CS450

Structure of Higher Level Languages

Lecture 08: foldl, looping last-to-first; tail-recursion

Tiago Cogumbreiro

Looping from last-to-first

Reversing a list

Exercise: write copy in terms of foldr

Using our recursion pattern

```
(define (copy l)
  (match l ; (list 1 2 3)
    [(list) (list)]
    [(list h l ...)
      ; h = 1
      ; l = (list 2 3)
      (define result (copy l))
      ; result = (list 2 3)
      (cons h result)
      ; output = (list 1 2 3)
      ]
  ))
```

Exercise: write copy in terms of foldr

Using our recursion pattern

```
(define (copy l)
  (match l ; (list 1 2 3)
    [(list) (list)]
    [(list h l ...)
      ; h = 1
      ; l = (list 2 3)
      (define result (copy l))
      ; result = (list 2 3)
      (cons h result)
      ; output = (list 1 2 3)
      ]
  ))
```

Using foldr

```
(define (copy-foldr l)
  (foldr
    ; step
    (lambda (h result)
      (cons h result)
    )
    ; base-case
    (list)
    ; list we are iterating over
    l
  )
)
```

Reversing a list

Aka copy in reverse

```
(define (reverse-foldr l)
  (foldr
    ; step
    (lambda (h result)
      (cons h result)
    )
    ; base-case
    (list)
    ; list we are iterating over
    l
  )
)
```

- The difference between `foldr` and `foldl` is the traversal order
- Copying is traversing last-to-first (since `cons` adds to the l-h-s)
- Reversing is traversing first-to-last (first becomes last with `cons`)

UMass Boston

Reversing

Adding to the right-hand-side

```
(define (cons-right x l)
  ; x = 4
  ; l = (list 1 2 3)
  (match l
    [(list) (list x)]
    [(list h l ...)
      ; h = 1
      ; l = (list 2 3)
      (define result (cons-right x l))
      ; result = (list 2 3 4)
      (cons h result)
      ; output = (list 1 2 3 4)
    ]
    ))
```

```
(define (reverse-slow l)
  (match l ; (list 1 2 3)
    [(list) (list)]
    [(list h l ...)
      ; h = 1
      ; l = (list 2 3)
      (define result (reverse-slow l))
      ; result = (list 3 2)
      (cons-right h result)
    ]
  )
)
```

Reversing

Adding to the right-hand-side

- The problem with `cons-right` is that it needs to traverse the whole list
- Reversing effectively traverses the list $\sum_{i \leq n} i = \frac{(n-1)n}{2}$ times, thus makes a linear-time operation into quadratic!
- **Can you implement `cons-right` and `reverse-slow` with either foldl or with foldr?**

UMass Boston

Reversing a list

Implement function `(reverse l)` that reverses a list.

Spec

```
(check-equal? (list 4 3 2 1) (reverse (list 1 2 3 4)))
```

UMass Boston

Reversing a list

Implement function `(reverse l)` that reverses a list.

Spec

```
(check-equal? (list 4 3 2 1) (reverse (list 1 2 3 4)))
```

Solution

```
(define (reverse l)
  (define (rev accum l)
    (match l
      [(list) accum]
      [(list h l ...) (rev (cons h accum) l)]))
  (rev l (list)))
```

- We use a parameter `accum` as a "loop variable", that is, a variable that we update at each iteration.
- The idea behind `rev` is that recursing can be thought of as "calling" the next iteration of the loop where we update the loop variables (ie, `accum` and `l` with their new respective values)

UMass Boston

Joining strings in Racket

Joining strings in Python

```
>>> ", ".join(["x", "y", "z"])
"x, y, z"
>>> ", ".join([])
""
>>> ", ".join(["x"])
"x"
```

UMass Boston

Joining strings in Racket

```
(require rackunit)
(check-equal? (join ", " '("x" "y" "z")) "x, y, z")
(check-equal? (join ", " '()) "")
(check-equal? (join ", " '("x")) "x")
```

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**
  "x, y, " + "z"
  How do you check the last?

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**
  "x, y, " + "z"
  How do you check the last?
- **Separator as a prefix:**

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**
  "x, y, " + "z"
  How do you check the last?
- **Separator as a prefix:**
  ", x, y, z"

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**
  "x, y, " + "z"
  How do you check the last?
- **Separator as a prefix:**
  ", x, y, z"
- **Separator as a prefix, but not on first:**

UMass Boston

Joining strings in Racket

- **Separator as a suffix:**
  "x, y, z, "
- **Separator as a suffix, but not on last:**
  "x, y, " + "z"
  How do you check the last?
- **Separator as a prefix:**
  ", x, y, z"
- **Separator as a prefix, but not on first:**
  "x" + ", y, z"

UMass Boston

Separator as suffix

```
(define (join-suffix sep l)
  (match l
    [(list) ""]
    [(list x) x]
    [(list h l ...)
      (string-append h sep (join-suffix sep l))
    ]
  )
)
```

- We can use pattern `(list x)` to identify the ***last element***
- When we reach the last-element, we do not add a suffix
- Otherwise, (third pattern) we suffix the element with the separator.
- **Insight:** recursion+pattern matching lets us easily mention the last k elements.
- A problem of this solution is that it is less trivial to adapt it to `fold`s.

Separator as a prefix

```
(define (join-prefix sep l)
  (match l
    [(list) ""]
    [(list h l ...)
      (define (join-iter l)
        (match l
          [(list) ""]
          [(list h l ...)
            ; h = "b"
            ; l = (list "c")
            (string-append sep h
              (join-iter l))
            ; output = ", b, c"
          ]
        )
      )
      (string-append h (join-iter l))]))
```

- If the list is empty, there is no separator to handle.
- If the list is non-empty, we handle the first element differently, the rest of the list is handled recursively.
- Recursively, traverse last-to-first, so add the separator as a prefix, in-order
- **Try implementing `join-iter` with foldr!**

UMass Boston

Separator as prefix

```
(define (join sep l)
  (define (join-iter accum l)
    (match l
      [(list) accum]
      [(list h l ...)
        (join-iter
         (string-append accum sep h)
         l)]))
  (match l
    [(list) ""]
    [(list h l ...) (join-iter h l)]))
```

- This solution uses a parameter `accum` to build the list, rather than handling the result of recursion
- This solution is more complicated, but faster (next lecture)

UMass Boston

Imperative versus functional

```
def join(sep, l):
   if l == []:
     return ""
   accum = l[0]
   l = l[1:]
   for x in l:
     accum = accum + sep + x
   return accum

>>> l = ["x", "y", "z"]
>>> join(", ", l)
'x, y, z'
```

```
(define (join sep l)
  (define (join-iter accum l)
    (match l
      [(list) accum]
      [(list h l ...)
        (join-iter
         (string-append accum sep h)
         l)]))
  (match l
    [(list) ""]
    [(list h l ...) (join-iter h l)]))
```

Another pattern arises

```
; Example 1
(define (reverse l)
  (define (rev accum l)
    (match l
      [(list) accum]
      [(list h l ...) (rev (cons h accum) l)]))
  (rev l (list)))
; Example 2
(define (join sep l)
  (define (join-iter accum l)
    (match l
      [(list) accum]
      [(list h l ...)
        (join-iter
         (string-append accum sep h)
         l)]))
  (match l
    [(list) ""]
    [(list h l ...) (join-iter h l)]))
```

A generalized recursion pattern for lists

```
(define (rec base-case l)
  (match l
    [(list) base-case]
    [(list h l ...)
      (rec (step h accum) l)]))
```

For instance,

```
(cons h accum)
```

maps to

```
(step h accum)
```

Implementing this recursion pattern

Recursive pattern for lists

```
(define (rec base-case l)
  (match l
    [(list) base-case]
    [(list h l ...)
      (rec (step h accum) l)]))
```

Fold left reduction

```
(define (foldl step base-case l)
  (match l
    [(list) base-case]
    [(list h l ...)
      (foldl (step h accum) l)]))
```

Implementing join with foldl

Before

```
(define (join sep l)
  (define (join-iter accum l)
    (match l
      [(list) accum]
      [(list h l ...)
        (join-iter
         (string-append accum sep h)
         l)]))
  (match l
    [(list) ""]
    [(list h l ...) (join-iter h l)]))
```

Implementing join with foldl

Before

```
(define (join sep l)
  (define (join-iter accum l)
    (match l
      [(list) accum]
      [(list h l ...)
        (join-iter
         (string-append accum sep h)
         l)]))
  (match l
    [(list) ""]
    [(list h l ...) (join-iter h l)]))
```

After

```
(define (join sep l)
  (match l
    [(list) ""]
    [(list h l ...)
          (define (step elem accum)
            (string-append accum sep elem))
          (foldl step h l)]))
```

Python version suggested by Dakai Tzou:

```
from functools import reduce
def join(l, sep):
  if l == []:
    return l
  def step(elem, accum):
    return elem + sep + accum
  return reduce(step, l[1:], l[0])
```

Implementing reverse with foldl

Original

```
(define (reverse l)
  (define (rev accum l)
    (match l
      [(list) accum]
      [(list h l ...) (rev (cons h accum) l)]))
  (rev l (list)))
```

Implementing reverse with foldl

Original

```
(define (reverse l)
  (define (rev accum l)
    (match l
      [(list) accum]
      [(list h l ...) (rev (cons h accum) l)]))
  (rev l (list)))
```

Solution

```
(define (reverse l)
  (define (on-elem elem accum)
    (cons elem accum))
  (foldl on-elem (list) l))
```

or

```
(define (reverse l)
  (foldl cons (list) l))
```

UMass Boston

Exercise

> Write a function `(to-string l)` that takes a list of numbers and renders them as `[n1, n2, ...]`. Example, `(list-to-string '(1 2 3))` should output `[1, 2, 3]`. Function `number->string` converts a number into a string

Exercise

> Write a function `(to-string l)` that takes a list of numbers and renders them as `[n1, n2, ...]`. Example, `(list-to-string '(1 2 3))` should output `[1, 2, 3]`. Function `number->string` converts a number into a string

```
(define (to-string l)
  (string-append
    "["
    (join ", " (map number->string l))
    "]"))
```

1. convert a list of string to a list of numbers (so that `join` can use the list)
2. convert a list of numbers into `1, 2, 3`
3. Surround the string with brackets.

UMass Boston