Week 12

A Brief Introduction to Lisp

@nebu

Outline

Learn (most of) Lisp

Most of the rest of Lisp

Data Structures from Nothing at All (AKA Data is Code)

Infinite-Size Data Structures

A Lisp Interpreter in Lisp

Σ

Section 1

Learn (most of) Lisp

Previously on SIGma

- We covered the lambda calculus, a model of computation radically different from Turing's.
- How can we use it to build an actual programming language that actually *does* stuff?

By the End...

- Know most of a simple Lisp (subset of Racket).

By the End...

- Know most of a simple Lisp (subset of Racket).
- Realize that code is data, data is code, often in weird ways.

By the End...

- Know most of a simple Lisp (subset of Racket).
- Realize that code is data, data is code, often in weird ways.
- See some cool programming structures you may not have seen before, and

By the End...

- Know most of a simple Lisp (subset of Racket).
- Realize that code is data, data is code, often in weird ways.
- See some cool programming structures you may not have seen before, and
- Not hate the parens too much.

Introducing Lisp

- Lisp is a family of interpreted programming languages which use a very specific kind of parenthesis-based notation.

Introducing Lisp

- Lisp is a family of interpreted programming languages which use a very specific kind of parenthesis-based notation.
- Second oldest programming language.

Introducing Lisp

- Lisp is a family of interpreted programming languages which use a very specific kind of parenthesis-based notation.
- Second oldest programming language.
- Ideas from Lisp are *still* being ported to modern languages: see higher order functions, lambdas, etc.

Introducing Lisp

- Lisp is a family of interpreted programming languages which use a very specific kind of parenthesis-based notation.
- Second oldest programming language.
- Ideas from Lisp are *still* being ported to modern languages: see higher order functions, lambdas, etc.
- ...so it's probably worth knowing!

Atoms

Let's start talking to Racket (the variant of Lisp I like).

Atoms

Let's start talking to Racket (the variant of Lisp I like).
Numbers and strings are called atoms. The interpreter echoes the atom it gets. So...

Atoms

Let's start talking to Racket (the variant of Lisp I like).

Numbers and strings are called atoms. The interpreter echoes the atom it gets. So...

```
3
```

Atoms

Let's start talking to Racket (the variant of Lisp I like).
Numbers and strings are called atoms. The interpreter echoes the atom it gets. So...

```
3
```

3

More Atoms

```
"Sigma is a Greek letter."
```

Σ

More Atoms

```
"Sigma is a Greek letter."
```

```
Sigma is a Greek letter.
```

More Atoms

```
"Sigma is a Greek letter."
```

```
Sigma is a Greek letter.
```

```
17.5524
```

More Atoms

```
"Sigma is a Greek letter."
```

```
Sigma is a Greek letter.
```

```
17.5524
```

```
17.5524
```

Let's Try More Than One Atom

```
17.4 19.5 "sigma"
```

Let's Try More Than One Atom

```
17.4 19.5 "sigma"
```

```
17.4
19.5
"sigma"
```

What Did You Learn About Lisp?

- Atoms *evaluate* to themselves.

What Did You Learn About Lisp?

- Atoms *evaluate* to themselves.
- Untyped — there seems to be no distinction between floating point numbers, integers, and strings — everything is an “atom”.

Means of Combination

- Atoms by themselves aren't very useful.
- For instance, *having* 3 and 4 as numbers is fine, but you really want to operate on them somehow.

Means of Combination

We do so by using Lisp's parentheses. Thus:

```
(+ 3 4)
```

Means of Combination

We do so by using Lisp's parentheses. Thus:

```
(+ 3 4)
```

```
7
```

- The parens mean *function application*. The first element in the parens is treated as a function, and the rest as arguments to the function.
- `(+ 3 4)` means: apply `+` to `3` and `4`. The result is an atom, `7`, which the interpreter prints out for us.

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

10

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

```
10
```

```
(- 10 4)
```

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

10

```
(- 10 4)
```

6

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

10

```
(- 10 4)
```

6

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

Now, Guess the Outputs:

```
(+ 3 4 1 2)
```

10

```
(- 10 4)
```

6

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

6

Guess the Output...

```
(= 3 4)
```

Guess the Output...

```
(= 3 4)
```

#f

Guess the Output...

```
(= 3 4)
```

#f

```
(= 4 4)
```

Guess the Output...

```
(= 3 4)
```

#f

```
(= 4 4)
```

#t

Parens Are Easy!

- No operator precedence issues, see:

```
(* 2 (+ 4 5))
```

Parens Are Easy!

- No operator precedence issues, see:

```
(* 2 (+ 4 5))
```

```
18
```

Parens Are Easy!

- No operator precedence issues, see:

```
(* 2 (+ 4 5))
```

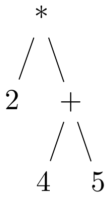
```
18
```

- If you think about it, it's actually the AST written out, so Lisp is one of the easiest languages to parse.

For Example

```
(* 2 (+ 4 5))
```

is equivalent to:



Means of Abstraction

- So, we have atoms we can combine to compute stuff.
- How can we name things?

```
(define x 13)
x
```

Means of Abstraction

- So, we have atoms we can combine to compute stuff.
- How can we name things?

```
(define x 13)
x
```

```
13
```

- This is risky territory: this is not an assignment.
- Think of it like setting up an alias: "x" refers to the same thing as "13". `x` is not a variable that you'd mutate using sequential operations, like in C.

How to Define Your Own Function

You saw me use `+`, `*`, etc. Setting up our own function is easy:

```
(lambda (x) (* x x))
```

How to Define Your Own Function

You saw me use `+`, `*`, etc. Setting up our own function is easy:

```
(lambda (x) (* x x))
```

```
#<procedure:...pp2qm/ob-FqzJ1n.rkt:3:0>
```

- This is a function that takes in a single argument, `x`, and returns `(* x x)`, the square.

Using the Function

```
((lambda (x) (* x x)) 5)
```

Using the Function

```
((lambda (x) (* x x)) 5)
```

25

It's Nicer to Name Stuff...

```
(define square
  (lambda (x) (* x x)))

(square (square 5))
```

It's Nicer to Name Stuff...

```
(define square
  (lambda (x) (* x x)))

(square (square 5))
```

625

Or use the syntactic sugar:

```
(define (square x) (* x x))
(square (square 5))
```

It's Nicer to Name Stuff...

```
(define square
  (lambda (x) (* x x)))

(square (square 5))
```

625

Or use the syntactic sugar:

```
(define (square x) (* x x))
(square (square 5))
```

625

Questions? Questions!

1. Write a Lisp procedure **`average`**, that takes two arguments and computes their arithmetic mean:

$$\texttt{average}(\texttt{x}, \texttt{y}) = \frac{x + y}{2}$$

2. Using **`average`** and the **`square`** function we defined earlier, define a function:

$$\texttt{mean-square(x, y)} = \frac{x^2 + y^2}{2}$$

Good Job!

At this point, you understand the hard part of the language!

Section 2

Most of the rest of Lisp

Fibonacci and Conditionals

Fibonacci numbers are defined as:

$$F(x) = \begin{cases} 0, & x = 0 \\ 1, & x = 1 \\ F(x-1) + F(x-2), & \text{otherwise} \end{cases}$$

- To implement this in Lisp, we need conditional statements.
- We derived them from the lambda calculus last time, so we'll go ahead and use them.

Lisponacci

Recall the the `if` is a ternary, of form `(if cond then a else b)`:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 10)
```

55

Questions? Questions!

Write the absolute value function in Lisp:

$$\texttt{abs}(x) = \begin{cases} -x, & x < 0 \\ 0, & x = 0 \\ x, & x > 0 \end{cases}$$

Challenge question: Write an iterative implementation of Fibonacci that runs in $O(n)$.

Section 3

Data Structures from Nothing at All (AKA Data is Code)

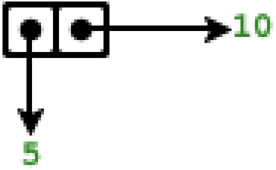
Pairs

Lisp provides a primitive called `cons`, that lets us create pairs of things. Using it:

```
(cons 5 10)
```

```
'(5 . 10)
```

The way to picture a pair is:



If You Have Two Things, You Can Have as Many as You Want

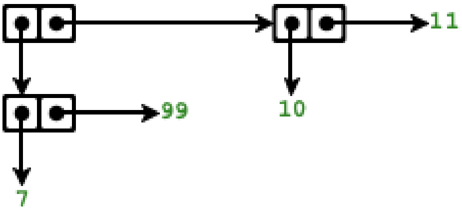
```
(cons (cons 7 99) (cons 10 11))
```

What's the box and pointer for this structure?

If You Have Two Things, You Can Have as Many as You Want

```
(cons (cons 7 99) (cons 10 11))
```

What's the box and pointer for this structure?



Lists

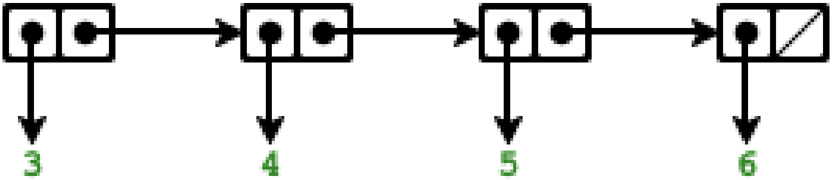
What about this one?

```
(cons 3 (cons 4 (cons 5 (cons 6 '()))))
```

Lists

What about this one?

```
(cons 3 (cons 4 (cons 5 (cons 6 '()))))
```

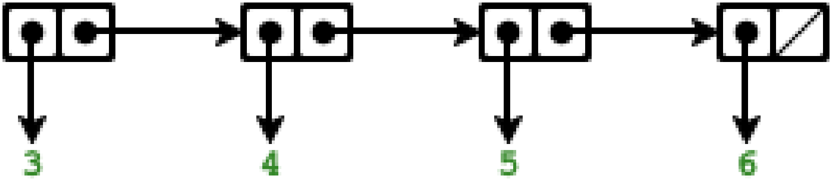


This is the linked list, a fundamental structure to Lisp. In fact, Lisp stands for LISt Processing.

Easier Lists

Our Lisp provides an easier way to declare a list to make life easier:

```
(list 3 4 5 6)
```



Talking to Pairs

Lisp provides `car` (get the first element of a pair) and `cdr` (get the second element of a pair).

```
(define x (cons 5 10))
(car x)
(cdr x)
```

Talking to Pairs

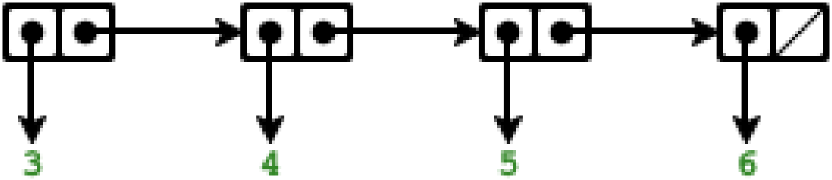
Lisp provides `car` (get the first element of a pair) and `cdr` (get the second element of a pair).

```
(define x (cons 5 10))
(car x)
(cdr x)
```

```
5
10
```

Talking to Lists

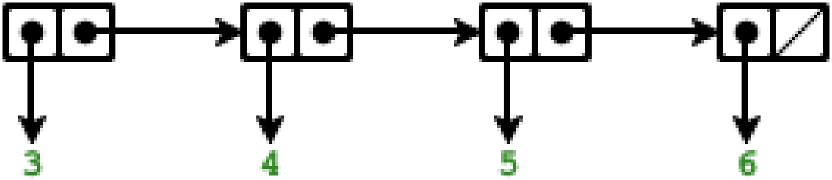
We can `car` and `cdr` a list too. Think about what it means...




```
(define lst (list 3 4 5 6))
(car lst)
(cdr lst)
```

Talking to Lists

We can `car` and `cdr` a list too. Think about what it means...




```
(define lst (list 3 4 5 6))
(car lst)
(cdr lst)
```

```
3
'(4 5 6)
```

Processing Lists

What if we want to take a list and return a list that's its element-wise square? Easy.

```
(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst)))))

(square-list (list 3 4 5 6))
```

```
'(9 16 25 36)
```

Higher Order Procedures on Data: Code as Data

Now, write the following functions:

- Cube every element in a list.
- Multiply every element in a list by k.
- Subtract 2 from every element in a list.
- Take the square root of every element in a list.

Hopefully, you recognize that this is the same pattern as `square-list`: we're just using `cube` or some other function instead of `square`.

Map

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
            (map fn (cdr lst)))))

(map square (list 3 4 5 6))
(map cube (list 3 4 5 6))
```

```
'(9 16 25 36)
'(27 64 125 216)
```

So...

We just passed a function as an input to a function. Code is being treated as data in this case.

By the way, Python, Rust, and lots of new languages have `map`.

Where Did the Data Come From?: Data as Code

I tricked you, a bit. Lambda calculus doesn't have pairs, so it can't have lists or other data structures!

I'm glad you bring that up...

Church's Pairs are Unary Functions

```
(define (cons x y)
  (lambda (m)
    (m x y)))

(define (car x)
  (x (lambda (a d)
       a)))

(define (cdr x)
  (x (lambda (a d)
       d)))
```

And so, all our data structures are actually code?
(This basically proves Lisp/lambda calculus is Turing complete, since you can implement the tape of the machine using a list.)

Pairs are Unary Functions

```
(define (cons x y) ; cons is a binary function on x and y
  (lambda (m) ; that returns a unary function on m
    (m x y))) ; which applies m to x and y.

(define (car x) ; car is a unary function on x (a pair)
  (x (lambda (a d) ;  that applies x to a function that
       a))) ; returns its first input

(define (cdr x) ; cdr is a unary function on x (a pair)
  (x (lambda (a d) ;  that applies x to a function that
       d))) ; retrns its second input
```

And so, all our data structures are actually code?
(This basically proves Lisp/lambda calculus is Turing complete, since you can implement the tape of the machine using a list.)

Another Formulation

This is an easier to follow pair implementation, but it differs from Church's original construction.
(This one requires conditionals and integers to be implemented in the λ calculus, and is thus less "pure".)

```
(define (our-cons a b)
  (lambda (pick)
    (cond ((= pick 1) a)
          ((= pick 2) b))))

(define (our-car x) (x 1))
(define (our-cdr x) (x 2))
```

Questions? Questions!

Here's Lisp code to sum all the elements of a list.

```
(define (sum-list lst)
  (if (null? lst)
      0
      (+ (car lst)
         (sum-list (cdr lst)))))

(sum-list (list 1 2 3 4)) ;; 10
```

Write a higher-order function `fold(fn, init, lst)` that combines all the elements of `lst` and `init` using the binary function `fn`. Then we should be able to do `sum-list` as:

```
(fold + 0 lst)
```

Section 4

Infinite-Size Data Structures

Lisp is Inefficient!

Well, yeah. Here's a certain form of inefficiency:

```
(require math/number-theory) ;; for prime?

(define (get-prime low high)
  (filter prime? (range low high)))

(first (rest (get-prime 10 1000)))
```

```
13
```

We generate all the primes till 1000, but only use the second one. Lots of wasted computation here.

This is Fixable

Swap “lists” for “streams”, and this code runs instantly.

```
(require math/number-theory) ;; for prime?

(define (get-prime low)
  (stream-filter prime? (in-naturals low)))

(stream-first (stream-rest (get-prime 10)))
```

```
13
```

`(in-naturals low)` is all natural numbers, from `low` to infinity.

Infinite Data in a Finite Memory?

As usual, I'm cheating.

A stream is an on-demand or "lazy" data structure, i.e., it is basically a pair:

```
(10, promise)
```

It is the first element of the stream, along with a promise to compute the rest, at some point.

Laziness is Good

- The stream is a pair `(first-element, promise)`.
- *Only* when we want more than the `first-element` is the `promise` `force`'d to be computed.

Laziness is Magic!

Not really. Here's the function to create and operate on a stream:

```
(define (cons-stream a e)
  (cons a (promise e)))

(define (first s)
  (car s))

(define (rest s)
  (force (cdr s)))
```

(Note in Racket you'd use `define-syntax` because of evaluation order.)

So the Magic is in `promise`. Or `force`!

Again, not really:

```
(define (promise expr)
     (lambda () expr))

(define (force p) (p))
```

And Now, We Have Infinitely Long Data Structures!

Python uses this concept with generator expressions, which are again lazy/on-demand. Haskell's lists are streams by default.

```
print(range(10**10**3).index(2))
```

```
2
```

Section 5

A Lisp Interpreter in Lisp

There are Further Mysteries...

Interpreting Lisp with Lisp is two (tiny) functions:

```
(define (eval exp env)
  (cond
    [(self-evaluating? exp) exp]
    [(variable? exp) (lookup-variable-value exp env)]
    [(quoted? exp) (text-of-quotation exp)]
    [(assignment? exp) (eval-assignment exp env)]
    [(definition? exp) (eval-definition exp env)]
    [(if? exp) (eval-if exp env)]
    [(lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env)]
    [(begin? exp) (eval-sequence (begin-actions exp) env)]
    [(cond? exp) (eval (cond->if exp) env)]
    [(application? exp) (apply (eval (operator exp) env) (list-of-values (operands exp) env))]
    [else (error "Unknown expression type: EVAL" exp)]))

(define (apply procedure arguments)
  (cond
    [(primitive-procedure? procedure) (apply-primitive-procedure procedure arguments)]
    [(compound-procedure? procedure)
     (eval-sequence (procedure-body procedure)
                    (extend-environment (procedure-parameters procedure)
                                        arguments
                                        (procedure-environment procedure)))]
    [else (error "Unknown procedure type: APPLY" procedure)]))
```

Greenspun's tenth rule of programming

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.