***2020***



**函数式编程原理 课程总结报告**

|  |  |
| --- | --- |
| 题 目： | 5段流水CPU设计 |
| 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](mailto:13456@qq.com) |

目 录

[1 目标与选题动机 2](#_Toc38664117)

[1.1 课设目的 2](#_Toc38664118)

[1.2 设计任务 2](#_Toc38664119)

[1.3 设计要求 2](#_Toc38664120)

[1.4 技术指标 3](#_Toc38664121)

[2 函数式编程的起源和发展过程 5](#_Toc38664122)

[2.1 单周期CPU设计 5](#_Toc38664123)

[2.2 中断机制设计 7](#_Toc38664124)

[2.3 流水CPU设计 8](#_Toc38664125)

[2.4 气泡式流水线设计 9](#_Toc38664126)

[2.5 数据转发流水线设计 9](#_Toc38664127)

[2.6 动态分支预测机制（未完成部分可不写，直接删除） 9](#_Toc38664128)

[3 函数式编程的特点 10](#_Toc38664129)

[3.1 单周期CPU 实现 10](#_Toc38664130)

[3.2 中断机制实现 16](#_Toc38664131)

[3.3 流水CPU实现 16](#_Toc38664132)

[3.4 气泡式流水线实现 16](#_Toc38664133)

[3.5 数据转发流水线实现 17](#_Toc38664134)

[3.6 动态分支预测机制实现 17](#_Toc38664135)

[4 心得收获与建议 18](#_Toc38664136)

[4.1 课程收获 18](#_Toc38664137)

[4.2 建议 18](#_Toc38664138)

[参考文献 20](#_Toc38664139)

# 目标与选题动机

## 目标

主要通过实验验证哪些知识点。

## 选题动机

报告选题的原因、主要拟表达的学习内容和收获、报告提纲挈领摘要。

# 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。

## 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。

### 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StandardML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StandardML最著名的编译器Standard ML New Jersey（SML/NJ）。Caml的具体实现是Ocaml语言，详见本节的第2.1.4小节的介绍。

### 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。

### 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。

### 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。

### 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。

### 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。

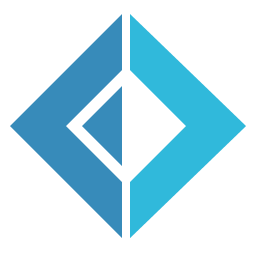
F#语言的标志如下：

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。

## 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。

### 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。

### 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。

### 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。

### 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。

# 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。

## 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。

## 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：

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。

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。

## 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);

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。

## 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。

## 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

## 引用透明

## 递归

# 心得收获与建议

## 课程收获

基于。

## 建议

本

# 参考文献

1. DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第4版).北京：机械工业出版社.
2. David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社
3. 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011年.
4. 谭志虎，秦磊华，胡迪青.计算机组成原理实践教程.北京：清华大学出版社，2018.
5. 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011年.
6. 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008年.

|  |
| --- |
| 一、原创性声明 |
| 本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。  特此声明！  **作者签字:** |