

函数式编程原理

课程报告

**姓 名： 彭子晨**

**学 号： U201814755**

**班 级： CS1804**

**指导教师： 顾琳**

**计算机科学与技术学院**

**2021年 4月 26 日**

一、函数式语言家族成员调研

1. **发展历史**

函数式编程（Functional Programming）或称函数程序设计、泛函编程，是一种[编程范式](https://zh.wikipedia.org/wiki/%E7%BC%96%E7%A8%8B%E8%8C%83%E5%BC%8F)，它将[电脑运算](https://zh.wikipedia.org/wiki/%E9%9B%BB%E8%85%A6%E9%81%8B%E7%AE%97)视为[函数](https://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B0)运算，并且避免使用程序[状态](https://zh.wikipedia.org/w/index.php?title=%E7%8A%B6%E6%80%81_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)&action=edit&redlink=1)以及[易变对象](https://zh.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E8%AE%8A%E7%89%A9%E4%BB%B6)。其中，[λ演算](https://zh.wikipedia.org/wiki/%CE%9B%E6%BC%94%E7%AE%97)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。比起[指令式编程](https://zh.wikipedia.org/wiki/%E6%8C%87%E4%BB%A4%E5%BC%8F%E7%B7%A8%E7%A8%8B)，函数式编程更加强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而不是设计一个复杂的执行过程。

函数式编程的理论基础是[阿隆佐·邱奇](https://zh.wikipedia.org/wiki/%E9%98%BF%E9%9A%86%E4%BD%90%C2%B7%E9%82%B1%E5%A5%87" \o "阿隆佐·邱奇)在1930年代开发的[Lambda演算](https://zh.wikipedia.org/wiki/Lambda%E6%BC%94%E7%AE%97" \o "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](https://zh.wikipedia.org/wiki/%E7%BB%84%E5%90%88%E5%AD%90%E9%80%BB%E8%BE%91)是比它更加古老和基础的数学根基，由[Moses Schönfinkel](https://zh.wikipedia.org/w/index.php?title=Moses_Sch%C3%B6nfinkel&action=edit&redlink=1)和[哈斯凯尔·柯里](https://zh.wikipedia.org/wiki/%E5%93%88%E6%96%AF%E5%87%AF%E5%B0%94%C2%B7%E6%9F%AF%E9%87%8C" \o "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](https://zh.wikipedia.org/wiki/%E6%95%B0%E5%AD%A6%E5%9F%BA%E7%A1%80)才被开发的。

于20世纪50年代后期，[John McCarthy](https://zh.wikipedia.org/wiki/John_McCarthy_(computer_scientist)" \o "John McCarthy (computer scientist))在[麻省理工学院](https://zh.wikipedia.org/wiki/%E9%BA%BB%E7%9C%81%E7%90%86%E5%B7%A5%E5%AD%A6%E9%99%A2" \o "麻省理工学院)，开发了早期的函数式语言[LISP](https://zh.wikipedia.org/wiki/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](https://zh.wikipedia.org/wiki/%E9%80%92%E5%BD%92_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)" \o "递归 (计算机科学))函数。最开始的LISP是多[范型](https://zh.wikipedia.org/wiki/%E7%BC%96%E7%A8%8B%E8%8C%83%E5%9E%8B" \o "编程范型)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](https://zh.wikipedia.org/wiki/Scheme)、[Clojure](https://zh.wikipedia.org/wiki/Clojure" \o "Clojure)，和分支语言比如[Dylan](https://zh.wikipedia.org/wiki/Dylan_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Dylan (编程语言))和[Julia](https://zh.wikipedia.org/wiki/Julia_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Julia (编程语言))等，试图简化LISP，使它围绕一个函数式核心，而[Common Lisp](https://zh.wikipedia.org/wiki/Common_Lisp" \o "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](https://zh.wikipedia.org/wiki/%E4%BF%A1%E6%81%AF%E5%A4%84%E7%90%86%E8%AF%AD%E8%A8%80" \o "信息处理语言)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](https://zh.wikipedia.org/wiki/%E9%AB%98%E9%98%B6%E5%87%BD%E6%95%B0" \o "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](https://zh.wikipedia.org/wiki/Kenneth_E._Iverson" \o "Kenneth E. Iverson)开发了[APL](https://zh.wikipedia.org/wiki/APL%E8%AF%AD%E8%A8%80" \o "APL语言) ，在他1962年出版的《A Programming Language》一书中有介绍。 APL给John Backus的[FP](https://zh.wikipedia.org/wiki/FP_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "FP (编程语言))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](https://zh.wikipedia.org/wiki/J%E8%AF%AD%E8%A8%80" \o "J语言)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](https://zh.wikipedia.org/wiki/K_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "K (编程语言))语言，后者在金融行业中与其派生出来的[Q](https://zh.wikipedia.org/w/index.php?title=Q_(%E9%98%B5%E5%88%97%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)&action=edit&redlink=1)语言一起被商业化使用。

1977年[John Backus](https://zh.wikipedia.org/wiki/John_Backus" \o "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](https://zh.wikipedia.org/wiki/FP_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "FP (编程语言))。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](https://zh.wikipedia.org/wiki/%E5%A4%8D%E5%90%88%E6%80%A7%E5%8E%9F%E7%90%86" \o "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](https://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B0%E7%BA%A7%E7%BC%96%E7%A8%8B" \o "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](https://zh.wikipedia.org/wiki/%E7%88%B1%E4%B8%81%E5%A0%A1%E5%A4%A7%E5%AD%A6" \o "爱丁堡大学)的[Robin Milner](https://zh.wikipedia.org/wiki/Robin_Milner)发明了[ML](https://zh.wikipedia.org/wiki/ML%E8%AF%AD%E8%A8%80" \o "ML语言)语言，它的语法受到了[ISWIM](https://zh.wikipedia.org/wiki/ISWIM" \o "ISWIM)的启发。同年，David Turner在[圣安德鲁斯大学](https://zh.wikipedia.org/wiki/%E5%9C%A3%E5%AE%89%E5%BE%B7%E9%B2%81%E6%96%AF%E5%A4%A7%E5%AD%A6" \o "圣安德鲁斯大学)开发[SASL](https://zh.wikipedia.org/wiki/SASL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80))语言，它基于了[ISWIM](https://zh.wikipedia.org/wiki/ISWIM" \o "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](https://zh.wikipedia.org/wiki/%E6%83%B0%E6%80%A7%E6%B1%82%E5%80%BC" \o "惰性求值)语言。在20世纪70年代的爱丁堡，Burstall和Darlington开发了[NPL](https://zh.wikipedia.org/wiki/NPL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "NPL (编程语言))语言。NPL基于[Kleene递归方程](https://zh.wikipedia.org/w/index.php?title=Kleene%E9%80%92%E5%BD%92%E6%96%B9%E7%A8%8B&action=edit&redlink=1) ，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](https://zh.wikipedia.org/wiki/NPL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "NPL (编程语言))派生出了[Hope](https://zh.wikipedia.org/wiki/Hope_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Hope (编程语言))语言。ML最终发展成几种语言，其中最常见的是[OCaml](https://zh.wikipedia.org/wiki/OCaml" \o "OCaml)和[Standard ML](https://zh.wikipedia.org/wiki/Standard_ML" \o "Standard ML)。

在20世纪70年代，[Guy L. Steele](https://zh.wikipedia.org/wiki/Guy_L._Steele" \o "Guy L. Steele)和[Gerald Jay Sussman](https://zh.wikipedia.org/wiki/Gerald_Jay_Sussman" \o "Gerald Jay Sussman)开发了[Scheme](https://zh.wikipedia.org/wiki/Scheme" \o "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](https://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F%E7%9A%84%E6%9E%84%E9%80%A0%E5%92%8C%E8%A7%A3%E9%87%8A" \o "计算机程序的构造和解释)》中所描述的那样。[Scheme](https://zh.wikipedia.org/wiki/Scheme)是使用[词法作用域](https://zh.wikipedia.org/wiki/%E8%AF%8D%E6%B3%95%E4%BD%9C%E7%94%A8%E5%9F%9F" \o "词法作用域)和[尾调用优化](https://zh.wikipedia.org/wiki/%E5%B0%BE%E8%B0%83%E7%94%A8)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](https://zh.wikipedia.org/wiki/%E4%BD%A9%E5%B0%94%C2%B7%E9%A9%AC%E4%B8%81-%E6%B4%9B%E5%A4%AB" \o "佩尔·马丁-洛夫)开发了[直觉类型论](https://zh.wikipedia.org/wiki/%E7%9B%B4%E8%A7%89%E7%B1%BB%E5%9E%8B%E8%AE%BA)（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](https://zh.wikipedia.org/w/index.php?title=%E4%BA%A4%E4%BA%92%E5%BC%8F%E5%AE%9A%E7%90%86%E8%AF%81%E6%98%8E&action=edit&redlink=1)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](https://zh.wikipedia.org/wiki/Miranda_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Miranda (编程语言))最初出现在1985年，采用来自[ML](https://zh.wikipedia.org/wiki/ML%E8%AF%AD%E8%A8%80" \o "ML语言)与[Hope](https://zh.wikipedia.org/wiki/Hope_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Hope (编程语言))语言的概念，作为他先前所设计的[SASL](https://zh.wikipedia.org/wiki/SASL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "SASL (编程语言))和[KRC](https://zh.wikipedia.org/wiki/%E8%82%AF%E7%89%B9%E9%80%92%E5%BD%92%E8%AE%A1%E7%AE%97%E5%99%A8" \o "肯特递归计算器)语言的后继者。Miranda对后来的[Haskell](https://zh.wikipedia.org/wiki/Haskell" \o "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. **语言分类**

典型的函数式编程语言分类如下：

1. 纯函数式编程语言

[纯函数式编程](https://zh.wikipedia.org/wiki/%E7%BA%AF%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B)语言通常不允许直接使用程序[状态](https://zh.wikipedia.org/w/index.php?title=%E7%8A%B6%E6%80%81_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)&action=edit&redlink=1)以及[易变对象](https://zh.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E8%AE%8A%E7%89%A9%E4%BB%B6)：

* + - [Miranda](https://zh.wikipedia.org/wiki/Miranda_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80))
    - [Clean](https://zh.wikipedia.org/wiki/Clean_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80))
    - [Haskell](https://zh.wikipedia.org/wiki/Haskell)

1. 非纯函数式编程语言

静态类型

* + - [ML](https://zh.wikipedia.org/wiki/ML%E8%AA%9E%E8%A8%80)
    - [OCaml](https://zh.wikipedia.org/wiki/OCaml)
    - [F#](https://zh.wikipedia.org/wiki/F%E2%99%AF)
    - [Scala](https://zh.wikipedia.org/wiki/Scala)
    - Typed Racket

动态类型

* + - [Lisp](https://zh.wikipedia.org/wiki/Lisp)
    - [Scheme](https://zh.wikipedia.org/wiki/Scheme)
    - [Clojure](https://zh.wikipedia.org/wiki/Clojure)
    - [Racket](https://zh.wikipedia.org/wiki/Racket)
    - [LOGO](https://zh.wikipedia.org/wiki/Logo_(%E7%A8%8B%E5%BA%8F%E8%AF%AD%E8%A8%80))
    - [Erlang](https://zh.wikipedia.org/wiki/Erlang)
    - [Wolfram语言](https://zh.wikipedia.org/wiki/Wolfram%E8%AF%AD%E8%A8%80)
    - [R](https://zh.wikipedia.org/wiki/R%E8%AA%9E%E8%A8%80)

1. 其他函数式编程语言
   * + [APL](https://zh.wikipedia.org/wiki/APL%E8%AA%9E%E8%A8%80)/[J](https://zh.wikipedia.org/wiki/J%E8%AF%AD%E8%A8%80)
     + [XSLT](https://zh.wikipedia.org/wiki/XSL_Transformations)

二、上机实验心得体会

1. **实验心得与体会**

在本次课程实验中，我印象最深的主要是lab2和lab3。

lab2：

在写树的时候，老师上课讲到了split方法，而我在实际实验的时候，按自己的思路利用了之前学习的函数式知识实现了树的建立。利用let in end建造树，其中有用到两个自定义函数：

indexN函数：接收参数（L:int list, i:int）用于获得数组L中L[i]的值；

get函数：接收参数（L:int list,l:int,r:int）用于获得数组L中[L[l],L[l+1],…,L[r-1]]也就是下标范围在[l,r)的子数组；

通过这两个自定义函数可以轻松取到数组的L[mid]值，然后通过get分别获得左边和右边的数组，进行递归建树。这次实验过程算是我第一次用函数式编程的思想去实现一个自己之前学过的数据结构，并且自己完整地设计生成函数以及辅助函数。也让我真正体会到了函数式编程的特色和魅力。

lab3：

lab3给我印象最深的是第5个实验写最小堆。在最开始写SwapDown函数的时候，一直陷入思维误区，没有想清楚。最开始的思路是在let里让左边的堆和右边的堆分别SwapDown，最后对整棵树SwapDown，然后发现自己漏看了题目的条件，左右本身就是最小堆了，才重新改换思路。最后总结的这个函数的思路如下：

在左右已经是最小堆的情况下，只要确保根节点、左子树根节点、右子树根节点这三个之中最小的一个值在根节点位置即可。所以在let中只需要获得左右节点的最小值v，然后和根节点x的value比较，如果value(x)<v，那么这个最小堆就已经成立了；反之如果value(x)>v，则需要节点x和左/右节点中小的那个进行交换操作。同时被交换的左/右子树不一定能够保持最小堆的特性，应该对其再进行SwapDown操作。

SwapDown的work和span分析都是O(D)，D表示子树的高度，这是由于无论串行还是并行，SwapDown的操作就是从根节点到叶子节点的一条线，代表了树的高度，没有能并行的地方。在这里我最初的分析是O(logn)，n表示节点个数，不过老师的指点提醒了我，输入并未要求是二叉平衡树，所以D≠logn。

解决了SwapDown之后heapify就很简单了，只要使用递归的操作使左子树为最小堆、右子树为最小堆，再对整体使用一个SwapDown就可。递归公式如下所示：

heapify(Node(left,x,right)) = SwapDown(Node(heapify(left),x,heapify(right)))

SwapDown的work和span分析则是对每个节点都用了一次SwapDown，所以对N个节点的堆来说work = O(N\*D),而span是指各层并行处理，也就是span=O(D\*D)。

1. **感想和建议**

通过这次实验，我更深入地理解了函数式编程的思想。原本对减少副作用、数学方式等等理论词汇还有些模糊不清的理解，但在实操之后就慢慢理解了它的思想，并学会了主动利用这种思想去解决一些编程上的问题。虽然函数式编程语言本身没有太多人使用，但是函数式编程的思想是可以学习并在实际操作中运用的。

实验检查过程中老师十分认真负责，会一边看我运行案例解说，并引导我发现自己错误的地方。我认为实验课时和实验内容的安排也比较适当，不会特别紧迫，也不会过于宽松。从我的体验来说，我想提的建议是，在前两个实验中会有PPT上课堂问题的回答，也需要自己填写答案检查，但是由于PPT上另开文本框答题不是特别方便，位置也不太够，而且检查起来也比较耗时，后面的同学往往要等上很久。我个人的建议是把课堂任务的答案作为课后作业提交，和实验分隔开来会比较好。

整体上来说函数式编程实验给我的体验良好，我在此由衷地感谢老师们这学期在课堂上对我们的殷切教导，在实验课上对我们的耐心检查。