Assignment 11

一、概念题

1、什么是函数式编程？它有什么优缺点。

函数式程序设计是指把程序组织成一组数学函数，计算过程体现为基于一系列函数应用的表达式求值。

优点：代码简介，开发快速；接近自然语言，容易理解；易于“并发编程”；更少的出错率。

缺点：性能较低；不适合处理可变状态；不适合IO，也不适合GUI。

> https://zhuanlan.zhihu.com/p/81302150

言之有理即可。

2、什么是尾递归和尾递归优化?

一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。

递归调用可重用本次调用的栈空间，可以自动转换为迭代，防止栈溢出错误（无限创造栈帧导致栈内存耗尽）。

3、C++中的Filter、Map和Reduce操作各自是什么含义。

Filter：筛选集合中符合条件的元素出来构成一个新集合。
Map：对集合中每个元素进行某种操作并将结果放到一个新集合中。
Reduce：对集合中所有元素进行某个操作后得到一个结果。

4、C++是如何实现Currying操作的？并阐述一下该操作的重要性。

通过std::bind或者lambda表达式，将多个参数转换为一个参数。

参数复用，或者说是固定参数，避免重复传参；提前返回，或者说是提前确认，避免重复判断；延迟执行。

言之有理即可。

二、编程题

1、使用尾部递归实现算法来找到二叉树中最长的Z型路径长度。

```
int answer = 0;
struct Node {
  Node *left;
  Node *right;
};
void dfs(Node *node, bool toLeft, int length) {
  answer = answer > length ? answer : length;
```

```
  if (toLeft) {
    if (node->right && node->left) {
      dfs(node->right, false, length + 1);
      dfs(node->left, true, 1);
    } else if (node->right){
      dfs(node->right, false, length + 1);
    } else if (node->left) {
      dfs(node->left, true, 1);
    }
  } else {
    if (node->right && node->left) {
      dfs(node->left, true, length + 1);
      dfs(node->right, false, 1);
    } else if (node->left) {
      dfs(node->left, true, length + 1);
    } else if (node->right) {
      dfs(node->right, false, 1);
    }
  }
}
int solution(Node *root) {
  if (!root) return 0;
  dfs(root, true, 0);
  dfs(root, false, 0);
  return answer;
}
```

2、函数求导数（currying）。

```
#include <vector>
#include <functional>
#include <cmath>
double derivative(double x, double d, double (*f)(double)) {
  return (f(x) - f(x - d)) / d;
}
std::function<double(double(*)(double))> bind_derivative(double x, double d) {
  return std::bind(derivative, x, d, std::placeholders::_1);
}
std::function<std::function<double(double (*)(double))>(double)>
bind_derivative(double x) {
  return [x](double y) { return std::bind(derivative, x, y, std::placeholders::_1);
};
}
int main() {
  std::vector<double (*)(double)> funcs = {sin, cos, tan, exp, sqrt, log, log10};
  auto d1 = bind_derivative(1, 0.000001);
  auto d2 = bind_derivative(1)(0.000001);
```

```
  std::vector<double> result1, result2;
  std::transform(funcs.begin(), funcs.end(), std::back_inserter(result1), d1);
  std::transform(funcs.begin(), funcs.end(), std::back_inserter(result2), d2);
  return 0;
}
```