Lists of Lists

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 5.3

© Mitchell Wand, 2012-2017
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Learning Outcomes

- At the end of this lesson, the student should be able to
  - Give examples of S-expressions
  - Write the data definition and template for S-expressions
  - Write functions on S-expressions using the template

S-expressions (informally)

- An S-expression is something that is either a string or a list of S-expressions.
- So if it's a list, it could contain strings, or lists of strings, or lists of lists of strings, etc.
- Think of it as a nested list, where there's no bound on how deep the nesting can get.
- Another way of thinking of it is as a multi-way tree, except that the data is all at the leaves instead of in the interior nodes.

Some History

- An S-expression is a kind of nested list, that is, a list whose elements may be other lists. Here is an informal history of S-expressions.
- S-expressions were invented by John McCarthy (1927-2011) for the programming language Lisp in 1958. McCarthy invented Lisp to solve problems in artificial intelligence.
- Lisp introduced lists, S-expressions, and parenthesized syntax. The syntax of Lisp and its descendants, like Racket, is based on S-expressions.
- The use of S-expressions for syntax makes it easy to read and write programs: all you have to do is balance parentheses. This is much simpler than the syntax of other programming languages, which have semicolons and other rules that can make programs harder to read.
- S-expressions are one of the great inventions of modern programming. They were the original idea from which things like XML and JSON grew.

Examples

```
"alice"
"bob"
"carole"
(list "alice" "bob")
(list (list "alice" "bob") "carole")
(list "dave"
      (list "alice" "bob")
      "carole")
(list (list "alice" "bob")
      (list (list "ted" "carole")))
```

Here are some examples of S-expressions, in **list** notation (See Lesson 4.1)

Examples

```
"alice"
"bob"
"carole"
("alice" "bob")
(("alice" "bob") "carole")
("dave" ("alice" "bob") "carole")
(("alice" "bob")
 (("ted" "carole")))
```

Here are the same examples of S-expressions, in **write** notation (See Lesson 4.1). We often use write notation because it is more compact.

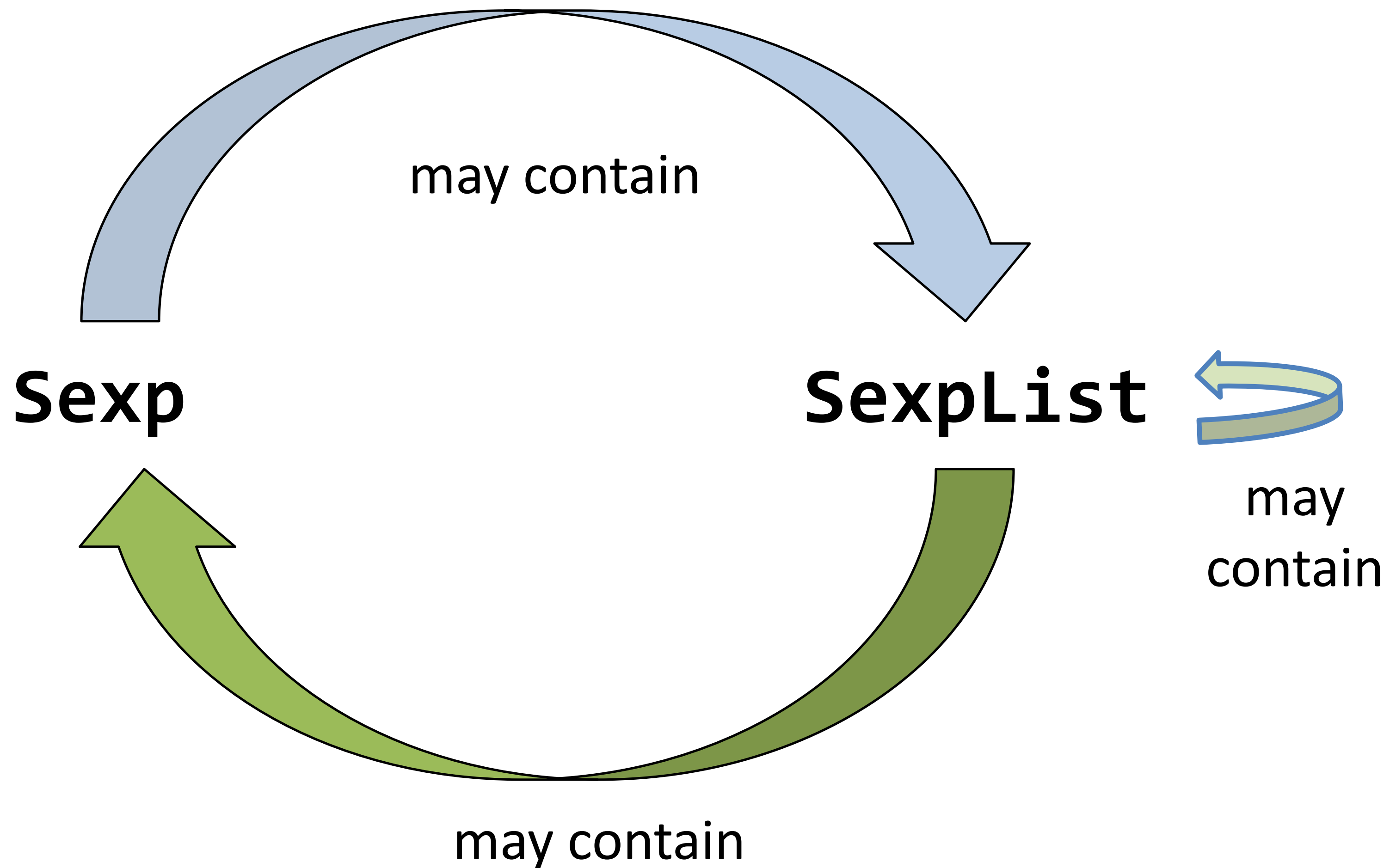
Data Definition

An Sexp is either
-- a String (any string will do), or
-- an SexpList

An SexpList is either
-- empty
-- (cons Sexp SexpList)

Here we've built S-expressions where the basic data is strings, but we could build S-expressions of numbers, cats, sardines, or whatever. We'll see that later in this lesson.

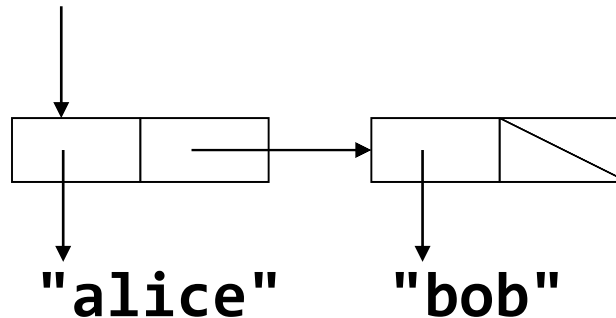
This is mutual recursion



Data Structures

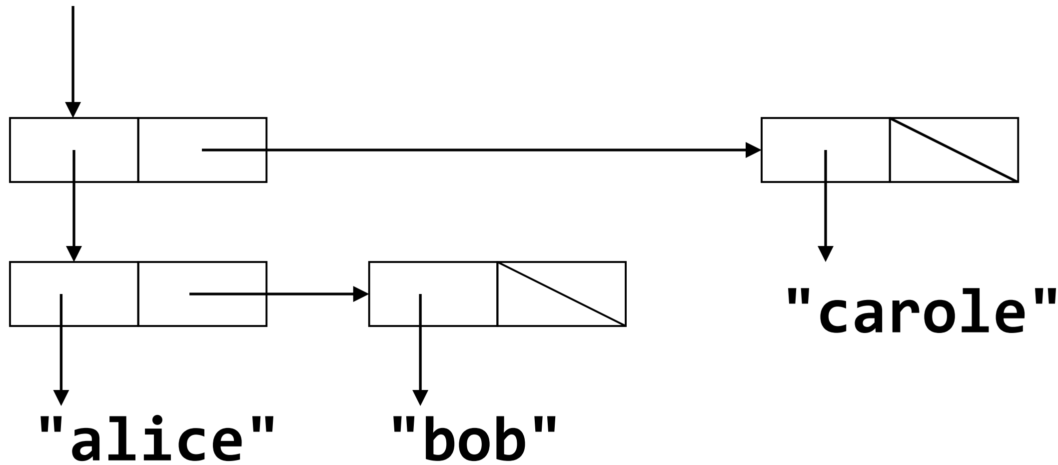
```
"alice"
"bob"
"carole"
("alice" "bob")
```

A list of S-expressions is implemented as a singly-linked list. Here is an example.



Data Structures

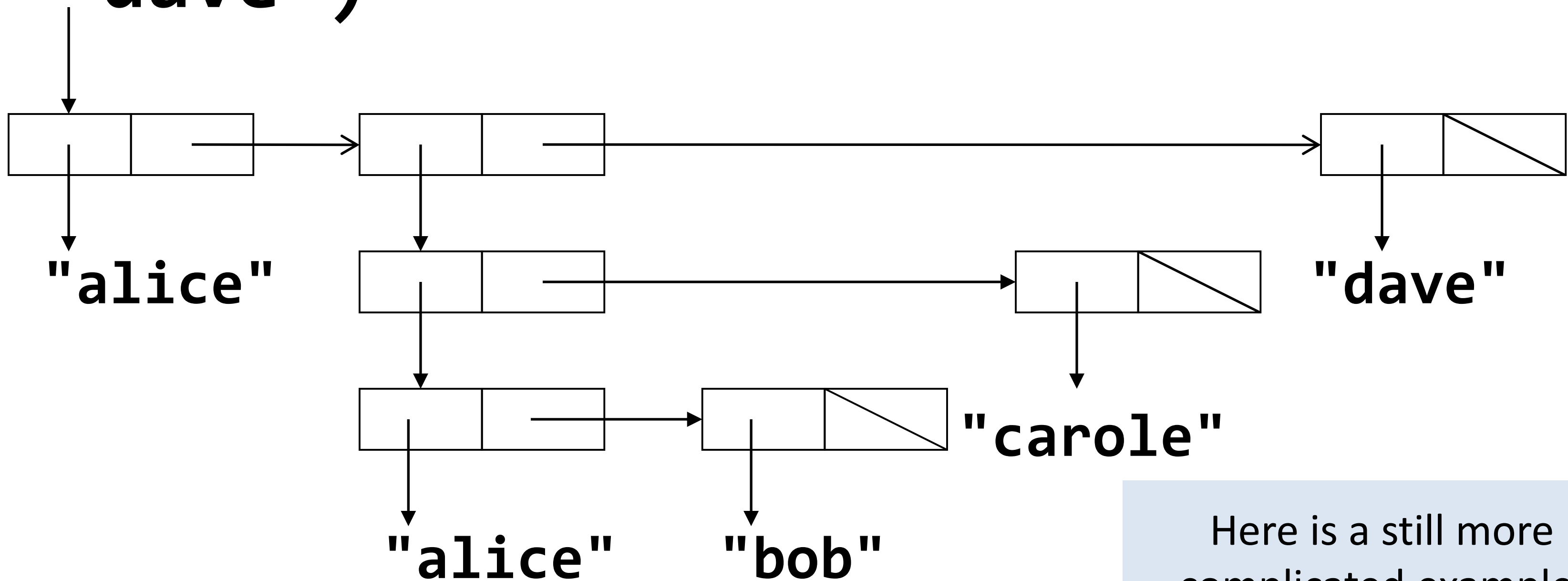
(("alice" "bob") "carole")



Here is a slightly more complicated example. Observe that the **first** of this list is another list. The **first** of the **first** is the string **"alice"**.

Data Structures (cont'd)

```
("alice"
 (("alice" "bob") "carole")
 "dave")
```

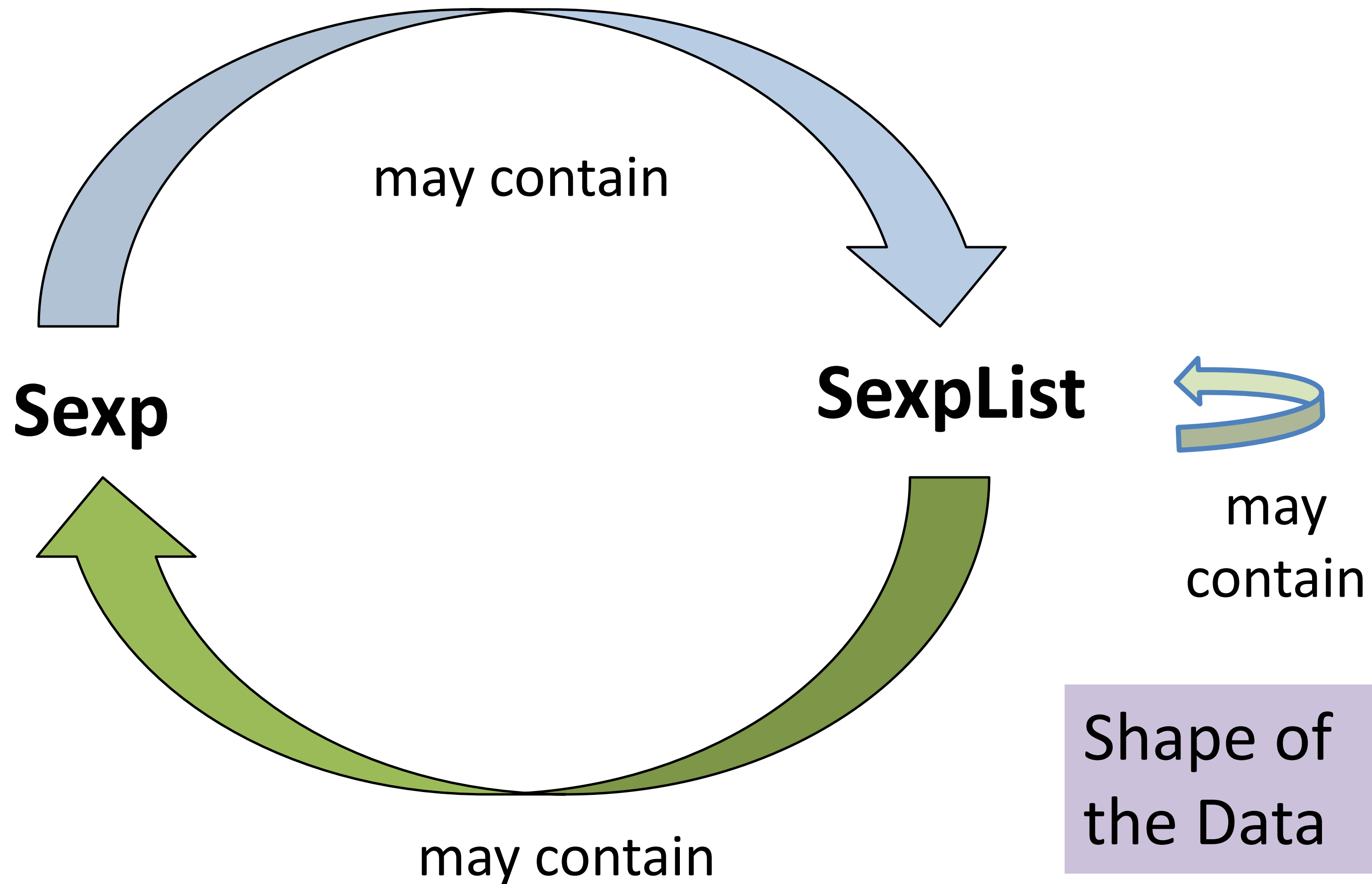


Here is a still more complicated example.

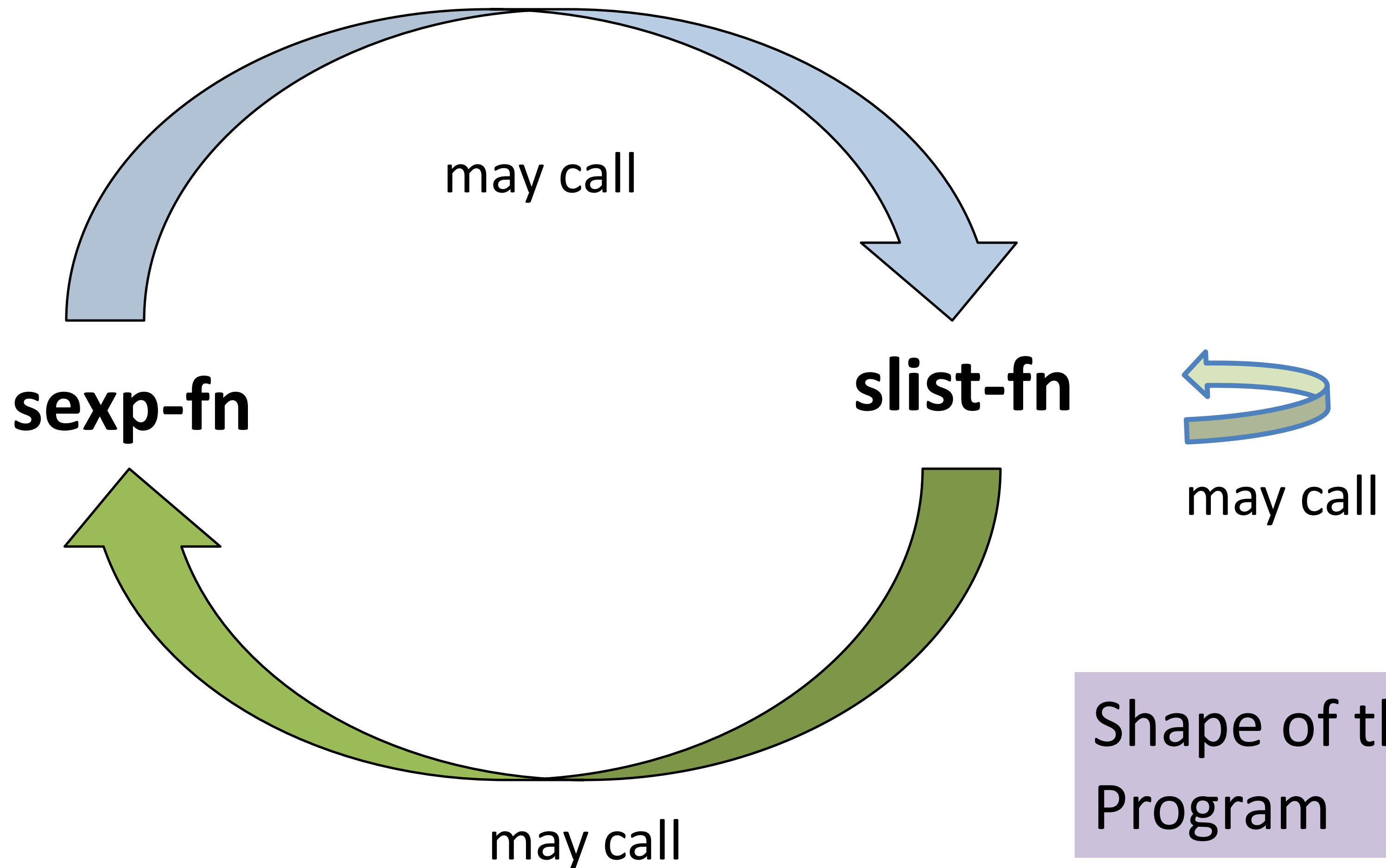
Observer Template: functions come in pairs

```
;; sexp-fn : Sexp -> ??
;; slist-fn : SexpList -> ??

(define (sexp-fn s)
  (cond
    [(string? s) ...]
    [else (... (slist-fn s))]))

(define (slist-fn sexps)
  (cond
    [(empty? sexps) ...]
    [else (... (sexp-fn  (first sexps))
               (slist-fn (rest sexps)))]))
```

Remember: the shape of the program follows the shape of the data



Remember: the shape of the program follows the shape of the data



One function, one task

- Each function deals with exactly one data definition.
- So functions will come in pairs
- Write contracts and purpose statements together, **or**
- Write one, and the other one will appear as a wishlist function

occurs-in?

```
;; occurs-in? : Sexp String -> Boolean
;; returns true iff the given string occurs somewhere in
   the given Sexp.
;; occurs-in-slist? : SexpList String -> Boolean
;; returns true iff the given string occurs somewhere in
   the given list of Sexps.
```

Here's an example of a pair of related functions: **occurs-in?** , which works on a **Sexp**, and **occurs-in-slist?** , which works on a **SexpList**.

Examples/Tests

```
(check-equal?
  (occurs-in? "alice" "alice")
  true)

(check-equal?
  (occurs-in? "bob" "alice")
  false)

(check-equal?
 (occurs-in?
  (list "alice" "bob")
  "cathy")
 false)
```

```
(check-equal?
 (occurs-in?
  (list (list "alice" "bob")
        "carole")
  "bob")
 true)

(check-equal?
 (occurs-in?
  (list "alice"
        (list (list "alice" "bob")
              "dave")
        "eve")
  "bob")
 true)
```

More Examples

```
;; number-of-strings-in-sexp  : Sexp -> Number
;; number-of-strings-in-sexps : SexpList -> Number
;; returns the number of strings in the given Sexp or
   SexpList.

;; characters-in-sexp : Sexp -> Number
;; characters-in-sexps : SexpList -> Number
;; returns the total number of characters in the strings
   in the given Sexp or SexpList.
```

The S-expression pattern

Can do this for things other than strings:

```
An XSexp is either
-- an X
-- an XSexpList

A XSexpList is either
-- empty
-- (cons XSexp XSexpList)
```

The Template for **XSexp**

```
;; sexp-fn : XSexp-> ??
(define (sexp-fn s)
  (cond
    [(X? s) ...]
    [else (slist-fn s)]))

;; slist-fn : XSexpList -> ??
(define (slist-fn sexps)
  (cond
    [(empty? sexps) ...]
    [else (... (sexp-fn  (first sexps))
               (slist-fn (rest sexps)))]))
```

(first sexps) is a **XSexp**. This is mixed data, so our rule about the shape of the program following the shape of the data tells us that we should expect to wrap it in an **(sexp-fn ...)** .

Sexps with Sardines as the data

```
A SardineSexp is either
-- a Sardine
-- a SardineSexpList

A SardineSexpList is either
-- empty
-- (cons SardineSexp
         SardineSexpList)
```

An example of the **XSexp** pattern.

The Template for SardineSexp

```
;; sardine-sexp-fn : SardineSexp -> ??
(define (sardine-sexp-fn s)
  (cond
    [(sardine? s) ...]
    [else (sardine-sexp-list-fn s)]))

;; sardine-sexp-list-fn : SardineSexpList -> ??
(define (sardine-sexp-list-fn sexps)
  (cond
    [(empty? sexps) ...]
    [else (... (sardine-sexp-fn   (first sexps))
               (sardine-sexp-list-fn (rest sexps)))]))
```

Summary

- Nested Lists occur all the time
- Mutually recursive data definitions
- Mutual recursion in the data definition leads to mutual recursion in the template
- Mutual recursion in the template leads to mutual recursion in the code

Summary

- You should now be able to:
  - Give examples of S-expressions
  - Give some reasons why S-expressions are important
  - Write the data definition and template for S-expressions
  - Write functions on S-expressions using the template

Next Steps

- Study the file 05-3-sexps.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 5.3
- Go on to the next lesson