```
; a list-of-numbers is either
;   empty,
; or a structure
;   (cons first rest)
; where first is a number and rest is a list-of-numbers
;
; We will use Scheme's builtin list facility (cons, first, rest, and empty)

; Template for list
; (define (ListProg fee)
;    (cond
;       ((empty? fee)  ...)
;       (else
;            ... (first fee) ...
;            ... (ListProg (rest fee)) ... )
;     ))


; Problem 1

; Contract, purpose, & header
;
; List-search : list number -> Boolean         [Boolean means a true or
false value]
; Purpose: returns true if the input number occurs in the input lise
;          returns false otherwise
; (define (List-search TheList  TheNumber) ...)

; Test data
;
;  Input                        | Result
; ---------------------------------------
;  empty 12                     | false
;  (cons 12 empty)  12          | true
;  (cons 1 (cons 12 empty) 12   | true

; Final code
; List-search : list number -> Boolean         [Boolean means a true or
false value]
; Purpose: returns true if the input number occurs in the input lise
;          returns false otherwise
(define (List-search TheList  TheNumber)
  (cond
    ((empty? TheList) false)
    (else   ; could also be (cons? TheList)
        (cond
```

```
           ((= (first TheList) TheNumber) true)
           (else (List-search (rest TheList) TheNumber)))
        )
    ))

; Of course, this problem admits to several alternate solutions

; The nested cond expressions can be combined and simplified...

; List-search : list number -> Boolean         [Boolean means a true or
false value]
; Purpose: returns true if the input number occurs in the input lise
;          returns false otherwise
(define (List-search-alt1 TheList  TheNumber)
  (cond
    ((empty? TheList) false)
    ((= (first TheList) TheNumber) true)
    (else (List-search (rest TheList) TheNumber))
  ))


; The simplest combines the answer with a boolean or function to avoid the
inner cond
; List-search : list number -> Boolean         [Boolean means a true or
false value]
; Purpose: returns true if the input number occurs in the input lise
;          returns false otherwise
(define (List-search-alt2 TheList  TheNumber)
  (cond
    ((empty? TheList) false)
    (else   ; could also be (cons? TheList)
        (or
           (= (first TheList) TheNumber)
           (List-search (rest TheList) TheNumber)))
    ))


; Problem 2 -- same data analysis and template

; Contract, purpose, & header
; List-large : list-of-numbers -> number      [Problem statement is
actually wrong in the
;                                             assignment ... has an extra
argument to List-large]
; Purpose: count list elements that are numerically greater than or equal
to 100
```

```
; (define (List-large fie) ... )


; Test data
;
; Input                              |  Result
; -----------------------------------------
; empty                              |  0
; (cons 100 (cons 200 empty))        |  2
; (cons 1   (cons 101 empty))        |  1
; (cons 1   (cons 2   empty))        |  0

; Final Code
;
; Contract, purpose, & header
; List-large : list-of-numbers -> number      [Problem statement is
actually wrong in the
;                                              assignment ... has an extra
argument to List-large]
; Purpose: count list elements that are numerically greater than or equal
to 100
(define (List-large fie)
   (cond
     ((empty? fie) 0)
     (else
        (cond
          ((<= 100 (first fie)) (+ 1 (List-large (rest fie))))
          (else (List-large (rest fie)))
          ))
     ))


; Of course, you could implement the body of this function in several
different ways

; The simplest change is just a recognition that the cond clauses evaluate
in order,
; so that you don't need to nest the cond clauses...

; List-large : list-of-numbers -> number      [Problem statement is
actually wrong in the
;                                              assignment ... has an extra
argument to List-large]
; Purpose: count list elements that are numerically greater than or equal
to 100
(define (List-large-alt1 fie)
```

```
  (cond
     ((empty? fie) 0)
     ((<= 100 (first fie)) (+ 1 (List-large (rest fie))))
     (else (List-large-alt1 (rest fie)))
     ))

; The second is a more elegant solution that produces a much easier program
to read.
; It requires a helper function, which we will call BigNumberFilter

; BigNumberFilter : number -> number
; Purpose: if the input number is larger than 100, it returns one
;          Otherwise, it returns zero
(define (BigNumberFilter N)
  (cond
    ((> 100 N) 0)
    (else 1)
    ))


; Given BigNumberFilter, List-large is trivial to write
; List-large : list-of-numbers -> number      [Problem statement is
actually wrong in the
;                                              assignment ... has an extra
argument to List-large]
; Purpose: count list elements that are numerically greater than or equal
to 100
(define (List-large-alt2 fie)
  (cond
     ((empty? fie) 0)
     (else (+ (BigNumberFilter (first fie))
              (List-large-alt2 (rest fie))))
     ))
```

Welcome to DrScheme, version 208p1.
Language: Beginning Student.

```
> (List-search empty 12)
false
> (List-search (cons 12 empty) 12)
true
> (List-search (cons 1 (cons 12 empty)) 12)
true
> (List-search (cons 1 (cons 2 empty) 12))
List-search: this procedure expects 2 arguments, here it is provided 1
argument
> (List-search (cons 1 (cons 2 empty)) 12)
false
>
```

Welcome to DrScheme, version 208p1.
Language: Beginning Student.

```
> (List-large empty)
0
> (List-large (cons 100 (cons 200 empty)))
2
> (List-large (cons 1   (cons 101 empty)))
1
> (List-large (cons 1   (cons 2   empty)))
0
>
```