COMP 210, FALL 2000
Lecture 2: Building and Testing Simple Programs

Things to do
- Register for a laboratory session (on the web site, the source of all knowledge)
- Start reading the book (sections 1-6)

Preliminary Discussion
- Programming in the small versus in the moderate, versus in the large
- COMP 210 teaches a disciplined, data-driven design methodology for programming in the small. Because the course is about design methodology, rather than *just* programming, some of the answers that are possible, even workable, are regarded as undesirable—just because they are not what the methodology will generate.
- You'll have to trust our judgement on this issue, but the designers of this course, and the staff of this course have a large collective base of experience.

Review
1. Last class we talked about computation and programming. Computation is the automatic evaluation of expressions. Programming is the process of writing down concise specifications for computations, in a form where the computer can automate their evaluation.
2. We looked at three expressions,
    3. Owe(S) to compute the cost of S slices of pizza
        Owe(S) = S * (12 / 8)
    4. Wage(H) to compute gross pay at work-study rates
        Wage(H) = H * 6
    5. Area(R) to compute the area occupied by a pizza from its radius
        Area(R) = pi * (R * R)

Mathematically, all of these are functions.

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

```
(define Owe(S) (*  S  (/  12  8) ) )

(define Wage(H) (*  H  6) )

(define  Area( R) (* pi (*  R  R) ) )
```

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

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 Saturday 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 Saturday 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 two places rather than one.

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 the first principle of program construction in COMP 210:

Always use an existing function when one is available

Back to Weekend-Pizza. How do we know if it is correct? We need to TEST it and convince ourselves that it is performing the computation that we designed it to do. We need a set of ARGUMENTs and expected results; we can invoke Weekend-Pizza on each set of ARGUMENTs and see if it returns the expected results.

Test Case	Expected Result	Actual Result
(Weekend-Pizza 2 3)	15/2	15/2
(Weekend-Pizza 6 4)	15	15
(Weekend-Pizza 0 0)	0	0

How do we choose these values? We'll spend more time on that issue over the next week. How do we determine the expected results? From the program's contract and our knowledge of the problem—the underlying facts that we used to specify the problem before writing a single line of Scheme. How do we get the actual results? From running the actual code on Dr. Scheme.

If these test values are well-chosen, we can state, with some confidence, that our program, Weekend-Pizza, calculates the correct answer.

Methodology

We can formalize these activities into the first four steps of our methodology.

a) Write a contract, purpose, and a header
b) Develop several examples, so that we know what results to expect and so that we have test inputs for later use
c) Write the program's body
d) Test the program

Summary

1. Computation is (roughly) the automatic evaluation of expressions
2. Programming is the discipline of writing down rules that specify computations
3. Programs can be evaluated automatically. Scheme programs can be evaluated using a simple rewriting engine—replace defined terms with their bodies until all the operations are elementary (or built-in to Dr. Scheme). Evaluate the elementary operations.
4. Complicated programs can be built on the shoulders of smaller programs.

   1St Principle of COMP 210: **Always reuse an existing function if one exists.**
5. Introduced design methodology
   a) Write a contract, purpose, and header
   b) Develop several examples—chosen to elucidate the program's behavior
   c) Write the program's body
   d) Test the program to ensure that it behaves as specified (and as desired)

Next class—*Conditionals, the innovation that makes interesting programs possible*