COMP 210, Spring 2001
Lecture 16: More on Functional Abstraction

Reminders:

1. Next exam will be Wednesday evening, March 21 from 7-9 P.M.
2. Current homework is due Wednesday, March 14.
3. Current reading: Intermezzo 3, Sections 19-23, Intermezzo 4.

Review

1. We started to talk about functional abstraction. We built a series of similar programs and showed how they could be expressed as instantiations of a common abstraction:

   `keep-rel: (num num -> num) num alon -> alon.`

```
;; keep-rel  (num num -> bool) num  list-of-nums -> list-of-nums
;; Purpose: given binary relation rel, number n, and (list-of num)
;;          alon, keep all the numbers i in the input list such
;;          that i rel n
(define (keep-rel rel n alon)
  (local [(define filter-rel alon)
          ;; treat rel & n as invariant
            (cond [(cons? alon)
                   (cond
                     [(rel (first alon) n)
                      (cons (first alon) (filter-rel alon)))
                     [else (filter-rel (rest alon))])]))]

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

We talked about the fact that functions are values. All of the Scheme "operations" you've seen so far, with the exception of **`define, define-struct`**, **`local`**, **`and`**, and **`or`** is a function.

Back to Work

So far, all of these examples have looked at an open-ended interval, *e.g.*, (5,∞), (∞,9). [Of course, we could use **`filter-rel`** to find all of the numbers equal to some number, 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 5 and 9?

```
;; keep-bet-5-9: list-of-numbers -> list-of-numbers
;; Purpose: returns a list containing those numbers i in the input list
;;          alon such that 5 <= i <= 9.
(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 num) -> (list-of num)
;; 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 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 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` and `keep-bet`

Look at the definitions of **`keep-rel`** and **`keep-bet`**. They are very 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 an operation that takes **`(first alon)`** and returns a boolean that tells **`filter`** whether or not to keep a list element. Let's simply make that operation a parameter of **`keep`**.

```
(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 generalization of our various **`keep`** functions, 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.`** Note that **`filter`** does not care what type elements appear in the list. The only restriction on the elements of the input list is imposed by the function parameter! For any type **`T`**, **`filter`** maps a list of **`T`** and a function of type **`(T -> boolean)`** to a list of **`T.`**

`filter: <T> (list-of T) (T -> boolean) -> list-of-T`

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

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, a significant hassle in large programs because they contain so many names.
2. It is wordy.

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 functions –- it is a second way to define a function (instead of 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) )
```

In Scheme, a **`lambda`** expression is written

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

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

`lambda`-expressions are values so they evaluate to themselves—just like numbers and symbols.

To evaluate the application of a **`lambda`** expression, DrScheme evaluates the argument expressions and replaces the application by the body of the **`lambda`** expression with the argument values safely substituted (no capture) for the corresponding parameters.

There is an alternate way to interpret **`lambda`** expressions. The **`lambda`** expression

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

can be expanded into

```
(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.

These two reduction strategies for handling **`lambda`** are equivalent. The first is conceptually simpler but the second may be more notationally convenient when evaluating programs by hand.