COMP 210, FALL 2000
Lecture 20: More on Functional Abstraction

Reminders:

1. Next exam is a take-home; handed out Friday, due one week later (10/27/00@5pm).
2. Go to the lab lecture this week. Many functions that will be needed in future labs and lectures and will only be covered in the lab lecture. You are responsible for the lab lecture on the test.
3. Review session Friday, 3PM, DH 1070.

Review

1. We started to talk about functional abstraction. Built a series of repetitive examples, intended to show you that they were similar. Showed how to abstract them out. Ended with an example, **keep-rel**: (num num -> num) num alon -> alon.

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

This shocked several of you. We were able to pass the relational operator as an argument to a function and it works.

We talked about the fact that programs are values. I pointed out that everything you've seen so far, with the exception of **define**, **define-struct**, and **local** , is a program. The rationale that I gave you is bogus. I suggested that you couldn't type **define** in the interactions window. I was wrong.

Back to Work

So far, all of these examples have looked at an open-ended interval (> 5, < 9). [Of course, we could use filter-rel to find all of the numbers equal to *x*, but that's only interesting if we wanted to count them. (length (keep-rel = 5 somelist)).] What if we wanted to pull out the numbers between five and nine?

```
;; keep-bet-5-9: list-of-numbers -> list-of-numbers
;; Purpose: returns a list containing those numbers in the input list
;;              whose value is between 5 and 9, inclusive
(define (keep-bet-5-9 alon)
  (cond
    [(empty? alon)   empty]
    [(cons?    alon)
      (cond
        [(and (>= (first alon) 5) (<= (first alon) 9))
         (cons (first alon) (keep-bet-5-9 (rest alon)))]
        [else  (keep-bet-5-9 (rest alon))]
      ) ] ) )
```

We should really write a helper function to replace the complex test.

```
;; bet-5-9?: number -> boolean
;; Purpose:  test if the argument is between five and nine, inclusive
(define (bet-5-9? anum)
    (and (>= num 5) (<= num 9)))

;; keep-bet-5-9: list-of-numbers -> list-of-numbers
;; Purpose: returns a list containing those numbers in the input list
;;                  whose value is between 5 and 9, inclusive
(define (keep-bet-5-9 alon)
  (cond
     [(empty? alon)   empty]
     [(cons?    alon)
        (cond
           [(bet-5-9? (first alon))
             (cons (first alon) (keep-bet-5-9 (rest alon)))]
           [else  (keep-bet-5-9 (rest alon))]
        ) ] ) )
```

You know, by now, where we are going. What if we want to change the range of numbers? We can change the helper function, but we might be using it somewhere else.
We can write a version that takes an arbitrary range of numbers…

```
;; bet? :  num num num -> boolean
;; Purpose: determines if the third argument lies numerically between
;;              the first and second arguments
(define (bet?  lower upper  anum)
   (and  (>= num lower) (<= num upper)))

;; keep-bet : num num list-of-numbers -> list-of-numbers
;; Purpose: keeps all the numbers lying between first and second
arguments
(define (keep-bet  lower  upper  alon)
  (local
       [(define (filter-bet alon)
          (cond
               [(empty? alon)  empty]
               [(cons?    alon)
                    (cond
                       [(bet? lower upper (first alon))
                         (cons (first alon) (filter-bet (rest alon)))]
                       [else  (filter-bet  (rest alon))])]))]
       (filter-bet alon) ))

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

Notice that we used a local to avoid passing around lower and upper at the recursive calls inside filter-bet and keep-bet.

Abstracting from keep-rel & keep-bet
Look at the definitions of keep-rel and keep-bet. They are reasonably similar. They differ in their parameters and the first case in the innermost cond–where they decide whether or not to keep the first element of the input list. The parameter differences boil down to inputs to that test. The cond clause differs in the implementation of that test.

Can we write one program to capture all of that common code? We start by copying down all of the information that is common to both programs, leaving an ellipsis in places where they differ…

```
(define (keep  …  alon)
  (local
     [(define (filter  alon)
        (cond
          [(empty? alon) empty)]
          [(cons?    alon)
             (cond
               [( … (first alon))
                 (cons (first alon) (filter (rest alon)))]
               [else (filter (rest alon))] )] ))]
     (filter alon) ))
```

Let's fill in the gaps. First, look at the gap inside **filter**. We need a helper function that takes (first alon) and returns a boolean that tells filter whether or not to keep the number. Let's make a number for that helper function and make it a parameter. That fills in the gap in the parameters, too.

```
(define (keep  keep-elt? alon)
  (local
     [(define (filter  alon)
        (cond
          [(empty? alon) empty)]
          [(cons?    alon)
             (cond
               [(keep-elt? (first alon))
                 (cons (first alon) (filter (rest alon)))]
               [else (filter (rest alon))] )] ))]
     (filter alon) ))
```

To use this, we just need to write the appropriate helper functions. For example

```
(define (keep-lt-5 alon)
   (local  [(define (lt-5? num) (< num 5))]
      (keep  lt-5?  alon)  ))
```

```
(define (keep-bet-5-9 alon)
   (local [(define (bet-5-9?  num) (bet?  5  9 num))]
         (keep  bet-5-9?  alon) ))
```

The function **keep** is so useful that Scheme provides a built-in version of it. We call the built-in version of it **filter.**

The Professor is a Liar

Well, I told you that everything you learned outside of the compilers class is a lie. In fact, Scheme does provide **filter**, but it is not as limited as the one we wrote.

What if you wanted to write a function that counted the number of times the symbol 'fee appeared in a list? You'd like to write it as

```
;; keep-fee : list-of-symbol -> list-of-symbol
;; Purpose: return the list containing every occurrence of 'fee
(define (keep-fee  alos)
   (local [(define (is-fee?  asym)(=  'fee  asym))]
          (keep   is-fee?  alos) ))
```

Except, this violates the contract for keep.

Look at the body of keep. Is there anything in the body of keep that depends on the fact that the argument list contains numbers?

The contract is overly restrictive. In fact, we'd like keep to work on a list of symbol, or a list of ftn, or a list of mechanic, or a list of plane, or a list of just-about-anything.

To do this, we could rewrite the contract by defining a list of several types of things. Imagine **list-of-just-about-everything** where it has a clause for each kind of object that we want **keep** to use. Does this solve our problem? NO. In particular, the list-of-just-about-everything allows one list to contain many different kinds of elements. That makes the **keep-elt** function much harder to write correctly. What we need is a notation that allows us to specify that keep takes a list of something—where every element of the list is a something—but that keep doesn't care what that something is. The precise requirement for keep is that **keep-elt** must have a contract of **something-> boolean**.

To write this down as a Scheme contract, we want to say

```
;; a list-of-alpha is either
;;   – empty, or
;;   – (cons f r) where f is an alpha and r is a list of alpha.
```

Then, we can write the contract for keep as

```
;; keep :  (alpha→boolean) list-of-alpha -> list-of-alpha
```

This concisely expresses the requirement for **keep**. The function **keep-elt** must process the elements of the argument list and return a boolean. The elements of the input and output list both have that same type. This is the real contract for **filter**.

Lambda

If we're going to use abstract functions, such as **filter**, we're going to end up creating a large number of helper functions. Many of these functions will have only one purpose–-they will be created to pass into an abstract function. In this case, there is little (or no) point in forcing you to invent clever (or unique) names for all of them. Scheme gives us two mechanisms to avoid naming problems with these helper functions.

We could, of course, encapsulate them inside a local, as we did with **keep-fee**.

```
;; keep-fee : list-of-symbol -> list-of-symbol
;; Purpose: return the list containing every occurrence of 'fee
(define (keep-fee  alos)
   (local [(define (is-fee?  asym)(symbol=?  'fee  asym))]
            (keep   is-fee?  alos) ))
```

This hides **is-fee?** from the world outside **keep-fee** and avoids the potential for a name conflict. However, there are two problems with writing **keep-fee** this way.

1. It forces you to invent a name for double. (minor hassle)

2. It violates the whole philosophical purpose of using local. The real justifications for using a local are:
    1. To avoid computing some complicated value more than once.
    2. To make complicated expressions more readable by introducing helper functions that break the expression up into more tractable parts. (Notice that avoiding the use of invariant parameters might fall under either case!)

This example fits neither criterion. The expression is not complicated; in fact, it is about as simple as a Scheme expression can get. The expression is not used in many places; it is used exactly once. The only reason for introducing double is because we need a function (symbol→symbol) that we can pass to **keep** (or **filter**) – this lets us avoid writing a lot of code by using the abstract function.

To handle this situation, Scheme includes a construct called λ. Unfortunately, DrScheme operates under the limited typographic conventions of computer keyboards, so we end up writing it out as **lambda**. Lambda lets us create unnamed programs –- it is a second way to write out a program (without using **define**).

```
(define (is-fee? asym)              (lambda (asym)
  (symbol=? asym 'fee))               (symbol=? asym 'fee))
```

These are equivalent, in the sense that they both create programs that "do" the same thing. They differ, in the sense that you can use **is-fee?** anywhere that its name can be seen, while the **lambda** expression occurs somewhere in the code, is created, is evaluated, and cannot be used elsewhere *because it has no name*.

Using **lambda**, we could rewrite **keep-fee** as

```
(define (keep-fee alos)
   (filter (lambda (asym) (symbol=? asym 'fee))  alos) )
```

Formally, lambda is written

```
(lambda
  (arg1  arg2 … argn)
  body
```

```
    )
```

where arg1, arg2, …, argn and body are arbitrary Scheme expressions.

To evaluate a lambda expression, DrScheme rewrites it as

```
(local  [(define (a-unique-new-name  arg1  arg2 … argn)
                   body)]
        a-unique-new-name)
```

The body-expression cannot refer to *a-unique-new-name* because the programmer does not know how to write it. The unique name is introduced by the rewriting process, not by the programmer, so the programmer cannot write a lambda expression that *directly* calls itself.   To slightly simplify the explanation, assume the list being sorted contains no duplicates.