CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test Solutions

7 March 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 **TWELVE (12) 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).

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 = 9
y = 5
def f(x):
    return g(x) + y
def g(x):
    return y + x
print(g(f(y)))
```

[5 marks]

B.

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

[5 marks]

C.

```
t = 'True'
f = 'False'
if f == False:
    t, f = f, t
elif f[-1] == t[-1]:
    t += f
if not f == t:
    f += t
else:
    f = 'ARGH'
print(t)
print(f)
```

[5 marks]

D.

```
n = 10
while n:
    if n > 0:
        n -= 2*n
    n += 1
    if n % 3 == 0:
        continue
    n += 1
    if n % 2 == 0:
        break
print(n)
```

[5 marks]

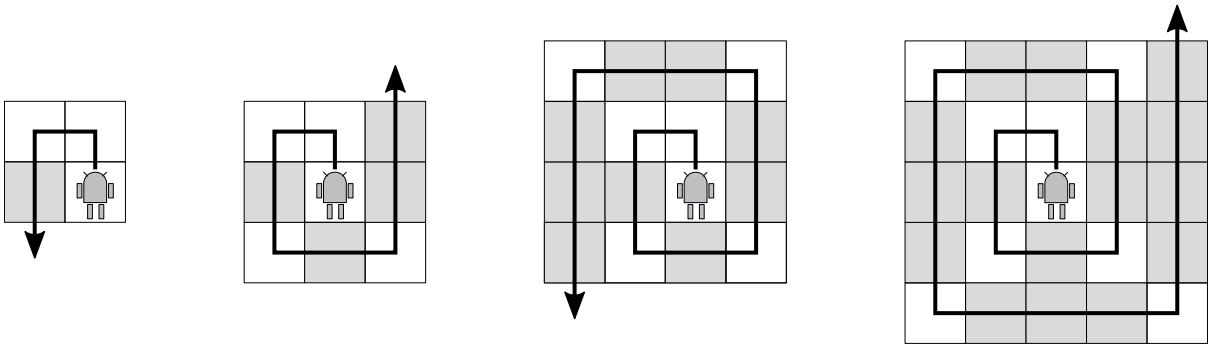
E.

```
def foo(y):
    return lambda x: x(x(y))
def bar(x):
    return lambda y: x(y)
print((bar)(bar)(foo)(2)(lambda x:x+1))
```

[5 marks]

Question 2: Spiral Path [18 marks]

A robot starts in the centre of a square grid and moves along a spiral path to cover every square in the grid and then exits the grid, as shown below for grids of size 2, 3, 4 and 5 respectively:



Basically, the robot moves forward and turns left whenever it encounters an unvisited grid square to its left.

The function `num_straights` takes as input the grid size, and returns the number of squares which the robot travels straight through without turning, i.e., the number of shaded squares in the above grids. While there is an obvious pattern of square numbers, your code should solve it using recursion/iteration.

A. Provide a recursive implementation of the function `num_straights`. [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_straights`. [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 sequence of instructions given to the robot to transverse a size 3 gird is:

```
('F', 'T', 'F', 'T', 'F', 'F', 'T', 'F', 'F', 'T', 'F', 'F', 'F')
```

where `'F'` means move one square forward, and `'T'` means turn 90 degrees to the left. Note that the last instruction causes the robot to exit the grid.

The function `gen_seq` takes as input the size of a grid, and returns a sequence of instructions for the robot to transverse the grid. The sequence of instructions is a tuple containing the strings `'F'` and `'T'` which represent forward and turn commands.

Provide a recursive or iterative implementation of the function `gen_seq`.
Hint: Recall `int` *can be multiplied with* `tuple`. [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 Spirals [8 marks]

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

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

The function `num_straights` in Question 2 can be defined in terms of `sum` as follows:

```
def num_straights(n):
    <PRE>
    return sum(<S1>, <S2>, <S3>, <S4>)
```

Please provide possible implementations for the terms `S1`, `S2`, `S3` and `S4`. You may optionally define other functions in `PRE` if needed. [4 marks]

B. 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 `gen_seq` in Question 2 can be defined in terms of `fold` as follows:

```
def gen_seq(n):
    <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. [4 marks]

Question 4: Pizzas [24 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.

Pizza is a traditional Italian dish consisting of a yeasted flatbread typically topped with tomato sauce and cheese and baked in an oven. It can also be topped with additional vegetables, meats, and condiments, and can be made without cheese. *Source: Wikipedia.*

For this question, you will be modelling pizzas and the toppings.

A topping is has two properties: a name (`str`) and the number of calories (`int`). It is supported by the following functions:

- `make_top(name, calories)` takes the topping name and number of calories as inputs, and returns a new topping.
- `name(top)` takes a topping as input, returns its name.
- `calories(top)` takes a topping as input, returns the number of calories of the topping.

A pizza is first created with a name (`str`) and size (`int`). Various toppings can then be added to the pizza. It is supported by the following functions:

- `make_pizza(name, size)` takes the pizza name and pizza size as inputs, and returns a new pizza.
- `add_topping(top, pizza)` takes a topping and a pizza as inputs, and returns a pizza with the topping added to it. Note that it is possible to add the same topping multiple times.

Consider the following sample run:

```
>>> pizza = make_pizza("Hawaiian", 10)  # makes a 10-inch Hawaiian pizza

>>> tomato = make_top("Tomato", 8)
>>> cheese = make_top("Cheese", 50)

>>> pizza = add_topping(tomato, pizza)
>>> pizza = add_topping(cheese, pizza)
>>> pizza = add_topping(cheese, pizza)  # adds extra cheese
```

A. Draw the **box-pointer diagram** of `pizza`, `tomato` and `cheese` at the end of the sample run above. Your diagram should be consistent with your implementations of *topping* and *pizza* functions in Part (B) and Part(C). [2 marks]

B. Provide an implementation for the *topping* functions `make_top`, `name` and `calories`. [4 marks]

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

C. Provide an implementation for the *pizza* functions `make_pizza` and `add_topping`. [4 marks]

D. Rachel is closely watching her diet and wishes to know how many calories are in a pizza. The total number of calories in a pizza is the total calories of all the toppings, as well as the calories of the base bread.

The calories of the base bread is dependent on the size of the pizza, according to the formula: $\pi(\frac{s}{2})^2 \times 5$ where s is the size of the pizza.

Implement the function `total_calories` which takes a pizza as input, and returns the total number of calories (`float`) of the entire pizza. *Note:* `math.pi` *returns the constant* π. [4 marks]

E. Rachel now wants to know what toppings are included in a pizza to help plan her diet.

Implement the function `get_toppings` which takes a pizza as input, and returns a tuple of the names of toppings of the pizza. Note that while the same topping can be added several times to a pizza, the names returned in the tuple should be unique, i.e. no repeats.

Reminder: You are not to use any Python data types which have not yet been taught in class, i.e. other than `tuple`. [4 marks]

F. Rachel does not want to eat an entire pizza. She slices a fraction of pizza to eat. Assume that a slice of pizza proportionally divides all the toppings and calories according to its fraction.

Suppose we want to also model a partially-eaten pizza as a pizza, i.e., using the same data representation. The function `eat(pizza, frac)` takes a pizza and a fraction from `0` to `1.0` as inputs, and returns a new pizza without the slice, i.e. the slice has been eaten.

```
>>> total_calories(pizza)
500.69908169872417

>>> pizza = eat(pizza, .2)   # eat 1/5 of the pizza
>>> total_calories(pizza)
400.55926535897936           # left 4/5 of the calories
```

Describe all issues, if any, of using your pizza data representation and suggest modifications to fix it. Finally, provide an implementation for `eat`.

Reminder: You should not break the abstraction of topping. [4 marks]

G. Oh no! Rachel accidentally topped a pizza with another pizza by calling:

```
>>> pizza = add_topping(pizza, pizza)
```

Based on your initial implementation of *topping* and *pizza*, describe what will happen when she tries to use the functions `get_toppings` and `total_calories` on this pizza. [2 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 —

CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test Solutions — Answer Sheet

7 March 2018 **Time allowed:** 1 hour 30 minutes

Student No:	**S**	**O**	**L**	**U**	**T**	**I**	**O**	**N**	**S**

Instructions (please read carefully):

1. Write down your **student number** on this answer sheet. DO NOT WRITE YOUR NAME!
2. This answer sheet comprises **TWELVE (12) 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).

GOOD LUCK!

For Examiner's Use Only

Question	Marks	Remarks
Q1	/ 25	
Q2	/ 18	
Q3	/ 8	
Q4	/ 24	
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 = 9
y = 5
def f(x):
    return g(x) + y
def g(x):
    return y + x
print(g(f(y)))
```

`20`
Tests the understanding of scoping.

Question 1B

[5 marks]

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

`((1, 2, (1, 2)), (1, 2), 3, 4)` Tests understanding of tuple addition.

Question 1C

[5 marks]

```
t = 'True'
f = 'False'
if f == False:
    t, f = f, t
elif f[-1] == t[-1]:
    t += f
if not f == t:
    f += t
else:
    f = 'ARGH'
print(t)
print(f)
```

`'TrueFalse'` Tests distinction between strings and booleans, and string indexing.
~~`'FalseTrueFalse'` Tests understanding of~~ independence of multiple if-else statements.

Question 1D

[5 marks]

```
n = 10
while n:
    if n > 0:
        n -= 2*n
    n += 1
    if n % 3 == 0:
        continue
    n += 1
    if n % 2 == 0:
        break
print(n)
```

`-4`

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

Question 1E

[5 marks]

```
def foo(y):
    return lambda x: x(x(y))
def bar(x):
    return lambda y: x(y)
print((bar)(bar)(foo)(2)(lambda x:x+1))
```

`4`

Test knowledge of evaluation of lambda expressions

Question 2A

[4 marks]

```
def num_straights(n):
    if n == 1:
        return 0          # no shaded squares on grid size 1
    else:
        return num_straights(n-1) + (n-1) + (n-2)  # 1st side + 2nd side
```

Question 2B

[2 marks]

Time: $O(n)$, there is a total of n recursive calls.

Space:$O(n)$, there is a total of n recursive calls, and each call will take up space on the stack.

Question 2C

[4 marks]

```
def num_straights(n):
    num = 0
    for i in range(2, n+1):
        num += (n-1) + (n-2)
    return num
```

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 gen_seq(n): # recursive
    if n == 1:
        return ('F',)
    else:
        side = ('T',) + (n-1)*('F',)
        return gen_seq(n-1) + side + side + ('F',)

def gen_seq(n): # iterative
    seq = ('F',)
    for i in range(2, n+1):
        side = ('T',) + (n-1)*('F',)
        seq += side + side + ('F',)
    return seq
```

Question 2F

[2 marks]

Time: $O(n^3)$. In every function call (recursion) or loop (iteration), a new tuple of the length of the path of each "layer" of the spiral is created. Since the length of the spirals is n^2, and there are n function calls or the loop runs n times, so the total time is $n^2 \times n = O(n^3)$.

In other words it is the sum of squares: $1^2 + 2^2 + 3^2 + \ldots + n^2 = O(n^3)$

Space: $O(n^2)$. End of the day, a new tuple of size n^2 is created and returned.

Question 3A

[4 marks]

*optional <PRE>:

<S1>:
```
lambda x: x-1 + x-2
```

<S2>:
```
2
```

<S3>:
```
lambda x: x+1
```

<S4>:
```
n
```

Question 3B

[4 marks]

*optional <PRE>:

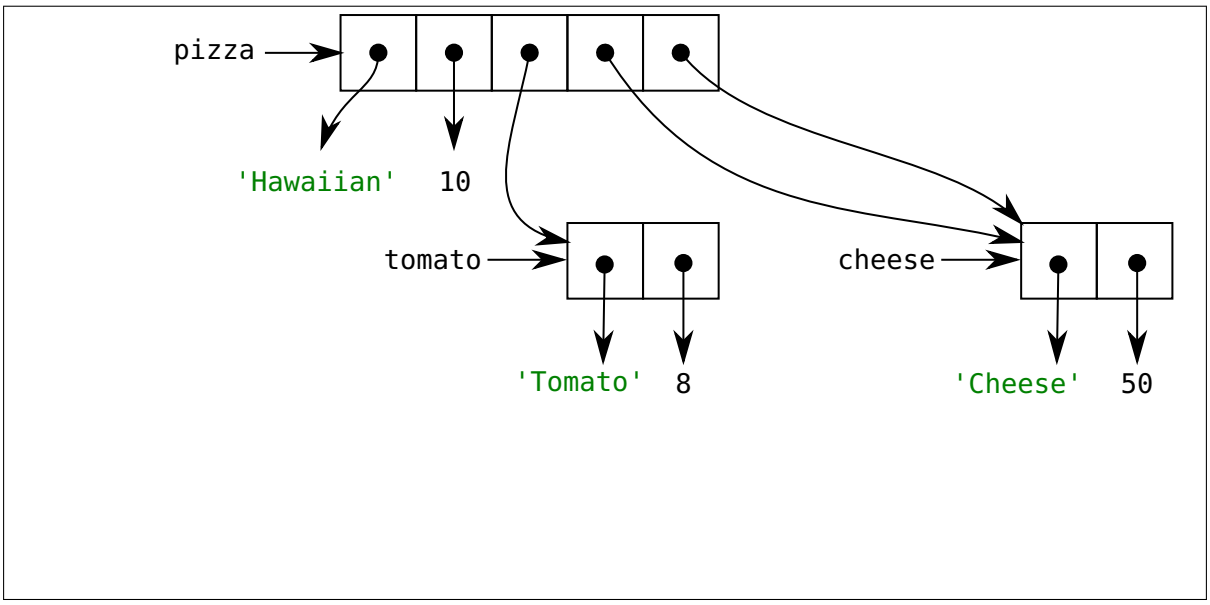
<T1>:
```
lambda a, b: b + a
```

<T2>:
```
lambda x: 2*(('T',) + x*('F',)) + ('F',) if x else ('F',)
```

<T3>:
```
n-1
```

Question 4A

[2 marks]



Question 4B

[4 marks]

```
def make_top(name, calories):
    return (name, calories)



def name(top):
    return top[0]



def calories(top):
    return top[1]
```

Question 4C

[4 marks]

```
def make_pizza(name, size):
    return (name, size)









def add_topping(top, pizza):
    return pizza + (top,)
```

Question 4D

[4 marks]

```
import math

def total_calories(pizza):
    base = math.pi * (pizza[1]/2)**2 * 5
    return base + sum(map(lambda t: calories(t), pizza[2:]))
```

Question 4E

[4 marks]

```
def get_toppings(pizza):
    top = ()
    for t in pizza[2:]:
        if name(t) not in top:
            top += (name(t),)
    return top
```

Question 4F

[4 marks]

The issue is that `make_top` can only be created with an integer calorie, so we cannot divide the calories of a topping without losing precision.

One way of solving is to include a fraction in the pizza, and `make_pizza` will create a pizza with a fraction of 1. I will insert the fraction as the first element and when calculating the calories, multiply the result with this fraction.

```
def eat(pizza, frac):
    return (pizza[0]*(1-frac),) + pizza[1:]
```

Question 4G

[2 marks]

While the structure of a pizza is fundamentally different from that of a topping, the first element of both pizza and topping are their names, so `get_topping` will simply return the pizza name `'Hawaiian'` as one of the toppings.

The second element of a pizza is the size. So `total_calories` will wrongly compute the number of calories, assuming that the size of the pizza is the number of calories of it as a topping.

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.

— END OF ANSWER SHEET —