Midterm 2 Practice Solutions

CS 111

1.

\_ \_ \_

| X | Y | Z | XYZ + XYZ |

|---|---|---|---------------|

| 0 | 0 | 0 | 0 |

| 0 | 0 | 1 | 0 |

| 0 | 1 | 0 | 1 |

| 0 | 1 | 1 | 0 |

| 1 | 0 | 0 | 0 |

| 1 | 0 | 1 | 1 |

| 1 | 1 | 0 | 0 |

| 1 | 1 | 1 | 0 |

2.

\_\_ \_

XY + XY

3.

--- ---

|X| |Y|

--- ---

| |

|------- |-------

| | | |

| v | v

| ----- | -----

| |NOT| | |NOT|

| ----- | -----

| | | |

| | | | -------

| o------+------+--->| |

| | | | | AND |--------

| | | o--->| | |

| | | | ------- | ------

| | | | |---->| | ----------

| | | | | OR |------>| f(x,y) |

| | | | ------- |---->| | ----------

o------+------+------+--->| | | ------

| | | | | AND |--------

| | | o--->| |

| | | | -------

4.

1 11 111 1111 11111 11111

+ 1101 + 1101 + 1101 + 1101 + 1101 + 1101 + 1101

10111 10111 10111 10111 10111 10111 10111

------- ------- ------- ------- ------- ------- -------

????? 0 00 100 0100 00100 100100

5.

x 1011 x 1011 x 1011 x 1011 x 1011 x 1011 x 1011

11101 11101 11101 11101 11101 11101 11101

------- ------- ------- ------- ------- ------- -------

????? 1011 1011 1011 1011 1011 1011

00000 00000 00000 00000 00000

101100 101100 101100 101100 101100

1011000 1011000 1011000 1011000 1011000

10110000 10110000 10110000 10110000 10110000

---------

100111111

6.

Truth table:

| x\_1 | x\_0 | y\_1 | y\_0 | x\_1x\_0 > y\_1y\_0 |

|-----|-----|-----|-----|-----------------|

| 0 | 0 | 0 | 0 | 0 |

| 0 | 0 | 0 | 1 | 0 |

| 0 | 0 | 1 | 0 | 0 |

| 0 | 0 | 1 | 1 | 0 |

| 0 | 1 | 0 | 0 | 1 |

| 0 | 1 | 0 | 1 | 0 |

| 0 | 1 | 1 | 0 | 0 |

| 0 | 1 | 1 | 1 | 0 |

| 1 | 0 | 0 | 0 | 1 |

| 1 | 0 | 0 | 1 | 1 |

| 1 | 0 | 1 | 0 | 0 |

| 1 | 0 | 1 | 1 | 0 |

| 1 | 1 | 0 | 0 | 1 |

| 1 | 1 | 0 | 1 | 1 |

| 1 | 1 | 1 | 0 | 1 |

| 1 | 1 | 1 | 1 | 0 |

Only rows where result is a 1:

| x\_1 | x\_0 | y\_1 | y\_0 | x\_1x\_0 > y\_1y\_0 |

|-----|-----|-----|-----|-----------------|

| 0 | 1 | 0 | 0 | 1 |

| 1 | 0 | 0 | 0 | 1 |

| 1 | 0 | 0 | 1 | 1 |

| 1 | 1 | 0 | 0 | 1 |

| 1 | 1 | 0 | 1 | 1 |

| 1 | 1 | 1 | 0 | 1 |

Minterm formula:

\_ \_ \_ \_ \_ \_ \_ \_

x\_1x\_0y\_1y\_0 + x\_1x\_0y\_1y\_0 + x\_1x\_0y\_1y\_0 +

\_ \_ \_ \_

x\_1x\_0y\_1y\_0 + x\_1x\_0y\_1y\_0 + x\_1x\_0y\_1y\_0

Circuit:

See file m2practice.circ.

7.

00 read r1 # read a positive number into r1

01 read r2 # read a positive number into r2

02 sub r1 r1 r2 # r1 = r1 - r2

03 mul r1 r1 r1 # square the difference

04 write r1 # output the square of the difference

05 halt

8.

00 setn r1 0 # initialize counter for positive numbers

01 setn r2 0 # initialize counter for negative numbers

02 read r3 # read number from user

03 jeqzn r3 09 # if zero, stop accepting input

04 jltzn r3 07 # if negative, go to negative counter

05 addn r1 1 # otherwise, add 1 to positive counter, and

06 jumpn 08 # go to end of loop

07 addn r2 1 # add 1 to negative counter, and

08 jumpn 02 # go to beginning of loop

09 write r1 # output number of positive numbers

10 write r2 # output number of negative numbers

11 halt

9.

00 read r1 # read number from user

01 setn r2 111 # set register to number that will be output

02 jeqzn r1 08 # if we don't need to call the function again, exit

03 call r14 06 # call the function

04 addn r1 -1 # subtract one from the number of times to call the fn

05 jumpn 01 # go to beginning of loop

06 write r2 # start of function; output 111

07 jumpr r14 # return from function

08 halt

# Exercise 10

def years\_needed(principal, rate, target):

"""Return the number of years needed to reach the

target from the given principal and rate.

Parameters:

principal -- a number

rate -- a number

target -- a number

"""

years = 0

while principal < target:

principal = principal \* (1 + rate)

years += 1

return years

# Exercise 11

def count\_vowels(s):

"""Return the number of vowels in s.

Parameters:

s -- a string

"""

num\_vowels = 0

for ch in s:

if ch in 'AEIOUaeiou':

num\_vowels += 1

return num\_vowels

# Exercise 12

def stars(n):

"""Print a triangle of asterisks with a

height of n and a base of n.

n -- an integer

"""

for i in range(1, n + 1):

for j in range(1, i + 1):

print('\*', end= '')

print()

# Exercise 13

def all\_perfect(lst):

"""Return whether every number in the list is a 100.

Parameters: a list of numbers

"""

for n in lst:

if n != 100:

return False

return True

# Exercise 14

def index\_nearest(n, lst):

"""Return the index of the number closest to the number n.

Parameters:

n -- a number

lst -- a list of numbers

"""

nearest\_val = abs(n - lst[0])

nearest\_index = 0

for i in range(1, len(lst)):

if abs(n - lst[i]) < nearest\_val:

nearest\_val = abs(n - lst[i])

nearest\_index = i

return nearest\_index

# Exercise 15

def num\_appearances(substring, s):

"""Return the number of appearances of the specified substring

(which we assume is of length 2) in the string s.

Parameters:

substring -- a string of length 2

s -- a string

"""

appearances = 0

for i in range(len(s) - 1):

if substring == s[i:i + 2]:

appearances += 1

return appearances

def most\_common\_pair(s):

"""Return the most common substring of length 2 in a string.

Parameters:

s -- a string of length greater than 1

"""

most\_common\_val = s[0:2]

most\_common\_appearances = num\_appearances(most\_common\_val, s)

for i in range(1, len(s) - 1):

val = s[i:i + 2]

appearances = num\_appearances(val, s[i:])

if appearances > most\_common\_appearances:

most\_common\_val = val

most\_common\_appearances = appearances

return most\_common\_val

# Exercise 16

# Solution #1 -- using a single while loop

def longest\_dna(s):

"""Return the longest string of 'A', 'C', 'G', 'T'

characters in a string.

Parameters:

s -- a non-empty string

"""

longest\_streak = 0

longest\_index = 0

current\_streak = 0

i = 0

while i < len(s):

if s[i] not in 'ACGT':

current\_streak = 0

else:

current\_streak += 1

if current\_streak > longest\_streak:

longest\_index = i - current\_streak + 1

longest\_streak = current\_streak

i += 1

return s[longest\_index:longest\_index + longest\_streak]

# Solution #2 -- using a while loop nested in a for loop.

# This approach will typically require more steps,

# but the logic may be somewhat easier to follow.

def longest\_dna2(s):

"""Return the longest string of 'A', 'C', 'G', 'T'

characters in a string.

Parameters:

s -- a non-empty string

"""

longest\_streak = 0

longest\_index = 0

for i in range(len(s)):

if s[i] in 'ACGT':

# find the end of the current streak

j = i + 1

while j < len(s) and s[j] in 'ACGT':

j += 1

# is this the longest streak?

current\_streak = j - i

if current\_streak > longest\_streak:

longest\_streak = current\_streak

longest\_index = i

return s[longest\_index:longest\_index + longest\_streak]

17.

We start by assigning 1 to x and 8 to y, which are global variables:

x = 1

y = 8

global

|-----------|

| x | y |

|-----|-----|

| 1 | 8 |

| | |

When we encounter the first loopy function call, we must evaluate the

variables x and y to their current values, 1 and 8. Therefore, we invoke the

loopy function with values of 1 and 8 for the parameters, which sets loopy's

local variable x to 1 and y to 8. Note that the variables x and y in loopy

are \*not\* the same as the variables x and y in the global scope. We

therefore also now will maintain a table for the first call to loopy:

y = loopy(x, y) --> y = loopy(1, 8)

global first loopy

|-----------| |-----------|

| x | y | | x | y |

|-----|-----| |-----|-----|

| 1 | 8 | | 1 | 8 |

| | | | | |

In the loopy function, we first print some output:

print('starting loopy:', x, y) --> print('starting loopy:', 1, 8)

global first loopy output

|-----------| |-----------| ------

| x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----|

| 1 | 8 | | 1 | 8 |

| | | | | |

We then change the values of x and y inside a while loop. After the loop is

finished, the tables look like:

global first loopy output

|-----------| |-----------| ------

| x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----|

| 1 | 8 | | 1 | 8 |

| | | | 2 | 6 |

| 3 | 4 |

| 4 | 2 |

| | |

We then print some more output:

print('after loop:', x, y) --> print('after loop:', 4, 2)

global first loopy output

|-----------| |-----------| ------

| x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----| after loop: 4 2

| 1 | 8 | | 1 | 8 |

| | | | 2 | 6 |

| 3 | 4 |

| 4 | 2 |

| | |

When we return 4, the return value replaces the function call where we

left off. Where we left off, the return value of the loopy function was

to be assigned to the y variable in global scope:

y = loopy(x, y) --> y = loopy(1, 8) --> y = 4

global first loopy output

|-----------| |-----------| ------

| x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----| after loop: 4 2

| 1 | 8 | | 1 | 8 |

| | 4 | | 2 | 6 |

| 3 | 4 |

| 4 | 2 |

| | |

We then print some output:

print('after first call:', x, y) --> print('after first call:', 1, 4)

global first loopy output

|-----------| |-----------| ------

| x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----| after loop: 4 2

| 1 | 8 | | 1 | 8 | after first call: 1 4

| | 4 | | 2 | 6 |

| 3 | 4 |

| 4 | 2 |

| | |

We then call loopy for a second time. Here are the tables when the second

call to loopy returns:

global first loopy sec. loopy output

|-----------| |-----------| |-----------| ------

| x | y | | x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----| |-----|-----| after loop: 4 2

| 1 | 8 | | 1 | 8 | | 4 | 1 | after first call: 1 4

| | 4 | | 2 | 6 | | | | starting loopy: 4 1

| 3 | 4 | after loop: 4 1

| 4 | 2 |

| | |

When we return 4, the return value replaces the function call where we

left of. Where we left off, the return value of the loopy function

was thrown away. Therefore, the only thing left to do is to print output:

print('after second call:', x, y) --> print('after second call:', 1, 4)

global first loopy sec. loopy output

|-----------| |-----------| |-----------| ------

| x | y | | x | y | | x | y | starting loopy: 1 8

|-----|-----| |-----|-----| |-----|-----| after loop: 4 2

| 1 | 8 | | 1 | 8 | | 4 | 1 | after first call: 1 4

| | 4 | | 2 | 6 | | | | starting loopy: 4 1

| 3 | 4 | after loop: 4 1

| 4 | 2 | after second call: 1 4

| | |

18.

The first assignment creates a new list in memory and assigns a reference

to that list to the variable a:

\_

a |-|-----------> [1, 2, 3, 4]

The second assignment simply assigns the reference held in a to the

variable b. Therefore, no new list is created, but a and b share

a reference to the same list:

\_

a |-|-----------> [1, 2, 3, 4]

^

\_ |

b |-|--------------

Therefore, the following two changes affect the list that both a and

b reference:

a[3] = 5

b[1] = 7

\_

a |-|-----------> [1, 7, 3, 5]

^

\_ |

b |-|--------------

Therefore, the output is:

a is [1, 7, 3, 5]

b is [1, 7, 3, 5]

19.

In Program 1, the foo function accepts only an integer as a parameter,

which is immutable. Therefore, any changes that foo makes to that

variable will be local changes only. To illustrate this, we can look

at the first function call that is made.

To start with, in the global scope the following list is set up:

----------

| \_ |

global | b |-|--+--------> [1, 2, 3]

| |

----------

In the first iteration through the for loop, foo(b[0]) is called. This

evaluates to foo(1), so foo is passed an integer. Note that it is

not passed a reference to the list or to any element of the list.

We get the following picture in memory:

----------

| \_ |

foo | a |1| |

| |

----------

----------

| \_ |

global | b |-|--+--------> [1, 2, 3]

| |

----------

In the foo function, the variable a is doubled. This changes the

local variable a in foo, but has no effect on the list!

----------

| \_ |

foo | a |2| |

| |

----------

----------

| \_ |

global | b |-|--+--------> [1, 2, 3]

| |

----------

Therefore, when the foo function returns, the list is not changed.

----------

| \_ |

global | b |-|--+--------> [1, 2, 3]

| |

----------

The same happens for the next two function calls (foo(2) and foo(3)),

so when the print statement is executed, the list is unchanged and

the output is:

b is [1, 2, 3]

----------------------------------------------------------------------------

In Program 2, the bar function accepts both the (mutable) list and

the index of the number that we want to double. Therefore, any changes

that are made to the list inside of bar will also be reflected when

we return to the global scope, since there is only one list in memory.

To start with, in the global scope the following list is set up:

----------

| \_ |

global | b |-|--+--------> [1, 2, 3]

| |

----------

In he first iteration through the for loop, bar(b, 0) is called.

Note that bar is passed a reference to the list, so we get the

following picture in memory:

----------

| \_ |

| i |0| |

bar | \_ |

|lst|-|--+-----------

| | |

---------- |

---------- |

| \_ | v

global | b |-|--+--------> [1, 2, 3]

| |

----------

In the bar function, the element in the list at index i is

doubled. This changes the list in the following way:

----------

| \_ |

| i |0| |

bar | \_ |

|lst|-|--+-----------

| | |

---------- |

---------- |

| \_ | v

global | b |-|--+--------> [2, 2, 3]

| |

----------

When the bar function returns, the list is still changed since

there is only one list in memory between the two functions:

----------

| \_ |

global | b |-|--+--------> [2, 2, 3]

| |

----------

The same happens for the next two function calls (bar(b, 1) and

bar(b, 2)), so when the print statement is executed, every element

in the list has been doubled and the output is:

b is [2, 4, 6]