CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

1 March 2017 **Time allowed:** 1 hour 30 minutes

Student No:	**A**								

Instructions (please read carefully):

1. Write down your **student number** on the question paper. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet test**. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises **FOUR (4) questions** and **TWENTY (20) pages**. The time allowed for solving this test is **1 hour 30 minutes**.
4. The maximum score of this test is **75 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The pages marked "scratch paper" in the question set may be used as scratch paper.
8. You are allowed to un-staple the sheets while you solve the questions. Please make sure you staple them back in the right order at the end of the test.
9. You are allowed to use pencils, ball-pens or fountain pens, as you like as long as it is legible (no red color, please).

GOOD LUCK!

Question	Marks	Remark
Q1	/ 20	
Q2	/ 20	
Q3	/ 10	
Q4	/ 25	
Total	/ 75	

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: Python Expressions [20 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). Determine the response printed by the interpreter for the final expression entered and **write the exact output in the answer box**. If the interpreter produces an error message, or enters an infinite loop, explain why. You may show your workings **outside the answer box** in the space beside the code. Partial marks will be awarded for workings if the final answer is wrong.

A. [4 marks]

```
x = 7
y = 11
def f(y):
    return y - x
def g(x):
    x =  5
    return f(x * y)
print(g(y))
```

B. [4 marks]

```
t = (1, 2)
def foo(tup):
    if tup == ():
        return 0
    else:
        return (tup[0], foo(tup[1:]))
print(foo((t, t)))
```

C. [4 marks]

```
f = lambda f, g: (f, (g))
g = lambda g: g(g, g)
print(g(f)[0](4, 2))
```

D. [4 marks]

```
result = 0
for i in range(1, 13, 3):
    if i % 2 == 0:
        i += 2
    else:
        result //= i
    result += i
print(result)
```

E. [4 marks]

```
a, b, c = "easy", "east", "easter"
if a < b:
    a, b = b, a
else:
    if b < c:
        a += b
b = a + c
print(a)
print(b)
print(c)
```

Question 2: Shish Kebab [20 marks]

You have found a part-time job at a middle-eastern cafe and have been tasked to grill shish kebabs.

"A shish kebab is a skewer with meat and vegetables and is usually grilled. It can contain lamb, beef, fish, or chicken, as well as vegetables like green peppers, onions, and mushrooms."

Source: thespruce.com

Figure 1: A shish kebab

For this question, a kebab is represented by a string of digits, and each digit represents a piece of meat or vegetable in the kebab. The value of the digit represents the time needed to cook the respective piece in that position. For example, the string `"3333"` represents a kebab made of 4 pieces of food with each piece requiring 3 mins cooking time on the grill.

A. [Warm up] As a new intern, you are first given a micro-grill that is large enough to cook only one piece of the kebab at a time.

The function `grill_piece` which takes as input a kebab (which is a string), a position (which is an integer), and a time in minutes (which is an integer).

The function returns a kebab with the piece at the given position grilled for the given time. That is, the number of minutes required to cook that particular piece would have decreased by the amount of time spent on the grill.

If the amount of time being grilled is more than the cooking time required, then the digit representing the piece will become the string `"X"`.

Note that the chefs are actually programmer too, so they will count the pieces starting from 0! You may also assume that the kebab does not contain any burnt pieces.

Examples:

```
>>> grill_piece("333", 0, 2) # will grill the first piece for 2 mins
"133"

>>> grill_piece("333", 1, 3) # will grill the second piece for 3 mins
"303"

>>> grill_piece("333", 2, 4) # will grill the third piece for 4 mins
"33X"
```

Provide an implementation for the function `grill_piece`. [4 marks]

```
def grill_piece(kebab, at, time):
```

After mastering the art of cooking kebabs without burning them on the single flame grill, you are now given access to larger grills to cook the kebabs. Yay.

However, these larger grills emit very different flame patterns. A flame pattern is also represented by a string of digits. Each digit represents the amount of heat that particular position of the flame will produce, with 1 being the norm, 2 being twice as much, 3 being thrice, and so on.

For example, when the flame `"121"` is used to grill a kebab for one minute, the pieces at the edges of the flame will be cooked as per norm but the piece at the centre will be cooked twice as fast, and will appear as if it has been cooked for 2 mins. Thus, a `"333"` kebab when cooked using a `"121"` flame for one minute will become a `"212"` kebab. Cooking it a further one minute with the same flame will result in a `"1X1"` burnt kebab.

The function `grill_on_flame` takes as input a kebab (which is a string), a flame pattern (which is a string), and a starting position (which is an integer), and grills the kebab over the flame pattern at the given starting position for one minute.

You can assume that the flame pattern of the grill will never be longer than the kebab and the entire flame will always be positioned within the kebab.

Examples:

```
>>> grill_on_flame("3333", "121", 1)  # grills with flame  "3333"
"3212"                                # from second piece    1̂2̂1̂

>>> grill_on_flame("3333", "121", 0)  # grills with flame  "3333"
"2123"                                # from first piece    1̂2̂1̂

>>> grill_on_flame("3333333", "212", 2)  #  "3333333"
"3312133"                                #     2̂1̂2̂

>>> grill_on_flame("2XX2", "1111", 0)    # already burnt
"1XX1"                                   # stays burnt
```

The function `grill_piece` from part A is given and works correctly **even when given a kebab with burnt pieces**. You may call this function in your implementation.

B. Provide an **<u>iterative</u>** implementation of the function `grill_on_flame`. [4 marks]

```
def grill_on_flame(kebab, flame, at):
```

C. What is the order of growth in terms of time and space for the function you wrote in Part (B). You may assume that `grill_piece` runs in constant time and space if you have used it in your function. Briefly explain your answer. [2 marks]

Time:

Space:

D. Provide a recursive implementation of the function `grill_on_flame`. [4 marks]

```
def grill_on_flame(kebab, flame, at):
```

E. What is the order of growth in terms of time and space for the function you wrote in Part (D). You may again assume that `grill_piece` runs in constant time and space if you have used it in your function. Briefly explain your answer.

Hint: Note the time and space complexity of string slicing. [2 marks]

Time:

Space:

F. Now you think it is quite troublesome to have to place the flame on arbitrary positions of the kebab and decide to just evenly run the flame along the kebab for one minute at a time.

In other words, the flame will be placed starting at the first piece for one minute, then starting at the second for the next minute, then the third and so on until the flame reaches the end of the kebab.

For example, the kebab `"33333"` when grilled with a `"121"` flame using this strategy will look like:

```
kebab:  33333  >>  21233  >>  20023  >>  20X02
flame:  1̂2̂1̂         1̂2̂1̂         1̂2̂1̂
```

Write a function `grill_evenly` that takes as input a kebab and a flame, and returns a kebab that has been grilled using the above-mentioned strategy.

Example:

```
>>> grill_evenly("33333", "121")
"20X02"    # Noo.. burnt kebab :(

>>> grill_evenly("13431", "121")
"00000"    # Yay! Perfectly cooked.
```

Hint: You may assume the functions `grill_piece` *and* `grill_on_flame` *defined in the previous parts are given and correct.* [4 marks]

```
def grill_evenly(kebab, flame):
```

Question 3: Higher-Order Shish Kebab [10 marks]

INSTRUCTIONS: Parts B and C should be solved using the given higher-order function, and not by recursion OR iteration.

A. Ok, so trying to grill a kebab evenly was not such a good idea and you ended up burning most of them. However, you still wish to make your job easier.

Rather than having to look up the manual to figure out the flame pattern for a particular grill, you think you can use a higher-order function to define a grill of a given flame pattern, then use this grill to cook your kebabs without needing the flame pattern as the input.

Example:

```
>>> lousy_grill = make_grill("121")  # grill has lousy flame pattern
>>> lousy_grill("3333" , 0)
"2123"

>>> nice_grill = make_grill("111")  # grill has nice flame pattern
>>> nice _grill("3333" , 0)
"2223"
```

Provide an implementation of the function `make_grill`. You may reuse the functions defined in Question 2. [2 marks]

```
def make_grill(flame):
```

Consider this higher-order function:

```
def tail(f, a, n):
    if n == 0:
        return a
    else:
        return tail(f, f(n, a), n-1)
```

B. **[Warning: HARD]** It turns out that the function `grill_on_flame` in Question 2 can be defined in terms of the higher-order function `tail` as follows:

```
def grill_on_flame(kebab, flame, at):
    <PRE>
    return tail(<T1>,
                <T2>,
                <T3>)
```

Please provide possible implementations for the terms `T1`, `T2` and `T3`. You may use the function `grill_piece` that was defined previously for this part, and may also optionally define other functions in `<PRE>` if needed. [4 marks]

*optional `<PRE>`:

`<T1>`:

`<T2>`:

`<T3>`:

C. **[Warning: HARD]** We can also define the function `grill_evenly` in Question 2 in terms of the higher-order function `tail` as follows:

```
def grill_evenly(kebab, flame):
    <PRE>
    return tail(<T4>,
                <T5>,
                <T6>)
```

Please provide possible implementations for the terms `T4`, `T5` and `T6`. You may use the function `grill_on_flame` that was defined previously for this part, and may also optionally define functions in `<PRE>` if needed. [4 marks]

*optional `<PRE>`:

`<T4>`:

`<T5>`:

`<T6>`:

Question 4: Chemistry [25 marks]

INSTRUCTIONS: Please read the entire question clearly before you attempt this problem!! You are also not to use any Python data types which have not yet been taught in class.

In Chemistry, we learn that objects in the world are made up of molecules, which are in turn made up of atoms. For this question, you are to model atoms and molecules.

An atom can be vaguely represented by a symbol, which is a string and a mass number, which is an integer.

The following functions support the atom data type:

- `make_atom(symbol, mass)` takes as input the symbol and the mass of the atom, and returns an atom data type.
- `symbol_of(atom)` takes as input an atom, and returns the symbol of the atom.
- `mass_of(atom)` takes as input an atom, and returns the mass of the atom.

Sample run:

```
>>> hydrogen = make_atom("H", 1)

>>> symbol_of(hydrogen)
"H"

>>> mass_of(hydrogen)
1
```

A. Explain how you will use tuples to represent an atom. You may use a box-pointer diagram to aid your explanation. [2 marks]

B. Provide an implementation for the functions `make_atom`, `symbol_of` and `mass_of`.
[2 marks]

```
def make_atom(symbol, mass):




def symbol_of(atom):




def mass_of(atom):
```

[Important!] For the remaining parts of this question, **you should not break the abstraction of an atom in your code.**

Molecules are simply made up of atoms bonded together. In addition, a molecule should also contain a name, which is a string. The molecule data type should support the following functions:

- `make_empty_molecule(name)` takes as input a name, which is a string, and returns a molecule of the given name with no atoms.
- `get_name(molecule)` takes as input a molecule, and returns the name of the molecule.
- `add_atom(atom, molecule)` takes as inputs an atom and a molecule, and returns a new molecule with the atom added to it.
- `remove_atom(atom, molecule)` takes as inputs an atom and a molecule, and returns a new molecule with one of such atom removed from it. If atom is not found in molecule, the input molecule is simply returned.

C. Explain how you will use tuples to represent a molecule. You may use a box-pointer diagram to aid your explanation. [2 marks]

D. Provide an implementation for the functions `make_empty_molecule`, `get_name`, `add_atom` and `remove_atom`. [5 marks]

```
def make_empty_molecule(name):




def get_name(molecule):




def add_atom(atom, molecule):




def remove_atom(atom, molecule):
```

E. Write a function `count_symbols` that takes as input a symbol, which is a string, and a molecule and returns the number of atoms with the given symbol that is contained in the molecule.

You may wish to use the higher-order functions in the Appendix. [4 marks]

```
def count_symbols(symbol, molecule):
```

F. Write a function `add` which takes as inputs two molecules, and creates a chemical reaction which returns a large molecule that contains all the atoms of both molecules. The name of this new molecule is simply both names concatenated together. [2 marks]

```
def add(m1, m2):
```

Instead of using `add_atom` to add an atom to a molecule, a chemist accidentally used the `add` function instead and got a bad result when he tried to use the `count_symbols` function.

G. Explain what will be the result of using the function `add` on an atom and molecule given your implementation. You may use examples or illustrations to aid your explanation. [4 marks]

H. Now the chemist thinks he can modify the function `count_symbols` to recursively search deeply to still work with wrongly added molecules. Do you think his idea will work given your implementation of both the atom and molecule data types? Please explain why. You may explain in writing or in code. [4 marks]

Appendix

The following are some functions that were introduced in class. For your reference, they are reproduced here.

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) * product(term, next(a), next, b)

def fold(op, f, n):
  if n == 0:
    return f(0)
  else:
    return op(f(n), fold(op, f, n-1))

def enumerate_interval(low, high):
    return tuple(range(low,high+1))

def map(fn, seq):
    if seq == ():
        return ()
    else:
        return (fn(seq[0]),) + map(fn, seq[1:])

def filter(pred, seq):
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],) + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])

def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))
```

Scratch Paper

Scratch Paper

— E N D O F P A P E R —