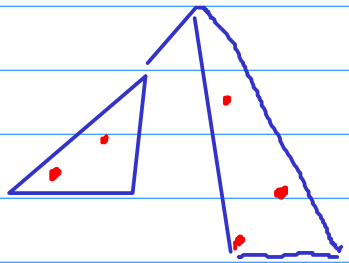
CSc 335 Class 18 April 2 2024

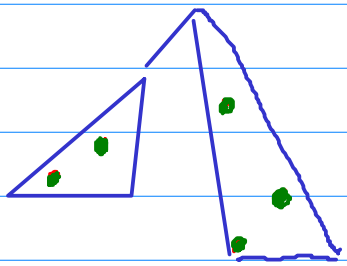
① Resume with subst* as an example of tree recursion.



Replace each occurrence of old by new (both old and new are atoms)

↳ actually new could perfectly well be an arbitrary tree including ()?

a pattern in which the recursive calls are on the car and cdr — ie - The divide and conquer decision is to work on the car and the cdr separately.



```
(define (subst* tree old new)
  (cond ((null? tree) tree)
        ((atom? tree) (if (eq? tree old) new tree))
        (else (cons (subst* (car tree) old new)
                    (subst* (cdr tree) old new)))
  ))
```

What to do with the results of the recursive calls?

By the post, we know subst* is to return a tree. And from that we know that the recursive calls return trees — so the question becomes: how do we put two trees together to get another tree. The only option: cons.

Since (car tree) and (cdr tree) are proper components of tree, we may assume that the recursive calls work.

(2) a restricted form of equal? introduced as another example of tree recursion

```
(define same-shape?
  (lambda (l1 l2)
    (cond
      ((or (not (pair? l1)) (not (pair? l2))) (eq? l1 l2))
      (else
       ; both l1 and l2 are pairs
       (and (same-shape? (car l1) (car l2))
            (same-shape? (cdr l1) (cdr l2)))))))
```

The goal here is to identify when l_1 and l_2 have the same structure as trees.

Their cars must (recursively) have the same structure, and their cdrs must (recursively) have the same structure

if either l_1 or l_2 is not a pair, then they must be eq?

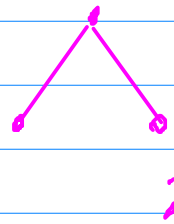
eg: if $l_1 = 2$ and $l_2 = '(2)$ then (eq? l_1 l_2) = #f, and surely l_1 and l_2 have different structures

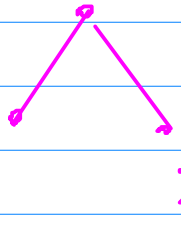
But stronger, since eq (same-shape? a b) = #t for atoms a and b only if (eq? a b).

So (same-shape? t (subst* t a b)) = #f

(even though t and (subst t a b) are "tree-isomorphic" — have the same shape as trees, at every level, they are not identical)

↳ WAIT a minute! For

$t_1 =$ 

and $t_2 =$ 

, we do have (same-shape? t_1 t_2) = #t, even though t_1 and t_2 occupy different memory locations. [ie, (eq? t_1 t_2) = #f]

So we mean Logically Identical, as opposed to Physically Identical.

This is what eq? worries about.

(3) A simple interpreter — simple, yes, but it serves as a template for all the interpreters yet to come in 335.

The goal is to produce an evaluator / a calculator for a class of algebraic expressions — ie — for a particularly simple programming language.

As you might expect, we begin by defining the syntax of the programming language — ie — by defining the class Aexp of algebraic expressions. I say this is expected — because once we have the syntax — the structure of Aexp — we more or less automatically know how to write the interpreter/evaluator/calculator.

Let's say that Aexp is the least class containing non-negative scheme integers which is closed under the operations of @, # and !, defined as follows;

if $e_1 \in$ Aexp and $e_2 \in$ Aexp, then

$(e_1 @ e_2) \in Aexp$

$(e_1 \# e_2) \in Aexp$

$(e_1 ! e_2) \in Aexp$

This is just syntax! Nothing at all has been said about the meanings of @, #, !

Some examples of expressions belonging to Aexp

0, 1, 2, ...

(1 @ 2), (1 # 2), ...

(1 @ (1 # 2)), ...

In BNF: Aexp ::= non-neg scheme ints | (Aexp @ Aexp) | (Aexp # Aexp) | (Aexp ! Aexp)

going back to the discussion of structural induction

The proper components of, say,

((1 ! (2 # 3)) @ (4 ! 5))

are:

1, 2, 3, 4, 5

(1 ! (2 # 3))

(2 # 3)

(4 ! 5)

Note that (1 ! is NOT a component — components must themselves be well-formed — ie - must belong to Aexp.

It's useful to set up Aexp as a data structure. Data structures have constructors, selectors, and classifiers — here, we use these to define a user-interface:

constructors:

(define (make-@ e1 e2)

```
(list e1 '@ e2))
```

our current low-level representation as a list.

```
(define (make-# e1 e2)
  (list e1 '# e2))

(define (make-! e1 e2)
  (list e1 '! e2))
```

Selectors

```
(define (first-operand e)
  (car e))

(define (second-operand e)
  (caddr e))

(define (operator e)
  (cadr e))
```

(e1 # e2)

classifiers

```
(define (#-exp? e)
  (eq? (operator e) '#))

(define (@-exp? e)
  (eq? (operator e) '@))

(define (!-exp? e)
  (eq? (operator e) '!))
```