National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2017/2018

Tutorial 4
Higher-Order Functions

1. The *Composite Simpson's Rule* is a method of numerical integration. Using Composite Simpson's Rule, the integral of a function f from a to b is approximated as

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n]$$

   where n is an even integer, $h = \frac{b-a}{n}$, $y_k = f(a + kh)$ and the coefficients of y are 1 for y_0 and y_n, 2 for other even values of k and 4 for odd values of k. (Increasing n increases the accuracy of the approximation.) Define a function that takes as arguments f, a, b, and n and returns the value of the integral, computed using the above Composite Simpson's Rule. Use your function to integrate a cube between 0 and 1 (with $n = 100$ and $n = 1000$).

   Hint: A sample call with $f(x) = x^3, a = 0, b = 1, n = 100$ should be/look like this:

```
calc_integral(lambda x: x*x*x, 0, 1, 100)
```

2. Write a function `g(k)` that solves the following product using the higher-order function `fold`.

$$g(k) = \prod_{x=0}^{k} (x - (x+1)^2)$$

   Note that big-Pi (Π) notation used for product in the same way Sigma (Σ) notation is for `sum`. The code for `fold` is reproduced below for your convenience.

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

3. (a) Show that `sum` (discussed in lecture) is a special case of a still more general notion called `accumulate` that combines a collection of terms. It uses a general accumulation function, which is the argument `combiner` in the example call below:

```
accumulate(combiner, base, term, a, next, b)
```

   Accumulate takes as arguments the same term and range specifications as `sum`. In addition, it takes a two-parameter combiner function that specifies how the current term is to be combined with the accumulation of the preceding terms and a value `base` that specifies what value to use when the terms run out. The equations below show how `accumulate` works, where the combiner is denoted by $\oplus$.

$$a_1 = a,\ a_n \leq b$$

$$accumulate(\oplus, base, f, a, next, b):\ (f(a_1) \oplus (f(a_2) \oplus (\ldots \oplus (f(a_n) \oplus base)\ldots)))$$

Write the `accumulate` function and show how `sum` can be defined as a simple call to accumulate.

(b) If your `accumulate` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

4. (a) In this question, you will implement a representation of line segments in a 2D plane. Some sample executions are provided for you to test your implementations but you are **strongly encouraged** to create your own test cases. ☺

i. A point can be represented as a pair of numbers: the x coordinate and the y coordinate.

Based on this representation of a point, define a point constructor `make_point`.

Define a selector `x_point` which returns the x coordinate of a given point.

Define a selector `y_point` which returns the y coordinate of a given point.

To test your functions, you may find the following function (that prints a given point argument) useful:

```
def print_point(p):
    print("(", x_point(p), ",", y_point(p), ")")
```

This is an example of why data abstraction is useful. Because `print_point` only uses the selectors for your new compound data object, it does not need to know anything about how your object is implemented. Furthermore, there is no need to modify `print_point` when you decide to modify your object's implementation.

Sample Runs:

```
p1 = make_point(2, 3)
print(x_point(p1)) #expected printout: 2
print(y_point(p1)) #expected printout: 3
print_point(p1) #expected printout: ( 2 , 3 )
```

ii. A line segment has two endpoints. It can be represented by a pair of points: a starting point and an ending point.

Based on this representation of a line segment, define a line segment constructor `make_segment`.

Define a selector `start_segment` which returns the starting point of a given line segment.

Define a selector `end_segment` which returns the ending point of a given line segment.

Sample Runs (continued from part 4(a)i):

```
p2 = make_point(5, 7)
s = make_segment(p1, p2)
print_point(start_segment(s)) #expected printout: ( 2 , 3 )
print_point(end_segment(s)) #expected printout: ( 5 , 7 )
```

iii. Finally, using the selectors and constructors which you have defined, define a function `midpoint_segment` that takes a line segment as argument and returns its midpoint. (The midpoint of a line segment is the point whose coordinates are the averages of the coordinates of the endpoints of the line segment.)

Sample Runs (continued from 4(a)ii):

```
m = midpoint_segment(s)
print_point(m) #expected printout: ( 3.5 , 5.0 )
```

(b) Implement a representation for a rectangle in a 2D plane. In terms of your constructors and selectors, create functions that compute the perimeter and the area of a given rectangle.

Now implement an alternative representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area functions will work using either representation?