**# Daniyar Nazarbayev, H00204990.**

**# exercise H #1**

def mult1 (list\_num):

x = 0

total = 1

while (x<len(list\_num)):

total = total \* list\_num[x]

x = x + 1

return total

**# exercise H #2**

def mult2 (list\_num):

if len(list\_num)==0:

return 1

else:

return list\_num.pop() \* mult2(list\_num)

**# exercise H #3**

# just make a big list and give it as an argument

x = list(range(1,1000001))

# mult2 is not tail recursive.

# each value from each recursive call is necessary to give a proper result,

# so new memory space will be created for every recursive call.

# which can result in a stack overflow error.

**# exercise H #4**

from functools import reduce

def mult3 (list\_num):

if len(list\_num)==0:

return 1

else:

return reduce((lambda x, y: x \* y), list\_num)

**# exercise H #5**

# 60.0 in every case.

# it means that python is not strictly typed.

# in SML's case that would not be possible.

# first of all, lists can only contain the same type

# and even if i were to try to use functions floor and real

# to convert everything to the same type

# i won't be able to, since i cannot check for types, cause SML is static typed.

**# exercise H #6**

# mult3 will be fastest, mult1 second fastest, and mult2 the slowest.

# mult3 is using the reduce function, and reduce carries over

# the value it calculates.

#

# for element in it:

# value = function(value, element)

#

# this is not recursion, this is just a loop that calls a function

# for each element, but it carries the total calculated and the next element

# so it is sort of like tail recursion.

# mult1 is second fastest, i think. It's a loop, and it uses the same variable

# for all its calculations, meaning only a single memory slot

# (don't quote me on that)

#

# total = total \* list[x]

#

# maybe both mult3 and mult1 take the same time, since they don't do a lot

# of assignments, but reuse the same memory space instead.

# mult2 should take the most time, since it does a lot of assignments.

# for each recursive call, it has to allocate memory for the returned values.

# and after the base case is met, all the values are then multiplied.

# when a function is called, it's put in the stack part of the memory

# each recursive call is going to take up a slot in the stack memory

# there is going to be a lot of assignments, and at the end, the program

# will have to extract the value from each of those stack slots.

**# exercise H #7**

def multpoly(list\_poly):

if len(list\_poly)>0:

if type(list\_poly[0]) is str:

x = 0

total = ""

while (x<len(list\_poly)):

total = total + list\_poly[x]

x = x + 1

return total

if type(list\_poly[0]) is list:

x = 0

total = []

while (x<len(list\_poly)):

total.extend(list\_poly[x])

x = x + 1

return total

if type(list\_poly[0]) is int or type(list\_poly[0]) is float:

x = 0

total = 1

while (x<len(list\_poly)):

total = total \* list\_poly[x]

x = x + 1

return total

else:

return 1

**# exercise H #8**

def flatten(list1, list2=None):

if list2 == None:

list2 = []

if type(list1) is list and len(list1)>0:

if type(list1[0]) is list:

temp = list1.pop(0) + list1

return flatten(temp, list2)

else:

list2.append(list1.pop(0))

return flatten(list1, list2)

else:

return list2

# this was a tricky question.

# I wasn't able to make it work with just one variable.

# I created a global variable with an empty

# where all the non list elements would get appended to.

# unfortunately the function itself would not output a list

# i would have to print that global variable to access the new list.

# then i remembered about the reduce function

# which has a local variable already set to None

# i initially set my list2 to None, and there is an if statement that checks

# whether it's None and sets it to a list ([])

# then i append to that list whenever an element in list1 is not of type list.

# and then i send the newly updated list2 as a parameter

# for the next recursive call.