

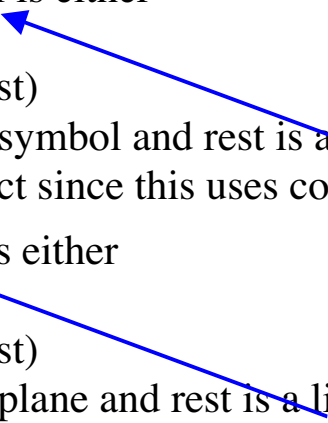
```
;; a brand is a
;; (make-brand type speed seats service)
;; where type is a symbol, and speed, seats, and service
;;     are numbers
(define-struct brand (type speed seats service))

;; a plane is a
;; (make-plane tailnum kind miles mechanic)
;; where tailnum is a symbol, kind is a brand
;;     miles is a number, and mechanic is a list-of-symbols
(define-struct plane (tailnum kind miles mechanic))

;; a list-of-symbol is either
;; - empty, or
;; - (cons first rest)
;; where first is a symbol and rest is a list-of-symbol
;; [No define-struct since this uses cons, first, and rest]

;; a list-of-plane is either
;; - empty, or
;; - (cons first rest)
;; where first is a plane and rest is a list-of-plane
;; [No define-struct since this uses cons, first, and rest]

;; Example Data
;;
(define brand1 (make-brand 'DC-10 550 282 15000))
(define brand2 (make-brand 'MD-80 505 141 10000))
(define brand3 (make-brand 'ATR-72 300 46 5000))
```



```
;; And some planes
(define N1701 (make-plane 'N1701 brand1 0 empty))
(define N3217 (make-plane 'N3217 brand2 100
                          (cons 'Susan (cons 'Mike (cons 'Pam empty))))
              ))
(define N1079 (make-plane 'N1079 brand1 3500
                          (cons 'Mike (cons 'Bubba (cons 'Susan empty))))
              ))
(define N9824 (make-plane 'N9824 brand3 500
                          (cons 'Bess (cons 'Felix (cons 'Jane empty))))
              ))
(define N3141 (make-plane 'N3141 brand3 1000
                          (cons 'Fred (cons 'Bubba empty)))
              ))

;;
(define JetSetFleet
  (cons N1701
        (cons N3217
              (cons N1079
                    (cons N9824
                          (cons N3141 empty)))))))
```

;; just-DC-10s: list-of-plane -> number

;; Purpose: consumes a list-of-plane and returns the number that

;; are DC-10s

(define (count-DC-10s a-lop)

(cond

[(empty? a-lop) 0]

[(cons? a-lop)

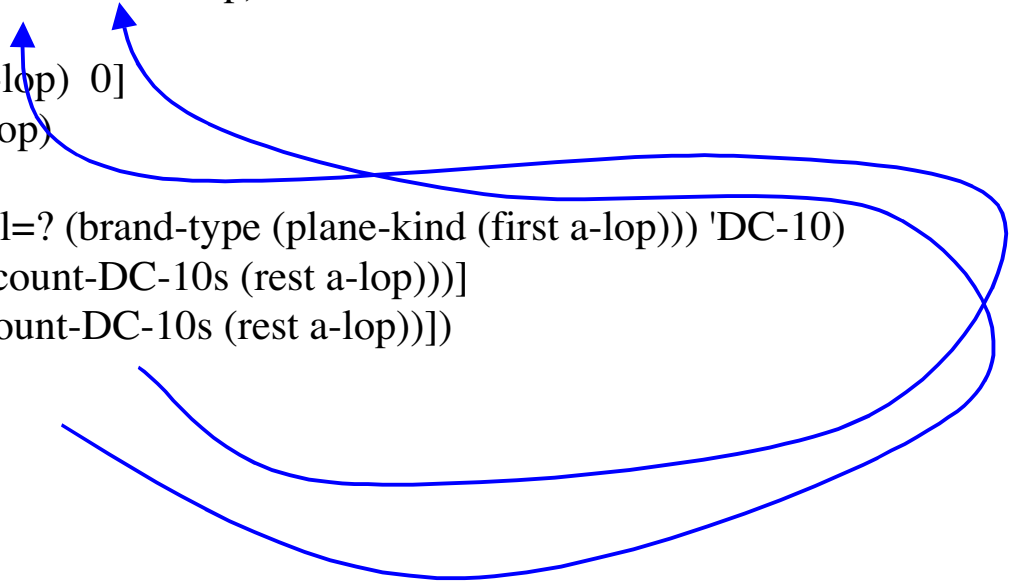
(cond

[(symbol=? (brand-type (plane-kind (first a-lop))) 'DC-10)

(add1 (count-DC-10s (rest a-lop)))]

[else (count-DC-10s (rest a-lop))])

]))



;; just-DC-10s: list-of-plane -> number

;; Purpose: consumes a list-of-plane and returns the number that

;; are DC-10s

(define (count-DC-10s a-lop)

(cond

[(empty? a-lop) 0]

[(cons? a-lop)

(add

(cond

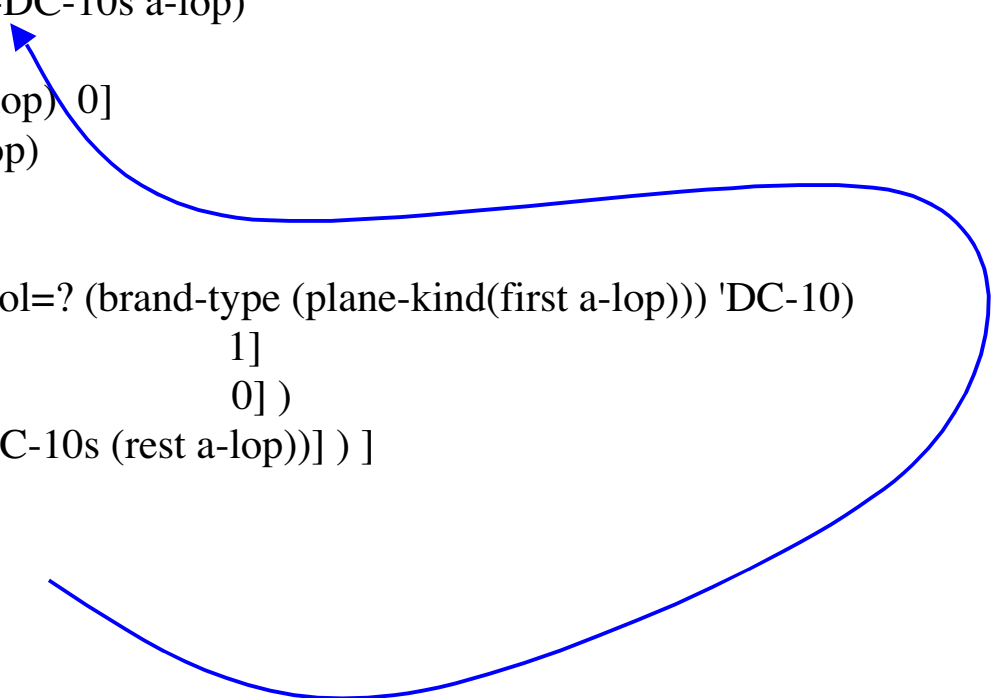
[(symbol=? (brand-type (plane-kind(first a-lop))) 'DC-10)

1]

[else 0])

(count-DC-10s (rest a-lop))])]

))



```
;; A cleaner formulation that uses a helper function because
;; we are going to access two distinct kinds of data (planes
;; and brands)
```

```
;;
;; First, the helper function
```

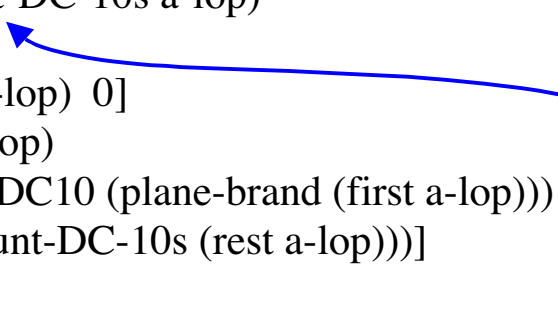
```
;; Is-DC10: brand → number
;; Purpose: consumes a brand and returns 1 if the brand's
;;          type is DC10 and returns 0 otherwise
```

```
(define (Is-DC10 a-brand)
  (cond
    [(symbol=? (brand-type a-brand) `DC10) 1]
    [else 0]
  ))
```

```
;; and, the desired program
```

```
;; just-DC-10s: list-of-plane -> number
;; Purpose: consumes a list-of-plane and returns the
;;          number that are DC-10s
```

```
(define (count-DC-10s a-lop)
  (cond
    [(empty? a-lop) 0]
    [(cons? a-lop)
     (add (Is-DC10 (plane-brand (first a-lop)))
           (count-DC-10s (rest a-lop)))]
  ))
```



```
:: Templates
```

```
:: for brand
```

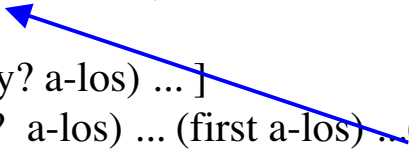
```
:: (define ( ... a-brand ... )  
;; ( ... (brand-type a-brand) ...  
;; ... (brand-speed a-brand) ...  
;; ... (brand-seats a-brand) ...  
;; ... (brand-service a-brand) ... ))
```

```
:: for plane
```

```
:: (define ( ... a-plane ... )  
;; ( ... (plane-tailnum a-plane) ...  
;; ... (plane-kind a-plane) ...  
;; ... (plane-miles a-plane) ...  
;; ... (plane-mechanics a-plane) ... ))
```

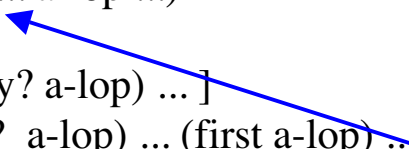
```
:: for list-of-symbols
```

```
:: (define (f... a-los ... )  
;; (cond  
;; [(empty? a-los) ... ]  
;; [(cons? a-los) ... (first a-los) ... (f (rest a-los) ... )... ]  
;; ))
```



```
:: for list-of-planes
```

```
:: (define (g ... a-lop ... )  
;; (cond  
;; [(empty? a-lop) ... ]  
;; [(cons? a-lop) ... (first a-lop) ... (g (rest a-lop) ... ) ... ]  
;; ))
```



Design Methodology

1. Data Analysis -- determine how many pieces of data describe interesting aspects of a typical object mentioned in the problem statement; add a data definitions for each kind ("class") of object in the problem
2. Contract, Purpose, and Header -- write the basic documentation for the program
3. Test Cases -- develop several test cases. Be sure to pick end conditions on intervals, unusual values, along with a couple that are normal values with obvious answers that you can easily check. (*e.g.*, (area 10) = 314.15 mumble)
4. Templates -- write the templates for each argument that is a compound object. The template must contain the various selector functions defined for the object. It serves as a reminder of what kinds of information we have available. For recursive structures, indicate the recursive call by drawing an arrow from the call back to the header.
5. Develop the body -- use your knowledge of the problem, the data, and Scheme to fill in the body of the program. This may involve developing helper functions (especially if the problem involves a compound object that contains another compound object).
6. Test your program -- either hand evaluate your code on the test cases from step 3, or use DrScheme to evaluate them.