中国科学技术大学 University of Science and Technology of China

编译技术回顾

《程序设计语言理论》

张昱

0551-63603804，yuzhang@ustc.edu.cn
中国科学技术大学
计算机科学与技术学院

主要内容

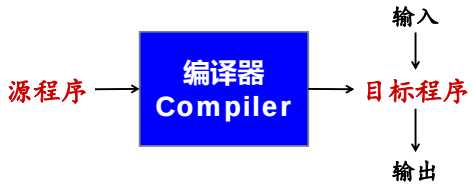
1. 编译器的形式和阶段
2. 运行时数据的组织
3. 抽象机模型

中国科学技术大学 University of Science and Technology of China

1. 编译器的形式和阶段

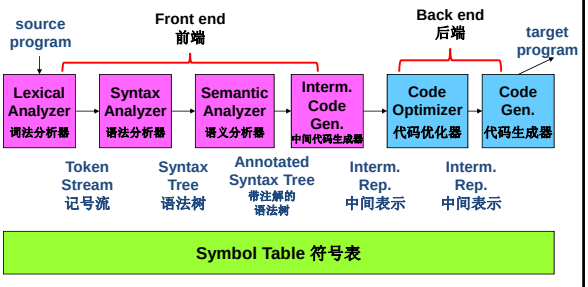
- 编译器的形式
- 编译器的主要阶段

编译器是什么

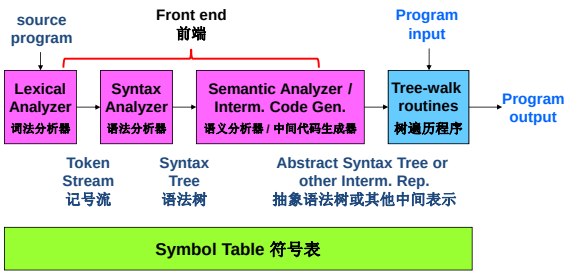


- 目标语言
  - 一种编程语言、CISCs（复杂指令集）、RISCs（精简指令集）
  - 面向多核/众核、GPUs、FPGAs、量子计算机等

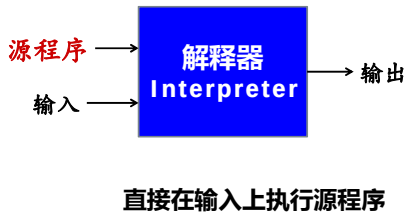
编译器的阶段



编译器的阶段

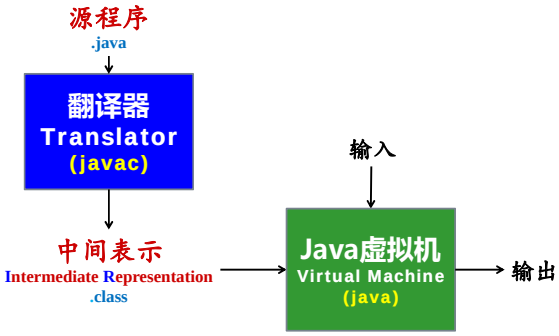


解释器



直接在输入上执行源程序

Java编译器

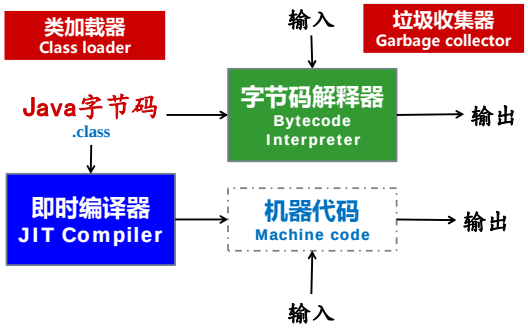


编译器的其他形式

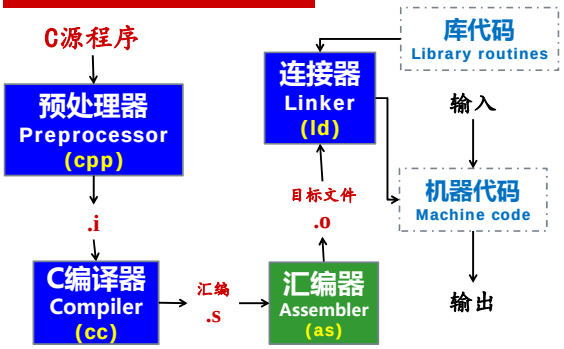
- 交叉编译器（Cross compiler）
  - 在一个平台上生成另一个平台上的代码

    PC → arm-linux-gcc → ARM
- 增量编译器（Incremental compiler）
  - 以增量地编译源程序,只编译修改的部分, 如 Freeline
- 即时编译器（Just-in-time compiler）
  - 在运行时对IR中每个被调用的方法进行编译，得到目标机器的本地代码，如 Java VM 中的即时编译器
- 预先编译器（Ahead-of-time compiler ）
  - 在程序执行之前将IR翻译成本地码，如 ART中的AOT

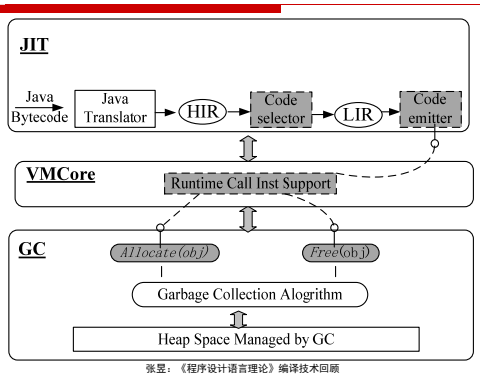
Java虚拟机



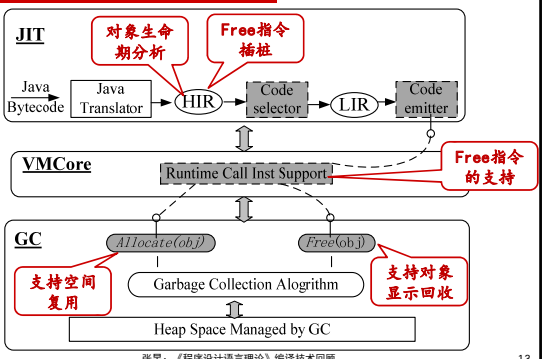
C 编译器



Java虚拟机



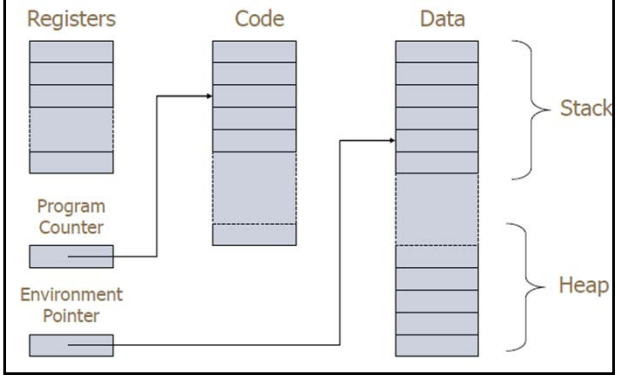
Java虚拟机：改善内存管理



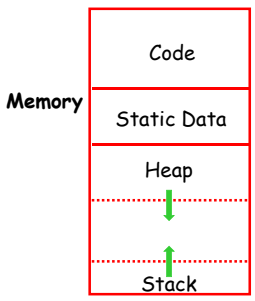
2. 运行时数据的组织

- 运行时的数据组织、地址空间
- 活动记录：语言特征对其的影响

简化的机器模型



进程地址空间：64位Linux



低地址 0x0000000000400000

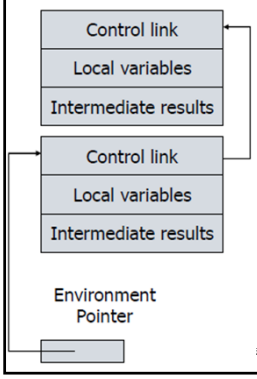
64位Linux系统：

- 使用低48位虚拟地址，48位至63位必须与47位一致（256TB）
- 用户空间：低128TB 0-0x7FFFFFFFFFFF
- 内核空间：64TB 0xFFFF880000000000-0xFFFFc7FFFFFFFFFF

高地址 0x00007FFFFFFF0000

TASK_SIZE

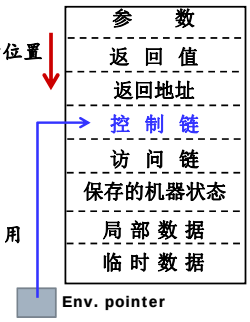
内嵌块的活动记录(activation record)



- 控制链
  - 指向外围块在栈中的活动记录
- 活动记录入栈
  - 设置新的控制链，使之指向旧的env ptr
  - 将env ptr指向新的活动记录
- 活动记录出栈
  - 用当前活动记录的控制链来重置env ptr

函数的活动记录

- 返回地址
  - 函数返回时要执行的代码的位置
- 返回值
  - 存储返回结果（或其地址）
- 参数
  - 从调用者处获取的数据
  - 参数传递方式：传值、传引用

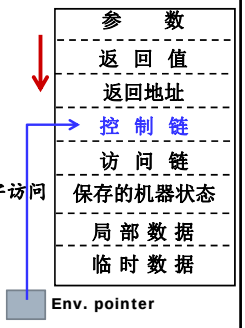


函数的活动记录

- 控制链
  - 指向调用者的活动记录
- 访问链
  - 指向在程序正文中包含其的最近外围块的活动记录

支持过程定义嵌套时的非局部名字访问

- 二者的区别
  - 控制链依赖于程序的动态行为
  - 访问链依赖于程序正文的静态形式



高阶函数

- 语言特征：函数是first class values
  - 函数可以作为参数被传递
  - 函数可以作为返回值返回

需要维护函数所处的环境（上下文）

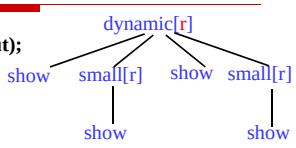
- 静态作用域？（上下文按程序正文中的静态包含关系）
- 动态作用域？（上下文按程序运行时的调用关系）
- 实现
  - 函数作为参数：还要传递其环境所在的活动记录指针
  - 函数作为返回值：需要保存返回函数的活动记录

闭包 Closures

- 函数值是一个元组
  - closure = <env, code>
- 当调用用闭包表示的函数时
  - 为调用分配活动记录
  - 按闭包中的环境指针env来设置活动记录中的访问链

示例：基于静态作用域时

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



show在dynamic中定义，show中访问的r是dynamic中定义的

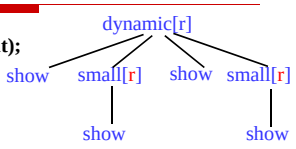
执行后输出：

0.250 0.250

0.250 0.250

示例：基于动态作用域时

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



dynamic中调用的show所访问的r是dynamic中定义的；small中调用的show所访问的r是small中定义的

执行后输出：

0.250 0.125

0.250 0.125

动态作用域

- 使用动态作用域的语言
  - Pascal、Emacs Lisp、Common Lisp(兼有静态作用域)、Perl（兼有静态作用域）、Shell语言（bash, dash, PowerShell）
- 其他
  - 宏展开

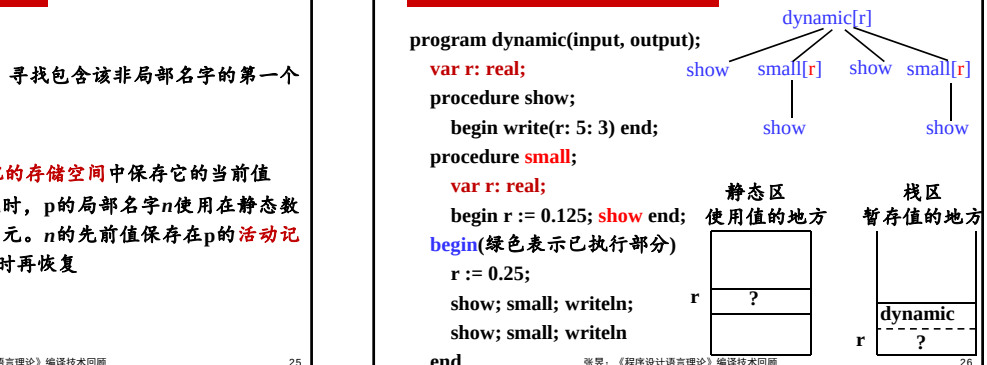
https://en.wikipedia.org/wiki/Scope_(computer_science)

实现动态作用域的方法

- 深访问
  - 用控制链搜索运行栈，寻找包含该非局部名字的第一个活动记录
- 浅访问
  - 为每个名字在静态分配的存储空间中保存它的当前值
  - 当过程p的新活动出现时，p的局部名字*n*使用在静态数据区分配给*n*的存储单元。*n*的先前值保存在p的活动记录中，当p的活动结束时再恢复

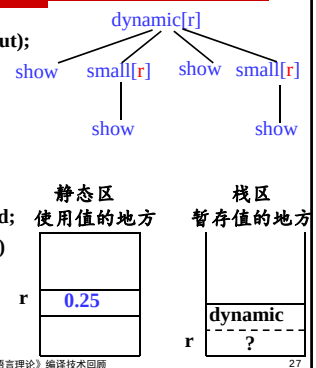
基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



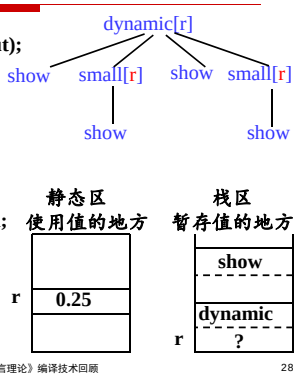
基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



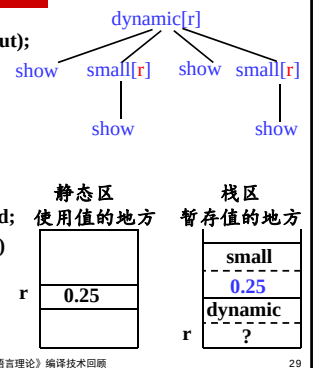
基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



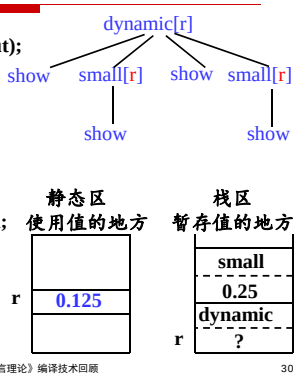
基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



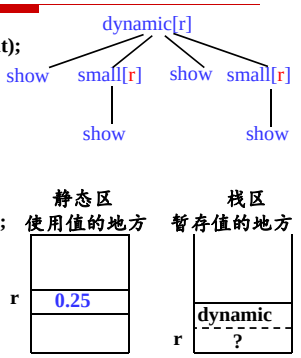
基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
  begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
  end.
```



基于浅访问实现动态作用域

```
program dynamic(input, output);
  var r: real;
  procedure show;
    begin write(r: 5: 3) end;
  procedure small;
    var r: real;
    begin r := 0.125; show end;
begin(绿色表示已执行部分)
  r := 0.25;
  show; small; writeln;
  show; small; writeln
end.
```



3. 抽象机模型

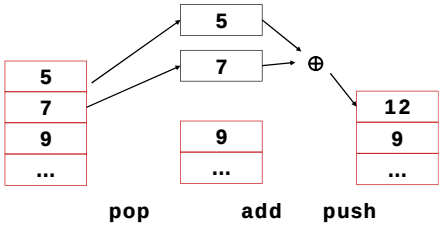
- 栈型机器(Stack Machines)
- 寄存器机器(Register Machines)

栈型机器

- 栈型机器模型
  - 是一种简单的求值模型
  - 没有变量和寄存器
  - 中间结果存放在栈中
  - 每条指令：
    1）从栈顶取操作数
    2）将这些操作数从栈中删除
    3）在这些操作数上计算所需要的操作
    4）将结果入栈

栈型机器操作示例

例 在栈型机器上执行加法操作



栈型机器操作示例

- 指令
  - push i　　将 i 置于栈顶
  - add　　从栈中弹出两个元素，将它们相加，再将结果入栈
- 计算 7+5 的程序

```
push 7
push 5
add
```

```
iconst 1
iconst 2
iadd
```

Java字节码、.Net的 CLR、Pascal的P-code 、Perl 5属于此

栈型机器的优点

- 统一的编译机制 => 编译器实现简单
  每个操作从同一地方取操作数，并把结果放到同一地方
- 操作数的位置是隐式的
  - 总是在栈顶
  - 无需显式指定操作数和结果的位置
- 代码量小、程序更紧凑
  - 如，用 add，而不是 add r1, r2

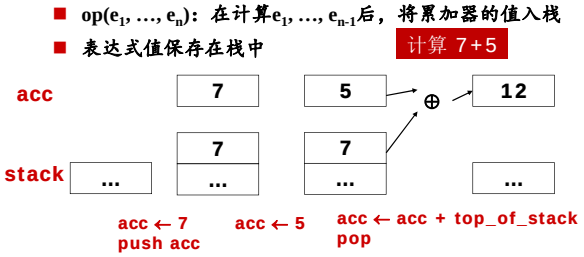
—— Java字节码使用栈求值模型的原因

栈型机器的优化

- 存在的问题
  - add指令涉及3条访存操作：2 reads、1 write
  - 栈顶被频繁访问
- 优化思路
  - 将栈顶内容保存在寄存器中（寄存器访问速度快）
    ——累加器(accumulator)
  - add指令改为acc ← acc + top_of_stack
    这样就只有一条访存指令了

带累加器的栈型机器

- 不变式(invariants)
  - 表达式的结果总是在累加器中
  - op(e_1, …, e_n)：在计算e_1, …, e_{n-1}后，将累加器的值入栈
  - 表达式值保存在栈中



寄存器型机器

- 寄存器型机器模型
  - 操作数保存在寄存器上
  - 指令中需要参数指定操作数的地址，如 add t1, t2, t3
    如，Dalvik VM、Lua VM、Parrot VM等均属于此类
- 相比栈型机器的优势和劣势
  - 劣势：编译器复杂（寄存器分配和指派等、跟踪变量存放的场所）、代码量大、单条指令的代价高
  - 优势：指令数少（执行得快）、还可以做一些栈型机器无法实现的优化（如公共表达式的结果保存到寄存器中，不必重复计算）

栈 vs. 寄存器

- 栈代码容易翻译到寄存器代码，反之则不成立
  - 局部变量直接映射
  - 栈存储单元映射到虚拟寄存器

栈代码	寄存器代码	说明
iload 4	imove r10, r4	；加载局部变量4
bipush 57	biload r11, 57	；加载立即数57
iadd	iadd r10, r10, r11	；整数加
istore 6	imove r6, r10	；存储结果到局部变量6
iload 6	imove r10, r6	；加载局部变量6
ifreq 7	ifreq r10, 7	；如果结果为0，跳到7