Loops in Scheme, II

(early slides assume map/filter)

Recap: filter and map

- filter and map are Scheme’s “loops”
  - filter : (α → boolean) list[α] → list[α]
    extract list of elts that satisfy a predicate
  - map : (α → β) list[α] → list[β]
    applies function to all elts, returning list of results

Recall sum

```
;; sum : list[num] → num
;; adds up the elements of a list of numbers
(define (sum alon)
   (cond [(empty? alon) 0]
              [(cons? alon) (+ (first alon)
                                        (sum (rest alon)))]))
```

Sum also loops; how to write it with filter/map?

[try it]

filter/map don’t work for sum

- Both return lists -- sum returns a number
- Sum requires another kind of loop
- We derived filter by looking at two programs with similar structure and abstracting the common parts into a helper function …

sum and product

```
;; sum : list[num] → num
;; adds elts of a list of nums
(define (sum alon)
   (cond
    [(empty? alon) 0]
    [(cons? alon)
      (+ (first alon)
           (sum (rest alon)))]))
```

```
;; prod : list[num] → num
;; multiplies list of nums
(define (prod alon)
   (cond
    [(empty? alon) 1]
    [(cons? alon)
      (* (first alon)
           (prod (rest alon)))]))
```

Where do these two programs differ?

sum and product

```
;; sum : list[num] → num
;; adds elts of a list of nums
(define (sum alon)
   (cond
    [(empty? alon) 0]
    [(cons? alon)
      (+ (first alon)
           (sum (rest alon)))]))
```

```
;; prod : list[num] → num
;; multiplies list of nums
(define (prod alon)
   (cond
    [(empty? alon) 1]
    [(cons? alon)
      (* (first alon)
           (prod (rest alon)))]))
```

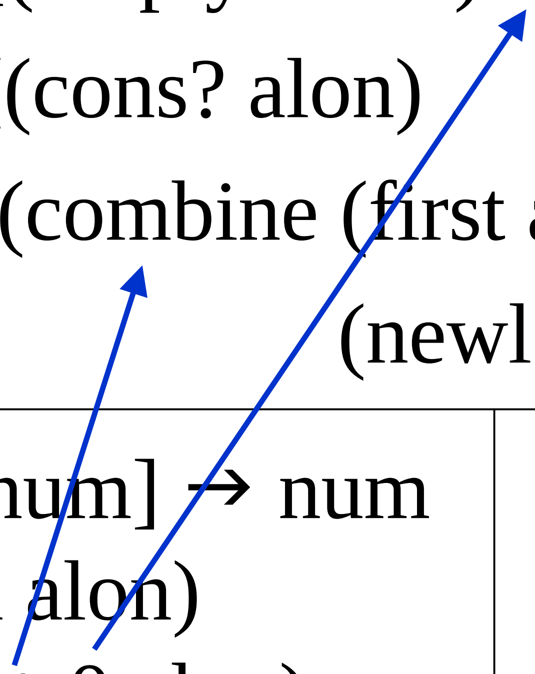
Make the blue parts parameters to a new function [try it]

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

Write sum and product using newloop [try it]

The “New Loop”



```
;; newloop : _?_ num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

```
;; sum : list[num] → num
(define (sum alon)
  (newloop + 0 alon))
```

```
;; prod : list[num] → num
(define (prod alon)
  (newloop * 1 alon))
```

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                              (newloop (rest alon)))]))
```

Write length (of a list) using newloop [try it]

base and alon arguments are easy … but combine …

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

What is combine’s contract? [try it]

;; combine : ______ ______ → ______

(we see from its use that it takes two arguments)

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                                (newloop (rest alon)))]))
```

What is combine’s contract?

;; combine : ______ ______ → ______

What type is (first alon)?   A number, by contract

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                           (newloop (rest alon)))]))
```

What is combine’s contract?

;; combine :   num   ______ → ______

What type is (first alon)?   A number, by contract

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                            (newloop (rest alon)))]))
```

What is combine’s contract?

;; combine : num ______ → ______

What type is (newloop (rest alon))?

A number, by contract

The “New Loop”

```
;; newloop : __?__ num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
         [(cons? alon)
          (combine (first alon)
                   (newloop (rest alon)))]))
```

What is combine’s contract?

;; combine : num num → ______

What type is (newloop (rest alon))?

A number, by contract

The “New Loop”

```
;; newloop : _?_ num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

A number
(by contract)
since
newloop
returns the
result of
combine

What is combine’s contract?

;; combine : num num → ______

What does combine return?

The “New Loop”

```
;; newloop : ___?___ num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

What is combine’s contract?

;; combine :   num   num   →   num

What does combine return?

A number (by contract) since newloop returns the result of combine

The “New Loop”

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
              [(cons? alon)
               (combine (first alon)
                               (newloop (rest alon)))]))
```

So, combine has contract

;; combine : num num → num

OK, but how do we write combine for length?

The “New Loop”

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
            [(cons? alon)
             (combine (first alon)
                             (newloop (rest alon)))]))
```

Combine takes the first elt of the list and the result of looping on the rest of the list. So, your combine function determines how to put these together …

The “New Loop”

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
            [(cons? alon)
             (combine (first alon)
                          (newloop (rest alon)))]))
```

```
;; combine : num num → num
(lambda (elt result-rest)
    …)
```

(this naming convention on combine functions reminds you what the args stand for)

The “New Loop”

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
         [(cons? alon)
          (combine (first alon)
                   (newloop (rest alon)))]))
```

```
;; combine : num num → num
(lambda (elt result-rest)
   (+ 1 result-rest))
```

For length, we don’t care about the contents of the elt, just that it exists. Combine therefore ignores elt.

The “New Loop”

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                              (newloop (rest alon)))]))
```

```
;; length : list[α] → num
(define (length alst)
   (newloop (lambda (elt result-rest) (+ 1 result-rest))
                    0 alst))
```

[stretch break]

But wait …

```
;; newloop : (num num → num) num list[num] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

The contracts don’t match!

```
;; length : list[α] → num
(define (length alst)
  (newloop (lambda (elt result-rest) (+ 1 result-rest))
           0 alst))
```

Fixing the newloop contract

```
;; newloop : (num num → num) num list[α] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                              (newloop (rest alon)))]))
```

```
;; length : list[α] → num
(define (length alst)
   (newloop (lambda (elt result-rest) (+ 1 result-rest))
                     0 alst))
```

If we change num to α, what else must change in the newloop contract?

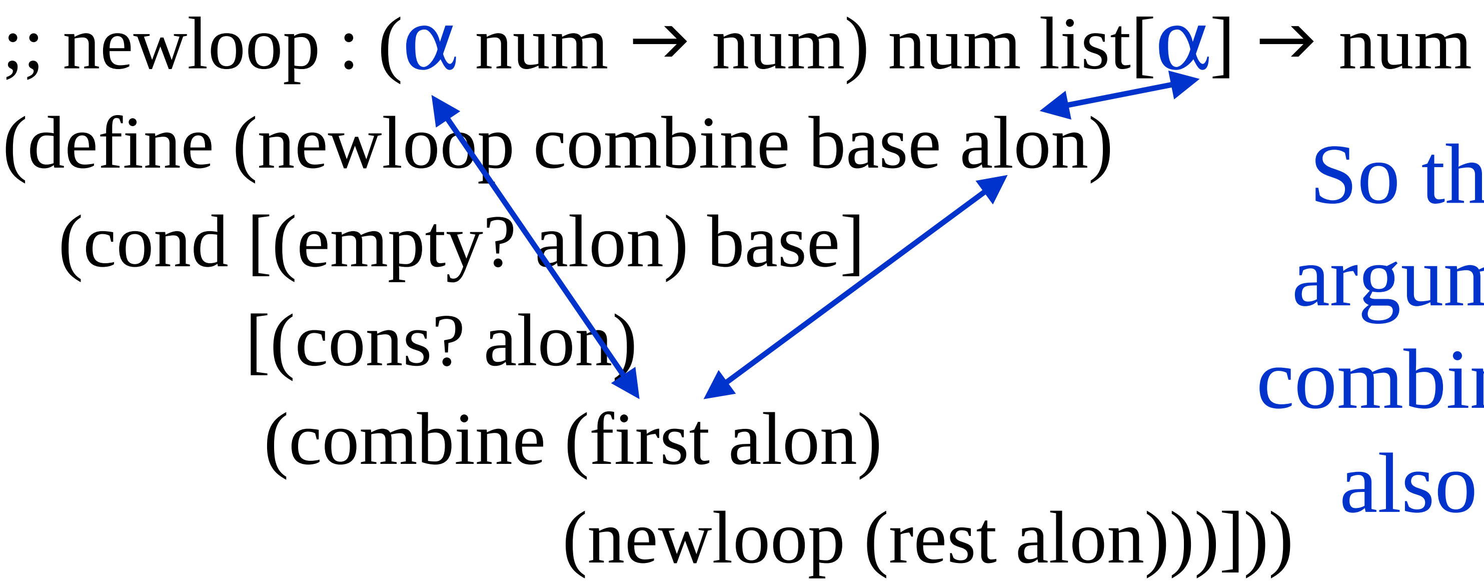
Fixing the newloop contract

```
;; newloop : (num num → num) num list[α] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                            (newloop (rest alon)))]))
```

Where is the α processed in newloop?

```
;; length : list[α] → num
(define (length alst)
   (newloop (lambda (elt result-rest) (+ 1 result-rest))
                 0 alst))
```

Fixing the newloop contract



```
;; newloop : (α num → num) num list[α] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                            (newloop (rest alon)))]))
```

So the first argument to combine must also be α

```
;; length : list[α] → num
(define (length alst)
   (newloop (lambda (elt result-rest) (+ 1 result-rest))
                    0 alst))
```

Fixing the newloop contract

```
;; newloop : (α num → num) num list[α] → num
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

This fixes the contract wrt length; now consider newloop alone

---

```
;; length : list[α] → num
(define (length alst)
  (newloop (lambda (elt result-rest) (+ 1 result-rest))
           0 alst))
```

Stepping back: newloop

```
;; newloop : (α num → num) num list[α] → num
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                            (newloop (rest alon)))]))
```

- What in the definition of newloop requires it to output a number? (newloop has no arith ops…)
- What if we wanted a loop that returned a boolean, or a structure, or …?

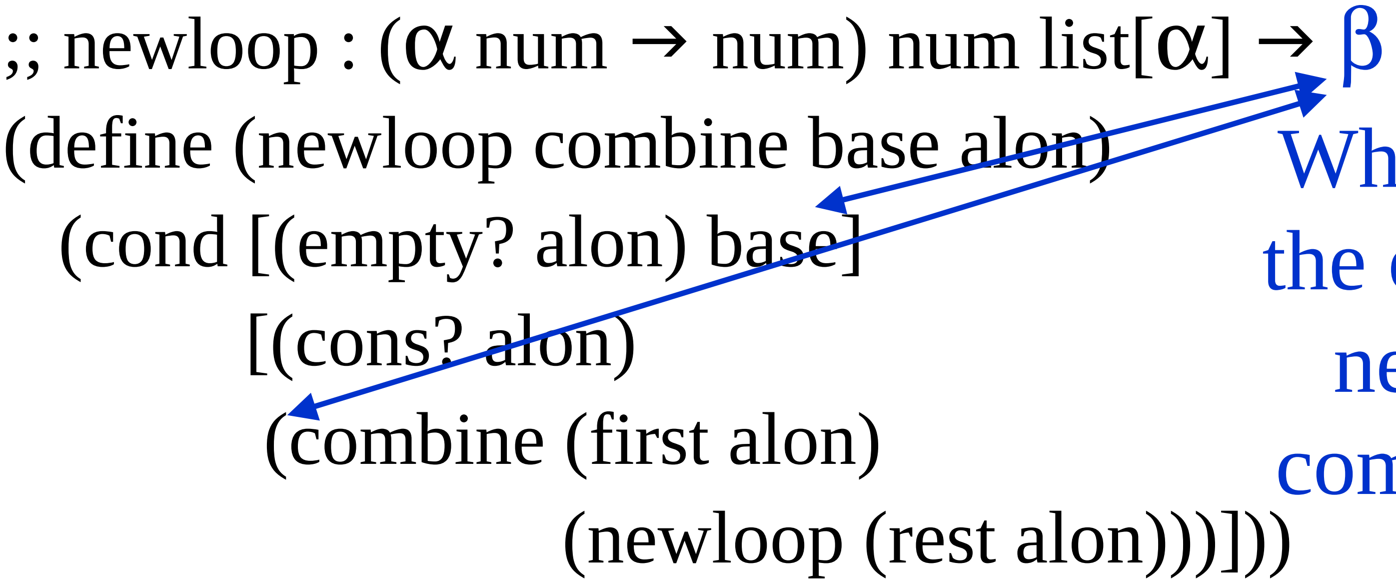
Generalizing newloop

```
;; newloop : (α num → num) num list[α] → β
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                              (newloop (rest alon)))]))
```

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop



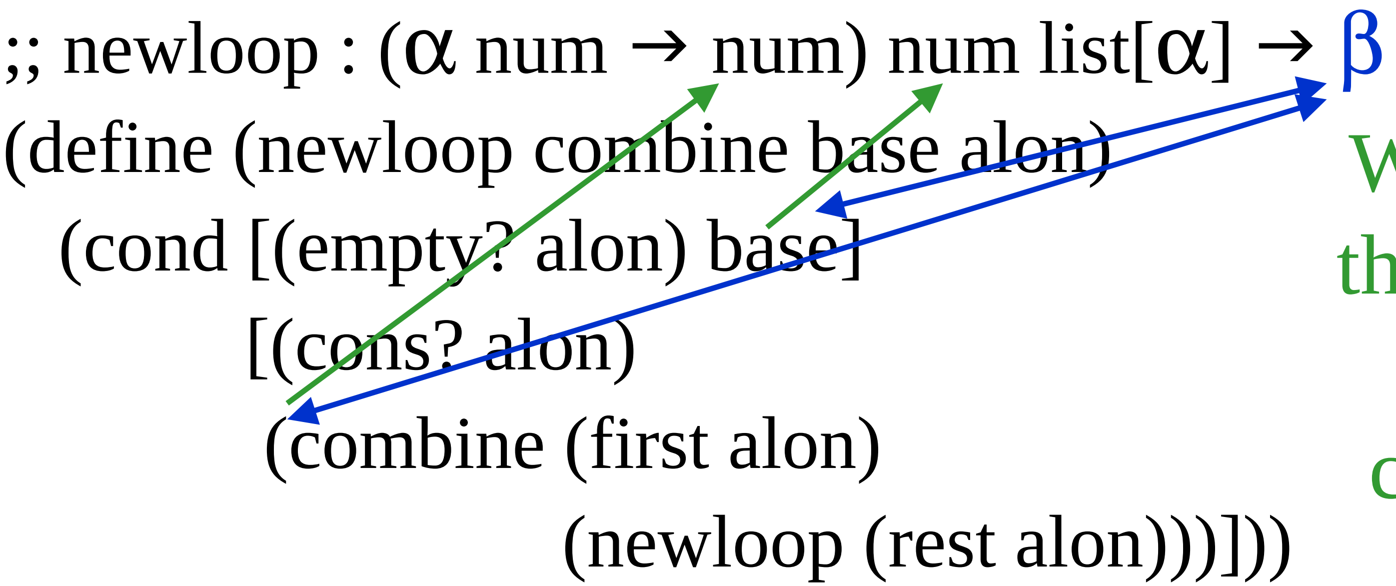
```
;; newloop : (α num → num) num list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

Where does the output of newloop come from?

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop



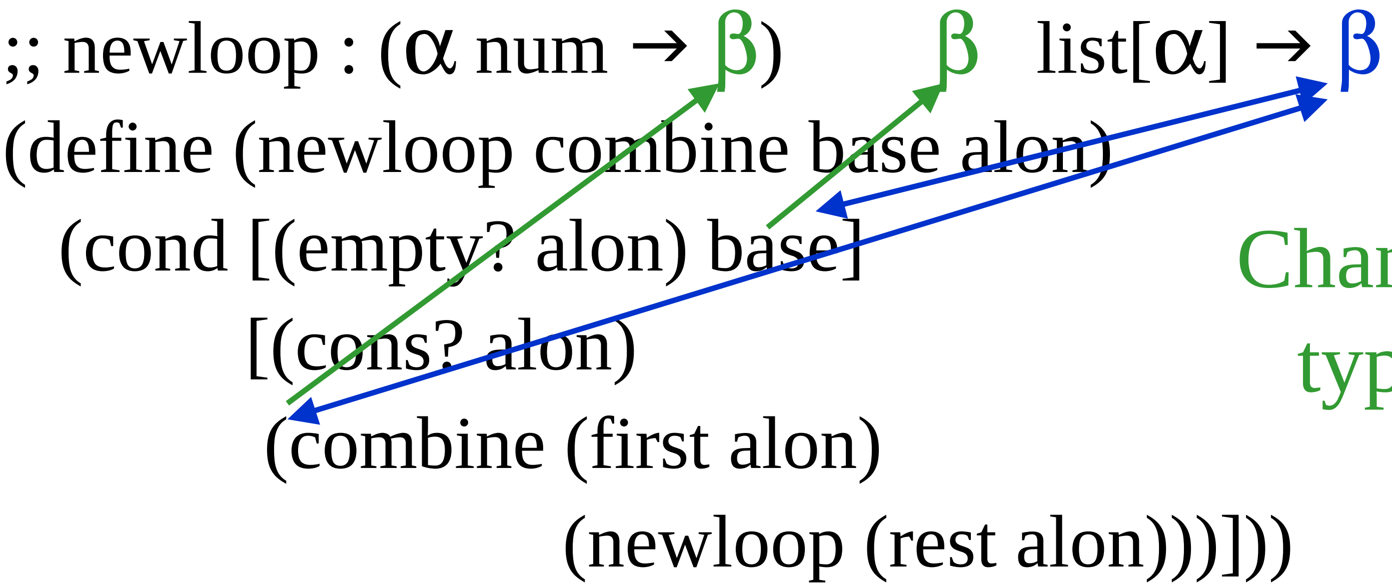
```
;; newloop : (α num → num) num list[α] → β
(define (newloop combine base alon)
   (cond [(empty? alon) base]
              [(cons? alon)
               (combine (first alon)
                                (newloop (rest alon)))]))
```

Where are these types in the contract?

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop



```
;; newloop : (α num → β)      β  list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

Change these types to β

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop

```
;; newloop : (α num → β)     β  list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

What about that lingering num? (where is it from)?

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop

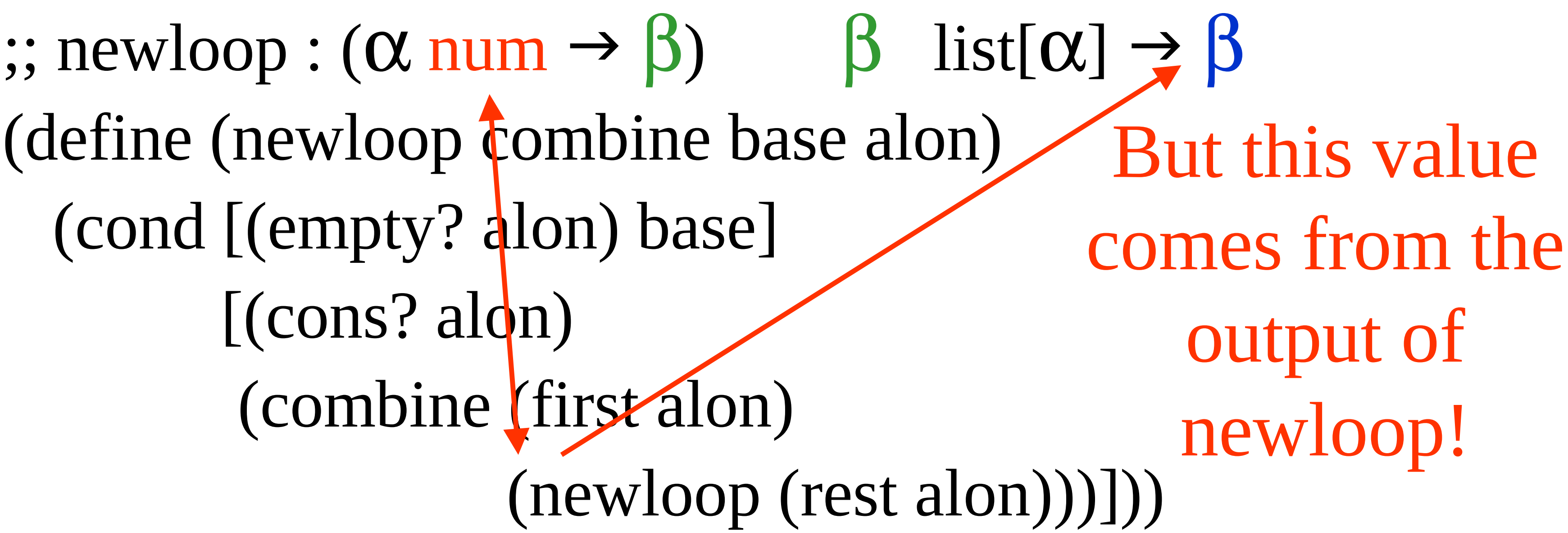
```
;; newloop : (α num → β)      β  list[α] → β
(define (newloop combine base alon)
   (cond [(empty? alon) base]
             [(cons? alon)
              (combine (first alon)
                              (newloop (rest alon)))]))
```

The num is the second argument to combine

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop



```
;; newloop : (α num → β)      β  list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

But this value comes from the output of newloop!

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

Generalizing newloop

```
;; newloop : (α  β → β)      β  list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                  (newloop (rest alon)))]))
```

So this num must also become a β

Let’s change the contract to let newloop return a value of any type.

What else in the contract must change to β?

At long last …

```
;; newloop : (α β → β)  β  list[α] → β
(define (newloop combine base alon)
  (cond [(empty? alon) base]
        [(cons? alon)
         (combine (first alon)
                    (newloop (rest alon)))]))
```

Actually, newloop is built-in. It’s called *foldr*

The foldr loop

```
;; foldr : (α β → β)  β  list[α] → β
(define (foldr combine base alst)
   (cond [(empty? alst) base]
               [(cons? alst)
                (combine (first alst)
                              (foldr (rest alst)))]))
```

```
;; length : list[α] → num
(define (length alst)
   (foldr (lambda (elt result-rest) (+ 1 result-rest))
            0 alon))
```

Phew!

- We now have three loops at our disposal:
  - filter : ($\alpha \rightarrow$ boolean) list[α] $\rightarrow$ list[α]
    extract list of elts that satisfy a predicate
  - map : ($\alpha \rightarrow \beta$) list[α] $\rightarrow$ list[β]
    applies function to all elts, returning list of results
  - foldr : ($\alpha\ \beta \rightarrow \beta$) β list[α] $\rightarrow \beta$
    combines elts of list according to given function

Time to practice!

Recall the data defns for animal/boa/armadillo

- ;; A boa is a (make-boa symbol num symbol)
  (define-struct boa (name length food))

- ;; An armadillo is a (make-dillo symbol num bool)
  (define-struct dillo (name length dead?))

- ;; An animal is one of
  ;; - a boa
  ;; - an armadillo

Time to practice!

Write the following programs with Scheme loops

- ;; large-animals : list[animal] num → list[animal]
  ;; return list of all animals longer than given num

- ;; eats-pets-count : list[animal] → num
  ;; return number of boas in list that eat pets

- ;; kill-all-dillos : list[animal] → list[animal]
  ;; return list containing all animals in the input list
  ;;   but with all armadillos dead