The List Data Structure

A data structure is a data organization, management and storage format that enables efficient access and modification.

There are other data structures in Racket, but lists are a fundamental part of the language.

Lists are self-referential

When a data structure is self-referential, it means that is composed of smaller versions of itself.

A well-formed list is one that is formed by a sequence of cons expressions that contain ‘() at the end.

Compare the list function to using the cons function. The result of the list function is always well-formed because it is actually using the cons function behind the scene.

Self-referential data type definitions

SELF-REFERENTIAL DATA DEFINITION FOR A LIST OF STRINGS:
A list of strings (LOS) is either:
1. empty, or
2. it’s a (cons s stlist), where s is a string and stlist is a LOS

You’ll see that functions used to process lists have a similar structure to this definition.

Pattern of Recursive Functions

1. The body of the function contains a decision statement.
2. There is at least one base case, or stopping condition, to end the recursion.
3. There is at least one recursive case in which the function calls itself with a value that is closer to the base case than the current value of the parameter to the function.
Purpose of Recursive Expressions
These expressions allow us to process every element in a list in sequence and return a list, containing an arbitrary number of values. It allows us to iterate through a list in a non-destructive manner.
Lists, like strings, are immutable data structures. When a list is consumed by a function and the return value is a non-empty list, the original list is not changed...a new one is created.

Shape of Recursive Functions
Template for recursive functions:
(define (rec-temp lissy)
  (cond
    [(null? lissy) ...]
    ...[else (...(first lissy)...(rec-temp (rest lissy)))]
 where CO is some "combining operation" such as *, +, string-append, cons,....

Example Recursive Function
Write a function that consumes a list of any type (loat) and produces a list containing only the quoted symbols.
(define (only-sym loat)
  (cond
    [(empty? loat) '()]
    [(symbol? (first loat))
      (cons (first loat) (only-sym (rest loat)))]
    [else (only-sym (rest loat))])
  (check-expect (only-sym (1 a "oil" b d)) '(a b d))
  (check-expect (only-sym (')) ')') ;; always test empty list
  (check-expect (only-sym (1 3 5 7 4)) '())
  (check-expect (only-sym (jack and jill)) ' (jack and jill))

Define a function that consumes a list of anything (loat) and produces only the quoted symbols in loat.
(define (only-sym loat)
  (cond
    [(empty? loat) '()]
    [(symbol? (first loat))
      (cons (first loat) (only-sym (rest loat)))]
    [else (only-sym (rest loat))])
  (check-expect (only-sym (1 a d f 3 g l)) '(a d f g l))
  (check-expect (only-sym (')) ')')
  (check-expect (only-sym ("abc def ghi")) ' (abc def ghi))
  (check-expect (only-sym ("Hello, "world" !"))) ' (Hello, "world" !")