```
;; Sample solution for A05

(require "namelist.rkt")

;-----------------------------------------------------------------------
; Q1: data definitions
;-----------------------------------------------------------------------

; A Namelist is one of:
; * empty
; * (cons Nameinfo Namelist)

; my-namelist-fn: Namelist -> Any
; (define (my-namelist-fn names)
;    (cond [(empty? names) ...]
;          [(cons? names) ... (my-nameinfo-fn (first names)) ...
;                         ... (my-namelist-fn (rest names))...]))

; my-nameinfo-fn: Nameinfo -> Any
; (define (my-nameinfo-fn info)
;    ... (nameinfo-name info)...
;    ... (nameinfo-decade info)...
;    ... (nameinfo-rank info)...
;    ... (nameinfo-gender info)...
; )

; Note:  A less desireable solution is to combine the two
; templates:
; (define (my-namelist-fn names)
;    (cond [(empty? names) ...]
;          [(cons? names)
;           ... (nameinfo-name (first names)) ...
;           ... (nameinfo-decade (first names)) ...
;           ... (nameinfo-rank (first names)) ...
;           ... (nameinfo-gender (first names)) ...
;           ... (my-namelist-fn (rest names))...]))
;
; Even though it's less desireable, this approach is
; presented in the lecture notes and should receive
; full marks.



; A short example list used for testing.
(define aNameList
  (list (make-nameinfo "Lydia" 2000 3 'Female)
        (make-nameinfo "John" 2000 1 'Male)
        (make-nameinfo "John" 1990 2 'Male)))


;-----------------------------------------------------------------------
; Q2: find-rank
;-----------------------------------------------------------------------

; find-rank: Namelist String Symbol Nat -> (union Nat false)
; Purpose: Find the popularity ranking of name for the given gender
; in the given decade; false if can't be found.
; Examples:
```

```
(check-expect (find-rank empty "John" 'Male 2000) false)
(check-expect (find-rank aNameList "John" 'Male 2000) 1)

(define (find-rank names name gender decade)
  (cond [(empty? names) false]
        [(and (cons? names)
              (= decade (nameinfo-decade (first names)))
              (symbol=? gender (nameinfo-gender (first names)))
              (string=? name (nameinfo-name (first names))))
         (nameinfo-rank (first names))]
        [(cons? names) (find-rank (rest names) name gender decade)]
        [else (error "find-rank: expected first argument to be a namelist")]))

;Note: You are allowed to assume that inputs into the function
;will be of the correct data type.

; Tests:
(check-expect (find-rank aNameList "John" 'Male 1990) 2)
(check-expect (find-rank aNameList "Lydia" 'Female 2000) 3)
(check-error (find-rank 3 "John" 'Male 1990)
             "find-rank: expected first argument to be a namelist")
(check-expect (find-rank aNameList "John" 'Male 2010) false)
(check-expect (find-rank aNameList "Curtis" 'Male 1990) false)
;-----------------------------------------------------------------------
; Q3: collect-name
;-----------------------------------------------------------------------

; collect-name: Namelist String Symbol --> Namelist
; Purpose: Collect all the structures from names that match
; name and gender.
; Examples:
(check-expect (collect-name aNameList "John" 'Male)
              (list (make-nameinfo "John" 2000 1  'Male)
                    (make-nameinfo "John" 1990 2 'Male)))

(define (collect-name names name gender)
  (cond [(empty? names) empty]
        [(and (cons? names)
              (symbol=? gender (nameinfo-gender (first names)))
              (string=? name (nameinfo-name (first names))))
         (cons (first names)
               (collect-name (rest names) name gender))]
        [(cons? names) (collect-name (rest names) name gender)]
        [else (error "Argument not a list.")]))

;Note: You are allowed to assume that inputs into the function
;will be of the correct data type.

; Tests:
(check-expect (collect-name empty "John" 'Male) empty)
(check-expect (collect-name aNameList "Lydia" 'Female)
              (list (make-nameinfo "Lydia" 2000 3  'Female)))
(check-error (collect-name 1 "John" 'Male) "Argument not a list.")



;-----------------------------------------------------------------------
; Q4: first-n
```

```
;-----------------------------------------------------------------

;; enough?: (listof Any) Nat -> Boolean
;; Purpose:  produces true if there at least n elements in the list lst,
;; and false otherwise
;; Examples:
(check-expect (enough? (list 1 2) 2) true)
(check-expect (enough? (list 1 2 3) 4) false)
(define (enough? lst n)
  (cond
    [(empty? lst) (= n 0)]
    [(= n 0) true]
    [else (enough? (rest lst) (sub1 n))]))
;; Tests
(check-expect (enough? empty 0) true)
(check-expect (enough? empty 1) false)
(check-expect (enough? (list 1 2 3 4) 2) true)

;; extract: (listof Any) Nat --> (listof Any)
;; Purpose: Produce the first n items on the list lst,
;; knowing that there are at least n items in the list.
;; Examples:
(check-expect (extract (list 1 2 3) 2) (list 1 2))
(define (extract lst n)
  (cond
    [(= n 0) empty]
    [else (cons (first lst) (extract (rest lst) (sub1 n)))]))
;; Tests:
(check-expect (extract empty 0) empty)
(check-expect (extract (list 1 2 3) 1)
              (list 1))

;; first-n: (listof Any) Nat --> (listof Any)
;; Purpose: Produce the first n items on the list lst if there
;; are enough items in the list, and 'NotEnoughItems otherwise
; Examples:
(check-expect (first-n (list 1 2 3) 2)
              (list 1 2))
(define (first-n lst n)
  (cond
    [(enough? lst n) (extract lst n)]
    [else 'NotEnoughItems]))

;Note: You are allowed to assume that inputs into the function
;will be of the correct data type.
; Tests:
(check-expect (first-n empty 0) empty)
(check-expect (first-n empty 10) 'NotEnoughItems)
(check-expect (first-n (list 1 2 3) 1)
              (list 1))




;-----------------------------------------------------------------
; Q5: name-sort
;-----------------------------------------------------------------
; name-sort: Namelist -> Namelist
```

```
; Purpose: Sort names so they are increasing order by name and decade.
; Example:
(check-expect (name-sort aNameList)
              (list (make-nameinfo "John" 1990 2 'Male)
                    (make-nameinfo "John" 2000 1  'Male)
                    (make-nameinfo "Lydia" 2000 3 'Female)))
(define (name-sort lst)
  (cond [(empty? lst) empty]
        [(cons? lst) (name-insert (first lst) (name-sort (rest lst)))]))

; Tests:
(check-expect (name-sort
               (list (make-nameinfo "John" 1990 2 'Male)
                     (make-nameinfo "John" 2000 1 'Male)
                     (make-nameinfo "Lydia" 2000 3 'Female)))
              (list (make-nameinfo "John" 1990 2 'Male)
                    (make-nameinfo "John" 2000 1 'Male)
                    (make-nameinfo "Lydia" 2000 3 'Female)))
(check-expect (name-sort empty) empty)

; Helper function:
; name<?:  Nameinfo Nameinfo -> Boolean
; Purpose: Return true if info1 is ordered before info2; false otherwise.
; Sort order:  name, gender ('Female < 'Male), decade
(check-expect (name<? (make-nameinfo "John" 1990 2 'Male)
                      (make-nameinfo "Andy" 1990 2 'Male)) false)
(check-expect (name<? (make-nameinfo "Andy" 1990 2 'Male)
                      (make-nameinfo "John" 1990 2 'Male)) true)

(define (name<? info1 info2)
  (or (string<? (nameinfo-name info1)
                (nameinfo-name info2))
      (and (string=? (nameinfo-name info1)
                     (nameinfo-name info2))
           (or (and (symbol=? (nameinfo-gender info1) 'Female)
                    (symbol=? (nameinfo-gender info2) 'Male))
               (and (symbol=? (nameinfo-gender info1)
                              (nameinfo-gender info2))

                    (< (nameinfo-decade info1)
                       (nameinfo-decade info2)))))))

; Additional tests
(check-expect (name<? (make-nameinfo "Pat" 1990 2 'Male)
                      (make-nameinfo "Pat" 1990 2 'Female)) false)
(check-expect (name<? (make-nameinfo "Pat" 1990 2 'Female)
                      (make-nameinfo "Pat" 1990 2 'Male)) true)
(check-expect (name<? (make-nameinfo "Pat" 1980 2 'Female)
                      (make-nameinfo "Pat" 1990 2 'Female)) true)
(check-expect (name<? (make-nameinfo "Pat" 2000 2 'Female)
                      (make-nameinfo "Pat" 1990 2 'Female)) false)

; Helper function:
; name-insert: Nameinfo Namelist -> Namelist
; Purpose: Insert name into the sorted Namelist names such that the
; resulting list is also sorted.  Ordering determined by name, gender,
; decade.
; Example:
```

```
(check-expect (name-insert (make-nameinfo "John" 2000 1  'Male)
                           (list (make-nameinfo "John" 1990 2  'Male)
                                 (make-nameinfo "Lydia" 2000 3  'Female)))
              (list (make-nameinfo "John" 1990 2  'Male)
                    (make-nameinfo "John" 2000 1  'Male)
                    (make-nameinfo "Lydia" 2000 3  'Female)))

(define (name-insert name names)
  (cond [(empty? names) (cons name empty)]
        [(name<? name (first names))  (cons name names)]
        [else  (cons (first names)
                     (name-insert name (rest names)))]))

; Tests:
(check-expect (name-insert (make-nameinfo "John" 1990 2 'Male)
                           (list (make-nameinfo "John" 2000 1  'Male)
                                 (make-nameinfo "Lydia" 2000 3  'Female)))
              (list (make-nameinfo "John" 1990 2  'Male)
                    (make-nameinfo "John" 2000 1  'Male)
                    (make-nameinfo "Lydia" 2000 3  'Female)))
(check-expect (name-insert (make-nameinfo "Lydia" 2000 3  'Female)
                           (list (make-nameinfo "John" 1990 2  'Male)
                                 (make-nameinfo "John" 2000 1  'Male)))
              (list (make-nameinfo "John" 1990 2  'Male)
                    (make-nameinfo "John" 2000 1  'Male)
                    (make-nameinfo "Lydia" 2000 3  'Female)))
(check-expect (name-insert (make-nameinfo "John" 1990 2  'Male) empty)
              (list (make-nameinfo "John" 1990 2  'Male)))

;---------------------------------------------------------------------
; Q6: bar-graph
;---------------------------------------------------------------------

; bar-graph: (listof Nat) -> String
; Produce a bar graph with one bar of "*" characters for each number of lon.
; Examples:
(check-expect (bar-graph (list 2 5 3))
              "**\n*****\n***\n")

(define (bar-graph lon) (list->string (bg-helper lon)))

; Tests:
(check-expect (bar-graph empty)        "")
(check-expect (bar-graph (list 0))     "\n")
(check-expect (bar-graph (list 2))     "**\n")
(check-expect (bar-graph (list 0 2))   "\n**\n")
(check-expect (bar-graph (list 3 2))   "***\n**\n")

; bg-helper:  (listof Nat) -> (listof Char)
; Produce a bar graph as a list of characters (instead of a string).
(check-expect (bg-helper (list 1 2)) (list #\* #\newline #\* #\* #\newline))
(define (bg-helper lon)
  (cond [(empty? lon) empty]
        [(<= (first lon) 0) (cons #\newline (bg-helper (rest lon)))]
        [else (cons #\* (bg-helper (cons (sub1 (first lon))
                                         (rest lon))))]))
; bar-graph and bg-helper are so closely related that the tests for
; bar-graph completely test bg-helper.  Thus there are no additional
```

```
; tests for bg-helper.



;-----------------------------------------------------------------------
; Q7: name-popularity-graph:
;-----------------------------------------------------------------------

; name-popularity-graph: Namelist String Symbol -> String
; Purpose: Produce a bar graph of the popularity of name in nlst with one
; bar per decade.
;Examples:
(check-expect (name-popularity-graph
               (list (make-nameinfo "Test" 2000 1 'Male)
                     (make-nameinfo "Test" 1990 900 'Male)
                     (make-nameinfo "Test" 1980 1000 'Male))
               "Test" 'Male)

"\n\n\n\n\n\n\n\n\n*\n*******\n**************************************************
**************\n")

(define (name-popularity-graph nlst name gender)
  (bar-graph (scale-ranks
              (extract-ranks
               (insert-missing-decades
                1890 2000
                (name-sort (collect-name nlst name gender)))))))

; Tests:
(check-expect (name-popularity-graph empty "Test" 'Male)
              "\n\n\n\n\n\n\n\n\n\n\n\n")
(check-expect (name-popularity-graph
               (list (make-nameinfo "Test" 2000 1 'Male)
                     (make-nameinfo "Test" 1920 900 'Male)
                     (make-nameinfo "Test" 1980 1000 'Male))
               "Test" 'Male)

"\n\n\n*******\n\n\n\n\n\n*\n\n**************************************************
**************\n")
(check-expect (name-popularity-graph aNameList "John" 'Male)

"\n\n\n\n\n\n\n\n\n\n**************************************************************
****\n**********************************************************\n")

; Helper functions:
; extract-ranks: Namelist -> (listof Nat)
; Extract the rank of each nameinfo record in names.
; Examples:
(check-expect (extract-ranks aNameList) (list 3 1 2))

(define (extract-ranks names)
  (cond [(empty? names) empty]
        [(cons? names) (cons (nameinfo-rank (first names))
                             (extract-ranks (rest names)))]))

; Tests:
(check-expect (extract-ranks
               (list (make-nameinfo "Test" 2000 1 'Male)
```

```
                    (make-nameinfo "Test" 1920 900 'Male)
                    (make-nameinfo "Test" 1980 1000 'Male)))
              (list 1 900 1000))

; scale-ranks: (listof Nat) -> (listof Nat)
; Scale the numbers on lon using the formula given in the assignment.
; Example:
(check-expect (scale-ranks (list 1000 900 1))
              (list 1 7 67 ))

(define (scale-ranks lon)
  (cond [(empty? lon) empty]
        [(cons? lon) (cons (- 67 (floor (/ (first lon) 15)))
                           (scale-ranks (rest lon)))]))

; Tests:
(check-expect (scale-ranks (list 5000 5))
              (list -266 67))
(check-expect (scale-ranks empty) empty)


;-----------------------------------------------------------------------
; Bonus
;-----------------------------------------------------------------------

; name-popularity-graph2: Namelist String Symbol -> String
; Purpose: Produce a bar graph of the popularity of name in nlst with one
; bar per decade.
;Examples:
(check-expect (name-popularity-graph2
               (cons (make-nameinfo "Test" 2000 1 'Male)
                     (cons (make-nameinfo "Test" 1990 900 'Male)
                           (cons (make-nameinfo "Test" 1980 1000 'Male)
                                 empty)))
               "Test" 'Male)
              "1890 \n1900 \n1910 \n1920 \n1930 \n1940 \n1950 \n1960 \n1970 \n1980
*\n1990 *******\n2000
*****************************************************\n")

(define (name-popularity-graph2 nlst name gender)
  (bar-graph-with-decades
   1890
   (scale-ranks
               (extract-ranks
                (insert-missing-decades
                 1890 2000
                 (name-sort (collect-name nlst name gender)))))))

; Tests:
(check-expect (name-popularity-graph2 empty "Test" 'Male)
              "1890 \n1900 \n1910 \n1920 \n1930 \n1940 \n1950 \n1960 \n1970 \n1980
\n1990 \n2000 \n")
(check-expect (name-popularity-graph2
               (cons (make-nameinfo "Test" 2000 1 'Male)
                     (cons (make-nameinfo "Test" 1920 900 'Male)
                           (cons (make-nameinfo "Test" 1980 1000 'Male)
                                 empty)))
               "Test" 'Male)
```

```
              "1890 \n1900 \n1910 \n1920 *******\n1930 \n1940 \n1950 \n1960 \n1970
\n1980 *\n1990 \n2000
**********************************************************************\n")
(check-expect (name-popularity-graph2 aNameList "John" 'Male)
              "1890 \n1900 \n1910 \n1920 \n1930 \n1940 \n1950 \n1960 \n1970 \n1980
\n1990 **********************************************************************\n2000
**********************************************************************\n")



; bar-graph-with-decades: Nat (listof Nat) --> String
; dec labeling the first bar of the bar graph having one bar
; for each item on lon, the number representing the length of the bar.
; dec (the decade) is incremented by 10 for each bar.
(check-expect (bar-graph-with-decades 1890 (list 2 3))
              "1890 **\n1900 ***\n")
(define (bar-graph-with-decades dec lon) (list->string (bgwd-helper dec lon)))

; bgwd-helper:  (listof Nat) -> (listof Char)
; Produce a bar graph as a list of characters (instead of a string).
; Otherwise, identical to bar-graph-with-decades
(define (bgwd-helper dec lon)
  (cond [(empty? lon) empty]
        [else
         (myappend (string->list (number->string dec))
                   (cons #\space
                         (myappend (bar (first lon))
                                   (bgwd-helper (+ 10 dec) (rest lon)))))]))

; bar-graph-with-decades and bgwd-helper are so closely related
; that the tests for bar-graph-with-decades completely test bgwd-helper.
; Thus there are no additional tests for bg-helper.


; myappend:  (listof Any) (listof Any) --> (listof Any)
; Return a single list composed of the elements of s1 followed
; by the elements of s2.
(check-expect (myappend (list 1 2 3) (list 4 5)) (list 1 2 3 4 5))
(define (myappend s1 s2)
  (cond [(empty? s1) s2]
        [else (cons (first s1)
                    (myappend (rest s1) s2))]))



; Helper functions:
; bar: Nat -> (listof Char)
; Purpose: Return a list of n "*" characters.
; Examples:
(check-expect (bar 5)
              (list #\* #\* #\* #\* #\* #\newline))

(define (bar n)
  (cond [(<= n 0) (cons #\newline empty)]
        [else (cons #\* (bar (sub1 n)))]))
```

```
                    (myappend (rest s1) s2))]))


; Helper functions:
; bar: Nat -> (listof Char)
; Purpose: Return a list of n "*" characters.
; Examples:
(check-expect (bar 5)
              (list #\* #\* #\* #\* #\* #\newline))

(define (bar n)
  (cond [(<= n 0) (cons #\newline empty)]
        [else (cons #\* (bar (sub1 n)))]))
```