CSCI-C311 Programming Languages

Racket: Lists, Iteration, Recursion

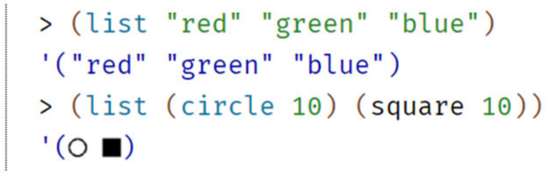
Dr. Hang Dinh

Reading Assignment for This Lecture

- [Tutorial] Quick: An Introduction to Racket with Pictures
  - https://docs.racket-lang.org/quick/index.html
  - Part 8 - Lists
- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - 2.3 Lists, Iteration, and Recursion

The `list` function

- Rackets inherits LISP → list is an important part of Racket
- The list function takes any number of arguments and returns a list containing the given values:



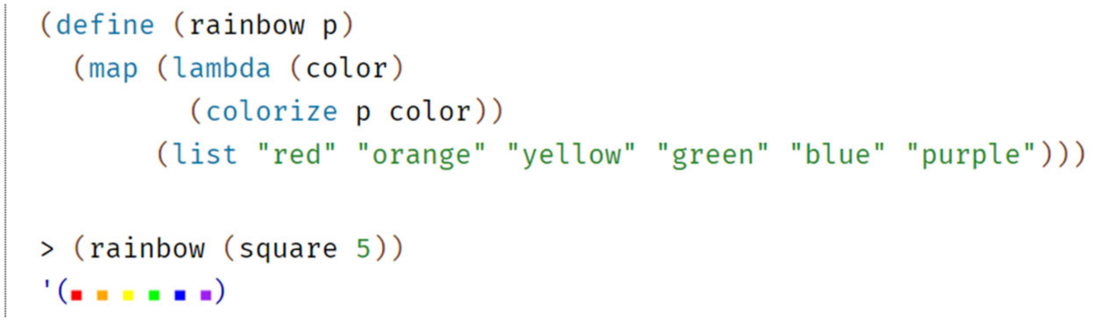
```
> (list "red" "green" "blue")
'("red" "green" "blue")
> (list (circle 10) (square 10))
'(○ ■)
```

  - A list prints as a single quote and then pair of parentheses wrapped around the printed form of the list elements.

3

The `map` function

- The map function takes a list and a function to apply to each element of the list; it returns a new list to combine the function's results



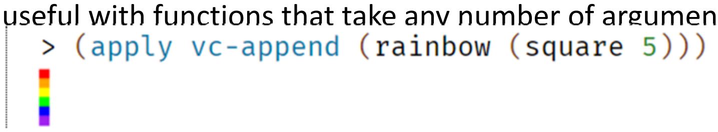
```
(define (rainbow p)
  (map (lambda (color)
         (colorize p color))
       (list "red" "orange" "yellow" "green" "blue" "purple")))

> (rainbow (square 5))
'(🟥 🟧 🟨 🟩 🟦 🟪)
```

4

The apply function

- Like map, the apply function takes a function and a list,
  - but a function given to apply should take all of the arguments at once, instead of each one individually.
  - useful with functions that take any number of arguments, such as vc-append

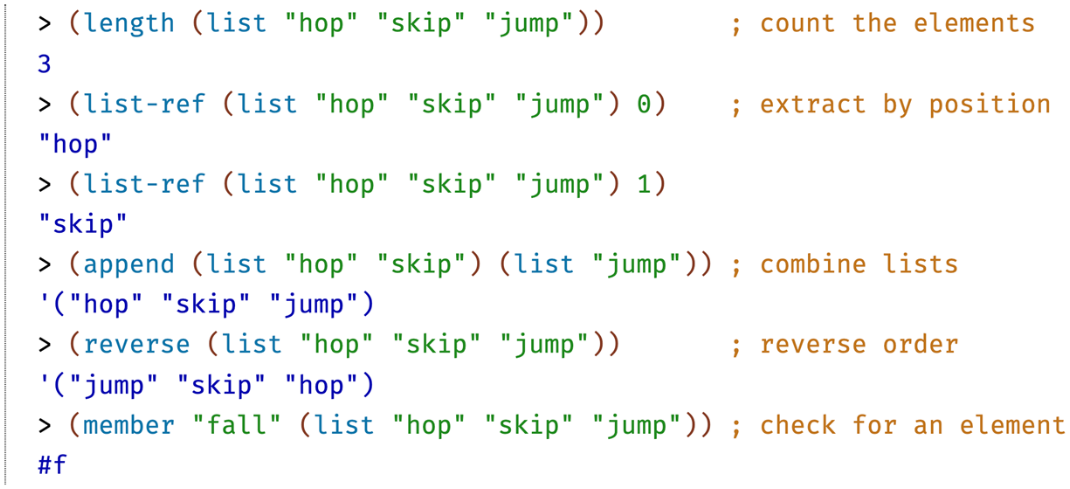


```
> (apply vc-append (rainbow (square 5)))
```

- The apply function bridges the gap between a function that wants many arguments and a list of those arguments as a single value
  - (vc-append (rainbow (square 5))) would not work,
  - because vc-append does not want a list as an argument; it wants a picture as an argument, and it is willing to accept any number of them.

Predefined Functions Operating on Lists

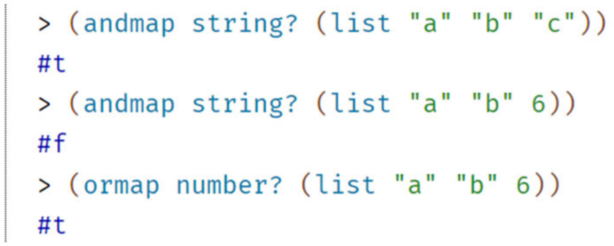
- Examples: Simple operations on lists



```
> (length (list "hop" "skip" "jump"))        ; count the elements
3
> (list-ref (list "hop" "skip" "jump") 0)    ; extract by position
"hop"
> (list-ref (list "hop" "skip" "jump") 1)
"skip"
> (append (list "hop" "skip") (list "jump")) ; combine lists
'("hop" "skip" "jump")
> (reverse (list "hop" "skip" "jump"))       ; reverse order
'("jump" "skip" "hop")
> (member "fall" (list "hop" "skip" "jump")) ; check for an element
#f
```

Predefined List Loops

- Racket includes functions that iterate over the elements of a list
  - These iteration functions play a role similar to for in Java, and other languages
  - Typically combined with `lambda` forms.
- Different iteration functions combine iteration results in different ways
  - The map function uses the per-element results to create a new list.
  - The andmap and ormap functions combine the results by anding or oring

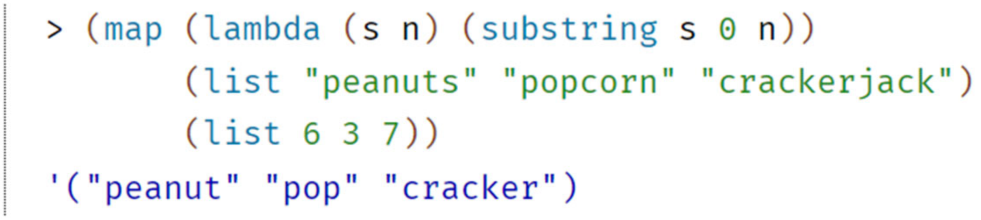


```
> (andmap string? (list "a" "b" "c"))
#t
> (andmap string? (list "a" "b" 6))
#f
> (ormap number? (list "a" "b" 6))
#t
```

8

Iteration Functions: `map` , `andmap`, `ormap`

- The map, andmap, and ormap functions can all handle multiple lists, instead of just a single list.
- The lists must all have the same length, and the given function must accept one argument for each list

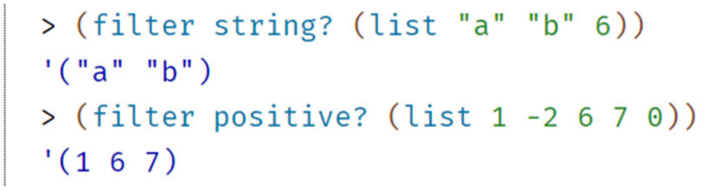


```
> (map (lambda (s n) (substring s 0 n))
       (list "peanuts" "popcorn" "crackerjack")
       (list 6 3 7))
'("peanut" "pop" "cracker")
```

9

Iteration Functions: filter, foldl

- The filter function keeps elements for which the body result is true, and discards elements for which it is #f:

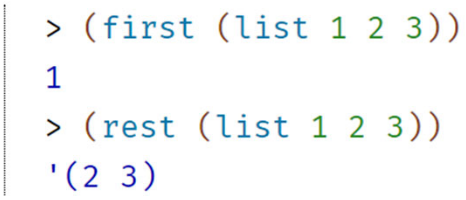


```
> (filter string? (list "a" "b" 6))
'("a" "b")
> (filter positive? (list 1 -2 6 7 0))
'(1 6 7)
```

- The foldl function generalizes some iteration functions.
  - foldl is not as popular as the other functions.
  - Since map, ormap, andmap, and filter cover the most common kinds of list loops.

List Primitives: first, rest

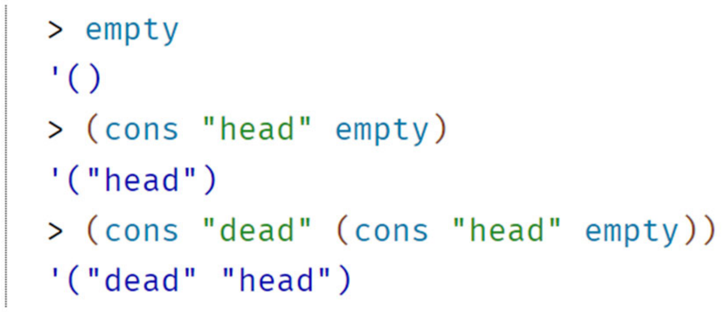
- You can write equivalent iterations using a handful of list primitives
- Since a Racket list is a linked list, the two core operations on a non-empty list are
  - first: get the first thing in the list; and
  - rest: get the rest of the list.



```
> (first (list 1 2 3))
1
> (rest (list 1 2 3))
'(2 3)
```

List Primitives: cons, constant empty

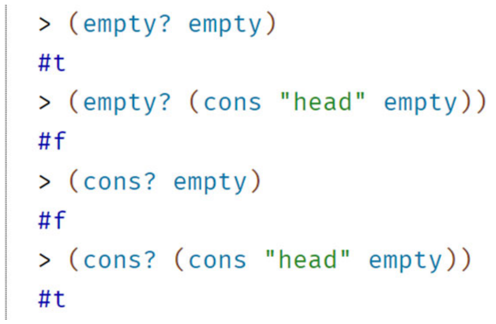
- To create a new node for a linked list—that is, to add to the front of the list—use the cons function ("construct")
- To get an empty list to start with, use the empty constant:



```
> empty
'()
> (cons "head" empty)
'("head")
> (cons "dead" (cons "head" empty))
'("dead" "head")
```

List Primitives: empty?, cons?

- To process a list, need to test if the list is non-empty
  - because first and rest work only on non-empty lists.
- The empty? function detects empty lists, and cons? detects non-empty lists:



```
> (empty? empty)
#t
> (empty? (cons "head" empty))
#f
> (cons? empty)
#f
> (cons? (cons "head" empty))
#t
```

List Iteration From Scratch: Recursion

- Write your own version of the length function using list primitives:

```
(define (my-length lst)
     (cond  [(empty? lst) 0]
            [else (+ 1 (my-length (rest lst)))]))
```

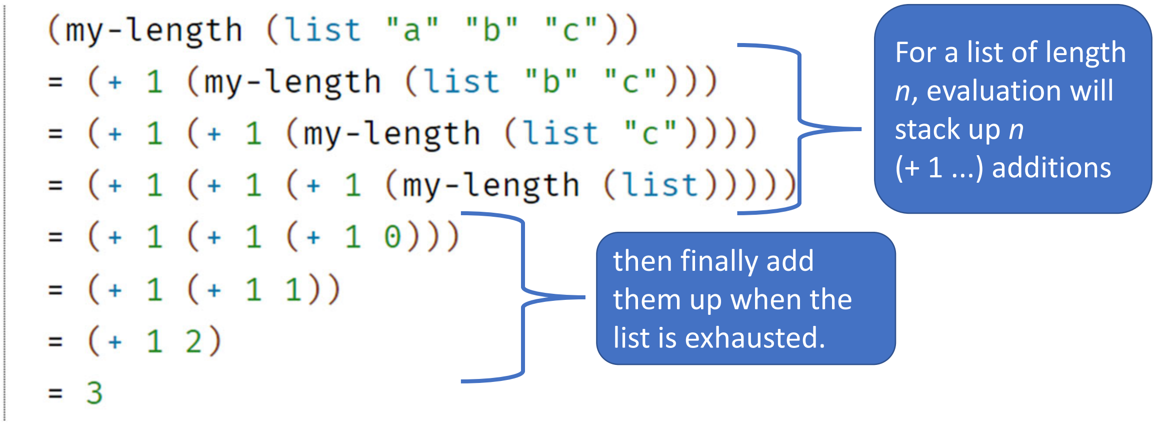
- Write your own version of the map function using list primitives:

```
(define (my-map f lst)
     (cond  [(empty? lst) empty]
            [else (cons (f (first lst))
                        (my-map f (rest lst)))]))
```

15

Tail Recursion Evaluation

- Both the my-length and my-map functions run in *O*(*n*) space for a list of length *n*

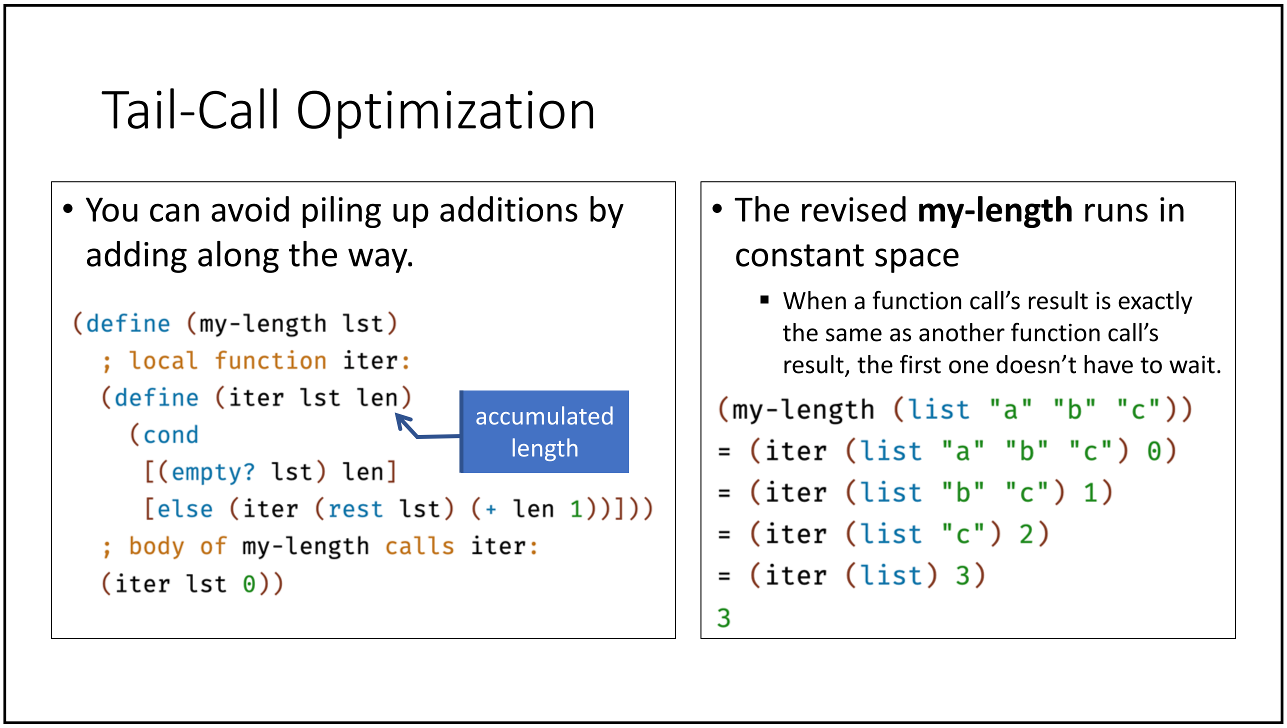


```
(my-length (list "a" "b" "c"))
= (+ 1 (my-length (list "b" "c")))
= (+ 1 (+ 1 (my-length (list "c"))))
= (+ 1 (+ 1 (+ 1 (my-length (list)))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
= (+ 1 2)
= 3
```

For a list of length *n*, evaluation will stack up *n* (+ 1 ...) additions

then finally add them up when the list is exhausted.

16



Tail-Call Optimization

- You can avoid piling up additions by adding along the way.

```
(define (my-length lst)
  ; local function iter:
  (define (iter lst len)
    (cond
     [(empty? lst) len]
     [else (iter (rest lst) (+ len 1))]))
  ; body of my-length calls iter:
  (iter lst 0))
```

accumulated length

- The revised **my-length** runs in constant space
  - When a function call's result is exactly the same as another function call's result, the first one doesn't have to wait.

```
(my-length (list "a" "b" "c"))
= (iter (list "a" "b" "c") 0)
= (iter (list "b" "c") 1)
= (iter (list "c") 2)
= (iter (list) 3)
3
```

17