CS135 Tutorial 10

Higher Order Functions, Lambda

Perfect Squares

Write a function, `(perfect-squares lon)`, that consumes a list of numbers and produces a list of the perfect squares it contains (maintaining the original order).

```
(check-expect (perfect-squares (list 1 2 3 4 5 6 7 8 9 10)) (list 1 4 9))
```

Recall that a perfect square is a number, n, where $n=i^2$ for some integer i.

Restrictions:

- Use only implicit recursion (i.e. you can’t write a function that applies itself, either directly or via mutual recursion).

Perfect Squares Revisited

Write a function, `(generate-perfect-squares lo hi)`, that generates a list in ascending order of perfect squares between `lo` and `hi`, inclusive.

Restrictions:

- Use only implicit recursion (i.e. you can’t write a function that applies itself, either directly or via mutual recursion).

Simplify

We've studied arithmetic expressions several times. In M14 we represented them with quoted lists, so we could do `(eval '(+ 2 (* 3 4) (+ 5 6)))` to get 25.

In A07 we added identifiers such as `'x`, `'y`, and `'z` to our expressions, getting the values from a symbol table. Combine these ideas into a new data definition:

```
;; An Op is (anyof '+ '*)

;; An Arithmetic Expression (AExp)
;; is one of:
;; * Num
;; * Sym
;; * (cons Op (listof AExp))
```

Write `(simplify ex)` which simplifies an arithmetic expression.

Look for opportunities to use `filter`, `map`, etc. as well as `lambda`.

Simplify: Examples

```
(check-expect (simplify 1) 1)
(check-expect (simplify 'x) 'x)

;; collapse constants into a single value
(check-expect (simplify '(+ 1 2 3 4)) 10)
(check-expect (simplify '(* 1 2 3 4)) 24)
(check-expect (simplify '(+ 1 (* 2 3) 4 (* 5 6))) 41)

;; leave other parts of the expression alone
(check-expect (simplify '(+ x y z)) '(+ x y z))
(check-expect (simplify '(* x y z)) '(* x y z))

;; move constants to the front of the expression
(check-expect (simplify '(+ 1 (* x y) z (* 5 6))) '(+ 31 (* x y) z))
(check-expect (simplify '(+ 1 (* x y (+ 2 3)) z (* 5 6))) '(+ 31 (* 5 x y) z))
```

Strategy

If we develop templates from the data definition and rename for our problem:

```
(define (simplify ex)
  (cond [(number? ex) ...]
        [(symbol? ex) ...]
        [(cons? ex) (simplify/lst (first ex)
                                  (rest ex))]))

(define (simplify/lst op lox)
  (cond [(empty? lox) ...]
        [else (... (simplify (first lox))
                   (simplify/lst op (rest lox)))]))
```

Strategy

- Given an expression, start by simplifying all the subexpressions. That is,

```
'(+ 1                        '(+ 1
    (* x y (+ 2 3))              (* 5 x y)
    z                  =>        z
    (* 5 6))                     30)
```

- Pass the operator and simplified arguments to a helper function, `simplify/lst`.
- Partition the list of arguments into a list of numbers and a list of non-numbers.
- Collapse the list of numbers into one number (watch out for empty!).
- Combine operator, number, and non-numeric expressions to produce the new expression.