```
;; -----------------------------------------------------------
;;  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-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
(define trio (make-family-order (list burger+o
                                      cheeseburger+o
                                      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))

(need-fries? empty) "should be" false
(need-fries? (list burger+f)) "should be" true
(need-fries? (list burger+o burger+o)) "should be" false
(need-fries? (list burger+o trio)) "should be" true
(need-fries? (list not-hungry)) "should be" 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))

(need-cheese? empty) "should be" false
(need-cheese? (list cheeseburger+o)) "should be" true
(need-cheese? (list burger+f burger+o)) "should be" false
(need-cheese? (list burger+o trio)) "should be" true
(need-cheese? (list not-hungry)) "should be" false

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

(need-onions? empty) "should be" false
(need-onions? (list burger+f)) "should be" true
(need-onions? (list hold-the-onions)) "should be" false
(need-onions? (list hold-the-onions burger+f)) "should be" true
(need-onions? (list trio)) "should be" true
(need-onions? (list trio/hold-the-onions)) "should be" false
(need-onions? (list not-hungry)) "should be" 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))]))

(count-sides 'fries burger+f) "should be" 1
(count-sides 'fries burger+o) "should be" 0
(count-sides 'fries trio) "should be" 1
(count-sides 'onion-rings trio) "should be" 2

(need-fries-more/given-counts? (list burger+f) 0 0) "should be" true
(need-fries-more/given-counts? (list burger+o) 0 0) "should be" false
(need-fries-more/given-counts? (list burger+o) 1 0) "should be" true
(need-fries-more/given-counts? (list burger+f) 1 1) "should be" true
(need-fries-more/given-counts? (list burger+f burger+o) 0 0) "should be" true
(need-fries-more/given-counts? (list burger+o burger+f) 0 0) "should be" false
(need-fries-more/given-counts? (list trio) 0 0) "should be" false
(need-fries-more/given-counts? (list trio) 1 0) "should be" true
(need-fries-more/given-counts? (list trio burger+o) 1 0) "should be" false

(need-fries-more? (list burger+f)) "should be" true
(need-fries-more? (list burger+f burger+o burger+f)) "should be" true
(need-fries-more? (list burger+f burger+o burger+o)) "should be" false
(need-fries-more? (list trio)) "should be" false
(need-fries-more? (list burger+f trio)) "should be" true

;; -----------------------------------------------------------
;;  State

;; ORDERS : list-of-order
(define ORDERS empty)

;; FAMILY-ORDER : list-of-simple-order
(define FAMILY-ORDERS empty)

;; add-simple-order! : burger side -> void
;;  Add an order for a burger and side to the end of the order list
;;  Effect: sets ORDERS to the new order list
(define (add-simple-order! b s)
  (set! ORDERS (append ORDERS (list (make-simple-order b s)))))

(set! ORDERS empty)
(add-simple-order! (make-burger true true) 'fries) "should be" (void)
ORDERS "should be" (list burger+f)
(add-simple-order! (make-burger true false) 'onion-rings) "should be" (void)
                   (list
                    (make-simple-order (make-burger true true) 'fries)
ORDERS "should be"  (make-simple-order (make-burger true false) 'onion-rings))

;; add-family-order! : burger side drink -> void
;;  Add an order for a burger and side to the end of the current
;;   family order list
;;  Effect: sets FAMILY-ORDERS to the new order list
(define (add-family-order! b s)
  (set! FAMILY-ORDERS (append FAMILY-ORDERS (list (make-simple-order b s)))))
```

```
(set! FAMILY-ORDERS empty)
(add-family-order! (make-burger true true) 'fries) "should be" (void)
FAMILY-ORDERS "should be" (list
                           (make-simple-order (make-burger true true) 'fries))
(add-family-order! (make-burger true false) 'onion-rings) "should be" (void)
FAMILY-ORDERS "should be" (list
                           (make-simple-order (make-burger true true) 'fries)
                           (make-simple-order (make-burger true false) 'onion-rings))

;; family-order-complete! : -> void
;;  Moves the current family order into the main order list
;;   Effect: add a family order to ORDERS, resets FAMILY-ORDERS to empty
(define (family-order-complete!)
  (begin
    (set! ORDERS (cons (make-family-order FAMILY-ORDERS)
                       ORDERS))
    (set! FAMILY-ORDERS empty)))

(set! ORDERS empty)
(set! FAMILY-ORDERS (list (make-simple-order (make-burger true false) 'onion-rings)
                          (make-simple-order (make-burger true true) 'fries)))
(family-order-complete!) "should be" void
ORDERS "should be" (list (make-family-order
                          (list (make-simple-order (make-burger true false) 'onion-rings)
                                (make-simple-order (make-burger true true) 'fries))))
FAMILY-ORDERS "should be" empty
```