Module 7: 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.

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.

In Module 05, we introduced the following sorting function to produce a list in increasing numerical order.

```
;; requires: values is in sorted order
(define (insert x values)
  (cond [(empty? values) (cons x empty)]
        [(<= x (first values)) (cons x values)]
        [else (cons (first values) (insert x (rest values)))]))
(define (my-sort values)
  (cond [(empty? values) empty]
        [else (insert (first values) (my-sort (rest values)))]))
```

Sorting different ways

If the comparison $<=$ in insert is changed to $>=$, then sorting a (listof Num) produces a list in decreasing numerical order.

If the comparison $<=$ in insert is changed to string$<=$?, then sorting a (listof Str) produces a list in increasing alphabetical order.

If the comparison $<=$ in insert is changed to sum$<=$? where

(define (sum$<=$? pt1 pt2)
   ($<=$ ($+$ (first pt1) (second pt1))
        ($+$ (first pt2) (second pt2))))

then sorting a (listof (list Num Num)) produces a list in increasing order of the sum.

It would be nice if we could simply apply a sorting function to a list and a comparison function, in order to sort different types of lists into different orders.

Fortunately, we can!

The built-in function sort

Functions are *first-class values* in the intermediate student language level, which means that they can be bound to constants, put in lists, consumed as arguments, and produced as results.

In the case of sort, in addition to a list, it consumes a predicate that consumes two elements of the list (x and y). The predicate produces true if x should come before y in the sorted list.

;; (sort lst comes-before?) produces an ordered version of lst,
;;   based on the order determined by comes-before?

The contract for sort shows the relationship between the parameters, where χ can be any type.

The type of a function is its contract (with brackets).

;; sort: (listof χ) (χ χ $\rightarrow$ Bool) $\rightarrow$ (listof χ)

For example:

(sort (list 4 8 6 2 10) >) $\Rightarrow$ (list 10 8 6 4 2)
(sort (list (list 10 4) (list 8 $-$3) (list 0 3)) sum<=?)
        $\Rightarrow$ (list (list 0 3) (lit 8 $-$3) (list 10 4))

7: Local and functional abstraction

A more complicated example:

```
(define (str-len<? s1 s2)
  (or (< (string-length s1) (string-length s2))
      (and (= (string-length s1) (string-length s2))
           (string<? s1 s2))))
```

```
(sort (list "horse" "owl" "elephant" "cat") str-len<?)
⇒ (list "cat" "owl" "horse" "elephant")
```

Abstract list functions

Sorting is a common list application, and sort can be used to sort any type of list, as long we have a way to compare two elements.

Racket includes other functions for common list applications, such as selecting values from a list (*filter*) or applying the same operation to every element in a list (*map*).

By including a function as a parameter to these abstract list functions, we are able to filter and map onto any type list, similar to sorting any type of list.

Selecting certain symbols

```
(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))]))
```

Selecting even numbers

```
(define (select-even alist)
  (cond
    [(empty? alist) empty]
    [(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.

The abstract list function filter

The following is a possible implementation of the built-in function
filter. You do not need to write this yourself.

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

Tracing **my-filter**

(my-filter even? (list 6 7 8))

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

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

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

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

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

filter may not be implemented exactly as my-filter is, but it behaves the same way.

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))

filter consumes a predicate specifying which elements to include in the new list. The predicate must consume only one parameter while producing a Boolean. The elements in the list are the same type as consumed by the predicate.

;; filter: $(\chi \rightarrow \text{Bool})$ (listof χ) $\rightarrow$ (listof χ)

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

A value can now be a primitive operation or a function.

Names of functions are now expressions.

Abstracting from examples

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

(define (name-list slist)
  (cond
    [(empty? slist) empty]
    [else (cons (first (first slist))
                (name-list (rest slist)))]))
```

The abstract list function map

The following is a possible implementation of the built-in function map. You do not need to write it yourself.

```
(define (my-map f alist)
  (cond
    [(empty? alist) empty]
    [else (cons (f (first alist)) (my-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 **my-map**

(my-map sqr (list 3 6 5))

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

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

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

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

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

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

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

7: Local and functional abstraction

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 examples using map:

(define (negate-list numlist) (map − numlist))
(define (name-list slist) (map first slist))

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.

Its contract can be written as follows, where χ and Ψ can be any types.

;; map: $(\chi \rightarrow \Psi)$ (listof χ) $\rightarrow$ (listof Ψ)

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 (list k (+ (* k a) b))

               (points-from (add1 k) n a b))]))

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

These functions create a list by applying an operation to 0,1,...,n-1.

The abstract list function **build-list**

The following are two possible implementations of the built-in function build-list. You do not need to write these.

```
(define (build-from q n fn)
  (cond [(= n q) empty]
        [else (cons (fn q) (build-from (add1 q) n fn))]))
(define (my-build-list n fn) (build-from 0 n fn))


(define (another-build-list n fn)
  (map fn (range 0 n 1)))
```

Tracing **my-build-list**

(my-build-list 3 add1)

$\Rightarrow$ (my-build-from 0 3 add1)

$\Rightarrow$ (cons 1 (my-build-from 1 3 add1))

$\Rightarrow$ (cons 1 (cons 2 (my-build-from 2 3 add1)))

$\Rightarrow$ (cons 1 (cons 2 (cons 3 (my-build-from 3 3 add1))))

$\Rightarrow$ (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.

Again, the contract can be used to show the relationship between the parameters to build-list.

;; build-list: Nat (Nat $\rightarrow \chi$) $\rightarrow$ (listof χ)

Abstracting from examples

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

;; requires: Strings in alist are not empty
(define (concat-firsts alist)
  (cond
    [(empty? alist) " "]
    [else (string-append (substring (first alist) 0 1)
                         (concat-firsts (rest alist)))]))
```

7: Local and functional abstraction

```
(define (list-template alist)
  (cond
    [(empty? alist) …]
    [else (… (first alist) … (list-template (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

The following is a possible implementation of foldr. You do not need to write this.

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

Tracing **my-foldr**

(my-foldr f 0 (list 3 6 5))

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

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

$\Rightarrow$ (f 3 (f 6 (f 5 (my-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.

7: Local and functional abstraction

The contract for foldr, for any types χ and Ψ:

;; foldr: $(\chi\ \Psi \to \Psi)\ \Psi$ (listof χ) $\to \Psi$

The combine function provided to foldr consumes two parameters:

- an item (of type χ) in the list that foldr consumes and

- the result (of type Ψ) 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?

```
;; combine-first: Str Str → Str

;; requires: s is not empty

(define (combine-nonempty s t)
    (string-append (substring s 0 1) t))


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

Consider the function my-length that calculates the length of a list:

```
(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 combine the following to get the new list?

- 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) → (listof Num)
(define (neg-combine item result-on-rest)
   (cons (− item) result-on-rest))



;; negate-list: (listof Num) → (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). Similarly, ormap is provided.

Importantly, andmap and ormap use short-circuit evaluation in the same way and and or do.

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.

7: Local and functional abstraction

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 (constants or functions) 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.

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.

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.

While possible, they are not used that often.

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.

Suppose you are trying to calculate the roots of a quadratic equation $ax^2 + bx + c = 0$. Assuming they exist, they are calculated as and $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$. and $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$.

The function roots

Which implementation do you prefer?

```
(define (roots a b c)
        (list (/ (− (− b) (sqrt (− (sqr b) (∗ 4 a c)))) (∗ 2 a))
              (/ (+ (− b) (sqrt (− (sqr b) (∗ 4 a c)))) (∗ 2 a))))


(define (roots a b c)
    (local
        [(define d (sqrt (− (sqr b) (∗ 4 a c))))]
        (list (/ (− (− b) d) (∗ 2 a))
              (/ (+ (− b) d) (∗ 2 a)))))
```

Using local can improve efficiency

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 significantly 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))])]))
```

7: Local and functional abstraction

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]))]))
```

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

list-max

list-max2
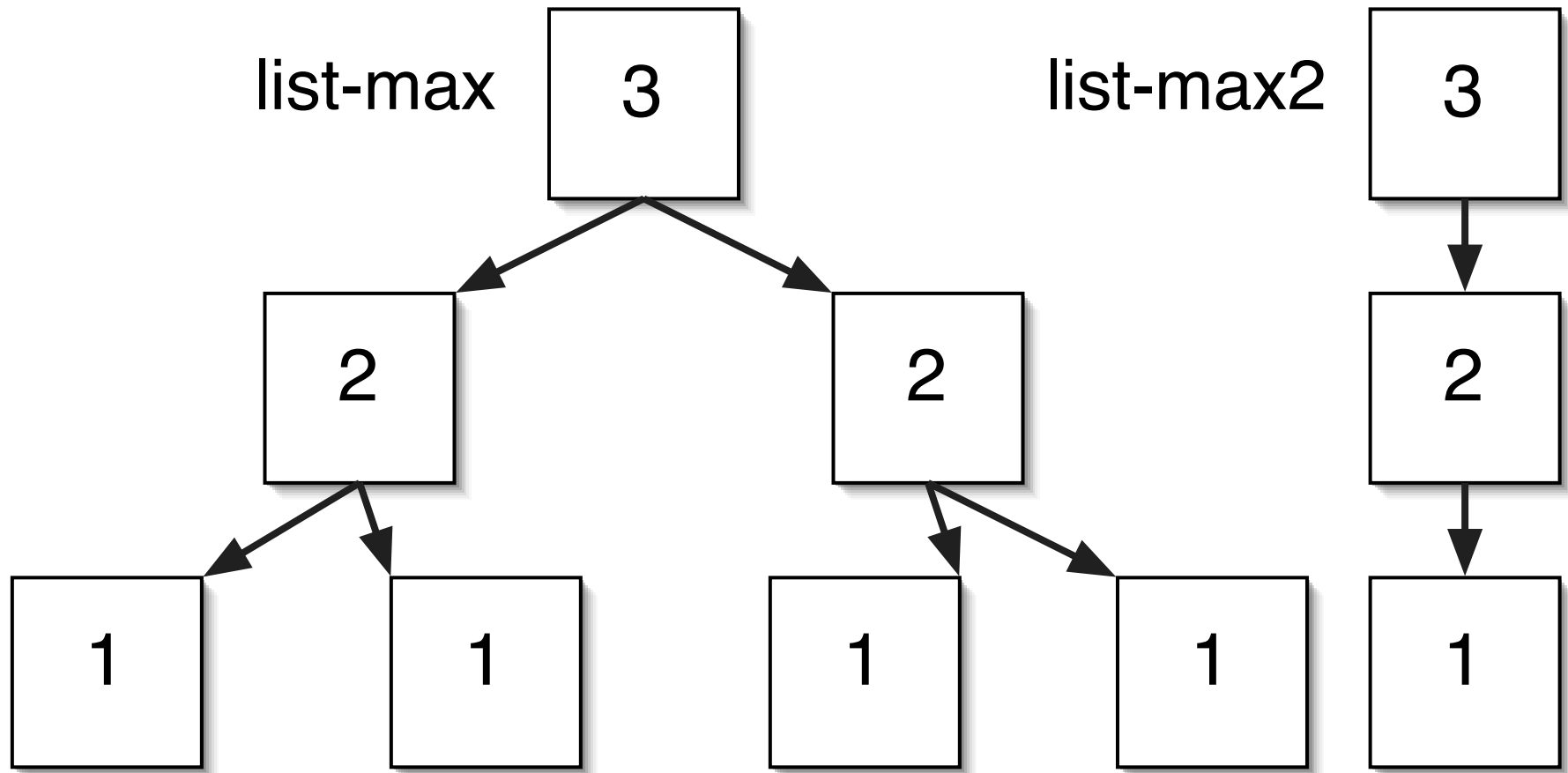
Using **local** for smaller tasks

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 the type Point from Module 05.

```
;; distance: Point Point → Num
(define (distance point1 point2)
  (sqrt (+ (sqr (- (first point1) (first point2)))
           (sqr (- (second point1) (second point2))))))

(define (distance point1 point2)
  (local [(define delta-x (- (first point1) (first point2)))
          (define delta-y (- (second point1) (second point2)))
          (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".

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 (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) → (listof Num)
          ;; requires: alon is sorted in nondecreasing order
          (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)))])])] ...))
```

;; 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.

A contract and purpose are still required but examples and tests are not.

Sometimes we will omit the design recipe due to space limitations on the slides.

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.

7: Local and functional abstraction

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

;;   points (list x y)

;;   where x = 0,1,...,n-1, and y = a*x+b.

(define (first-points n a b)
  (build-list n
       (lambda (x) (list 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 … 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,

((lambda (x) (∗ 2 x)) 4) ⇒ (∗ 2 4) ⇒ 8

((lambda (s t) (+ (string-length s) t)) "hello" 10)

⇒ (+ (string-length "hello") 10)

⇒ (+ 5 10)

⇒ 15

When to use **lambda**?

Use lambda when the function

- is single use

- is reasonably short (2-3 lines)

- does not require recursion.

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

Two solutions - which do you prefer?

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


(define (alt-count-starters phrases start)
  (length (filter (lambda (s) (string=? (substring s 0 1) start))
                  phrases)))
```

Goals of this module

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

You should be familiar with map, filter, foldr, build-list, and sort, and 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.

7: Local and functional abstraction