CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

4 October 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 **EIGHTEEN (18) 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	/ 25	
Q2	/ 18	
Q3	/ 12	
Q4	/ 20	
Total	/ 75	

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, 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. [5 marks]

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

B. [5 marks]

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

C. [5 marks]

```
s = "bananab"
while s[0] == s[-1]:
    mid = len(s)//2
    s = s[1:mid]+s[0]+s[mid:-1]
print(s)
```

D. [5 marks]

```
s = 0
for i in range(10):
    if (i % 2 == 0):
        i = i+1
    if (i % 3 == 0):
        i = i-1
    else:
        i = 0
    s += i
print(s)
```

E. [5 marks]

```
def f(x):
    return lambda y: x(y+1)
def g(x):
    return f(g)(x+1)
print(g(1))
```

Question 2: Loan [18 marks]

It requires a huge amount of Galleons (ɢ) to conjure up an infinitely repeating gosper curve. Grandmaster Ben will need to take a loan to obtain all the gosper curves for the class. To avoid compounding interest, he decides to make a fixed repayment every month to pay off the loan. This is known as an amortized loan.

One can easily compute the loan using derived formulas, but the idea to compute it is simple: every month, an interest is charged upon the outstanding loan amount (known as the principal) and Grandmaster Ben makes a payment to pay off the interest and some of the principal. This continues until the entire loan is paid off.

For example, suppose Grandmaster Ben takes a loan amount of ɢ1,000,000 with a **monthly** interest rate of 0.5% and he wishes to make a monthly payment of ɢ6,000.

At month 1, the interest of the loan would be $ɢ1,000,000 \times 0.5\% = ɢ5,000$. Ben pays ɢ6,000 so the remainder of the principal is $ɢ1,000,000 + ɢ5,000 - ɢ6,000 = ɢ999,000$.

In month 2, the interest would be $0.5\% \times ɢ999,000 = ɢ4,995$, and the principal is reduced to $ɢ999,000 + ɢ4,995 - ɢ6,000 = ɢ997,995$.

We can summarize it a simple table:

Month	Principal	Interest	Payment	Remaining
1	1,000,000.000	5,000.000	6,000	999,000.000
2	999,000.000	4,995.000	6,000	997,995.000
3	997,995.000	4,989.975	6,000	996,984.975
4	996,984.975	4,984.925	6,000	995,969.900
...	...	...	...	...
359	7,440.368	37.202	6,000	1,477.570
360	1,477.570	7.388	1,485	0.000

As you can tell, it will take a long time, 30 years to be exact, for Ben to pay off the loan.

With the help of a table, it is easy to determine what is the remaining amount of the loan on any given month. For example, in month 4, Ben would have about ɢ995,970 of loan left to repay.

The function `loan_left(prin, intr, pymt, n)` takes as inputs the loan principal (amount loaned), the monthly interest, the monthly payment and a month number. It returns the amount of loan left to be paid after month n. It should return 0 when the loan has been completely repaid.

Examples:

```
>>> loan_left(1000000, 0.005, 6000, 1)
999000

>>> loan_left(1000000, 0.005, 6000, 359)
1477.570
```

```
>>> loan_left(1000000, 0.005, 6000, 360)
0
```

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

```
def loan_left(prin, intr, pymt, n):
```

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

Time:

Space:

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

```
def loan_left(prin, intr, pymt, n):
```

D. What is the order of growth in terms of time and space for the function you wrote in Part (C). Briefly explain your answer.

[2 marks]

Time:

Space:

Grandmaster Ben realises that his wand manufacturing business has a cyclical revenue pattern, peaking twice a year at January and July. It seems that demand for new wands are higher at the start of every semester.

So instead of repaying a fixed amount very month, Ben can afford to make higher payments in January and July, and lower payments for the other months of the year. For example, he can pay ₲16,000 in January and July, and ₲4,000 for the other months.

We can modify the `loan_left` function to `loan_left_cycle`, which take in payments as a tuple of monthly amounts. The monthly payment will repeatedly cycle through the tuple, restarting from the beginning when the end is reached, until the loan is paid off.

Example:

```
>>> cycle = (16000, 4000, 4000, 4000, 4000, 4000)
>>> loan_left_cycle(1000000, 0.005, cycle, 1)
989000
```

E. Provide an implementation of the function `loan_left_cycle`. [4 marks]

```
def loan_left_cycle(prin, intr, pymts, n):
```

F. What is the order of growth in terms of time and space for the function you wrote in Part (E). Briefly explain your answer. [2 marks]

Time:

Space:

Question 3: Higher-Order Loan [12 marks]

A. Consider this higher-order function `tail`

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

The function `loan_left` in Question 2 can be defined in terms of the higher-order function `tail` as follows:

```
def loan_left(prin, intr, pymt, n):
    <PRE>
    return tail(<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]

*optional `<PRE>`:

`<T1>`:

`<T2>`:

`<T3>`:

B. For some loans, the interest rate is variable and not fixed. That is, the monthly interest can be dependent on the length of the loan. For example, the interest might increase every year, or interest can be compounded only every 3 months.

We can modify the `loan_left` function to `var_loan_left`, such that it takes in a *function* to compute the interest for each month instead of a fixed rate.

Example:

```
>>> intf = increase_every_n(0.005, 12)  # Increase interest every 12 months
>>> var_loan_left(1000000, intf, 5000, 1)  # No interest
995000.0
>>> var_loan_left(1000000, intf, 5000, 2)
990000.0
>>> var_loan_left(1000000, intf, 5000, 12)
940000.0
>>> var_loan_left(1000000, intf, 5000, 13)  # 0.005% interest
939700.0
>>> var_loan_left(1000000, intf, 5000, 14)  # 0.005% interest
939398.5
```

The function `increase_every_n(rate, n)` returns an interest function that is interest free for the first n months, but increases the interest by $rate$ every n months. In the example above, the interest would be 0.5% for months 13–24, 1.0% for months 25–36, etc.

Another function `every_n(intf, n)` takes as input an interest function and an integer n, and returns an interest function that charges the interest according to $intf$ only every n month. For example, if $n = 3$, the interest function $intf$ would be charged on month 1, month 4, month 7, etc.

Example:

```
>>> f1 = increase_every_n(0.01, 4)  # increase by 1% every 4 months
>>> f2 = every_n(f1, 2)  # charge interest every 2 months
>>> for i in range(1, 12):
...     print (i, var_loan_left(1000, f2, 100, i))
1 900.0     # 0% interest
2 800.0
3 700.0     # 0% interest
4 600.0
5 506.0     # 1% interest
6 406.0
7 310.06    # 1% interest
8 210.06
9 114.2612  # 2% interest
10 14.2612
11 0        # 2% interest
```

Decide how the interest functions can be used in `var_loan_left` and provide an implementation for the functions `increase_every_n`, `every_n` and `var_loan_left`.

[8 marks]

```
def increase_every_n(rate, n):




def every_n(intf, n):




def var_loan_left(prin, intf, pymt, n):
```

Question 4: Shipping Containers [20 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.

A shipping container is a container with strength suitable to withstand shipment, storage, and handling. In our model, items can be placed in a container, with each item having a specified weight.

The following functions support the container data type:

- `make_empty_container(weight)` takes as input a weight (which is an integer), and returns an empty container of the given weight.
- `get_items(container)` takes as input a container, and returns a tuple of items that is in the container.
- `get_weight(container)` takes as input a container, and returns the total weight of the container, which is the sum of the weight of the container and the weight of all the items in it.
- `add_item(item, weight, container)` takes as input an item (which is a string) and a weight (which is an integer) and a container, and returns a container with the item added to it.
- `remove_item(item, container)` takes as input an item (which is a string) and a container, and returns a container with the specified item removed.

Sample run:

```
>>> container = make_empty_container(2)  # empty container weighs 2 tons

>>> container = add_item("bananas", 10, container)  # add 10 tons of bananas
>>> container = add_item("oranges", 5, container)   # add 5 tons of oranges
>>> container = add_item("bananas", 3, container)   # add 3 tons of bananas

>>> get_items(container)
('bananas', 'oranges', 'bananas')  # order does not matter

>>> get_weight(container)
20  # 2 + 10 + 5 + 3 = 20

>>> container = remove_item("bananas", container)
>>> get_weight(container)
7  # 2 + 5 = 7

>>> container = remove_item("apples", container)
>>> get_weight(container)
7  # nothing changed
```

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

B. Provide an implementation for the functions `make_empty_container`, `get_items` and `get_weight`. [4 marks]

```
def make_empty_container(weight):




def get_items(container):









def get_weight(container):









```

C. Provide an implementation for the functions `add_item` and `remove_item`.

[4 marks]

```
def add_item(item, weight, container):




def remove_item(item, container):
```

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

Containers can be loaded onto ships. Our model of ships is supported by the following functions:

- `make_ship(capacity)` takes as input a capacity (which is an integer), and returns an empty ship that is capable of carrying the specified capacity of weight.
- `load_container(container, ship)` takes as input a container and a ship, and adds the container to the ship. If the total weight of all the containers in the ship exceeds its capacity, then the ship sinks and `None` is returned. Otherwise, a ship with the added container is returned.
- `contains(item, ship)` takes as input an item (which is a string) and a ship, and returns `True` if the item is in any of the containers on the ship. Otherwise, it returns `False`.

D. Decide on an implementation of ship and provide an implementation for each of the functions `make_ship`, `load_container` and `contains`. [6 marks]

```
def make_ship(capacity):




def load_container(container, ship):
```

```
def contains(item, ship):
```

E. Your friend Gerrie thinks we should lock the containers to prevent thief and piracy of the goods. Her idea is to hide the container in a closure which can only be accessed by the secret code. Below is her implementation:

```
def lock(container, code):
    return lambda x: container if x == code else None

def unlock(container, code):
    return container(code)
```

Gerrie then tries to load containers locked with her functions onto the ships using the function `load_container` as follows but got an error:

```
>>> maersk = make_ship(100)
>>> get_weight(container)  # container of apples from previous part
7

>>> container = lock(container, "passcode")  # locks container using passcode
>>> load_container(container, maersk)
Error: xxx
```

What is the cause of the error and is there any way to resolve it without breaking the abstraction of container? [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

— E N D O F P A P E R —