CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

3 Oct 2018 **Time allowed:** 1 hour 30 minutes

Instructions (please read carefully):

1. This is **an open-sheet test**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
2. The QUESTION SET comprises **FOUR (4) questions** and **TEN (10) pages**, and the ANSWER SHEET comprises of **TEN (10) pages**.
3. 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. You must submit only the **ANSWER SHEET** and no other documents. The question set may be used as scratch paper.
8. Use of calculators are not allowed in 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).
10. **Marks may be deducted** for i) unrecognisable handwriting, and/or ii) excessively long code. A general guide would be not more than twice the length of our model answers.

GOOD LUCK!

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: Python Expressions [25 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, **state and explain why**. You may show your workings **outside the answer box**. Partial marks may be awarded for workings even if the final answer is wrong.

The code is replicated on the answer booklet. 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.

```
x = 2
y = 3
def f(x, y):
    x = x * 10
    y = y + x
    return y
result = f(f(y, x), y-x)
print(result)
```

[5 marks]

B.

```
a = (1, 2)
b = (a, 3, 4)
c = a + b + (5, 6)
print(c)
```

[5 marks]

C.

```
t = 'test'
if t == "test":
    print('pass')
if t == 't':
    print('T')
elif t*2:
    print('T_T')
else:
    print(wake-up)
```

[5 marks]

D.

```
n = 5
while True:
    print(n)
    if n % 3 == 0:
        break
    elif n % 2 == 0:
        n = n//2
        continue
    n = n-1
```

[5 marks]

E.

```
def f(x, y):
    return lambda z: (x)(y)(z)
def g(x):
    return lambda y: y(x)
print(f(g, 0)(g)(42))
```

[5 marks]

Question 2: Double Trouble [18 marks]

The function `num_pairs` takes as input a string and returns the number of consecutive character pairs in the string. For example, the strings `'balloon'` and `'missisippi'` given as input both return 2 because "balloon" has the pairs "ll" and "oo" while "missisippi" has the pairs "ss" and "pp".

Note that a character may count towards multiple pairs. For example, the string `'crosssection'` contains 2 pairs of "ss", and the string `'ahhhh!!'` contains 4 pairs (3 "hh" and 1 "!!").

A. Provide a recursive implementation of the function `num_pairs`. [4 marks]

B. State the order of growth in terms of time and space for the function you wrote in Part (A). Briefly explain your answer. [2 marks]

C. Provide a iterative implementation of the function `num_pairs`. [4 marks]

D. State the order of growth in terms of time and space for the function you wrote in Part (C). Briefly explain your answer. [2 marks]

E. The function `num_consec(s, n)` takes as input a string s and an integer $n \geq 2$, and returns the number of n-consecutive character groupings found in s. For example, the string `'ahhhh!!'` contains two instances of 3-consecutive characters "hhh", therefore `num_consec('ahhhh!!', 3)` will return `2`.

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

F. Is your implementation in Part (E) recursive or iterative? State the order of growth in terms of time and space of your implementation and briefly explain your answer. [2 marks]

Question 3: Higher-Order Doubles [10 marks]

A. Consider the higher-order function `fold` which was taught in class.

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

The function `num_pairs` in Question 2 can be defined in terms of `fold` as follows:

```
def num_pairs(s):
    <PRE>
    return fold(<T1>, <T2>, <T3>)
```

Please provide possible implementations for the terms `T1`, `T2`, and `T3`. You may optionally define other functions in `PRE` if needed.

Note you are to use the higher-order function and not solve it recursively or iteratively.

[4 marks]

B. [Warning: HARD] Your friendly biologist friend, Watson, has used Python strings to represent genetic sequences that he is working with. He has also used some *string manipulation functions (SMF)* to help modify and manipulate his genetic sequences. **A string manipulation function is a function that takes in a string and returns a new and possibly modified string**.

Watson is currently researching on and have managed to construct genetic sequences that do not contain consecutive identical characters. However, he is concerned that some of the SMFs that he plans to use will introduce consecutive identical characters into his genes.

For example,

```
>>> my_gene = 'atgcgtacgatc'  # gene with no duplicate consecutive characters
>>> every_two(my_gene)        # SMF that removes every second character
'aggagt'                      # result has duplicate consecutive characters
```

Watson wish that the **SMFs will instead return** `None` if the resulting sequence contains consecutive characters. Unfortunately, he does not have access to the function definitions and can not modify them directly.

However, he has heard that it is possible to define a higher-order function as a *decorator* that can be used to redefine an pre-existing function, like so:

```
>>> every_two = assert_single(every_two)   # redefine every_two
>>> every_two(my_gene)
None     # return value is None when result has double letters

>>> every_two('atgcagctacg')
'agacag'   # Otherwise works normally

>>> rotate('atgcagctacg')  # SMF that swaps every pair
'tacggatccag'              # result has double letters

>>> rotate = assert_single(rotate)  # redefine rotate
>>> rotate('atgcagctacg')
None      # return value is None when result has double letters

>>> rotate('atgcgtacagtc')
'tacgtgcagact'  # Otherwise works normally
```

Watson did not learn how to use higher-order functions and will need your help.

Implement the function `assert_single` that behaves according the the sample execution shown above. You may reuse functions that were defined earlier. [6 marks]

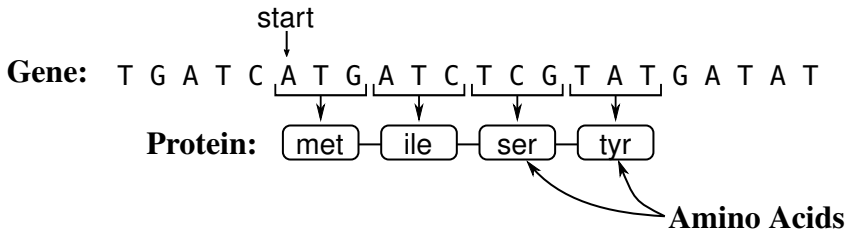
Question 4: Genetics [22 marks]

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.

Amino acids are compounds that combine to make up proteins, which are essential for organic life. In other words, a protein consists of a sequence or chain of amino acids.

Proteins can be expressed from genes. A gene is section of DNA which consists of a sequence of nucleotide bases. There are four possible nucleotide bases, represented as the strings `'A'`, `'T'`, `'G'`, `'C'`.

To express a protein from a gene, we begin by reading from a given starting position in the gene nucleotide sequence. Every 3 bases will encode for an amino acid, which will be grouped together to form a protein.



An amino acid can be coded from different bases. For example, the base sequences `'ATC'` and `'ATA'` both will encode the amino acid Isoleucine (ile).

Your other friendly biologist friend, Crick, has created an `amino` acid data type, which is supported by the following functions:

- `make_amino(seq)` takes in a string of three characters and returns an `amino` acid which the sequence codes for.
- `amino_name(amino)` takes in an `amino` acid, and returns the `name` of the amino acid.

Amino acids are considered to be equivalent if they have the same name.

A. Crick is interested in sequences, or chains, of amino acids. A *chain* of amino acids can be represented using a tuple.

Implement a function `equal_chains(c1, c2)` that takes as inputs two chains of amino acids (represented as tuples). The function returns `True` if both chains contains the *exact* same sequence of amino acids, and `False` otherwise. [4 marks]

For our purposes, it is sufficient for a *protein* to be represented by a name and the its sequence of amino acids.

Design a data structure to represent a protein, which is supported by the following functions:

- `make_protein(name)` creates and returns a new protein of the given name, containing no amino acids.
- `protein_name(protein)` returns the name of the protein.
- `protein_seq(protein)` returns the sequence of amino acids of the protein as a tuple.
- `add_amino(amino, protein)` takes as inputs an amino acid and a protein, returns a new protein with the amino acid appended to the end of its current sequence of amino acid.

Consider the following sample run:

```
>>> met = make_amino('ATG')
>>> tyr = make_amino('TAT')

>>> myelin = make_protein('Myelin')
>>> myelin = add_amino(met, myelin)
>>> myelin = add_amino(tyr, myelin)
>>> myelin = add_amino(met, myelin)
```

B. Draw the **box-pointer diagram** of `myelin`, `met` and `tyr` at the end of the sample run above. Your diagram should be consistent with your implementations of a protein in part (C). [2 marks]

C. Provide an implementation of the functions `make_protein`, `protein_name`, `protein_seq` and `add_amino`. [6 marks]

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

D. Two proteins are considered to be equivalent if they have the same name and same sequence of amino acids. Implement the function `equal_proteins(p1, p2)` that takes two proteins as input and returns `True` if the two proteins are equivalent, and `False` otherwise. [4 marks]

E. Similar to proteins, it is sufficient for a gene to be represented by its name and sequence of nucleotide bases.

Design a data structure to represent a gene, which is supported by the following functions:

- `make_gene(name, seq)` takes two strings as input, the first being the name of the gene, and the second, the sequence of nucleotide bases.

- `express_protein(name, start, length, gene)` takes as input a string, two integers and a gene. It returns a new protein of the given name that contains a sequence of amino acids that is encoded from the start position in the gene's nucleotide base sequence, for the given length.

Sample execution:

```
>>> gene = make_gene('T708', 'TGATCATGATCTCGTATGATA')

>>> actin = express_protein("Actin", 5, 4, gene)
# expresses the protein shown in the figure above, which contains
# a sequence of amino acids met-ile-ser-tyr (ATG)-(ATC)-(TCG)-(TAT)
```

Provide an implementation for the functions `make_gene` and `express_protein`. You may assume the inputs are always valid and will not express past the length of the gene sequence.

[6 marks]

— END OF QUESTIONS —

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

— E N D O F P A P E R —

CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test — Answer Sheet

3 Oct 2018 **Time allowed:** 1 hour 30 minutes

Student No:	**A**								

Instructions (please read carefully):

1. Write down your **student number** on this answer sheet. DO NOT WRITE YOUR NAME!
2. This answer sheet comprises **TEN (10) pages**.
3. All questions must be answered in the space provided; no extra sheets will be accepted as answers. You may use the extra page at the back if you need more space for your answers.
4. You must submit only the **ANSWER SHEET** and no other documents. The question set may be used as scratch paper.
5. You are allowed to use pencils, ball-pens or fountain pens, as you like as long as it is legible (no red color, please).
6. **Marks may be deducted** for i) unrecognisable handwriting, and/or ii) excessively long code. A general guide would be not more than twice the length of our model answers.

GOOD LUCK!

For Examiner's Use Only

Question	Marks	Remarks
Q1	/ 25	
Q2	/ 18	
Q3	/ 10	
Q4	/ 22	
Total	/ 75	

This page is intentionally left blank.
Use it ONLY if you need extra space for your answers, in which case indicate the **question number clearly**. **DO NOT** use it for your rough work.

Question 1A

[5 marks]

```
x = 2
y = 3
def f(x, y):
    x = x * 10
    y = y + x
    return y
result = f(f(y, x), y-x)
print(result)
```

Question 1B

[5 marks]

```
a = (1, 2)
b = (a, 3, 4)
c = a + b + (5, 6)
print(c)
```

Question 1C

[5 marks]

```
t = 'test'
if t == "test":
    print('pass')
if t == 't':
    print('T')
elif t*2:
    print('T_T')
else:
    print(wake-up)
```

Question 1D

[5 marks]

```
n = 5
while True:
    print(n)
    if n % 3 == 0:
        break
    elif n % 2 == 0:
        n = n//2
        continue
    n = n-1
```

Question 1E

[5 marks]

```
def f(x, y):
    return lambda z: (x)(y)(z)
def g(x):
    return lambda y: y(x)
print(f(g, 0)(g)(42))
```

Question 2A

[4 marks]

```
def num_pairs(s):
```

Question 2B

[2 marks]

Time:

Space:

Question 2C

[4 marks]

```
def num_pairs(s):
```

Question 2D

[2 marks]

Time:

Space:

Question 2E

[4 marks]

```
def num_consec(s, n):
```

Question 2F

[2 marks]

My implementation is **RECURSIVE / ITERATIVE** (circle/delete accordingly)

Time:

Space:

Question 3A

[4 marks]

*optional `<PRE>:`

`<T1>:`

`<T2>:`

`<T3>:`

Question 3B

[6 marks]

```
def assert_single(                ):  # fill in the correct parameters
```

Question 4A

[4 marks]

```
def equal_chains(c1, c2):
```

Question 4B

[2 marks]

Question 4C

[6 marks]

```
def make_protein(name):




def protein_name(protein):




def protein_seq(protein):




def add_amino(amino, protein):
```

Question 4D

[4 marks]

```
def equal_proteins(p1, p2):
```

Question 4E

[6 marks]

```
def make_gene(name, seq):
```

```
def express_protein(name, start, length, gene):
```

— END OF ANSWER SHEET —

Question 1A

[5 marks]

```
x = 2
y = 3
def f(x, y):
    x = x * 10
    y = y + x
    return y
result = f(f(y, x), y-x)
print(result)
```

`321`
Tests the understanding of scoping and whether students are careful with parameter substitution.
+2 ea: correct evaluation of `f(y, x)` or `f(sub_result, diff)`
+1: correctly evaluates `y-x`

Question 1B

[5 marks]

```
a = (1, 2)
b = (a, 3, 4)
c = a + b + (5, 6)
print(c)
```

`(1, 2, (1, 2), 3, 4, 5, 6)` Tests understanding of tuple addition.
-2 ea: incorrect evaluation of `(a, 3, 4)` or `a + b` or `b + (5,6)`
~~(min 0 marks)~~

Question 1C

[5 marks]

```
t = 'test'
if t == "test":
    print('pass')
if t == 't':
    print('T')
elif t*2:
    print('T_T')
else:
    print(wake-up)
```

`'pass'` +2
`'T_T'` +3
+1: points out error is not thrown if `wake-up` not executed
~~(max 5 marks)~~

Question 1D

[5 marks]

```
n = 5
while True:
    print(n)
    if n % 3 == 0:
        break
    elif n % 2 == 0:
        n = n//2
        continue
    n = n-1
```

```
5
4
2
1
0
```

Tests the understanding of `while` loops, and `break` and `continue`

+2: skips 3 due to `n=n//2` when `n=4`
+1: skips `n=n-1` when `n=4` due to `continue`
+2: `break` when `n=0`

Question 1E

[5 marks]

```
def f(x, y):
    return lambda z: (x)(y)(z)
def g(x):
    return lambda y: y(x)
print(f(g, 0)(g)(42))
```

Error. Evaluates to `42(0)` and an error occurs, as integer 42 is not a function.
Test knowledge of evaluation of lambda expressions

+2: state that error occurred.
+3: explains `42(0)` is the cause of error
OR
3: state `TypeError` without explanation
1: correctly evaluates `f(g,0)` but did not state error occurred
3: correctly evaluates to `42(0)` but did not state error occurred

Question 2A

[4 marks]

```
def num_pairs(s):
    if len(s) < 2:
        return 0
    elif s[0] == s[1]:
        return 1 + num_pairs(s[1:])
    else:
        return num_pairs(s[1:])
```

Question 2B

[2 marks]

Time: $O(n^2)$, where n = len(s). There is a total of n recursive calls, each requiring creating a new tuple of length n.

Space:$O(n^2)$, where n = len(s). There is a total of n recursive calls, each requiring creating a new tuple of length n.

Question 2C

[4 marks]

```
def num_pairs(s):
    count = 0
    for i in range(1, len(s)):
        if s[i-1] == s[i]:
            count += 1
    return count

def num_pairs(s):
    count, last = 0, ''
    for c in s:
        if last == c:
            count += 1
        last = c
    return count
```

Question 2D

[2 marks]

Time: $O(n)$, the loop will iterate n times.

Space:$O(1)$, no extra memory is needed because the variables are overwritten with the new values.

Question 2E

[4 marks]

```
def num_consec(s, n): # recursive
    if len(s) < n:
        return 0
    elif s[0]*n == s[:n]:
        return 1 + num_consec(s[1:], n)
    else:
        return num_consec(s[1:], n)


def num_consec(s, n): # iterative
    count = 0;
    for i in range(len(s)):
        if s[i]*n == s[i:i+n]:
            count += 1
    return count
```

Question 2F

[2 marks]

Exact answer will depend on the actual code written. If code is incomplete, or has glaring errors, e.g. infinite loop, obviously different algorithm, this part will be awarded zero marks.

Time: Let $k = \texttt{len(s)}$
$O(k^2)$ due to slicing the length of s for every recursive call.
$O(kn)$ for iterative code since it loops k times and each loop slices string of length k. We will also accept $O(k)$ by assuming n is fixed.

Space:
$O(k^2)$ for recursive due to new string created at every recursion
$O(n)$ for new string created every loop. We will also accept $O(1)$ by assuming n is fixed.

Question 3A

[4 marks]

*optional <PRE>:

<T1>:
```
lambda a, b: b + a
```

<T2>:
```
lambda x: 1 if s[x] == s[x+1] else 0
```
This line in the crux of the solution. 0 marks if this line is completely incorrect.

<T3>:
```
len(s)-2
```
-1 mark for each out-of-bound reference

Question 3B

[6 marks]

```
def assert_single(f):  # fill in the correct parameters
    return lambda x: None if num_pairs(f(x)) else f(x)
```

Purpose of this question is to test understanding of higher order function.
0-1 marks if solution does not return a function.
-2 marks for each mistake.

Question 4A

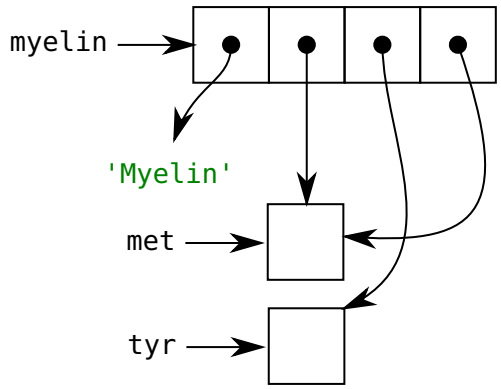
[4 marks]

```
def equal_chains(c1, c2):
    return map(amino_name, c1) == map(amino_name, c2)
# using map defined in the appendix

def equal_chains(c1, c2):
    t1, t2 = (), ()
    for a in c1:
        t1 = t1 + (amino_name(c1),)
    for a in c2:
        t2 = t2 + (amino_name(c2),)
    return t1 == t2

def equal_chains(c1, c2):
    i = 0
    while i < min(len(c1), len(c2)):
        if amino_name(c1) != amino_name(c2):
            return False
    return len(c1) == len(c2)
```

Question 4B

[2 marks]

One possible implementation is this:



Other representations are also valid as long as it works with the implemented functions.

-1 mark if the two met acids do not point to the same tuple.

Question 4C

[6 marks]

```
def make_protein(name):
    return (name,)




def protein_name(protein):
    return protein[0]




def protein_seq(protein):
    return protein[1:]




def add_amino(amino, protein):
    return protein + (amino,)
```

Question 4D

[4 marks]

```
def equal_proteins(p1, p2):
    return protein_name(p1) == protein_name(p2) and \
           equal_chains(protein_seq(p1), protein_seq(p2))
```

-2 marks for breaking abstraction of protein. -1 mark if amino acids are not compared using `amino_name` (or using `equal_chains`)

Question 4E

[6 marks]

```
def make_gene(name, seq):
    return (name, seq)




def express_protein(name, start, length, gene):
    p = make_protein(name)
    for i in range(start, start+3*length, 3):
        p = add_amino(make_amino(gene[1][i:i+3]), p)
    return p
```

— END OF ANSWER SHEET —