

华中科技大学

2020

函数式编程原理

课程总结报告

题目：函数式编程的语言及特点介绍

专业：计算机科学与技术

班级：CS1701

学号：U201714501

姓名：熊逸钦

电话：18507110246

邮件：yiqin0411@qq.com

目 录

1	目标与选题动机	2
1.1	目标	2
1.2	选题动机	3
2	函数式编程语言的家族成员及其简介	5
2.1	静态类型的函数式编程语言	5
2.2	动态类型的函数式编程语言	8
3	函数式编程的特点	12
3.1	函数是第一等公民	12
3.2	多态类型	13
3.3	高阶函数	14
3.4	柯里化	15
3.5	无副作用	16
3.6	“无状态”和递归	17
4	心得收获与建议	18
4.1	课程收获	18
4.2	建议	18

1 目标与选题动机

1.1 目标

课程中总共安排了三次实验，实验目标和具体验证的知识点如下：

1.1.1 第一次实验

实验的总体目标是：

- (1) 熟悉 SML/NJ 开发环境及使用；
- (2) 掌握 SML 基本语法和书写规则；
- (3) SML 简单程序设计和程序编写

具体的，实验中验证的知识点包括：

- (1) SML 是强类型语言，不同数据类型不可以进行运算；
- (2) 字符串类型没有+操作符，要用^操作符拼接；
- (3) 形如 $x = 1$ 的表达式不是给 x 赋值，而是一个布尔值；
- (4) 要使用变量前需先进行绑定；
- (5) 元组中元素数据类型可以不同，列表中元素数据类型必须相同；
- (6) 掌握函数的递归写法和对列表的处理；
- (7) 掌握基本的模式匹配和临界情况处理；
- (8) 函数之间可以相互调用。

1.1.2 第二次实验

实验的总体目标是：

- (1) 掌握 list 结构的 ML 编程方法和程序性能分析方法；
- (2) 掌握基于树结构的 ML 编程方法和程序性能分析方法。

具体的，实验中验证的知识点包括：

- (1) 与 list 相关的运算符的使用，比如::符号是取表中元素，@符号是表的拼接；

华中科技大学课程总结报告

(2) 作用域限定 `let-in-end` 的使用，在 `let` 之后 `in` 之前赋值的变量只在 `in` 之后 `end` 之前有效；

(3) 掌握递归函数结构的分析，并能够通过判断运行次数来确定时间复杂度；

(4) 掌握 `list` 结构的遍历、插入排序和归并排序算法；

(5) 掌握 `tree` 结构的遍历、插入排序和归并排序算法；

(6) 掌握使用 `datatype` 进行树结点类型定义；

(7) 掌握包含并行递归过程的函数的性能分析，从 `work` 和 `span` 两个角度分析；

(8) 掌握二分搜索算法的实现。

1.1.3 第三次实验

实验的总体目标是：

(1) 掌握多态类型、`option` 类型和高阶函数的编程方法；

(2) 用 ML 语言求解实际问题。

具体的，实验中验证的知识点包括：

(1) 高阶函数，将函数作为参数传递；

(2) 对列表中所有元素使用同一函数进行作用的 `mapList` 函数；

(3) 掌握用高阶函数实现 `mapList` 的方法；

(4) 掌握用柯里化函数实现 `mapList` 的方法；

(5) 掌握将数据包装为 `option` 类型的方法；

(6) 掌握找零问题的解决方法，并利用找零问题的思路求解类似问题。

1.2 选题动机

本次报告选取的两个主题方向分别是：

(1) 函数式编程语言的家族成员及其简介；

(2) 函数式编程的特点。

首先，选择调查和总结函数式编程语言的家族成员体系的原因是，我认为一种编程思想要代入实践就必须有这种思想的载体，而函数式编程语言恰恰就是将函数式编程引入实践的载体。因为语言是用来描述思想的工具，所以通过语言的发展可以看到函数式编程的发展，通过语言的特性也可以看到函数式编程的特性。此外，语言的发

华中科技大学课程总结报告

展是多元化的，除了在课程中学习的 Standard ML 之外，还有许多不同的函数式编程语言。例如函数式编程语言的理论基础基本都是 λ 演算，而后出现了以 Lisp 为代表的动态类型的函数式编程语言，以及以 ML 为代表的静态类型的函数式编程语言，以 Lisp 和 ML 为源头又各有许多方言，形成各自的家族体系。这么多的函数式编程语言的关系错综复杂，从这个角度来说，我也认为有必要对它们进行梳理。

然后，选择调查和总结函数式编程的特点的原因是，我在课程学习的过程中，一开始非常不习惯和不理解函数式编程的一些特性，一直使用命令式的思维去理解，感觉很多特性是“反人类”的，比如无法对 list 进行下标操作、没有循环结构、不能随意使用变量保存状态等。但随着学习的深入，逐渐可以理解这些特性在函数式编程中发挥的作用。在整个课程学习完之后，我认为有必要对这些有趣的特性进行一个总结，巩固自己对函数式编程思想的理解。

下面的第二章将围绕函数式编程语言的家族成员及其简介这一主题展开，主要根据静态类型和动态类型对函数式编程语言进行划分，并对一些较为主流的函数式编程语言的诞生、特点以及和其他语言之间的联系进行简要介绍。

下面的第三章将围绕函数式编程语言的特点展开，主要包括函数是第一等公民、多态类型、高阶函数、柯里化、无副作用、“无状态”和递归这些特点的介绍和举例。在介绍特点时会以课程中所学习的 Standard ML 语言为例，结合具体代码进行特点的分析，并会介绍这些特点所带来的一些好处。

2 函数式编程语言的家族成员及其简介

本章主要探讨函数式编程语言的分类，以及介绍各类型函数式编程语言中具有代表性的家族成员。

按照一般思路，可以从强类型/弱类型，静态类型/动态类型这两个维度来对函数式编程语言进行分类。但是我经过了解之后发现绝大多数的主流函数式编程语言均采用强类型的设计。弱类型的函数式编程语言不具有主流地位，例如法国数学家 David Madore 于 1999 年设计的 Unlambda 语言，由 s、k、i 三个符号（组合子）完成基本操作，语言形式非常有趣，但实际应用中很少使用。

因此下文所描述的较为主流的函数式编程语言均为强类型语言，下面主要按照静态类型和动态类型对它们进行分类介绍。同一类型中按照诞生年份的前后进行排序。

2.1 静态类型的函数式编程语言

静态类型的语言指的是在运行之前的编译阶段就进行类型检查，并确定变量的数据类型的一类语言。静态类型的较为主流的函数式编程语言如下。

2.1.1 ML

ML(Meta Language)由爱丁堡大学的 Robin Milner 等人在二十世纪七十年代晚期开发。ML 并非纯函数式编程语言，因为它在一定程度上允许副作用和进行指令式的编程。同时 ML 具有函数式语言常有的惰性求值的策略，也具有多态、模式匹配、高阶、递归等特点。

ML 具有非常好的灵活性，表达式的值可以是函数，这个值可以作为其他函数的参数，也可以作为其他函数的返回值。这个特点在本次课程学习 Standard ML 语言时，尤其在学习高阶、多态和柯里化等特性时体现非常明显。

目前 ML 语言家族具有两个较大的分支，一个是我们课程中所学的 Standard ML（标准 ML），另一个是 Caml。本次课程中使用的是 Standard ML 最著名的编译器 Standard ML New Jersey（SML/NJ）。Caml 的具体实现是 Ocaml 语言，详见本节的 2.1.4 小节介绍。

2.1.2 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。

2.1.3 Haskell

Haskell 是于 1990 年在 Miranda 语言的基础上进行标准化的，它的理论基础也是 λ 演算。Haskell 的命名来源于美国逻辑学家哈斯凯尔·加里，他在数理逻辑方面的工作很有造诣。

值得一提的是 Haskell 的标志，如下图所示：



图 2.1 Haskell 语言标志

图中的标志构成一个字母 λ 的样子，就是为了致敬 λ 演算这一套基于数学的形式系统。

Haskell 和前面提到的 Miranda 语言一样是纯函数式编程语言，函数没有副作用，支持惰性求值、模式匹配、列表解析、类型类和类型多态等特性。

目前 Haskell 经历了从 Haskell 1.0 至 1.4、Haskell 98、Haskell Prime 到 Haskell 2010 这几个大的版本更迭，现在官方网站上指定下载的编译器是 GHC，它可以跨平台运行，具有高性能的并发和并行能力。

目前 Haskell 语言的用户群体比较庞大，社区也很活跃，拥有一个名为 Hackage 的线上 Haskell 代码仓库，上面有很多开源的第三方库可供下载。

2.1.4 OCaml

OCaml 全称 Objective Caml，由 Xavier Leroy, Jérôme Vouillon, Damien Doligez,

Didier Rémy 及其他人于 1996 年创立，是一个开源项目。

OCaml 的标志是一只骆驼，如下图 2.2 所示：



图 2.2 OCaml 语言标志

就像它的名字，Objective Caml 是 Caml 语言在面向对象方面拓展之后的产物，Caml 是前文中 ML 的一个主要分支。

OCaml 的开发工具包括一个交互式的顶层解释器、字节码编译器以及最优本地代码编译器。它的一个最大特点就是具有健壮的模块化结构以及引入了面向对象的编程结构，在一定程度上提高了函数式编程语言在大型软件工程项目中的应用能力。

2.1.5 Scala

Scala 语言由联邦理工学院洛桑（EPFL）的 Martin Odersky 于 2001 年基于 Funnel 的工作开始设计。Funnel 是把函数式编程思想和 Petri 网相结合的一种编程语言。正因为此，Scala 语言作为一门纯粹的面向对象的语言，无缝结合了函数式编程语言的特性。

Scala 语言的标志如下：



图 2.3 Scala 语言标志

Scala 语言在函数式编程层面提供了许多关键概念的支持，比如它支持高阶函数、函数柯里化、函数嵌套、多态、匿名函数等特性。

Scala 运行于 JVM 虚拟机，加上它融合了面向对象、命令式编程和函数式编程的思想，使得它吸引了很多开发者进行尝试。它现在是一门非常热门的语言，拥有很活跃的开源社区和很多使用者。

2.1.6 F#

F#语言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。自 2002 年开始研发, 2005 年发布了第一个版本, 2007 年底正式从研发专案转移至产品部门, 并决定将 F#置入 Visual Studio 2010。

F#语言的标志如下:



图 2.4 F#语言标志

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。

2.2 动态类型的函数式编程语言

动态类型的语言指的是在运行期间才进行数据类型检查, 把类型绑定的过程延迟到运行阶段的一类语言。动态类型的较为主流的函数式编程语言如下。

2.2.1 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。

Lisp 语言的标志如下：



图 2.5 Lisp 语言标志

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。

2.2.2 Scheme

Scheme 语言由麻省理工大学的 Gerald J. Sussman 和 Guy L. Steele Jr. 创造，诞生于 1975 年。

这门语言是现代化的 Lisp 语言的一种方言，它的特点是精简，以一个非常小的语言核心作为标准，再以语法糖的形式来对语言本身进行拓展。

Scheme 语言除了具有函数式编程语言的基本特点外，还具有的特点有：继承于 Lisp 的括号嵌套、自动内存管理、支持尾递归、可以作为脚本语言进行嵌入。它所支持的数据结构在 Lisp 的基础上有所拓展，支持数字、字符、字符串、布尔值、列表、数组、函数这些广泛的数据类型。相比于 Lisp 只支持原子和列表而言有了很大的扩充。

2.2.3 Clojure

Clojure 是一种运行在 Java 平台上的 Lisp 方言，尽管 Lisp 具有强大的表达能力

和强悍的功能，但是由于它的表达形式过于规范化和数学化，难以应用到实际场合之中。Clojure 语言的出现就是为了改变这一点。Clojure 借助了 JVM 即 Java 虚拟机的支持，使得在任何 JVM 平台上都可以很方便地使用 Clojure 语言来实现 Lisp 的一些强大的特性和功能。

Clojure 语言的标志如下：



图 2.6 Clojure 语言标志

Clojure 和 Lisp 一样是一门动态类型的语言，它的语法和 Lisp 非常接近，在 JVM 平台上运行的时候，会被编译为 JVM 的字节码进行运算。由于运行在 Java 虚拟机上，因此 Clojure 语言在保持了函数式语言主要特点的同时，还可以方便地调用 Java 类库的 API，大大提高了实际可用性。

2.2.4 Erlang

Erlang 是一种通用的面向并发的编程语言，它由瑞典电信设备制造商爱立信所辖的 CS-Lab 开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境。

Erlang 语言的标志如下：



图 2.6 Erlang 语言标志

Erlang 于 1987 年首次被创造，直到 1998 年才发布开源版本，主要是因为当年对高并发的要求还不高，但是由于现在高并发的场景增多，Erlang 也迎来了快速发展的阶段。

首先 Erlang 作为函数式编程语言，具有函数式编程语言的基本特性，在 Erlang 中，函数是基本单位，是第一等公民，可以参与计算，可以作为参数、作为返回值传值。几乎所有的概念都是由函数表达，所有的操作也都是由函数完成。

然后 Erlang 具有非常好的并发支持，得益于它的 Erlang 虚拟机，以及轻量级的

华中科技大学课程总结报告

Erlang 进程，它可以支持超大规模的并发应用，而且无需依赖第三方库或是操作系统的调度。

此外，Erlang 还具有热更新的性质，也就是说它支持代码在运行时被修改。这个特性对于更新和检修操作非常友好，能够最大程度的保证 Erlang 系统的运行，不会因为业务更新造成系统的暂停。Erlang 支持热更新的基础在于它是一门动态类型的语言，只有在运行期间才进行数据类型的检查，对数据类型的绑定延迟到运行阶段完成。

3 函数式编程的特点

本章主要探讨函数式编程的一些突出特点，每个小节将讨论一个特点，并会以我们课程所学的 Standard ML 语言为例进行具体描述。

3.1 函数是第一等公民

第一等公民（first-class values）指的是具有和其他数据类型一样的平等地位。不同等次（class）类型的地位如下表 2.1 所示：

表 2.1 不同等次类型的地位

等次	地位
First Class	可作为函数参数、可返回、可赋值给其他变量
Second Class	可作为函数参数、不可返回、不可赋值给其他变量
Third Class	不可作为函数参数、不可返回、不可赋值给其他变量

函数作为第一等公民，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。函数与其他数据类型一样，处于平等的地位，这就意味着函数与程序和过程之间没有区别，函数可以看作数据来进行使用。

下面使用 standard ML 语言的语法举三个例子来说明函数作为第一等公民的特性。

（1）作为函数参数

```
(* thenAddOne: ((int->int)*int)->int *)  
fun thenAddOne (f, x) = f x + 1;
```

上例中的 thenAddOne 函数的参数中就包含了另一个函数 f，函数功能是将参数 x 使用函数 f 映射之后的值再加 1。

（2）作为返回值

```
(* mapList': ('a->'b)->('a list->'b list) *)  
fun mapList' f =  
  fn L => case L of  
    [] => []  
  | x::R => (f x)::(mapList' f R);
```

上例中的 mapList' 函数接受一个函数 f 作为参数，返回一个将 f 作用到列表中每

一个元素的函数。

(3) 赋值给其他变量

```
(* addOne: int->int *)  
fun addOne x = x + 1;  
(* 赋值给 copyFun *)  
val copyFun = addOne;
```

上例中函数 `addOne` 的功能是将整型 `x` 的值加 1 之后返回。赋值语句“`val copyFun = addOne;`”将 `addOne` 函数的职能赋给了 `copyFun`，此时 `copyFun x` 的函数功能与 `addOne x` 一致。

3.2 多态类型

多态的含义是多种形态，指的是在类型推导过程中一些无约束的类型。多态类型是一个类型模式，用一个具体的类型去匹配这样一个类型模式，就可以将多态的类型模式实例化为一个具体的类型。

在 `standard ML` 语法中，多态类型在类型推导中使用单引号+小写字母的形式来代指。具体而言，在类型推导中出现的第一个多态类型为 `'a`，第二个为 `'b`，以此类推……如下图 2.1 所示：

```
- fun f(x, y, z) = (x, y, z);  
val f = fn : 'a * 'b * 'c -> 'a * 'b * 'c
```

图 2.1 多态类型举例

在上例中，函数 `f` 的参数 `x`、`y`、`z` 在类型推导时无法确定，因此被认为是多态类型，但 `x`、`y`、`z` 的具体类型可以不同，因此使用不同的小写字母进行区分，类型分别为 `'a`、`'b`、`'c`。

使用语句 `f(1, 3.14, "hello")` 对上例中的多态类型进行实例化，如下图 2.2 所示。

```
- f(1, 3.14, "hello");  
val it = (1, 3.14, "hello") : int * real * string
```

图 2.2 多态类型实例化

在上例中，`x` 值为 1，类型为 `int`；`y` 值为 3.14，类型为 `real`；`z` 值为“hello”，类型为 `string`，这是将多态类型 `'a`、`'b`、`'c` 分别实例化得到的结果。

函数式编程中引入多态类型，可以避免写很多多余的代码，同时便于维护。因为不需要对每一种类型都编写一个对应的函数，只需要利用多态类型，在实际匹配时将

多态类型实例化为所需的数据类型即可，需要维护的函数也就只有一个，不需要过多考虑类型问题。

3.3 高阶函数

高阶函数可以使用另一个函数作为其输入参数，也可以返回一个函数。如果说上一节中的多态特性是为了简化多个类型的相同操作，那么高阶函数特性则是简化了多个参数的函数操作，也简化了同类型批量数据的不同函数操作。

具体来说，有了高阶函数之后，可以通过内部的一个函数把所有的参数处理完之后得到一个中间结果，再把这个中间结果作为变量传递给上一级函数。比如需要对一个整型列表中的全部元素进行递增、翻倍和平方这三种不同的操作，可以使用下面三个单独的函数：

```
(* addList: int list->int list *)  
fun addList [ ] = [ ]  
  | addList (x::L) = (x+1) :: addList L;  
  
(* doubleList: int list->int list *)  
fun doubleList [ ] = [ ]  
  | doubleList (x::L) = (x*2) :: doubleList L;  
  
(* sqList: int list->int list *)  
fun sqList [ ] = [ ]  
  | sqList (x::L) = (x*x) :: sqList L;
```

上例中的三个函数 `addList`、`doubleList`、`sqList` 分别实现了对列表中所有元素的加 1、乘 2 和平方操作。但是经过观察可以发现上述三个函数的结构都是类似的，都是对每个元素的操作，只是操作的类型不同。根据这个思想可以使用高阶函数的概念，把这个不同类型的操作抽象成对单个元素的函数 `f`，把 `f` 作为参数传递到高阶函数中，对列表中的每一个元素都使用函数 `f` 进行映射，如下例中 `mapList` 函数所示：

```
(* mapList: (('a->'b)*'a list)->'b list *)  
fun mapList(f, [ ]) = [ ]  
  | mapList(f, x::L) = (f x)::mapList(f, L);
```

华中科技大学课程总结报告

有了 `mapList` 这个高阶函数之后，只需要实现 `int->int` 类型的加 1、乘 2 和平方函数，然后再将对应的函数作为参数送入 `mapList` 函数中，对抽象的函数 `f` 进行具体化，就可以灵活实现对列表中所有元素进行同一操作。

3.4 柯里化

当一个函数面临接收的参数数量不止一个时，对多个参数有两种处理方法，一种是把多个参数当成一个元组，第二种则是接收第一个参数，然后随即返回一个函数，这个函数可以接收第二个参数，以此类推直到所有参数被接收。第二种方式的函数就是柯里函数，柯里化也就是把接收多参数的函数转变为接收单一参数（第一个参数），并返回一个能够接收后续参数的函数的技术。

举例而言，一个 2 参数的函数使用第一种方式进行处理时定义为 `(a * b) -> t` 类型，它是一个作用于元组的单参数函数。而使用第二种方式进行处理时则定义为 `a -> (b -> t)`，它是一个柯里函数，其接收第一个参数 `a`，返回一个能够接收参数 `b` 的函数。

下面是一个代码示例：

```
(* toBase: int -> int -> int list *)
(* REQUIRE: n 为 10 进制，且 n>=0,b>1 *)
(* ENSURE: toBase b n 将十进制数 n 转换为 b 进制数的 int list 形式 *)
fun toBase b 0 = []
  | toBase b n =
    let
      val (r,q) = (n mod b, n div b)
    in
      r::(toBase b q)
    end;
```

在这个例子中 `toBase` 函数接受一个参数 `b`，返回一个接收参数 `n` 并将十进制数 `n` 转换为 `b` 进制数的 `int list` 形式的函数。

函数的柯里化有以下优点：首先可以提高函数的适用范围，其次由于是逐个接收参数逐个累积，最后才进行执行，具有延迟执行的特性，最后可以提前把易变因素确定下来，减少了后续生成函数的复杂程度，易于理解和设计函数。

3.5 无副作用

副作用指的是函数内部和外部进行互动，产生了运算之外的其他结果。比如说，在函数内部修改了全局变量的值。副作用在命令式编程中非常常见，比如下面的 C 语言代码：

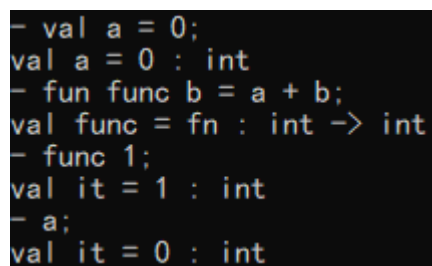
```
int a = 0;
int func(int b) {
    a = a + b;
    return a;
}
int main() {
    int b = 1;
    return func(b);
}
```

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。

再看函数式编程的例子，以 Standard ML 语言为例，代码如下：

```
val a = 0;
fun func b = a + b;
func 1;
a;
```

上例中 func 函数的功能和 C 代码相同，将 a 的值增加 b 后返回，但区别是完全没有副作用。当调用 func 1 后函数正确返回 1，但 a 的值仍然为 0，如图 2.3 所示。



```
- val a = 0;
val a = 0 : int
- fun func b = a + b;
val func = fn : int -> int
- func 1;
val it = 1 : int
- a;
val it = 0 : int
```

图 2.3 调用函数 func 后 a 的值不变

无副作用的特点保持了函数的独立性，函数不依赖于外部的状态，也不会修改外

部的状态。有了无副作用的特性，使得函数的单元测试和整个程序的调试都更加简单。

3.6 “无状态”和递归

在命令式编程中，程序是具有状态的，程序的一个运行状态包括各个寄存器的值、所申请的内存空间的位置、大小和其中的数值等等信息。但在函数式编程中强调无状态，无需将状态保存在变量中，而是将状态锁定在函数内部，不依赖任何外部状态。也就是在函数式编程中是通过函数创建新的参数或者返回值，通过这些参数和返回值来保存状态的。在函数式编程中并非真正的无状态，而是将状态完全存放在栈中，对状态的读取和修改是通过一层层函数调用中，利用函数的参数和返回值进行传递。

这样的机制使得函数式编程天生就具有递归特性。函数式编程中没有循环语句，因为循环需要使用变量保存状态来进行控制，取而代之的形式就是递归。如果要遍历一个列表，在函数式编程中就是每次取用列表的首元素，然后对剩余列表使用该函数的递归调用，Standard ML 描述的代码样例如下：

```
(* max: int list->int *)  
fun max [ ] = 0  
  | max (x::L) =  
    let  
      val y = max L  
    in  
      if x > y then x else y  
    end;
```

函数 `max` 的功能是遍历一个元素均为自然数的整型列表中所有元素并找出最大值。可以看到整个函数结构是先用模式匹配取出列表的首元素，然后再递归调用 `max` 函数找到剩余列表元素中的最大值，与当前首元素进行比较，选取较大值作为返回值。这是一个典型的递归思路，递归求解问题可以将问题简单化，只需要完善地考虑问题的一个子问题。

由于在函数式编程中所有的计算都是静态的，所以在递归过程中并没有函数“调用”的概念，编译器会默认所有的递归函数是已知的，结合惰性求值的特点，使得中间开销并不大，程序的性能有所保证。

4 心得收获与建议

4.1 课程收获

在函数式编程原理这门课程的学习中，我学到了函数式编程语言的发展和使用场景，并具体学习了 Standard ML 这一门函数式编程语言。

在逐步摸索 Standard ML 的语法和进行课程实验的过程中，加深了我对函数式编程的理解，逐渐熟练了对 list 结构和 tree 结构编写递归形式的函数进行操作的过程，具体实现了插入排序和归并排序，而 list 和 tree 结构都是非常贴合递归思想的数据结构。

再后来接触到了类型推导、模式匹配、多态类型、高阶函数、函数柯里化等等函数式编程的特点，也通过实验对这些特点进行了实践巩固。最后通过学习实际的找零问题将函数式编程的思想推广到了实际应用的层面，我学会了使用找零问题的思想解决一些类似的问题，即在递归函数中构造两个子问题的分支，对递归过程中的每个状态进行判断并选择进入到哪一个分支之中，还简要介绍了 Standard ML 中的错误处理机制。在课程的最后老师还安排了分享的机会，使我有机会能够和同学们分享我的一些认识和想法，还锻炼了我的表达能力。

总而言之，我在这门课程的学习中受益匪浅，第一次接触到了函数式编程的思想，与此前所学的命令式编程有很大的差异，但是它不仅非常有趣，而且在一些特定的应用场景（比如高并发）以及特定问题（比如斐波那契、树的归并排序）的求解上能够取得非常好的效果，给我提供了一种全新的编程思想。

4.2 建议

老师可以在讲课的时候在 SML 界面中对课件中的一些代码内容进行实际演示，或者对每一章节设计一个小的实践任务，让同学们将课件中提到的一些经典的代码自己实践一遍，会更便于理解一些抽象的问题。