NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

RE-MID-TERM TEST SOLUTION SKETCHES FOR

CS1010S Programming Methodology

19 March 2014 Time Allowed: 1 hr 40 min

INSTRUCTIONS TO CANDIDATES

1. This examination paper consists of **FOUR (4)** questions and comprises **ELEVEN (11)** printed pages. Answer **ALL** questions.
2. Read ALL questions before you start.
3. Write your answers in the space provided. If you need more space, write at the back of the sheet containing the question only. DO NOT put part of the answer to one problem on the back of the sheet containing another problem.
4. Please write LEGIBLY! No marks will be given for answers that cannot be deciphered.
5. This is an **OPEN-SHEET** quiz. You are allowed to bring one A4 sheet of notes (written on both sides).
6. Please fill in your **Name** and **Matriculation Number** below.

Name: _____

Matriculation Number: | | | | | | | | | |

For Examiner's Use Only		
	Marks	Max. Marks
Question 1		**30**
Question 2		**26**
Question 3		**24**
Question 4		**20**
TOTAL:		**100**

Problem 1 (30 marks)

Answer each part below independently and separately. In each part, one or more Python expressions are entered into the interpreter (Python shell). The shell prompt ">>>" is omitted on each line. Determine the output printed by the interpreter for the final expression entered. If the interpreter produces an error message, or enters an infinite loop, explain why.

a) [5 marks]

```
def test():
    for i in range(2):
        for j in range(1, len("abc")):
            print(str(i) + " and " + str(j))
    return True


test()
```

Ans:

```
0 and 1
0 and 2
1 and 1
1 and 2
True
```

b) [5 marks]

```
x = 5
y = 10

def bundle(y):
    print(x)
    return lambda y: y + 5

bundle(2)(5)
```

Ans:

```
5
10
```

c) [5 marks]

```
x = "star trek"
y = "adventures"
i = 4
j = 2

while j <= 2:
    if i > 0:
        print(y[0:i])
    else:
        print(x[0:j])
    i = i - 1
    j = j + 1
```

Ans:
adve

d) [5 marks]

```
def top(x):
    def two(x):
        return x + 2
    def three (x):
        return x + 3
    def zip (x):
        return x * 0
    return three(zip(two(x)))

top(10000)
```

Ans:
3

e) [5 marks]

```
def top(x):
    def two(y):
        def three(z):
            def zip(x):
                return 2*y + 3*z + 0*x
            return zip(z)
        return three(y)
    return two(x)

top(5)
```

Ans:
25

f) [5 marks]

```
a = (1, 2)
b = ('one', 'two', 'three')

def convert(a, b):
    t1 = type(a)
    t2 = type(b)
    if t1 == t2:
        print(a + b)
    else:
        print("Cannot print!")

convert(a, b)
```

Ans:
(1, 2, 'one', 'two', 'three')

Problem 2 (26 marks)

A positive integer `n` is a *prime number* if its smallest divisor is itself. The smallest prime number is 2.

Assume that you have a predicate `is_Prime(n)` that checks whether the positive integer `n` is a prime number (*Note*: you will be asked to define this predicate in Part (c) of this problem, but for Part (a) and Part (b) you can just assume that this function is already defined correctly).

a) Write an **iterative** Python function `prime_generator_i` that takes in as argument a positive integer `n`, and prints out all the prime numbers less than or equal to `n`, in REVERSED order. (Note: this function does not need to RETURN anything.) [6 marks]

```
def prime_generator_i(n):

    # prints all primes between n and 2

    for i in range(n, 1, -1):
        if is_Prime(i):
            print(i)
```

b) Write a **recursive** Python function `prime_generator_r` that takes in as argument a positive integer `n`, and prints out all the prime numbers less than or equal to `n`, in REVERSED order. (Note: this function does not need to RETURN anything.) [6 marks]

```
def prime_generator_r(n):

    # prints all primes between n and 2

    if n < 2:
        return
    elif is_Prime(n):
        print(n)
        prime_generator_r(n - 1)
```

Now define the predicate function `is_Prime(n)` as follows:

1. Define an internal function that finds the smallest divisor of a number `m`, by testing `m` for divisibility by successive integers starting with 2.
2. Check if the smallest divisor of `n` is itself.

Complete the function definition by providing the code segments **`T1`** and **`T2`** below.

```
def is_Prime(n):
    def smallest_divisor(m, guess):
        <T1>
    return <T2>
```

Assumptions:

i. You can make use of this fact: If *m* is not prime it must have a divisor less than or equal to $\sqrt{m}$. So you only need to test for integers between 1 and $\sqrt{m}$. In other words, when the number $guess^2$ (square of *guess*) is greater than *m*, the smallest divisor is *m*.

ii. You can also assume the availability of the functions `sqrt(x)` and `square(x)`, which will return $\sqrt{x}$ and x^2 respectively, for any integer *x*.

c) **`T1`** specifies the different conditions to successively check for the smallest divisor of a number `m`, starting with a number `guess` [8 marks]:

Grading policy:

There are many ways to calculate the smallest divisor. Three conditions must be satisfied for full credit: the terminating condition, checking if number m is divisible by another number (guess), and iteration or recursion through successive values of guess. The solution shown below is a recursive implementation; an iterative implementation is also acceptable

T1:

```
if square(guess) > m:
    return m
elif m % guess == 0:
    return guess
else:
    return smallest_divisor(m, guess + 1)
```

d) **`T2`** specifies the comparison check of whether the smallest divisor of `n` is equal to `n` itself [6 marks]:

T2

```
smallest_divisor(n, 2) == n
```

Problem 3 (24 marks) Magic Wands – Part 1

Harry, Ron, and Ermie are three good friends taking the same Introductory Wand Making course at the Happy Magic School. Being in the digital age, the first class assignment is to write a Python program (!) that simulates the creation, management, and usage of the wands in the magical world.

Harry has defined a data abstraction for representing a wand by implementing it as a list of two integers: `[n, m]`, where `n` is the number of **charges** (i.e., how many shots that the wand can be "fired" or "used"), and `m` is the number of **lives** (i.e., how many times the wand can be "recharged" or "topped-up with additional charges").

```
def make_wand(n, m):
    return [n, m]
```

a) Help Harry define the selectors of the wand data type as follows:

i. Define the selector `get_charges(w)`, which returns the number of charges left in a wand `w` [3 marks]

ii. Define the selector `get_lives(w)`, which returns the number of lives left in a wand `w` [3 marks]

```
def get_charges(w):
    return w[0]



def get_lives(w):
    return w[1]
```

b) Help Harry define the modifiers of the wand data type as follows:

i. Define the modifier `add_charges(w, c)`, which adds the number of charges `c` to a wand `w`. [3 marks]

ii. Define the modifier `add_lives(w, d)`, which adds the number of lives `d` to a wand `w`. [3 marks]

```
def add_charges(w, c):
    w[0] += c


def add_lives(w, d):
    w[1] += d
```

c) Now write a Python function `use_wand(w)` that simulates the usage of the wand `w`. You should use only the selectors and modifiers as defined in Part (a) and Part (b) in your new function to manipulate the wand. Executing the function will decrease the number of charges of the wand `w` by 1. If there are no charges left to use the wand, the function should print out: 'Cannot use!' The function will return the updated wand `w`. [5 marks]

```
def use_wand(w):

    n = get_charges(w)
    m = get_lives(w)
    if n > 0:
        add_charges(w, -1)
    else:
        print("Cannot use!")
    return w
```

d) Now write another Python function `recharge_wand(w,c)` that simulates the recharging of the wand `w` by `c` number of charges. You should use only the selectors and modifiers as defined in Part (a) and Part (b) in your new function to manipulate the wand. Executing the function will increase the number of charges of the wand `w` by `c`, and decrease the number of lives of the wand by 1. If there are no lives left to recharge the wand, however, the function should print out: 'Cannot recharge!' The function will return the updated wand `w`. [5 marks]

```
def recharge_wand(w,c):

    n = get_charges(w)
    m = get_lives(w)
    if m > 0:
         add_charges(w, c)
         add_lives(w, -1)
    else:
        print("Cannot recharge!")
    return w
```

e) Ron does not like to work with lists and wants to change the underlying representation of a wand to a tuple `(n, m)`, where again `n` is the number of charges and `m` is the number of lives left in the wand. "Why," Ron declares, "I just need to change the main implementation of `make_wand`. All the selectors and modifiers remain the same as a tuple is also a sequence type in Python. I also do not need to change the `use_wand` and the `recharge_wand` functions!"

In the space below, **briefly** explain why you agree OR do not agree with Ron. State any assumptions you make in your answer. [2 marks]

The original list implementation is mutable. When a tuple implementation is used, all the modifiers, add_lives and add_charges need to be rewritten to return a new wand object each time the wand is updated. Consequently, the functions use_wand and recharge_wand also need to be rewritten to deal with the new wand objects created.

Problem 4 (20 marks) Magic Wands – Part 2

Ermie, the smartest among the three friends in the Happy Magic School, has defined a higher-order function `sum_magical_power(maker, c, d)` to make a new magical item (e.g., wand, etc.) by combining the power of multiple magical items (e.g., a set of wands) and adding some new charges and new lives to it. `sum_magical_power` takes in 3 arguments: `maker`, which is a constructor for the magical item (e.g., `make_wand`), `c` number of new **charges**, and `d` number of new **lives** (with the same meanings of charges and lives as defined in Problem 3), and RETURNS a function that takes in as argument `items`, which is a sequence of magical items as follows:

```
def sum_magical_power(maker, c, d):
    def collect_magic(items):
        total_charges = 0
        total_lives = 0

       # sums up the charges of items into total_charges
       # sums up the lives of items into total_lives

       <T1>

        return maker(total_charges + c, total_lives + d)
    return collect_magic
```

a) Provide a possible implementation of the code segment **<T1>** for the `sum_magical_power` higher-order function in Python. State any assumptions you make in your answer. [8 marks]

Grading policy:

Points will be deducted if the abstraction barrier is violated, i.e., accessors are not used for getting the charges and lives of the magical items.

T1:

```
for i in items:
    n = get_charges(i)
    m = get_lives(i)
    total_charges += n
    total_lives += m
```

b) Assume that you have access to the constructor, selectors, and modifiers for wands as defined in Problem 3, and the answer to Part (a) for the full definition of `sum_magical_power(maker, c, d)`. Write a Python function `make_great_wand(c, d)` that returns a function for creating a new wand by combining the power of a set of wands, and adding `c` new charges and `d` new lives to it. State any other assumptions you make in your answer. [6 marks]

Grading policy:

This should return a function.

```
def make_great_wand(c, d):

    return sum_magical_power(make_wand, c, d)
```

An example call to the `make_great_wand` function is shown below:

```
>>> w1 = make_wand(5, 3)
>>> w2 = make_wand(2, 4)
>>> wands = (w1, w2)
>>> w3 = <T2>  # Use make_great_wand to combine and add to the
               # power of w1 and w2
>>> get_charges(w3)
8
>>> get_lives(w3)
8
```

c) Provide a possible expression of the call to the function `make_great_wand` at **`<T2>`** that would lead to the results as shown in the rest of the example above. State any assumptions you make in your answer. [6 marks]

T2:

```
w3 = make_great_wand(1, 1)(wands)
```

*** END OF PAPER ***