NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5001—Software Development Fundamentals
Academic Year 2023/2024, Semester 2
PROBLEM SET 5

FUNCTIONS AS FIRST-CLASS CITIZENS

These exercises are optional and ungraded, and for your own practice only. Contact `yongqi@nus.edu.sg` for queries.

SIMPLE EXERCISES

Question 1. Without using IDLE, evaluate the results of the following expressions:

1. `(lambda x: (lambda y: y + 1)(x))(1)`
2. `list((lambda x: (i + 'a' for i in x))('abc'))`
3. `lambda x: x()(1)`
4. `(lambda x, y, z: x + y + z)(1, 2, 3)`
5. `map(lambda x: x > 2, [9, 8, 7, 1, 2, 3])`
6. `list(map(lambda x: x > 2, [9, 8, 7, 1, 2, 3]))`
7. `filter(lambda x: x > 2, [9, 8, 7, 1, 2, 3])`
8. `list(filter(lambda x: x > 2, [9, 8, 7, 1, 2, 3]))`
9. `list(map(lambda x: 'o' if x % 2 else 'e', range(8)))`
10. `list(filter(lambda x: 'o' if x % 2 else 'e', range(8)))`
11. `list(map(str, filter(lambda x: x % 2, range(10))))`
12. `str(list(filter(lambda x: x > 30, map(lambda x: x * x, range(10)))))`

Question 2. β-reduce $(\lambda x.\lambda y.\lambda z.xyz)(\lambda x.\lambda y.xy)(\lambda x.2x)(5)$ as much as possible.

SHORT PROGRAMMING QUESTIONS

Question 3 (Stolen from CS1010S AY 18/19 Semester 2 Midterm Assessment). What is the output of the following program:

```
def foo(x):
    return lambda a: a(x)(a)
def bar(x):
```

```
    return lambda a: a(x + 1)
print(foo(5)(bar)(lambda x: x))
```

Question 4. Suppose you are given the following higher-order functions:

```
def sum(term, a, next, b):
    if (a > b):
        return 0
    return term(a) + sum(term, next(a), next, b)

def accumulate(fn, init, ls, get):
    if not ls:
        return init
    return fn(get(ls), accumulate(fn, init, ls[1:], get))

def fold(op, f, n):
    if n == 0:
        return f(0)
    return op(f(n), fold(op, f, n - 1))
```

We are going to use them to perform some computations with Bowling.

"Ten-pin bowling is a type of bowling in which a bowler rolls a bowling ball down a wood or synthetic lane toward ten pins positioned in a tetractys (equilateral triangle-based pattern) at the far end of the lane. The objective is to knock down all ten pins on the first roll of the ball (a strike), or failing that, on the second roll (a spare)." Source: Wikipedia.

A game consists of ten frames, where each frame the player is given a chance to knock down as many pins as possible (10). A strike is represented by `'X'`, a spare is represented by `'/'`, and in any other case, we simply display the number of pins knocked on a single roll as the number itself. For example, the frame `'X'` means that player made a strike, the frame `'5/'` means that the player knocked down 5 pins on the first roll then made a spare in the second roll, and the frame `'45'` means that the player knocked down 4 pins in the first roll and 5 pins in the second.

We are going to use a simplified scoring system. For each frame, a strike gives 30 points, a spare gives 20 points and for any other frame, the score given is simply the total number of pins knocked down in the first and second rolls.

Question 4 (i). Write a function `knocked_down(game)` that takes an input game as a string and returns the number of pins knocked down in the game. Your function should use `sum` as described above, and your solution must comply with the following template. All of the terms defined cannot be iterative or recursive.

```
def knocked_down(game):
    # you may define t1, t2, t3 and t4 here.
    # you are only allowed to return the following.
    return sum(t1, t2, t3, t4)
```

Example runs follow:

```
>>> knocked_down('XXX9/9/45421/XX')
95
>>> knocked_down('12345/12345/12345/1/')
70
>>> knocked_down('XXXXXXXXXX')
100
```

Assume that the input game is always a valid game of ten frames.

Question 4 (ii). Write a function `score(game)` that takes in a game and returns the score of the game. In a similar fashion, use the following template.

```
def score(game):
    # you may define t1, t2, t3 and t4 here.
    # you are only allowed to return the following.
    return accumulate(t1, t2, t3, t4)
```

Example runs follow:

```
>>> score('XXX9/9/45421/XX')
225
>>> score('12345/12345/12345/1/')
110
>>> score('XXXXXXXXXX')
300
```

Question 4 (iii). A triple is defined as three Xs in a row. Write a function `has_triple(game)` that determines whether a triple exists in `game`. In a similar fashion, use the following template.

```
def has_triple(game):
    # you may define t1, t2, and t3 here.
    # you are only allowed to return the following.
    return fold(t1, t2, t3)
```

Example runs follow:

```
>>> has_triple('XXX9/9/45421/XX')
True
>>> has_triple('12345/12345/12345/1/')
False
>>> has_triple('XXXXXXXXXX')
True
```

Question 5. Write one `burger_price(burger)` solution recursively, then another using `map`, `filter` and/or `reduce`. You may use the following iterative version to help you.

```
def burger_price(burger):
    ingredient_price = {
        'B': 50,
        'C': 80,
        'P': 150,
        'V': 70,
        'O': 40,
        'M': 90
    }
    acc = 0
    for ing in burger:
        acc += ingredient_price[ing]
    return acc
```

Question 6. Consider the following function:

```
def scale(ls, n):
    acc = []
    for i in ls:
        acc.append(i * n)
    return acc
```

Question 6 (i). Write `scale` using `map`, `filter` and `reduce` without any loops or recursion, and it should work for tuples instead of lists.

Question 6 (ii). Write `scale` recursively.

Question 7. Write the following function using `map`, `filter` and `reduce` without any loops or recursion:

```
def sum_square_of_digits(n):
    acc = 0
    while n:
        acc += (n % 10) ** 2
        n //= 10
    return acc
```

Question 8. Observe our basic BFS algorithm from Lecture 7:

```
def traversable(d, src, dst):
    # d is a dictionary of k-v pairs such that all elements in v are
    # neighbours of k
    frontier = [src]
    visited = set()
    while frontier:
        current = frontier.pop(0)
        if current in visited: continue
        if current == dst: return True
        visited.add(current)
        frontier.extend(d[current])
    return False
```

Write a function `bfs(d, src, dst)` that also 1) receives `d` as a dictionary mapping cities to a list of their neighbours, 2) the source city and 3) the destination city, that does exactly the same as `traversable` except that it is implemented declaratively (no loops, use recursion/map/filter/reduce).

Then observe another BFS implementation from Problem Set 4:

```
def neighbours(maze, r, c):
    def is_valid(coordinates):
        # check if coordinates is a valid and traversable cell
        x, y = coordinates
        return 0 <= x < len(maze) and    \
               0 <= y < len(maze[x]) and \
               maze[x][y] == 0
    # all possible neighbours are top, bottom, left and right
    possible_neighbours = [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]
    # remove the invalid neighbours
    return [i for i in possible_neighbours if is_valid(i)]

def solvable(maze):
    # maze is a 2D list of 0s and 1s; this represents a maze
    frontier = [(0, 0)]
    visited = set()
    while frontier:
        current = frontier.pop(0)
        if current in visited: continue
        if current == (len(maze) - 1, len(maze[0]) - 1): return True
        visited.add(current)
        frontier.extend(neighbours(maze, *current))
    return False
```

Notice that in `traversable`, `d` is only used to obtain the neighbours of the current city. Also, in `solvable`, we really only need the `maze` to obtain the neighbours of a cell at the current coordinates.

Both `traversable` and `solvable` are implemented using BFS. Thus, we should write a single BFS implementation and allow `traversable` and `solvable` to make use of this BFS implementation.

Re-write your `bfs` function earlier, except that its function signature is now `bfs(neighbours, src, dst)` and it receives 1) `neighbours` which is a function that receives some object and produces a list of its neighbours (either in a map of cities or in a maze), the source and the destination, and returns `True` if it is possible to traverse from the source to the destination. Then implement `traversable` and `solvable` using your brand new `bfs(neighbours, src, dst)` function.

Question 9. The Happy Sum is defined as $n^2+\ldots+4+1+4+\ldots+n^2$ for some positive n. We're obviously not going to ask you to define the function iteratively or recursively, that is too lenient.

Given the definition of `combine`, fill in the appropriate implementations in `happy_sum`.

```
def combine(f, op, n):
    res = f(0)
    for i in range(n):
        res = op(res, f(i))
    return res

def happy_sum(n):
    def f(x):
        'whatever you want'
    def op(x, y):
        'more stuff'
    n = 'hehehehehehehe'
    # obviously, don't change the return expression
    return combine(f, op, n)
```

LONG PROGRAMMING QUESTION

We are going to work on some really simple questions but using our maximum brain power to think differently.

sin

The Taylor Series expansion of sin is as follows:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots \quad \text{for all } x$$

Write a function `sin(x, n)` that approximates $\sin x$ using the first n terms of the Taylor Series. Do

so iteratively, then recursively, and finally only using `map`, `filter` and `reduce` without any loops or recursion.

cos

The Taylor Series expansion of cos is as follows:

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = x - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots \quad \text{for all } x$$

Write a function `cos(x, n)` that approximates $\cos x$ using the first n terms of the Taylor Series. Do so iteratively, then recursively, and finally only using `map`, `filter` and `reduce` without any loops or recursion.

Taylor Series, generally

Write a function `taylor` that approximates any Taylor Series (such as the expansion of sin and cos). How would someone use this function to compute, let's say `sin(0, 10)`, and then use the same function to compute `cos(90, 5)`? You may do so iteratively, recursively or any other method of your choosing, aside from the mind-numbingly obvious method of importing sin and cos from the math library.

SOLUTIONS

Question 1. Finding out the solutions to these is quite simple—just write the expressions in IDLE to see the results after evaluating them.

Here are the explanations for each expression.

(1.1) Answer: 2. `(lambda x: (lambda y: y + 1)(x))(1)` becomes
`(lambda y: y + 1)(1)` which becomes
`2`.

(1.2) Answer: `['aa', 'ba', 'ca']`. We first evaluate the argument to `list`.
`(lambda x: (i + 'a' for i in x))('abc')` becomes
`(i + 'a' for i in 'abc')` which gives a generator containing
`'aa', 'ba', 'ca'`. The result follows.

(1.3) Answer: `lambda x: x()(1)`. Essentially nothing happens because the function is not called lol.

(1.4) Answer: 6. `(lambda x, y, z: x + y + z)(1, 2, 3)` becomes `1 + 2 + 3`.

(1.5) Answer: `<map object at ...>`. The `map` function returns a map object. The elements in the map are lazily-evaluated and therefore nothing significant happens, only until someone retrieves elements from them. The elements in the map object are shown in 1.6.

(1.6) Answer: `[True, True, True, False, False, True]`.
Let the lambda expression be $f(x) = \textit{True}$ if $x > 2$, *False* otherwise.
So, the map object contains $f(9), f(8), f(7), f(1), f(2), f(3)$.

We know that $f(9) = \textit{True}$, $f(8) = \textit{True}$, $f(7) = \textit{True}$, $f(1) = \textit{False}$, $f(2) = \textit{False}$ and $f(3) = \textit{True}$. Therefore, the map object contains `True, True, True, False, False, True`. The result follows.

(1.7) Answer: `<filter object at ...>`. Similar to 1.5.

(1.8) Answer: `[9, 8, 7, 3]`. Just like in 1.6, we let the lambda expression be $f(x) = \textit{True}$ if $x > 2$, *False* otherwise. Only the elements in `[9, 8, 7, 1, 2, 3]` that passes the predicate f will be included in the filter object. We know that $f(9) = \textit{True}$, $f(8) = \textit{True}$, $f(7) = \textit{True}$, $f(1) = \textit{False}$, $f(2) = \textit{False}$ and $f(3) = \textit{True}$. Therefore, only 9, 8, 7 and 3 remain in the filter object. The result follows.

(1.9) Answer: `['e', 'o', 'e', 'o', 'e', 'o', 'e', 'o']`. This is straightforward. We map each number in `range(8)` to `'o'` if it is odd and `'e'` if it is even.

(1.10) Answer: `[0, 1, 2, 3, 4, 5, 6, 7]`. This is a trick question; the lambda expression is a predicate that determines whether an element should remain in the iterable. The function returns `'e'` or `'o'`, both of which are nonempty strings and hence truthy. Therefore, all elements in `range(8)` will remain in the filter object because the predicate returns true regardless of the element passed into it.

(1.11) Answer: `['1', '3', '5', '7', '9']`.
`filter(lambda x: x % 2: range(10))` keeps all elements in `range(10)` which are odd (because `lambda x: x % 2` returns 1 (true) when given an odd number and 0 (false) when given an even number). Therefore, only 1, 3, 5, 7, 9 remain in the filter object.

`map(str, ...)` takes each number in the filter object and converts them into a string. Therefore, the map object contains `'1'`, `'3'`, `'5'`, `'7'`, `'9'`. The result follows.

(1.12) Answer: `'[36, 49, 64, 81]'`. `map(lambda x: x * x, range(10))` takes each element in `range(10)` and squares them, therefore the resulting map object contains the squares of numbers 0 to 9.

`filter(lambda x: x > 30, ...)` keeps only the elements in the map object that are greater than 30. Therefore, the filter object only contains `36`, `49`, `64`, `81`. We then convert it into a list, which results in `[36, 49, 64, 81]`, and finally converting it into a string results in the above.

Question 2. You might take the following steps to obtain 10.

$$
\begin{aligned}
(\lambda x.\lambda y.\lambda z.xyz)(\lambda x.\lambda y.xy)(\lambda x.2x)(5) &\equiv_\alpha (\lambda x.\lambda y.\lambda z.xyz)(\lambda a.\lambda b.ab)(\lambda c.2c)(5) \\
&\triangleright_\beta (\lambda y.\lambda z.xyz)[x := \lambda a.\lambda b.ab](\lambda c.2c)(5) \\
&= (\lambda y.\lambda z.(\lambda a.\lambda b.ab)yz)(\lambda c.2c)(5) \\
&\triangleright_\beta (\lambda y.\lambda z.(\lambda b.ab)[a := y]z)(\lambda c.2c)(5) \\
&= (\lambda y.\lambda z.(\lambda b.yb)z)(\lambda c.2c)(5) \\
&\triangleright_\beta (\lambda y.\lambda z.yb[b := z])(\lambda c.2c)(5) \\
&= (\lambda y.\lambda z.yz)(\lambda c.2c)(5) \\
&\triangleright_\beta (\lambda z.yz)[y := \lambda c.2c](5) \\
&= (\lambda z.(\lambda c.2c)z)(5) \\
&\triangleright_\beta (\lambda z.2c[c := z])(5) \\
&= (\lambda z.2z)(5) \\
&\triangleright_\beta 2z[z := 5] \\
&= 10
\end{aligned}
$$

That is the long method. The shortcut is to see that the first two terms on the left don't really do very much. For example, $(\lambda x.\lambda y.\lambda z.xyz)abc$ just gives you abc. $(\lambda x.\lambda y.xy)ab$ just gives you ab.

$$
\begin{aligned}
(\lambda x.\lambda y.\lambda z.xyz)(\lambda x.\lambda y.xy)(\lambda x.2x)(5) &\triangleright_\beta (\lambda x.\lambda y.xy)(\lambda x.2x)(5) \\
&\triangleright_\beta (\lambda x.2x)(5) \\
&\triangleright_\beta 2(5) \\
&= 10
\end{aligned}
$$

Question 3. Let's do this using Python, then see what it looks like with lambda calculus.

Python method (go from one line to the next.)

```
def foo(x):
    return lambda a: a(x)(a)
def bar(x):
    return lambda a: a(x + 1)
foo(5)                          (bar)(lambda x: x)
(lambda a: a(5)(a))(bar)        (lambda x: x)
bar(5)                          (bar)(lambda x: x)
(lambda a: a(6))(bar)           (lambda x: x)
bar(6)                          (lambda x: x)
(lambda a: a(7))(lambda x: x)
(lambda x: x)(7)
7
```

Let $foo = \lambda x.\lambda y.yxy$ and $bar = \lambda a.\lambda b.b(a+1)$.

Short method:

$$\begin{aligned}
foo(5)(bar)(\lambda z.z) &\triangleright_\beta (\lambda y.yxy)[x := 5](bar)(\lambda z.z) \\
&= (\lambda y.y(5)(y))(bar)(\lambda z.z) \\
&\triangleright_\beta (y(5)(y))[y := bar](\lambda z.z) \\
&= (bar)(5)(bar)(\lambda z.z) \\
&\triangleright_\beta (\lambda b.b(a+1))[a := 5](bar)(\lambda z.z) \\
&= (\lambda b.b(6))(bar)(\lambda z.z) \\
&\triangleright_\beta (b(6))[b := bar](\lambda z.z) \\
&= bar(6)(\lambda z.z) \\
&\triangleright_\beta (\lambda b.b(a+1))[a := 6](\lambda z.z) \\
&= (\lambda b.b(7))(\lambda z.z) \\
&\triangleright_\beta (b(7))[b := (\lambda z.z)] \\
&= (\lambda z.z)(7) \\
&\triangleright_\beta z[z := 7] \\
&= 7
\end{aligned}$$

Long method (ditching all names):

$$
\begin{aligned}
(\lambda x.\lambda y.yxy)(5)(\lambda a.\lambda b.b(a+1))(\lambda z.z) &\triangleright_\beta (\lambda y.yxy)[x := 5](\lambda a.\lambda b.b(a+1))(\lambda z.z) \\
&= (\lambda y.y(5)(y))(\lambda a.\lambda b.b(a+1))(\lambda z.z) \\
&\triangleright_\beta (y(5)(y))[y := \lambda a.\lambda b.b(a+1)](\lambda z.z) \\
&= (\lambda a.\lambda b.b(a+1))(5)(\lambda a.\lambda b.b(a+1))(\lambda z.z) \\
&\triangleright_\beta (\lambda b.b(a+1))[a := 5](\lambda a.\lambda b.b(a+1))(\lambda z.z) \\
&= (\lambda b.b(6))(\lambda a.\lambda b.b(a+1))(\lambda z.z) \\
&\triangleright_\beta (b(6))[b := (\lambda a.\lambda b.b(a+1))](\lambda z.z) \\
&= (\lambda a.\lambda b.b(a+1))(6)(\lambda z.z) \\
&\triangleright_\beta (\lambda b.b(a+1))[a := 6](\lambda z.z) \\
&= (\lambda b.b(7))(\lambda z.z) \\
&\triangleright_\beta (b(7))[b := (\lambda z.z)] \\
&= (\lambda z.z)(7) \\
&\triangleright_\beta z[z := 7] \\
&= 7
\end{aligned}
$$

Question 4. The entire question 4 is hard.

a. The idea behind the arguments of `sum` are:

- `term` obtains the a^{th} term in the sequence to sum
- `a` is simply the position of the sequence whose term you want to obtain
- `next` gives you the next a
- `b` is the limit beyond which you will stop summing.

We need to think about what these arguments might be. We are trying to determine the total number of pins knocked down in ten frames, so we could phrase it as "summing the number of pins knocked down in ten frames". Therefore, `term` can help us decide what is the number of pins knocked down in a frame. `term` requires `a`, therefore `a` can be the index of the first roll of the frame of interest. This gives us the definition of `term(a)` as returning the number of pins knocked down in the frame starting at index `a` in `game`. `next` should simply give us the index of the first roll of the next frame, and `b` should therefore be the index of the last element in `game`, where if `a > b` we would know that our desired sum is obtained.

The explanation above results in the following solution.

```
def knocked_down(game):

    def frame_knocked_down(idx_of_first_roll):
        # the first roll of the frame of consideration is
        # game[idx_of_first_roll].
        # if game[idx_of_first_roll] is an X then we know immediately that
        # this frame is a strike. otherwise, we will have to look at the
        # second roll.
        idx_of_second_roll = idx_of_first_roll + 1
        if game[idx_of_first_roll] == 'X' or game[idx_of_second_roll] == '/':
            return 10
        return int(game[idx_of_first_roll]) + int(game[idx_of_second_roll])

    def idx_of_first_roll_of_next_frame(idx_of_first_roll_of_curr_frame):
        # we know that if the current frame at index i is just an X,
        # then the next frame starts at i + 1. Otherwise, then the frame
        # must look like 3/ or 51, therefore the next frame would start at
        # index i + 2.
        if game[idx_of_first_roll_of_curr_frame] == 'X':
            return idx_of_first_roll_of_curr_frame + 1
        return idx_of_first_roll_of_curr_frame + 2

    return sum(frame_knocked_down,
               0,
               idx_of_first_roll_of_next_frame,
               len(game) - 1)
```

4b. Here we need to understand what `accumulate` does. For brevity, let a be `accumulate`, f be `fn`, i be `init` and g be `get`. Mathematically,

$$\begin{aligned}
a(f, i, ls, g) &= f(g(l), a(f, i, ls[1:], g)) \\
&= f(g(ls), f(g(ls[1:]), a(f, i, ls[2:], g))) \\
&= f(g(ls), f(g(ls[1:]), f(g(ls[2:]), a(f, i, ls[3:], g)))) \\
&= f(g(ls), f(g(ls[1:]), f(g(ls[2:]), f(g(ls[3:]), a(f, i, ls[4:], g))))) \\
&= \ldots \\
&= f(g(ls), f(g(ls[1:]), f(g(ls[2:]), f(g(ls[3:]), f(\ldots, i)))))
\end{aligned}$$

Does this pattern seem familiar to you? This looks pretty much like summing the elements of a list recursively, given some initial value i. Suppose $g(ls) = ls[0]$. Then,

$$
\begin{aligned}
a(add, i, ls, g) &= add(g(l), a(add, i, ls[1:], g)) \\
&= add(g(ls), add(g(ls[1:]), a(add, i, ls[2:], g))) \\
&= add(g(ls), add(g(ls[1:]), add(g(ls[2:]), a(add, i, ls[3:], g)))) \\
&= add(g(ls), add(g(ls[1:]), add(g(ls[2:]), add(g(ls[3:]), a(add, i, ls[4:], g))))) \\
&= \ldots \\
&= add(g(ls), add(g(ls[1:]), add(g(ls[2:]), add(g(ls[3:]), add(\ldots, i))))) \\
&= add(ls[0], add(ls[1], add(ls[2], add(ls[3], add(\ldots, i)))))
\end{aligned}
$$

This pattern is very similar to what we need to do—to obtain the sum of the scores of the frames. However, we need to be a bit more clever with this, because we have to recurse downwards using `ls[1:]` indiscriminately, regardless of whether the frame contains one or two rolls. Therefore, we might instead choose to sum the scores of the *rolls* instead.

The following solution becomes intuitively clear.

```
def score(game):
    from operator import add

    def score_of_roll(game):
        # the roll of interest is game[0]. if we know that game[1] is a /, then
        # we simply ignore game[0] because the score of the frame
        # (game[0], game[1]) is simply score_of_roll(game[1]), which we will get
        # to in the next recursive call of accumulate.
        if game[0] == '/':
            return 20
        if game[0] == 'X':
            return 30
        if len(game) > 1 and game[1] == '/':
            return 0
        return int(game[0])

    return accumulate(add, 0, game, score_of_roll)
```

4c. Similar to 4b we can expand `fold` to see what it looks like.

$$
\begin{aligned}
fold(op, f, n) &= op(f(n), fold(op, f, n-1)) \\
&= op(f(n), op(f(n-1), fold(op, f, n-2))) \\
&= \ldots \\
&= op(f(n), op(f(n-1), op(f(n-2), op(\ldots, f(0)))))
\end{aligned}
$$

Our goal is to determine whether `game[0:3] == 'XXX'` or `game[1:4] == 'XXX'` or `game[2:5] == 'XXX'` and so on. Given this manner of expressing what we need to do, the following solution becomes intuitively clear.

```
def has_triple(game):
    op = lambda x, y: x or y

    def triple_at_index(i):
        return game[i:i + 3] == 'XXX'

    return fold(op, triple_at_index, len(game) - 3)
```

Question 5. For recursion, the base case can be the empty burger where we return 0. The subproblem of a burger can be burger[1:]. Given `burger_price(burger[1:])`, the price of the whole burger is that plus the price of the first ingredient.

```
def burger_price(burger):
    if not burger: return burger
    ingredient_price = {
        'B': 50,
        'C': 80,
        'P': 150,
        'V': 70,
        'O': 40,
        'M': 90
    }
    return ingredient_price[burger[0]] + burger_price(burger[1:])
```

We can also transform each character of the burger into its price, then reduce by summation.

```
from functools import reduce
def ingredient_price(ingredient):
    return {
            'B': 50,
            'C': 80,
            'P': 150,
            'V': 70,
            'O': 40,
            'M': 90
        }[ingredient]
def burger_price(burger):
    prices = map(ingredient_price, burger)
    # can use sum too:
    # return sum(prices)
    return reduce(lambda x, y: x + y, prices)
```

Question 6. a. `filter` is clearly not needed. The idea is, map each element in the input tuple to a single-element tuple containing the scaled element, then reduce via concatenation. Recall that the optional third argument to `reduce` is the initial value.

```
from functools import reduce
def scale(tp, n):
    # alternatively:
    # tuple(map(lambda i: (i * n,), tp))
    return reduce(lambda x, y: x + y,
                  map(lambda i: (i * n,), tp),
                  ())
```

b. Base case: empty list, return empty list; subproblem is the slice (as per usual); given the scaled slice, we simply need to 'add back' the scaled first element:

```
def scale(ls, n):
    if not ls: return []
    return [ls[0] * n] + scale(ls[1:], n)
```

Question 7. Again, `filter` is not needed. We need to somehow convert `n` into an iterable which we can map/reduce over. We can do that by converting it into a string. Then, we map each character (digit) in the string into the square of its integer equivalent, then reduce by addition.

```
from functools import reduce
def sum_square_of_digits(n):
    from operator import add
    return reduce(add, map(lambda i: int(i) ** 2, str(n)))
```

Question 8. We can use recursion. Let *aux* be an auxiliary function that helps us perform BFS. One of four things can happen during BFS:

1. frontier is empty; return false
2. current is in visited; continue with BFS and exclude current from the frontier
3. current is the destination; return true
4. add neighbours of current to frontier, add current to visited and continue with BFS.

This gives rise to the following program.

```
1 def bfs(d, src, dst):
2     def aux(frontier, visited):
3         if not frontier: return False
4         current, *remaining = frontier
5         return aux(remaining, visited) if current in visited else \
6                True if current == dst else \
7                aux(remaining + d[current], visited | {current})
8     return aux([src], set())
```

To implement a general BFS algorithm, instead of receiving the dictionary `d`, we allow our function to receive any function such that given a current node, we can obtain its traversable neighbours (the only change is in line 7):

```
1  def bfs(neighbours, src, dst):
2      def aux(frontier, visited):
3          if not frontier: return False
4          current, *remaining = frontier
5          return aux(remaining, visited) if current in visited else \
6                 True if current == dst else \
7                 aux(remaining + neighbours(current), visited | {current})
8      return aux([src], set())
```

Then, our `traversable` function from Lecture 7 and `solvable` function from Problem Set 4 can be implemented as the following:

```
1  # Lecture 7
2  def traversable(d, src, dst):
3      return bfs(lambda x: d[x], src, dst)
4
5  # Problem Set 4
6  def solvable(maze):
7      # define the neighbours function
8      def neighbours(current):
9          r, c = current
10         def is_valid(coordinates):
11         # check if coordinates is a valid and traversable cell
12             x, y = coordinates
13             return 0 <= x < len(maze) and    \
14                 0 <= y < len(maze[x]) and \
15                 maze[x][y] == 0
16         # all possible neighbours are top, bottom, left and right
17         possible_neighbours = [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]
18         # remove the invalid neighbours
19         return [i for i in possible_neighbours if is_valid(i)]
20     return bfs(neighbours, (0, 0), (len(maze) - 1, len(maze[0]) - 1))
```

Question 9. `combine` should remind you of the accumulator pattern. Since we are performing some kind of summation operation, we can allow `op` to be a function that sums the two input arguments. Then, `f(i)` should be the i^{th} term that we want to add to our accumulator.

We can rewrite our happy sum to be defined as $1 + 2(2^2) + 2(3^2) + \cdots + 2(n^2)$. Thus, we simply allow `f(i)` to return $2(i+1)^2$. However, notice that for all positive n, `f(0)` is added to our accumulator twice. Thus, we allow $f(0)$ to be 0.5. The solution below should then be self-explanatory.

```python
def combine(f, op, n):
    res = f(0)
    for i in range(n):
        res = op(res, f(i))
    return res

def happy_sum(n):
    def f(x):
        if x == 0:
            return 0.5
        return 2 * (x + 1) ** 2
    def op(x, y):
        return int(x + y)
    n = n
    return combine(f, op, n)
```

Long Programming Question. Approximating sin and cosine using the Taylor Series expansion is relatively straightforward; for each we simply need a function to compute the i^{th} term for each expansion, and use it in our approximation.

Computing both follows the same approach. In the iterative version, use the accumulator pattern. In the recursive case, the base case is simply for $n = 1$ terms, and in the recursive case, simply add the $n - 1$th term to the recursive call to the subproblem. In the map/filter/reduce case, simply transform each i in `range(n - 1)` into the i^{th} term and reduce via summation.

```python
from math import factorial
from operator import add
from functools import reduce

def sin_it(x, n):
    # function to compute the ith term for Taylor Series expansion for sin
    def term(i):
        return (-1) ** i * x ** (2 * i + 1) / factorial(2 * i + 1)
    total = 0
    for i in range(n):
        total += term(i)
    return total

def sin_rec(x, n):
    def term(i):
        return (-1) ** i * x ** (2 * i + 1) / factorial(2 * i + 1)
    # base case
    if n == 1:
        return term(0)
    # recursive case
    return term(n - 1) + sin_rec(x, n - 1)

```

```python
def sin_mfr(x, n):
    def term(i):
        return (-1) ** i * x ** (2 * i + 1) / factorial(2 * i + 1)
    return reduce(add, map(term, range(n)))

def cos_it(x, n):
    # function to compute the ith term for Taylor Series expansion for cos
    def term(i):
        return (-1) ** i / factorial(2 * i) * x ** (2 * i)
    total = 0
    for i in range(n):
        total += term(i)
    return total

def cos_rec(x, n):
    def term(i):
        return (-1) ** i / factorial(2 * i) * x ** (2 * i)
    if n == 1:
        return term(n - 1)
    return term(n - 1) + cos_rec(x, n - 1)

def cos_mfr(x, n):
    def term(i):
        return (-1) ** i / factorial(2 * i) * x ** (2 * i)
    return reduce(add, map(term, range(n)))
```

Since the only difference between each version of sin and cos is in the function that computes the i^{th} term, we simply allow that to be a parameter of the function.

```python
def sin_term(x):
    return lambda i: (-1) ** i * x ** (2 * i + 1) / factorial(2 * i + 1)

def cos_term(x):
    return lambda i: (-1) ** i / factorial(2 * i) * x ** (2 * i)

def taylor_it(x, n, term):
    total = 0
    for i in range(n):
        total += term(x)(i)
    return total

def taylor_rec(x, n, term):
    if n == 1:
        return term(x)(0)
    return term(x)(n - 1) + taylor_rec(x, n - 1, term)

def taylor_mfr(x, n, term):
    return reduce(add, map(term(x), range(n)))
```

```
20
21 sin_it = lambda x, n: taylor_it(x, n, sin_term)
22 sin_rec = lambda x, n: taylor_rec(x, n, sin_term)
23 sin_mfr = lambda x, n: taylor_mfr(x, n, sin_term)
24 cos_it = lambda x, n: taylor_it(x, n, cos_term)
25 cos_rec = lambda x, n: taylor_rec(x, n, cos_term)
26 cos_mfr = lambda x, n: taylor_mfr(x, n, cos_term)
```

– End of Problem Set 5 –