

```

1 =====
2 =====
3 ;;
4 ;;           2017 Fall CS 145 Midterm II Review Notes V3.0
5 ;;
6 ;;           Asst. 6~8
7 ;;           Lecture/Tutorial Notes
8 ;;           Prof: Gordon Cormack
9 ;;           ISA: Ashish Mahto
10 ;;
11 ;;           By David Duan
12 ;;
13 ;;           Outline
14 ;;           Part A: Asymptotic Analysis
15 ;;           Part B: Abstract data type
16 ;;           Part C: Higher order functions
17 ;;           Part D: Other minor topics
18 ;;           Part E: Appendix
19 ;;
20 =====
21 ;; If you find any errors (I'm sure there are a lot of them), please
22 ;; email j32duan@edu.uwaterloo.ca, David Duan. Thanks in advance.
23 =====
24 ;; Release Note
25 ;;
26 ;; V0.X
27 ;;   Notes from lecture
28 ;;   Notes from tutorial
29 ;;   Ashish's messages (which are, indeed, full of wisdom xd)
30 ;;   My own research
31 ;;
32 ;; V1.X
33 ;;   Raw note
34 ;;
35 ;; V2.0 Update: (Approx 1k lines)
36 ;;   Finished part A, B, and C.
37 ;;   Still need to finish foldr and foldl.
38 ;;   I'll do that tomorrow after asking Ashish questions.
39 ;;
40 ;; V2.1 Update: (Approx 1.5k lines)
41 ;;   Added foldr, foldl, and foldr vs. foldl.
42 ;;   Added or/list, disjoint/list, and f^n
43 ;;   Changed formatting
44 ;;
45 ;; V2.2 Update: (Approx 2k lines)
46 ;;   Added Part D, including D1, D2, D3
47 ;;   Added Part E, including E1, practice questions from TUT Oct.25
48 ;;   Need to add: assignment 6~8
49 ;;
50 ;; V2.3 Update: (Approx 2k lines)
51 ;;   Modified A7
52 ;;   Modified Part E, but haven't finished yet.
53 ;;   Modified Part C, added filter and map
54 ;;   Fixed typo
55 ;;
56 ;; V3.0 Release: 2155 lines
57 ;;   Modified Part E
58 ;;   Fixed typo
59 ;;
60 =====
61 ;; Big thanks to Teresa Kang and Steven Wong for pointing out countless
62 ;; typos and logic errors.
63 =====

```

64 Part A: Asymptotic Analysis

65

66 Table of Contents

67 A1. Introduction

68 A2. Examples

69 A3. /subset and /strict-subset

70 A4. Order

71 A5. Examples

72 A6. Examples for using definitions to prove statements involving
73 Big-O notation.

74 A7. Additional remarks

75

76 -----

77 A1. Introduction.

78

79 **Defn:** (Informal)80 $f(x) \text{ /in } O(g(x)) \text{ iff } O(f(x)) \leq O(g(x)).$

81

82 **Defn:** (Formal)83 There exists some constant x_0 and c such that $f(x) < c * g(x)$
84 for every $x > x_0$.

85

86 **Defn:** (About Big-O notation)87 $O(g(n))$ is the set of functions $f(n)$ such that there exists
88 constant c and n_0 such that for all $n \geq n_0$, $f(n) \leq c * g(n)$.

89

90 **Remark:**

- 91 1. $c * g(x)$ is an upper bound for $f(x)$ when $x > x_0$.
- 92 2. $O(g(x))$ serves as a placeholder for all $f(x) \text{ /in } O(g(x))$.
- 93 3. If $O(f(x)) \leq O(g(x))$, then for all $h(x) \text{ /in } O(f(x))$,
94 $h(x) \text{ /in } O(g(x))$.

95

96 -----

97 A2. Examples.

98

99 **Ex.**

100 $x \text{ /in } O(x)$
 101 $x^2 \text{ /in } O(x^2)$
 102 $f(x) \text{ /in } O(f(x))$

103

104 **Ex.**

105 $O(f(x)) + k = O(f(x))$
 106 $O(f(x)) * k = O(f(x))$

107

108 **Ex.**

109 $O(k) = O(1)$
 110 $O(\log_n(x)) = O(\log(x))$
 111 $O(x) + O(x^2) + \dots + O(x^n) = O(x^n)$

112

113 -----

114 A3. /subset and /strict-subset

115

116 **Thm:**

117 $O(x) \text{ /subset } O(x)$.
 118 $O(x) \text{ /subset } O(x^m)$ for any $m \geq 1$.

119

120 **Defn:**

121 $O(f(x)) \text{ /strict-subset } O(g(x))$ if
 122 a. $O(f(x)) \text{ /subset } O(g(x))$, and
 123 b. $O(g(x)) \text{ /not /subset } O(f(x))$

124

125 **Thm:**126 $O(a^x) \text{ /strict-subset } O((a+e)^x)$

127 $O(x^n)$ /strict-subset $O(a^x)$ given that $a > 1$

128

129 -----

130 A4. Order

131

132 $O(1) < O(\log n)$
 133 $< O(n^k)$
 134 $< O(n)$
 135 $< O(n \log n)$
 136 $< O(n^p), p > 1$
 137 $< O(a^n), a > 1$
 138 $< O(b^n), b > a$
 139 $< O(n!)$

140

141 -----

142 A5. Examples

143

144 **Ex.**

145

```
146 |%Racket_file
147 |
148 | (define (dedupe lst)
149 |   (cond
150 |     [(empty? lst) empty]
151 |     [(member (first lst) (rest lst))
152 |      (dedupe (rest lst))]
153 |     [else
154 |      (cons (first lst) (dedupe (rest lst)))]))
155 |
156 |%END
```

157

158 **Note:** The function `member` takes $O(n)$, and we apply it to every
 159 element in the list, so overall the running time is $O(n^2)$.

160

161 **Ex.**

162

```
163 |%Racket_file
164 |
165 | (define (append l1 l2)
166 |   (cond
167 |     [(empty? l1) l2]
168 |     [else (cons (first l1) (append (rest l1) l2))]))
169 |
170 |%END
```

171

172 **Note:** The function `cons` takes $O(1)$, and we apply it to every
 173 element in list1, so overall the running time is $O(n)$
 174 where n is the size of list1.

175

176 **Ex.**

177

```
178 |%Racket_file
179 |
180 | (define (reverse l)
181 |   (cond
182 |     [(empty? l) empty]
183 |     [else (append (reverse rest l)
184 |                    (list first l))]))
185 |
186 |%END
```

187

188 **Note:** This is a bad list recursion. We apply `append` and
 189 `reverse` which both takes $O(n)$ time on each element,

190 so overall it will be $O(n^2)$.

191

192 -----

193 A6. Examples for using definitions to prove statements involving
194 Big-O notation.

195

196 **Ex.** Prove " $3n^2 + 6n$ is $O(n^2)$ " using definition 1.

197

198 We need to start with "there exists c and n_0 ", but we don't
199 know what value would make the rest of the statement true,
200 so we leave them as symbolic constants and accumulate
201 information about it, as long as we make sure to specify
202 its value by the end of the proof.

203

204 **Remark:**

205 This part is very similar to our proof in math 147 where
206 we assume ϵ and δ/N exists, then do calculations
207 until we find an appropriate pair.

208

209 The next step is doing algebra. Assume our c works, we want
210 to find an appropriate n_0 .

211

212 **Remark:**

213 This is similar to assuming our N/δ works and work
214 towards ϵ .

215

216 |

$$\begin{aligned} 217 \quad 3n^2 + 6n &\leq c * n^2 \\ 218 \quad \quad 6n &\leq (c-3) * n^2 \\ 219 \quad \quad \quad 6 &\leq (c-3) * n \\ 220 \quad \quad 6 / (c-3) &\leq n \end{aligned}$$

221 |

222

223 At this point, we can let c be an arbitrary number that
224 is greater than 3, so the left side would be positive.
225 By the Archimedean principle, the set of natural numbers
226 is not bounded, so we can always find some natural number
227 n_0 such that $n \geq n_0$ implies $6 / (c-3) \leq n$.

228

229 Since the last inequality satisfies the requirement of
230 " $3n^2 + 6n \leq c * n^2$ for all $n > n_0$ for some c in real number
231 and n in natural number", we can conclude that $3n^2 + 6n$ is
232 indeed $O(n^2)$.

233

234 **Remark:**

235 Our proof works only because everything we did was
236 reversible (we performed the same operations on both sides).
237 This is not always the case with inequalities. For example,
238 if we use the fact that we can increase the larger side of an
239 inequality, we would not be able to work backward.

240

241

242 **Ex.** Prove that " $3n^2 - 6n$ is not $O(n)$."

243

244 Express this using definition, we are saying "not" there exists
245 c and n_0 such that for all $n > n_0$, $3n^2 - 6n \leq cn$. This is equivalent
246 of saying, there exists some $n > n_0$ that does not make our
247 statement true.

248

249 Let c in \mathbb{R} . We want to prove "there exists n such that for any c ,
250 $3n^2 - 6n$ is greater than cn ."

251

252 |

```

253 | 3n^2 - 6n > cn
254 |     3n^2 > (c+6) * n
255 |     n > (c+6) / 3
256 |

```

257
 258 There are two restrictions for n . First we have $n > n_0$, and secondly
 259 we have $n > (c+6)/3$, so we can let $n > \max\{n_0, (c+6)/3\}$. This way, we
 260 have found an appropriate counter example of n that supports our
 261 statement.

262 **!IMPORTANT!** EXAM PREPARATION

```
263 """
```

264 At least one of the problem will be in the following form:

265
 266 For each of a number of pairs of definitions for $f(x)$ and $g(x)$,
 267 you must submit a module to do the following:

268
 269 -> Provide a function (findx c x0) that consumes positive values c
 270 and x_0 and produces $x \geq x_0$ such that $f(x) > c * g(x)$. If no such
 271 value exists, produce 'impossible.'

272
 273 -> Provide values c and x_0 for which (findx c x0) produces 'impossible.'
 274 If there is no such pair of values, define c and x_0 both to have
 275 the value 'none.'

276
 277 If your implementation of findx is correct, you will be able to
 278 find suitable values of c and x_0 iff $f(x)$ is $O(g(x))$. In either case,
 279 you should justify your answer using embedded comments.

```
280 """
```

281 **Note:**

282 To prove that $f(x)$ is $O(g(x))$, assume c exists, find appropriate n ,
 283 and rewrite the proof in the normal order. This is similar to limits.

284
 285 To prove that $f(x)$ is **not** $O(g(x))$, try to prove the negation of
 286 it. That is, find an example of n such that for any c , $f(x) > O(g(x))$.

287 ----- 288 A7. Additional remarks

289 **Remark:**

290 We sometimes use the equal sign to express the relationship,
 291 as in " $3n^2 + 6n = O(n^2)$ ". But $O(n^2)$ is **not** a function nor
 292 an algebraic object, so this equal sign does **not** share any
 293 properties such as reflexivity as the normal mathematical
 294 equality sign.

295 **Remark:**

296 Running time of common Racket functions

297 **Note:**

298 $O(1)$: cons, first, rest, list, make-foo, foo-x, foo?, eq?

299 **Note:**

300 $O(n)$, where $n = (\text{size } x)$:
 301 (append x y) ;; independent of size of y
 302 (length x)
 303 (member e x)
 304 (reverse x)

305 **Note:**

```

316 O(n*T(n)), where n = (size x) and f has O(T(n)) time
317   (foldr f e x)
318   (foldl f e x) ;; requires O(n) space
319   (map f x)

```

Note:

```

322 sort: O(n log n)
323 quicksort: on average O(n log n), worst O(n^2)
324 (equal? x y): O(n), where n = min((size x), (size y))

```

```

326 =====
327 =====

```

Part B: Abstract Data Type

Defn:

```

331 ADT: A set of values and a finite set of operations
332   (functions), defined entirely by the behavior of the
333   operators.

```

Remark:

```

336 Contrast to: Concrete Data Types
337   - Defined by its representations.
338   - Has infinite number of operations.
339   - For example, list of number '(10, 20, 30)

```

```

342 Ex. Abstract implementation of set.

```

```

344 Remark: Big thanks to Teresa for pointing out a crucial mistake.

```

Note:

```

347 We represent the set as a function.

```

```

349 |%Racket_file
350 |
351 | (provide make-empty-set insert-set member-set)
352 |
353 | (define (make-empty-set) (lambda (e) false))
354 |
355 | (define (member-set e s) (s e))
356 |
357 | (define (insert-set e s)
358 |   (lambda (x)
359 |     (if (= e x) true (s x))))
360 |
361 | ;; You need to define what equality means.
362 |
363 |%END

```

Note:

- 366 1. This implementation is essentially a `member` function,
367 it tests `if` the argument passed in is in the set or not.
- 368
- 369 2. `(make-empty-set)` creates a `lambda` function which returns
370 false no matter what argument is passed in.
- 371
- 372 3. `(member-set e s)` returns the result of function application
373 s applied onto e, ie. returns `(s e)`.
- 374
- 375 4. `(insert-set e s)` returns a `lambda` function which takes in
376 one argument; its output depends on `if` the argument x is equal
377 to the argument e we passed in in the first place. The variable
378 e is what's stored in the `lambda` function. The variable x is

379 newly-consumed and will be checked if it's in the set already.

380

381 5. If it is still confusing, let's walk through some examples.

382

383

384 **Ex.** (member-set 5 (make-empty-set))

385 This function checks if 5 is in the empty set.

386

387 |%STEPPER

388

389 | (member-set 5 (make-empty-set))

390

390 | ((make-empty-set) 5)

391

391 | ((lambda (e) false) 5)

392

392 | false

393

394 |%END

395

396

397 **Ex.** (insert-set 5 (make-empty-set))

398 This function inserts 5 into an empty set.

399

400 |%STEPPER

401

402 | (insert-set 5 (make-empty-set))

403

404 | ;; local_var: e = 5, s = (make-empty-set)

405

405 | ((lambda (e s)

406

406 | (lambda (x)

407

407 | (if (= e x) true (s x))))

408

408 | 5 (make-empty-set))

409

410 | ;; (make-empty-set) = (lambda (e) false)

411

411 | (lambda (x) (if (= 5 x) true ((lambda (e) false) x)))

412

413 |%END

414

415 **Note:** Instead of returning a value, we got a lambda function.

416 This function consumes an additional variable x, then

417 compares if it equals 5. If yes, the function returns true.

418 Otherwise our variable x gets passed into the next layer

419 of lambda function, which in this case is the empty

420 set function that always returns false.

421

422

423 **Ex.** (insert-set 3 (insert-set 5 (make-empty-set)))

424 This function inserts 3 into a set containing 5.

425

426 |%STEPPER

427

427 | (insert-set 3 (insert-set 5 (make-empty-set)))

428

429 | ;; local_var: e = 3, s = (insert-set 5 (make-empty-set))

430

430 | ((lambda (e s)

431

431 | (lambda (x)

432

432 | (if (= e x) true (s x))))

433

433 | 3 (insert-set 5 (make-empty-set)))

434

435 | ;; substitution for (insert-set 5 (make-empty-set))

436

436 | ((lambda (e s)

437

437 | (lambda (x)

438

438 | (if (= e x) true (s x))))

439

439 | 3

440

440 | (lambda (e s)

441

441 | (lambda (x)

```

442     (if (= e x) true (s x)))
443     5 (make-empty-set))
444
445 ;; substitution for (make-empty-set)
446 ((lambda (e s)
447   (lambda (x)
448     (if (= e x) true (s x))))
449   3
450   (lambda (e s)
451     (lambda (x)
452       (if (= e x) true (s x))))
453     5
454     (lambda (e) false))
455
456 ;; Now 5 gets consumed to produce a new function
457 ((lambda (e s)
458   (lambda (x)
459     (if (= e x) true (s x))))
460   3
461   (lambda (x)
462     (if (= 5 x) true ((lambda (e) false) x))))
463
464 ;; Now 3 gets consumes to produce a new function
465 (lambda (x)
466   (if (= 3 x)
467     true
468     (lambda (x)
469       (if (= 5 x) true ((lambda (e) false) x)) x))
470
471 %END

```

Note:

Take a look at what our output function does. This `lambda` function (call it `lambda1`) takes in 1 argument `x` and compares `x` with 3. If they are equal, we return `true`. Otherwise we call the next `lambda` function (call it `lambda2`), which takes in the same `x` and compare it with 5. If `x = 5` then return `true`, otherwise call the next `lambda` function, in our case it's the empty set/function so it always returns `false`.

Note that every time the `if` statement fails, we are calling `(s x)`. The `s` is the `lambda` function, and we are feeding it with an argument `x`, which is the variable right after the close bracket of `lambda` function.

Ex. `(insert-set 5 (insert-set 3 (insert-set 5 (make-empty-set))))`

This function (tries to) insert the number 5 into a set containing number 3 and 5.

```

493 %STEPPER
494
495 (insert-set 5 (insert-set 3 (insert-set 5 (make-empty-set))))
496
497 ;; local_var: e = 5, s = (insert-set 3 (insert-set 5
498 ;; (make-empty-set)))
499 ((lambda (e s)
500   (lambda (x)
501     (if (= e x) true (s x))))
502   5 (insert-set 3 (insert-set 5 (make-empty-set))))
503
504 ;; substitution for (insert-set 3 (insert-set 5 (make-empty-set)))

```

```

505 ((lambda (e s)
506   (lambda (x)
507     (if (= e x) true (s x))))
508   5
509   ((lambda (e s)
510     (lambda (x)
511       (if (= e x) true (s x))))
512     3
513     (insert-set 5 (make-empty-set))))
514
515 ;; substitution for both insert-set 5 and the empty set.
516
517 ((lambda (e s)
518   (lambda (x)
519     (if (= e x) true (s x))))
520   5
521   ((lambda (e s)
522     (lambda (x)
523       (if (= e x) true (s x))))
524     3
525     ((lambda (e s)
526       (lambda (x)
527         (if (= e x) true (s x))))
528       5
529       (lambda (e) false))))
530
531 ;; now consumes 5 to produce a new function
532
533 ((lambda (e s)
534   (lambda (x)
535     (if (= e x) true (s x))))
536   5
537   ((lambda (e s)
538     (lambda (x)
539       (if (= e x) true (s x))))
540     3
541     (lambda (x)
542       (if (= 5 x) true
543         ((lambda (e) false) x))))
544
545 ;; consumes 3
546
547 ((lambda (e s)
548   (lambda (x)
549     (if (= e x) true (s x))))
550   5
551   (lambda (x)
552     (if (= 3 x) true
553       ((lambda (x)
554         (if (= 5 x) true
555           ((lambda (e) false) x))) x))))
556
557 ;; consumes 5
558
559 (lambda (x)
560   (if (= 5 x) true
561     ((lambda (x)
562       (if (= 3 x) true
563         ((lambda (x)
564           (if (= 5 x) true
565             ((lambda (e) false) x))) x))) x)))
566
567 %END

```

568
 569 **Note:** This `lambda` has similar logic as the last one, except
 570 it actually contains two functions checking for 5.
 571 Nevertheless it doesn't affect the set operations since
 572 if the argument is 5, we will directly return true.
 573

574
 575 **Note:** To further see that this implementation is like a
 576 `member` function, try `((insert 5 (make-empty-set)) 5)` and see
 577 what would happen.
 578

579 =====
 580 =====

581 Part C: Higher-order function

582
 583 Table of Contents
 584 C1. `compose`
 585 C2. `disjoin`
 586 C3. `foldr`
 587 C4. `foldl`
 588 C5. `foldr` vs. `foldl`
 589 C6. `or/list`
 590 C7. `disjoin/list`
 591 C8: `f^n`
 592 C9. `currying`
 593 C10. `map`
 594 C11. `filter`

595
 596 -----
 597 C1. `compose`

598
 599 **Note:**
 600 Consider the contract `(f : (b : B) -> C)`.
 601 This is the type contract for a generic one-argument function.
 602 It takes an argument `b` of type `B` and returns an output of type `C`.
 603

604 Now suppose we have another function `g` with type contract
 605 `(g : (a : A) -> B)`. We can `define` a new function called
 606 `compose`, which represents the function composition of
 607 function `g` and `f`:
 608

609 **Implementation:**

610
 611 `;; Type contract:`
 612 `;; (compose : (f : B -> C) -> (g : A -> B) => (A -> C))`
 613

```
614 |%Racket_file
615 |
616 | (define compose
617 |   (lambda (f g)
618 |     (lambda (x)
619 |       (f (g x))))))
620 |
621 |%END
```

622
 623 **Remark:**

624 - Arguments:
 625 `f : B -> C`
 626 `g : A -> B`
 627 - Return:
 628 A `lambda` function
 629 - Arguments:
 630 `x : A`

```
631     - Return:
632       (f (g x)) : C
```

Ex.

```
636 |%Racket_file
637 |
638 | > (define neg-sqrt (compose - sqrt))
639 | > neg-sqrt
640 | #<procedure:compose>
641 |
642 | > (neg-sqrt 3)
643 | -1.732.. ;; first (sqrt 3), then (- (sqrt 3))
644 |
645 | > ((compose sqrt -) 3)
646 | 0+1.732..i ;; first (- 3), then (sqrt (- 3))
647 |
648 |%END
```

Note:

Compare **and** contrast the following function with the `compose` defined above.

Implementation:

```
656 ;; Type contract:
657 ;; (compose-then-compute : (f : B -> C) -> (g : A -> B) -> (a : A) => C)
```

```
659 |%Racket_file
660 |
661 | (define compose-then-compute
662 |   (lambda (f g a)
663 |     (f (g a))))
664 |
665 |%END
```

Remark:

```
668 - Arguments:
669   f : B -> C
670   g : A -> B
671   a : A
672 - Return:
673   (f (g a)) : C
```

Remark:

The difference is that `compose` returns a function, where `compose-then-compute` returns a value.

679 -----

C2. `disjoin`**Note:**

The function `disjoin` consumes two functions that returns booleans **and** returns a function that returns the disjunction (or-value).

Implementation:

```
689 |%Racket_file
690 |
691 | (define disjoin
692 |   (lambda (f g)
693 |     (lambda (x)
```

```

694 |         (or (f x) (g x))))))
695 |
696 |%END

```

Remark:

- Arguments:
 - f : A -> Bool
 - g : A -> Bool
- Return:
 - A function
 - Argument:
 - x : A
 - Return:
 - (or (f x) (g x)) : Bool

Ex.

Check if the list has length less than 2?

First thought:

```

713 |         (or (empty? lst)
714 |             (empty? (rest lst)))

```

Note that the second argument inside the `or` function is applying two 1-arg functions onto the same argument, thus we can rewrite it using `compose`:

```

722 |         (empty? (rest lst)
723 |             ;; is equivalent to
724 |             ((compose empty? rest) lst)

```

Also, since we are applying two functions onto the same argument (`((compose empty? rest) produces a function!`), we can rewrite the whole thing using `compose`:

```

731 |         (or (empty? lst)
732 |             (compose empty? rest) lst)
733 |         ;; is equivalent to
734 |         ((disjoin empty? (compose empty? rest)) lst)

```

This way, we can `define` our new function:

Implementation:

```

740 |%Racket_file
741 |
742 | (define len<2?
743 |   (disjoin empty? (compose empty? rest)))
744 |
745 | (len<2? '(1 2)) => False
746 |
747 |%END

```

C3: foldr

Thm:

Main characteristic of `foldr`: PURE RECURSION

Implementation:

756

```

757 |%Racket_file
758 |
759 | (define (foldr f z ls)
760 |   (cond
761 |     [(empty? ls) z]
762 |     [else (f (first ls) (foldr f z (rest ls)))]))
763 |
764 |%END

```

Ex.

```

768 |%Racket_file
769 |
770 | (foldr cons '() '(1 2 3))
771 |
772 | (foldr + 0 '(1 2 3))
773 |
774 |%END

776 |%STEPPER
777 |
778 | (foldr cons '() '(1 2 3))
779 | (cons 1 (foldr cons '() '(2 3)))
780 | (cons 1 (cons 2 (foldr cons '() '(3))))
781 | (cons 1 (cons 2 (cons 3 (foldr cons '() '()))))
782 | (cons 1 (cons 2 (cons 3 '())))
783 | (cons 1 (cons 2 '(3)))
784 | (cons 1 '(2 3))
785 | '(1 2 3)

787 | (foldr + 0 '(1 2 3))
788 | (+ 1 (foldr + 0 '(2 3)))
789 | (+ 1 (+ 2 (foldr + 0 '(3))))
790 | (+ 1 (+ 2 (+ 3 (foldr + 0 '()))))
791 | (+ 1 (+ 2 (+ 3 0)))
792 | (+ 1 (+ 2 3))
793 | (+ 1 5)
794 | 6
795 |
796 |%END

```

Remark:

As you can see, in each step, the second argument for function `f` is the recursive function application. The calculation starts when the function reaches the end of the `list` (which returns the init value `z`), and fold from right towards left.

Thm:

Because of this, we can rewrite `foldr` like this:

```

808 |%Racket_file
809 |
810 | (foldr f z '(e1 e2 e3 e4 ... eN))
811 |
812 |%END

```

Step1: Expand the `foldr`

```

816 |%STEPPER
817 |
818 | (foldr f z '(e1 e2 e3 e4 ... eN))
819 | (f e1 (foldr f z '(e2 e3 e4 ... eN)))

```

```

820 | (f e1 (f e2 (foldr f z '(e3 e4 ... eN))))
821 | (f e1 (f e2 (f e3 (foldr f z '(e4 ... eN))))))
822 | (f e1 (f e2 (f e3 (f e4 (foldr f z '(... eN))))))
823 | ...
824 | (f e1 (f e2 (f e3 (f e4 (... (f eN (foldr f z '()))))))))
825 |
826 | %END

```

Remark:

Note that `(foldr f z '())` returns `z`, so we can start doing calculations now.

Step2: Evaluate expressions.

Let r_k be the result when applying f onto $e_k, r^{(k+1)}$. Note that when $k = N, e^{(k+1)} = z$.

```

837 | %STEPPER
838 |
839 | (f e1 (f e2 (f e3 (f e4 (... (f eN (foldr f z '()))))))))
840 | (f e1 (f e2 (f e3 (f e4 (... (f eN z))))))
841 | ...
842 | (f e1 (f e2 (f e3 (f e4 r5))))
843 | (f e1 (f e2 (f e3 r4)))
844 | (f e1 (f e2 r3))
845 | (f e1 r2)
846 | r1
847 |
848 | %END

```

C4: foldl

Thm:

Main characteristic of `foldl`: ACCUMULATIVE RECURSION

Implementation:

```

858 | %Racket_file
859 |
860 | (define (foldl f z ls)
861 |   (define (calc ls acc)
862 |     (cond
863 |       [(empty? ls) acc]
864 |       [else (calc (rest ls)
865 |                   (f (first ls) acc))]))
866 |   (calc ls z))
867 |
868 | %END

```

Ex.

```

872 | %Racket_file
873 |
874 | (foldl cons '() '(1 2 3))
875 |
876 | (foldl + 0 '(1 2 3))
877 |
878 | %END
879 |
880 | %STEPPER
881 |
882 | (foldl cons '() '(1 2 3))

```

```

883 | (calc '(1 2 3) '())
884 | (calc '(2 3) (cons 1 '()))
885 | (calc '(3) (cons 2 (cons 1 '())))
886 | (calc '() (cons 3 (cons 2 (cons 1 '()))))
887 | (cons 3 (cons 2 (cons 1 '())))
888 | (cons 3 (cons 2 '(1)))
889 | (cons 3 '(2 1))
890 | '(3 2 1)
891 |
892 | (foldl + 0 '(1 2 3))
893 | (calc '(1 2 3) 0)
894 | (calc '(2 3) (+ 1 0))
895 | (calc '(3) (+ 2 (+ 1 0)))
896 | (calc '() (+ 3 (+ 2 (+ 1 0))))
897 | (+ 3 (+ 2 (+ 1 0)))
898 | (+ 3 (+ 2 1))
899 | (+ 3 3)
900 | 6
901 |
902 | %END

```

Remark:

In foldl, the second argument for each function application of f is the accumulative result.

Thm:

We can rewrite foldl like this:

```

911 | %Racket_file
912 |
913 | (foldl f z '(e1 e2 e3 e4 ... eN))
914 |
915 | %END

```

Step1: Expand the foldl / Use the helper

```

919 | %STEPPER
920 |
921 | (foldl f z '(e1 e2 e3 e4 ... eN))
922 | (calc '(e1 e2 e3 e4 ... eN) z)
923 | (calc '(e2 e3 e4 ... eN) (f e1 z))
924 | (calc '(e3 e4 ... eN) (f e2 (f e1 z)))
925 | (calc '(e4 ... eN) (f e3 (f e2 (f e1 z))))
926 | (calc '(... eN) (f e4 (f e3 (f e2 (f e1 z))))))
927 | (calc '() (f eN (... (f e4 (f e3 (f e2 (f e1 z)))))))
928 |
929 | %END

```

Remark:

Note that (calc '() <acc>) returns <acc>, so we can start doing calculations now.

Step2: Evaluate expressions.

Let r_k be the result when applying f onto e_k , $r(k-1)$.
 Note that when $k = 1$, $e(k-1) = e_0 = z$.

```

940 | %STEPPER
941 |
942 | (f eN (... (f e4 (f e3 (f e2 (f e1 z))))))
943 | (f eN (... (f e4 (f e3 (f e2 r1))))))
944 | (f eN (... (f e4 (f e3 r2))))
945 | (f eN (... (f e4 r3)))

```

```

946 | (f eN r(N-1))
947 | rN
948 |
949 |%END

```

```

950
951 -----

```

952 C5: foldr vs. foldl

```

953
954

```

Thm: Equivalence

For most simple pure-functional functions, `(foldr f z ls)` is equivalent to `(foldl f z (reverse ls))`.

```

957
958

```

Ex.

```

959
960

```

```
|%Racket_file
```

```

962
963

```

```
| (foldr f z '(e1 e2 e3 e4 e5))
| (foldl f z '(e5 e4 e3 e2 e1))
```

```

965
966

```

```
|%END
```

```

967
968

```

```
|%STEPPER
```

```

969
970

```

```
| (foldr f z '(e1 e2 e3 e4 e5))
| (f e1 (foldr f z '(e2 e3 e4 e5)))
| (f e1 (f e2 (foldr f z '(e3 e4 e5))))
| (f e1 (f e2 (f e3 (foldr f z '(e4 e5))))))
| (f e1 (f e2 (f e3 (f e4 (foldr f z '(e5))))))
| (f e1 (f e2 (f e3 (f e4 (f e5 (foldr f z '()))))))
| (f e1 (f e2 (f e3 (f e4 (f e5 z))))))
```

```

977
978

```

```
| (foldl f z '(e5 e4 e3 e2 e1))
| (calc '(e5 e4 e3 e2 e1) z)
| (calc '(e4 e3 e2 e1) (f e5 z))
| (calc '(e3 e2 e1) (f e4 (f e5 z)))
| (calc '(e2 e1) (f e3 (f e4 (f e5 z))))
| (calc '(e1) (f e2 (f e3 (f e4 (f e5 z))))))
| (calc '() (f e1 (f e2 (f e3 (f e4 (f e5 z))))))
| (f e1 (f e2 (f e3 (f e4 (f e5 z))))))
```

```

986
987

```

```
|%END
```

```

988
989

```

Remark:

They produce the same outcome!!

```

991
992

```

Thm: Memory cost

In terms of memory costs, `foldl <= foldr`.

The main reason is that we use an accumulator for `foldl`, where in `foldr` we need memory to store everything.

```

995
996

```

```

997 -----
998

```

```

999
1000

```

Note:

Suppose we want to `define` a function which takes the "or" value of all elements in a list.

```

1003
1004

```

Implementation:

```

1005
1006

```

```
;; Type contract:
;; (or/list (listof Bool) -> Bool)
;;
```

```

1007
1008

```

```

1009 ;; Example:
1010 ;; (or/list '(true false false)) -> true
1011
1012 |%Racket_file
1013 |
1014 | (define or/list
1015 |   (lambda (bool-list)
1016 |     (foldr
1017 |       (lambda (bool acc-val)
1018 |         (or bool acc-val))
1019 |       false
1020 |       bool-list)))
1021 |
1022 |%END

```

Remark:

For main function or/list

- Argument:
 - bool-list: listof b : Bool
- Return:
 - b : Bool created by foldr

Remark:

For function foldr

- Argument:
 - A function
 - Argument:
 - bool : Bool, an element from bool-list
 - acc-val : Bool, accumulator
 - Return:
 - (or bool acc-val) : Bool
 - false : Bool

Remark:

For or function, the base case is false.

- bool-list : listof b : Bool
- Return:
 - c : Bool

C7: disjoin/list**Note:**

We want to disjoin all functions in a list.

Implementation:

```

1057 |%Racket_file
1058 |
1059 | (define disjoin/list
1060 |   (lambda (list-of-func)
1061 |     (foldr
1062 |       disjoin
1063 |       (lambda (x) false)
1064 |       list-of-func)))
1065 |
1066 |%END

```

Remark:

For main function disjoin/list

- Argument:
 - listof f : A -> Bool

1072 - Returns:
 1073 A function created by foldr

Remark:

1074
 1075 For function foldr

1076 - Arguments:

1077 `disjoin : (f : A -> Bool) (g : A -> Bool) -> (h : A -> Bool)`
 1078 `(lambda (x) false)`

Remark:

1081 Recall that the base `case` for `or` is `false`, so the initial value
 1082 of `foldr` inside the `or` is `false`. In this case, the base `case`
 1083 for `disjoin` is also `false`, but since we are working with
 1084 functions instead of Booleans, we need a function version of
 1085 "False". In a sense, `(lambda (x) false)` can be seen as the
 1086 function version of `false`.
 1087

1088 `listof f : A -> Bool`

1089 - Return:

1090 A function `h : A -> Bool`

1091

1092

1093

1094

1095

1096

1097

1098

1099

1100

1101

1102

1103

1104

1105

1106

1107

1108

1109

1110

1111

1112

1113

1114

1115

1116

1117

1118

1119

1120

1121

1122

1123

1124

1125

1126

1127

1128

1129

1130

1131

1132

1133

1134

 C8: `f^n`: repetitive application

Note:

1097 If you want to `apply` one function many times to one list, you can
 1098 `let` this `f^n` function help you.

Implementation:

```
1102 |%Racket_file
1103 |
1104 | (define f^n
1105 |   (lambda (f n)
1106 |     (foldr
1107 |       compose
1108 |       identity
1109 |       (make-list n f))))
1110 |
1111 |%END
```

Remark:

1114 For main function `f^n`

1115 - Argument:

1116 `f : A -> B`

1117 `n : Nat`

1118 - Return:

1119 A function `g : A -> B` created by `foldr`.

Remark:

1122 For `foldr`

1123 - Argument:

1124 `compose : (f : B -> C) (g : A -> B) -> (A -> C)`

1125 `identity : (f : X -> X)`

Remark:

1128 Recall that in `disjoin/list` we used `(lambda (x) false)` as the
 1129 function version of `false`. Here, we want to compose a `list` of
 1130 functions together, `and` we need a start point. This is very
 1131 much alike to `+` and `*`. For those two functions, when we fold them,
 1132 we used their identities as the initial value inside `foldr`
 1133 (`0` and `1`, respectively). Now we want to find the functional
 1134 version of `identity` -- the function `identity` returns the same

1135 thing it consumes, which is the ideal function version of
 1136 identity we are looking for.

1137
 1138 (make-list n f) : listof f : A -> B, with length n.

1139 **Ex.**

```

1140 |%Racket_file
1141 |
1142 |> (f^n rest 2)
1143 |#<procedure:compose>
1144 |
1145 |%END
1146 |
1147 |%STEPPER
1148 |
1149 |
1150 | (f^n rest 2)
1151 |
1152 | ((lambda (f n)
1153 |   (foldr
1154 |     compose
1155 |     identity
1156 |     (make-list n f))) rest 2)
1157 |
1158 | (foldr
1159 |   compose
1160 |   identity
1161 |   (make-list 2 rest))
1162 |
1163 | (foldr
1164 |   compose
1165 |   identity
1166 |   '(rest rest))
1167 |
1168 | (compose rest (foldr compose identity '(rest)))
1169 |
1170 | (compose rest (compose rest (foldr compose identity '())))
1171 |
1172 | (compose rest (compose rest identity))
1173 |
1174 | (compose rest rest)
1175 |
1176 | ((lambda (f g)
1177 |   (lambda (x)
1178 |     (f (g x)))) rest rest)
1179 |
1180 | (lambda (x) (rest (rest x)))
1181 |
1182 |%END
  
```

1183 **Ex.**

```

1184 |%Racket_file
1185 |
1186 |> ((f^n rest 2) '(1 2 3 4 5))
1187 |'(3 4 5)
1188 |
1189 |%END
1190 |
1191 |%STEPPER
1192 |
1193 | ((f^n rest 2) '(1 2 3 4 5))
1194 |
1195 |
1196 |
1197 |
  
```

```

1198 | ...
1199 |
1200 | ((lambda (x) (rest (rest x))) '(1 2 3 4 5))
1201 |
1202 | (rest (rest '(1 2 3 4 5)))
1203 |
1204 | (rest '(2 3 4 5))
1205 |
1206 | '(3 4 5)
1207 |
1208 | %END

```

Ex.

```

1211 |
1212 | %Racket_file
1213 |
1214 | > (define cons-7
1215 |     (lambda (lst)
1216 |       (cons 7 lst)))
1217 |
1218 | > ((f^n cons-7 3) '(1 2 3))
1219 | '(7 7 7 1 2 3)
1220 |
1221 | %END

```

Look at `((f^n cons-7 3) first)`.

```

1225 | %STEPPER
1226 |
1227 | (f^n con-7 3)
1228 |
1229 | ((lambda (f n)
1230 |   (foldr
1231 |     compose
1232 |     identity
1233 |     (make-list n f))) cons-7 3)
1234 |
1235 | (foldr
1236 |   compose
1237 |   identity
1238 |   (make-list 3 cons-7))
1239 |
1240 | (foldr
1241 |   compose
1242 |   identity
1243 |   '(cons-7 cons-7 cons-7))
1244 |
1245 | (compose cons-7 (compose cons-7 cons-7))
1246 |
1247 | ((lambda (x) (cons-7 (cons-7 (cons-7 x))))
1248 |   '(1 2 3))
1249 | %END

```

Now back to `((f^n cons-7 3) '(1 2 3))`

```

1253 | %STEPPER
1254 |
1255 | ((f^n cons-7 3) '(1 2 3))
1256 |
1257 | ((lambda (x) (cons-7 (cons-7 (cons-7 x)))) '(1 2 3))
1258 |
1259 | (cons-7 (cons-7 '(7 1 2 3)))
1260 |

```

```

1261 | (cons-7 '(7 7 1 2 3))
1262 |
1263 | '(7 7 7 1 2 3)
1264 |
1265 |%END

```

1266

1267

1268 -----

C9: currying

1269

1270 **Note:**

1271 We can curry a 2-arg function into a "wrapped" 1-arg function.

1272

1273 **Note:**

1274 Before: (f : A -> B -> C)

1275 After: (f' : A -> (B -> C))

1276

1277 **Note:**

1278 Before: (A -> B -> C)

1279 After: (A -> (B -> C))

1280

1281

1282 **Implementation:**

1283

```

1284 |%Racket_file
1285 |
1286 | (define curry
1287 |   (lambda (f)
1288 |     (lambda (a)
1289 |       (lambda (b)
1290 |         (f a b))))))
1291 |
1292 |%END

```

1293

1294 **Remark:**

1295 - Argument:

1296 f : (A -> B -> C)

1297 - Return:

1298 A function

1299 - Argument

1300 a : A

1301 - Return

1302 A function

1303 - Argument

1304 b : B

1305 - Return

1306 (f a b) : C

1307

1308 **Ex.**

```

1309 |%Racket_file
1310 |
1311 | > (+ 1 2)
1312 | 3
1313 | > (+ 1)
1314 | 1
1315 | > (curry +)
1316 | #<procedure>
1317 | ;; At this point, we have only provided "f", so we get a
1318 | ;; procedure
1319 | ;;           (lambda (a)
1320 | ;;             (lambda (b)
1321 | ;;               (lambda (+ a b))))
1322 |
1323 | > ((curry +) 1)

```

```

1324 | #<procedure>
1325 | ;; Now we have provided both f and a, so we have a
1326 | ;; "partially applied" function. Our output lambda function
1327 | ;; looks like this:
1328 | ;;           (lambda (b)
1329 | ;;             (lambda (+ 1 b)))
1330 |
1331 | > (((curry +) 1) 2)
1332 | 3
1333 | ;; We have finally provided all three arguments, so we get
1334 | ;; a value back in return. In fact, ((curry +) 1) is equivalent
1335 | ;; to the function add1.
1336 |
1337 | %END

```

Note:

Let's look at the stepper.

```

1342 | %STEPPER
1343 |
1344 | ((curry +) 1)
1345 |
1346 | (((lambda (f)
1347 |   (lambda (a)
1348 |     (lambda (b)
1349 |       (f a b)))) +) 1)
1350 |
1351 | ((lambda (a)
1352 |   (lambda (b)
1353 |     (+ a b))) 1)
1354 |
1355 | (lambda (b) (+ 1 b))
1356 |
1357 | %END

```

Remark:

You can see how at every step, one more argument gets accumulated into the final expression. `(f a b)` eventually turns into `(lambda (b) (+ 1 b))` in a few steps.

Note:

In a sense, what we've done is "storing" some inputs inside a `lambda` function. Consider the following example:

Implementation:

```

1370 | %Racket_file
1371 |
1372 | (define curry*
1373 |   (lambda (a)
1374 |     (lambda (b)
1375 |       (lambda (f)
1376 |         (f a b))))))
1377 |
1378 | %END

```

Note:

What if we want to uncurry a function?

Let's start with type contract. We want to `reverse` the curry process.

Note:

For curry:

1387
 1388 Before: $(f : A \rightarrow B \rightarrow C)$
 1389 After: $(f' : A \rightarrow (B \rightarrow C))$

1390
 1391 Before: $(A \rightarrow B \rightarrow C)$
 1392 After: $(A \rightarrow (B \rightarrow C))$

1393
 1394 Then for uncurry:

1395
 1396 Before: $(f' : A \rightarrow (B \rightarrow C))$
 1397 After: $(f : A \rightarrow B \rightarrow C)$

1398
 1399 Before: $(A \rightarrow (B \rightarrow C))$
 1400 After: $(A \rightarrow B \rightarrow C)$

1401
 1402 **Implementation:**

```
1403 |%Racket_file
1404 |
1405 | (define uncurry
1406 |   (λ (f)
1407 |     (λ (a b)
1408 |       ((f a) b))))
1409 |
1410 |%END
```

1411
 1412
 1413 **Remark:**

1414 - Arguments:
 1415 **Warning:** f must be a curried function.
 1416 A function
 1417 - Arguments:
 1418 $a : A$
 1419 - Return:
 1420 A function
 1421 - Arguments:
 1422 $b : B$
 1423 - Return:
 1424 $(f a b) : C$
 1425 - Return:
 1426 A function
 1427 - Arguments:
 1428 $a : A$
 1429 $b : B$
 1430 - Return:
 1431 $((f a) b) : C$

1432
 1433 **Ex.**

```
1434 |%Racket_file
1435 |
1436 | (define curried-add
1437 |   (curry +))
1438 |
1439 | (uncurry curried-add)
1440 |
1441 |%END
1442 |
1443 |%STEPPER
1444 |
1445 | (uncurry curried-add)
1446 |
1447 | ;; rewrite the uncurry function
1448 | ((lambda (f)
1449 |
```

```

1450 | (lambda (a b)
1451 |   ((f a) b))) curried-add)
1452 |
1453 | ;; rewrite the curried-add function
1454 | ((lambda (f)
1455 |   (lambda (a b)
1456 |     ((f a) b)))
1457 |   ((lambda (f)
1458 |     (lambda (a)
1459 |       (lambda (b)
1460 |         (f a b)))))) +))
1461 |
1462 | ;; pass in +
1463 | (lambda (f)
1464 |   (lambda (a b)
1465 |     ((f a) b))
1466 |   (lambda (a)
1467 |     (lambda (b)
1468 |       (+ a b))))))
1469 |
1470 | ;; pass in the lambda function as f
1471 | (lambda (a b)
1472 |   (((lambda (a)
1473 |     (lambda (b)
1474 |       (+ a b)))
1475 |    a)
1476 |    b))
1477 |
1478 | %END

```

Remark:

- Argument:
 - a : A
 - b : B
- Return:
 - ((A function
 - Argument:
 - a : A
 - Return:
 - A function
 - Argument:
 - b : B
 - Return:
 - (+ a b) : C) applied onto a and b) : C

Warning:

This part is very messy. Please read carefully and make sure you understand it fully.

1499

1500 C10. map

1501

Implementation:

1502

1503 | %Racket_file

1504

```

1505 | (define map
1506 |   (lambda (f ls)
1507 |     (foldr
1508 |       (lambda (x acc) (cons (f x) acc))
1509 |       '()
1510 |       ls))))
1511 |
1512 |

```

1512

```

1513 |%END
1514
1515 Ex.
1516 | (map add1 '(1 2 3)) => '(2 3 4)
1517
1518
1519 -----

```

C11. filter

Implementation:

```

1524 |%Racket_file
1525 |
1526 | (define filter
1527 |   (lambda (f ls)
1528 |     (foldr
1529 |       (lambda (x acc) (if (f x) (cons x acc) acc))
1530 |       '()
1531 |       ls)))
1532 |
1533 |%END

```

```

1536 =====
1537 =====
1538 Part D. Other Minor Topics.

```

```

1539
1540 Table of contents
1541 D1. Modules
1542 D2. Smart helpers
1543 D3. Parameterized by total order
1544
1545 -----

```

D1. Modules

Theory: Modules

```

1550 |%Racket_file
1551 |
1552 | #lang racket
1553 |
1554 | ;; (sumto n) sums the integers from 0 to n, where n
1555 | ;;       is a non-negative integer.
1556 | ;; running time: O(n)
1557 |
1558 | (provide sumto)
1559 |
1560 | (define (sumto n)
1561 |   (if (zero? n) 0 (+ n (sumto (sub1 n)))))
1562 |
1563 |%END

```

Note: Provide statement

This `provide` statement is called a "directive", which tells Racket that other files may use this.

```

1568 |%Racket_file
1569 |
1570 | #lang racket
1571 |
1572 | ;; running time: O(n^2)
1573 |
1574 | (require "sum.rkt")
1575 |

```

```

1576 |
1577 | (define (sumsumto m)
1578 |   (if (zero? m) 0 (+ (sumto m) (sumsumto (sub1 m)))))
1579 |
1580 | ;; Don't panic. This is just a double summation.
1581 |
1582 | %END

```

Note: Requirement statement

The `require` statement in Racket is just like `import` in Python in other imperative languages (Python, Java, etc.)

D2. Smart helpers

Note:

You can use helper functions in a smart way when dealing with the following situations:

1. a calculation costs too much so you want to avoid doing it more than once, `or`
2. you want to record the value produced by the current function application.

In short, you want to create a "variable" to store some `values` for later use.

Implementation:

```

1604 | %Racket_file
1605 |
1606 | (define (f x)
1607 |   (h (g x) (g x)))
1608 |
1609 | ;; Let h be an arbitrary 2-arg function that you don't care about.
1610 | ;; (For example, h can be + or *).
1611 | ;; Let f, g be arbitrary 1-arg functions.
1612 | ;; Suppose (g x) costs too much and thus
1613 | ;; you want to avoid performing it twice.
1614 | ;; Then this function can be rewritten as:
1615 |
1616 | (define (f' x)
1617 |   (helper (g x)))
1618 |
1619 | (define (helper y)
1620 |   (h y y))
1621 |
1622 | ;; or use local helper function:
1623 |
1624 | (define (f'' x)
1625 |   (define (helper y)
1626 |     (h y y))
1627 |   (helepr (g x)))
1628 |
1629 | %END

```

Ex.

Now consider the function `(drawcard n)`.

Suppose you want to `do` the following:

1. Given a `list` of cards called `list-of-cards`.
2. Draw a new card `(drawcard n)`
3. If the new card's secret number is in the list, discard it. Otherwise `cons` the new card into the list.

```

1639
1640 If you use the old approach like this:
1641
1642 |%Racket_file
1643 |
1644 | ;; (drawcard n) produces a card which is represented
1645 | ;;   by a two-element list.
1646 | ;; (first (drawcard n)) returns the secret number of the card.
1647 | ;; (second (drawcard n)) returns n
1648 |
1649 | ;; Suppose we have defined a helper function called unique?
1650 | ;;   which can determine if there is an identical card (a card
1651 | ;;   with the same secret number) in the list or not.
1652 | ;; (unique : (c : Card) -> (l : listof Card) -> Bool)
1653 |
1654 | ;; (main : (n : Nat) -> (lst : listof Cards) -> (l : listof Cards))
1655 | (define (main n lst)
1656 |   (cond
1657 |     [(unique? (drawcard n) lst) ;; [Line #1]
1658 |      (cons (drawcard n) lst)] ;; [Line #2]
1659 |     [else lst]))
1660 |
1661 |%END

```

You would soon realize that the two `(drawcard n)` applications in line #1 and line #2 actually produce different cards and there's a very high chance these two cards have different secret numbers. Therefore your entire algorithm would be incorrect.

To fix this, you can use the above helper:

```

1670 |%Racket_file
1671 |
1672 | ;; main : (n : Nat) -> (lst : listof Card) -> (l : listof Card)
1673 | (define (main n lst)
1674 |   ;; helper : (c : Card) -> (ls : listof Card) -> (l : listof Card)
1675 |   (define (helper c ls)
1676 |     [(unique? c ls) (cons c ls)]
1677 |     [else ls])
1678 |   (helper (drawcard n) lst))
1679 |
1680 |%END

```

Note:

This way, the `c` inside the helper function stores your value for `(drawcard n)` and your program is correct.

D3. Parameterized by total order

Note:

In short, we can `define` our own rules for ordering, and we can also pass in `>` and `<` as arguments.

Note:

When we are defining our own struct, sometimes we need to `define` our own rules for order. To determine which element is "greater", we can use the following function to consume a function for comparison:

Implementation:

```

1700 |%Racket_file
1701 |

```

```

1702 | (define (less-than? f)
1703 |   (lambda (a b)
1704 |     (cond
1705 |       [(< (f a) (f b)) true]
1706 |       [(< (f b) (f a)) false]
1707 |       [else (< a b)])))
1708 |
1709 |%END

```

Ex.

```

1713 |%Racket_file
1714 |
1715 | > ((less-than? abs) -1 -2)
1716 | #true
1717 |
1718 |%END
1719 |
1720 |%STEPPER
1721 |
1722 | ((less-than? abs) -1 -2)
1723 |
1724 | ;; rewrite less-than?
1725 | ;; plug in abs as f
1726 | ((lambda (a b)
1727 |   (cond
1728 |     [(< (abs a) (abs b)) true]
1729 |     [(< (abs b) (abs a)) false]
1730 |     [else (< a b)]))
1731 |   -1 -2)
1732 |
1733 | ;; plug in -1 and -2 as a and b
1734 | (cond
1735 |   [(< (abs -1) (abs -2)) true]
1736 |   [(< (abs -2) (abs -1)) false]
1737 |   [else (< -1 -2)]))
1738 |
1739 | #true
1740 |
1741 |%END

```

```

1743 | =====
1744 | =====
1745 | Part E. Appendix

```

Table of contents

- 1748 E1. Practice from tutorial Oct.25 by Ashish.
- 1749 E2. Practice from tutorial Nov.1 by Ashish.
- 1750 E3. Assignment 6~8 Recap.
- 1751 E4. Functions you should 100% memorize.
- 1752 E5. Functions that may help you on the test.
- 1753 E6. Possible questions on the test.

Ex.

1. Rewrite the function in an unsugared form:

[Level of difficulty: easy]

Q1.rkt

```

1762 |%Racket_file
1763 |
1764 | (define (make-identity)

```

```

1765 | (lambda (x) x))
1766 |
1767 |%END
1768
1769 S1.rkt
1770
1771 |%Racket_file
1772 |
1773 | (define (make-identity x) x)
1774 |
1775 |%END
1776

```

Ex.

- Define "negate", which takes a boolean function `f`, and returns a function that produces the "not" of the output.

[Level of difficulty: easy]

```

;; Type contract:
;; (negate : (A -> Bool) -> (A -> Bool))

```

```

1787 S2.rkt
1788
1789 |%Racket_file
1790 |
1791 | (define negate
1792 |   (lambda (f)
1793 |     (lambda (x)
1794 |       (not (f x)))))
1795 |
1796 |%END
1797

```

Ex.

- Define `equal-to?`, which is a higher-order function. It should take a value that can be compared using `equal?` and return a function, that when applied to another argument, returns the `equal?` of arguments.

[Level of difficulty: easy]

```

;; Type contract:
;; equal-to? : (a1 : A) -> ((a2 : A) -> Bool)

```

```

1809 S3.rkt
1810
1811 |%Racket_file
1812 |
1813 | (define equal-to?
1814 |   (lambda (a1)
1815 |     (lambda (a2)
1816 |       (equal? a1 a2))))
1817 |
1818 |%END
1819

```

Ex.

- Define a function "dedup" which takes an ordered list and removes any duplicate elements. Write this function using `foldr`.

[Level of difficulty: medium]

```

;; Type contract:

```

```

1828 ;; dedup : (listof A) -> (listof A)
1829
1830 S4.rkt
1831
1832 |%Racket_file
1833 |
1834 | (define dedup
1835 |   (lambda (lst)
1836 |     (foldr
1837 |       (lambda (x acc) (if (not (member x acc))
1838 |                           (cons x acc) acc))
1839 |       '()
1840 |       lst)))
1841 |
1842 |%END

```

Ex.

5. Define "my-map", which takes a function **and** a list, **and** produces a **list** with **f** applied to all elements of the list. Use **foldr**.

[Level of difficulty: medium]

```

1850 ;; Type contract:
1851 ;; my-map : (A -> B) -> (listof A) -> (listof B)

```

```

1854 S5.rkt
1855
1856 |%Racket_file
1857 |
1858 | (define my-map
1859 |   (lambda (f lst)
1860 |     (foldr
1861 |       (lambda (x acc) (cons (f x) acc))
1862 |       '()
1863 |       lst)))
1864 |
1865 |%END

```

Ex.

6. Create a function "plus" that adds two natural numbers together. Use only above "f^n" **and** "add1", **do not** use recursion.

[Level of difficulty: medium]

```

1874 ;; Type contract:
1875 ;; plus: Nat -> Nat -> Nat

```

```

1877 S6.rkt
1878
1879 |%Racket_file
1880 |
1881 | (define plus
1882 |   (lambda (a b)
1883 |     ((f^n add1 a) b)))
1884 |
1885 |%END

```

Remark:

Here is an example **and** the step-by-step process.

```

1890 |%Racket_file

```

```

1891 |
1892 | (plus 3 15)
1893 |
1894 |%END
1895 |
1896 |%STEPPER
1897 |
1898 | (plus 3 15)
1899 |
1900 | ;; rewrite plus
1901 | ((lambda (a b) ((f^n add1 a) b)) 3 15)
1902 |
1903 | ;; pass in a, b
1904 | ((f^n add1 3) 15)
1905 |
1906 | ;; rewrite f^n
1907 | (((lambda (f n)
1908 |   (foldr compose identity
1909 |     (make-list n f))) add1 3) 15)
1910 |
1911 | ;; pass in add1, 3
1912 | ((foldr compose identity '(add1 add1 add1)) 15)
1913 |
1914 | ;; rewrite foldr
1915 | ((lambda (x)
1916 |   (add1 (add1 (add1 x)))) 15)
1917 |
1918 | ;; pass in 15
1919 | (add1 (add1 (add1 15)))
1920 |
1921 | 18
1922 |
1923 |%END

```

Ex.

7. Create a function "mult" that multiplies two natural numbers together. Use only the above "f^n" and "plus", do not use recursion.

[Level of difficulty: hard]

```

;; Type contract:
;; mult: Nat -> Nat -> Nat

```

S7.rkt

```

|%Racket_file
|
| (define mult
|   (lambda (a b)
|     ((f^n ((curry plus) a) b) 0))
|
|%END

```

Remark:

This is very very confusing.
I'll provide an example and step-by-step analysis.

```

|%Racket_file
|
| (mult 3 5)
|
|%END

```

```

1954
1955 |%STEPPER
1956
1957 | (mult 3 5)
1958
1959 | ;; rewrite mult
1960 | ((lambda (a b) ((f^n ((curry plus) a) b) 0)) 3 5)
1961
1962 | ;; plus in 3, 5
1963 | (((f^n (curry plus) 3) 5) 0)
1964
1965 | ;; rewrite curry
1966 | ((f^n
1967 |   ((lambda (f)
1968 |     (lambda (a)
1969 |       (lambda (b)
1970 |         (f a b)))) plus 3)
1971 |   5) 0)
1972
1973 | ;; plug in plus, 3
1974 | (((f^n
1975 |   (lambda (b) (plus 3 b))) 5) 0)
1976
1977 | ;; in the step above, we get a partial application of 3.
1978 | ;; now, rewrite f^n
1979 | ((lambda (f n) compose identity (make-list f n))
1980 |   (lambda (b) (plus 3 b)) ;; this thing is the f we are passing
1981 |                               ;; into the outer lambda.
1982 |   5) 0)
1983
1984 | ;; plug in 5 as n and (lambda (b) (plus 3 b))
1985 | ;; as f in the outer lambda
1986 | ((compose identity (make-list (lambda (b) (plus 3 b)) 5)) 0)
1987
1988 | ;; compose
1989 | ;; now it's the beauty of partial application.
1990 | (plus 3 (plus 3 (plus 3 (plus 3 (plus 3 0)))))
1991
1992 | 15
1993
1994 |%END

```

Remark:

This process is very challenging. Read it carefully.

E2. Practice from tutorial Nov.1 by Ashish

Note:

The following two functions are equivalent:

```

2005 |%Racket_file
2006
2007 | (define map-add1
2008 |   (lambda (lst)
2009 |     (map add1 lst)))
2010
2011 | (define map-adder
2012 |   ((curry map) add1))
2013
2014 |%END

```

Note that `((curry map) add1)` creates the

2017 function (lambda (lst) (map add1 lst)).

2018

2019 **Note:**

2020 The following two functions are equivalent:

2021

2022 |**%Racket_file**

2023

2024 | (define sum-list

2025 | (lambda (lst)

2026 | (foldr + 0 lst)))

2027

2028 | (define sum-list-curried

2029 | (((curry foldr) + 0)))

2030

2031 |**%END**

2032

2033 Currying and partial application too op.

2034

2035 -----

2036 E3. Assignment 6~8 Recap

2037

2038 Asst.6 a~e:

2039 Practice on time complexity.

2040 Read A1 and A6 and you should be good.

2041

2042 Asst.6 f:

2043 Given AVL, define a new ADT set.

2044

2045 **Remark:**

2046 Try to provide a wrapper struct.

2047

2048 Asst.7:

2049 Create a game to draw cards and collect prizes.

2050 Nothing too crazy but remember to check your running time.

2051

2052 Asst.8:

2053 Use generate function to write (prime? n),

2054 (my-build-list n f), (my-foldl f z l), (my-insert e l <),

2055 (my-insertion-sort l <). This is an exercise to help us

2056 what fold does.

2057

2058 Also there is a merge sort exercise. By now we should be

2059 comfortable dealing with sorting and searching algorithms.

2060

2061 -----

2062 E4. Functions you should 100% memorize.

2063

2064 |**%Racket_file**

2065

2066 | (define compose

2067 | (lambda (f g)

2068 | (lambda (x)

2069 | (f (g x))))

2070

2071 | (define map

2072 | (lambda (f lst)

2073 | (foldr

2074 | (lambda (x z) (cons (f x) z))

2075 | '()

2076 | lst)))

2077

2078 | (define filter

2079 | (lambda (f lst)

```

2080 | (foldr
2081 |   (lambda (x z) (if (f x) (cons x z) z))
2082 |   '()
2083 |   lst)))
2084 |
2085 | (define (foldr f z l)
2086 |   (cond
2087 |     [(empty? l) z]
2088 |     [else (f (first l) (foldr f z (rest l)))]))
2089 |
2090 | (define (foldl f z l)
2091 |   (define (calc l z)
2092 |     (cond
2093 |       [(empty? l) z]
2094 |       [else (calc (rest l) (f (first l) z))]))
2095 |   (calc l z))
2096 |
2097 | (define (f'' x)
2098 |   (define (helper y)
2099 |     (h y))
2100 |   (helper (g x)))
2101 |
2102 | %END

```

E5. Functions that may help you on the test.

```

2107 | %Racket_file
2108 |
2109 | (define curry
2110 |   (lambda (f)
2111 |     (lambda (a)
2112 |       (lambda (b)
2113 |         (f a b)))))
2114 |
2115 | (define f^n
2116 |   (lambda (f n)
2117 |     (foldr
2118 |       compose
2119 |       identity
2120 |       (make-list n f))))
2121 |
2122 | %END

```

E6. Possible questions on the test.

1. Relatively short problems
 - 1.1 Theory
 - One question about big-O
 - One question about ADT
 - One question about modules
 - 1.2 Short programs
 - One question about big-O
 - One question about ADT
 - One question about writing a `lambda` function
 - One question about application of a given `lambda` function
2. Topic: Big-O
 - 2.1 Prove why $f(x)$ is $O(g(x))$
 - 2.2 Prove why $h(x)$ is **not** $O(g(x))$
3. Topic: Lambda

2143 3.1 Write a function to ...
2144 3.2 Write a function to ...
2145 3.3 Given this function, do ...
2146
2147 4. Topic: Stuff from midterm I
2148 4.1 Use tree to define a new ADT
2149 4.2 Use tree to do some operations
2150
2151 5. Prove that merge sort / insertion sort is correct using induction.
2152
2153 -----
2154
2155 V3.0 Complete. 2017, Nov.3 by David Duan