CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 10

The elements of a list can be any type of Scheme expression.

- So far we've mostly avoided processing lists that contain other lists as elements.
- We'll address that today.

To indicate that a list is an element of some other lists, we'll refer to it as a ***sublist***.

Thus, the list

```
(1 (2 3 (4 5)) 6 (7) ())
```

has five elements, three of which happen to be lists in their own right:

```
(2 3 (4 5))
(7)
()
```

We often refer to the ***top-level structure*** of a nested lists.

- This just means thinking about the elements of that list, regardless of whether those elements happen to be lists themselves.
- So, the top-level structure of `(1 (2 3 (4 5)) 6 (7) ())` is simply that it's a list containing five elements, three of which happen to be lists.

Like with "flat lists", it's much easier to deal with nested lists of arbitrary size using recursion.

In fact, it's only a *tiny* bit more complicated to do "deep" recursion than it is to do "flat" recursion!

Flat recursion

A call to (***procedure*** `lst`) may lead to a recursive call to (***procedure*** `(rest lst)`).

The calls form a linear tree.

Each tree node has zero or one child nodes.

Example: The `length` procedure.

Length of a list

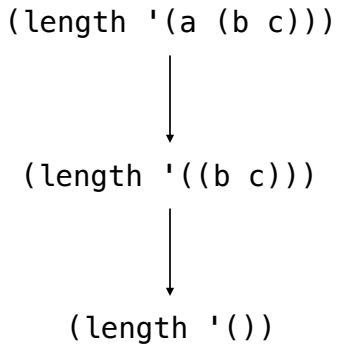
```
(define length (lambda (lst) ...?...))
```

```
> (length '(a b c))
3
> (length '(a (b c)))
2
> (length '())
0
```

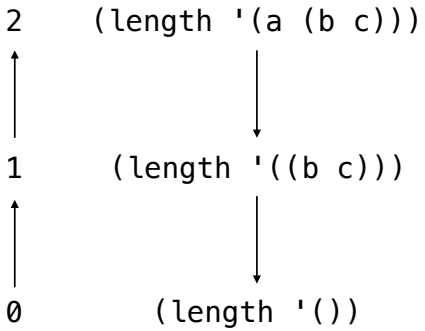
Length of a list

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (rest lst))))))
```

Procedure call tree



Parent node value results from incrementing child node value.



Counting the items in an expression

```
(define item-count (lambda (x) ...?...))
```

```
> (item-count '(a b c)
3
> (item-count '(a (b c)))
3
> (item-count 'a)
1
> (item-count '())
0
```

Counting the items in an expression

```
(define item-count
  (lambda (x)
    (cond ((null? x) 0)
          ((not (list? x)) 1)
          (else (+ (item-count (first x))
                   (item-count (rest x)))))))
```

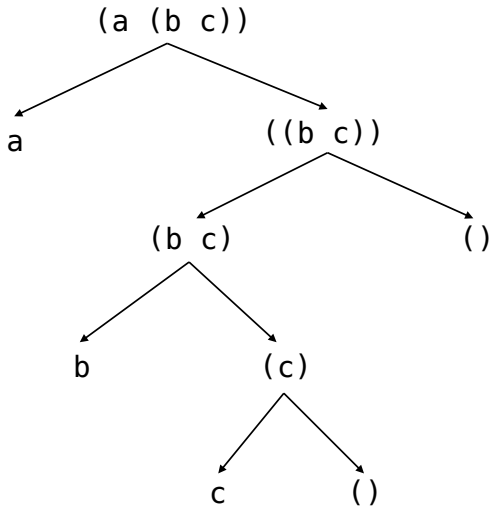
Data-driven recursion

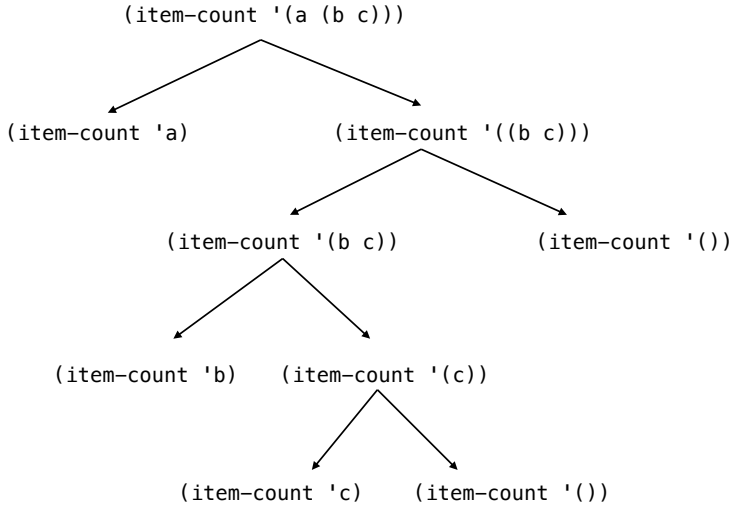
The data has the structure of a tree.

The calls to a recursive procedure have the structure of a tree.

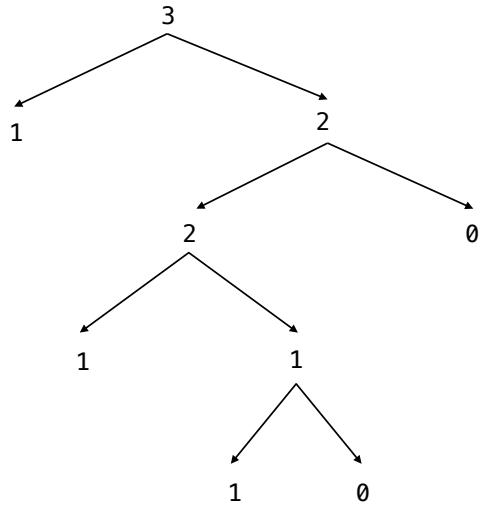
The two trees have the *same* structure.

Parent is the result of consing the left child into the right child.





Parent is the result of adding the left child to the right child.



Deep recursion

A call to (***procedure*** `lst`) may lead to a recursive call to (***procedure*** `(first lst)`) and (***procedure*** `(rest lst)`).

The calls form a *binary tree* – each tree node has zero or two child nodes.

Example: The `item-count` procedure.

cons cells and deep recursion

Compare the `cons` cell representation of `(a (b c))` to the tree we used to analyze the `item-count` procedure on `(a (b c))`.

They are both trees, and the trees have the same structure.

In deep recursion, the recursive procedure is called once on each `cons` cell.

Recursion stops at the data that are not in `cons` cells.

Recall how we've used flat recursion to square all the elements of a flat list:

```
(define square (lambda (x) (* x x)))
(define square-all
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (square (first lst))
              (square-all (rest lst))))))
(square-all '(1 2 3 4))
```

Now let's do the *deep recursion* version, `square-all*`.

> We'll often – though not always – use * in a function name to indicate that it works on nested lists.

```
(define square-all*
  (lambda (lst)
    (cond
      ;; Base case: lst is empty
      ((null? lst)
       '())

      ;; Recursive case 1:
      ;;   First element is a list
      ((list? (first lst))
       (cons (square-all* (first lst))
             (square-all* (rest lst))))

      ;; Recursive case 2:
      ;;   First element is not a list
      (else
       (cons (square (first lst))
             (square-all* (rest lst)))))))

(square-all* '(1 (2) (3 (4 5) 6) () (7)))
```

Is an item a member of a list?

```
(define member? (lambda (item lst) ...?...))
```

```
> (member? 'b '(a b (c) d))
#t
> (member? 'c '(a b (c) d))
#f
> (member? 'groucho '())
#f
```

Is an item a member of a list?

```
(define member?
  (lambda (item lst)
    (and (not (null? lst))
         (or (equal? item (first lst))
             (member? item (rest lst))))))
```

Is an item found anywhere within an expression?

```
(define within? (lambda (item x) ...?...))
```

```
> (within? 'b '(a b (c) d))
#t
> (within? 'c '(a b (c) d))
#t
> (within? 'c 'c)
#t
> (within? 'groucho '())
#f
```

Is an item found anywhere within an expression?

```
(define within?
  (lambda (item x)
    (or (equal? item x)
        (and (pair? x)
             (or (within? item (first x))
                 (within? item (rest x)))))))
```

Appending two lists

```
(define append (lambda (lst1 lst2) ...?...))
```

```
> (append '(a b c) '(d e f))
'(a b c d e f)
> (append '() '(a b c))
'(a b c)
> (append '(a b c) '())
'(a b c)
```

Appending two lists

```
(define append
  (lambda (lst1 lst2)
    (if (null? lst1)
        lst2
        (cons (first lst1)
              (append (rest lst1)
                      lst2)))))
```

Flatten an expression

```
(define flatten (lambda (x) ...?...))
```

```
> (flatten '(a (b) c))
'(a b c)
> (flatten '(a b c))
'(a b c)
> (flatten 'z)
'(z)
> (flatten '())
'()
```

Flatten an expression

```
(define flatten
   (lambda (x)
     (cond ((null? x)
            '())

           ((not (pair? x))
            (list x))

           (else
            (append
              (flatten (first x))
              (flatten (rest x)))))))
```

The depth of an expression

```
(define depth* (lambda (x) ...?...))


> (depth* 'b)
0 ;; atomic element

> (depth* '())
0 ;; atomic element

> (depth* '(a b c))
1 ;; all elements at depth 1

> (depth* '(1 (2 (3 (4 (5))))))
5
```

The depth of an expression

```
(define depth*
  (lambda (x)
    (cond
      ;; Base case: list is empty
      ((null? x) 0)

      ;; Recursive case 1:
      ;;   First item is a non-empty list
      ((pair? (first x))
       (max (+ 1 (depth* (first x)))
            (depth* (rest x))))

      ;; Recursive case 2:
      ;;   First item is atomic, including
      ;;   when it's the empty list.
      (else
       (max 1 (depth* (rest x)))))))
```


Acknowledgments

This lecture incorporates material from:

Tom Ellman