# 函数式编程第一次实验报告

## 历史调研报告

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

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。

## 实验主体

1.下列模式能否与类型为int list的L匹配成功？不成功的话，请指出该模式的类型

x::L 能匹配除了空的list以外的int list

\_::\_ 不一定能匹配成功，因为\_表示它是任意类型，如果左边\_是int类型则能匹配成功，否则匹配不成功，实际的模式无法确定。同时不能匹配空的int list

x::(y::L) 由于表中必须存储相同类型的数据，所以y也一定是int类型，否则会报错，分析应该是list L里面加入了x和y两个int类型的数据，可以匹配成功。但是匹配的是长度至少为2的int list

(x::y)::L 括号优先级最高，先看括号，括号内可以看出y是一个int list，可以得到L是一个存储int list的list，即嵌套的list，所以匹配不成功。

[x, y] 因为是用中括号包裹起来的，所以一定是list，又因为list里的元素类型必须相同，所以y也是int类型，所以匹配。

2. 试写出与下列表述相对应的模式。如果没有模式与其对应，试说明原因。

list of length 3 [x,y,z]

lists of length 2 or 3 没有这样的模式，一个表达式只有一种模式，一个列表只有一个长度

Non-empty lists of pairs (x,y)::L

Pairs with both components being non-empty lists (x::y,m::n)

3. 第四行x = 2，第五行m =12.4,第六行x绑定的是int类型，值为9001,第十四行表达式assemble(x,3.0)计算结果是27。

1. (\*f:int -> int)

fun f(3: int) : int = 9

f\_ = 4;

错误一：f\_ = 4这句话前面前面少了竖线 |

(\* circ: real -> real \*)

fun circ(r : real) : real = 2 \* pi \* r;

错误二：2是整数，pi是实数，实数整数之间不能运算

(\*semicirc: real -> real \*)

fun semicirc : real = pie \* r;

错误三：函数semicirc不符合描述，没有参数， r都无定义

(\* area: real -> real\*)

fun area(r : int) : real = pi \* r \* r;

错误四：函数area参数类型错误，应该是real，实际上却是int

1. 输入和输出结果中间用分号隔开

3+4; val it = 7 : int

3+2.0;Error:operator and operand don’t agree 类型不匹配，一个是real，一个是int

It +6; val it = 13 : int

val it = “hello”; val it = "hello" : string

It + “world”; Error: overloaded variable not defined at type+运算符不能用于字符串相加

It+5; Error: overloaded variable not defined at type+运算符也不能用于字符串和整数相加

val a = 5; val a = 5 : int

a=6; val it = false : bool=运算符用于判断等号两边是否相等，返回bool值

a+8; val it = 13 : int

val twice = (fn x => 2 \* x); val twice = fn : int -> int

twice a; val it = 10 : int 因为函数twice参数为x，返回2x，而a=5，故返回10

let x = 1 in x end; stdIn:13.1-13.8 Error: syntax error: deleting LET ID EQUALOP

stdIn:13.11 Error: syntax error found at IN

foo; stdIn:1.2-1.5 Error: unbound variable or constructor: foo未定义变量

[1,”foo”];列表里的数据的类型必须一样

完成函数mult的编写，实现求解整数列表中所有整数的乘积。

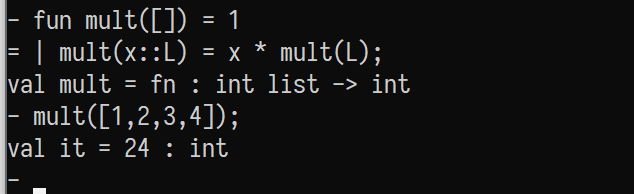
(\* mult : int list -> int \*)

(\* REQUIRES: true \*)

(\* ENSURES: mult(L) evaluates to the product of the integers in L. \*)

fun mult[] = 1

| mult(x::L) = x \* mult(L);

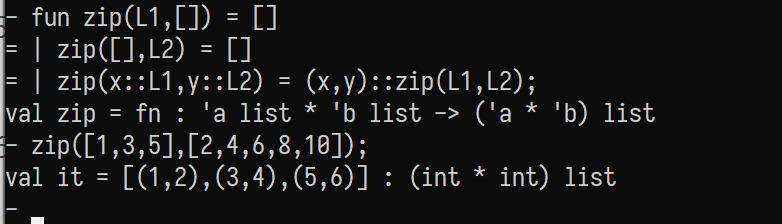


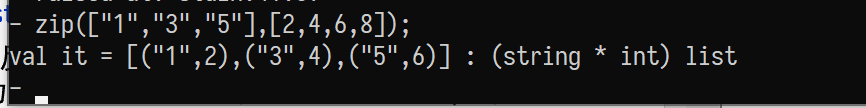
1. (\* zip:string list \* int list -> (string \* int) list \*)

fun zip(L1,[])=[]

| zip([],L2)=[]

| zip (x::L1 y::L2) = (x,y)::zip(L1,L2)





(\* unzip: (string \* int) list -> string list \* int list \*)

unzip((x,y)::L) = let val (L1,L2)=unzip(L) in (x::L1,y::L2) end

8.

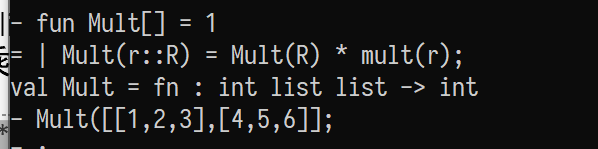
(\* mult : int list list -> int \*)

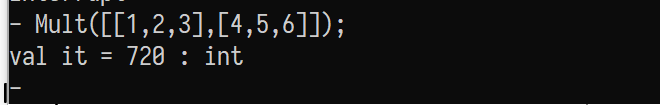
(\* REQUIRES: true \*)

(\* 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);





9.

(\* mult’ : int list \* int -> int \*)

(\* REQUIRES: true \*)

(\* ENSURES: mult’(L, a) … (\* FILL IN \*) \*)

 fun mult’ ([ ], a) = a

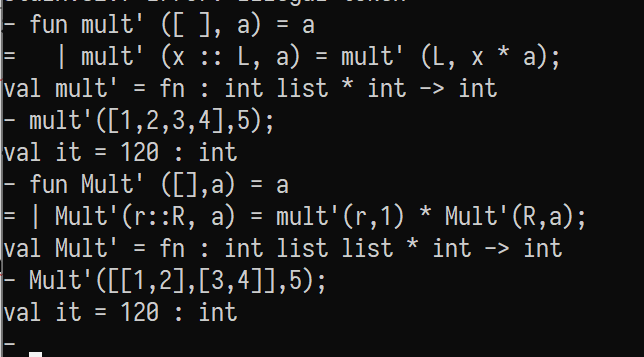
| mult’ (x :: L, a) = mult’ (L, x \* a);

mult’函数是求整数列表中的所有整数与a的乘积。

(\* Mult’ : int list list \* int -> int \*)

fun Mult’ ([],a) = a

| Mult’ (r::R, a) = mult’(r, 1) \* Mult’(R, a);



10.

编写递归函数square实现整数平方的计算，即square n = n \* n。

要求：程序中可调用函数double，但不能使用整数乘法（\*）运算。

(\* double : int -> int \*)

(\* REQUIRES: n >= 0 \*)

(\* ENSURES: double n evaluates to 2 \* n.\*)

fun double (0 : int) : int = 0

| double n = 2 + double (n - 1)

(\* square : int -> int \*)

(\* REQUIRES: n >= 0 \*)

(\* ENSURES: double n evaluates to n \* n.\*)

fun square(0 : int) : int = 0

| square(x) = square(x – 1) + double(x) – 1;

