CS1010S — Programming Methodology
School of Computing
National University of Singapore

Solutions for Re-Mid-Term Quiz

19 October 2013 **Time allowed:** 1 hour 45 minutes

Matriculation No: | | | | | | | | | |

Instructions (please read carefully):

1. Write down your matriculation number on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises **FOUR (4) questions** and **SIXTEEN (16) pages**. The time allowed for solving this quiz is **1 hour 45 minutes**.
4. The maximum score of this quiz is **100 marks**. The weight of each question is given in square brackets beside the question number. Note however that the final grade will be capped at **60 marks** because this is a re-exam.
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 back-sides of the sheets and 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 quiz.
9. You are allowed to use pencils, ball-pens or fountain pens, as you like (no red colour, please).

GOOD LUCK!

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Total		

Question 1: Python Expressions [24 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. If the interpreter produces an error message, or enters an infinite loop, explain why.

A.
```
a = "mid"
b = "term"
c = "quiz"
if len(b) < len(a):
    print(len(c))
else:
    print(a[-1] + c[-2] + b[1])
```

[4 marks]

```
die
```

This question tests if the student understands conditional `if` and string indexing.

B.
```
good = ("luck", "day", "will", "win")
def lucky(good):
    if "friday" < good[3]:
        print("awesome")
    else:
        print("broke")
lucky("money")
```

[4 marks]

```
broke
```

This question tests if the student understands function call and string comparisons

C.
```
total = 5
bag = ("chocolates", "jellybeans", "candies", "gummies")
for sweet in bag:
    if len(sweet) % 2:
        total += len(sweet)
    else:
        total -= len(sweet)
print(total)
```

[4 marks]

```
-1
```

This question tests if the student understands looping through a collection and the modulus operation

D.
```
x = 26
y = 5
while x % y <= 3:
    x = x - 3
print(x)
```

[4 marks]

```
14
```

This question tests if the student understands the `while` loop.

E.
```
def foo(x, y):
    return x + y
def bar(x, y):
    return y + (x,)
print(bar(21, foo(bar(foo(2, 3), (1,)), (9, 15))))
```
[4 marks]

```
(1, 5, 9, 15, 21)
```

This question tests if the student understands how to perform tuple concatenation as well as how to evaluate complex function calls.

F.
```
print(accumulate(lambda x, y: x * y, 1, enumerate_interval(1, 5)))
```
[4 marks]

```
120
```

This question tests if the student understands how to evaluate higher order functions.

Question 2: Magic Bags!!! [24 marks]

A Python Magic Bag is a bag that has infinite capacity. Suppose you are provided with a function `is_magic_bag` which takes an object and returns `True` if it is a Python Magic Bag.

A. Write a recursive function `count_bags(objects)` that takes a tuple `objects` and returns the number of objects that are Magic Bags. If you use the splicing operator (`[a:b]`), keep in mind that it is costly and will take time equal to the length of the spliced segment because of copying. [6 marks]

```
def count_bags(objects):
    if objects == ():
        return 0
    elif is_magic_bag(objects[0]):
        return 1 + count_bags(objects[1:])
    else:
        return count_bags(objects[1:])
```

B. What is the order of growth in terms of time and space for the function you wrote Part(a) in terms of n, where n is the total number of items in contents? [4 marks]

Time: $O(n^2)$

Space: $O(n)$

C. Write an iterative version of `count_bags`. [6 marks]

```
def count_bags(objects):
    total = 0
    for item in objects:
        if is_magic_bag(item):
            total = total + 1
    return total
```

D. What is the order of growth in terms of time and space for the function you wrote Part(c) in terms of n, where n is the total number of items in contents? [2 marks]

Time: $O(n)$

Space: $O(1)$

E. Python Mages love to abuse the Python Magic Bag by stuffing other Python Magic Bags into itself. You are provided with a function `get_contents(bag)` which takes a Python Magic Bag and returns a tuple of objects in the Magic Bag. Write a function `deep_count_bags(objects)` that takes in a tuple of objects and returns the total number of Magic Bags (some potentially nested) in this tuple of objects. [6 marks]

```
def deep_count_bags(objects):
    total = 0
    for item in objects:
       if is_magic_bag(item):
          total = total + 1 + deep_count_bags(get_contents(item))
    return total
```

Question 3: Higher Order Functions [22 marks]

Consider the following higher-order function that we call `mash`:

```
def mash(seq, checker, combiner, fixer, base):
    if seq == ():
        return base
    elif checker(seq[0]):
        return combiner(seq[0], mash(seq[1:], checker, combiner, fixer, base))
    else:
        return mash(fixer(seq[0]) + seq[1:], checker, combiner, fixer, base)
```

A. Suppose the function `sum_integers(n)` computes the sum of integers from 1 to *n* (inclusive) and `sum_integers(n)` is defined as follows:

```
def sum_integers(n):
    return mash(<T1>,
                <T2>,
                <T3>,
                <T4>,
                <T5>)
```

Please provide possible implementations for the terms `T1`, `T2`, `T3`, `T4`, and `T5`. [11 marks]

T1:	`enumerate_interval(1, n)`
T2:	`lambda x:  True`
T3:	`lambda x, y:  x + y`
T4:	`lambda x:  x`
T5:	`0`

B. Suppose the function `add_all(tpl)` calculates the sum of all the integers (of type int) within a tuple `tpl`. `tpl` may contain integers or other tuples. `add_all(tpl)` is defined as follows:

```
def add_all(tpl):
    return mash(<T6>,
                <T7>,
                <T8>,
                <T9>,
                <T10>)
```

Please provide possible implementations for the terms `T6`, `T7`, `T8`, `T9`, and `T10`. [11 marks]

`T6:`

```
tpl
```

`T7:`

```
lambda x:  type(x) == int
```

`T8:`

```
lambda x, y:  x + y
```

`T9:`

```
lambda x:  x
```

`T10:`

```
0
```

Question 4: Bluble Mania [30 marks]

Blubles colourful creatures that live in an intricate society. When a bluble meets another bluble, the larger bluble will eat the smaller bluble and the size of the larger bluble will increase accordingly, i.e. the size of the bigger bluble will become the sum of its original size and that of all the eaten blubles. However, a bluble may not contain more than one bluble of each colour. In the event where a larger bluble needs to eat a smaller bluble with a colour that it has already eaten, it must blurp out the originally eaten bluble of the same colour first before eating the next bluble. You can assume that the colour will be represented with a string.

Your implementations must satisfy the following sample execution:

```
>>> tulip = make_bluble(9, "white")
>>> get_size(tulip)
9
>>> get_colour(tulip)
"white"
>>> get_eaten(tulip)
()
```

A. Describe how you would represent a bluble object and write a function `make_bluble(size, colour)` that takes 2 arguments, the original size of the bluble and the colour of the bluble, and returns a new bluble object that has not eaten any other blubles. **Hint:** Please read the entire problem before you do this question. You will need to keep track of what a bluble has eaten. [4 marks]

```
def make_bluble_with_eatee(size, color, eatee):
    return ('bluble', size, color, eatee)

def make_bluble(size, color):
    return make_bluble_with_eatee(size, color, ())
```

B. Write accessors `get_size`, `get_colour` and `get_eaten` for the bluble object you defined in Part (a) above, that return the size of a bluble, its colour and also a tuple of eaten blubles, respectively. Keep in mind that the size of a bluble is the sum of its original size and also the sizes of all the eaten blubles. [6 marks]

```
def get_size(bluble):
    return bluble[1]

def get_colour(bluble):
    return bluble[2]

def get_eaten(bluble):
    return bluble[3]
```

C. Write a function `is_bluble(a)` which returns `True` if `a` is a bluble, or `False` otherwise. [3 marks]

```
def is_bluble(a):
    return type(a) == tuple and len(a) == 4 and a[0] == 'bluble'
```

D. Write a function `eat(eater, eatee)`, which takes as arguments a bluble which is to be the eater, and another bluble which is to be eaten ("eatee") and returns a **new** bluble object which represents the eater after it has eaten the eatee. You may assume that the eatee will be of a color that the eater has yet to eat. If the eatee is too big to be eaten, `eat` will just return the eater object. [5 marks]

```
def eat(eater, eatee):
    eater_size = get_size(eater)
    eatee_size = get_size(eatee)
    if eater_size > eatee_size:
        return make_bluble_with_eatee(eater_size + eatee_size,
                                      get_colour(eater),
                                      get_eaten(eater) + (eatee,))
    else:
        return eater
```

E. Write a function `blurp(blurper, colour)`, which takes as arguments blurper, a bluble object, and the colour of the bluble that it needs to blurp out and returns a **new** bluble object which is the blurper after it has blurped out the required coloured bluble. The size of resultant bluble should be reduced appropriately. If blurper has not eaten a bluble of the specified colour, `blurp` will return the original blurper object. [6 marks]

Your implementations must satisfy the following sample execution:

```
>>> daisy = make_bluble(3, "white")
>>> tulip = make_bluble(9, "red")
>>> tulip = eat(tulip, daisy)
>>> tulip = blurp(tulip, "red")
>>> get_size(tulip)
12
>>> get_eaten(tulip)
(daisy, )
>>> tulip = blurp(tulip, "white")
>>> get_eaten(tulip)
()
>>> get_size(tulip)
9
```

```
def blurp(blurper, colour):
    eatees = get_eaten(blurper)
    for index in range(len(eatees)):
        e = eatees[index]
        if get_colour(e) == colour:
            new_size = get_size(blurper) - get_size(e)
            new_estees = eatees[:index] + eatees[index+1:]
            return make_bluble_with_eatee(new_size,
                                          get_colour(blurper),
                                          new_estees)
    return blurper
```

F. Write a function `encounter(bluble1, bluble2)`, which takes two blubles and returns a **new** bluble object representing the larger bluble after the encounter. If the two blubles are of the same size (which means that neither ean eat each other), `encounter` can return either bluble. Remember that a bluble may not eat another bluble of a color that it has already eaten without first blurping. [6 marks]

```
def encounter(bluble1, bluble2):
    size_bluble1 = get_size(bluble1)
    size_bluble2 = get_size(bluble2)
    if size_bluble1 == size_bluble2:
       return bluble1
    elif size_bluble1 > size_bluble2:
       eater, eatee = bluble1, bluble2
    else:
       eater, eatee = bluble2, bluble1
    eater = blurp(eater, get_colour(eatee))
    return eat(eater, eatee)
```

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 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 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

— END OF PAPER —