List Recursion

Design of Programming Languages

Modified Slides of Lyn Turbak

Recursive List Functions in Racket

Because Racket lists are defined recursively, it's natural to process them recursively.

Typically (but not always) a recursive function `recf` on a list argument `L` has two cases:

- **base case:** what does `recf` return when L is empty? (Use `null?` to test for an empty list.)
- **recursive case:** if L is the nonempty list `(cons` ***Vfirst Vrest***`)` how are ***Vfirst*** and `(recf` ***Vrest***`)` combined to give the result for `(recf L)`?

  Note that we always apply `recf` directly to ***Vrest*** (and nothing else)!

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in words]

Step 1 (concrete example): pick a concrete input list, typically 3 or 4 elements long. What should the function return on this input?

E.g. A `sum` function that returns the sum of all the numbers in a list:
`(sum '(5 7 2 4))` $\Rightarrow^*$ `18`

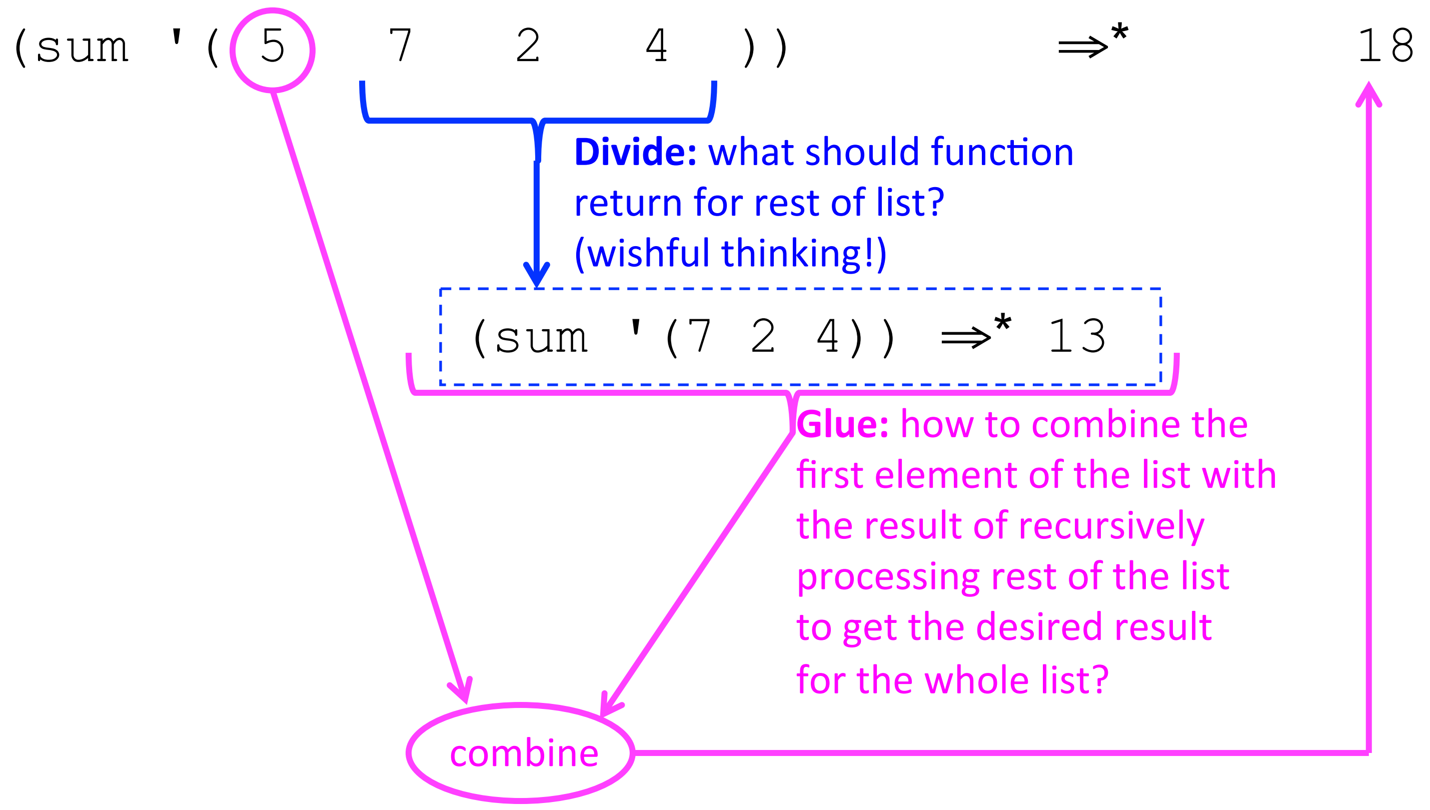
Step 2 (divide): without even thinking, **always** apply the function to the `rest` of the list. What does it return? `(sum '(7 2 4))` $\Rightarrow^*$ `13`

Step 3 (glue): How to combine the first element of the list (in this case, `5`) with the result from processing the rest (in this case, `13`) to give the result for processing the whole list (in this case, `18`)? `(+ 5 (sum '(7 2 4))` $\Rightarrow^*$ `18`

Step 4 (generalize): Express the general case in terms of an arbitrary input:

```
(define (sum nums)
   … (+ (first nums) (sum (rest nums)) … )
```

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in diagram]



(sum '(5 7 2 4)) ⇒* 18
Divide: what should function return for rest of list? (wishful thinking!)
(sum '(7 2 4)) ⇒* 13
Glue: how to combine the first element of the list with the result of recursively processing rest of the list to get the desired result for the whole list?
combine

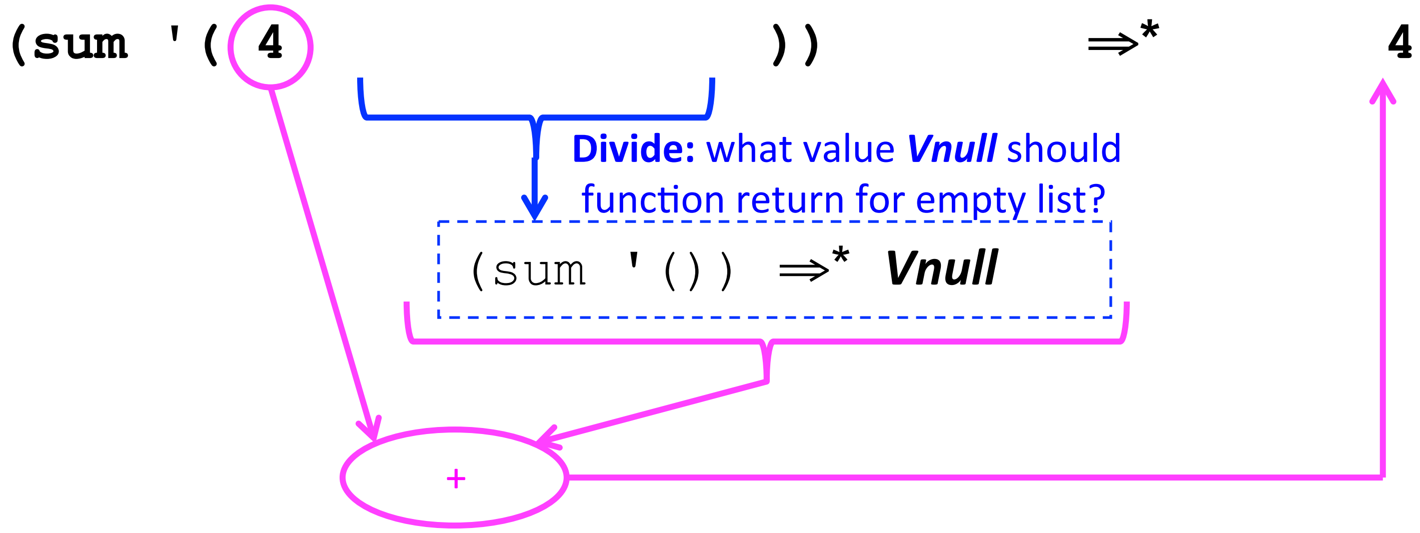
Solution for concrete example: `(+ 5 (sum '(7 2 4))`

Generalization of concrete solution: `(+ (first nums) (sum (rest nums))`

Recursive List Functions: base case via singleton case

Deciding what a recursive list function should return for the empty list is not always obvious and can be tricky. E.g. what should `(sum '())` return?

If the answer isn't obvious, consider the ``penultimate case" in the recursion, which involves a list of one element:



(sum '(4))
⇒* 4
Divide: what value Vnull should function return for empty list?
(sum '()) ⇒* Vnull
+

In this case, ***Vnull*** should be 0, which is the identity element for addition.

But in general it depends on the details of the particular combiner determined from the general case. So solve the general case before the base case!

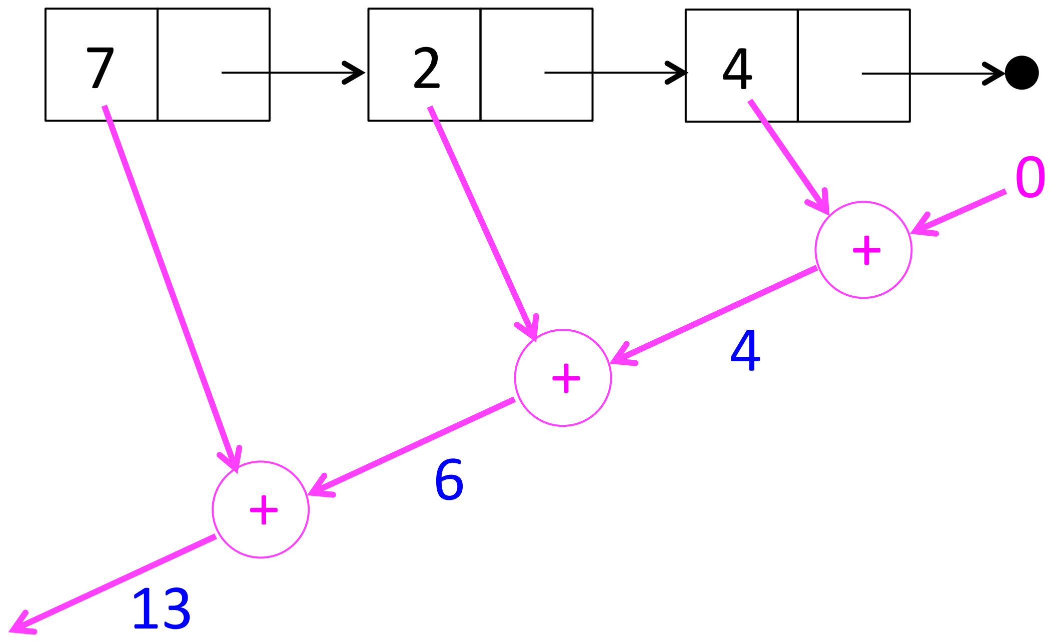
Putting it all together: base & general cases

`(sum nums)` returns the sum of the numbers in the list `nums`

```
(define (sum ns)
  (if (null? ns)
      0
      (+ (first ns)
         (sum (rest ns)))))
```

Understanding `sum`: Approach #1

```
(sum '(7 2 4))
```



We'll call this the **recursive accumulation** pattern

Understanding `sum`: Approach #2

In `(sum (list 7 2 4))`, the list argument to `sum` is

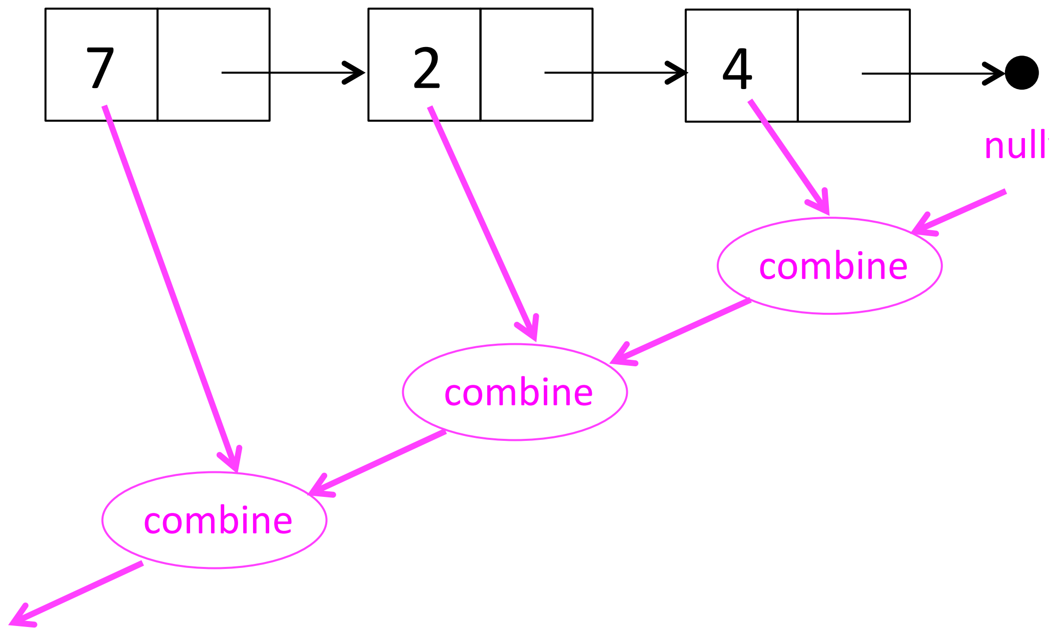
```
(cons 7 (cons 2 (cons 4 null))))
```

Replace `cons` by **+** and `null` by **0** and simplify:

```
(+ 7 (+ 2 (+ 4 0))))
⇒ (+ 7 (+ 2 4)))
⇒ (+ 7 6)
⇒ 13
```

Generalizing `sum`: Approach #1

```
(recf (list 7 2 4))
```



Generalizing `sum`: Approach #2

In `(recf (list 7 2 4))`, the list argument to `recf` is

```
(cons 7 (cons 2 (cons 4 null)))
```

Replace `cons` by **`combine`** and `null` by **`nullval`** and simplify:

```
(combine 7 (combine 2 (combine 4 nullval)))
```

Generalizing the `sum` definition

```
(define (recf ns)
  (if (null? ns)
      nullval
      (combine (first ns)
               (recf (rest ns)))))
```

Your turn

Define the following recursive list functions and test them in Racket:

`(product ns)` returns the product of the numbers in `ns`

`(min-list ns)` returns the minimum of the numbers in `ns`
Hint: use `min` and `+inf.0` (positive infinity)

`(max-list ns)` returns the minimum of the numbers in `ns`
Hint: use `max` and `-inf.0` (negative infinity)

`(all-true? bs)` returns `#t` if all the elements in `bs` are truthy; otherwise returns `#f`. *Hint*: use `and`

`(some-true? bs)` returns a truthy value if at least one element in `bs` is truthy; otherwise returns `#f`. Hint: use `or`

`(my-length xs)` returns the length of the list `xs`

Recursive Accumulation Pattern Summary

	combine	nullval
`sum`		
`product`		
`min-list`		
`max-list`		
`all-true?`		
`some-true?`		
`my-length`		

Define these using Divide/Conquer/Glue

it's *your* turn

```
> (snoc 11 '(7 2 4))
'(7 2 4 11)
```

```
> (my-append '(7 2 4) '(5 8))
'(7 2 4 5 8)
```

```
> (append-all '((7 2 4) (9) () (5 8)))
'(7 2 4 9 5 8)
```

```
> (my-reverse '(5 7 2 4))
'(4 2 7 5)
```

Mapping Example: `map-double`

it's your turn

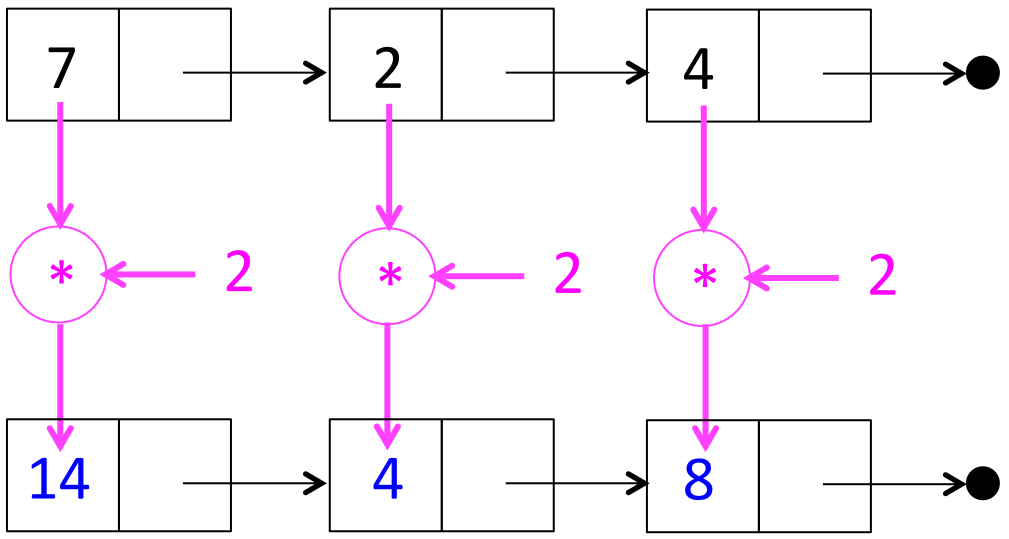
`(map-double ns)` returns a new list the same length as `ns` in which each element is the double of the corresponding element in `ns`.

```
> (map-double (list 7 2 4))
'(14 4 8)
```

```
(define (map-double ns)
  (if (null? ns)
      ; Flesh out base case

      ; Flesh out general case


      ))
```

Understanding `map-double`

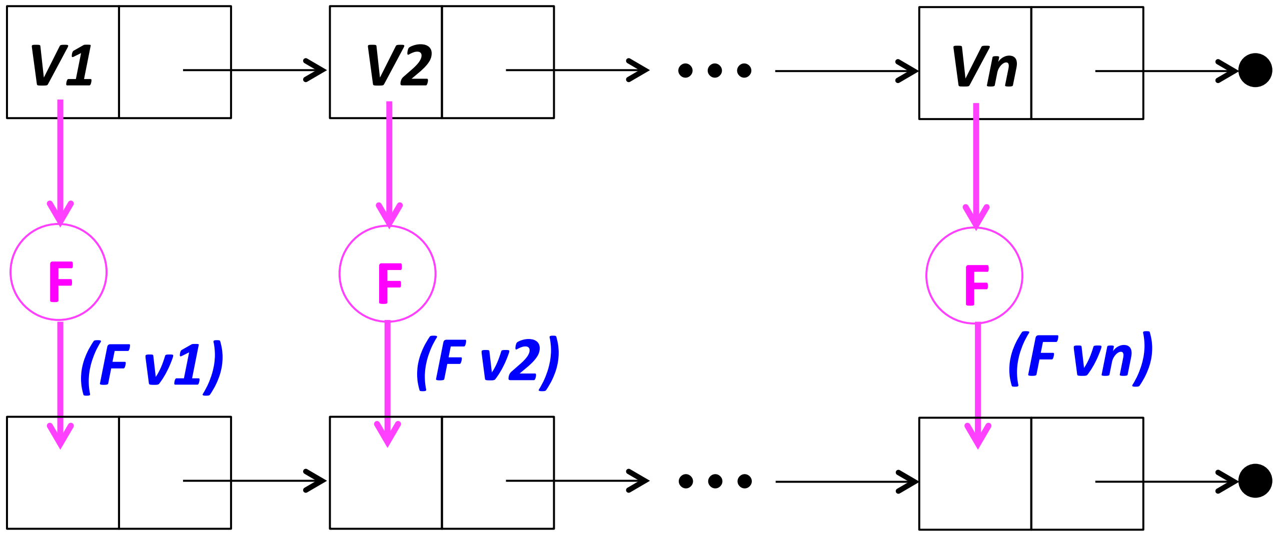
```
(map-double '(7 2 4))
```



We'll call this the **mapping** pattern

Generalizing `map-double`

`(mapF (list V1 V2 … Vn))`



```
(define (mapF xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (mapF (rest xs)))))
```

Expressing `mapF` as an accumulation

it's *your* turn

```
(define (mapF xs)
  (if (null? xs)
      null
      ((λ (fst subres)
                                       ) ; Flesh this out
       (first xs)
       (mapF (rest xs)))))
```

Some Recursive Listfuns Need Extra Args

```
(define (map-scale factor ns)
  (if (null? ns)
      null
      (cons (* factor (first ns))
            (map-scale factor (rest ns)))))
```

Filtering Example: `filter-positive`

it's *your* turn

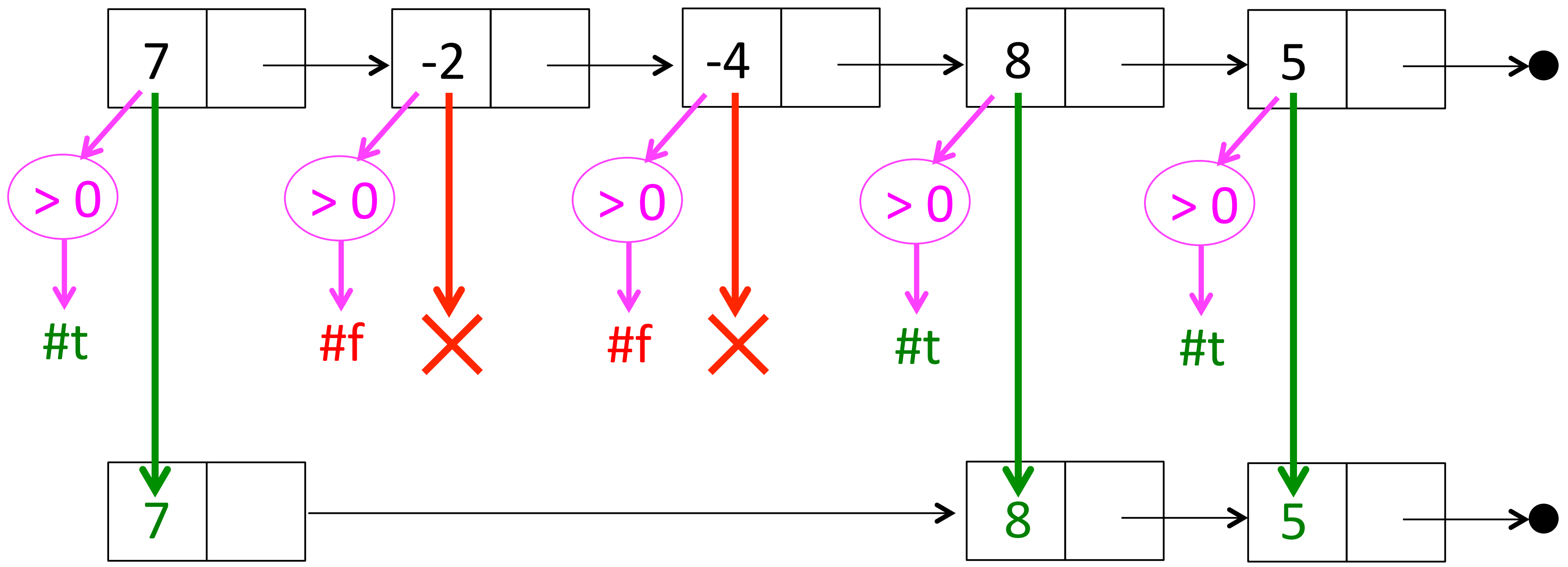
`(filter-positive ns)` returns a new list that contains only the positive elements in the list of numbers `ns`, in the same relative order as in `ns`.

```
> (filter-positive (list 7 -2 -4 8 5))
'(7 8 5)
```

```
(define (filter-positive ns)
  (if (null? ns)
      ; Flesh out base case

      ; Flesh out recursive case




      ))
```

Understanding `filter-positive`

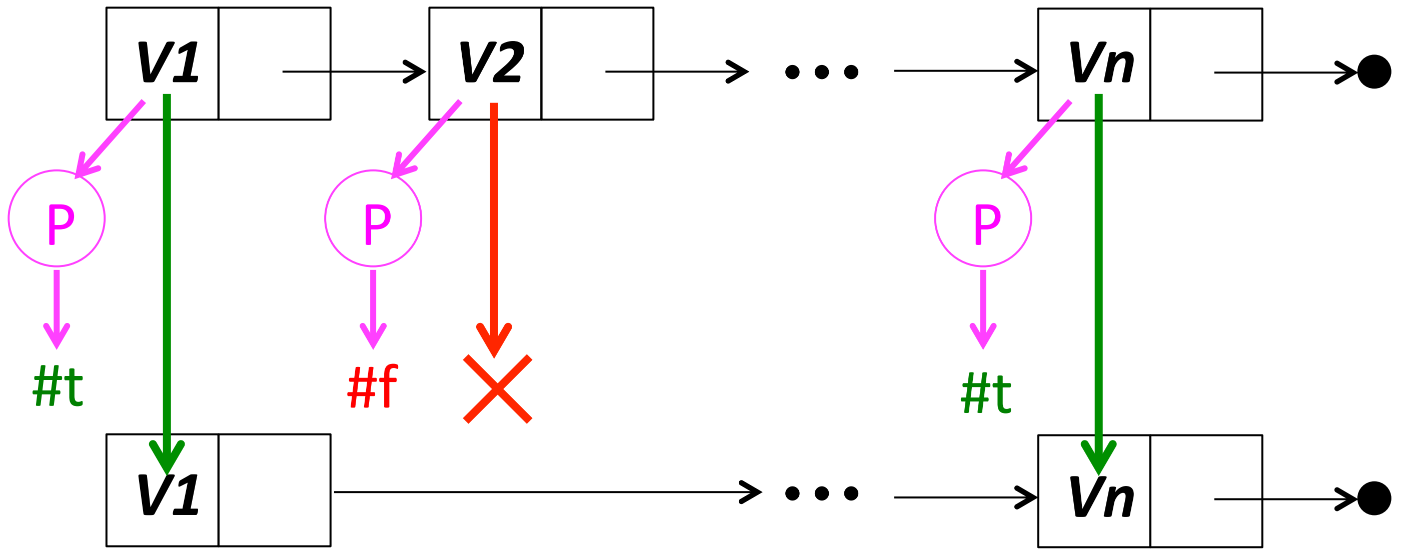
```
(filter-positive (list 7 -2 -4 8 5))
```



We'll call this the **filtering** pattern

Generalizing `filter-positive`

`(filterP (list V1 V2 … Vn))`



```
(define (filterP xs)
  (if (null? xs)
      null
      (if (P (first xs))
          (cons (first xs) (filterP (rest xs)))
          (filterP (rest xs)))))
```

Expressing `filterP` as an accumulation

```
(define (filterP xs)
  (if (null? xs)
      null
      ((lambda (fst subres)



                     ) ; Flesh this out
        (first xs)
        (filterP (rest xs)))))
```

Recursive Accumulation Pattern Summary Solutions

	combine	nullval
`sum`	`(λ (fst subres) (+ fst subres))` simpler: + Note: below we show only simpler form, if it exists	`0`
`product`	`*`	`1`
`min-list`	`min`	`+inf.0`
`max-list`	`max`	`-inf.0`
`all-true?`	`and`	`#t`
`some-true?`	`or`	`#f`
`my-length`	`(λ (fst subres) (+ 1 subres))`	`0`

Define these using Divide/Conquer/Glue **Solutions**

```
> (snoc 11 '(7 2 4))
'(7 2 4 11)
```

```
(define (snoc y xs)
  (if (null? xs)
      (list y)
      (cons (first xs) (snoc y (rest xs)))))
```

```
> (my-append '(7 2 4) '(5 8))
'(7 2 4 5 8)
```

```
(define (my-append xs ys)
  (if (null? xs)
      ys
      (cons (first xs)
            (my-append (rest xs) ys))))
```

```
> (append-all
     '((7 2 4) (9) () (5 8)))
'(7 2 4 9 5 8)
```

```
(define (append-all xss) ; xss means list
  (if (null? xss)        ; of list of elts
      '()
      (my-append (first xss)
                 (append-all (rest xss)))))
```

```
> (my-reverse '(5 7 2 4))
'(4 2 7 5)
```

```
(define (my-reverse xs)
  (if (null? xs)
      '()
      (snoc (first xs)
            (my-reverse (rest xs))
```

Mapping Example: `map-double` **Solutions**

it's *your* turn

`(map-double ns)` returns a new list the same length as `ns` in which each element is the double of the corresponding element in `ns`.

```
> (map-double (list 7 2 4))
'(14 4 8)
```

```
(define (map-double ns)
  (if (null? ns)
      ; Flesh out base case
      '()  ; Can also write null or ns
      ; Flesh out general case
      (cons (* 2 (first ns))
            (map-double (rest ns)))
      ))
```

Expressing `mapF` as an accumulation **Solutions**

it's *your* turn

```
(define (mapF xs)
  (if (null? xs)
      null
      ((λ (fst subres)
          (cons (F fst) subres) ) ; Flesh this out
       (first xs)
       (mapF (rest xs)))))
```

Filtering Example: `filter-positive` **Solutions**

it's your turn

`(filter-positive ns)` returns a new list that contains only the positive elements in the list of numbers `ns`, in the same relative order as in `ns`.

```
> (filter-positive (list 7 -2 -4 8 5))
'(7 8 5)
```

```
(define (filter-positive ns)
  (if (null? ns)
      ; Flesh out base case
      '()  ; Can also write null or ns
      ; Flesh out recursive case
      (if (> (first ns) 0)
          (cons (first ns)
                (filter-positive (rest ns)))
          (filter-positive (rest ns)))
      ))
```

Expressing `filterP` as an accumulation **Solutions**

it's
your
turn

```
(define (filterP xs)
  (if (null? xs)
       null
       ((lambda (fst subres)
           (if (P fst)
               (cons fst subres)
               subres) ) ; Flesh this out
        (first xs)
        (filterP (rest xs)))))
```