**CH2 词法分析**

**2.1 扫描处理**

1.某些记号只有一个词义:保留字；某些记号有无限多个语义:标识符ID表示.

**2.2正则表达式**

1.幂>连接>并

2.相同的语言可以用不同的RE表示

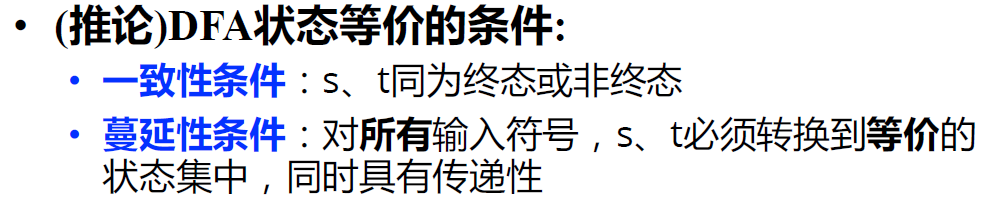
2.RE匹配最长字串优先；规则优先（按顺序）

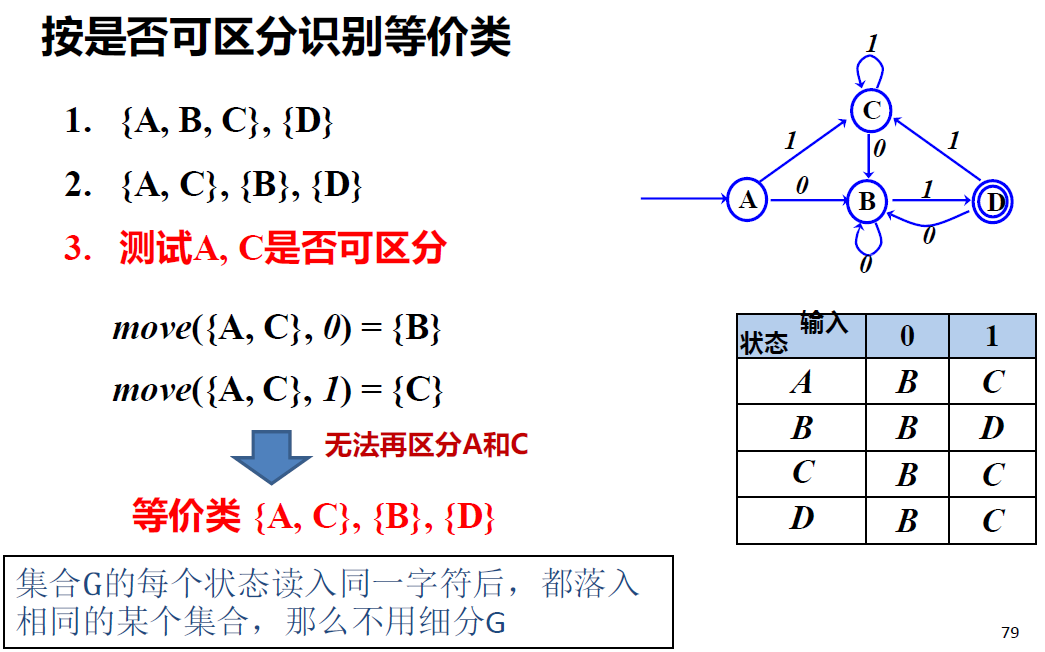
**2.4正则表达式到DFA**

**1.子集构造的过程:**首先列出所有状态的闭包；然后将初始状态的闭包作为新的初始状态；然后计算在每个新状态下在各个字符上的转移的闭包作为新的状态,转移自然成为新的转移；包含原接受状态的所有新状态都是接受状. PS:闭包首先包含自身.

**2.DFA状态数最小化** (Hopcroft’s algorithm)

可区分的状态：如果存在串x，使得从s、t出发，一个到达接受状态，一个到达非接受状态，那么x就区分了s和t。





**CH3 上下文无关文法分析**

**3.1 CFG**

1.左递归:定义A的推导式的右边第一个出现的是A；右递归:定义A的推导式右边最后一个出现的是A；

2.仅含终结符号的句型是一个句子；

3.最左推导: 每步代换最左边的非终结符

**3.2分析数和抽象语法树**

1.分析树是一个作了标记labeled的树,内部节点是非终结符,树叶是终结符；对一个内部节点运用推导时,推导结果从左到右依次成为该内部节点的子节点

2.AST(syntax tree)去除了终结符和非终结符信息,仅保留了语义信息

**3.3 Ambiguity二义性**

**1.定义:** 某些句子存在多棵分析树。

**2.解决方法：**分层，规定符号的优先级和结合性。

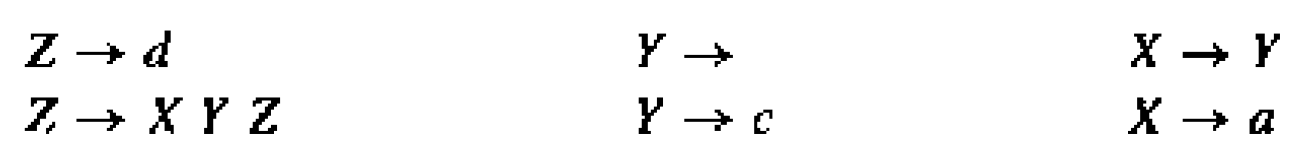
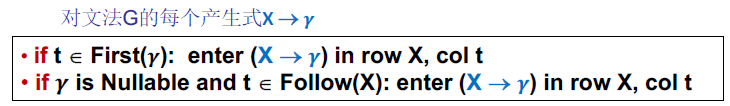
3.越接近根（S）,优先级越低；左递归左结合

**自顶向下分析: LL(k)**

第一个L是从左到右处理,第二个L是最左推导,k代表使用k个符号预测分析方向

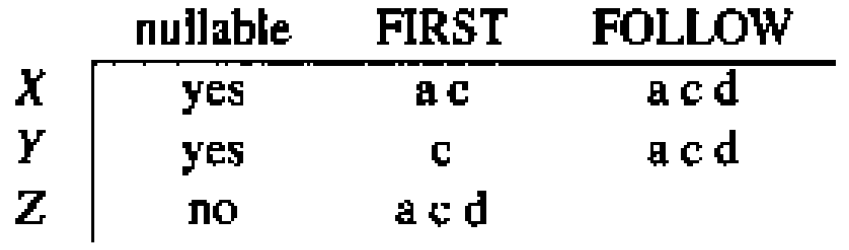
1. LL(1)文法是无二义性的, 对任意规则, First()∩First(为空,若,则不能推导空串且First()∩Follow(为空。

2.LL(1)分析表M[N,T]的构造算法:

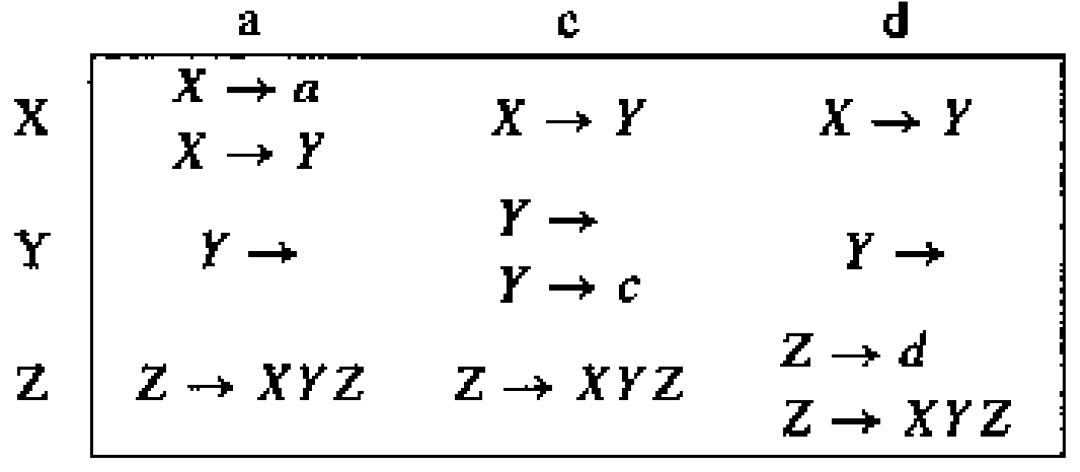


1) 检查有无左递归左因子

2)nullable|First|Follow



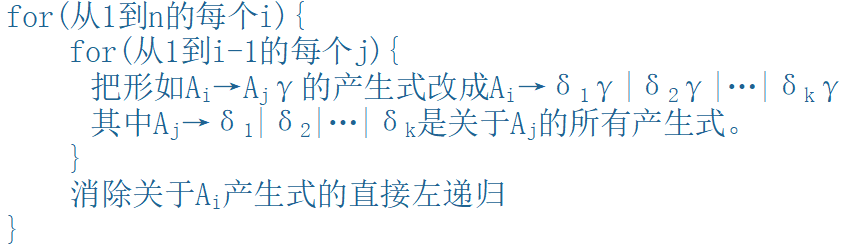
3) 按照算法填写规则进表



3.消除左递归:

一般的左递归，不能带有产生式和循环

按照某个顺序，将产生式排序为A1,A2,…,An



4. **提取左因子**

。，

5. LL(1)分析的优点

• 运行高效(线性时间)

• 递归实现符合文法结构、适合手动构造&自动生成

LL(1)分析的局限性

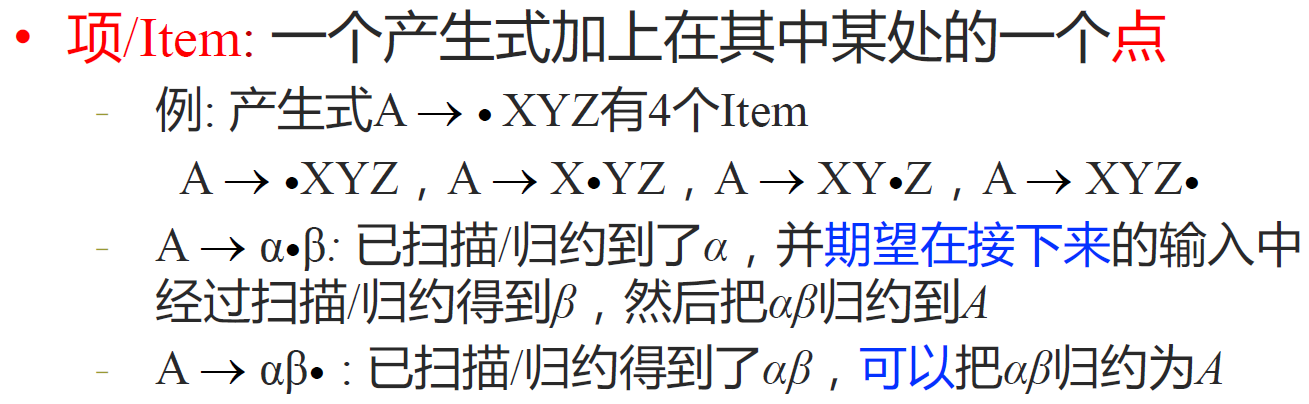
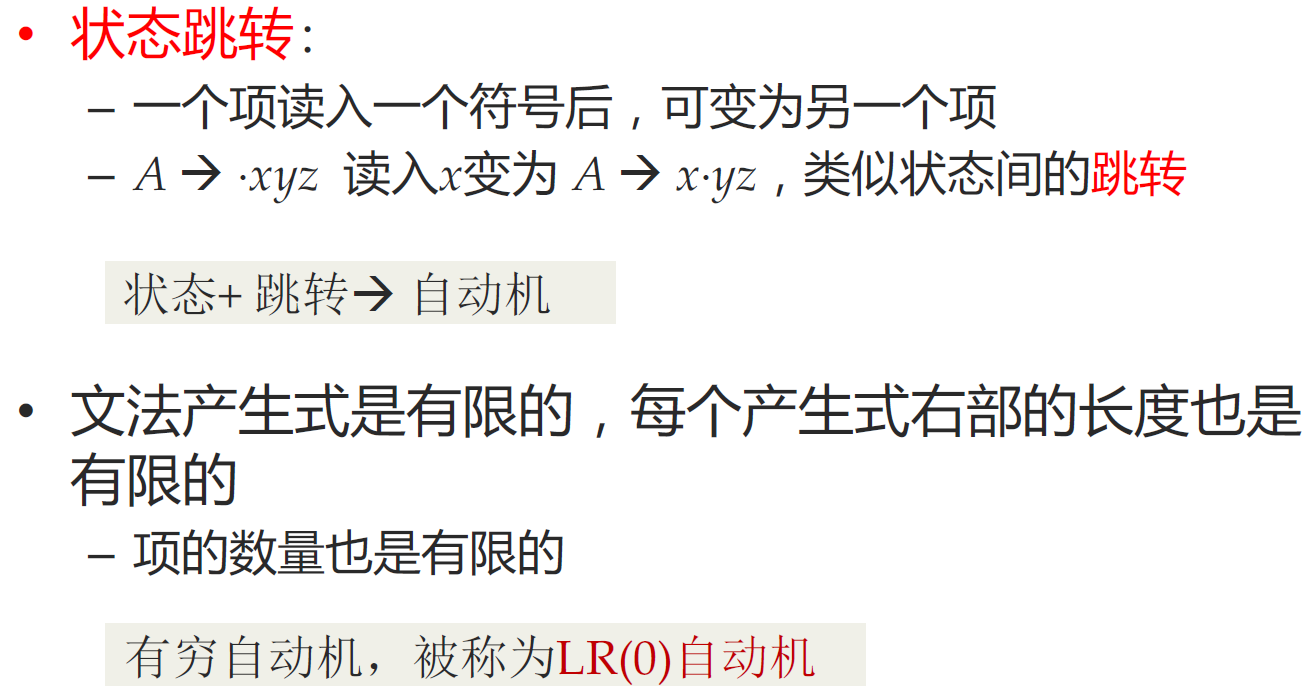
• 能分析的文法类型受限

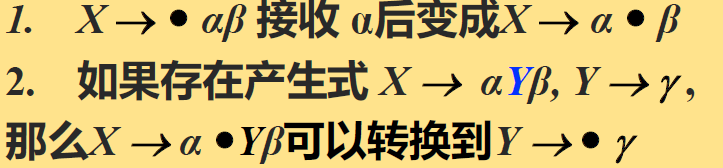
**自底向上分析LR(k)**

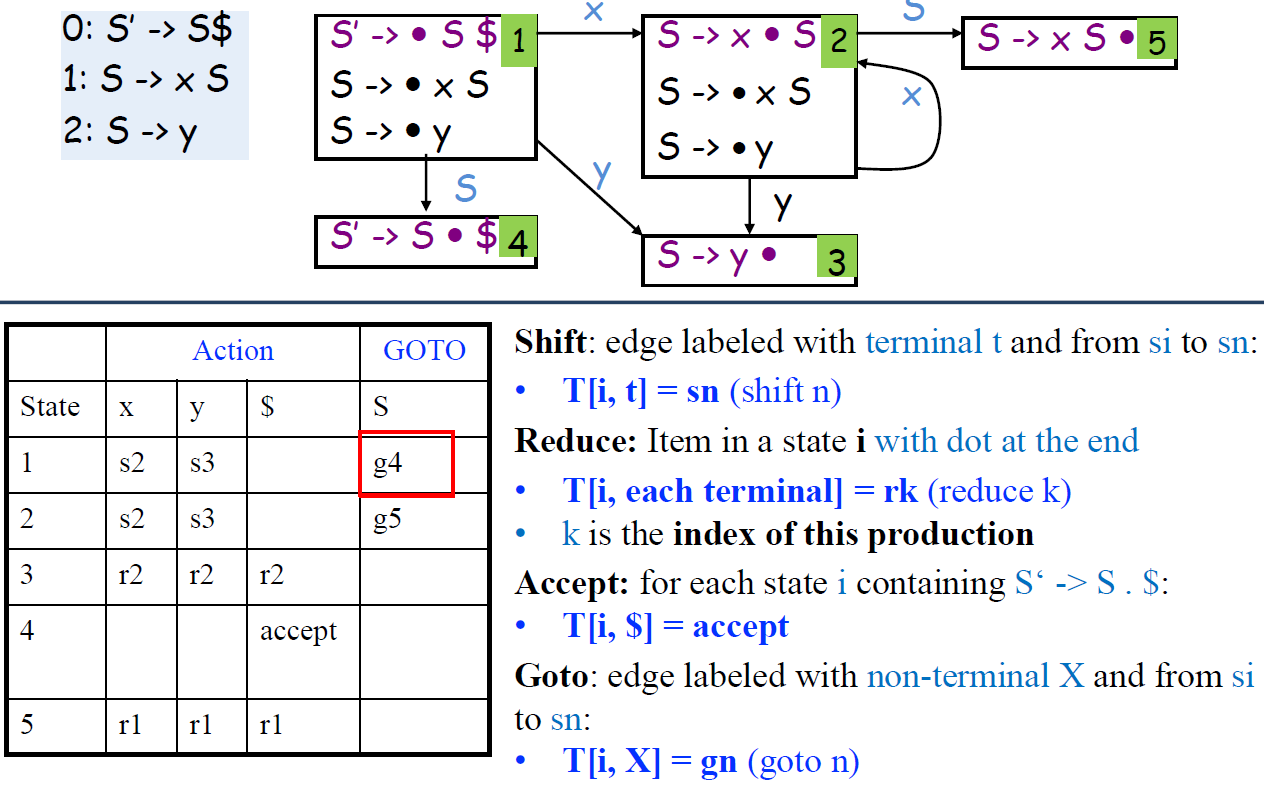
L从左向右扫描，R最右推导的逆过程,k向前看k个字符

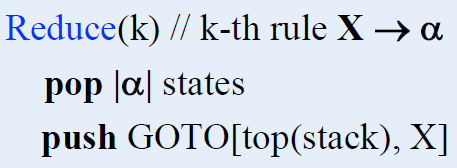
**LR(0), SLR(1), LR(1), LALR(1)**

**1. LR(0)分析**







**2. SLR(1)分析**

每步归约都应该满足t ∈ Follow(E)

– E是用来归约的产生式的左部

– t是Next Token

优化分析表，只在follow集中填入reduce

**缺陷**：如不是任何最右句型的前缀，那么即使x在某个句型中跟在A之后，仍不应该按归约

**3. LR(1)分析**

LR(1)项中包含更多信息来消除一些归约动作，精确指明何时应该规约

项的形式例如：

其中a称为向前看符号，为终结符或$

**LR(1) Item 在如下两种情况下存在转化：**

1. 接收 后就会变成 ；

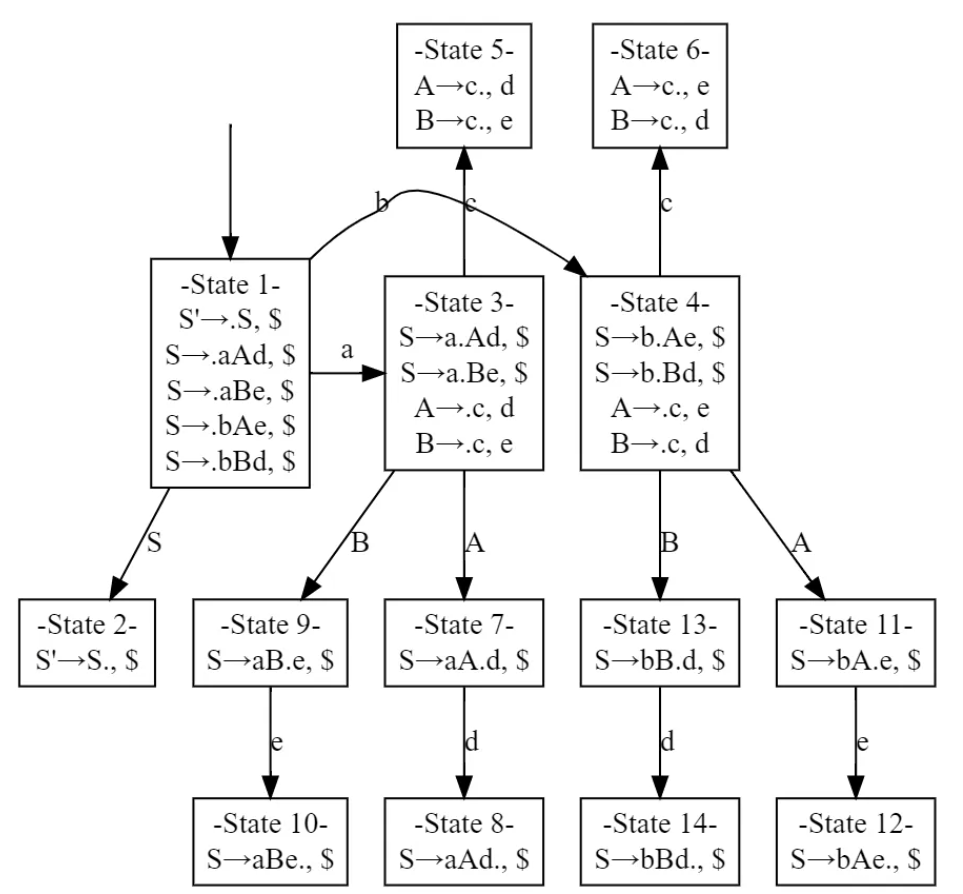
2.如果存在，那么对于每一个 (可以为)， 可以经 转换到 。

例如：

S'-> S$

S -> aAd|bBd|aBe|bAe

A -> c; B -> c



**4. LALR(1)分析**

将具有相同核心（忽略后缀）的状态进行合并,如上图中的状态5和6

5.s-r conflict:表中同时出现了sn和rn,即构成s-r冲突

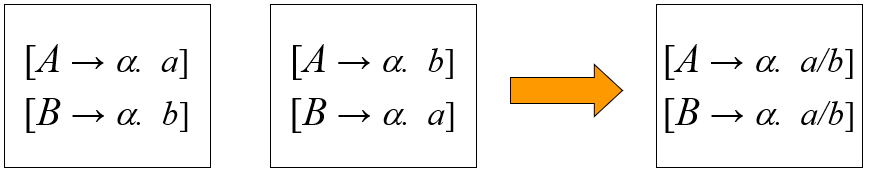
6.r-r conflict:表中同时出现两个及以上的rn,则构成r-r

**7. Yacc冲突解决**

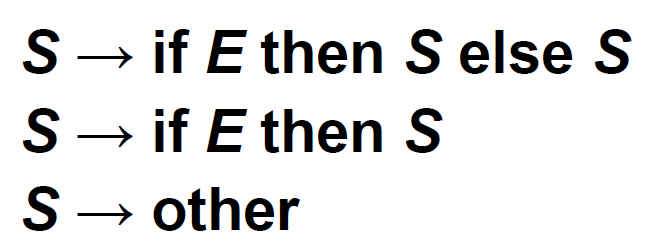
**-** 归约/归约冲突：选择Yacc说明中先出现的产生式

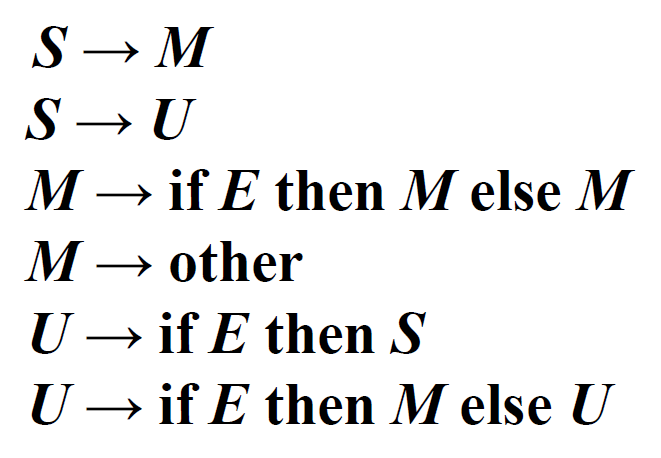
- 移进/归约冲突：移近优先

8.如果文法是LR(1),那么LALR(1)中必然没有sr冲突,但是可能有rr冲突.

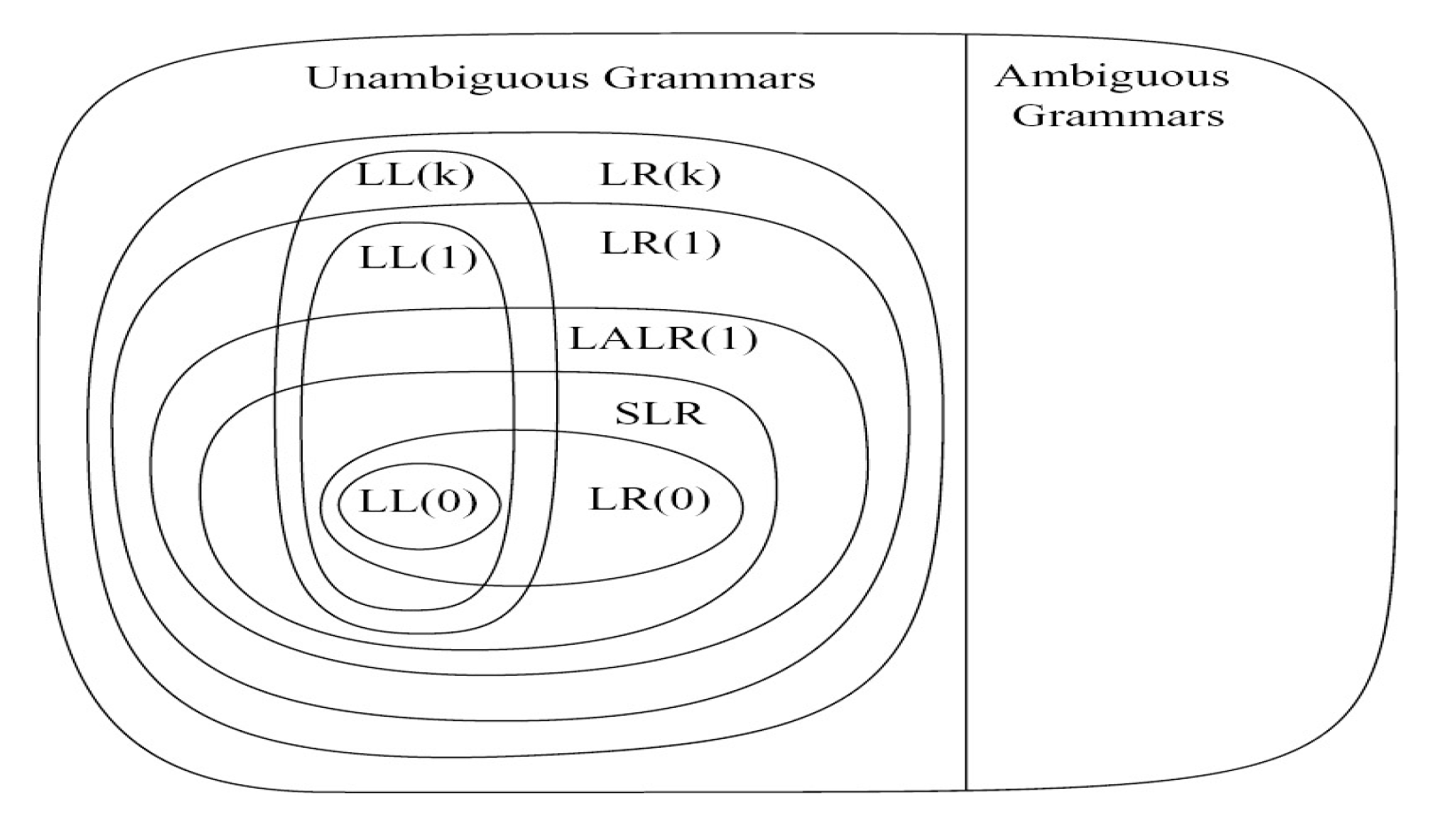


9.悬挂else(dangling else)的处理:悬挂else会导致s-r冲突,冲突文法如下:



解决办法:引入新非终结符用于if-else的匹配问题.M是匹配的else;U是未匹配的else

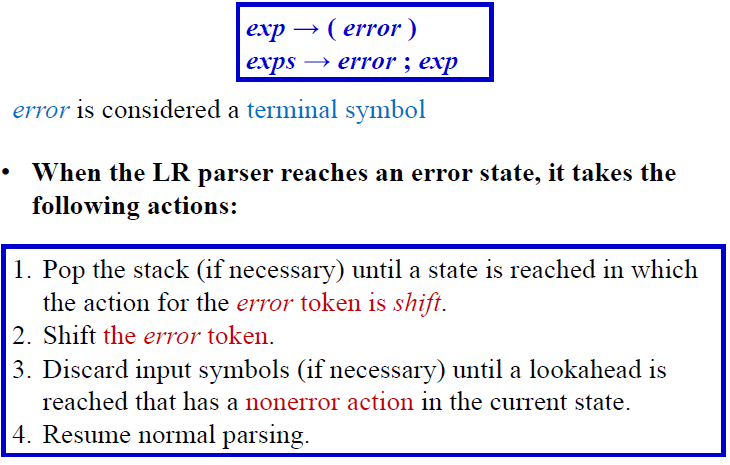
10. 各类文法的层次如下:



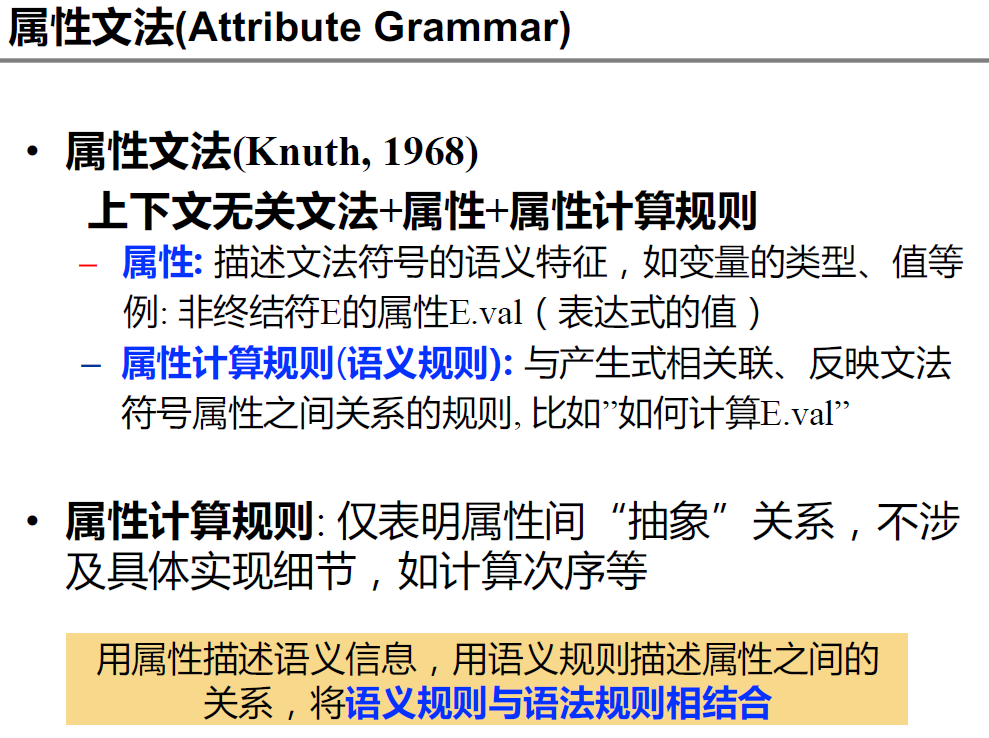
**11. 错误恢复**

动机：开发者希望一次编译报出所有错误

**Local recovery**



**CH4 抽象语法**



应用：

**·推导类：**例如很多语言的编译期求值

·**生成类：**生成AST， 中间代码等

**2．语义动作（semantic action）**

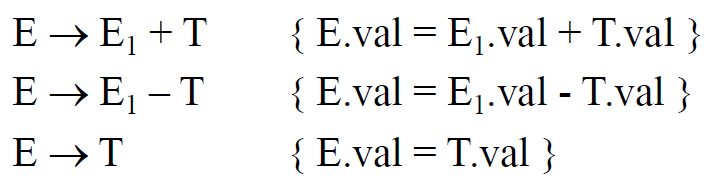
我们可以给产生式绑定一个语义动作，使得按照这个产生式规约时/推导时完成特定操作。

每个token都可能有独属于自己的**语义值**(Semantic Value) 。每种token的语义值类型可以不同，我们把A的语义值的类型称为“A的关联类型”。

例如对于产生式 :A→BCD

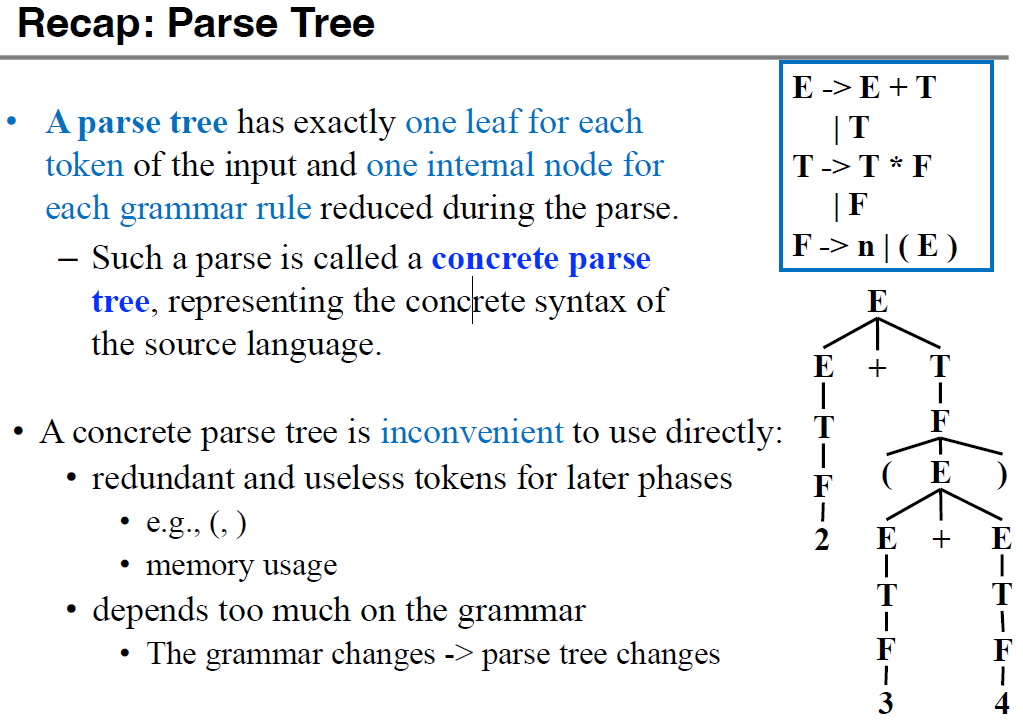
·语义动作返回值必须是A的关联类型

·这个值可以通过B C D各自的语义值进行运算得出



**3. 抽象解析树APT**

APT = Abstract Parse Tree 是语义动作的一种应用。



**4.位置Position**

在one-pass编译器中，词法分析、语法分析、语义分析是同步进行的。而错误发生时，词法分析器lexer的位置可以用来作为错误发生位置的合理估计反馈给用户。所以，lexer存有一个全局变量维护当前位置信息。

然而，对于使用AST的编译器，词法分析结束后才开始语法分析，因此这是不可行的。

**解决方案**：AST每个节点记录自己在源文件中的位置，标记自己是具体哪几个字符派生而来的。

**CH5 语义分析**

**5.1广义语义分析**

因此我们需要通过检查、遍历 程序表示(Program Representation) 来完成 （广义的）语义分析。

常用的程序表示：AST、CFG、programdependence graph (PDG)、valueflowgraph (VFG)、SSA

然后我们就可以：类型检查、代码生成、去除dead code、寄存器分配

**5.2狭义语义分析**

本课重点关注的（狭义的）语义分析指的是通过检查AST获知程序的静态属性，包括：

·作用域(Scope)与变量可见性

·变量、函数、表达式的类型

·将AST转为中间代码IR

**5.3符号表**



**Symbol Table需实现的接口**:

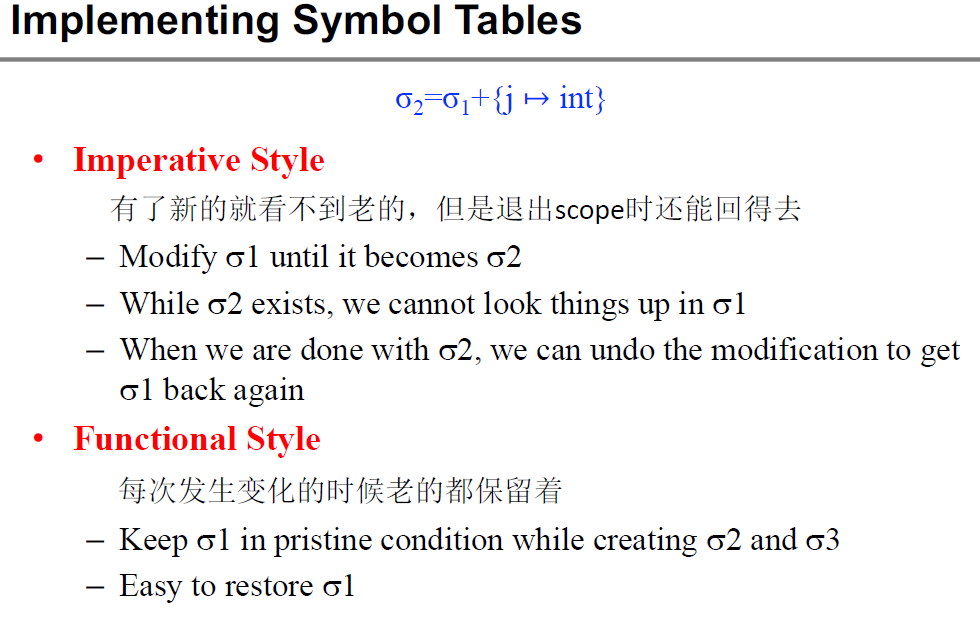
• insert:将名称绑定到相关信息(如类型), 如果名称已在符号表中定义，则新的绑定优先于旧的绑定

• lookup:在符号表中查找名称，找到名称绑定的信息

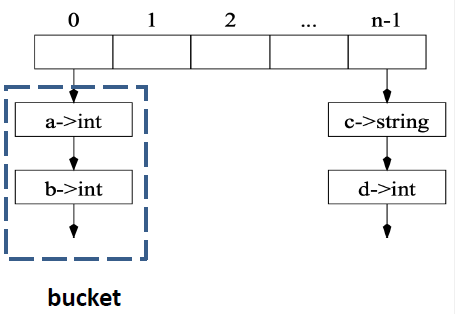
• beginScope:进入一个新的作用域

• endScope:退出作用域，将符号表恢复

在Java等语言中，可能有多个环境同时活跃（对应不同的module, class等），他们都需要一个符号表。这被称为多符号表。



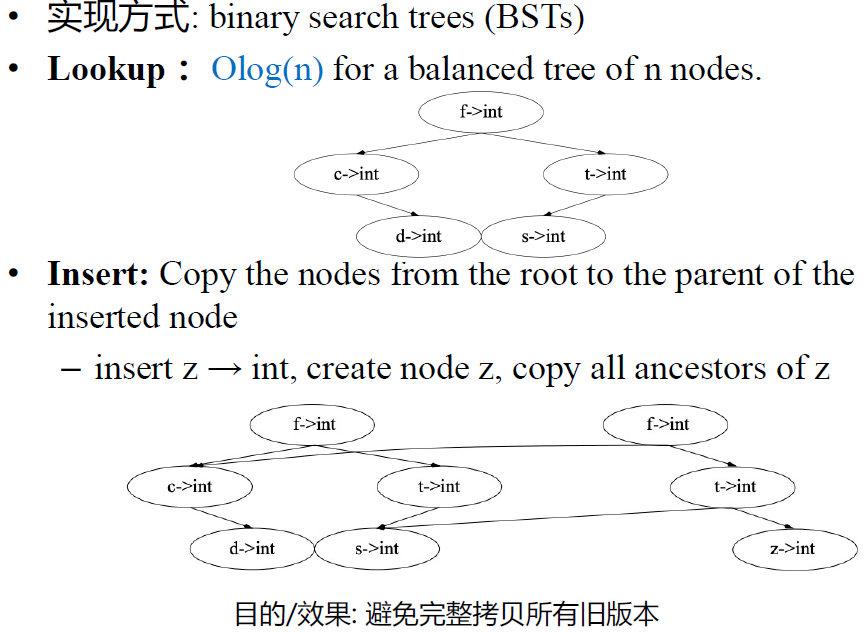
**Imperative Style:** 哈希表



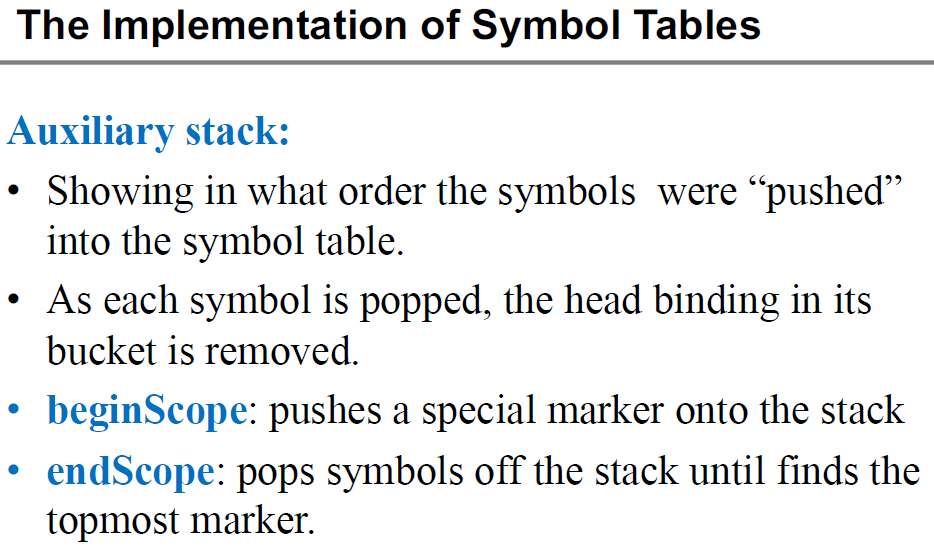
**Functional Style:** 永远保留老的，只是查询时做一些额外处理（可以理解为只是renaming）。这样还原更简单。

·直接使用BST（红黑树等）实现查找。

·可以使用可持久化数据结构完成删除、复原等操作，进一步降低单次操作的空间复杂度，非常方便。



**Tiger 符号表实现：**



**5.4 类型检查**

Tiger的类型系统包含：

1.原始类型(primitive type): int和string

2.构造类型(constructed type): record（结构体）和array

根据不同的判别法，类型等价这一关系分为：

Name equivalence (NE): 必须声明是同一个类型

Structure equivalence (SE): 内部结构一样

显然前者被广泛采用，Tiger语言也不例外。

Tiger存在两个独立的命名空间，不同命名空间的同名identifier不会互相遮蔽对方：Types 和Functions and variables

**CH6 活动记录**

**6.1 寄存器**

调用函数需要保护现场：因为寄存器的值可能会被子函数改变，返回时“现场已经被破坏”。为此部分寄存器的值需要在栈上进行备份。分为以下两种：

Caller-saved: 也被称为易失(volatile)寄存器。例如t系列临时寄存器。调用者如果用到则需要自己保存，子函数可以任意修改

Callee-saved: 例如FP/SP, 由子函数负责保存与恢复（进入子函数时push到栈，退出时从栈里pop），调用者无需关心

寄存器并非万能：有时，在栈上分配空间（实体化）是不可避免的。例如：

**1.** 对象过大，无法放在寄存器中

**2.** 数组对象，需要通过地址偏移访问

**3.** 寄存器被特殊需要，例如上文提到可能用于传参

**4.** 太多中间值/局部变量，有限的寄存器放不下，称为 Spill, 在寄存器分配部分会展开讨论

**5.** 变量 “逃逸(escape)” 了（也就是脱离了当前scope/无法确定变量有效的生命周期）：

·引用传参：需要内存地址（虽然对于现代语言经过优化并不总是需要一个地址）

·显式地取变量地址（C语言等）

·被嵌套函数访问（Tiger语言不需要考虑）

**块结构Block Structure**

动机：在允许函数嵌套定义的语言（比如Tiger）中，内部函数可能使用外部函数中的局部变量。

**1. 静态链接Static Link**

当内部函数g被调用时，调用者f传入一个指针指向f的栈帧（或者说活动记录）

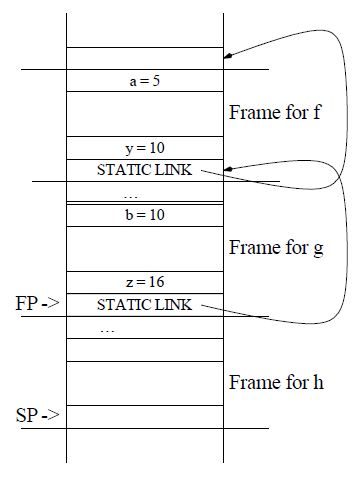
**·**这种情况下，我们说“f statically encloses g”

**·**如果多次嵌套，嵌套次数为N，这些指针会构成一个长为N的单向链表串联起栈帧

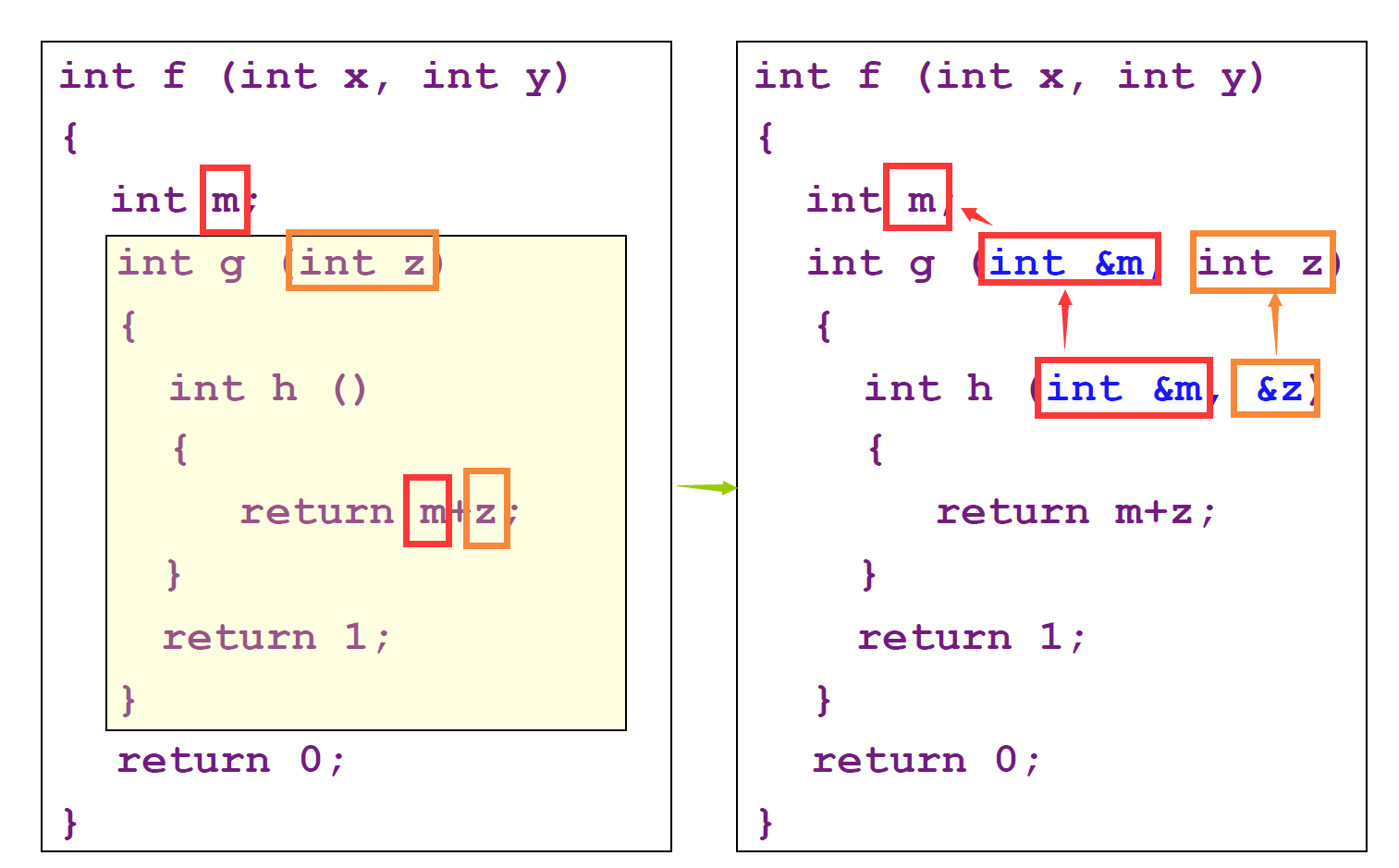
**·**每个函数记录自己的嵌套深度n

**·**如果访问了在深度m的变量，只需沿着该链向上n-m次就可以找到该变量所在的栈帧

**·优缺点**：Overhead小，但是因为要通过链表向上经过多层速度较慢

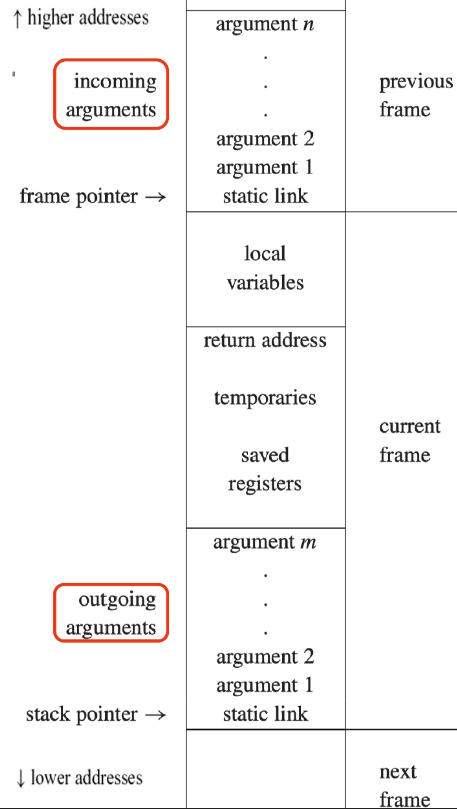


**2．Lamda lifting:** 从最深的一层叶过程开始，把所有g(a1)用到的外部变量o1 o2改写为真正传入的参数，于是变为g(o1, o2, a1). 如此逐渐向上改写每一层即可。



**3. Display数组：**一个全局数组，记录当前每个嵌套深度i对应的栈帧地址。这样不需要经过链表即可直接找到变量对应的栈帧。

Tiger栈帧布局：



incoming parameters: 调用者传入

return address: 返回CALL指令

local variables: 部分必须在栈帧中放置的局部变量

saved registers: 该函数保存的一些寄存器值

out-going arguments: 调用其他函数时传递的参数

FP/SP: 指向基址/栈顶

**CH7 中间代码生成**

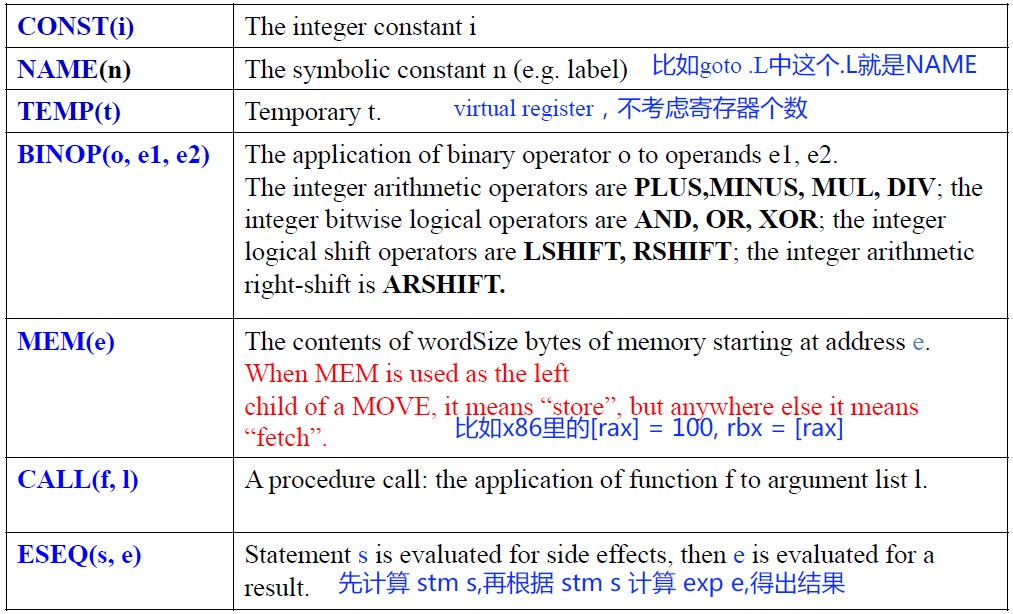
**7.1为何需要IR**

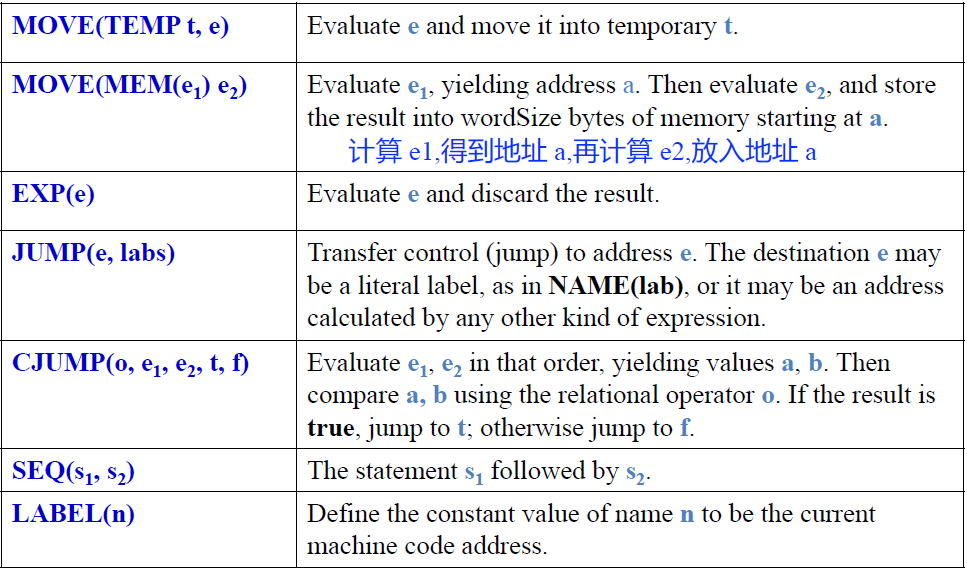
·更模块化，更可迁移（跨平台）

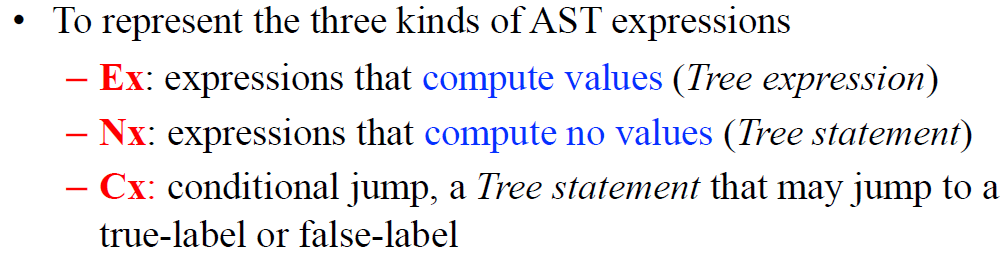
·多层的：分层应用不同的分析和优化（i.e. 变换）

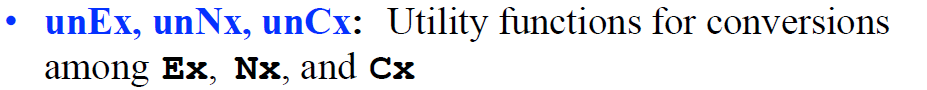
·可能丢失少部分机器特定的细节，但不会损失太多

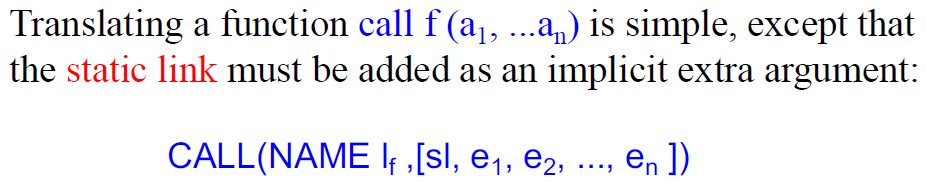
**7.2 Tiger IR**











函数定义分为三部分：

prologue:

函数入口的LABEL标记

更改栈帧的FP/SP以创建新栈帧

保存不逃逸的参数到可用寄存器

保存逃逸的参数到栈帧（包含Static link）

保存callee-saved的寄存器到栈帧（例如返回地址）

body: 翻译的函数体

epiloge:

保存返回值到寄存器（或栈上某地址）

从栈帧中恢复callee-saved的寄存器

回退到调用者的栈帧（更改FP/SP）

Return指令

（可选）伪指令（标记函数在此结束）

**CH8 基本块与traces**

**8.1正规树Canonical Tree**

Canonical tree 的性质：

1. **根节点为语句(statement)** ，所以正规树就对应IR中的一个语句，也就是IR Tree中的一个子树。

只不过因为这些额外约束，所以子树可能需要经过变换。**其他所有节点都是表达式。**

2. **不存在SEQ或者ESEQ**。

SEQ: 用于分割出这些canonical tree, 自然被删了不存在。

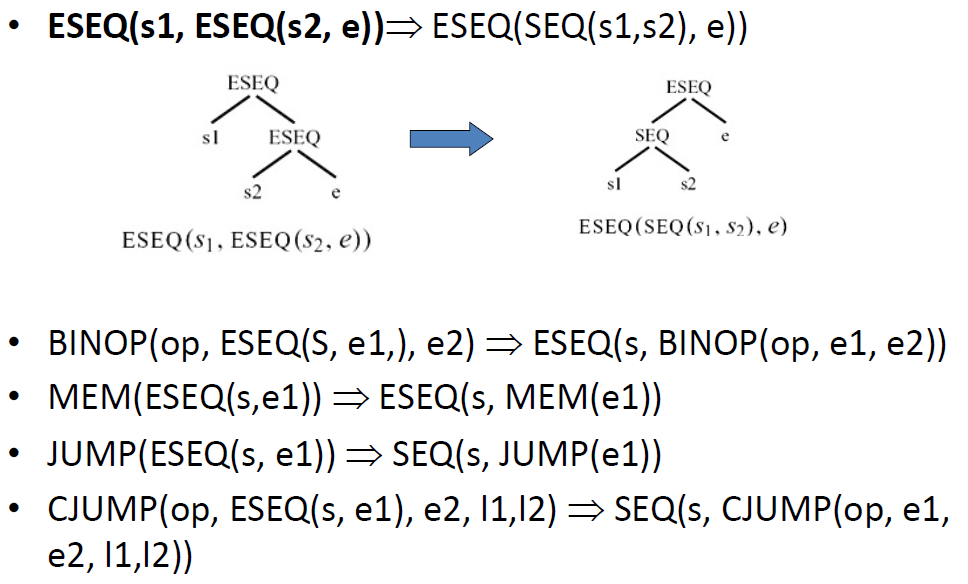
ESEQ: 被变换成SEQ然后也被删了（详见下文）。

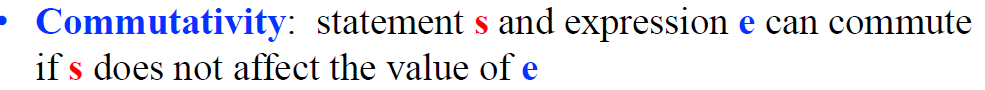
3. **每个CALL节点的父节点即为canonical tree的根节点**，且只能为EXP或MOVE.

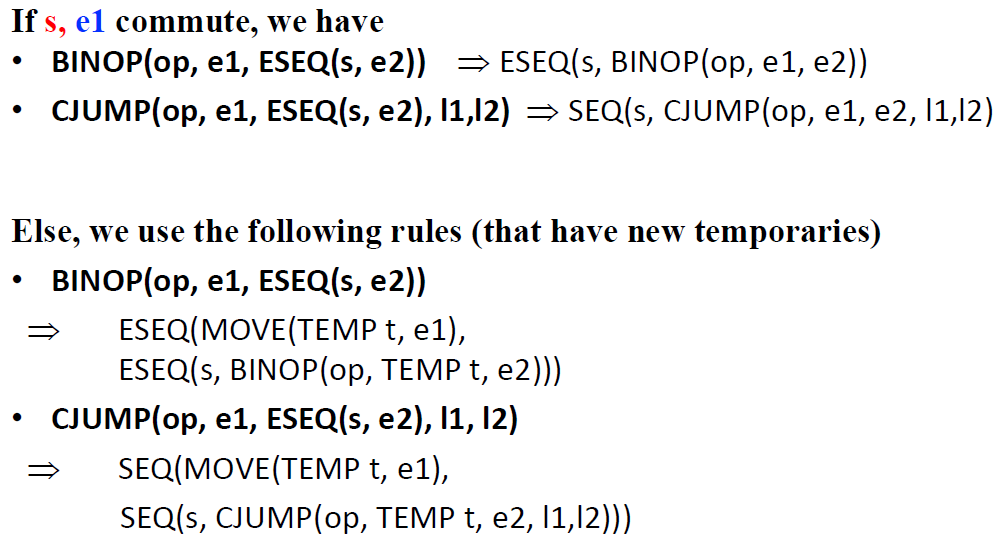
事实上至多只能有一个CALL节点，因为我们规定EXP或MOVE只能包含一个CALL.

**8.2线性化为正规树**

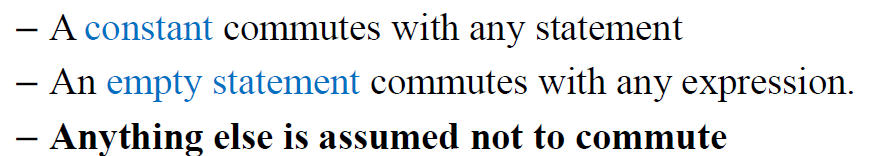
**1. 消除ESEQ**







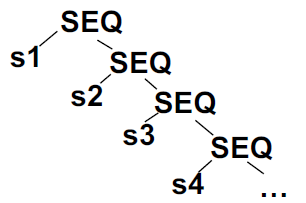
保守估计是否commute



**2．移动CALL到顶层**

对于多余的CALL(fun, args) 转化为 ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

**3. 重排并消除SEQ**



**8.3 处理条件跳转**

**1. 组装为基本块(basic block)**

基本块指的是一串代码，进入则一路执行到块末尾。性质：

·开头是一个 label 伪指令（入口唯一！）

·最后一个指令是跳转、条件跳转或返回等terminator

·中间不包含其他 label/terminator

**2. 生成trace**

基本块如何排列并不影响程序执行结果。

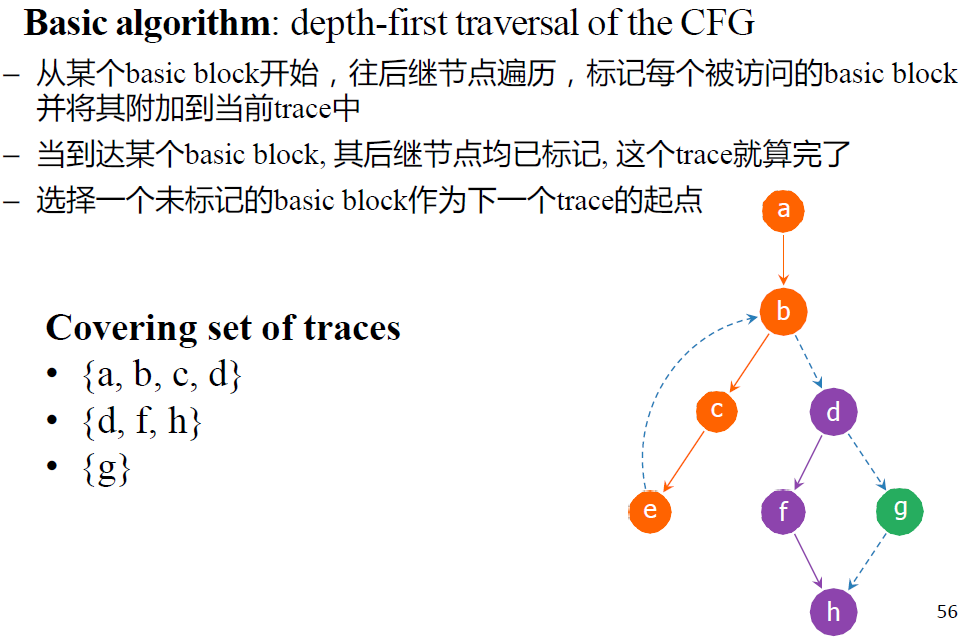
然而我们需要进行一些优化，尽可能减少跳转或提高跳转的效率。例如：

·对于CJUMP(cond, true, false), 将false label对应的块紧挨着放置在CJUMP指令后，这样执行时就可以fall through

·尝试将无条件跳转JUMP的目标label对应的块直接放在JUMP后

·如果可以，甚至可以直接删除部分JUMP, 例如将函数内联

·优化缓存命中率



**CH9 指令选择**

**9.1 把IR转化为 抽象汇编代码：**

·具有有限个寄存器的汇编代码；

·为中间结果创造新的临时寄存器

·在之后的流程中，会把这些寄存器映射到物理寄存器

指令选择要解决的问题：IR的一个语句有多种可能的实现方式，需要确定为其中“最好”的一种。

**9.2 基于树覆盖的指令选择**

为了方便起见，接下来假设目标机器码是一个简单的架构（指令集）：Jouette架构。它基本上就是RISC-V的子集。我们约定：

·寄存器r0永远为0

·数据/地址能直接在寄存器中放得下

·每时钟周期只有1个指令执行

·每个指令延迟1时钟周期（MOVEM指令除外）

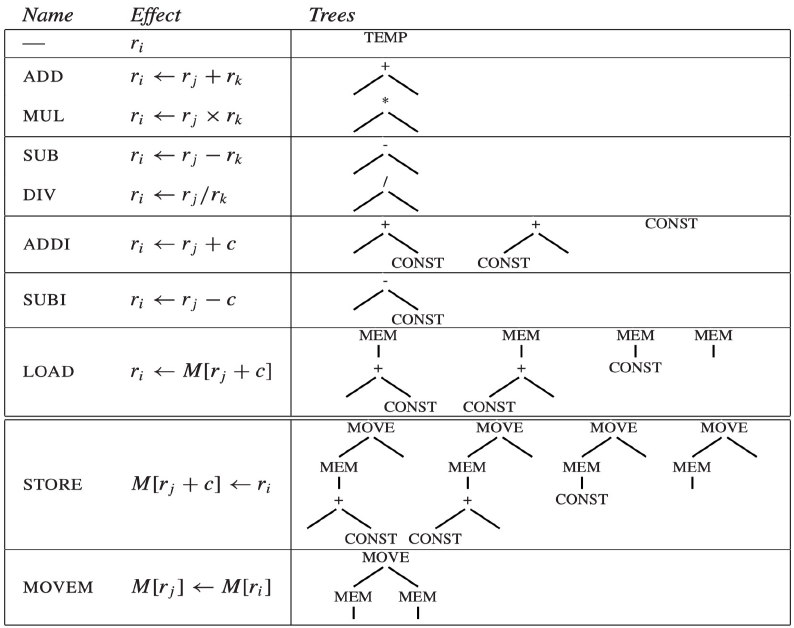
**1. 宏展开**

·公式化地，把每条IR直接展开成一或多条机器指令

·缺点：缺乏优化，只能1->1或1->N, 但多条IR指令有时明明可以合并(N->1)

**2. Tree Patterns**

**Tile** ：模式匹配到的一个实例



**Optimum tiling全局最优：**所有tiles总代价最小

**Optimal tiling局部最优：**没有相邻的tile可以被合并成一个tile了

**9.3 指令选择算法**

**1.** **Maximal Munch**

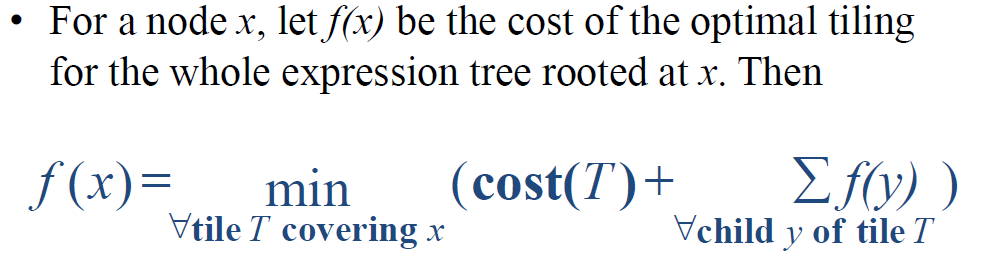
·自顶向下

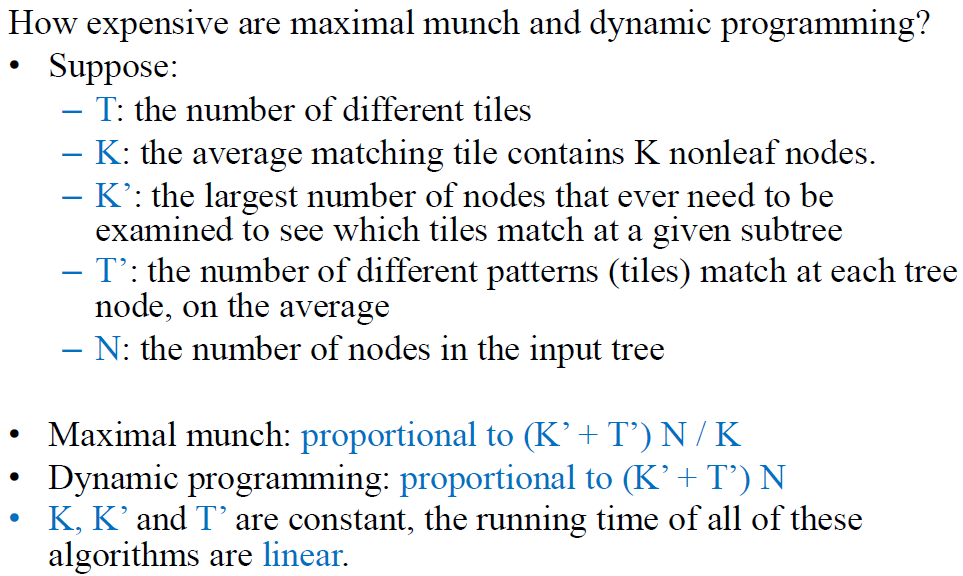
·用可用的最大tile覆盖当前节点

·递归地对（那些根节点）还未被覆盖的每个子树应用该算法

·把所有tile对应的指令按照覆盖顺序的逆序组成机器码

**2．Dynamic Programing**

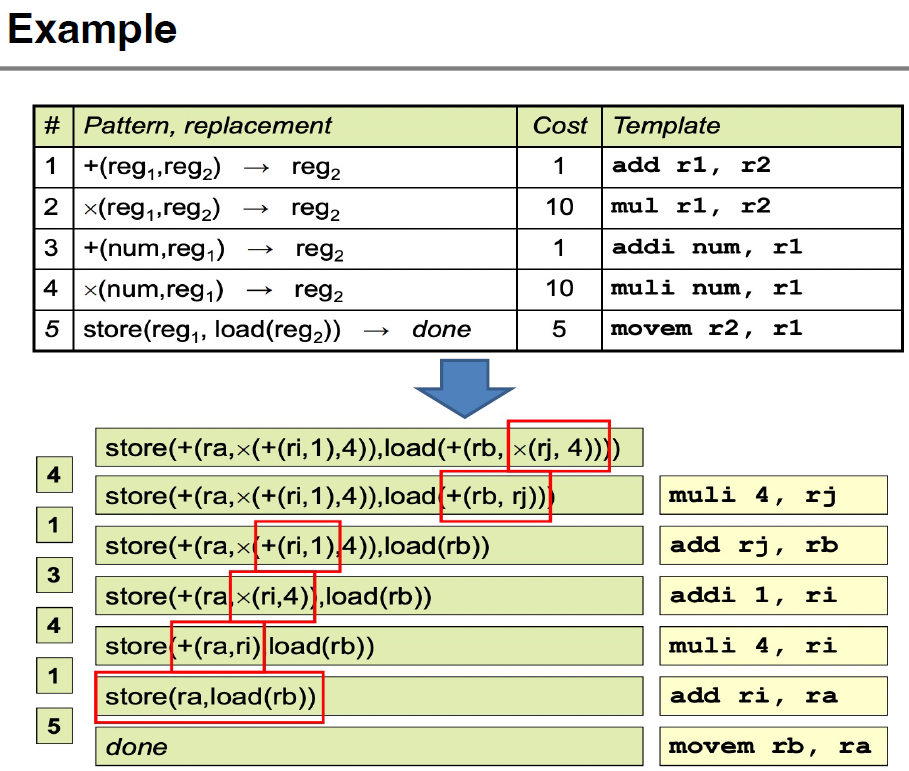




**3. Tree Grammar**

–将指令选择简化为parse问题！

–使用动态编程算法的泛化进行



**CH10 活跃变量分析**

**10.1 数据流分析Dataflow Analysis**

我们可以从CFG（或其他类型的中间表示）中静态（i.e. 不需要真的运行程序）推导出一些程序的行为信息。

本节中CFG节点代表一个语句(Stmt)

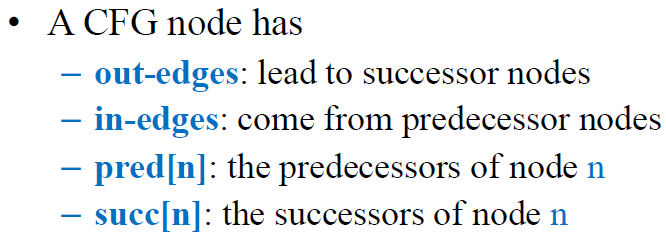
**10.2 活跃性分析Liveness Analysis**

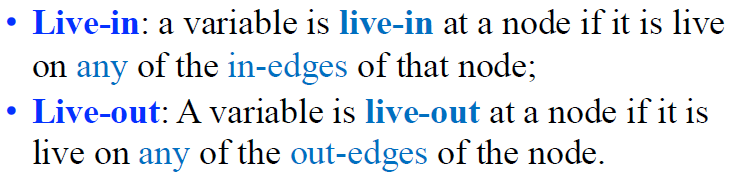
定义变量的liveness:变量x在执行到一个语句s时是活跃(live)的，当且仅当：

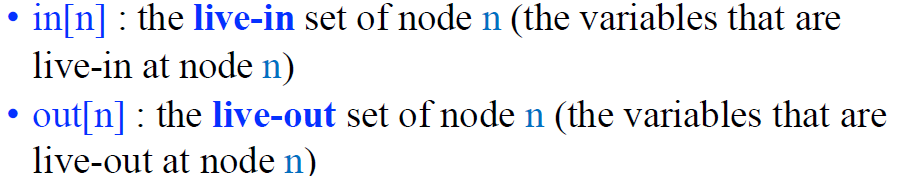
·存在某个语句s' use x

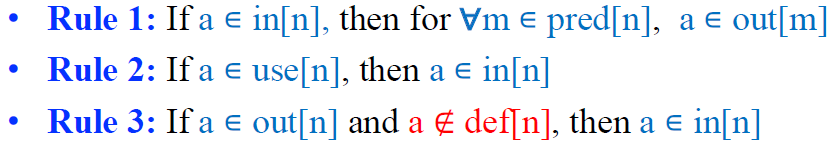
·存在一个执行路径，能从s执行到s'

·在某个这样的路径上，x没有经过def









**根据上述定义，可以迭代计算活跃变量集合：**



**活跃性分析杂项**

1. 把多个相邻的节点尽可能合并为基本块可以加速。

2. 根据变量多少，视具体情况用 bitmap 或有序列表等方式储存集合可以更快。

3. 实践中也可以每次专门求解某一变量x的数据流（in/out），因为大部分中间临时值的 live range 很短。

4. 这一算法最差的时间复杂度可能为, 其中N为节点个数

5. 算法求解的是“最小不动点”：极端情况下，你当然可以让所有变量都是live的，这也是个不动点，但不是我们想要的。

6. 进一步优化算法：可以用队列记录有可能需要被进一步更新的节点（类似BFS）。

7. 变量活跃性可以分为：

* 动态的dynamic liveness: 运行时判断，显然是一种under-approximation, 因为不能覆盖所有可能执行路径。
* 静态的static liveness: 像刚刚做的一样在编译期判断，显然是一种over-approximation, 因为估计得很保守，而事实上有些路径不可能被执行到。

**CH11 寄存器分配**

**11.1** **寄存器分配算法的任务：**

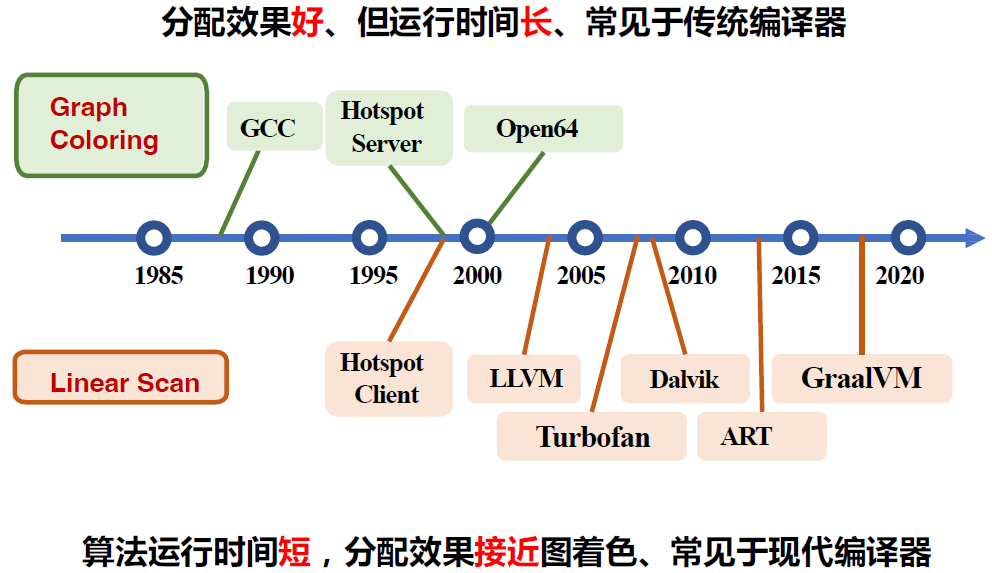
将 IR 中大量的虚拟寄存器（储存变量或者运算的中间临时结果，统称 temporaries）分配到固定数量的 k 个物理寄存器上，保证代码使用不超过 k 个寄存器

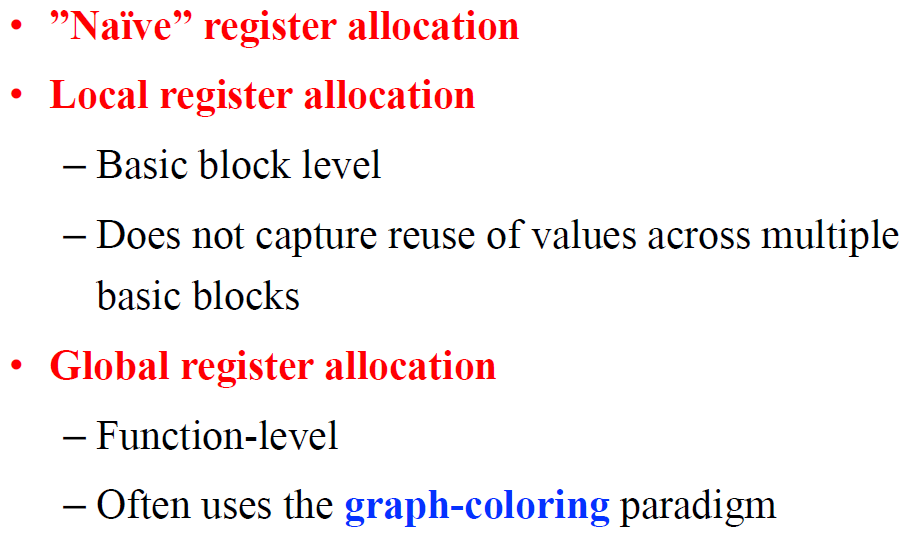
·使得内存访问、储存尽可能少

·使得内存上用于储存 spilled 值的空间尽可能小

·算法需要尽可能高效

典型的复杂度：O(n) 或 O(nlogn)





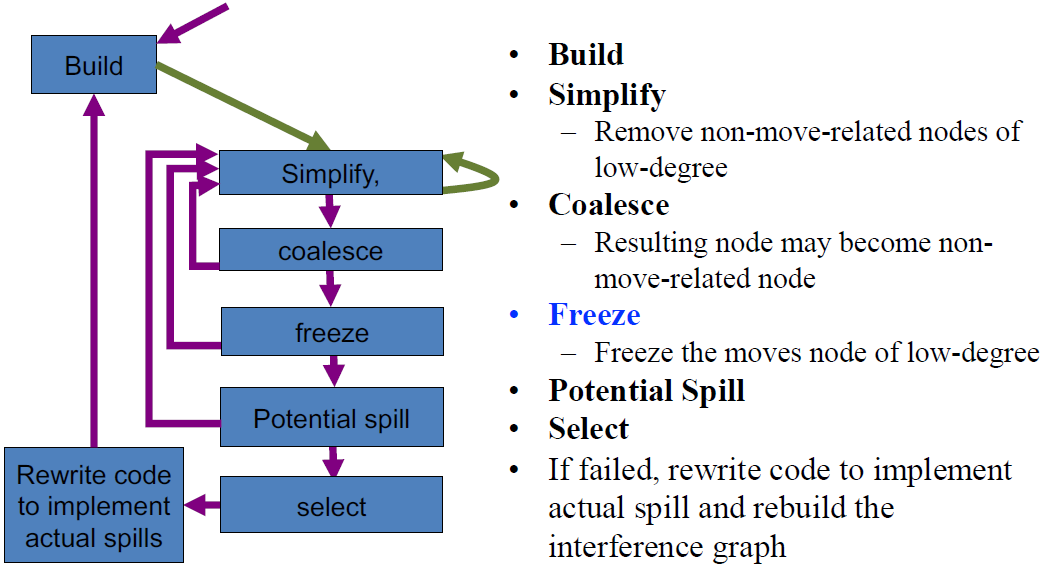
**11.2 图染色Graph Coloring**

相交图 = 干扰图 = interference graph = conflict graph

如果两个虚拟寄存器a和b因为某种约束限制，使得它们无法被分配到同一个物理寄存器中，我们就说a与b之间存在 干扰(interference) ，这些约束包括但不限于：

·live range 有重叠：某个时刻a和b都是活跃的，值需要得到保留

·产出/使用a这个值的指令比较特殊，不能使用某些特定寄存器



**以逆序对节点进行染色**

**保守合并策略** coalesce:

**Briggs**: 确保合并出的新节点的相邻节点中，significant 节点（度数>=k的节点）数量少于 k 个。

**George**: 对于a的每个相邻节点t, 要么t是 non-significant 节点，要么t和b之间本来就有 interference 边。

**预染色(pre-colored) 节点：**栈顶指针寄存器、参数寄存器等 special purposed 寄存器不是通用的。

·可以把这些物理寄存器一一对应成虚拟寄存器，或者说为每个 special purposed 寄存器分配一个专属 temporary / node.

·每个 pre-colored node 的颜色是唯一的

·pre-colored nodes 之间互相两两 interfere （不一定需要真的全连接，也可以特殊处理）

·预染色节点不能被 simplify 掉。

·预染色节点不能被 spill 到内存上。

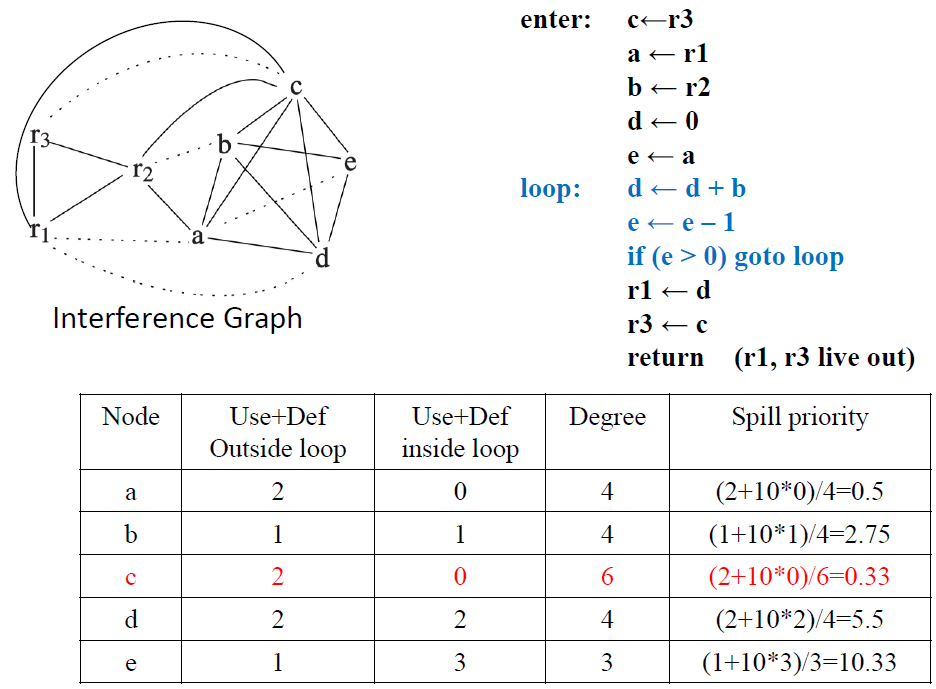
·预染色节点能参与合并(coalescing).

·普通的 temporary / node 可以使用预染色的颜色（只要不 interfere）

**编译器前端需要保证预染色 temporary 的 live range 足够短**

·temporary 的 live range 如果不需要跨越某个函数调用，最好放置在 caller-saved 寄存器中

·temporary 的 live range 如果需要跨越某个或多个函数调用，只能放置在 callee-saved 寄存器中



其中r1, r2为caller-saved, r3为callee-saved

**CH12 垃圾回收**

**12.1 运行时Runtime**

Runtime System = 程序默认的、不在编写时描述的builtin 方法等运行时环境信息。可能有：POSIX 信号处理，任务智能分配到多核 CPU 的不同核心，虚拟机（比如 JVM），智能内存管理（例如垃圾回收）

**12.2 内存管理概述**

手动内存管理存在许多安全隐患：内存泄漏，double frees, use-after frees

内存管理需要平衡性能、安全以及开发难度。

**内存区域划分：**

1. Static area：编译期分配，全局变量、字符串字面量等

2. Runtime stack：也就是栈，储存许多栈帧 / 活动记录，包含局部变量、函数调用返回地址等信息

3. 堆(heap)：通过malloc new 等方式在程序中动态分配的对象，如果这些对象不再使用则称之为 垃圾(garbage)

**垃圾回收判定是NP问题**，因此我们近似通过判断内存对象的可达性(reachability)来找出哪些对象可以 / 无法通过指针，或是指针链访问到。不可达的对象一定不可用。

垃圾回收分为两个阶段

garbage detection: 检测已经成为垃圾的内存对象

garbage reclamation: 释放这些垃圾对象

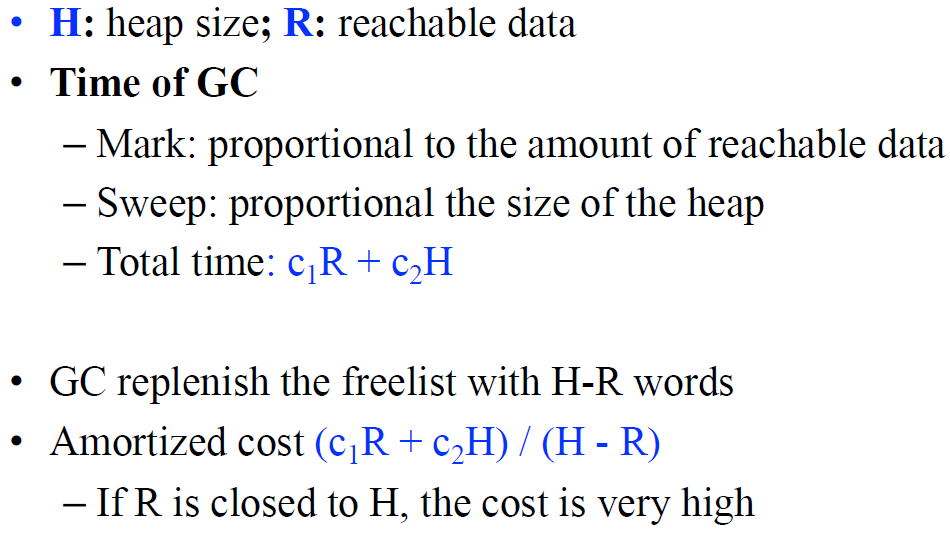
**12.3垃圾回收算法**

**1. Mark and Sweep**

一种朴素的垃圾回收方式。有如下两步：

·Mark: 从各个根节点出发，搜索并标记能到达的全部节点。

·Sweep: 线性扫描整个堆，把未被标记的节点（不可达节点）链接到 freelist 中, 最后清除标记。



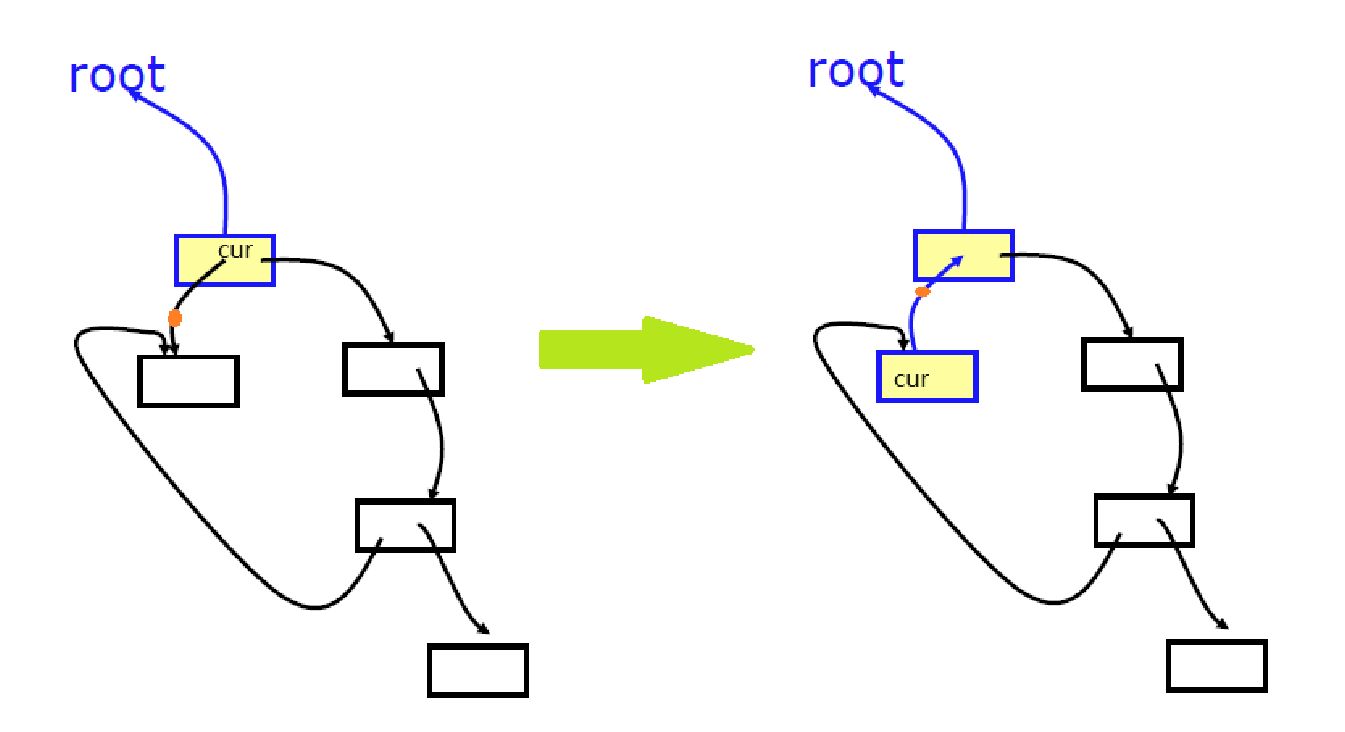
**Problem 1:** 遍历可达对象时如果直接 DFS, 在极端情况下容易递归过深爆栈。

Sol : **Explicit stack**：手写栈模拟递归

**Problem 2:** 即使模拟递归，栈的空间开销过大（最多为heap大小）

**Sol：Deutsch-Schorr-Waite (DSW) pointer reversal**

不使用显式的栈，而是复用图中的元素（节点与边），把栈“放在”图中：在图中用指针串起来一连串节点，也可以充当栈



+ 垃圾数量较少时很高效

+ 可以回收循环引用（A->B->C->A...）

+ 对象在内存中不需要移动，也就是其位置不需要更改（有时影响程序正确性：参考隔壁 Rust 的 std::pin）

- 垃圾数量多时很低效

- GC 过程会阻塞程序运行，程序必须暂停以回收垃圾

- 容易导致堆的碎片化，尤其是会产生许多内存（外部）碎片

**2. Reference Counting**

**核心思想**：在释放对象时就进行垃圾回收，而不是等到申请内存时发现空间不足才试图回收垃圾。

1. 当执行x.fi:= p时，我们需要维护 RC(reference count) 值：

·p的 RC 值+=1 .

·x.fi之前指向的对象 RC 值-=1 .

2. 如果此时发现某个对象r的 RC 值变为 0, 也就是无人指向它时：

·将r加入 freelist.

·r的各个 fields 指向的对象 RC 值-=1 .

·这可能会使得其他对象 RC 值也下降到 0, 从而触发连续回收过程。

+ GC 带来的开销(overhead)是渐进式的：不需要专门为了进行 GC 暂停程序执行，而是融合到程序执行过程中（无需 stop-and-collection）

+ 实现简单

+ 可以和手动内存管理并存（例如同时使用裸指针和智能指针）

+ 内存局部性好：RC 值和对象储存在一起

+ 垃圾能被立即回收（而不是等待直到下一批统一回收）

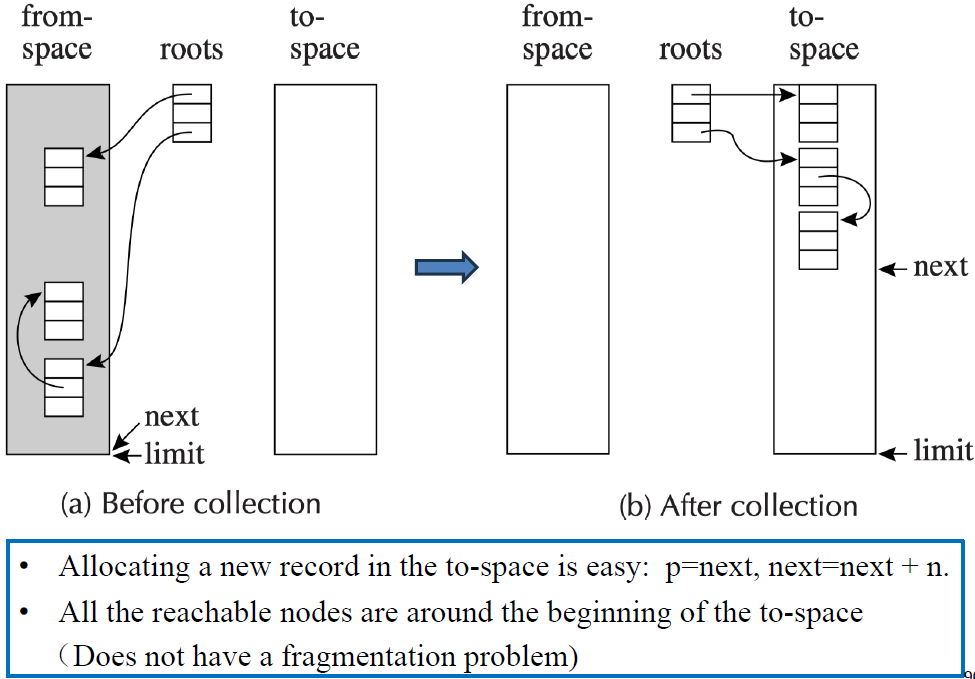
两个严重问题：

- 不能回收循环引用（A->B->C->A...）。相互引用可能导致它们的 RC 值都不为 0, 成为一个永远不可达的”孤岛“

- 增加/减少 RC 值是非常频繁的，带来非常昂贵的开销

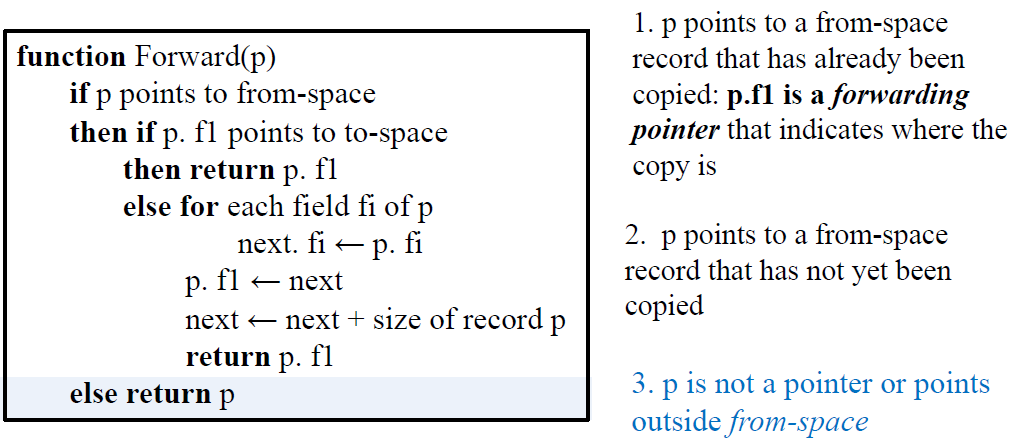
- 增加/减少 RC 值需要在目标代码中加入大量指令

**3. Copy Collection**



此外，我们还需要进行 **指针转发(pointer forwarding)** . 这是因为我们移动了一些对象，改变了其内存地址，因此原先指向这些对象的指针地址也需要相应地被修改。

……但这样做需要在找到、储存每个对象对应的新地址，开销太大了，因此我们在每次拷贝时，直接把新对象x'的地址储存在 from-space 已经用不到的对象x开头,然后在拷贝时，每次访问指针p时，通过forward(p)翻译到真实地址

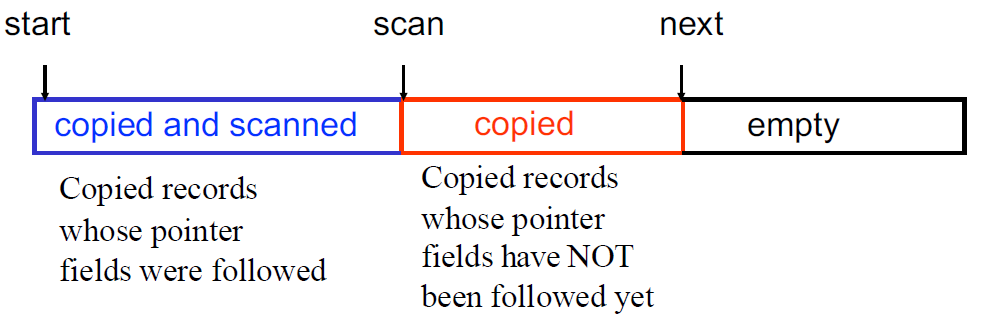


**Cheney’s Algorithm**: 使用 BFS 遍历可达对象并拷贝：

1. 初始时，拷贝根变量(roots)指向的内存对象，这些对象进入 copied 队列。

2. 处理 copied 队列中对象内部的指针，这些对象指向的内存对象被拷贝，进入 copied 队列；这些对象本身成为 copied-and-scanned 对象，移出队列。

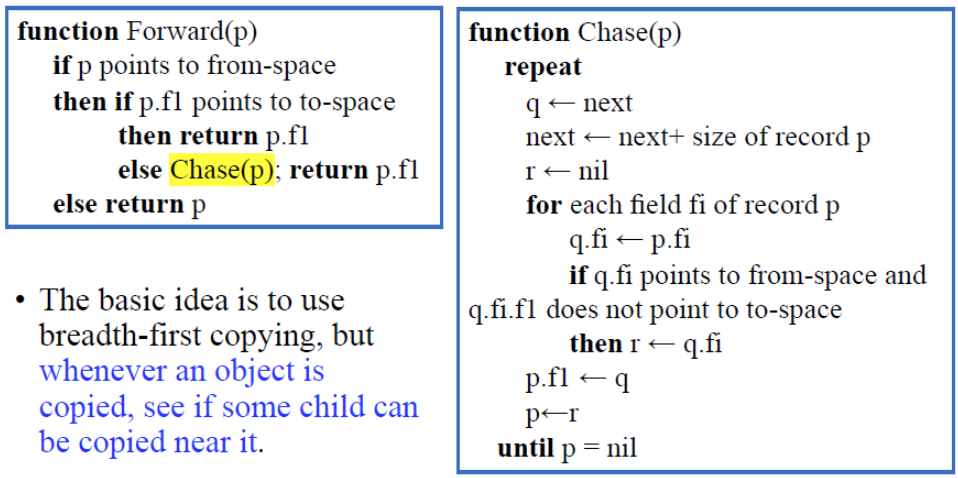
3. 重复以上两步直到队列空。



·BFS算法的内存局部性较差：x中的指针x.fi指向的对象可能地址离x很远！这导致无法很好的利用虚拟内存，也会造成许多 cache misses.

·DFS算法的内存局部性较好，但需要 pointer-reversal, 也很慢，且麻烦。

**Hybrid algorithm**



+ 实现简单：无需用栈或 pointer-reversal

+ 跑得比香港记者快，时间复杂度优秀：和活跃对象数量成正比

+ 可用的 free space 在内存上连续：每次 GC 时自动进行了“压缩”，去除了内存碎片

+ 是更新的一些算法的基础范式，起到了示范作用

- 高达一半的空间被浪费了

- 内存局部性差：Cheney's algorithm

- 需要精确的类型信息标出所有指针（如果不知道某个 field 是否为指针，就无法在移动对象后转发该指针）

**12.4 快速内存分配**

为了高效分配内存，我们可采用 Copying Collection: 可分配空间连续，且起始位置(next)、区域末尾(limit)已知。

每次在内存上分配一个大小为N的 record 时，有如下开销与对应优化：

1．~~Call the allocate function~~通过内联展开，可消除函数调用

2．Test next + N < limit ? (If the test fails, call GC)

3．~~Move next to result~~

可以和A合并：Move next into some computationally useful place

4．~~Clear M[next], M[next+1], …, M[next + N - 1]~~

可以通过和 B(见下)合并消除（或者干脆不初始化）

5．next <- next + N

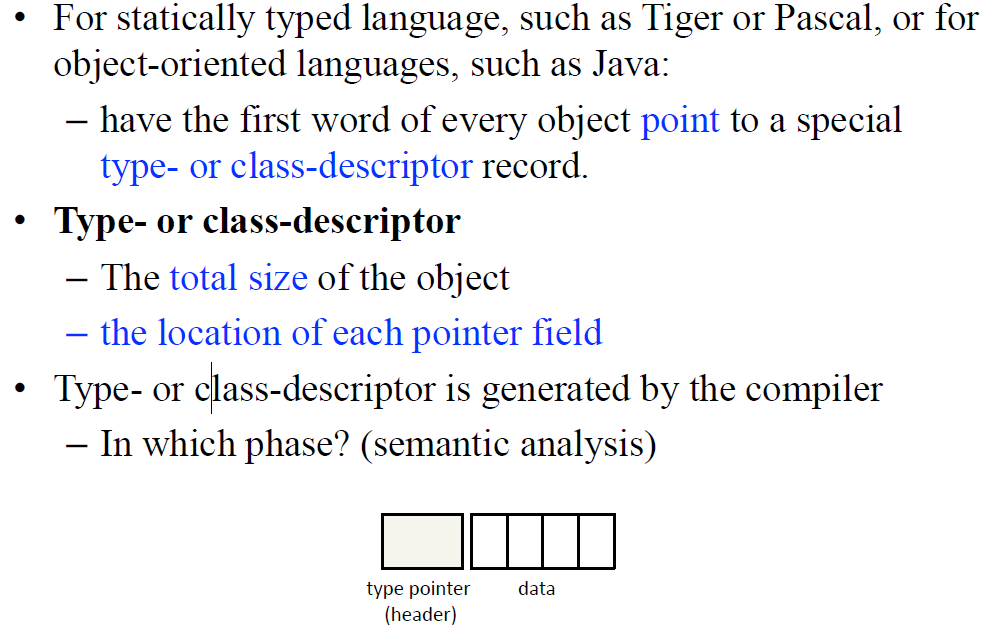
6．~~Return from the allocate function~~通过内联展开，可消除函数调用

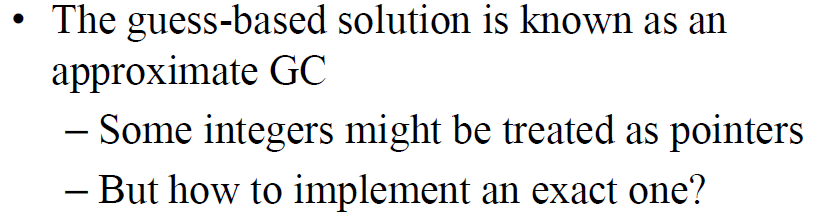
以及如下不属于分配过程的开销：

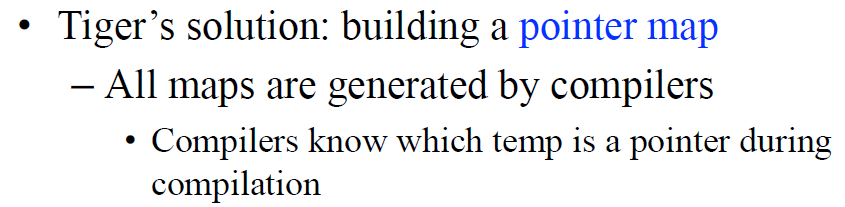
A. Move result into some computationally useful place (Done in step#3)

B. Store useful values into the record (Done in step#4)

**12.5 垃圾回收接口**







**·**如果垃圾回收器在分配时调用：在执行每次分配 record 的指令前生成 pointer map

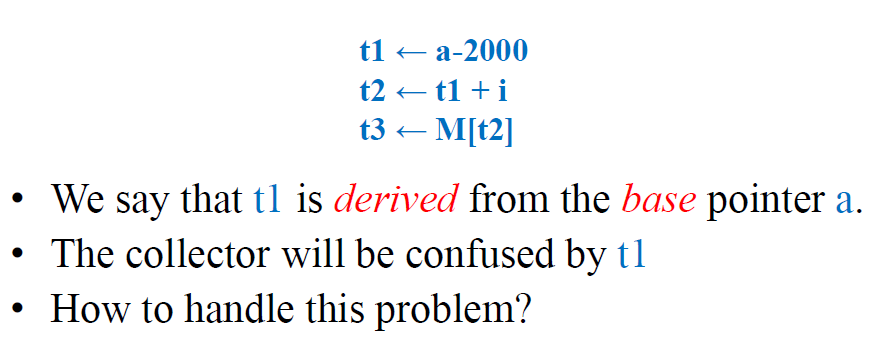
**·**递归调用时：每次调用函数CALL前生成 pointer map

**·**由于我们调用函数CALL前生成 pointer map, 所以通过每个返回地址可以索引到下个（被调用函数）栈帧的 pointer map.

**·**垃圾收集器通过从栈顶向下扫描调用栈,依次处理每个栈帧。每处理完一个栈帧,就可以通过该栈帧的返回地址来获取下一个栈帧的 pointer map, 识别出栈帧中的指针。

**·**需要特殊处理 callee-saved 寄存器：考虑调用链 f->g->h, g 在调用 h 前，对于这些 callee-saved 寄存器，必须在 pointer map 中额外记录哪些是 g 自己使用的，哪些是从 f “继承”来的。

**12.6 Derived pointe:**有的指针可能指向某个record中间，或者根本不指向一个record.



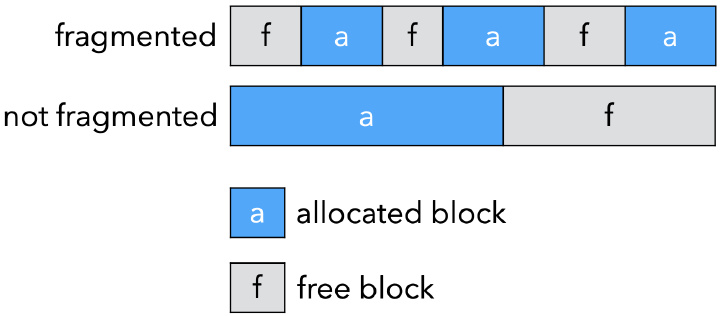
**pointer map必须识别出每个derived pointer并把信息给base pointer:**

·如果a的地址在 GC 过程中发生了改变，则t1的地址也需要被修正

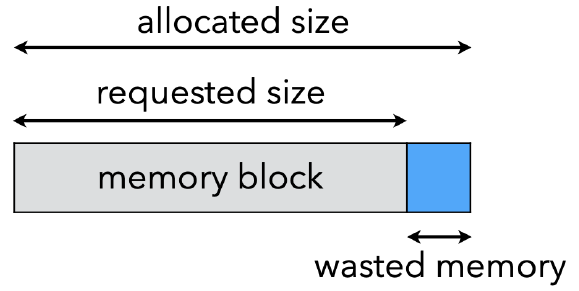
·a至少要比t1活得久，否则在 GC 时 pointer map 就无法有足够的信息修正 derived pointer.

**12.6 内存碎片**

**外部碎片(external fragmentation):** 当程序要求分配一块内存时，尽管堆中看起来可用空间足够，但由于这些可用空间零散、碎片化地分布在各个位置，成为了外部碎片，我们找不到足够大的一段连续内存分配给程序，分配失败。



**内部碎片(internal fragmentation):** 当程序要求分配一块内存时，由于地址对齐等原因，内存管理管理器可能返回一块比要求的更大的内存，用不完的部分即为内部碎片，将不可避免地被浪费。



**CH14 面向对象(OOP)**

**14.1 类 Class**

语法：class B extends A {...}

·子类B继承自父类A, 或者称派生(derived)类B继承自基类A.

传递性：如果C继承了B, B继承了A, 我们可以说C继承了A.

·B隐式地继承A的所有 fields(成员变量) 和 methods(成员方法).

·B可以重写(override) 部分 methods. 注意和重载(overload) 区分：

**重写(override)**: 函数名相同，且函数签名必须一致；派生类重写某个 method 以添加和基类不同的实现。

**重载(overload)**: 函数名相同，且函数传入参数必须不同；可以为某个函数在不同传参下的多个版本、多个实现。

·**fields不能被重写**(override). 派生类中定义的所有fields都是在派生类中新定义的.

例如基类A有一个 field name, 而派生类B也有一个 field name, 则实际上它们是不同的 fields A.name 与 B.name. 并非重写基类的 field

·所有类都是预定义Object类的派生类，Object没有field或method.

·关于self: 在 Tiger 中不是关键字，而是每个 method 都有的隐式的参数。在运行时自动传入，指向当前实例，用以引用当前实例对应的 fields 和 methods。

**14.2 类的层次结构**

·单继承single-inheritance(SI): 每个类最多只能继承一个基类，因此继承关系图是一棵树。

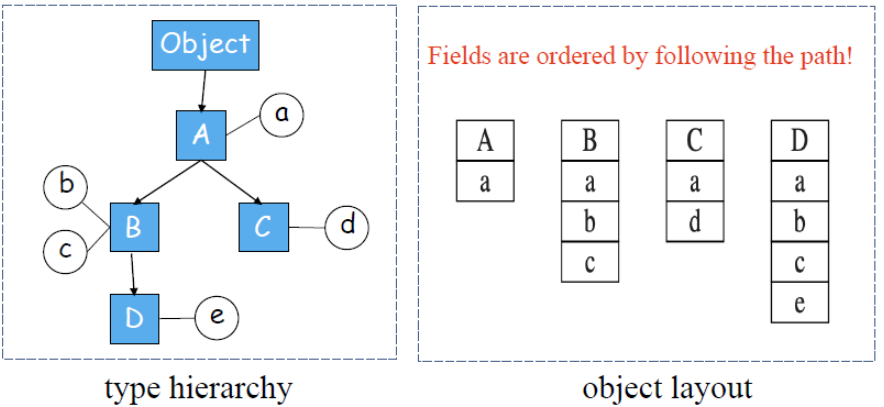
例如 Python

·多继承multiple-inheritance(MI): 每个类可以继承多个基类，因此继承关系图是有向无环图(DAG).

例如 C++, Perl, Python

实现起来更 tricky

**1. 单继承**



**Method dispatch:**

**1. Static Method**

· 每个 method 编译成一段代码（称为 method instance）

· 每个类都对应一个 class decriptor, 里面包含了描述这个类的一些必要信息：指向基类的指针，包含这个类所有的 method instances列表

·对于 **static method** 的调用x.f(), 编译器将会：

1. 找到对象x对应的类，记为C.

2. 如果C中有f, 则直接得出x.f() 翻译结果为C\_f; 否则继续向上寻找。

3.假设C的基类为B, 在B中查找f, 如找到则得出x.f()翻译结果为B\_f; 否则继续向上（在基类中）寻找。

4. ……

5. 直到在某个祖先中找到为止（若到Object还没有则报错）

**2. Dynamic method**

· 每个类维护一个 dispatch vector (例如 C++ 中的虚表 vtable) 储存每个 method 的地址。

· 建立方式类似 prefixing: 派生类中新声明的方法跟在基类 dispatch vector 的后面；不过如果有基类的 method 被重写了，也要替换成自己重写后 method 的地址。

·每个对象都关联某个 vtable: 对象的开头储存一个指针指向对应 class descriptor , 里面就有 vtable

·需要动态查找(lookup), 有额外的开销

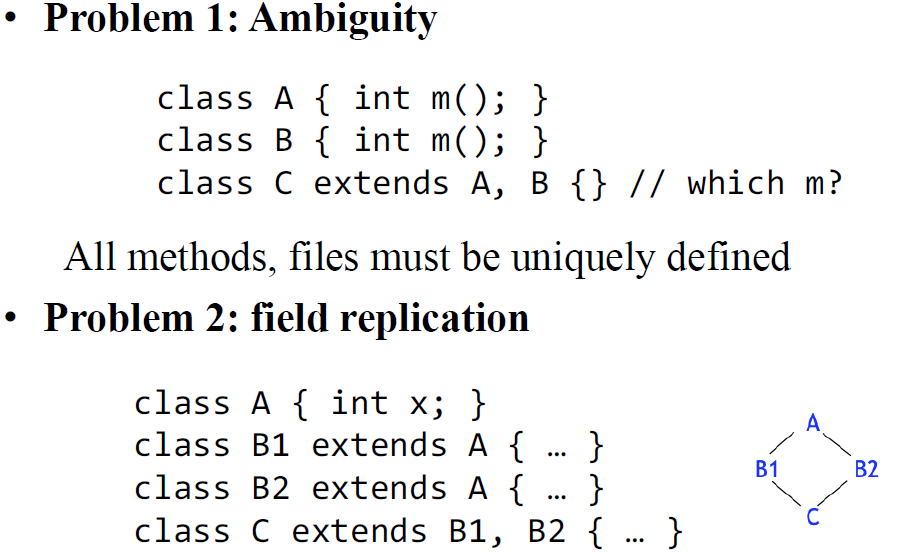
·对于 dynamic method 的调用x.f(), 编译器将会：

1. 在x的0偏移处（开头）找到 class descriptor d.

2. 由于方法f的偏移量是确定的，从d中F偏移处获取f的函数地址。

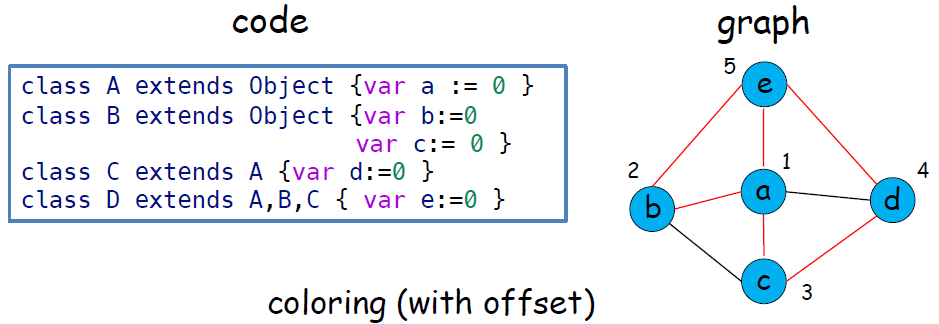
3. 调用f.

**2. 多继承**

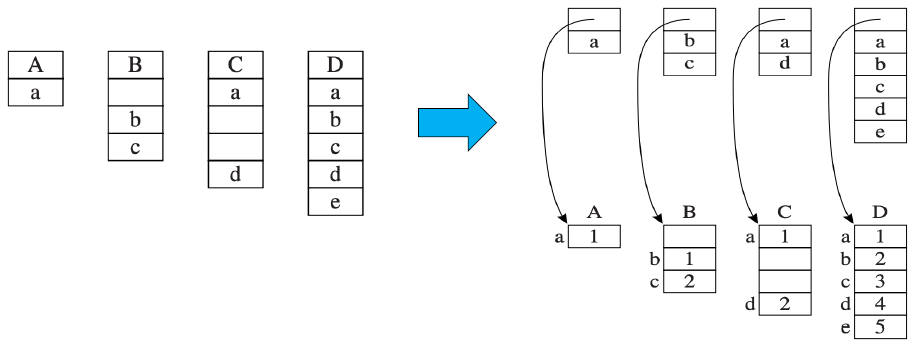


**Field layout**: 通过图染色获取。

目标：静态分析所有类，为每个 field 的找到一个固定偏移量。如果不同 field 在同一个类中出现，则不能共享同一个偏移量。



优化：许多空的 slot 被浪费了。我们可以把 fields 在内存上合并，转而在每个类的 class descriptors 中记录各个 field 的真实偏移量。



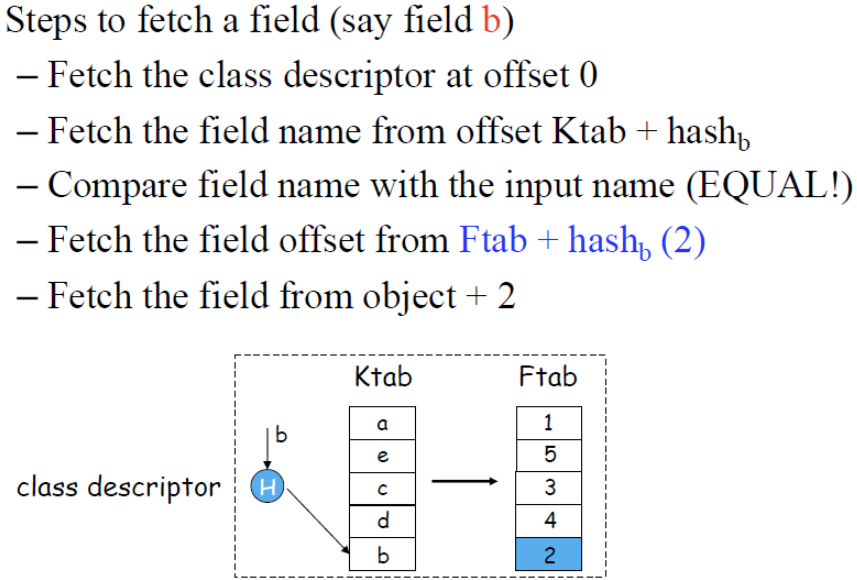
·**由于类的数量远少于对象数量**，所以这样能节省空间。

·不过这种优化导致每个 field 的具体偏移不固定了，因此需要在运行时在 class descriptor 中动态查找(lookup) field 的真实偏移量.

**Method dispatch**: 仍然使用图染色。我们可以直接把 method 名混合进上述的图中，一起染色；也就是不仅记录 field 的偏移量，也记录 method 的地址。

并非所有时候都可以静态知晓所有类的存在并统筹规划。例如 Java 里类可以被动态加载。

**Hashing：**



**14.3 基于类的类型系统**

每个类都是一种类型。在进行类型转换时：

upcast: 派生类转为基类。永远是安全的。

downcast: 反之，基类转为派生类不一定安全。

**如何判断x是不是类C的实例？**

·朴素（慢）的做法：

1. 向上递归检查x 的class descriptor (记作x.0) 开始的继承链

2. 如果发现某个是C的class descriptor 则x是C的实例；否则如果到最上层（...super==NIL）仍未发现，说明不是C的实例。

·**display：**每个 class descriptor 储存一个 display, 也就是一个的定长列表，记录对象的整条继承链。就像：

0: Object

1: GrandparentClass

2: ParentClass

3: MeClass

4: (nil)

…

·给每个类一个专属的数字ID， 然后display中储存这些ID代表类

·由于对每个类，其继承关系的嵌套深度在编译期已知，因此可以立刻找到需要比较 display 中的哪一项。

·例如，假设MeClass的继承深度是第 3 层，要检查x是否为MeClass的实例，只需检查x的继承深度是否大于等于 3, 且x.0.display[3]是否指向MeClass的 class descriptor 即可。

**14.4私有 field/method**

private field/method: 私有的 field/method 只能被类的其他 method 访问/调用，而不能被外部调用。

·这是封装思想的体现：调用者不该知道内部实现细节。

·通过类型检查确保私密性(privacy)：每个访问处检查是否private.

**CH18 循环优化**

**Low level**: 针对单个循环

·loop invariant code motion: 把循环中不变化的代码移动到循环外

·strength reduction: 强度折减，用代价低的运算取代代价高的运算

·loop unrolling: 循环展开，增加每次循环迭代的元素个数，减少小循环的循环开销

**High level:** 重构、修改多个循环

·loop fusion: 融合多个循环，减少循环开销

·loop interchange: 调换内外循环顺序，提高内存局部性等

·loop tiling: 循环遍历数组时，通过适当的重新分块，提高内存局部性等

**18.1 循环Loop：**在CFG体现为 一个节点集合S, 包含 header node h, 并且对于S中的任何节点x:

·都有一条从x到h的路径

·都有一条从h到x的路径

·除了h, 没有任何其他S以外的节点能到达x

**Loop entry**: h是唯一一个能从外部到达的节点，是循环的唯一入口

**Loop exit:** 可以有多个节点能跳出循环（i.e. 后继节点不都属于S）

**Predecessor**: pred, 前驱节点

**Successor**: succ, 后继节点

**Dominator**: 如果 CFG 的入口节点s0到节点n的所有路径都经过节点d, 我们就称d是n的支配节点(dominator)

·记作d dom n

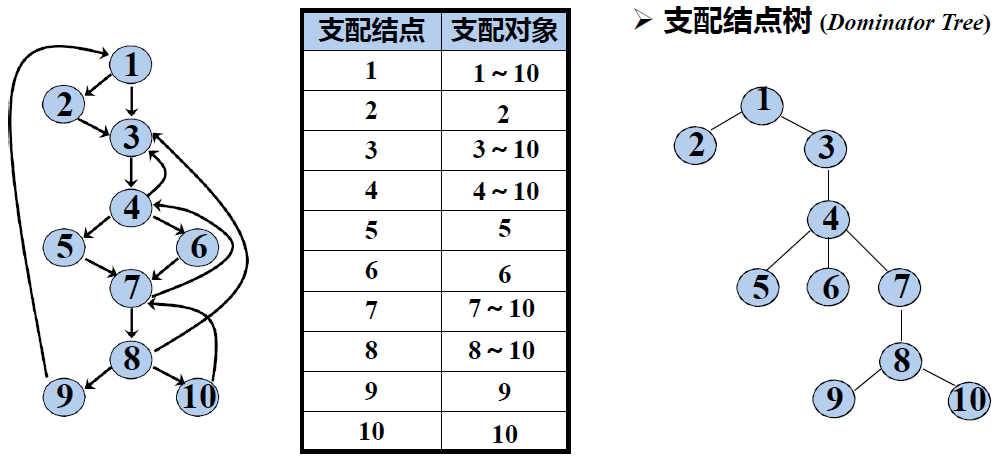
·可以理解为简单版的图论中的“割点”

·每个节点都支配(dominate)自己

·节点可以有多个 dominators

**Immediate dominator：**从entry到达n的任何路径(不含n) 中，它是路径中最后一个支配n的结点

**Dominator tree**：对于每个节点n, 其父亲为Idom(n)



**18.2 Natural loop** 自然循环

·Back edge: 如果边 n->h 满足 h dom n, 则这是一个 back edge

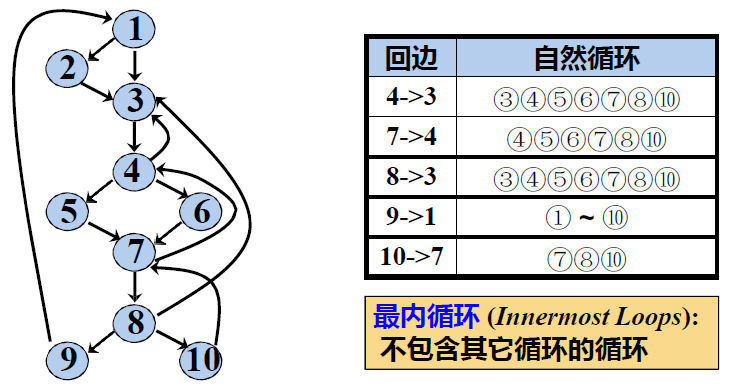
·Back edge 指向的节点 h 称为 loop header.

Back edge 是从循环尾回到循环头(loop header)的边；循环体是其中被 loop header 支配的节点。

·循环可以共享首节点

·除了共享首节点的情况外，两个循环要么完全不相交，要么一个完全嵌入另一个（或者说后者包含前者）。

·**最内循环**(innermost loop): 不包含其他循环（不被其他循环嵌入）



**Loop-nest tree**: 表达循环的嵌套关系

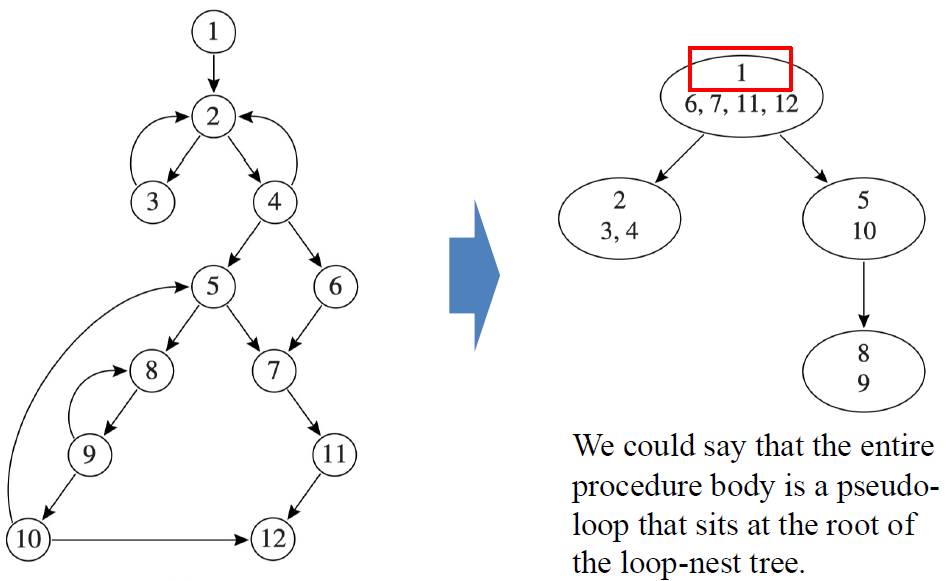
·树中每个节点对应一个loop header及其对应的natural loops节点集合

·如果这个loop heder对应多个natural loops (共享首节点)，也合并到同一个节点。

·节点的上半部分表示该节点对应的 loop header.

·可以把整个 procedure 视为在一个假想的大循环中，作为树的根节点。

·叶节点即对应最内循环



**Loop preheader**: 前置首节点

许多优化操作会在进入循环前进行一些准备工作，也就是在紧挨着循环头之前插入一些语句。因此我们可以在 loop header 之前插入一个 loop preheader 用来安置这些语句。

loop preheader 的唯一后继就是 loop header

·循环L里到达首节点的边x->h改为进入前置首节点：x->p

·循环L外到达首节点的边不变

**18.3循环不变代码外提Loop Invariant Hoisting**

**Loop-invariant:** 如果某个表达式的值在循环中不会改变，对循环来讲是固定值，则称表达式 loop-invariant.

我们可以归纳地检查赋值语句 x := v1 OP v2 的操作数是否 invariant:

**Base cases:**

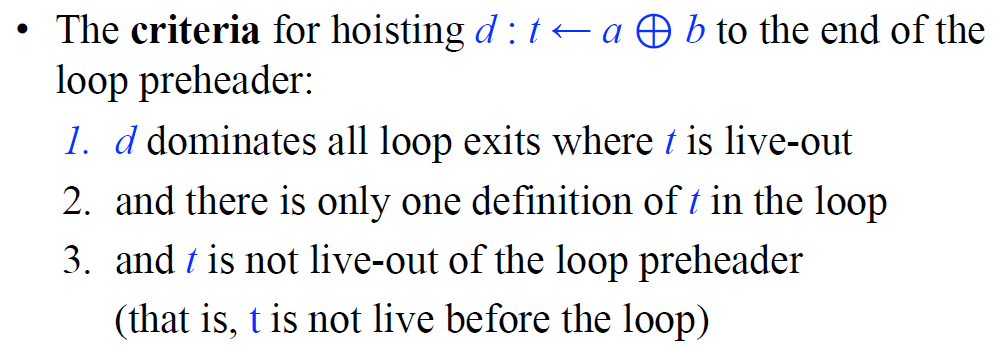
·常数：一定是 invariant 的。

·变量的 use: 该变量所有 defs 都在循环外则 invariant.

**Inductive cases:**

·表达式：多个 invariant 表达式进行运算仍然 invariant.

·变量的 use: 要求在执行该语句时只可能有唯一的 def 有效，且这个 def 的右侧(RHS) 是 loop-invariant 的。



**反例：**

