

# Module 9: Local and functional abstraction

Readings: HtDP, Intermezzo 3 (Section 18); Sections 19-23.

We will cover material on functional abstraction in a somewhat different order than the text.

CS 115 will cover built-in functions that consume functions as inputs.

CS 116 will cover functions that produce functions, and how to write functions that produce functions.

We move to the Intermediate teaching language with the introduction of local definitions and abstract list functions.

# What is abstraction?

Abstraction consists of

- finding similarities or common aspects, and
- forgetting unimportant differences.

For a single function, differences in parameter values are forgotten, and the similarity is captured in the function body.

For multiple functions, similarity is captured in templates.

For multiple functions, further abstraction is possible.

# Eating 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))])]))
```

# Selecting even numbers

```
(define (select-even alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
     (cond
       [(even? (first alist)) (cons (first alist) (select-even (rest alist)))]
       [else (select-even (rest alist))]))])
```

# Abstracting from these examples

Similarity: general structure (removing certain items)

Difference: predicate used to decide what to remove

Goal: form an **abstract list function** that consumes the predicate.

Functions are *first-class* values in the Intermediate Student Language.

First-class values can be bound to constants, put in lists and structures, consumed as arguments, and produced as results.

# The abstract list function **filter**

```
(define (filter pred alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
     (cond
       [(pred (first alist))
        (cons (first alist) (filter pred (rest alist)))]
       [else
        (filter pred (rest alist))])]))
```

We can simplify the code somewhat.

```
(define (filter pred alist)
  (cond
    [(empty? alist) empty]
    [(pred (first alist))
     (cons (first alist) (filter pred (rest alist)))]
    [else (filter pred (rest alist))]))
```

# Tracing filter

`(filter even? (list 6 7 8))`

$\Rightarrow$  `(cons 6 (filter even? (list 7 8)))`

$\Rightarrow$  `(cons 6 (filter even? (list 8)))`

$\Rightarrow$  `(cons 6 (cons 8 (filter even? empty)))`

$\Rightarrow$  `(cons 6 (cons 8 empty))`

The **abstract list function** `filter` performs the general operation of selecting items from lists.

Racket provides such functions to apply common patterns of structural recursion.



# Using **filter**

```
(define (select-even alist) (filter even? alist))
```

```
(define (symbol-not-apple? item) (not (symbol=? item 'apple)))
```

```
(define (eat-apples alist) (filter symbol-not-apple? alist))
```

The built-in function **filter** consumes a predicate specifying which elements of the list are to be kept.

The predicate must be a one-parameter function producing a boolean, where the type of the parameter is same as the type of the elements of the list.

# Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as `filter`.

It reduces code size.

It avoids cut-and-paste.

Bugs can be fixed in one place instead of many.

Improving one functional abstraction improves many applications.

# Additions to syntax and semantics with Intermediate

An expression can now be a primitive operation or a function.

Names of functions are now values.

# Abstracting from examples

```
(define (negate-list numlist)
  (cond
    [(empty? numlist) empty]
    [else (cons (— (first numlist)) (negate-list (rest numlist)))]
```

```
(define (compute-grades rlist)
  (cond
    [(empty? rlist) empty]
    [else (cons (final-grade (first rlist))
                  (compute-grades (rest rlist)))]))
```

# The abstract list function **map**

```
(define (map f alist)
  (cond
    [(empty? alist) empty]
    [else (cons (f (first alist))
                  (map f (rest alist)))]))
```

For this and other built-in abstract list functions, see the table on page 313 of the text (Figure 57 in Section 21.2).

# Tracing map

`(map sqr (list 3 6 5))`

$\Rightarrow$  `(cons (sqr 3) (map sqr (list 6 5)))`

$\Rightarrow$  `(cons 9 (map sqr (list 6 5)))`

$\Rightarrow$  `(cons 9 (cons (sqr 6) (map sqr (list 5))))`

$\Rightarrow$  `(cons 9 (cons 36 (map sqr (list 5))))`

$\Rightarrow$  `(cons 9 (cons 36 (cons (sqr 5) (map sqr empty))))`

$\Rightarrow$  `(cons 9 (cons 36 (cons 25 (map sqr empty))))`

$\Rightarrow$  `(cons 9 (cons 36 (cons 25 empty)))`

The abstract list function `map` performs the operation of transforming a list element-by-element into another list of the same length.

`(map f (list x1 x2 ... xn))` has the same effect as  
`(list (f x1) (f x2) ... (f xn))`.

Short definitions using `map`:

`(define (negate-list numlist) (map — numlist))`

`(define (compute-grades rlist) (map final-grade rlist))`

The function consumed by `map` must be a one-parameter function where the type of the parameter is the same as the type of the elements of the list.

# Abstracting from examples

```
(define (first-evens-from k n)
  (cond [(= n k) empty]
        [else (cons (* k 2) (first-evens-from (add1 k) n))]))

(define (first-evens n) (first-evens-from 0 n))

(define (points-from k n a b)
  (cond [(= n k) empty]
        [else (cons (make-posn k (+ (* k a) b))
                      (points-from (add1 k) n a b))]))

(define (first-points n a b) (points-from 0 n a b))
```



# The abstract list function **build-list**

```
(define (build-from q n fn)
  (cond [(= n q) empty]
        [else (cons (fn q)
                      (build-from (add1 q) n fn))]))
```

```
(define (build-list n fn)
  (build-from 0 n fn))
```

# Tracing **build-list**

(build-list 3 add1)

⇒ (build-from 0 3 add1)

⇒ (cons 1 (build-from 1 3 add1))

⇒ (cons 1 (cons 2 (build-from 2 3 add1)))

⇒ (cons 1 (cons 2 (cons 3 (build-from 3 3 add1))))

⇒ (cons 1 (cons 2 (cons 3 empty)))

The abstract list function `build-list` creates a list of length `n` by applying the function `fn` to the values `0,1,2,...,n-1`.

`(build-list n fn)` creates `(list (fn 0) (fn 1) ... (fn (— n 1)))`

Short definitions using `build-list`

`(define (first-squares n) (build-list n sqr))`

`(define (make-odd k) (+ (* 2 k) 1))`

`(define (first-odds n) (build-list n make-odd))`

The function consumed by `build-list` must be a one-parameter function that consumes a `Nat`. The type in the new list is the type produced by the consumed function.

# Abstracting from examples

```
(define (product-of-numbers alist)
  (cond
    [(empty? alist) 1]
    [else (* (first alist) (product-of-numbers (rest alist)))]))
```

```
(define (concat-firsts alist)
  (cond
    [(empty? alist) " "]
    [(string=? " " (first alist)) (concat-firsts (rest alist))]
    [else (string-append (substring (first alist) 0 1)
                          (concat-firsts (rest alist)))]))
```

```
(define (my-list-fun alist)
  (cond
    [(empty? alist) ...]
    [else ... (first alist) ... (my-list-fun (rest alist)) ... ]))
```

To fill in the template:

Replace the first ellipsis by a base value.

Combine `(first alist)` and the result of a recursive call on `(rest alist)`.

Parameters for the abstract list function: base value and combining function.

# The abstract list function **foldr**

```
(define (foldr combine base alist)
  (cond
    [(empty? alist) base]
    [else (combine
              (first alist)
              (foldr combine base (rest alist)))]))
```

**foldr** is also a built-in function in Racket.

# Tracing foldr

`(foldr f 0 (list 3 6 5))`

$\Rightarrow$  `(f 3 (foldr f 0 (list 6 5)))`

$\Rightarrow$  `(f 3 (f 6 (foldr f 0 (list 5))))`

$\Rightarrow$  `(f 3 (f 6 (f 5 (foldr f 0 empty))))`

$\Rightarrow$  `(f 3 (f 6 (f 5 0)))`  $\Rightarrow$  ...

Intuitively, the effect of the application

`(foldr f b (list x1 x2 ... xn))` is to compute the value of the expression  
`(f x1 (f x2 (... (f xn b) ...)))`.

`foldr` is short for “fold right”.

It can be viewed as “folding” a list using the provided `combine` function, starting from the right-hand end of the list.

It can be used to implement `map`, `filter`, and other abstract list functions.



# Using **foldr**

```
(define (product-of-numbers alist) (foldr * 1 alist))
```

If **alist** is (**list** **x1** **x2** ... **xn**), then by our intuitive explanation of **foldr**, the expression (**foldr** \* 1 **alist**) reduces to

```
(* x1 (* x2 (* ... (* xn 1) ...)))
```

Thus **foldr** does all the work of the template for processing lists in the case of **product-of-numbers**.

The combine function provided to `foldr` consumes two parameters:

- an item in the list that `foldr` consumes and
- the result of applying `foldr` to the rest of the list.

In `product-of-numbers` the function `*` multiplies an element with the product of the rest of the list.

How does `concat-firsts` use these two parameters?

```
(define (combine-nonempty s t)
  (cond
    [(string=? "" s) t]
    [else (string-append (substring s 0 1) t)]))
```

```
(define (concat-firsts alist)
  (foldr combine-nonempty "" alist))
```

Consider the function `my-length` that calculates the length of a string:

```
(define (my-length alist)
  (cond [(empty? alist) 0]
        [else (+ 1 (my-length (rest alist)))]))
```

For `my-length`, the first element of the list contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to `foldr` in this case can ignore the value of the first parameter, and just add 1 to the result on the rest of the list.

```
(define (inc x y)
```

```
  (add1 y))
```

```
(define (my-length alist)
```

```
  (foldr inc 0 alist))
```

# Using **foldr** to produce lists

The functions we provide to **foldr** can also produce **cons** expressions, since these are also values.

Example: using **foldr** for **negate-list**.

How do we produce the negate list from

- an item in the list and
- the result of the recursive call on the **rest** of the list?

**neg-combine** takes the element, negates it, and **conses** it onto the result of the recursive call.

:: neg-combine: Num (listof Num)  $\rightarrow$  (listof Num)

```
(define (neg-combine item result-on-rest)  
  (cons (— item) result-on-rest))
```

:: negate-list: (listof Num)  $\rightarrow$  (listof Num)

```
(define (negate-list alist)  
  (foldr neg-combine empty alist))
```

# Boolean functions and foldr

To check whether a predicate `p` produces `true` for every element in a list `alist`, we might be tempted to try:

```
(foldr and true (map p alist))
```

Problem: `and` is not a function, but a special form, and this produces an error.

Solution: Racket provides `andmap`, which can be used like this:

```
(andmap p alist)
```

For the same reason, `ormap` is provided.



Imperative languages, which tend to provide inadequate support for recursion, usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.

Anything that can be done with the list template can be done using `foldr`, without explicit recursion.

Does that mean that the list template is obsolete?

No. Experienced Racket programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

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?

The helper 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.

However, `n` only exists within the body of `multiples-of`.

Can we define `is-mult?` inside `multiples-of`?

Yes, but we need a new construct: `local`.

# 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 (def1 ... defn) exp)
```

What use is this?

```
(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)))
```

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

# Another example

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))))
```

The helper function `mid` is a helper function of the helper functions `front-part` and `back-part`.

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



Our solution was perfectly acceptable.

However, repeated applications, such as

`(mid (string-length mystring))`,

make it a bit hard to read.

It would be nice to replace repeated applications by a constant.

The special form **local** allows us to define a constant or a function within another function.

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

Note: `mid`, `front`, and `back` are constants, not functions, so they have no contracts.

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-append back front)))
```

# Re-using names with local

We have reused names before:

- `n` is bound to a value by `define`
- `n` is the name of a parameter

```
(define n 10)
```

```
(define (myfun n) (+ 2 n))
```

```
(myfun 6)
```

The function application `(myfun 6)` produces 8, not 12, due to the substitution rules for function application.

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

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

The substitution rules for **local** must handle this.

# The semantics of local

Key ideas:

- Create a new, unique name for each identifier defined in a local definition (e.g. `mid` becomes `mid_0`).
- Bind the new name to the value.
- Substitute the new name for the old name everywhere in the expression.
- Move all of the local definitions to the top level, evaluate, and continue.

# Evaluating swap-parts

(swap-parts "aside")

⇒ ( local [(define mid (quotient (string-length "aside") 2))  
          (define front (substring "aside" 0 mid))  
          (define back (substring "aside" mid))]  
      (string-append back front))

⇒ (define mid\_0 (quotient (string-length "aside") 2))  
   (define front\_0 (substring "aside" 0 mid\_0))  
   (define back\_0 (substring "aside" mid\_0))  
   (string-append back\_0 front\_0))

```
⇒ (define mid_0 (quotient 5 2))  
   (define front_0 (substring "aside" 0 mid_0))  
   (define back_0 (substring "aside" mid_0))  
   (string-append back_0 front_0)  
  
⇒ (define mid_0 2)  
   (define front_0 (substring "aside" 0 mid_0))  
   (define back_0 (substring "aside" mid_0 ))  
   (string-append back_0 front_0)
```



```
⇒ (define mid_0 2)
   (define front_0 (substring "aside" 0 2))
   (define back_0 (substring "aside" mid_0))
   (string-append back_0 front_0)

⇒ (define mid_0 2)
   (define front_0 "as")
   (define back_0 (substring "aside" mid_0))
   (string-append back_0 front_0)
```

```
⇒ (define mid_0 2)
   (define front_0 "as")
   (define back_0 (substring "aside" 2))
   (string-append back_0 front_0)
```

⇒ (define mid\_0 2)  
(define front\_0 "as")  
(define back\_0 "ide")  
(string-append back\_0 front\_0)

⇒ (define mid\_0 2)  
(define front\_0 "as")  
(define back\_0 "ide")  
(string-append "ide" front\_0)

```
⇒ (define mid_0 2)
   (define front_0 "as")
   (define back_0 "ide")
   (string-append "ide" "as")
⇒ "ideas"
```

# Substitution rule

An expression of the form

(**local** [(**define**  $x_1$   $exp_1$ ) ... (**define**  $x_n$   $exp_n$ )]  $bodyexp$ ) is handled as follows:

$x_1$  is replaced with a **fresh** identifier (call it  $x_{1\_0}$ ) everywhere in  $exp_1$  through  $exp_n$  and  $bodyexp$ .

$x_2$  is replaced with  $x_{2\_0}$  everywhere in  $exp_1$  through  $exp_n$  and  $bodyexp$ .

This process is repeated with  $x_3$  through  $x_n$ .

Then, all the definitions are lifted out to the top level, yielding

```
(define x1_0 exp1)
(define x2_0 exp2)
...
(define xn_0 expn)
bodyexp
```

where the expressions have been modified to use the new names.

Read Intermezzo 3 (Section 18).

# Nested local expressions

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.

```
(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))
           (define avg (/ num-sum num-nums))]
          avg))]))]
```



# Using **local** for common subexpressions

A subexpression used twice within a function body always yields the same value.

Using **local** to give the reused subexpression 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 subexpression **(eat-apples (rest alist))** occurs twice in the code.

# The function eat-apples

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

# Using **local** in eat-apples

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

In the function `eat-apples`, the subexpression

`(eat-apples (rest alist))`

appears in two different answers of the same `cond` expression, so only one of them will ever be evaluated.

In the next example, the subexpression

`(list-max (rest alon))`

appears twice. The first appearance is always evaluated, and sometimes both are. In this case, using `local` is more efficient, as well as more readable.

# The function list-max

:: (list-max alon) produces maximum in alon.

:: list-max: (listof Num)  $\rightarrow$  Num

:: requires: alon is nonempty

```
(define (list-max alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else
     (cond
       [(> (first alon) (list-max (rest alon))) (first alon)]
       [else (list-max (rest alon))]))]))
```

# Using **local** in list-max

```
(define (list-max2 alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else
     (local [(define max-rest (list-max2 (rest alon)))]
       (cond
         [(> (first alon) max-rest) (first alon)]
         [else max-rest]))])))
```

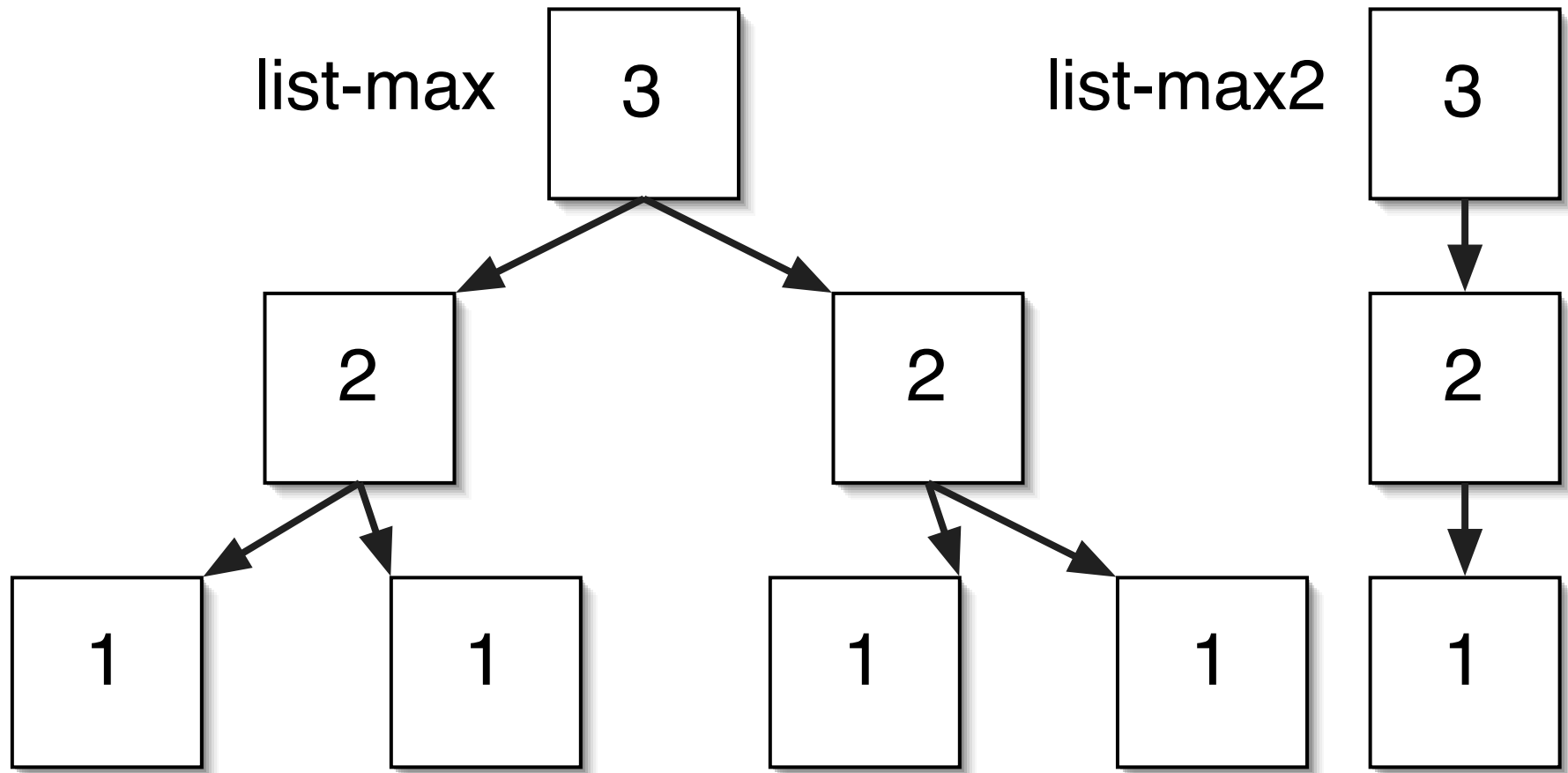
# Using **local** can improve efficiency

You might expect that the first version does no more than twice the work of the second version.

But the first version may make two recursive calls, and each of them may make two, and so on.

If we run each version on an increasing list of three numbers, and draw a box for each call containing the length of its argument, we can see how the work adds up.

# Tracing versions of list-max





# Using **local** for smaller tasks

Sometimes we choose to use **local** in order to name subexpressions mnemonically 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))))))
```

```
(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 (have no effect) outside the local expression.

We can bind names to functions as well as values in a local definition.

Evaluating the local expression creates new, unique names for the functions just as for the values.

This is known as **behaviour** encapsulation.

Behaviour encapsulation allows us to move helper functions within the function that uses them, so they are invisible outside the function.

```
(define (final-grade astudent)
  (make-grade
    (student-name astudent)
    (+ (* assts-weight (student-assts astudent))
      (* mid-weight (student-mid astudent))
      (* final-weight (student-final astudent)))))

(define (compute-grades slist)
  (cond [(empty? slist) empty]
        [else (cons (final-grade (first slist))
                      (compute-grades (rest slist)))])])
```

```
(define (compute-grades slist)
  (local [(define assts-weight 0.2)
          (define mid-weight 0.3)
          (define final-weight 0.5)]
```

:: (final-grade astudent) produces Grade for astudent

:: final-grade: Student  $\rightarrow$  Grade

```
(define (final-grade astudent)
  (make-grade (student-name astudent)
    (+ (* assts-weight (student-assts astudent))
      (* mid-weight (student-mid astudent))
      (* final-weight (student-final astudent))))))
(map final-grade slist)))
```

```
(define (my-sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (my-sort (rest alon)))]))
```

```
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
      [(<= n (first alon)) (cons n alon)]
      [else (cons (first alon) (insert n (rest alon)))])]))
```

(define (my-sort alon)

(local [;; (insert n alon) produces sorted list including n and all

;; values in alon

;; insert: Num (listof Num)  $\rightarrow$  (listof Num)

;; requires: alon is sorted in nondecreasing order

(define (insert n alon)

(cond

[(empty? alon) (cons n empty)]

[ else (cond

[( $\leq$  n (first alon)) (cons n alon)]

[ else (cons (first alon) (insert n (rest alon))))]])) ... ))



;; Body of my-sort

```
(cond  
  [(empty? alon) empty]  
  [ else (insert (first alon) (my-sort (rest alon))))]))
```

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).

# Another example

Use abstract list functions and **local** to write a function **shorter-than-avg** that consumes a list of strings, **slist**, and produces a list of all those strings in **slist** that are shorter than the average length of all strings in **slist**.

For example,

**(shorter-than-avg (list "abc" " " "12345" "b"))**  $\Rightarrow$  **(list " " "b")**

**(shorter-than-avg empty)**  $\Rightarrow$  **empty**

# “Single use” helper functions

Consider the following:

```
(define (longer-strings los n)
  (local
    [(define (longer? s)
      (> (string-length s) n))]
    (filter longer? los)))
```

The function `longer?` is defined so it can be passed as a parameter to `filter`. It only exists inside `longer-strings`.

There is a Racket feature called `lambda` which allows us to simplify the definition of `longer-strings`.

# lambda

- Creates a function value without assigning it a name.
- The function can consume any number of parameters.
- For example, `(lambda (x) (+ 1 (sqr x)))` is a function that consumes one parameter, and produces its square plus 1. More generally

```
(lambda  
  (p1 p2 ... pN)  
  body-of-function)
```

- `lambda` expressions cannot be simplified further.

# Using **lambda**

Requires new Racket level: Intermediate student with **lambda**

;; (first-points n a b) produces a line containing the

;;     Posns (make-posn x y)

;;     where  $x = 0, 1, \dots, n-1$ , and  $y = a \cdot x + b$ .

```
(define (first-points n a b)
```

```
  (build-list n
```

```
    (lambda (x) (make-posn x (+ (* a x) b)))))
```

# Defining functions with **lambda**

```
(define (double k)  
  (* k 2))
```

can be rewritten using **lambda** as

```
(define double  
  (lambda (k) (* k 2)))
```

The two definitions are equivalent and define a function called **double**.

# Evaluating a **lambda** expression

Recall that a function application has the form  $(f\ a1\ a2\ \dots\ an)$ , where  $f$  is the name of a built-in or user-defined function.

Now,  $f$  just needs to be a function value, so it can be a **lambda** expression. For example,

$$((\text{lambda}\ (x)\ (*\ 2\ x))\ 4) \Rightarrow (*\ 2\ 4) \Rightarrow 8$$
$$((\text{lambda}\ (s\ t)\ (+\ (\text{string-length}\ s)\ t))\ \text{"hello"}\ 10)$$
$$\Rightarrow (+\ (\text{string-length}\ \text{"hello"})\ 10)$$
$$\Rightarrow (+\ 5\ 10)$$
$$\Rightarrow 15$$



# When to use **lambda**?

Use **lambda** when the function is

- single use
- reasonably short (2-3 lines)

Complete **count-starters** that consumes a list of Strings (called **phrases**) and a String of length one (called **start**) and produces the number of Strings in **phrases** that start with **start**.

# Solution

```
(define (count-starters phrases start)
  (foldr
    (lambda (s count)
      (+ count
        (cond [(string=? " " s) 0]
              [(string=? (substring s 0 1) start) 1]
              [else 0])))
    0
    phrases))
```

# Goals of this module

You should understand the idea of encapsulation of local helper functions.

You should be familiar with `map`, `filter`, `foldr`, and `build-list`, understand how they abstract common recursive patterns, and be able to use them to write code.

You should understand the idea of functions as first-class values, and how they can be supplied as arguments.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.

You should understand the syntax, informal semantics, and formal substitution semantics for the **local** special form.

You should be able to use **local** to avoid repetition of common subexpressions, 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.

You should be able to understand and write programs that use **lambda** expressions in place of named functions.