higher-order procedures: `fold`

`(foldr op id lst)`

- `op` : a binary procedure
- `lst` : list of arguments
- apply `op` right-associatively to elements of `lst`
- return result of evaluation
- the identity element `id` is always used

That is:

- `(foldr op id '())` $\Rightarrow$ `id`
- `(foldr op id '(e))` $\Rightarrow$ `(op e id)`
- `(foldr op id '(e1 e2 ... en))`
  $\Rightarrow$ `(op e1 (op e2 (op ... (op en id))))`

higher-order procedures: `foldr`

Let's define our version first.

higher-order procedures: `foldr`

`(foldr + 0 '(1 2 3 4))` $\Rightarrow$

higher-order procedures: `foldr`

```
(foldr list '() '(1 2 3 4)) ⇒
```

higher-order procedures: `foldr`

```
(foldr op id lst)
```

- `op` $\Leftrightarrow$ `cons`
- `id` $\Leftrightarrow$ `'()`

```
   (1 2 3 4)
=> (cons 1 (cons 2 (cons 3 (cons 4 '()))))

   ( +   1 ( +   2 ( +   3 ( +   4 0 ))))
=> (foldr + 0 '(1 2 3 4)

   (list 1 (list 2 (list 3 (list 4 '()))))
=> (foldr list '() '(1 2 3 4)
```

higher-order procedures: `foldl`

`(foldl op id lst)`

- `op` : an binary procedure
- `lst` : list of arguments
- apply `op` left-associatively to elements of `lst`
- return result of evaluation
- the identity element `id` is always used

That is:

- `(foldl op id '())` $\Rightarrow$ `id`
- `(foldl op id '(e))` $\Rightarrow$ `(op e id)`
- `(foldl op id '(e1 e2 ... en))`
  $\Rightarrow$ `(op en (op en-1 (op ... (op e1 id))))`

higher-order procedures: `foldl`

`(foldl op id lst)`

In other implementations:

- `(foldl op id '())` $\Rightarrow$ `id`
- `(foldl op id '(e))` $\Rightarrow$ `(op id e)`
- `(foldl op id '(e1 e2 ... en))`
  $\Rightarrow$ `(op ... (op (op id e1) e2) ...) en )`

higher-order procedures: `apply`

`(apply op lst)`

- `op` : an *n*-ary procedure
- `lst` : list of *n* arguments to `op`
- apply `op` to elements of `lst`
- return result of evaluation

higher-order procedures: apply

`(apply op lst)`

- `(apply + '(1 2 3 4))`
  $\Rightarrow$

- `(apply cons '(a (b c)))`
  $\Rightarrow$

- `(apply list '(1 2 3 4))`
  $\Rightarrow$

- `(apply cons '(a (b c) (d)))`
  $\Rightarrow$

higher-order procedures: `foldr`

`(foldr op id lst)`

- `op` $\Leftrightarrow$ `cons`
- `id` $\Leftrightarrow$ `'()`

```
(define (append list1 list2)
```

```
(define (map p lst)
```

higher-order procedures: `foldr`

```
   (append '(1 2 3 4) '(5 6 7))
=>
```

parameter lists

- bind a formal parameter to a list of actual parameters

```
(define list-args
  (lambda varparam
    varparam))

(list-args) ==>

(list-args 'a) ==>

(list-args 'a 'b 'c 'd) ==>
```

parameter lists

- bind a formal parameter to a list of actual parameters

```
(define rev-args
  (lambda varparam
    (reverse varparam)))

(rev-args ) ==>

(rev-args 'a 'b 'c) ==>

(rev-args 5 4 3 2 1) ==>
```

parameter lists

- bind a formal parameter to a list of actual parameters

```
(define sum-non1-args
  (lambda (fst . varparam)
    (apply + varparam)))

(sum-non1-args 1) ==>

(sum-non1-args 1 2) ==>

(sum-non1-args 1 2 3) ==>

(sum-non1-args ) ==>
```

parameter lists

- bind a formal parameter to a list of actual parameters

```
(define sum-non12-args
  (lambda (fst sec . varparam)
    (apply + varparam)))

(sum-non12-args 1 2) ==>

(sum-non12-args 1 2 3 4) ==>

(sum-non12-args 1) ==>
```