COMP 200: Elements of Computer Science
Fall 2004
Lecture 5: September 1, 2004

On the Board

Homework 2: Assigned Friday
Get access to Dr. Scheme (http://www.drscheme.org to download)

Back to Programming

Last class, we wrote some Scheme expressions, and introduced the notion of a parameterized function.

```
(/ 12 8)

(* 1 2 3 4)

(* pi 11 11)

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

Today, we will go back and put a veneer of organization and approach over that haphazard lecture. Programming in COMP 200 (and 210) involves using a fairly rigid design methodology (laid out in Felleisen et al's book, *How to Design Programs.*)

Any nontrivial expression in Scheme begins with an open parenthesis followed by the name of a function. Builtin functions include the standard arithmetic operators: sum, difference, product, division. When we created our own function, **Owe**, we were able to use it in the same way that we used the builtin functions (or operators).

Pi is an example of a named object in scheme. The code that defined **Owe** creates another named object, the function **Owe**. We can create named objects whose value is a number (similar to **pi**).

```
(define r 11)
(* pi r r)
```

should produce the same result as

```
(* pi 11 11)
```

The temptation to create named objects is big, so we need to adopt some discipline. First, any named object must be documented with a comment — in scheme, we can insert a textual line by beginning it with a semicolon:

```
; R is the radius for our disc
(define R 11)
```

Scheme ignores the comment. Comments exist for the sole purpose of enlightening a human reader.

Anytime that we write a define (in Dr. Scheme's definition window), it must be accompanied by a meaningful comment. Something such as

```
; This line is the required comment
(define R 11)
```

is not acceptable.

The first step in designing a program involves documenting its purpose. In our design methodology, the first step of the task involves writing down some comments that describe what function the program will perform. We call these two comments a **contract** and a **purpose**.

As an example, consider a simple program that calculates the first class postage for a letter as a function of its weight, in ounces. To begin, we must understand the formula that the Post Office uses to calculate postage. For first class mail on a standard-sized envelope, the first 1 ounce costs 37 cents and each additional ounce costs 23 cents. (Square envelopes, oversize envelopes, and so on cost more.)

To encode this knowledge in a scheme function, FirstClassPostage, we start with a contract and purpose:

```
; FirstClassMail : num → num
; Purpose: takes a weight in ounces and returns the postage
;              in cents required for US domestic mail
```

The contract is written in a cryptic form. We read it as

> "FirstClassMail takes as input a number (singular) and returns a number."

The words appearing between the colon and the arrow describe the arguments of the function, in order. The words to the right of the arrow describe its result.

We wrote down "num" for the argument and the result. In fact, we can define matters more carefully. We know from the semantics of the problem

that a letter cannot have a negative weight. We might use a notation such as positive num or non-negative num.

To the contract and purpose, we add a header:

```
; FirstClassMail : posnum → num
; Purpose: takes a weight in ounces and returns the postage
;              in cents required for US domestic mail
(define  (FirstClassMail Ozs) …)
```

The header simply lays out the syntax (function name and name of its argument — Ozs in this case) and leaves the function body as a set of ellipses.

The next step is to fill in the program's body with an expression that correctly fulfills its purpose. To figure out that expression, it is helpful to work several examples:

Ozs	Result
1 ounce	37 cents
2 ounces	60 cents
11 ounces	267 cents

We hope that working out the examples leads us to a method of computing the result. Equally important, the examples provide us with data for testing our program.

In this case, one formula that appears to fit the data is

37 + (Ozs –1) * 23

In Scheme, this formula becomes

```
(+ 37 (* (- Ozs 1) 23))
```

Now, we can fill in the program and, we hope, complete it.

Of course

```
(- (* Ozs 37) (* (- Ozs 1) 14))
```

also works, but is less straightforward

```
; FirstClassMail : posnum → num
; Purpose: takes a weight in ounces and returns the postage
;              in cents required for US domestic mail
(define  (FirstClassMail Ozs)
        (+ 37 (* (- Ozs 1) 23)) )
```

At this point, we need to test our program to determine if it functions correctly. We can certainly type it into Dr. Scheme and check the test cases given above against the results.

At this point, we need to explain the rewriting semantics of Scheme.

(FirstClassMail 1)

⇒ (+ 37 (* (- 1 1) 23))

⇒ (+ 37 (* 0 23))

⇒ (+ 37 0)

⇒ 37

Dr Scheme copies the body of the function, replacing the arguments with their actual values. It then evaluates expressions innermost to outermost (and first to last) until it can simplify the expression so that it contains no operators or functions. When the expression can be simplified no further, Dr Scheme is done and it reports the result.

work the other examples

The rewrite rules for invoking a function and for evaluating arithmetic are simple and intuitive. Careful application of the rewrite rules leads to a hand evaluation of the expression that should produce the same result as evaluating it using Dr. Scheme.

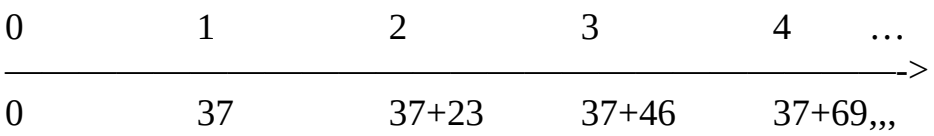
What about an argument that is less than one ounce?

Fractional arguments break the formula, so we can introduce a helper function

```
; FCM:  posnum → posint
; Purpose: rounds its argument upward to the next integer and
;              invokes FirstClassMail
(define (FCM  X)
        (FirstClassMail (ceiling X)))
```

What about an argument that is less than or equal to zero?

Both cause problems, albeit of a somewhat different nature. It turns out that the behavior of FirstClassMail depends on the value of its input argument. We can represent this with a number line:



Our algebraic formula only works in the last part of the interval (>= 2).

For the lower parts of the interval (0 and 1), simpler formulas pertain. Thus, we need a way to encode in the program three kinds of behavior over three different ranges.

We can write expressions that pick out these distinct ranges

Condition	**Formula**
(= Ozs 0)	0
(= Ozs 1)	37
(>= Ozs 2)	(* 37 (* (- Ozs 1) 23))

Full set of relational operataors is <, <=, =, >, >=. Can use and, or, & not to create combinations (as in (not (= 1 2)).

To express this in Scheme, we use a **cond** expression

```
; FirstClassMail: num -> num
; Purpose: takes a weight in ounces and returns the postage
;         in cents required for US Domestic Mail
(define (FirstClassMail W)
  (cond
     ((= W 0) 0)
     ((= W 1) (<= W 1)) 37)
     ((>= Ozs 2)  (+ 37 (* (- W 1) 23)))
   ))
```