605.629: Programming Languages
Assignment 06
Sabbir Ahmed

October 17, 2021

1. [20 pts, Scheme]

Write a function count so that `(count a lat)` counts how often the atom `a` appears in the list of atoms `lat`.

```
> (count 'a '(d a t a))
2
> (count 's '(m i s s i s s i p p i))
4
```

Answer

```
(define (count a lat)
  (if (null? lat)                   ; if list is empty
    0                               ; return 0
    (if (equal? a (car lat))        ; else if a equals the first value of lat
      (+ 1 (count a (cdr lat))) ; recurse function with remainder of lat and add 1
      (count a (cdr lat)))))
```

---

2. [20 pts, Scheme]

Write a function `(count-all lat)` that produces a list pairing every individual atom in the list of atoms `lat` with a count of how often the atom appears in `lat`.

```
> (count-all '(m i s s i s s i p p i))
((m 1) (i 4) (s 4) (p 2))
> (count-all '(to be or not to be that is the question))
((to 2) (be 2) (or 1) (not 1) (that 1) (is 1) (the 1) (question 1))
```

The order of the items in the output list does not matter. Make sure to include the code if you use any auxiliary functions (helper functions). Note that this function is computing a histogram.

Answer

```
(define (count a lat)
  (if (null? lat)                   ; if list is empty
    0                               ; return 0
    (if (equal? a (car lat))        ; else if a equals the first value of lat
      (+ 1 (count a (cdr lat))) ; recurse function with remainder of lat and add 1
      (count a (cdr lat)))))

(define (make-unique hist term)
  (fold (lambda (alist result)      ; fold lambda into hist
    (if (equal? (car alist) term)   ; if term equals the first value of the alist
      result                        ; return the result of the lambda
      (cons alist result)))         ; else append cons pair of the alist and the result
  '() hist))

(define (count-all lat)
  (fold (lambda (term hist)                 ; fold lambda into lat
    (cond
      ((assoc term hist) => (lambda (x)     ; if term already exists in hist alist
          (cons                             ; replace value in hist
            (cons term (count term lat))    ; increment previous frequency of term
            (make-unique hist term))))      ; only the new term-freq pair is in hist
      (else =>
          (cons (cons term 1) hist))))      ; append default frequency to hist
  '() lat))
```

---

3. [20 pts, Scheme]

Consider the following code:

```
(define mystery
  (lambda (x)
    (cond
       ((null? x) (list x))
      (else (append
        (mystery (cdr x))
        (map (lambda (y)
          (cons (car x) y))
          (mystery (cdr x))))))))
```

a. What is the value of `(mystery '(a b c))`?

Answer

Breaking down the function:

```
(define mystery
  (lambda (x)
    (cond
      ((null? x) (list x))  ; if nil, return empty list
      (else (append         ; else append the following
        (mystery (cdr x))   ; return value of the recursive call with the rest of x
        (map (lambda (y)    ; map the lambda with the recursive call
          (cons (car x) y)) ; make a pair of the first value of x and y
          (mystery (cdr x))))))))
```

The function appears to create lists with all the possible combinations of the elements of the input list. The following is generated from using an interpreter:

```
(() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

---

b. Suggest a better name for mystery that better conveys its purpose.

Answer

The function appears to create a power set from the input list. `mystery` can be renamed to `power-set`.

---

c. What is the value of `(length (mystery '(a b c d e f g h i j)))`?

Answer

To find the number of power sets possible for a set of n elements, compute 2^n. Since `(length '(a b c d e f g h i j)) == 10`, then `(length (mystery '(a b c d e f g h i j)))` $= 2^{10} = 1024$.

---

4. [50 pts, Scheme] (including 10 pts bonus)

Recall that in lexical binding (or scoping) a variable reference is associated with the nearest lexically enclosing binding of the variable. Dynamic binding (or scoping) resolves a variable reference in the body of a function in the environment at the point of call (extended with bindings of actuals to formals). Below are expressions in a Scheme-like syntax. Evaluate each using obvious meanings in both a lexical and a dynamic setting. If something happens to prevent straightforward evaluation, explain the "error" succinctly. You may assume that each of these is to be evaluated in a pristine initial environment: no previous definitions have been made.

For each of the code pieces below write down Lexical and Dynamic evaluations of the expressions.

1.

```scheme
(let ((x 22))
    (let ((f (lambda (y) x)))
        (let ((x 30))
            (f 42))))
```

Resolution of lexical:

```
(f 42)
(lambda (42) x)
(lambda (42) 22)
22
```

Resolution of dynamic:

```
(f 42)
(x 30)
(lambda (42) 30)
30
```

2.

```
(let ((f (lambda (y) x)))
    (let ((x 33))
        (f 420)))
```

Resolution of lexical

```
(f 420)
(lambda (420) x)
x
```

Resolution of dynamic

```
(f 420)
(x 33)
(lambda (420) 33)
33
```

3.

```
(let ((f (lambda (x)
    (if (= x 0)
        0
        (+ x (f (1 - x)))))))
    (f 10))
```

Resolution of lexical

```
(f 10)
(lambda (10) (= 10 0))
(= 10 0)
(+ 10 f (-9))
(f -9)
(lambda (-9) (= -9 0))
(= -9 0)
(+ 10 f (10))
(f 10)
(lambda (10) (= 10 0))
(= 10 0)
(+ 10 f (-9))
(f -9)
(lambda (-9) (= -9 0))
(= -9 0)
(+ 10 f (10))
...
```

The function reaches its maximum recursion depth.

Resolution of dynamic

```
(f 10)
(lambda (10) (= 10 0))
(= 10 0)
(+ 10 f (-9))
(f -9)
(lambda (-9) (= -9 0))
(= -9 0)
(+ 10 f (10))
(f 10)
(lambda (10) (= 10 0))
(= 10 0)
(+ 10 f (-9))
(f -9)
(lambda (-9) (= -9 0))
(= -9 0)
(+ 10 f (10))
...
```

The function reaches its maximum recursion depth.

4.

```
(let ((a 2))
    (let ((f (lambda () a)))
        (let ((g (lambda (x) (f))))
            (let ((a 4))
                (g 1)))))
```

Resolution of lexical

```
(g 1)
(lambda (1) f)
(lambda (1) a)
(lambda (1) 2)
2
```

Resolution of dynamic

```
(g 1)
(a 4)
(lambda (1) f)
(f a)
(f 4)
4
```

5.

```
(let ((a 2))
    (let ((f (lambda () a)))
        (let ((g (lambda (a) (f))))
            (let ((a 4))
                (g 1)))))
```

Resolution of lexical

```
(g 1)
(lambda (1) f)
(lambda (1) a)
(lambda (1) 2)
2
```

Resolution of dynamic

```
(g 1)
(a 4)
(lambda (1) f)
(f a)
(f 4)
4
```
