Procedures which take procedures as arguments:
Assume the following definition of sum

\[
\text{(define sum (lambda (term-proc a next-proc b)
  (if (> a b)
    0
    (+ (term-proc a) (sum term-proc (next-proc a) next-proc b))))}
\]

Work out what happens when we evaluate

\[
\text{(sum (lambda (x) (x)) 1 (lambda (x) (+ x 1)) 10)}
\]

ANSWER: sums the integers between 1 and 10 inclusive

How do we compute the sum for \( i = 1 \) to 10 of \( 1/i \)?

ANSWER:

\[
\text{(sum (lambda (x) (/ 1 x)) 1 (lambda (x) (+ x 1)) 10)}
\]

How do we compute the sum for \( i = \{1, 3, 5, 7, 9\} \) of \( (i^3 + 2i) \)?

ANSWER:

\[
\text{(sum (lambda (x) (+ (* x x x) (* 2 x))) 1 (lambda (x) (+ x 2)) 9)}
\]

Now consider another definition of sum:

\[
\text{(define sum (lambda (term-proc a next-proc condition)
  (if (condition a)
    0
    (+ (term-proc a) (sum term-proc (next-proc a) next-proc condition))))}
\]

How do we sum the integers between 1 and 10 inclusive using this definition?

ANSWER: (sum (lambda (x) (x))

\[
\begin{align*}
1 \\
\text{(lambda (x) (+ x 1))} \\
\text{(lambda (x) (> x 10))}
\end{align*}
\]

Procedures which return procedures:
write a procedure (sum-func f g) which takes two procedures \( f(x) \) and \( g(x) \) and returns a new procedure which is \( f(x)+g(x) \)

ANSWER:

\[
\text{(define sum-func (lambda (f g)
  (lambda (x) (+ (f x) (g x))))}
\]

What is the type of this procedure?
ANSWER:
\[(A\to\text{number}), (A\to\text{number}) \to (A\to\text{number})\]

========================================================================
Procedures which return procedures:
write a procedure \(\text{max-func} \ f \ g\) which takes two procedures \(f(x)\) and \(g(x)\)
and returns a new procedure \(h\) which is \(h(x) = \max\{f(x), g(x)\}\)

ANSWER:
\[
\text{max-func} = (\lambda (f \ g) \ (\lambda (x) \ (\text{if} \ (> \ (f \ x) \ (g \ x)) \ (f \ x) \ (g \ x))))
\]

BETTER ANSWER:
\[
\text{max-func} = (\lambda (f \ g) \ (\lambda (x) \ (\text{let} \ (a \ (f \ x)) \ (b \ (g \ x)) \ (\text{if} \ (> \ a \ b) \ a \ b))))
\]

========================================================================
Procedures which return procedures:
Assume we have defined ‘‘\text{compose}’’ as follows

\[
\text{compose} = (\lambda (f \ g) \ (\lambda (x) \ (f \ (g \ x))))
\]

Now define a new procedure \(\text{compose-n}\) which composes a function with itself \(n\) times

ANSWER:
\[
\text{compose-n} = (\lambda (f \ n) \ (\text{if} \ (= \ n \ 1) \ f \ (\text{compose} \ f \ (\text{compose-n} \ f \ (- \ n \ 1)))))
\]

OR (without using compose):

\[
\text{compose-n} = (\lambda (f \ n) \ (\text{if} \ (= \ n \ 1) \ f \ (\lambda (x) \ (f \ ((\text{compose-n} \ f \ (- \ n \ 1)) \ x)))))
\]

========================================================================
Abstracting common patterns over lists:
Assume the following definitions for ‘‘\text{map}, ‘‘\text{accumulate}’’ and ‘‘\text{filter}’’

\[
\text{map} \ proc \ seq = (\text{if} \ (\text{null?} \ seq)
\]

2
(define (accumulate op init seq)
  (if (null? seq)
      init
      (op (car seq) (accumulate op init (cdr seq)))))

(define (filter pred seq)
  (if (null? seq)
      nil
      (let ((rest (filter pred (cdr seq))))
        (if (pred (car seq))
            (cons (car seq) rest)
            rest)))))

What are the values for the following expressions?:

(map (lambda (x) (- x 5)) (list 1 2 3 4))
ANSWER: (-4 -3 -2 -1)

(accumulate * 1 (list 1 2 3 4))
ANSWER: 24

(filter (lambda (x) (< x 3)) (list 1 2 3 4))
ANSWER: (1 2)

Now write functions using map, filter or accumulate which:

1) calculates the sum of elements in a list, e.g. (list 1 2 3 4) => 10
ANSWER: (define sum (lambda (x) (accumulate + 0 x)))

2) calculates the length of a list
ANSWER: (define length (lambda (x) (accumulate (lambda (x y) (+ 1 y)) 0 x)))

A final (tough) question: say we define ls to be a list of *procedures*:

(define (square x) (* x x))
(define (double x) (* x 2))
(define (inc x) (+ x 1))

(define ls (list square double inc))

Now say we want a function ‘‘apply-procs’’ that behaves as follows:
(apply-procs ls 4)
=> ((square 4) (double 4) (inc 4)) = (16 8 5)

(apply-procs ls 3)
=> ((square 3) (double 3) (inc 3)) = (9 6 4)

How do we achieve this using map?

ANSWER:
(define (apply-procs ls x)
  (map (lambda (f) (f x)) ls))