have fun!

第一类值（first-class value）

函数式编程范式本身主要思想来自于数学，λ演算是其重要的理论基础，所以函数式编程可以几乎很简单的就能直接表示数学问题而不必进一步抽象为命令式程序设计语言的是算法。

函数式编程范式是与值打交道，而不是与状态打交道。所以函数式编程中，一切都是值，而处理值的基本工具就是表达式而非命令（语句）。这是相对于命令式语言最大的一个不同。

表达式主要是由函数调用组成，而函数则可以应用任何类型的参数，包括函数本身。函数也是值，可以参与各种运算，因而大大简化了解决问题的方式。

高阶函数与闭包（high-order function & closure）

正是由于函数可以作为值来使用，就有了很多高级的应用。

一些可以支持以函数作为参数的函数，被称之为高阶函数。下面是 Scheme 语言中定义的两个函数，函数 fact 使用递归的方法来实现阶乘运算，函数 map 则是高阶的映射操作：将一个作为其参数的函数 F 作用于作为其参数的列表 L 的每一个元素之上，从而生成一个新的列表。

```
1  (define (fact x)
2      (if (= 1 x)
3          1
4          (* x (fact (- x 1)))))
5
6  (define (map f l)
7      (if (null? l)
8          '()
9          (cons (f (car l)) (map f (cdr l)))))
10
11 (map fact '(1 2 3 4 5))
12 ;#=> '(1 2 6 24 120)
13
```

（代码 1[scheme]：null?用于判断函数是否列表是否为空，car 返回列表头部元素，cdr 返回除列表头部元素之外的所有部分组成的列表，cons 则是用于构造新的列表）

在这段程序中，函数 fact 作为 map 函数的实参传入进函数，通过一系列的 cons/car/cdr 操作和 map 的自身递归，得到一个新的包含旧表的元素阶乘的列表'(1 2 6 24 120)。

函数不仅可以作为参数传递，还能够作为返回值。看以下代码：

```scheme
(define (plusN n)
    (lambda (x)
        (+ x n)))

(define plus3 (plusN 3))

(plus3 5)
;#=> 8
```

（代码 2[scheme]：plusN 函数返回一个可以接受数字并返回其值+n 的函数，这样就直接相当于一个构建工厂，可以一步步的产生一系列相似并且实用的函数，lambda 会产生一个匿名函数，其作用我们会在后面叙述）

没看出来哪里实用？那你错了。这就是**闭包**，Design Pattern 里面啰嗦了多久才能把一个工厂模式讲清楚，试着去看一下他理解一下，再回来看这段代码吧。

匿名函数（lambda expression）

前面提到过，函数式程序设计范式的一个重要理论基础是 λ 演算（lambda calculus），在很多函数式编程语言中，lambda 表达式（或者是匿名函数，因为有些关键字也不一定非得用 lambda）就成了标配。

```scheme
(define (map f l)
    (if (null? l)
        '()
        (cons (f (car l)) (map f (cdr l)))))

(map (lambda (x)
        (* (fact (+ x 1)) (fact x)))
     '(1 2 3 4 5 6))

;#=> '(2 12 144 2880 86400 3628800)
```

（代码 3[scheme]：lambda 表达式通过 map 作用列表之上）

表面上看，匿名函数的用处也不大。如果没办法通过名称来区分函数的话，要他还有什么用？但是，试想想，总有些东西其实并没必要保存多么持久，用过一次就可以弃掉的。比如 C 语言标准库 stdlib.h 中的 qsort 函数，每次都要单独定义一个比较函数，将函数指针传递给它才能够正常工作，而如果 C 语言支持字面级的（literal）函数/匿名函数的话，完全就没必要多浪费这么一部份工作了。

另外 lambda 函数作为函数的返回值也是一个比较广的应用（见代码 2）。

（高阶函数应用如柯里化（Currying）等目前暂不提及。）

副作用和引用透明（side effect & reference transparency）

前面提到过，函数式编程是针对值而非针对状态的，而函数的执行过程也就是对表达式求值的过程。命令式语言中，表达式可以有副作用（side effect），如下表达式：

$$f(u)+t$$

和：

$$t + f(u)$$

则并不一定会返回同样的结果，因为 f(u)可能会影响到 t 的值。

函数式编程则不必担心这种情况。函数式编程只进行表达式的计算，而表达式则是简单的求值，对于认给定的 t、u，返回的结果都是固定的，并且不会对 t，u 进行任何破坏性的改变（因为那样会影响到程序的状态）。如代码 1 的第 9 行，传入的表并没有任何改变，只是新生成了一个另外的包含计算结果的表。

命令式编程语言不同之处在于其每一条命令都有可能涉及到状态的改变，如迭代循环，每一步都会对迭代器/迭代变量进行判断（状态）。而这种影响到程序状态的情况很容易导致一些莫名的错误。

尾调用优化（tail-call elimination）

先看另外一个阶乘函数，并与代码 1 中的 fact 比较：

```
1  (define (fact x)
2      (define (facti n m)
3          (if (= 0 n)
4              m
5              (facti (- n 1) (* m n))))
6      (facti x 1))
7
8  (fact 100)
9  ;#=>
   933262154439441526816992388562667004907159682643816214685929638952175999932299 15
```

（代码 4[scheme]：尾递归形式的阶乘函数，效率较普通递归大大提升）

所谓尾调用，指的是一种类似于 goto 的调用。当一个函数调用是另一个函数的最后一个动作时，该调用被称为尾调用。

这样一个尾调用进行时，“尾调用”所在的函数并没有别的动作可以执行，那么理论上可以不用回到之前那个状态，直接继续进行就可以了。所以“尾调用”进行后，不必保存执行调用的函数的任何栈信息，这样尾调用也就不会耗费多余的栈空间，而正是这个特点，一个程序可以使用无数的尾调用，而不必担心栈溢出（stack overflow）。

尾调用应用到递归中，则被称为尾递归。因为调用和返回栈的优化，完全不用考虑通常递归的效率问题，从而直接把“递归”换成了另外一种形式的“迭代”，同时还保持了函数设编程的风格和特性。

类型推导（type inference）

绝大多数函数式编程语言是动态类型或者是基于类型推导和多态的静态类型。动态类型的优点和缺点自不必言喻，自由度和舒适程度的提升伴随着执行效率下降和查错的难度增大，是得失兼有。而静态多态类型则是另外一个对于动态类型和静态类型的相互取补。

多态类型检测提供了一个相对于动态类型和静态类型之间的新立场：首先，静态检查可以提供安全性和高效的查错；其次，因为多态的加入，灵活性大大优于静态类型语言。而且

因为由于类型推导机制，所以很多时候没必要在程序中堆砌类型声明，大多数都是可以由编译器/解释器自动推导出来。

如下代码：

```haskell
let f x = x + 1
#=> f :: (Num a) => a -> a
```

（代码 5[haskell]：第一行是一个函数定义，第二行是编译器推导的该函数的参数类型和结果类型）

在代码 5 中，因为编译器对函数 f 解析的过程中一步步的类型推导，确定了 f 只能接受数字（Num）作为参数同时返回数字的函数类型。

模式匹配（pattern matching）

这里的模式匹配没有正则表达式那么强大，但是却仍然是函数式范式一个十分重要的特性。

```haskell
let xsum [] m = m
let xsum (x:xs) m = xsum xs (x+m)
let sumList l = xsum l 0

sumList [1,2,3,4,5]
#=> 25
```

（代码 6[haskell]：Haskell 中的模式匹配实现对列表元素求和/同时使用了尾递归的特性）

没有了 car 和 cdr，看起来确实清爽了许多，而且也在某种程度上简单清晰了许多，如果说 Haskell 的模式匹配还只是这么简单的用于函数参数列表和 Guard，那么 Erlang 的则是无处不在，无处不匹配。

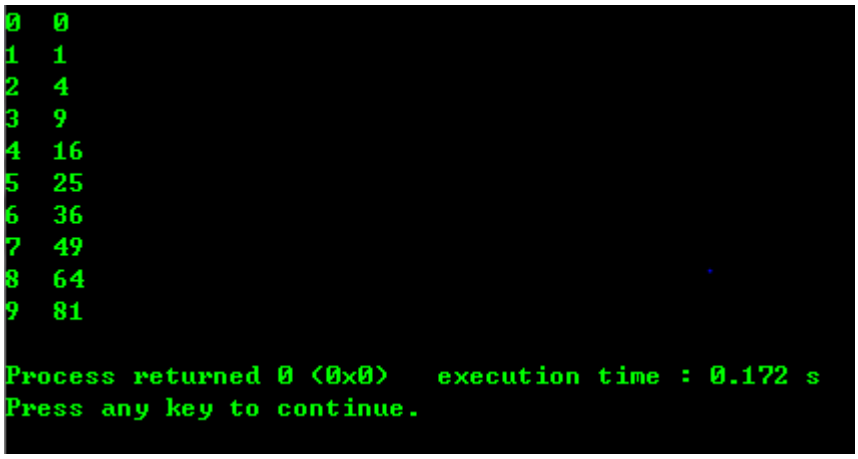
并发原语（concurrency）

并发原语 send、receive 和 spawn 来自于 Erlang。通过这样简单的消息传递模式可以实现大量高效的并发线程：下面是并发原语在 D 语言中的应用：

```d
import std.stdio;
import std.concurrency;
void fun(){
    int m,n;
    while(1){
        receive(
                (int x){m = x;}
        );
        if (m == 99)
            return;
        else
            for(n=0;n<10;n++)
                if(n*n == m)
                    writeln(n,"  ",m);
    }
}
void main(){
    auto tid = spawn(&fun);
    int i = 0;
    while(i++ != 100){
        send(tid,i);
    }
}
```

（代码 7[dmd]：D 语言对并发原语的支持）

屏幕输出结果为：



以上就是函数式编程的几大特性及其为程序设计语言范式带来的一些优秀基因。新生代语言中，绝大多数都正在向着函数式程序设计语言靠拢，而较之于面向对象相对程序设计的提升，函数式范式则又是一个更为重要的进步。

参考

《ML 程序设计教程（原书第二版）》 Paulson L. C.著；柯韦译. 机械工业出版社，2005.5
《计算机程序的构造与解释（SICP）》

后记与附注

待加入内容

1、高级数据结构：表（List）、序对（Pair）、元组（Tuple）、向量（Vector）和自定义数据类型。

2、简洁的语法结构：

a) 空白和括号作为语法分隔符=>可以保留很多的符号来用作其他方面。

b) 前缀表达式为主，无需考虑过多的优先级和结合性问题。

内容改进

1、从针对语言改成针对范式。

2、减少盲目的攻击性。

3、强调思想而非内容。、

4、加入更多的数学性质的东西。

深入研究方向

1、std.concurrency.d(D 语言并发类库)

2、FC++（Functional C++类库）

3、C++ Meta Programming

小记

本来准备有一段时间在《天方夜谭》停止之后能够找一些东西继续，曾经拟出一个题目来叫做《理想主义者的程序设计语言发展史》（简称 HOPLi）。

目的不是在于要再度写一个历史性的东西什么的，更多的是想纠正好多人对于绝大多数程序设计语言的错误认识（JavaScript 应该是最不幸的一个）。可是，半途就没办法继续下去了，无论是从知识层面还是认识层面都还没能够达到相应的水平来写下来这么有深度的一个东西。

但是当时因为各种原因对老师讲课的内容不满：一个正宗的 MS ED（微软系的 Enterprise Developer）来通过 C++讲面向对象基础，自我感觉其内容的深度和广度都尚不如我一两前的认知水平。果断决定要拿出一些东西来给他看一下，用什么来证明他所坚持的东西其实并不怎么样。

但是后来就绝望了。发现这根本就是徒劳的时候，这篇内容已经完成到这个地步了，在上课和一个同学神游的时候发现自己有种回归数学的感觉，而且本来就在慢慢的深入计算机科学方面，所以不如这样子慢慢的一边深入的研究数学与证明，一边研究程序设计语言发展史，一边完成 Have Fun，一边完善 HOPLi。

嗯，这有难度。