COMP 210, FALL 2001
Lecture 15: Functional Abstraction

Reminders:
1. Homework due on Wednesday.
2. Graded exams will be returned in class on Wednesday.

Review

1. Last class before exam review: rewriting rules for **`local`**, other uses for **`local`**.

On to Functional Abstraction

Write a simple function that consumes a list of numbers and produces a list of numbers. The numbers in the returned list should be exactly those numbers in the original list that are less than 5 (in the same order as the original list).

```
;; keep-lt-5:  list of numbers -> list of numbers
;; Purpose:  keeps all input numbers in alon less than 5
(define (keep-lt-5 alon)
  (cond
    [(empty? alon)   empty]
    [(cons?   alon)
     (cond [(< (first alon) 5)
            (cons (first alon) (keep-lt-5 (rest alon)))]
           [else  (keep-lt-5 (rest alon))])]))
```

What about **`keep-lt-9`** ?

```
;; keep-lt-9:  list of numbers -> list of numbers
;; Purpose:  keeps all input numbers in alon less than 9
(define (keep-lt-9 a-lon)
  (cond
    [(empty? alon)  empty]
    [(cons? alon)
     (cond [(< (first alon) 9)
            (cons (first alon) (keep-lt-9 (rest alon)))]
           [else  (keep-lt-9 (rest alon))])]))
```

Notice how much these two functions have in common. Can we write one function that captures all this common code (*single-point of control*) and use it to implement **`keep-lt-5`** and **`keep-lt-9`**?

```
;; keep-lt:  number list-of-numbers -> list-of-numbers
;; Purpose:  keep all input numbers in alon that are less
;;           than the given number num
(define (keep-lt num alon)
  (cond
    [(empty? alon)  empty)
```

```
    [(cons? alon)
     (cond [(< (first alon) num)
            (cons (first alon)  (keep-lt num (rest alon)))]
           [else  (keep-lt num (rest alon))])]))
```

Notice that **num** never changes. We could use a local to avoid passing it around in so many places (and save work) [But, efficiency is only a concern in early part of Comp 210 when it becomes an exponential problem.]

```
;; keep-lt:  number list-of-numbers -> list-of-numbers
;; Purpose:  keep all input numbers in alon that are less
;;           than the number num
(define (keep-lt num alon)
  (local
      [(define (filter-lt alon)
         (cond
           [(empty? alon)  empty)
           [(cons? alon)
            (cond
              [(< (first alon) num)
               (cons (first alon) (filter-lt (rest alon)))]
              [else  (filter-lt (rest alon))])]))]
    (filter-lt alon)))
```

Using **keep-lt**, we can define **keep-lt-5** and **keep-lt-9**

```
(define (keep-lt-5 alon)  (keep-lt 5 alon))
(define (keep-lt-9 alon)  (keep-lt 9 alon))
```

What if we wanted to write **keep-gt-5**

```
;; keep-gt-5:  list of numbers -> list of numbers
;; Purpose:  keeps all input numbers in alon greater than 5
(define (keep-gt-5 alon)
  (cond
    [(empty? alon)  empty]
    [(cons? alon)
     (cond [(> (first alon) 5)
            (cons (first alon) (keep-gt-5 (rest alon)))]
           [else  (keep-gt-5 (rest alon))])]))
```

Where do these functions differ? Only in the comparison operator and in the names of the functions. [The last lecture should have convinced you that the names are malleable.]

How can we reuse the common code here? Previously, we made the upper limit on the value into a parameter. Now, we need to make the comparison operation itself be a parameter. Can we pass in the comparison operator?

Critical Aside

How do we represent **>** in the contract? **(number number -> number)**
We've been writing these contracts for six weeks now. This should be pretty natural.

Back To Abstracting Out Comparison

```
;; keep-rel-5:  (num num -> num) list of numbers -> list of numbers
;; Purpose:  keeps all input numbers n in alon such that n rel 5
(define (keep-rel-5 rel alon)
  (cond
    [(empty? alon)  empty]
    [(cons? alon)
     (cond [(rel (first alon) 5)
            (cons (first alon) (keep-rel-5 rel (rest alon)))]
           [else  (keep-relation-5 (rest alon))])]))
```

and

```
(define (keep-lt-5 alon)  (keep-rel-5 < alon))

(define (keep-gt-5 alon)  (keep-rel-5 > alon))
```

As before, we can use local in the obvious way to avoid passing relation as a parameter.

```
;; keep-rel-5 :  (num num -> num) list of numbers -> list of numbers
;; Purpose:  keeps all input numbers that have relation than 5
(define (keep-rel-5 rel alon)
  (local
      [(define (filter-rel alon)
         (cond [(empty? alon)  empty]
               [(cons? alon)
                (cond [(rel (first alon) 5)
                (cons (first alon) (filter-rel (rest alon)))]
              [else  (filter-rel (rest alon))])]))]
    (filter-rel alon)))

(define (keep-lt-5 alon)  (keep-rel-5 < alon))
```

Of course, the next thing we want to do is abstract out the number 5. We should be able to write a function that takes both the relation and the limit as parameters and returns a list containing the specified subset of the numbers in the original list.

```
;; keep-rel:  (num num -> num) num  list-of-nums -> list-of-nums
;; Purpose: keep all the numbers n in the input list alon such that
;;          n rel num
;;     by the function argument to the number argument  (whew!)
(define (keep-rel  rel num alon)
  (local
      [(define filter-rel alon)  ;; treat relation & num as invariant
         (cond [(empty? alon)  empty]
               [(cons? alon)
                (cond [(rel (first alon) num)
                       (cons (first alon) (filter-rel (rest alon)))]
                      [else (filter-rel (rest alon))])]))]
    (filter-rel alon)))

(define (keep-gt-9 alon)  (keep-rel > 9 alon))
```

Programs as Values

How can we pass **>** or **<** or **=** as an argument (parameter) to a function such as **keep-rel** ?

A program (function) is just another value. Recall that we created test values using the **define** operation. We used it to create numbers and lists. These became named values in the world of Scheme values. We use the same syntax and operation to create programs, don't we? The **define** operator takes its two parts – the function name and argument list and its body. The way that DrScheme evaluates the function is a little more complex than evaluating

```
(define Seventeen 17)
```

because it involves rewriting the parameters with their values. Of course, Seventeen has no parameters, so the mechanism for handling functions will work on a defined numerical value as a trivial case.

In general, functions (programs) are values. We can use them anywhere that we use values. For example, we can build lists of functions, just as we build lists of symbols or numbers, or objects created with **define-struct**.

What about **>** ? Isn't it special? Isn't it a built-in operator?

> Yes. It is a built-in function, but that grants it no special exemptions from the rules that govern the execution of Scheme programs. The built-in functions are just functions themselves. Thus, **+**, *****, **/**, **-**, **<**, **=**, **>** are all functions akin to program-defined functions that you could write. So are **add1**, **sub1**, **cons**, **cons?**, **empty?**, **number?**. **But**, **define**, **define-struct**, **cond**, **and**, and **or** are not functions and cannot be used as values. The pseudo-functions **and**, and **or** are not conventional functions because all Scheme functions evaluate all of their arguments. The pseudo-functions **and**, and **or** are abbreviations for **cond** forms, which do not necessarily evaluate all of question or result expressions that appear in the embedded list of clauses.

What about returning a function? If we can treat functions as values, can we write a Scheme function that creates new functions and returns them?

> Yes. However, we have not (yet) seen an operator that builds a program. That's in the next lecture. With the right operator, **lambda**, you can write programs that create new programs and return them.