Custom functions

Using these built-in functions, we can write our own simple functions on lists.

```scheme
;; (next-concert loc) produces the next concert to attend or false if loc is empty

(check-expect
 (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) false)

(define (next-concert clst)
 (cond
 [(empty? clst) false]
 [else (first clst)]))
```
Custom functions

;;; (same-consec? loc) determines if next two concerts are the same

(check-expect
  (same-consec? (cons "a" (cons "b" empty))) false)
(check-expect
  (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" empty)) false)
(define (same-consec? clst)
  (and
    (not (empty? clst))
    (not (empty? (rest clst)))
    (string=? (first clst) (first (rest clst))))))
Contracts involving lists
Contracts

What is the contract for \( \text{next-concert \ clst} \)?

We could use “List” for \( \text{clst} \).

However, we almost always need to answer the question “list of what?”. A list of numbers? A list of strings? A list of any type at all?
Contracts – (listof X) notation

We use (listof X) in contracts, where X may be replaced with any type. For the concert list example in the previous slides, the list contains only strings and has type (listof Str).

Other examples include:

• (listof Num)
• (listof Bool), and
• (listof Any).

Replace X with the most specific type available. (listof X) always includes the empty list, empty.
Contracts – \((\text{anyof } \ldots)\) notation

What about the value produced by \texttt{next-concert}? It might be a string or it might be \texttt{false}.

Use \((\text{anyof } X \ Y \ldots)\) to mean any of the listed types or values.

Examples include:

\begin{itemize}
  \item \((\text{anyof } \text{Num} \ \text{Str})\)
  \item \((\text{anyof } \text{Str} \ \text{Num} \ \text{Bool})\)
  \item \((\text{anyof } 1 \ 2 \ 3)\), and
  \item \((\text{listof } (\text{anyof } \text{Str} \ \text{SomeSymbol}))\)
\end{itemize}
Syntax

List values are \((\text{cons} \ v \ \text{lst})\) where \(v\) is any Racket value (including list values) and \(\text{lst}\) is a list value (which includes \empty\).

The following are valid expressions:

\text{empty}

\((\text{cons} \ e1 \ e2)\), where \(e1\) and \(e2\) are expressions

\((\text{empty?} \ e1)\)

\((\text{first} \ e1)\)

\((\text{rest} \ e1)\)

\((\text{cons?} \ e1)\)
Substitution rules

The substitution rules for lists are:

(\textit{first} (\texttt{cons} \ a \ b)) \Rightarrow \ a, \text{ where } a \text{ and } b \text{ are values.}

(\textit{rest} (\texttt{cons} \ a \ b)) \Rightarrow \ b, \text{ where } a \text{ and } b \text{ are values.}

(\textit{empty?} \ \texttt{empty}) \Rightarrow \ true. 

(\textit{empty?} \ a) \Rightarrow \ \texttt{false}, \text{ where } a \text{ is any Racket value other than } \texttt{empty}.

(\textit{cons?} (\texttt{cons} \ a \ b)) \Rightarrow \ true, \text{ where } a \text{ and } b \text{ are values.}

(\textit{cons?} \ a) \Rightarrow \ \texttt{false}, \text{ where } a \text{ is any Racket value not created using } \texttt{cons}. 


Substitution rules – Processing lists

Most interesting functions will process the entire consumed list. How many concerts are on the list? How many times does "Kamelot" appear? Which artists are duplicated in the list?

The structure of a function often mirrors the structure of the data it consumes. As we encounter more complex data types, we will find it useful to be precise about their structures.

We will do this by developing data definitions.

We can even go so far as developing function templates based on the data definitions it consumes.
Data definitions & templates
Data definitions

Informally: a list of strings is either empty, or consists of a first string followed by a list of strings (the rest of the list).

;; A (listof Str) is one of:
;; ⋆ empty
;; ⋆ (cons Str (listof Str))

This is a recursive data definition, with a base case and a recursive (self-referential) case.

We can use this data definition to show rigourously that
(cons "a" (cons "b" empty)) is a (listof Str).
Data definitions – Generalization

We can generalize lists of strings to other types by using an X:

;;;; A (listof X) is one of:
;;;; * empty
;;;; * (cons X (listof X))
Data definitions –
Template for processing a \((\text{listof } X)\)

The structure of a function often reflects the structure of the data it consumes. The structure of a \((\text{listof } X)\) is given by its data definition:

```plaintext
;; A (\text{listof } X) is one of:
;;  * empty
;;  * (\text{cons } X (\text{listof } X))
```

So a function consuming a \((\text{listof } X)\) will need to distinguish between these two cases.
Data definitions – 
Template for processing a \((\text{listof } X)\)

\[\text{;; listof-X-template: } (\text{listof } X) \rightarrow \text{Any} \]

\[
\text{(define } (\text{listof-X-template } \text{lox}) \text{)}
\]

\[
\text{(cond}
\]
\[
\quad [(\text{empty? } \text{lox}) \ldots]\; ; \text{takes care of } \ast \text{ empty}
\]
\[
\quad [(\text{cons? } \text{lox}) \ldots)]\)\); \text{takes care of}
\]
\[
\quad \ast (\text{cons } X (\text{listof } X))
\]

The \ldots represents a place to fill in code specific to the problem.

\((\text{cons? } \text{lox})\) could be replaced with \texttt{else}.

In the last case, we know from the data definition that there is a
• first value \((X)\) on the list as well as a
• rest of the list \((\text{(listof } X))\)
Data definitions – Template for processing a (listof X)

;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond
   [(empty? lox) ...]
   [(cons? lox) (... ... (first lox) ... (rest lox))]])

Because (rest lox) also is of type (listof X), we apply the same computation to it, that is, we apply listof-X-template.
Data definitions –
Template for processing a \((\text{listof } X)\)

\[
;; \text{listof}-X\text{-template}: (\text{listof } X) \rightarrow \text{Any}
\]

\[
\text{(define } (\text{listof}-X\text{-template } \text{lox})
\text{(cond}
\text{[(empty? } \text{lox}) ...]
\text{[(cons? } \text{lox}) (... 
\text{... (first } \text{lox)}
\text{ (listof}-X\text{-template (rest } \text{lox)})])])\]
\]

This is the resulting template for a function consuming a \((\text{listof } X)\), which matches the data definition.

We can now fill in the dots for a specific example.

A great approach is simply copying the template!
Example: how many concerts?

;; (count-concerts los) counts the number of concerts in los
;; count-concerts: (listof Str) -> Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)
(define (count-concerts los)
  (cond
   [(empty? los) ...]
   [else (... ... (first los)
     (count-concerts (rest los)))]))
Example: how many concerts?

```
(define (count-concerts los)
  (cond
    [(empty? los) ...]
    [else (... ...(first los) (count-concerts (rest los)))]))
```

To proceed, think about list functions:

• What does the function produce in the base case?

• What does the function do to the first element in a non-empty list?

• How does the function combine the value produced from the first element with the value obtained by applying the function recursively to the rest of the list?
Example: how many concerts?

;; (count-concerts los) counts the number of concerts in los
;; count-concerts: (listof Str) -> Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)
(define (count-concerts los)
  (cond
   [(empty? los) 0]
   [else (+ 1 (count-concerts (rest los)))]))

This is a recursive function (i.e., it uses recursion).
Recursion

(define (count-concerts los)
  (cond
   [(empty? los) 0]
   [else (+ 1 (count-concerts (rest los)))]))

A function is recursive when the body of the function involves an application of the same function.

This is an important technique which we will use quite frequently throughout the course.

Fortunately, our substitution rules allow us to trace such a function without much difficulty.
Tracing **count-concerts**

\[
\text{(count-concerts (cons "a" (cons "b" empty)))}
\]

\[
\text{(define (count-concerts los)}
\text{  (cond [(empty? los) 0]}
\text{    [else (+ 1 (count-concerts (rest los)))]})]
\]
Tracing `count-concerts`

```
(count-concerts (cons "a" (cons "b" empty)))

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (cond [(empty? (cons "a" (cons "b" empty))) 0]
          [else (+ 1 (count-concerts (rest
                                      (cons "a" (cons "b" empty))))))])
```
Tracing count-concerts

(count-concerts (cons "a" (cons "b" empty)))

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (cond [(empty? (cons "a" (cons "b" empty))) 0]
         [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])
Tracing count-concerts

(count-concerts (cons "a" (cons "b" empty)))

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (cond [(empty? (cons "a" (cons "b" empty))) 0]
         [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])
Tracing **count-concerts**

\[
\text{(count-concerts \ (cons \ "a" \ (cons \ "b" \ empty)))}
\]

\[
\text{(define \ (count-concerts \ los) \}
\text{ \ (cond \ [(empty? \ los) 0]}
\text{ \ \ \ \ \ \ \ \ \ [else \ (+ \ 1 \ (count-concerts \ (rest \ los)))])})
\]

\[
\Rightarrow \ (\text{cond} \ [(empty? \ \text{(cons \ "a" \ (cons \ "b" \ empty)))}) \ 0]}
\text{ \ \ \ \ \ \ \ \ \ \ [else \ (+ \ 1 \ (count-concerts \ \text{(rest} \ \text{(cons \ "a" \ (cons \ "b" \ empty))}))])])}
\]

\[
\Rightarrow \ (\text{cond} \ [false \ 0]}
\text{ \ \ \ \ \ \ \ \ \ \ [else \ (+ \ 1 \ (count-concerts \ \text{(rest} \ \text{(cons \ "a" \ (cons \ "b" \ empty))})])])}
\]
Tracing **count-concerts**

\[
(count\text{-concerts} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty})))
\]

\[
(\text{define} \ (count\text{-concerts} \ los)
\quad (\text{cond}
\quad \quad [(\text{empty?} \ los) \ 0]
\quad \quad [\text{else} \ (+ \ 1 \ (count\text{-concerts} \ (\text{rest} \ los)))]))
\]

=> (\text{cond}
\quad \quad [(\text{empty?} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))) \ 0]
\quad \quad [\text{else} \ (+ \ 1 \ (count\text{-concerts} \ (\text{rest}
\quad \quad \quad (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])

=> (\text{cond}
\quad \quad \text{false} \ 0]
\quad \quad [\text{else} \ (+ \ 1 \ (count\text{-concerts} \ (\text{rest}
\quad \quad \quad \quad (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])
Tracing \texttt{count-concerts}

\[
\text{(count-concerts (cons "a" (cons "b" empty)))}
\]

\[
\text{(define (count-concerts los)}
\text{(cond [(empty? los) 0]
[else (+ 1 (count-concerts (rest los)))]))}
\]

\[
\Rightarrow \text{(cond [(empty? (cons "a" (cons "b" empty))) 0]
[else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))]))}
\]

\[
\Rightarrow \text{(cond [false 0]
[else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))]))}
\]

\[
\Rightarrow \text{(cond [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))]))}
\]
Tracing \textit{count-concerts}

\[(\text{count-concerts} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty})))\]

\[(\text{define} \ \ (\text{count-concerts} \ \text{los}) \ \\
  \text{(cond} \ [(\text{empty?} \ \text{los}) \ 0] \ [\text{else} \ (+ \ 1 \ (\text{count-concerts} \ (\text{rest} \ \text{los})))])\)]

\[\Rightarrow (\text{cond} \ [(\text{empty?} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))) \ 0] \ [\text{else} \ (+ \ 1 \ (\text{count-concerts} \ (\text{rest} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])\]

\[\Rightarrow (\text{cond} \ [\text{false} \ 0] \ [\text{else} \ (+ \ 1 \ (\text{count-concerts} \ (\text{rest} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])\]

\[\Rightarrow (\text{cond} \ [\text{else} \ (+ \ 1 \ (\text{count-concerts} \ (\text{rest} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])\]
Tracing \textit{count-concerts}

\[
=> (\text{cond} \ [\text{else} \ (+ \ 1 \ (\text{count-concerts} \ (\text{rest} \ (\text{cons} \ "a" \ (\text{cons} \ "b" \ \text{empty}))))))])
\]

Tracing `count-concerts`

=> (cond [else (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))])

=> (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))
Tracing \texttt{count-concerts}

\[
\Rightarrow (\texttt{cond} \ [\texttt{else} \ (+ \ 1 \ (\texttt{count-concerts} \ \texttt{(rest} \ \texttt{(cons} \ "a" \ \texttt{(cons} \ "b" \ \texttt{empty})))))])
\]

\[
\Rightarrow (+ \ 1 \ (\texttt{count-concerts} \ \texttt{(rest} \ \texttt{(cons} \ "a" \ \texttt{(cons} \ "b" \ \texttt{empty})))))
\]

\[
\Rightarrow (+ \ 1 \ (\texttt{count-concerts} \ \texttt{(cons} \ "b" \ \texttt{empty})))
\]

Here we have finished the first call to \texttt{count-concerts} and now perform the first \texttt{recursive} call to \texttt{count-concerts}. 
Tracing **count-concerts**

\[ (+ 1 \ (\text{count-concerts} \ (\text{cons} \ "b" \ \text{empty}))) \]

\[
(\text{define} \ (\text{count-concerts} \ \text{los})
  (\text{cond} \ [(\text{empty?} \ \text{los}) 0]
  [\text{else} \ (+ 1 \ (\text{count-concerts} \ (\text{rest} \ \text{los})))]))
\]
Tracing count-concerts

(+ 1 (count-concerts (cons "b" empty)))

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (+ 1 (cond [(empty? (cons "b" empty)) 0]
             [else (+ 1 (count-concerts (rest (cons "b" empty))))])])
Tracing **count-concerts**

\[
(+ 1 (\text{count-concerts} (\text{cons} \ "b" \ \text{empty})))
\]

\[
(\text{define} \ (\text{count-concerts} \ \text{los})
  (\text{cond} \ [(\text{empty?} \ \text{los}) \ 0]
    [\text{else} \ (+ 1 (.\text{count-concerts} (\text{rest} \ \text{los}))))])
\]

\[
=> (+ 1 (\text{cond} \ [(\text{empty?} \ (\text{cons} \ "b" \ \text{empty})) \ 0]
    [\text{else} \ (+ 1 (.\text{count-concerts} (\text{rest} \ (\text{cons} \ "b" \ \text{empty}))))]))
\]

\[
=> (+ 1 (\text{cond} \ [\text{false} \ 0]
    [\text{else} \ (+ 1 (.\text{count-concerts} (\text{rest} \ (\text{cons} \ "b" \ \text{empty}))))]))
\]
Tracing `count-concerts`

```
(+ 1 (count-concerts (cons "b" empty)))

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (+ 1 (cond [(empty? (cons "b" empty)) 0]
                [else (+ 1 (count-concerts (rest (cons "b" empty))))]))

=> (+ 1 (cond [false 0]
                [else (+ 1 (count-concerts (rest (cons "b" empty))))]))

=> (+ 1 (cond [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
```
Tracing \texttt{count-concerts}

\[
\Rightarrow (+ 1 (\texttt{cond [else (+ 1 (\texttt{count-concerts (rest (cons "b" empty))))]]}))
\]
Tracing \textit{count-concerts}

\[
\Rightarrow ( + 1 (\text{cond} [\text{else} (+ 1 (\text{count-concerts} (\text{rest} (\text{cons} "b" \text{empty}))))]))
\]

\[
\Rightarrow ( + 1 (+ 1 (\text{count-concerts} (\text{rest} (\text{cons} "b" \text{empty}))))))
\]
Tracing `count-concerts`

```lisp
=> (+ 1 (cond [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
=> (+ 1 (+ 1 (count-concerts (rest (cons "b" empty)))))
=> (+ 1 (+ 1 (count-concerts empty)))
```

Here we have finished the first **recursive** call to `count-concerts` and now perform the second **recursive** call to `count-concerts`. 
Tracing **count-concerts**

\[ (+ 1 (+ 1 (\text{count-concerts} \ \text{empty}))) \]

\[ \text{(define} \ (\text{count-concerts} \ \text{los}) \]
\[ \text{(cond} \ [(\text{empty?} \ \text{los}) 0] \]
\[ \text{[else} \ (+ 1 \ (\text{count-concerts} \ (\text{rest} \ \text{los})))])) \]
Tracing `count-concerts`

\[
(+ 1 (+ 1 (count-concerts empty)))
\]

(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))

=> (+ 1 (+ 1 (cond [(empty? empty) 0]
           [else (+ 1 (count-concerts (rest empty)))])))))
Tracing **count-concerts**

\[ (+ 1 (+ 1 \text{ (count-concerts empty)})) \]

\[
\text{(define (count-concerts los)}
  \text{(cond [(empty? los) 0]} \]
  \text{[else (+ 1 (count-concerts (rest los)))]})
\]

\[ \Rightarrow (+ 1 (+ 1 \text{ (cond [(empty? empty) 0]} \]
  \text{[else (+ 1 (count-concerts (rest empty)))]})] \]

\[ \Rightarrow (+ 1 (+ 1 \text{ (cond [true 0]} \]
  \text{[else (+ 1 (count-concerts (rest empty)))]})] \]
Tracing \textit{count-concerts}

(+ 1 (+ 1 (count-concerts empty)))

\begin{verbatim}
(define (count-concerts los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-concerts (rest los)))]))
\end{verbatim}

=> (+ 1 (+ 1 (cond [(empty? empty) 0]
                    [else (+ 1 (count-concerts (rest empty)))])))

=> (+ 1 (+ 1 (cond [true 0]
                    [else (+ 1 (count-concerts (rest empty)))])))

=> (+ 1 (+ 1 0))
Tracing \textit{count-concerts}

\[ \Rightarrow (+ 1 (+ 1 0)) \]

From the third (= second recursive) call to \textit{count-concerts}.
From the second (= first recursive) call to \textit{count-concerts}.
From the first call to \textit{count-concerts}. 
Tracing `count-concerts`

=> (+ 1 (+ 1 0))

=> (+ 1 1)

=> 2
Tracing `count-concerts` – Condensed traces

The full trace contains too much detail, so we instead use a condensed trace of the recursive function. This shows the important steps and skips over the trivial details.

This is a space saving tool we use in these slides, not a rule that you have to understand.
This condensed trace shows more clearly how the application of a recursive function leads to an application of the same function to a smaller list, until the base case is reached.
Tracing **count-concerts** – Condensed traces

From now on, for the sake of readability, we will tend to use condensed traces. At times we will condense even more (for example, not fully expanding constants).

If you wish to see a full trace, you can use the Stepper.

But as we start working on larger and more complex forms of data, it becomes harder to use the Stepper, because intermediate expressions are so large.
Example: `count-kamelot`

`; (count-kamelot los) produces the number of occurrences of "Kamelot" in los
`; count-kamelot: (listof Str) -> Nat
`; Examples:
(check-expect (count-kamelot empty) 0)
(check-expect (count-kamelot (cons "Kamelot" empty)) 1)
(check-expect (count-kamelot (cons "Edguy" (cons "Delain" empty))) 0)
(define (count-kamelot los) ...)"
Example: **count-kamelot** (cont)

The template is a good place to start writing code.

1. Write the template.
2. Alter it according to the specific function you want to write.

For instance, we can generalize `count-kamelot` to a function which also consumes the string to be counted.

```scheme
;; count-string: Str (listof Str) -> Nat
(define (count-string str los) ...)
```

The recursive function application will be

```scheme
(count-string str (rest los)).
```
More about templates

Tip: Filling in the templates

In the Function Definition part of the design recipe: First write the cond-answers corresponding to base cases (which don’t involve recursion).

For self-referential or recursive cases, figure out how to combine the values provided by the recursive application(s) and the selector expressions.

As always, create examples that exercise all parts of the data definition, and tests that exercise all parts of the code.
Refining the (listof X) template

Sometimes, each $X$ in a (listof $X$) may require further processing. Indicate this with a template for $X$ as a helper function.

```scheme
;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
       [else (... (X-template (first lox)) ...
                 (listof-X-template (rest lox)))]))
```

We assume this generic data definition and template from now on.
Templates as generalizations

A template provides the basic shape of the code as suggested by the data definition.

Later in the course, we will learn about an abstraction mechanism (higher-order functions) that can reduce the need for templates.

We will also discuss alternatives for tasks where the basic shape provided by the template is not right for a particular computation.
Patterns of recursion
Patterns of recursion

The list template has the property that the form of the code matches the form of the data definition.

We will call this simple recursion.

There are other patterns of recursion which we will see later on in the course.

Until we do, the functions we write (and ask you to write) will use simple recursion (and hence will fit the form described by such templates).

Use the templates.
Simple recursion

In simple recursion, every argument in a recursive function application is either:

• unchanged, or

• one step closer to a base case according to a data definition

(define (func lst) ... (func (rest lst)) ...) ; Simple
(define (func lst x) ... (func (rest lst) x) ...) ; Simple
(define (func lst x) ... (func (process lst) x) ... ) ; NOT Simple
(define (func lst x) ... (func (rest lst) (math-function x)) ...) ; NOT Simple