Programming Abstractions

Lecture 5: Variations on let

Stephen Checkoway

What values does this code return?

```
(define (foo x)
  (let ([y (add1 x)]
        [z (* 2 x)])
    (+ y z)))
(foo 3)
```

A. 10

B. 11

C. 12

D. Some other value

E. Error

What values does this code return?

```
(define (bar x)
  (let ([x (add1 x)]
        [z (* 2 x)])
    (+ x z)))
(bar 3)
```

A. 10

B. 11

C. 12

D. Some other value

E. Error

A common problem

When writing programs, it's not uncommon to define some local variables in terms of other local variables

Example: Return the elements of a list of numbers that are at least as large as the first element (the head) of the list, in reverse order

```
(define (at-least-as-large lst)
  (cond [(empty? lst) empty]
        [else
          (let ([head (first lst)]
                [bigger (filter (λ (x) (>= x head)) lst)])
            (reverse bigger))]))
```

This doesn't work; we can't use `head` in the definition of `bigger`

The issue

The issue is the scope of the binding for `head`: just the body of the `let`

One (bad) work around would be to use multiple `lets`

```
(define (at-least-as-large lst)
  (cond [(empty? lst) empty]
        [else
          (let ([head (first lst)])
            (let ([bigger (filter (λ (x) (>= x head)) lst)])
              (reverse bigger)))]))
```

Sequential let

`(let* ([id1 s-exp1] [id2 s-exp2]…) body)`

Later s-exps can use earlier ids, e.g.,

```
(let* ([x 5]
       [y (foo x)]
       [z (+ x y)])
  (bar z y))
```

Returning to our example

```
(define (bar x)
  (let* ([x (add1 x)]
         [z (* 2 x)])
    (+ x z)))
(bar 3)
```

A more realistic example

Write a procedure (split-by pred lst) that splits lst into two lists, the first contains all of the elements that match pred, the second contains all the elements that do not match pred

```
(split-by even? (range 10)) => '((0 2 4 6 8) (1 3 5 7 9))

(split-by (λ (x) (< x 3)) (range 5)) => '((0 1 2) (3 4))

(define (split-by pred lst)
```

Another problem: recursion

Often, we're going to want to define a recursive procedure but we can't do that with `let` or `let*`

```
(let ([fact (λ (n)
              (if (<= n 1)
                  1
                  (* n (fact (sub1 n)))))])
  (fact 5))
```

We can't use `fact` in the definition of `fact`

Recursive let

`(letrec ([id1 s-exp1] [id2 s-exp2]…) body)`

All of the s-exps can refer to all of the ids

- This is used to make recursive procedures

```
(letrec ([fact (λ (n)
                 (if (<= n 1)
                     1
                     (* n (fact (sub1 n)))))])
  (fact 5))
```

Recursive let drawback (subtle)

The values of the identifiers we're binding can't be used in the bindings

Invalid (the value of `x` is used to define `y`)

- ```
  (letrec ([x 1]
           [y (+ x 1)])
    y)
  ```

Valid (the value of `x` isn't used to *define* `y`, only when `y` is called)

- ```
  (letrec ([x 1]
           [y (λ () (+ x 1))])
    (y))
  ```