

函数式编程原理

课程报告

**姓 名：陈科**

**学 号：U201814630**

**班 级：CS1805**

**指导教师：顾琳**

**计算机科学与技术学院**

**2021年4月23日**

## 一、函数式语言家族成员调研

### 1.1函数式编程语言家族成员

#### 1.1.1 ML

ML (meta language)是一个通用的函数式编程语言，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。作为元语言的ML是为了帮助在LCF定理证明机中寻找证明策略而构想出来的。（之前的元语言是pplambda，它联合了一阶逻辑演算、多态及Λ演算）。它使用了Hindley-Milner类型推论算法来推测大多数值的类型，而不需要四处使用注解。

ML一般被归为非纯函数式编程语言，因为它允许副作用和指令式编程。ML可以算一种具备命令式语言特点的函数型语言，或者说面向函数的命令型语言。和Lisp一样，ML具有非常灵活的函数功能。例如一个表达式的值可能就是一个函数，这个函数可以被作为参数传递给另一个函数，或者函数的返回值就是一个函数。同时和Algol类的语言比较接近的是，ML的语法象命令型的，而且用起来象用Algol家族的很多比较新的后代们一样方便。而且ML有并行扩展，可以用来写并行系统；甚至还有面向对象扩展。

#### 1.1.2 Haskell

Haskell是一种标准化的，通用纯函数式编程语言，有非限定性语义和强静态类型。它的命名源自美国逻辑学家Haskell Brooks Curry，他在数学逻辑方面的工作使得函数式编程语言有了广泛的基础。在Haskell中，“函数是一等公民”。作为函数式编程语言，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。具有“证明即程序、结论公式即程序类型”的特征。这也是Haskell语言以希腊字母「λ」作为自己标志的原因。Haskell语言的最重要的两个应用是GHC（Glasgow Haskell Compiler）和Hugs（一个Haskell语言的解释器）。



#### 1.1.3 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

#### 1.1.4 OCaml

Objective Caml（OCaml）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是开放原始码项目。此项目的管理和大部分维护工作交由INRIA。



由于caml的读音与camel一致，OCaml的吉祥物自然被定为骆驼，而以OCaml服务为主的软件也通常以骆驼的形象作为其LOGO。

OCaml将Caml语言在面向对象方面做了延展。Caml 是函数式编程语言，它的扩展语言还有基于微软.net平台的 f# (fsharp)语言。Caml 的代码大多可以在f#中使用。F#的开发工具有VS .net，Caml的代码也可使用。

OCaml的开发工具包含交互式顶层解释器(interactive toplevel interpreter)，字节码编译器(bytecode compiler)，以及最优本地代码编译器(optimizing native code compiler)。OCaml有一个巨大并强悍的标准库，这使得她可以像Python或者Perl语言一样可以方便地开发各种应用程序，健壮的模块化与面向对象编程结构又使得她可以胜任大规模软件工程项目。

OCaml是Caml的继承者，CAML的缩写最初代表着Categorical Abstract Machine Language(分类抽象机语言)，不过后来OCaml将这个抽象机淘汰掉了。如今人们更喜欢把CAML理解为Collaborative Application Markup Language(协作应用程序标记语言)。

#### 1.1.5 Scala

Scala是一门多范式的编程语言，一种类似java的编程语言，设计初衷是实现可伸缩的语言、并集成面向对象编程和函数式编程的各种特性。



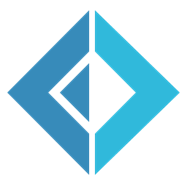
与只支持单继承的语言相比，Scala具有更广泛意义上的类重用。Scala允许定义新类的时候重用“一个类中新增的成员定义（即相较于其父类的差异之处）”。Scala称之为mixin类组合。

Scala还包含了若干函数式语言的关键概念，包括高阶函数（Higher-Order Function）、柯里化（Currying）、嵌套函数（Nested Function）、序列解读（Sequence Comprehensions）等等。

Scala是静态类型的，这就允许它提供泛型类、内部类、甚至多态方法（Polymorphic Method）。另外值得一提的是，Scala被特意设计成能够与Java和.NET互操作。Scala当前版本还不能在.NET上运行（虽然上一版可以-\_-b），但按照计划将来可以在.NET上运行。

#### 1.1.6 F#

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，而Ocaml是基于ML函数程序设计语言的。 这是一个用于显示·NET在不同编程语言间互通的程序设计。



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。

#### 1.1.7 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。

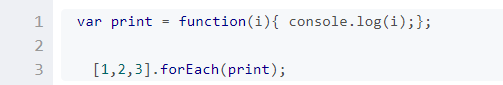
Lisp的语句结构也非常简单，和自然表达方式非常贴近。用圆括号把代表原子的标识符括起来形成一个列表，Lisp 程序中充满了一对对嵌套的小括号，这些嵌套的符号表达式体现了递归。例如( A1 A2 A3 A4 )，如果将它解释为数据，则表示一个四元素列表，若将它解释为代码，则表示名为A1的函数作用于A2、A3、A4这三个参数上。此外，圆括号可以嵌套，也就是LISP支持嵌套的表结构。

### 1.2函数式编程特点

#### 1.2.1函数是一等公民

所谓”第一等公民”（first class），指的是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。

举例来说，下面代码中的print变量就是一个函数，可以作为另一个函数的参数。



#### **1.2.2只用表达式，不用语句**

“表达式”（expression）是一个单纯的运算过程，总是有返回值；“语句”（statement）是执行某种操作，没有返回值。函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

原因是函数式编程的开发动机，一开始就是为了处理运算（computation），不考虑系统的读写（I/O）。”语句”属于对系统的读写操作，所以就被排斥在外。

当然，实际应用中，不做I/O是不可能的。因此，编程过程中，函数式编程只要求把I/O限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

#### 1.2.3无副作用

所谓”副作用”（side effect），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

副作用在命令式编程中非常常见，比如下面的C语言代码：

int a = 0;

int func(int b) {

a = a + b;

return a;

}

int main( ) {

int b = 1;

return func(b);;

}

在上面的例子中，main函数调用func函数。func函数接收整型参数b，功能是将全局变量a的值增加b，并返回修改后的a值。显然这个例子中的func函数是具有副作用的，它修改了全局变量a的值。调用func函数结束后a的值将变为1。

函数式编程强调没有”副作用”，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

#### 1.2.4不可变性

函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。

在其他类型的语言中，变量往往用来保存”状态”（state）。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。下面的代码是一个将字符串逆序排列的函数，它演示了不同的参数如何决定了运算所处的”状态”。

#### 1.2.5引用透明

引用透明（Referential transparency），指的是函数的运行不依赖于外部变量或”状态”，只依赖于输入的参数，任何时候只要参数相同，引用函数所得到的返回值总是相同的。

以上的特点都是函数式编程的核心，基于这些特点，又衍生出了许多应用场景：

• 纯函数：同样的输入得到同样的输出，无副作用。

• 函数组合：将多个依次调用的函数，组合成一个大函数，简化操作步骤。

• 高阶函数：可以加工函数的函数，接收一个或多个函数作为输入、输出一个函数。

• 闭包：函数作用域嵌套，实现的不同作用域变量共享。

• 柯里化：将一个多参数函数转化为多个嵌套的单参数函数。

• 偏函数：缓存一部分参数，然后让另一些参数在使用时传入。

• 惰性求值：预先定义多个操作，但不立即求值，在需要使用值时才去求值，可以避免不必要的求值，提升性能。

• 递归：控制函数循环调用的一种方式。

• 尾递归：避免多层级函数嵌套导致的内存溢出的优化。

链式调用：让代码更加优雅。

### 1.3函数式编程的兴衰

#### 1.3.1函数式编程（Composing Software）的兴起

在计算机科学初期，在很多计算机科学理论在电脑上实践在电脑上之前，有两个伟大的计算机科学家：Alonzo Church和 Alan Turing。他们发明了两种不同但是都被广泛使用的计算模型。两个模型都可以将任何可计算的模型计算。

Alonzo Church 发明了λ演算。λ演算是一个基于函数应用的普遍模型。Alan Turing以图灵机著称，图灵机是一个在一卷胶带定义了设备和操作符的普遍模型。

在这两组模型的合作中展示了，其实两种模型在功能上是殊途同归的。

#### 1.3.2函数式编程（Composing Software）的衰落

在二十世纪七十年代，创建程序的模式于简单的组合，成为了工业流水线式的生产。

之后又有了面向对象编程，一个关于组件封装和信息传递的好主意。不过由于一些热门的语言扭曲，变成了继承和类关系的噩梦。

函数是编程在学术和应用上都失去了地位，仅仅残存在一些有执念的极客，常青藤高塔中的教授，还有一些没有因Java风潮而着迷的学生那里。

对于大部分的人来说，这都是一个黑暗的时代。

#### 1.3.3函数式编程（Composing Software）的再次兴起

到了2010年左右，发生了一件好事：JavaScript大爆发。在2006年以前。JavaScript被认为是一个做网页动画的玩具语言，但是其实有很多强大的特性隐匿其中，即λ演算的特性。人们开始口耳相传的一个“新的概念”：“函数式编程”。

到了2015年，组合式程序的概念又开始兴起。为了让他更普遍，JavaScript也升级了新的特性，加入了箭头函数，使函数，柯里化，以及λ表达式更加直观可读。

箭头函数就像是函数式爆发的火箭燃料，以至如今已经很难见到不怎么使用函数式编程的JavaScript程序了。

组合让软件模型和行为都更加简单，优雅，直观。使用一些轻量而重要函数来构建大一些的程序组件，函数式的编写程序更加易于测试，理解，组织，引入。

## 二、上机实验心得体会

### 1.心得体会

四周的Standard ML实验课程的学习，让我对函数式编程语言产生了浓厚的兴趣，不同于面向过程的C语言和面向对象的Java语言，函数式编程将函数作为第一公民，针对逻辑操作将函数的结果和函数本身作为参数和返回类型，这和之前接触的语言有着本质的不同，经过几周的学习，我收获了很多新鲜的体验，从下面两个实验课题中我收获颇多：

①二叉搜索树的建立：

|  |
| --- |
| datatype tree = Empty | Node of tree \* int \* tree;  fun Ins(x, Empty) = Node(Empty, x, Empty)  | Ins(x, Node(t1, y, t2)) =  case Int.compare(x,y) of  LESS => Node(Ins(x, t1), y, t2)  | \_ => Node(t1, y, Ins(x, t2));  fun trav Empty = []  | trav(Node(t1,x,t2)) = trav t1 @ (x :: trav t2);  fun binarySearch(Empty,\_) = false  | binarySearch(Node(t1,x,t2),a:int) =  case Int.compare(a,x) of  LESS => binarySearch(t1,a)  | EQUAL => true  | GREATER => binarySearch(t2,a);  fun createSortedTree [] =Empty  | createSortedTree(x::L) =  Ins(x,createSortedTree(L)); |

这次实验题目的是完成一个二分查找的函数，针对一个表的二分查找，实验巧妙的设计了Ins函数，实现了将元素组装成表的函数，并通过插入元素与根节点比较，构造了一个左小右大的二叉搜索树。这种思路十分巧妙，而且将树拆分为表格和逐个元素插入构成树的方式打开了我的思路。这点在实验三中也被我所引用，解决了很多问题。

②递归的调用：

|  |
| --- |
| fun heapify Empty =Empty  | heapify(Node(Empty,x,Empty))= Node(Empty,x,Empty)  | heapify(Node(t1,x,t2)) =  let  val l = heapify t1  val r = heapify t2  in  SwapDown(Node(l,x,r))  end |

实验三选做题的第五题递归调用让我了解到递归调用的方式，从顶向下的依次递归和最终状态的条件判断赋值都是非常好的思路，原本非常复杂的小堆排序树的构建，通过递归调用子函数来实现就大大简化了实验的逻辑。

经过四周实验的学习，函数式编程的思想给我带来了很多新的感受，锻炼了我的编程能力，拓宽了思路。

### 2.课程建议

课程难度是以一种递增的形式逐渐引导大家进行功能的实现，但是希望可以在实验二树的部分增加一下提示和样例函数，方便学生作为参考，函数式编程语言对很多学生来说都是初次接触，在最开始的实验部分能够有更多的解析。