函数式编程原理

课程报告

姓　　名：朱槐志
学　　号：U201814745
班　　级：CS1807
指导教师：郑然

计算机科学与技术学院

2021 年 4 月 25 日

目录

一、函数式语言家族成员调研

Lisp，提出者John McCarthy，提出时间1960年4月，特点：

1. 函数性，用函数定义和函数调用组成的表达式来描述求解问题的算法，表达式的值就是问题的解
2. 递归性，LISP 的主要数据结构是表，而且表是用递归方法定义的，即表中的一个元素也可以定义为一个表
3. 数据与程序的一致性，LISP 的一段程序是用户的一个自定义函数，这个函数可被其他函数调用，一个函数被其他函数调用，就是调用了这个函数的返回值
4. 自动进行存储分配，用 LISP 语言编程时，程序员完全可以不考虑存储分配问题。程序中定义的函数、数据和表等都能在程序运行时，由 LISP 自动提供。对不再需要的数据，LISP 自动释放其占用的存储区。
5. 语法简单，对变量和数据不需要事先定义和说明类型，LISP 语言的基本语法就是函数定义和函数调用

兴衰：LISP1 -> LISP1.5 ->MACLISP ->Common LISP，还有许多其他版本的 LISP。原始定义简洁的缺点使得大型开发工程变得困难，自底层到高层，自二维表查询到面向对象，使用者需要嵌入更多的函数来实现，致使 LISP 众多方言衍生，也使它无法流行。

ML，提出者Robin Milner，提出时间二十世纪七十年代，特点：

1. 非纯函数式语言，它算是一种具备命令式语言特点的函数式语言
2. 尾部递归代替循环
3. 纯表达式风格
4. 模式匹配和多态
5. 允许副作用和指令式编程

兴衰：

ML中的思想影响了众多的语言，例如Haskell，Cyclone和Nemerle。ML大多被用于语言设计和操作，但是他作为通用语言也被用于生化，金融系统和宗谱数据库，P2P客户/服务端程序等。

Miranda，提出者

Clean，全名Concurrent Clean，由荷兰的尼兹梅根大学制作和维护，

特点：

它是用C写成的纯函数式程序设计语言，和Haskell有很多相似之处，Clean程序很容易跨平台。

Erlang，提出者Joe Armstrong，出现于1987年，

特点：

它是基于进程并发的，这些进程对于操作系统而言是透明的。支持热代码升级，可以实现不停机就进行代码的升级，来实现软件运行中的热升级。但是它是一种弱类型语言，语言抽象能力不强。Erlang是属于IO密集型的语言，适合分布式特征明显的项目。

兴衰：

Erlang近些年在国内的发展十分迅速，主要用于游戏服务器系统开发。Erlang上手容易，容错率高，快速迭代等优点非常适用于页游和手游的开发。但中小型公司注重产品大于技术，这对Erlang在于国内的发展是不利的，大部分公司都用着同一套已经成型的Erlang框架，着重于处理具体的业务逻辑忽视了底层优化。这对于Erlang的发展是极其不利的。

Erlang在国外发展的时间较长，作为一种成熟的语言已经得到了证实，目前应用于包括爱立信的宽带，GPRS和ATM交换解决方案系统等几百个重大的开发项目，有着丰富的项目经验，编程框架（OTP）为Erlang系统的提供了一套实现健壮性和容错性的工具和类库和完整的结构化框架。

Haskell，提出者Hudak P，Wadler P，

特点：

Haskell 支持惰性求值、模式匹配、列表解析、类型类和类型多态。它是一门纯函数编程语言，这意味着大体上，Haskell 中的函数没有副作用。Haskell 拥有一个基于 Hindley-Milner 类型推论的静态、强类型系统

兴衰：

Haskell有一个活跃的社区，在线上包仓库Hackage上有丰富的第三方开源库或工具。以Haskell为基础的衍生出了很多其他语言，它们分别是：并行Haskell，扩充Haskell(旧名 Goffin)，Eager Haskell， Eden， DNA-Hakell 和面向对象的变体(Haskell++，O'Haskell， Mondrian)。另外 Haskell 还被作为其他语言设计新功能时的样板，例如 Python 中的 Lambda 标记语句。

OCaml，提出者：avier Leroy，Jérôme Vouillon，Damien Doligez，Didier Rémy及其他人

特点：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。

Scala，提出者Oderksy M

特点：Scala是一种纯粹的面向对象编程语言，而又无缝地结合了命令式编程和函数式编程风格。既有面向对象的风格，又有函数式风格。支持更高层的并发模型。拥有轻量级的函数语法，与Java无缝的互操作。

兴衰：被 David Rupp 评价为，Scala 可能是下一代 Java。Scala 和 Groovy 两种语言都在快速发展的过程中。就情况来看，Groovy 的优势在于易用性以及与 Java 无缝衔接，Scala 的优势在于性能和一些高级特性，如果在发展过程中两者能互相借鉴对方的优点来充实自身，对开发者来讲无疑是福音。

F#，提出者：微软

特点：它提供了类型推断，提供了丰富的模式匹配结构，具有交互式脚本和调试功能。它允许写入高阶函数，并提供了发达的对象模型。

兴衰：F#通常用在：制定科学模型，数学解题，人工智能的研究工作，金融建模，平面设计，CPU设计，编译器编程。

Clojure，提出者Hickey R

- 特点：Clojure 保持了函数式语言的主要特点，例如 immutable state，Full Lisp-style macro support，persistent data structures 等等，并且还能够非常方便的调用 Java 类库的 API，和 Java 类库进行良好的整合。它专注于基本概念的不变性，不能对创建的对象进行任何修改，它可以管理程序员的应用程序的状态，它还支持并发。

兴衰：Clojure 用户拥有丰富的开发环境，每种环境都适合不同的社区和口味，并都日趋完善。当然 Clojure 还是一门小众语言。

二、上机实验心得体会

2.1.

对我来说感受比较深的一个实验是编写 interleave:int list * int list -> int list 这一题。它要求我们把两个 List 合并成一个 List。

可能是因为写实验二的时候对递归的理解还不很透彻，现在写报告的时候再想这个问题其实很简单。当时编写这个函数的时候我在想怎么让 interleave([x],[m,n])和 interleave([],[n])之间构建起联系，我当时想的是有很多种可能都能到达最终([],L)或者(L,[])的状态，但是我怎么才能得到上一个状态的结果从而进行 List 连接操作呢进而得出结果？因为 ml 不具备保存状态的能力，所以当时想了挺久。

当时我还是采取一种命令式的思维去思考，最终我编写出的代码是这样的：

```
fun interleave(A,[],l) = l @ A
  | interleave([],A,l) = l @ A
  | interleave(x1::L1,x2::L2,l) = interleave(L1,L2,l@[x1,x2]);
```

这种写法通过第三个参数把上一层的结果传递给下一层，执行到最后一层的时候就能得出结果。此时写报告时，我也写出了更符合函数式编程思想的解法：

```
fun interleave([],_) = []
  | interleave(_,[]) = []
  | interleave(x :: L, y :: R) = x::y::interleave(L,R)
```

这种写法就是经典的递归，上一层的结果依赖下一层的结果，只有先递归到最底层，再回溯上来，才能得出结果。我觉得这两种解法相比较的话，递归的写法更加的简洁，容易理解，它更符合我们过程式编程的思想。第一种解法类似于命令式语言中的 for 循环，但是这种写法在 ml 中显得有点“异类”我感觉，因为它将在每一层的计算结果作为状态传递给了下一层，其实不太符合函数式编程的思想。

2.2.

另一个对我来说感受很深的实验是实验三的第四题。这题相当于是让我们将一棵树转换成最小堆。其实这题看起来很难，但是用 sml 写起来其实很简单。

最小堆的定义就是根节点的值不大于左右子树节点的值。定义两个函数 SwapDown 和 heapify。其中函数 SwapDown 是在左右子树为最小堆的情况下对这棵树进行调整。如果

根节点的值没有比左右子树的节点值都小的话，我们就需要对这棵树进行调整，把三个节点值中最小的和根节点进行交换，并且给交换过的那颗子树递归进行 SwapDown。因为交换过节点值会影响子树的最小堆特性，需要进一步调整。整个 SwapDown 操作下来，整棵树也就成为了一个最小堆。

完成了 SwapDown 函数之后剩下的就很简单了，heapify 就是先给左右子树调用 heapify，使左右子树都为最小堆，然后对调整后的整棵树进行 SwapDown。即 heapify(Node(L,v,R)) = SwapDown(Node(heapify(L),v,heapify(R)))。

对这个函数的编写让我感受到 ml 语言在某些方面相比命令式语言的强大功能。在我使用 ml 的过程中，对我感触最深的就是 ml 中对 List 的操作非常的易于理解和简单，甚至对于树这种复杂的数据结构，在 sml 中都变得非常的简单，交换节点的操作很通俗易懂。如果是在 java 或者 c++中对于树的节点进行交换的话，需要的操作会更加的繁琐。另外 sml 中的 let 语句这种特性，可以弥补 sml 因为无法保存状态而可能造成的缺陷。

2.3 总结

ml 语言特点总结：

1）并行。

在函数式编程中，我们并不需要像 java 一样需要对代码增加额外操作，比如实现 Runnable 接口，重写 run 方法什么的，程序本身就可以并发运行。程序运行期间，我们也不用考虑死锁问题。原因是通过函数式编程所得到的程序，在程序中不会出现某一数据被同时修改两次及以上的情况，同样的，两个不同的线程就更不用说了。由于函数式编程有这样的优点，因此对于程序员编写并发代码十分友好。

（2）不修改状态、无副作用。

在函数式程序中，所有的功能的结果都是一个返回值，没有其他的行为，包括对外部变量的修改。在命令式语言中，变量是用来保存状态的。由于函数式编程不修改变量，导致了这些状态不能存在于变量中。函数式编程语言保存状态的方法是使用参数来保存，递归方法是最好的例子。不过也正是因为采用了递归方法，函数式编程语言在运行速度上相对于其他语言较慢，这也是函数式编程语言长期没能流行起来的主要原因。

（3）惰性求值

惰性求值即表达式不在它被绑定到变量之后就立即求值，而是在该值被取用的时候求值。惰性求值的一个好处是能够建立可计算的无限列表而没有妨碍计算的无限循环或大小问题。例如，可以建立生成无限斐波那契数列表的函数（经常叫做“流”）。第 n 个斐波那契数的计算仅是从这个无限列表上提取出这个元素，它只要求计算这个列表的前 n 个成员。

（4）柯里化

链接库往往将函数定义得比较一般化，具有通用性。这样的函数，需要传入比较多的参数。利用柯里化的方式，可以定义出“特殊化”的函数。所谓柯里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的方法。柯里化与部分求值是相关的，但不完全相同。

（5）模式匹配

模式匹配类比于命令式语言中函数的重载，即执行同一功能的函数可以根据输入参数类型和个数的不同做出不同的反应。从而实现分支功能。模式匹配可以让系统自动帮我们进行分支与变量的指定，有了模式匹配，函数式编程可以降低依赖 switch/case 而且写出来的程序代码也不会像 switch/case 那样一大块。

（6）递归

函数式编程中没有其他语言中 while、for 等循环语句，想要实现类似的功能只能通过递归实现。函数式语言设计者为了防止多重递归调用导致的爆栈等问题，函数式语言中往往会对递归进行优化，变化为等价的尾递归，某些编译器在编译器会对尾递归进行优化，使得每一次调用都使用相同的栈空间。

（7）高阶函数

在函数式编程中，函数也可以视为一个值，所以可以将一个函数作为另外一个函数的参数。一个函数就接收另一个函数作为参数，那么这么函数就是高阶函数。

由于以上的特点，使得 ml 语言有下列优点：

（1）更方便测试

编写可测试的代码是当下软件开发行业的基本要求，代码在交付前要编写测试用例对每一个功能做单元测试。副作用增加了代码测试难度，原因在于当你尝试测试依赖于外部条件的函数，测试用例也必须运行在相同的条件下。纯函数不依赖也不改变外界的状态，只要给定输入参数，返回的结果必定相同。因为没有副作用，所以不必提供特定的单元测试环境、编写单元测试将变得更加简单、轻松、

（2）易于并行编程

编写多线程代码尤其注意资源的互斥访问及死锁的问题。当纯函数在多线程环境执行的时候，不必担心互斥与死锁。纯函数不会去读取或修改外部对象的值，因此不必编写互斥或者读写锁之类的代码来保证资源被有序访问。

（3）开发快速

由于高阶函数的特性，多个函数可以组合成一个更强大的函数。柯里化技术主张用现有函数来构建新函数的理念。这极大地提高了代码的复用程度。

2.4 建议

我的建议是把实验开始的时间推迟一周，因为星期二刚刚上完第一节课，星期三就开始做实验我觉得还是有点跳跃，应该留一周的时间给我们缓冲会比较好，让我们有充足的时间对上课讲的知识消化，这样做起实验来也更加的快速高效。

另外就是在实验的时候会出现这样的情况：某个助教后面排了很多检查的人，而另外一个助教那边一个检查的人都没有，但是检查的人却必须在这边排队，不能给空闲的助教检查。这样的情况下实验检查的效率是很低的，我们学生的体验也非常不好，大量的时间都用来排队检查了。所以希望课程组能够给这种情况提出解决方案。我觉得就大家自由到老师那里检查就好了，没必要按班级分配老师，只是这样的话要辛苦老师每次实验之后要把检查表汇总一下。