**CrsA Pre-Assessment Problems**

**1 Programming Problems**

**1.1 Palindrome Detection Modulo Spaces and Capitalization**

A palindrome is a string with the same letters in each of forward and reverse order (ignoring capitalization). Design a program called is‐palindrome that consumes a string and produces a boolean indicating whether the string with all spaces and punctuation

removed is a palindrome. Treat all non-alphanumeric characters (i.e., ones that are not digits or letters) as punctuation.

Examples:

is‐palindrome("a man, a plan, a canal: Panama") is true

is‐palindrome("abca") is false

is‐palindrome("yes, he did it") is false

**1.2 Sum Over Table**

Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths. Design a program sum‐largest that consumes a table of numbers and produces the sum of the largest item from each row.

Assume that no row is empty.

Example:

sum‐largest([list: [list: 1, 7, 5, 3], [list: 20], [list: 6, 9]]) is (7 + 20 + 9)

**1.3 Adding Machine**

Design a program called adding‐machine that consumes a list of numbers and produces a list of the sums of each nonempty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Example:

adding‐machine([list: 1, 2, 0, 7, 0, 5, 4, 1, 0, 0, 6]) is [list: 3, 7, 10]

**2 Review Problems**

Below you are given problem statements followed by multiple solutions to each problem. Assume that the solutions are correct; ignore any small deviations in behavior. Also assume that any missing helper functions are defined in the obvious way. Finally, ignore stylistic differences in naming. Instead, focus on the structure of the solutions.

For each problem, rank the solutions in order (from most to least) of your preference. You are allowed to have ties. Below the ranking grid, explain why you picked that ordering, mentioning briefly all solutions in your response.

The solutions are labeled A, B, etc. Indicate your ordering by selecting the appropriate radio button for each solution. For instance, selecting “1st” for B, “2nd” for A and C, and “3rd” for D means you liked B the *most*, followed by A and C (tied), followed by D.

Remember to explain your choice!

**2.1 Rainfall**

Design a program called rainfall that consumes a list of real numbers representing daily rainfall readings. The list may contain the number ‐999 indicating the end of the data of interest. Produce the average of the nonnegative values in the list up to the first ‐999 (if it shows up). There may be negative numbers other than ‐999 in the list (representing faulty readings). Assume that there is at least one nonnegative number before ‐999.

Example:

rainfall([list: 1, ‐2, 5, ‐999, 8]) is 3

Solution A:

fun rainfall(l :: List<Number>) ‐> Number:

fun helper(rds :: List<Number>, total :: Number, days :: Number) ‐> Number:

cases (List) rds:

| empty => total / days

| link(f, r) =>

ask:

| f == ‐999 then: total / days

| f < 0 then: helper(r, total, days)

| otherwise: helper(r, total + f, days + 1)

end

end

end

helper(l, 0, 0)

end

Solution B:

fun rainfall(l :: List<Number>) ‐> Number:

fun sum‐rfs(shadow l :: List<Number>) ‐> Number:

cases (List) l:

| empty => 0

| link(f, r) =>

ask:

| f == ‐999 then: 0

| f < 0 then: sum‐rfs(r)

| otherwise: f + sum‐rfs(r)

end

end

end

fun count‐days(shadow l :: List<Number>) ‐> Number:

cases (List) l:

| empty => 0

| link(f, r) =>

ask:

| f == ‐999 then: 0

| f < 0 then: count‐days(r)

| otherwise: 1 + count‐days(r)

end

end

end

total = sum‐rfs(l)

days = count‐days(l)

total / days

end

Solution C:

fun rainfall(l :: List<Number>) ‐> Number:

fun cleanse(shadow l :: List<Number>) ‐> List<Number>:

cases (List) l:

| empty => empty

| link(f, r) =>

ask:

| f == ‐999 then: empty

| f < 0 then: cleanse(r)

| otherwise:

link(f, cleanse(r))

end

end

end

actual = cleanse(l)

total = sum‐of(actual)

days = actual.length()

total / days

end

**2.2 Length of Triples**

Design a program called max‐triple‐length that consumes a list of strings and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings. Also assume we are given

data Triple: triple(a, b, c) end

Example:

max‐triple‐length([list: "a", "bb", "c", "dd"]) is 5

Solution A:

fun max‐triple‐length(l :: List<String>) ‐> Number:

fun break‐into‐triples(shadow l :: List<String>) ‐> List<Triple>:

link(triple(l.first, l.rest.first, l.rest.rest.first),

ask:

| is‐empty(l.rest.rest.rest) then: empty

| otherwise: break‐into‐triples(l.rest)

end)

end

triple‐lengths = map(lam(t): string‐length(t.a) + string-length(t.b) + string‐length(t.c) end,

break‐into‐triples(l))

max‐of(triple‐lengths)

end

Solution B:

fun max‐triple‐length(l :: List<String>) ‐> Number:

fun break‐into‐triples(shadow l :: List<Number>) ‐> List<Triple>:

link(triple(l.first, l.rest.first, l.rest.rest.first),

ask:

| is‐empty(l.rest.rest.rest) then: empty

| otherwise: break‐into‐triples(l.rest)

end)

end

triples‐as‐nums = map(string‐length, l)

triple‐lengths = map(lam(t): t.a + t.b + t.c end, break‐into‐triples(triples‐as‐nums))

max‐of(triple‐lengths)

end

Solution C:

fun max‐triple‐length(l :: List<String>) ‐> Number:

shadow l = map(string‐length, l)

fun helper(shadow l :: List<Number>, max‐so‐far :: Number, prev‐2 :: Number,

prev‐1 :: Number) ‐> Number:

cases (List) l:

| empty => max‐so‐far

| link(f, r) =>

prev‐3 = prev‐2 + f

helper(r,

if prev‐3 > max‐so‐far: prev‐3 else: max‐so‐far end,

prev‐1 + f,

f)

end

end

helper(l.rest.rest.rest,

l.first + l.rest.first + l.rest.rest.first,

l.rest.first + l.rest.rest.first,

l.rest.rest.first)

end

**2.3 Shopping Discount**

An online clothing store applies discounts during checkout. A shopping cart is a list of the items being purchased. Each item has a name (a string like “shoes”) and a price (a real number like 12.50). Design a program called checkout that consumes a shopping

cart and produces the total cost of the cart after applying the following two discounts:

* if the cart contains at least 100 worth of shoes, take 20% off the cost of all shoes (match only items whose exact name is "shoes")
* if the cart contains at least two hats, take 10 off the total of the cart (match only items whose exact name is "hat")

Assume the cart is represented as follows:

data CartItem: ci(name :: String, cost :: Number) end

type Cart = List<CartItem>

Example:

checkout(

[list: ci("shoes", 25), ci("bag", 50),

ci("shoes", 85), ci("hat", 15)])

is 153

Solution A:

fun checkout(cart :: Cart) ‐> Number:

shoes = filter(lam(c): c.name == "shoes" end, cart)

shoe‐cost = sum‐of(map(lam(c): c.cost end, shoes))

shoe‐discount = if shoe‐cost >= 100: shoe‐cost \* 0.20 else: 0 end

hats = filter(lam(c): c.name == "hat" end, cart)

hat‐count = hats.length()

hat‐discount = if hat‐count >= 2: 10 else: 0 end

init‐cost = sum‐of(map(lam(c): c.cost end, cart))

init‐cost ‐ shoe‐discount ‐ hat‐discount

end

Solution B:

fun checkout(cart :: Cart) ‐> Number:

fun helper(ct :: Cart, total :: Number, shoe‐cost :: Number, hatcount :: Number) ‐> Number:

cases (List) ct:

| empty =>

shoe‐discount = if shoe‐cost >= 100: shoe‐cost \* 0.20 else: 0 end

hat‐discount = if hat‐count >= 2: 10 else: 0 end

total ‐ shoe‐discount ‐ hat‐discount

| link(f, r) =>

new‐total = total + f.cost

ask:

| f.name == "shoes" then: helper(r, new‐total, shoe‐cost + f.cost, hat‐count)

| f.name == "hat" then: helper(r, new‐total, shoe‐cost, hat‐count + 1)

| otherwise: helper(r, new‐total, shoe‐cost, hat‐count)

end

end

end

helper(cart, 0, 0, 0)

end