Programming Abstractions

About the class:

This is a course about Programming Languages

We will use the language Scheme to discuss, analyze and implement various aspects of programming languages.

The course has 3 parts:

A. Scheme and things you can do with it (5 weeks)
B. Implementing Scheme and other langauges (5 weeks)
C. Advanced issues (delayed evaluation, continuatons, etc.) (3 weeks)

A Quick History of Scheme

- John McCarthy invented LISP at MIT around 1960 as a language for AI.
- LISP grew quickly in both popularity and power. As the language grew more powerful it required more and more of a system's resources. By 1980 5 simulataneous LISP users would bring a moderately powerful PDP-11 to its knees.
- Guy Steele developed Scheme at MIT 1975-1980 as a minimalist alternative to LISP.
- Scheme is an elegant, efficient subset of LISP. It has some nice properties that we will look at that allow it to be implemented efficiently. For example, most recursions in Scheme turn into loops.

Why Scheme?

- All LISP-type languages have lists as their main data structure; their programs are written in lists. This means it is easy to have programs take other programs as input. Scheme programs can reason about other programs. This makes Scheme useful for thinking about programming languages in general.
- Scheme is a different programming paradigm. Java, C and other languages are <u>imperative</u> languages. Programs in these languages do their work by changing data stored in variables. Scheme programs can be written as <u>functional</u> programs -- they compute by evaluating functions and avoid variable assignments.
- Scheme is very elegant. It is much less verbose than Java, which means it is easier to see what is happening in a Scheme program.
- It is fun!

The course

- We will have about 10 labs (I call them labs; they are assignments you do on your own). You will have between 7 and 10 days for each lab. We will have one exam around the end of February, another in mid April, and a final exam.

- It is very important that you do the labs on time. This course differs from 151 in that each lab has a lot of small, independent parts. You can get some of it done and running even if it isn't all done. So hand the lab in on time, even if it isn't completely finished. If you fall behind in the labs you won't follow the lectures and you will be lost.

We will use Dr. Racket as our Scheme interpreter. The history of this is rather tortured. First there was a group called PLT (Programming Langauage Theory) containing Matthias Felleisen, Shriram Krishnamurthy, Robby Findler and others. They produced a Scheme interpreter called Dr. Scheme. Over the years a catfight developed over what features belonged in "real" Scheme. Eventually the PLT group started calling its language "Racket" instead of Scheme. The interpreter changed its name from "Dr. Scheme" to "Dr. Racket". Feelings were hurt. No one says "Racket is really Scheme", though it is.

You can obtain Racket for free (Windows, Mac or Linux). See
    www.racket-lang.org

Look at the syllabus.

There is a Blackboard site for the class where I will post the class notes and examples and the labs.

For this week:

- Read Chapters 1 - 5 of The Little Schemer
- Do Lab 1; this is due on Monday, Feb. 10.

Begin Scheme

There are two kinds of expressions in Scheme: S-expressions and Procedures. Procedures are functions; they take arguments and return values.

An S-expression is either an *atom* or a *list*.

An atom can be

- A symbol
- A number
- #t or #f (Boolean values)
- A string (which we will seldom use)

A list is either

- null
- or a pair consisting of a head and a tail. The head must be an S-expression and the tail must be a list.

The Scheme interpreter evaluates expressions.

- The value of a number, #t, #f, or a string is the atom itself.
- The value of a symbol is the value bound to it.
- the value of a procedure is a closure; we'll talk about this later.
- The value of a null list is null.
- The value of a non-null list is the result of calling the head of the list as a procedure with the tail of the list as its arguments. For example, the value of

  ```
  (+ 3 4)
  ```

  is 7.
- Note that we can't evaluate the list (1 2 3) because the head of this list, 1, is not a procedure.

The quote ' is used to prevent evaluation.

'(1 2 3) evaluates to the list (1 2 3)

Basic Scheme procedures:

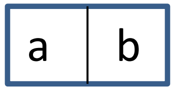
- car (Contents of the Address Register on an old IBM 704) (pronounced "car", like an automobile)
- cdr (Contents of the Decrement Register on that 704) (pronounced "could - er"; rhymes with "should stir")
- cons (Construct)

By the way, the IBM 704, introduced in 1954, was the first commercial computer with floating-point hardware.   Transistors were just being invented; the 704 was a vacuum tube computer.

Here are the meanings:

- (cons a b) creates a new pair, where a is the head and b is the tail. If b is a list, this makes a new list.

  (cons a b) creates a *cons-box*

- (car x) is an error unless x is a pair, in which case (car x) is the head of x.
- (cdr x) is an error unless x is a pair, in which case (cdr x) is the tail of x.

Procedures always evaluate their arguments before performing their actions:

> (car (1 2 3)) is an error because the argument (1 2 3)
>> can't be evaluated.
>
> (car '(1 2 3)) performs the car procedure on the value of
>> the argument, which is the list (1 2 3).
>> The car of this list is 1, so
>> (car '(1 2 3)) => 1.

Examples

- (cons 3 null) => (3)
- (cons 2 (cons 3 null)) => (2 3)
- (cons '(1 2) '(3 4)) => ((1 2) 3 4)
- (car '((1 2) (3 4 5 6)) => (1 2)
- (cdr '(1 2 3)) => (2 3)

cadr is shorthand for "car of the cdr"

```
        (cadr '(1 2 3 4)) => (car (cdr '(1 2 3 4)))
                           => (car '(2 3 4))
                          => 2
```

In other words (cadr x) is the second element of list .

Similarly there are procedures caddr, cadddr, etc.

define changes the global environment by binding a value to a symbol.

e.g. (define Beatles '(john paul george ringo))

The most common things to define are procedures.

Procedures are created with lambda-expressions.

```
        (lambda (parameter-list) body)
```

e.g.

```
        (lambda (x) (+ x 5))
```

or

```
        (lambda () 23)
```

For example

```
(define f (lambda (x y) (+ (* x y) 1)))

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

Note that define binds symbols to values, not to expressions.

```
(define f (lambda (x) (+ x 1)))
(define bob (f 0))
(define f (lambda (x) (* x x )))

bob => 1,  not 0
```

There are two *conditional* expressions:

```
(if <test>  <test-true exp> <test-false exp>)

(cond  [test1 exp1] [test2 exp2] ... )
```

you can use the symbol *else* for the final test.

E.g.

```
(if (< 1 2) 3 4) => 3

(cond [(< 2 1) 3] [(< 5 6) 4] [else 5]) => 4
```

We can put all of this together to get our first interesting procedure:

```
(define f (lambda (x)
        (cond
                [(= x 1) 1]
                [else (* x (f (- x 1)))])))
```