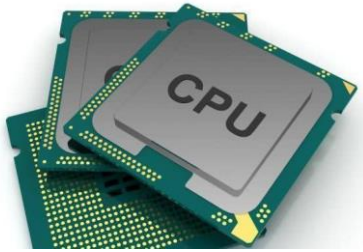
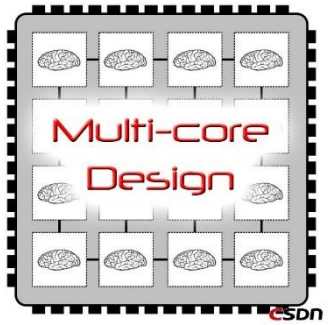
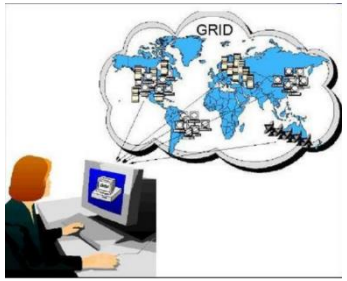
函数式编程原理

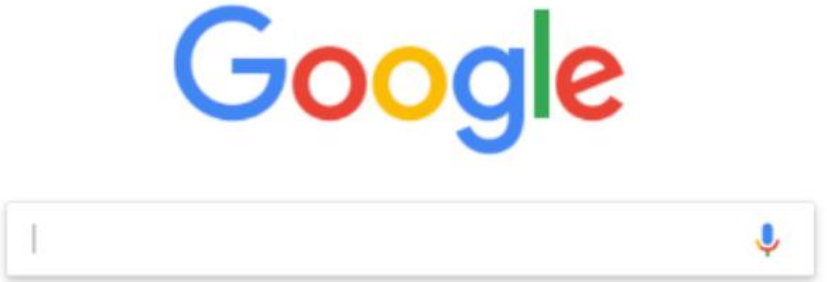
郑　然

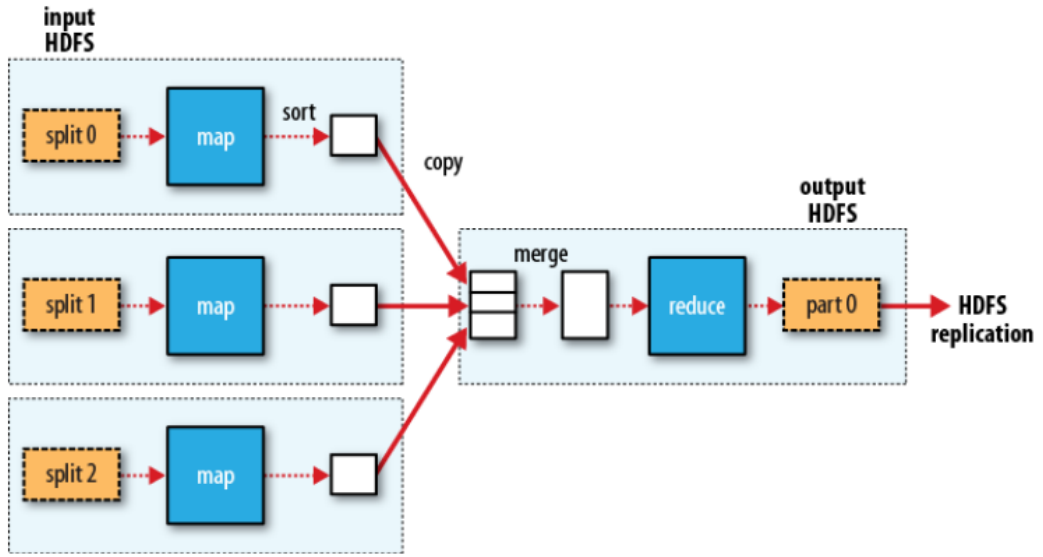
分布式计算 与 MapReduce







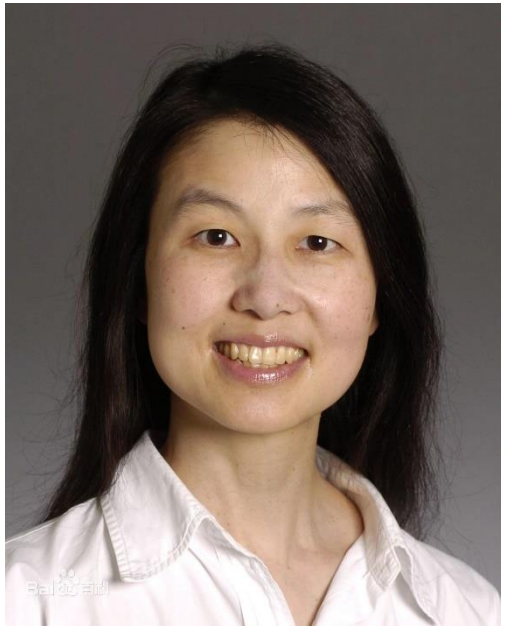




My heart is in the work

做一只特立独行的——狗

Carnegie Mellon.
TARTANS

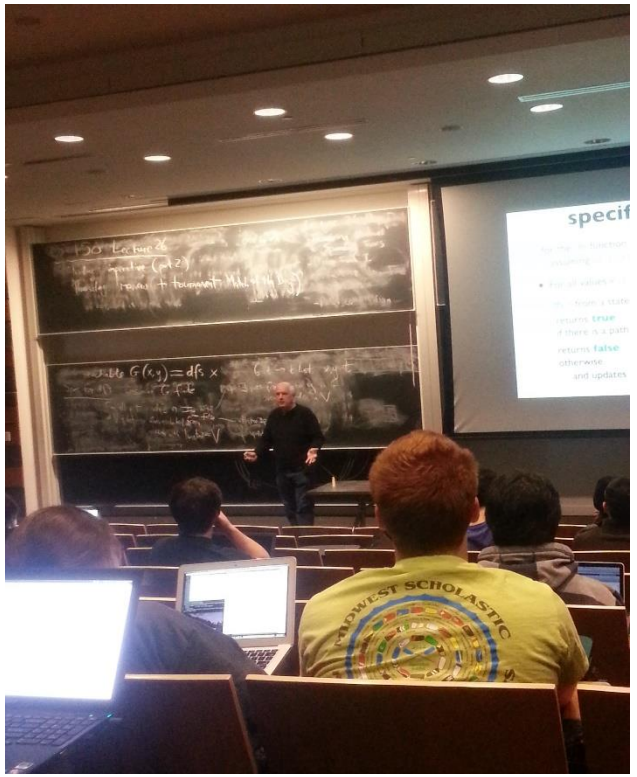


周以真

计算思维 (2006)

CMU 15-150 Functional Programming

- Lectures: Twice every week (12:00-13:20)
- Homework:
  - Once every week (Electronic submission)
  - Lateness, Compilation, Validity
- Lab: Once every week
- Taking Notes
  - by writing or typing
  - Do not record or tape lectures electronically, whether by audio or video.
- Self-studying time: More than 8 hours every week
- Tools: Andrew Unix, Git, Emacs/VI, LaTex, Piazza, SML/NJ, ……
- Grading Policy: Homework 40%, Labs 10%, Midterm 20%, Final 30%



函数式编程原理

- 授课方式：讲授+上机
- 学时要求：　16+16
- 涉及内容：
  - 程序正确性/有效性证明
  - 类型, 声明, 表达式, 函数
  - 递归，模式匹配，多态类型检测
  - 高阶函数，惰性求值
  - ……

- 课程目标：
  - 掌握函数式编程方法
  - 掌握程序书写规范，并采用严格的推导方法证明程序的正确性
  - 掌握串/并行程序的性能分析方法
  - 掌握各种数据结构的特点，学会选择合适的数据结构进行功能设计，提高程序效率

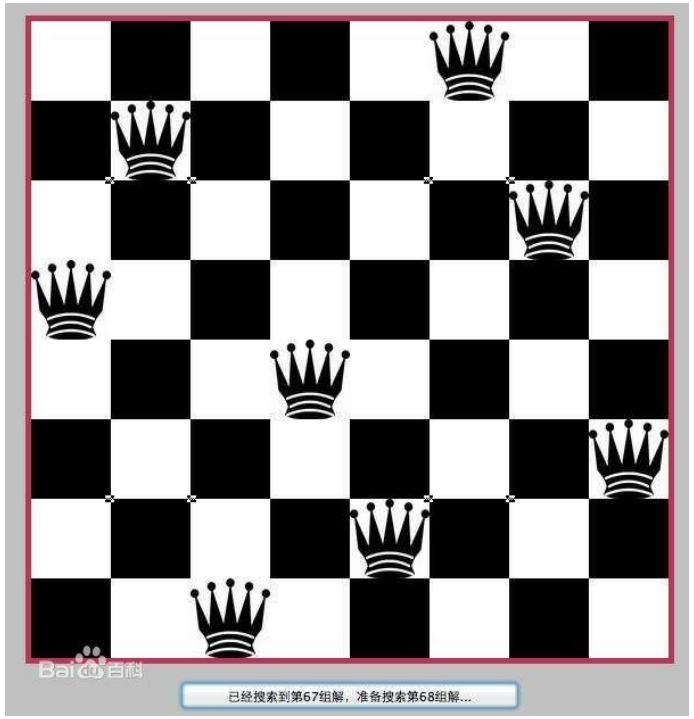
  - 能看的懂函数式代码
  - 能编写简单的ML函数式程序
  - 能分析串/并行程序性能

八皇后问题

- 在8x8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法？

在8!=40320种不同行/列/斜线的排列中共有92种解决方案：

从第0列开始，逐列进行搜索，找到一个不受任何现有皇后攻击的位置。如果找不到安全位置，则后退一步(改变前一个皇后的位置)重新查找，直到找到安全位置……



学习资料及工具

- CMU 15-150：
  http://www.cs.cmu.edu/~15150/index.html
- Programming in Standard ML, CMU
- ML for the Working Programmer，PAULSON, LAWRENCE C. (Univ. of Cambridge, Cambridge, UK)
- ML程序设计教程(第2版)，机械工业出版社

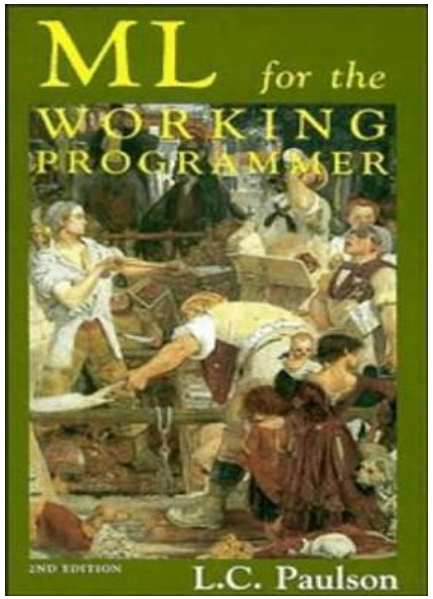
- 编程环境及语言： SML of New Jersey (SML/NJ)
  - Welcome! - SML Help
    https://smlhelp.github.io/book/index.html

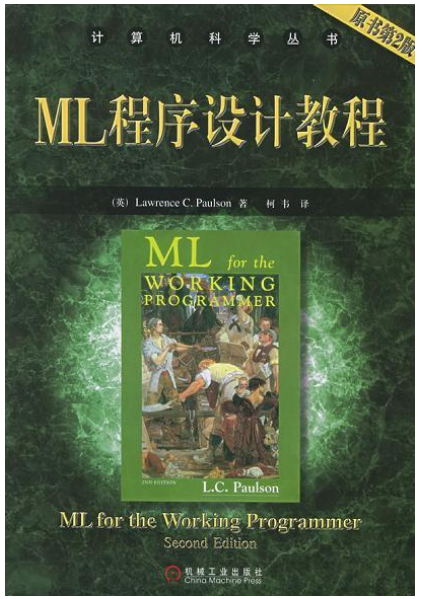
Programming in Standard ML

(DRAFT: VERSION 1.2 OF 11.02.11.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2011





成绩评定

- 平时与作业：30%
- 上机实验：30%
- 结课随堂测：10%
- 结课报告：30%

学习平台

- 头歌链接：https://www.educoder.net/classrooms
- 邀请码：2MJ37

今天的主要内容

- 函数式语言的发展历程
- 函数式语言家族成员
- 函数式语言的定义
- 函数式语言的特点

几种编程语言

Pascal、C

命令式语言，基于动作的语言，以冯诺依曼计算机体系结构为背景
通过修改存储器的值产生**副作用**的方式去影响后续的计算
Fortran、ALGOL、Pascal、C　　**关注的焦点：如何进行计算**

Java、C++

面向对象语言，以对象作为基本程序结构单位，提供类、继承等成分
Smalltalk、Simula、Java、C++

Prolog

逻辑式语言，建立在逻辑学的理论基础之上，广泛应用在人工智能的研究中，可以用来建造专家系统、自然语言理解、智能知识库等

LISP、SML

函数式语言，数学函数是基础，没有内部状态，也没有**副作用**
程序的执行是表达式的计算
LISP、Scheme、ML、Haskell、Ocaml　　**关注的焦点：计算什么**

判定丢番图方程的可解性问题

- “希尔伯特23问”（Hilbert 23 Problems）的第10个问题（1900年）
  - “给定一个系数均为有理整数，包含任意个未知数的丢番图方程：设计一个过程，通过有限次的计算，能够判定该方程在有理数整数上是否可解”
- 形式化描述：数理的完整性、一致性和可判定性（1928年，Principles of Mathematical Logic）
  - 通过有限次的步骤，对数学函数进行有效计算的方法——递归函数、可计算性理论

大卫•希尔伯特
(哥廷根大学)

计算机科学的理论基础

图灵机和λ演算

1936年，针对数理的可判定性问题的否定答案

- 以λ演算为基础：“An unsolvable problem of elementary number theory”　　一切皆函数

阿隆左•丘奇 (美国)

- 图灵机模型：“On computable numbers, with an application to the entscheidungs problem”

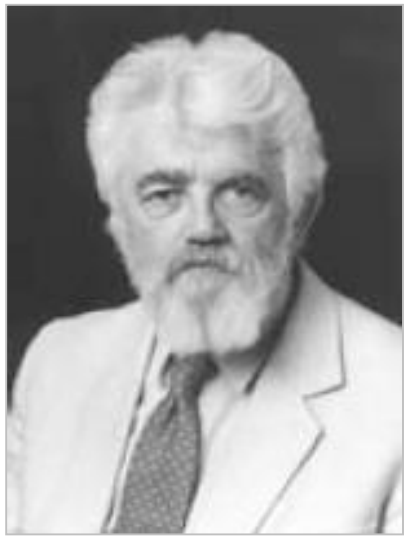
——计算机的基本理论模型

冯•诺依曼架构、历史上第一台电子计算机

阿兰•图灵 (英国)

函数式编程

- 不依赖于冯·诺依曼体系结构的计算机
- 设计的基础：数学函数
  - 程序的输出定义为其输入的一个数学函数
  - 没有内部状态，也没有副作用
- 第一个函数式语言：LISP
  - 初始动机：用于人工智能研究的表处理语言
  - 实现λ演算理论，采用符号表达式定义数据和函数，采用抽象数据列表与递归符号演算衍生人工智能



约翰•麦卡锡
(麻省理工)

几个函数式编程语言

- ISWIM(If you See What I Mean)(1966年)
  - 以函数式为核心的指令式语言，函数由Sugared λ 演算组成
  - 没有完全实现，在一定程度上奠定了函数式语言设计的基础
- ML语言(1973年)
  - 具备命令式语言特点的函数式编程语言灵活的函数功能，允许副作用和指令式编程的使用
  - 有并行扩展，可以用来写并行系统
- Standard ML语言(20世纪90年代)
  - 高阶函数、I/O机制、参数化的模块系统和完善的类型系统
  - 交互式编译器，SML/NJ最著名

彼得•兰达
(牛津大学)

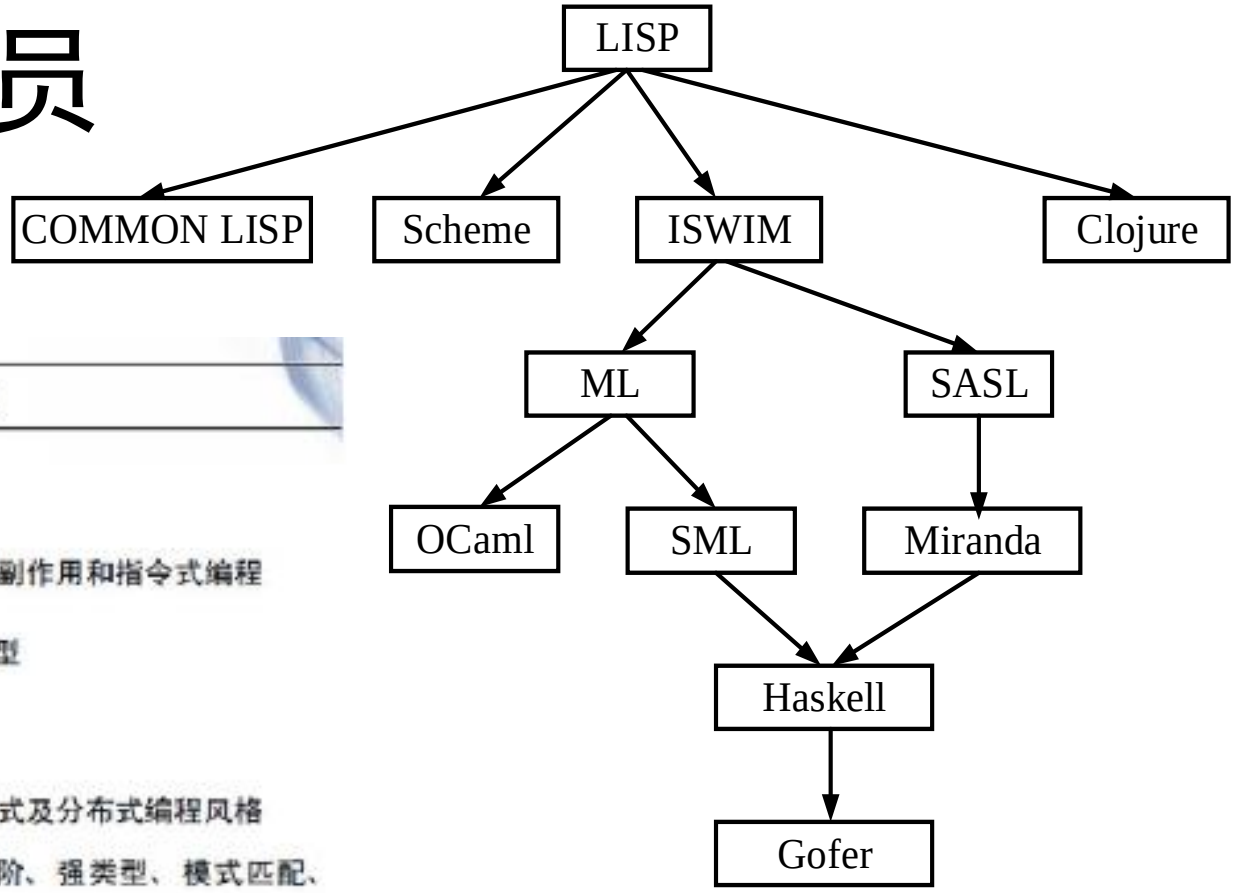
罗宾•米尔纳
(爱丁堡大学)

FP语言：开创纯函数语言研究之先河

- 1977年， ACM年会上获得计算机界最高奖：图灵奖
  “Can Programming be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs”
- 取消了变量，用固定的泛函数结合和一些简单定义作为从现存函数构造新函数的唯一工具
- 不是最早的函数式编程语言，但发明了Functional programming这个概念

约翰•巴克斯(美国)

函数式语言家族及主要成员



时间	名称	提出者/或提出机构	特征
1958	Lisp（包括 Common Lisp 和 Scheme）	McCarthy J	表处理语言，适合于数学运算
1979	ML（包括 Standard ML 和 Caml）	Milner R	非纯函数式编程语言，强类型，允许副作用和指令式编程
1985	Miranda	Turner D	基于 ML 的惰性纯函数式语言，强类型
1987	Clean	Radboud University Nijmegen	惰性高阶纯函数式语言，强类型
1987	Erlang	Erlang A K.	多范式编程语言，支持函数式、并发式及分布式编程风格
1988	Haskell	Hudak P, Wadler P	纯函数式语言，支持惰性求值、高阶、强类型、模式匹配、列表内包、类型类和类型多态
1996	OCaml	Leroy X, Vouillon J	非纯函数式编程语言，强静态类型
2001	Scala	Odersky M	多范式编程语言，支持对象式和函数式编程的典型特性
2002	F#	微软	基于 OCaml 的非纯函数式编程语言，强静态类型，适合程序核心数据多线程处理
2007	Clojure	Hickey R	针对 JVM 平台且基于 Lisp 的动态函数式编程语言，支持高阶函数和惰性计算

函数式编程语言的兴衰

- 2000年以前的没落
  - 自身缺点：慢，消耗更多的资源
  - 外界因素：面向对象编程的崛起
  - 一直被学术界重视，很少应用到业界
- 2000年以后的兴盛
  - 摩尔定律失效，芯片工艺限制
  - 多核计算机的诞生，并发与数据共享需求
  - 命令式编程的天生缺陷 vs. 函数式编程的优势
  - 走出实验室，为业界所用

  - 工业和商业应用，如符号数学、统计、金融分析等

图灵机 vs. λ演算
命令式语言 vs. 函数式语言

- 图灵机和命令式语言
  - 遵循图灵机模型，核心概念是命令
  - 通过修改存储器的值而产生**副作用**的方式去影响后续的计算
  - 命令式程序的“函数”有副作用，如改变全局变量

函数或表达式计算时除了有返回值之外，还修改了某个状态或与调用它的函数或外部环境进行了明显的交互

X=X+1

- λ演算和函数式语言
  - 一切皆函数，即用函数组合的方式描述计算过程
  - 避免繁琐的内存管理，没有内部状态，也没有副作用
  - 函数的结果仅仅依赖于提供给它的参数

Y=X+1

什么是函数式编程

- In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

  --From wikepedia

- 函数式编程是一种编程模型，它将计算机运算看作是数学中函数的计算，并且避免了**状态**以及**变量**的概念。

函数式编程语言的特点

- 函数是“first-class values”
  - 程序、函数和过程可以使用数学意义下的函数来表示

    $y = f(x)$　　//变量y就是值$f(x)$的一个名字
    - 程序、函数和过程之间没有区别，函数可以作为数据来使用
    - 变量总是代表实际值，变量没有存储位置和左值的概念，没有赋值操作，只有常量、参数和值

$$x = x + 1$$

没有意义

函数式编程语言的特点

- 引用透明性
  - 任何函数的值只取决于它的参数的值，而与求得参数值的先后或调用函数的路径无关

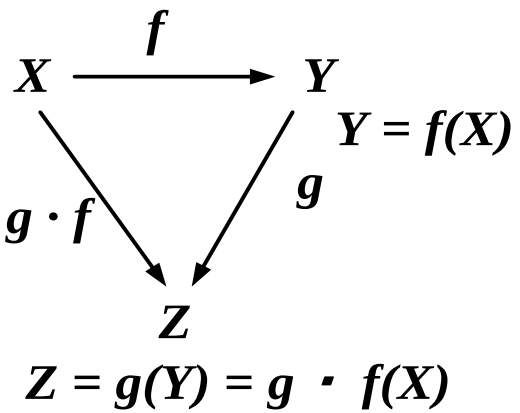
```
int x = 10;
int u = 0;

int f() {
  u++;
   return x+u;
}

int g() {
  return f()+u;
}
```

副作用，速度快

- 表达式求值，避免繁琐的内存管理，没有内部状态，也没有副作用
- 变量只是一个名称，而不是一个存储单元
- 变量只能被定义一次，将反复拷贝大量数据

函数式编程语言的特点

- 高阶函数（Higher-Order Function）
  - 函数可以不依赖于任何其他的对象而独立存在，可以将函数作为参数传入另一个函数，也可以作为函数的返回值
  - 两个基本运算：合成和柯里化



把接受多个参数的函数变换为接受一个单一参数（最初函数的第一个参数）的函数，并返回接受余下参数而且返回结果的新函数

```
real add(real x, real y, real z) {
    return x+y+z;
}
```

```
fun plus x y z: int = x + y + z
```

- 提高函数的实用性，扩大适用范围，提前把易变因素传参固定下来
- 函数会延迟执行（惰性求值）

函数式编程语言的特点

- 惰性求值（延迟计算）与并行
  - 当调用函数时，不是盲目的计算所有实参的值后再进入函数体，而是先进入函数体，只有当需要实参值时才计算所需的实参值（按需调用）

```
int a = f(x);
int b = g(y);
int c = a + b;
```

```
int a = f(x);
int b = g(y);
if x == 0  c = a   else c = b;
```

- 用数学方法分析处理代码
  - 抵消相同项、避免执行无谓的代码
  - 重新调整代码执行顺序、效率更高
  - 重整代码以减少错误
- 不能处理I/O, 不能和外界交互

函数式编程

- 函数式编程是一种编程模型，它将计算机运算看作是数学中函数的计算，并且避免了**状态**以及**变量**的概念
  - 函数是一等公民
  - 引用透明性
  - 高阶函数
  - 惰性求值（延迟计算）

函数式编程语言的特点

- 递归调用及其优化

【例】返回整数i和j之间的所有整数的和。

$$sum(i, j) = i + (i+1) + ... + (j-1) + j$$

```
int sum(int i, int j) {
 int k, temp=0;
  for(k=i; k<=j; k++)
      temp+=k;
  return temp;
}
```

（a）命令式循环版本

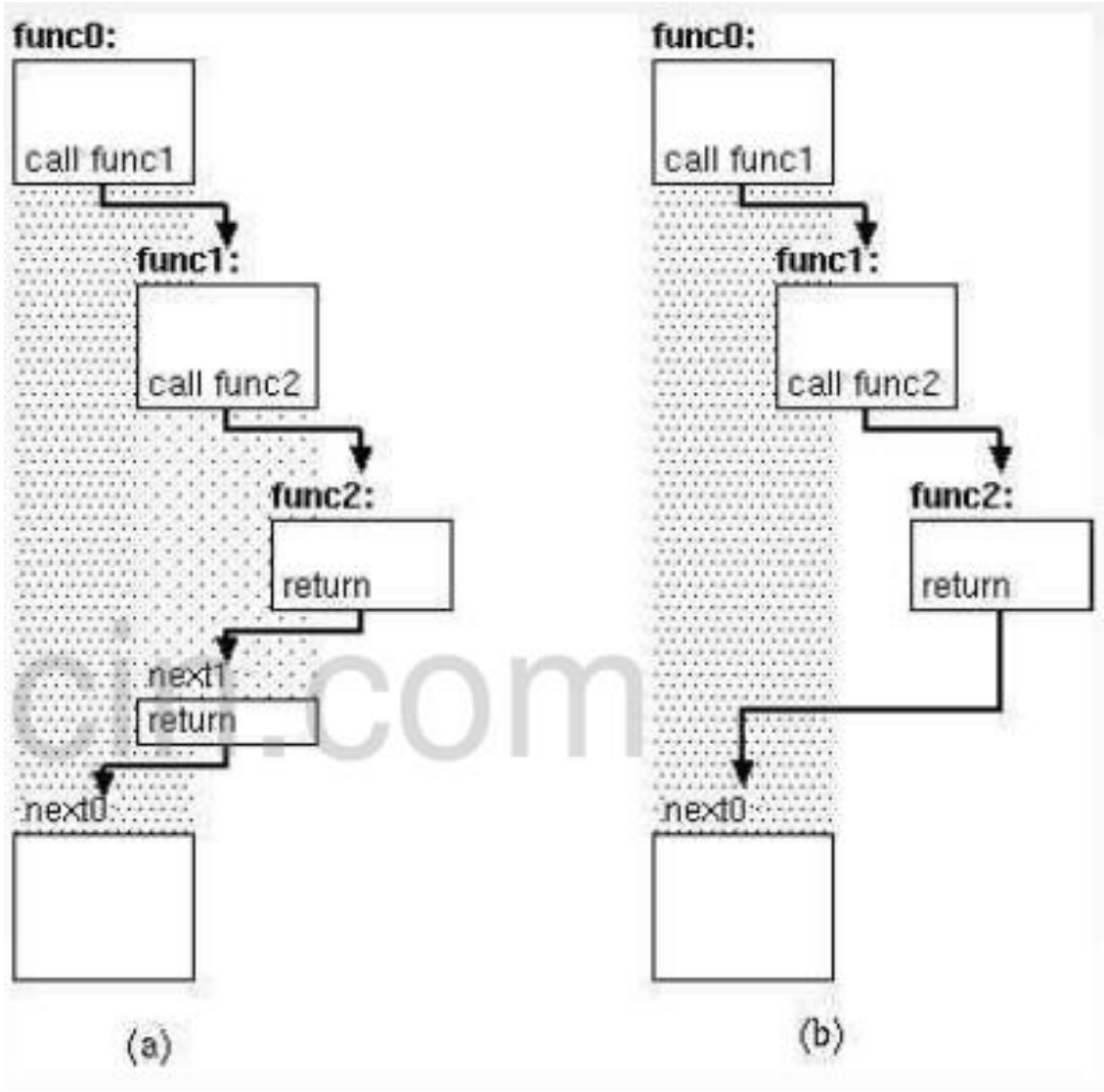
```
int sum(int i, int j) {
 if (i>j) return 0;
   else return i+sum(i+1, j);
}
```

有什么问题？

sum(3,9) = 3+sum(4,9)
= 3+4+sum(5,9)
=……

需要栈，空间浪费！
可能堆栈溢出

（b）函数式递归版本



- 尾递归
  - 一个函数(调用者, caller)调用另一个函数(被调用者, callee)，而且callee产生的返回值被caller立刻返回出去，这种形式的调用称为尾调用(tail call)。
  - 如果caller和callee是同一个函数，那么便将这个尾调用称为尾递归(tail recursion)。

- 问题：如何把函数转换成尾递归？
  - 累积参数法
    - 将递归调用之后要完成的操作预先计算好，并将结果传给递归调用。

```
int sum(int i, int j) {
  if (i>j) return 0;
   else return i+sum(i+1, j);
}
```

将sum函数转换成尾递归

```
int sum1(int i, int j, int sumSoFar) {
  if (i>j) return sumSoFar;
    else return sum1(i+1, j, sumSoFar+i);
}
int sum(int i, int j)  {
  return sum1(i, j, 0);
}
```

尾调用并不需要为callee开辟一个新的栈帧（stack frame），但需要参数和局部数据（覆盖方式），使栈空间大大缩小，提高实际运行效率

函数式编程语言的特点

- 递归调用及其优化

```
int sum(int i, int j) {
 if (i>j) return 0;
   else return i+sum(i+1, j);
}
```

```
int sum1(int i, int j, int sumSoFar) {
  if (i>j) return sumSoFar;
    else return sum1(i+1, j, sumSoFar+i);
}
```

ML语言的整数求和函数：

```
fun sum [ ] = 0
| sum (x::L) = x + sum(L);
```

```
fun sum' ([ ], a) = a
| sum' (x::L, a) = sum' (L, x+a);
```

函数式编程语言的特点

- 递归调用及其优化

斐波那契数列F_n：$F_0 = 0$，$F_1 = 1$

$$F_n = F_{n-2} + F_{n-1} \quad (n \geqslant 2)$$

```
int Fib(int n) {
  int res=b=1, a=0;
  if (n=0) return 0;
  n = n - 1;
  while (n > 0) {
    res = a + b;
    a = b;
    b = res;
    n = n – 1;
  }
  return res;
}.
```

```
fun fib 0 = 0
   | fib 1 = 1
   | fib n = fib(n-2) + fib(n-1)
```

效率太低！不断计算相同的子问题

```
fun nextfib (prev, curr: int) = (curr, prev + curr)
 fun fibpair 1 = (0, 1)
             | fibpair n = nextfib(fibpair(n-1))
```

尾递归

函数式编程语言的特点

- 递归调用及其优化

斐波那契数列F_n：$F_0 = 0$，$F_1 = 1$

$$F_n = F_{n-2} + F_{n-1} \quad (n \geqslant 2)$$

```
int Fib(int n) {
   int temp, a = b = 1;
   n = n - 1;
   while (n > 0) {
      temp = a;
      a = a + b;
      b = temp;
      n = n – 1;
      return b;
}
```

空间浪费！需要栈，可能堆栈溢出

```
fun nextfib (prev, curr: int) = (curr, prev + curr)
 fun fibpair 1 = (0, 1)
             | fibpair n = nextfib(fibpair(n-1))
```

累积参数

```
fun itfib (1, prev, curr): int = curr
             | itfib(n, prev, curr) = itfib(n-1, curr, prev + curr)
fun fib n = itfib(n, 0, 1)
```

函数式编程语言的特点

- 模式匹配
  - 模式定义为只包含变量、构造子（数值、字符等）和通配符的表达式
  - 模式与值进行匹配，如果匹配成功，将产生一个绑定（Binding）；如果匹配不成功，就会失败并抛出异常

    模式d::L和[2,4]匹配的结果为：d=2，L=[4]

    模式d::L和[ ]无法匹配，抛出异常

```
int Fib(n) {
 if (n==0) return 0;
   else if (n==1) return 1;
        else return Fib(n-2) + Fib(n-1);
}
```

```
fun fib 0 = 0
    | fib 1 = 1
    | fib n = fib(n-2) + fib(n-1)
```

- 对复杂的嵌套if语句，可以用更少的代码更好的完成任务
- 修改代码时，如增加或修改条件，只需增加或修改合适的定义即可