CS 61A Lecture Notes Week 1

Topic: Functional programming

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

Welcome to CS 61A, the world's best computer science course, because we use the world's best CS book as the textbook. The only thing wrong with this course is that all the rest of the CS courses for the rest of your life will seem a little disappointing (and repetitive).

Course overview comes next lecture; now we're going to jump right in so you can get started exploring on your own in the lab.

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 next 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.

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).

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 (2 months)
- Object-oriented programming (1 month)
- Client-server programming (1 week)
- 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 61C we look at lower levels; all are important but we want to start at the highest level to get you thinking right.

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?)

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 next week is section 1.3, skipping 1.2 for the time being.

• **Unix Shell Programming**

[This topic may be in week 1, week 4, or week 11 depending on the holidays each semester.]

Sometimes the best way to solve a programming problem is not to write a program at all, but instead to glue together existing programs that solve the problem.

As an example, we'll construct a spelling-checker program. It will take a text file as input, and will generate a list of words that are in the file but not in the online dictionary.

For this demonstration I'll use the file named `summary` as the text I want to spell-check.

We are given a file that contains several words per line, including things we don't want to compare against the dictionary, such as spaces and punctuation. Our job will be easier if we transform the input into a file with exactly one word per line, with no spaces or punctuation (except that we'll keep apostrophes, which are part of words — contractions such as "we'll" — rather than word delimiters).

```
tr -d '.,;:"\!\[]()' < summary > nopunct
```

`Tr` is a Unix utility program that translates one character into another. In this case, because of the `-d` switch, we are asking it to delete from the input any periods, commas, semicolons, colons, exclamation points, braces, and parentheses.

The single-quote (`'`) characters are used to tell the shell that the characters between them should not be interpreted according to the shell's usual syntax. For example, a semicolon in a shell command line separates two distinct commands; the shell does one and then does the other. Here we want the semicolon to be passed to the `tr` program just as if it were a letter of the alphabet.

When a program asks to read or print text, without specifying a particular source or destination, its input comes from something called the *standard input*; its output goes to the *standard output.* Ordinarily, the standard input is the keyboard and the standard output is the screen. But you can *redirect* them using the characters `<` and `>` in shell commands. In this case, we are asking the shell to connect `tr`'s standard input to the file named `summary`, and to connect its standard output to a new file named `nopunct`.

Spacing characters in text files include the space character and the tab character. To simplify things, let's translate tabs to spaces.

```
tr '    ' ' ' < nopunct > notab
```

Between the first pair of single-quotes is a tab character. `Tr` will translate every tab in the file to a space.

We really want one word per line, so let's translate spaces to newline characters.

```
tr ' ' '\n' < notab > oneword
```

The notation `\n` is a standard Unix notation for the newline character.

In English text, we capitalize the first word of each sentence. The words in the dictionary aren't capitalized (unless they're proper nouns). So let's translate capital letters to lower case:

```
tr '[A-Z]' '[a-z]' < oneword > lowcase
```

The notation `[A-Z]` means all the characters between the given extremes; it's equivalent to

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What `tr` does is convert every instance of the nth character of its first argument into the nth character of the second argument. So we are asking it to convert `A` to `a`, `B` to `b`, and so on.

Our plan is to compare each word of the text against the words in the dictionary. But we don't have to read every word of the dictionary for each word of the file; since the dictionary is sorted alphabetically, if we sort the words in our text file, we can just make one pass through both files in parallel.

```
sort < lowcase > sorted
```

The `sort` program can take arguments to do sorting in many different ways; for example, you can ask it to sort the lines of a file based on the third word of each line. But in this case, we want the simplest possible sort: The "sort key" is the entire line, and we're sorting in character-code order (which is the same as alphabetical order since we eliminated capital letters).

Common words like "the" will occur many times in our text. There's no need to spell-check the same word repeatedly. Since we've sorted the file, all instances of the same word are next to each other in the file, so we can ask Unix to eliminate consecutive equal lines:

```
uniq < sorted > nodup
```

`Uniq` has that name because in its output file every line is unique; it eliminates (consecutive) duplicates.

Now we're ready to compare our file against the dictionary. This may seem like a special-purpose operation for which we'll finally have to write our own program, but in fact what we want to do is a common database operation, combining entries from two different databases that have a certain element in common. Our application is a trivial one in which each "database" entry has only one element, and we are just checking for matches.

This combining of databases is called a *join* in the technical terminology of database programming. In our case, what we want is not the join itself, but rather a report of those elements of one database (our text file) that don't occur in the other database (the online dictionary). This is what the `-v1` switch means:

```
join -v1 nodup words > errors
```

That's it! We now have a list of misspelled words that we can correct in Emacs. The user interface of this spelling program isn't fancy, but it gets the job done, and we didn't have to work hard to get it.

It's a little ugly that we've created all these intermediate files. But we don't have to, because the shell includes a "pipe" feature that lets you connect the standard output of one program to the standard input of another. So we can do it this way:

```
tr -d '.,;:"\!\[]()' | tr '    ' ' ' | tr ' ' '\n' \
  | tr '[A-Z]' '[a-z]' | sort | uniq | join -v1 - words
```

The backslash at the end of the first line is the shell's *continuation* character, telling it that the same command continues on the next line. The minus sign (`-`) as the second argument to `join` tells it to take its first database input from its standard input rather than from a named file. (This is a standard Unix convention for programs that read more than one input file.)

We can write a file named `spell` containing this command line, mark the file as executable (a program, rather than data) with the command

```
chmod +x spell
```

and then instead of the sequence of commands I used earlier we can just say

```
spell < summary > errors
```

What we've done is use existing programs, glued together in a "scripting language" (the Unix shell), to solve a new problem with very little effort. This is called "code re-use." The huge collection of Unix utility programs have been written with this technique in mind; that's why almost all of them are what the Unix designers call "filters," meaning programs that take a text stream as input and produce a modified text stream as output. Each filter doesn't care where its input and output are; they can be files, or they can be pipes connected to another filter program.