```
;; ------------------------------------------------------------
;;  Data definitions

;; A burger is
;;  (make-burger bool bool)
(define-struct burger (cheese? onions?))

;; A side is either
;;  'fries
;;  'onion-rings

;; A simple-order is
;;  - (make-simple-order burger side)
(define-struct simple-order (burger side))

;; A family-order is
;;  - (make-family-order list-of-simple-order)
(define-struct family-order (orders))

;; An order is either
;;  - simple-order
;;  - family-order

;; To remind us, for list-of-order and list-of-simple-order:
;;
;;  A list-of-X is
      either - empty
;;           - (cons X list-of-X)

;; ------------------------------------------------------------
;;  Examples for testing

;  Burger with onions (no cheese), fries on the side
(define burger+f (make-simple-order (make-burger false true) 'fries))
;  Burger with onions (no cheese), onion rings on the side
(define burger+o
(make-simple-order (make-burger false true) 'onion-rings))
;  Burger with cheese and onions, onion rings on the side
(define cheeseburger+o
(make-simple-order (make-burger true true) 'onion-rings))
;  Burger with chese (no onions), fires on the side
(define hold-the-onions
(make-simple-order (make-burger true false) 'fries))

;  An family order with no order inside (family apparently changed its mind)
(define not-hungry (make-family-order empty))
;  Family of three: burger+o, cheeseburger+o, and hold-the-onions
             (make-family-order (list burger+o
                                      cheeseburger+o
(define trio                          hold-the-onions)))
```

```
;  Family of three: hold-the-onions, hold-the-onions, and hold-the-onions
(define trio/hold-the-onions
  (make-family-order (list hold-the-onions
                           hold-the-onions
                           hold-the-onions)))

;; ------------------------------------------------------------
;;  Checking orders

;; Original functions, later abstracted to need-something? and
;   need-something-for-order?:
;
; ;; need-fries? : list-of-order -> bool
; ; Checks whether any order in l includes 'fries
; (define (need-fries? l)
;   (ormap (lambda (o)
;            (need-fries-for-order? o))
;          l))
;
; ;; need-fries-for-order? : order -> bool
; ; Checks whether any order in o includes 'fries
; (define (need-fries-for-order? o)
;   (cond
;     [(simple-order? o) (eq? 'fries (simple-order-side o))]
;     [(family-order? o) (need-fries? (family-order-orders o))]))

;; need-something? : (simple-order -> bool) list-of-order -> bool
;  Return true if CHECK is produces true for every
;  order in l (including each order within each family order)
(define (need-something? CHECK l)
  (ormap (lambda (o)
           (need-something-for-order? CHECK o))
         l))

;; need-something-for-order? : (simple-order -> bool) order -> bool
;  Return true if CHECK is produces true for every
;  order in o (including each order within a family order)
(define (need-something-for-order? CHECK o)
  (cond
    [(simple-order? o) (CHECK o)]
    [(family-order? o) (need-something? CHECK (family-order-orders o))]))

;; Make sure that uses of `need-something?' cover all cases in
;;  both list-of-order and order...

;; need-fries? : list-of-order -> bool
;   Checks whether any order in l includes 'fries
(define (need-fries? l)
  (need-something? (lambda (o) (eq? 'fries (simple-order-side o)))
                   l))

(check-expect (need-fries? empty) false)
(check-expect (need-fries? (list burger+f)) true)
```

```
(check-expect (need-fries? (list burger+o burger+o)) false)
(check-expect (need-fries? (list burger+o trio)) true)
(check-expect (need-fries? (list not-hungry)) false)

;; need-cheese? : list-of-order -> bool
;   Checks whether any order in l includes cheese
(define (need-cheese? l)
  (need-something?
(lambda (o) (burger-cheese? (simple-order-burger o)))
                   l))

(check-expect (need-cheese? empty) false)
(check-expect (need-cheese? (list cheeseburger+o)) true)
(check-expect (need-cheese? (list burger+f burger+o)) false)
(check-expect (need-cheese? (list burger+o trio)) true)
(check-expect (need-cheese? (list not-hungry)) false)

;; need-onions? : list-of-order -> bool
;   Checks whether any order in l includes onions (on burgers
;   or as rings)
(define (need-onions? l)
                   (lambda (o)
                     (or (burger-onions? (simple-order-burger o))
  (need-something?       (eq? 'onion-rings (simple-order-side o))))
                   l))

(check-expect (need-onions? empty) false)
(check-expect (need-onions? (list burger+f)) true)
(check-expect (need-onions? (list hold-the-onions)) false)
(check-expect (need-onions? (list hold-the-onions burger+f)) true)
(check-expect (need-onions? (list trio)) true)
(check-expect (need-onions? (list trio/hold-the-onions)) false)
(check-expect (need-onions? (list not-hungry)) false)

;; ------------------------------------------------------------
;;  Prioritizing orders

;; need-fries-more? : list-of-order -> bool
;;  We need fries more if, no matter how far we look ahead
;;  in the order list, the number of fries we need is never
;;  less than the number of onions that we need.
(define (need-fries-more? l)
  (need-fries-more/given-counts? l 0 0))

;; need-fries-more/given-counts? : list-of-order num num -> bool
;;  Like need-fries-more?, but assumes that we've so far
;;  seen fr orders for fries and on orders for onion rings
;;  (with fr >= or)
(define (need-fries-more/given-counts? l fr on)
  (cond
    [(empty? l) true]
    [else (local [(define n-fr (+ fr (count-sides 'fries (first l))))
                  (define n-on (+ on (count-sides 'onion-rings (first l)))))]
```

```
            (cond
             [(< n-fr n-on) false]
             [else (need-fries-more/given-counts? (rest l) n-fr n-on)]))]))

;; count-sides : sym order -> num
;;  Counts the number of "which" sides ('fries or 'onion-rings) in o
(define (count-sides which o)
  (cond
    [(simple-order? o) (cond
                         [(symbol=? which (simple-order-side o)) 1]
                         [else 0])]
    [else (foldl (lambda (o n)
                   (+ (count-sides which o) n))
                 0
                 (family-order-orders o))]))

(check-expect (count-sides 'fries burger+f) 1)
(check-expect (count-sides 'fries burger+o) 0)
(check-expect (count-sides 'fries trio) 1)
(check-expect (count-sides 'onion-rings trio) 2)

(check-expect (need-fries-more/given-counts? (list burger+f) 0 0) true)
(check-expect (need-fries-more/given-counts? (list burger+o) 0 0) false)
(check-expect (need-fries-more/given-counts? (list burger+o) 1 0) true)
(check-expect (need-fries-more/given-counts? (list burger+f) 1 1) true)
(check-expect (need-fries-more/given-counts? (list burger+f burger+o) 0 0) true)
(check-expect (need-fries-more/given-counts? (list burger+o burger+f) 0 0) false)
(check-expect (need-fries-more/given-counts? (list trio) 0 0) false)
(check-expect (need-fries-more/given-counts? (list trio) 1 0) true)
(check-expect (need-fries-moare/given-counts? (list trio burger+o) 1 0) false)

(check-expect (need-fries-more? (list burger+f)) true)
(check-expect (need-fries-more? (list burger+f burger+o burger+f)) true)
(check-expect (need-fries-more? (list burger+f burger+o burger+o)) false)
(check-expect (need-fries-more? (list trio)) false)
(check-expect (need-fries-more? (list burger+f trio)) true)
```