Local definitions

- The functions and special forms we've seen so far can be **arbitrarily nested** – except define and check-expect.
- So far, **definitions** have to be made "at the top level", outside any expression.
- The Intermediate language provides the special form **local**, which contains a series of **local definitions** plus an **expression using them**, of the form

  (local
    (def_1 . . . def_n)
    exp)

  What use is this?

- **local** groups related definitions for use in expression. Each definition can be either a define or a define-struct.

Function multiples-of

Suppose we want to use abstract list functions to solve:

Write a function multiples-of that consumes a positive integer, n, and a list of integers, ilist, and produces a new list containing ONLY those values in ilist which are multiples of n.

Our attempt:

```
(define (is-mult? m)
    (zero? (remainder m n)))
(define (multiples-of n ilist)
            (filter is-mult? ilist))
```

fails. Why?

```
(define ilist (list 1 2 3 4 5 6)
;; Example
(multiples-of 2 ilist) => (list 2 4)
(multiples-of 3 ilist) => (list 3 6)
```

Local definitions (cont.)

```
(define (multiples-of n ilist)
  (local
  (  ;; (is-mult? m) produces
     ;; true if m is a multiple of n,
     ;;  and false otherwise.
     ;; is-mult?: Int → Bool
  (define (is-mult? m)
  (zero? (remainder m n))))
  (filter is-mult? ilist)))
```

```
(define (is-mult? m)
    (zero? (remainder m n)))

(define (multiples-of n ilist)
(filter is-mult? ilist))
```

n only exists within the body of multiples-of

- The combine function is-mult? needs the value of n but to be used by filter, it can only accept one parameter - an element of the list.

Function multiples-of (cont.)

- Note: Provide **purpose, contract, and requirements** for local helper functions.

REMEMBER

```
(define (multiples-of n ilist)
    (local
    [  ;; (is-mult? m) produces
       ;; true if m is a multiple of n,
       ;;  and false otherwise.
       ;; is-mult?: Int → Bool
    (define (is-mult? m)
    (zero? (remainder m n)))]

    (filter is-mult? ilist)))
```

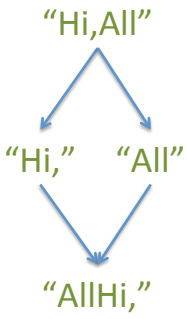
Another Example: function swap-parts

Recall the function swap-parts from Module 2. The function used three helper functions.

```
(define (mid num)
    (quotient num 2))

(define (front-part mystring)
    (substring mystring 0 (mid (string-length
mystring))))

(define (back-part mystring)
    (substring mystring (mid (string-length
mystring))
    (string-length mystring)))
```



Function swap-parts (cont.)

```
(define (swap-parts mystring)
    (string-append (back-part mystring)
    (front-part mystring)))
```

- The helper function mid is a helper function of the helper functions front-part and back-part.
- Our solution was perfectly acceptable. However, repeated applications, such as

  (mid (string-length mystring)),

  make it a bit hard to read.

Function swap-parts (cont.)

```
(define (swap-parts mystring)
(local
 ((define mid (quotient (string-length mystring) 2))
  (define front (substring mystring 0 mid))
  (define back (substring mystring mid
                          (string-length mystring))))
 (string-append back front)))
```

```
(local
    (def1
     def2
     def3)

     exp)
```

- The special form local allows us to define a **constant** or a **function** within another function.
- It would be nice to replace repeated applications by a constant.
- Note: mid, front, and back are constants, not functions, **so they have no contracts**.

Local double parentheses

- Like cond, local results in double parentheses.
- Optional: use **square brackets** to improve readability.

```
(define (swap-parts mystring)
   (local
      [(define mid (quotient (string-length mystring) 2))
       (define front (substring mystring 0 mid))
       (define back (substring mystring mid
                               (string-length mystring)))]
      (string-append back front)))
```

Definition Part

Re-using names with local

A define within a local expression may rebind a name that has already been bound to another value or expression.

```
(define (my-fun n m)
    (local
     [(define n 12)
      (define (local-fun n) (* n 10))]
        (+ m (local-fun n))))
```


Practical Exercise

Using abstract list functions and local, write the function ***scale-points*** which consumes a list of posns and a number *k*, and produces a list of posns where each co-ordinate value (i.e., both x and y) are scaled (i.e., **multiplied by**) the value *k*. Any helper functions must be declared using local. You must not use recursion to solve this problem.

```
(define (scale-points alop k)
(local
[(define (scale-posn aposn)
          (make-posn (* k (posn-x aposn))
                     (* k (posn-y aposn))))]
(map scale-posn alop)))
```

Nested local expression

- It isn't always possible to define local at the beginning of the function definition, **because** the definition might make assumptions that are only true in part of the code.
- A typical example is that of using a **list** function, like first or rest, which must consume a nonempty list.
- When there is one local definition that can be used throughout and one not, we end up with nested local expressions.

Nested local expression

```
(define (avg-nums lst)
     (local
      [(define only-nums (filter number? lst))]
      (cond
      [(empty? only-nums) 'no-nums]
      [else
       (local
         [(define num-sum (foldr + 0 only-nums))
          (define num-nums (length only-nums))]
              (/ num-sum num-nums)) ]
     )
))
```

Using local for common sub-expressions

- A **sub-expression** used twice within a function body always yields the same value.
- Using local to give the reused sub-expression a name improves the readability of the code.
- In the following example, the function eat-apples removes all occurrences of the symbol 'apple from a list of symbols.
- The sub-expression (eat-apples (rest alist)) occurs twice in the code.

The function eat-apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty
     [(cons? alist)
      (cond
      [(not (symbol=? (first alist)
'apple))
      (cons (first alist)
(eat-apples (rest alist)))]
       [else
(eat-apples (rest alist))])]))
```

```
(define (eat-apples alist)
 (cond
   [(empty? alist) empty]
   [(cons? alist)
  (local [(define ate-rest (eat-
apples (rest alist)))]
(cond
   [(not (symbol=? (first alist)
'apple))
   (cons (first alist) ate-rest)]
   [else ate-rest])]))
```

Using local

Using local can improve efficiency

Module 2

```
(define (mid num)
      (quotient num 2))

(define (front-part mystring)
      (substring mystring 0 (mid (string-
length mystring))))

(define (back-part mystring)
      (substring mystring (mid (string-length
mystring))
      (string-length mystring)))

(define (swap-parts mystring)
      (string-append (back-part mystring)
      (front-part mystring)))
```

Module 10

```
(define (swap-parts mystring)
(local
  ((define mid (quotient (string-length
mystring) 2))
   (define front (substring mystring 0 mid))
   (define back (substring mystring mid
(string-length mystring))))
  (string-append back front)))
```


Using local for smaller tasks

- Sometimes we choose to use local in order to name sub-expressions in a way to make the code more **readable**, even if they are not **reused**.
- This may make the code longer, but may improve clarity.
- Recall our function to compute the distance between two points.

```
(define (distance posn1 posn2)
     (sqrt (+ (sqr (– (posn-x posn1) (posn-x posn2)))
              (sqr (– (posn-y posn1) (posn-y posn2))))))
```

Using local for smaller tasks (cont.)

```
(define (distance posn1 posn2)
    (sqrt (+ (sqr (– (posn-x posn1) (posn-x posn2)))
             (sqr (– (posn-y posn1) (posn-y posn2))))))

(define (distance posn1 posn2)
    (local [(define delta-x (– (posn-x posn1) (posn-x posn2)))
            (define delta-y (– (posn-y posn1) (posn-y posn2)))
            (define sqrsum (+ (sqr delta-x) (sqr delta-y)))]
      (sqrt sqrsum)))
```


Using local for encapsulation

- Encapsulation is the process of grouping things together in a "**capsule**".
- We have already seen data encapsulation in the use of structures.
- Encapsulation can also be used to hide information. Here the local bindings are not visible outside the local expression.

Full design recipe

- Note that a full design recipe is not needed for local helper functions on **assignment submissions**.
- You should still develop **examples** and **tests** for helper functions, and test them outside local expressions if possible.
- Once you have confidence that they work, you can move them into a local expression, and delete the examples and tests.
- A **contract** and **purpose** are still required (omitted in these slides for space reasons).

Goals of this module

- You should understand the idea of **encapsulation** of local helper functions.
- You should be familiar with map, filter, and foldr, understand how they abstract common recursive patterns, and be able to use them to write code.
- You should understand the idea of functions as f**irst-class values**, and how they can be supplied as arguments.

Goals of this module (cont.)

- You should be able to use local to avoid repetition of common sub-expressions, to improve readability of expressions, and to improve efficiency of code.
- You should be able to match the use of any constant or function name in a program to the binding to which it refers.