CS1010S — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

2 March 2016 **Time allowed:** 1 hour 45 minutes

Student No:	**S**	**O**	**L**	**U**	**T**	**I**	**O**	**N**	**S**

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 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 = 2
y = 7
def f(x):
    return g(y + x)
def g(y):
    return x * y
print(f(y))
```

```
28
```

B. [5 marks]

```
t = ()
for i in range(5):
    if i % 2:  # odd?
        t = t + (i,)
    else:
        t = (i, t)
print(t)
```

```
(4, (2, (0, (), 1), 3))

4 marks for (4, (2, (0, 1), 3))
4 marks for (4, 2, 0, (), 1, 3)
```

C. [5 marks]

```
def bar(x):
        return lambda y: x(x(y))
def foo(y):
        return lambda x: x(y)
print((foo(bar)(bar)(lambda x:x*2)(2)))
```

```
32
```

D. [5 marks]

```
a = 5
b = 10
if a < b:
    b = 5
    if b < a:
        a = 3
else:
    b = 7
if b > 5:
    a = 9
print(a + b)
```

```
10
```

E. [5 marks]

```
n = 24
i = 2
while n > 0:
    if n % i == 0:
        i = i + 1
        continue
    if n > i:
        i = i - 1
    break
print(i)
```

```
4
```

4 marks for 5

F. [5 marks]

```
def hi(a, b):
    return a // b
def he(a, b):
    return a * b
print(he(3, hi(7, 3)))
```

```
6
```

Question 2: Nine-pin Bowling [19 marks]

Nine-pins bowling is a game where a player rolls a ball to strike at nine bowling pins. In our version of nine-pin bowling, players only have one chance to knock down nine pins before they are reset after every bowl. (This is different from the commonly played ten-pin bowling where players get two tries to knock down ten pins.)

A game of nine-pins can thus be represented as a string of digits, with each digit indicating the number of pins knocked down in a bowl.

For example, a player with a game of 5 bowls where he knocks down 4 pins in his first bowl, and one more pin every subsequent bowl would be represented by the string `"45678"`. A player scoring a perfect game of 10 bowls would be `"9999999999"`.

A. [Warm up] the total number of pins felled. Without using any higher-order functions, write a function `pins_felled` that takes as input a game and returns the total number of pins knocked down in a game.

Example:

```
>>> pins_felled("45678")
30

>>> pins_felled("450927")
27

>>> pins_felled("9999999999")
90
```

[4 marks]

```
def pins_felled(game):
    total = 0
    for i in s:
        total += int(i)
    return total
```

or

```
def pins_felled(game):
    if s:
        return int(s[0]) + pins_felled(s[1:])
    else:
        return 0
```

B. Is the function you wrote in Part (A) recursive or iterative? State the order of growth in terms of time and space for the function you wrote in Part (A). Explain your answer. [3 marks]

The function is **RECURSIVE / ITERATIVE** (circle one)

Time:
Iterative: $O(n)$, where n is the length of the input string.
Recursive: $O(n^2)$, where n is the length of the input string.

Space:
Iterative: $O(1)$, where n is the length of the input string.
Recursive: $O(n^2)$, where n is the length of the input string.

We mark according to the code written in part A, even though it might be wrong.

To reward players who manage to hit all nine pins in a bowl, the number of pins felled in the next bowl will be added to the score.

For example, the game `"4927"` will be $4+9+2+2+7=24$ as the score for the second bowl is $9+2$.

The score for a perfect game of `"9999999999"` will be 171, because each bowl except for the last bowl is worth $9+9=18$ points. $(9+9)\times 9+9=171$.

The score for the game `"049090"` is $4+9+9=22$ because after knocking down nine pins in a bowl, the player did not knock down any in the next bowl.

C. Write a **recursive** function `compute_score(game)` which takes in a game and returns the score of the game using this scoring scheme. [4 marks]

```
def compute_score(game):
    if len(game) == 1:
        return int(game[0])
    elif game[0] == '9':
        return 9 + int(game[1]) + compute_score(game[1:])
    else:
        return int(game[0]) + compute_score(game[1:])
```

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: $O(n^2)$ where n is the length of the string. This is because of string slicing at every of the n recursions.

Space: $O(n^2)$ where n is the length of the string. This is because of string slicing at every of the n recursions takes up space in the heap.

Other answers are acceptable according to what was written in part C. If the code does not make sense, then we cannot compute an order of growth, so no marks will be given.

E. Write an iterative function `compute_score(game)` which takes in a game and returns the score of the game using this scoring scheme. [4 marks]

```
def compute_score(game):
    score = 0
        for i in range(len(game)-1):
        score += int(game[i])
        if game[i] == '9':
            score += int(game[i+1])
    return score + int(game[-1])
```

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: $O(n)$ where n is the length on the input string. This is because the for-loop loops n times.

Space: $O(1)$, because there are no delayed operations or new objects being created every iteration.

Other answers are acceptable according to what was written in part E. If the code does not make sense, then we cannot compute an order of growth, so no marks will be given.

Question 3: Higher-Order Function [22 marks]

For this question, you are not to reuse the functions defined in Question 2. Parts A, B and C should be solved using the given higher-order function, and not by recursion or iteration.

A. It turns out that the function `pins_felled` in Question 2A can be is defined in terms of the higher-order function `sum` (refer to the Appendix) as follows:

```
def pins_felled(game):
    <PRE>
    return sum(<T1>,
               <T2>,
               <T3>,
               <T4>)
```

Please provide possible implementations for the terms `T1`, `T2`, `T3` and `T4`. You may also optionally define functions in `<PRE>` if needed. [5 marks]

Correct answer, which uses `sum` to increment an index:

```
def pins_felled(game):
    return sum(lambda x: int(game[x]),
               0,
               lambda x: x+1,
               len(game)-1)
```

Note the ending condition is `len(game)-1` due of the implementation of `sum` (terminates when `a > b`).

Some students attempted another approach which is to slice the string in the series, or convert the game into an integer and then start "slicing" the digits, as such:

```
def pins_felled(game):
    return sum(lambda x: x[0],
               game,
               lambda x :x[1:],
               "")
```

or

```
def pins_felled(game):
    return sum(lambda x: x % 10,
               int(game),
               lambda x: x // 10,
               0)
```

Now the idea is commendable, but there is one fatal flaw. That is that the terminating condition for `sum` is `a > b`, which is immediately satisfied for the above two incorrect solutions. (3 marks for such solutions.)

This question is marked as a whole, so simply filling common values in the parts will not give you the marks. There has to be some understanding shown in the answer.

B. Likewise, we can also express the `compute_score` function from Question 2C in terms of `sum` as follows:

```
def compute_score(game):
    <PRE>
    return sum(<T5>,
               <T6>,
               <T7>,
               <T8>)
```

Please provide possible implementations for the terms `T8`, `T9`, `T10` and `T11`. You may also optionally define functions in `<PRE>` if needed. [5 marks]

The solution is similar to the previous part, just that `term` is modified to check either the next bowl or earlier bowl. There is a need to check the bounds so you do not check beyond the last bowl (or the first bowl depending on your implementation).

```
def compute_score(game):
    return sum(lambda x: 9 + int(game[x+1]) if int(game[x]) == 9 \
                         and x < len(game)-1 else int(game[x]),
               0,
               lambda x: x+1,
               len(game)-1)
```

or

```
def compute_score(game):
    return sum(lambda x: 2 * int(game[x]) if int(game[x-1]) == 9 \
                         and x > 0 else int(game[x]),
               0,
               lambda x: x+1,
               len(game)-1)
```

-1 mark for each minor mistake, like wrong bounds or not converting to int.
0 marks if the idea of checking the next or previous bowl is not seen.

C. **[Challenging]** Knocking down all 9 pins twice in a row is known as a "double". For example, the game `"4099012"` contains a double as the player knocked down 9 pins in the third and fourth bowl. But the game `"9898989"` does not contain any doubles because after hitting 9 pins, the player only managed to hit 8 in the next bowl.

The function `has_double` takes as input a game and return `True` if the game contains at least one double, and `False` otherwise. `has_double` can be expressed in terms of the higher-order function `fold` (refer to the Appendix) as follows:

```
def has_double(game):
    <PRE>
    return fold(<T9>,
                <T10>,
                <T11>)
```

Please provide possible implementations for the terms `T9`, `T10` and `T11`. You may also optionally define functions in `<PRE>` if needed. [5 marks]

```
def has_double(game):
    return fold(lambda a, b: a or b,
                lambda x: game[x] == '9' and game[x+1] == '9',
                len(game) - 2)
```

-1 mark for minor mistakes like out-of-bounds.
Essentially the idea is to "fold" boolean values using `or`.

D. The junior version of nine-pin bowling is played exactly the same, except that it has fewer pins. For example, one version might be played with 7 pins per bowl. Depending on the age of the players, the number of pins can range from 1 to 9.

Write a function `n_score` that takes as inputs a game, and the maximum number of pins being played in the game. The function returns the score (as described in Question 2C) of the game. [3 marks]

```
def n_score(game, n):
    return sum(lambda x: n + int(game[x+1]) if int(game[x]) == n \
                         and x < len(game)-1 else int(game[x]),
               0,
               lambda x:x+1,
               len(game)-1)
```

The answer is almost identical to `compute_score`, where the only difference is checking agaist `n` instead of a hard-coded value `9`. Full marks is given if the student duplicates his/her code from Q2 and correctly replaces the hard-coded values with the variable, even though the answer in Q2 is incorrect.

This question is testing the understanding of replacing hard-coded values with a variable, which is what we will not penalize again for wrong code.

E. Now we can write a function `scorer` that will take as input n, the number of pins, and return a function that can compute the real score of a n-pin bowling game.

Example:

```
>>> seven_scorer = scorer(7)
>>> seven_scorer("77777")
63
```

Provide an implementation of `scorer`. [4 marks]

```
def scorer(n):
    return lambda game: n_score(game, n)
```

1 mark only if function does not return a function, since the main point of this question is to test returning a function.

Question 4: Ten-pin Bowling [29 marks]

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

The commonly played bowling game is ten-pin bowling. A game of bowling is made up of frames, and in each frame, the player is given two bowls to knock down all 10 pins before they are reset. (This is different from nine-pin bowling where the pins are reset after every bowl.)

To compute the score for a game, the number of pins knocked down during each of the two bowls for each frame has to be recorded. Thus, a game consists of a sequence of frames, with a complete game having 10 frames.

The following functions are to be implemented:

- `new_frame(b1, b2)` takes as inputs the number of pins knocked down in the first and second bowl and returns a frame. If the player knocks down all 10 pins in his first bowl, it is called a *strike* and the player does not need to bowl a second time. In this case, the frame will be created as `new_frame(10, 0)`.
- `get_first` and `get_second` takes as input a frame, and returns, respectively, the number of pins knocked down in the first and second bowl of the frame.
- `is_strike` takes as input a frame and returns `True` if the frame is a strike, i.e., all 10 pins were knocked down in the first bowl.
- `is_spare` takes as input a frame and returns `True` if all 10 pins were knocked down in exactly two bowls.
- `create_new_game` takes no inputs and return a new game
- `add_to_game` takes as inputs a game and a frame, and returns a game with the frame appended to it.

A. Explain how you will use tuples to represent a frame as well as a game. [2 marks]

Each frame will be a tuple of two elements, each will be the first and second bowl respectively. A game will be a tuple of frames with the first element being the first frame of the game.

B. Provide an implementation for the functions `new_frame`, `get_first`, `get_second`, `is_strike` and `is_spare`. [4 marks]

```
def new_frame(b1, b2):
    return (b1, b2)



def get_first(frame):
    return frame[0]



def get_second(frame):
    return frame[1]



def is_strike(frame):
    return frame[0] == 10



def is_spare(frame):
    return (frame[0] < 10) and (frame[0] + frame[1] == 10)
```

-1 mark for each mistake in each function.
One common mistake is to not check for strike in `is_spare` as the question states that a spare must use exactly 2 bowls to fell all ten pins.

C. Provide an implementation for the functions `create_new_game` and `add_to_game`. [2 marks]

```
def create_new_game():
    return ()


def add_to_game(game, frame):
    return game + (frame,)
```

[Important!] For the remaining parts of this question, **you should not break the abstraction of a frame.**

D. Write a function `count_strikes` which takes as input a game and returns the number of strikes in the game. [3 marks]

```
def count_strikes(game):
    total = 0
    for frame in game:
        if is_strike(frame):
            total += 1
    return total
```

or

```
def count_strikes(game):
    return len(filter(is_strike, game)) # using filter in appendix
```

-2 marks for breaking abstraction, since it is stated explicitly above!

E. A double is when a player gets two strikes in a row, i.e., a strike immediately following after another strike. Write a function `has_double` that takes as input a game and returns `True` if the game contains a double, and `False` otherwise. [4 marks]

```
def has_double(game):
    for i in range(len(game)-1):
        if is_strike(game[i]) and is_strike(game[i+1]):
            return True
    return False
```

F. The computer in charge of monitoring the pins will simply return the number of pins felled for each bowl. Thus, it will output a tuple of integers to represent each game. For example, an output of `(4, 6, 10, 3, 2, 5, 5)` is a game comprising of 4 frames with 1 strike (in the 2nd frame) and 2 spares (in the 1st and 4th frame). Note that only one bowl is recorded for a strike and the subsequent bowl is the first bowl of the next frame.

Between this representation and what you proposed in Part A, which do you think is better? Explain briefly. [2 marks]

The represetation in part A is better because it closely reflects the actual organization of a game into frames containing bowls. It is then easier to count the strikes and spares and obtain the score of a frame.

The bowls representation can be useful to perform computations like calculating the score because the bonus points are based on bowls, not frames.

Any sensible answer will be accepted, keywords being the representation in part A fits the natural organization of a game.

An unacceptable answer would be that the storage would be smaller, because the space complexity of both representation is the same.

G. Write a function `bowls_to_game` which takes as input a tuple of integers representing the number of pins felled per bowl (as described in Part F), and returns a game, which is the representation you proposed in part A. [4 marks]

```
def bowls_to_game(bowls):
    i = 0
    game = ()
    while i < len(bowls):
        if bowls[i] == 10:
            game = add_to_game(create_frame(10, 0), game)
            i += 1
        else:
            game = add_to_game(create_frame(bowls[i], bowls[i+1]), game)
            i += 2
    return game
```

-1 mark for each mistake, such as breaking abstraction, wrong range resulting in out-of-bounds, using `for i in game` then doing `i+=2`, putting `return False` in the loop, etc.

H. Write a function `game_to_bowls` that takes as input a game, and return a tuple of pins felled per bowls (as described in Part F). Essentially it reverses the function `bowls_to_game`. [4 marks]

```
def game_to_bowls(game):
    bowls = ()
    for frame in game:
        if is_strike(frame):
            bowls = bowls + (10,)
        else:
            bowls = bowls + (get_first(frame), get_second(frame))
    return bowls
```

-1 mark for each mistake, such as breaking abstraction, wrong range resulting in out-of-bounds, putting `return False` in the loop, etc.

I. **[Warning: Hard]** Computing the score for ten-pin bowling is more complicated than nine-pin bowling. The final score is the total number of pins felled, plus some additional bonuses:

- When a player bowls a strike, that frame is awarded 10 points (for the 10 pins felled), plus the number of pins felled in his next **two bowls** (not to be confused with frames).
- When a player bowls a spare, the frame is awarded 10 points (for the 10 pins felled), plus the number of pins felled in his **next bowl** (not to be confused with frame).
- For simplicity, a strike or spare in the last frame will not be able to increase its score. Likewise, a double in the 9th and 10th frame would only give an addition 10 points to the 9th frame.

The following example shows a game with the score of each frame tabulated below the pins felled. The score of the game is the total sum of each frame score:

Frame	1	2	3	4	5	6	7	8	9	10
Pins felled	10	7, 3	9, 0	10	0, 8	8, 2	0, 5	10	10	10
Frame Score	20	19	9	18	8	10	5	30	20	10

Write the function `compute_score` which takes as input a game and returns the final score of the game. If you find it easier to work with a sequence of bowls instead, you can use the functions defined earlier to do the conversion. [4 marks]

```
def compute_score(game):
    def helper(bowl)
        if len(bowl) == 0:
            return 0
        elif len(bowl) == 1:
            return bowl[0]
        elif len(bowl) == 2:
            if bowl[0] == 10:
                return 10 + 2*bowl[1]
            else:
                return bowl[0] + bowl[1]
        elif bowl[0] == 10:
            return 10 + bowl[1] + bowl[2] + helper(bowl[1:])
        elif bowl[0] + bowl[1] == 10:
            return 10 + bowl[2] + helper(bowl[2:])
        else:
        return bowl[0] + bowl[1] + helper(bowl[2:])
    return helper(game_to_bowls(game))
```

-1 mark for each mistake, such as breaking abstraction, wrong range resulting in out-of-bounds, not checking for strike after a strike, etc.

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 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 —