

```

1 ;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: nutrition.rkt
2
3
4 (define-struct nutri-fact (name serving fat carbs sugar protein))
5 ;; A Nutri-Fact is a (make-nutri-fact Str Num Num Num Num Num)
6 ;; requires: 0 < serving
7 ;;           fat + carbs + protein <= serving
8 ;;           0 <= sugar <= carbs
9 ;;           0 <= fat, protein
10
11
12 (define coke-zero (make-nutri-fact "Coke Zero" 355 0 0 0 0))
13 (define cheerios (make-nutri-fact "Honey Nut Cheerios" 29 1.5 23 9 2))
14 (define cashews (make-nutri-fact "Cashews" 30 14 9 2 5))
15 (define ketchup (make-nutri-fact "Ketchup" 15 0 5 4 0.3))
16 (define tuna (make-nutri-fact "Canned Tuna" 55 1 0 0 11))
17
18 (define less-sugar (make-nutri-fact "less sugar" 4 0 0 0 0))
19 (define more-sugar (make-nutri-fact "more sugar" 4 0 1 1 0))
20 (define less-protein (make-nutri-fact "less protein" 4 0 1 1 0))
21 (define more-protein (make-nutri-fact "more protein" 4 0 1 1 1))
22 (define less-carbs (make-nutri-fact "less carbs" 4 0 1 1 1))
23 (define more-carbs (make-nutri-fact "more carbs" 4 0 2 1 1))
24 (define less-fat (make-nutri-fact "less fat" 4 0 1 1 1))
25 (define more-fat (make-nutri-fact "more fat" 4 1 1 1 1))
26
27
28 ;; 2(a)
29
30 ;; my-nutri-fact-fn: Nutri-Fact -> Any
31 (define (my-nutri-fact-fn food)
32   ( ... (nutri-fact-name food) ...
33         (nutri-fact-serving food) ...
34         (nutri-fact-fat food) ...
35         (nutri-fact-carbs food) ...
36         (nutri-fact-sugar food) ...
37         (nutri-fact-protein food) ...))
38
39
40 ;; 2(b)
41
42 ;;(resize food new-size) produces a new Nutri-Fact from food
43 ;;  with a serving size of new-size grams
44 ;; resize: Nutri-Fact Num -> Nutri-Fact
45 ;; requires: 0 < new-size
46 ;; Example:
47 (check-expect (resize cheerios 58)
48               (make-nutri-fact "Honey Nut Cheerios" 58 3 46 18 4))
49
50 (define (resize food new-size)
51   (make-nutri-fact (nutri-fact-name food)
52                   new-size
53                   (/ (* new-size (nutri-fact-fat food))
54                     (nutri-fact-serving food))
55                   (/ (* new-size (nutri-fact-carbs food))
56                     (nutri-fact-serving food))
57                   (/ (* new-size (nutri-fact-sugar food))
58                     (nutri-fact-serving food))

```

```

59         (/ (* new-size (nutri-fact-protein food))
60            (nutri-fact-serving food))))
61
62 ;; Tests:
63 (check-expect (resize cheerios 14.5)
64               (make-nutri-fact "Honey Nut Cheerios" 29/2 3/4 23/2 9/2 1))
65
66
67 ;; 2(c)
68
69 (define fat-calories/gram 9)
70 (define carbs-calories/gram 4)
71 (define protein-calories/gram 4)
72
73
74 ;;(calories food) determines the number of calories in food
75 ;; calories: Nutri-Fact -> Num
76 ;; Example:
77 (check-expect (calories cheerios) 113.5)
78
79 (define (calories food)
80   (+ (* (nutri-fact-fat food) fat-calories/gram)
81      (* (nutri-fact-carbs food) carbs-calories/gram)
82      (* (nutri-fact-protein food) protein-calories/gram)))
83
84 ;; Tests:
85 (check-expect (calories cashews) 182)
86
87
88 ;; 2(d)
89
90 ;;(choose-for-diet food1 food2) produces which of food1 and food2
91 ;; is healthier, prioritizing lower sugar proportion, then
92 ;; higher protein, then lower carbs, then lower fat
93 ;; choose-for-diet: Nutri-Fact Nutri-Fact -> Nutri-Fact
94 ;; Examples:
95 (check-expect (choose-for-diet cashews coke-zero) coke-zero)
96 (check-expect (choose-for-diet tuna (resize tuna 100)) tuna)
97
98 (define (choose-for-diet food1 food2)
99   (cond
100     [(< (/ (nutri-fact-sugar food1)
101            (nutri-fact-serving food1))
102          (/ (nutri-fact-sugar food2)
103             (nutri-fact-serving food2))) food1]
104     [(= (/ (nutri-fact-sugar food1)
105            (nutri-fact-serving food1))
106          (/ (nutri-fact-sugar food2)
107             (nutri-fact-serving food2)))
108      (cond
109        [(> (/ (nutri-fact-protein food1)
110                (nutri-fact-serving food1))
111              (/ (nutri-fact-protein food2)
112                 (nutri-fact-serving food2))) food1]
113        [(= (/ (nutri-fact-protein food1)
114                (nutri-fact-serving food1))
115              (/ (nutri-fact-protein food2)
116                 (nutri-fact-serving food2)))

```

```

117 (cond
118   [(< (/ (nutri-fact-carbs food1)
119         (nutri-fact-serving food1))
120        (/ (nutri-fact-carbs food2)
121            (nutri-fact-serving food2))) food1]
122   [(= (/ (nutri-fact-carbs food1)
123         (nutri-fact-serving food1))
124        (/ (nutri-fact-carbs food2)
125            (nutri-fact-serving food2)))
126   (cond
127     [(< (/ (nutri-fact-fat food1)
128           (nutri-fact-serving food1))
129          (/ (nutri-fact-fat food2)
130              (nutri-fact-serving food2))) food1]
131     [(= (/ (nutri-fact-fat food1)
132           (nutri-fact-serving food1))
133          (/ (nutri-fact-fat food2)
134              (nutri-fact-serving food2))) food1]
135     [else food2]]])
136   [else food2]])
137 [else food2]])
138 [else food2]])
139
140 ;; Tests:
141 (check-expect (choose-for-diet less-sugar more-sugar) less-sugar)
142 (check-expect (choose-for-diet more-sugar less-sugar) less-sugar)
143 (check-expect (choose-for-diet more-protein less-protein) more-protein)
144 (check-expect (choose-for-diet less-protein more-protein) more-protein)
145 (check-expect (choose-for-diet more-carbs less-carbs) less-carbs)
146 (check-expect (choose-for-diet less-carbs more-carbs) less-carbs)
147 (check-expect (choose-for-diet less-fat more-fat) less-fat)
148 (check-expect (choose-for-diet more-fat less-fat) less-fat)
149 (check-expect (choose-for-diet
150               (make-nutri-fact "first identical" 4 1 1 1 1)
151               (make-nutri-fact "second identical" 4 1 1 1 1))
152               (make-nutri-fact "first identical" 4 1 1 1 1))
153
154
155 ;; 2(e)
156
157 ;; (valid-nutri-fact? food) determines if food is a valid nutri-fact
158 ;; valid-nutri-fact?: Any -> Bool
159 ;; Examples:
160 (check-expect (valid-nutri-fact? cheerios) true)
161 (check-expect (valid-nutri-fact? "Pizza") false)
162 (check-expect (valid-nutri-fact? (make-nutri-fact "toomuch" 10 10 10 10 10)) false)
163
164 (define (valid-nutri-fact? food)
165   (and (nutri-fact? food)
166        (string? (nutri-fact-name food))
167        (number? (nutri-fact-serving food))
168        (number? (nutri-fact-fat food))
169        (number? (nutri-fact-carbs food))
170        (number? (nutri-fact-sugar food))
171        (number? (nutri-fact-protein food))
172        (< 0 (nutri-fact-serving food))
173        (<= (+ (nutri-fact-fat food)
174                (nutri-fact-carbs food)

```

```

175         (nutri-fact-protein food))
176         (nutri-fact-serving food))
177     (<= 0
178         (nutri-fact-sugar food)
179         (nutri-fact-carbs food))
180     (<= 0 (nutri-fact-fat food))
181     (<= 0 (nutri-fact-protein food))))
182
183 ;; Tests:
184 (check-expect (valid-nutri-fact? 'no) false)
185 (check-expect (valid-nutri-fact? (make-nutri-fact 1 10 15 0 0 0)) false)
186 (check-expect (valid-nutri-fact? (make-nutri-fact "types" 'no 15 0 0 0)) false)
187 (check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 'no 0 0 0)) false)
188 (check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 'no 0 0)) false)
189 (check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 0 'no 0)) false)
190 (check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 0 0 'no)) false)
191 (check-expect (valid-nutri-fact? (make-nutri-fact "empty" 0 0 0 0 0)) false)
192 (check-expect (valid-nutri-fact? (make-nutri-fact "neg" -5 0 0 0 0)) false)
193 (check-expect (valid-nutri-fact? (make-nutri-fact "justenough" 5 1 2 1 1)) true)
194 (check-expect (valid-nutri-fact? (make-nutri-fact "over-sugared" 10 0 5 6 0)) false)
195 (check-expect (valid-nutri-fact? (make-nutri-fact "negsug" 5 0 2 -1 0)) false)
196 (check-expect (valid-nutri-fact? (make-nutri-fact "nosug" 5 1 2 0 1)) true)
197 (check-expect (valid-nutri-fact? (make-nutri-fact "allsug" 5 0 2 2 0)) true)
198 (check-expect (valid-nutri-fact? (make-nutri-fact "nocarbs" 5 1 0 0 1)) true)
199 (check-expect (valid-nutri-fact? (make-nutri-fact "negfat" 5 -1 0 0 0)) false)
200 (check-expect (valid-nutri-fact? (make-nutri-fact "nofat" 5 0 1 1 1)) true)
201 (check-expect (valid-nutri-fact? (make-nutri-fact "negprot" 5 1 1 1 -1)) false)
202 (check-expect (valid-nutri-fact? (make-nutri-fact "noprot" 5 1 1 1 0)) true)
203
204
205
206 ;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: creditcheck.rkt
207
208 (define-struct date (year month day))
209 ;; A Date is a (make-date Nat Nat Nat)
210 ;; requires: year/month/day corresponds to a valid date
211 ;;           (in the Gregorian calendar)
212
213 (define-struct transaction (tdate amount category))
214 ;; A Transaction is a (make-transaction Date Num Sym)
215
216 (define-struct account (name expires limit threshold exception))
217 ;; An Account is a (make-account Str Date Num Num Sym)
218 ;; requires: 0 < threshold < limit
219
220 (define thanksgiving (make-date 2017 10 9))
221 (define halloween (make-date 2017 10 31))
222 (define allsouls (make-date 2017 11 2))
223 (define turkey (make-transaction thanksgiving 20 'food))
224 (define candy (make-transaction halloween 10 'food))
225 (define advil (make-transaction allsouls 20 'medicine))
226
227
228 ;; 3(a)
229
230 ;; (date<=? date1 date2) determines if date1 occurs before date2
231 ;; or is the same date as date2
232 ;; date<=? : Date Date -> Bool

```

```

233 ;; Examples:
234 (check-expect (date<=? allsouls halloween) false)
235 (check-expect (date<=? halloween halloween) true)
236
237 (define (date<=? date1 date2)
238   (or (< (date-year date1) (date-year date2))
239       (and (= (date-year date1) (date-year date2))
240             (or (< (date-month date1) (date-month date2))
241                 (and (= (date-month date1) (date-month date2))
242                       (<= (date-day date1) (date-day date2)))))))
243
244 ;; Tests:
245 (check-expect (date<=? (make-date 2016 2 4) (make-date 2017 2 4)) true)
246 (check-expect (date<=? (make-date 2017 2 4) (make-date 2016 2 4)) false)
247 (check-expect (date<=? (make-date 2017 1 4) (make-date 2017 2 4)) true)
248 (check-expect (date<=? (make-date 2017 2 4) (make-date 2017 1 4)) false)
249 (check-expect (date<=? (make-date 2017 2 3) (make-date 2017 2 4)) true)
250 (check-expect (date<=? (make-date 2017 2 4) (make-date 2017 2 3)) false)
251
252
253 ;; 3(b)
254
255 ;;(approve? purchase customer) determines if the purchase
256 ;; is valid for the given customer account
257 ;; approve? Transaction Account -> Bool
258 ;; Examples:
259 (check-expect (approve? candy (make-account "Bob" allsouls 100 50 'none))
260               true)
261 (check-expect (approve? candy (make-account "late" thanksgiving 100 50 'none))
262               false)
263
264 (define (approve? purchase customer)
265   (and (date<=? (transaction-date purchase) (account-expires customer))
266        (<= (transaction-amount purchase) (account-limit customer))))
267
268 ;; Tests:
269 (check-expect (approve? candy (make-account "" halloween 100 50 'none))
270               true)
271 (check-expect (approve? candy (make-account "" halloween 10.1 5 'none))
272               true)
273 (check-expect (approve? candy (make-account "" halloween 10 5 'none))
274               true)
275 (check-expect (approve? candy (make-account "" halloween 9.9 5 'none))
276               false)
277
278
279 ;; 3(c)
280
281 ;;(alert? purchase customer) determines if the purchase is approved but
282 ;; exceeds the threshold limit set by the customer and is not in
283 ;; the exception category set by the customer
284 ;; alert?: Transaction Account -> Bool
285 ;; Examples:
286 (check-expect true (alert? candy
287                      (make-account "Bob" allsouls 100 5 'none)))
288 (check-expect false (alert? candy
289                      (make-account "Bob" allsouls 100 5 'food)))
290

```



```

291 (define (alert? purchase customer)
292   (and (approve? purchase customer)
293     (not (symbol=? (transaction-category purchase)
294       (account-exception customer))))
295     (> (transaction-amount purchase)
296       (account-threshold customer))))
297
298 ;; Tests:
299 (check-expect false (alert? candy
300   (make-account "" thanksgiving 100 50 'none)))
301 (check-expect true (alert? candy
302   (make-account "" allsouls 100 9.9 'none)))
303 (check-expect false (alert? candy
304   (make-account "" allsouls 100 10 'none)))
305 (check-expect false (alert? candy
306   (make-account "" allsouls 100 10.1 'none)))
307
308
309
310 ;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: battle.rkt
311
312 (define-struct card (strength colour))
313 ;; A Card is a (make-card Nat Sym)
314 ;; requires: 1 <= strength <= 9
315 ;;           colour is one of
316 ;;           ('red 'yellow 'green 'blue 'purple 'brown)
317
318 (define-struct hand (c1 c2 c3))
319 ;; A Hand is a (make-hand Card Card Card)
320
321
322 (define strong-colour-run (make-hand (make-card 7 'red)
323   (make-card 8 'red)
324   (make-card 9 'red)))
325 (define weak-colour-run (make-hand (make-card 3 'red)
326   (make-card 2 'red)
327   (make-card 1 'red)))
328 (define strong-3kind (make-hand (make-card 9 'red)
329   (make-card 9 'yellow)
330   (make-card 9 'green)))
331 (define weak-3kind (make-hand (make-card 1 'red)
332   (make-card 1 'yellow)
333   (make-card 1 'green)))
334 (define strong-colour (make-hand (make-card 6 'red)
335   (make-card 8 'red)
336   (make-card 9 'red)))
337 (define weak-colour (make-hand (make-card 1 'red)
338   (make-card 2 'red)
339   (make-card 4 'red)))
340 (define strong-run (make-hand (make-card 7 'red)
341   (make-card 9 'red)
342   (make-card 8 'green)))
343 (define weak-run (make-hand (make-card 3 'green)
344   (make-card 1 'red)
345   (make-card 2 'red)))
346 (define strong-sum (make-hand (make-card 6 'red)
347   (make-card 8 'green)
348   (make-card 9 'red)))

```

```

349 (define weak-sum (make-hand (make-card 1 'red)
350                             (make-card 2 'yellow)
351                             (make-card 4 'green)))
352
353
354 ;;(run? cards) determines if the strengths of three cards
355 ;; form a consecutive sequence
356 ;; run?: Hand -> Bool
357 ;; Examples:
358 (check-expect (run? strong-run) true)
359 (check-expect (run? weak-run) true)
360 (check-expect (run? weak-sum) false)
361
362 (define (run? cards)
363   (and (= 2 (- (max (card-strength (hand-c1 cards))
364                     (card-strength (hand-c2 cards))
365                     (card-strength (hand-c3 cards)))
366             (min (card-strength (hand-c1 cards))
367                   (card-strength (hand-c2 cards))
368                   (card-strength (hand-c3 cards))))))
369   (= (+ 1 (min (card-strength (hand-c1 cards))
370                (card-strength (hand-c2 cards))
371                (card-strength (hand-c3 cards))))
       (- (+ (card-strength (hand-c1 cards))
              (card-strength (hand-c2 cards))
              (card-strength (hand-c3 cards)))
          (+ (min (card-strength (hand-c1 cards))
                  (card-strength (hand-c2 cards))
                  (card-strength (hand-c3 cards)))
              (max (card-strength (hand-c1 cards))
                    (card-strength (hand-c2 cards))
                    (card-strength (hand-c3 cards))))))))
381
382
383 ;;(colour? cards) determines if all cards share a colour
384 ;; colour?: Hand -> Bool
385 ;; Examples:
386 (check-expect (colour? strong-colour) true)
387 (check-expect (colour? weak-sum) false)
388
389 (define (colour? cards)
390   (and (symbol=? (card-colour (hand-c1 cards))
391              (card-colour (hand-c2 cards)))
392        (symbol=? (card-colour (hand-c1 cards))
393                  (card-colour (hand-c3 cards))))
394
395
396 ;;(strength-sum cards) produces the sum of the strengths of cards
397 ;; strength-sum: Hand -> Nat
398 ;; Example:
399 (check-expect (strength-sum strong-sum) 23)
400
401 (define (strength-sum cards)
402   (+ (card-strength (hand-c1 cards))
403      (card-strength (hand-c2 cards))
404      (card-strength (hand-c3 cards))))
405
406

```

```

407 ;;(three-kind? cards) determines if all strengths in cards are the same
408 ;; three-kind?: Hand -> Bool
409 ;; Examples:
410 (check-expect (three-kind? strong-3kind) true)
411 (check-expect (three-kind? weak-sum) false)
412
413 (define (three-kind? cards)
414   (= (card-strength (hand-c1 cards))
415      (card-strength (hand-c2 cards))
416      (card-strength (hand-c3 cards))))
417
418
419 ;;(battle hand1 hand2) determines which of player1 (hand1) and player2 (hand2) wins
420 ;;   in a battle of Schotten Totten, with player1 winning ties
421 ;; battle: Hand Hand -> (anyof 'player1 'player2)
422 ;; Examples:
423 (check-expect (battle weak-sum strong-colour-run) 'player2)
424 (check-expect (battle weak-sum weak-sum) 'player1)
425
426 (define (battle hand1 hand2)
427   (cond
428     [(and (colour? hand1) (run? hand1)
429           (or (not (and (colour? hand2) (run? hand2)))
430               (>= (strength-sum hand1)
431                   (strength-sum hand2)))) 'player1]
432     [(and (colour? hand2) (run? hand2)) 'player2]
433     [(and (three-kind? hand1) (or (not (three-kind? hand2))
434                                   (>= (strength-sum hand1)
435                                       (strength-sum hand2)))) 'player1]
436     [(three-kind? hand2) 'player2]
437     [(and (colour? hand1)
438           (or (not (colour? hand2))
439               (>= (strength-sum hand1)
440                   (strength-sum hand2)))) 'player1]
441     [(colour? hand2) 'player2]
442     [(and (run? hand1)
443           (or (not (run? hand2))
444               (>= (strength-sum hand1)
445                   (strength-sum hand2)))) 'player1]
446     [(run? hand2) 'player2]
447     [(>= (strength-sum hand1)
448          (strength-sum hand2)) 'player1]
449     [else 'player2]))
450
451 ;; Tests:
452 (check-expect (battle strong-colour-run strong-colour-run) 'player1)
453 (check-expect (battle strong-colour-run weak-colour-run) 'player1)
454 (check-expect (battle weak-colour-run strong-colour-run) 'player2)
455 (check-expect (battle strong-colour-run weak-sum) 'player1)
456
457 (check-expect (battle strong-3kind strong-3kind) 'player1)
458 (check-expect (battle strong-3kind weak-3kind) 'player1)
459 (check-expect (battle weak-3kind strong-3kind) 'player2)
460 (check-expect (battle strong-3kind weak-sum) 'player1)
461 (check-expect (battle weak-sum strong-3kind) 'player2)
462
463 (check-expect (battle strong-colour strong-colour) 'player1)
464 (check-expect (battle strong-colour weak-colour) 'player1)

```



```

465 (check-expect (battle weak-colour strong-colour) 'player2)
466 (check-expect (battle strong-colour weak-sum) 'player1)
467 (check-expect (battle weak-sum strong-colour) 'player2)
468
469 (check-expect (battle strong-run strong-run) 'player1)
470 (check-expect (battle strong-run weak-run) 'player1)
471 (check-expect (battle weak-run strong-run) 'player2)
472 (check-expect (battle strong-run weak-sum) 'player1)
473 (check-expect (battle weak-sum strong-run) 'player2)
474
475 (check-expect (battle strong-sum strong-sum) 'player1)
476 (check-expect (battle strong-sum weak-sum) 'player1)
477 (check-expect (battle weak-sum strong-sum) 'player2)
478
479
480
481 ;; ***** Alternate Battle Solution *****
482
483
484 ;; This solution maps the score of a hand to a Nat, where the score is simply
485 ;; the sum of the three strengths, plus an additive "bonus" if the
486 ;; hand satisfies any of the following combinations
487 (define bonus-run 200)      ;; three cards in sequence
488 (define bonus-colour 300)   ;; three cards with the same colour
489 (define bonus-3kind 400)    ;; three cards with the same strength
490
491
492 ;;(score cards) determines a "score" for the given hand of cards
493 ;; (see text and constants above)
494 ;; score: Hand -> Nat
495 ;; Examples:
496 (check-expect (score weak-sum) 7)
497 (check-expect (score weak-run) 200)
498 (check-expect (score strong-colour) 323)
499 (check-expect (score weak-3kind) 403)
500 (check-expect (score strong-colour-run) 524)
501
502 (define (score cards)
503   (+ (strength-sum cards)
504     (cond [(run? cards) bonus-run]
505           [else 0])
506     (cond [(colour? cards) bonus-colour]
507           [else 0])
508     (cond [(three-kind? cards) bonus-3kind]
509           [else 0])))
510
511
512 ;;(battle/alt hand1 hand2) determines which of player1 (hand1) and player2 (hand2)
512 wins
513 ;; in a battle of Schotten Totten, with player1 winning ties
514 ;; battle/alt: Hand Hand -> (anyof 'player1 'player2)
515 ;; Examples:
516 (check-expect (battle/alt weak-sum strong-colour-run) 'player2)
517 (check-expect (battle/alt weak-sum weak-sum) 'player1)
518
519 (define (battle/alt hand1 hand2)
520   (cond [(>= (score hand1) (score hand2)) 'player1]
521         [else 'player2]))

```

```
522 |
523 | ;; Tests:
524 | (check-expect (battle/alt weak-sum strong-sum) 'player2)
525 | (check-expect (battle/alt strong-sum weak-sum) 'player1)
526 | (check-expect (battle/alt strong-sum weak-run) 'player2)
527 | (check-expect (battle/alt strong-run weak-run) 'player1)
528 | (check-expect (battle/alt strong-run weak-colour) 'player2)
529 | (check-expect (battle/alt strong-colour weak-colour) 'player1)
530 | (check-expect (battle/alt strong-colour weak-3kind) 'player2)
531 | (check-expect (battle/alt strong-3kind weak-3kind) 'player1)
532 | (check-expect (battle/alt strong-3kind weak-colour-run) 'player2)
533 | (check-expect (battle/alt strong-colour-run weak-colour-run) 'player1)
534 |
535 |
```

sdhuang