

Laboratory: More Higher-Order Procedures

Exercise 1: Insert

`insert` is a procedure which takes two parameters, a binary procedure and a list, and gives the result of applying the procedure to neighboring values. There are two ways to think about `insert` based on the way we interpret

```
(insert proc (list v0
                v1
                v2
                ...
                vn-1
                vn))
```

i. We could interpret the call to `insert` as

```
(proc v0
  (proc v1
    (proc v2
      ...
      (proc vn-1
        vn ) ...))))
```

ii. We could interpret the call to `insert` as

```
(proc
  (proc
    ...
    (proc
      (proc
        (proc v0
          v1 )
        v2 )
      v3 )
    ...
    vn-1 )
  vn )
```

That is, we can apply the operation in a rightmost manner (i) or in a leftmost manner (ii). For addition, the difference is between grouping like this

$$(v_0 + (v_1 + (v_2 + \dots + (v_{n-1} + v_n) \dots)))$$

or like this

$((\dots ((v_0 + v_1) + v_2) + \dots + v_{n-1}) + v_n)$

- Does it make a difference which way we do things? (Hint, consider subtraction as a binary operation.)
- Implement the rightmost version of `insert`. You should find that this works like the standard version of `sum`.
- Implement the leftmost version of `insert`. You should find that this works more like the version of `sum` we did in class.

Exercise 2: Making Lists

a. Define and test a `generate-list` procedure that takes two arguments: (1) a one-argument procedure, `proc`, that can be applied to a natural number and (2) `n`, a natural number. Your procedure should generate a list of length `n` whose *i*th element is the result of applying `proc` to *i*. For example,

```
> (generate-list (lambda (x) (* x x)) 6)
(0 1 4 9 16 25)
```

b. Define and test a `generate-lister` procedure that takes one argument -- a one-argument procedure, `proc`, that can be applied to a natural number -- and generates a new procedure that takes one parameter, a natural number, `n`, and returns a list of length `n` whose *i*th element is the result of applying `proc` to *i*.

Exercise 3: Right section

In the reading, you saw how we might define `left-section`, which fills in the left of the two arguments of a binary procedure. Define and test the analogous higher-order procedure `right-section`, which takes a procedure of two arguments and a value to drop in as its *second* argument, and returns the operator section that expects the *first* argument. (For instance, `(right-section expt 3)` is a procedure that computes the cube of any number it is given.)

Exercise 4: Powers of Two

Using the `generate-lister` procedure from you defined in a previous lab and an appropriate operator section, define a procedure `powers-of-two` that constructs and returns a list of powers of two, in ascending order, given the length of the desired list:

```
> (powers-of-two 7)
(1 2 4 8 16 32 64)
```

Exercise 5: Removing Elements

Here is an interesting list of natural numbers:

```
(define republican-voter-ids (list 1471 4270))
```

Define a Scheme procedure `remove-republicans` that takes a list of non-empty lists as its argument and filters out of it the lists in which the first element is also an element of `republican-voter-ids`.

Exercise 6: Intersection

Define `intersection` (given two lists with no duplicates, return the list containing only elements that appear in both lists) using `remove`, `complement`, `member`, and `right-section`.

Exercise 7: Filtering

The filters constructed by `remove` are designed to *exclude* list elements that satisfy a given predicate. Define a higher-order procedure `make-filter` that returns a filtering procedure that *retains* the elements that satisfy a given predicate (excluding those that *fail* to satisfy it). For instance, applying the filter (`make-filter even?`) to a list of integers should return a list consisting of just the even elements of the given list.