函数式编程原理
课程报告

姓　　名：张文浩
学　　号：U202015084
班　　级：计科 2011 班
指导教师：顾琳

计算机科学与技术学院

2022 年 1 月 **8** 日

目 录

一、函数式语言家族成员调研

1.1 函数式编程语言综述

函数式编程（函数程序设计、泛函编程）是不同于命令式编程的一种编程范式，它将电脑运算视为函数计算，是结构化编程的一种。其主要成分是原始函数、定义函数和函数型，可以把函数作为参数，函数的结果也可以是一个函数。

1.2 函数式编程语言的发展历史

函数式编程的起源是 1936 年由阿隆佐·邱奇设计的名为 λ 演算的形式系统。在其中，函数的参数是函数，返回值也是函数。

1958 年，MIT 教授 John McCarthy 发明了一种列表处理语言 Lisp，这种语言是 λ 演算在现实生活中的实现，而且能够在冯诺依曼架构的计算机上运行。1973 年，MIT 人工智能实验室开发出 Lisp 机。

之后函数式编程迅速发展，也诞生了很多函数式编程语言。但函数式编程还是在学术和应用上失去了价值，函数式编程开始衰落。

2010 年左右 JavaScript 大爆发，由于 JavaScript 中蕴含着 λ 演算的特性，使得函数式编程又一次兴起，目前仍然在大量使用。

1.3 函数式编程语言的特点

1. 函数是一等公民：

函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

2. 没有副作用：

函数保持独立，所有功能就是返回一个新的值，没有其他行为，不会修改外部变量的值。

3. 不修改状态：

函数式编程只是返回一个新的值，不修改系统变量。在其他类型的语言中，变量往往用来保存状态。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归。

4. 引用透明：

函数的运行不依赖于外部变量或者状态，只依赖于输入的参数，如果参数相同，那么函数得到的返回值相同。

5. 只使用表达式，不使用语句：

表达式是一个单纯的运算过程，总会有返回值；语句是执行某种操作，没

有返回值。函数式编程中，每一步都是单纯的运算，都会有返回值。

6. 并行：

在函数式编程中，程序员无需对程序修改，程序就可以并发运行，程序运行期间，不会产生死锁现象。因为通过函数式编程得到的程序，在程序中不会出现某一数据被同时修改两次。

7. 代码部署热：

在完全不停止系统任何组件的情况下，达到更新相关代码的目的。对函数式的程序，所有的状态即传递给函数的参数都被保存在了堆栈上，这使得热部署轻而易举。

1.4 典型函数式编程语言：

1.纯函数式编程语言：

Haskell:

Haskell（发音为/ˈhæskəl/）是一种标准化的，通用的纯函数编程语言，有非限定性语义和强静态类型。在 Haskell 中，“函数是第一类对象”。作为一门函数编程语言，主要控制结构是函数。Haskell 语言是 1990 年在编程语言 Miranda 的基础上标准化的，并且以λ演算为基础发展而来。这也是为什么 Haskell 语言以希腊字母“λ”（Lambda）作为自己的标志。Haskell 具有“证明即程序、命题为类型”的特征。

2.非纯函数式编程语言：

强静态类型：

①ML：

ML (meta language)是一个通用的函数式编程语言，是一种非纯函数编程语言，允许副作用和指令式编程。它是由爱丁堡大学的 Robin Milner 及他人在二十世纪七十年代晚期开发的。它的语法是从 ISWIM 得到的灵感。

ML 特性有：传值调用的求值策略，头等函数，带有垃圾收集的自动内存管理，参数多态，静态类型，类型推论，代数数据类型，模式匹配和异常处理。

基于 ML 的程序设计语言有 Ocaml，F#等。

②Ocaml：

Objective Caml（OCaml）是 Caml 编程语言的主要实现，基于 ML 语言，由 Xavier Leroy，Jérôme Vouillon，Damien Doligez，Didier Rémy 及其他人于 1996 年创立。OCaml 是开放原始码项目。此项目的管理和大部分维护工作交由

INRIA。

Ocaml 将 Caml 语言在面向对象方面做了延展。

③F#：

F#是由微软发展的为微软.NET 语言提供运行环境的程序设计语言。它是基于 Ocaml 的。这是一个用于显示.NET 在不同编程语言间互通的程序设计。

F#同样支持面向对象。同时 F#已经接近成熟，支持高阶函数、柯里化、惰性求值、Continuations、模式匹配、闭包、列表处理和元编程等。

④Scala：

洛桑联邦理工学院的 Martin Odersky 于 2001 年基于 Funnel 的工作开始设计 Scala。Scala 是一门多范式的编程语言，一种类似 java 的编程语言 。

Scala 的既有函数式编程特性，又有面向对象特性。它支持高阶函数、柯里化、嵌套函数、序列解读等函数式编程特性，同时 Scala 中每个值都是对象，这也体现了其函数式编程特性。

强动态类型：

①LISP：

LISP 名称源自列表处理（List Processing）的英语缩写，由来自麻省理工学院的人工智能研究先驱约翰·麦卡锡（John McCarthy）在 1958 年基于λ演算所创造，采用抽象数据列表与递归作符号演算来衍生人工智能。

LISP 是一种通用高级计算机程序语言，长期以来垄断人工智能领域的应用。LISP 作为应用人工智能而设计的语言，是第一个声明式系内函数式程序设计语言。

特征：弱类型、动态推断。LISP 只有两种数据结构，原子（atom）和表（list）。原子为标识符形式的符号或数字的字面值，表则是由零个或多个表达式组成的序列。基本上，LISP 程序，并不需要使用一般表处理所必需的任意插入及删除操作。LISP 的语法是简洁的典型，程序代码与数据的形式完全相同——以圆括号为边界的表。

②Scheme：

Scheme 编程语言是一种 Lisp 方言，诞生于 1975 年，由 MIT 的 Gerald J. Sussman 和 Guy L. Steele Jr. 完成。它是现代两大 Lisp 方言之一。

特点：括号嵌套、语法简洁、自动内存管理、支持尾递归等。

③Clojure：

Clojure 语言的创造者是里奇·希基，是一种动态的、强类型的、寄居在 JVM 上的语言。而且是 LISP 语言的方言。

特征：Clojure 对待变化的方式以标识的概念为特征。标识是指随着时间的推移而产生的一系列状态。而状态则是指标识在某一特定时间点上的值。需要强调的是，这里的值是不可变的。由此引申，由于状态是不可变的值，任意数量的工作单位都可以在其上以并行的方式实施操作。因此，并发性就成为一道管理状态间变化的问题为此，Clojure 提供了几个可变的引用类型。每个引用类型都有其明确定义的语义用于控制状态之间的跃迁。

④R 语言：

R 语言是为数学研究工作者设计的一种数学编程语言，主要用于统计分析、绘图、数据挖掘。R 语言来自 S 语言，是 S 语言的一个变种。S 语言由 Rick Becker, John Chambers 等人在贝尔实验室开发。

特点：可以进行函数式编程，同时也支持面向对象。

二、上机实验心得体会

函数式编程与之前所学的 C 语言，C++等有很大的差别，其“函数是一等公民”的核心思想是让我感觉最新奇最有趣的地方。因为函数式编程中没有循环语法，所以所有循环都必须改成递归来实现。在这几周的实验中，我们大量编写递归函数，既熟悉了 SML 语法，又深刻地锻炼了自己的思考能力，让自己在面对递归的时候，不再感觉那么奇特而困难。

下面是让我记忆犹新的实验题目：

①第二次实验第八题：

编写函数，输入一个数组，输出其前缀和数组，要求 work 分别为$O(n^2)$和$O(n)$。

```
fun helper (a,[]) = []
  | helper(a,b::A) = (a+b)::helper(a,A)
fun PrefixSum [] = []
  | PrefixSum (a::L) = a::helper(a,PrefixSum(L))

fun fastPrefixSum [] = []
| fastPrefixSum [x] = [x]
| fastPrefixSum (x::y::L) = x::fastPrefixSum((x+y)::L)
```

对于我自己来说，第二个函数还是比较容易实现的，主要思想就是每次都将第一个元素加到第二个元素之上，这样实现的 work 为$O(n)$。

但第一个函数就稍微有一些困难，不过思想很简单，就是将遍历数组中的每个元素，将该元素加到其后数组的每一个元素之上，但当时并没有很快想出如何用递归来实现这个功能。

这里就体现了 helper 函数的用处，首先编写这样一个将元素加到数组的每个位置上的函数，然后将该函数运用到求前缀和当中，就可以很轻松的实现该功能。

通过这次实验我发现，灵活运用 helper 函数，可以通过分析函数的实现过程，在原函数中使用 helper 函数，会更容易实现某些复杂的功能，同时让函数更加简洁，更加清晰。

②第三次实验第四题：

编写函数实现最小堆的构建。

```
fun treecompare (Empty,Empty) = EQUAL
  |  treecompare (Empty,t) = LESS
  |  treecompare (t,Empty) = GREATER
  |  treecompare (Node(t0,x,t1),Node(t2,y,t3)) = Int.compare(x,y)

fun SwapDown Empty = Empty
  | SwapDown (Node(Empty,x, Empty)) = Node(Empty,x, Empty)
```

```
8.    | SwapDown (Node(Empty, x, R)) =
9.     let
10.       val Node(t1, r, t2) = R
11.    in
12.       case treecompare (Node(Empty, x, R), R) of
13.         GREATER => Node(Empty,r,SwapDown(Node(t1,x,t2)))
14.         | _ => Node(Empty, x, R)
15.    end
16.   | SwapDown (Node(L, x, Empty)) =
17.    let
18.       val Node(t1, l, t2) = L
19.    in
20.       case treecompare (Node(L, x, Empty), L) of
21.         GREATER => Node(SwapDown(Node(t1,x,t2)), l, Empty)
22.         | _ => Node(L, x, Empty)
23.    end
24.   | SwapDown (Node(Node(1l,y1,1r),x,Node(2l,y2,2r))) =
25.    case treecompare(Node(1l,y1,1r),Node(2l,y2,2r)) of
26.       GREATER =>
27.         if x>y2 then
28.           Node(Node(1l,y1,1r),y2,SwapDown(Node(2l,x,2r)))
29.         else Node(Node(1l,y1,1r),x,Node(2l,y2,2r))
30.       | _ =>
31.         if x>y1 then
32.           Node(SwapDown(Node(1l,x,1r)),y1,Node(2l,y2,2r)))
33.         else Node(Node(1l,y1,1r),x,Node(2l,y2,2r));
34.
35.  fun heapify Empty = Empty
36.    | heapify (Node(L, x, R)) = SwapDown(Node(heapify(L), x , heapif
   y(R)));
```

这个题目将一个复杂的功能拆分成了多个函数。

首先是两棵树比较大小的函数，实现这个函数比较容易。

第二个函数是将一个子树都是最小堆的树转化成一个最小堆。这个函数思想比较简单，唯一需要注意的地方就是所分出的子情况比较多，需要仔细分析认真考虑，将每一种情况都要包含到，才能正确实现功能，同时注意使用之前已经编写好的比较两树大小的函数，可以让函数更简洁。

第三个函数就是是实现构建最小堆的函数。巧妙的运用前面编写好的函数，就会十分简洁地写出该函数。这里注意，每一次调用 heapify 函数，就会调用一次 SwapDown 函数，两次 heapify，如果层数是 d，节点个数就会是 2^d 量级，运行时效率很低。引入并行处理，就会让同层节点同时处理，而层数是 d，就会极大地提高了程序运行的效率。

三、课程建议和意见

由于是第一次接触函数式编程的语言，很多语法对学生来说是很陌生的，所以建议充实一下 ppt 的内容，因为基本上所有同学都是通过 ppt 来复习的，我自己感觉 ppt 有点简略，很多内容都是一笔带过，起不到很好的复习效果。而且可以在课上或者实验课的时候多讲一些示例程序，让学生先充分了解函数式编程的语法和思想，再开始编写自己的程序。如果可以的话，也可以选择教材，直接讲述教材内容，或者根据教材内容来编写 ppt，因为教材上的内容比较充实一点，这样也会更便于课后的复习。

上课时我能很深刻地感受到老师是非常负责任的，一遍遍确定我们有没有听懂，而且在实验课的时候耐心解答我们的问题，非常感谢老师在这段时间的付出！