Local

Last lecture, we introduced a new Scheme construct, "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

Introduced rewriting rules for local

(local [(definitions)] (expression))

1. Dr. Scheme creates a unique name for each name defined in the local
2. Dr. Scheme rewrites the entire body of the local using those new names
3. Dr. Scheme evaluates the (expression) part of the local
4. Dr. Scheme replaces the local with the result

This replacement destroys every copy of the local names

Yet Another Example

```
;; IsIn?:  list-of-symbol   symbol -> boolean
;; Purpose: returns true of the symbol appears in the list and false
;;                if it is not in the list
(define  (IsIn? a-los key)
   (cond
      [(empty? a-los)             false]
      [(cons?   a-los)
       (or   (symbol=? (first a-los) key)
             (IsIn?  (rest a-los) key) )]
    )
)
```

Key is never changed

Key is used

We can use local to ellide this invariant (unchanging) parameter and to simplify IsIn?

(Another use for local)

Follows classic list template

- Empty list ⇒ return false
- Non-empty list ⇒ check (first a-los) & recur on (rest a-los)
- What can we complain about?

Yet Another Example

```
;; IsIn?:  list-of-symbol   symbol -> boolean
;; Purpose: returns true of the symbol appears in the list and false
;;                if it is not in the list
(define  (IsIn? a-los key)
   (local
      [ (define (Search the-list)
            (cond
               [(empty? the-list)             false]
               [(cons?   the-list)
                (or     (symbol=? (first the-list) key )
                        (Search (rest the-list)) ) ]
            )) ]
        (search a-los)
    )
)
```

Parameter to IsIn?, but not to Search

This version passes fewer parameters
⇒ Cleaner interface
⇒ Faster execution

Yet Another Example

```
;; IsIn?:  list-of-symbol   symbol -> boolean
;; Purpose: returns true of the symbol appears in the list and false
;;              if it is not in the list
(define  (IsIn? a-los key)
   (local
      [ (define (Search the-list )
           (cond
              [(empty? the-list )          false]
              [(cons?   the-list )
               (or     (symbol=? (first the-list ) key )
                       (Search (rest the-list)) ) ]
           )) ]
        (search a-los)
    )
)
```

What happens if we write this as "a-los", rather than "the-list"?

Yet Another Example

```
;; IsIn?:  list-of-symbol   symbol -> boolean
;; Purpose: returns true of the symbol appears in the list and false
;;              if it is not in the list
(define  (IsIn? a-los key)
   (local
      [ (define (Search a-los )
           (cond
              [(empty? a-los )             false]
              [(cons?   a-los )
               (or     (symbol=? (first a-los ) key )
                       (Search (rest a-los)) ) ]
           )) ]
        (search a-los)
    )
)
```

How does Scheme resolve these different references to a-los?

Which refer to IsIn?'s parameter?

Which refer to Search's parameter?

Apply the rewriting rules …

(shift to Dr. Scheme & run lecture18a.scm in the stepper)

Yet Another Example

```
;; IsIn?:  list-of-symbol   symbol -> boolean
;; Purpose: returns true of the symbol appears in the list and false
;;               if it is not in the list
(define  (IsIn? a-los key)
   (local
       [ (define (Search a-los )
            (cond
               [(empty? a-los )             false]
               [(cons?   a-los )
                (or     (symbol=? (first a-los ) key )
                         (Search (rest a-los)) ) ]
            )) ]
        (search a-los)
    )
)
```

The parameter __a-los__ to Search occludes the parameter __a-los__ to IsIn?

An expression sees the closest occurrence of a-los

Lexical Scoping Rules

Names are defined inside a scope

- A procedure definition creates a scope
  - → Scope of a procedure is its entire body
  - → Procedure's parameters are visible throughout its scope
- A local expression contains its own scope
  - → The scope of a local covers both the *__definition__* part and the *__expression__* part
  - → Any name defined in the *__definition__* part is visible throughout the entire local (the definition part & the expression part)
- Local inside a procedure
  - → The scopes nest, in order of appearance (*lexical order*)
  - → Local inside ⇒ local names prevail

Nesting Locals

One last example

- Develop a program to intersect two lists of symbols
- Consumes two lists & produces a list containing their common elements
  - → Familiar operation from set theory
  - → Common operation in many contexts *(sets are fundamental)*

```
;; Intersect:  list-of-symbol   list-of-symbol -> list-of-symbol
;; Purpose: returns a list containing the intersection of the two arguments
(define  (Intersection a-los1 a-los2) … )
```

Clearly, a program with two complex arguments …

Nesting Locals

Data Analysis

- Use standard list definitions
- Two arguments are general lists of symbols
  - → No restriction on length or order
- Use template that we developed for merge

	(empty? a-los2)	(cons? a-los2)
(empty? a-los1)	(and (empty? a-los1) (empty? a-los2))	(and (empty? a-los1) (cons? a-los2))
(cons? a-los1)	(and (cons? a-los1) (empty? a-los2))	(and (cons? a-los1) (cons? a-los2))

Possible cases

Nesting Locals

Template

```
(define (f  a-los1  a-los2)
  (cond
    [(and (empty? a-los1) (empty? a-los2))     …]
    [(and (empty? a-los1) (cons? a-los2))
         … (first a-los2) …  (f  a-los1 (rest a-los2)) …]
    [(and (cons? a-los1) (empty? a-los2))
         … (first a-los1) …  (f (rest a-los1) a-los2)…]
    [(and (cons? a-los1) (cons? a-los2))
         … (first a-los1) … (first a-los2) …
         … (f  a-los1 (rest a-los2)) …
         … (f (rest a-los1) a-los2) …
         … (f (rest a-los1) (rest a-los2)) …]
   )
)
```


Nesting Locals

Working through the cases

```
(define (Intersect  a-los1  a-los2)
  (cond
    [(and (empty? a-los1) (empty? a-los2))     empty]
    [(and (empty? a-los1) (cons? a-los2))
         … (first a-los2) …  (f  a-los1 (rest a-los2)) …]
    [(and (cons? a-los1) (empty? a-los2))
         … (first a-los1) …  (f (rest a-los1) a-los2)…]
    [(and (cons? a-los1) (cons? a-los2))
         … (first a-los1) … (first a-los2) …
         … (f  a-los1 (rest a-los2)) …
         … (f (rest a-los1) a-los2) …
         … (f (rest a-los1) (rest a-los2)) …]
   )
)
```

obviously

Nesting Locals

Working through the cases

```
(define (Intersect  a-los1  a-los2)
  (cond
    [(and (empty? a-los1) (empty? a-los2))     empty]
    [(and (empty? a-los1) (cons? a-los2))      empty]
    [(and (cons? a-los1) (empty? a-los2))
          … (first a-los1) …  (f (rest a-los1) a-los2)…]
    [(and (cons? a-los1) (cons? a-los2))
          … (first a-los1) … (first a-los2) …
          … (f  a-los1 (rest a-los2)) …
          … (f (rest a-los1) a-los2) …
          … (f (rest a-los1) (rest a-los2)) …]
   )
)
```

Less obviously

Nesting Locals

Working through the cases

```
(define (Intersect  a-los1  a-los2)
  (cond
    [(and (empty? a-los1) (empty? a-los2))     empty]
    [(and (empty? a-los1) (cons? a-los2))      empty]
    [(and (cons? a-los1) (empty? a-los2))      empty]
    [(and (cons? a-los1) (cons? a-los2))
          … (first a-los1) … (first a-los2) …
          … (f  a-los1 (rest a-los2)) …
          … (f (rest a-los1) a-los2) …
          … (f (rest a-los1) (rest a-los2)) …]
   )
)
```

Follows last case

Nesting Locals

Working through the cases

```
(define (Intersect  a-los1  a-los2)
  (cond
    [(and (empty? a-los1) (empty? a-los2))     empty]
    [(and (empty? a-los1) (cons? a-los2))      empty]
    [(and (cons? a-los1) (empty? a-los2))      empty]
    [(and (cons? a-los1) (cons? a-los2))
          … (first a-los1) … (first a-los2) …
          … (f  a-los1 (rest a-los2)) …
          … (f (rest a-los1) a-los2) …
          … (f (rest a-los1) (rest a-los2)) …]
  )
)
```

The clause with all the work

Like Search

Need to search for (first a-los1) in a-los2 and intersect (rest a-los1) with a-los2

Nesting Locals

Where do we go from here?

```
 …
    [(and (cons? a-los1) (cons? a-los2))
          (append (ISearch (first a-los1) a-los2)
                  (Intersect (rest a-los1) a-los2) ) ]
…

;; ISearch: symbol   list-of-symbol -> list-of-symbol
;; Purpose: return a singleton list containing symbol if symbol
;;          is found in the list; return empty otherwise
(define (ISearch key a-los) … )
```

We can hide ISearch in a local …

Nesting Locals

This leads to the following code:

```
(define (Intersect  a-los1  a-los2)
  (local
    [ (define (ISearch key a-los)
        (cond [(empty? a-los)         empty]
              [(cons?   a-los)
               (cond [(symbol=? (first a-los) key ) (list (first a-los))]
                     [else (ISearch key (rest a-los))  ] ) ]) ) ]
    (cond
      [(and (empty? a-los1) (empty? a-los2))            empty]
      [(and (empty? a-los1) (cons? a-los2))  empty]
      [(and (cons? a-los1) (empty? a-los2))  empty]
      [(and (cons? a-los1) (cons? a-los2))
       (append (ISearch (first a-los1) a-los2)
               (Intersect (rest a-los1) a-los2) ) ]  ) ;; close the cond
  ) ;; close the local
)
```

For append

Nesting Locals

Elliding invariant parameters:

```
(define (Intersect  a-los1  a-los2 )
  (local
    [ (define (ISearch key a-los)
        (cond [(empty? a-los)         empty]
              [(cons?   a-los)
               (cond [(symbol=? (first a-los) key ) (list (first a-los))]
                     [else (ISearch key (rest a-los))  ] ) ]) ) ]
    (cond
      [(and (empty? a-los1) (empty? a-los2))            empty]
      [(and (empty? a-los1) (cons? a-los2))  empty]
      [(and (cons? a-los1) (empty? a-los2))  empty]
      [(and (cons? a-los1) (cons? a-los2))
       (append (ISearch (first a-los1) a-los2)
               (Intersect (rest a-los1) a-los2) ) ]  ) ;; close the cond
  ) ;; close the local
)
```

Can use local to avoid passing these

On homework?

Using Local

Use a local when

- It lets the program compute a complicated value once instead of multiple times
- It makes a complicated expression more readable
- It eliminates the need for passing an invariant parameter
- It hides helper functions that should not be exposed to the outside world
  - → A matter of defined & exposed interfaces
  - → Local lets us manage the shared name space