foldr, end of lexical scoping

Review of foldr

`foldr` (sometimes also called accumulate, reduce, or inject) is another very famous iterator over recursive structures

Accumulates an answer by repeatedly applying **`f`** to answer so far

- **`(foldr f base (x1 x2 x3 x4))`** computes
  `(f x1 (f x2 (f x3 (f x4 base))))`

```
(define (foldr f base lst)
  (if (null? lst) base
    (f (car lst)
       (foldr f base (cdr lst)))))
```

- This version “folds right”; another version “folds left”
- Whether the direction matters depends on `f` (often not)

Examples with foldr

These are useful and do not use “private data”

```
(define (f1 lst) (foldr + 0 lst))
(define (f2 lst)
  (foldr (lambda (x y) (and (>= x 0) y)) #t lst))
```

These are useful and do use “private data”

```
(define (f3 lo hi lst)
  (foldr
    (lambda (x y)
      (+ (if (and (>= x lo) (<= x hi)) 1 0) y))
    0 lst))

(define (f4 g lst)
  (foldr (lambda (x y) (and (g x) y)) #t lst))
```

Lexical scoping vs dynamic scoping

- The alternative to lexical scoping is called dynamic scoping.
- In dynamic scoping, if a function f references a non-local variable x, the language will look for x in the function that **called** f.
  - If it's not found, will look in the function that called the function that called f (and so on).
- Contrast with lexical scoping, where the language would look for x in the scope where f was **defined**.

Why lexical scope?

1. **Function meaning does not depend on variable names used**

Example: Can change body to rename a variable **`q`** instead of **`x`**

- Lexical scope: guaranteed to have no effects
  Dynamic scope: might change function

```
(define (f y)
  (let ((x (+ y 1)))
    (lambda (z) (+ x y z)))
```

When the anonymous function that f returns is called, in lexical scoping, we always know where the values of x, y, and z will be (what frames they're in). With dynamic scoping, x and y will be searched for in the functions that called the anonymous function, so who knows where they'll be.

Why lexical scope?

1. **Function meaning does not depend on variable names used**

Example: Can remove unused variables in lexical scoping

– Dynamic scope: May change meaning of a function (weird)

```
(define (f g)
  (let ((x 3))
    (g 2)))
```

– You would never write this in a lexically-scoped language, because the binding of x to 3 is never used.

  - (No way for g to access this particular binding of x.)

– In a dynamically-scoped language, g might refer to a non-local variable x, and this binding might be necessary.

Why lexical scope?

2. Easy to reason about functions where they're defined.

```
(define x 1)
(define (f y)
    (+ x y))
(define g
  (let ((x "hello"))
    (f 4))
```

Example: Dynamic scope tries to add a string to a number (b/c in the call to (+ x y), x will be "hello")

Why lexical scope?

3. Closures can easily store the data they need
   - Many more examples and idioms to come

```
(define (gteq x) (lambda (y) (>= y x)))
(define (no-negs lst) (filter (gteq 0) lst))
```

- The anonymous function returned by gteq references a non-local variable x.
- In lexical scoping, the closure created for the anonymous function will point to gteq's frame so x can be found.
- In dynamic scoping, x would not be found at all.

Does dynamic scope exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages
- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Racket) have special ways to do it
  - But most don't bother
- Historically, dynamic scoping was used more frequently in older languages because it's easier to implement than lexical scoping.
  - Strategy: Just search through the call stack until variable is found. No closures needed.
  - Call stack maintains list of functions that are currently being called, so might as well use it to find non-local variables.

Iterators made better

- Functions like **`map`** and **`filter`** are *much* more powerful thanks to closures and lexical scope

- Function passed in can use any “private” data in its environment

- Iterator (e.g., map or filter) “doesn’t even know the data is there”
  - It just calls the function that it's passed, and that function will take care of everything.