CS1010S — Programming Methodology
School of Computing
National University of Singapore

Re-Mid-Term Quiz

17 October 2014 **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 **NINETEEN (19) 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 color, please).

GOOD LUCK!

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Total		

Question 1: Python Expressions [30 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.

```
x = 0
if x <= 0:
    print("Yeah")
elif x >= 0:
    print("No!!")
if x == 0:
    print("No idea!")
```

[5 marks]

```
Yeah
No idea!!
```

This question tests if the student understands the `if` statement, and is careful to notice the scope of the if-block.

B.

```
def t(a,b):
    return (b,)+(a)
print(t(t((1,2),2),3))
```

[5 marks]

```
(3, 2, 1, 2)
```

This question tests if the student understanding of tuples.

C.

```
x = 5
y = 7
def g(y):
    return x + y
def f(x):
    y = 4
    return g(x-y)
print(f(y))
```

[5 marks]

```
8
```

This question tests if the student understands the scope of the variables in nested function calls.

D.

```
sum = 60
for i in range(1,10):
    if i % 3:
        continue
    if sum < 10:
        break
    sum //= i
print(sum)
```

[5 marks]

```
3
```

This question tests the understanding of `for`-loops, and the `continue` and `break` statements in loops.

E.

```
a = "Happy"
b = "Birthday!"
while (a):
    b += a[0]
    a = a[1:]
print(b)
```

[5 marks]

```
Birthday!Happy
```

This question tests the understanding of slicing and maniupling strings as sequences.

F.

```
once = lambda f: lambda x: f(x)
twice = lambda f: lambda x: f(f(x))
thrice = lambda f: lambda x: f(f(f(x)))
print(once(twice(thrice(lambda x: 2*x)))(2))
```

[5 marks]

128 This question tests the student's proficiency in evaluating multiple `lambda` statements.

Question 2: Palindromic numbers [22 marks]

A palindromic number is a number that remains the same when its digits are reversed. For example, 27372 is a palindromic number, i.e. it is "symmetrical".

The first 20 palindromic numbers (starting from 1) are:

1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, 99, 101, 111

A. Write a **recursive** function `is_palindromic(n)` that returns `True` if `n` is a palindromic number and return `False` otherwise.
(*Hint: It might be helpful to use strings.*) [5 marks]

```
def is_palindromic(n):
    n = str(n)
    def reverse(str):
        if str=="":
            return ""
        else:
            return reverse(str[1:]) + str[0]
    return n == reverse(n)
```

B. What is the order of growth in terms of time and space for the function you wrote in Part (A) in terms of n? **Explain** your answer, state your assumptions (if any).
(*Hint: String slicing takes $O(k)$ space and time where k is the length of the returned string. Adding two strings takes $O(k_1 + k_2)$ space and time where k_1 and k_2 are the lengths of the two strings.*) [4 marks]

Time: $O(\log^2 n)$ due to string slicing

Space: $O(\log^2 n)$ due to n recursive calls, and each call creating a new string of $n-1$ length by slicing.

C. Write an **iterative** function `is_palindromic(n)` that returns `True` if `n` is a palindromic number and return `False` otherwise. [5 marks]

```
def is_palindromic_iter(n):
    n = str(n)
    n1 = ""
    for char in n:
        n1 = char + n1
    return n1 == n
```

D. What is the order of growth in terms of time and space for the function you wrote in Part (C) in terms of n? [2 marks]

Time: $O(\log n)$ cos the loop iterates through the number of digits of n.

Space: $O(\log n)$ due to new string n1 being created.

Recall that the first 20 palindromic numbers (starting from 1) are:

1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, 99, 101, 111

where 1 is the 1^{st} palindromic number, and 111 is the 20^{th} palindromic number.

E. Write a function `nth_palindrome(n)` that returns the $\texttt{n}^{th}$ palindromic number. [6 marks]

```
def nth_palindrome(n):
    counter = 0
    count = 0
    while count < n:
        counter+=1
        if is_palindromic(counter):
            count+=1
    return counter
```

Question 3: Higher-Order Functions [22 marks]

Consider the following sum $S(i,j,k)$ where i,j,k are integers and $k > 0$:

$$S(i,j,k) = i + (i+k) + (i+2k) + \cdots (i+nk), \text{ where } i+nk \leq j < i+(n+1)k$$

A. Suppose the function `compute_s(i,j,k)` will return the sum $S(i,j,k)$. We can express $S(i,j,k)$ in terms of `sum` (see Appendix) as follows:

```
def compute_s(i,j,k):
    return sum( <T1>, <T2>, <T3>, <T4> )
```

Please provide possible implementations for the terms `T1`, `T2`, `T3`, and `T4`. [5 marks]

`T1`: [1 mark]

```
lambda x:  x
```

`T2`: [1 mark]

```
i
```

`T3`: [2 marks]

```
lambda x:  x+k
```

`T4`: [1 mark]

```
j
```

B. We can also express $S(i, j, k)$ in terms of `fold` (see Appendix) as follows:

```
def compute_s(i,j,k):
    return fold( <T5>, <T6>, <T7> )
```

Please provide possible implementations for the terms `T5`, `T6`, and `T7`. [5 marks]

`T5`:
[1 mark]

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

`T6`:
[2 marks]

```
lambda n:  i + n*k
```

`T7`:
[2 marks]

```
(j // k) - 1
```

C. Besides `sum` and `fold`, `map` and `reduce` (see Appendix) are higher-order functions that are commonly used in practice.

It turns out that we can also express $S(i, j, k)$ in terms of `map` and `reduce`:

```
def compute_s(i,j,k):
    return reduce( <T8>, <T9> , map( <T10> , tuple(range( <T11> ))))
```

Please provide possible implementations for the terms `T8`, `T9`, `T10`, and `T11`. [6 marks]

`T8`: [1 mark]

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

`T9`: [1 mark]

```
0
```

`T10`: [2 marks]

```
lambda n:  i + n*k
```

`T11`: [2 marks]

```
j//k
```

D. Indeed, the functions `map` and `reduce` are very handy. For example, we can use them to compute the factorial of a non-negative number n:

```
def factorial(n):
    return reduce( T<15> , <T16> , map( <T17>, tuple(range( <T18> ))))
```

Recall that $factorial(n) = 1 \times 2 \times 3 \times \ldots \times n$.

Please provide possible implementations for the terms `T15`, `T16`, `T17`, and `T18`. [6 marks]

`T15`: [2 marks]

```
lambda x,y:  x * y
```

`T16`: [1 mark]

```
1
```

`T17`: [1 mark]

```
lambda x:  x
```

`T18`: [2 marks]

```
1, n+1
```

Question 4: Genealogy [26 marks]

Warning: Please read the question description clearly before you attempt this problem!!

Genealogy is the study of families and the tracing of their lineages and history. For this question, we will specialize in studying the lineage of fathers, which is called a patriline. In other words, it is a family line of fathers. For example, this is a 4 generation patriline of the Lee family:

Philip Lee
↓
Andrew Lee
↓
Ben Lee
↓
Matthew Lee

You will create a family patriline with the function `make_patriline(family, ancestor)`, where `family` is the family name and `ancestor` is the first name of the first ancestor of this line. Thereafter, sons can be added to the patriline with the function `begat(patriline, name)`, where `patriline` is the patriline and `name` is the name of the next son in line. For example, the Lee's patriline can be created by:

```
>>> lee = make_patriline("Lee", "Philip")
>>> lee2 = begat(lee, "Andrew")
>>> lee3 = begat(lee2, "Ben")
>>> lee4 = begat(lee3, "Matthew")
```

The function `get_length(patriline)` returns the number of generations in the patriline, and the function `get_person(patriline, n)` is used to get the nth person of the patriline, with the first ancestor being $n = 1$.

Example:

```
>>> get_length(lee4)
4

>>> get_person(lee4, 2)
'Andrew'

>>> get_person(lee4, 5)
None
```

A. Describe how you will represent a patriline using a tuple. [3 marks]

A patriline is represented by a tuple, with the family name at index 0, and the first ancestor at index 1 and his son at index 2 and so on.

Note: These two points must be mentioned to obtain the full marks: 1) how the family name is stored, 2) the ordering of the patriline in the tuple.

B. Please provide a possible implementation for the constructor `make_patriline` and `begat`. [4 marks]

```
def make_patriline(family, ancestor):
    return (ancestor, family)

def begat(patriline, name):
    return patriline + (name,)
```

C. Please provide a possible implementation for the accessors functions `get_length` and `get_person`. [6 marks]

```
def get_length(patriline):
    return len(patriline) - 1
```

```
def get_person(patriline, n):
    if n <= get_length(patriline):
        return patriline[n]
```

D. Bob, a genealogist, would like to use Python for his work. In particular, he wants to examine the patrilines that we have created. One function that he needs in his work is to get the name of the father of a given person in the patriline.

For example:

```
>>> get_father(lee4, "Ben")
'Andrew'

>>> get_father(lee4, "Amos")
None

>>> get_father(lee4, "Philip")
None
```

Give a possible implementation of the function `get_father`, bearing in mind that Bob has no knowledge of your patriline state. You may assume that no two persons in the patriline have the same name. [3 marks]

```
def get_father(patriline, son):
    for i in range(2, get_length(patriline) + 1):
        if (get_person(patriline, i) == son):
           return get_person(patriline, i-1)
    return None
```

E. Now it is entirely possible for a father to have several sons, in which case a patriline might be split from a common ancestor. For example, consider the Tan's patriline:

```
>>> tan = make_patriline("Tan", "John")
>>> tan2 = begat(tan, "Samuel")
>>> tan3 = begat(tan2, "Elvis")
>>> tan4 = begat(tan2, "Elvin")
>>> tan5 = begat(tan4, "Brian")
```

where Samuel has two sons, Elvis and Elvin, thus forking the Tan patriline into two.

When studying patrilines, it is important to know where patrilines have forked. Help Bob write a function to find the lowest common ancestor of two patrilines, provided that they are of the same family and origin. You may assume that two patrilines with a common ancestry will both have the same first ancestor. [6 marks]

Example:

```
>>> lowest_common_ancestor(tan4, tan5)
'Samuel'

>>> lowest_common_ancestor(lee3, tan3)
None
```

```
def lowest_common_descendant(p1, p2):
    person = None
    i = 1
    while i <= get_length(p1) and i <= get_length(p2):
        if get_person(p1, i) != get_person(p2, i):
            break
        person = get_person(p1, i)
        i += 1
    return person
```

F. Oh no! In his research, Bob has discovered that "Tan" is a very common family name. It is entirely possible to have different patrilines of two different persons with the same name! For example:

```
>>> atan = make_patriline("Tan", "John")
>>> atan1 = begat(atan, "Mark")
```

Since there are two different persons named John Tan, the correct output should be:

```
>>> lowest_common_ancestor(tan3, atan1)
None
```

Does your implementation of `lowest_common_ancestor(p1, p2)` produce the correct result? If yes, please explain why? If no, please explain what can be done to change it.
Hint: The `is` *comparison operator on strings does not guarantee consistent results.* [4 marks]

No. Because the accessor functions only return strings, which is the same for two different persons of the same name. Using the `is` comparison different strings is not consistently False.
[**1 mark for recognising the limitations of string comparision**]

Thus, the solution is to store each person in the patriline as a tuple.Only then the `is` comparator will return false for two persons having the same name.
[**2 marks for suggesting an alternative like tuple**]

The abstraction, in particular, the `get_person` function also has to be redefined to return a person object and not just the name. Additional accessor functions need to be introduced to retrieve the name from the person object as well as a constructor to create a person.
[**1 mark for recognising that the abstraction has to be modified too**]

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

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

Scratch Paper

— E N D O F P A P E R —