## Non-empty lists

What's wrong with max-of-nelon?

*We wrote this expression twice*

```
;; max-of-nelon:  nelon -> number
(define  (max-of-nelon  a-nelon)
   (cond
      [(empty?  (rest a-nelon))          (first a-nelon) ]
      [(cons?    (rest a-nelon))
       (cond
           [( >= (first a-nelon) (max-of-nelon (rest a-nelon)) ) (first a-nelon)]
           [else (max-of-nelon (rest a-nelon)) ]
       ) ]
   ))
```

*In this form, max-of-nelon takes time proportional to* $2^{(\text{length a-nelon})}$

*⇒ Efficiency is not an objective, but this is a major waste of time*

## Non-empty lists

How bad can it get?

- Let's try it
- (max (list 1 2 3 4 5 6)) 1
  - → Recurs twice on (list 2 3 4 5 6) 2
  - → Each of those recurs twice on (list 3 4 5 6) 4
  - → Each of those recurs twice on (list 4 5 6) 8
  - → Each of those recurs twice on (list 5 6) 16
  - → Each of those recurs twice on (list 6) 32
  - → Phew! This is getting ridiculous ⇒ 63
- It's a little better if the list is not in order, but …
  - → List of length n calls <u>max</u> $2^n - 1$ times
  - → This is <u>too</u> much
  - → List of length 7 would take 127 calls, 8 would take 255, …

## *What's the answer?*

Need a new (for COMP 210) idea

- Save the value of <u>max-of-list</u>
- Makes it recur only once
- (max (list 1 2 3 4 5 6)) 1
  - → Recurs once on (list 2 3 4 5 6) 1
  - → Recurs once on (list 3 4 5 6) 1
  - → Recurs once on (list 4 5 6) 1
  - → Recurs once on (list 5 6) 1
  - → Recurs once on (list 6) 1
  - → And is done ⇒ 6
- Reduces work to n calls for list of length n
  - → Exponential savings in work are always worth pursuing

## *Local*

How can we preserve value of (max (rest a-nelon))?

- Need a new construct – Scheme's <u>local</u> expression

*WARNING: set language level in Dr. Scheme to intermediate*

Local

- Takes two complicated arguments
  - → List of definitions
  - → An expression
- (local [ (definitions) ] (expression))
- Evaluates the expression <u>*in the context*</u> of the definitions

## *Local*

We said that local

"Evaluates the expression in the context of the definitions"

1. It creates a new <u>*scope*</u> –
   - Think of this as a box that can hold objects in Scheme world
   - Can see out of the box from inside
   - Cannot see into the box from outside
2. Evaluates all the definitions inside the box
   - Create new objects and new results, by normal evaluation
3. Evaluates the expression inside the box
   - Uses objects inside the box
4. Replaces the local with the result, discarding the box

## *Local*

Rewriting max-of-nelon with local

```
;; max-of-nelon:  nelon -> number
(define  (max-of-nelon  a-nelon)
   (cond
      [(empty?  (rest a-nelon))          (first a-nelon) ]
      [(cons?    (rest a-nelon))
       (local
           [ (define maxrest (max-of-nelon (rest a-nelon))) ]
           (cond
               [( >= (first a-nelon) maxrest ) (first a-nelon)]
               [else maxrest]
           ) ) ]     ;; closing the (cons? ) clause
   ))
```

*Evaluates* (max-of-nelon (rest a-nelon)) *once, but uses it twice*

## Local

*Scheme world*

```
(max-of-nelon (list 1 2 3 4 5 6))
```

*Type the code and this example into the definitions window in Dr. Scheme.*

*Use the stepper to run through the example.*

*Pay attention to renaming of "maxrest" each time a local is evaluated in the recursion.*

## Local

What was it really doing?

- Executed a local for each element of the list
- Created a nest of n boxes (or scopes)
- Each scope defines maxrest as largest element found by the computation in an inner box

What happened to all the scopes?

1. Dr. Scheme creates a unique name for each define in the local
2. Dr. Scheme rewrites the local using those new names
3. Those names are never used outside the local
4. Nothing can ever refer to their value

This is how Dr. Scheme actually implements it

*Another process recycles space for unusable names*

## Another Example

```
;; exp-5: number -> number
;; Purpose: compute the fifth power of the input number
(define  (exp-5 x)
   (local
      [ (define (square y) ( * y y))
        (define (cube z)    (* z  (square z))) ]
       (* (square x)  (cube x))
    ))
```

What happens?

- In intermediate language level, we click execute
- Try  (exp-5  2)

## Another Example

(exp-5  2)

- Creates box, with functions square and cube
- Evaluates (* (square 2) (cube 2))

```
→(* (square 2) (cube 2))
→(* (*  2  2) (*  2  (square 2)) )
→ (* (*  2  2) (*  2  (*  2  2)) )
→ (* (*  2  2) (*  2  4) )
→ (* (*  2  2)  8)
→ (* 4  8)
→ 32
```

*Type this one into Dr. Scheme and run it with the stepper.*

## *Another Example*

What about (cube 3) ?

- It fails, because cube has no definition outside the local
- Its name was rewritten with some unique string
- We don't know that name *(& Dr. Scheme cannot tell us)*

What about (+ 3 (exp-5 2))?

- Copies (+ 3 (… body of exp-5 with 2 substituted for x …))
- Follows all those steps
- Replaces (exp-5 2) with 32
- Performs the addition to yield 35

*Try this one in Dr. Scheme with the stepper, too!*

## *Local*

So what good is local?

- Sped up max-of-nelon *($2^n$-1 calls to n-1 calls)*
- When should we use it? *(210 has all these rules!)*

Use a local when

- It lets the program compute a complicated value once instead of multiple times
- It makes a complicated expression more readable
  - → Use it to introduce private helper functions like square & cube
  - → Break up complex expression into tractable parts

We will see more uses for local in the next couple of weeks