Producing lists

CS135 Lecture 07

L07.0 Eating apples

We bought some groceries

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

Let's eat the apples, i.e., we want to produce an equivalent list with the symbol **`'apple`** removed.

We call this a *filter*. We are *filtering* the list to remove apples.

Purpose, contract, and header

```
;; Filter a list of symbols to remove the symbol 'apple
;; eat-apples: (listof Sym) -> (listof Sym)
(define (eat-apples lst) ...)
```

Test cases

```
(check-expect (eat-apples empty) empty)

(define test0
  (cons 'apple
        (cons 'eggs
              (cons 'bread
                    (cons 'apple
                          (cons 'milk
                                (cons 'bread empty)))))))

(define result0
  (cons 'eggs (cons 'bread (cons 'milk (cons 'bread empty)))))


(check-expect (eat-apples test0) result0)
```

Remove all apples from a list

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

The list is empty. It does not contain any apples. Produce the empty list.

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

If it's an apple, ignore it. It it's not an apple, we need retain it in the filtered list.

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

The rest of the list filtered to remove the apples.

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

If we are retaining the symbol in #2, we `cons` *it to the filtered list from #3.*

Filling in the template

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(symbol=? 'apple (first lst))
         (eat-apples (rest lst))]
        [else (cons (first lst) (eat-apples (rest lst)))]))
```

Generalizing **`eat-apples`** to eat anything

```
;; Filter a list of symbols to remove a target symbol
;; eat-symbol: Sym (listof Sym) -> (listof Sym)
(define (eat-symbol target lst)
  (cond [(empty? lst) empty]
        [(symbol=? target (first lst))
         (eat-symbol target (rest lst))]
        [else
         (cons (first lst)
               (eat-symbol target (rest lst)))]))
```

List idiom - Filtering a list

Filtering is another common list idiom, which is easier to describe than folding. We start with a list, together with a property we'd like to test for every element of that list. Our goal is to produce a sublist of the original list, consisting of just those elements that have the property.

For example, if we're filtering for positive numbers, then filtering the list

```
    (cons 4 (cons -3 (cons -8 (cons 1 empty))))
```

would produce

```
    (cons 4 (cons 1 empty))
```

If we were instead asking for even numbers, we'd end up with

```
    (cons 4 (cons -8 empty))
```

Keeping even numbers in a list of numbers

```
(define (keep-even lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (keep-even (rest lst))]
        [else (cons (first lst) (keep-even (rest lst)))]))

(check-expect
 (keep-even (cons 4 (cons -3 (cons -8 (cons 1 empty)))))
            (cons 4 (cons -8 empty)))
```

Note: **`odd?`** is a built-in predicate; there is also a built-in **`even?`**

L07.1 Apples to oranges

Transforming a list

We decide that we want to buy oranges instead of apples. We want to transform our grocery list to replace all apples with oranges.

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

$\downarrow$

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

Replacing all apples with oranges

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

The list is empty. It does not contain any apples. Produce the empty list.

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

If it's an apple, replace it with an orange. Otherwise, retain it in the filtered list.

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

The rest of the list filtered to replace the apples with oranges.

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

***`cons`** the symbol from #2 onto the filtered list from #3.*

Apples to oranges

```
;; Transform a list of symbols by replacing
;; all occurrences of 'apple with 'orange
;; apples-to-oranges: (listof Sym) -> (listof Sym)
(define (apples-to-oranges lst)
  (cond [(empty? lst) empty]
        [(symbol=? 'apple (first lst))
         (cons 'orange
               (apples-to-oranges (rest lst)))]
        [else
         (cons (first lst)
               (apples-to-oranges (rest lst)))]))
```

List idiom - Mapping a list

Transforming a list in this way is also called “mapping a list”.

We have a list and some sort of transformation operation, and we want to produce a new list of the same length, where each value from the original list has had the transformation applied to it.

For example, if we want to map the built-in sqr function over the list

```
    (cons 2 (cons 4 (cons 5 empty)))
```

we would expect to get back

```
    (cons 4 (cons 16 (cons 25 empty)))
```

Mapping lists of numbers

We can map lists of numbers to lists of numbers by applying mathematical operations to each element of the list and producing the result.

For example, we could add one to each element of a list.

```
(cons 10 (cons -6 (cons 999 empty)))
```

$\downarrow$

```
(cons 11 (cons -5 (cons 1000 empty)))
```

Add one to each element of a list of numbers

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

The list is empty. There is nothing to transform. Produce the empty list.

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

Add one to the first element of the list.

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

Call the function recursively to add one to the elements in the rest of the list.

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

***`cons`** the value from #2 onto the transformed list from #3.*

Add one to each element of a list of numbers

```
;; Add one to each element of a list
;; add1-list: (listof Num) -> (listof Num)
(define (add1-list lst)
  (cond [(empty? lst) empty]
        [else (cons (add1 (first lst))
                    (add1-list (rest lst)))]))

(check-expect (add1-list empty) empty)
(check-expect
 (add1-list (cons 10 (cons -6 (cons 999 empty))))
 (cons 11 (cons -5 (cons 1000 empty))))
```

L07.2 Ordered lists

Ordered lists

Lists are more than just a collection, or set, of elements. The elements can have properties relative to one another. For example, a list can be ordered.

Increasing or “ascending” order:

```
(cons 10 (cons 20 (cons 30 empty)))
```

Decreasing or “descending” order:

```
(cons 64 (cons 53 (cons 42 (cons 42 (cons 20 empty)))))
```

We say “strictly increasing” or “strictly decreasing” if elements can’t be equal.

Predicate to check if a list is in ascending order

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

The list is empty. The list is ordered. Produce **`true`**.

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?**

`true`, *if the rest of the list in ascending order;* ***`false`***, *otherwise.*

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

`and` *the values from #2 and #3 (could "short circuit").*

Predicate to check if a list is in ascending order

For this problem, we have two base cases:

1. If the list is empty, the list is ordered.
2. If the rest of the list is empty, the list is ordered.

If neither of these conditions is true, then we can compare the first element of the list, with the second element of the list. We know that the second element exists because the rest of the list is not empty.

Predicate to check if a list is in ascending order

```
;; Check that a list of numbers is in ascending order
;; ascending?: (listof Num) -> Bool
(define (ascending? lst)
  (cond [(or (empty? lst) (empty? (rest lst))) true]
        [(> (first lst) (first (rest lst))) false]
        [else (ascending? (rest lst))]))
```

Notice that the base cases are combined and that we “short circuit” if the first element is greater than the second element.

Change **>** to **>=** if we want the list to be in *strictly increasing* order.

Test cases

```
(check-expect (ascending? empty) true)
(check-expect (ascending? (cons -10 empty)) true)

(check-expect
 (ascending? (cons 10 (cons 20 (cons 30 empty)))) true)

(check-expect
 (ascending?
  (cons 64 (cons 53 (cons 42 (cons 31 (cons 20 empty))))))
 false)

(check-expect (ascending? (cons 1 (cons 1 empty))) true)
```

Inserting into an ordered list

Normally we add an element to a list with `cons`, but suppose we want to insert an element into a list while maintaining some property, such as ascending order.

What happens if we insert into an empty list?

What happens if the element is less than the first of the list?

What happens if the element is greater than the first of the list?

(any other cases?)

Inserting into an ordered list

```
;; Insert a number into an ordered list
;; insert: Num (listof Num) -> (listof Num)
;; Requires: values must be in ascending order

(define (insert n lst)
  (cond [(empty? lst) (cons n empty)]
        [(< n (first lst)) (cons n lst)]
        [else (cons (first lst) (insert n (rest lst)))]))
```

How do we test this function?

The “requires” indicates the special limitations of this function.

Test cases

```
(check-expect (insert -1 empty) (cons -1 empty))
(check-expect
  (insert 3 (cons 1 (cons 2 (cons 4 (cons 5 empty)))))
  (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty))))))
(check-expect
  (insert 0 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
  (cons 0 (cons 1 (cons 2 (cons 3 (cons 4 empty))))))
(check-expect
  (insert 5 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
  (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty))))))
```

L07.3 Lastly

`first`, **`second`**, **`third`**, **`fourth`**, …

`first`	Consumes a list with length ≥ 1. Produces the first value in that list.
`second`	Consumes a list with length ≥ 2. Produces the second value in that list.
`third`	Consumes a list with length ≥ 3. Produces the third value in that list.
`fourth`	Consumes a list with length ≥ 4. Produces the fourth value in that list.

…and so on to **`eighth`**, but you probably will never need more than **`third`**.

In the list of allowed constructs, we limit you to **`first`**, **`second`**, and **`third`**. If you want to use **`fourth`** or higher, you might want to think again about your approach. It’s probably not the simplest way to solve the problem.

How do we find the last element in a non-empty list?

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

If the rest of the list is empty, produce the first (and therefore the last) element in the list.

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

By #1, the rest of the list is not empty, so we can ignore the first element.

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

The last element in the rest of the list.

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

Since we ignore the first of the list in #2, we just produce the element from #3.

Finding the last element in a non-empty list.

```
;; Produce the last element of a non-empty list
;; last: (listof Any) -> (listof Any)
;; Requires: list should be non-empty
(define (last lst)
  (cond [(empty? (rest lst)) (first lst)]
        [else (last (rest lst))]))
```

It’s essential for the purpose to indicate that the list must be non-empty.

It makes no sense for `last` to work on an empty list. It’s okay under the Rules for Recursion for the base case to be `(empty? (rest lst))`. You should add a “requires” under a contract to indicate special limitations of a function.

Lecture 07 Summary

L07: You should know

- How to filter lists.
- How to transform elements of a list with the map idiom.
- How to check that a list has a defined ordering property, e.g. ascending
- How to insert into an ordered list
- How to use the built-in functions **`second`**, and **`third`**.
- How to find the last element of a list.

L07: Allowed constructs

Newly allowed constructs:
`even? odd? second third`

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and asin atan boolean? check-expect
check-within cond cons cons? cos define e else empty empty?
exp expt false first inexact? integer? length list? log max
min not number? odd? or pi quotient rational? remainder rest
sin sqr sqrt sub1 symbol? symbol=? tan true zero?
listof Any anyof Bool Int Nat Num Rat Sym
```

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