INTRODUCTIONS, OVERVIEW, SCHEME 1

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June 21, 2010

1 Overview

My name is George Wang, and we've got a great staff of TA's, Readers, and LA's to help all of you this semester. Please take a minute to say 'hi' to me and them in Lab later today :). Before I go over course content and administrivia, I'd like to jump straight into the course.

2 The Big Picture

Computer Science is an interesting name for the field, since really, it is neither about computers nor is it a science. If you want to study the construction and design of computers, you want to be studying electrical engineering. And excepting more theoretical elements, computer science is much more of of an abstract engineering discipline. Abstract, in the sense that in most fields you have to be concerned with the physical limitations of the world. In Computer Science, we can deal with problems devoid of the constraints of reality. This is something I think is really great.

We're not teaching you Scheme, since as I said, you'll know Scheme by tomorrow. We're teaching Computer Science. This is analogous to teaching Literature through Shakespeare or Milton, or whatever. And the central idea to this course and CS and much of engineering is Abstraction.

Consider this. Maybe, but hopefully not, you drive a car to school. To speed up, you press the gas pedal. To slow down, you hit the brakes. To turn you use the steering wheel. But what's amazing is that once you learn these things, they're the same whether you're driving a hybrid or a truck. What the brakes are doing is utterly different. What the gas pedal is doing is mindblowingly complicated. But the point is, that you don't care. That is the power of abstraction. All you need to know is that gas pedal = speed. You don't think about the fuel injectors injecting gas and air engine, or each individual molecule forming in contained explosions driving pistons which give the car your motion. You don't think about all of that, because if you did, you wouldn't be able to actually do anything. You would be so focused on thinking about the details that you miss the big picture.

But the abstraction are not limited to one single layer. What you should think about is the idea of layers of abstraction. The engineers who build the engine don't have to consider in detail the synthesis and analysis of the carbon chains that forms the gas. The chemists that study that don't have to deal with the subatomic particles that in turn compose those. Thus, these layers of abstraction keep you sane as you are trying to tackle these topics.

3 Scheme

In this class, we've settled on teaching CS through the medium of a language called Scheme. Students have often asked why we use Scheme and not something more powerful and cooler like Python, or Java, or C, or whatever is popular in the day. One big reason is simple: by the end of today you will know virtually all the rules of Scheme. However, knowing rules and understanding rules are fundamentally different, much like knowing the rules of chess and understanding the implications of these rules are very very different.

Scheme is a interactive language. That means that instead of writing a great big program and then cranking it through all at once, you can type in a single expression and find out its value. For example:

`3`	self evaluating
`(+ 2 3)`	function notation
`(sqrt 16)`	names don't have to be punctuation
`(+ (* 3 4) 5)`	composition of functions
`+`	functions are things in themselves
`'+`	quoting
`'hello`	can quote any word
`'(+ 2 3)`	can quote any expression
`'(good morning)`	quote non-expression sentences
`(first 274)`	functions dont have to be arithmetic
`(butfirst 274)`	(abbreviated bf)
`(first 'hello)`	works for non-numbers
`(first hello)`	reminder about quoting
`(first (bf 'hello))`	composition of non-numeric functions
`(+ (first 23) (last 45))`	combining numeric and nonnumeric
`(define pi 3.14159265)`	special form
`pi`	value of a symbol
`'pi`	contrast with quoted
`(+ pi 7)`	symbols work in expressions
`(* pi pi)`	

4 Functions

Now, let's talk about functions. You've heard of these from grade school. Functions here are essentially the same thing. First, let's look at how to do them in Scheme:

`(define (square x)` `(* x x))`	defining a function
`(square 5)`	invoking the function
`(square (+ 2 3))`	composition with defined functions

Here's a few things to note:

- Procedures have something called formal parameters and a body. In the case above, the parameter is `x`, and the body is `(* x x)`. There is also a difference between functions and procedures. Take for example:

  $$\begin{aligned} f(x) &= 3x + 6 \\ g(x) &= 3(x+2) \end{aligned}$$

  f and g are the same *function*, but they are different *procedures*. In other words, procedures is a set of instructions to compute a function. Most of the time, these won't matter, but occasionally, they will, and I'll try to be careful when I can.

- A function can have any number of arguments, including none. However, they must have exactly one return value. It's not a function unless you always get the same answer for the same arguments.

- If each little computation is independent of the past history, then we can *reorder* the computation. In particular, this is great for parallelism where we may have many processors dividing up the work in such a way as to be both efficient and correct.

- It is not about a sequence of events! We don't do programming like:

  ```
  (do-thing-1)
  (do-thing-2)
  (do-thing-3)
  ```

- Instead, we do composition of functions. Imagine things like $f(g(x))$, like in high school. We can represent this visually as well.

5 Words and Conditionals

Let's try to write a procedure that returns the plural of a word:

```
(define (plural wd)

    (word wd 's))
```

`Word` is a procedure that builds a word given its arguments. This example works for a lot of simple words, such as word, work, book, elephant, but not for words that end in y. Thus, we will include it by introducing a conditional, `if`.

```
(define (plural wd)
  (if (equal? (last wd) 'y)
      (word (bl wd) 'ies)
      (word wd 's)))
```

`If` is a special form that only evaluates one of the two alternatives. `Equal?` is a procedure that given two arguments, tests if they are equal.
Finally, let's examine one more special form, `cond`.

```
(define (buzz n)
  (cond ((equal? (remainder n 7) 0) 'buzz)
        ((member? 7 n) 'buzz)
        (else n)))
```

This is an example of what we call syntactic sugar. The two conditionals are in fact the same thing. However, when you need to use multi-way decisions, this is much simpler.

6 Recursion

Let's examine this example, Pig Latin. Pig latin moves the initial consonants to the end of the word, and appends 'ay'. Thus, SCHEME becomes EMESCHAY.
Let's look at this in code:

```
;;;;; In file cs61a/lectures/1.1/pigl.scm
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))
(define (pl-done? wd)
  (vowel? (first wd)))
(define (vowel? letter)
  (member? letter '(a e i o u)))
```

So Recursion is the idea that it is a procedure that calls itself as a helper procedure. Before we start thinking this is that hard, let's think about something. A person's *ancestors* are defined as their parents and their parents' *ancestors*. Ta da!

7 Course Overview

I want to go over the overview of the course, as this course is structured much in the same way. We'll be starting at the bottom of Scheme. I'll start by showing you the atoms of Scheme. From there, we'll talk about the means of combining them into more powerful and more useful constructs. In the first week, we'll discuss the various ways to build up more and more complicated functions. We can think of functions as machines that operate on data. Weeks two and three, we then transition on describing what exactly we mean by data. By week 4, we proceed to look at how to think about larger codebases by using OOP. In week 5, we come back and talk about Scheme again, but in the context of how Scheme itself can be built. In week 6, we discuss some of the applications that can be done with Scheme. In week 7, as a parallel to week 4, we discuss having to harness procedures that are so large that they must run on many computers at once, and how to handle that. Finally, in week 8, we discuss alternate paradigms to programming, other than Scheme. Whew!