PPL知识点

- 引言
  - 编程语言是什么，不是什么
  - 编程语言的分类
    - 命令式 C/C++ Python Java JS
    - 函数式 LISP Scheme ML
    - 逻辑式 Prolog
- 编程语言的历史
  - 早期语言
    - Fortran：没有本地变量
    - LISP：函数式，前缀
    - BASIC：交互执行
    - Algol：结构化
    - Smalltalk：面向对象
  - 语言的评价
    - 可读性 整体简单 正交强（上下文无关，可读性强） 数据类型 语法设计
    - 可写性 整体简单 正交弱（太强难以查错） 抽象支持 表达性
    - 可靠性 类型检查 异常处理 别名 可写与可读
    - 成本 编译成本 培训成本 执行成本 维护成本
  - 语言的实现形式
    - 编译
      - 词法分析 拆分词单元
      - 语法分析 语法树
      - 语义分析 生成中间码
      - 代码生成 机器码
      - 链接装载
    - 解释 由其他程序解释执行，像机器的仿真一样
- 语法
  - 什么是语法正确?
  - BNF 用在了ALGOL 58里
    - 语素 lexeme 浙江 杭州 南京 标记 token 语素的集合 地
    - BNF是元语言 metalanguage 用来描述其他语言的语言
    - 原始BNF很难处理长度限制
  - BNF四部分
    - 终结符 token and lexeme
    - 非终结符 就是抽象 <>里面的都是非终结符
    - 起始符号
    - 产生规则

- BNF的推导
  - 左/右推导
    - LHS是单个非终结符，那么上下文无关
    - 左推导 每次都展开最左侧的非终结符
    - 推导可以是非左非右的
  - 推导式
  - 推导树
    - 从左到右读叶子就是推导结果
  - 编译器如何检查一个句子是否在语言中
    - 使用输入句子中的一个个token来进行推导或反推导
- BNF的实际问题
  - BNF不能做的
    - 无法描述local，BNF没法说变量在使用前必须被定义
    - 无法做数值比较
  - 歧义
    - 定义：某个句型（推导的中间形式）可以由多个不同的语法树生成
    - 原因：语法缺少操作符优先级的语义
    - 将高优先级的操作符放在低的下面（语法树里靠近叶子）L3.P48
  - 优先级
  - 结合律
- 扩展BNF

- 变量
  - 变量的语义
  - 变量的实现
    - 变量和内存的关系 变量是内存单元的抽象，变量或者名字指向一块储存值的内存区域
    - 常量
  - 变量的属性
    - 名字
    - 值
    - 类型
      - 决定了值的范围
      - 决定了值的操作
    - 地址
  - 变量的绑定
    - 定义：绑定是属性和实体之间的关联association，比如变量和类型或值之间的关联
    - 绑定的时机
      - 语言实现时绑定，浮点类型和其表示
      - 编译时绑定，Java和C中值和变量的绑定
      - 装载绑定，FORTRAN 77变量和内存单元的绑定，或者C中的static变量
      - 链接时绑定，Java中子程序的调用在链接时与子程序绑定

        - 运行时绑定，本地变量到栈内存
    - 静态vs动态
        - 运行时绑定之前发生的绑定都叫做静态，并在程序运行期间不会改变
        - 执行时进行首次绑定是动态绑定
- 储存绑定
    - 静态变量
        - 在执行前就与内存单元绑定，贯穿整个程序都存在，static和global
        - 优点：高效，直接寻址，没有运行时内存分配开销，全局可访问，子程序中声明的变量具有历史敏感性
        - 缺点：不够灵活（不支持递归）不能共享内存；在编译时可必须知道大小；不能是动态数据结构
    - 栈动态变量
        - 运行时刻声明的变量，C和Java中的本地变量
        - 优点：支持递归，节省空间
        - allo和deallo的开销；子程序没有历史敏感性；间接寻址，效率低
    - 显示堆动态变量
        - 隐式deallo就是GC
        - java的所有对象都是显示堆动态，采用隐式dealloc
        - 优点：动态数据结构
        - 缺点：低效、可靠性低、堆管理开销
    - 隐式堆动态变量
        - 赋值时，才绑定到堆空间上
            - APL中的所有变量，Perl和JS中的字符串和数组
        - 优点：灵活
        - 缺点：运行时维护动态属性的开销，编译会遗漏一些错误检测
- 变量的作用域和生存期
    - 作用域的静态和动态 L4.P28 P29
        - 静态作用域
            - 变量的作用域可以静态确定
            - 基于代码和空间概念
            - 可以出现变量隐藏（有更近的变量时，远变量被隐藏）
        - 动态作用域
            - 基于程序单元的调用顺序，而不是文本上的结构
            - 优点：方便，无需传递参数
            - 缺点：子程序的本地变量对其他子程序可见，可靠性差。无法对非本地变量做静态类型检查。可读性差。非本地变量寻址复杂
    - 生存期的静态和动态
    - 跨域的访问
- 变量类型
    - 有类型，需要事先声明
    - 有类型，可以自动推断（需事先声明变量，但是可以不声明类型）
    - 有类型，完全自动推断（不事先声明）

        - 有类型，可以随时切换
    - 非基础类型
        - 字符串的实现
            - C字符串
                - 非基本数据类型
                - 通过char数组实现，以\0作为结束标志
                - 有独影的库函数来支持操作，而不是基本操作
                - 问题：没有保证防止overflow的机制，strcpy(src,dst)，size src>dst时
            - 非C字符串
            - 字符串长度
                - COBOL java都是静态字符串，不可以被修改 编译时确定
                - C/C++是有限制的动态长度 可能需要运行时确定长度（C/C++例外）
                - JS perl都是动态无限的 一定需要运行时确定，需要alloc和dealloc
                - Ada支持以上三种
        - 枚举和限定范围的类型
            - 叫做叙述类型 ordinal types
            - 枚举类型
                - 一般当作小int处理
        - 数组
            - 类型
                - 静态 下标和存储都是固定的 C/C++的static 数组 效率高 无需动态分配
                - 固定栈动态 下标是静态限定，存储空间在运行时分配。节省空间（不同子程序的空间重利用） C/C++的普通数组
                - 栈动态数组 下标和存储空间都是动态限定 运行时才进行分配 用数组前可以不知道数组的大小
                - 固定堆动态=固定栈动态，就是在堆中申请而已 malloc java中所有的所有泛型数组
                - 堆动态全部都是随意的
        - 结构
        - 指针和引用
            - 指针
                - 内存值和nil组成
                - 提供了直接寻址的能力 提供了动态管理内存的方法
                - 悬挂指针 指向已经被释放的内存空间 墓碑法解决 L5.P40
                - 丢失的堆动态变量 空间还在，指针丢了
            - 引用
                - 引用指向内存中的对象或者值
            - 可选值
    - 堆的管理（自动垃圾回收）
- 表达式计算

- 运算符：一元、二元、三元
- 优先级和结合律
- 表达式的副作用 当函数改变其参数或者全局参数 叫做副作用
    - 好比a+fun(a)，如果fun会修改掉a的值，那么操作数的运算顺序就会影响表达式的结果，产生了副作用
    - 函数是没有副作用的，纯函数式语言也是这样
    - 由于冯诺依曼体系以及命令式语言和计算模型（状态机）之间的联系导致了副作用的出现
    - 如何消除副作用
        - 方案1：通过禁止函数式副作用来定义语言。禁止双向参数传递、禁止函数中访问全局变量、缺点在于：单向传参太不灵活，而且禁止访问非本地变量并不合适
        - 方案2：通过定义操作数的计算顺序来避免副作用。缺点是：限制了一定的编译器优化能力。Java就要求操作数的计算必须是从左到右的
- 运算符重载
- 类型转换
    - 收缩转换，double转float，不一定安全
    - 拓展转换，int转double，基本安全，但是精度不行
- 关系运算
- 逻辑运算
    - 短路：一个表达式不需要计算完所有的操作数和运算符
        - 逻辑运算
        - 乘法、除法运算
- 赋值是否是计算

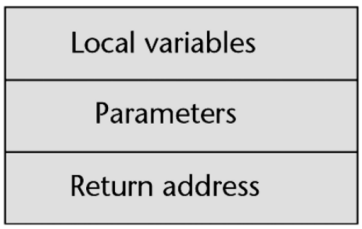
- 控制
    - 选择语句
        - 两路
        - 多路
    - 循环语句

- 子程序及实现
    - 子程序的定义
        - 两类：过程和函数
        - 过程：语句的集合描述了参数化计算
        - 函数：结构上类似过程，但是在语义上模拟了数学函数
    - 参数表及绑定（形参实参关系）
    - 参数传递和返回值
        - 传参三种语义模式 L8.P11
            - 输入型 输出型 输入输出型
        - 语义
            - 传值：实参的值用来初始化对应的形参，是输入型
                - 一般通过拷贝实现
                - 缺点是：双倍存储空间，如果参数太大，拷贝代价很高

- 传结果 输出型 就是函数结束后把形参的结果传给实参
  - 和按值传递一样，通过访问路径实现很难
  - 所以也要复制
  - 额外的问题，存在实参冲突 sub(a,a)这样的问题
- 传值-结果 输入输出型 先复制实参的值然后计算完再传回实参
  - 优点与传引用相同，缺点同上
- 传引用，同样是输入输出模型
  - 但是不通过拷贝获取数据，而是通过访问路径，有了引用，就可以访问到储存实参的单元
  - 优点：时间空间效率高
  - 缺点：对形参的访问需要额外的间接寻址，可以创建别名

- 数组和结构（对象）的传递
- 函数的传递

- 访问外界数据
- 子程序的实现
  - AR的概念
    - AR活动记录，格式或布局是子程序中的非代码部分
    - 描述的数据之灾子程序执行期间相关的
    - AR是静态的，但是不在栈里
    - 简单子程序的AR（局部变量全部静态，不支持嵌套）

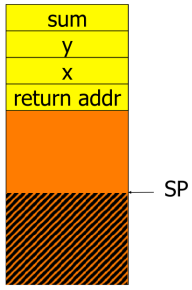


  - 通过栈动态局部变量实现子程序
    - 栈动态局部变量
    - 栈动态局部变量在运行时alloc到栈上，因此支持递归
    -

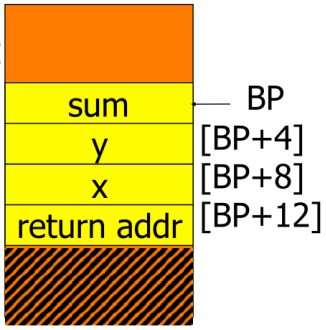
- When AddTwo is called, its AR is dynamically created and pushed onto the run-time stack
- How to reference the variables in stack, i.e., x, y, sum?

```
AddTwo PROC
   mov  eax,x
   add  eax,y
   mov  sum,eax
   ret
AddTwo ENDP
```

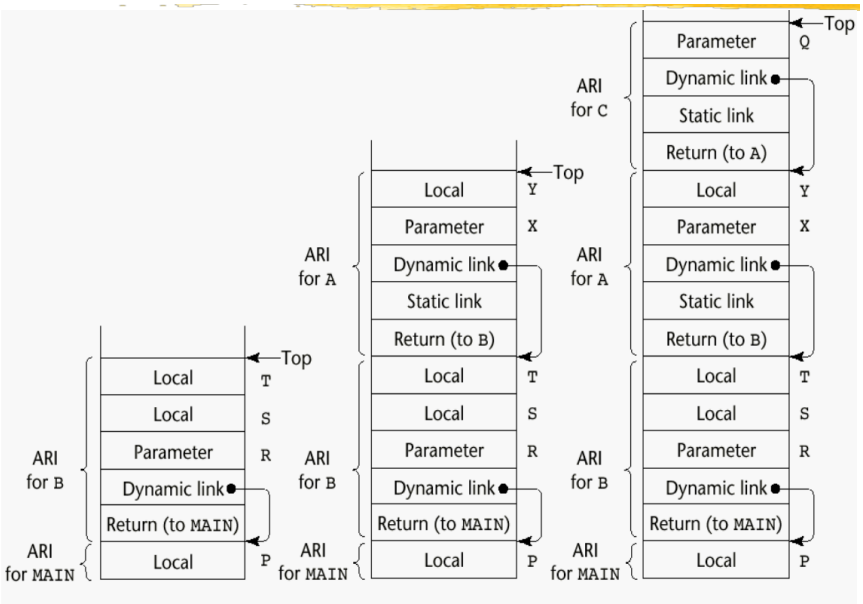
- How about SP of caller?



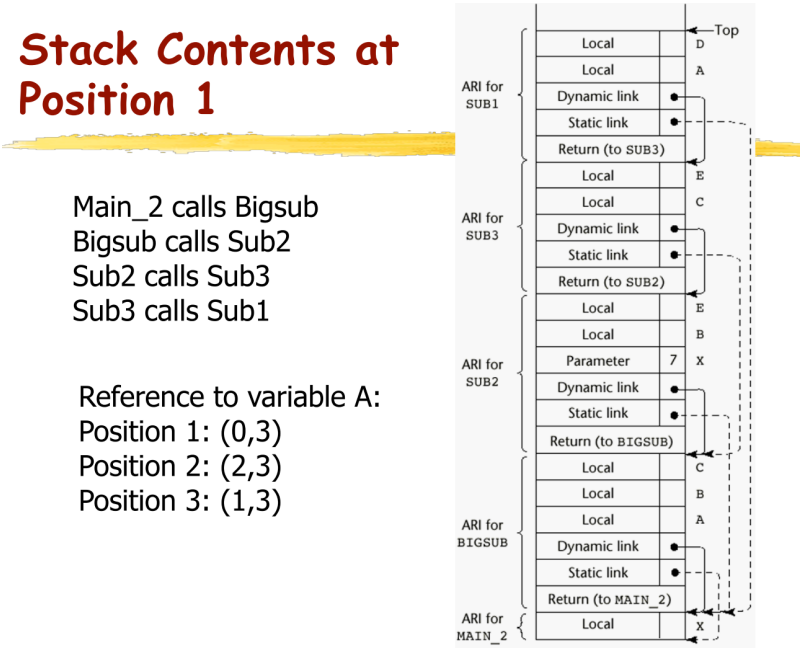
    - 在子程序调用时，通过帧指针/基址/动态链接，这三个是一个东西来引用栈里的x y sum这些变量。子程序结束后BP也恢复为原值
    -



- BP base pointer
  - 总是指向当前执行的程序单元的AR基地址
  - 子程序被调用，当前的BP会被存到新的AR里，然后BP被置为新AR的基地址
  - 子程序返回时，BP恢复到callee中存的值
  -



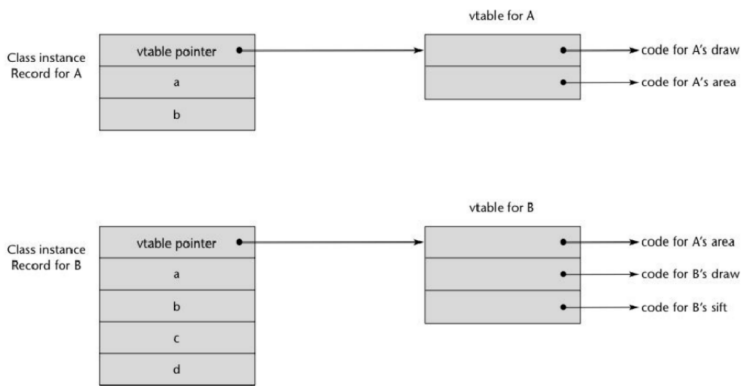
  - 动态链指出的是调用顺序。动态链指向AR的top，实际上就是老的SP栈指针
  - 静态链指出了变量作用域的寻找顺序。静态链指向AR的bottom
  - 下面的(0,3)对叫做引用链偏移量/局部偏移量对，第一个是向外找几层作用域，第二个是在该作用域的AR中，从底部到变量偏移多少。
  -



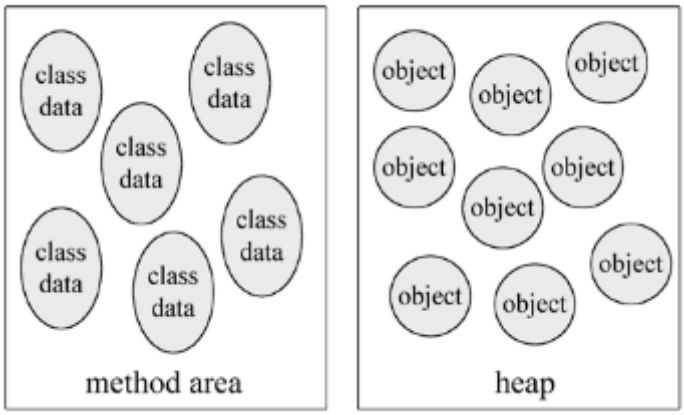
- 静态链存在的问题
  - 嵌套太深时，访问非本地变量是很慢的
  - 代码的修改会影响到嵌套深度，从而影响开销，对于时间敏感的代码很烦
- 块的处理，两种方式
  - 当作无参数的子程序，用同样的方法处理，但是这样每次执行块，就有对应的AR出现
  - 将块的AR放到包含它的子程序里，因为块的最大存储空间是静态确定的，这些空间可以在local段之后alloc好在包含它的AR里
- 参数的传递
- 返回地址
- 值的返回

- 抽象及实现
  - 类的定义
  - OOP关键特性
    - ADT抽象数据类型
    - 继承
    - 多态和动态绑定
  - OOP设计上的问题
    - 对象的排他性
      - 选择1：所有都是对象
        - 优点：优雅且一致
        - 缺点：即使是很简单的操作，都需要通过对象间的消息传递来完成
      - 选择2：向现有的类型系统中添加对象
        - 优点：解决了上面的缺点，在原始的简单类型上的速度很快
        - 缺点：出现类型的混杂
      - 选择3：原始的类型不变，其他结构化的类型作为对象
        - 优点：原始类型上的操作很快，类型系统相对较小

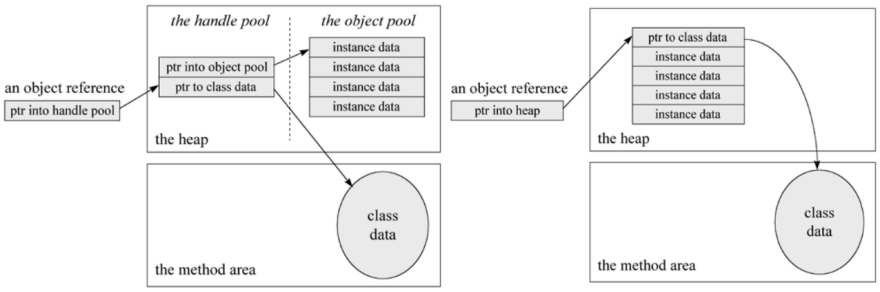
        - 缺点：类型混杂
    - 子类是子类型吗 subclass subtype
        - 子类是子类型，当子类继承掉的方法是类型完全兼容的，也就是在任何调用override函数的地方没有发生类型错误
    - 多继承会导致菱形继承
    - 对象的分配和释放
        - alloc的时候，放到哪里
        - 要不要显式dealloc
    - 静态和动态绑定
        - 需不需要所有的消息与方法之间的绑定是动态的
            - 如果需要就会低效，不需要就没有动态绑定的好处（动态绑定使程序更容易拓展）
            - 可以的方案是：程序员可以自己明确绑定方法
- OOP例子
    - C++
        - C++支持私有继承和公有继承
            - 私有继承的子类会把父类的pub和pro改成pri
                - 私有继承的子类不是子类型，因为改变了访问权限
            - 公有就是原来的
- OO构造的实现
    - 大部分OO构造可以编译器实现
        - ADT被实现为作用域规则
    - 两个大部分
        - 储存实例数据
            - CIR class instance record 类实例记录储存着对性爱难过的状态，他是静态的，编译时确定，每个类都有自己的CIR
        - 方法调用到方法的动态绑定
            - 静态绑定不需要CIR参与
            - 动态绑定的方法必须在CIR中设立入口
                - 入口可以是指向方法代码的指针
                - 使用vtable来储存所有的动态绑定方法
                - 然后通过指向vtable的指针和偏移量访问方法
                -



- 对象的生成
- 成员变量的绑定
- 成员函数的绑定
- 继承的实现
- 范型的实现

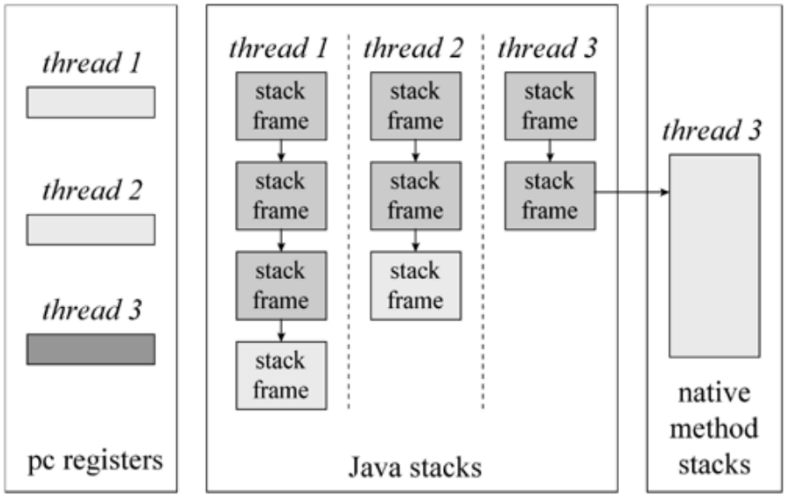
- JVM
  - JVM和OS的关系
    - JVM是在OS里的
  - 类装载器
  - JVM内存模型
    - JVM中的每个实体都有一个方法区和堆，它们通过VM中运行的线程共享。当JVM装载一个calss文件，它会从class文件中的二进制数据解析有关类型的信息。然后将类型信息放置到方法区。程序运行起来后，JVM将程序实例化的所有对象放到堆上
    - 

    - 方法区
      - 所有的线程共享同一个方法区，所以对方法区进行访问的数据结构必须是线程安全的
      - 方法区的大小可以不固定
      - 方法区内存可以不连续
      - 方法区可以进行GC
    - 堆
      - 所有线程共享一个堆
    - 类数据vs类对象
      - 类数据和静类态变量在方法区
      - 类对象在堆
      - 从对象到类数据有一个指针
      -



- 堆栈
  - 线程产生后，拿到自己的PC和Java栈。如果线程执行Java方法，那么PC就是下一条指令。
    - 线程Java栈储存着该线程的Java方法调用状态，包含局部变量、调用参数、返回值、中间计算
    - 如果不是Java方法，而是本机方法，就存在本机栈方法栈中，也可能在寄存器或者其他存储区
  - Java栈由堆栈帧连接成链表组成，
    -

Java Stacks



  - 计算栈和线程栈
  - 栈帧
    - 三部分：局部变量 操作栈 帧数据
    -
  - 方法栈
    - 线程调用Java方法，JVM创建一个新栈帧，将其放到Java栈中作为新的栈帧（链表链过去），方法执行时，使用该栈帧储存参数、局部变量中间计算和数据
- 常量池
  - 常量池里有什么
    - 常量池是会被类型使用到的有序常数集，包括string int float以及对类型、字段和方法的符号引用

        - 常量池使用索引，常量池包含类型使用的所有类型、字段和方法的符号引用，常量池在动态链接里十分重要
    - 装载和链接的过程
        - 装载，找并且import一个类型的二进制数据
        - 链接，执行验证、准备和解决
            - 验证：保证引入类型的正确性
            - 准备：为类变量alloc内存并且初始化
            - 解决：将符号引用转化为直接引用
        - 初始化：调用Java代码初始化类变量
- 函数式计算
    - 基本概念
    - 流式计算
        - 筛选
        - 映射
        - 化约
    - 高阶函数/闭包
        - 实现
        - 访问环境中的变量
- 并行
    - 并发、并行和分布式的概念
        - 并行的原因：速度很难提升，但是核心数可以增加
        - 并行：一个任务被分为多个任务执行
        - 并发：一个程序的多个任务同时执行
        - 分布式：并行计算在多个机器上进行
    - 并行计算的概念
        - 计算图
        - 加速比
            - work就是所有数字加起来
            - span是最长路数字加起来
            - 加速比=work/span
        - 多核优化
        - Amdahl定律
            - 实际的加速比<=work/span
    - 并行计算的实现
        - 结构并行
            - Folk/Join框架
        - 函数并行
            - Furture框架
            - 记忆优化
        - 循环并行
            - Forall框架

        - 栅栏问题
- 递归
    - 线性递归和迭代
        - 牛顿逼近算平方根
        - 阶乘
    - 树递归和迭代
        - 下面的是尾递归，可以改成迭代 上面是递归

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

        - 斐波那契数列
        - 帕斯卡三角
        - 指数计算的优化
    - 高阶函数和Lambda实现
        - 累加
        - 求方程的根
    - 消除尾递归
        - 使用当前子程序的AR，修改本地变量字段的值，然后再使用该AR运行子程序即可