

函数式编程原理

课程报告

**姓 名：次天钊**

**学 号：U201816799**

**班 级：校交1801班**

**指导教师：顾琳**

**计算机科学与技术学院**

**2021 年 4月 27日**

目录

[一、函数式语言家族成员调研 2](#_Toc70541854)

[1.1 函数式编程思想简介 2](#_Toc70541855)

[1.2 函数式编程发展历史 2](#_Toc70541856)

[1.3 函数式编程语言成员简介 4](#_Toc70541857)

[1.3.1 静态类型 4](#_Toc70541858)

[1.3.2 动态类型 6](#_Toc70541859)

[二、上机实验心得体会 9](#_Toc70541860)

[2.1 心得体会 9](#_Toc70541861)

[2.1.1 ML语言的多态性 9](#_Toc70541862)

[2.1.2 ML语言的并行处理 9](#_Toc70541863)

[2.2 课程建议 11](#_Toc70541864)

一、函数式语言家族成员调研

1.1 函数式编程思想简介

函数式编程（英语：functional programming）或称函数程序设计、泛函编程，是一种编程范式，它将电脑运算视为函数运算，并且避免使用程序状态以及易变对象。其中，λ演算为该语言最重要的基础。而且，λ演算的函数可以接受函数作为输入参数和输出返回值。

在函数式编程中，函数是第一类对象，意思是说一个函数，既可以作为其它函数的输入参数值，也可以从函数中返回值，被修改或者被分配给一个变量。

1.2 函数式编程发展历史

函数式编程的**理论基础**是**阿隆佐·邱奇**在1930年代开发的**Lambda演算**，其本身是一种数学的抽象而不是编程语言。另一个组合子逻辑是比它更加古老和基础的数学根基，由Moses Schönfinkel和哈斯凯尔·柯里在1920年代和1930年代开发。两者都是为了更好的表达数学基础才被开发的。

于20世纪50年代后期，**John McCarthy**在麻省理工学院，开发了早期的函数式语言**LISP**，运行在大型IBM主机（IBM700/7000系列）上。LISP的函数是使用邱奇的lambda表示法定义的，并扩展了标签构造来允许递归函数。最开始的LISP是多范型语言，并且随着新的范型的发展，越来越多的编程风格得到了支持。后来发展出来的方言比如Scheme、Clojure，和分支语言比如Dylan和Julia等，试图简化LISP，使它围绕一个函数式核心，而Common Lisp旨在保留并更新它所替代的各种更早先LISP方言的那些范型特征。

而于1956年发明的**IPL**语言，一般被认为是**第一个基于计算机的函数式编程语言**。它是一种用于操纵符号列表的汇编式语言。它有一个生成器的概念，相当于一个接受函数作为参数的函数，并且，由于它是汇编级语言，代码可以是数据，因此IPL可以被视为具有高阶函数。但是，它在很大程度上依赖于改变列表的结构和类似的指令式编程特征。

在20世纪60年代早期，**Kenneth E. Iverson**开发了**APL** ，在他1962年出版的《A Programming Language》一书中有介绍。 APL给John Backus的FP提供了巨大的影响。在20世纪90年代早期，Iverson和Roger Hui创造了J语言。在20世纪90年代中期，以前曾与Iverson合作过的Arthur Whitney创建了K语言，后者在金融行业中与其派生出来的Q语言一起被商业化使用。

1977年John Backus在他的图灵奖颁奖演讲《可以从冯·诺依曼式的编程风格中解放出来的程序设计和函数式风格及其程序代数》中，展示了他提出的FP。他将函数式编程定义为通过“组合形式”以分层方式构建，允许“程序代数”; 在现代语言中，这意味着函数式程序应遵循复合性原理。Backus的论文推广了函数式编程的研究，虽然它强调的是函数级编程而不是现在所说的lambda演算风格。

1973年爱丁堡大学的**Robin Milner**发明了**ML语言**，它的语法受到了ISWIM的启发。同年，**David Turner**在圣安德鲁斯大学开发**SASL**语言，它基于了ISWIM的应用式子集。在1976年，Turner重新设计并重新实现它为惰性求值语言。在20世纪70年代的爱丁堡，Burstall和Darlington开发了NPL语言。NPL基于Kleene递归方程 ，并在他们的程序转换工作中首次引入。然后Burstall、MacQueen和Sannella结合了来自ML的多态类型检查，从NPL派生出了Hope语言。ML最终发展成几种语言，其中最常见的是OCaml和Standard ML。

在20世纪70年代，**Guy L. Steele和Gerald Jay Sussman**开发了**Scheme**，如有影响力的“Lambda论文集”和经典的1985年教科书《计算机程序的构造和解释》中所描述的那样。Scheme是使用词法作用域和尾调用优化的第一个Lisp方言，将函数式编程的影响力提升到更广泛的范围，让更多的编程语言社区接触到它们。

在20世纪80年代，**佩尔·马丁-洛夫**开发了**直觉类型论**（也称为构造类型论），它将函数式编程与表现为类型依赖的数学证明联系起来。这导致了交互式定理证明的新方法的产生，并影响了后续的函数式编程语言的发展。David Turner开发的惰性求值函数式语言Miranda最初出现在1985年，采用来自ML与Hope语言的概念，作为他先前所设计的SASL和KRC语言的后继者。Miranda对后来的Haskell有很强的影响，由于它是专有软件，所以Haskell社区于1987年开始达成共识，以形成函数式编程研究的开放标准，对标准的实现自1990年以来一直在进行中。

最近，它在基于CSG几何框架构建的OpenSCAD语言的参数CAD中得到了应用，虽然它无法区分左值和右值，导致了不熟悉函数式编程的用户混淆。

1.3 函数式编程语言成员简介

函数式编程可以从强类型/弱类型或者静态类型/动态类型这两个角度来对其进行分类，但我在了解后发现绝大多数的函数式编程语言均采用强类型的设计，因此以下分类采用静态类型/动态类型这两方面对一些主流的函数式编程语言进行介绍。

1.3.1 静态类型

**1.ML**

ML是由爱丁堡大学的Robin Milner及他人在二十世纪七十年代早期开发的，它的语法是从ISWIM得到的灵感。

ML特性包括：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。

今天在ML家族中有好几种语言：两种主要的方言是Standard ML和OCaml，其他的包括F#，它是针对Microsoft .NET平台的开放研究项目。ML中的思想影响了众多的语言，例如Haskell，Cyclone和Nemerle，ATS和Elm。

ML的实力大多被用于语言设计和操作（编译器、分析器、定理证明器），但是它作为通用语言也被用于生物信息和财务系统等领域。

**2.OCaml**



图1.1 OCaml语言标志

OCaml，最早称为Objective Caml，是Caml编程语言的主要实现，由Xavier Leroy，Jérôme Vouillon，Damien Doligez，Didier Rémy及其他人于1996年创立。OCaml是开放源代码项目。此项目的管理和大部分维护工作交由INRIA。

**3.F#**

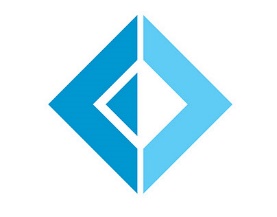


图1.2 F#语言标志

F#是由微软发展的为.NET语言提供运行环境的程序设计语言，是函数编程语言（FP，Functional Programming），函数编程语言最重要的基础是Lambda Calculus。它是基于OCaml的，而OCaml是基于ML函数编程语言。有时F#和OCaml的程序是可以交互编译的。

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。这是一个用于显示.NET在不同编程语言间互通的程序设计，可以被.NET中的任意其它代码编译和调用。

2002年微软开始由Don Syme带领研发F#，从C#，LINQ和Haskell中获取了经验，2005年推出第一个版本，2007年7月31日释出1.9.2.9版。2007年底，微软宣布F#进入产品化的阶段。

**4.Scala**



图1.3 Scala语言标志

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。Funnel是把函数式编程思想和佩特里网相结合的一种编程语言。Odersky之前工作于Java泛型和javac。Java平台的Scala于2003年底/2004年初发布。.NET平台的Scala发布于2004年6月。

Scala 2.8的特性包括重写的Scala容器库、命名参数和默认参数、包对象，以及Continuation.

1.3.2 动态类型

**1.Lisp**



图1.4 Lisp语言标志

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。

在Lisp中，数据类型的构成很简单，只有原子（atom）和表（list）两种数据结构，其中原子为标识符形式的符号或数字的字面值，而表是由零个或多个表达式组成的序列。由于使用递归的思想，每次只需要取出表头或表尾元素，因此不需要支持表的任意位置插入和删除的操作。

Lisp的语句结构也非常简单，和自然表达方式非常贴近。用圆括号把代表原子的标识符括起来形成一个列表，Lisp 程序中充满了一对对嵌套的小括号，这些嵌套的符号表达式体现了递归。例如( A1 A2 A3 A4 )，如果将它解释为数据，则表示一个四元素列表，若将它解释为代码，则表示名为A1的函数作用于A2、A3、A4这三个参数上。此外，圆括号可以嵌套，也就是LISP支持嵌套的表结构。

**2.Scheme**

Scheme最早由麻省理工学院的盖伊·史提尔二世与杰拉德·杰伊·萨斯曼在1970年代发展出来，并由两人发表的“λ论文集”推广开来。 Scheme语言与λ演算关系十分密切。小写字母“λ”是Scheme语言的标志。

Scheme的哲学是：设计计算机语言不应该进行功能的堆砌，而应该尽可能减少弱点和限制，使剩下的功能显得必要[4]。Scheme是第一个使用静态作用域的Lisp方言，也是第一个引入“干净宏”和第一类续延的编程语言。

**3.** **Clojure**



图1.5 Clojure语言标志

里奇·希基（Rich Hickey）是Clojure语言的创造者。里奇·希基开发Clojure的原因是因为他想要一款适合函数式编程的现代Lisp。该语言既需要与已创建的Java平台共生又需要有适合并发性的设计。

Clojure对待变化（change）的方式以标识（identity）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（state）。而状态则是指标识在某一特定时间点上的值（value）。需要强调的是，这里的值是不可变的（immutable）。由此引申，由于状态是不可变的值，任意数量的工作单位（workers）都可以在其上以并行（parallel）的方式实施操作。因此，并发性（concurrency）就成为一道管理状态间变化的问题［注意，这里的“变化”是指从一个状态到另外一个状态的跃迁（transition）而不是状态本身的变化（mutation）。］为此，Clojure提供了几个可变的（mutable）引用类型（reference type）。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。

**4.****Erlang**



图1.6 Erlang语言标志

Erlang是一种通用的并发程序设计语言，它由乔·阿姆斯特朗（Joe Armstrong）在瑞典电信设备制造商爱立信所辖的计算机科学研究室开发，目的是创造一种可以应付大规模并发活动的程序设计语言和运行环境。

Erlang于1987年首次被创造，直到1998年才发布开源版本，主要是因为当年对高并发的要求还不高，但是由于现在高并发的场景增多，Erlang也迎来了快速发展的阶段。

首先Erlang作为函数式编程语言，具有函数式编程语言的基本特性，在Erlang中，函数是基本单位，是第一等公民，可以参与计算，可以作为参数、作为返回值传值。几乎所有的概念都是由函数表达，所有的操作也都是由函数完成。

然后Erlang具有非常好的并发支持，得益于它的Erlang虚拟机，以及轻量级的Erlang进程，它可以支持超大规模的并发应用，而且无需依赖第三方库或是操作系统的调度。

此外，Erlang还具有热更新的性质，也就是说它支持代码在运行时被修改。这个特性对于更新和检修操作非常友好，能够最大程度的保证Erlang系统的运行，不会因为业务更新造成系统的暂停。Erlang支持热更新的基础在于它是一门动态类型的语言，只有在运行期间才进行数据类型的检查，对数据类型的绑定延迟到运行阶段完成。

二、上机实验心得体会

2.1 心得体会

1. 在熟悉安装SML/NJ开发环境的基础上，完成实验内容。在完成实验中选取2-3个感受最深的实验，谈谈在函数式编程学习和实践中的心得和体会。

2.1.1 ML语言的多态性

**第三次实验第二题：**

编写函数mapList’，要求： ① 函数类型为: (‘a -> ‘b) -> (‘a list -> ‘b list)； ② 功能为实现整数集的数学变换(如翻倍、求平方或求阶 乘)

具体代码：

1. (\*函数类型： (‘a -> ‘b) -> (‘a list -> ‘b list) \*)
2. (\*函数功能：实现整数集的数学变换(如翻倍、求平方或求阶乘)\*)
3. fun mapList' f = fn L =>
4. **case** L of [] => []
5. | x::R => (f x):: mapList' f R;

在本函数的编写中充分体现了ML语言的多态性。该函数以函数为参数，返回值仍然是一个函数，返回的函数将作为参数的函数用在列表的每一个元素上，返回一个列表。

虽然ML是强类型匹配的，但匹配完类型后如果有一些参数的类型不确定，那么这些不确定类型的参数就可以匹配成任意类型的实参，从而一个多态函数可以作用在多种类型的输入上。

2.1.2 ML语言的并行处理

**第三次实验第5题：**

一棵minheap树定义为： 1. t is Empty; 2. t is a Node(L, x, R), where R, L are minheaps and values(L), value(R) >= x (value(T)函数用于获取树T的根节点的值）

编写函数treecompare, SwapDown 和heapify：

① treecompare: tree \* tree -> order

(\* when given two trees, returns a value of type order, based on which tree has a larger value at the root node \*)

②SwapDown: tree -> tree

(\* REQUIRES the subtrees of t are both minheaps \* ENSURES swapDown(t) = if t is Empty or all of t’s immediate children are empty then \* just return t, otherwise returns a minheap which contains exactly the elements in t. \*)

③heapify : tree -> tree (\* given an arbitrary tree t, evaluates to a minheap with exactly the elements of t. \*)

**具体代码：**

2. fun treecompare (Empty,Empty)=EQUAL
3. |   treecompare(Empty,Node(t1,x,t2))=GREATER
4. |   treecompare(Node(t1,x,t2),Empty)=LESS
5. |   treecompare(Node(left1,x1,right1),Node(left2,x2,right2))=
6. **if** (x1<x2) then
7. LESS
8. **else** **if** (x1>x2) then
9. GREATER
10. **else** EQUAL;

13. fun SwapDown Empty=Empty
14. | SwapDown (Node(Empty,x,Empty))=Node(Empty,x,Empty)
15. | SwapDown (Node(left,x,right))=
17. **if** treecompare(Node(left,x,right),left)=LESS andalso treecompare(Node(left,x,right),right)=LESS then
18. Node(left,x,right)
19. **else** **if** treecompare(left,right)=LESS then
20. let
21. val Node(l,root,r)=left
22. in
23. Node(SwapDown( Node(l,x,r)),root,right)
24. end
25. **else**
26. let
27. val Node(l,root,r)=right
28. in
29. Node(left,root,SwapDown (Node(l,x,r)))
30. end;

33. fun heapify Empty =Empty
34. | heapify (Node(left,root,right))=
35. let
36. val t1=heapify(left)
37. val t2=heapify(right)
38. val t3=Node(t1,root,t2);
39. in
40. SwapDown(t3)
41. end;

根据分析易知道，函数SwapDown的work和span均为O（n）。分析函数heapify可知，该函数的每一次处理都需要递归调用两次，然后再调用一次SwapDown函数。

使用递归树来对函数heapify来进行分析：

对于一个规模为n的问题来说，递归树的第一层为1个节点，第二层为2个节点，第k层有2^k个节点。因此节点的总个数是2^n量级的，如果没有并行处理的话这种指数级的复杂度在问题规模较大时会消耗较长时间来求解。

但如果引入并行处理，递归树的每一层均可同时处理，而该递归树的层数为log(n)量级的，因此可见引入并行处理，该问题的复杂度大大降低了。

2.2 课程建议

虽然通过本课程我大致了解了函数式编程的思想，完成了老师布置的实验，但仍有一种浅尝辄止的感觉。还是觉得对一些问题的理解不够深刻，对一些语法的掌握不够熟练。希望老师在以后的课程讲授中完善上课使用的PPT，将一些问题的解决表述的更加清楚，这样可以方便同学们在课后对着PPT进行复习。

同时希望老师能够增加课程的配套练习，以加强巩固课上知识。