华中科技大学

函数式编程原理

课程报告

姓　　名：　木林

学　　号：　I201920024

班　　级：　CS2002

指导教师：　顾琳

计算机科学与技术学院

2023 年 10 月 28 日

一、函数式语言家族成员调研

函数式编程语言家族是一个庞大而多样化的编程语言群体，它们共同强调函数的使用和纯计算。函数式编程语言有很多家族，但其中最著名的成员包括 Lisp、Scheme、ML、Haskell、Erlang、Clojure、OCaml、F# 和 Scala。

这些语言由不同的研究人员和实践者开发，但它们都有一些共同的特点。

1. 功能语言的特征

功能语言通常具有以下特点：

强调函数： 函数式语言将函数视为一等公民，这意味着函数可以像其他数据类型一样被传递和赋值给变量。这使得编写模块化和可重用的代码变得容易。

纯计算： 函数式语言鼓励使用纯函数，即没有任何副作用的函数。这使得代码更具可预测性，也更易于推理。

不变性： 函数式语言通常使用不可变数据结构，即一旦创建就无法更改的数据结构。这使得代码更加线程安全，也更容易调试。

衰落和繁荣的原因，多年来，函数式语言经历了衰落和繁荣两个时期。在计算机发展的早期，函数式语言非常流行，但在 20 世纪 70 年代和 80 年代，随着 C 和 Pascal 等命令式语言被更广泛地采用，函数式语言逐渐失宠。

不过，近年来函数式语言卷土重来，这要归功于一系列因素，包括多核编程的兴起以及对并发和并行编程语言的需求。大数据和机器学习日益流行，而这两者都非常适合函数式编程。新函数式语言的发展，这些语言更友好、更易学。与命令式语言相比，函数式语言具有许多优势，包括模块化、可重用性、可预测性和线程安全性。因此，函数式语言非常适合大数据、机器学习和并发编程等广泛应用。

除上述语言外，还有 Elm、PureScript 和 Racket 等许多其他函数式语言也在使用中。函数式语言家族是一个充满活力、不断发展壮大的社区，新语言层出不穷。

- Lisp

Lisp 是一种多范式编程语言，由约翰-麦卡锡（John McCarthy）设计，最早

出现于 1960 年。它以函数式、过程式、反射式和元编程范式而闻名。虽然 Lisp 可用于函数式编程，但它更像是一种多语言工具包。你可以进行函数式编程、面向对象编程（OOP）、逻辑编程，也可以为某种新范式定义自己的特定领域语言（DSL），等等。例如，Clojure 是 Lisp 的一种方言，它最接近函数式编程风格，因为它具有懒评估功能，不鼓励 OOP，自始至终使用不可变的数据结构，限制不受控制的突变（通过软件事务内存），并且在核心库中拥有大量纯函数/高阶函数。另一方面，Common Lisp 确实支持函数式特性，但命令式技术的使用在 Common Lisp 程序中更为普遍。

因此，虽然 Lisp 有能力支持函数式编程，但它并不是严格意义上的函数式语言，因为它支持多种范式。

- Scheme

Scheme 是 Lisp 系列编程语言的一种方言。它于 20 世纪 70 年代在麻省理工学院计算机科学与人工智能实验室（MIT AI Lab）创建，由其开发者 Guy L. Steele 和 Gerald Jay Sussman 发布。

Scheme 强调函数式编程和特定领域语言，但也能适应其他风格。它以简洁和极简的设计著称。Scheme 是历史最悠久、研究最深入的动态语言之一，有许多快速、可移植的实现。它是第一种选择词法范围的 Lisp 方言，也是第一种要求实现执行尾调用优化的 Lisp 方言，为函数式编程和递归算法等相关技术提供了更强大的支持。它也是最早支持一流连续性的编程语言之一。

与 Lisp 一样，虽然 Scheme 具有支持函数式编程的功能，但它并不是严格意义上的函数式语言，因为它支持多种范式。

- ML

ML（元语言）是一种函数式编程语言，由 Robin Milner 等人于 20 世纪 70 年代初在爱丁堡大学开发。它以使用多态 Hindley-Milner 类型系统而闻名，该系统可自动为大多数表达式分配类型，无需显式类型注释（类型推断），并确保类型安全。

ML 为函数参数、垃圾回收、命令式编程、逐值调用和卷曲提供了模式匹配。虽然 ML 是一种通用编程语言，但它在编程语言研究中被大量使用，而且是少数几种完全使用形式语义进行指定和验证的语言之一。它的类型和模式匹

配使其非常适合并常用于其他形式语言，如编译器编写、自动定理证明和形式验证。

ML 可以被称为不纯粹的函数式语言，因为尽管它鼓励函数式编程，但也允许使用副作用（与 Lisp 等语言类似，但与 Haskell 等纯函数式语言不同）。与大多数编程语言一样，ML 使用急于求值，这意味着所有子表达式都会被求值，不过可以通过使用闭包实现懒求值。如今，ML 家族有多种语言，其中最著名的三种是标准 ML（SML）、OCaml 和 F#。

- Haskell

Haskell 是一种纯函数式编程语言，于 1990 年首次推出。它以 Haskell Brooks Curry 命名，Haskell Brooks Curry 在数理逻辑方面的研究为函数式语言奠定了基础。Haskell 基于 lambda 微积分，因此以 lambda 作为标识。

在函数式编程中，程序是通过评估表达式来执行的，而命令式编程则不同，程序是由语句组成的，在执行时会改变全局状态。函数式编程通常避免使用可变状态。Haskell 以其强大的类型系统、懒评估和优雅的语法而著称。它使用多态静态类型系统，支持高阶函数、纯粹性（包括不可变数据和引用透明）、懒评估和递归等特性。Haskell 特别适用于需要高度可修改和可维护的程序。它被广泛应用于复杂软件系统的开发，对于希望编写高质量、可维护代码的开发人员来说，是一种非常好的语言。

- Erlang

Erlang 是一种函数式编程语言，由爱立信公司于 1986 年开发，用于电信应用。Erlang 本身支持并发、分布和容错。Erlang 可用于构建具有高可用性要求的大规模可扩展软实时系统。它的一些用途包括电信、银行、电子商务、计算机电话和即时消息。

Erlang 是一种通用语言，也支持函数式编程。它允许出现故障的软件或硬件不会导致系统瘫痪，从而实现了分布式容错。不过，值得注意的是，虽然 Erlang 支持函数式编程，但它也支持并发编程。Erlang 中的每个进程都是一个正在进行的计算，都有一个当前状态，该状态会随着其他进程的变化而变化。因此，虽然 Erlang 有能力支持函数式编程，但它并不是严格意义上的函数式语言，因为它支持多种范式。

- Clojure

Clojure 是一种函数式编程语言，由 Rich Hickey 创建，首次出现于 2008 年。它是 Lisp 的一种方言，可在 Java 虚拟机（JVM）、通用语言运行时（CLR）和 JavaScript 平台上运行。

Clojure 提供了避免可变状态的工具，将函数作为一级对象提供，并强调递归迭代而不是基于副作用的循环。Clojure 主要是一种函数式编程语言，拥有丰富的不可变、持久的数据结构。当需要可变状态时，Clojure 提供了一个软件事务性内存系统和反应式代理系统，可确保干净、正确的多线程设计。

不过，Clojure 并不纯粹，因为它并不强迫你的程序在引用上透明，也不追求程序的 "可证明性"。Clojure 背后的理念是，大多数程序的大多数部分都应该是功能性的，而功能性越强的程序就越健壮。

- OCaml

OCaml 是一种函数式编程语言，于 1993 年首次推出。它由法国国家研究机构 INRIA 开发，在学术界和工业界都很流行。

OCaml 以其强大的类型推断和安全性、垃圾回收和高级抽象而著称。它支持函数式、命令式和面向对象编程风格。函数式语言的关键语言抽象是数学函数。函数将输入映射到输出；对于相同的输入，函数总是产生相同的输出。

OCaml 提供模式匹配、垃圾回收和强大的模块系统。它还支持许多高级特性，如代数数据类型、参数多态性和模块。OCaml 并非纯粹的函数式语言，它在鼓励函数式编程的同时，也允许使用命令式特性。这使得 OCaml 成为一种多范式语言，可以灵活地为特定问题选择最合适的编程风格。

- F#

F# 是一种函数式编程语言，于 2005 年首次推出。它由微软研究院开发，目前由 F# 软件基金会维护。

F# 以其简洁、健壮和高性能的代码而著称。它允许您编写简洁、自文档化的代码，您可以将注意力集中在问题领域，而不是编程细节上。F# 是一种通用编程语言，兼具编译语言的高效性和解释型语言的简洁性。F# 支持函数式编程概念，如高阶函数、不变性和模式匹配。它还支持面向对象和命令式编程风格。这使得 F# 成为一种多范式语言，可以灵活地为特定问题选择最合适的编程风格。

- Scala

Scala 是一种函数式编程语言，于 2004 年首次推出。它被设计为 "更好的 Java"，可以与 Java 和 JavaScript 无缝互操作。Scala 是一种类型安全的 JVM 语言，它将面向对象编程和函数式编程都融入了一种极其简洁、高级和富有表现力的语言中。

Scala 提供了定义匿名函数的轻量级语法，支持高阶函数，允许函数嵌套，并支持卷曲。它还提供了用于数据处理的 Apache Spark 等框架，以及根据需要扩展程序的工具。函数式编程是一种强调只使用纯函数和不可变值编写应用程序的编程风格。Scala 强制执行不可变性，并使用纯函数，对于相同的输入总是返回相同的值，并且没有副作用。

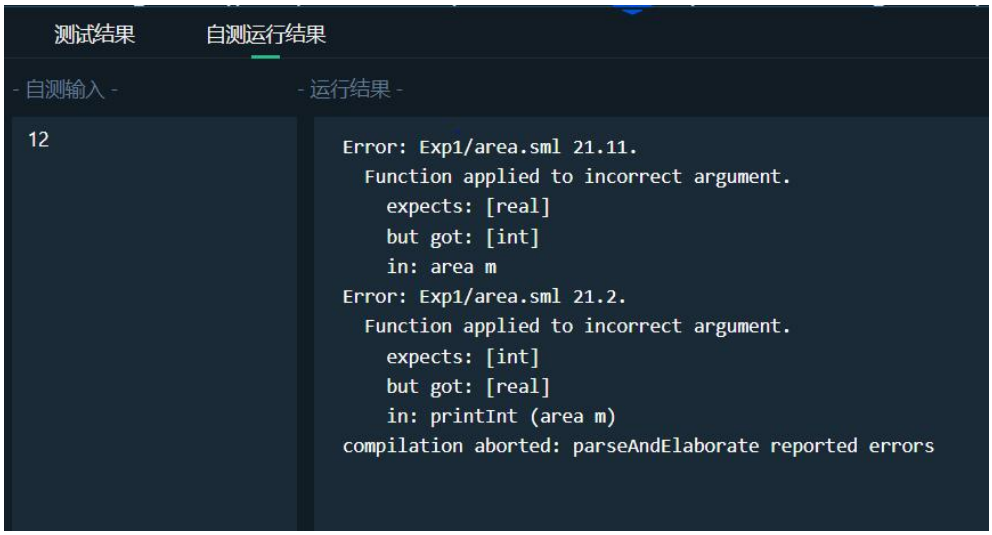
二、上机实验心得体会

1. 第一实验第一关

- 实验内容

```
(* 输出一个整数 *)
fun printInt (a:int) = print(Int.toString(a)^" ");
(* 输出一个实数 *)
fun printReal (a:real) = print(Real.toString(a)^" ");
(* 读取一个整数 *)
fun getInt () = Option.valOf (TextIO.scanStream (Int.scan StringCvt.DEC) TextIO.stdIn);
(* 读取一个实数 *)
fun getReal () = Option.valOf (TextIO.scanStream (Real.scan) TextIO.stdIn);
(*  完成 Begin 和 End 间代码的修改  *)
(*****Begin*****)
 val t = 5.0;
 fun area r = t * r;
 val m = getInt ();
 printInt(area m);
(*****End*****)
```

编译时会显示错误：



- 结题思路、代码、和运行结果

在函数 area 中，你使用了变量 t 乘以半径 r 来计算面积。这里我假设你需要使用实数作为半径来计算面积，所以 t 也应该是一个实数。我们可以看到，出现错误的原因是，如果 t 的值是 float，那么打印语句需要一个 float 变量参数，同样，如果 t 的值是 int，那么打印语句也需要一个 int 变量参数。因此，答案是要么将 t 的值改为 t=5，或者将 printInt 语句改为 printReal。

```
(* 输出一个整数 *)
fun printInt (a:int) = print(Int.toString(a)^" ");
(* 输出一个实数 *)
fun printReal (a:real) = print(Real.toString(a)^" ");
(* 读取一个整数 *)
fun getInt () = Option.valOf (TextIO.scanStream (Int.scan StringCvt.DEC) TextIO.stdIn);
(* 读取一个实数 *)
fun getReal () = Option.valOf (TextIO.scanStream (Real.scan) TextIO.stdIn);
(*  完成 Begin 和 End 间代码的修改  *)
(*****Begin*****)
 val t = 5;
 fun area r = t * r;
 val m = getInt ();
 printInt(area m);
```

(*****End*****)

预期结果为 12*5=60，实验结果如下图：

2. 第一实验第二关

- 实验内容

参照函数 sum 完成 mult 函数的编写，实现求解整数列表中所有整数的乘。函数 sum 用于求解整数列表中所有整数的和，函数定义如下：

```
(* sum : int list -> int *）
(* REQUIRES: true        *)
(* ENSURES: sum(L) evaluates to the sum of the integers in L. *)
fun sum [ ] = 0
   | sum (x ::L) = x + (sum L);
```

完成函数 mult 的编写，实现求解整数列表中所有整数的乘积。

```
(* mult : int list -> int         *)
(* REQUIRES: true        *)
(* ENSURES: mult(L) evaluates to the product of the integers in L. *)
fun mult [ ] =         (* FILL IN *)
   | mult (x ::L) =     (* FILL IN *)
```

- 结题思路、代码、和运行结果

mult [ ] 返回 1，因为空列表的乘积为 1。

mult (x :: L) 返回列表头元素 x 乘以剩余部分 (mult L) 的结果。这里使用了递归调用 mult L 来计算剩余部分的乘积。

```
fun printInt (a:int) =
   print(Int.toString(a)^" ");
fun getInt () =
   Option.valOf (TextIO.scanStream (Int.scan StringCvt.DEC) TextIO.stdIn);
fun printIntList ( [] ) = ()
  | printIntList ( x::xs ) =
   let
     val tmp = printInt(x)
   in
     printIntList(xs)
   end;
fun getIntList ( 0 ) = []
  | getIntList ( N:int) = getInt()::getIntList(N-1);
(*  完成 Begin 和 End 间代码的编写  *)
(*****Begin*****)
(* sum : int list -> int             *)
(* REQUIRES: true        *)
(* ENSURES: sum(L) evaluates to the sum of the integers in L. *)
fun sum [ ] = 0
   | sum (x ::L) = x + (sum L);
(* mult : int list -> int            *)
(* REQUIRES: true        *)
(* ENSURES: mult(L) evaluates to the product of the integers in L. *)
fun mult [ ] = 1
   | mult (x ::L) = x * (mult L);
(*****End*****)
val n = getInt();
val L = getIntList(n);
printInt(mult L);
```

预期结果为 9*5*2*3=270，实验结果如下图：

3. 第一实验第三关

- 实验内容

完成函数 Mult: int list list -> int 的编写,该函数调用 mult 实现 int list list 中所有整数乘积的求解。

```
(* Mult : int list list -> int     *)
(* REQUIRES: true        *)
(* ENSURES: Mult(R) evaluates to the product of all the integers in the lists of R. *)
fun Mult [ ] =     (* FILL IN *)
 | Mult (r :: R) =     (* FILL IN *)
```

- 结题思路、代码、和运行结果

如果列表为空，则返回 1（因为任何乘法运算的空列表都将返回 1）。

如果列表非空，则将第一个元素与列表中剩余元素的乘积相乘。这里使用了递归调用 Mult R，其中 R 是除第一个元素之外的剩余元素。

```
fun printInt (a:int) =
   print(Int.toString(a)^" ");
fun getInt () =
   Option.valOf (TextIO.scanStream (Int.scan StringCvt.DEC) TextIO.stdIn);
fun printIntList ( [] ) = ()
 | printIntList ( x::xs ) =
   let
    val tmp = printInt(x)
   in
    printIntList(xs)
```

```
    end;
fun getIntList ( 0 ) = []
  | getIntList ( N:int) = getInt()::getIntList(N-1);
fun getsubL ([], i) = ([],[])
  | getsubL (xs, 0) = ([],xs)
  | getsubL (x::xs, i) =
      let
          val (a,b) = getsubL(xs,i-1)
      in ((x::a),b)
      end;
(*  完成 Begin 和 End 间代码的编写  *)
(*****Begin*****)
(* mult : int list -> int            *)
(* REQUIRES: true        *)
(* ENSURES: mult(L) evaluates to the product of the integers in L. *)
fun mult [ ] = 1 | mult (x ::L) = x * (mult L);
(* Mult : int list list -> int       *)
(* REQUIRES: true        *)
(* ENSURES: Mult(R) evaluates to the product of all the integers in the lists of
R. *)
fun Mult [ ] = 1 | Mult (r :: R) = mult(r) * (Mult R);
(*****end*****)
val L = getIntList(10);
val (L1,L2) = getsubL(L,3);
val (L3,L4) = getsubL(L2,2);
val L5 = L1::L3::L4::[];
printInt(Mult L5);
```

预期结果为 3^10=59049，实验结果如下图：

三、课程建议和意见

总之，学习函数式编程语言可以为开发人员带来很多好处，包括提高代码质量、减少错误、提高生产率和更好的并发支持。函数式编程语言在金融、机器学习和大数据等行业也越来越受欢迎。因此，对于希望提升职业生涯的开发人员来说，学习函数式编程语言是一项非常有价值的投资。以下是学习函数式编程语言的一些额外好处：

提高代码质量： 函数式编程语言鼓励开发人员编写更简洁、可读性和可维护性更强的代码。这是因为函数式程序通常由纯函数组成，即接受输入并返回输出而不修改函数本身之外的任何状态的函数。这使得函数式代码更容易推理和调试。

减少错误： 函数式编程语言还鼓励开发人员使用不可变数据结构和纯函数，从而有助于减少错误。不可变数据结构一旦创建就无法更改，因此不易出错。纯函数也没有副作用，这意味着它们不会修改函数本身之外的任何状态。这使得跟踪和修复错误变得更加容易。

提高生产力： 函数式编程语言还提供了许多功能，使编写和维护代码变得更加容易，从而有助于提高开发人员的工作效率。例如，函数式编程语言通常支持 "懒评估"，这意味着表达式只在需要结果时才进行评估。对于某些类型的

程序来说，这可以大大提高性能。

更好的并发支持： 函数式编程语言也非常适合并发编程。这是因为函数式代码通常是线程安全的，这意味着它可以并发执行而不会导致错误。这使得函数式编程语言成为开发需要同时处理多个任务的应用程序的良好选择。

本课程的实验课结束以后，对 Educoder 实验平台有一个建议，即不应该对学生隐藏测试集结果，特别是对于只有一个测试集结果的实验，或者至少学生可以通过花费自己获得的硬币来查看。我们可能无法在第一次尝试时就写出一个包含所有数据条件的完全可行的程序，但我们可以利用偷看答案的方式来帮助发现我们程序编写中的错误，这也是提高学生思考编写一个在所有情况下都可行的正确程序的能力的一种方法，否则，这将使学生在处理为什么我们的编程不能像预期的那样工作时非常困难，完全迷失和困惑，被困在一个问题上，可能会因为尽力解决问题而毫无进展。