**CH1 介绍**

**module**（指每一个功能模块）**interface**（指前后模块中规定的输入输出）

**front end**：analysis, 将代码文本转化为abstract syntax，这部分完成lexing和parsing。它需要对语言进行理解，检查语法正确性.生成中间代码

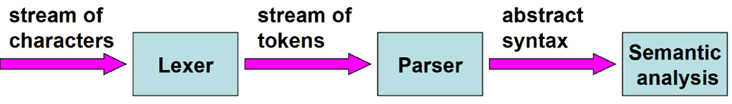
**middle end**：将抽象表示变为intermediate form，这部分完成type checking、analysis、optimizaition。在这个阶段，编译器会尝试使用各种技术来改善程序效率和可读性，比如代码优化、循环展开、函数内联等

**back end**：synthesis, 负责将中间形式转换成目标机器代码。这个过程需要进行寄存器分配、代码生成、对象文件链接等操作，最终生成可执行文件

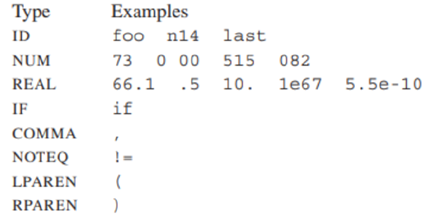
**CH2 词法分析 Lexical Analysis**

**2.1 扫描处理**

词法分析:将输入字符串识别为有意义的子串

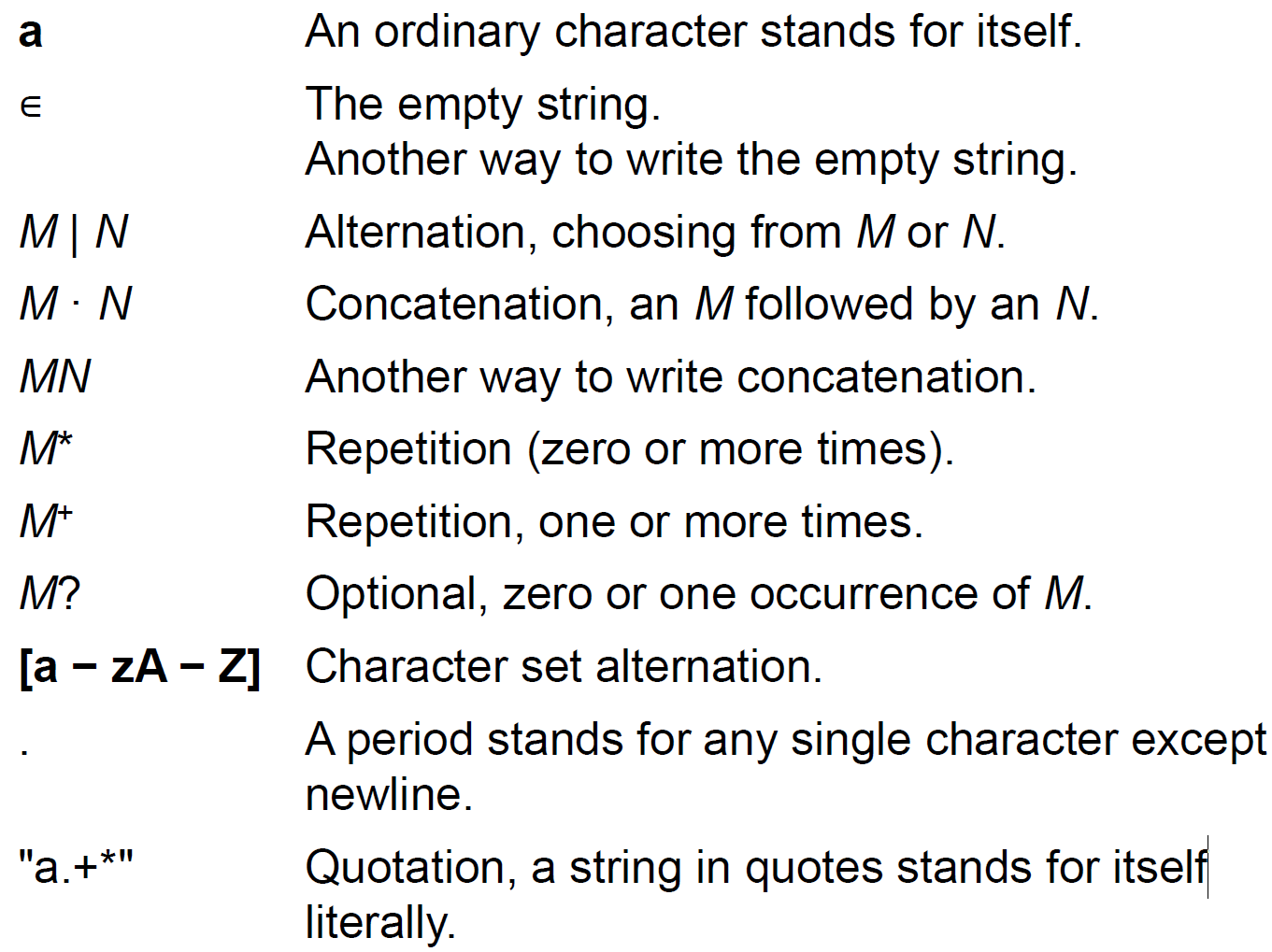
****

**Token**: 是指程序设计语言中具有独立含义的最小词法单位，包含单词（ID）、标点符号（，！；）、操作符（+，+=，?:）、分隔符（{}，[]）等

****

**2.2正则表达式**

1.一些notation



**缩写表达规则**：我们有时省略·和ϵ，并规定\*优先级高于·高于|。还有一些缩写形式：

[abcd] 表示 (a|b|c|d)

[b-gM-Qkr] 表示 [bcdefgMNOPQkr]

M? 表示 (M|ϵ) ，即 M 出现 0 次或 1 次

M+ 表示 (M·M\*) ，即 M 出现至少 1 次

. 表示除换行符外的所有单个字符

"a.+\*" 引号中的字符串匹配其自身

RE匹配**优先匹配保留字**；**最长字串优先**

🍔 Strings over the alphabet {a, b, c } with an even number of a's.

((b|c) \* a (b|c)\* a (b|c) \* ) \* | (b|c)\*也就是，对于aa的情况列举所有可能，然后再加上没有aa的情况

🍔 Strings over the alphabet {a, b, c} that don't contain the contiguous sub-string baa.

a\*(b|ba|ca\*)\* or a\*((b|c)\*(b|c)+a)\*(c+aa+)\*)\*

正则表达式：不具备记忆、计数比较、递归的能力

**2.3自动机**

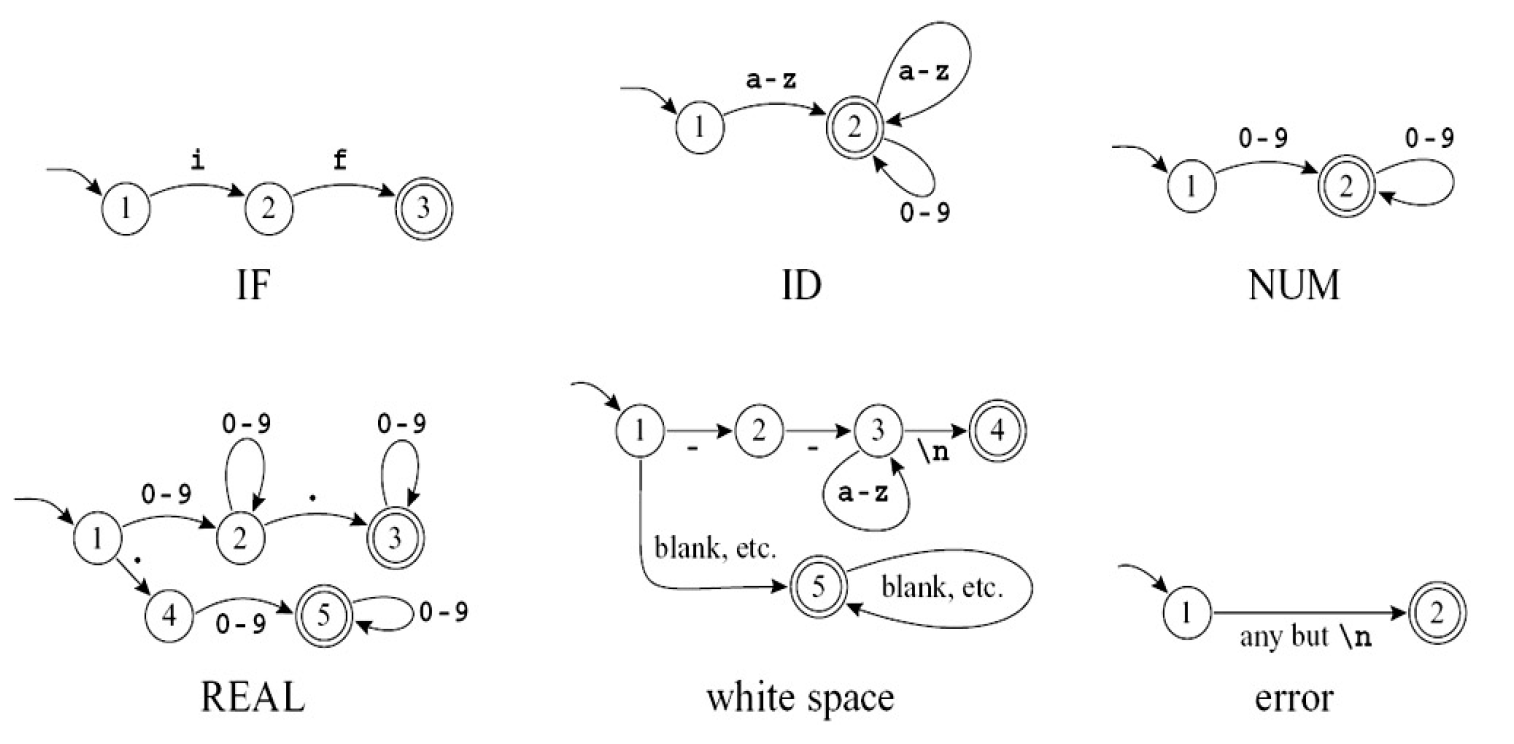
正则表达式⇔ DFA ⇔ NFA

**DFA**:M由字母表、状态集S、转换函数T:S→S、初始状态S0∈S以及接受状态AS.

错误状态默认不画,但是存在；错误状态下的任何转移均回到自身,永远无法进入接受.

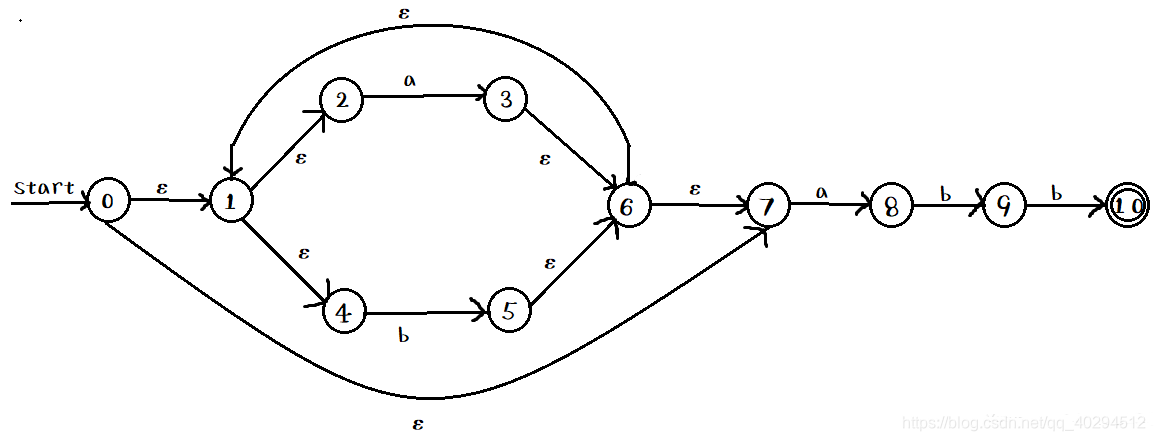
**NFA**:M由字母表、状态集S、转换函数T:S→(S)、初始状态S0及接受状态A的集合.

一些给出的基本FA图:

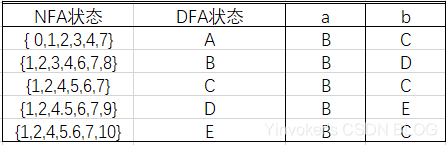


**2.4正则表达式到DFA**

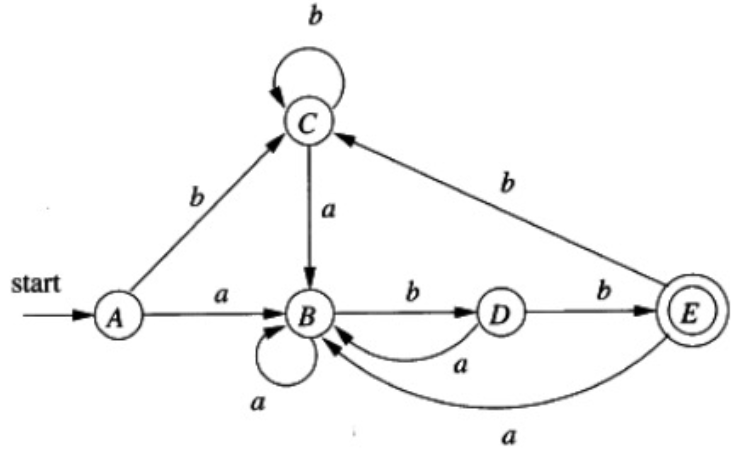
**DFA→NFA:**列出所有状态的闭包，如下DFA



从A填到E就可以得到下表：



将状态表转换为状态图（这里需要注意，A为第一个起始状态，所以A为起始态。由于E包含接收态10，故E为接收态）：



**DFA状态数最小化**:

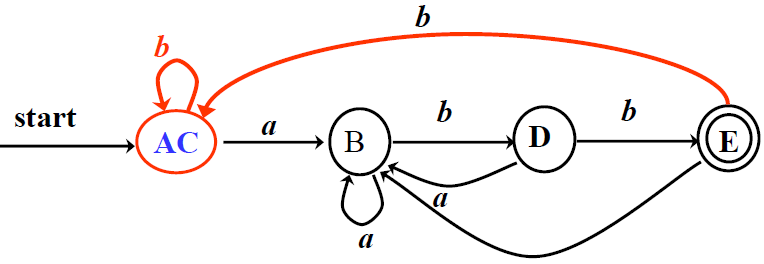
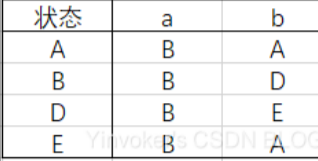
分组，得到 {A,B,C,D} {E}

{E} 独自分组，无法操作。 {A,B,C,D}做a操作，发现都转为状态B，做b操作，A，B，C在 {A,B,C,D} 组内成员，而D在另一个组内，重新分组得到 {A,B,C} {D} {E}

{D} {E}无法操作， {A,B,C} 做a操作，都在同一组内，做b操作，B不在同一组内，重新划分，重新分组得到 {A,C} {B} {D} {E}

{A,C} 进行a操作和b操作都在同一组内，无法继续向下划分了。操作到此结束。可以将 {A,C}合并

得到以下转换表和图



lex的源程序：{定义}%%{规则}%%{用户程序}

定义段：常量：表示常数的标识符；正则定义

规则段：{**正则表达式**（模式）} {**C语言**（动作）}

用户程序段：动作中要用到的函数

**CH3 上下文无关文法分析 Parsing**

从词法分析器获得Token序列，确认该序

列是否可以由语言的文法生成

– 对于语法错误的程序，报告错误信息

– 对于语法正确的程序，生成抽象语法树

**3.1 CFG**

**G = (T , N , P , S )**

T：终结符集合(Terminals)

N：非终结符集合(Non-terminals)

P：产生式集合(Productions)

S：开始符号(Start symbol)

**最左(右)推导**: 每步代换最左边的非终结符，反向是最右(左)规约，如果不ambiguous则只有一个。每个左推导都有对应的唯一的右推导

**3.2 二义性**

**1.二义性**(ambiguous)**:**同一个串存在多个推导即多个分析树

2.**分析树**(sytax tree)**根节点**是文法的初始符合,**内部节点**是非终结符,**树叶**是终结符；每一个父节点和他的子节点构成一条产生式。最左推导和前序编号对应,最右推导后序。语法树中,越接近根,越高,优先级越低；左递归导致左结合,右递归会右结合

为了高效语法分析，可能对文法做处理/限制：

1.消除二义性

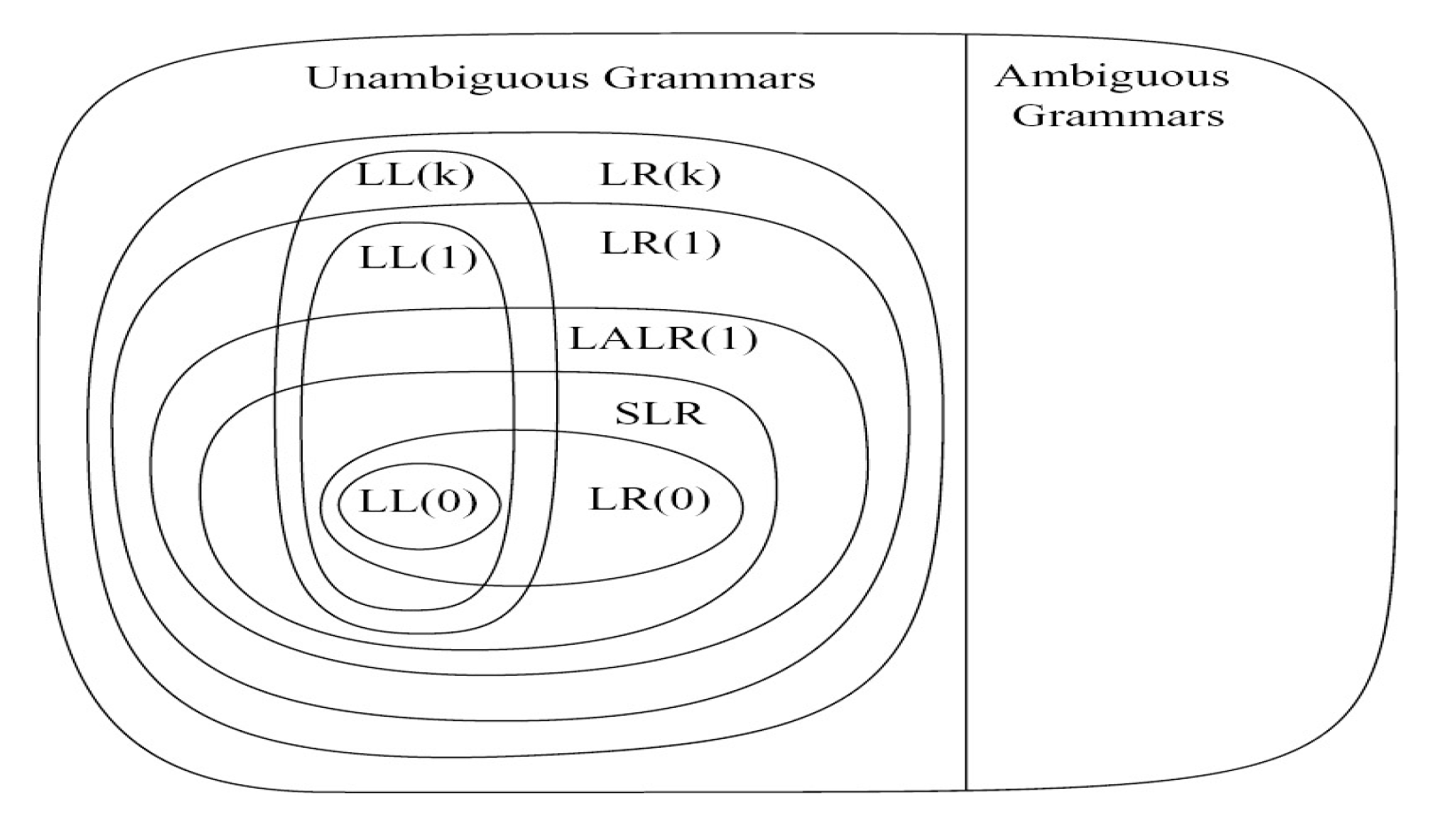
2.消除左递归(常使用于自顶向下)

3.提左公因子(常使用于自顶向下)

消除二义性的方法有

– 规定符号的优先级

– 规定符号的结合性



**自顶向下分析: LL(k)**

自顶向下(Top-down)分析

从开始符号S出发，尝试根据产生式规则推导(derive)出x；Parse Tree的构造方法：从根部开始。自顶向下分析总是选择每个句型的**最左非终**结符进行替换

第一个L是从左到右处理,第二个L是**最左推导**,1代表仅使用1个符号预测分析方向

LL(1)的前提：对任意规则,① First()∩First(β为空, 即α和β推导不出以同一个单词为首的串 ②若，那么不能有 , 且需要First()∩First(为空，即α和β不能同时推出e; First(α)不应在Follow(A) 中

LL(1)不会有二义，但是无歧义文法不一定是LL(1), 是无左递归的, 是无左公因子的

LL(1)面对重复和选择的解决方法:**消除左递归**left recursion removal(比如:不是ε,且均不以A开头).)和**提取左因子**left factoring.

**消左递归**:

**提取左因子**:

**Fisrt(X)**:可从X推导得到的串的首个终结符的集合。

若X为T或,Fisrt(X)={X}②若X为N,对于每个产生式X→X1 X2 …Xn,First(X)都包含了First(X1)-.若对于某个i<n,所有的Fisrt(X1),…,First(Xi)都含有,则First(X)也包括了First(Xi+1)-.若所有Fisrt(X1),…,First(Xn)都含有,则First(X)也包含.

A non-terminal A is **nullable** if and only if First(A) contains

**Follow(A)**: 从S出发,可能在推导过程中跟在A右边的终结符号集

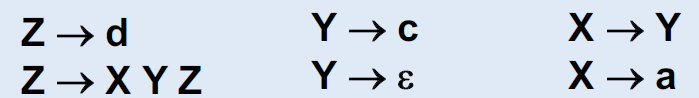
若A是一个N,那么Follow(A)由T和$组成.①若A是$,直接进入Follow(A) ②若存在产生式B→A,则First()-在Follow(A)中 ③若存在产生式B→A,且在First()中,则Follow(A)包括Follow(B)④更常见的情况是B→A,那么Follow(A)包括Follow(B)

First关注点在产生式左边,Follow在右边

**LL(1)分析表M[N,T]的构造算法**:为每个非终结符A和产生式A→重复以下:首先把能直接首T推出的nullable置为True, 其余为False.

**项M(A,a)** 表示: 针对非终结符为A，当下一个输入Token为a时，可选的产生式集合

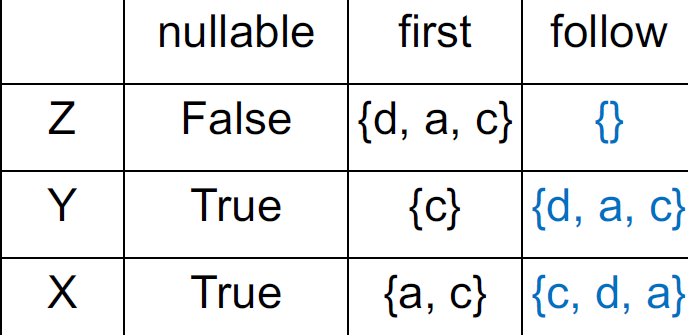
①对于First()中的每个记号a,都将A→添加到项目M[A,a]②若可空(nullable),则Follow(A)中的每个元素a (包括$),都将A→添加到M[A,a]



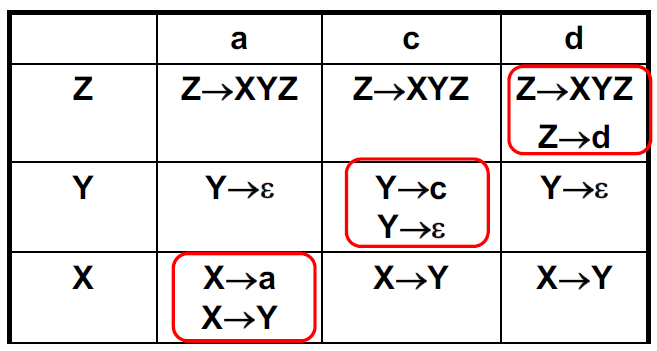
1) 检查有无左递归左因子

使得右边不存在以相同符号开头的式子，如 E -> T+E | T 中，有两个T, 需要引入一个新的符号，比如E -> TX , X -> +E | ϵ

2)nullable|First|Follow



3) 按照算法填写规则进表



4). LL(1)判别:2 grammar rules in the same box 所以上例不是LL(1)

5). 具体用法直接消除即可

**自底向上分析**

尝试根据产生式规则归约(reduce)到文法的开始符号S。Parse Tree的构造方法: 从叶子开始

通过shift和reduce逐个把terminal转到non-termina。自底向上分析更加高效，能够处理更复杂的语言结构。自底向上分析是最右分析的一个逆过程。可以用**栈**完成

①**shift,**将 | 右移一位，即将下一个 terminal 纳入考虑②**reduce**使用产生式A→将栈顶的规约成A③**accept**分析栈为开始符号,输入栈为空时的动作④**error** ⑤转移(**goto)**: 吃non-terminal

**LR(0)<SLR(1)<LALR(1)<LR(1)**，如果前边一个文法判断是后边都是

**LR(0)**

L从左向右扫描，R最右推导的逆

LR(0)的状态中包含带**’.’**的规则,’.’左边表示已读,右边表示未读；如果圆点’.’右边的第一个非终结符有规则,那么也写到这一状态中.

LR(0)文法不可能二义的

**rj** j是新状态，把j压入栈

**si**用式子i归约A→β：把β归约为A (Pop栈顶状态,并根据GOTO表压入新状态)

**accept**接受输入，完成分析

**gi,** 进入状态i(第i行)

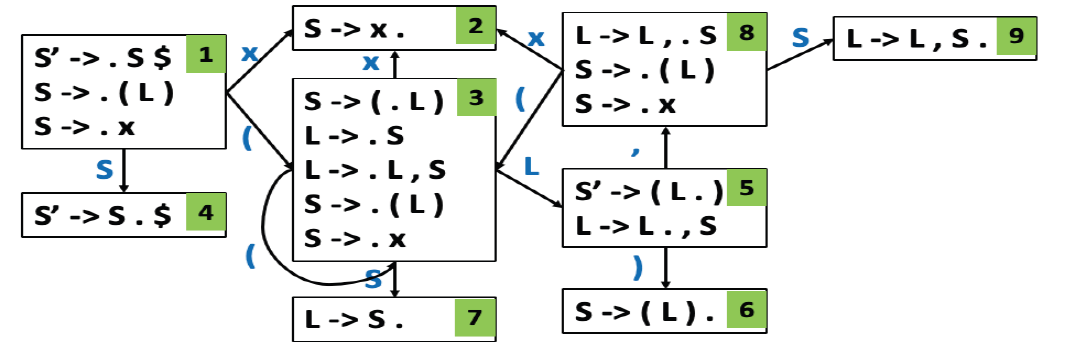
ACTION是遇到终结符，GOTO是遇到非终结符

**1) 新的开始符号**

0: S’ -> S$ 1: S -> (L) 2: S -> x

3: L -> S 4: L -> L, S

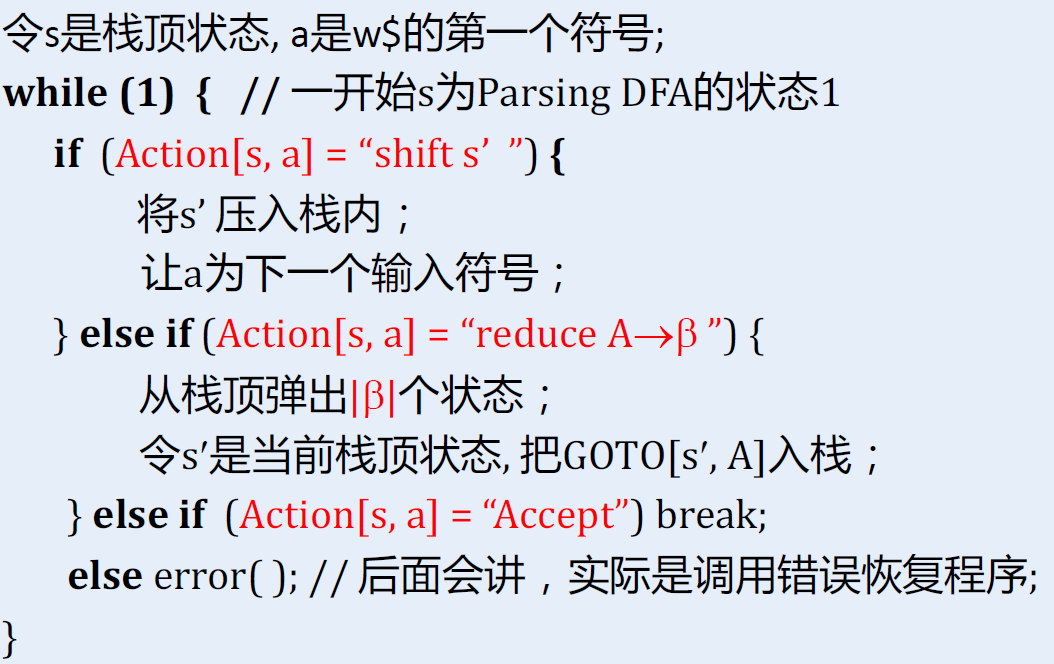
**2) 构造转移DFA**对应上面的5个状态，构造一个对应的DFA。主要是分清楚·符号在哪个位置：（1）**在非终结符的前面**，状态集要增加，如：下面的表中的I2状态，由于·符号后面是非终结符A，就要增加以A开始的状态集。（2）**在终结符的前面**，状态集不变，如I3状态，·符号后面是终结符c，I3状态不变（3）**在末尾，是规约状态**。如I8

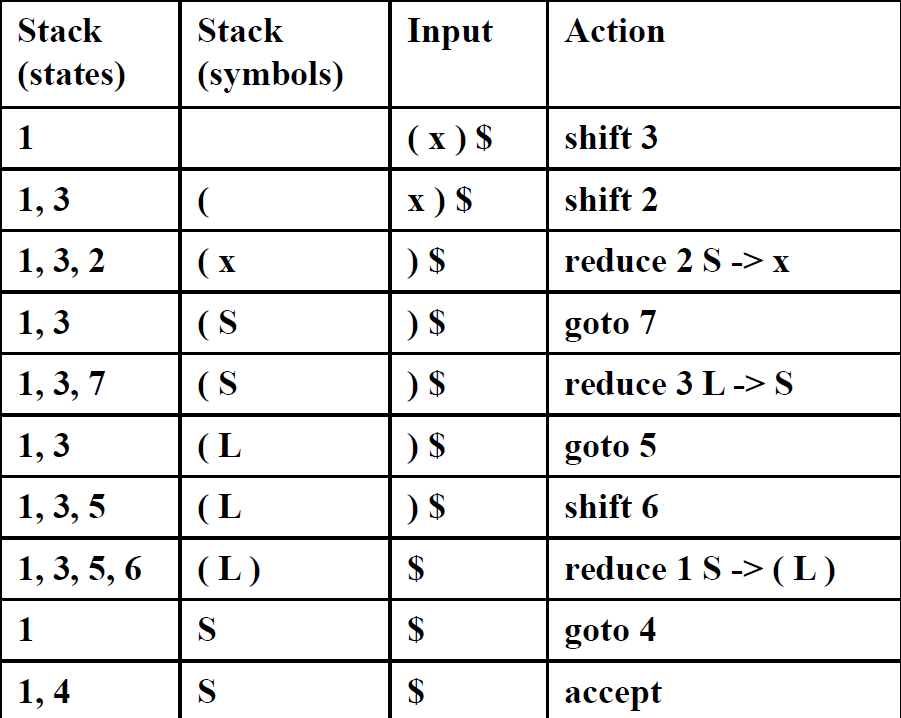


**3) 根据DFA构表并转移**



**4) 应用**





**4) 判断**

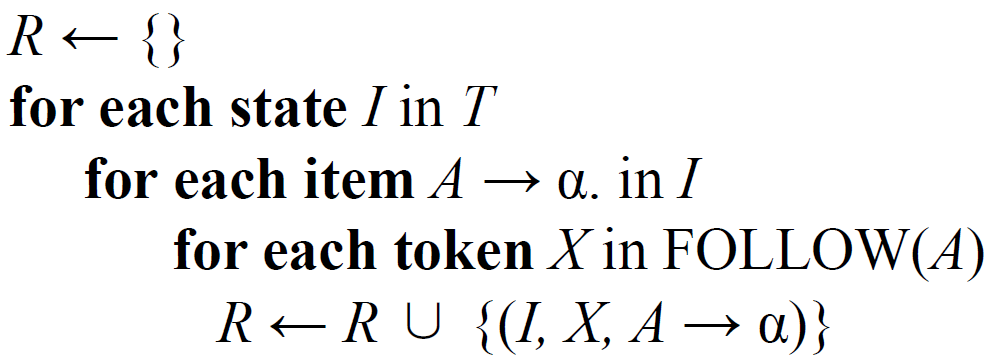
**s-r conflict**:同一个格子里面中同时出现了s和r,即构成s-r冲突。Yacc解决方案：移进动作优先于归约动作

**r-r conflict**:表中同时出现两个及以上的rn,则构成r-r, 有多种可以做归约的方案，不知道归约到哪一个, 比如

**SLR:**

区别在于，对于r，这里不是填满一整行，而是只对转移表达式follow(左边)进行填写

按照进行归约的条件是： 下一个输入符号x可以在某个句型中跟在A之后，也就是x ∈ Follow(A)



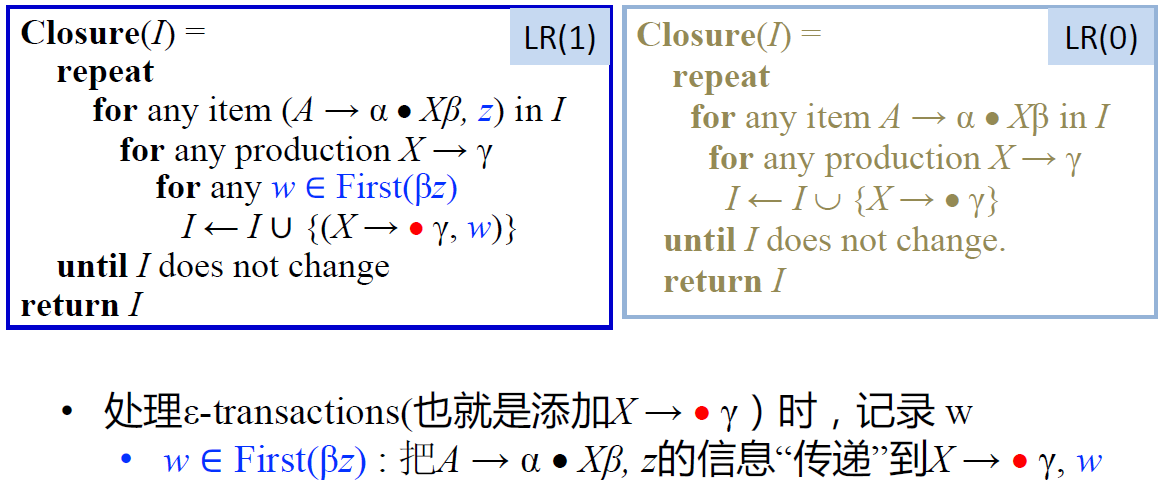
SLR不可能是二义性

自底向上右递归可能引起栈溢出,需要避免

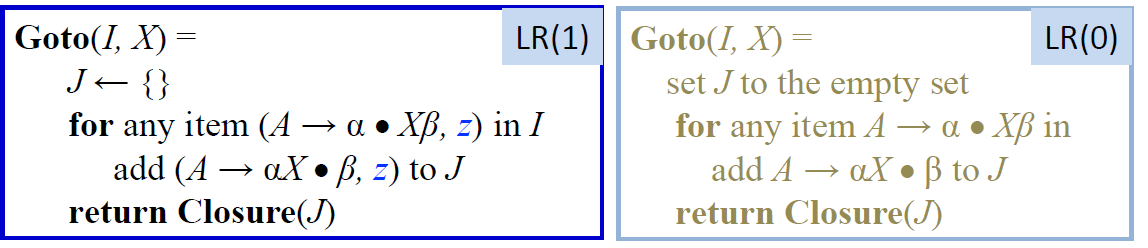
SLR中的两种冲突,sr冲突使用消岐规则:优先移进；rr冲突基本是设计出问题

**LR(1):**

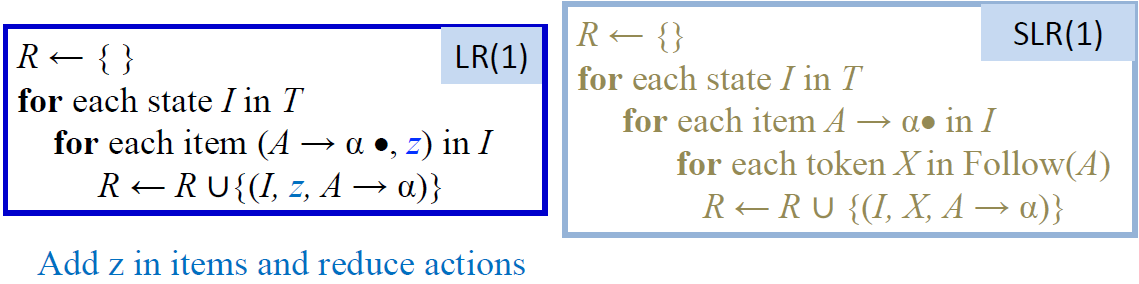
比SLR(1)多一个展望符（未来的输入将会是），后面跟着的是follow set



goto基本与LR(0)相同



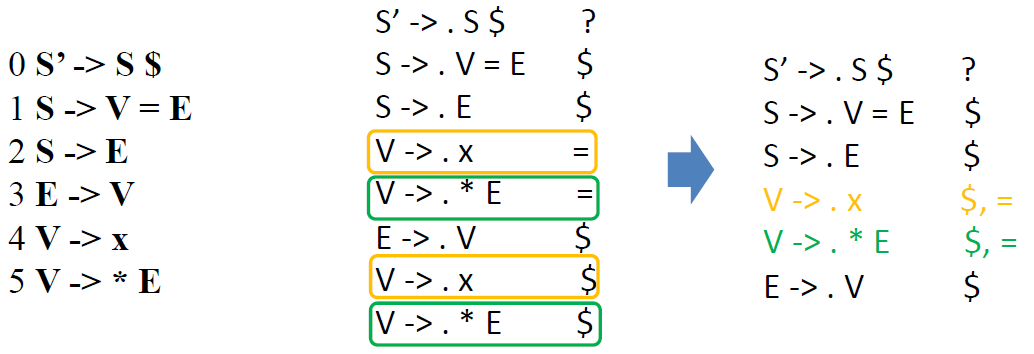
Reduce: 如果要根据归约，下一个输入符号必须是展望

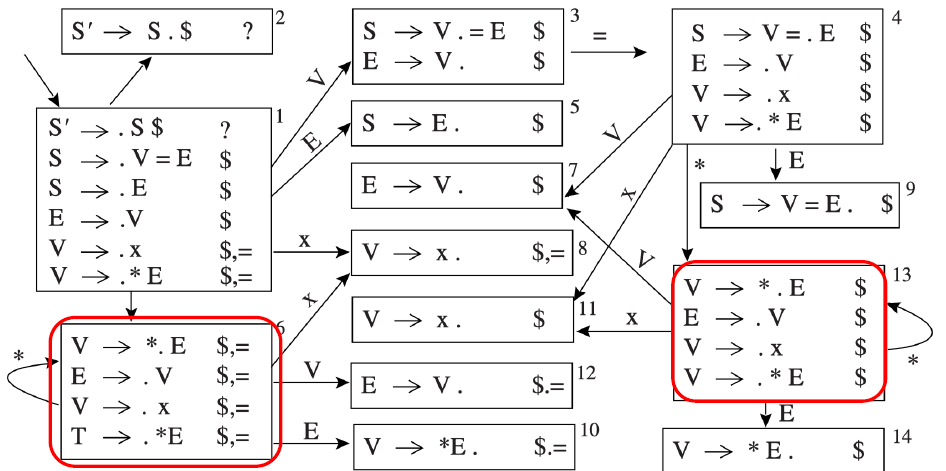


LR(1)文法不可能二义性

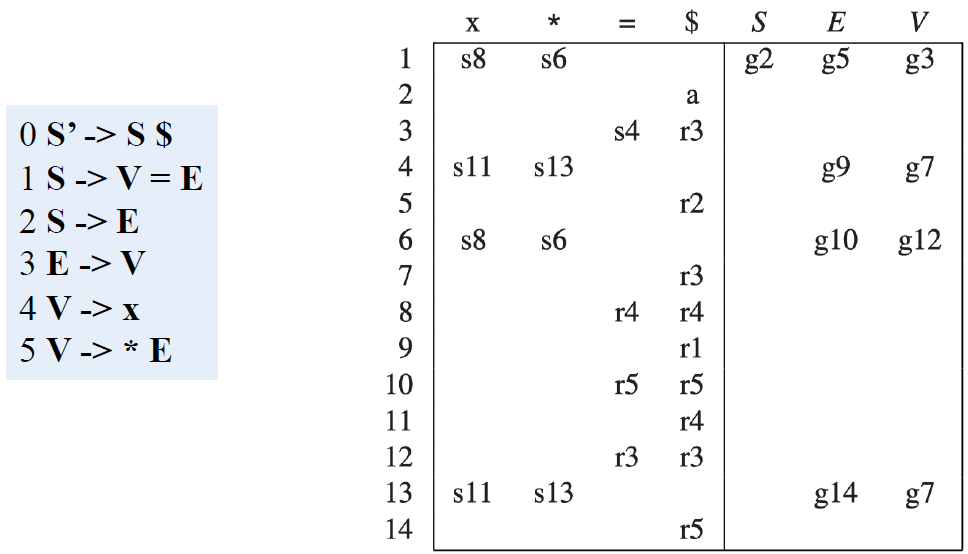
**1) 新的开始符号**

**2) 构造转移DFA**



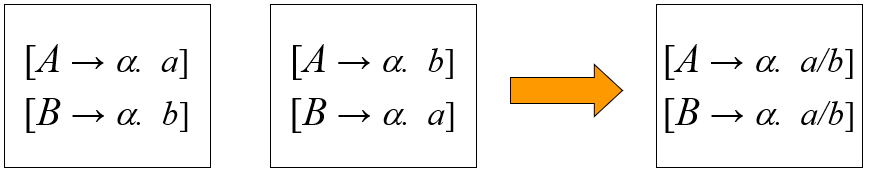


**3) 根据DFA构表并转移**



**LALR(1):**

将look ahead symbol进行合并:除了look ahead symbol不同以外全都相同的规则合并成一个,目的在于减少表项. 这有时可能引入 reduce-reduce conflict



如果在 LALR(1) 解析算法中没有出现解析冲突，则语法为 LALR(1) 语法

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.

**Yacc基于LALR(1).**

LR(1):把期望的向前看符号也加入项中成为LR(1)项，向前看符号(串)的长度即为LR(k)中的k，充分利用向前看符号，但是状态很多

LALR(1)：介于SLR(1)和LR(1)之间，且分析表和SLR一样大;LALR已经可以处理大部分的程序设计语言

🍔为什么说LR(0)没有Lookahead?

LR(0) Item中没有lookahead terminal等信息

“是否归约、选择哪个产生式规约”仅由栈顶状态决定

🍔 A grammar is ambiguous if it has only one parse trees for all sentences. **Answer: T**

**CH4 抽象语义 Abstract Syntax**

**属性文法**: 上下文无关文法+属性+属性计算规则

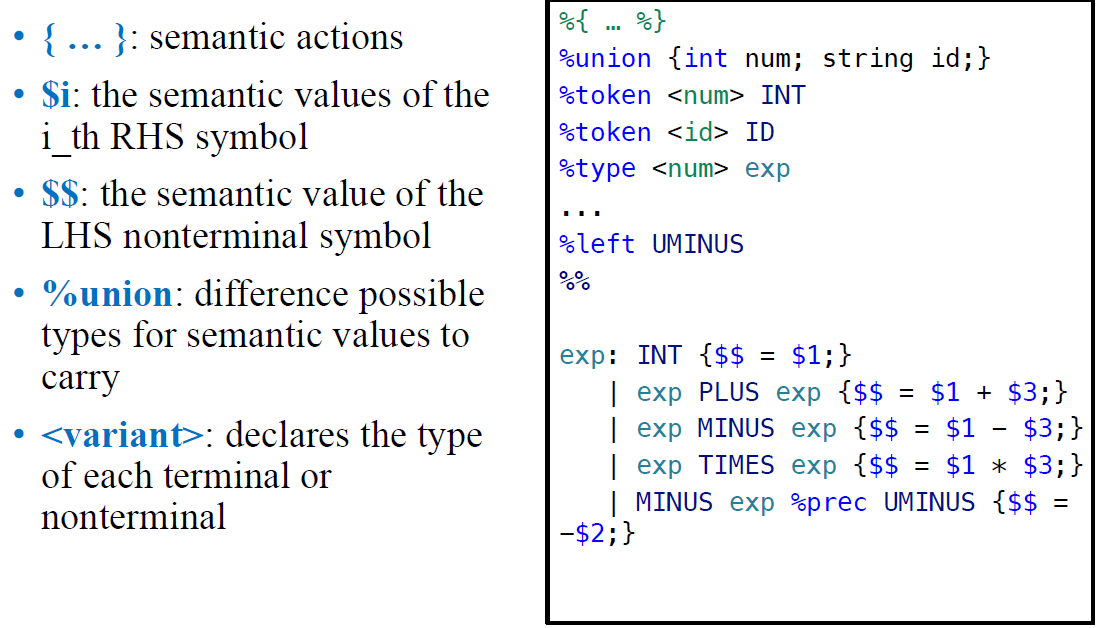
– 属性: 描述文法符号的语义特征，如变量的类型、值等。例: 非终结符E的属性E.val（表达式的值）

– 属性计算规则(语义规则): 与产生式相关联、反映文法符号属性之间关系的规则, 比如”如何计算E.val”

可通过Parser生成器支持的**Semantic action**实现计算并应用于抽象语法树生成等场景

**Recursive Descent Parsing**中**语义动作**是解析函数返回的值，或这些函数的副作用，或两者兼而有之。对于每个终结符和非终结符，我们关联一种表示从该符号派生的短语的语义值。

**Yacc-Generated Parsers**中



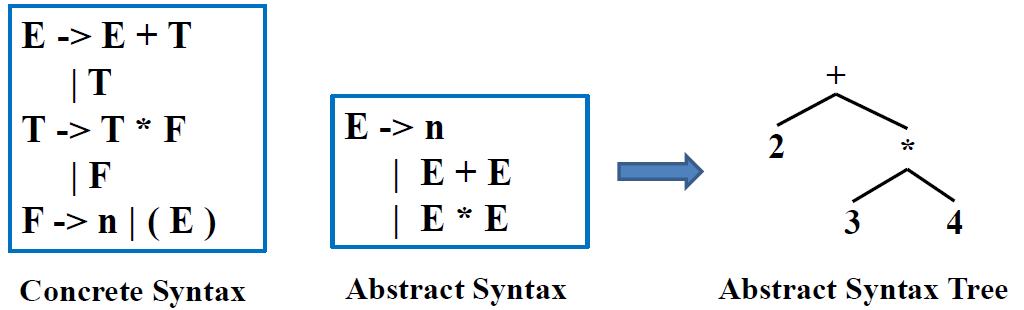
Yacc生成的解析器会保留一个与状态堆栈并行的语义值堆栈。当解析器执行归约时，它必须执行相应的 C 语言语义操作

$i for a rule A -> Y1…Yk的值就存在栈顶上k个

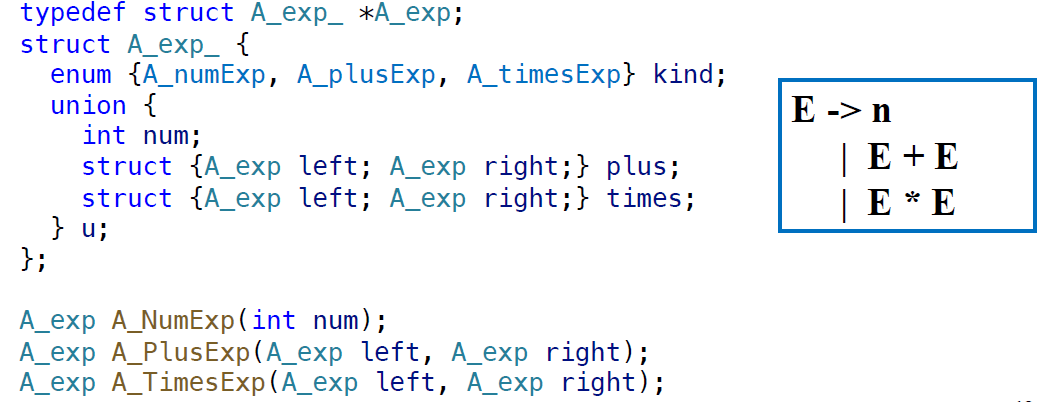
**parse tree**：对于输入中的每个token，都有一个叶子节点，并且对于解析过程中规约的每个文法规则，都有一个内部节点。

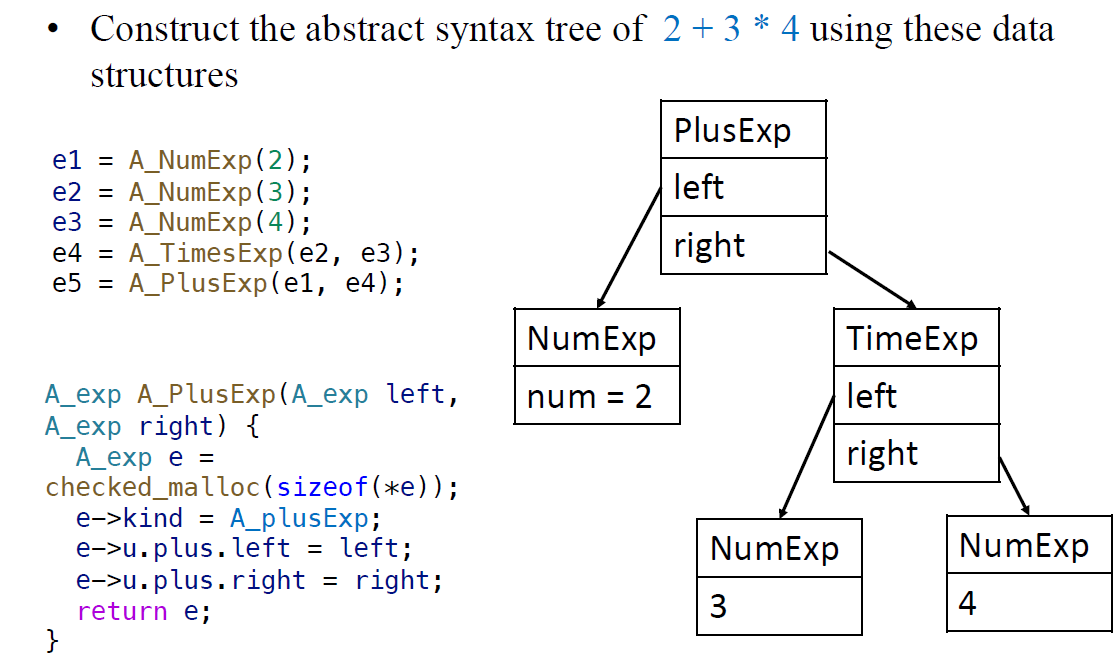
**concrete parse tree**：代表了源语言的具体语法结构。使用起来不方便，有多余的token和依赖过多的语法

**abstract parse tree**：传达源程序的短语结构，解决了所有语法分析问题，但不包含任何语义解释和细节(不包含规则节点和括号)。对于同一种语言，不同的语法规则会产生不同的解析树，但最终都应该生成相同的AST。AST并不关注它们是如何由语法规则生成的。AST使用运算符或操作作为根节点和内部节点，并将操作数作为这些节点的子节点。与解析树不同，抽象语法树不使用内部节点来表示语法规则。



要在后续阶段使用抽象语法树，编译器需要将抽象语法树表示为数据结构并加以操作。具体来说，为每个非终结符指定一个typedef，为每个产生式指定一个 union-variant





**定位**：在one-pass编译器中，词法分析器会保留一个“当前位置”全局变量。

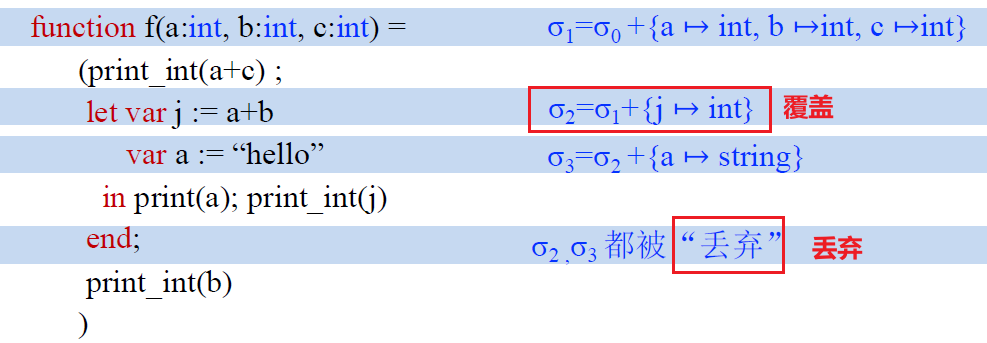
对于使用抽象语法树数据结构的编译器：不需要在一次过程中完成所有解析和语义分析;词法分析器在语义分析开始之前就到达文件末尾。抽象语法数据结构必须包含 pos，这些字段指示这些抽象语法结构所衍生的字符在原始源文件中的位置。Yacc的解决方案是定义一个非终结符号 pos，其语义值是源位置（行号，或行号和行内位置）

**CH5 语义分析 Semantic Analysis**

1.connect AST 的一些静态属性（比如类型，变量访问范围） 2.把AST翻译为静态环境

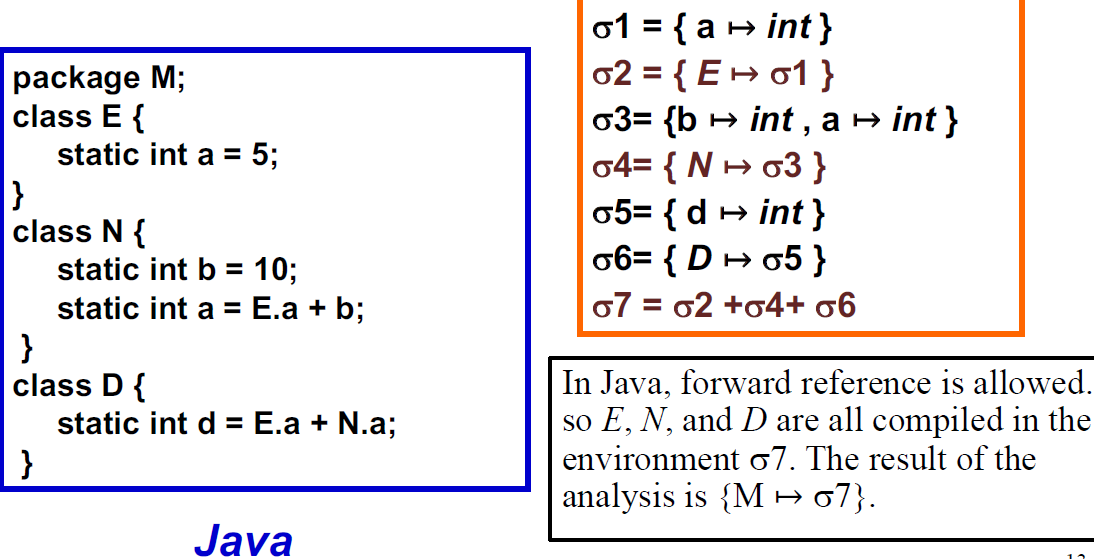
**5.1 符号表**

**Binding**把symbol和对应的meaning/attribute关联； **Environment**一组绑定，比如{g ↦ string, a ↦ int}；**符号表**, environment的实现,将标识符映射到类型和储存位置.（一般hash table



变量仅在自己的作用域中可见.当语义分析到达每一个作用域的结尾时,所有属于该作用域的变量都被符号表抛弃不用.在C/C++以及Java中,变量的作用域不可以交叉(scopes of vars cannot be intercrossed)

**Multiple Symbol Tables.** 在某些语言中，可以同时存在多个活动环境：程序中的每个模块、类或记录都有自己的符号表。



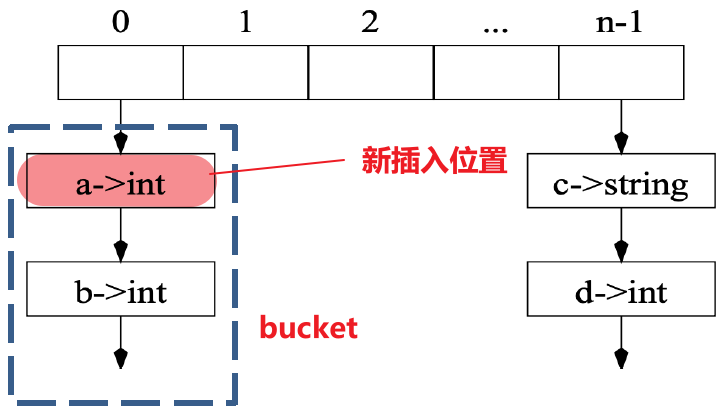
**5.2 命令式符号表(imperative)**

散列表(bucket list) 实现, 有了新的就看不到老的，但是退出scope时还能回得去，动态变化

-修改直到它变成

-当存在时，我们无法在中查找内容

-当我们完成后，我们可以撤消修改以再次取回



**破坏式更新(side effect)**:将变量名作为键值直接插入到现有散列表,外加变量的类型; 插入时将变量插入到对应bucket的链表头部,便于实现作用域的删除操作以恢复环境

查找的时候，先从0开始找到对应的符号表中的位置，然后再找bucket

**5.3 函数式(functional)**

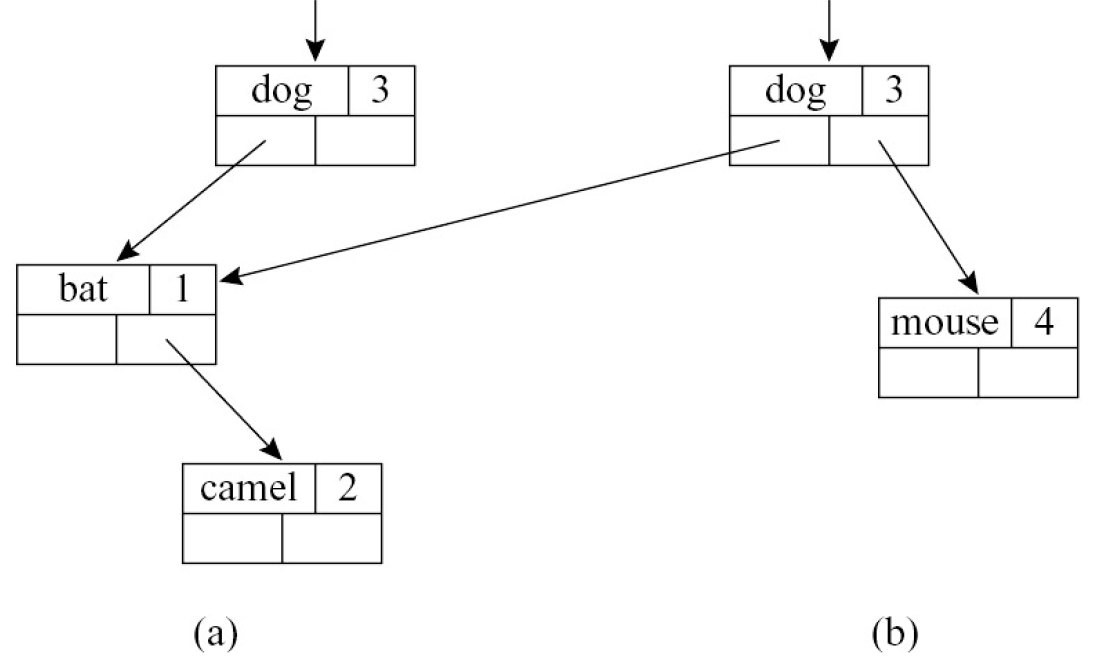
二叉搜索树(BST)实现，左小右大，每次发生变化的时候老的都保留着

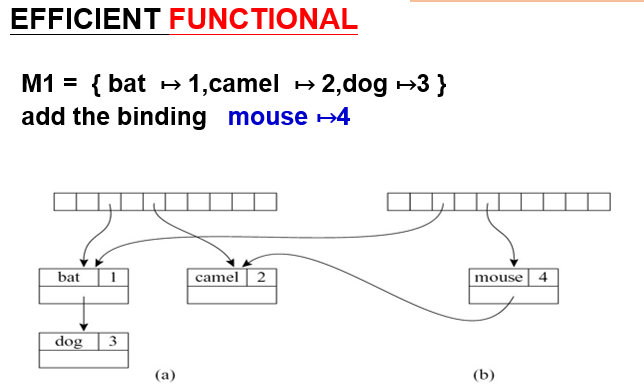
-创建和时保持于原始状态

-轻松恢复

创建新的BST节点的时候会先拷贝所有的祖先的属性然后修改(和查找元素效率相同logN),长效数据结构(persistent data structure);长效红黑树.

例子:已知,现在添加新绑定mouse→4.需要创建复制d个BST节点(直到被插入的深度d),剩余部分共用).然后插入新节点结果如下,两图等价:



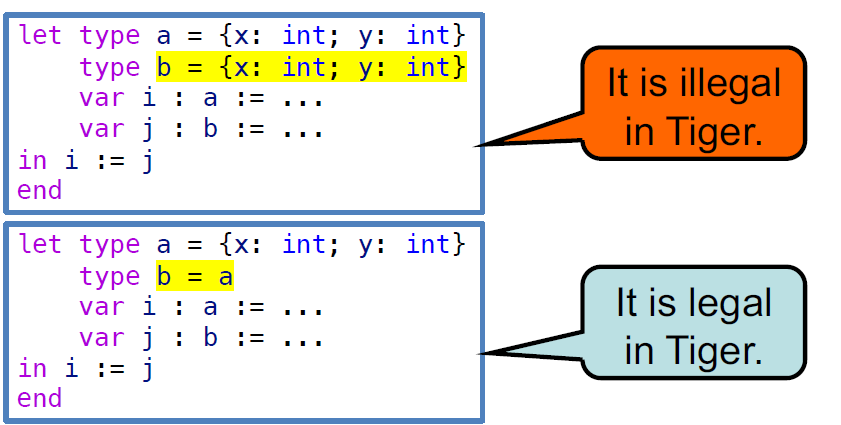


**5.4 Tiger特性**

**Structural equivalence**。T1 和 T2 等价当且仅当 T1 和T2 由以相同顺序应用的相同构造函数组成

**Name equivalence（Tiger使用这个）** T1和T2 等价当且仅当 T1 和 T2是由完全相同的类型声明定义的相同类型名称。例如，定义 typedef t1 = int, typedef t2 = int，但是 t1, t2, int 仍然两两不等价

**Declaration equivalence**弱化版的名等价，即别名之间可以等价



**CH6 活动记录 Activation Record**

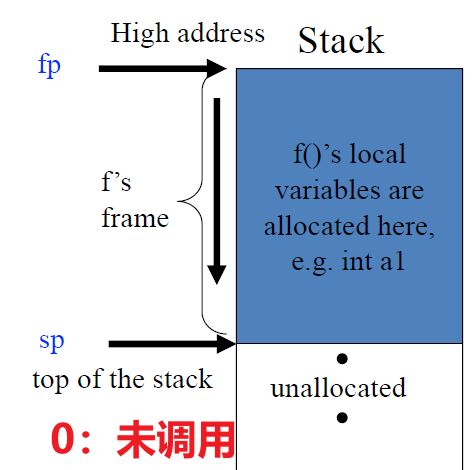
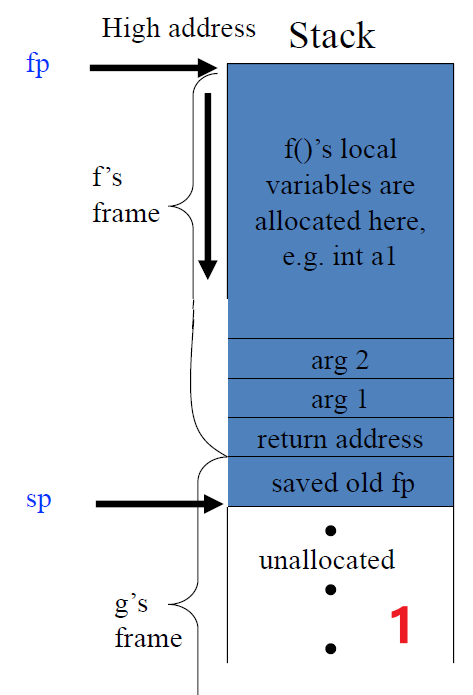
栈中用来存放函数的局部变量、参数、返回地址、其他临时变量的区域称之为**activation record**或**stack frame**存储的内容有：Relevant machine state (saved registers, return address), Space for return value, Space for local data, Pointer to activation for accessing non-local data

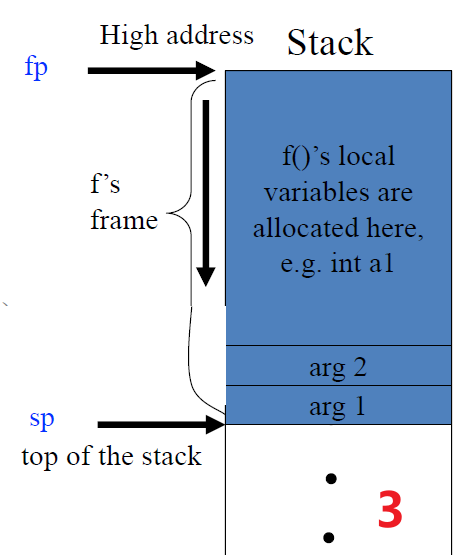
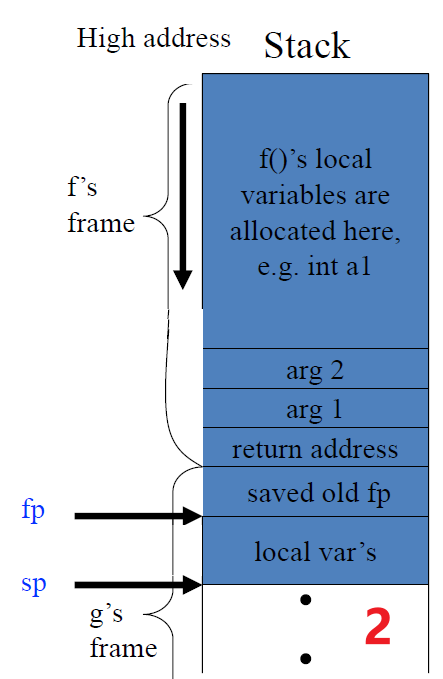
活动记录的具体组织和实现不唯一,即使是同一语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异

**frame pointer**(base pointer, 基址寄存器):指向当前帧的开始, 编译后的代码通过使用帧指针的偏移量来引用局部变量和参数

**Stack Pointer**(栈顶寄存器) :指向当前帧的末尾,指栈顶

**一个函数f调用g时**①**当f调用g时**：sp指向f传给g的第一个参数,然后g分配栈帧(sp-栈帧大小)②**进入g** fp先存一下，fp=sp，然后sp往下走③**g退出**: sp=fp,然后fp取回原先保存的值(并且pop掉返回地址等)



**栈帧内变量Frame-Resident Variables**: 一般来说(局部变量和中间结果,一些参数，返回地址，返回值)会放到**寄存器**中,以下情况需要将变量储存到**栈帧**内: ①变量传地址/引用(passed by reference) ②被嵌套在当前过程的函数调用(nested accessed) ③太大了放不下(too big to fit)④变量是数组⑤有特殊用途的变量(传参等) ⑥存在过多的临时变量和局部变量(溢出spill) ⑦取地址address is taken，比如&a

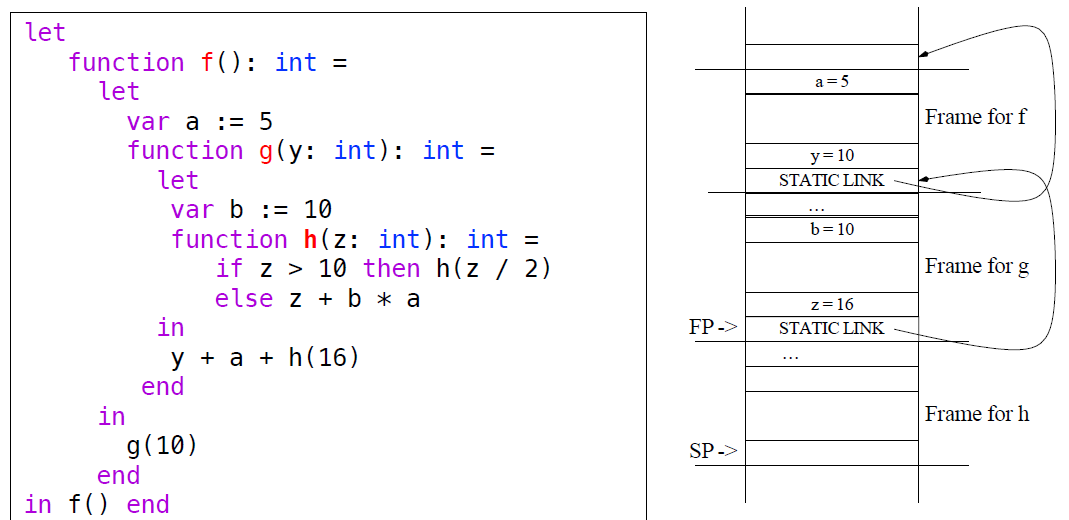
**参数传递**:现代计算机传参约定:前K个参数放在寄存器里传递,剩余在储存器传递.寄存器传参的方法(4种): ①不给叶子过程(leaf procedure)分配栈帧 ②过程间寄存器分配(interprocedural register allocation) ③直接重写寄存器不做保护 ④寄存器窗口技术(register windows)

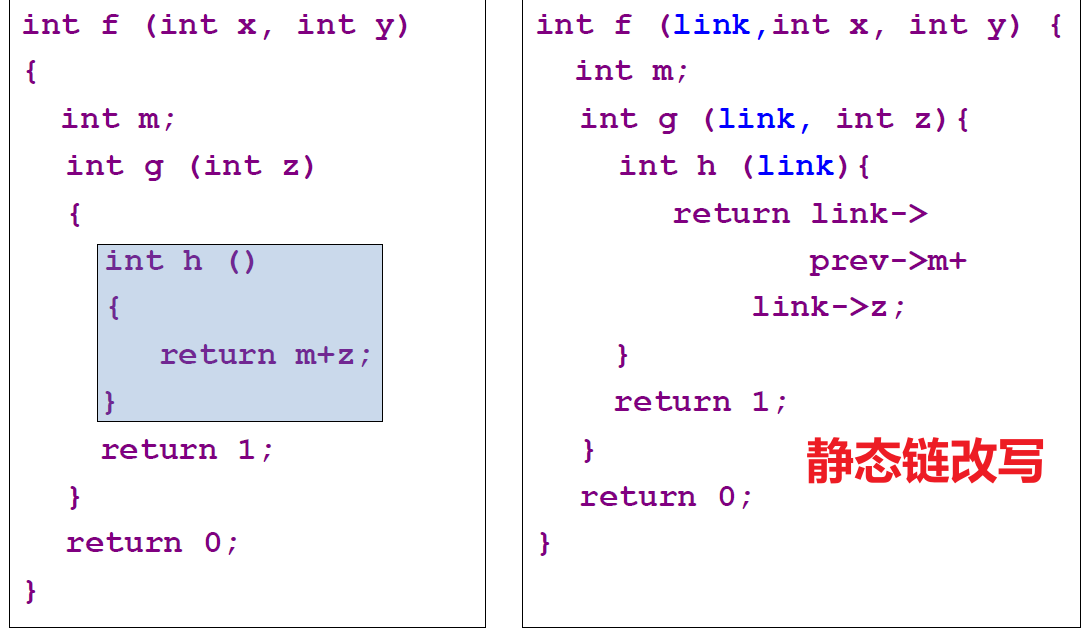
**返回地址**:call指令地址的下一条指令地址.

**逃逸变量(elapsed variable)**:①由引用传递(Pascal 支持，但 Tiger 不支持) ②被取地址 ③被内层嵌套函数访问的变量

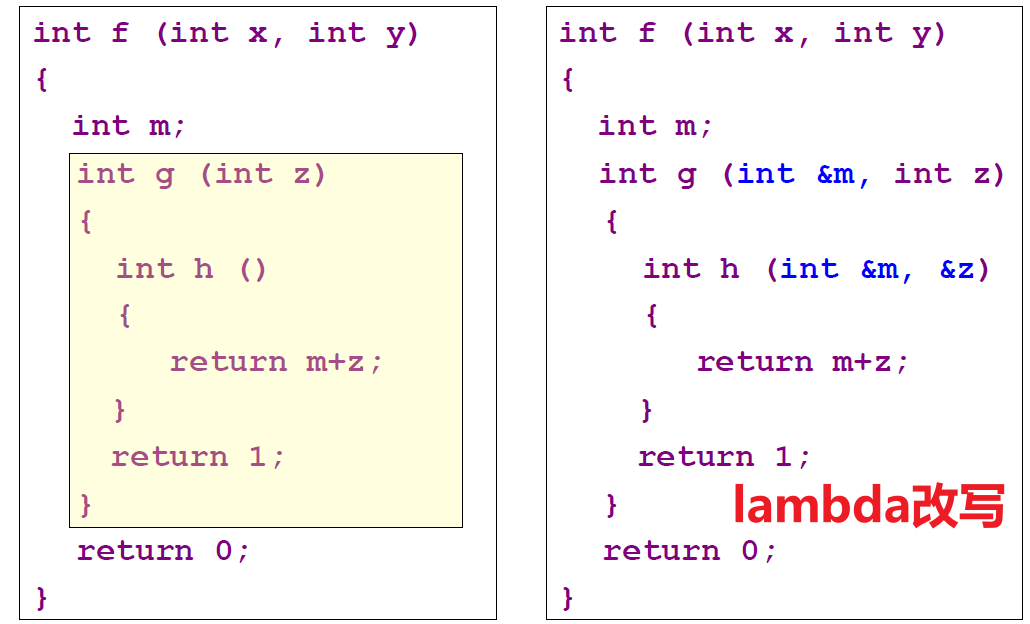
**静态链(static link)**:指向上一层嵌套层级的栈帧的指针.内层嵌套函数调用外层定义的变量需要用到静态链,否则无法寻址。The static link is a pointer to the activation record of the enclosing procedure

注意:静态链层级是函数的嵌套深度,不是递归调用的深度,两者不同概念.当深度为 n 的函数访问深度为 m 的变量时发出代码以爬**n-m**个链接来访问适当的激活记录

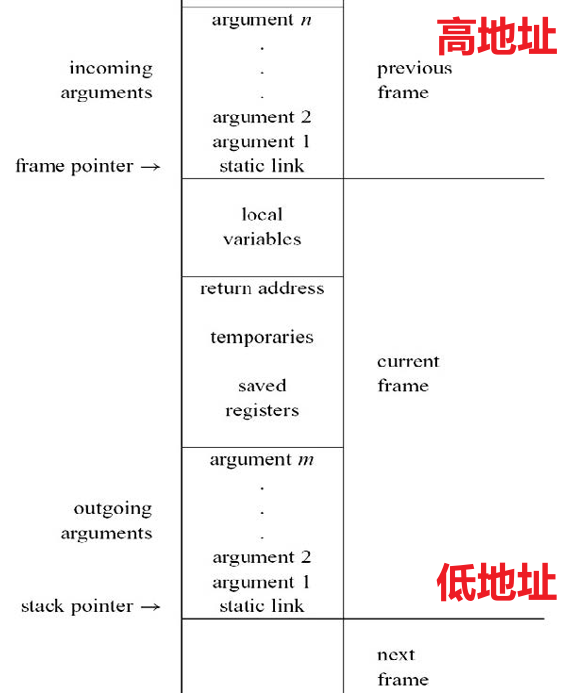




其他访问外层变量的方法:①**嵌套层次显示表(display)**: 一个全局数组,位置i存放最近一次的,静态嵌套深度为i的过程的栈帧.是管理静态链的全局数组(期中错过,管理的是静态链不是栈指针) ②**λ提升(lambda shifting):** 当g调用f时，f（或任何嵌套在f内的函数）实际访问的每个g变量都会作为额外参数传递给f（通过将非局部变量视为形式参数来重写程序；翻译/转换过程从最内层的程序开始，然后向外进行）



**Stack Frame从高地址往低地址增长** **layout**: ①传入参数(incoming arguments) ②返回地址(由call指令创造) ③局部变量 ④传出参数(outgoing) ⑤静态链(static link).



**CH7 中间代码(IR code)**

**IR:**抽象的机器语言，能够表达目标机器操作无需承诺太多特定于机器的细节。链接前端和后端,解决了高级语言和目标机器汇编语言之间的转化，适用于不同源语言和不同目标机器的可移植编译器(N\*M → N+M)

三地址码的一般形式：**x = y op z**



Tigher只转一次IR

中间语法树的表达式:

①**CONST(i)**:整型常数 ②**NAME(n)**:符号常数

③**TEMP(t)**:临时变量

④**BINOP(o, e1,e2)**:对操作数e1,e2的二元操作。整数运算操作有PLUS,MINUS, MUL, DIV; 按位逻辑操作有AND, OR, XOR; 逻辑移位有LSHIFT, RSHIFT; 整数算术右移是ARSHIFT

⑤**MEM(e)**:作为MOVE操作的左子式时表示对储存器e地址的存入;其他位置表示读取该地址的内容

⑥**CALL(f,l)**:过程调用

⑦**ESEQ(s,e)**: 先计算语句s形成副作用(s不返回值但是影响e的计算),然后计算e作为该表达式的值，比如ESEQ(a=5, a + 5)结果是10

⑧**MOVE(TEMP t, e)**: 计算e的值然后存到临时变量t中

⑨**MOVE(MEM(e1),e2))**:计算e2的值然后存入到e1作为地址的内存中(先计算e1再计算e2)

⑩**JUMP(e, labs)**:跳转到e地址或者labs为label的地址

①**CJUMP(o, e1,e2,t,f)**:依次计算e1和e2, 生成值a,b;然后用比较运算符操作aob,如果结果为true跳到t,反之跳转到f; 对于CJUMP和JUMP语句,还不知道label的具体值,需要使用两张表:真值标号回填表(true patch list)和假值标号回填表(false patch list).

②**SEQ(s1,s2)**:语句s1后面跟s2

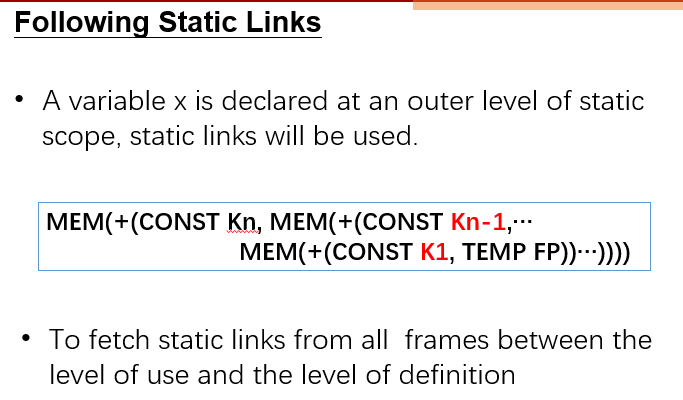
③**LABEL(n)**:定会一名字后的常数值为当前机器代码的地址(Name是使用这个symbol，比如goto .L，而Label是定义这个symbol，

④**EXP(e)** 评估 e 并丢弃结果。

翻译为IR

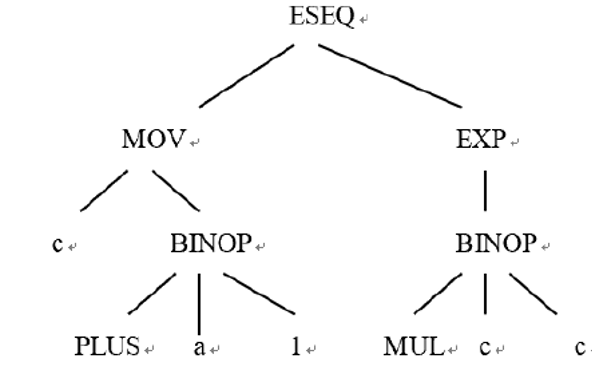
**Simple Variables**:存放在栈帧的变量v转化为**MEM(BINOP(PLUS, TEMP fp, CONST k))**, k是栈帧内v的地址偏移

**追踪静态链**: k1~kn-1是各个嵌套函数的静态链位移



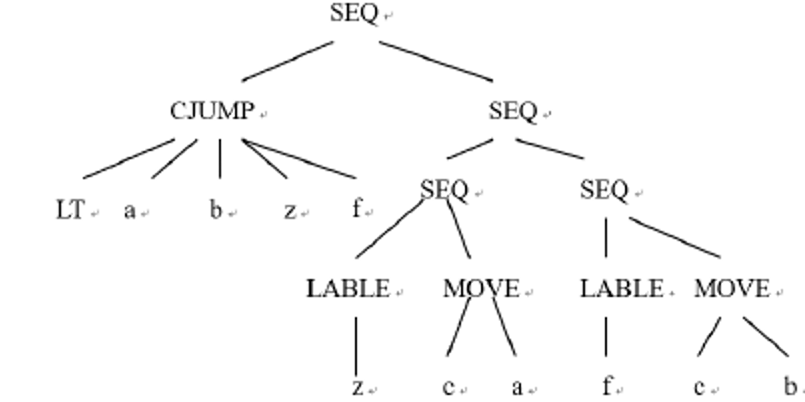
**数组变量下标**:a[i]表示为**MEM(+(MEM(e), BINOP(MUL, I, CONST W))**

**顺序语句** c=a+1; c\*c



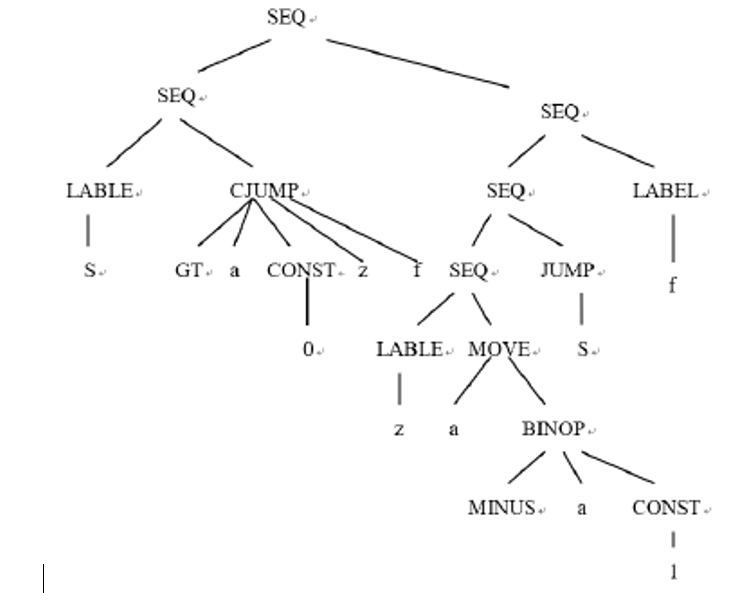
**条件语句** if a<b then c=a else c=b:

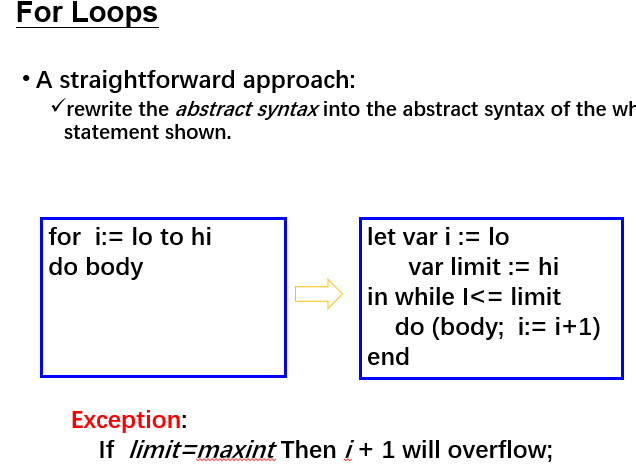
首先由一个CJUMP确认两类跳转，再有seq做连接，其中的n LABEL是if的结束



**循环语句** while a>0: a=a-1

循环语句有：初始化，循环体，循环条件判断。首先，有4个标签，分别是循环条件标签z，循环体标签z2，还有为了凑CJUMP的标签z1，循环体结束后的n：如果a>0已经不存在了，会跳转到z1，再由SEQ跳到循环体结束的n；如果还要继续做，那么就是从z2那里继续做，知道SEQ全做完重新跳回判断z





**声明** 函数被翻译为入口处理代码(prologue)/函数体(body)和出口处理函数(epilogue)组成的汇编语言代码.

**入口处理函数**包含:(1)声明一个函数开始的伪指令(2)函数名字的label定义(3)调整栈指针的一条指令用于分配新的栈帧(4)将逃逸参数保存至栈帧的指令,以及将非逃逸参数传送的新临时寄存器指令(5)保存在此函数用到的caller-save寄存器

**出口函数**位于函数体之后,包含:(1)将返回值传送至专用与返回结果的寄存器(2)用于恢复callee-save的寄存器取数指令(3)恢复栈指针,释放栈帧(4)return指令(5)声明函数结束的伪指令

**CH8 基本块**

**CJUMP**能够跳转到两个标号的任意一个,但实际的是条件为假时跳转到下一条②**ESEQ**会使得子树的不同计算顺序产生不同结果③表达式使用**CALL**会有计算顺序不同的问题④CALL的嵌套调用(作为另一个CALL的参数)会出问题,覆盖存放返回值的寄存器的值

规范树(canonical tree)

没有SEQ与ESEQ，并且call的父节点是exp或者move

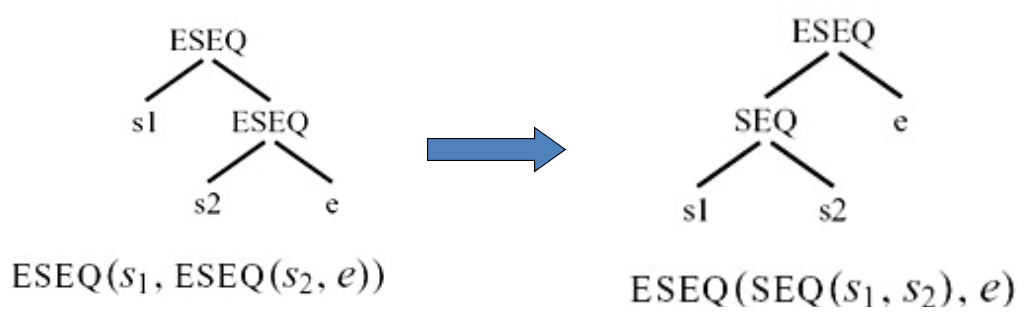
**重写流程**:①一棵树去掉SEQ和ESEQ，重写成规范树 ②将树分组合成不含转移和标号的基本块(basic block)集合 ③对基本块进行排序形成一组轨迹(trace);每一个CJUMP后就是其false标号

**linearize**(消除eseq，将call移动到顶层)、**basic blocks**(一组语句序列)、**Trace schedule**（使得每一块后面的block就是cjump为假的跳转）

**ESEQ消除**， 整体方法是在树中一级级的网上提升，直至ESEQ变为SEQ

**ESEQ(s1, ESEQ(s2,e))→ESEQ(SEQ(s1,s2), e)**

要执行e是s1和s2都要有effect，等价于SEQ这个s1,s2然后看是否有effect，再做e

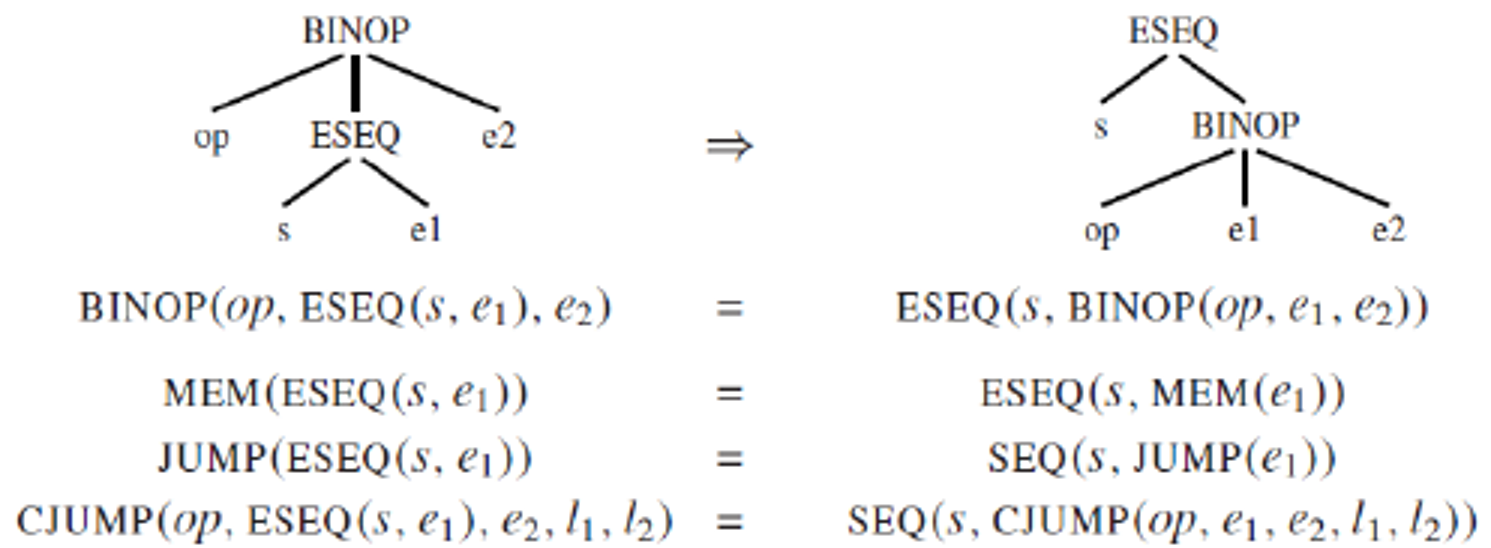


**BINOP(op,ESEQ(s,e1),e2)→ESEQ(BINOP(op,e1,e2))**

**MEM(ESEQ(s,e1))→ESEQ(s, MEM(e1))**

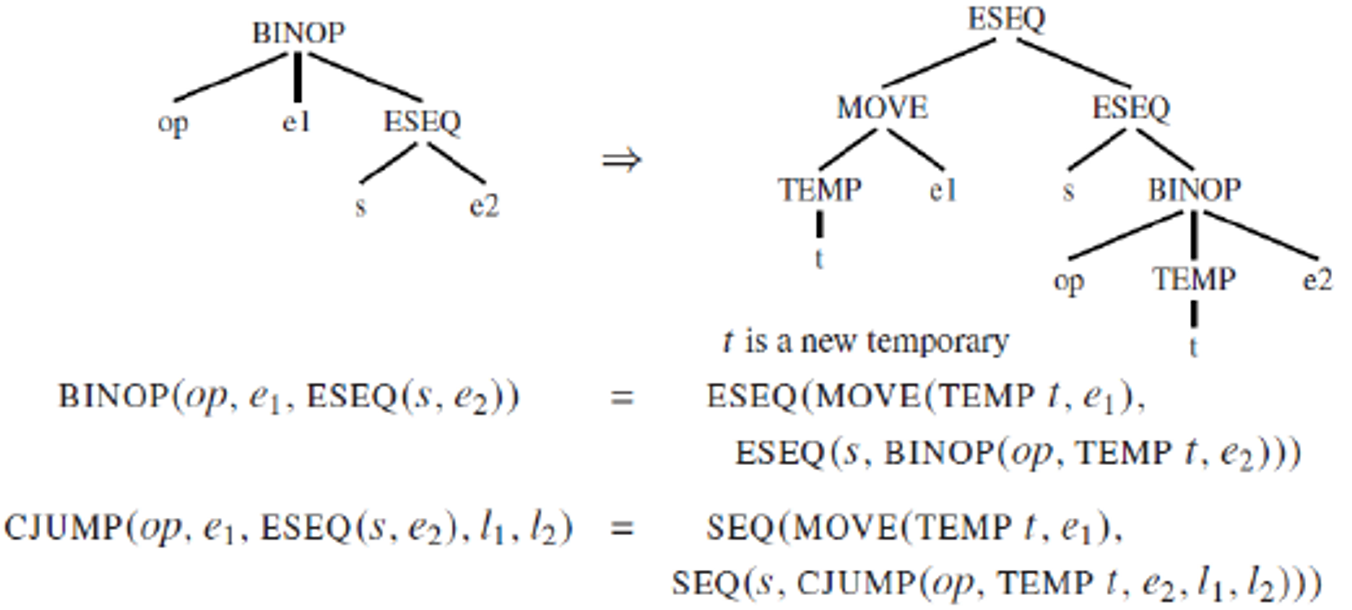
**JUMP(ESEQ(s, e1))→SEQ(s,JUMP(e1))**

**CJUMP(op,ESEQ(s,e1),e2,l1,l2)→SEQ(s,CJUMP(op, e1, e2, l1,l2))**

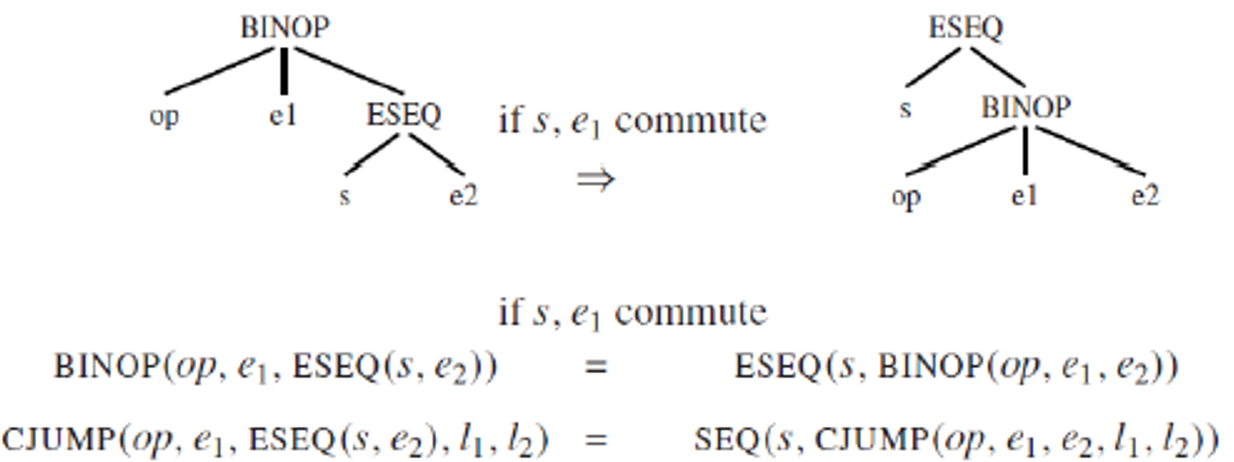


**BINOP(op,e1, ESEQ(s,e2))→ESEQ(MOVE(TEMP t,e1), ESEQ(s,BINOP(op,TEMP t,e2)))**

**CJUMP(op,e1,ESEQ(s,e2),l1,l2)→SEQ(MOVE(TEMP t,e1), SEQ(s,CJUMP(op,TEMP t,e2,l1,l2))).**



如果ESEQ中s和e1是可交换的(commute),那么可以直接把s和e1交换,ESEQ提出来。commute的思路是：常数可以与任何语句交换，空语句可与任何表达式交换，其余的都不能交换。



**CALL移到顶层**: 以BINOP(op,CALL(),CALL()….)为例, 第二个CALL会在BINOP执行前覆盖第一个CALL返回在RV寄存器里的值.

解决办法是使用ESEQ将返回值保存到一个新的临时变量:CALL(fun,args)→ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

最后对得到的**SEQ(SEQ(SEQ(..., sx), sy), sz)**，应用**SEQ(SEQ(a, b), c) = SEQ(a, seq(b, c))**，得到SEQ(s1, SEQ(s2, ..., SEQ(sn-1,sn)...))

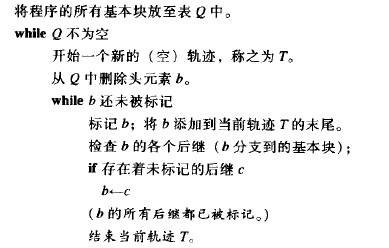
**处理条件分支**

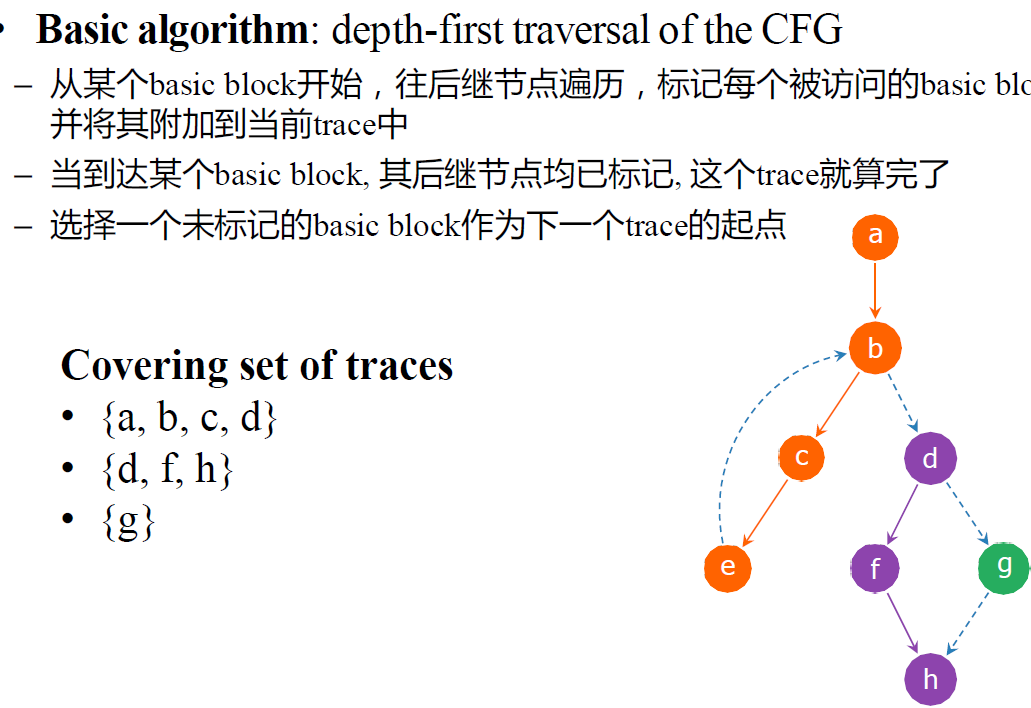
**用于解决CJUMP的问题，步骤1.找basic blocks 2.reorder得到trace**

**基本块(basic block):**取一列规范树:①第一个语句是LABEL②最后一个语句是JUMP或CJUMP③没有其他的LABEL,JUMP或CJUMP

**2.划分基本块方法:**从头到尾扫描语句序列,每次发现一个LABEL就开始一个新的基本块并结束上一个基本块;每发现一个JUMP或CJUMP就结束一个基本块(并开始下一个基本块).如果过程还遗留任何基本块不是JUMP或CJUMP结尾的,则在块末尾增加一条转移到下一个基本快标号处的JUMP;如果有任何基本块不是以LABEL开始的,则生成一个新的标号插入到基本块开始;在末尾添加done LABEL,将JUMP(NAME done)放到最后一个基本快末尾.

**轨迹(trace):**程序执行期间可能连贯执行的语句序列.要寻找一组能够覆盖整个程序的轨迹集合,且每一个基本块仅出现在一条轨迹中.





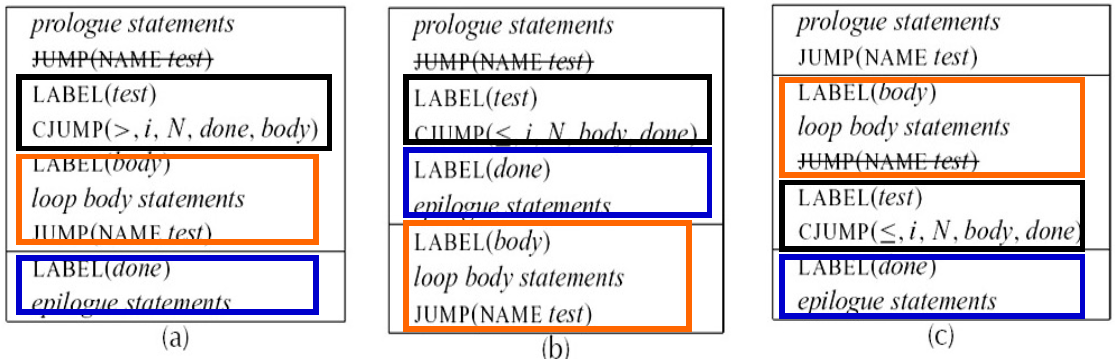
**完善:**①所有后面跟false标号的CJUMP不变②对任何后面跟true标号的CJUMP,交换器true标号和false标号以及判断条件取反③对气候跟随的既不是true也不是false标号的CJUMP,生成新的标号f’并重写CJUMP,使得其false标号紧跟其后.

**最优轨迹**：频繁的循环体有单独的轨迹，其中的迭代跳转要最少。比如下例c最优：

(a):While循环的每个迭代有一个CJUMP和一个JUMP

(b):使用了不同traces,但每个迭代仍有一个CJUMP和一个JUMP

(c):每个迭代都没有JUMP



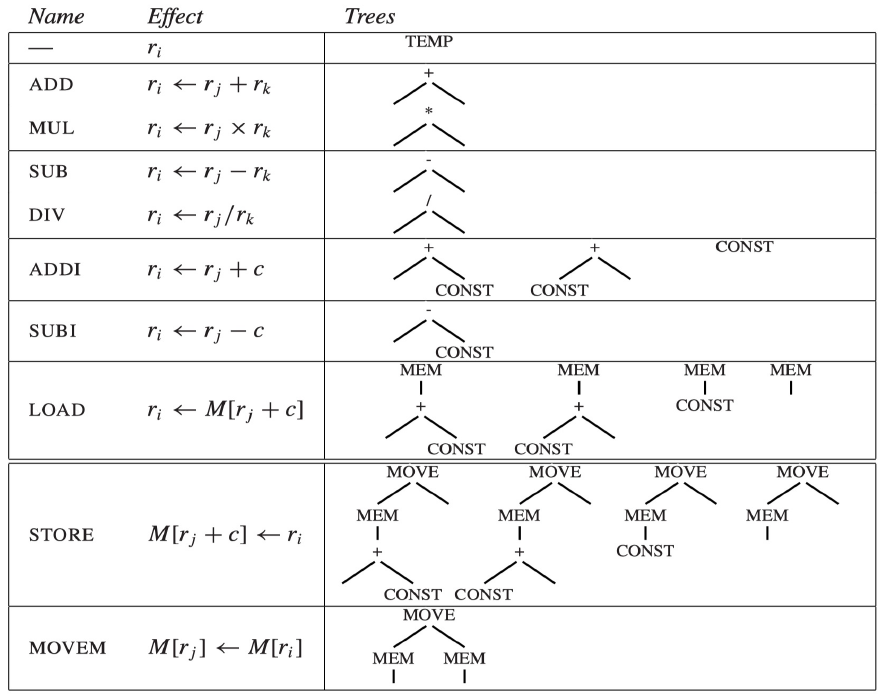
**CH9 指令选择**

**Jouette architecture**：RISC 差不多，加载/存储架构,r0 始终包含零，MOVEM 除外每个周期仅执行一条指令。有两类寄存器(a寄存器:存地址;d寄存器:存数据)，使用CFG来描述瓦片,文法具有高度歧义性,但是动规可以很好处理.

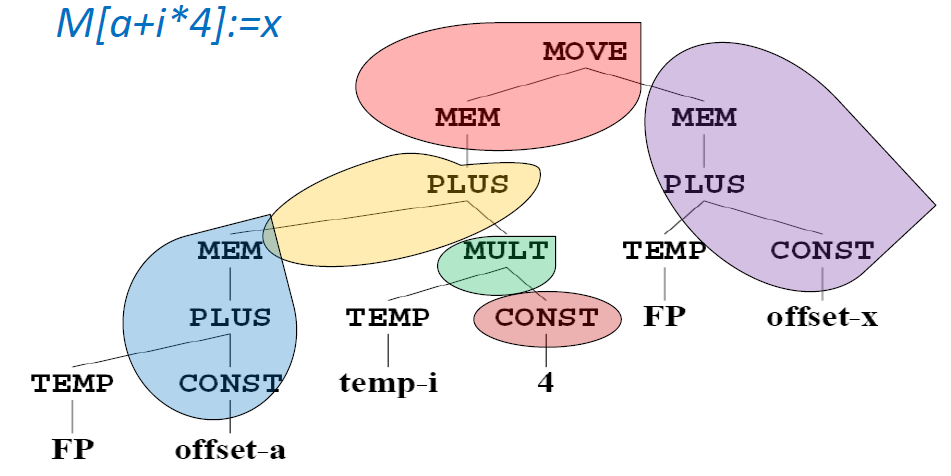
Each machine instruction can be expressed a fragment of an IR tree, called a **tree pattern**

树型到指令

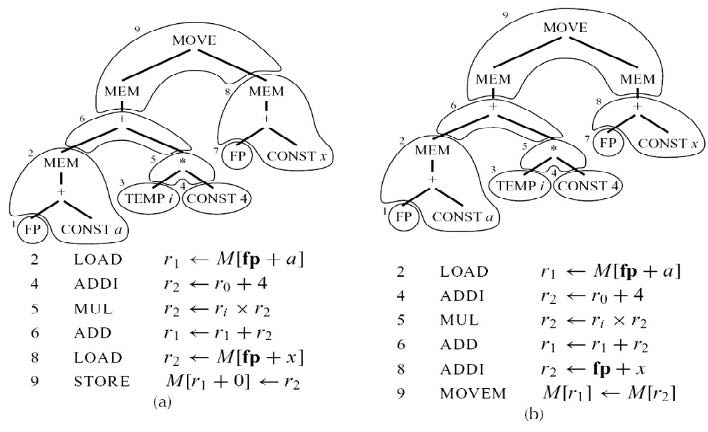
可以把一条机器指令表示成IR书的一段树枝,称为树型(tree pattern)。**指令选择的任务就变成了使用树型的最小集合来覆盖(tiling)**。使用Jouette体系结构,将树型映射成指令.指令和树型的映射如下表:



可以想象为铺地板的过程，每一个tree pattern就是各种大小不一的瓷砖（tile），指令选择就类似用瓷砖铺满屋子。比如，a[i] := x (M[a+i\*4]:=x)



一棵树可以有多种tiling的方式,但是一定要按照给定的结构去tiling:

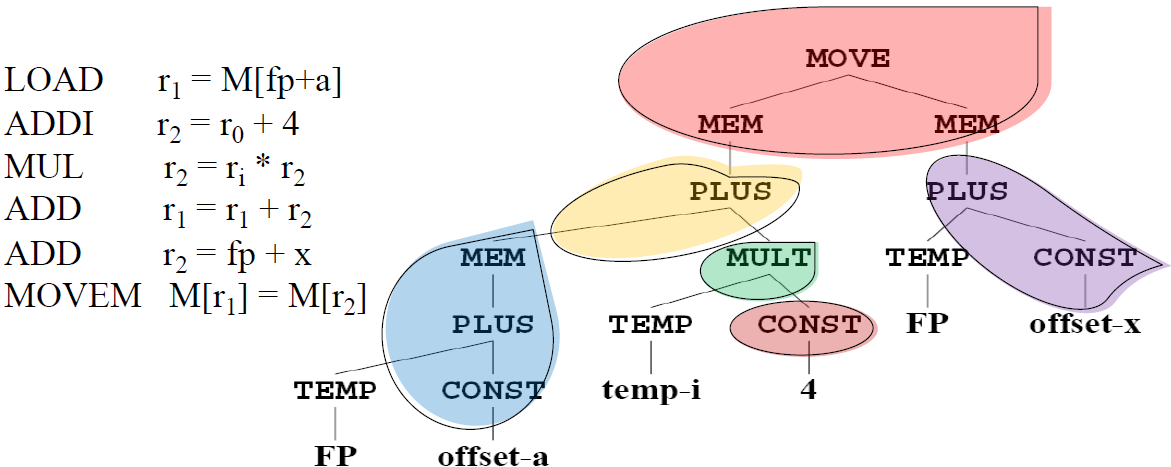


**如何选择能够以最少执行时间覆盖 IR 语句树的图块？**小图块确保我们可以对每棵树进行图块处理，大图块以提高效率。通常希望选择大图块，更少的指令

**最佳覆盖(optimum)**:瓦片的代价和可能是最小的覆盖,类似于全局最优；**最佳覆盖(optimal)**:不存在两个相邻的瓦片能连接成一个代价更小的瓦片覆盖. 每一个最优覆盖同时也是最佳的,反之不然.

指令选择算法

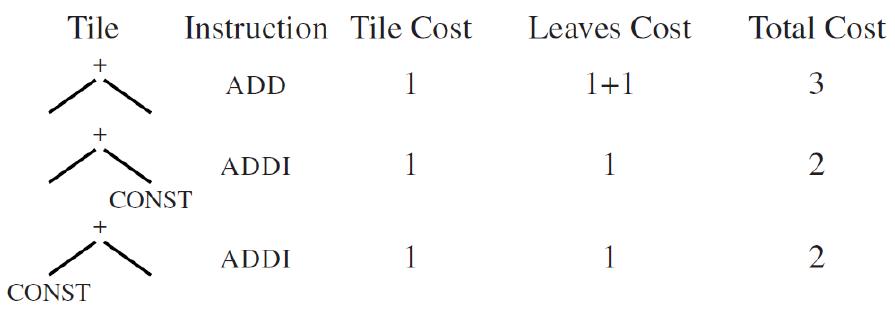
**Maximal Munch算法 for** optimual: 从树的根节点开始greedy寻找适合他的最大瓦片(包含的node数量最多),按照Jouette体系结构可能会覆盖其他几个节点（就是左边那张表），如果有两种方案，任选;对遗留的其他子树也进行相同操作;Maximal Munch算法的tiling是top-down的,但是指令的生成是逆序的(很好理解,因为上层的覆盖指令需要下层的指令提供操作数,所以是逆序).



**动态规划 for optimum**: bottom-up,会给每个节点指定一个代价:可以覆盖该节点为根的字数的最优指令序列的指令代价之和. Cost=cost of selected tile + cost of subtrees. **选择成本最小的图块并向上递归**

🍔书上给出的例子如下：

假设要做一个1+2这样两个常数的加法，那么采取不同的瓦片得到不同的结果，如果用ADD的话，需要把叶子的代价先求出，拿一个const的代价是1，所以第一行就是先把两个const放到寄存器里再做加法，因此是1+1的叶子代价和一个加法的1代价。而下面两行都是用不同的瓦片，只需要把一个const放到寄存器里，然后就可以直接做了



再看一个，第一行是运用了上面的最优代价作为子代价，下面则是用了不同的瓦片

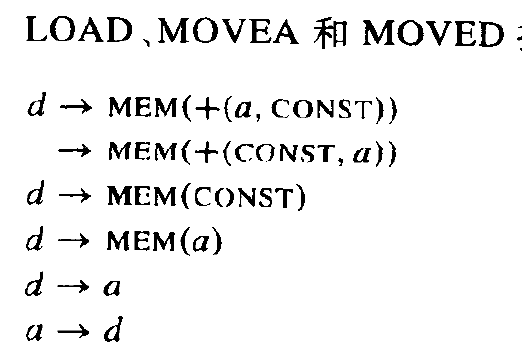


一旦求出了根节点（也就是整棵树）的代价，便进入到指令流出阶段（instruction emmsion），

从下到上构建指令。比如上面的就是：ADDI R1<-R0+1, LOAD R1<-M[r1+2]

**树文法(Tree Grammar)**:了解即可，基于树形语法的算法，这种算法是动态规划算法的一种扩展形式。而这种算法特别适用于指令集比较

复杂、寄存器和寻址模式比较多的计算机。它的好处是，可以在编译过程中自动选择最佳的寄存器和寻址方式，从而减少程序的指令数和执行时间。下面是生成load,move,moved指令的树文法生成过程参考.



**算法效率**:T个瓦片,平均每个匹配的瓦片有K个非叶子节点.K’是在给定子树中为确定匹配那个瓦片需要检查的最大节点个数(近似于最大瓦片的大小).假定平均每个树节点可以与T’个瓦片匹配.输入树的节点为N.①Maximal Munch:②动态规划:动规的比例常数比Maximal大,因为要遍历两遍.K’,K,T是常数,线性复杂度.

CISC机器

**RISC机器特征**:①32个寄存器②仅有一类整数/指针寄存器③算数运算仅对寄存器进行操作④采用”三地址”指令(r1←r1+r2)⑤取指令和村指令只有M[reg+const]模式⑥每条指令长度固定为32位⑦每一条指令产生一个结果或作用,无副作用

**CISC机器特征**:①不多的几个寄存器(16,8,6)②寄存器分不同类型,某些操作只能在特定种类的寄存器上进行③算术运算可以通过不同的寻址模式访问寄存器和储存器④指令是”两地址”指令⑤有不同的寻址模式⑥有由变长操作码加变长寻址模式形成的变长指令⑦指令具有副作用(自增寻址方式)

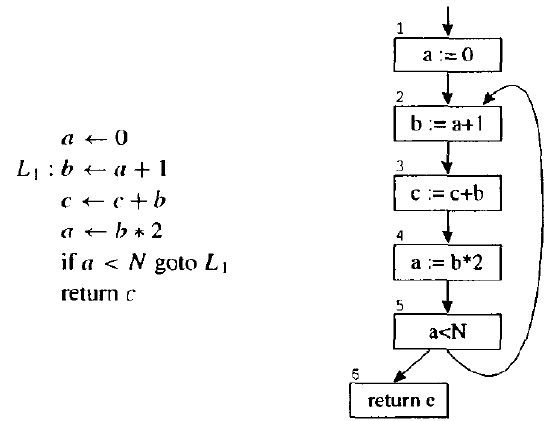
**CISC机器的特点解决难题**:①寄存器较少:不限制生成TEMP节点,假设寄存器分配能完成分配工作②寄存器分类:将操作数显示地传送到相应的寄存器中③两地址指令:增加一条额外的传送指令④算数运算可以访问存储器:指令选择阶段将每一个TEMP节点转化成一个寄存器引用.⑤若干种寻址模式:优点(破坏寄存器少;指令代码短)⑥变长指令:不管;⑦副作用指令:三种解决办法(a)忽略地址自增指令,希望其自动消失(b)在采取树型匹配的代码生成器的上下文中使用特别方式匹配方言(c)使用完全不同的指令算法,基于DAG样式.

**CH10 活跃分析 Liveness Analysis**

编译器需要分析程序的中间表示,以确定那些临时变量在同时被使用.如果一个变量的值在将来还需要使用,则变量是活跃的(live),这种分析叫做**活跃分析**

**control flow graph**: 如果语句之后可以紧接着执行，就有一条边(这里一个块里就一条语句)

**Liveness**: 变量位索引的集合,变量在那几条边上活跃的边集合.如果x在p'语句块被使用到(取用它的值)，那么便有一条从p->p'的活跃。如下例： b的活跃范围是{2->3，3->4}，a的活跃范围是{1->2，4->5->2}活跃（这里的a<N是对a的使用，a=b\*2不算对a的使用)。这两个范围内，a/b可以放在一个寄存器里，另外的c可以放在另一个寄存器里。这就是，靠活跃分析，来做寄存器分配。



**出边(out-edge)**:节点引向后继节点的边;**入边(in-edge)**:由前继节点指向的边.**succ[n]**是节点n的后继节点;**pred[n]**是节点n的前驱节点

**定值(define)**:对变量和临时变量的赋值成为变量的定值;**使用(use)**:出现在赋值号右边的变量为其使用.

**Liveness variable**: 变量在边上活跃是指存在一条边通向该变量的一个use的有向路径,且不经过该变量的任何def.

如果变量在一个节点的所有入边上全是活跃的,那么该变量是**入口活跃的(live-in)**;若一个变量在一个节点的所有出边上都是活跃的,那么该变量在该节点是**出口活跃的(live-out)**.对应的有集合**in[n]**和**out[n]**

下面的rule很直观，想想就行了

Rule 1: If ain[n], then for ∀mpred[n], aout[m]

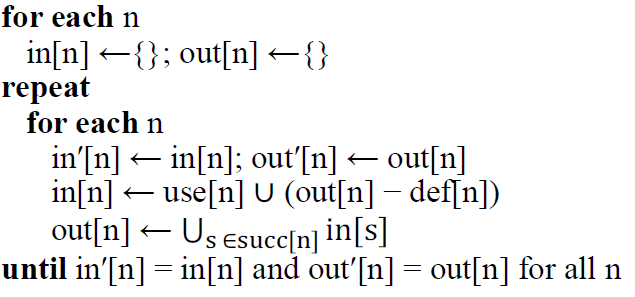
Rule 2: If ause[n], then ain[n]

Rule 3: If aout[n] and a ∉ def[n], then ain[n]

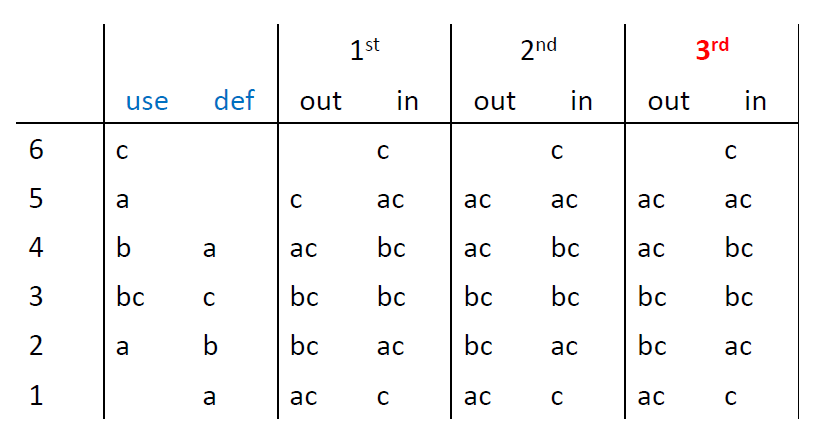
数据流方活跃性计算

活跃性计算:就是计算流图每一个节点的in和out集合(方程10-2):

迭代计算直到不动点

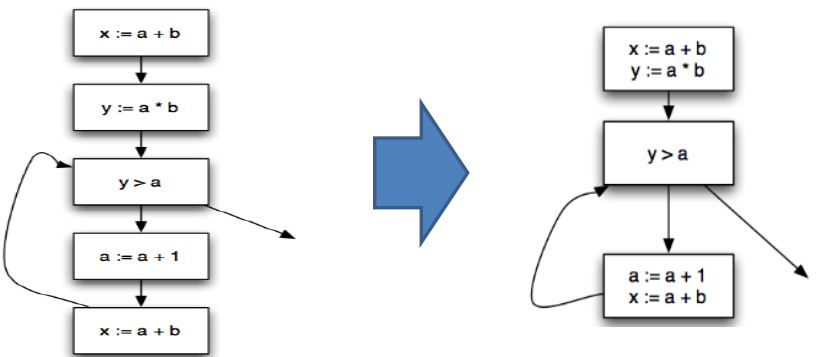


适当排序可以显著加快算法的收敛过程,一般要**从程序末尾往前算**,**先算out再算in**,可以显著提高速度和正确率.信息活跃性是沿控制流箭头的反方向流动的,计算顺序同理.比如上一张图中的CFG的计算结果如下：



**关于CFG优化的讨论**

**Basic block**如果一个点只有一个前驱和一个后继，那么可以合并



**One variable at a time**：上面我们是基于整体性的做数据流方程的，但是，对于大型程序或者复杂的代码，同时计算所有变量的数据流信息可能会非常耗时。因此，当只需要某个变量的数据流信息时，可以单独使用深搜方法计算该变量的信息。

**集合表示**

**位数组(bit array) for dense set**:程序中有N个变量,用N位数组表示集合①求并集对位数组求按位或②时间效率:对每个字有K位的计算机,并运算需要N/K次操作

**有序变量表**(sorted lists) for sparse set:链表的成员是组成集合的元素①并集通过合并链表实现②时间开销和求并集的集合大小成正比.集合稀疏(平均少于N/K)用有序链表表示速度会更快(越稀疏越快)

**时间复杂度**:for循环初始化节点in,out需要O(N^2);repeat循环的最坏时间复杂度是**O(N^4)**.由于活跃信息大部分稀疏,实际运行时间在**O(N)和O(N^2)之间**

**最小不动点**: 数据流方程的解只是保守的近似解,只能保证生成的代码一定是正确的,但是产生代码的指令所使用的寄存器比实际需要的多. **定理**: 方程10.2有一个以上的解(in,out计算公式方程); **定理**: 方程10.2所有解都包含最小解(least solution)；**定理**:不存在程序H,它以任意程序P和输入X作为自己的输入.当P(X)停止时返回真,当P(X)无限循环时返回假. 证明:假设存在这样一个程序H,我们会得出如下矛盾.从H构造函数F,F(Y)=if H(X,Y) then (while true do()) else true. 推论:不存在程序H’(X,L),对任何程序X和X中标号L,可以判断出X在执行中是否曾经到达了标号L.

**动态活跃**under-approximation: 如果程序的某些执行从 n 转到 a 的使用，而不需要经过 a 的任何定义，则变量 a 在节点 n 处动态有效。动态活跃一定是静态活跃的

**静态活跃**over-approximation: 如果存在从 n 到 a 的某个使用位置的控制流边的某条路径，且该路径不经过 a 的定义，则变量 a 在节点 n 处静态有效。

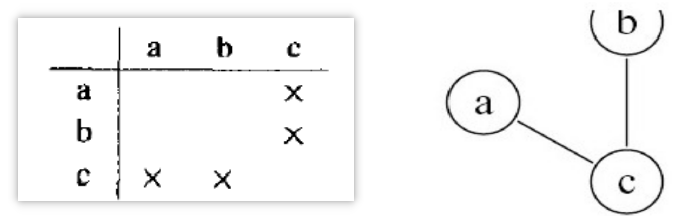
冲突图

**冲突**:阻止将两个同时活跃(冲突)的临时变量分配到同一个寄存器的条件称为冲突(interference).

**冲突原因**:①临时变量在程序的同一点同时活跃②某些寄存器必须被使用时,临时变量不能占用这些寄存器.

**冲突表示**:①冲突矩阵:n\*n的矩阵,n时临时变量的数目,(i,j)打叉表示i,j冲突.②冲突图:冲突矩阵的另一种表现形式.

绘制冲突图的办法:为新定值(def)添加冲突边的办法是①对变量a定值的非MOVE指令,以及在该指令节点n处,任意bi∈out[n],添加冲突边(a,bi) ②对于节点标号为n的MOVE指令a←c,对bi∈out[n]且bi≠c,添加边(a,bi). 注:可以给(a,c)画上虚线,便于寄存器分配的coalesce.



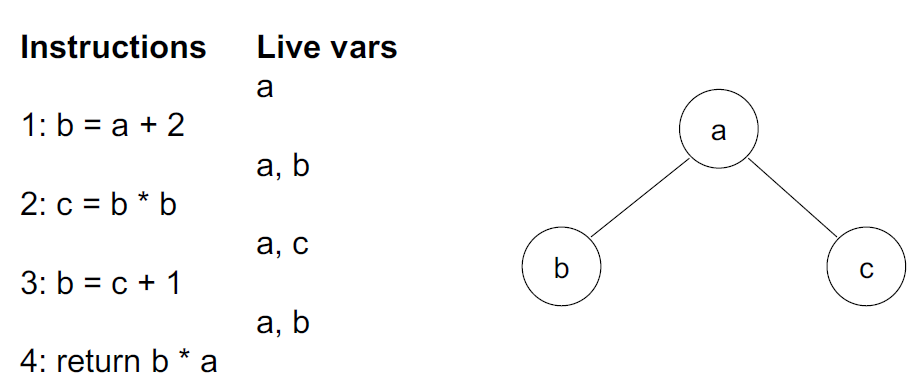
**CH11 寄存器分配 Register Allocation**

”颜色”就是寄存器,相邻节点不能着同一种颜色.这会导致部分变量必须保存到memory里,称之为**”溢出”spill**. 可以将MOVE指令的源操作数和目的操作数分配到同一个寄存器中，以便MOVE指令被删除

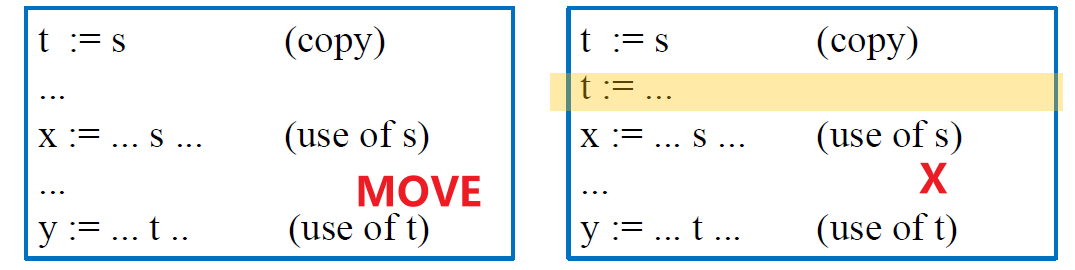
找到最小k着色是NP-hard，判断一个图是不是k-colored是NP-complete

图着色算法

**构造(build)**:构造冲突图。Node变量，edge两变量不能同时在一起(两变量同时live)。



特别注意，这个地方MOVE指令不能连边（下图中如果S,t没有其他用途的话，是MOVE）



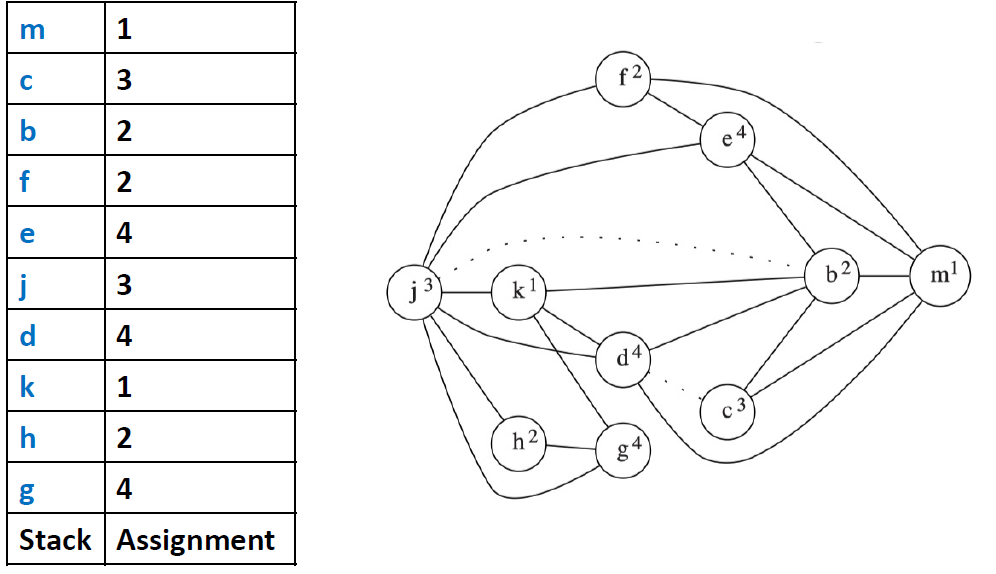
**简化(simplify)**: 把邻居少于寄存器个数K的点从图中去除并且压栈。如果一个图G的节点n的度小于颜色K,那么去掉该节点后的图G’如果能被K着色,G也可以.

**溢出(spill)**:简化过程中如果只有**高度数**(significant degree)点(度≥K),此时简化失效;可以把一个变量放入内存中。我们还会**乐观着色**估计这个点有可能不冲突，先暂时放入栈里，视为潜在溢出节点，留待检查。（潜在的原因是：它的邻居有可能同色，使得该节点也能分配到一种颜色）

**选择(select)**:从栈顶开始弹出着色，如果potential spill node真的无法分配，就成了实际溢出节点

**重开(start over):**如果不能为某些节点着色,那么需要重写程序.重写后还需要重复上述所有流程,直到没有实际溢出

书上的例子：



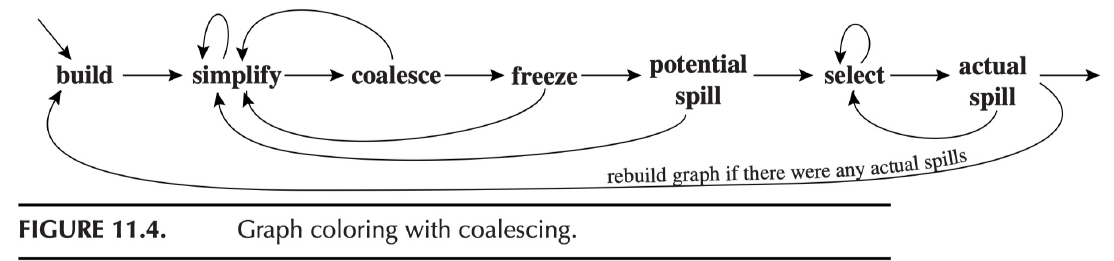
合并(coalesce)

合并的原因是两个变量有move关系，也就是在某一刻会出现a=b，并且a与b之间没有干涉边（即不存在，不能放入同一个寄存器的冲突），那么新节点ab可以合并，其所有边就是a和b之前的边。但盲目合并可能使得图不可着色，我们采取保守的合并策略，简单说就是，确保这次合并不会影响图着色。

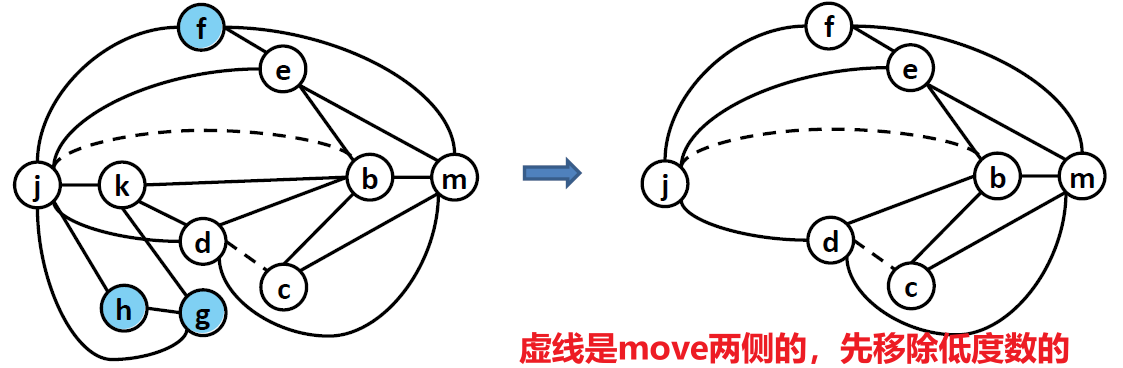
保守的(conservative)的合并策略:**Briggs策略**:如果a,b合并后的节点a&b的高度数邻节点个数少于K,则a和b可以合并.**George策略**:对于a的每一个邻居t,t与b已有冲突或者t是低度数节点。这两种策略下合并不会改变图的着色性,合并不成功时仍然是安全的.可能有多余的MOVE指令,但是不会出现溢出

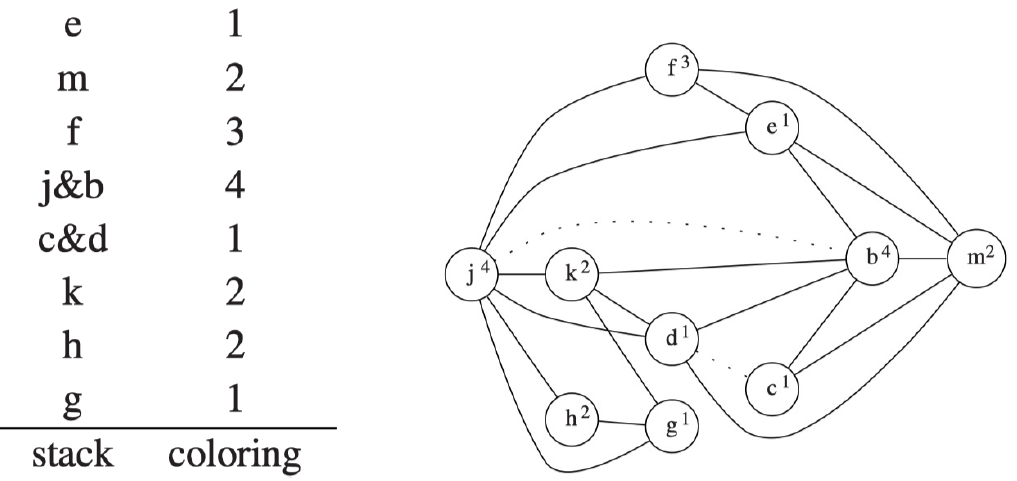
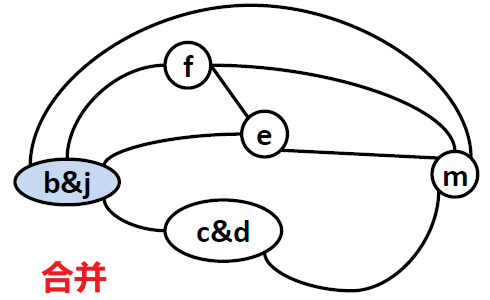
带合并的图着色算法

①构造:构造冲突图,将节点分类为传送有关(move-related)和传送无关的(non-move-related).传送有关节点是传送指令的源操作数或目的操作数,可以在一对传送有关节点之间画一条虚线 ②简化 ③合并:对简化的成果按照上述合并策略进行保守合并,并且重复简化和合并的过程,直到剩下的节点全部是高度数节点或传送有关节点. ④冻结(freeze):简化和合并都不能进行时,寻找一个低度数的传送有关节点,冻结这个节点所关联的那些传送指令,放弃合并的希望(把虚线画成直线)创造更多的简化机会.然后重新开始简化和合并.（能合并的两个点，必须只有一条虚线，如果既虚线也实线（称为**constrained**）就不能合并，必须要冻结）⑤溢出:如果没有低度数节点,选择潜在可能溢出的高度数节点并压入栈中 ⑥选择:弹出整个栈并指派颜色.



比如





另外，课本在这里放了一个小概念，之前我们提到了遇到要放在内存的真溢出节点时候，需要重新开始，但是一种更有效的算法是，如果在发现需要溢出之前已经进行了节点合并，那么这些合并操作将会被保留下来；但是，如果在发现需要溢出之后进行了节点合并，那么这些合并操作将会被丢弃（**解除合并**）

预着色(precolor)

一些变量直接使用了真实寄存器,相当于已经固定了寄存器的使用。预着色的节点一定相互冲突, 在冲突图上表现为两两互联，①无法简化②无法指派颜色③无法溢出(认为寄存器节点的度是无限大)④可以参与合并

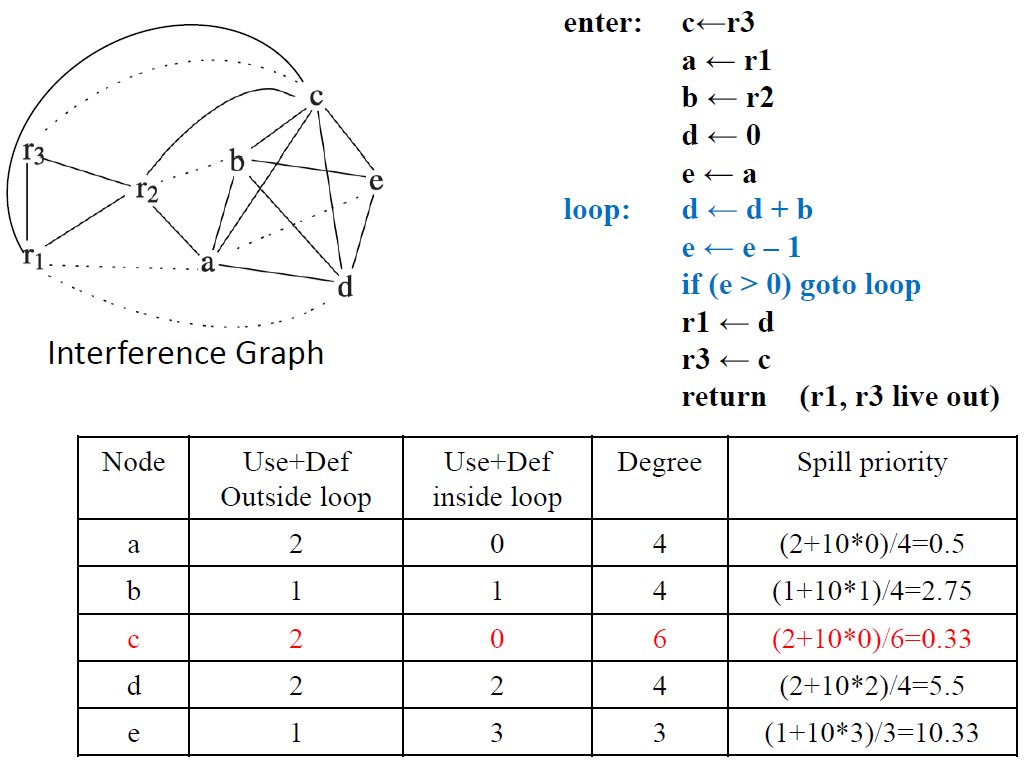
活跃范围主要靠**move给一个新的临时变量**，过程结束再move回来，这样它在这个过程中就是不活跃的。而当这个函数存在较大压力的时，t231会溢出，否则把t231和r7合并。

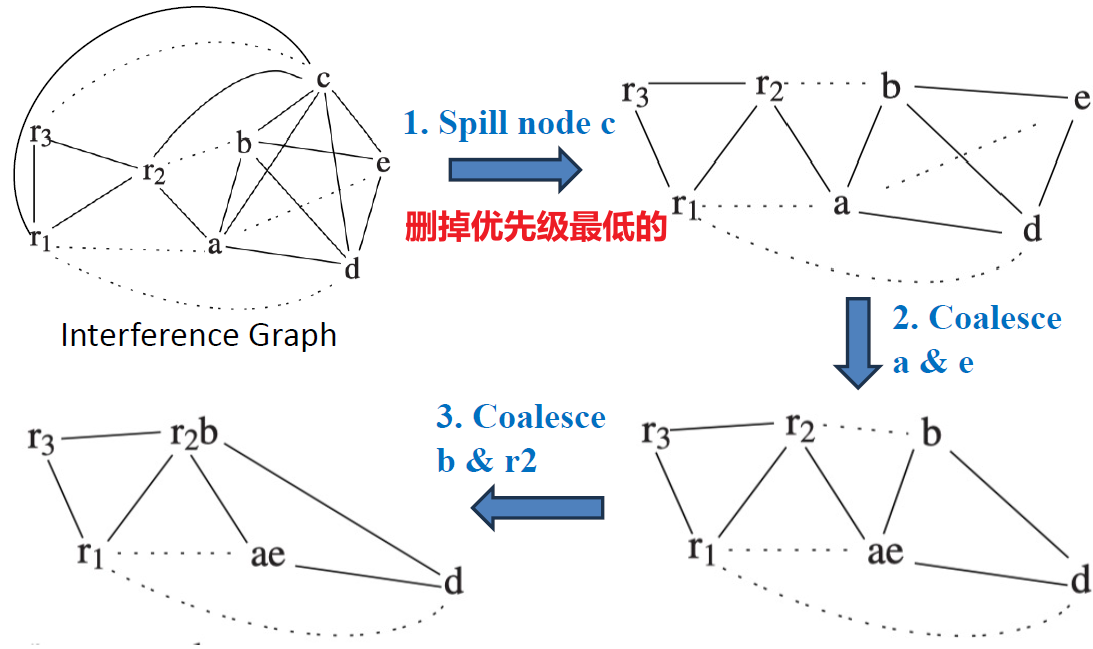


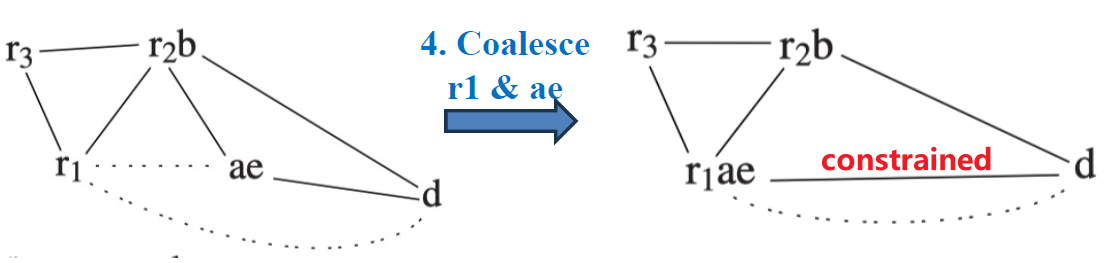
对于一个本地变量或编译器临时变量，如果它不跨越任何过程调用，通常应该分配到一个调用者**caller-save register**，这样就可以避免寄存器的保存和恢复操作。但是，如果一个变量需要跨越多个过程调用，那么就应该将它分配到一个被调用者**callee-save register**中，这样只需要在进入和退出被调用过程时保存和恢复该寄存器一次即可

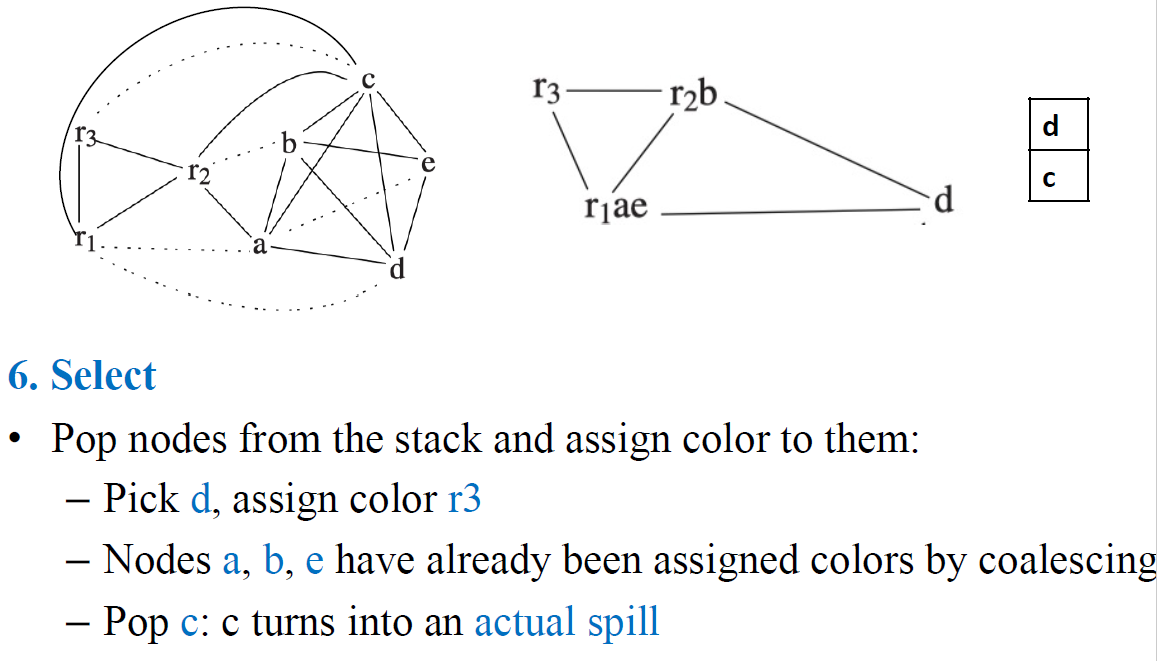
**溢出优先级**:对于节点a而言,变量a在循环外层的use和def的总数记为,循环内层的use,def总数记为;节点的度为.节点a的溢出优先级:

注意:D不包含虚线的计数,只包含实线的计数;Priority的值越小,优先级越高(优先溢出那些不经常被使用的高度数节点).



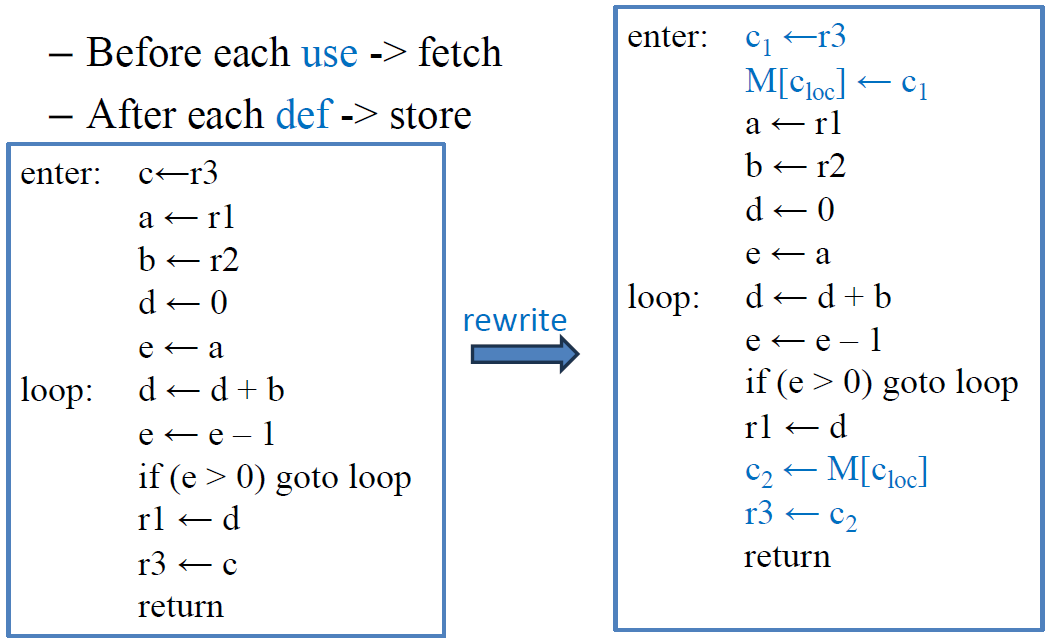


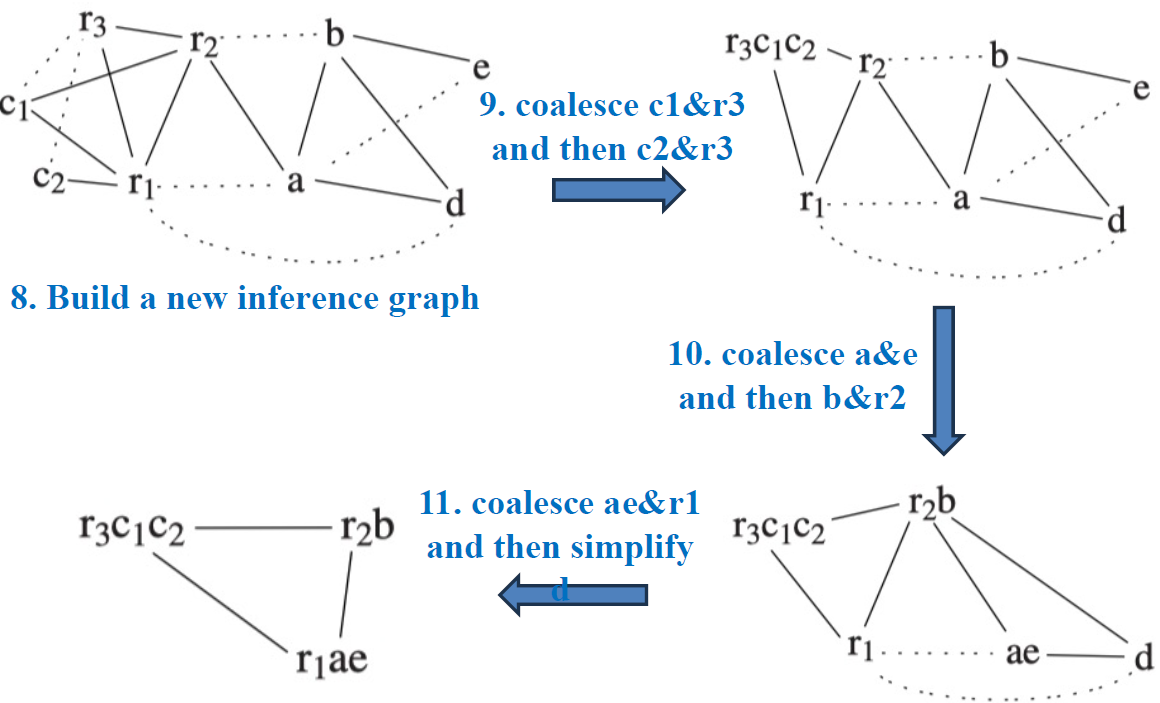


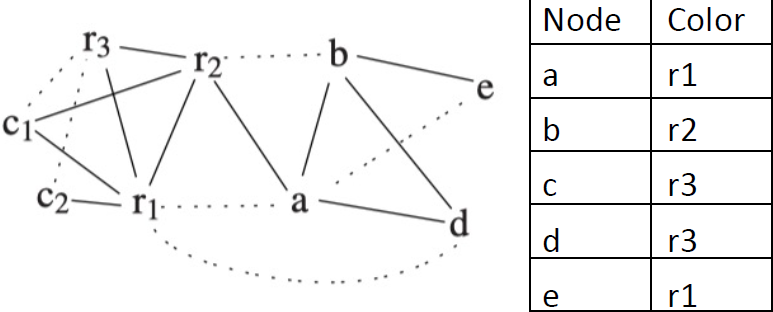


对**实际溢出**节点的处理:假设变量a发生了实际溢出,那么a必须保存到memory中(定义),同时**重写原先**的程序. 对def和use的修改如下：①对于a的每一个use,都要新建一个临时变量,从M[]位置读取a的值(因为a现在被存到内存中):变成 ②对于a的每一次def,也要生成一个新的,首先给定值,然后赋值到M[]: 变成

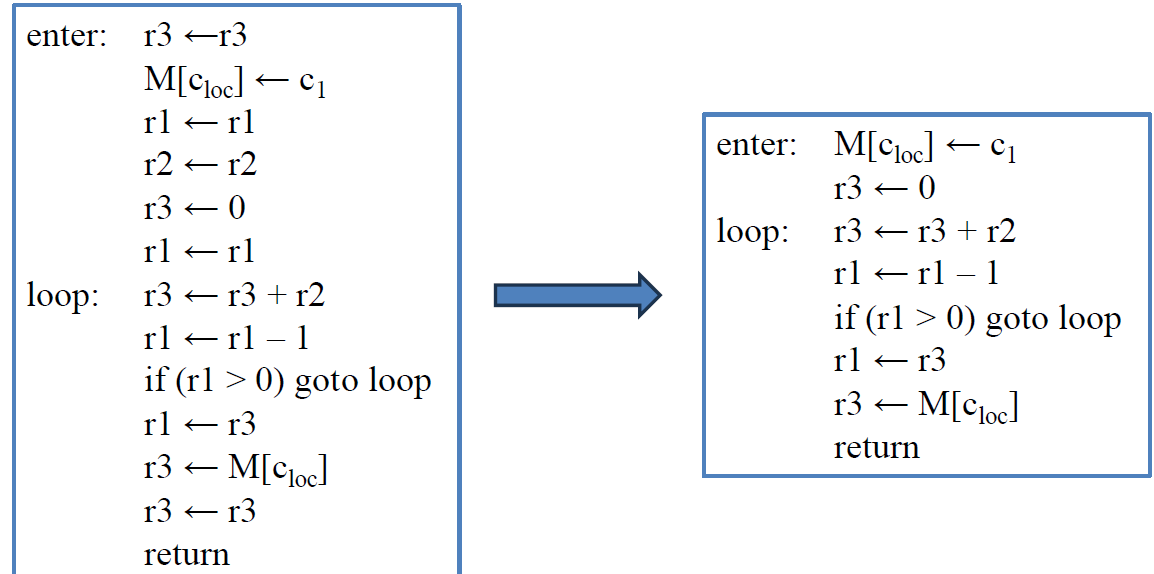
继续上例，进行重写







最近再进行一步化简，删除任何源和目标相同的移动指令：



**CH13 垃圾收集 Garbage Collection**

**静态区域**：编译时分配

**运行时堆栈**：激活记录,用于管理函数调用和返回

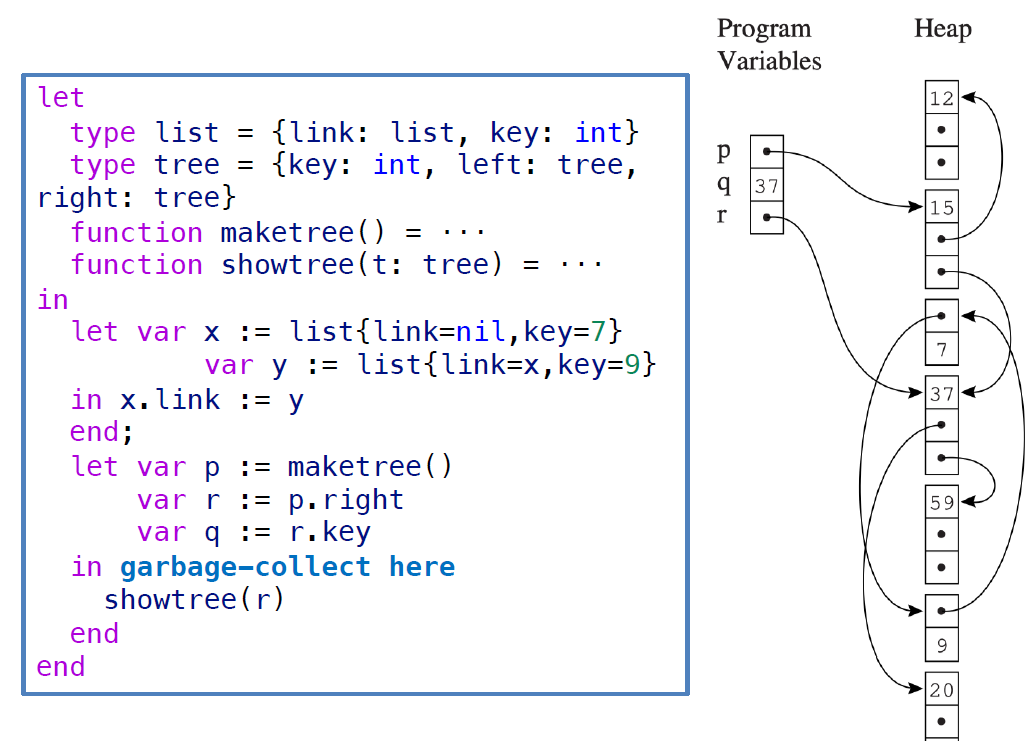
**堆**: 动态分配的对象和数据结，如malloc

**垃圾(garbage)**:在堆中分配且通过任何程序变量形成的指针链都无法到达的记录。**垃圾收集**，使得那部分空间可以重新使用，这不是由编译器而是由运行时的系统完成的。我们需要采用一种保守的方法，保证活跃记录都是可到达的，并尽可能减少那些可到达但非活跃的记录数量

**保守近似**:由于变量的活跃性不总能知道(停机问题的等价),只是将不可到达的内存当作垃圾.

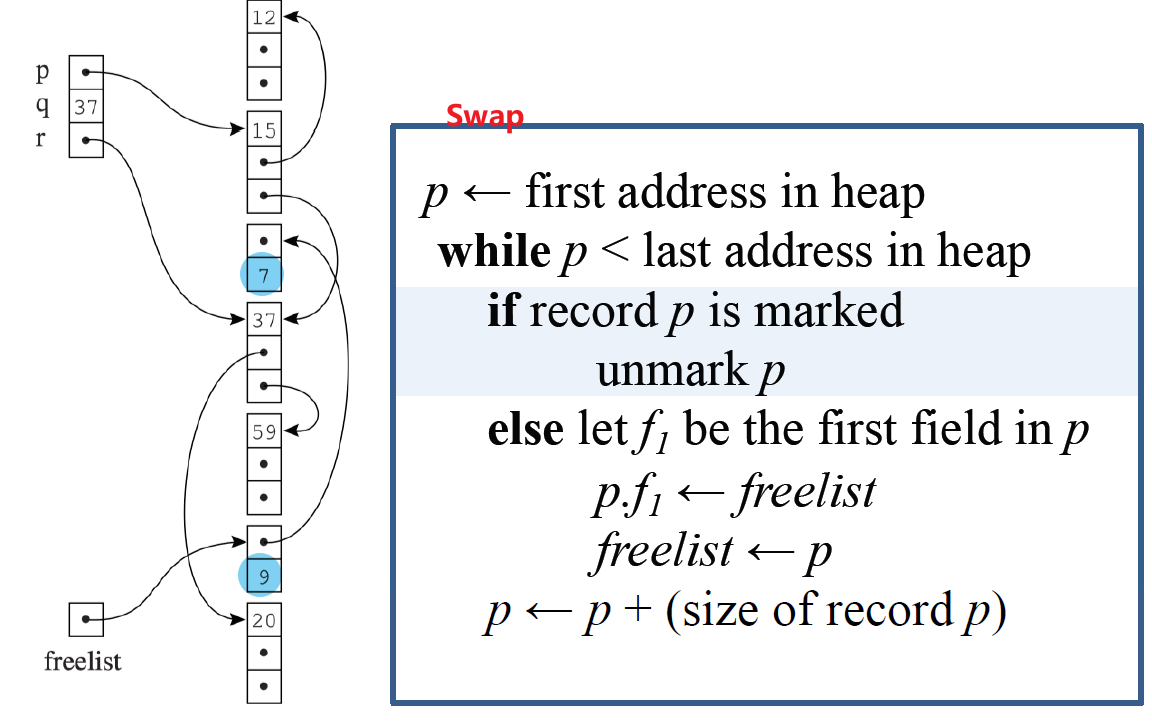
程序变量和堆的记录作为节点构成一个有向图.每个程序变量是图中的一个根.如果存在一条从根节点出发到达n的有向路径,则称堆内存节点n是**reachable**. p points to a record y: we mean the value of p is the address of y (let y be the name /identifier of the record （虎书中p是指针或者指向的意思）

首先给出书中示例，实际上是一个环+一棵树



Mark-Swap 算法

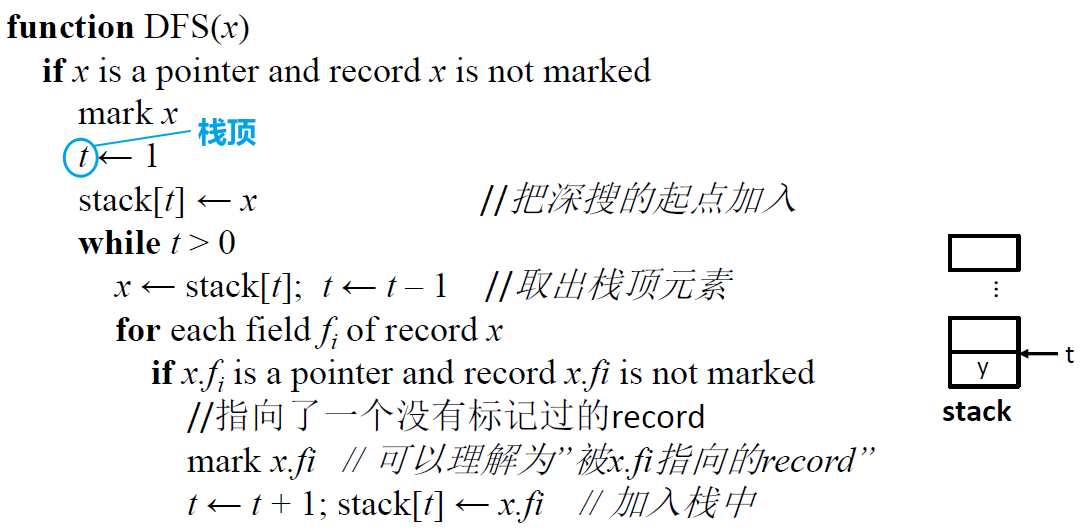
①标记阶段:使用DFS标记所有可达节点 ②清扫阶段:未被标记的节点一定是垃圾:通过从头到尾扫描堆内存,对未标记的节点连接到freelist中.同时清除所有已标记节点的标记.



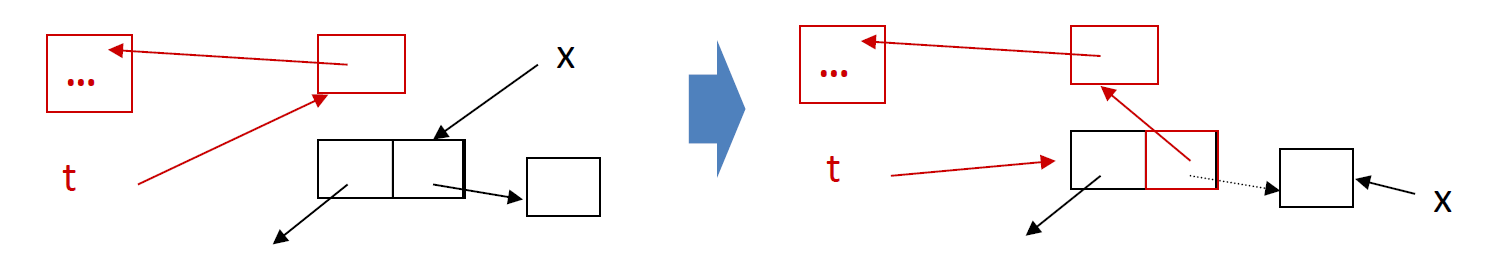
时间复杂度:标记阶段时间和标记节点个数成正比;清扫阶段时间与堆大小成正比. 假设大小为H的堆中有R个字可到达数据,则一次垃圾收集的代价是, 为常数;好处是可用大小为个字的自由储存单元补充空闲表. 通过摊还分析,最终的垃圾收集代价为

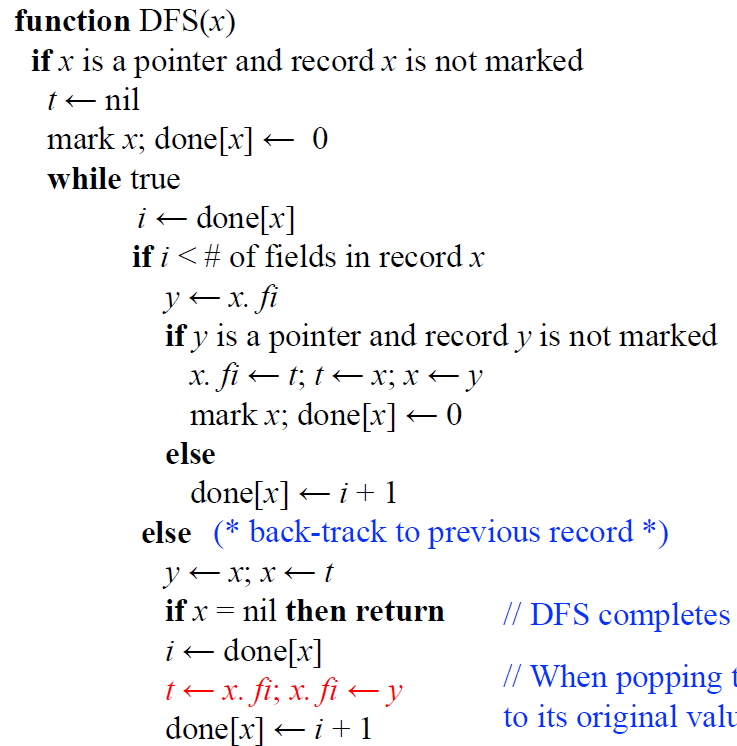
改进

上面的方法会碰到爆栈的问题，因此使用如下的手工栈是一个好选择，这里其实就是手动模拟了入栈和出栈，用t来控制。H words instead of H activation records！



但是辅助栈的空间大小与被分配的堆空间相同仍然不可接受. 于是又练一种更节省空间的方法：**指针逆转(pointer reversal)**:在记录域x.fi被压入栈后,不再查看原来的x.fi,而是用x.fi存储其父节点的指针(指向父节点x).当从栈中弹出x.fi的内容时,再将域x.fi恢复为原来值. 同时要求每个记录有一个名为done域,用以记录中有多少域已经被处理过.使用指针逆转的DFS:





变量t用于指明栈顶,栈内每一个记录x都是已经标记的记录.如果i=done[x]则x.fi是连接下面一个节点的”栈链”.当对栈执行弹出操作时,x.fi恢复为原来值.

引用计数算法

在这个机制中，每个记录都包含一个引用计数，用于记录有多少指针指向它。每当一个指针（例如指针x.fi）被赋值给记录中的一个字段，就会对指针(x.fi)所指向的原记录和现记录修改：指向新记录的引用计数将加一，而原来指向的记录（如 x.fi 原来指向的记录）的引用计数将减一。通过这种方式，程序可以跟踪每个记录有多少指针在引用它，以判断何时可以安全地释放该记录所占用的内存资源。当某个记录的引用计数降至零时，说明没有任何指针再指向该记录，可以将该记录放入空闲链表（Free List）中以供后续的内存分配使用。同时，也需要将该记录所指向的所有记录的引用计数减一，以确保内存资源能够被正确地回收和重复利用。

**优点**:简单易于实现 **缺点**:(1)无法回收成环的垃圾.(2)增加引用计数所需的操作代价很大.

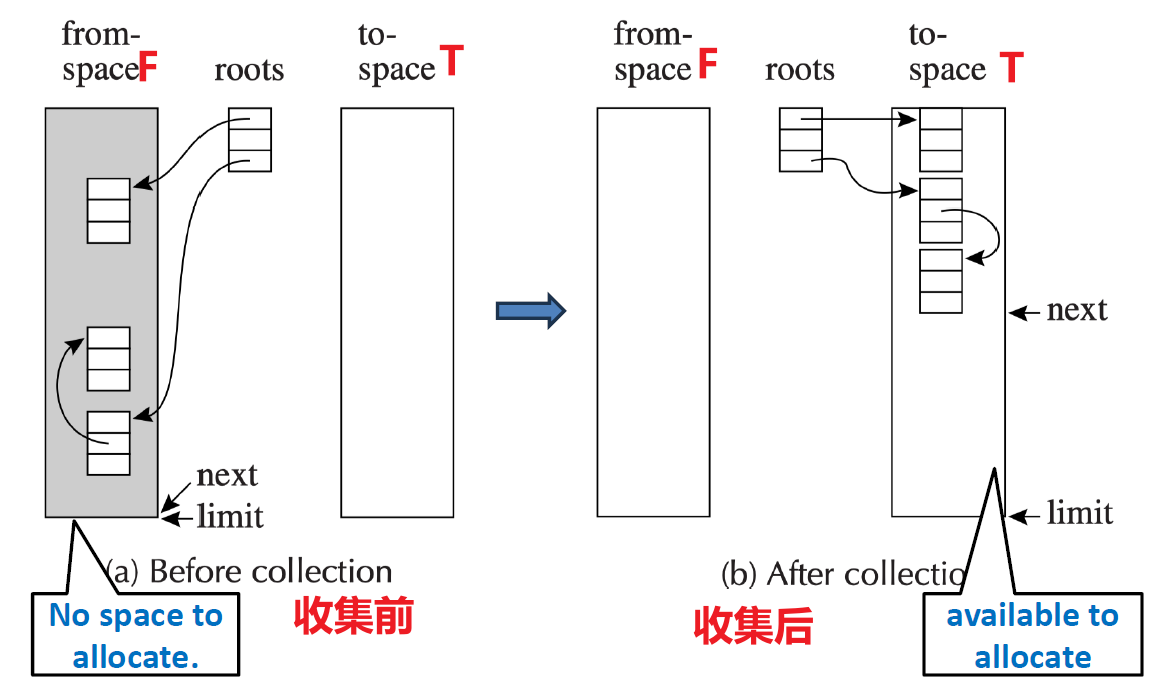
**解决”环问题”办法**:(1)简单地使用数据结构时显式地解开所有的环,但是难度较大(2)将标记清理和引用计数相结合.

**每当将p存储到x.fi中（即x.fi = p）时**，p的引用计数会增加，而x.fi先前指向的对象的引用计数会减少。**如果某个记录r的引用计数降至零**，r会被放入空闲列表中，并且r所指向的所有其他记录的引用计数都会减少。

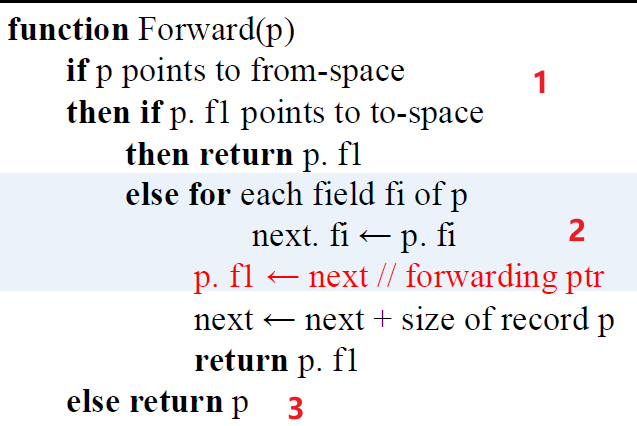
**Copying Collection**

使用两个堆，**from-space**：程序使用的堆，**to-space**：在垃圾回收之前不使用的堆

当from-space用尽时，遍历from-space，并将所有可达的节点复制到to-space，垃圾会被留在from-space，当next达到限制时，交换from-space和to-space的角色

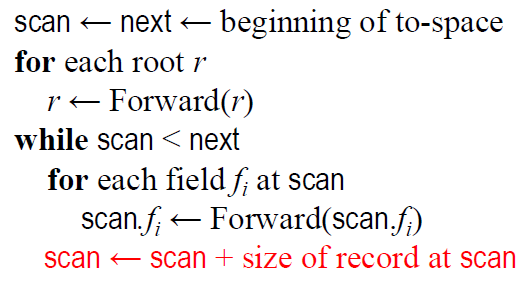


**交换算法流程**：①收集初始化：初始化指针next指向to-space的开始.每当from-space发现一个可到达记录,便把它复制到to-space的next所指位置,同时式next增加该记录的大小②**转递(forwarding)**：使一个指向from-space的指针p转而指向to-space，分3种情况(1)p指向的使from-space中已经复制的记录,则p.fi是指明副本在何处的特殊的转递指针(2)p指向from-space中一个尚未复制过的记录,则将它复制到next所指的位置;同时将转递指针赋给p.f1(3)p不是指针或者指向from-space以外的指针,对p不做任何事情. 伪代码如下：

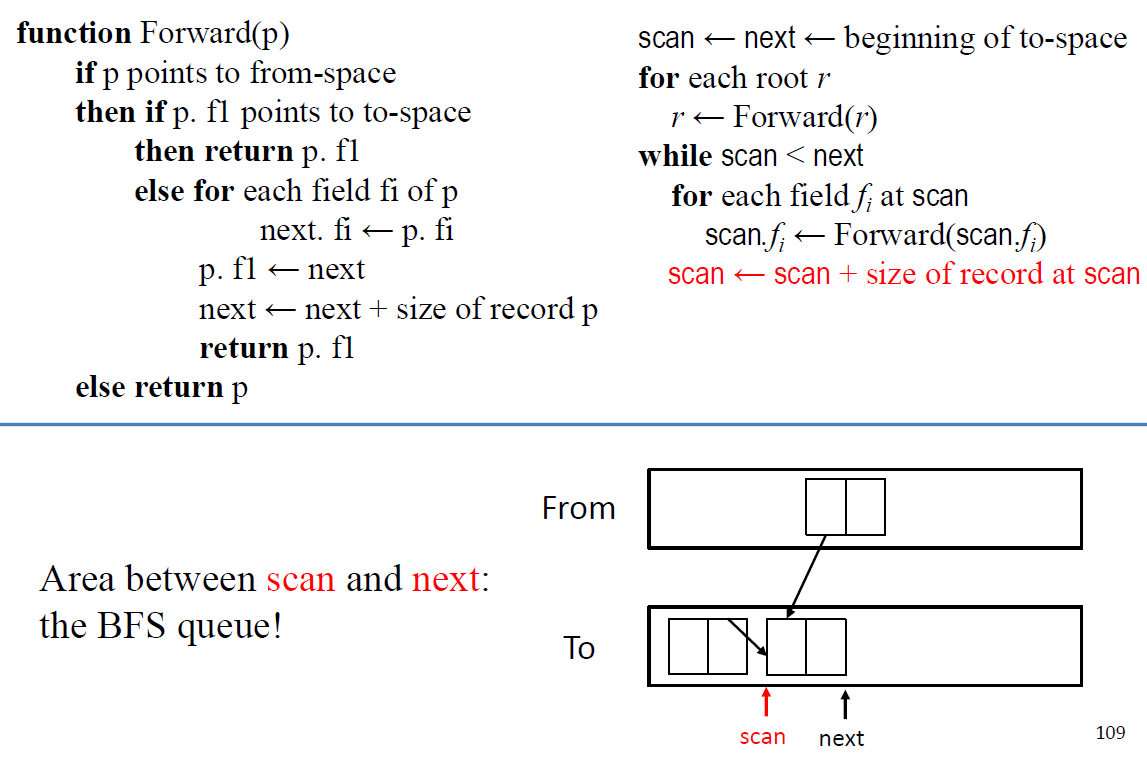


**Cheney’s Algorithm**

**使用BFS的方法**，在 Cheney 算法中，垃圾回收器首先将指针 scan 和next 初始化为指向To-Space的开始位置。然后对于每个程序变量（Root） r ，都需要通过转发操作将其指向的对象修改为对应的到空间中的记录。这样可以确保从空间中所有可达的记录都会被复制到到空间中，并得到有效对象的转发



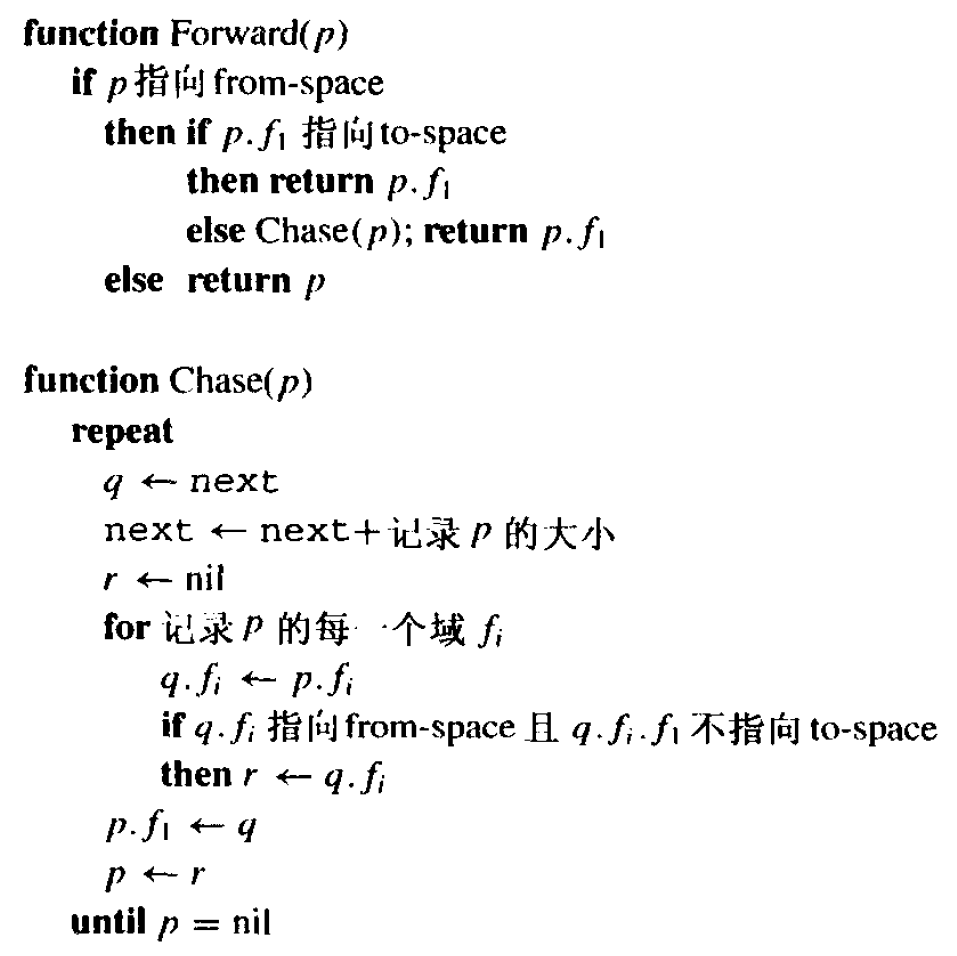
通过使用广度优先搜索（这样可以从浅层到深层遍历，一定程度上避免问题），垃圾回收器可以按顺序遍历所有可达的记录，并将其复制到新的位置。具体来说，对于到空间中每个位置的scan操作，需要遍历该位置上记录的所有字段，并将这些字段指向的对象进行转发。遍历完所有字段后，将scan的值增加该记录所占用的空间大小，以便继续遍历下一个可达的位置。如此往复，直到遍历完所有可达的记录为止



**算法优点**:不需要外部栈和逆转指针;使用scan和next之间的区间作为BFS队列;实现简单.**算法缺点**:引用局部性差(没有把根相关的引用放到一起而是错开了,相邻的是不同变量引用)

**算法代价**:每一次回收个字.摊还代价是每个分配字为条指令.

**混合式算法(半深度优先搜索)**:缓解Cheney空间局部性不好的缺点.时间复杂度也是摊还代价是每个分配字为条指令.当H远超R时代价接近于零,即没有固有的垃圾收集代价的下界,空间时间代价都很大.



整个Copying Collection的**优点**： 简单：不需要堆栈或指针反转；运行时间与存活对象的数量成正比；保持空闲空间的连续性；自动压缩消除碎片；为许多后续算法奠定基础。**缺点**：一半的内存被浪费；局部性差（至少对于Cheney算法来说）；需要精确的类型信息（是否是指针）

编译器接口（编译器与垃圾收集交互）

**快速分配(for Copying Collection)**:复制收集的分配空间使得分配的空间是连续的空间；区域的末端是**limit**, **next**指向下一个空闲单元.

分配大小为N的记录的步骤如下:①调用储存分配函数②测试next+N<limit是否成立(不成立则调用垃圾收集器)③将next复制到result④清除M[next]-M[next+N-1]⑤next:=next+N⑥从分配函数返回.A.将result传送到计算上有用的某个地方B.将要用到的值储存到该记录.

其中①⑥可以被内联拓展**(inline expanding**)消除;③可以与A结合被消除;④可以与B结合被消除;②⑤不可以被消除,但如果在同一基本块内有多个分配,则可以在多个分配间公用比较操作和自增操作,把next和limit放到寄存器里②⑤只需要**3**条指令.综上,分配记录的指令开销.可以被减少到**4**条指令.

**数据布局(data layout)描述**:①收集器可以对任意类型记录进行操作②简单办法是让每个对象的第一个字指向特殊的类型(或类)的描述字记录(descripter)：包含对象的总大小以及每一个指针域的位置

**对于静态类型语**言，如Tiger或Pascal，或面向对象语言，如Java：让每个对象的第一个字指向一个特殊的类型或类描述符记录。**类型或类描述符包括**：对象的总大小，每个指针字段的位置

**指针映射(pointer map)**：①编译器必须能给收集器标出存放指针的临时变量和局部变量(寄存器中or活动记录)②由于每条指令都可能使活跃临时变量集合发生改变,故指针映像在程序的每一点都是不同的.③因此一个较简单的办法使仅在那些可以开始新的垃圾收集的点才描述指针映像；这些点才是alloc函数的调用点；每个函数的调用点也必须描述指针映像.④指针映像最好用返回地址作为键值：(1)为了找到所有根,收集器从栈顶向下扫描(2)每一个返回地址的键值对应一个指针映像的登记项,登记项描述下一个栈帧(3)每个栈帧内收集器从站真的指针开始标记(4)callee-save的寄存器需要特殊处理：f调用g,g调用h,h知道自己保存了callee-save的寄存器但是不知道那些使指针.所以g的指针影响必须指出在调h时他的callee-save的寄存器那些是指针那些从f继承.

**导出指针(derived pointer):**①对于表达式a[i-2000],内部被编译器计算成为M[a-2000+i]:对应t1:=a-2000;t2:=t1+i;t3:=M[t2]

②为避免重复计算,会把t1:=a-2000;提到循环外计算.③若循环中包含alloc且收集器在t1活跃时开始工作,那么收集器可能会被搞糊涂④此处定义t1是由基指针(base)a导出的(derived).pointer map必须标识每一个导出指针(derived pointer)并指出其导出的基指针.⑤就是收集器在把a重定位到地址a’时,也必须调整t1到t1+a’-a;把导出的指针和其基类相互绑定:只要基类活跃,导出指针也必须活跃.一个导出的指针将隐式地保持其基指针活跃.

空闲表数组:使用简单freelist的效率低,为了找到空间大小合适的记录需要找到很深.使用空闲表组成的数组,freelist[i]中存放大小为i的空闲区域.当要分配大小为i的记录时,从freelist[i]的表头取一个即可.清扫垃圾时可以把大小为j的插入到freelist[j]中.若想从freelist[i]的空表中分配,可以从freelist[j](j>i)抢夺一个较大的记录,然后把剩余的(j-i)插入到freelist[j-i].

6.碎片(fragment):①外部碎片:想分配一个n大小的空间,但是空闲空间均小于n②内部碎片:实际使用大小为n的分配了大小为K的空间(K>n),未使用的空间在记录内而不是空闲记录中

**Ch14 Object-Oriented Languages**

Object-Tiger: Extend the Tiger language with new declaration syntax to create classes

**Single-inheritance** of data fields. Static method VS Dynamic method

Multiple-inheritance of data fields. **Global graphic-coloring** for fields offsets in descriptors, hashing

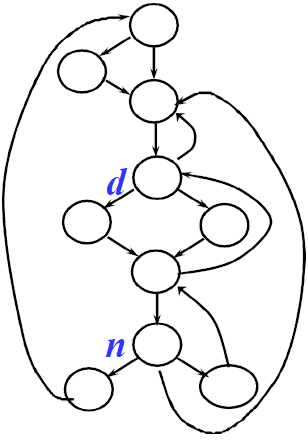
Facilities for type testing and safe casting. Using a **display** of parent classes, type coercions

Varieties of **privacy and protection**. statically enforced by compile-time type-checking for class-based languages

Classless languages. Type-checking dynamic instead of static, Objects are **created by cloning**, similar to the class-based language with **multiple inheritance and dynamic linking**

Optimizing Object-oriented programs. do **global program analysis**, the dynamic method call replaced by a static function call

**CH18 Loop Optimizations**

• 低级优化：在单个循环中移动代码，比如loop invariant code motion, strength reduction, loop unrolling

• 高级优化:重构循环，通常会影响多个循环，比如loop fusion, loop interchange, loop tiling

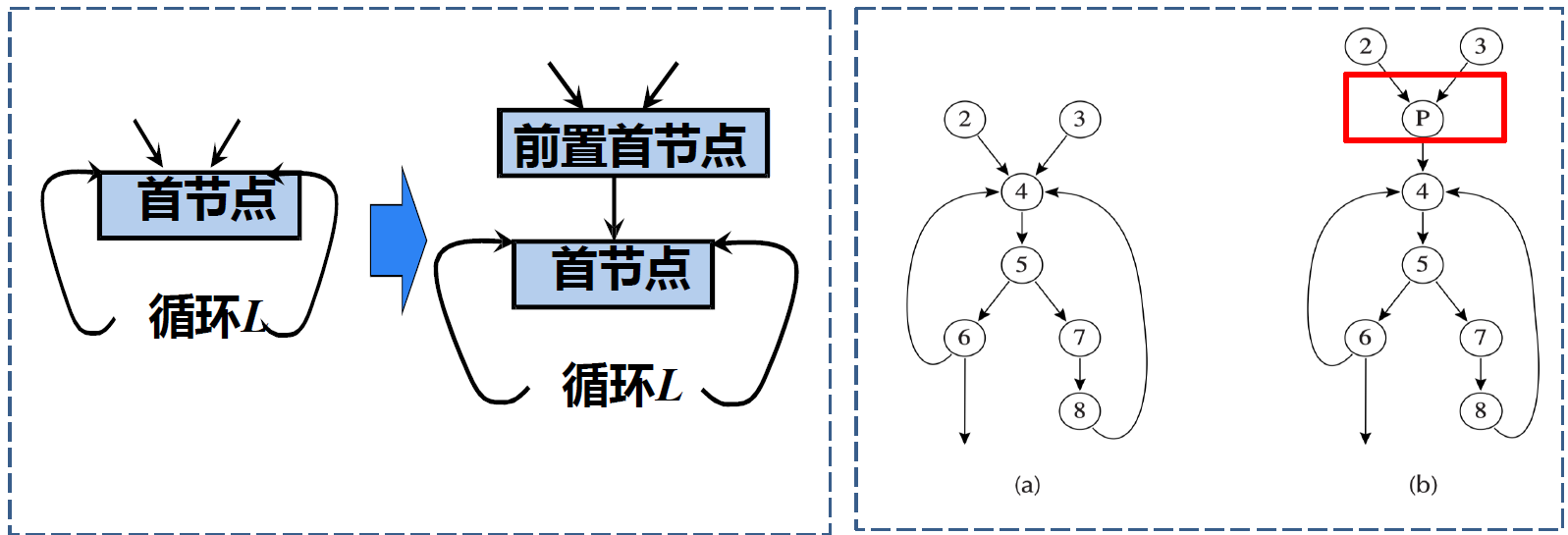
**Dominator**: d是n的支配结点(d dom n):从流图的入口结点到结点n的每条路径都经过节点d

**直接支配结点(immediate dominator**)，从入口结点到达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

在**Dom tree**中, 每个结点只支配它和它的后代结点

**自然循环**满足以下性质:1.有唯一的入口结点，称为首结点(header)。首结点支配循环中的所有结点 2.循环中至少有一条返回首结点的路径

Loop Preheader (前置首结点) 前置首结点的唯一后继是首结点: 1. 原来从循环L外到达L首结点的边都改成进入前置首结点; 2. 从循环L里面到达首结点的边不变



some expressions evaluated in a loop never

change; they are **loop invariant循环不变量**

Step1: find invariant computations in loop

– invariant: computes same result each time evaluated

Step 2: move them outside loop

–to top if used within loop: **code hoisting上升**

–to bottom if used after loop: **code sinking下沉**

assignment **x := v1 op v2** is invariant for a loop if for each operand v1 and v2 either？

1. The operand is constant, or

2. All of the definitions that reach the assignment are outside the loop, or

3. Only one definition reaches the assignment and it is a loop invariant

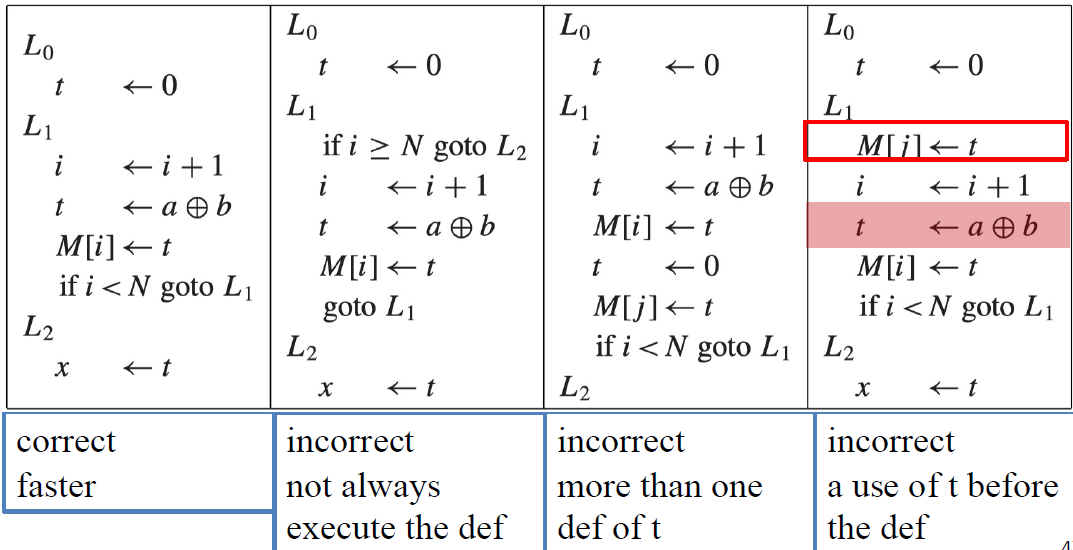
代码是循环不变的，并不意味着我们可以移动它！（应该保留语义。The criteria for hoisting

d : t ← a ⊕ b to the end of the loop preheader:

1. d dominates all loop exits where t is live-out

2. and there is only one definition of t in the loop

3. and t is not live-out of the loop preheader (that is, t is not live before the loop)



**历年卷概念**

1. There is only one parse tree for a string of an unambiguous grammar. True(已消除二义性)

2. LL(1)没有reduce,有match generate accept

3. Left recursion is commonly used to make operations left associative

4. LR(1) parser can detect errors earlier than LR(0) True(外面的早一些发现错误)

5. Yacc use **LALR(1)**

6. The symbol table will not carry the data type

7. 如果画NFA没有要求，可以简化画，但是注意像(01)\*这样的其实是两个状态间的来回

8. **A grammar is ambiguous if it has 2 different derivations or 2 different parse trees for a sentence** **(False)**

9. If a grammar is LR(1) but not LALR(1),There are not shift-reduce conflicts in its parsing table of LALR(1) **(True,只有reduce-reduce的问题)**

10. LR(1) item[A->a.Br,a],FOLLOW(B)={ra}

11. A LR(1) parser cannot parse any left-recursive CFG without ambiguity **False**

12. There is only one parse tree for the string of an ambiguous grammar **True**

13. Finding the next handle is the man task of LR parser **True**

14. The parse tree will not completely reflect the derivation steps for a string **False**

15.four parameters passing methods:

(1)值传递 | Pass by Value

就是 C 语言中的参数传递方式。

(2)引用传递 | Pass by Reference

传递变量的引用，即传递其在存储空间中的位置。在函数中所做的一切更改都会作用于这个变量本身。

(3)值结果传递 | Pass by Value-Result

将实参从左到右逐个复制到形参中，在函数运行结束后再逐个将其复制回原来的位置。

(4)名字传递 | Pass by Name

名字传递的思想是：直到函数真正使用了某个参数时才对其赋值，因此也称为 延迟赋值 (delayed evaluation)。等价的做法是将函数在调用的原位进行展开。

int i = 0;

void p(int x, int y)

{ x += 1;i += 1;y += 1;}

main()

{

int a[2]={1,1}; p(a[i], a[i]);

printf("%d %d\n",a[0],a[1]);

}

(1)的结果是1 1，不变

(2)的结果是3 1，相当于两次a[0]+1

(3)的结果是2 1，相当于先取p(a0,a0)两个1到里面，把x=2放入a0,再把y=2放入a0

(4)的结果是2 2，相当于是a[i] += 1; i += 1; a[i] += 1;

16. The output of the scanner is token

17. Lex is a tool that is a lexical analyzer generator

18. Static variables is not commonly found in the stack frame

19. **caller operation push the return address in the stack frame**

20. 终结符$不应该出现在LR里

21. In the production B->aAr,e不在follow(a)里

22. Scopes of the variables are intercrossed sometimes **Flase**

23. Yacc can not use ambiguous grammars **False**

24. The best choice of data structure of the symbol table is HASH table

25. LR(0) is the least powerful

26. We do error recovery with (1)add error production(2)modify the parsing tables(3)modify the parsing engine **but not** eliminate the conflict

27. **sp pointer** is commonly found in a stack frame(activation record)

28. The motivation to divide the complier is to **provide portability of costomer**

29.Heap management is related to mark and sweep;memory compaction;display but not have **stop-and-copy**

30.历年题里，我认为compute the argument should be done by the caller

31.bookkeeping-control link

32.delayed used in pass by name

33. semantic analysis output an annotated tree

34. use access link to retrieve nonlocal data declared within another procedure(stack-based environment with local procedure)

35.yacc keep a value stack parallel to parsing stack

36. a traverse order of the dependency graph must be undirected and cylic

37. Intermediate code can be very high level or it can closely resemble target code. **True**

38. The general organization of runtime storage will contain the code area, the global/static area, a stack, a free space as well as a heap. **True**

39. When a procedure is called, an activation record P is generated on stack. There may be activation records on stack corresponding to sibling children nodes of P in the activation tree-True

40. 3-address codes have 3 fields **True**

41. The activation record kept in the stack is always directly pointed by frame pointer (fp) **F**