前言

函数式编程原理，是整个大学生涯中带给我启发最大的一门课程。它给人带来了完全不同的角度去审视编程的艺术，因为引入了函数（Function）的以及纯函数（Pure Function）的概念:

```
1.函数不会产生side effect(no side effect)
2.函数满足referential transparency
```

使得编程时保证了函数在任何时候传递相同参数时，得到的结果都相同。这个特性可谓是在构建大型工程时最为优异的性质。

为了方便以后反复翻阅来巩固，全文主要围绕着函数式编程的起源、发展以及SML编程语法规范来进行撰写。当然也会穿插其他相关知识图谱，来对其知识框架进行补充。

全文同步更新到我的个人博客主页HUST-CS1703 SML，以后也会继续迭代更新，欢迎共同学习进步。

一.函数式编程概述

如果仅仅从历史故事角度去讨论函数式编程起源和发展，可能会显得十分冗长与无聊，Google上也有大把的文章去罗列其背景故事，如果想看故事的话，推荐知乎上的函数式编程的早期历史和一篇博主的分享理解函数式编程，这里就不过多的赘述了。以下主要是围绕其数学层面上的起源和发展历程。

范畴论

函数式编程的起源，是一门叫做范畴论（Category Theory）的数学分支。

```
"范畴就是使用箭头连接的物体。"
In mathematics, a category is an algebraic structure that comprises "objects"
that are linked by "arrows".
```

也就是说，彼此之间存在某种关系的概念、事物、对象等等，都构成"范畴"。随便什么东西，只要能找出它们之间的关系，就能定义一个"范畴"。

箭头表示范畴成员之间的关系，正式的名称叫做"态射"（morphism）。范畴论认为，同一个范畴的所有成员，就是不同状态的"变形"（transformation）。通过"态射"，一个成员可以变形成另一个成员。

这也引申出了函数式编程中的基本数学模型：

- 所有成员是一个集合
- 变形关系是函数

可以发现，范畴论是集合论更上层的抽象，简单的理解就是"集合 + 函数"。理论上通过函数，就可以从范畴的一个成员，算出其他所有成员。

有了这个数学模型后，我们可以把把"范畴"想象成是一个容器，里面包含两样东西：

- 值（value）
- 值到值的变形关系，也就是函数（function）

范畴论使用函数，最初只是用于数学运算，后来有人将其在计算机上实现，就逐渐变成了今天的函数式编程。

这里引用一句阮一峰对函数式编程的概括：

> 本质上，函数式编程只是范畴论的运算方法，跟数理逻辑、微积分、行列式是同一类东西，
> 都是数学方法，只是碰巧它能用来写程序。

函数式 VS 命令式

- 函数式编程是一种编程范式，我们常见的编程范式有命令式编程，函数式编程，逻辑式编程，常见的面向对象编程和面向过程编程都是命令式编程。
- 命令式编程是面向计算机硬件的抽象，如变量(抽象存储单元)，表达式(算数运算与内存读写)，控制语句(跳转指令)，最终得到一个冯诺依曼机的指令序列。

函数式编程的基础模型来源于λ演算，是阿隆佐思想的在现实世界中的实现。不过不是全部的lambda演算思想都可以运用到实际中，因为lambda演算在设计的时候就不是为了在各种现实世界中的限制下工作的(毕竟是数学家捣鼓出来的东西)。 目前的函数式编程语言基本都是翻译为冯诺依曼指令实现的。

从另外一个角度来说（关于变量和函数）：

变量：

- 命令式: 代表存储可变状态的单元(内存地址)，相当于地址的别名 x = x + 1
- 函数式: 代表数学函数中的变量，映射到某个值，相当于值的别名 2x = 4

函数：

- 命令式: 描述求解过程(怎么做)，本质上是一系列的冯诺依曼机指令，can do anything
- 函数式: 数学概念里的函数，描述映射(计算)关系(做什么)，也称为纯函数/无状态函数

函数式编程特性

- 不可变语法义：没有可变状态，也就没有 for, while 循环，使得函数式编程严重依赖递归。
- 纯函数：相同输入得到相同输出，且函数没有语义上可观察的副作用
- 尾递归：一个函数里的最后一个动作是返回一个函数的调用结果的情形，即最后一步新调用的返回值直接被当前函数的返回结果。
- 惰性求值/乱序求值：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。
- 柯里化：也是课程中重点介绍的性质，指将一个多参数函数分解成多个单参数函数，然后将单参函数多层封装起来，每层函数都返回一个函数去接收下一个参数，这可以简化函数的多个参数。
- 高阶函数：以函数为参数或返回值的函数，有了高阶函数，就可以将复用的粒度降至函数级别，相对面向对象而言，复用粒度更低。
- Monad：函数式处理可变状态的一种方法。

函数式优点 VS 缺点

优点：

- 并发性: 函数无副作用(天然可重入)，原生并发友好
- 确定性: 可读性高，易于测试和调试，错误易于重现
- 没有锁和指针
- 具有很大的优化潜力，如惰性求值，并发，缓存函数计算结果等，很多原本需要程序来做的事情，都可以由编译器来做。比如动态规划的缓存，MapReduce 等。

缺点：

- 处理可变状态如 IO 的能力弱
- 为了维持不可变性，拷贝的开销
- 运行效率，依靠并发

二.Standard ML编程语法规范

SML语言的详细语法教程十分稀少，而且大多是纯英文版本，所以在此整理出一篇中文版的SML语法规范。以后随着更深入的了解，也会不断迭代更新。

基础语法

注释

```
(*注释内容*) 在SML中没有单行注释//` 与块注释/* */
```

运算

- 元操作符没有+和-，表示负数应该使用~
- 整数除法div（除术算法），实数除法/，取余(模)mod（不是%）
- 函数运用比中缀运算符优先级更高。（SML中函数也可以视为运算符）

ML标准类型

- 基础类型：unit、int、real、bool、string

类型	值	操作数
real	3.14, 2.17, 0.1E6, . . .	+, -, *, /, =, <, . . .
char	#“a”, #“b”, . . .	ord,chr,=, <, . . .
string	“abc”, “1234”, . . .	^, size, =, <, . . .
bool	true, false	if exp then exp1 else exp2

- 表:int list,(int -> int) list
- 元组: int*int ,int*int*real.....
- 函数: int ->int, real -> int *int.....

所有对象的类型不一定需要显示说明，可以通过上下文推导出来

类型断言(*typing assertion*)： `exp : typ`（*exp*表达式，*typ*类型），例如：

```
3:int
3+4:int
4div3:int
4mod3:int
```

类型约束：

所有对象的类型不一定需要显示说明，可以通过上下文推导出来。例如：

```
3+3.14 (*这里的3虽然看起来是int，但是根据上下文可知3是real型*)
```

变量、作用域

- 变量

```
val x = 1;
val s = "Hello,world";
```

通过val来声明，不需要注明类型。

- 作用域 ML语言的作用域是静态的、词法的(*static, lexical*)，与C语言类似。 声明的变量、类型的作用域具有全局的作用域(*Global Scope*)，即剩下的代码块都是它的作用域。

绑定

- 类型绑定 一般格式：$type \cdot typcon1 = typ1$

```
type float = real
type count = int and average = real
```

称float为type constructor，上面type bindings语句中引入了type constructor float；

- 变量绑定

一般格式：val ide : typ= value

```
val pi:real = 3.14
```

value的类型必须和所绑定的变量一致，变量的命名即为标识符ide

声明

- 限制作用域

格式和作用 let语句：let dec in exp end; dec的作用域限制在exp内，当exp执行完，dec即销毁 local 声明：local dec in dec0 end;

```
let
    D
in
    E1;
    E2;
    ...;
    En
end
```

- 同名变量

覆盖原则同所有静态作用域的语言一样（如C、Java等）

```
val m : int = 2
val r : int =
  let
    val m : int = 3
    val n : int = m*m
  in
    m*n
  end * m
```

最终结果为54

函数

基础数学模型：$f(x) = x + 1$

- 函数类型说明

  格式：`typ -> typ'` 说明：*typ* 指参数的类型（*domain type*），*typ'* 指结果的类型（*range type*）

- 声明方式

  val 声明法，格式为`val var1 : typ1 = exp1`

  fun声明，格式：`fun fname (var:dtyp):rtyp = fexps`

  （说明：`fname` 函数名，`var`参数，`dtyp`参数类型，`rtyp`返回值类型，`fexps`操作表达式其结果为返回值）

```
fun mult[] = 0
    | mult(x::L) = if L=[] then x else x *(mult L);

fun Mult[] = 0
    | Mult(x::L) = if L=[] then mult(x) else mult(x) *(Mult L);
```

异常

机制、目的以及思想和其他语言(如Java)中的异常几乎没有什么不同，所以我们重点关注异常在ML语言中的语法特性。

本质：ML语言中的内置类型exn(关于数据类型详见(四)数据类型) 特性：1).exn类型的构造子集合可以通过声明语句被扩展　　2).异常包不是ML的值，能识别它们的只有raise和handle

- 声明

  `nullary exception` $exceptionEx$ 其中exception为ML关键字，Ex为异常的名字(名字通常首字母大写) `value-carrying exception` $exceptioneExoftyp$

```
exception Failure
exception Failedbecause of string;
exception Badvalues of int;
```

- 抛出异常

  $raiseEx$

  其中Ex为异常e或者值为exn类型的表达式，作用就是计算出一个含值e的异常包。

- 处理异常

  类似于case表达式

```
E handle E1 => e1
(|E2 =>e2
......
En=>en)
```

参考

1.sml基本语法—异常

2.函数式编程的早期历史

3.理解函数式编程

4.阮一峰的网络日志

5.The Standard ML Basis Library