

函数式编程原理

课程报告

**姓 名： 路昊东**

**学 号： U202010755**

**班 级：计算机2011班**

**指导教师： 顾琳**

**计算机科学与技术学院**

**2021年12 月 25 日**

**目录**

[一、函数式语言家族成员调研 2](#_Toc15614)

[二、上机实验心得体会 8](#_Toc6221)

[三、课程建议和意见 10](#_Toc6012)

一、函数式语言家族成员调研

要谈论这个话题，我感觉可以首先从函数式语言的历史发展谈起。

搜集资料了解到，1930年代，Princeton大学的四个人Alonzo Church, Alan Turing, John von Neumann、Kurt Godel都对formal system做研究，他们对于实体世界兴趣不大，他们探讨的问题都是抽象的数学证明题。虽然各自做研究，但他们的问题确有共同点：想要回答关于计算的问题：

如果我们具有一部机器，它的运算威力无限，那么我们能用它来解决哪些问题？

这些问题能否自动解决？

哪些问题不能解决？

为什么不能解决？

如果两部机器具有不同的设计，可否具有相同的运算威力？

……

对于这些问题，Alonzo Church开发出一套formal system，名为lambda calculus。这个系统本质上是一个编程语言，为一部“想象中的机器”所设计的语言。lambda calculus的函数可以接受函数当作输入和输出。这样的函数用希腊字母λ当作识别，所以这个语言才名为lambda calculuse 。利用lambda calculus, Alonzo能够回答上述的许多问题，提出最终的答案。

Alan Turing也在做类似的研究，开发出不同的系统，就是大名鼎鼎的Turing machine，他得到的结论和Alonzo Church类似。后来证实Turing machine和lambda calculus的威力一样强大。

在信息科学的研究领域，资金充裕的美国军方一直都是很重要的推手之一。二次世界大战时，IBM为了帮美国军方解决弹道计算的问题，建立了知名的Mark I计算机。1949年EDVAC计算机诞生，这是第一部采用von Neumann架构的计算机，也是Turning Machine的真实版本。Turing machine领先lambda calculus做出实体机器。

1958年，对lambda calculus相当感兴趣的MIT教授John McCarthy设计出Lisp语言，Lisp实践了lambda calculus，让lambda calculus可以在von Neumann计算机上执行!大家开始注意到Lisp的威力。1973年，MIT的人工智能实验室开发出所谓的Lisp machine硬件，等于是将lambda calculus的机器实践出来了!

函数式编程长期以来没有出现在主流的商业软件世界。欠缺主流语言的支持，函数式编程只能偏安一隅，躲在学术界。而学术界欠缺书籍文章浅显且正确的介绍，这就是为何函数式编程会依然躲在象牙塔中的原因。

一直以来，真正让FP无法被接受的原因可能是“执行效率”。传统上，函数式编程语言的效率确实比命令式编程语言来得差，这在商业系统上是不能忍受的。命令式语言让我们用贴近冯纽曼架构机器的方式写程序，比较低阶，所以效率会比较高。而函数式编程语言却是使用较高阶的数学抽象，所以效率比较低。

但是这个原因却有了变化。过去这十多年， 我们有了新的衡量标准:“简单”“快速开发”比其它因素都更重要，因为现在软件的复杂度已经到了我们无法忍受的地步了，而IT产业的竞争也比以往激烈许多。想要“简单”、“快速开发”，就要用比较高阶的抽象，因此函数式编程比命令式编程更适合现在的开发环境。这些年来硬件的进步，让我们对于函数式编程的效率不再是大问题;甚至由于编译技术的进步，函数式编程语言的执行速度，现在也已经不再令人难以忍受。另外，多CPU、多核心、超线程(HT)的硬件架构普及，以及分布式运算的流行，几乎就是专为滋养FP繁殖而打造的环境。

谈论完历史，基于一定的了解基础，按照一般思路，可以从强类型/弱类型，静态类型/动态类型这两个维度来对函数式编程语言进行分类。

但我经过了解之后，发现绝大多数的主流函数式编程语言均采用强类型的设计。弱类型的函数式编程语言不具有主流地位，例如法国数学家David Madore于1999年设计的Unlambda语言，由s、k、i三个符号（组合子）完成基本操作，语言形式非常有趣，但实际应用中很少使用。

下面主要按照静态类型和动态类型对它们进行分类介绍。同一类型中按照诞生年份的前后进行排序。

**静态类型的函数式编程语言**

静态类型的语言指的是在运行之前的编译阶段就进行类型检查，并确定变量的数据类型的一类语言。静态类型的较为主流的函数式编程语言如下。

1. **ML**

ML（Meta Language：元语言），是一个[函数式](https://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80" \o "函数式编程语言)、[指令式](https://zh.wikipedia.org/wiki/%E6%8C%87%E4%BB%A4%E5%BC%8F%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80" \o "指令式编程语言)的[通用](https://zh.wikipedia.org/wiki/%E9%80%9A%E7%94%A8%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80" \o "通用编程语言)的[编程语言](https://zh.wikipedia.org/wiki/%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80" \o "编程语言)，它著称于使用了[多态](https://zh.wikipedia.org/wiki/%E5%A4%9A%E6%80%81_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)" \o "多态 (计算机科学))的[Hindley–Milner类型推论](https://zh.wikipedia.org/wiki/%E7%B1%BB%E5%9E%8B%E6%8E%A8%E8%AE%BA" \l "Hindley%E2%80%93Milner_%E7%B1%BB%E5%9E%8B%E6%8E%A8%E8%AE%BA%E7%AE%97%E6%B3%95" \o "类型推论)。由爱丁堡大学的Robin Milner等人在二十世纪七十年代晚期开发。

ML提供了对函数实际参数的[模式匹配](https://zh.wikipedia.org/wiki/%E6%A8%A1%E5%BC%8F%E5%8C%B9%E9%85%8D" \o "模式匹配)、[垃圾回收](https://zh.wikipedia.org/wiki/%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6_(%E8%A8%88%E7%AE%97%E6%A9%9F%E7%A7%91%E5%AD%B8)" \o "垃圾回收 (计算机科学))、[指令式编程](https://zh.wikipedia.org/wiki/%E6%8C%87%E4%BB%A4%E5%BC%8F%E7%BC%96%E7%A8%8B" \o "指令式编程)、[传值调用](https://zh.wikipedia.org/wiki/%E6%B1%82%E5%80%BC%E7%AD%96%E7%95%A5" \l "%E4%BC%A0%E5%80%BC%E8%B0%83%E7%94%A8" \o "求值策略)和[柯里化](https://zh.wikipedia.org/wiki/%E6%9F%AF%E9%87%8C%E5%8C%96" \o "柯里化)。它被大量的用于编程语言研究之中，并且是全面规定了的和使用[形式语义](https://zh.wikipedia.org/wiki/%E5%BD%A2%E5%BC%8F%E8%AF%AD%E4%B9%89%E5%AD%A6" \o "形式语义学)验证了的少数语言之一。它的类型和模式匹配使得它非常适合并且经常用于在其他形式语言上进行操作，比如在[编译器构造](https://zh.wikipedia.org/wiki/%E7%BC%96%E8%AF%91%E5%99%A8" \o "编译器)、[自动化定理证明](https://zh.wikipedia.org/wiki/%E8%87%AA%E5%8B%95%E5%8C%96%E5%AE%9A%E7%90%86%E8%AD%89%E6%98%8E" \o "自动化定理证明)和[形式验证](https://zh.wikipedia.org/wiki/%E5%BD%A2%E5%BC%8F%E9%AA%8C%E8%AF%81" \o "形式验证)中。

亲身体验发现，ML具有非常好的灵活性，表达式的值可以是函数，这个值可以作为其他函数的参数，也可以作为其他函数的返回值。这个特点在本次课程学习Standard ML语言时，尤其在学习高阶、多态和柯里化等特性时体现非常明显。

目前ML语言家族具有两个较大的分支，一个是我们课程中所学的Standard ML（标准ML），另一个是Caml。

本次课程中使用的是Standard ML最著名的编译器Standard ML New Jersey（SML/NJ）。Caml的具体实现是Ocaml语言。

1. **Miranda**

Miranda是由英国学者大卫·特纳（David Turner）所设计的纯函数式编程语言，其同样具有惰性求值特点，并遵循函数式编程语言的全部要求。Miranda语言于1985年发表，配套发布了一个C语言编写的编译器，可以在Linux、Unix系统上运行，之后再1987年和1989年发布了更新。

这门语言部分采用了来自ML语言的设计，并为后续发展出的Haskell语言提供了很多新的理念。

1. **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。值得一提的是Haskell的标志，构成一个字母λ的样子，就是为了致敬λ演算这一套基于数学的形式系统。

Haskell和前面提到的Miranda语言一样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，现在官方网站上指定下载的编译器是GHC，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。

1. **OCaml**

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的标志是一只骆驼。

就像它的名字，Objective Caml是Caml语言在面向对象方面拓展之后的产物，Caml是前文中ML的一个主要分支。

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。

1. **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。

1. **F#**

F#语言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。自2002年开始研发，2005年发布了第一个版本，2007年底正式从研发专案转移至产品部门，并决定将F#置入Visual Studio 2010。

由于F#语言是基于上文中的OCaml语言的，而OCaml语言又是ML语言的一大分支Caml语言的具体实现，因此F#也是一门非纯粹的函数式编程语言，同样延续了OCaml一样对面向对象特性的良好支持。

目前Microsoft计划将慢慢整合F#语言到.NET平台中，并让它在.NET平台中提供计算支持。由于函数式编程在并行和多线程处理时的良好性能表现，F#可能在程序核心数据的多线程处理中得到很好的利用。

**动态类型的函数式编程语言**

动态类型的语言指的是在运行期间才进行数据类型检查，把类型绑定的过程延迟到运行阶段的一类语言。动态类型的较为主流的函数式编程语言如下。

1. **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。

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。

1. **Scheme**

Scheme语言由麻省理工大学的Gerald J. Sussman 和 Guy L. Steele Jr. 创造，诞生于1975年。

这门语言是现代化的Lisp语言的一种方言，它的特点是精简，以一个非常小的语言核心作为标准，再以语法糖的形式来对语言本身进行拓展。

Scheme语言除了具有函数式编程语言的基本特点外，还具有的特点有：继承于Lisp的括号嵌套、自动内存管理、支持尾递归、可以作为脚本语言进行嵌入。它所支持的数据结构在Lisp的基础上有所拓展，支持数字、字符、字符串、布尔值、列表、数组、函数这些广泛的数据类型。相比于Lisp只支持原子和列表而言有了很大的扩充。

1. **Clojure**

Clojure是一种运行在Java平台上的 Lisp 方言，尽管Lisp具有强大的表达能力和强悍的功能，但是由于它的表达形式过于规范化和数学化，难以应用到实际场合之中。Clojure语言的出现就是为了改变这一点。Clojure借助了JVM即Java虚拟机的支持，使得在任何JVM平台上都可以很方便地使用Clojure语言来实现Lisp的一些强大的特性和功能。

Clojure和Lisp一样是一门动态类型的语言，它的语法和Lisp非常接近，在JVM平台上运行的时候，会被编译为JVM的字节码进行运算。由于运行在Java虚拟机上，因此Clojure语言在保持了函数式语言主要特点的同时，还可以方便地调用Java类库的API，大大提高了实际可用性。

1. **Erlang**

Erlang是一种通用的面向并发的编程语言，它由瑞典电信设备制造商爱立信所辖的CS-Lab开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境。

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。

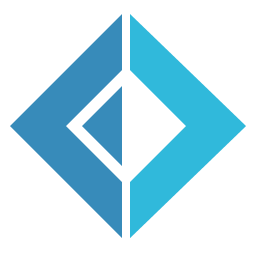
此外，Erlang还具有热更新的性质，也就是说它支持代码在运行时被修改。这个特性对于更新和检修操作非常友好，能够最大程度的保证Erlang系统的运行，不会因为业务更新造成系统的暂停。Erlang支持热更新的基础在于它是一门动态类型的语言，只有在运行期间才进行数据类型的检查，对数据类型的绑定延迟到运行阶段完成。

1. **R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，可以进行函数式编程， 同时也支持面向对象。

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。

**图例：** Haskell语言、OCaml语言、Scala语言、 F#语言、 Lisp语言、Clojure语言、Erlang语言标志如下：

二、上机实验心得体会

由于之前对Rust语言有过学习，了解过FP的一些思想，也看过一些Haskell的语法和代码，所以在整个课程中，感觉比较熟悉，也相当于是巩固了之前所学同时有了新的拓展。

而且本课程中，基本上都是使用ML语言的基本语法和ML语言自带的数据类型类完成一些简单的操作，解决一些抽象一点的问题。题目函数几乎没有多少关于函数复杂度的限制，所以我课下拓展了一些有关的知识，也算是颇有收获。

经过四次上机课以及平时的学习，关于FP思想，我的理解大概是这样的：给定x，当作输入（即x为输入值的命名），然后，将一个函数（方法）作用到这个x上，得到一个中间结果，命名为y1，函数中x为自变量（类似数学中的函数定义和运用）。然后将y1作为输入，在下一个函数中调用，得到下一个中间结果并将这个结果命名为y2……直到最后得到我们需要的结果y。因为x、y1、y2、y3等等都只是这些结果的一个命名，所以它们中的每一个的内容都不能被修改。FP编程，就是实现从x到输出y中间所有的函数。

对于四次实验题而言，第一次实验相当于是对概念有个基本的了解，第四次实验相当于是对一些题目的补充。

至于实验二，相当于是一次过渡，让我们仔细体验FP中无循环， 所有循环都必须改成递归来实现的特点，比如第二题——用归纳法证明函数的正确性、第七题——编写函数reverse和reverse'、还有有些难度的第八题——前缀和数组的实现，这里要求W分别为O(n^2)和O(n)。第一个函数PrefixSum就是将遍历数组，将每次遍历的元素加到其后数组的每一个元素之上。这里根据提示，我使用了帮助函数PrefixSumHelp，将元素加到数组的每个位置上，比较轻松实现了功能。第二个函数fastPrefixSum，如果每次都将第一个元素加到第二个元素之上，W也满足了要求。递归，确实是一件美事！

最有难度也或者说综合程度最强的是第三次实验，最小堆的编写，其中关于SwapDown和heapify函数时间复杂度的要求的确让我遭受了一些“掉头发”的“磨难”。

首先是treecompare 函数，用来实现两棵树大小比较，写起来比较简单。第二个函数是SwapDown，将一个子树都是最小堆的树转化成一个最小堆。这里子情况还是很多的，只有将每一种情况都考虑到，才能正确实现功能。当然，SwapDown中调用treecompare 函数能简化不少步骤。第三个函数就是是实现构建最小堆的函数heapify。同样的，这里需要调用之前写好的SwapDown函数，能够做到对于函数的简化。这里需要思考的地方就是对于效率的考虑。每调用一次heapify 函数，就会调用一次SwapDown 函数，那如果层数为h，两次heapify，节点个数就会是2^h的 量级，效率很低！此时要并行处理，即同层节点同时处理，时间复杂度自然就降低了。

结合自己课余时间的学习，我深入体会了函数**first-class**的地位（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，不同于second-class和third-class）、**多态类型**（一个类型模式，用一个具体的类型去匹配这样一个类型模式，就可以将多态的类型模式实例化为一个具体的类型，在standard ML语法中，多态类型在类型推导中使用单引号+小写字母的形式来代指。具体而言，在类型推导中出现的第一个多态类型为’a，第二个为’b，以此类推……FP中引入多态类型，可以避免写多余的代码，同时便于维护, 因为不需要对每一种类型都编写一个对应的函数，只需要利用多态类型，在实际匹配时将多态类型实例化为所需的数据类型即可，需要维护的函数也就只有一个，不需要过多考虑类型问题）、**高阶函数**（使用另一个函数作为其输入参数，也可以返回一个函数。如果说多态特性是为了简化多个类型的相同操作，那么高阶函数特性则是简化了多个参数的函数操作，也简化了同类型批量数据的不同函数操作。有了高阶函数之后，可以通过内部的一个函数把所有的参数处理完之后得到一个中间结果，再把这个中间结果作为变量传递给上一级函数）、**函数柯里化**（当一个函数面临接收的参数数量不止一个时，对多个参数有两种处理方法，一种是把多个参数当成一个元组，第二种则是接收第一个参数，然后随即返回一个函数，这个函数可以接收第二个参数，以此类推直到所有参数被接收。第二种方式的函数就是柯里函数，柯里化也就是把接收多参数的函数转变为接收单一参数（第一个参数），并返回一个能够接收后续参数的函数的技术）、**无副作用、“无状态”和递归**（FP天生就具有递归特性）等FP的特点，也通过实验对这些特点进行了实践巩固。

当然，关于函数式的路还很长，和命令式编程相比，FP显然更具有魅力，在高并发等等场景中有着更加独特的魅力和作用。我将继续学习，继续实践！

三、课程建议和意见

SML终究还是一门编程语言，思想重要，代码实现也重要，开始只是理解了思想，写实验时我真正使用了SML语法就确实感觉到了难度。还是希望老师在有限的课时中，除了给我们介绍思想，还可以在PPT中增加一些代码量，这样我们课下根据PPT复习知识点的时候也会更加高效有效，或者老师可以在课堂中多进行一些演示。

另外，这门课感觉是因为课时少的缘故，对很多内容以及FP中很多优秀的特性都没来得及介绍或者仅仅只是浅尝辄止，上机实验也稍微偏简单一点，不过题目还是很有代表性的。希望老师可以稍微增加一下实验难度或者增加一下课时，毕竟至少我是真的真的感觉函数式编程是一门很重要的、很有用的工具。

最后还是非常感谢老师在这段时间的辛勤付出！无论是课堂上还是实验课都十分耐心地解答我们的问题，十分感谢！