Tutorial for Week 10: A08 Coverage

Recommended prep for tutorial leaders:

- Familiarize yourself with the content of Assignment 8.
- Open the Tutorials page and be ready to remind students where they can find the slides after the tutorials.
- Make sure you are aware of upcoming deadlines to remind students at the start of the tutorial.
- I have also uploaded a file containing the solutions to the problems below (in case it is difficult to read in this document).

This tutorial will cover the following topics:

- Writing functions that produce and/or consume other functions.
- Practice with local definitions.
- Understanding the heap property.

For all problems in this tutorial, you may only define local helper functions.

Question 1: Line Functions
Write a function (make-line m b) which takes a slope m and an intercept b and produces a function that evaluates the line defined by m and b at the point x (where x is a parameter for the produced function). That is, the produced function should compute \( y = mx + b \).

;; Examples:
(define example-line (make-line 1.5 -5))
(check-expect (example-line 0) -5)
(check-expect (example-line 4.5) 1.3)

Solution:
;; (make-line m b) consumes a slope [m] and an intercept [b] and produces a function that evaluates the line defined by m and b at a position x.

;; Examples:
(check-expect (example-line 0) -5)
(check-expect (example-line 4.5) 1.75)

;; make-line: Num Num -> (Num -> Num)
(define (make-line m b)
  (local ;; (f x) consumes a position [x] and produces the corresponding y coordinate for the line defined by m and b.
    ;; f: Num -> Num
    (define (f x)
      (+ (* m x) b))
    f))

When solving this problem:

- Because functions are first class values, we can define a function using local that can be produced directly in the local body.
• Keep in mind that functions defined in local can access the parameters to the greater function as if they were constants! This is very useful because it allows us to make a function that consumes only x, but uses the values of m and b.
• Remind students of the rules for contracts involving functions that are produced or consumed:
  o We can use the contract for a function as its type.
  o Don’t forget to surround the function’s type with round brackets!
• Also remind students of important style rules involving local:
  o Local functions need a purpose and contract, but not examples.
  o There should be a blank line between each function defined using local.
  o There should be a blank line after the local definitions and before the local body.

Question 2: Manipulating Strings
Write a function (str-change pred? func-change str) to perform custom string modifications. The function consumes a predicate (pred?) that determines which characters to modify and a change function (func-change) that converts characters in the string (str) to their new values.

For example, the code below uses str-change to convert all vowels in a string to uppercase.

```
(define (vowel? ch)
  (member? ch (string->list "aeiou")));; Example:
(check-expect (str-change vowel? char-upcase "this is a test") "thIs Is A tEst")
```

Solution:

```
;; (str-change pred? func-change str) converts the characters in [str] according to the predicate function [pred?] and the conversion function [func-change] and produces a new modified string.

;; Example:
(check-expect (str-change vowel? char-upcase "this is a test") "thIs Is A tEst");; str-change: (Char -> Bool) (Char -> Char) Str -> Str
(define (str-change pred? func-change str)
  (local [;; (str-change/loc loc) converts the characters in loc according to the predicate function pred? and conversion function func-change, producing the modified list of characters.
    ;; str-change/loc: (listof Char) -> (listof Char)
    (define (str-change/loc loc)
      (cond [(empty? loc) empty]
            [(pred? (first loc)) (cons (func-change (first loc))
              (str-change/loc (rest loc)))]
            [else (cons (first loc) (str-change/loc (rest loc)))]))
    (list->string (str-change/loc (string->list str)))))
```

When solving this problem:
• The first step is to ensure that students fully understand the example. The function consumes a string and two functions:
  o One is a predicate that consumes a single character and produces a boolean (true if that character should be modified and false otherwise). In this example, the function vowel? produces true if the character is a vowel (not including y) and false otherwise.
  o The second function tells us what to do with the characters selected by the predicate – in this case, we want to convert them to uppercase using the built-in function char-upcase. Note: To be thorough, you could open the Intermediate Student documentation and show students where this function is defined. The second function must always consume a character and produce the modified version.

• When solving this problem, we really need to work with lists of characters in order to make use of the predicate and change function (which both consume a character). Therefore, we will define str-change as a wrapper function: first, we will convert str to a list of characters, call a helper function to solve the problem, and convert the output back to a string at the end.

• The helper function is fairly straightforward, but there are a few things to point out along the way:
  o Notice that our helper function str-change/loc only needs to consume a list of characters. We can access the predicate and change function directly.
  o At each step, we examine a character from the original string. Based on the result of pred?, we either modify the character using func-change or leave the character unchanged. Either way, the character will be added to the output using cons.
  o It may be a useful exercise to write the helper function first using the specific example of “vowel?” and “char-upcase”. Then show students how replacing those with pred? and func-change produces the same result.

Question 3: Heap Property
In Assignment 8, you were introduced to a new type of binary tree called a heap. Every valid heap must satisfy the Heap Property: every key is less than or equal to every key in its subtrees. Here is the struct definition and data definition for a heap:

```scheme
(define-struct hnode (key left right))
;; A (heapof X) is one of:
;; * empty
;; * (make-hnode X (heapof X) (heapof X))
;; requires: all elements in left are >= key
;;           all elements in right are >= key
```

Write a function (heap? bt) which determines whether a binary tree bt satisfies the heap property. We use the following definitions for binary trees (you may assume that the comparison function <= is used when checking the heap property):

```scheme
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)
;; A binary tree (BT) is one of:
;; * empty
;; * Node
```
;; Examples:
(define example-heap (make-node 1 (make-node 2 empty empty)
  (make-node 3 (make-node 4 empty
    empty)
  (make-node 5 empty
    empty)))))
(define not-a-heap (make-node 100 (make-node 2 empty empty)
  (make-node 3 empty empty)))
(check-expect (heap? example-heap) true)
(check-expect (heap? not-a-heap) false)

Solution:
;; (heap? bt) produces true if the binary tree [bt] is a valid heap
;; according to the comparison function <= and false otherwise.

;; Examples:
(check-expect (heap? example-heap) true)
(check-expect (heap? not-a-heap) false)

;; heap?: BT -> Bool
(define (heap? bt)
  (local [;; (heap-property? child) determines whether the [child]
    satisfies
      ;; the heap property with respect to its parent and the given
      ;; comparison function <=.
      ;; heap-property?: BT -> Bool
      (define (heap-property? child)
        (or (empty? child)
          (<= (node-key bt) (node-key child))))]
    (or (empty? bt)
      (and (heap-property? (node-left bt))
        (heap-property? (node-right bt))
        (heap? (node-left bt))
        (heap? (node-right bt))))))

When solving this problem:
- First, go over the heap data definition and make sure students understand the examples.
  - example-heap is a heap because the root’s key is <= all the keys of its descendants (and the same is true of the node with key 3, which has children 4 and 5).
  - not-a-heap has a root that is greater than both its children’s keys, so it is not a heap.
- There are two key observations to solving this problem:
  - We know that the binary tree is either empty or is an hnode. If it’s empty, we know that it is a valid heap. If it is an hnode, it could be a (heapof Nat), but only if it satisfies the heap property.
To check if a Node satisfies the heap property, we can make the following key observation: we can use the property of transitivity to simplify our solution! If \( a \leq b \) and \( b \leq c \), then \( a \leq c \).

- In the example-tree, \( 1 \leq 3 \) and \( 3 \leq 4 \), so \( 1 \leq 4 \).
- This means that we only need to check the heap property for each node and its children individually. We don’t need to explicitly compare each node to all of its descendants.
- Another way to think about this is to use the recursive (heapof Nat) definition: If my key is \( \leq \) my children’s keys AND my children are both valid heaps (meaning the heap property holds), then I must be a valid heap as well.

• Note that the heap-property? helper function avoids repeated code, because we must perform the same check for each of the children (left and right).

• Finally, here is a clever alternative solution to this problem:
  - Instead of comparing each node’s key to those of its children, which requires checking whether or not a child is empty, what if we could compare a child’s key to that of its parent instead?
  - We can define a helper function called (heap-property? bt parent-key) which consumes a binary tree and a parent-key that is initially set to the key of the root. Then, for each node, we simply need to check if the node’s key is \( \geq \) the key of its parent. If so, we can check if the heap property is satisfied for the children by passing our key in as the parent-key for each child.

• Here is the alternative solution:

```scheme
;; Alternate version of heap? that compares each node to its parent
;; (avoiding the need to check if a child is empty (every child has a
;; parent, but not every parent has a child).

;;; heap-v2?: BT -> Bool
(define (heap-v2? bt)
  (local [;;; (heap-property? bt parent-key) determines whether this binary tree [bt]
  ;; satisfies the heap property, with respect to its parent’s key [parent-key]
  ;; and the comparison function <=
  ;; heap-property?: BT Nat -> Bool
  (define (heap-property? bt parent-key)
    (cond [;; (empty? bt) true]
      [(<= parent-key (node-key bt)) (and (heap-property? (node-left bt) (node-key bt))
        (heap-property? (node-right bt) (node-key bt)))]
      [else false]))
    (cond [;; (empty? bt) true]
      [(<= parent-key (node-key bt)) (and (heap-property? (node-left bt) (node-key bt))
        (heap-property? (node-right bt) (node-key bt)))]
      [else (and (heap-property? (node-left bt) (node-key bt))
        (heap-property? (node-right bt) (node-key bt)))])))
```

```