Khoi Duong

Prof. Yang

CE450

10/25/2022

MIDTERM

GitHub link: <https://github.com/MynameisKoi/CE450/tree/main/Midterm>

1. The embedded system's basic requirements consist of hardware and software integrating and working together to perform tasks. In concrete, the needs of the embedded system are microprocessors, microcontrollers, integrated memory (SD card), input/output communication interfaces (peripheral interfaces), and power supply.

The ARM processor architecture sets rules that rule the hardware process when a particular instruction is executed. It is a contract between the hardware and the software, defining how they interact with one another. The ARM processor architecture mainly instructs the computer processor through sets of rules.



Source code with explanation: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/2.py>

def Ton(*now*):

then = 42

def no(*know*): # in the case below, no(4) is executed

# know = 4

no = then # no = 42, however this is a local variable

return *know* \* *now*(*know*) # this equals to 4 \* now(know)

# and now = lambda oh: oh \* 4

# so now(4) = 4 \* 4 = 16

# know \* now(know) = 4 \* 16 = 64

return no # return the result of 'no' function, not the value of local no variable

# 64

then, no = 7,4 # then = 7, no = 4

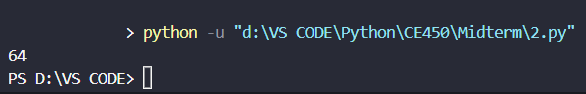
now = lambda *oh*: *oh* \* no # now = lambda oh: oh \* 4

ok = Ton(now)(no) # ok = Ton(lambda oh: oh \* 4)(4)

print(ok)

The high-order function will assign now = lambda oh: oh \* no, and the line know\*now(know) will take the argument of the function “no(know)” from the line “ok = Ton(now)(no)”. This means it will execute “no(no)” with *no* value equals to 4

Run program & result:





Source code & explanation: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/3.py>

woo = 6

def much(*woo*):

if much == *woo*:

# first order of the function:

# since the value of the argument 'woo' is the same as the function name 'much'

# the condition is true and the following code is executed

# second order of the function:

# the function is called again, but this time the argument 'woo' is the value of the local variable 'woo'

# which is 6

# the condition is false

such = lambda *woo*: 5

# the 'such' function will take the second 'woo' value in 'much(much(much))(woo)'

# however, whatever the argument is, the function will always return 5

def *woo*():

# this will change 'woo' from a value to a function name

return such

return *woo*

# the function will return itself

such = lambda *woo*: 4

# in the second order of the function, the 'such' function will take the value of the local variable 'woo'

# which is 6, however, whatever the argument is, the function will always return 4

# such = 4 (local variable of the second order 'much' function)

return *woo*()

# when this line is executed, it goes back to the 'def woo():' line

# it will return the such function (whose parent function is the first order of 'much')

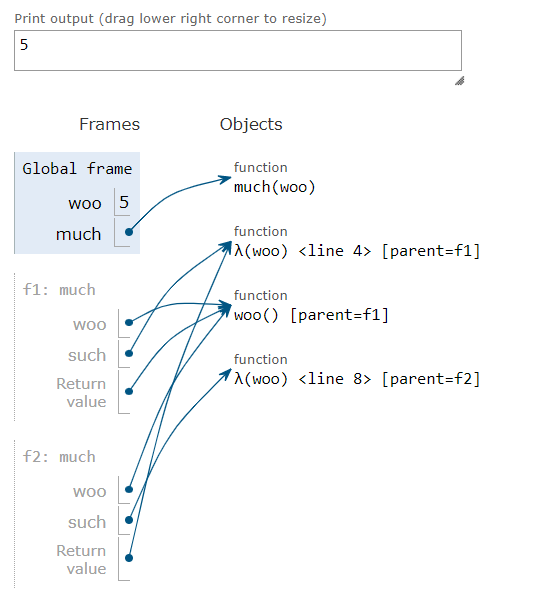
# thus, it will return 5

woo = much(much(much))(woo)

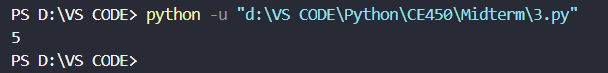
# this assign 'woo' value to the value of 'much' function

print(woo)

Diagram of variables and functions:



Run program & result:





Source code with explanation: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/4.py>

def horn(*hood*):

horn = *hood* # in this case, horn = hood = lambda horn: horn(2)

def *hood* (*horn*):

# first time called:

# the line will change 'hood' from a value to a function name

# now hood != horn of the line above

# second time called:

# hood(horn) = hood(2) => horn = 2

return *horn* # return 2

return horn(*hood*)

# the line will call back to the lambda horn of the code line below

hood = lambda *horn*: *horn*(2)

print(horn(hood))

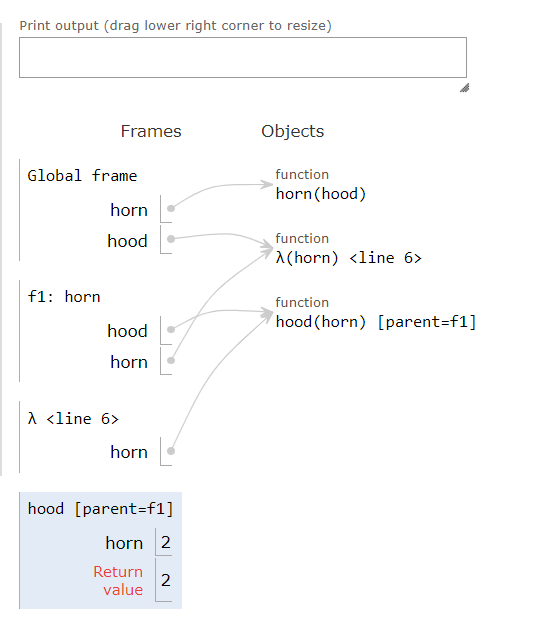
# the function 'horn' takes an argument 'hood'

# which is lambda horn: horn(2)

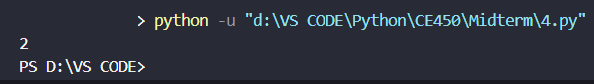
# this will call the 'horn(hood)' function

# horn(lambda horn: horn(2))

Diagram of variables and functions:



Run program & result:





Source code & explanation: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/5.py>

pear = "ni"

def apple(*banana*): # apple(lambda peach: 'ni' + peach)

# banana = lambda peach: 'ni' + peach

def plum(*peach*): # plum(lambda peach: 'ni' + peach)

pear = lambda *pear*: *peach*(*pear*)

# this line will take the argument 'ck' from the 'plum(banana)("ck") line'

# peach('ck') => 'ni' + 'ck' => 'nick'

return pear # pear = 'ck'

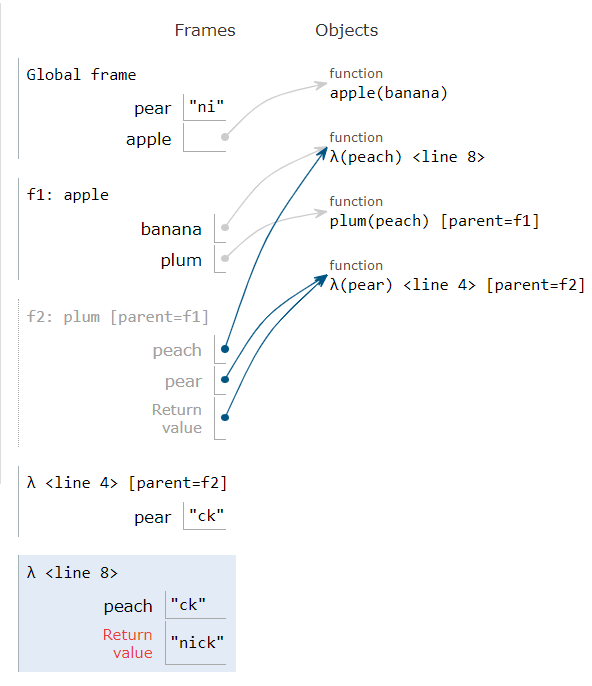
return plum(*banana*)("ck")

# plum(lambda peach: 'ni' + peach)('ck')

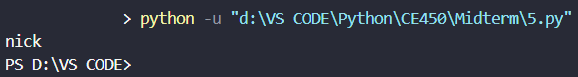
print(apple(lambda *peach*: pear + *peach*)) # peach('ck') => 'ni' + 'ck' => 'nick'

# return 'nick'

Diagram of variables and functions:



Run program & result:





Source code & explanation: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/6.py>

x = "x"

g = x # g = x = "x"

def x(*x*): # x becomes the name of function, x != "x"

g = "h" # local variable g = "h"

if *x* == g: # this false because g = "h" and x != "x"

return *x* + "i"

*x* = lambda *x*: *x*(g) # x = lambda x: x("h")

# now x does not point to the function name 'x' anymore

# this 'x' has the parent function which is the function 'x' above

return lambda *g*: *x*(*g*)

#

x = x(x)(x)

# x(x) = x(lambda x: x("h")) = lambda g: x(g)

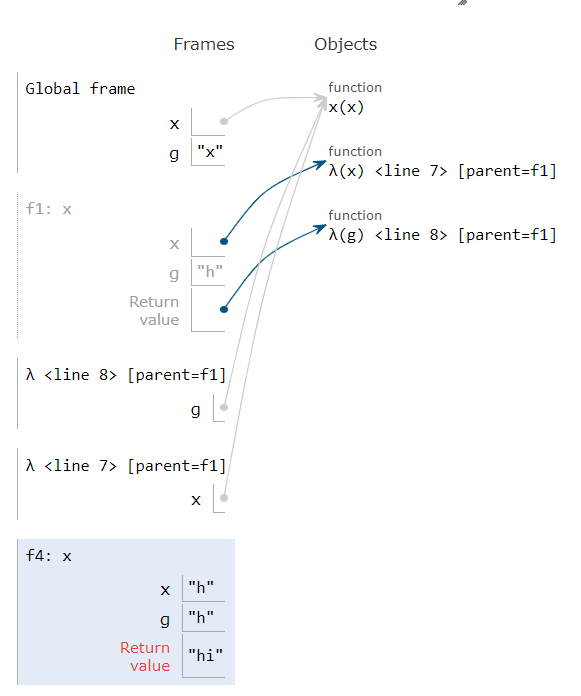
# x(x)(x) = x(lambda x: x("h"))(x) = x("h")

# on the second call of x("h"), the condition is true since argument x = "h" and g = "h"

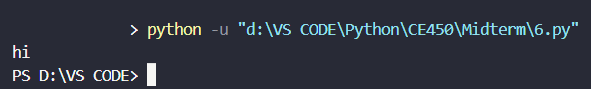
# thus, the function will return "hi"

print(x)

Diagram of variables and functions:



Run program & result:





Source code: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/7.py>

import math

def nrst\_two(*x*):

if *x* <= 0:

return 0

else:

a = math.log2(*x*)

up = math.ceil(a)

down = math.floor(a)

go\_up = 2\*\*up

go\_down = 2\*\*down

if go\_up - *x* > *x* - go\_down:

return go\_down

else:

return go\_up

print("nrst\_two(8) = ", nrst\_two(8)) # 8

print("nrst\_two(11.5) = ", nrst\_two(11.5)) # 8, since it is closer to 2\*\*3 than 2\*\*4

print("nrst\_two(14) = ", nrst\_two(14)) # 16

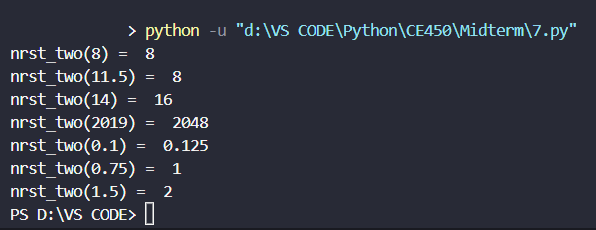
print("nrst\_two(2019) = ", nrst\_two(2019)) # 2048

print("nrst\_two(0.1) = ", nrst\_two(0.1)) # 0.125

print("nrst\_two(0.75) = ", nrst\_two(0.75)) # 1

print("nrst\_two(1.5) = ", nrst\_two(1.5)) # 2

Run program & result:





Source code: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/8.py>

def is\_plndrm(*n*):

# use recursion to check whether input argument is palindrome

if *n* < 10:

return True

else:

if *n* % 10 == *n* // 10 \*\* (len(str(*n*)) - 1):

return is\_plndrm(*n* % 10 \*\* (len(str(*n*)) - 1) // 10)

else:

return False

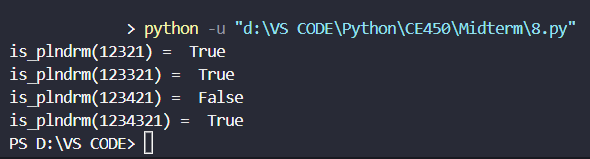
print("is\_plndrm(45654) = ", is\_plndrm(45654)) # True

print("is\_plndrm(42) = ", is\_plndrm(42)) # False

print("is\_plndrm(2019) = ", is\_plndrm(2019)) # False

print("is\_plndrm(10101) = ", is\_plndrm(10101)) # True

Run program & result:





Source code: <https://github.com/MynameisKoi/CE450/blob/main/Midterm/9.py>

def has\_subls(*ls*, *subls*):

# return if the elements of subls appear in ls in any order

# return True if subls is empty

if *subls* == []:

return True

else:

if *subls*[0] in *ls*:

return has\_subls(*ls*, *subls*[1:])

else:

return False

print("has\_subls([], []) = ", has\_subls([], [])) # True

print("has\_subls([3, 3, 2, 1], []) = ", has\_subls([3, 3, 2, 1], [])) # True

print("has\_subls([], [3, 3, 2, 1]) = ", has\_subls([], [3, 3, 2, 1])) # False

print("has\_subls([3, 3, 2, 1], [3, 2, 1]) = ", has\_subls([3, 3, 2, 1], [3, 2, 1])) # True

print("has\_subls([3, 2, 1], [3, 2, 1]) = ", has\_subls([3, 2, 1], [3, 2, 1])) # True

Run program & result:

