CSCI 334: Principles of Programming Languages

Lecture 9: Lisp

Instructor: Dan Barowy

Williams

Announcements

Announcements

Midterm exam *next* Thursday

Announcements

Midterm exam *next* Thursday

Optional Lisp homework posted later today

Announcements

Midterm exam *next* Thursday

Optional Lisp homework posted later today

Feedback: slides insufficient (clarification?)

Choosing a language

Choosing a language

Many good technical reasons

Choosing a language

Many good technical reasons

Few people decide for technical reasons

Choosing a language

Many good technical reasons

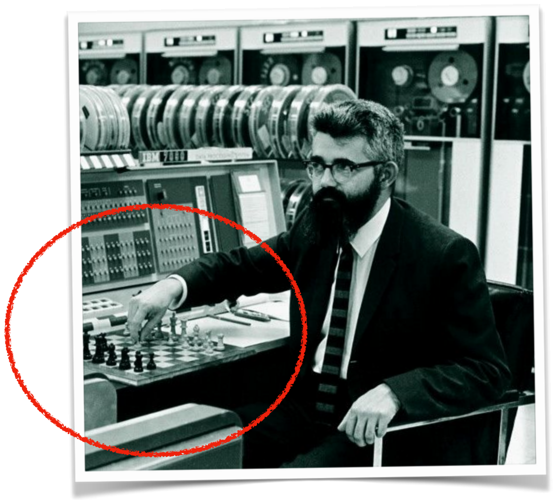
Few people decide for technical reasons

Seibel: Why do people get so religious about their computer languages?

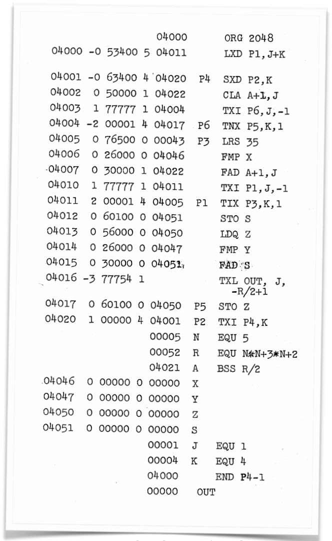
Bloch: I don't know. But when you choose a language, you're choosing more than a set of technical trade-offs—you're choosing a community. It's like choosing a bar. Yes, you want to go to a bar that serves good drinks, but that's not the most important thing. It's who hangs out there and what they talk about. And that's the way you choose computer languages. Over time the community builds up the language—not only the people, but the software artifacts: tools, libraries, and so forth. That's one of the reasons that sometimes languages that are, on paper, better than other languages don't win—because they just haven't built the right communities around themselves.

John McCarthy

IBM 704



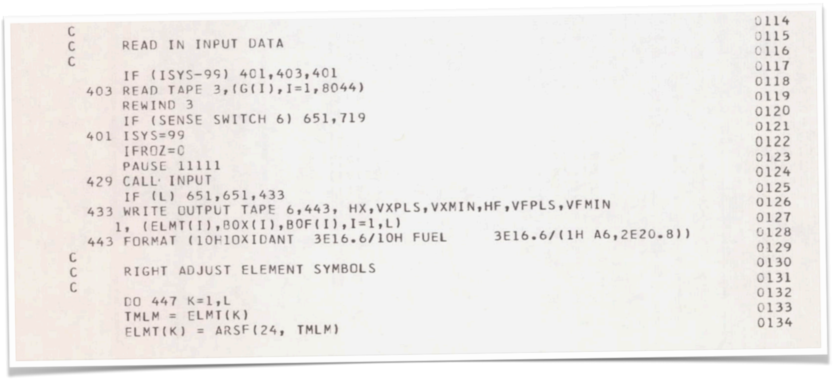
Lisp was invented for AI research



```
                    04000      ORG 2048
04000 -0 53400 5 04011         LXD P1,J+K

04001 -0 63400 4 04020   P4    SXD P2,K
04002  0 50000 1 04022         CLA A+1,J
04003  1 77777 1 04004         TXI P6,J,-1
04004 -2 00001 4 04017   P6    TNX P5,K,1
04005  0 76500 0 00043   P3    LRS 35
04006  0 26000 0 04046         FMP X
04007  0 30000 1 04022         FAD A+1,J
04010  1 77777 1 04011         TXI P1,J,-1
04011  2 00001 4 04005   P1    TIX P3,K,1
04012  0 60100 0 04051         STO S
04013  0 56000 0 04050         LDQ Z
04014  0 26000 0 04047         FMP Y
04015  0 30000 0 04051         FAD S
04016 -3 77754 1               TXL OUT, J,
                                 -R/2+1
04017  0 60100 0 04050   P5    STO Z
04020  1 00000 4 04001   P2    TXI P4,K
                 00005   N     EQU 5
                 00052   R     EQU N*N+3*N+2
                 04021   A     BSS R/2
04046  0 00000 0 00000   X
04047  0 00000 0 00000   Y
04050  0 00000 0 00000   Z
04051  0 00000 0 00000   S
                 00001   J     EQU 1
                 00004   K     EQU 4
                 04000         END P4-1
                 00000   OUT
```

704 Assembly (circa 1954)
(From Coding the MIT-IBM 704 Computer)



```
C
C     READ IN INPUT DATA
C
      IF (ISYS-99) 401,403,401
  403 READ TAPE 3,(G(I),I=1,8044)
      REWIND 3
      IF (SENSE SWITCH 6) 651,719
  401 ISYS=99
      IFROZ=0
      PAUSE 11111
  429 CALL INPUT
      IF (L) 651,651,433
  433 WRITE OUTPUT TAPE 6,443, HX,VXPLS,VXMIN,HF,VFPLS,VFMIN
     1, (ELMT(I),BOX(I),BOF(I),I=1,L)
  443 FORMAT (10H1OXIDANT  3E16.6/10H FUEL     3E16.6/(1H A6,2E20.8))
C
C     RIGHT ADJUST ELEMENT SYMBOLS
C
      DO 447 K=1,L
      TMLM = ELMT(K)
      ELMT(K) = ARSF(24, TMLM)
```

FORTRAN (circa 1956)
(From NASA Technical Note D-1737)

```
(defun fact (n)
    (cond ((eq n 0) 1)
          (t (* n (fact (- n 1))))))
```

LISP (circa 1958)

LISP is a “functional” language

LISP is a "functional" language

- programs are modeled after math. functions

LISP is a "functional" language

- programs are modeled after math. functions
- no statements, only expressions

LISP is a "functional" language

- programs are modeled after math. functions
- no statements, only expressions
- no "mutable" variables, only declarations

LISP is a "functional" language

- programs are modeled after math. functions
- no statements, only expressions
- no "mutable" variables, only declarations
- therefore, the effect of running a program ("evaluation") is purely the effect of applying a function to an input.

What does this mean?

What does this mean?

- there is no hidden state (globals in C) or state-change rules (as in Java).

What does this mean?

- there is no hidden state (globals in C) or state-change rules (as in Java).
- complex parts are composed out of simple parts

What does this mean?

- there is no hidden state (globals in C) or state-change rules (as in Java).
- complex parts are composed out of simple parts
- we care about what a program means and not what it does: you can understand a program without having to mentally execute it

What does this mean?

- there is no hidden state (globals in C) or state-change rules (as in Java).
- complex parts are composed out of simple parts
- we care about what a program means and not what it does: you can understand a program without having to mentally execute it
- *programs are easier to reason about*

functional language (again)

functional language (again)

- programs are modeled after math. functions

functional language (again)

- programs are modeled after math. functions
- no statements, only expressions

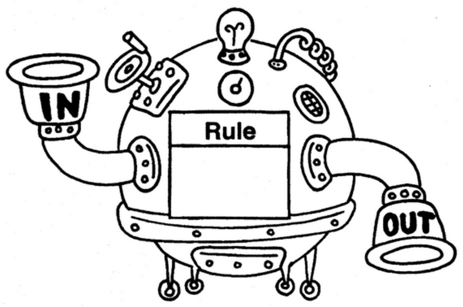
functional language (again)

- programs are modeled after math. functions
- no statements, only expressions
- no "mutable" variables, only declarations

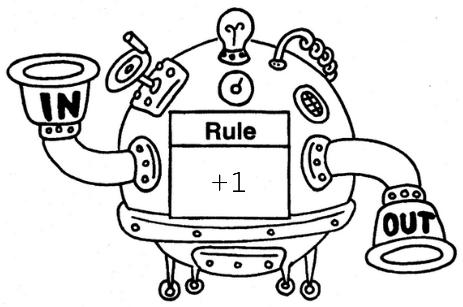
functional language (again)

- programs are modeled after math. functions
- no statements, only expressions
- no "mutable" variables, only declarations
- therefore, the effect of running a program ("evaluation") is purely the effect of applying a function to an input.

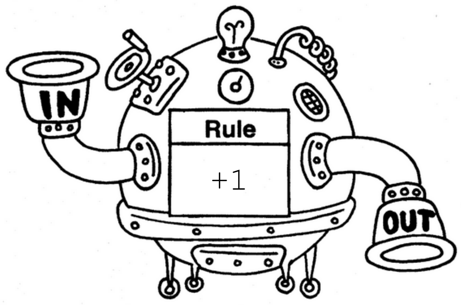
LISP is a "functional" language



LISP is a "functional" language

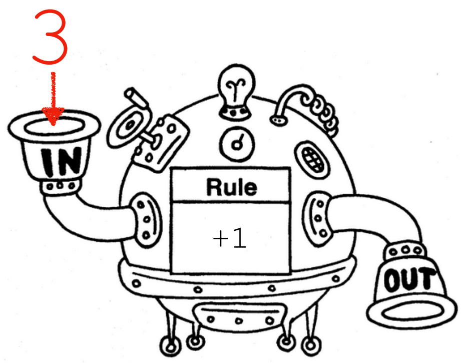


LISP is a “functional” language



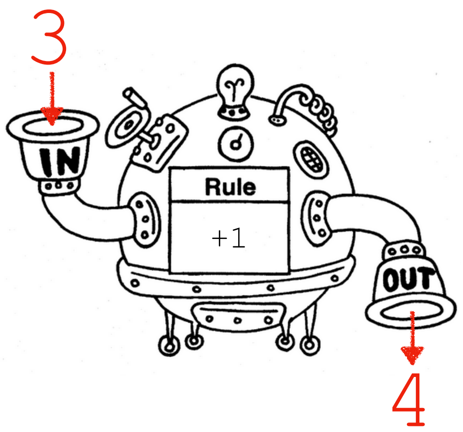
```
(defun add-one (n) (+ n 1))
```

LISP is a “functional” language



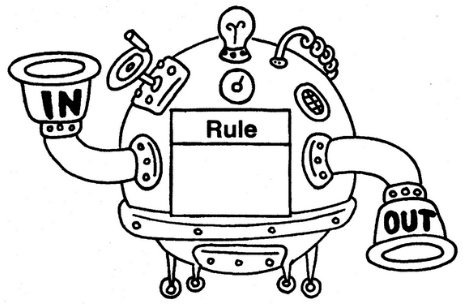
```
(defun add-one (n) (+ n 1))
```

LISP is a “functional” language

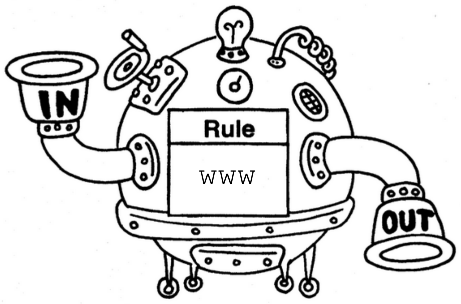


```
(defun add-one (n) (+ n 1))
```

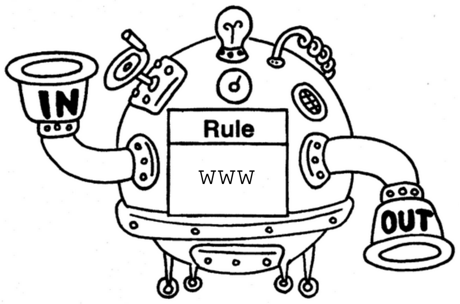
LISP is a “functional” language



LISP is a “functional” language



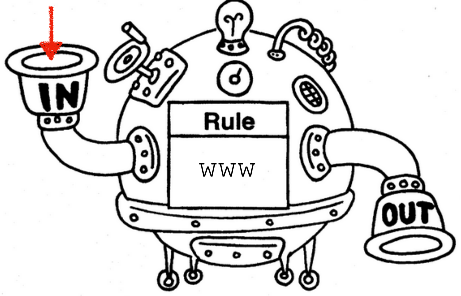
LISP is a “functional” language



```
(defun cleaning-robot (dirt) …)
```

LISP is a “functional” language

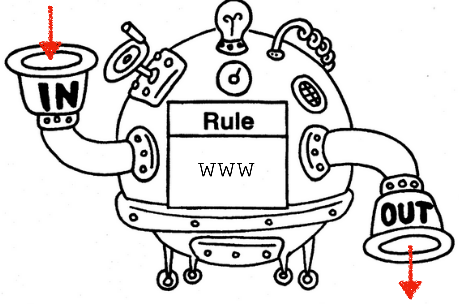
dirty house



```
(defun cleaning-robot (dirt) …)
```

LISP is a “functional” language

dirty house

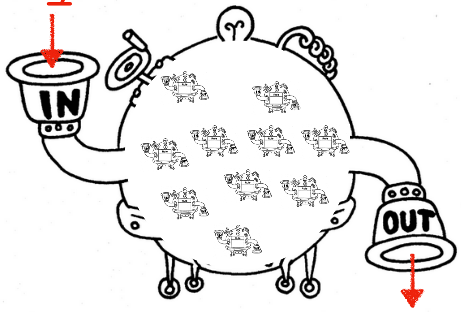


clean house

```
(defun cleaning-robot (dirt) …)
```

Big functions are “composed” of little functions

dirty house

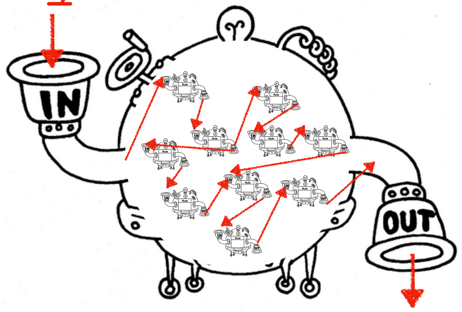


clean house

```
(defun cleaning-robot (dirt) …)
```

Big functions are “composed” of little functions

dirty house

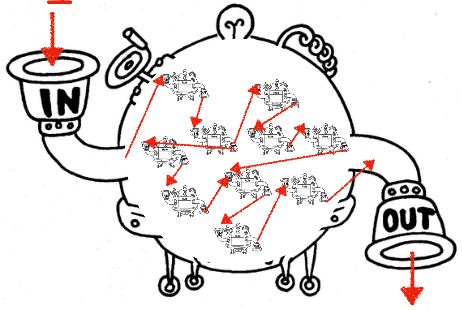


clean house

```
(defun cleaning-robot (dirt) …)
```

Program correctness is easier to achieve

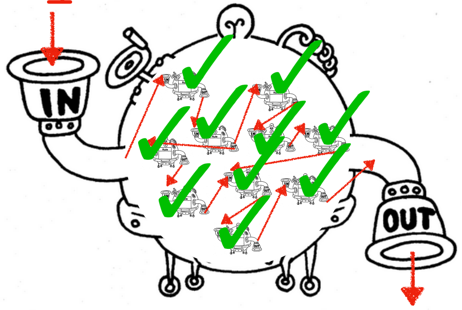
dirty house



clean house

Program correctness is easier to achieve

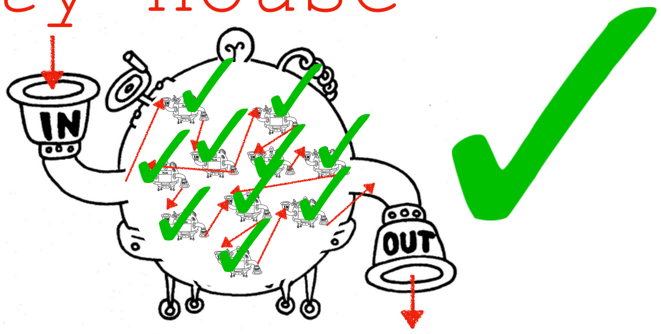
dirty house



clean house

Program correctness is easier to achieve

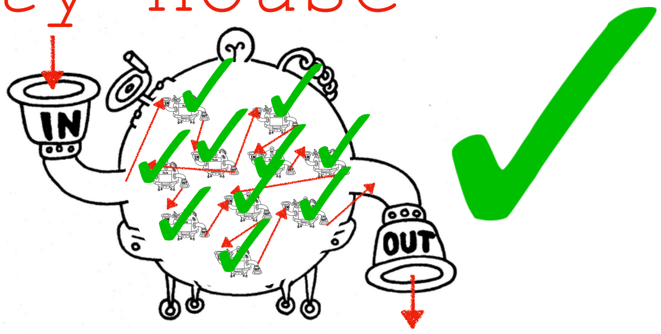
dirty house



clean house

Program correctness is easier to achieve

dirty house



clean house

I.e., whole is correct if pieces are correct.

LISP is inspired by the lambda calculus

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

  function of two values: `(+ 1 1)`

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

  function of two values: `(+ 1 1)`
- syntax is (mind-numbingly) regular

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

  function of two values: `(+ 1 1)`
- syntax is (mind-numbingly) regular

  functions: `(function-name arguments …)`

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

  function of two values: `(+ 1 1)`

- syntax is (mind-numbingly) regular

  functions: `(function-name arguments …)`

  values: anything that is not a function

LISP is inspired by the lambda calculus

- everything either a value or a funct. of a value

  value: `1`

  function of two values: `(+ 1 1)`

- syntax is (mind-numbingly) regular

  functions: `(function-name arguments …)`

  values: anything that is not a function

- evaluating a function produces a value:

  `(+ 1 1) ↠ 2`

Statements vs. expressions

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

  Java: `i++`

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

  Java: `i++`

  value stored at location i incremented by one

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

  Java: `i++`

  value stored at location i incremented by one
- An *expression* is a combination of functions and values that yields a new value

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

  Java: `i++`

  value stored at location i incremented by one
- An *expression* is a combination of functions and values that yields a new value

  Lisp: `(+ i 1)`

Statements vs. expressions

- A *statement* is an operation that changes the state of the computer

  Java: `i++`

  value stored at location i incremented by one
- An *expression* is a combination of functions and values that yields a new value

  Lisp: `(+ i 1)`

  evaluating + with i and 1 returns i + 1

REPL
(read-eval-print loop)

Batch mode

Mutable variables

Mutable variables

- If you can update a variable, your language has mutable variables

Mutable variables

- If you can update a variable, your language has mutable variables

  Java:
  ```
  int i = 3;
  i = 4;
  ```

Mutable variables

- If you can update a variable, your language has mutable variables

  Java:
  ```
  int i = 3;
  i = 4;
  ```
- Notice that both lines of code are statements

Mutable variables

- If you can update a variable, your language has mutable variables

  Java:
  ```
  int i = 3;
  i = 4;
  ```
- Notice that both lines of code are statements
- (pure) Lisp does not have mutable variables

Immutable variables

Immutable variables

- Variables cannot be updated in Lisp

Immutable variables

- Variables cannot be updated in Lisp

  Lisp: `(defun my-func (i) …)`

  `(my-func 3)`

Immutable variables

- Variables cannot be updated in Lisp

  Lisp: `(defun my-func (i) …)`

  `(my-func 3)`

  or the shorter

  `((lambda (i) …) 3)`

Immutable variables

- Variables cannot be updated in Lisp

  Lisp: `(defun my-func (i) …)`

  `(my-func 3)`

  or the shorter

  `((lambda (i) …) 3)`
- Notice that all of the above are expressions

Immutable variables

- Variables cannot be updated in Lisp

  Lisp: `(defun my-func (i) …)`

  `(my-func 3)`

  or the shorter

  `((lambda (i) …) 3)`
- Notice that all of the above are expressions

- In fact, functions are the only way to bind values to names in (pure) Lisp

Lisp syntax: atoms

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
```

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
112.75
```

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
112.75
“hello”
```

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
112.75
“hello”
‘foo
```

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
112.75
“hello”
‘foo
t
```

Lisp syntax: atoms

- An atom is the most basic unit in Lisp: data

```
4
112.75
“hello”
‘foo
t
nil
```

Lisp syntax: cons cells

Lisp syntax: cons cells

- cons cells are the basic unit of composition in Lisp (recall C composes data with `struct`).

Lisp syntax: cons cells

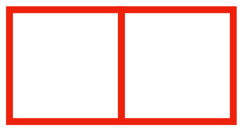
- cons cells are the basic unit of composition in Lisp (recall C composes data with `struct`).

```
(cons "hello" 4)
```

Lisp syntax: cons cells

- cons cells are the basic unit of composition in Lisp (recall C composes data with `struct`).

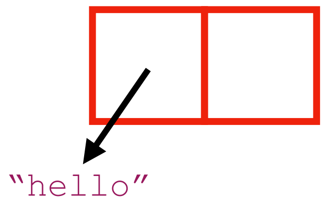
```
(cons "hello" 4)
```



Lisp syntax: cons cells

- cons cells are the basic unit of composition in Lisp (recall C composes data with `struct`).

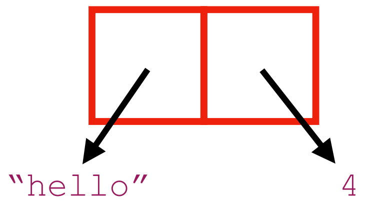
```
(cons "hello" 4)
```



Lisp syntax: cons cells

- cons cells are the basic unit of composition in Lisp (recall C composes data with `struct`).

```
(cons "hello" 4)
```



Lisp = LISt Processor

Lisp syntax: lists

Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

Lisp syntax: lists

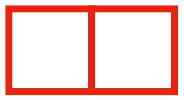
- E.g., lists in Lisp are just made out of cons cells

```
(cons 1 (cons 2 (cons 3 nil)))
```

Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

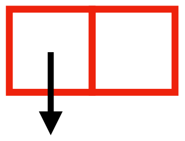
`(cons 1 (cons 2 (cons 3 nil)))`



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

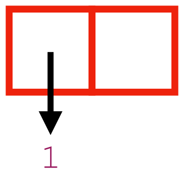
`(cons 1 (cons 2 (cons 3 nil)))`



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

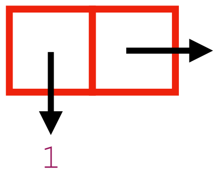
`(cons 1 (cons 2 (cons 3 nil)))`



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

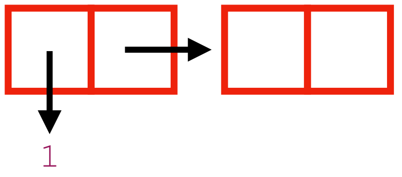
`(cons 1 (cons 2 (cons 3 nil)))`



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

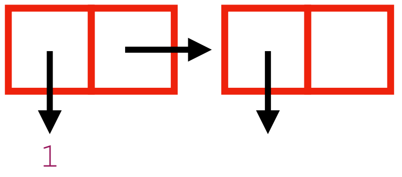
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

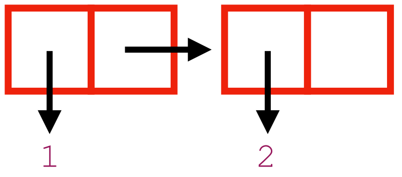
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

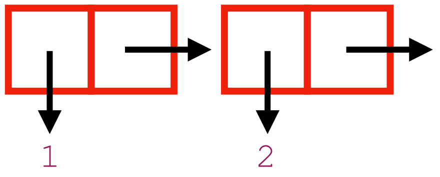
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

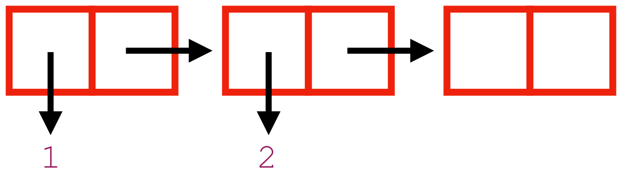
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

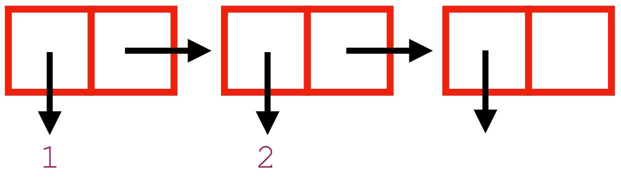
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

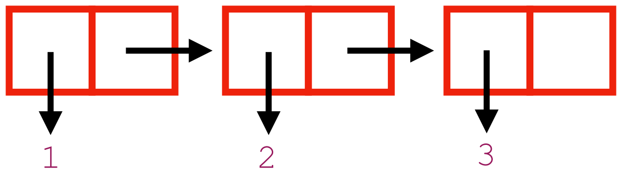
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

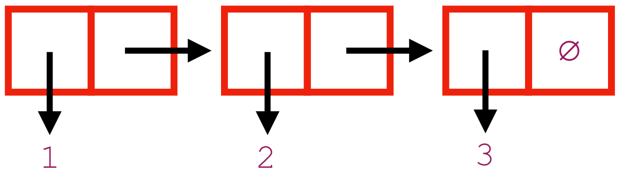
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

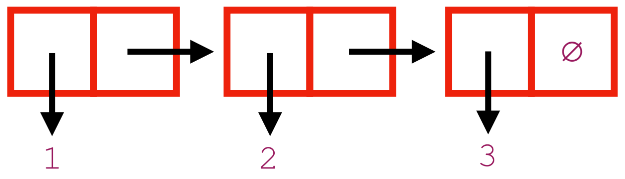
```
(cons 1 (cons 2 (cons 3 nil)))
```



Lisp syntax: lists

- E.g., lists in Lisp are just made out of cons cells

  `(cons 1 (cons 2 (cons 3 nil)))`



- Lisp has a shorthand for this:

  `'(1 2 3)`

Lisp syntax: car and cdr

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

  `(car (cons 1 2)) = 1`

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

```
(car (cons 1 2)) = 1
```

- Access the second element with `cdr`

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

```
(car (cons 1 2)) = 1
```

- Access the second element with `cdr`

```
(cdr (cons 1 2)) = 2
```

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

```
(car (cons 1 2)) = 1
```

- Access the second element with `cdr`

```
(cdr (cons 1 2)) = 2
```

- What's the value of the following expression?

```
(car '(1 2 3))
```

Lisp syntax: car and cdr

- Access the first element of a cons cell with `car`

```
(car (cons 1 2)) = 1
```

- Access the second element with `cdr`

```
(cdr (cons 1 2)) = 2
```

- What's the value of the following expression?

```
(car '(1 2 3))
```

- What about this?

```
(cdr '(1 2 3))
```

Lisp syntax: functions

Lisp syntax: functions

- Everything else is a function (or special form)

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

```
(car …)
```

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

  `(car …)`

  `(cdr …)`

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

  `(car …)`

  `(cdr …)`

  `(append …)`, etc.

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

  `(car …)`

  `(cdr …)`

  `(append …)`, etc.
- And you can define your own

  `(defun my-func (arg) (value))`

Lisp syntax: functions

- Everything else is a function (or special form)
- There are a bunch of built-in functions

  `(car …)`

  `(cdr …)`

  `(append …)`, etc.
- And you can define your own

  `(defun my-func (arg) (value))`

Lisp syntax: conditionals

Lisp syntax: conditionals

- In Lisp, there is no `if/else`

Lisp syntax: conditionals

- In Lisp, there is no `if/else`

```
(cond ((test) (value)) …)
```

Lisp syntax: conditionals

- In Lisp, there is no `if/else`

```
(cond ((test) (value)) …)
```

- E.g.,

```
(cond ((eq 1 x) (cons x xs)) …)
```

Lisp syntax: conditionals

- In Lisp, there is no `if/else`

  `(cond ((test) (value)) …)`

- E.g.,

  `(cond ((eq 1 x) (cons x xs)) …)`

- Does the same as the Java

```
if (x == 1) {
  xs.add(x);
}
```

Lisp syntax: conditionals

Lisp syntax: conditionals

- `cond` is more general than `if/else`.

Lisp syntax: conditionals

- `cond` is more general than `if/else`.

```
(cond ((test1) (value1))
      ((test2) (value2)
      …)
```

Lisp syntax: conditionals

- `cond` is more general than `if/else`.

```
(cond ((test1) (value1))
      ((test2) (value2)
      ...)
```

```
(defun insert (x l)
  (cond ((eq l nil) (cons x nil))
        ((< x (car l)) (cons x l))
        (t (cons
             (car l)
             (insert x (cdr l))))))
```

That's pretty much it!

That's pretty much it!

- See "334 Lisp FAQ" for all the syntax you need to know on course webpage

That's pretty much it!

- See "334 Lisp FAQ" for all the syntax you need to know on course webpage
- If you happen to be looking at the book, a slightly different syntax is used (mostly Scheme).

That's pretty much it!

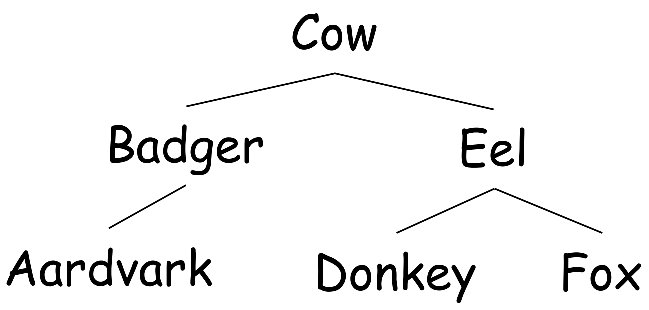
- See "334 Lisp FAQ" for all the syntax you need to know on course webpage
- If you happen to be looking at the book, a slightly different syntax is used (mostly Scheme).

Activity

list length

```
(length-list '(1 2 3 4 5 6)) ↠ 6
```

Activity



Memory management

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

  ```
  MyObject *m = malloc(sizeof(MyObject));
  ```

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

  ```
  MyObject *m = malloc(sizeof(MyObject));
  m->foo = 2;
  ```

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

  ```
  MyObject *m = malloc(sizeof(MyObject));
  m->foo = 2;
  m->bar = 3;
  ```

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

```
MyObject *m = malloc(sizeof(MyObject));
m->foo = 2;
m->bar = 3;
… do stuff with m …
```

Memory management

- C:

  When you want to use a variable, you have to *allocate* it first, then *decallocate* it when done.

```
MyObject *m = malloc(sizeof(MyObject));
m->foo = 2;
m->bar = 3;
… do stuff with m …
free(m);
```

Memory management

Memory management

- Lisp and Java:

  You barely need to think about this at all.

Memory management

- Lisp and Java:

  You barely need to think about this at all.

  ```
  MyObject m = new MyObject(2,3);
  … do stuff with m …
  ```

Memory management

- Lisp and Java:

  You barely need to think about this at all.

  ```
  MyObject m = new MyObject(2,3);
  … do stuff with m …
  (cons 2 3)
  ```