```racket
;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: nutrition.rkt


(define-struct nutri-fact (name serving fat carbs sugar protein))
;; A Nutri-Fact is a (make-nutri-fact Str Num Num Num Num Num)
;; requires: 0 < serving
;;           fat + carbs + protein <= serving
;;           0 <= sugar <= carbs
;;           0 <= fat, protein


(define coke-zero (make-nutri-fact "Coke Zero" 355 0 0 0 0))
(define cheerios (make-nutri-fact "Honey Nut Cheerios" 29 1.5 23 9 2))
(define cashews (make-nutri-fact "Cashews" 30 14 9 2 5))
(define ketchup (make-nutri-fact "Ketchup" 15 0 5 4 0.3))
(define tuna (make-nutri-fact "Canned Tuna" 55 1 0 0 11))

(define less-sugar (make-nutri-fact "less sugar" 4 0 0 0 0))
(define more-sugar (make-nutri-fact "more sugar" 4 0 1 1 0))
(define less-protein (make-nutri-fact "less protein" 4 0 1 1 0))
(define more-protein (make-nutri-fact "more protein" 4 0 1 1 1))
(define less-carbs (make-nutri-fact "less carbs" 4 0 1 1 1))
(define more-carbs (make-nutri-fact "more carbs" 4 0 2 1 1))
(define less-fat (make-nutri-fact "less fat" 4 0 1 1 1))
(define more-fat (make-nutri-fact "more fat" 4 1 1 1 1))


;; 2(a)

;; my-nutri-fact-fn: Nutri-Fact -> Any
(define (my-nutri-fact-fn food)
  ( ... (nutri-fact-name food) ...
        (nutri-fact-serving food) ...
        (nutri-fact-fat food) ...
        (nutri-fact-carbs food) ...
        (nutri-fact-sugar food) ...
        (nutri-fact-protein food) ...))


;; 2(b)

;;(resize food new-size) produces a new Nutri-Fact from food
;;   with a serving size of new-size grams
;; resize: Nutri-Fact Num -> Nutri-Fact
;; requires: 0 < new-size
;; Example:
(check-expect (resize cheerios 58)
              (make-nutri-fact "Honey Nut Cheerios" 58 3 46 18 4))

(define (resize food new-size)
  (make-nutri-fact (nutri-fact-name food)
                   new-size
                   (/ (* new-size (nutri-fact-fat food))
                      (nutri-fact-serving food))
                   (/ (* new-size (nutri-fact-carbs food))
                      (nutri-fact-serving food))
                   (/ (* new-size (nutri-fact-sugar food))
                      (nutri-fact-serving food))
```

```
                  (/ (* new-size (nutri-fact-protein food))
                     (nutri-fact-serving food))))

;; Tests:
(check-expect (resize cheerios 14.5)
              (make-nutri-fact "Honey Nut Cheerios" 29/2 3/4 23/2 9/2 1))


;; 2(c)

(define fat-calories/gram 9)
(define carbs-calories/gram 4)
(define protein-calories/gram 4)


;;(calories food) determines the number of calories in food
;; calories: Nutri-Fact -> Num
;; Example:
(check-expect (calories cheerios) 113.5)

(define (calories food)
  (+ (* (nutri-fact-fat food) fat-calories/gram)
     (* (nutri-fact-carbs food) carbs-calories/gram)
     (* (nutri-fact-protein food) protein-calories/gram)))

;; Tests:
(check-expect (calories cashews) 182)


;; 2(d)

;;(choose-for-diet food1 food2) produces which of food1 and food2
;;  is healthier, prioritizing lower sugar proportion, then
;;  higher protein, then lower carbs, then lower fat
;; choose-for-diet: Nutri-Fact Nutri-Fact -> Nutri-Fact
;; Examples:
(check-expect (choose-for-diet cashews coke-zero) coke-zero)
(check-expect (choose-for-diet tuna (resize tuna 100)) tuna)

(define (choose-for-diet food1 food2)
  (cond
    [(< (/ (nutri-fact-sugar food1)
           (nutri-fact-serving food1))
        (/ (nutri-fact-sugar food2)
           (nutri-fact-serving food2))) food1]
    [(= (/ (nutri-fact-sugar food1)
           (nutri-fact-serving food1))
        (/ (nutri-fact-sugar food2)
           (nutri-fact-serving food2)))
     (cond
       [(> (/ (nutri-fact-protein food1)
              (nutri-fact-serving food1))
           (/ (nutri-fact-protein food2)
              (nutri-fact-serving food2))) food1]
       [(= (/ (nutri-fact-protein food1)
              (nutri-fact-serving food1))
           (/ (nutri-fact-protein food2)
              (nutri-fact-serving food2)))
```

```
        (cond
          [(< (/ (nutri-fact-carbs food1)
                 (nutri-fact-serving food1))
              (/ (nutri-fact-carbs food2)
                 (nutri-fact-serving food2))) food1]
          [(= (/ (nutri-fact-carbs food1)
                 (nutri-fact-serving food1))
              (/ (nutri-fact-carbs food2)
                 (nutri-fact-serving food2)))
           (cond
             [(< (/ (nutri-fact-fat food1)
                    (nutri-fact-serving food1))
                 (/ (nutri-fact-fat food2)
                    (nutri-fact-serving food2))) food1]
             [(= (/ (nutri-fact-fat food1)
                    (nutri-fact-serving food1))
                 (/ (nutri-fact-fat food2)
                    (nutri-fact-serving food2))) food1]
             [else food2])]
          [else food2])]
       [else food2]))

;; Tests:
(check-expect (choose-for-diet less-sugar more-sugar) less-sugar)
(check-expect (choose-for-diet more-sugar less-sugar) less-sugar)
(check-expect (choose-for-diet more-protein less-protein) more-protein)
(check-expect (choose-for-diet less-protein more-protein) more-protein)
(check-expect (choose-for-diet more-carbs less-carbs) less-carbs)
(check-expect (choose-for-diet less-carbs more-carbs) less-carbs)
(check-expect (choose-for-diet less-fat more-fat) less-fat)
(check-expect (choose-for-diet more-fat less-fat) less-fat)
(check-expect (choose-for-diet
               (make-nutri-fact "first identical" 4 1 1 1 1)
               (make-nutri-fact "second identical" 4 1 1 1 1))
              (make-nutri-fact "first identical" 4 1 1 1 1))


;; 2(e)

;;(valid-nutri-fact? food) determines if food is a valid nutri-fact
;; valid-nutri-fact?: Any -> Bool
;; Examples:
(check-expect (valid-nutri-fact? cheerios) true)
(check-expect (valid-nutri-fact? "Pizza") false)
(check-expect (valid-nutri-fact? (make-nutri-fact "toomuch" 10 10 10 10 10)) false)

(define (valid-nutri-fact? food)
  (and (nutri-fact? food)
       (string? (nutri-fact-name food))
       (number? (nutri-fact-serving food))
       (number? (nutri-fact-fat food))
       (number? (nutri-fact-carbs food))
       (number? (nutri-fact-sugar food))
       (number? (nutri-fact-protein food))
       (< 0 (nutri-fact-serving food))
       (<= (+ (nutri-fact-fat food)
              (nutri-fact-carbs food)
```

```
             (nutri-fact-protein food))
          (nutri-fact-serving food))
       (<= 0
          (nutri-fact-sugar food)
          (nutri-fact-carbs food))
       (<= 0 (nutri-fact-fat food))
       (<= 0 (nutri-fact-protein food))))

;; Tests:
(check-expect (valid-nutri-fact? 'no) false)
(check-expect (valid-nutri-fact? (make-nutri-fact 1 10 15 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "types" 'no 15 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 'no 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 'no 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 0 'no 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "types" 10 15 0 0 'no)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "empty" 0 0 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "neg" -5 0 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "justenough" 5 1 2 1 1)) true)
(check-expect (valid-nutri-fact? (make-nutri-fact "over-sugared" 10 0 5 6 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "negsug" 5 0 2 -1 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "nosug" 5 1 2 0 1)) true)
(check-expect (valid-nutri-fact? (make-nutri-fact "allsug" 5 0 2 2 0)) true)
(check-expect (valid-nutri-fact? (make-nutri-fact "nocarbs" 5 1 0 0 1)) true)
(check-expect (valid-nutri-fact? (make-nutri-fact "negfat" 5 -1 0 0 0)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "nofat" 5 0 1 1 1)) true)
(check-expect (valid-nutri-fact? (make-nutri-fact "negprot" 5 1 1 1 -1)) false)
(check-expect (valid-nutri-fact? (make-nutri-fact "noprot" 5 1 1 1 0)) true)



;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: creditcheck.rkt

(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)
;; requires: year/month/day corresponds to a valid date
;;           (in the Gregorian calendar)

(define-struct transaction (tdate amount category))
;; A Transaction is a (make-transaction Date Num Sym)

(define-struct account (name expires limit threshold exception))
;; An Account is a (make-account Str Date Num Num Sym)
;; requires: 0 < threshold < limit

(define thanksgiving (make-date 2017 10 9))
(define halloween (make-date 2017 10 31))
(define allsouls (make-date 2017 11 2))
(define turkey (make-transaction thanksgiving 20 'food))
(define candy (make-transaction halloween 10 'food))
(define advil (make-transaction allsouls 20 'medicine))


;; 3(a)

;;(date<=? date1 date2) determines if date1 occurs before date2
;;   or is the same date as date2
;; date<=?: Date Date -> Bool
```

```
;; Examples:
(check-expect (date<=? allsouls halloween) false)
(check-expect (date<=? halloween halloween) true)

(define (date<=? date1 date2)
  (or (< (date-year date1) (date-year date2))
      (and (= (date-year date1) (date-year date2))
           (or (< (date-month date1) (date-month date2))
               (and (= (date-month date1) (date-month date2))
                    (<= (date-day date1) (date-day date2)))))))

;; Tests:
(check-expect (date<=? (make-date 2016 2 4) (make-date 2017 2 4)) true)
(check-expect (date<=? (make-date 2017 2 4) (make-date 2016 2 4)) false)
(check-expect (date<=? (make-date 2017 1 4) (make-date 2017 2 4)) true)
(check-expect (date<=? (make-date 2017 2 4) (make-date 2017 1 4)) false)
(check-expect (date<=? (make-date 2017 2 3) (make-date 2017 2 4)) true)
(check-expect (date<=? (make-date 2017 2 4) (make-date 2017 2 3)) false)


;; 3(b)

;;(approve? purchase customer) determines if the purchase
;;   is valid for the given customer account
;; approve? Transaction Account -> Bool
;; Examples:
(check-expect (approve? candy (make-account "Bob" allsouls 100 50 'none))
              true)
(check-expect (approve? candy (make-account "late" thanksgiving 100 50 'none))
              false)

(define (approve? purchase customer)
  (and (date<=? (transaction-tdate purchase) (account-expires customer))
       (<= (transaction-amount purchase) (account-limit customer))))

;; Tests:
(check-expect (approve? candy (make-account "" halloween 100 50 'none))
              true)
(check-expect (approve? candy (make-account "" halloween 10.1 5 'none))
              true)
(check-expect (approve? candy (make-account "" halloween 10 5 'none))
              true)
(check-expect (approve? candy (make-account "" halloween 9.9 5 'none))
              false)


;; 3(c)

;;(alert? purchase customer) determines if the purchase is approved but
;;   exceeds the threshold limit set by the customer and is not in
;;   the exception category set by the customer
;; alert?: Transaction Account -> Bool
;; Examples:
(check-expect true (alert? candy
                           (make-account "Bob" allsouls 100 5 'none)))
(check-expect false (alert? candy
                            (make-account "Bob" allsouls 100 5 'food)))

```

```
(define (alert? purchase customer)
  (and (approve? purchase customer)
       (not (symbol=? (transaction-category purchase)
                      (account-exception customer)))
       (> (transaction-amount purchase)
          (account-threshold customer))))

;; Tests:
(check-expect false (alert? candy
                            (make-account "" thanksgiving 100 50 'none)))
(check-expect true (alert? candy
                           (make-account "" allsouls 100 9.9 'none)))
(check-expect false (alert? candy
                            (make-account "" allsouls 100 10 'none)))
(check-expect false (alert? candy
                            (make-account "" allsouls 100 10.1 'none)))



;; CS 135 :: Fall 2017 :: Posted solution :: A03 :: battle.rkt

(define-struct card (strength colour))
;; A Card is a (make-card Nat Sym)
;; requires: 1 <= strength <= 9
;;           colour is one of
;;           ('red 'yellow 'green 'blue 'purple 'brown)

(define-struct hand (c1 c2 c3))
;; A Hand is a (make-hand Card Card Card)


(define strong-colour-run (make-hand (make-card 7 'red)
                                     (make-card 8 'red)
                                     (make-card 9 'red)))
(define weak-colour-run (make-hand (make-card 3 'red)
                                   (make-card 2 'red)
                                   (make-card 1 'red)))
(define strong-3kind (make-hand (make-card 9 'red)
                                (make-card 9 'yellow)
                                (make-card 9 'green)))
(define weak-3kind (make-hand (make-card 1 'red)
                              (make-card 1 'yellow)
                              (make-card 1 'green)))
(define strong-colour (make-hand (make-card 6 'red)
                                 (make-card 8 'red)
                                 (make-card 9 'red)))
(define weak-colour (make-hand (make-card 1 'red)
                               (make-card 2 'red)
                               (make-card 4 'red)))
(define strong-run (make-hand (make-card 7 'red)
                              (make-card 9 'red)
                              (make-card 8 'green)))
(define weak-run (make-hand (make-card 3 'green)
                            (make-card 1 'red)
                            (make-card 2 'red)))
(define strong-sum (make-hand (make-card 6 'red)
                              (make-card 8 'green)
                              (make-card 9 'red)))
```

```
(define weak-sum (make-hand (make-card 1 'red)
                            (make-card 2 'yellow)
                            (make-card 4 'green)))


;;(run? cards) determines if the strengths of three cards
;;   form a consecutive sequence
;; run?: Hand -> Bool
;; Examples:
(check-expect (run? strong-run) true)
(check-expect (run? weak-run) true)
(check-expect (run? weak-sum) false)

(define (run? cards)
  (and (= 2 (- (max (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)))))
       (= (+ 1 (min (card-strength (hand-c1 cards))
                    (card-strength (hand-c2 cards))
                    (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)))))))


;;(colour? cards) determines if all cards share a colour
;; colour?: Hand -> Bool
;; Examples:
(check-expect (colour? strong-colour) true)
(check-expect (colour? weak-sum) false)

(define (colour? cards)
  (and (symbol=? (card-colour (hand-c1 cards))
                 (card-colour (hand-c2 cards)))
       (symbol=? (card-colour (hand-c1 cards))
                 (card-colour (hand-c3 cards)))))


;;(strength-sum cards) produces the sum of the strengths of cards
;; strength-sum: Hand -> Nat
;; Example:
(check-expect (strength-sum strong-sum) 23)

(define (strength-sum cards)
  (+ (card-strength (hand-c1 cards))
     (card-strength (hand-c2 cards))
     (card-strength (hand-c3 cards))))
```

```
;;(three-kind? cards) determines if all strengths in cards are the same
;; three-kind?: Hand -> Bool
;; Examples:
(check-expect (three-kind? strong-3kind) true)
(check-expect (three-kind? weak-sum) false)

(define (three-kind? cards)
  (= (card-strength (hand-c1 cards))
     (card-strength (hand-c2 cards))
     (card-strength (hand-c3 cards))))


;;(battle hand1 hand2) determines which of player1 (hand1) and player2 (hand2) wins
;;   in a battle of Schotten Totten, with player1 winning ties
;; battle: Hand Hand -> (anyof 'player1 'player2)
;; Examples:
(check-expect (battle weak-sum strong-colour-run) 'player2)
(check-expect (battle weak-sum weak-sum) 'player1)

(define (battle hand1 hand2)
  (cond
    [(and (colour? hand1) (run? hand1)
          (or (not (and (colour? hand2) (run? hand2)))
              (>= (strength-sum hand1)
                  (strength-sum hand2)))) 'player1]
    [(and (colour? hand2) (run? hand2)) 'player2]
    [(and (three-kind? hand1) (or (not (three-kind? hand2))
                                  (>= (strength-sum hand1)
                                      (strength-sum hand2)))) 'player1]
    [(three-kind? hand2) 'player2]
    [(and (colour? hand1)
          (or (not (colour? hand2))
              (>= (strength-sum hand1)
                  (strength-sum hand2)))) 'player1]
    [(colour? hand2) 'player2]
    [(and (run? hand1)
          (or (not (run? hand2))
              (>= (strength-sum hand1)
                  (strength-sum hand2)))) 'player1]
    [(run? hand2) 'player2]
    [(>= (strength-sum hand1)
         (strength-sum hand2)) 'player1]
    [else 'player2]))

;; Tests:
(check-expect (battle strong-colour-run strong-colour-run) 'player1)
(check-expect (battle strong-colour-run weak-colour-run) 'player1)
(check-expect (battle weak-colour-run strong-colour-run) 'player2)
(check-expect (battle strong-colour-run weak-sum) 'player1)

(check-expect (battle strong-3kind strong-3kind) 'player1)
(check-expect (battle strong-3kind weak-3kind) 'player1)
(check-expect (battle weak-3kind strong-3kind) 'player2)
(check-expect (battle strong-3kind weak-sum) 'player1)
(check-expect (battle weak-sum strong-3kind) 'player2)

(check-expect (battle strong-colour strong-colour) 'player1)
(check-expect (battle strong-colour weak-colour) 'player1)
```

```
(check-expect (battle weak-colour strong-colour) 'player2)
(check-expect (battle strong-colour weak-sum) 'player1)
(check-expect (battle weak-sum strong-colour) 'player2)

(check-expect (battle strong-run strong-run) 'player1)
(check-expect (battle strong-run weak-run) 'player1)
(check-expect (battle weak-run strong-run) 'player2)
(check-expect (battle strong-run weak-sum) 'player1)
(check-expect (battle weak-sum strong-run) 'player2)

(check-expect (battle strong-sum strong-sum) 'player1)
(check-expect (battle strong-sum weak-sum) 'player1)
(check-expect (battle weak-sum strong-sum) 'player2)



;; ************************ Alternate Battle Solution ************************


;; This solution maps the score of a hand to a Nat, where the score is simply
;; the sum of the three strengths, plus an additive "bonus" if the
;; hand satisfies any of the following combinations
(define bonus-run 200)      ;; three cards in sequence
(define bonus-colour 300)   ;; three cards with the same colour
(define bonus-3kind 400)    ;; three cards with the same strength


;;(score cards) determines a "score" for the given hand of cards
;;  (see text and constants above)
;; score: Hand -> Nat
;; Examples:
(check-expect (score weak-sum) 7)
(check-expect (score weak-run) 206)
(check-expect (score strong-colour) 323)
(check-expect (score weak-3kind) 403)
(check-expect (score strong-colour-run) 524)

(define (score cards)
  (+ (strength-sum cards)
     (cond [(run? cards) bonus-run]
           [else 0])
     (cond [(colour? cards) bonus-colour]
           [else 0])
     (cond [(three-kind? cards) bonus-3kind]
           [else 0])))


;;(battle/alt hand1 hand2) determines which of player1 (hand1) and player2 (hand2)
wins
;;   in a battle of Schotten Totten, with player1 winning ties
;; battle/alt: Hand Hand -> (anyof 'player1 'player2)
;; Examples:
(check-expect (battle/alt weak-sum strong-colour-run) 'player2)
(check-expect (battle/alt weak-sum weak-sum) 'player1)

(define (battle/alt hand1 hand2)
  (cond [(>= (score hand1) (score hand2)) 'player1]
        [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
```