华中科技大学

函数式编程原理
课程报告

姓　　名：周楷
学　　号：U202115469
班　　级：CS2105
指导教师：顾琳

计算机科学与技术学院
2023 年 10 月 21 日

一、函数式语言家族成员调研

函数式语言家族中有 ML、Miranda、Haskell、OCmal、Scala、Lisp、Scheme、Lux 等成员，接下来将挑选其中几个进行介绍

1.1 ML

ML(Meta Language)由爱丁堡大学的 Robin Milner 等人在二十世纪七十年代晚期开发。

在 ML 中，函数被视为一等公民，可以像其他数据类型一样进行操作。函数可以作为参数传递给其他函数，也可以作为返回值返回。

ML 支持高阶函数，即函数可以接受其他函数作为参数或返回其他函数作为结果。这种能力使得编写灵活的、抽象的代码变得更加容易。

本次课程学习的语言正是 Standard ML 语言，它很好地代表了函数式编程语言。

1.2 Haskell

Haskell 是于 1990 年在 Miranda 语言的基础上进行标准化的，它的理论基础也是 λ 演算。Haskell 的命名来源于美国逻辑学家哈斯凯尔·加里，他在数理逻辑方面的工作很有造诣。

Haskell 是纯函数式编程语言，函数没有副作用，支持惰性求值、模式匹配、列表解析、类型类和类型多态等特性。

目前 Haskell 经历了从 Haskell 1.0 至 1.4、Haskell 98、Haskell Prime 到 Haskell 2010 这几个大的版本更迭，它可以跨平台运行，具有高性能的并发和并行能力。

目前 Haskell 语言的用户群体比较庞大，社区也很活跃，拥有一个名为 Hackage 的线上 Haskell 代码仓库，上面有很多开源的第三方库可供下载。

图 1.1 Haskell 语言标志

1.3 OCmal

OCaml 全称 Objective Caml，由 Xavier Leroy，Jérôme Vouillon，Damien Doligez，Didier Rémy 及其他人于 1996 年创立，是一个开源项目。

OCaml 的开发工具包括一个交互式的顶层解释器、字节码编译器以及最优本地代码编译器。它的一个最大特点就是具有健壮的模块化结构以及引入了面向对象的编程结构，在一定程度上提高了函数式编程语言在大型软件工程项目中的应用能力。

图 1.2 OCaml 语言标志

1.4 Scala

Scala 语言由联邦理工学院洛桑（EPFL）的 Martin Odersky 于 2001 年基于 Funnel 的工作开始设计。Funnel 是把函数式编程思想和 Petri 网相结合的一种编程语言。正因为此，Scala 语言作为一门纯粹的面向对象的语言，无缝结合了函数式编程语言的特性。

Scala 语言在函数式编程层面提供了许多关键概念的支持，比如它支持高阶函数、函数柯里化、函数嵌套、多态、匿名函数等特性。

Scala 运行于 JVM 虚拟机，加上它融合了面向对象、命令式编程和函数式编程的思想，使得它吸引了很多开发者进行尝试。它现在是一门非常热门的语言，拥有很活跃的开源社区和很多使用者。

图 1.3 Scala 语言标志

1.5 Lisp

Lisp（List Processing）由麻省理工学院的人工智能研究先驱 John McCarthy 于 1958 年创造，Lisp 的理论依据是 λ 演算。

正如 Lisp 的名字，它最初是一门表处理语言，因为表天生具有递归的性质，Lisp 采用抽象数据列表和递归符号演算的方式运行。递归是数学层面上的基本概念之一，从递归理论出发，一切可以计算的函数最终都可以划归为几种基本的递归函数的种种组合。

图 1.4 Lisp 语言标志

二、上机实验心得体会

2.1 整数列表乘积 Multx

2.1.1 解题思路

本题要求我们设计一个 Multx 函数，其要求为(* Multx: int list list * int -> int *)。

我们可以先设计一个 multx，其要求为(* multx : int list * int -> int *)，接着理由递归的思想，要完成 Multx 的设计要求，就是对其中每个 list 调用 multx 函数，最后一起返回就是正确答案。

2.1.2 代码

```
(* multx : int list * int -> int    *)
(* REQUIRES: true    *)
(* ENSURES: multx(L, a) … (* FILL IN *)  *)

fun multx ([ ], a) = a
  | multx (x :: L, a) = multx (L, x * a);

(* Multx: int list list * int -> int *)
(* REQUIRES: True               *)
(* ENSURES: Multx(R, a) evaluates to the product of integers in R times a*)

fun Multx( [ ], a) =  a
  | Multx(r::R, a) =  Multx(R,multx(r,a));
```

2.1.3 运行结果

Multx 函数能够正确返回结果。

2.1.4 性能分析

multx 函数遍历 list 中的每一个元素，因而时间复杂度为 $O(n)$，Multx 对其中的每一个 list 调用 multx 函数，因而时间复杂度为 $O(n^2)$。

2.1.5 问题与解决

这是第一个实验中的关卡，此时我还对函数式编程的模式不太理解。由于平时习惯于面向过程的编程语言，我比较擅长使用循环、条件分支等方式。但函数式编程中没有循环这种方法，因而只能使用递归来代替循环。

对于 multx 函数，我们需要递归遍历 list 中的每一个元素，并将中间结果保存在参数中。对于 Multx 函数也是相同的操作。

2.2 PrefixSum 函数

2.2.1 解题思路

本题要求我们计算一个数组的前缀和数组，分别实现一个低性能与一个高性能的函数。

对于低性能的 PrefixSum()函数，我们可以定义一个辅助函数 Sum 用于计算

一个数组的和。然后在遍历数组的过程中记录已经遍历的元素，对每一项调用辅助函数 Sum 计算已经遍历的元素，从而得到当前项的前缀和。

对于高性能的 fastPrefixSum()函数，我们可以在参数中保存当前的前缀和，对于每一项只用将该元素添加到已保存的前缀和中就可以得到当前前缀和。

2.2.2 代码

```
1.    fun PrefixSum(nums) =
2.        let
3.            fun sum([]) = 0
4.                | sum(x::xs) = sum(xs) + x
5.            fun sumHelp(added,[],sums) = sums
6.                | sumHelp(added,x::xs,sums) = sumHelp(added@[x],xs,sums@[sum(add
    ed@[x])])
7.        in
8.            sumHelp([],nums,[])
9.        end;
10.
11.   fun fastPrefixSum(nums) =
12.       let
13.           fun sumHelp(sums, [], curSum) = sums
14.               | sumHelp(sums, cur::rest, curSum) = sumHelp(sums@[cur+curSum],
    rest, curSum+cur)
15.       in
16.           sumHelp([], nums, 0)
17.       end;
```

2.2.3 运行结果

PrefixSum()函数和 fastPrefixSum()函数均能返回正确结果。

2.2.4 性能分析

对于 PrefixSum()函数，辅助函数 Sum 的时间复杂度为 O(n)，对每个遍历的元素调用了 Sum 函数，因而总的时间复杂度为 $O(n^2)$。

对于 fastPrefixSum()函数，由于在参数中保存中间结果，时间复杂度为 O(n)。

2.2.5 问题与解决

本关卡的重要点在于定义好函数内的辅助函数，通常来说我们在面向过程的语言中都是在函数内直接编写指令的，但由于函数式编程特性，所有的功能都需要由函数实现，因而就需要用到辅助函数来帮助我们完成函数功能。

2.3 binarySearch

2.3.1 解题思路

本题要求我们在将 list 转为 tree 的基础上，实现一个基于树的二分查找功

能。

由于我们得到的 tree 本身就是二分查找树，因而我们要做的就是深入这棵树，对比需要查找的数和根节点，从而逐步所有搜索范围。

2.3.2 代码

```
fun listToTree ([] : int list) : tree = Empty
  | listToTree (x::l) =
        let val (l, r) = split(l)
        in Br(listToTree(l), x, listToTree(r))
        end;
fun binarySearch (Empty : tree, _ : int) : bool = false
    | binarySearch(Br(lt,a,rt),x) =
        case Int.compare(a,x) of
            GREATER => binarySearch(lt,x)
            | EQUAL => true
            | LESS => binarySearch(rt,x);
```

2.3.3 运行结果

binarySearch()函数能正确返回结果。

2.3.4 性能分析

根据二分查找特性，binarySearch()函数的时间复杂度为 O(n)。

2.3.5 问题与解决

本关卡需要我们实现一个经典的二分查找算法。我们需要利用函数式编程的特点，利用递归不断查找，最终得到正确结果。

三、课程建议和意见

在函数式编程原理这门课程的学习中，我学到了函数式编程语言的发展和使用场景，并具体学习了 Standard ML 这一门函数式编程语言。

在课程中，我接触到了类型推导、模式匹配、多态类型、高阶函数、函数柯里化等等函数式编程的特点。在四个实验中，我加深了对函数式编程的理解，同时也让我在实践中摸索函数式编程的原理。

总而言之，我在这门课程的学习中受益匪浅，这是我第一次接触到了函数式编程的思想，与此前所学的面向过程的编程和面向对象的编程都有很大的差异。

对于今后函数式编程原理的教学，建议可以采用 JavaScript 语言进行教学，一方面 JavaScript 是比较现代的编程语言，易于理解。另一方面 JavaScript 中本身就带有很多函数式编程的思想。