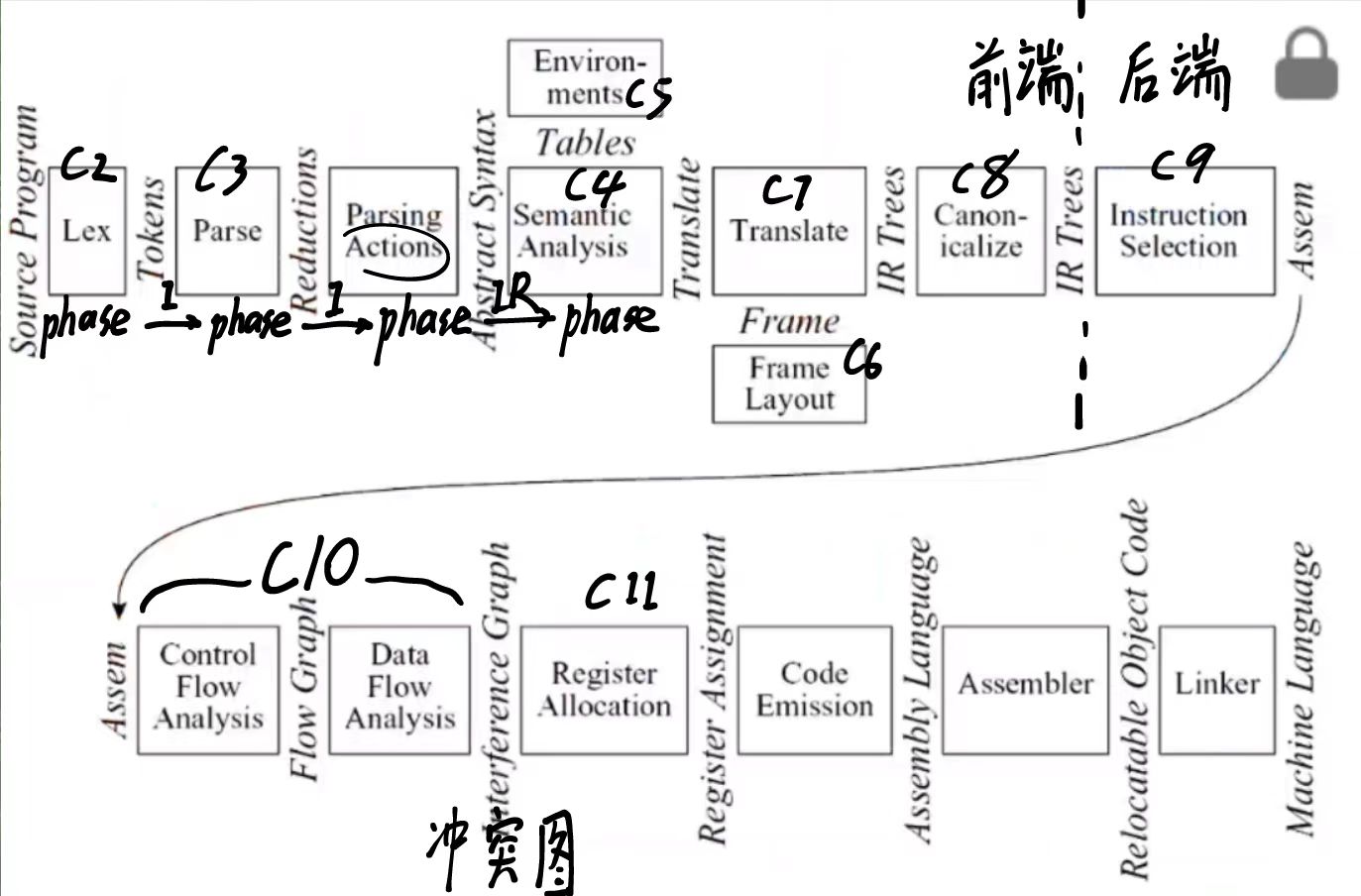
****

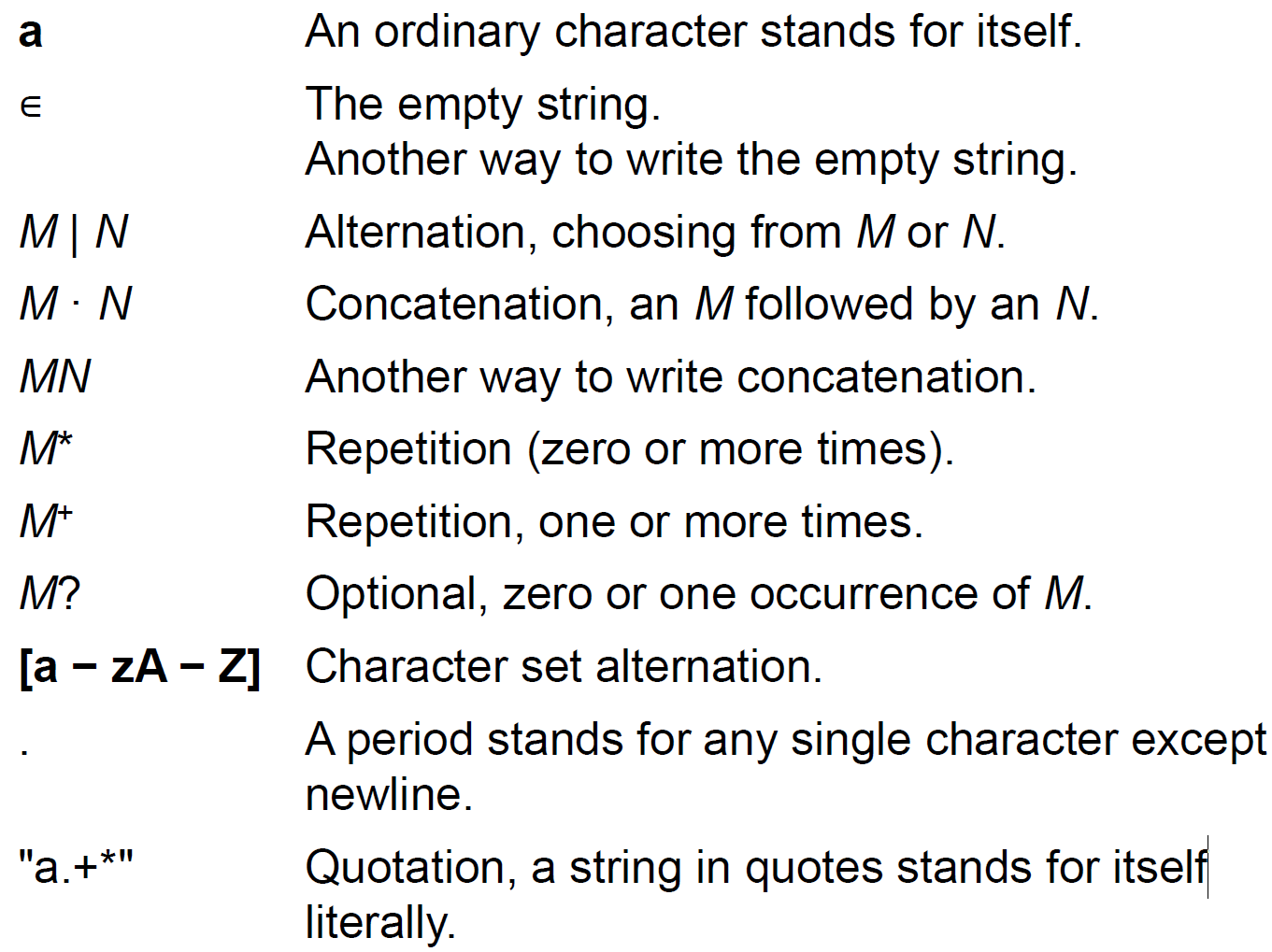
**CH2 词法分析**

**2.1 扫描处理**

1.某些token只有一个词义:保留字；某些token有无限多个语义:identifier；non-token 如 # // CR-LF

**2.2正则表达式**

1.一些notation



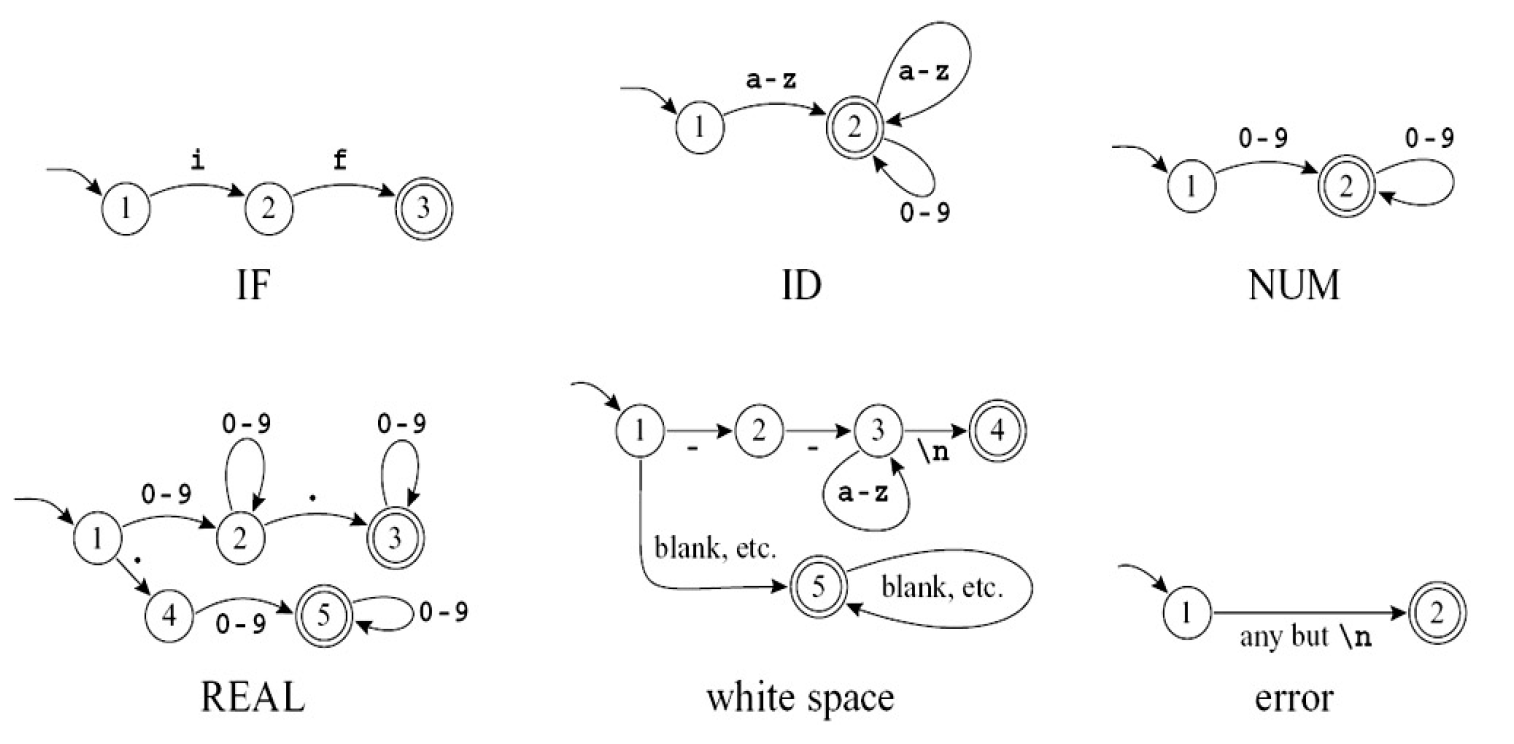
2.RE匹配:最长匹配、规则优先级（保留字优先）

**2.3有限自动机**

1.DFA:M=(K,,,S,F)

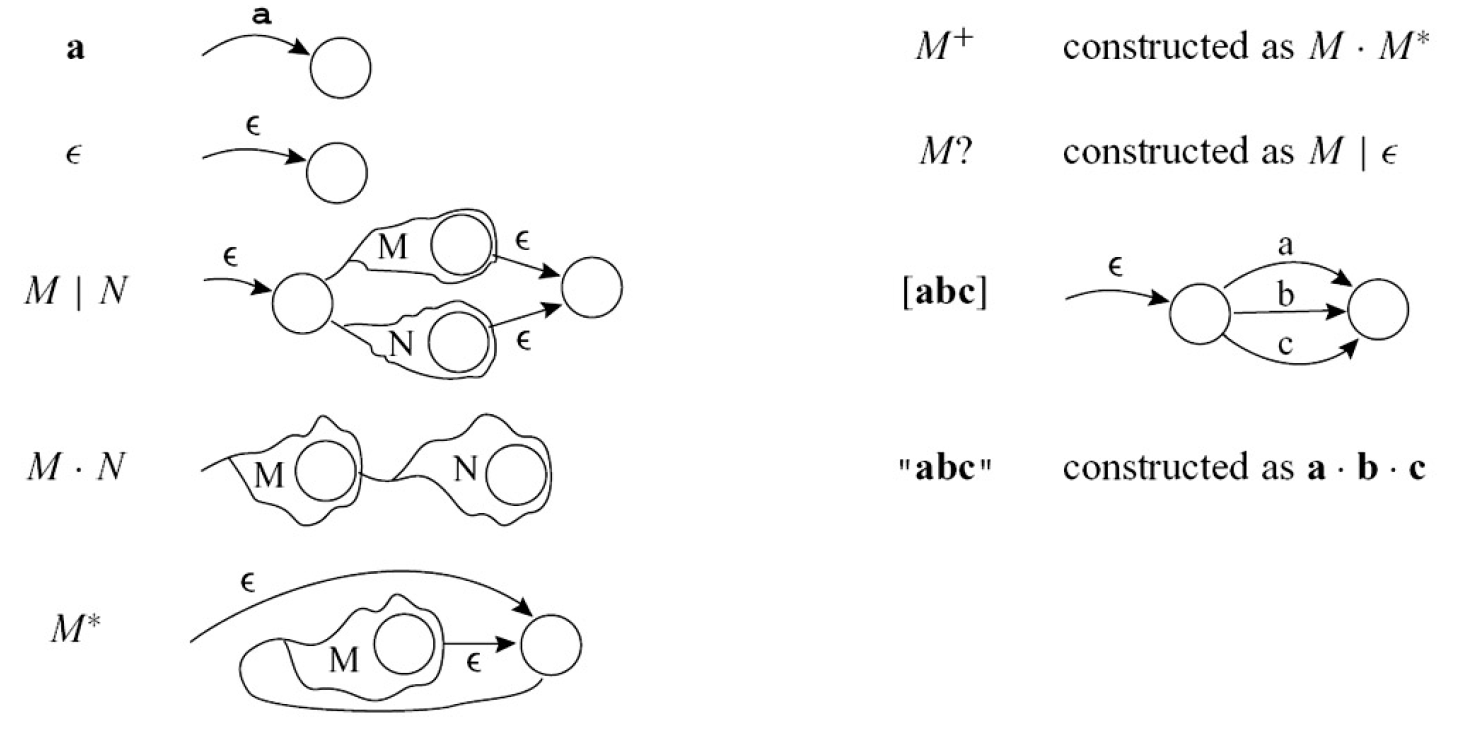
2.NFA:M=(K,,,S,F)

一些 token的NFA图:（NFA错误状态省略）

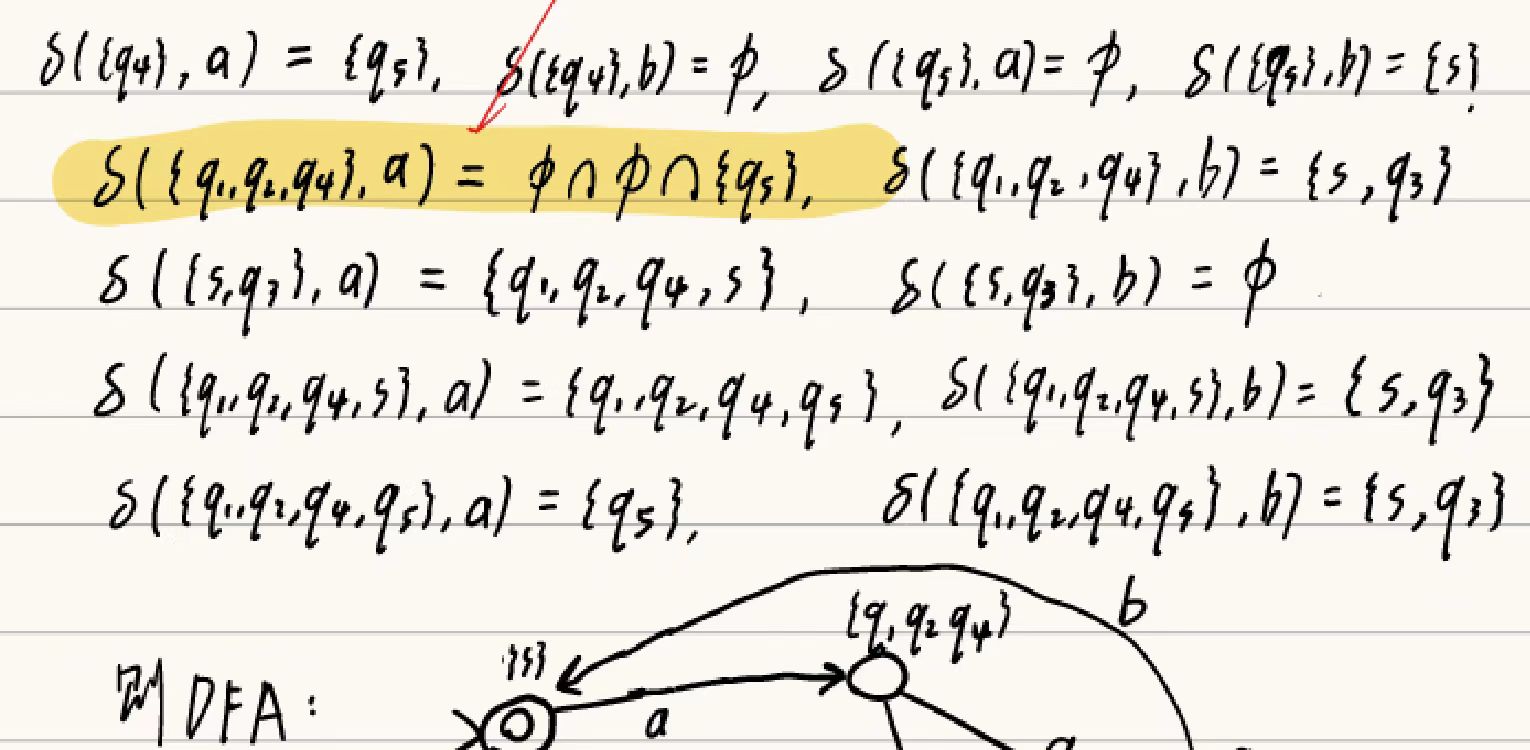


**2.4正则表达式到DFA**

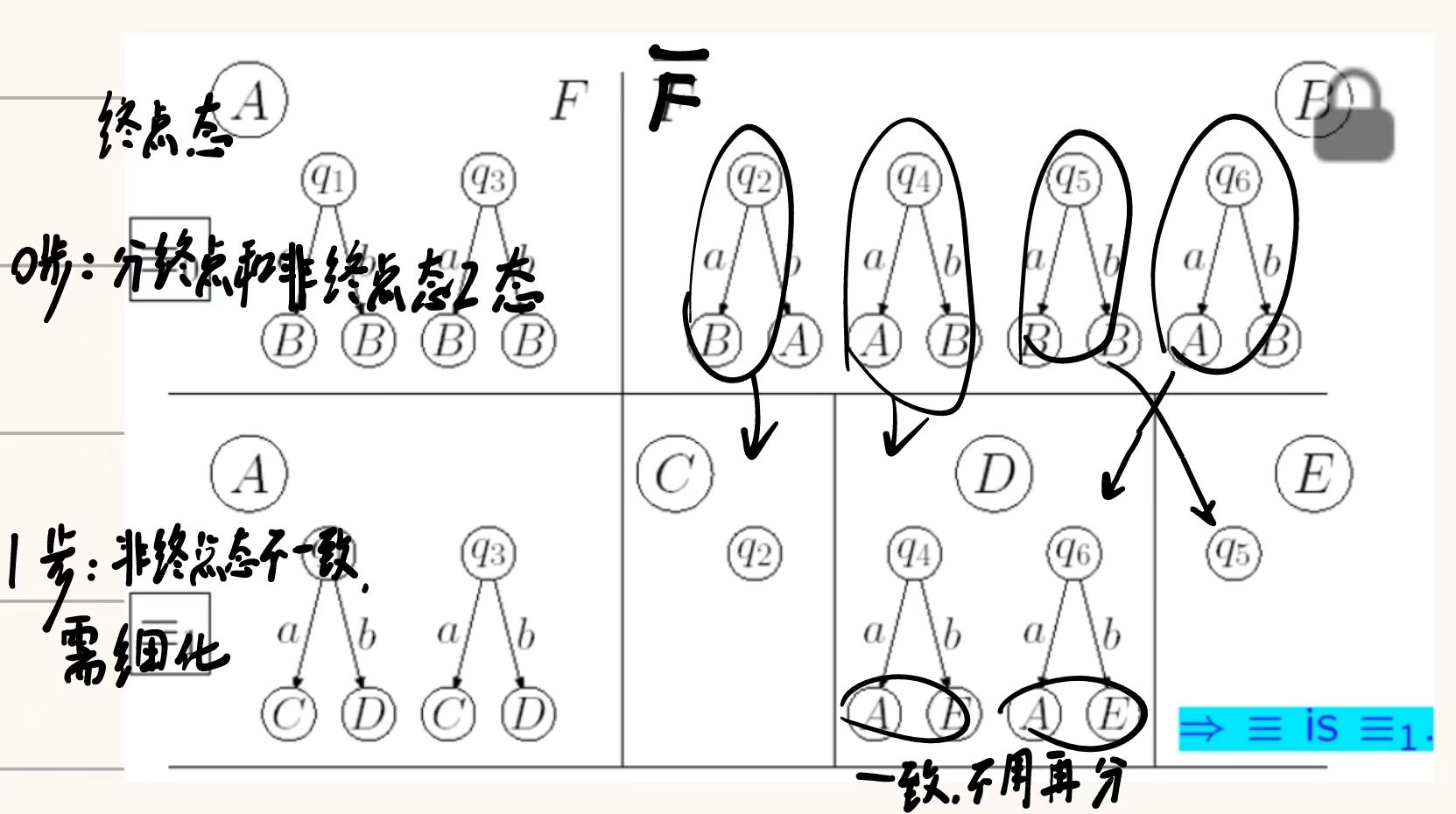
**1.RE to NFA:** 各个原操作

****

**2. NFA to DFA:**首先列出S的 闭包 E(S)作为新S’；然后计算在每个状态元素下在**各个字符**上的转移的闭包作为新状态（若未出现过）；包含原F的所有{...,F}都是同一接受状态 F’。



**3.DFA状态数最小化:**

****

**CH3 上下文无关文法分析**

**3.1 CFG**

1.左递归文法:L->La

2.右递归:L->aL

**3.2具体/抽象语法树**

1.**CST(concrete sytax tree / parse tree)**是一个作了**语法标记**的树,内部节点是非终结符,树叶是终结符。含有多余标点符号，带有语义解释；

2.最左推导和前序遍历对应,最右推导后序遍历

3.**AST(abstract syntax tree)去除了终结符和非终结符、标点符号语法，每个节点有数据结构并完成语义动作。抽象文法可能有二义性，但已经建树无所谓**。一般用左孩子右兄弟

**3.3 Ambiguity二义性**

**1.定义:**带有两个不同的分析树的串的文法

**2.解决方法 Adhoc disambiguating rule** ,如 precedence优先级

associativity 结合率

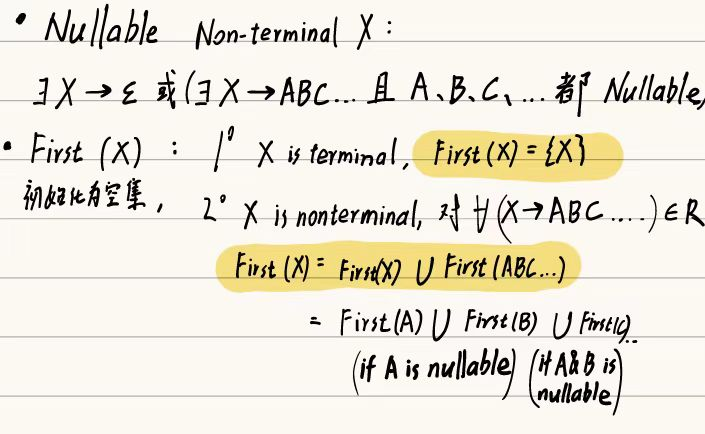
3.在语法树中,越接近根,优先级越低；左递归导致左结合,右递归会右结合

**Predictive Parsing自顶向下分析: Left-to-right Leftmost-derivative**

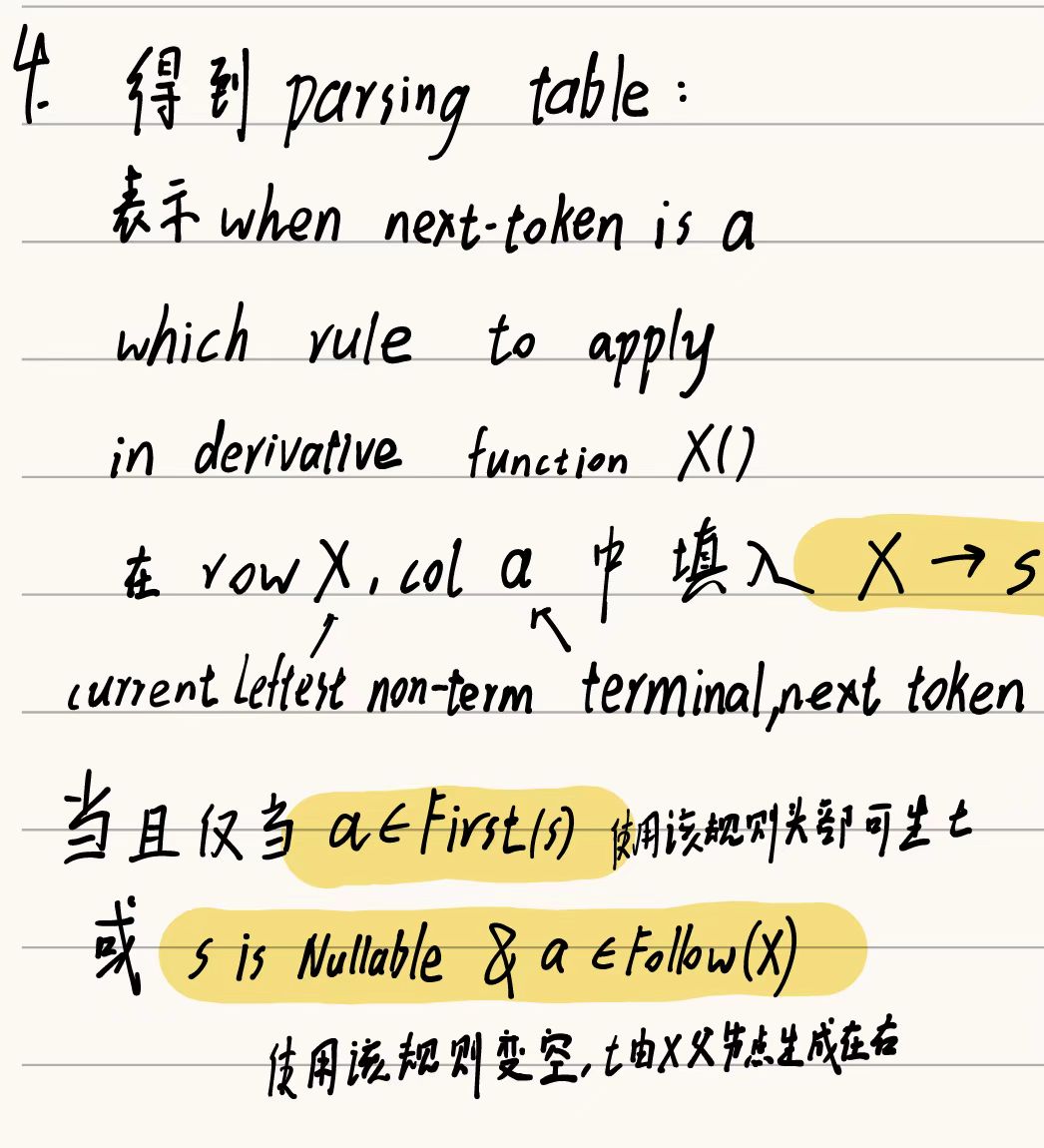
**k symbol lookahead**

**递归下降:**1.将一个非终结符A的文法规则看作将识别A的一个过程的定义.递归下降需要使用EBNF；将可选[]翻译成if,将重复{}翻译成while循环

**LL(1)1.动作:**①生成**(generate)**,利用文法将栈顶的N替换成串,串反向进栈 ②**匹配(match/eat)**:将栈顶的记号和下一个输入记号匹配 ③**接受(accept)**:接受字符串 ④**错误(error)**

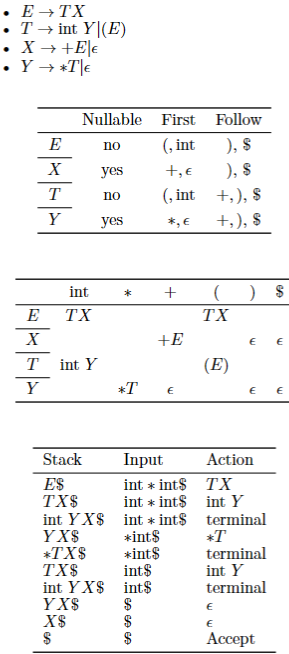


****

****

5. 依据 grammar 构造的 parsing table 每格至多一条 production，无二义性，才能被称为 LL(1) grammar.

用栈实现：



6. **LL(1)减少冲突必做:消除左递归**

left recursion removal

E E+T | T 改成

E TE’

E’ +TE’|

或者：

改成

**提取左因子**

left factoring.

改成

2.定理:A non-terminal A is nullable if and only if First(A) contains

**自底向上分析: LR(k)**

**Yacc基于LALR(1),它还用Value Stack 完成语义动作,需要父/左兄弟token 的inherited attributes (不止 CFG 在合成时子传父的 synthesized attr），通过先序遍历**

**1.动作:** ①shift,将T从输入开头移到栈顶 ②reduce使用产生式A→将栈顶的规约成A ③accept分析栈为$,输入栈为空时④error ⑤**转移(goto):** 分析栈吃掉non-terminal

**2.S’:**新的开始符号,新增S’→S$规则来引入栈底 $

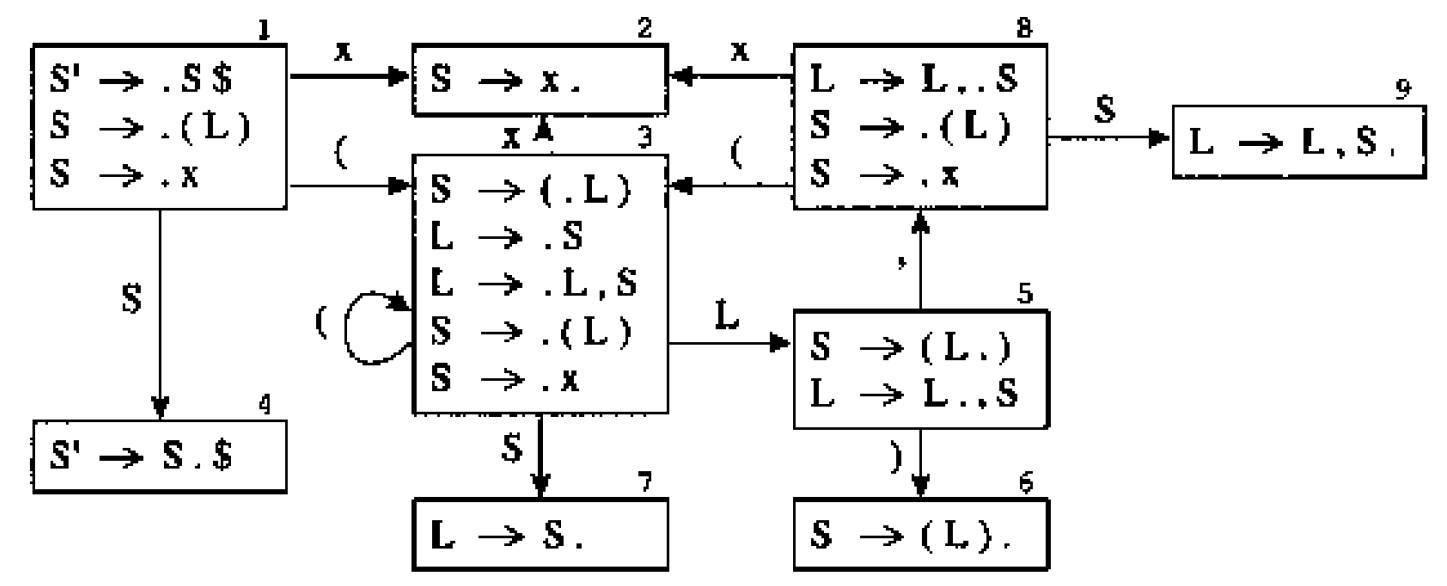
**3. 句柄 handle**： 最左可规约串，可reduce; **viable prefix 活前缀：**句柄的任何前缀，不可 reduce。

**4.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S->.AXX，将A->.XX也写到这一状态中。

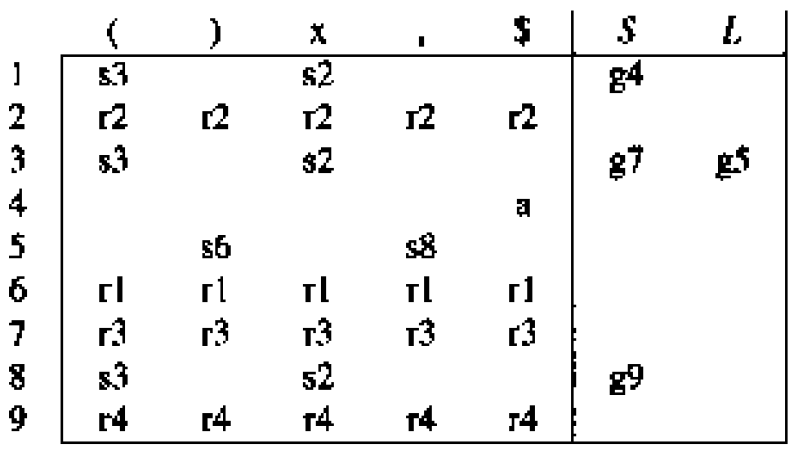
如下文法构造LR(0)



状态图如下:



由上面的状态图构造表如下，注意 reduce 用规则标号填满一行，shift 读入终结符并用状态标号



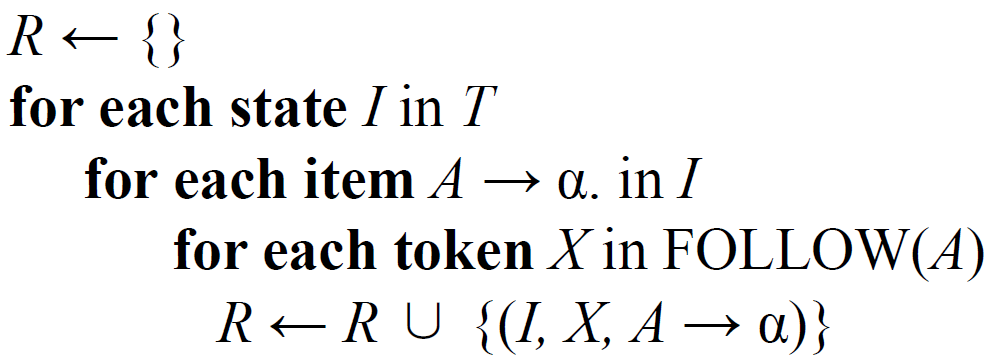
使用：input token string，从 初始状态1 开始，遇到 non-terminal 做状态转移

**5.s-r conflict**:一格同时出现了shift 和reduce,即构成s-r冲突

**6.r-r conflict:**一格同时出现两个及以上的reduce

**SLR(1):**

**1.SLR算法定义:** shift/goto 一致，但Reduce 设 A 是规约后非终结符，只放 Follow（A）而非一整行 ：



2.SLR中的两种冲突:s-r冲突:优先shift；但该CFG已经不是 SLR(1)。

r-r冲突：无解

**LR(1) and LALR(1)**

1.LR(1) items:[]前面是LR(0)项,后面是lookahead token

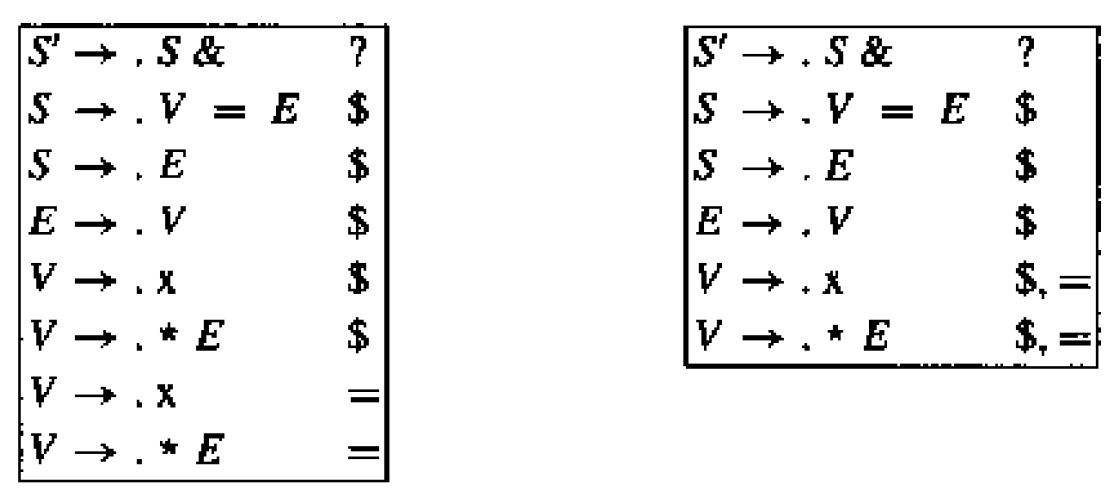
2.LR(1)的起始状态：[S’→.S,$]的closure。

该 closure 在

S->.Aβz,$ 时，

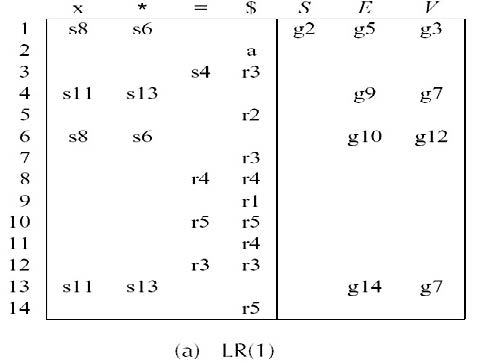
将所有 A->.XX,ti

ti∈First(βz) 搜集起来



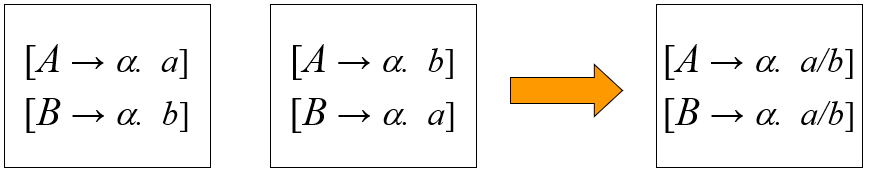
3.LR(1)Parsing Table的构造: shift/goto 与 LR(0) 一致，reduce 时只填入 lookahead symbol 的一格。

6. LR(1)分析表示例

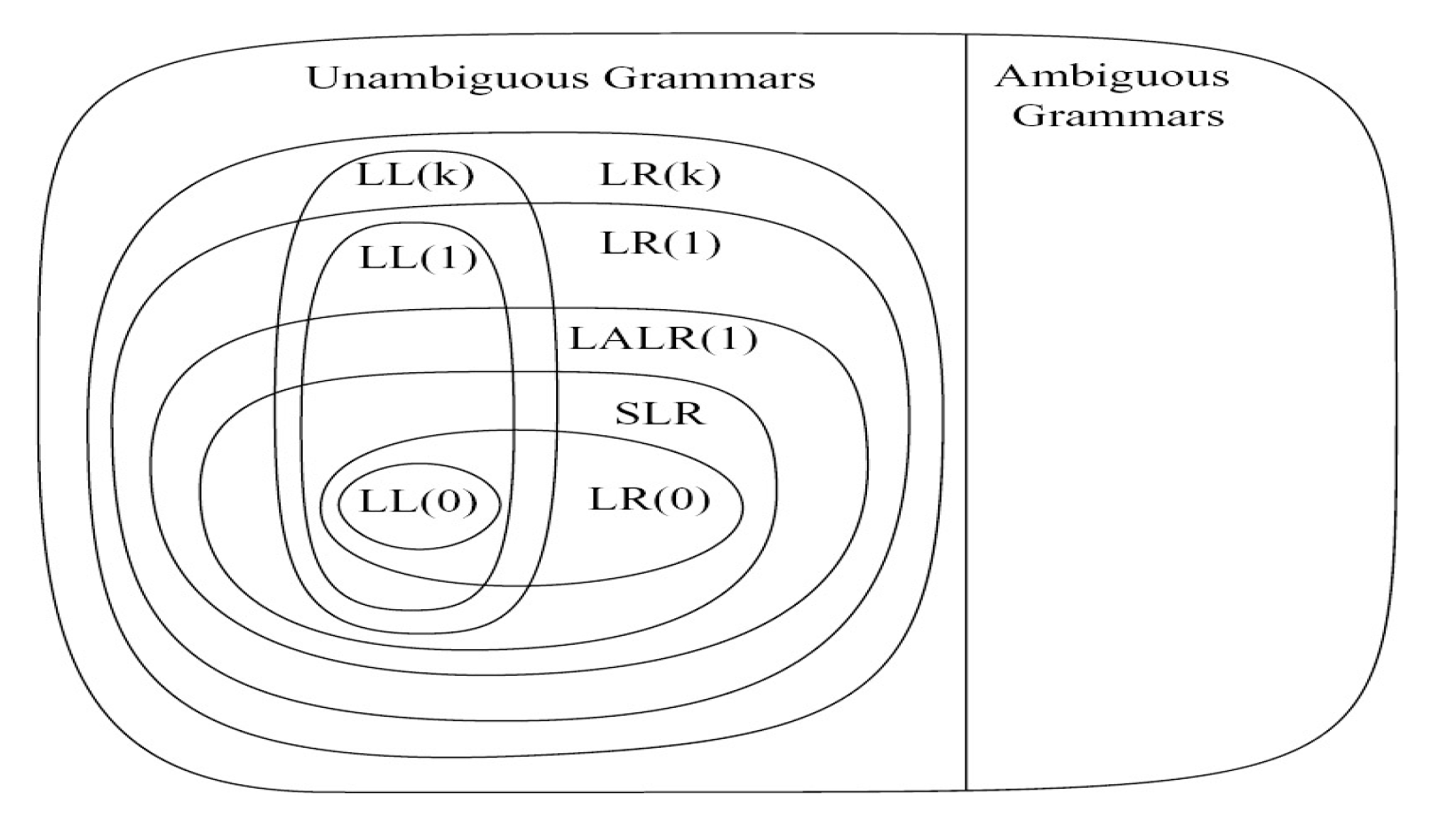


8.LALR(1)合并items 除 lookahead sets 之外完全一致的state，reduce 又可以填多格。

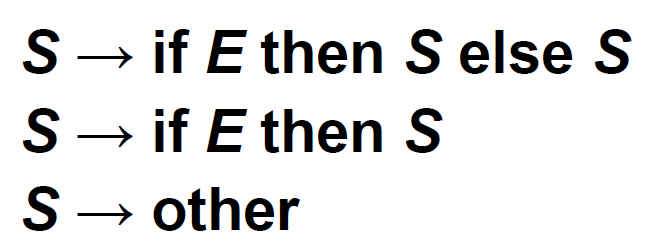
10.如果文法是LR(1),那么LALR(1)中必然没有s-r冲突,但是可能有r-r冲突.



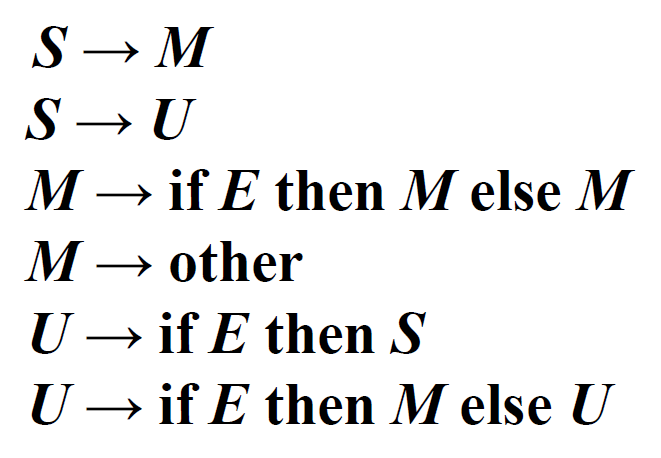
12. 各类文法的层次如下:



13.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 必定含else;U至少一个 if 没有 else



RE 和 CFG 最大区别：RE用 \* 表示简单重复，CFG 用文法表示递归生成；

Tiger 语法：

let

x:int //delcare

function f(x:int):int={..}

in

x = 0; //Expr

end

构成一个基本块

Local Error repair: use terminal `error` and synchronize token `;` to skip + resume.

Global Error repair:

bucke-fisher 在错误点之前 K 个单词从后往前做删除 + 插入 + 替换尝试 K+2\*KN

**CH5 语义分析**

编译器完成的是static semantic analysis

**5.1 符号表(symbol table)**

1.**Binding:** 把类型、值等信息绑定到一个identifier.{g→string, a→int}

**符号表/环境:**一些绑定的集合

2.局部变量都有一个**作用域(scope)**, 变量仅在自己的作用域中可见.

**3. Type Equal:**

· Structural equiv:

ts/c，只要有same struct

· Name equiv：

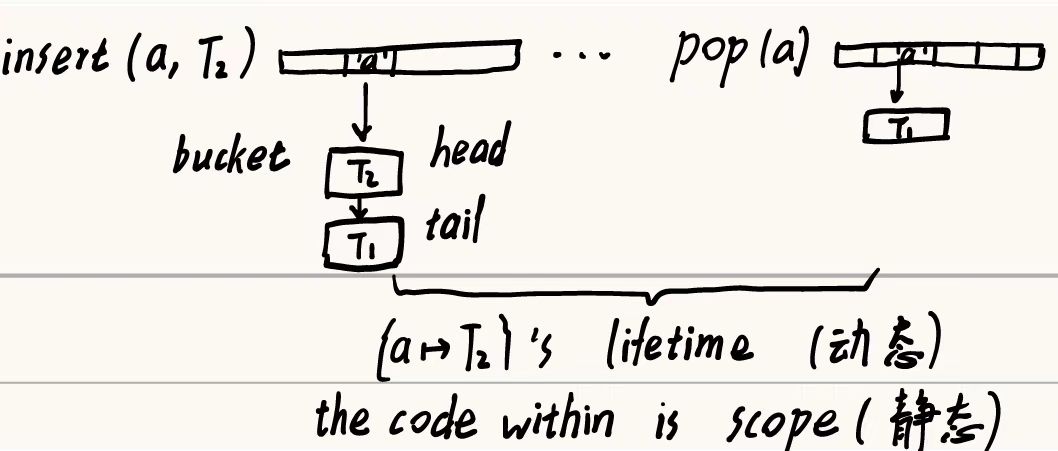
c++，必须严格名称

· Declare equiv：

tiger，t2=t1可alias

**注意: 在C/C++以及Java中,变量的作用域都不可交叉(scopes of vars cannot be intercrossed)。但 jave 定义 global range，允许函数先调用后定义；c 不允许，渐进变大**

**5.2 命令式(imperative)**

1.散列表(bucket list) 

2.方法：冲突时push pop**以恢复环境. 但旧环境总不可见**

**5.3 函数式(functional)**

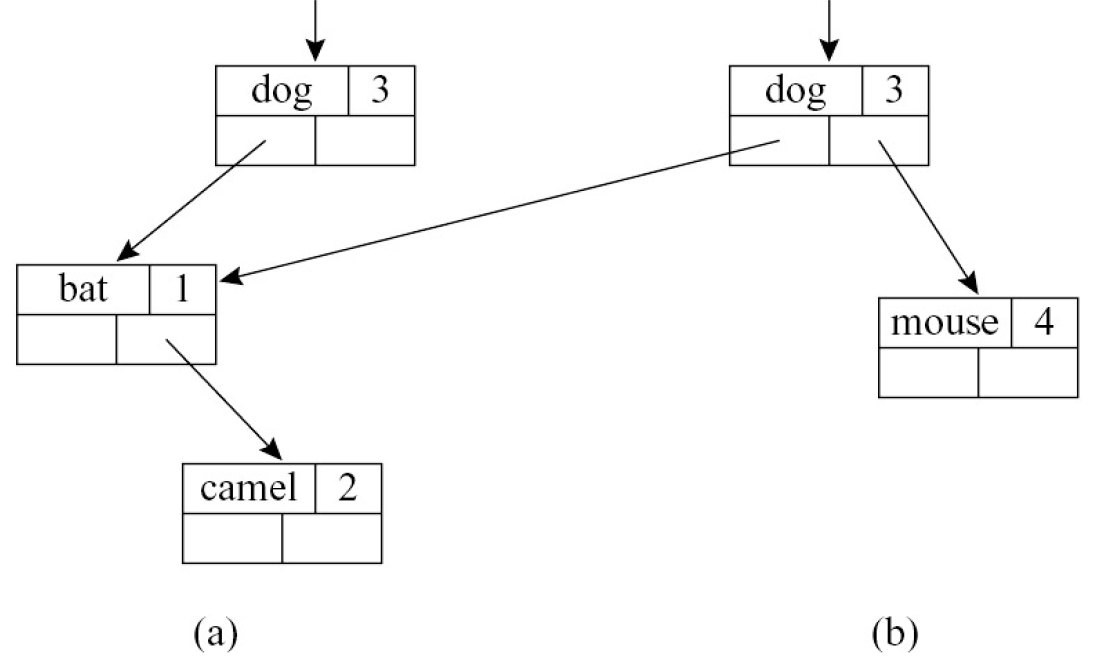
多哈希表：



1.**更省空间,但不适合大量symbol查找的**实现方法:二叉搜索树(BST)

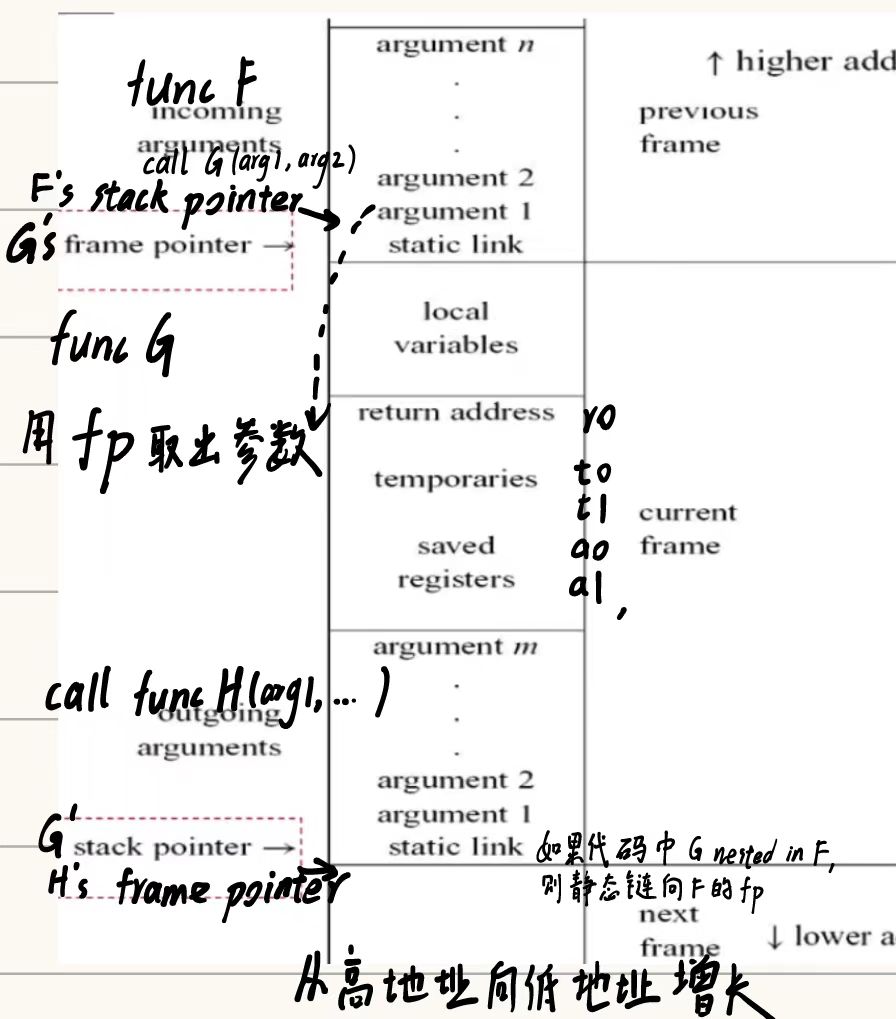
2.特点:不会直接操作原符号表,而是创建新的父BST节点;长效红黑树.

3. 例子:已知,现在添加新绑定(分隔符号表新作用域)mouse→4.需要创建复制d个父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:



**CH6 活动记录**

**6.1 栈帧,也叫活动记录**



1.**定义:**栈中存放函数的局部变量/参数/返回地址/临时变量的这片区域为该函数的活动记录(activation record)或栈帧(stack frame).

2.**帧指针(frame pointer):**指向当前帧的指针,一般是上一个sp;gcc -o2 不用;有些栈帧会分配一个寄存器存fp;fp=sp+size(frame),

帧指针的变化:

函数g调用f时

①sp指向g传给f的第一个参数

②fp旧值被保存到栈帧内,新的fp=sp

③f分配栈帧(sp-栈帧大小)

④f退出时把fp拷贝给sp,取回原先保存的fp即可.

从 g 退出:

6) 返回值拷贝至特殊寄存器

7) sp=fp (释放 g 的栈帧)

8) 从栈上取回旧的 fp 值到 fp 中

·Caller-save:a0-a7,$r0(zero)-$r1(ra),t0-t6 , 调用者如果用到则需要自己保存, 子函数可以任意修改.

•Callee-save: s0(fp), sp, s0-s7，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。

**3.参数传递**:计算机传参约定:前8个参数放在寄存器里传递,剩余在memory传递.

**寄存器传参,callee不用保护到栈帧的情况：**

•从变量生命周期入手: 如果寄存器对应的变量或参数在当前函数/调用子函数后不再使用, 子函数覆盖了自然也无妨

•过程间寄存器分配: 全局分析，每个函数使用不同的一组寄存器传参。

•叶过程 (Leaf Procedure): 如果某函数不调用任何其他过程, 不需要保存自己的参数。

• Register Windows: 每次调用函数时,尽可能利用尚未用到的寄存器,然后为子函数分配新的一套可用的寄存器 (SPARC 采用该策略)

**5.栈帧内变量**:一般来说局部/临时变量会放到寄存器中,以下情况需要将变量储存到栈帧内(memory):

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.逃逸变量(elapsed variable)**:**也就是脱离了当前 scope/无法确定变量有效的生命周期**

①地址实参指向的变量；

②被取地址；

③被内层嵌套函数访问的变量；

**8.Block structure**

为了使得内部函数访问非局部定义的外部变量, 有以下几种方法：

**1)静态链(static/access link)**:指向上一层fp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.

**2)嵌套层次显示表(display):** 一个**全局数组**，记录当前每个嵌套深度对应的栈帧地址(嵌套函数只能在外层栈帧注册后调用）。不需要链表可直接找外层 frame。

**3) λ提升(lambda shifting):**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

**附：**delayed evaluation，指函数pass by name之后动态获取实参，不做语义分析；语义分析得到的是 Anotated tree

**CH7 中间代码(IR code)**

**1.中间表示(intermediate represent):**抽象的机器语言,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(N\*M → N+M)

**2.基本概念:**

①前端(front end):词法分析|语法分析|语义分析|翻译成中间代码

**②后端(back end):**IR优化翻译成机器语言.

**7.1 中间表示树**

**1.中间语法树的表达式:**

①CONST(i):整型常数

②NAME(n):符号常数

③TEMP(t):临时变量

④BINOP(o, e1,e2):对操作数e1,e2的二元操作

⑤MEM(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

⑥CALL(f,[a1]):过程调用

⑦ESEQ(s,e): 先计算语句s形成副作用,然后计算e违该表达式的值

⑧MOVE(TEMP t, e): 计算e的值然后存到临时变量t中

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

⑩ JUMP(e, labs):跳转到e地址或者labs为label的地址

(11)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;

(12)SEQ(s1,s2):语句s1后面跟s2

(13)LABEL(n):定会一名字后的常数值为当前机器代码的地址.

**7.2 一般语句翻译**

• Ex 代表 expression

•Nx 代表无结果的 statement

• Cx 代表条件分支, 可能跳转到true label 或false

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).

**①简单变量:**

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.

**②追踪静态链:**

MEM(+(CONST Kn, MEM(+CONST Kn-1, … MEM(+(CONST K1, TEMP fp))…))); k1~kn-1是各个嵌套函数的静态链位移

**③数组变量下标:**

**a[i]表示为MEM(+(MEM(e), BINOP(MUL, I, CONST W))**

**7.3 函数翻译**

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.

**①入口处理函数包含**:

1)声明一个函数开始的伪指令

2)函数名字的标号定义

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

**②入口处理之后是:**函数的函数体

**③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**

**8.1 规范树(canonical tree)**

**1.定义:** 1) No SEQ or ESEQ

2) The parent of each CALL is either EXP(...) or MOVE(TEMP t, ...).

**2.为什么要规范?**

①CJUMP能够跳转到两个标号的任意一个,但机器指令条件为假时只能继续下一条

②ESEQ会使得子树的不同计算顺序产生不同结果，应使程序线性化。

③算数表达式操作数使用CALL，会有计算顺序不同的问题 f()+g()

④CALL的嵌套调用(作为另一个CALL的参数)会出问题,来不及caller-save，会覆盖内层返回值RV寄存器

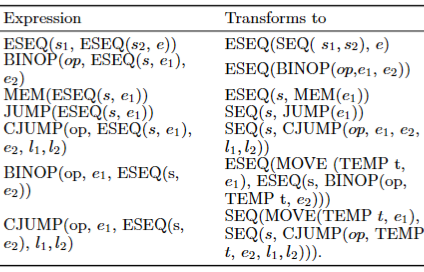
**3.重写流程:**

①一棵树重写成规范树

②将树分组合成不含CJUMP或JUMP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

**4.ESEQ转化:**



如果ESEQ中s和e2是**可交换的(commute)**,即s不影响 e2,那么可以直接先执行s，再取e2计算,不用临时保存

**5.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a0/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里:

CALL(fun,args)→ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

但这样 ESEQ 又会在 BINOP 下，直接提出来即可。

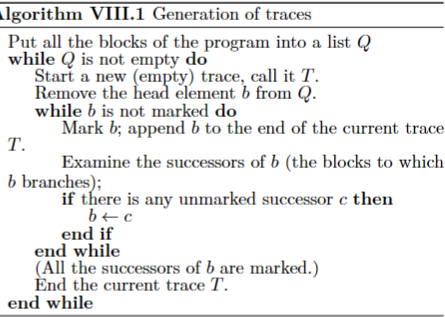
**8.2 处理条件分支**

**1.基本块(basic block):**取一列规范树,块的开始是label,以跳转指令为结尾.即: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;没发现一个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,则在街边那块末尾增加一条转移到下一个基本快标号处的JUMP;

**单个/多个 RETURN：**在末尾添加done LABEL,将JUMP(NAME done)放到最后一个基本快末尾.

**3.轨迹(trace):**程序执行期间可能连贯执行的语句序列. 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：

**4.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对其后跟随的既不是true也不是false标号的CJUMP,生成新基本块lf:

CJUMP(cond,a,b,lt,l’f)

LABEL l’f

JUMP(NAME lf)

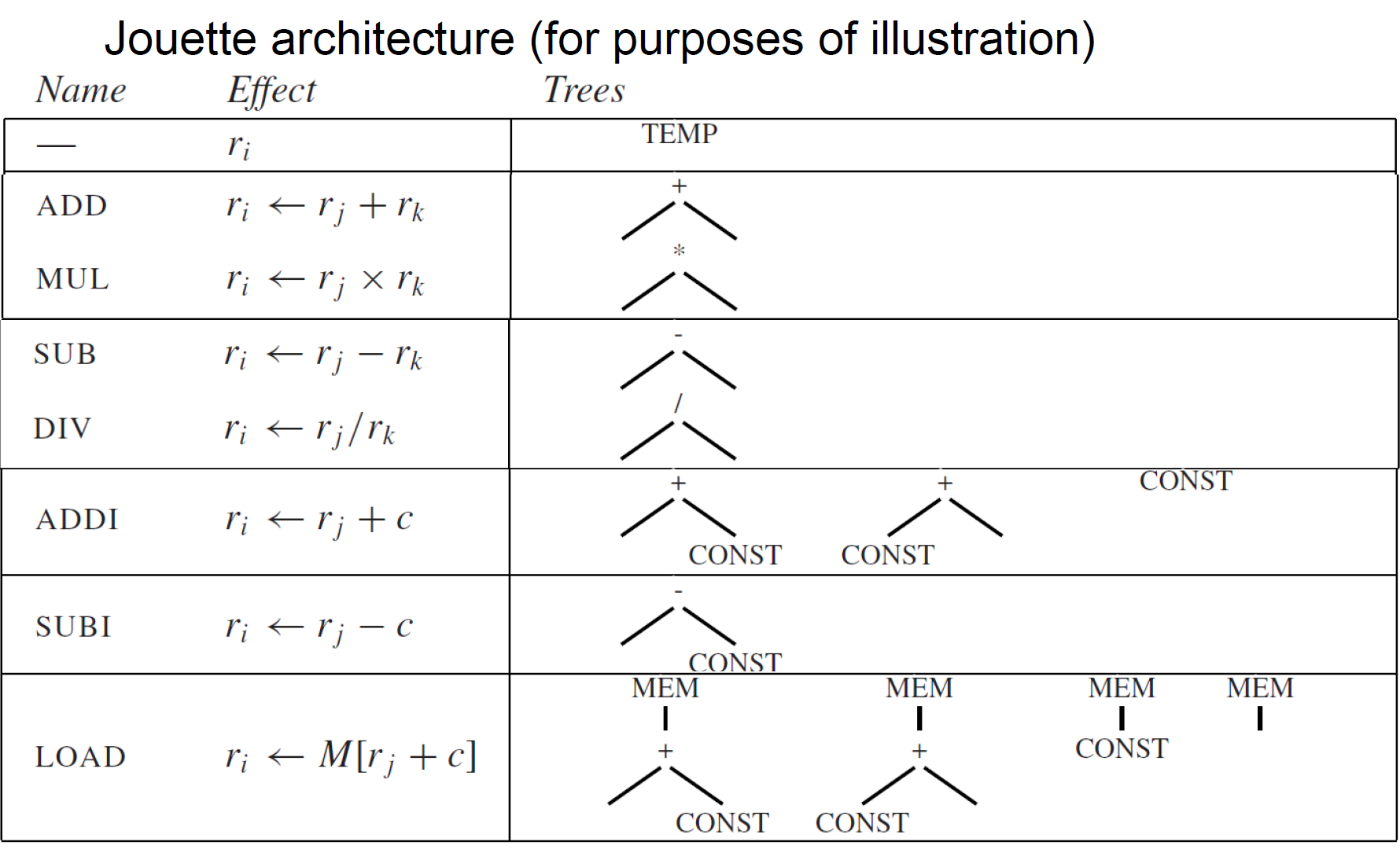
**CH9 指令选择**

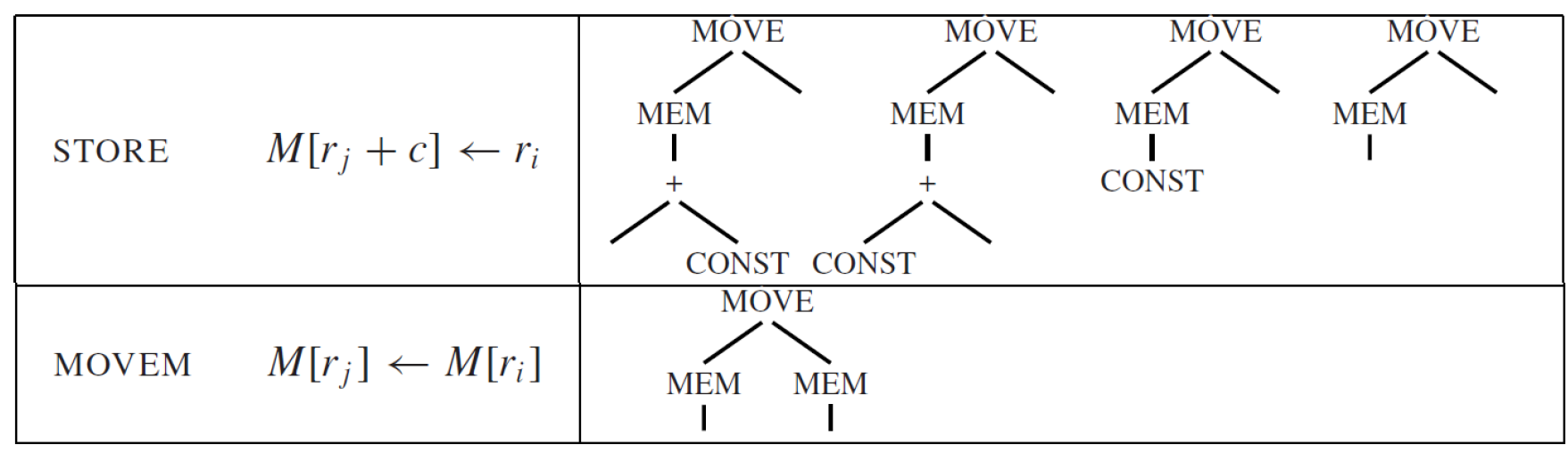
**9.1 树型到指令**

1.可以把一条机器指令表示成IR树的一段twig,称为树型(tree pattern).

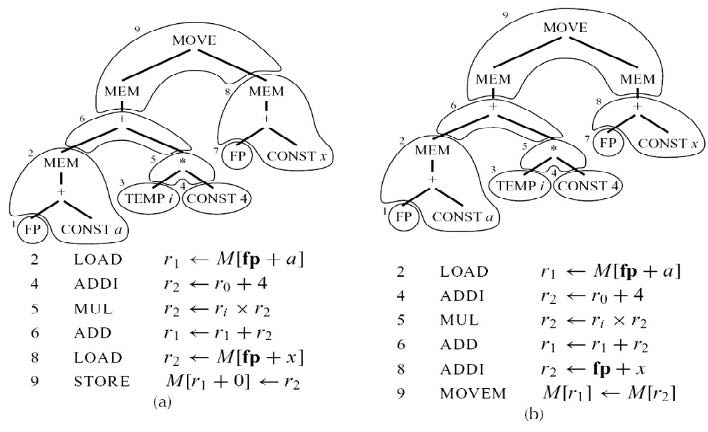
2.指令选择任务：使用树型的最小集合来覆盖(tiling)

3.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:





4.一棵树可以有多种tiling的方式:



**9.2 指令选择算法**

**1.Maximal Munch算法**:

①是贪心、局部最佳覆盖(optimal)算法

②自顶向下**Select指令**：从树的根节点开始寻找适合**最大瓦片；**对遗留的其他子树也进行相同操作 从 T’ 种 tile 选择，一个tile 需检查 K’ 节点，每次平均匹配掉的节点数K

**2.动态规划:**

①是全局最优覆盖 (optimum),子问题是子树的覆盖

②**自底向上**给每个节点计算代价:可以覆盖该节点为根的最优指令序列的指令代价之和,,每节点都要选代价最小方案。

③**递归向下**Select指令

**附：自底向上逆序Emit指令**：(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).

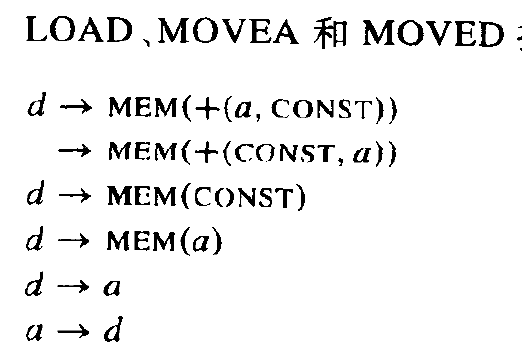
**3.树文法(Tree Grammar):**①动规的推广

②使用brain-damaged Jouette体系:

有两类寄存器(a寄存器:存地址;d寄存器:存数据)

③使用CFG来描述瓦片,文法有高度歧义性,但用动规只计算最小代价生成树

下面是生成load,movea,moved指令的树文法生成过程参考.



**4.快速匹配(fast match):**使用switch-case来匹配非叶子节点的label.

**9.3 CISC机器**

**1.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取指令和存指令只有M[reg+const]基址+offset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

**2.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 ADD AX,BX表示AX+=BX 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 iterator方式)

**3.CISC机器的特点解决难题:**①寄存器较少:IR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MOV传送指令

④算数运算可以访问存储器ADD [VAR1], [VAR2]:拆成4条.

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:INC [mem] (\*mem)++ 三种解决办法(a)忽略地址自增指令,性能差(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,在DAG中展开节点.

**CH10 活跃分析**

**10.0 定义**

1.编译器需要分析程序的中间表示,以确定哪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.

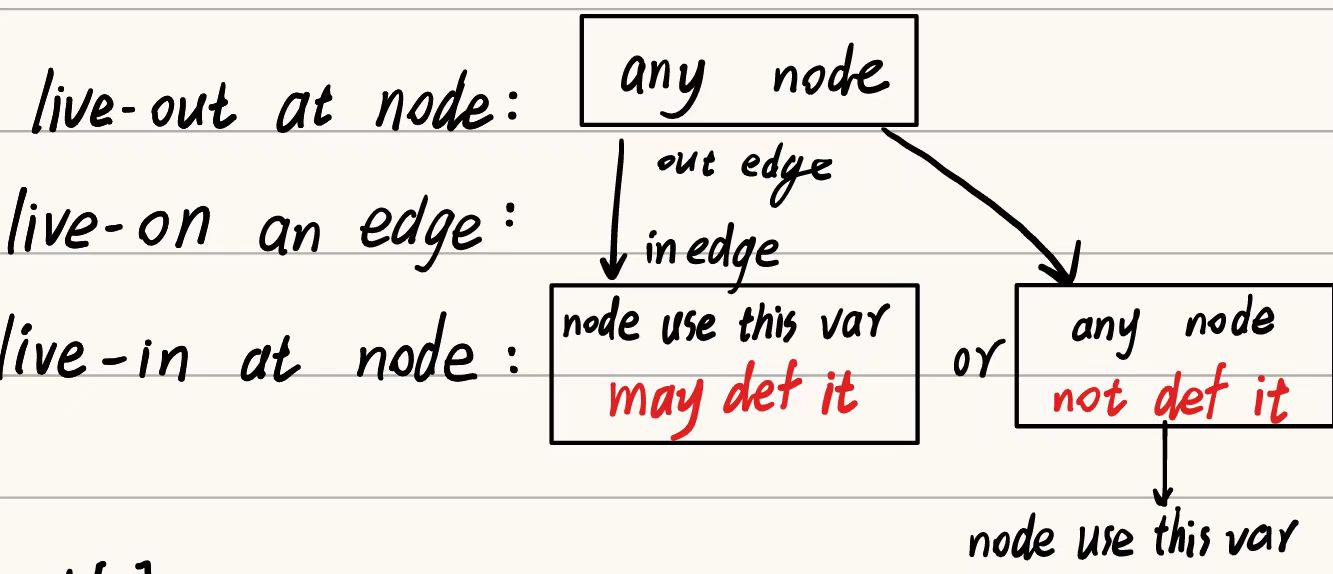
**2.控制流图(control flow graph):**程序的每条语句都是节点（不要合并BB）.

**3.活跃范围:**变量在那几条边上活跃的边集合.

**10.1 数据流方程的解**

succ[n]是节点n的后继节点;pred[n]是节点n的前驱节点

**2.赋值(def)；使用(use)**:



**10.2 活跃性计算**

**1.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

**2.活跃性计算的迭代方法:**

3.use作为in初值，指出 succ，适当排序,**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。**最后def**

**4.时间复杂度:**for循环初始化节点in,out需要O(N2);repeat循环的时间复杂度是O(N4).由于活跃信息大部分**稀疏**,实际运行时间在O(N)和O(N2)之间.

**10.3 集合表示**

**1.位数组(bit array):**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

**2.有序变量表:**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.

**3.方法比较:**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);集合密集:位数组表示更好.

**10.4 最小不动点**

**1.**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,如长链贯通 use ,产生代码指令所使用的寄存器比实际需要的多.

**2.定理:** **方程**10.2有一个以上的解(in,out计算公式方程)

**3.定理:** 方程10.2所有解都包含**最小解(least solution)**.

**10.5 静态/动态活跃性**

**1.停机问题:**不存在程序/TM 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:**假设存在程序H,我们会得出如下矛盾.从H构造函数F,F(Y)=if H(F,Y) then (while true{}) 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。

反证：若有 H’，则构造 U(‘X’) 在停机处加标号 Lhalt，则 H’(U(’X’),Lhalt)解决停机问题。

因此不可达和死代码消除没有 Universal，只有 Adhoc，无法确定是否提前经过 def。

**3.动态活跃:**程序**真正可达的一条**n到a的一个use路径，没有经过a的任何def（在更前面）**,**那么变量a在节点n时动态活跃的.

**4.静态活跃:**如果存在着一条从n到a的某个use路径,路径没有a的任何def,那么变量a在节点n静态活跃的.

**10.6 冲突图**

**1.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称为**冲突(interference).**

**2.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（与预染色冲突）

**3.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.

**4.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**任意bi∈out[n]且bi≠c,添加边(a,bi)**. **注:**可给(a,c)画虚线,但之后有冲突不忘实线,便于coalesce.

**CH11 寄存器分配**

**11.1 通过简化进行着色**

1.基于冲突图,寄存器分配问题转化为图着色问题:”颜色”=寄存器,相邻节点不能着同一种颜色.

部分变量必须保存到memory里,称之为”溢出”spill.

**2.简单图着色算法:**

**①构造(build):**构造冲突图.**②简化(simplify):**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.

方法:使用栈,将度小于K的节点压入栈中并从原图中删除,直到不能化简

**③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;需要按一定标准选择高度数点,将其**潜在溢出(potential spill)**,从图中删除并压入栈内打上标记.然后继续进行简化过程

**④选择(select):**将颜色指派给图节点,从空图开始重复地pop栈中图节点,重建图.当pop潜在溢出节点时,可能会发生无法着色的情况,这时发生**实际溢出(actual spill)** （潜在溢出可能有**邻接节点颜色相同**,可以着色不会成为实际溢出,成为乐观着色(optimistic color)）**⑤重开(start over):**如果不能为某些节点着色,那么需要重写程序，重复上述所有流程

**11.2 合并(coalesce)**

**1.合并的基础:**冲突图可以删除冗余的MOVE指令,但同时存在实边/不满足策略会 freeze。新节点的边是被合并的节点的边的并集.

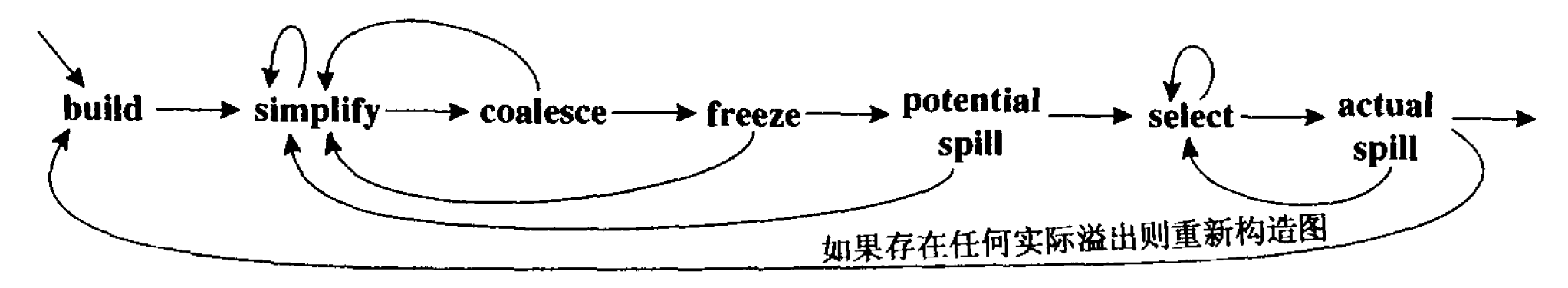
**2.安全的合并策略:**

**①Briggs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（邻居可先simplify）**

**②George策略:**节点a和b可以合并的条件是:对于a的**每一个邻居t**,满足两个条件之一即可合并a和b:(1)t与b已有冲突(2)t是低度数节点.

这两种策略都是**保守的(conservative)**,是因为合并不会改变图的着色性

**3.带合并的图着色算法图示**



**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).**②简化:**每次从图中删除低度数的**传送无关**节点,压入栈中. **③合并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(**把虚线画成直线**)重新开始简化和合并. **⑤溢出** **⑥选择:**弹出整个栈并指派颜色.

**4.受抑制的**

**(constrained):**就是冲突图中传送相关的两个节点之间有冲突边.既有虚线相连,又有实线相连.

**11.3 预着色(precolor)**

**1.定义:**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,节点已经有了”颜色”.所以叫”预着色”.**注意:预着色的节点构成完全冲突子图，两两互联.**

**2.**选择和合并阶段可以给普通临时变量分配与预着色相同的颜色,**只要不相互冲突**.

**3.预着色节点特性:**

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.

**5.溢出优先级计算公式:**对于节点a而言,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,使用少而冲突多,溢出优先级越高.

**6.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),改写程序：

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:

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[]:

**7. 对 CALL 的处理**：将所有 Caller-saved reg 写入该语句的 Defs，

Live-in 初始化为 arg regs，

Live-out 正常传递之后的值。

**8. 对 Callee-saved reg 处理：**

改写程序，在函数开头保护：

在函数末尾写回：

;

**CH13 垃圾收集**

**13.1 定义**

**1.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称之为垃圾(garbage).

**2.保守近似:**变量的活跃性不总能知道(停机问题的等价)

**3.可达性(reachable):** 程序变量和堆的记录构成一个有向图.每个程序指针是图中的一个根.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是可到达的.

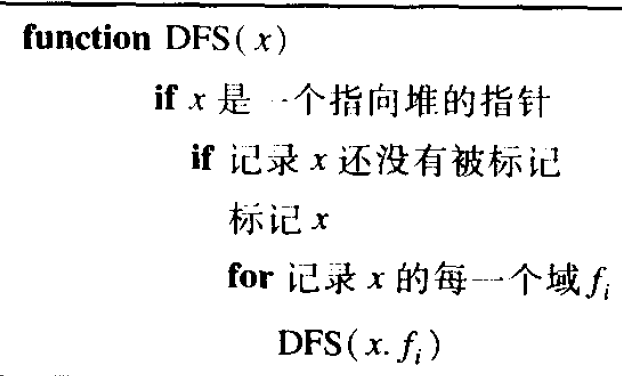
**13.2 标记-清扫式算法**

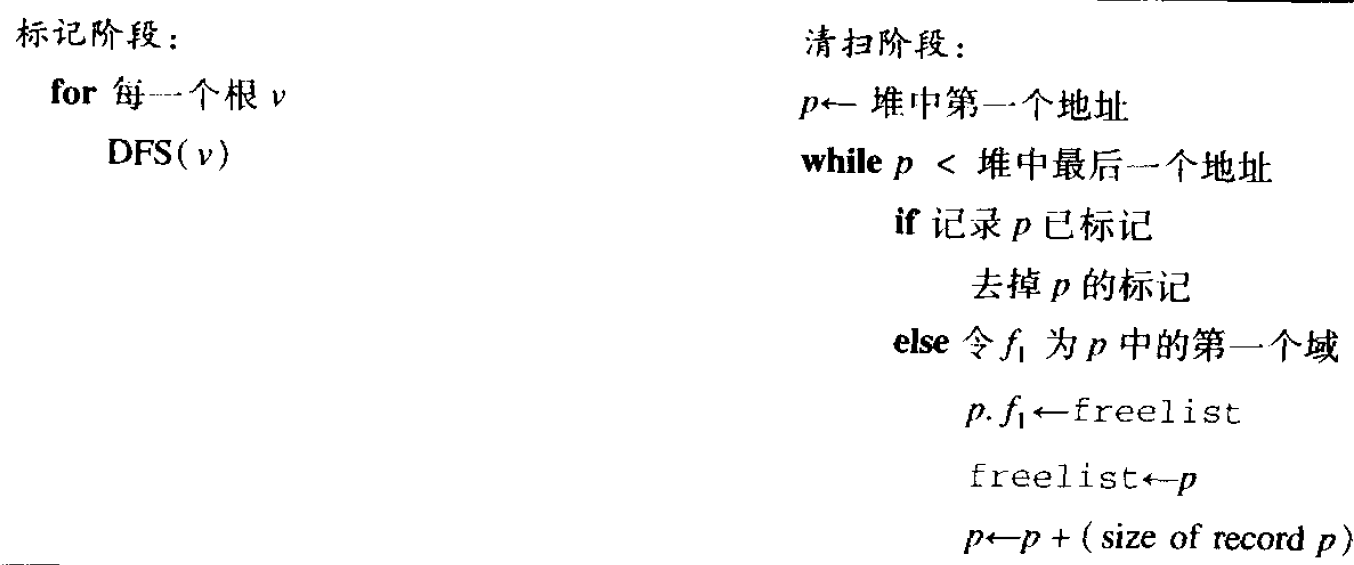
**1.算法原理:**

**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**空闲表(freelist)**中.同时清除所有已标记节点的标记.

伪代码如下(最朴素的算法):



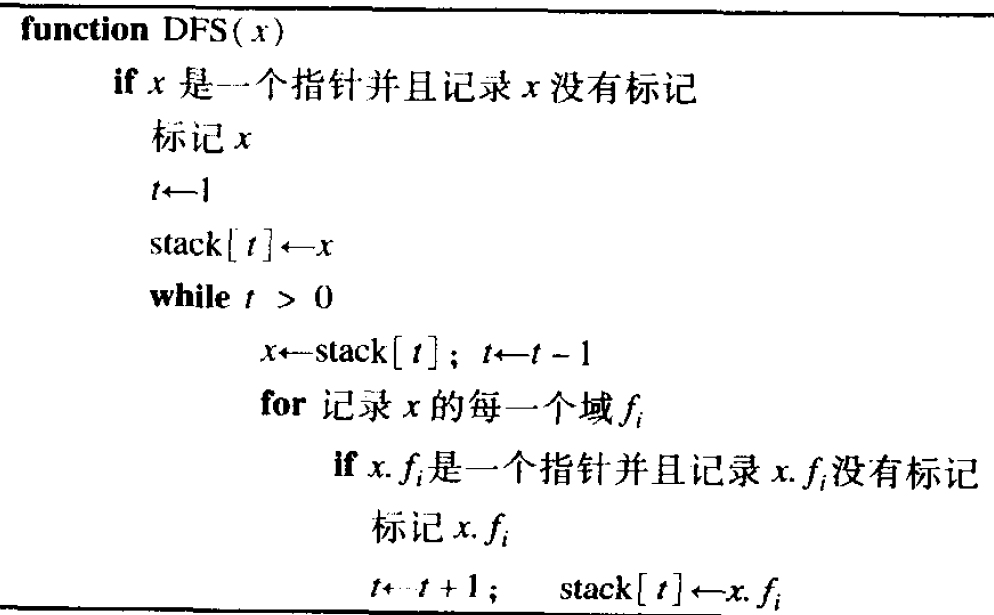


**2.朴素算法的时间复杂度:**标记阶段时间和标记节点个数成正比;清扫阶段时间与堆大小成正比. 假设大小为H的堆中有R个字可到达/将被标记,则一次垃圾收集的代价是,为常数;好处是可用大小为个字的自由储存单元补充空闲表. 摊还分析垃圾收集代价为

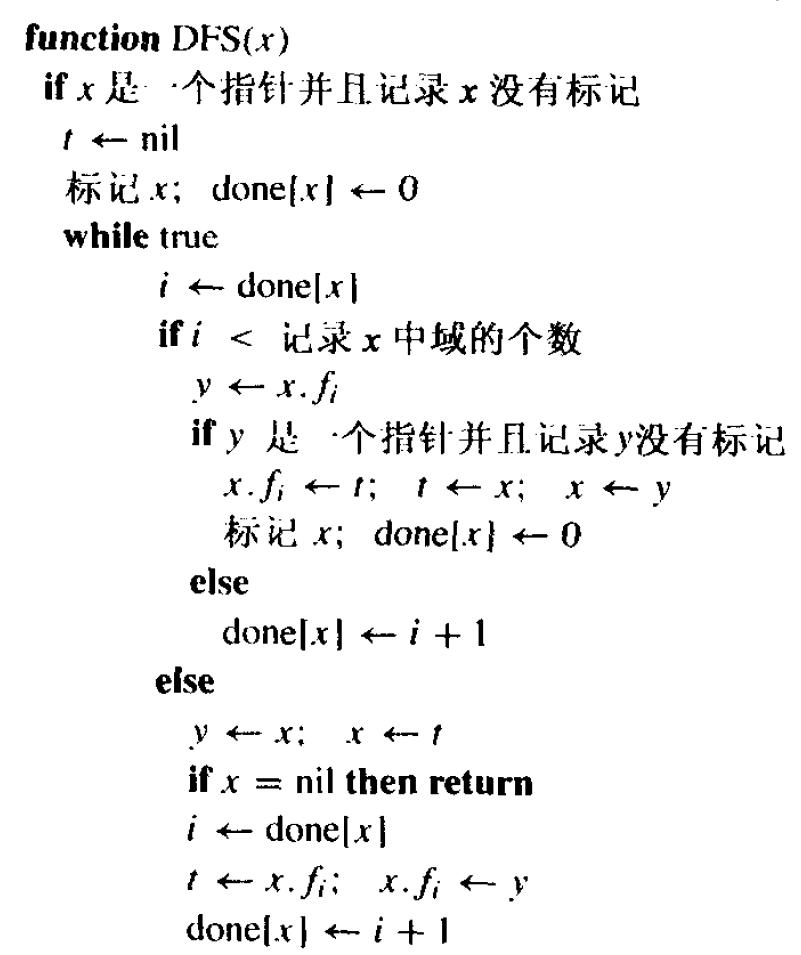
**3.朴素DFS改进:**

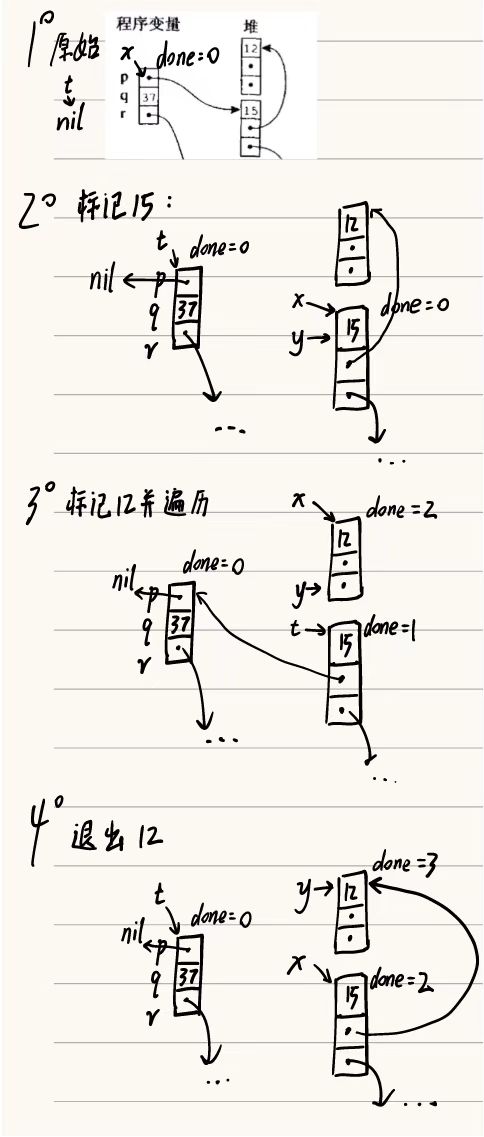
使用显式栈做 DFS,但显示的辅助栈在 H 个字串行可到达时,可能生长到H大小个字.

耗费空间太多，仍然不可接受.



**4.指针逆转(pointer reversal)：**在因为堆结构，指针相互引用的时候，本身构成图数据:





**5.空闲表数组:**使用简单链表效率低.

使用空闲桶（元素是空闲域）：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n

②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.

**13.3 引用计数算法(不常用）**

**1.算法原理:**类似 done,另一个辅助域。记住每一个记录有多少指针指向它. 计数的域和每一个记录储存在一起.改进：懒删除，将r从freelist中删除时，才递归减少r.fi计数:

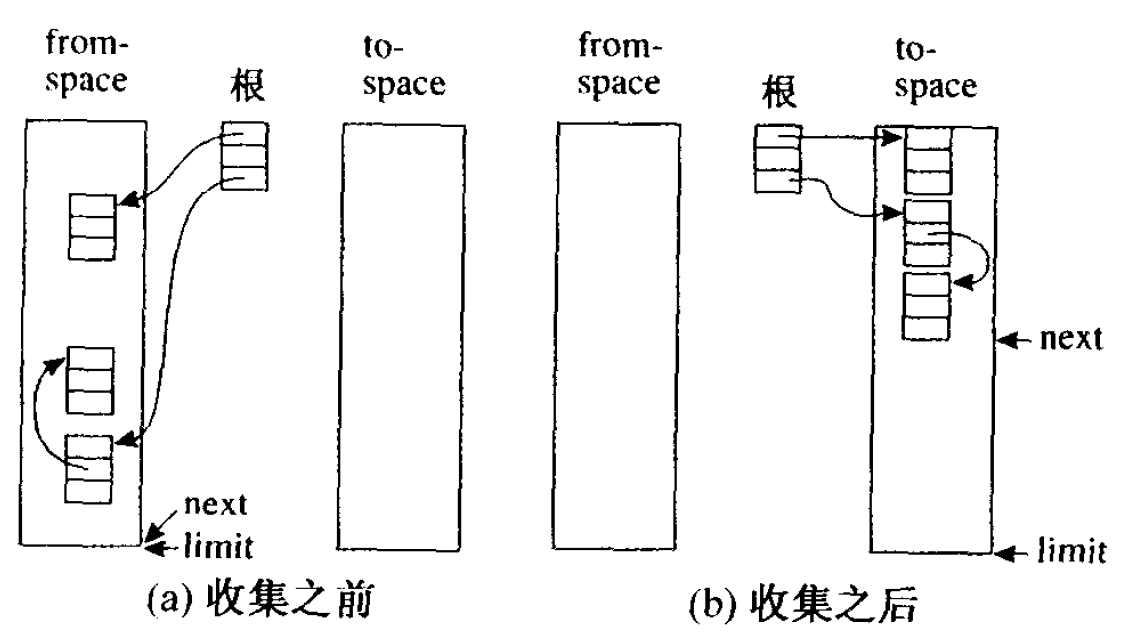
(1)能将递归减少的动作分解为较短的操作,是程序的运行更加平滑(对对交互式程序或实时程序好)

(2)递归减少的做法,递归减少动作只需要在分配器中进行.

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

**13.4 复制清理算法**

**1.图例**

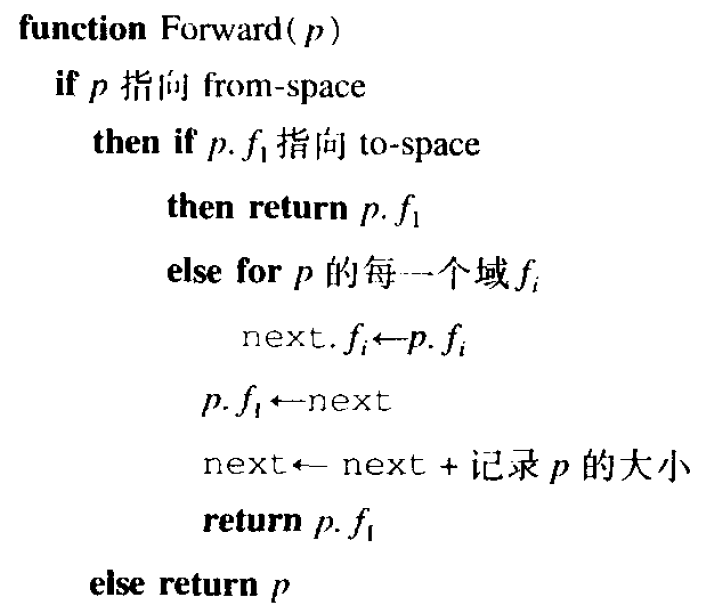


**2.算法流程：**

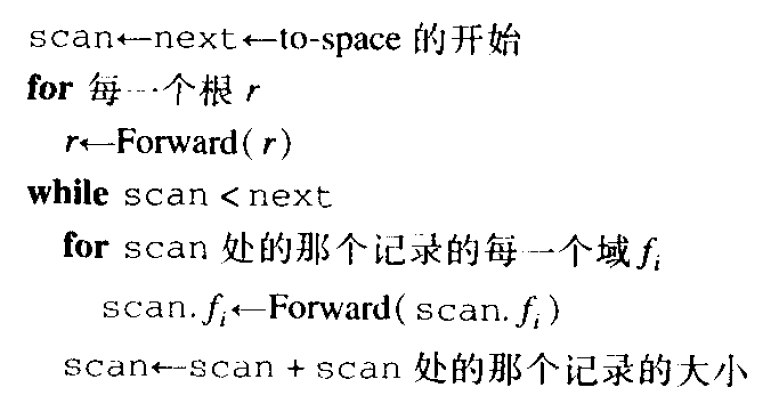
①回收初始化：初始化指针next指向to-space的开始.每当from-space发现一个可达记录,便复制到to-space的next所指位置,同时式next增加该记录的大小

②转递(forwarding)：使域内指向from-space的指针p转而指向to-space（3情况）

**3.传递指针算法伪代码:**



**4.Cheney算法:**用BFS对可达数据进行遍历.



①位于to-space开始和scan之间的是copied and scanned的记录,这一区域的所有指针均指向to-space.

②位于scan和next之间区域包含的是copied to-space，但子域还没有传递的记录:子域指向from-space

③while循从scan向next移动,复制记录也导致next移动.

④当所有可达数据都被复制到to-space后scan追上next.

**⑤算法优点:**copy deep的同时 scan deep, 不需要DFS scan;使用scan和next之间的区间作为BFS队列;实现简单.

⑥算法缺点:引用局部性差

⑦算法代价:共R个字可到达，需赋值，算法回收个字.摊还代价为 ；当H远超R时，代价接近于零

6.混合式算法(半深度优先搜索):缓解Cheney引用局部性差缺点，需要forward时，一次chase深度复制。

**13.5 编译器接口**

**1.Fast Allocation:**使用上述(copying collection) 垃圾收集，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.

**2.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result

④清除M[next] -M[next+N-1]

⑤next:=next+N

⑥从分配函数返回.A.将result作为分配结果指针返回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结合懒消除;②⑤不可以被消除,但可把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**.

**3.数据布局(data layout)**

done/size:让每个对象的第一个字**指向**descripter：包含对象的总大小以及每一个指针域的位置

**4.指针映射(pointer map)：**①编译器必须能给收集器标出**栈中**存放指针的临时变量和局部变量(寄存器中or活动记录)

②每条指令都可能使活跃临时变量集合发生改变,故指针映像变化频率高，不维护

③时机：仅在可开始新的垃圾收集的点才构建指针映像: alloc的调用点；每个函数的调用点

④指针映像用函数返回地址作为键：

(1)为找到所有根,收集器从栈顶向下扫描

(2)每一个函数返回地址在映射中,描述下一个栈帧中指针

(3)每个栈帧内GC从栈开始标记根，第二轮再到堆中

(4)callee-save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的寄存器但不知道哪些是指针.所以g的指针映像必指出,在调h时其callee-save的寄存器哪些指针从f继承.

**5.派生指针(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]

③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)并指出其基指针.

④GC在把a复制清理到地址a’时,也必须调整t1到t1+a’-a;**一个导出的指针将隐式地保持其基指针活跃.**

**CH14 面向对象语言**

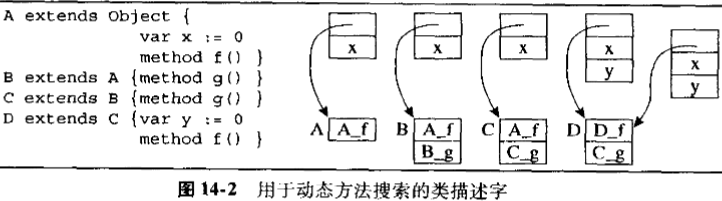
**1. 单继承 SI，**关系图是树

·Method dispatch：将method编译成instance

·Class decriptor:记录基类指针，Method instances

·Static method: 类似 static link 查询，利用类描述字+固定偏移+基类ptr。

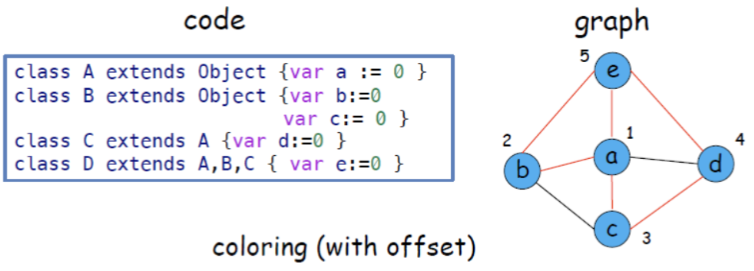
·Dynamic method:lw Dispatch Vector/vtable 虚函数表：运行时可 lookup

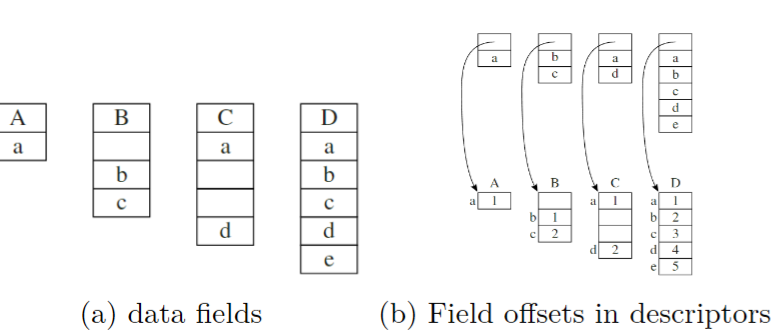


这样运行时，对象找其类描述自字，添加偏移直接调用。

**2. 多继承 MI**关系图是DAG

·field layout：为了让对象固定偏移找field，出现在同一类的 field 冲突，不可染色/占同一个 slot





·Method dispatch：同样图染色。空出 slot 会浪费真实内存，可使用hashmap<id, field\_offset/method\_addr>

若要在 object c 获取 field b：

1) 在c的0偏移处 (开头) 找到 class descriptor，d.

2) 从偏移量 d + Ktab + hashb 获取field name f

3) 对比 f 是否与 b 相同 (处理 hash 冲突)

4) 从偏移量 d + Ftab + hashb 获取 field b

5) 获取 field b 的内容

**3. Testing Class Membership**

安全向上封装，测试某对象是否是某个类C示例；朴素法：递归查找类描述字，找基类是否有 C。display 链：只适用单继承，树深度=继承深度，**每一对象都有深度递增**继承链，看该链C深度是否是C即可。

**4. Private Fields and Methods**

在 semantic **类型检查**中提早做，看非类 method访问属性是否标有private

**CH18 循环优化**

·循环的header/entry node h：CFG中，对循环体任何节点x，都有x到h（含反向边） 和 h 到 x的路径，且S以外进入循环体，必经过 h；（难以计算）

·Dominator tree：从CFG 入口 s 到达 n 必须经过 d，则d是n的 dominator， d dom n.

init D[s] = {s}，D[other]=whole graph；遍历

支配所有前驱，才支配该节点。

·Immediate dominator：距离n 最近 / 被dom(n)其他支配节点所支配的最弱的不是自己（非平凡）支配者，则 idom(n)=d

**除了 s 无 idom，排除不可达，其他都恰有一个 idom，且 idom tree父节点集合=dom**。

**2. Natural Loop**

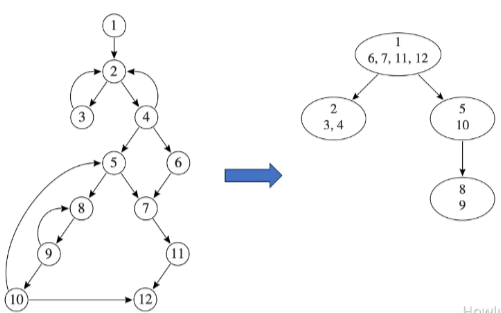
边 n → h 满足 h dom n, 则这是 back edge，所指向的h 称为 loop header

循环体：被 header 支配且可不经过header 到达 n

**注：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)。Loop-nest tree ：**每个节点第一行是循环头，想象整个procedure在假想大循环中。



**Loop preheader：**前置首节点，唯一后继是 header，用于：

**3. Loop Invariant Hoisting**

赋值语句 invairant：操作数是常数或def在循环外/def也不变（cascade）；有副作用（内存操作）谨慎外提。

