# COMP 200: Elements of Computer Science
**Fall 2004**
**Lecture 4: August 29, 2004**

## On the Board

Homework 1: due today (start of class)
http://www.drscheme.org — download DrScheme for your machine

## Elements of Programming

Algebraic expressions constitute a large part of most computer programs — in almost any programming language. Thus, it seems fitting to begin learning to program by learning to write and to evaluate algebraic expressions. (As a side benefit, this approach ties your intuitions about programming computers to your innate knowledge of algebra.)

Back to splitting a pizza: if a pizza costs $12 and divides into 8 slices, your share of the pie would be:

- For 1 slice ⇒ 1 x $12/8 = $12/8 = $1.50
- For 2 slices ⇒ 2 x $12/8 = $24/8 = $3
- For 12 slices ⇒ 12 x $12/8 = $144/8 = $18

To pay for your pizza addiction, you might take a job working in someone's research lab for, say, $7.35 per hour.

- 2 hours per week ⇒ 2 x $7.35 = $14.70 / week or 9.8 slices/week
- 5 hours per week ⇒ 5 x $7.35 = $36.75 / week or 24.5 slices/week
- 12 hours per week ⇒ 12 x 7.35 = $88.20 /week or 58.8 slices/week

Because you have geek friends, someone will want to compute the surface area of the pizza (and, subsequently, a single slice). From your high-school education, you know that the area of a circle is pi x radius$^2$.

- Radius 1 inch pizza ⇒ area of pi * 1 * 1 = pi = 3.14159 square inches
- Radius 10 inch pizza ⇒ area of pi * 10 * 10 = 314.159 square inches
- Radius 11 inch pizza ⇒ area of pi * 11 * 11 = ~380 square inches

And a single slice of an 11 inch pizza has roughly 380/8 square inches, or 47.5 square inches of pizza.

Geeking out a little more, what area of the pizza is covered with toppings? If the crust is one inch wide, then the area covered by toppings will be equal to the area of a disk with radius one inch smaller than the pizza's overall radius.

- A 14 inch (diameter) pizza has radius 7 and topping radius 6, so it has 6 * 6 * pi = 36 * 3.14159 = ~113 square inches of toppings.
- We computed that as

  pi x (d/2-1) x (d/2-1)

  We could also multiply that out to get

  pi/4 x (d-1) x (d-1) – pi x d + pi square inches

  Which formulation is "better"?

The first formulation is preferable because it reflects the way that we (I?) think about the problem. (d/2) is the radius. (d/2)-1 is the radius of the area covered by topping. Pi x r x r is the formula for area of a disc.

The latter formulation is completely equivalent from an arithmetic perspective. However, if you picked up something that used the formulation, it might be hard to intuit the steps that got us to the formula. In designing and building programs, we must be concerned about our ability to go back later and understand what we have done — or what others have done.

**Computing in Scheme**

Scheme is a programming language that has a particularly simple syntax and a clean, elegant semantics — the mapping between syntax and meaning.

| Algebraic Expression | Scheme Syntax | Comments |
|---|---|---|
| 5 | 5 | Numerals are values |
| 15 + 6 | (+ 15 6) | Prefix notation |
| 1 + 2 + 3 + 4 | (+ 1 2 3 4) | Arbitrary arity for ops |
| 12 ÷ 8 | (/ 12 8) | ÷ is written /; x is * |
| 7.35 * 12 | (* 7.35 12) | Decimals |
| pi * 11 * 11 | (* pi 11 11) | It knows about pi |
| pi * (d/2 –1) * (d/2 –1) | (* pi (- (/ d 2) 1) (- (/ d 2) 1) ) | Prefix notation must encode precedence |

*In Scheme*, every non-trivial expression starts with a "(" and a symbol that explains what comes next. The symbol is either an operator, such as +, -, *, and /, or the name of a scheme "function". Thus, (* 1 2 3 4) multiplies together its four arguments. The arguments can, themselves, be expressions, allowing us to evaluate complex expressions such as the topping area of the pizza.

*Evaluate the expressions in Dr. Scheme.*

Trivial expressions, like numbers and names, don't need parentheses.

**Formalizing Pizza Mathematics**

If we had to type in every expression that we wanted the computer to evaluate, every time that we needed to evaluate it, the process would be both tedious and time-wasting. We need a mechanism that lets us save important expressions and use them repeatedly. (Such a saved computation forms a simple *program*.)

To compute a single person's share of the cost of a pizza, we used the algebraic formula *S* * $12/8, where *S* is the number of slices that the person consumed (or ordered). We can formalize that into a function

Owe(S) = S * $12 / 8

Notice the dollar sign; it tells us (via dimensional analysis — the canceling and simplification of units of measure) that the result of this function is an amount in US currency.

This “function” reads as “Owe of S is S * 12 / 8 dollars”. We call “S” a parameter; we might say that the function “Owe” is parameterized by “S”.

To create a Scheme program or function that models Owe, we would use the **define** operator. In Dr. Scheme, we have a separate window for definitions.

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

Typing this into the definitions window, then clicking “run” creates a Scheme workspace where “Owe” is available in much the same way that the basic operators are. (Owe is monadic — it takes one argument.)

We can use similar definitions to figure out our Wage for hours worked

Wage(H) = $7.35 * H

And the area of a pizza from its Radius

Area(R) = pi * R * R

And from its Diameter

AreaFromDiameter(D) = pi * (d/2) * (d/2)

And the area covered with toppings

Toppings(D) = pi * (d/2 –1) * (d/2 –1), or

Toppings(D) = Area(d/2-1)

The second form of Toppings is preferable, given the existence of Area. It reuses the formula from Area and creates a “single point of control” where we can change the formula for the area of a disc. (This idea makes sense even if the example does not. We are unlikely to change the value of pi or its relationship to a circle.)

All of these can be defined in Scheme following the same basic syntactic form that we used for Owe.