Folding over Lists

CIS352 — Spring 2023

Kris Micinski

Iterating over a list to accumulate a result is one of the most typical programming patterns

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd)
         (cons hd (filter f tl))
         (filter f tl))]))
```

What do all these functions have in common?

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each matches on the list

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each returns an **initial** value

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each of them makes a recursive call and then **combines the result with hd**

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))

(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))

(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Let’s think about how sum-list operates over lists…

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(sum-list (cons 1 (cons 2 ‘())))
     … => (+    1 (+    2 0))
```

You can think of this as replacing cons with + and ‘() with 0

Now let’s look at list-product

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(list-product (cons 1 (cons 2 ‘())))
         … => (*    1 (*    2 1))
```

You can think of ***this*** as replacing cons with * and ‘() with 1

```
(fold f i     (cons 1 (cons 2 ‘())))
        … => (f    1 (f    2 i))
```

Folds abstract this common pattern:

- Iterating over list to **accumulate** some result
- Some **default** or **initial** value to handle empty list
- Some two-argument **reducer** function
  - Combines first element w/ processed tail

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write sum-list

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write list-product

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write filter-list

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

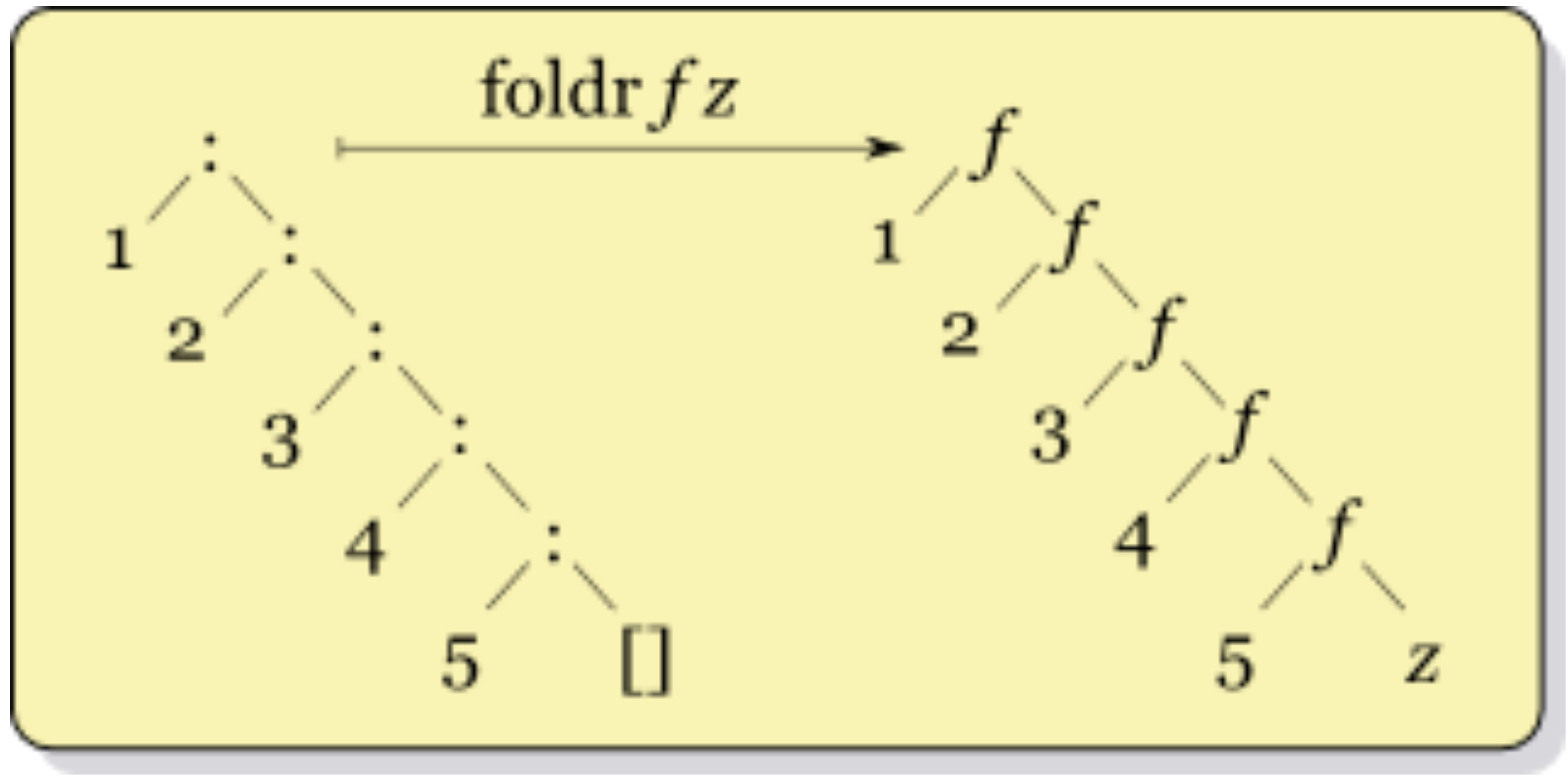
This version of fold is **direct-style**, meaning it will push stack frames

```
(define (foldr reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (foldr reducer init tl))]))
```

This version of fold is **direct-style**, meaning it will push stack frames

```
(define (foldr reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Traditionally this is called a “right” fold because it bottoms out at the end (right side) of the list, and reconstructs back up.

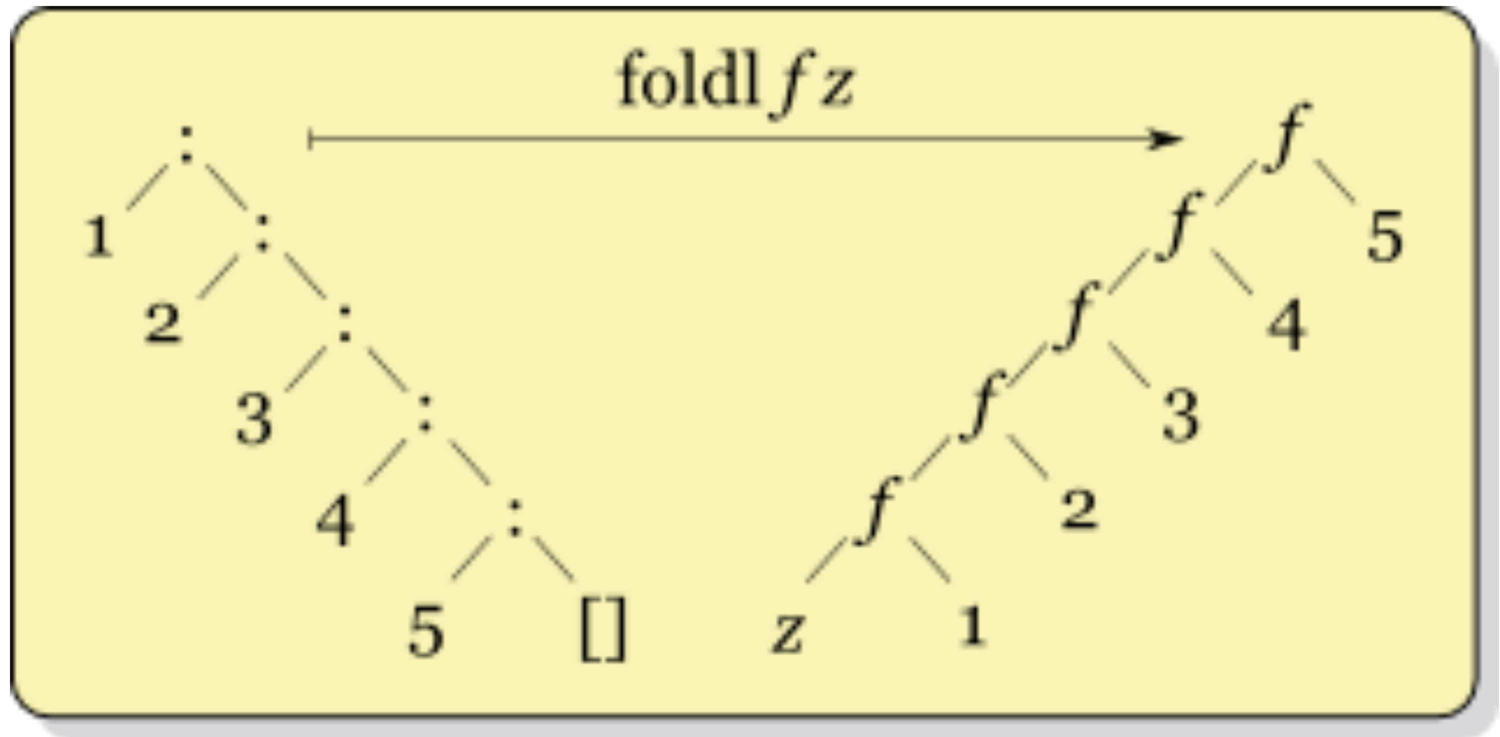


* **Diagram from the Haskell wiki**

We can also write a **tail-recursive** version of fold by swapping the argument order to reducer

```
(define (foldl reducer acc lst)
  (match lst
    ['() acc]
    [`(,hd . ,tl)
     (foldl reducer (reducer hd acc) tl)]))
```

This is called a **left fold** because it "starts" from the left (reducer will be called on first element w/ the "zero")



* Diagram from the Haskell wiki

Exercise

Use foldl to write reverse

```
(define (foldl reducer acc lst)
  (match lst
    ['() acc]
    [`(,hd . ,tl)
     (fold reducer (reducer hd acc) tl)]))
```

Biggest takeaways for you:

- Consider using fold when possible
- Use Racket's foldl or foldr
  - Mostly the same, but process list differently
- You need a two argument **reducer** function
- You need an **initial value**