Lists of Lists

CS 5010 Program Design Paradigms

Lesson 6.4

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.

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 S-expression of Strings (SoS) is either

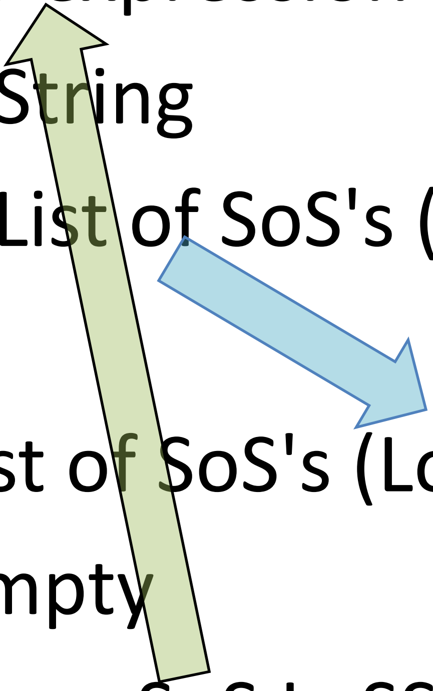
-- a String

-- a List of SoS's (LoSS)

A List of SoS's (LoSS) is either

-- empty

-- (cons SoS LoSS)



Let's write down a precise definition:

- An S-expression is either a string or a list of S-expressions
- A list of S-expressions is either empty or the cons of an S-expressions and another list of S-expressions.
- Note that the data definition for "list of S-expressions" follows the familiar pattern for lists.
- These two definitions are mutually recursive, as you can see from the two arrows

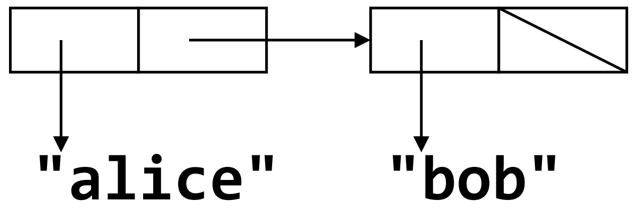
This is mutual recursion

$$\begin{tikzcd}
\textbf{SoS} \arrow[r, bend left=60, "\text{defined in terms of}"] & \textbf{LoSS} \arrow[l, bend left=60, "\text{defined in terms of}"]
\end{tikzcd}$$

Data Structures

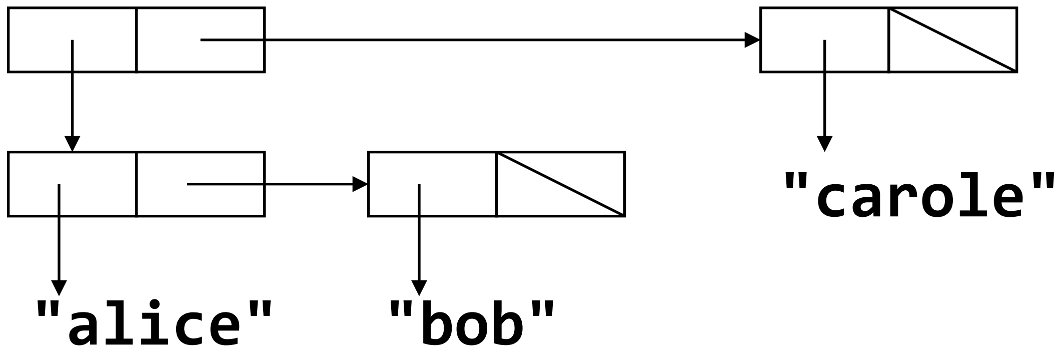
```
"alice"
"bob"
"carole"
("alice" "bob")
```

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



Data Structures

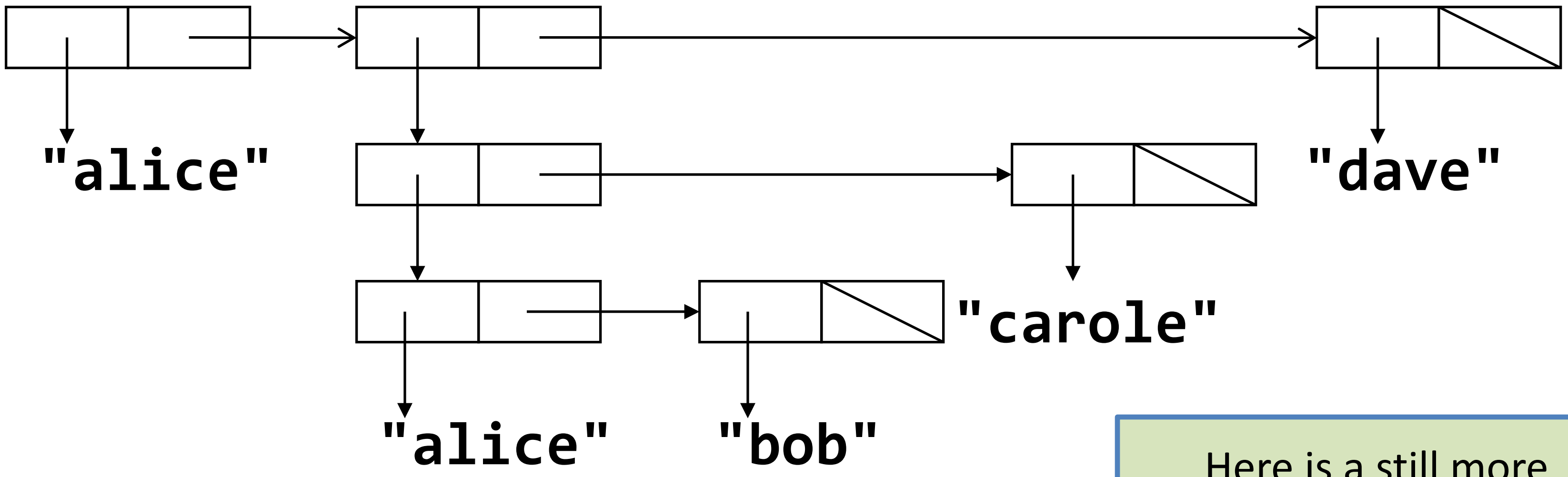
```
(("alice" "bob") "carole")
```



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.

The template recipe

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many cond clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate ``natural recursions'' for the template to represent the self-references of the data definition.
Do any of the fields contain compound or mixed data?	If the value of a field is a foo, add a call to a foo-fn to use it.

Remember the template recipe

Template: functions come in pairs

```
;; sos-fn : SoS -> ??
(define (sos-fn s)
  (cond
    [(string? s) ...]
    [else (loss-fn s)]))

;; loss-fn : LoSS -> ??
(define (loss-fn los)
  (cond
    [(empty? los) ...]
    [else (... (sos-fn (first los))
               (loss-fn (rest los)))]))
```

(first los) is a SoS. This is mixed data, so the last rule in the template recipe tells us we need to wrap it in a **(sos-fn ...)** .

This is mutual recursion

$$\begin{tikzcd}[column sep=huge] \texttt{sos-fn} \arrow[r, bend left=60, "\text{defined in terms of}"] & \texttt{loss-fn} \arrow[l, bend left=60, "\text{defined in terms of}"] \end{tikzcd}$$

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? : SoS String -> Boolean
;; returns true iff the given string occurs somewhere in
   the given sos.
;; occurs-in-loss? : LoSS String -> Boolean
;; returns true iff the given string occurs somewhere in
   the given loss.
```

Here's an example of a pair of related functions: **occurs-in?** , which works on a **SoS**, and **occurs-in-loss?** , which works on a **LoSS**.

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)
```

Livecoding: sos-and-loss.rkt

- occurs-in? : http://youtu.be/w_URqq2LrQU
- number-of-strings : http://youtu.be/9z-jdukgRx4

The inspiration for this livecoding exercise comes from here and here (ignore the sappy music). Background information

The S-expression pattern

Can do this for things other than strings:

```
An SexpOfX is either
-- an X
-- a ListOfSexpOfX

A ListOfSexpOfX is either
-- empty
-- (cons SexpOfX ListOfSexpOfX)
```

The Template for SexpX

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

;; losexp-fn : ListOfSexpOfX -> ??
(define (losexp-fn los)
  (cond
    [(empty? los) ...]
    [else (... (sexp-fn    (first los))
               (losexp-fn (rest los)))]))
```

(first los) is a **SexpOfX**. This is mixed data, so the last rule in the template recipe tells us we need to wrap it in a **(sexp-fn ...)** .

Sexp of Sardines

An Example of the **SexpOfX** pattern.

```
An SoSardines is either
-- a Sardine
-- a LoSSardines

A LoSSardines is either
-- empty
-- (cons SoSardines
         LoSSardines)
```

The Template for SoSardines

```
;; sosard-fn : SoSardines -> ??
(define (sosard-fn s)
  (cond
    [(sardine? s) ...]
    [else (lossard-fn s)]))

;; lossard-fn : LoSSardines -> ??
(define (lossard-fn los)
  (cond
    [(empty? los) ...]
    [else (... (sosard-fn   (first los))
               (lossard-fn (rest los)))]))
```

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

More Examples

```
;; number-of-strings : SoS -> Number
;; number-of-strings-in-loss : LoSS -> Number
;; returns the number of strings in the given sos or
   loss.

;; characters-in : SoS -> Number
;; characters-in-loss : LoSS -> Number
;; returns the total number of characters in the strings
   in the given sos or loss.

;; number-of-sardines : SoSardines -> Number
;; returns the total number of sardines in the given
   SoSardines.
```

Summary

- You should now be able to:
  - Give examples of S-expressions
  - Give 3 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 06-4-sos-and-loss.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.4
- Go on to the next lesson