Puzzle:
Can we overwrite lambda?
I.e. **(define (lambda n) (* n n))** ?

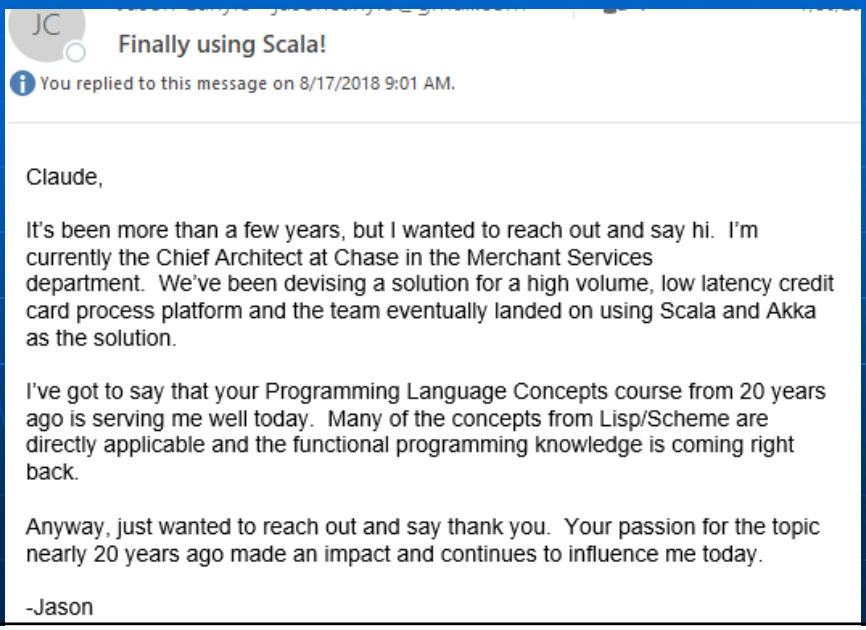
CSSE 304 Day 3

Announcements
Call Roll
Instructor Intro (different slides)
Course Intro
Recursion practice

Instructor Intro

- On separate slides.
- For more details, see the video on Moodle.

An email I received last summer



JC

Finally using Scala!

You replied to this message on 8/17/2018 9:01 AM.

Claude,

It's been more than a few years, but I wanted to reach out and say hi. I'm currently the Chief Architect at Chase in the Merchant Services department. We've been devising a solution for a high volume, low latency credit card process platform and the team eventually landed on using Scala and Akka as the solution.

I've got to say that your Programming Language Concepts course from 20 years ago is serving me well today. Many of the concepts from Lisp/Scheme are directly applicable and the functional programming knowledge is coming right back.

Anyway, just wanted to reach out and say thank you. Your passion for the topic nearly 20 years ago made an impact and continues to influence me today.

-Jason

Course Intro

- Education is not a spectator sport.
- You can only learn a little bit by watching me.
- Most of what you learn will be because of what you read for yourself, think for yourself, code for yourself.
- Don't take the "Here I am, teach me!" approach!

What the course is NOT called:

- Let's superficially learn about 5 new languages.
  - Which ones should we study?
  - Which are going to be most important in 20 years? 40 years?
  - How many people think you will program in C, C++, Python, PHP, or Java in 2050?
  - Actually, you probably will!
  - **Old languages never die, they just mutate to include new paradigms.**

I want you to become a linguist

- A linguist usually knows only few languages well, but she knows bits and pieces of other languages.
- What distinguishes the linguist is knowing **principles** behind languages.
- My dictionary defines *linguist* as "a person who studies the history and structure of language."

What to cover?

- "PLC" is such a broad area that we could easily offer 3 courses on just the elementary concepts central to programming languages today, with almost no intersection. (PLP)
- Some hard choices have to be made
- Will you like my choices?
  - No matter what choices I make, many important things will be left out.

Anatomy vs. Physiology

from Encyclopedia Americana:

Physiology is the study of processes common to all living organisms, as well as those special to particular groups of animals and plants.

Traditionally physiology has focused on the **FUNCTIONS** of these processes, relying on experimental methods to observe the processes under controlled conditions.

A visit to the zoo:

Feeding the animals vs. making a giraffe-tiger

Why Scheme for CSSE304?

- In PLC, Scheme is not an end in itself.
- **Scheme will serve two purposes in this course:**
  1. A place to see new programming concepts without having to learn syntax of lots of languages.
  2. A laboratory environment in which to better understand PL concepts by implementing them in our own interpreters.
- After a steep initial language learning curve, the overhead cost of introducing each new programming concept or paradigm in Scheme is low.

Why **start** the course with Scheme?

- In an introductory Chemistry lab, the first couple of weeks would mainly involve
  - Learning to use the lab equipment efficiently and safely
  - while doing a little bit of real chemistry
- **Scheme-a-thon in this course is similar.**
- At the end of that time, you should be comfortable with the Lab Equipment
  - Scheme should then be a springboard (not a barrier) when we use it later
  - During the Scheme introduction, we'll also encounter several important new PL concepts

Read the textbooks

- Most of TSPL is easy to read. But there are exceptions.
- You may need ultiple passes through some parts of EoPL .
  - Perhaps a few days between readings.
- Reading assignments are usually a little bit ahead of lectures.
  - Because of the above.

I won't pretend!

- This is a difficult course for some people.
- **Intellectual level is high.**
- Don't let it get away from you.
- **You'll have to "keep at it".**
  - as I should have done in Math 120 at Caltech. More on that soon!
- **Especially if it was a real struggle** for you to earn a B in CSSE 230.
  - OOTH, some people who did not do well in 230 do very well in 304!

Easy and hard problems

- Some problems are easy, some are hard.
- In some RHIT courses, there seems to be the expectation that all students will get all of the problems. Not here!
- In this class, I expect that all students will get most of the problems if they work hard at them.
- But there will be a few problems that only a few students will get.
- All students will learn something by *trying* all problems.
- You will sometimes need to cry "uncle" and move on to a different problem.
- Don't give up too easily, but don't be afraid to get help or to simply move on occasionally.

Thinking outside the box

- Different languages support many approaches (paradigms) to programming.
- Develop a mindset that welcomes and evaluates new paradigms.
  - "If it's not in Java, …" 1975 – 1997 - 2014

The best language is _____

- If you leave this course thinking that there is, will be, or ought to be a "holy grail" of languages--a language that will be all things for all people for all purposes...
  - ... then I will have failed.

Delayed reactions

- Over the years, many alumni have come to me and said:
  - I didn't 'get it' while I was taking 304,
  - but soon after graduation I used many of the things from this course.
  - Then I understood the value of it.

Some of my goals for you

- I want you to learn some PL terminology
- I want you to learn to ask (and sometimes answer) the "why" questions about language features
- I want you to come out of this course saying, "I have had the opportunity to think about programming in some very new ways"

You have things to contribute

- Many of you have used languages that I have not used
  - or language features that I have not used
- Your perspectives may add to the BoK (Body of Knowledge) in this course
  - Please don't be stingy!
  - Share your perspectives, ask questions, etc.

End of Course Intro

- Back to our regularly scheduled Scheme intro…

The "similar example in Java" Question from A0 hand-in

```
public class TryStringEqual {
    public static void main(String[] args) {
        int[] a = {1, 2}, b = {1, 2};
        String c = "abc", d = "abc";
        System.out.println("Arrays:    "+ (a==b));
        System.out.println("Strings:   "+ (c==d));
    }
}
```

```
> (eq? "abc" "abc")
#f
> (eq? 'abc 'abc)
#t
```

```
Arrays:    false
Strings:   true
```

https://en.wikipedia.org/wiki/String_interning

In computer science, **string interning** is a method of storing only one copy of each distinct string value, which must be immutable.[1] Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a **string intern pool**.

The single copy of each string is called its *intern* and is typically looked up by a method of the string class, for example String.intern()[2] in Java. All compile-time constant strings in Java are automatically interned using this method.[3]

String interning is supported by some modern object-oriented programming languages, including Java, Python, PHP (since 5.4), Lua,[4] Ruby (with its symbols), Julia and .NET languages.[5] Lisp, Scheme, and Smalltalk are among the languages with a symbol type that are basically interned strings. The library of the Standard ML of New Jersey contains an `atom` type that does the same thing. Objective-C's selectors, which are mainly used as method names, are interned strings.

Recap - Predicates

- What's a predicate?
- **How can you usually recognize that a given procedure is a predicate?**
- **eq?** vs **equal?**
- **eqv?** From TSPL:
  - **eq?** cannot be used to compare numbers and characters reliably. Although every inexact number is distinct from every exact number, two exact numbers, two inexact numbers, or two characters with the same value may or may not be identical (i.e., not **eq?**)
  - **eq?** is cheaper than **eqv?**

Some A1 solutions

```
; #4  interval-union
(define interval-union
  (lambda (i1 i2)
    (if (interval-intersects? i1 i2)
        (list (list (min (first i1) (first i2))
                    (max (second i1) (second i2))))
        (list i1 i2))))
```

```
(define vec-length
  (lambda (v)
    (sqrt (dot-product v v))))
```

```
(define distance
  (lambda (p1 p2)
    (vec-length (make-vec-from-points p1 p2))))
```

What is common to **all** procedures?

- What is it that every procedure application always does?
  - evaluates procedure and arguments first
  - **In which order?**
- Not necessarily true of non-procedures.
  - (quote x) ; x is **not** evaluated.
  - (define x 3) ; x is **not** evaluated.
  - (if x y z) ; either y or z is **not** evaluated.
  - (or x y z) ; y and z may **not** be evaluated.
  - (lambda (x) (+ x 3)) ; x is **not** evaluated.

fact example 1

```
> (define fact
    (lambda (n)
      (cond
       [(zero? n)  1]
       [else (* n (fact (- n 1)))])))
> (fact 4)
24
> (fact  -2)
 C-c C-c
break>q
```

Escape from infinite loop by repeatedly pressing ctrl-c

```
> (trace fact fact2 fact-acc)
(fact fact2 fact-acc)
> (fact 4)
|(fact 4)
| (fact 3)
| |(fact 2)
| | (fact 1)
| | |(fact 0)
| | |1
| | 1
| |2
| 6
|24
24
```

Fact example 2

```
> (define fact2
    (lambda (n)
      (if (or (negative? n)
              (not (integer? n)))
          "error"
          (fact-acc n 1))))

> (define fact-acc
  (lambda (n acc)
    (if (zero? n)
        acc
        (fact-acc (- n 1)
                  (* n acc)))))
```

```
> (trace fact fact2 fact-acc)
(fact fact2 fact-acc)
> (fact2 4)
|(fact2 4)
|(fact-acc 4 1)
|(fact-acc 3 4)
|(fact-acc 2 12)
|(fact-acc 1 24)
|(fact-acc 0 24)
|24
24
```

Cond

- Similar to **if-elif-…-else** in other languages.

```
(define member?
    (lambda (a ls)
      (cond
       [(null? ls) #f]
       [(eq? (car ls) a) #t]
       [else (member? a (cdr ls))])
      ))
```

```
; cond is like if ... else if ... else

(define largest-in-list
  (lambda (L)
    (cond [(null? L)
           (errorf 'largest-in-list
                   "empty list has ~s "
                   "no largest element")]
          [(null? (cdr L)) (car L)]
          [else (max (car L)
                     (largest-in-list
                       (cdr L)))])))

; What's the efficiency issue with this?
```

```
; more efficient:
(define largest-in-list
  (lambda (ls)
    (if (null? ls)
      (errorf 'largest-in-list
              "list cannot be empty")
      (largest-in-non-empty ls))))


(define largest-in-non-empty
  (lambda (ls)
    (if (null? (cdr ls))
      (car ls)
      (let ([largest-in-cdr
             (largest-in-non-empty (cdr ls))])
        (if (> (car ls) largest-in-cdr)
            (car ls)
            largest-in-cdr)))))
```

Using max is simpler, but this is how we could do it if we did not have max.

```
; Now define another version with an accumulator
;    (that is also more robust)

(define largest-in-list
  (lambda (ls)
    (if (null? ls)
      (errorf 'largest-in-list "list cannot be empty")
      (largest-in-list-acc (cdr ls) (car ls)))))


(define largest-in-list-acc
  (lambda (ls largest-so-far)
    (cond [(null? ls) largest-so-far]
          [(not (number? (car ls)))
           (errorf 'largest-in-list
                   "everything in the list must be a number")]
          [(> (car ls) largest-so-far)
           (largest-in-list-acc (cdr ls) (car ls))]
          [else (largest-in-list-acc (cdr ls)
                                     largest-so-far)])))
```

Count reflexive pairs

- A **relation** is a set of ordered pairs; the set of all first elements is the **domain**. The set of all second elements is the range.
- We represent a relation by a list of 2-lists. A 2-list is a list whose length is 2.
- A **reflexive pair** is a 2-list whose first and last elements are the same.
- count-reflexive-pairs (work it out live)

More recursive procedures

- Sum of squares of the first n non-negative integers
- **(square-all ls)** returns a list of the squares of the numbers in ls.
- **(make-list n obj)** returns a list of n "copies" of obj. [If obj is a “by-reference” object, such as a list, it makes n copies of the reference].

We may not get to some (and maybe all) of this slide today. If that is the case, I suggest trying them for some simple recursion practice.