CS339: Abstractions and Paradigms for Programming

Higher Order List Functions

Manas Thakur
CSE, IIT Bombay

Autumn 2025

Repetitions galore

- Recall the program to sum a list:

```
(define (sum l)
  (if (null? l)
      0
      (+ (car l) (sum (cdr l)))))
```

- What about product?

```
(define (prod l)
  (if (null? l)
      1
      (* (car l) (prod (cdr l)))))
```

- Length?

```
(define (length l)
  (if (null? l)
      0
      (+ 1 (length (cdr l)))))
```

Along with similar syntax, what are we doing in each of these?

Folding a list to a single value!

The fold higher-order operation

```
(define (foldr f v l)
  (if (null? l)
      v
      (f (car l) (foldr f v (cdr l)))))
```

Also called reduce.

- Updated sum:

```
(define (sum l)
  (foldr + 0 l))
```

- Product:

```
(define (prod l)
  (foldr * 1 l))
```

- Length?

```
(define (length l)
  (foldr (lambda (x y) (+ 1 y))
         0 l))
```

What was the ‘r’ in “foldr”?

```
(define (foldr f v l)
  (if (null? l)
      v
      (f (car l) (foldr f v (cdr l)))))
```

Folds from *right*.

```
> (define l (list 1 2 3 4))
> (foldr + 0 l)
```

```
(+ 1 (+ 2 (+ 3 (+ 4 0))))
```

- We also have a foldl:

```
(define (foldl f v l)
  (if (null? l)
      v
      (foldl f (f v (car l)) (cdr l))))
```

Folds from *left*.

```
> (foldl + 0 l)
```

```
(+ 0 1) ==> (+ 1 2) ==>
(+ 3 3) ==> (+ 6 4) ==> 10
```

Transforming lists

- Add 10 to each element of a list:

```
(define (add10 l)
  (if (null? l)
      nil
      (cons (+ 10 (car l)) (add10 (cdr l)))))
```

- How about squaring each element?

```
(define (sqr-list l)
  (if (null? l)
      nil
      (cons (square (car l)) (sqr-list (cdr l)))))
```

We are *mapping* a list to another list by applying a common function to all its elements!

The map higher-order operation

```
(define (map f l)
  (if (null? l)
      nil
      (cons (f (car l)) (map f (cdr l)))))
```

- What's squaring a list now?

```
(define (sqr-list l)
  (map square l))
```

- The new add10 function:

```
(define (add10 l)
  (map (lambda (x) (+ x 10)) l))
```

Map-based functions usually are also great candidates for parallelization!

Something complex (for now!)

- Compute the sum of the squares of the even numbers of a list:

```
(define (sum-sqr-evn l)
  (if (null? l)
      0
      (let ((x (car l)))
        (if (even? x)
            (+ (square x) (sum-sqr-evn (cdr l)))
            (sum-sqr-evn (cdr l))))))
```

Now keep this aside for some time.

Filtering lists

- Return numbers from a list that are >10:

```
(define (gt10 l)
  (if (null? l)
      nil
      (if (> (car l) 10)
          (cons (car l) (gt10 (cdr l)))
          (gt10 (cdr l)))))
```

- Return even numbers from a list:

```
(define (get-evn l)
  (if (null? l)
      nil
      (if (even? (car l))
          (cons (car l) (get-evn (cdr l)))
          (get-evn (cdr l)))))
```

You know what we're going to do next!

The filter higher-order operation

```
(define (filter pred l)
  (if (null? l)
      nil
      (let ((x (car l)))
        (if (pred x)
            (cons x (filter pred (cdr l)))
            (filter pred (cdr l))))))
```

- The new get-evn:

```
(define (get-evn l)
  (filter even? l))
```

- The new gt10:

```
(define (gt10 l)
  (filter (lambda (x) (> x 10)) l))
```

Now we can solve complex problems so elegantly!

- Get a list containing squares of the even numbers of a list:

```
(define (sqr-evn l)
  (map square (filter even? l)))
```

Next Class
We begin with OO (ooo)

- Compute the sum of the squares of the even numbers of a list:

```
(define (sum-sqr-evn l)
  (foldr + 0 (map square (filter even? l))))
```

Programming this way is called using sequences as conventional interfaces — a topic we would revisit again (that time, for efficiency) and enjoy a lot!

Comparison

```
(define (sum-sqr-evn l)
  (if (null? l)
      0
      (let ((x (car l)))
        (if (even? x)
            (+ (square x) (sum-sqr-evn (cdr l)))
            (sum-sqr-evn (cdr l))))))
```