Recursion on lists

CS135 Lecture 06

L06.0 Buying apples

A grocery list

```
(cons 'apple
      (cons 'eggs
            (cons 'bread
                  (cons 'apple
                        (cons 'milk
                              (cons 'bread empty))))))
```

How many apples should we buy? How many apples does the list contain?

Data definition

To make the problem clearer, let's write a data definition for a grocery list:

```
;; a Food is (anyof 'apple 'bread 'eggs 'milk)

;; a (listof Food) is one of:
;; * empty
;; * (cons Food (listof Food))
```

Our grocery list has a **recursive** data definition.

A list of food is **either** an empty list **or** a food item followed by a list of food.

Contract and header

We can now write a contract and header for a function that will count the number of apples we have to buy.

```
;; count-apples: (listof Food) -> Nat
(define (count-apples groceries) ...)
```

Notice that we aren't going to buy half-eaten apples and we aren't going to return apples to the store, so the function produces a **`Nat`**.

Body

The data definition gives us a hint about how to write the body of the function.

There are two cases:

1. The list is empty. We don't need to buy any apples.
2. The list is not empty. There are two possibilities for the first item.
    a. It's an apple. In which case we have to buy one apple, plus the number of apples on the rest of the list.
    b. It's not apple. In which case we have to buy the number of apples on the rest of the list.

Body

Translating the English to Racket:

```
;; Count the number of apples in a grocery list.
;; count-apples: (listof Food) -> Nat
(define (count-apples groceries)
  (cond [(empty? groceries) 0]
        [(symbol=? 'apple (first groceries))
         (add1 (count-apples (rest groceries)))]
        [else (count-apples (rest groceries))]))
```

Tests

```
(check-expect (count-apples empty) 0)

(define test0
  (cons 'apple
        (cons 'eggs
              (cons 'bread
                    (cons 'apple
                          (cons 'milk
                                (cons 'bread empty)))))))
(check-expect (count-apples test0) 2)
```

What else?

L06.1 Rules for recursion

Generalized list data definition

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

```
;; A (listof X) is one of:
;; ★ empty
;; ★ (cons X (listof X))
```

Replace **`X`** with a specific type such as **`Food`**, **`Int`**, or **`Point`**.

If **`X`** can be anything (the function does not care what the list contains) we use **`Any`**.

Template

The structure of a function manipulating a `(listof X)` directly follows from the data definition. We first handle the empty case, and then we can handle cases where the list is not empty.

```
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [... ...]
        [else (... (first lox)
                   (listof-X-template (rest lox)))]))
```

You can use this template as a starting point to write the body of a function processing a list.

Thinking about a list function

Here are four crucial questions to help think about functions consuming a list:

1. What should the function produce in the base case?
2. What should the function do to the first element in a non-empty list?
3. What should applying the function to the rest of the list produce?
4. How should the function combine #2 and #3 to produce the answer for the entire list?

When working with lists you should always recur on the **`rest`** of the list and check termination with **`empty?`**

Rules for recursion (second version)

1. Change one argument closer to termination while recurring. No other arguments can change.
2. When recurring on a natural number use `(sub1 n)` and test termination with `zero?`
3. When recurring on a list use `(rest lst)` and test termination with `empty?`

In this version, the argument in #1 is always the same argument.

Here, and elsewhere in the course, we use "`lst`" as the name of a generic list because "`list`" has special meaning in DrRacket, which we will discuss in a few lectures. We also sometimes use "`lon`" to mean list of `Nat`/`Num` and "`los`" to mean list of `Sym`, which provide extra clarity and readability.

Generalizing **`count-apples`**

The Rules for Recursion allow functions to have arguments that don't change. For example, we can write a function to count how often a target symbol occurs in a list.

```
;; Count the number of times a target symbol
;; occurs in a list of symbols.
;; count-symbol: Sym (listof Sym) -> Nat
(define (count-symbol target lst) ...)
```

Thinking about a recursive list function

1. **What should the function produce in the base case?**

The list is empty. The symbol does not appear in the list. Produce zero.

2. **What should the function do to the first element in a non-empty list?**

If it's the target symbol, count it (+1). If not, ignore it (+0).

3. **What should applying the function to the rest of the list produce?**

The number of times the target appears in the rest of the list.

4. **How should the function combine #2 and #3 to produce the answer for the entire list?**

Add the counts in #2 and #3.

Filling in the template

```
;; Count the number of times a target symbol
;; occurs in a list of symbols.
;; count-symbol: Sym (listof Sym) -> Nat
(define (count-symbol target lst)
  (cond [(empty? lst) ...]
        [(symbol=? target (first lst)) ...]
        [else (... (count-symbol target (rest lst)))]))
```

Start by getting our three cases (two in #2 and one in #3) into the template. The template can only be modified in ways that are consistent with the Rules for Recursion.

The first case to consider is always the empty list.

Filling in the template

```
;; Count the number of times a target symbol
;; occurs in a list of symbols.
;; count-symbol: Sym (listof Sym) -> Nat
(define (count-symbol target lst)
  (cond [(empty? lst) 0]
        [(symbol=? target (first lst))
         (add1 (count-symbol target (rest lst)))]
        [else (count-symbol target (rest lst))]))
```

Add the values produced by each case. It's okay to recurse twice in two different answers as long as the recursion follows the Rules. We almost never recurse in the question side of a question/answer pair.

Test cases

```
(check-expect (count-symbol 'apple empty) 0)
(define test0
  (cons 'apple
        (cons 'eggs
              (cons 'bread
                    (cons 'apple
                          (cons 'milk
                                (cons 'bread empty)))))))
(check-expect (count-symbol 'apple test0) 2)
(check-expect (count-symbol 'fish test0) 0)
```

What else?

L06.2 Length of a list

Purpose, contract and header

```
;; Compute the length of a list
;; len: (listof Any) -> Nat
(define (len lst) ...)
```

The function works with any list, so it consumes a `(listof Any)`.

DrRacket has a built-in function called **`length`**, which does exactly this, so we can't use that name. After this lecture, you can use the built-in function (unless, for some reason, we say you can't on an assignment or exam).

Thinking about a list function

1. **What should the function produce in the base case?**

The list is empty. The length is zero.

2. **What should the function do to the first element in a non-empty list?**

Count it. +1

3. **What should applying the function to the rest of the list produce?**

The length of the rest of the list.

4. **How should the function combine #2 and #3 to produce the answer for the entire list?**

Add them together.

Filling in the template and adding test cases

```
;; Computer the length of a list
;; len: (listof Any) -> Nat
(define (len lst)
  (cond [(empty? lst) 0]
        [else (add1 (len (rest lst)))]))

(check-expect (len empty) 0)
(check-expect (len (cons 'a (cons 'b empty))) 2)
(check-expect (len (cons 1 (cons 'a (cons true empty)))) 3)
```

List idioms

There are an unlimited number of ways that we might wish to process the information in a list. However, after working with lists in a lot of practical contexts, we start to see some common themes emerge: standard styles of processing that come up again and again. We'll refer to these as list processing **idioms**.

In this lecture and the next, we'll continue to practice writing list functions, with an emphasis on exploring the three most important list idioms: **folding** a list, **mapping** a list, and **filtering** a list.

So far, we have been folding a list.

List idiom - Folding a list

Folding a list (sometimes also called "reducing a list") is any process where we consume a list and boil all of its contents down to a single value, which often summarizes, combines, or otherwise accumulates the information of all of the individual list elements. Both `count-symbol` and `len` are examples of folding.

When we fold a list, we start with a "base" value, a foundation upon which the folding occurs, and which will serve as the result of folding the empty list. We also choose a process for "combining" each element of the list with the base, one at a time. Those successive combinations produce ever more complete answers to sub-problems on parts of the list, so that by the time we're done we have the answer for the whole list.

L06.3 Predicates over lists

Does a list of numbers contain any non-positive values?

```
;; Does a list of numbers contain only positive values?
;; all-positive?: (listof Num) -> Bool
(define (all-positive? list)...)

(check-expect
  (all-positive? (cons 1 (cons 2 (cons 3 empty)))) true)
(check-expect
  (all-positive? (cons 1 (cons -2 (cons 3 empty)))) false)
(check-expect (all-positive? empty) true)
```

Thinking about a list function

1. **What should the function produce in the base case?**

The list is empty. It does not contain any non-positive numbers. Produce true.

2. **What should the function do to the first element in a non-empty list?**

Is it non-positive? If so, we can "short circuit" and produce false.

3. **What should applying the function to the rest of the list produce?**

Produce true if the rest of the list is entirely positive; produce false otherwise

4. **How should the function combine #2 and #3 to produce the answer for the entire list?**

The "short circuit" in #2 avoids the need to combine results.

Filling in the template

```
;; Does a list of numbers contain only positive values?
;; all-positive?: (listof Num) -> Bool
(define (all-positive? list)
  (cond [(empty? list) true]
        [(<= (first list) 0) false]
        [else (all-positive? (rest list))]))
(check-expect
  (all-positive? (cons 1 (cons 2 (cons 3 empty)))) true)
(check-expect
  (all-positive? (cons 1 (cons -2 (cons 3 empty)))) false)
(check-expect (all-positive? empty) true)
```

No, you can't use the `member?` built-in list predicate

Yes, it's in the documentation. No, you can't use it. No need to ask.

If you think you need it to solve a homework problem, you need to think about how you can approach the problem differently. Please ask us for help.

`member?` is one of many built-in functions that test equality with `equal?`

In CS135 we want you to understand the types of the data you are working with and `equal?` works with anything (e.g., arbitrarily nested lists of lists) doing deep structural comparisons. At this point it's too magical, and as we start to work with more-and-more complex lists structures it can even lead you far astray.

Writing your own membership predicate

Here are some problems you can try solving to test your understanding of the material so far.

1. Write a function **`contains-symbol?`** that consumes a target symbol and list of symbols and produces true if the target is contained in the list.
2. Write a function **`contains?`** that works for a list of symbols and numbers. How do you test for equality if **`(first lst)`** could be a symbol or a number?

Now, when you want to use **`member?`** in an assignment, you can just paste in your version of **`contains?`** As we add more data types you may want to keep it updated.

If you are reading this sentence while studying for an exam, and you haven't tried these problems, we suggest you try them. If you can't solve them, ask for help.

Using DrRacket documentation

DrRacket document is available at https://docs.racket-lang.org/htdp-langs/

We are currently working at the "Beginning Student" level, but we will eventually reach the "Intermediate Student with Lambda" level. As we introduce more-and-more language constructs, the documentation may help to remind you about how to use these constructs.

You can also ask Google, ChatGPT, etc, for help but they don't understand the limitations we place on you in CS135 (e.g., they may suggest **`member?`**).

If the documentation says that a function uses **`equal?`** there's no need to ask if you can use the function. You can't. It's possible that we will accept some of the other functions, but more likely we will ask you to stick to the allowed constructs.

Lecture 6 Summary

L06: You should know

- How to write recursive functions that fold lists using the design recipe, templates, and the Rules for Recursion (second version).
- How to write predicates over lists using the design recipe, templates, and the Rules for Recursion (second version).
- No, you can't use **`member?`** or **`equal?`**

L06: Allowed constructs

Newly allowed constructs:
`add1 length listof`
Rules for Recursion (second version)

Previously allowed constructs:
```
( ) [ ] + - * / = < > <= >= ;
abs acos and asin atan boolean? check-expect check-within
cond cons cons? cos define e else empty empty? exp expt
false first inexact? integer? list? log max min not number?
or pi quotient rational? remainder rest sin sqr sqrt sub1
symbol? symbol=? tan true zero? Any anyof Bool Int Nat Num
Rat Sym
```

Recursion **must** follow the Rules for Recursion (second version)