An Alternative Introduction to Programming

Read: A Tutorial of the Scheme Programming language

Shen Zheyu

October 12, 2017

Table of Contents I

Introduction

- Shen Zheyu, sophomore ECE student
- My GitHub: `http://github.com/arsdragonfly/`
- SSTIA projects `https://github.com/SSTIA`

Overview

This seminar is intended to provide a different introduction to programming from VG101 (and arguably many other courses). In a (hopefully) friendly way, you'll learn many useful things unlikely to be found in other introductory material.

Overview

This seminar is intended to provide a different introduction to programming from VG101 (and arguably many other courses). In a (hopefully) friendly way, you'll learn many useful things unlikely to be found in other introductory material.

Much of the content is adapted from *The Little Schemer, Fourth Edition* by D. P. Friedman and M. Felleisen. If you're very interested, You may also want to read *Structure and Interpretation of Computer Programs* by H. Abelson and G. J. Sussman.

Setup

To begin with this seminar, you need to have a Scheme interpreter up and running. I personally recommend Racket (`www.racket-lang.org`), though other programs may also work (MIT Scheme, Guile, Chez Scheme, etc.).

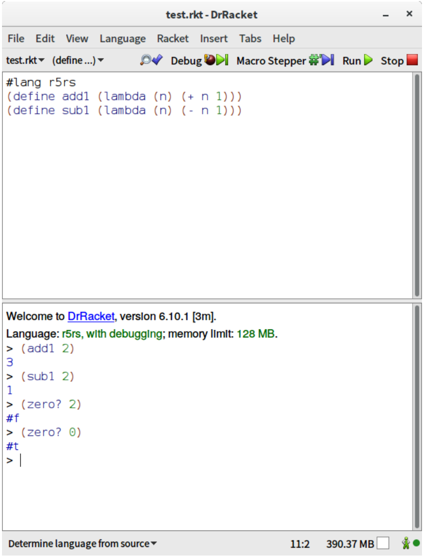
Setup

Before we start, we need to make sure that every program contains the following definitions of primitive functions:

```
(define add1
  (lambda (n)
    (+ n 1)))
(define sub1
  (lambda (n)
    (- n 1)))
(define atom? (lambda (x) (not (or (pair? x) (null? x)))))
```

It's recommended for this seminar now that you write your program in a single source code file, so that latter definitions of functions can build upon previous already-written ones.

Setup



test.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
test.rkt (define ...) Debug Macro Stepper Run Stop
#lang r5rs
(define add1 (lambda (n) (+ n 1)))
(define sub1 (lambda (n) (- n 1)))
Welcome to DrRacket, version 6.10.1 [3m].
Language: r5rs, with debugging; memory limit: 128 MB.
> (add1 2)
3
> (sub1 2)
1
> (zero? 2)
#f
> (zero? 0)
#t
>
Determine language from source
11:2 390.37 MB

Arithmetic on Natural Numbers

- What's the answer of (`add1 0`)?
- *1*.

Arithmetic on Natural Numbers

- What's the answer of (`add1 0`)?
- *1.*
- What's the answer of (`sub1 3`)?
- *2.*

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1.*
- What's the answer of `(sub1 3)`?
- *2.*
- What's the answer of `(add1 (add1 0))`?
- *It's the answer of `(add1 1)`.*

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1.*
- What's the answer of `(sub1 3)`?
- *2.*
- What's the answer of `(add1 (add1 0))`?
- *It's the answer of `(add1 1)`.*
- What's the answer of `(add1 1)` then?
- *2.*

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true"*.

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true".*
- What's the answer of `(zero? 810)`?
- `#f`, *which means "false".*

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true".*
- What's the answer of `(zero? 810)`?
- `#f`, *which means "false".*
- What's the answer of `(zero? (sub1 (sub1 2)))`?
- `#t`

λ ?

- What's the answer of `(lambda (x) (add1 (add1 x)))`?
- *A lambda expression, which is similar to a mathematical function.*

λ ?

- What's the answer of `(lambda (x) (add1 (add1 x)))`?
- *A lambda expression, which is similar to a mathematical function.*
- What's the answer of

  ```
  (define add2 (lambda (x) (add1 (add1 x))))
  ```

  ```
  (add2 (add1 3))
  ```

  ?
- *6.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is* `(add1 3)`*?*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*
- Then?
- *We substitute x for 4 in the $lambda$ $expression$.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is (add1 3)?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes ((lambda (x) (add1 (add1 x))) 4).*
- Then?
- *We substitute x for 4 in the lambda expression.*
- What will we get then?
- *(add1 (add1 4))*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*
- Then?
- *We substitute x for 4 in the $lambda$ $expression$.*
- What will we get then?
- *`(add1 (add1 4))`*
- Is that how we get 6?
- *Yes.*

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*
- What's the answer of `(one? 1)`?
- `#t`.

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*
- What's the answer of `(one? 1)`?
- `#t`.
- What's the answer of `(one? 2)`?
- `#f`.

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- How does the above definition work out?
- *Let's find it out step by step.*
- What's the answer of `(one? 1)`?
- it boils down to

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

cond?

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

- How does `cond` work then?
- *It works by asking questions: is* `(zero? (sub1 1))` *true?*

cond?

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

- How does `cond` work then?
- *It works by asking questions: is* `(zero? (sub1 1))` *true?*
- Yes. And then?
- *We get* `#t`.

cond?

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

- How does `cond` work then?
- *It works by asking questions: is* `(zero? (sub1 1))` *true?*
- Yes. And then?
- *We get* `#t`.
- Let's look at another example.

cond?

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

- What will happen when we try to find the answer of `(one? 2)`?
- *We ask questions again. Is `(zero? (sub1 2))` true?*

cond?

```
(cond
  ((zero? (sub1 1)) #t)
  (else #f))
```

- What will happen when we try to find the answer of `(one? 2)`?
- *We ask questions again. Is `(zero? (sub1 2))` true?*
- No. We then move on to the second question.
- *And `else` is always true.*
- Sure. It means "If the above don't work, try this." What do we get then?
- *We eventually get `#f`.*

Recur, recur, recur...

- Let's look at this definition.

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- *This looks complicated. How does it work?*

Recur, recur, recur...

- Let's look at this definition.

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- *This looks complicated. How does it work?*
- What's the answer of `(o+ 2 1)`?
- *3.*

- Let's look at this definition.

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- *This looks complicated. How does it work?*
- What's the answer of `(o+ 2 1)`?
- *3.*
- Let's follow it step by step. What's the answer of `(zero? 1)`?

Recur, recur, recur...

- Let's look at this definition.

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- *This looks complicated. How does it work?*
- What's the answer of `(o+ 2 1)`?
- *3.*
- Let's follow it step by step. What's the answer of `(zero? 1)`?
- `#f`.

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?
- *We ask for the answer of* `(add1 (o+ x (sub1 y)))`*.*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?
- *We ask for the answer of* `(add1 (o+ x (sub1 y)))`*.*
- Okay. What's the value of `(o+ x (sub1 y))`?
- *That's a good question. We need to first find the answer of* `x` *and* `(sub1 y)`*.*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?
- *We ask for the answer of* `(add1 (o+ x (sub1 y)))`*.*
- Okay. What's the value of `(o+ x (sub1 y))`?
- *That's a good question. We need to first find the answer of* `x` *and* `(sub1 y)`*.*
- What's the value of `x`? And what's the value of `(sub1 y)`?

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?
- *We ask for the answer of* `(add1 (o+ x (sub1 y)))`*.*
- Okay. What's the value of `(o+ x (sub1 y))`?
- *That's a good question. We need to first find the answer of* `x` *and* `(sub1 y)`*.*
- What's the value of `x`? And what's the value of `(sub1 y)`?
- *The value of* `x` *is 2, and the value of* `(sub1 y)` *is 0.*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- What do we do then?
- *We ask for the answer of `(add1 (o+ x (sub1 y)))`.*
- Okay. What's the value of `(o+ x (sub1 y))`?
- *That's a good question. We need to first find the answer of `x` and `(sub1 y)`.*
- What's the value of `x`? And what's the value of `(sub1 y)`?
- *The value of `x` is 2, and the value of `(sub1 y)` is 0.*
- All right. We need to find the value of `(o+ 2 0)` then.
- *What does this mean?*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- We enter the *lambda expression* again, but the value of `x` and `y` have changed!
- *Yes. the value of x is now 2, and the value of y is now 0.*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- We enter the *lambda expression* again, but the value of `x` and `y` have changed!
- *Yes. the value of x is now 2, and the value of y is now 0.*
- What's the value of `(zero? y)` now?
- `#t`.

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- We enter the *lambda expression* again, but the value of `x` and `y` have changed!
- *Yes. the value of x is now 2, and the value of y is now 0.*
- What's the value of `(zero? y)` now?
- `#t`.
- What do we get then?
- *The value of `(o+ 2 0)` is now the value of x, which is 2.*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- Are we done yet?

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- Are we done yet?
- *Nope. We still need to `add1` to the 2. Remember that `else`?*

Recur, recur, recur...

```
(define o+
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (add1 (o+ x (sub1 y)))))))
```

- Are we done yet?
- *Nope. We still need to `add1` to the 2. Remember that `else`?*
- Fine. Is that all?
- *Yes, and we finally get the 3.*

Recur, recur, recur...

- Now, let's try to write o-.
- *This is easy. Remember that we only need to deal with natural numbers.*

```
(define o-
  (lambda (x y)
    (cond
      (___________)
      (else (____ (___________))))))
```

Recur, recur, recur...

```
(define o-
  (lambda (x y)
    (cond
      (___________)
      (else (____ (___________))))))
```

- What do we do at the first line after `cond`?

```
(define o-
  (lambda (x y)
    (cond
      (___________)
      (else (____ (___________))))))
```

- What do we do at the first line after `cond`?
- *This is the same as before. We ask (`zero? y`) and if it is true, we get x.*

```
(define o-
  (lambda (x y)
    (cond
      (___________)
      (else (____ (___________))))))
```

- What do we do at the first line after `cond`?
- *This is the same as before. We ask (`zero? y`) and if it is true, we get x.*
- Isn't that exactly the same?
- *Yes. When dealing with numbers, always ask first if a number is zero.*

Recur, recur, recur...

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (____ (___________))))))
```

- What next?

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (____ (___________))))))
```

- What next?
- *Isn't that still very similar as before? We only need to change a tiny bit.*

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (____ (___________))))))
```

- What next?

Recur, recur, recur...

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (____ (___________))))))
```

- What next?
- *Isn't that still very similar as before? We only need to change a tiny bit.*

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (sub1 (o- x (sub1 y)))))))
```

- Is this okay?
- *Yes.*

Recur, recur, recur...

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (sub1 (o- x (sub1 y)))))))
```

- Isn't the following one also okay?

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

Recur, recur, recur...

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (sub1 (o- x (sub1 y)))))))
```

- Isn't the following one also okay?

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- *It seems so. but aren't they essentially the same?*
- Not quite, but we'll see it soon™.

Building upon

- Now, let's try to write `o*`.
- *Doesn't this require the `o+` that we have just written?*
- Yes, exactly.

- Now, let's try to write `o*`.
- *Doesn't this require the `o+` that we have just written?*
- Yes, exactly.

```
(define o*
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (+ x (o* x (sub1 y)))))))
```

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (sub1 (o- x (sub1 y)))))))
```

- Suppose we're using the definition above. What's left to do when `(zero? y)` finally becomes true when we're looking for the answer of `(o- 5 5)`?

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (sub1 (o- x (sub1 y)))))))
```

- Suppose we're using the definition above. What's left to do when `(zero? y)` finally becomes true when we're looking for the answer of `(o- 5 5)`?
- *It seems that we have a lot of `(sub1 ...)` to evaluate.*

Too many things to remember

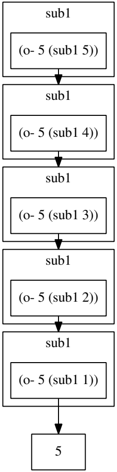


Figure: What we need to remember.

Too many things to remember

- What's the problem with this?
- *We need to remember how many `sub1`'s we need to evaluate.*
- What if we have too many things to remember?

Too many things to remember

- What's the problem with this?
- *We need to remember how many `sub1`'s we need to evaluate.*
- What if we have too many things to remember?
- *We run out of memory. Technically the graph that you saw illustrates what is called a stack and the situation that we run out of memory is called a stack overflow.*

Too many things to remember

- What's the problem with this?
- *We need to remember how many `sub1`'s we need to evaluate.*
- What if we have too many things to remember?
- *We run out of memory. Technically the graph that you saw illustrates what is called a stack and the situation that we run out of memory is called a stack overflow.*
- Is there any way to avoid this situation?

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- Now let's look at this definition of `o-`.
- *Isn't it only slightly different at the last line?*

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- Now let's look at this definition of `o-`.
- *Isn't it only slightly different at the last line?*
- Yes. But what would be the stack look like?

Too many things to remember

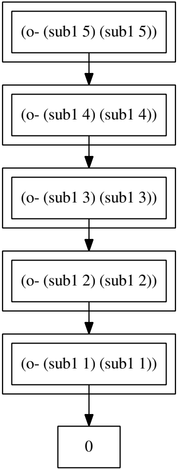


Figure: What we need to remember instead now.

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?
- *We don't need to remember how many `sub1`'s we have to do now.*

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?
- *We don't need to remember how many* `sub1`*'s we have to do now.*
- Do we really need to remember anything?

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?
- *We don't need to remember how many `sub1`'s we have to do now.*
- Do we really need to remember anything?
- *No, because we do not need to perform further calculations. All we need to do is to return the value.*

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?
- *We don't need to remember how many `sub1`'s we have to do now.*
- Do we really need to remember anything?
- *No, because we do not need to perform further calculations. All we need to do is to return the value.*
- Does this mean that we won't run out of memory?

Too many things to remember

```
(define o-
  (lambda (x y)
    (cond
      ((zero? y) x)
      (else (o- (sub1 x) (sub1 y))))))
```

- What becomes different this time?
- *We don't need to remember how many `sub1`'s we have to do now.*
- Do we really need to remember anything?
- *No, because we do not need to perform further calculations. All we need to do is to return the value.*
- Does this mean that we won't run out of memory?
- *Yes. When we call a function where no further calculation needs to be done, it becomes a tail call. The proper forgetting that prevents us from running out of memory is called tail-call optimization.*

Moving on to a higher order

- What's the answer of `(1 2)`?

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?
- *No answer.* `foo` *is not a function yet.*

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?
- *No answer.* `foo` *is not a function yet.*
- What's the answer of `(quote (1 2))`?

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?
- *No answer.* `foo` *is not a function yet.*
- What's the answer of `(quote (1 2))`?
- *`(1 2)`*.

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?
- *No answer. `foo` is not a function yet.*
- What's the answer of `(quote (1 2))`?
- *`(1 2)`.*
- What's the answer of `'(1 2)`?

Moving on to a higher order

- What's the answer of `(1 2)`?
- *No answer. 1 is not a function that accepts 2 as an argument.*
- What's the answer of `(foo 2)`?
- *No answer.* `foo` *is not a function yet.*
- What's the answer of `(quote (1 2))`?
- *`(1 2)`.*
- What's the answer of `'(1 2)`?
- *`(1 2)`. Same as before.*

Moving on to a higher order

- What's the answer of `(atom? 1)`?
- `#t`.

Moving on to a higher order

- What's the answer of (`atom? 1`)?
- `#t`.
- What's the answer of (`atom? '()`)?
- `#f`, *because it is an empty list.*

Moving on to a higher order

- What's the answer of `(atom? 1)`?
- `#t`.
- What's the answer of `(atom? '())`?
- `#f`, *because it is an empty list.*
- What's the answer of `(atom? '(1 (2)))`?
- `#f`, *because it is a list.*

Moving on to a higher order

- What's the answer of `(atom? 1)`?
- `#t`.
- What's the answer of `(atom? '())`?
- `#f`, *because it is an empty list.*
- What's the answer of `(atom? '(1 (2)))`?
- `#f`, *because it is a list.*
- What's the answer of `(null? '())`?
- `#f`, *because it is an empty list.*

Moving on to a higher order

- What's the answer of `(atom? 1)`?
- `#t`.
- What's the answer of `(atom? '())`?
- `#f`, *because it is an empty list.*
- What's the answer of `(atom? '(1 (2)))`?
- `#f`, *because it is a list.*
- What's the answer of `(null? '())`?
- `#f`, *because it is an empty list.*
- What's the answer of `(null? '(1 (2)))`?
- `#f`, *because it is a non-empty list.*

Moving on to a higher order

- What's the answer of `(car '())`?
- *No answer, because it's an empty list.*

Moving on to a higher order

- What's the answer of `(car '())`?
- *No answer, because it's an empty list.*
- What's the answer of `(car '(1 (2)))`?
- *1.*

Moving on to a higher order

- What's the answer of `(car '())`?
- *No answer, because it's an empty list.*
- What's the answer of `(car '(1 (2)))`?
- *1.*
- What's the answer of `(cdr '())`?
- *No answer, because it's an empty list.*

Moving on to a higher order

- What's the answer of `(car '())`?
- *No answer, because it's an empty list.*
- What's the answer of `(car '(1 (2)))`?
- *1.*
- What's the answer of `(cdr '())`?
- *No answer, because it's an empty list.*
- What's the answer of `(cdr '(1 (2)))`?
- `'((2))`. *Note that it's different from* `'(2)` !

Moving on to a higher order

- What's the answer of `(cons 1 '((2)))`?
- *`'(1 (2))`*.

Moving on to a higher order

- What's the answer of `(cons 1 '((2)))`?
- *'(1 (2))*.
- Can we build up the list from the empty list?

Moving on to a higher order

- What's the answer of `(cons 1 '((2)))`?
- *'(1 (2))*.
- Can we build up the list from the empty list?
- *(cons 1 (cons (cons 2 '()) '()))*.

Moving on to a higher order

- Define a function that removes all numbers equal to x in a list of atoms.
- *Sure.*

```
(define multirember
  (lambda (x lat)
    (cond
      ((null? lat) '())
      ((= (car lat) x) (multirember x (cdr lat)))
      (else (cons (car lat) (multirember x (cdr lat)))))))
```

Moving on to a higher order

- Define a function that removes all numbers equal to `x` in a list of atoms.
- *Sure.*

```
(define multirember
  (lambda (x lat)
    (cond
      ((null? lat) '())
      ((= (car lat) x) (multirember x (cdr lat)))
      (else (cons (car lat) (multirember x (cdr lat))))))
```

- What are the steps to go through when dealing with a list of atoms?
- We always ask `(null? lat)` first, then ask other questions.
- What if we're dealing with list of lists?

Moving on to a higher order

- Define a function that removes all numbers equal to x in a list of atoms.
- *Sure.*

```
(define multirember
  (lambda (x lat)
    (cond
      ((null? lat) '())
      ((= (car lat) x) (multirember x (cdr lat)))
      (else (cons (car lat) (multirember x (cdr lat)))))))
```

- What are the steps to go through when dealing with a list of atoms?
- We always ask (null? lat) first, then ask other questions.
- What if we're dealing with list of lists?
- We ask (null? l), (atom? (car l)) and other questions.

Moving on to a higher order

- Could you define a function that removes all numbers less than `x`?
- *Isn't this easy?*

```
(define multirember2
  (lambda (x lat)
    (cond
      ((null? lat) '())
      ((< (car lat) x) (multirember x (cdr lat)))
      (else (cons (car lat) (multirember x (cdr lat)))))))
```

Moving on to a higher order

- Could you define a function that removes all numbers greater than `x`?
- *Isn't this also easy?*

```
(define multirember2
  (lambda (x lat)
    (cond
      ((null? lat) '())
      ((> (car lat) x) (multirember x (cdr lat)))
      (else (cons (car lat) (multirember x (cdr lat)))))))
```

Moving on to a higher order

- Is it really so easy as you said?
- *What's so hard about this? we just copy the whole definition and change what we want.*

Moving on to a higher order

- Is it really so easy as you said?
- *What's so hard about this? we just copy the whole definition and change what we want.*
- Is there a way to not copy the whole thing?
- *What does it mean?*

Moving on to a higher order

- Is it really so easy as you said?
- *What's so hard about this? we just copy the whole definition and change what we want.*
- Is there a way to not copy the whole thing?
- *What does it mean?*
- Look at this function definition.

```
(define mr-f
  (lambda (tester)
    (lambda (x lat)
      (cond
        ((null? lat) '())
        ((tester (car lat) x) ((mr-f tester) x (cdr lat)))
        (else (cons (car lat) ((mr-f tester) x (cdr lat))))))))
```

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?
- `'(2 3)`.

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?
- `'(2 3)`.
- What's the answer of `((mr-f >) 2 '(1 2 3))`?

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?
- `'(2 3)`.
- What's the answer of `((mr-f >) 2 '(1 2 3))`?
- `'(1 2)`.

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?
- `'(2 3)`.
- What's the answer of `((mr-f >) 2 '(1 2 3))`?
- `'(1 2)`.
- What's so special about `mr-f`?

Moving on to a higher order

- What's the answer of `((mr-f =) 2 '(1 2 3))`?
- `'(1 3)`.
- What's the answer of `((mr-f <) 2 '(1 2 3))`?
- `'(2 3)`.
- What's the answer of `((mr-f >) 2 '(1 2 3))`?
- `'(1 2)`.
- What's so special about `mr-f`?
- *It accepts another function as an argument. Such functions are called higher-order functions.*

Add some curry

- Let's write the plus function slightly differently.
- *(define o-plus (lambda (x) (lambda (y) (+ x y))))*.
- Can we write all functions with multiple arguments in this way?

Add some curry

- Let's write the plus function slightly differently.
- *(define o-plus (lambda (x) (lambda (y) (+ x y))))*.
- Can we write all functions with multiple arguments in this way?
- *Sure. Why not? But What's the difference?*
- Suppose we have the following function:

```
(define maaaap
  (lambda (f)
    (lambda (lat)
      (cond
        ((null? lat) '())
        (else (cons (f (car lat))
                    ((maaaap f) (cdr lat))))))))
```

- How to add 2 to every element of `'(1 2 3)`?

Add some curry

```
(define maaaap
  (lambda (f)
    (lambda (lat)
      (cond
        ((null? lat) '())
        (else (cons (f (car lat))
                    ((maaaap f) (cdr lat))))))))
```

- How about this: `((maaaap (oplus 1)) '(1 2 3))`?

Add some curry

```scheme
(define maaaap
  (lambda (f)
    (lambda (lat)
      (cond
        ((null? lat) '())
        (else (cons (f (car lat))
                    ((maaaap f) (cdr lat))))))))
```

- How about this: `((maaaap (oplus 1)) '(1 2 3))`?
- *Yes, it works. Can we multiply it by 3?*
- Sure. This is left for you as an exercise.

Add some curry

```
(define maaaap
  (lambda (f)
    (lambda (lat)
      (cond
        ((null? lat) '())
        (else (cons (f (car lat))
                    ((maaaap f) (cdr lat))))))))
```

- How about this: `((maaaap (oplus 1)) '(1 2 3))`?
- *Yes, it works. Can we multiply it by 3?*
- Sure. This is left for you as an exercise.
- Does the process of taking functions apart into many lambdas get a name?
- *Yes. It's named Currying in honor of Haskell B. Curry.*

- We put the definition of o= here for convenience.

```
(define o=
  (lambda (x y)
    (cond
      ((zero? x) (zero? y))
      ((zero? y) #f)
      (else (o= (sub1 x) (sub1 y))))))
```

Now, try to write the Fibonacci function.

Are they really the same?

```
(define fibonacci
  (lambda (x)
    (cond
      ((zero? x) 1)
      ((zero? (sub1 x) 1))
      (else ...))))
```

- The code above is a good place to start.

Are they really the same?

```scheme
(define fibonacci
  (lambda (x)
    (cond
      ((zero? x) 1)
      ((zero? (sub1 x) 1))
      (else ...))))
```

- The code above is a good place to start.
- *Doesn't the following definition look natural?*

```scheme
(define fib
  (lambda (x)
    (cond
      ((zero? x) 1)
      ((zero? (sub1 x)) 1)
      (else (+ (fib (sub1 x)) (fib (- x 2)))))))
```

Are they really the same?

- It might look natural, but it's definitely not the optimal.
- *What does it mean?*

Are they really the same?

- It might look natural, but it's definitely not the optimal.
- *What does it mean?*
- See how long it takes for `(fib 10000)` to work out.

Are they really the same?

- How about this definition:

```
(define fibonacci-helper
  (lambda (r prev pprev)
    (cond
      ((zero? r) (+ prev pprev))
      (else (fibonacci-helper (sub1 r) (+ prev pprev) prev)))
(define fibonacci-alt
  (lambda (x)
    (fibonacci-helper x 0 1)))
```

Are they really the same?

- How about this definition:

```
(define fibonacci-helper
  (lambda (r prev pprev)
    (cond
      ((zero? r) (+ prev pprev))
      (else (fibonacci-helper (sub1 r) (+ prev pprev) prev))
(define fibonacci-alt
  (lambda (x)
    (fibonacci-helper x 0 1)))
```

- Do we need to be careful about how fast our program runs?

Are they really the same?

- How about this definition:

```
(define fibonacci-helper
  (lambda (r prev pprev)
    (cond
      ((zero? r) (+ prev pprev))
      (else (fibonacci-helper (sub1 r) (+ prev pprev) prev)
(define fibonacci-alt
  (lambda (x)
    (fibonacci-helper x 0 1)))
```

- Do we need to be careful about how fast our program runs?
- *Absolutely.*

Thank you!