COMP 210, Spring 2002
Lecture 2: From Expressions to Simple Scheme Programs

Things to do
- Lab sessions start today
- Homework 1 will be available this afternoon on the web site.
- Start reading the book (sections 1-6)

What is a computation?

When we speak of computation, we usually mean the automatic evaluation of some algebraic expression.

For example, if a pizza costs $12 and has 8 slices, what is your share of the bill for the pizza?

We need to know how many slices you ate

- 1 slice implies you owe (12 / 8) * 1 = $ 12/8 or $1.50
- 2 slices implies you owe (12 / 8) * 2 = $ 24/8 or $3
- 12 slices implies you owe (12 / 8) * 13 = $144/8 or $18

To pay for your pizaa, you might be considering a work-study job that pays $5.65 per hour. What would be your gross pay?

We need to know the number of hours worked

- 2 hours implies 5.65 * 2 = 11.30
- 10 hours implies 5.65 *10 = 56.50
- 12 hours implies 5.65 * 12 = 67.80

In a completely different vein, what if someone asks you for the surface area of the pizza?   We need to know its radius

- radius of 1 cm implies   pi * (1 * 1)   = pi * 1   = ~ 3.14…
- radius of 10 cm implies pi * (10 * 10)   = pi * 100 = ~ 314…
- radius of 11 cm implies pi * (11 * 11)   = pi * 121 = ~ 380 (380.1327….)

How much of the pizza is covered with toppings? The crust area takes one inch, so a 14 inch pizza has a 12 inch disk of toppings.

- Given diameter d, the toppings cover pi * [(d/2) –1] ^ 2 square inches, or

- pi/4 * d^2 – pi* d + pi square inches (multiply out (d/2-1)(d/2-1))

Which of these two formulations is better? The first, because, it reflects how we think about the problem. You can read it and figure out what it means, while the second one makes you scratch your head and question its correctness.

COMP 210 Axiom: Your code should reflect the way that you think about a problem.

Programming these calculations:

To make a computer perform these calculations, we need to write down a rule that it can follow. For our examples

If you at S slices of pizza, you owe (12 / 8) * S.

    Owe(S) = 12 / 8 * S

If you worked H hours, your gross pay is 5.65 * H

    Wage(H) = 5.65 * H

If the radius of a disk is R, its area is pi * (R * R)   [or R^2]

    Area(R) = R * R * π

If the pizza has diameter d, its toppings cover pi * (d/2-1) * (d/2-1)

    ToppingArea = Area(d/2-1)

[or ToppingArea(d) = (d/2-1) * (d/2-1) * pi]

These are all functions – they provide a unique map from argument into answer.

The languages that we use to express these rules to a computer have stricter notions of syntax than that—mostly to allow the computer to parse the rules automatically. These languages are called programming languages. In COMP 210, we will use a programming language with a particularly simple syntax, Scheme.

Example computations in Scheme:

Rule	Scheme Form
5	5
1.20 * 2	(* 1.20 2) [Prefix notation]
6 * pi * (3 * 3)	(* 6 (* pi (* 3 3)))
5.65 * 10	(* 5.65 10)
(12/8) * 3	(* (/ 12 8) 3)
pi * (10 * 10)	(* pi (* 10 10))

Every non-trivial expression starts with a "(" and a symbol that explains what comes next. Thus, * means a multiplication, so it should be followed by a list of argument expressions. Trivial expressions, like numbers and names, don't need parentheses.

These computations, expressed in a programming language such as Scheme, are programs. That is, they are concise, formal specifications that allow the computer and its various layers of software to evaluate your original computation. ARGUMENTS like H, S, and R create parameterized expressions or parameterized programs, so that we can reuse the basic computation with different sets of values.

Computing is the process of evaluating expressions. Computational science generalizes beyond high-school algebra to include algebras that operate over more complex domains, including numbers, names, symbols, and myriad other abstract spaces—(*i.e.*, the space of LR(1) grammars, or the space regular languages, or URLs on the WWW.)

[Note: the * operator in Scheme can actually take an arbitrary number of arguments, as can other operators. We will sometimes talk as if it is a binary (two-argument) operator so that it relates more intuitively to your experience with simple algebra.]

Formalizing Our Functions in Scheme
Scheme has a simple construct for recording these rules, called a *function.* Our four example rules can be written as Scheme functions:

    (define (Owe S) (* S (/ 12 8)))
    (define (Wage H) (* H 5.65))
    (define (Area R) (* pi (* R R)))
    (define (ToppingAread) (Area (- (d/2) 1)))

The process of writing down rules that specify a computation (in machine readable form) is called *programming*. The artificial languages that we use to write such specifications are called *programming languages*. (Brief commercial for 311, 412—after all, it is our strength.)

Given these definitions, the computer can automatically evaluate these functions on different ARGUMENTS. We use the function by calling it with an argument, as in

    (Owe 1)   produces 3/2 (or $1.50)
    (Owe 7)   produces 21/2 (or $7.50)
    (Wage 11)   produces 66

(Area 11)   produces 380.132711084365
(ToppingArea 12) produces 78.5398…

We call "S" a parameter and "7" an argument.

Discussed rewriting engine and worked several examples.

    (Owe 7)
    = (* 7 (/ 12 8))
    = (* 7 3/2)
    = 21/2   (or $10.50)

Another Program

Let's write another program. You and your COMP 210 lab partner do all your work on Saturday and Sunday nights. This requires a little more complex accounting, so you would like a program that computes your share of the weekend-long pizza bill.

Let's write a program, called Weekend-Pizza that takes the number of slices eaten on Saturday and on Sunday, and returns the total cost for a weekend of pizza eating. This takes us up to four programs in this lecture, so we need to start leaving behind records of what each program does.

In Scheme, a line that begins with a semi-colon is treated as a COMMENT. Scheme ignores COMMENTs; they exist only for human readers. In COMP 210, the first step in our programming methodology involves documenting the purpose of each program that we write. We do this in the form of a CONTRACT and a PURPOSE.

; Weekend-Pizza: num num -> num
; Purpose: calculate the total cost of pizza consumed on Sat. and Sunday
(define (Weekend-Pizza sat sun) …)

We add the program's header, to make the mapping between purpose and ARGUMENTs clear.   Now, we can fill in the body of the program as

; Weekend-Pizza: num num -> num
; Purpose: calculate the total cost of pizza consumed on Saturday and Sunday
 (define (Weekend-Pizza sat sun)
    (+ (Owe sat)
       (Owe sun)))

Does this work? What other ways might you write it?

    (Weekend-Pizza 3 4)
    = (+ (Owe 3)

```
                     (Owe 4) )
        = (+  (* 3 (/  12  8))
               (* 4 (/  12  8)) )
        = (+  (* 3   3/2)
               (* 4   3/2) )
        = (+  9/2
               12/2)
        = 21/2   or   $10.50
```

Alternatively, we could write Weekend-Pizza as
;Weekend-Pizza: num num -> num
; Purpose: calculate the total cost of pizza consumed on Sat. and Sunday
(define (Weekend-Pizza sat sun)
    (Owe (+ sat sun)))

Is this better or worse than the first version? Neither.

Another way that we could write it is:

;Weekend-Pizza: num num -> num
; Purpose: calculate the total cost of pizza consumed on Saturday and Sunday
(define (Weekend-Pizza sat sun)
    (+ (* sat (/ 12 8))
       (* sun (/ 12 8)))

How does this one compare to the previous two?   It is worse in several concrete ways.

a) If pizza prices rise to $13, we need to change it in 2 places rather than 1.

b) Because it duplicates the computation, it is much harder to read than the versions that use Owe directly. This makes it harder for you to come back and understand what you did, and for others to step in and use your code. (Assuming that you chose the program names in some reasonably self-documenting way.)

This brings us to our second axiom of program construction in COMP 210:

Always use an existing function when one is available

We often call this notion "keeping a single point of control." This is a general principle of management that extends beyond programs.

- The registrar creates several different lists that describe where classes are held. One is printed; another is online for general public access; still

another is in their own computer system where they can modify it. If all of these are generated from their own sources, the risk of an error rises with the number of copies. So does the amount of work to change a single fact!

- The use of symbolic names for jobs–-as in postmaster@rice.edu. I was one of the early postmasters of rice.edu; when I moved on, the mapping of the symbolic name to a specific person was changed.

Whenever you find yourself writing the same code to perform the same function multiple times, you should stop and think about how to restructure your program to create common code that you can reuse. We call this notion–-pulling a function out and making it independent–-abstraction.