CS1010FC — Programming Methodology
School of Computing
National University of Singapore

Solutions for Re-Mid-Term Test

18 April 2015 **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 test**. 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 test is **1 hour 45 minutes**.
4. The maximum score of this test is **100 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 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 test.
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 = 100
for i in range(11, 21):
    if i % 3:
        x += 1
    elif i % 3 == 2:
        x -= 2
print(x)
```

[5 marks]

```
107
```

This is to test that students understand the `for` loop, `if-else` and also the modulo operator (`%`).

B.
```
x = 200
y = 100
def f(x, y):
    x = 5
    x = (x + y) // 3
    return x
print(f(y, x))
```

[5 marks]

```
68
```

This is to test that students understand simple function calls, scope and also the quotient (`//`) operator.

C.
```
def thrice(f):
    return f(f(f))
def add1(x):
    return x+1
print(thrice(thrice(add1))(3))
```

[5 marks]

Error! `thrice(add1)` will end up trying to apply `add1` to itself, resulting in an error.

This is to test that students are truly familiar with nested function calls. This question is made very similar to a homework question on purpose.

D.
```
i = 2
s = (0, 1)
while i < 10:
    if i%3 != 0:
        s = s + (s[len(s)-2]+s[len(s)-1],)
    i += 1
print(s[4] + len(s))
```

[5 marks]

```
10
```

This is to test that students understand `while` loops and also tuples.

E.
```
x = (1,2,3)
for i in range(1,5):
    x = x*i
print(x[16:20])
```

[5 marks]

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

This is to test that students understand tuple multiplication and indexing.

F.
```
x = 5
y = 10
z = 15
def f(x,y):
    def g(y,z):
        def h(z,x):
            return x + y + z
        return h(z,y)
    return g(y,x)
print(f(y,x))
```

[5 marks]

```
20
```

This is to test that students are able to deal with complicated function calls and can keep track of them.

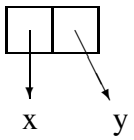
Question 2: Implementing Scheme Data Structures [24 marks]

In the Scheme programming language, the basic data structure is call a `cons` pair. In this question, we will simulate this data structure and work with it. Assume that the functions `cons`, `car` and `cdr` exist, such that:

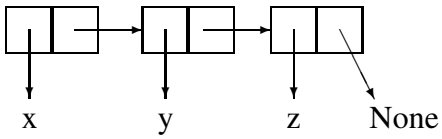
```
car(cons(x,y)) = x
cdr(cons(x,y)) = y
```

`cons(x,y)` returns a pair object that joins two objects `x` and `y`. There is also a function `is_pair` that takes in one parameter and returns `True` if the input is a `cons` pair. You can assume that the order of growth for `cons`, `car` and `cdr` are all $O(1)$ for both time and space.

The `cons` pair can be represented graphically as follows:



In order to combine more than two elements together, we create a chain of `cons` pairs called a Scheme list. A special property of a Scheme list is that the last element in the chain is always `None`. This is illustrated as follows:



The function `mylist` will create a new Scheme list object from an input sequence and the function `length` will return the number of elements in a Scheme list object. The function `to_tuple` will create a tuple corresponding to a Scheme list and the function `reverse` will reverse a Scheme list.

Sample execution:

```
>>> ml = mylist((1,2,3))

>>> car(ml)
1
>>> car(cdr(ml))
2
>>> car(cdr(cdr(ml)))
3
>>> cdr(cdr(cdr(ml)))
None

>>> length(ml)
3
>>> to_tuple(ml)
(1, 2, 3)
```

```
>>> rl = reverse(ml)
>>> car(rl)
3
>>> car(cdr(rl))
2
>>> car(cdr(cdr(rl)))
1
>>> cdr(cdr(cdr(rl)))
None
```

A. Please provide a possible implementation for `mylist`. [4 marks]

```
def mylist(seq):
    if seq == ():
        return None
    else:
        return cons(seq[0],mylist(seq[1:]))
```

B. Please provide a possible implementation for `length`. [4 marks]

```
def length(slst):
    if not is_pair(lst):
        return 0
    else:
        return 1+length(cdr(slst))
```

This question is actually the same as what was discussed in lecture. The student needs only to change the functions from the Python version to `car`, `cdr` and `is_pair`.

C. What is the order of growth in terms of time and space for the function you wrote in Part (B) in terms of n where n is the length of `slst`. [2 marks]

Time: $O(n)$, since the function will need to iterate down the full chain.

Space: $O(n)$, since the function is recursive and has pending operations.

D. If your function `length` in Part(B) was implemented as a recusive process, please provide a possible iterative implementation for `length`. If it were iterative, please provide the recursive implementation. [4 marks]

```
def length(slst):
    count = 0
    while is_pair(slst):
        count += 1
        slst = cdr(slst)
    return count
```

E. What is the order of growth in terms of time and space for the function you wrote in Part (D) in terms of n where n is the length of `slst`. [2 marks]

Time: $O(n)$, since the function will need to iterate down the full chain.

Space: $O(1)$, since we need only one variable `count`.

F. Please provide a possible implementation for `to_tuple`. [4 marks]

```
def to_tuple(slst):
    result = ()
    while is_pair(slst):
        result += (car(slst),)
        slst = cdr(slst)
    return result
```

G. Please provide a possible implementation for `reverse`. [4 marks]

```
def reverse(slst):
    return mylist(to_tuple(slst)[::-1])
```

Alternative:

```
def reverse(slst):
    def append(s1, s2):
        if not is_pair(s1):
            return s2
        else:
            return cons(car(s1), append(cdr(s1),s2))

    if not is_pair(slst):
        return None
    else:
        return append(reverse(cdr(slst)), cons(car(slst),None))
```

Question 3: Higher-Order Functions [22 marks]

A. **[Warm-up]** Consider the following sum:

$$a(1)+a(2)+...+a(n)$$

where a is a function. Suppose the function `asum(a,n)` computes this sum for $n \geq 1$ and is defined in terms of `sum` (see Appendix) as follows:

```
def asum(a,n):
    return sum(<T1>,
               <T2>,
               <T3>,
               <T4>)
```

Please provide possible implementations for T1, T2, T3, and T4. [6 marks]

`<T1>:` [2 marks]	`a`
`<T2>:` [1 mark]	`1`
`<T3>:` [2 marks]	`lambda x: x+1`
`<T4>:` [1 mark]	`n`

B. Next, suppose the function `asum(a,n)` from Part (A) is defined in terms of `fold` (see Appendix) as follows:

```
def asum(a,n):
    return fold(<T5>,
                <T6>,
                <T7>)
```

Please provide possible implementations for T5, T6, and T7. [6 marks]

`<T5>`: [2 marks]	`lambda x,y: x+y`
`<T6>`: [2 marks]	`lambda x: a(x+1)`
`<T7>`: [2 marks]	`n-1`

C. Consider the following expression:

$$(((\cdots(b(0)/b(1))/b(2))/\cdots)/b(n)$$

Suppose `bquotient(b,n)` computes this expression (for $n \geq 0$), i.e. `bquotient(b,0)` = $b(0)$, `bquotient(b,1)` = $b(0)/b(1)$, etc. and it is defined in terms of `fold` as follows:

```
def bquotient(b,n):
    return fold(<T8>,
                <T9>,
                <T10>)
```

Please provide possible implementations for T8, T9, and T10. [4 marks]

`<T8>`: [2 marks]	`lambda x,y: y/x`
`<T9>`: [1 mark]	`b`
`<T10>`: [1 mark]	`n`

D. Consider the following continued fraction:

$$\cfrac{1}{1+\cfrac{2}{1+\cfrac{\ddots}{1+n}}}$$

Suppose `cfraction(n)` computes this expression (for $n \geq 1$), i.e. `cfraction(1)` = 1, `cfraction(2)` = $\frac{1}{1+2}$, `cfraction(3)` = $\frac{1}{1+\frac{2}{1+3}}$, etc. and it is defined in terms of `fold` as follows:

```
def b cfraction(n):
    return fold(<T11>,
                <T12>,
                <T13>)
```

Please provide possible implementations for T11, T12, and T13. [6 marks]

`<T11>`: [2 marks]	`lambda x,y: x/(1+y)`
`<T12>`: [2 marks]	`lambda x: n-x`
`<T13>`: [2 marks]	`n-1`

Question 4: Chocolate Packing [24 marks]

You work for a chocolate factory that makes 2 types of chocolates, a small 1×1 chocolate and a big 2×2 chocolate. The chocolates are packed in square boxes of length n. i.e. of size $n \times n$. Each box can contain both small and big chocolates. For example, a box of length 2 (i.e. 2×2) can fit either one big chocolate or 4 small ones. A box of length 3 (i.e. 3×3) can fit either one big chocolate and 5 small ones or 9 small chocolates. Your goal is to model boxes of chocolates using the following functions:

- `make_box(n,b,s)` will create a box of length n, containing b big chocolates and s small chocolates.
- `size(b)` will return the length of box b.
- `get_big(b)` will return the number of big chocolates in box b.
- `get_small(b)` will return the number of small chocolates in box b.

Note: You are limited to using **tuples** for this question, i.e. you cannot use lists and other Python data structures.

A. Decide on an implementation for the chocolate box object and implement the functions `make_box(n,b,s)`, `size(b)`, `get_big(b)` and `get_small(b)`. [4 marks]

```
def make_box(n,big,small):
    return (n,big,small)

def size(box):
    return box[0]

def get_big(box):
    return box[1]

def get_small(box):
    return box[2]
```

B. Implement the function `is_valid` that takes in one parameter and returns `True` if the input is a valid box object (i.e. the format is correct and all the chocolates will fit inside), or `False` otherwise. [6 marks]

```
def is_valid(box):
    if type(box) != tuple or len(box) != 3:
        return False
    for i in box:
        if type(i) != int:
            return False
    if get_big(box)*4 + get_small(box) > size(box)**2: # Check all can fit
        return False
    return (size(box)//2)**2 >= get_big(box) # Make sure the big ones fit.
```

C. The functions `add_big(b,n)` and `add_small(b,n)` add *n* big and small chocolates to a box *b*, respectively, and return a **new** box. If it is not possible to add *n* chocolates, `False` is returned instead. Provide possible implementations of `add_big` and `add_small`. Note that you are allowed to define helper functions if you think they would be helpful. [6 marks]

Sample execution:

```
>>> b1 = make_box(3,0,0)
>>> size(b1)
3
>>> get_big(b1)
0
>>> get_small(b1)
0

>>> add_big(b1,2)
False
>>> add_small(b1,10)
False
```

```
>>> b2 = add_big(b1,1)
>>> size(b2)
3
>>> get_big(b2)
1
>>> get_small(b2)
0

>>> b3 = add_small(b2,5)
>>> size(b3)
3
>>> get_big(b3)
1
>>> get_small(b3)
5

>>> add_big(b3,2)
False
>>> add_small(b3,10)
False
```

```
def add_big(box, num):
    trybox = make_box(size(box), get_big(box)+num,get_small(box))
    if is_valid(trybox):
        return trybox
    else:
        return False

def add_small(box, num):
    trybox = make_box(size(box),get_big(box), get_small(box)+num)
    if is_valid(trybox):
        return trybox
    else:
        return False
```

D. The functions `add_big_fit(b,n)` and `add_small_fit(b,n)` add n big and small chocolates to a box b, respectively, and always return a **new** box. If the additional chocolates, the new box will be of the minimal size to fit the additional chocolates. You can assume that the functions `sqrt` and `ceil` are available. `sqrt(n)` returns the square root of n and `ceil(n)` returns integer larger than n if n is not an integer. [8 marks]

Sample execution:

```
>>> b4 = add_big_fit(b3,3)
>>> size(b4)
5
>>> get_big(b4)
4
>>> get_small(b4)
5

>>> b5 = add_small_fit(b4,5)
>>> size(b5)
6
>>> get_big(b5)
4
>>> get_small(b5)
10
```

```
def fit(big,small):
    min_size = ceil(sqrt(big))*2
    required = 4*big + small
    while min_size**2 < required:
        min_size += 1
    return make_box(min_size,big,small)

def add_big_fit(box, num):
    trybox = add_big(box, num)
    if not trybox:
        return fit(get_big(box)+num, get_small(box))
    else:
        return trybox

def add_small_fit(box, num):
    trybox = add_small(box, num)
    if not trybox:
        return fit(get_big(box), get_small(box)+num)
    else:
        return trybox
```

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 —