CS135 Tutorial 10

Higher Order Functions, Lambda

Eval Recap

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

```
;; An opnode is a (list (Num Num -> Num) (listof AExp))
;; An AExp is (anyof Num opnode)

;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
    (cond [(number? ex) ex]
          [(cons? ex) (my-apply (first ex) (rest ex))]))
```

Eval Recap Cont.

```
;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: Sym (listof AExp) → Num
(define (my-apply op args)
    (cond [(empty? args) (op )]
          [else (op (eval (first args)) (my-apply op (rest args)))]))
```

In `my-apply`, we explicitly recurse on the rest of the list to apply `eval` to each element in the list. Can we replace this with implicit recursion before we call `my-apply`?

Updated eval

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
    (cond [(number? ex) ex]
          [(cons? ex) (my-apply (first ex) (map eval (rest ex)))]))
;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: Sym (listof AExp) → Num
(define (my-apply op args)
    (cond [(empty? args) (op )]
          [else (op (first args)
                    (my-apply op (rest args)))]))
```

Now all `my-apply` does is use the operator, can we simplify it further with another higher order functions?

Updated my-apply cont.

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
    (cond [(number? ex) ex]
          [(cons? ex) (my-apply (first ex) (map eval (rest ex)))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: Sym (listof AExp) → Num
(define (my-apply op args)
    (foldr op (op ) args)]))
```

Update Eval cont.

In M14 we took `eval` a step further by using quoted lists. However, this prevented us from being able to store functions in our lists and we had to go back to symbols.
How can we adapt my apply to this change?

Update Eval cont.

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
    (cond [(number? ex) ex]
          [(cons? ex) (my-apply (first ex) (map eval (rest ex)))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: Sym (listof AExp) → Num
(define (my-apply op args)
    (cond [(symbol=? op '+) (foldr + 0 args)]
          [(symbol=? op '*) (foldr * 1 args)]))
```

Simplify

In A07 we also 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 more 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))
```

Eval -> Simplify

How can we update `eval` to meet the requirements for `simplify`?

```
(define (eval ex)
    (cond [(number? ex) ex]
          [(cons? ex) (my-apply (first ex) (map eval (rest ex)))]))

(define (my-apply op args)
    (cond [(symbol=? op '+) (foldr + 0 args)]
          [(symbol=? op '*) (foldr * 1 args)]))
```

Eval-Simplify

How can we update `eval` to meet the requirements for `simplify`?

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

(define (simplify/lst op simplified-args)
    (cond [(symbol=? op '+) (foldr + 0 simplified-args)]
          [(symbol=? op '*) (foldr * 1 simplified-args)]))
```

Problems to fix

- Following our original definition, we map simplify on all the elements in our lists which simplifies our expressions significantly

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

- Our current `simplify/lst` works on a list of just numbers!

Problems to fix

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

(define (simplify/lst op simplified-args)
    (local [(define only-nums (filter number? simplified-args))]
           (cond [(symbol=? op '+) (foldr + 0 only-nums)]
                 [(symbol=? op '*) (foldr * 1 only-nums)])))
```

Problems to fix

- Now we are completely ignoring the symbols and their expressions!
- We need to also get the non-numbers!
- Combine operator, number, and non-numeric expressions to produce the new expression. Watch out for empty!

Problems to fix

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

(define (simplify/lst op simplified-args)
    (local [(define only-nums (filter number? simplified-args))
            (define non-nums (filter (lambda (x) (not (number? x)))
                                     simplified-args))]
           (cond [(symbol=? op '+) (foldr + 0 only-nums)]
                 [(symbol=? op '*) (foldr * 1 only-nums)])))
```

How do we combine them?

		Contains Numbers?	
		Yes	No

Contains Symbols?	Yes	`'(+ 5 x (* 2 3) 10)`
		`=>`
		`'(+ 20 x)`
			`'(+ x y (* 2 z))`
			`=>`
			`'(+ x y (* 2 z))`

	No	`'(+ 1 2 3 4)`
		`=>`
		`10`
			`'(+ )`
			`=>`
			`0`

How do we combine them?

```
(define (simplify/lst op simplified-args)
    (local [(define only-nums (filter number? simplified-args))
            (define non-nums (filter (lambda (x) (not (number? x)))
                                     simplified-args))
            (define simplified-num (cond [(symbol=? op '+)
                                          (foldr + 0 only-nums)]
                                         [(symbol=? op '*)
                                          (foldr * 1 only-nums)]))]
            (cond [(empty? non-nums) simplified-num]
                  [(empty? only-num) (cons op non-nums)]
                  [else (cons op (cons simplified-num non-nums))])))
```