Alternate templates leading to the second solution

;; salary-rec-template: Payroll → Any
(define (salary-rec-template sr) (\ldots (name sr) \ldots (amount sr) \ldots ))

;; payroll-template pr

;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) \ldots ]
        [(cons? pr) (\ldots (salary-rec-template (first pr)) \ldots (payroll-template (rest pr)) \ldots )]))
Different kinds of lists

When we introduced lists in module 05, the items they contained were not lists. These were flat lists.

We have just seen lists of lists. A Payroll is a list containing a two-element flat list.

In later lecture modules, we will use lists containing unbounded flat lists.

We will also see nested lists, in which lists may contain lists that contain lists, and so on to an arbitrary depth.
Dictionaries

Once upon a time, a dictionary was a book in which you look up a word to find a definition. Nowadays, a dictionary is an app:

```
half of the words in his text were not in the dictionary: LEXICON, wordbook, glossary, vocabulary list, vocabulary, word list, wordfinder.
```
More generally, a dictionary contains a number of **keys**, each with an associated **value**.

Examples:

- Your contacts list. Keys are names, and values are telephone numbers.
- Your seat assignment for midterms. Keys are userids, and values are seat locations.
- Stock symbols (keys) and prices (values).

Many two-column tables can be viewed as dictionaries. The previous examples can all be viewed as two-column tables.
Dictionary operations

What operations might we wish to perform on dictionaries?

- **lookup**: given a key, produce the corresponding value
- **add**: add a (key, value) pair to the dictionary
- **remove**: given a key, remove it and its associated value
Association lists

One simple solution uses an association list, which is a list of (key, value) pairs.

We store the pair as a two-element list. For simplicity, we will use numbers as keys and strings as values.

;; An association list (AL) is one of:
;; ⋆ empty
;; ⋆ (cons (list Num Str) AL)
Association lists: an example

We can create association lists based on other types for keys and values. We use Nat and Str here just to provide a concrete example or an association list where given a student number you can look up the student’s name.

**Key Point:** We impose the additional restriction that an association list contains *at most one occurrence of any key.*

Since we have a data definition, we could use AL for the type of an association list, as given in a contract.

Another alternative is to use `(listof (list Nat Str))`. 
Association lists: template

We can use the data definition to produce a template.

;;; al-template: AL → Any
(define (al-template alst)
  (cond [(empty? alst) . . . ]
        [else (. . . (first (first alst)) . . . ; first key
                    (second (first alst)) . . . ; first value
                    (al-template (rest alst)))]))
Association lists: lookup operation

Recall that lookup consumes a key and a dictionary (association list) and produces the corresponding value.

In coding lookup, we have to make a decision. What should it produce if the lookup fails?

Since all valid values are strings, it can produce false to indicate that the key was not present in the association list.

(check-expect (lookup 3 (list 1 "John") (list 3 "Winnie")) "Winnie")
(check-expect (lookup 2 (list 1 "John") (list 3 "Winnie")) false)
Association lists: lookup-al implementation

;; (lookup-al k alst) produces the value corresponding to key k,
;; or false if k not present
;; lookup-al:

(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(= k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))])))
Association lists: further tasks

We will leave the `add` and `remove` functions as exercises.

This solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient in the case where the key is not present and the whole list must be searched.

In a future module, we will impose structure to improve this situation.
Two-dimensional data

Key Idea: Another use of lists of lists is to represent a two-dimensional table (or a matrix).

For example, here is a multiplication table:

\[(\text{mult-table } 3 4) \Rightarrow\]

\[(\text{list } (\text{list } 0 0 0 0)\]

\[(\text{list } 0 1 2 3)\]

\[(\text{list } 0 2 4 6))\]

The \(c^{th}\) entry of the \(r^{th}\) row (numbering from 0) is \(r \times c\).

We can write \text{mult-table} using two applications of the “count up” idea.
Two-dimensional data example: part 1

;; (mult-table nr nc) produces multiplication table
;; with nr rows and nc columns
;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
  (rows-from 0 nr nc))

;; (rows-from r nr nc) produces mult. table, rows r...(nr-1)
;; rows-from: Nat Nat Nat → (listof (listof Nat))
(define (rows-from r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (row r nc) (rows-from (add1 r) nr nc))])))
Two-dimensional data example: part 2

;; (row r nc) produces rth row of mult. table of length nc
;; row: Nat Nat → (listof Nat)
(define (row r nc)
  (cols-from 0 r nc))

;; (cols-from c r nc) produces entries c...(nc-1) of rth row of mult. table
;; cols-from: Nat Nat Nat → (listof Nat)
(define (cols-from c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (cols-from (add1 c) r nc))])))
Two-dimensional data - explained

• The parameters for the function \texttt{mult-table} specifies the number of rows (\texttt{nr}) and the number of columns (\texttt{nc}) in the table.

• \texttt{mult-table} uses \texttt{rows-from} to create a list of rows.

• \texttt{rows-from}: the parameter \texttt{r} goes from 0 up to \texttt{nr} and \texttt{rows} is used to create each row.

• \texttt{rows}: sets up the arguments for \texttt{cols-from}.

• \texttt{cols-from}: the parameter \texttt{c} goes from 0 up to \texttt{nc} and creates the row as a \texttt{(listof Nat)}. 
Processing two lists simultaneously

We now look at a more complicated recursion, namely writing functions which consume two lists (or two data types, each of which has a recursive definition).

Following the textbook, we will distinguish three different cases.

1. processing just one list recursively
2. processing two lists in lockstep
3. processing two lists at different rates

The simplest case is when one of the lists does not require recursive processing.
Case 1: processing just one list: e.g. my-append

As an example, consider the function my-append.

;; (my-append lst1 lst2) appends lst2 to the end of lst1
;; my-append: (listof Any) (listof Any) → (listof Any)
;; Examples:
(check-expect (my-append empty '(1 2)) '(1 2))
(check-expect (my-append '(3 4) '(1 2 5)) '(3 4 1 2 5))

(define (my-append lst1 lst2)
  ...
)
Case 1: **my-append** implementation

```
(define (my-append lst1 lst2)
  (cond [(empty? lst1) lst2]
       [else (cons (first lst1)
                     (my-append (rest lst1) lst2))]))
```

The code only does simple recursion on `lst1`.

The parameter `lst2` is “along for the ride”.

**append** is a built-in function in Racket.
Case 1: A condensed trace of **my-append**

\[
\text{(my-append (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))}
\Rightarrow \text{(cons 1 (my-append (cons 2 empty) (cons 3 (cons 4 empty))))}
\Rightarrow \text{(cons 1 (cons 2 (my-append empty (cons 3 (cons 4 empty))))))}
\Rightarrow \text{(cons 1 (cons 2 (cons 3 (cons 4 empty))))}
\]
Case 2: processing in lockstep

In general, two lists can either be empty or a cons (four combinations in total).

*Key Idea:* To process two list in *lockstep*, they must be *the same length* and are *consumed at the same rate*.

Since the two lists must be the same length, 
\((\text{empty? lst1})\) is true if and only if \((\text{empty? lst2})\) is true.

This means that out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.
Case 2: template for processing in lockstep

(define (lockstep-template lst1 lst2)
  (cond [(empty? lst1) . . . ]
        [else
         ( . . . (first lst1) . . . (first lst2) . . .
          (lockstep-template (rest lst1) (rest lst2)) . . . )]]))
Case 2: dot product example

To take the dot product of two vectors, we multiply entries in corresponding positions (first with first, second with second, and so on) and sum the results.

Example: the dot product of (1 2 3) and (4 5 6) is
\[1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32.\]

We can store the elements of a vector in a list, so (1 2 3) becomes '(1 2 3).

For convenience, we define the empty vector with no entries, represented by empty.
Case 2: **dot-product** implementation

;; (dot-product lon1 lon2) computes the dot product
;; of vectors lon1 and lon2
;; dot-product: (listof Num) (listof Num) → Num
;; requires: lon1 and lon2 are the same length
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product '(2) '(3)) 6)
(check-expect (dot-product '(2 3) '(4 5)) 23)

(define (dot-product lon1 lon2)
  ...
)

Case 2: **dot-product** implementation

(define (dot-product lon1 lon2)
  (cond [(empty? lon1) 0]
        [else (+ (∗ (first lon1) (first lon2))
                  (dot-product (rest lon1) (rest lon2)))]))
Case 2: \textbf{dot-product} condensed trace

\[(\text{dot-product} \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ \text{empty})))\]
\[(\text{cons} \ 4 \ (\text{cons} \ 5 \ \text{empty})))\]
\[\Rightarrow \ (\ + \ 8 \ (\text{dot-product} \ (\text{cons} \ 3 \ \text{empty})\]
\[(\text{cons} \ 5 \ \text{empty})))\]
\[\Rightarrow \ (\ + \ 8 \ (+ \ 15 \ (\text{dot-product} \ \text{empty}\]
\[\text{empty})))\]
\[\Rightarrow \ (\ + \ 8 \ (+ \ 15 \ 0)) \Rightarrow \ 23\]
Case 3: processing at different rates

Key Strategy: If the two lists lst1, lst2 being consumed are of different lengths, all four combinations of being empty/nonempty are possible:

(and (empty? lst1) (empty? lst2))
(and (empty? lst1) (cons? lst2))
(and (cons? lst1) (empty? lst2))
(and (cons? lst1) (cons? lst2))

Exactly one of these is true, but all must be tested in the template.
Case 3: the template so far

(define (twolist-template lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) . . . ]
        [(and (empty? lst1) (cons? lst2)) . . . ]
        [(and (cons? lst1) (empty? lst2)) . . . ]
        [(and (cons? lst1) (cons? lst2)) . . . ]))

The first case is a **base case**; the second and third may or may not be.
Case 3: refining the template

(define (twolist-template lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2)) (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2)) (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2)) ??? ])

The second and third cases may or may not require recursion.

The fourth case definitely does, but its form is unclear.
Case 3: further refinements

There are many different possible natural recursions for the last cond answer ???:

\[
\ldots (\text{first} \ \text{lst2}) \ldots (\text{twolist-template} \ \text{lst1} \ (\text{rest} \ \text{lst2})) \ldots \\
\ldots (\text{first} \ \text{lst1}) \ldots (\text{twolist-template} \ (\text{rest} \ \text{lst1}) \ \text{lst2}) \ldots \\
\ldots (\text{first} \ \text{lst1}) \ldots (\text{first} \ \text{lst2}) \ldots (\text{twolist-template} \ (\text{rest} \ \text{lst1}) \ (\text{rest} \ \text{lst2})) \ldots
\]

We need to reason further in specific cases to determine which is appropriate.
Case 3 Example: merging two sorted lists

We wish to design a function `merge` that consumes two lists.

Each list is sorted in ascending order (no duplicate values).

`merge` will produce one such list containing all elements.

As an example:

\[(\text{merge} \ (\text{list} \ 1 \ 8 \ 10) \ (\text{list} \ 2 \ 4 \ 6 \ 12)) \Rightarrow (\text{list} \ 1 \ 2 \ 4 \ 6 \ 8 \ 10 \ 12)\]

We need more examples to be confident of how to proceed.
Case 3 Example: more merging examples

(merge empty empty) ⇒ empty
(merge empty (list 2)) ⇒ (list 2)
(merge (list 1 3) empty) ⇒ (list 1 3)
(merge (list 1 4) (list 2)) ⇒ (list 1 2 4)
(merge (list 3 4) (list 2)) ⇒ (list 2 3 4)
Case 3 Example: reasoning about merge

If \textit{lon1} and \textit{lon2} are both nonempty, what is the first element of the
merged list?

It is the smaller of (first lon1) and (first lon2).

If (first lon1) is smaller, then the rest of the answer is the result of
merging (rest lon1) and lon2.

If (first lon2) is smaller, then the rest of the answer is the result of
merging lon1 and (rest lon2).
Case 3 Example: implementation of \texttt{merge}

\begin{verbatim}
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
       [(and (empty? lon1) (cons? lon2)) lon2]
       [(and (cons? lon1) (empty? lon2)) lon1]
       [(and (cons? lon1) (cons? lon2))
          (cond [(< (first lon1) (first lon2))
                  (cons (first lon1) (merge (rest lon1) lon2))] [else (cons (first lon2) (merge lon1 (rest lon2)))]))])
\end{verbatim}
Case 3 Example: a condensed trace

(merge (list 3 4)
    (list 2 5 6))
⇒ (cons 2 (merge (list 3 4)
    (list 5 6)))
⇒ (cons 2 (cons 3 (merge (list 4)
    (list 5 6))))
⇒ (cons 2 (cons 3 (cons 4 (merge empty
    (list 5 6)))))
⇒ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))
Consuming a list and a number

We defined recursion on natural numbers by showing how to view a natural number in a list-like fashion.

**Key Idea:** We can extend our idea for computing on two lists to computing on a list and a number, or on two numbers.

E.g. consider predicate “Does e appear at least $n$ times in this list?”

Example: “Does 2 appear at least 3 times in the list \(\text{list} \ 4 \ 2 \ 2 \ 3 \ 2 \ 4\)?” produces true.
Examples for the function `at-least?`

;; (at-least? n elem lst) determines if elem appears
;; at least n times in lst.
;; at-least?: Nat Any (listof Any) → Bool

(check-expect (at-least? 0 'red (list 1 2 3)) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) false)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)

(define (at-least? n elem lst) . . . )
Developing the code for \texttt{at-least}?

The recursion will involve the parameters \texttt{n} and \texttt{lst}, once again giving four possibilities:

\begin{verbatim}
(define (at-least? n elem lst)
  (cond [(and (zero? n) (empty? lst)) . . . ]
        [(and (zero? n) (cons? lst)) . . . ]
        [(and (> n 0) (empty? lst)) . . . ]
        [(and (> n 0) (cons? lst)) . . . ]))
\end{verbatim}

Once again, exactly one of these four possibilities is true.
Refining \textit{at-least}? \\

\begin{verbatim}
(define (at-least? n elem lst)
  (cond 
    [(and (zero? n) (empty? lst)) . . . ]
    [(and (zero? n) (cons? lst)) . . . ]
    [(and (> n 0) (empty? lst)) . . . ]
    [(and (> n 0) (cons? lst)) ???])
\end{verbatim}

In which cases can we produce the answer without further processing? \\
In which cases do we need further recursive processing to discover the answer?
Improving at-least?

In working out the details for each case, it becomes apparent that some of them can be combined.

If $n$ is zero, it doesn’t matter whether $lst$ is empty or not. Logically, every element always appears at least 0 times.

This leads to some rearrangement of the template, and eventually to the code that appears on the next slide.
Improved \textbf{at-least}?

\begin{verbatim}
(define (at-least? n elem lst)
  (cond [(zero? n) true]
        [(empty? lst) false]
        ; list is nonempty, \( n \geq 1 \)
        [(equal? (first lst) elem) (at-least? (sub1 n) elem (rest lst))]
        [else (at-least? n elem (rest lst))])
\end{verbatim}
Two condensed traces or \texttt{at-least}?

\begin{verbatim}
(at-least? 3 'green (list 'red 'green 'blue)) ⇒
(at-least? 3 'green (list 'green 'blue)) ⇒
(at-least? 2 'green (list 'blue)) ⇒
(at-least? 2 'green empty) ⇒ false
(at-least? 1 8 (list 4 8 15 16 23 42)) ⇒
(at-least? 1 8 (list 8 15 16 23 42)) ⇒
(at-least? 0 8 (list 15 16 23 42)) ⇒ true
\end{verbatim}
Another example: testing list equality

;; (list=? lst1 lst2) determines if lst1 and lst2 are equal
;; list=?: (listof Num) (listof Num) → Bool
(define (list=? lst1 lst2)
    (cond
        [(and (empty? lst1) (empty? lst2)) ... ]
        [(and (empty? lst1) (cons? lst2)) (... (first lst2) ... (rest lst2) ... )]
        [(and (cons? lst1) (empty? lst2)) (... (first lst1) ... (rest lst1) ... )]
        [(and (cons? lst1) (cons? lst2)) ??? ]))

Again there are four cases to consider.
Reasoning about list equality

Two empty lists are equal; if one is empty and the other is not, they are not equal.

If both are nonempty, then their first elements must be equal, and their rests must be equal.

The natural recursion in this case is

\[(\text{list}\,=\,?\,\, (\text{rest}\,\text{lst1})\, (\text{rest}\,\text{lst2}))\]
Implementation of list=?

(define (list=? lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) true]
        [(and (empty? lst1) (cons? lst2)) false]
        [(and (cons? lst1) (empty? lst2)) false]
        [(and (cons? lst1) (cons? lst2))
          (and (= (first lst1) (first lst2))
               (list=? (rest lst1) (rest lst2))))]

Some further simplifications are possible.
Built-in list equality

As you know, Racket provides the predicate `equal?` which tests structural equivalence. It can compare two atomic values, two structures, or two lists. Each of the nonatomic objects being compared can contain other lists or structures.

At this point, you can see how you might write `equal?` if it were not already built in. It would involve testing the type of data supplied, and doing the appropriate comparison, recursively if necessary.
Goals of this module

You should understand the principle of insertion sort, and how the functions involved can be created using the design recipe.

You should be able to use list abbreviations and quote notation for lists where appropriate.

You should be able to construct and work with lists that contain lists.

You should understand the three approaches to designing functions that consume two lists (or a list and a number, or two numbers) and know which one is suitable in a given situation.
Module 08 Summary

List Abbreviations and Dictionaries

1. The functions first, second,... eighth may be used to access elements at a particular location in a list. [12]
2. list creates a list whereas cons grows a list by one. [13]
3. Use ’ to create a list of Num, Sym, Str, Char or an empty list. [14]
4. You can create a list of lists. [15-18]
5. A flat list is a list that does not contain other lists. [29]
6. A nested list is a list which may contain lists which contain lists, and so on to an arbitrary depth. [29]
7. A dictionary associates a key with a value. [31]
Module 08 Summary

Dictionaries

8. Typical dictionary operations include [32]
   • lookup the value associated with a particular key,
   • add a key value pair to the dictionary
   • remove given a key, remove the key value pair.

9. An association list is a list of (key, value) pairs that contains at most one occurrence of any key. [33–34]

10. Use lists of lists to represent a 2-D table or a matrix. [39–41]
Module 08 Summary

Processing Two Lists Simultaneously

11. When processing two lists simultaneously, there are three types of approaches
   (a) processing just one list recursively [43–45]
   (b) processing two lists in lockstep [46–51]
   (c) processing two lists at different rates [52–60, 68-71]

12. The ideas behind processing two lists can be modified to process a list and a number. [61–67]