

**课 程 报 告**

**课程名称： 函数式编程**

**专业班级： CS1610**

**学 号： U201614795**

**姓 名： 覃 映 超**

**指导教师： 顾 琳**

**报告日期： 2019年5月6日**

**计算机科学与技术学院**

目录

[1. 课程内容梳理和和总结 1](#_Toc8144210)

[2. 实验问题分析和心得 2](#_Toc8144211)

[3. 课程学习体会和感悟 4](#_Toc8144212)

# 课程内容梳理和和总结

1. 对函数式编程的认识

函数式编程并不针对某一种特定的语言，它是一种编程思想，它将计算机运算看作是数学中函数的计算，并且尽量避免了状态以及变量的概念，这种编程思想可以运用在大部分编程语言中。

函数式编程的特点：将函数作为数据来使用。可以将函数作为参数传入另一个函数，也可以作为函数的返回值。惰性求值，在需要函数的结果时再计算。支持闭包，函数的状态可以保存在函数对象中。使用递归使得程序看起来很简洁。提高程序的并行性。

函数式编程的缺点：由于在程序中需要使用递归，这就带来一个问题就是导致栈溢出，虽然将函数转换成尾递归方式可以一定程度上减少栈开销，但数据量大了还是会出现栈溢出。

1. SML语言

上课主要是用的SML语言，学习了SML的一些基础语法和标准类型。在学习的过程中，我感觉Python语言在很多方面都与SML相似，惰性求值在Python中也有体现，Python中的filter函数就是在需要函数的结果时才会执行计算。

SML语言的基础类型主要有五种：unit, int, real, bool, string，还有三种拓展类型：lists、tuples、functions。在SML中数据类型不需要显示说明，编译器可以推导出来。

# 实验问题分析和心得

1. 实验一

实验一主要是熟悉SML语法，以及熟悉递归和尾递归。在计算列表内的整数的乘积时，使用递归进行计算，例如：

fun mult [] = 1

| mult(x::L) = x \* (mult L);

也可以将其转换为尾递归的形式，例如：

fun mult' ([],a) = a

| mult' (x::L, a) = mult' (L, x \* a);

当a为1时，mult`函数的功能与mult相同。

实验一的难度不大，了解了基础语法就很容易写出来。

1. 实验二

实验二主要是掌握list结构的操作，以及自定义二叉树结构。

List常用的操作就是“::”取出列表首元素也可以将元素添加到列表的首元素之前，还有一个常用的操作就是“@”连接运算，可以将两个list进行连接，但是@操作实际上也是用::操作实现的，即将一个list的元素不断添加到另一个list的首元素之前。

在SML中是没有tree这种类型的，因此在需要使用tree这种类型时，我们需要自己定义tree结构，tree的定义并不是唯一的。通过列表构造平衡树时，就要将列表大致分为两半。这样两棵子树的大小最多可能只差一个节点。创建平衡树的方式有三种分别是前序、中序、后序，但是使用前序的方式相对比较简单，直接取列表首元素作为根节点。了解了这些以后就可以写出创建平衡树的代码了，代码如下：

fun listToTree [] = Empty

| listToTree(x::xs) =

let

val k = length xs div 2

in

Node(x, listToTree(List.take(xs,k)), listToTree(List.drop(xs, k)))

end;

实验二比较难的就是创建二叉树的过程，其余的任务都是直接输入一棵树，对树进行操作，所以在了解树的结构之后就很容易做出来了。

1. 实验三

实验三的难度总体上比前两个实验的难度大一些，需要掌握多态类型、option类型和高阶函数的编程方法。

在SML中可以直接将函数对象作为参数传递给另一个函数，当函数体比较简单时，可以定义一个匿名函数对象，就跟lambda表达式一样，例如第一道题实现一个将整数经过函数变换的函数，如果函数变换的函数体比较简单（翻倍、求平方等）可以在调用函数的时候传递一个匿名函数：

val t1 = thenAddOne((fn x => x \* x), 3) 就可以完成。

这一次实验中还体现了惰性求值，具体细节如下：

fun mapList(f, []) = []

| mapList(f, x::L) = f(x)::mapList(f, L)

fun mapList' f =

let

fun f2 [] = []

| f2(x::L) = f(x)::f2(L)

in

f2

end

在调用mapList函数，该函数直接计算出函数结果，而调用mapList`函数时，不会直接计算函数结果，而是将这一次的函数状态保存在函数对象中，在后序过程中需要时再计算函数结果。

实验三中比较难的就是在子集中找出一个满足和为S的集合，则可以题目可以使用辅助函数找出输入集合的所有子集，然后再依次计算每个子集的和直到找出满足条件的集合，也可以取出一个元素，判断其是否在满足条件的结果集合中，如果不在则直接丢弃，不需要再考虑这个元素，具体实现如下：

fun subsetSumoption([],s) = NONE

|subsetSumoption(a::l,s) = if s=a then SOME (a::[])

else case subsetSumoption(l,s-a) of

NONE => subsetSumoption(l,s)

| SOME x => SOME (a::x)

# 课程学习体会和感悟

学习函数式编程给我的感觉就是，递归在函数式编程中很重要，然后一切皆函数的这种思想也值得学习。实验中的代码量不是很大，但是很多题目都很有意思，要是真的要学好这一门课还需要学习很多东西，因为有很多地方理解起来非常不容易。

老师上课讲得很好，讲课也很认真，也很有耐心，能把一个自己认为是显然的东西重复讲解给我们听，所以只要上课认真听课了，这些相对基础的东西就可以掌握了，掌握了基础的东西，做实验的时候也不会觉得太困难。