CS 61A Lecture Notes First Half of Week 1

Topic: Functional programming

Reading: Abelson & Sussman, Section 1.1 (pages 1–31)

Course overview:

Computer science isn't about computers (that's electrical engineering) and it isn't primarily a science (we invent things more than we discover them). CS is partly a form of engineering (concerned with building reliable, efficient mechanisms, but in software instead of metal) and partly an art form (using programming as a medium for creative expression). Most of all, however, CS is applied logic. At its best, CS is like getting logic and math to do interesting and useful things for you.

Programming is really easy, as long as you're solving small problems. Any kid in junior high school can write programs in BASIC, and not just exercises, either; kids do quite interesting and useful things with computers. But BASIC doesn't scale up; once the problem is so complicated that you can't keep it all in your head at once, you need help, in the form of more powerful ways of thinking about programming. (But in this course we mostly use small examples, because we'd never get finished otherwise, so you have to imagine how you think each technique would work out in a larger case.)

We deal with three big programming styles/approaches/paradigms:

- Functional programming (1 month)
- Object-oriented programming (2 weeks)
- Logic programming (1 week)

The big idea of the course is *abstraction*: inventing languages that let us talk more nearly in a problem's own terms and less in terms of the computer's mechanisms or capabilities. There is a hierarchy of abstraction:

```
Application programs High-level language (Scheme) Low-level
language (C) Machine language Architecture (registers, memory,
arithmetic unit, etc) circuit elements (gates) transistors
solid-state physics quantum mechanics
```

In 61A, we'll be dealing with only the very top levels of the pyramid; in 61C we look at lower levels. We want to start at the highest level to get you thinking right and help you avoid getting lost in the details.

Style of work: Cooperative learning. No grading curve, so no need to compete. Homework is to learn from; only tests are to test you. Don't cheat; ask for help instead. (This is the *first* CS course; if you're tempted to cheat now, how are you planning to get through the harder ones?)

Introducing ... Scheme

In 61A we program in Scheme, which is an *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)`	even non-expression sentences
`(first 274)`	functions don't have to be arithmetic
`(butfirst 274)`	(abbreviation `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 non-numeric
`(define pi 3.14159)`	special form
`pi`	value of a symbol
`'pi`	contrast with quoted symbol
`(+ pi 7)`	symbols work in larger expressions
`(* pi pi)`	
`(define (square x)` `  (* x x))`	defining a function
`(square 5)`	invoking the function
`(square (+ 2 3))`	composition with defined functions

Terminology: the *formal parameter* is the name of the argument (`x`); the *actual argument expression* is the expression used in the invocation (`(+ 2 3)`); the *actual argument value* is the value of the argument in the invocation (5). The argument's name comes from the function's definition; the argument's value comes from the invocation.

Examples:

```
(define (plural wd)
  (word wd 's))
```

This simple `plural` works for lots of words (book, computer, elephant) but not for words that end in `y` (fly, spy). So we improve it:

```
;;;;;                          In file cs61a/lectures/1.1/plural.scm
(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 alternatives.

Pig Latin: Move initial consonants to the end of the word and append "ay"; SCHEME becomes EMESCHAY.

```
;;;;;                          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)))
```

`Pigl` introduces *recursion*—a function that invokes itself. More about how this works later in the week.

Another example: Remember how to play Buzz? You go around the circle counting, but if your number is divisible by 7 or has a digit 7 you have to say "buzz" instead:

```
;;;;;                          In file cs61a/lectures/1.1/buzz.scm
(define (buzz n)
  (cond ((equal? (remainder n 7) 0) 'buzz)
        ((member? 7 n) 'buzz)
        (else n)))
```

This introduces the `cond` special form for multi-way choices.

`Cond` is the big exception to the rule about the meaning of parentheses; the clauses aren't invocations.

Functions.

• A function can have any number of arguments, including zero, but must have exactly one return value. (Suppose you want two? You combine them into one, e.g., in a sentence.) It's not a function unless you always get the same answer for the same arguments.

• Why does that matter? If each little computation is independent of the past history of the overall computation, then we can *reorder* the little computations. In particular, this helps cope with parallel processors.

• The function definition provides a formal parameter (a name), and the function invocation provides an actual argument (a value). These fit together like pieces of a jigsaw puzzle. *Don't write a "function" that only works for one particular argument value!*

• Instead of a sequence of events, we have composition of functions, like $f(g(x))$ in high school algebra. We can represent this visually with function machines and plumbing diagrams.

Recursion:

```
;;;;;                          In file cs61a/lectures/1.1/argue.scm
> (argue '(i like spinach))
(i hate spinach)
> (argue '(broccoli is awful))
(broccoli is great)

(define (argue s)
  (if (empty? s)
      '()
      (se (opposite (first s))
          (argue (bf s)))))

(define (opposite w)
  (cond ((equal? w 'like) 'hate)
        ((equal? w 'hate) 'like)
        ((equal? w 'wonderful) 'terrible)
        ((equal? w 'terrible) 'wonderful)
        ((equal? w 'great) 'awful)
        ((equal? w 'awful) 'great)
        ((equal? w 'terrific) 'yucky)
        ((equal? w 'yucky) 'terrific)
        (else w) ))
```

This computes a function (the `opposite` function) of each word in a sentence. It works by dividing the problem for the whole sentence into two subproblems: an easy subproblem for the first word of the sentence, and another subproblem for the rest of the sentence. This second subproblem is just like the original problem, but for a smaller sentence.

We can take `pigl` from last lecture and use it to translate a whole sentence into Pig Latin:

```
(define (pigl-sent s)
  (if (empty? s)
      '()
      (se (pigl (first s))
          (pigl-sent (bf s)))))
```

The structure of `pigl-sent` is a lot like that of `argue`. This common pattern is called *mapping* a function

over a sentence.

Not all recursion follows this pattern. Each element of Pascal's triangle is the sum of the two numbers above it:

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

Normal vs. applicative order.

To illustrate this point we use a modified Scheme evaluator that lets us show the process of applicative or normal order evaluation. We define functions using `def` instead of `define`. Then, we can evaluate expressions using `(applic (...))` for applicative order or `(normal (...))` for normal order. (Never mind how this modified evaluator itself works! Just take it on faith and concentrate on the results that it shows you.)

In the printed results, something like

```
(* 2 3) ==> 6
```

indicates the ultimate invocation of a primitive function. But

```
(f 5 9) ---->
(+ (g 5) 9)
```

indicates the substitution of actual arguments into the body of a function defined with `def`. (Of course, whether actual argument values or actual argument expressions are substituted depends on whether you used `applic` or `normal`, respectively.)

```
> (load "lectures/1.1/order.scm")
> (def (f a b) (+ (g a) b))     ; define a function
f
> (def (g x) (* 3 x))           ; another one
g
> (applic (f (+ 2 3) (- 15 6))) ; show applicative-order evaluation

(f (+ 2 3) (- 15 6))
   (+ 2 3) ==> 5
   (- 15 6) ==> 9
(f 5 9) ---->
(+ (g 5) 9)
   (g 5) ---->
   (* 3 5) ==> 15
(+ 15 9) ==> 24
24
> (normal (f (+ 2 3) (- 15 6))) ; show normal-order evaluation

(f (+ 2 3) (- 15 6)) ---->
(+ (g (+ 2 3)) (- 15 6))
   (g (+ 2 3)) ---->
   (* 3 (+ 2 3))
      (+ 2 3) ==> 5
   (* 3 5) ==> 15
   (- 15 6) ==> 9
(+ 15 9) ==> 24                 ; Same result, different process.
24
```

(continued on next page)

```
> (def (zero x) (- x x))        ; This function should always return 0.
zero
> (applic (zero (random 10)))

(zero (random 10))
   (random 10) ==> 5
(zero 5) ---->
(- 5 5) ==> 0
0                               ; Applicative order does return 0.

> (normal (zero (random 10)))

(zero (random 10)) ---->
(- (random 10) (random 10))
   (random 10) ==> 4
   (random 10) ==> 8
(- 4 8) ==> -4
-4                              ; Normal order doesn't.
```

The rule is that if you're doing functional programming, you get the same answer regardless of order of evaluation. Why doesn't this hold for `(zero (random 10))`? Because it's not a function! Why not?

Efficiency: Try computing

```
(square (square (+ 2 3)))
```

in normal and applicative order. Applicative order is more efficient because it only adds 2 to 3 once, not four times. (But later in the semester we'll see that sometimes normal order is more efficient.)

Note that the reading for the second half of the week is section 1.3, skipping 1.2 for the time being.