COMP 210, Spring 2001
Lecture 1B: Building and Testing Simple 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 pizza, 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*in* implies π* (1 * 1) = pi * 1 = ~ 3.14159265…
- radius of 10*in* implies π * (10 * 10) = pi * 100 = ~ 314.159265…
- radius of 11*in* implies π * (11 * 11) = π * 121 = ~ 380 (380.1327….)

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

If you ate *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 $\pi * r$^2]

$area(r) = \pi * r * r$

These are all functions: they map an input argument to a unique 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 * π * (3 * 3)	**`(* 6 (* pi (* 3 3)))`**	
5.65 * 10	**`(* 5.65 10)`**	
(12 / 8) * 3	**`(* (/ 12 8) 3)`**	
π * (10 * 10)	**`(* pi (* 10 10))`**	

Every non-trivial expression starts with a "(" and a symbol that explains what comes next. Thus, * means a two-operand multiplication, so it should be followed by two 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 Scheme expressions, 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 many other operators. We will treat it as a binary (two-argument) operator so that it corresponds more closely to your experience with simple algebra.]

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, 312, 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 Sat. and Sun.
(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 Sat. and Sun.
(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)) (owe 4))
= (+ (* 3 3/2) (owe 4))
= (+ 9/2 (owe 4))
= (+ 9/2 (* 4 (/ 12 8)))
= (+ 9/2 (* 4 3/2))
= (+ 9/2 6)
= 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 Sun.
(define (weekend-pizza sat sun)
    (owe (+ sat sun)))
```

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

Did this version first in class.

Another way that we could write it is:

```
; weekend-Pizza:  Num Num -> Num
; Purpose: calculate the total cost of pizza consumed on Sat. and Sun.
(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 the first principle of program construction in COMP 210:

Always use an existing function when one is available