复习提纲及题型总结

第一章 引论（基本概念）

- 什么是编译器：是一个程序、读入源语言编写的程序，并把该程序翻译成为一个等价的、用目标语言编写的程序
- 编译器结构：分析部分+综合部分
- 一个典型的编译程序包括：词法分析、语法分析、语义分析、中间代码生成、机器无关代码优化、代码生成、机器相关代码优化；
- 词法分析：读字符流输出词素，对每个词素生成词法单元传递给语法分析
- 语法分析：创建树形中间表示形式：语法树，指出词法单元流的语法结构
- 语义分析：语法树+符号表，检查源程序是否满足语言定义的语义约束；
  - 收集类型信息用于代码生成+类型检查、类型转换
- 中间代码生成：根据语义分析输出生成类机器语言的中间表示，例如三地址代码生成
- 代码优化：改进中间代码，有具体的设计目标
- 代码生成：把中间表示形式映射到目标语言，寄存器分配、指令选择、内存分配
- 符号表：可由各个步骤使用
  - 记录源程序中使用的变量的名字，收集各种属性
    - 名字的存储分配
    - 类型
    - 作用域
    - 过程名字的参数数量、参数类型等等
- 趟：以文件为输入输出单位的编译过程的个数，每趟可由一个或若干个步骤构成
- 语言的代分类：机器语言->汇编语言->高级程序设计语言->特定应用语言->基于逻辑和约束的语言
- 命令式语言和声明式语言：前者指明如何完成，后者指明完成哪些计算
- 参数传递机制：值调用、引用调用、名调用

第三章 词法分析

- 核心考点：L<==>RE==>NFA==>DFA==>最小化 DFA
- 词素：模式匹配的字符序列；词法单元：单元名+可选的属性值
  - eg：**comparison** 是词法单元，<=是词素
- **正则表达式？（重要）**
  - 表示“模式”
  - 概念
    - 字母表：有限的符号集合
    - 串：字母表中符号组成的一个有穷序列
    - 语言：给定字母表上一个任意的可数的串的集合
    - 子串：连续；子序列：不连续
  - 语言上的运算：并、连接、Kleene 闭包、正闭包、零个或一个(?)、字符类([a-z]=a|b|…|z)
    - 运算符优先级：* > 连接符> |，(a)|((b)*(c))可以改写为 a|b*c

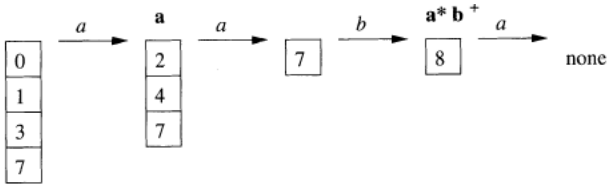
- 已知正则表达式写对应语言？
- 已知语言写正则表达式？
- 正则定义？命名正则表达式，使表示简洁
  - 怎么写正则定义？
    - d1→r1
    - d2→r2
    - ...
    - dn→rn
- 状态转换图？
  - 开始状态：start 边；回退输入：加“*”号；接受状态：双层圈
  - 保留字怎么处理？两种方法：
    - 在符号表中预先填写保留字，并指明它们不是普通标识符
    - 为关键字/保留字建立单独的状态转换图。并设定保留字的优先级高于标识符
- **有穷自动机 NFA/DFA？（重要）**
  - 注意二者等价
  - NFA：一个符号标记离开同一状态的多条边+可以有边的标号是 ε
    - 例子
      - 状态集合 S={0,1,2,3}
      - 开始状态 0
      - 接受状态集合{3}
      - 转换函数：(0,a)->{0,1} (0,b)->{0} (1,b)->2 (2,b)->3
    - 转换表？结构是什么？怎么写？示例：

状态	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

    - 什么时候认为 NFA 接受输入字符串 x？
      - 存在从开始状态到接受状态的路径
  - DFA：有且仅有一条离开该状态、以该符合为标号的边+没有标记为 ε 的边
- NFA==>DFA？子集构造法、转换表
  - 子集构造法？示例
    - A：=ε-closure(0)={0,1,2,4,7}
    - B：Dtran[A,a]=ε -closure(move(A,a))=ε -closure({3,8})={1,2,3,4,6,7,8}
    - ……
  - 转换表？示例：注意指明**开始状态和接受状态！**

NFA 状态	DFA 状态	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

- RE==>NFA？递归定义构造，注意 Kleene 闭包构造的时候前后要加新节点

- DFA 最小化？核心：同组状态不能被任意串区分
  - 可区分：只要有一条分别从状态 s 和状态 t 出发，沿着标号为 x 的路径到达的两个状态只有一个是接受状态，称为 s 和 t 可区分
  - 先分出接受状态，然后对剩余状态判断是否可区分。
  - 例子：
    - 初始分划：{A,B,C,D} {E}
    - 处理{A,B,C,D}，b 把它细分为{A,B,C}{D}
    - 处理{A,B,C}，b 把它细分为{A,C} {B}
    - 分划完毕。选取 A,B,D 和 E 作为代表，构造得到最小 DFA
  - 如果全部都是接受状态？
    - 一定有一个通过转化可以变成空集，在分划的时候先加入一个空集，然后把这个会转成空的单独分出，最后对剩余的所有状态最小化 DFA
    - 例如：{A,B,C,D} {空}，然后开始划分，最终得到{A,C,D}{B}{空}
- 使用 NFA 的词法分析器：往后找直到匹配
  - 例如： 

要找出所有可能性，如果多个匹配则匹配最前面的
- 使用 DFA 的词法分析器
  - 初始分划：所有非接受状态集合+对应于各个模式的接受状态集合，并且需要增加死状态Φ，用作词法分析的 DFA 可以丢掉 Φ
    - 例如：初始分划{0137, 7}{247}{8,58}{68}{Φ}

第四章 语法分析

- 自顶向下：LL 文法（应转化为无左递归版本），自底向上：LR 文法
- 上下文无关文法
  - 组成：中介符号、非终结符号、开始符号、产生式
    - 产生式：“产生式头（非终结符号）->产生式体”
  - 推导：单步推导、零步或多步推导、一步或多步推导、最左推导（lm）、最右推导/规范推导（rm）
  - 句型：S 推导 a，则 a 是句型
    - 最左句型（由最左推导得出）、最右句型
  - 句子：不包含非终结符号的句型
  - 语言：句子的集合
  - 语法分析树？推导构造语法树？语法树反写推导？
  - 验证文法生成的语言：证明 G 生成的每个串都属于 L，L 的每个串都由 G 生成
  - **上下文无关文法与正则表达式：（重点）**
    - 正则表达式不能计数
    - 为 NFA 构造等价文法？每个状态创非终结符，若有转换 B=(A,a)则 A→aB，最后接受状态则 A→ ε

- 例如：
  - A0→aA0| aA1| bA0
  - A1→bA2
  - A2→bA3
  - A3→ε

- **设计文法：重点**
  - 消除二义性：if then 和 if then else
    - 基本思想: 在一个 then 和一个 else 之间出现的语句必须是“已匹配的”。也就是说 then 和 else 中间的语句不能以一个尚未匹配的 then 结尾
  - 消除左递归：——自顶向下
    - A → Aα| β==>A → βA’，A’ → αA’ | ε
    - 非立即左递归？将非终结符从小排到大，把小的非终结符代到大的产生式体中可以消，例如：
      - S → Aa| b, A → Ac | Sd| ε
      - 代入 S 的产生式得：A → Ac | Aad| bd| ε，即可消除立即左递归
      - 结果为 S → Aa| b, A → bdA’|　　A’, A’ → cA’| adA’| ε
  - 提取左公因子：——自顶向下
    - A→αβ1| αβ2==> A → αA’，A’→β1 |β2
- 自顶向下分析：匹配-推导
  - **FIRST 集和 FOLLOW 集（重点）**
    - FIRST(α)：可以从α推导得到的串的首符号的集合；（ε可在 FIRST 集中）
      - X 终结符号，FIRST(X)={X}
      - X 非终结符号，且 X→ a…,那么将 a 添加到 FIRST(X)中；如果 X→ ε，那么 ε 也在 FIRST(X)中
      - 对于规则 X→ Y1Y2…Yn，把 FIRST(Y1)中的非 ε 符号添加到 FIRST(X)中。如果 ε 在 FIRST(Yn)中，把 ε 添加到 FIRST(X)中
    - FOLLOW(A)：可能在某些句型中紧跟在 A 右边的终结符号的集合。FOLLOW 集有 $
      - 将$放入 FOLLOW(S)，S 是开始符号，而$是输入串的结束标记
      - 如果存在产生式 A→αBβ，那么 First(β)中除 ε 之外的所有符号都在 Follow(B)中
      - 如果存在一个产生式 A→αB，或存在产生式 A→αBβ 且 First(β)包含 ε，那么 Follow(A)中的所有符号都在 Follow(B)中
  - LL(1)文法
    - 定义：<u>用于判断文法是不是 LL(1)的</u>
      - FIRST(α)∩FIRST(β) = Φ;（条件一、二）
      - 如果 ε ∈FIRST(β)，那么 FIRST(α)∩FOLLOW(A) = Φ；反之亦然。（条件三）
    - 预测分析表？结构？怎么用？
      - 对于 First(α)中的每个终结符号 a，将 A→α 加入到 M[A,a]
      - 如果 ε 在 First(α)中，那么对于 Follow(A)中的每个终结符号 b，将 A→α 加入到 M[A,b]中

- 如果 ε 在 First(α)中，且$在 Follow(A)中，将 A→α 加入到 M[A,$]中
- 例如：

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

- 多重定义条目？非 LL(1)文法

◆ 表驱动的非递归的预测语法分析

- 例如：

已匹配	栈	输入	动作
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $F \to \mathbf{id}$
id	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	匹配 id
id	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	输出 $T' \to \epsilon$
id	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	输出 $E' \to +\ TE'$
id +	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	匹配 +

- 匹配失败的例子：这里注意整个为什么会最终无法匹配

22	S3->aSa	aaaaaa$	匹配成功，返回调用上一层；
23	S2->aSa	aaaaaa$	匹配失败，换产生式；
24	S2->aa	aaaaaa$	匹配成功；
25	S2->aa	aaaaaa$	匹配成功，返回调用上一层；

■ 自底向上语法分析：移入-归约

◆ **句柄（重要）：句柄是最右推导的反向过程中被规约的那些部分**

- 然后可以根据产生式规约
- 和某个产生式匹配的最左子串不一定是句柄
- 例如

最右句型	句柄	归约用的产生式
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T\ *\ F$

- 问一个句型的句柄，就看这个句型怎么按最右推导的反向过程往上归约，第一步被归约的就是句柄

◆ 移入-归约分析技术：

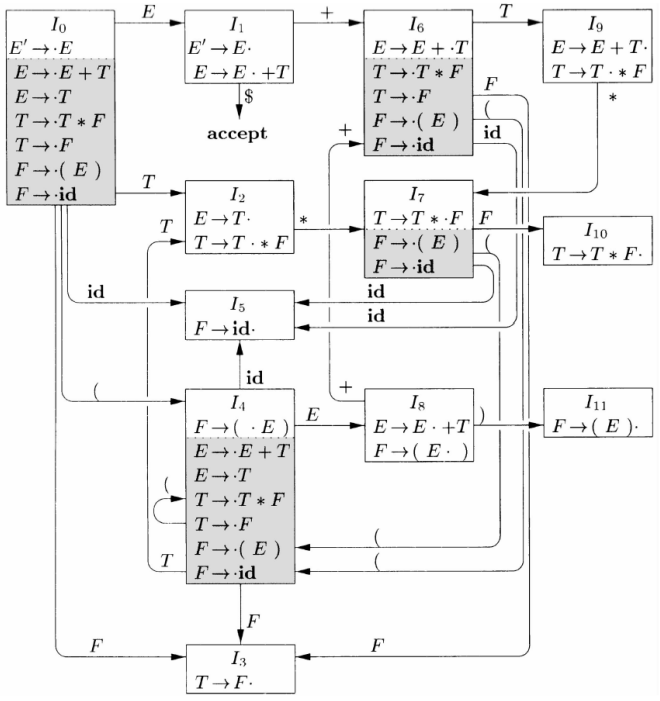
- 开始时刻：栈中只包含$，而输入为 w$
- 句柄总是在栈顶
- 成功结束时刻：栈中$S，而输入$
- 例如：

栈	输入	动作
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	移入
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	按照 $F \to \mathbf{id}$ 归约
$ F	$* \mathbf{id}_2$ $	按照 $T \to F$ 归约
$ T	$* \mathbf{id}_2$ $	移入
$ $T *$	$\mathbf{id}_2$ $	移入
$ $T * \mathbf{id}_2$	$	按照 $F \to \mathbf{id}$ 归约
$ $T * F$	$	按照 $T \to T * F$ 归约
$ T	$	按照 $E \to T$ 归约
$ E	$	accept

- **存在的冲突及原因——样题**
  - 移入/归约冲突：不能确定栈顶元素是否是句柄
  - 归约/归约冲突：不能确定句柄归约到哪个非终结符号

◆ LR 语法分析技术

- 规范 LR(0)项集族
  - 增广文法：加入产生式 S' →S
  - CLOSURE(I)：I 的项集闭包，对应于 DFA 化算法的 ε-CLOSURE
  - GOTO(I, X)：I 的 X 后继，对应于 DFA 化算法的 MOVE(I, X)
- 内核项和非内核项
- LR(0)自动机如何决定移入/归约？
  - 如果下一个输入符号为 a，且 j 上有 a 的转换，就移入 a；否则就归约
  - 核心：只有当下一个输入在 FOLLOW(A)中时才可以归约
- LR(0)语法分析过程：注意 7->8 是弹出 2710 再压入 2 得到的



行号	栈	符号	输入	动作
(1)	0	\$	**id** * **id** \$	移入到 5
(2)	0 5	\$ **id**	* **id** \$	按照$F \to$ **id** 归约
(3)	0 3	\$ F	* **id** \$	按照 $T \to F$ 归约
(4)	0 2	\$ T	* **id** \$	移入到 7
(5)	0 2 7	\$ T *	**id** \$	移入到 5
(6)	0 2 7 5	\$ T * **id**	\$	按照$F \to$ **id**归约
(7)	0 2 7 10	\$ T * F	\$	按照$T \to T * F$归约
(8)	0 2	\$ T	\$	按照$E \to T$ 归约
(9)	0 1	\$ E	\$	接受

- LR 语法分析表的结构
  - ACTION：就是下一个符号是什么来决定移入(s)/归约(r)
  - GOTO：就是非终结符是什么决定状态跳到哪里

状态	ACTION						GOTO		
	id	+	*	(	)	\$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- **构造方法（很重要）：**
  - 构造 G' LR(0)项集规范族 C={I0, I1,… In,}
  - 对于状态 i 中的项：
    - [A → α·aβ]且 GOTO[Ii,a]=Ij，那么 ACTION[i,a]=移入 j
    - [A → α·]，那么对于 FOLLOW(A)中的所有 a，ACTION[i,a]=归约 A → α
    - [S' →S·]，那么 ACTION[i,\$]=接受
  - 状态 i 对于非终结符号 A 的转换规则：
    - 如果 GOTO[Ii,A]=Ij，那么 GOTO[i,A]=j
  - 前述没有定义的所有条目设置为“报错”
- 没有重复条目，就是 SLR(1)文法；多个可选的动作/条目，就不是 SLR(1)文法

◆ LALR，LR(1)项集
- 核心：让 LR 分析器的每个状态精确指明哪些输入符号可以跟在句柄 α 后面，使 α 可能被规约为 A
- 构造关键点：

repeat
  for (I 中的每个项 $[A \to \alpha \cdot B\beta, a]$)
    for (G' 中的每个产生式 $B \to \gamma$)
      for (FIRST(βa)中的每个终结符号 b)
        将 $[B \to \cdot\gamma, b]$ 加入到集合 I 中;
until 不能向 I 中加入更多的项;

- 规范 LR(1)语法分析表
  - 与 SLR 的唯一区别是：SLR 对 FOLLOW(A)所有都设置归约，但此处仅对有向前看符号的项归约
- LALR 语法分析表：合并拥有相同项集核心（第一分量的集合）的 LR(1)项集（状态数与 SLR 分析表一样多，分析能力强一点）
  - 例如：
    - I3={ [C→c·C, c/d],[C→·cC, c/d],[C→·d, c/d] }
    - I6={ [C→c·C, $],[C→·cC, $],[C→·d, $] }
    - 合并后：I36= { [C→c·C, c/d/$],[C→·cC, c/d/$],[C→·d, c/d]/$ }
- 合并可能产生的冲突（很重要）
  - 不可能引入归约-移入型冲突：反证法（背！）
    - **假定合并后有移入-归约冲突，就是说：有项[A→α·, a]和项[B→β·aγ, b]。显然，原来的项集中都有[B→β·aγ, c]。而[A→α·, a]必然也在某个原来的项集中。这样，合并前的 LR(1)项集已经存在移入-归约冲突**
  - 但是可能引起归约-归约冲突
    - 例如：构造文法

      $S' \rightarrow S$

      $S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$

      $A \rightarrow c$

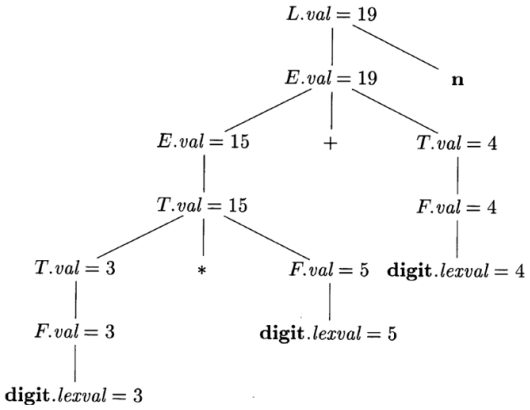
      $B \rightarrow c$
    - 可行前缀 ac 的有效项集{[A→c·,d],[B→c·,e]}
    - 可行前缀 bc 的有效项集{[A→c·,e],[B→c·,d]}
    - 合并之后的项集为{[A→c·,d/e],[B→c·,d/e]}当输入为 d 或 e 时，产生归约-归约冲突。
- 使用二义性文法：优先级、结合性（关键：*优先级>+，且+左结合；+归约，*移入）
- 预测分析中的错误恢复：
  - 恐慌模式（假装没有看见，synch 同步错误，然后弹出非终结符号）
  - 短语层次的恢复（空白条目中插入错误处理例程的函数指针，例程可以改变、插入或删除输入中的符号；发出适当的错误消息）

第五章 语义分析（语法制导翻译）

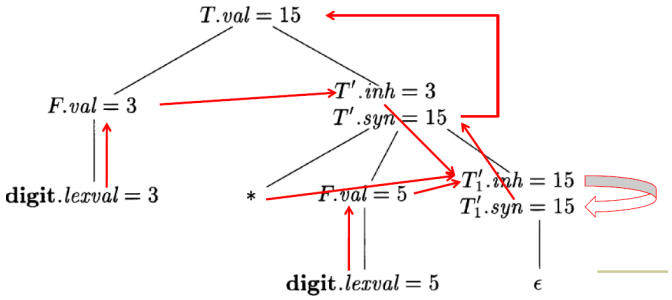
- 语法制导定义 SDD
  - 例子

产生式	语义规则
1) $L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2) $E \rightarrow E_1\ +\ T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1\ *\ F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (\ E\ )$	$F.val = E.val$
7) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\mathrm{lexval}$

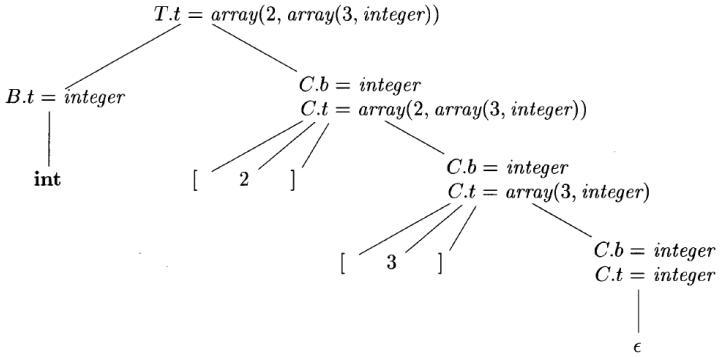
- 综合属性（子节点+本身）、继承属性（父节点、本身和兄弟节点）
- S 属性的 SDD：只包含综合属性，可与 LR 语法分析器一起实现
- L 属性的 SDD：综合属性+继承属性（必须从上往下，从左到右求 L 属性且无环）。判断是否可以计算 SDD 的依据：SDD 依赖图出现环
- 注释语法分析树
  - 例子：3*5+4n



  - 观察 inh 属性如何传递



- int[2][3]类型表达式的生成，例子：



- 语法制导的翻译方案——程序片段
  - 后缀翻译方案：所有动作都在产生式最右端的 SDT
    - 条件：文法可以自底向上且 SDD 是 S 属性
    - 构造方法
      - 将每个语义规则看作是一个赋值语义动作
      - 将所有的语义动作放在规则的最右端
    - 消除左递归:
      - $A \rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\}$
      - $A \rightarrow X\ \{A.a = f(X.x)\}$

$$A \rightarrow X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\}$$
$$R \rightarrow Y \ \{R_1.i = g(R.i, Y.y)\} \ R_1 \ \{R.s = R_1.s\}$$
$$R \rightarrow \epsilon \ \{R.s = R.i\}$$

- 有左递归的话不能构造 LL，不能自顶向下分析，即不能 L 属性 SDD 分析；
- 注意 LL 文法：L 属性 SDD，自顶向下；LR 文法：S 属性 SDD，自底向上。

■ L 属性定义的 SDT

◆ 位置？

- 计算 A 的继承属性的动作插入到产生式体中对应的 A 的左边
- 计算产生式头的综合属性的动作在产生式的最右边

◆ 例子：

$S \rightarrow$ **while** (	{ $L1 = new()$; $L2 = new()$; $C.false = S.next$; $C.true = L2$; }
C)	{ $S_1.next = L1$; }
S_1	{ $S.code =$ **label** $\|\|$ $L1$ $\|\|$ $C.code$ $\|\|$ **label** $\|\|$ $L2$ $\|\|$ $S_1.code$; }

第六章 中间代码生成

- 有向无环图可以指出表达式中的公共子表达式
- 三地址指令书写？
  - 四元式表示？格式（字段）：op、arg1、arg2、result
    - 单目运算符不使用 arg2
    - param 运算不使用 arg2 和 result
    - 条件转移/非条件转移将目标标号放在 result 字段
  - 三元式表示？op、arg1、arg2
    - y=x[i]需要拆分为两个三元式：t=x[i], y=t
    - y=x 即表示为= y x
- 类型等价
  - 结构等价
    - 或者它们是相同的基本类型
    - 或者是相同的构造算子作用于结构等价的类型而得到的。
    - 或者一个类型是另一个类型表达式的名字
  - 名等价：类型名仅仅代表其自身
- 表达式代码的 SDD

产生式	语义规则
$S \rightarrow$ **id** = E ;	$S.code = E.code \ \|\|$ $gen(top.get(\textbf{id}.lexeme) \ '=' \ E.addr)$
$E \rightarrow E_1$ + E_2	$E.addr =$ **new** $Temp()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr \ '=' \ E_1.addr \ '+' \ E_2.addr)$
| - E_1	$E.addr =$ **new** $Temp()$ $E.code = E_1.code \ \|\|$ $gen(E.addr \ '=' \ '\textbf{minus}' \ E_1.addr)$
| (E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$
| **id**	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

- 增量翻译：主属性 code 满足增量式翻译的条件
- 数组寻址：A[i]地址为：base+(i-low)*width
- 类型检查：类型转换、函数和运算符重载等
- 控制流：
  - 思路：画代码布局->写语义规则，注意 S 有 code 和 next 属性，B 有 true 和 false 属性都要规定
  - 循环注意可能需要添加 begin=newlabel()，以及 S1.next=begin 什么的问题！因为这样才能“循环”。反正一定要先画代码布局，帮助自己理清思路。
  - 控制流语句 B 没有 code 属性，但是布尔表达式里面 B 要考虑 code 属性
  - 布尔表达式
    - 带穿越的语义规则
      - B->B1||B2

$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \,||\, B_2.code \\
&\quad\ \textbf{else } B_1.code \,||\, B_2.code \,||\, label(B_1.true)
\end{aligned}
$$

      - B->B1&&B2
        - B1.false=if(B.false=fall)newlabel()elseB.false
        - B1.true=fall
        - B2.true=B.true
        - B2.false=B.false
        - B.code=if(B.false=fall)thenB1.code||B2.code||label(B1.false)elseB1.code||B2.code
- 回填！（样题有，但刚才不会，一定要搞懂！）
  - B.true/false：继承属性，goto 的跳转目标
  - B.truelist/falselist：综合属性，指 goto 所在的指令位置的列表
  - 语句的综合属性：S.nextlist
  - 辅助函数
    - Makelist(i)：创建一个只包含 i 的列表
    - Merge(p1,p2)：将 p1 和 p2 指向的列表合并
    - Backpatch(p,i)：将 i 作为目标标号插入到 p 所指列表中的各指令中
  - 回填与非回填方案的比较：
    - true/false 赋值，在回填方案中对应 list 赋值或 merge
    - 原生成 label 的地方用 M 记录相应代码位置，M.instr 是 label 标号
    - 原方案生成 goto B1.false，现在 goto M.inst
    - 回填时生成指令坯，然后加入相应的 list
    - 原来跳转到 B.true 的指令，现在被加入到 B.truelist 中
  - 回填写产生式应该考虑四种情况？
    - 求值？回填？merge？额外的 goto 语句？
    - 例子

1) $S \rightarrow \textbf{if}\,(\,B\,)\,M\,S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\,(\,B\,)\,M_1\,S_1\,N\,\textbf{else}\,M_2\,S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

3) $S \rightarrow \textbf{while}\,M_1\,(\,B\,)\,M_2\,S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$gen('\texttt{goto}\ M_1.instr)$; }

4) $S \rightarrow \{\,L\,\}$ { $S.nextlist\ =\ L.nextlist$; }

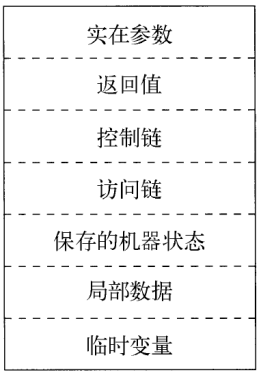
5) $S \rightarrow A\ ;$ { $S.nextlist\ =\ \textbf{null}$; }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

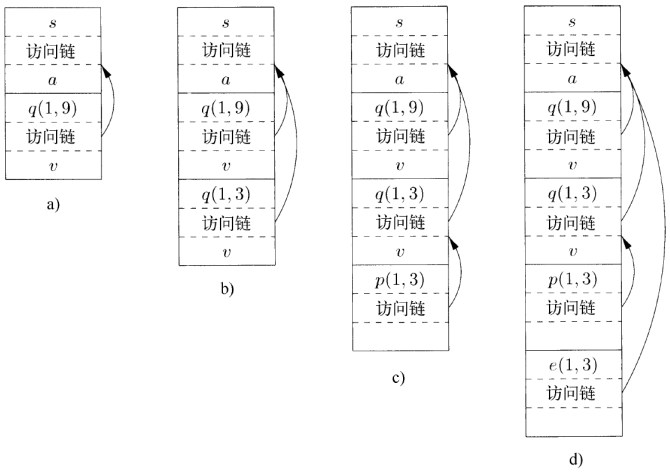
第七章 运行时环境

- 存储管理
  - 栈管理
    - 传递的值（实在参数、返回值）放在开始；固定长度的项（控制链、访问链、保存的机器状态）放中间；不知道大小的项（局部数据、临时变量）放尾部



    - 非局部数据的访问
      - 嵌套深度：不内嵌的过程嵌套深度为 1，嵌套在深度 i 的过程中的过程的嵌套深度为 i+1
      - 访问链：访问链访问一次就是嵌套深度-1
        - 设深度为 np 的过程 p 访问变量 x，而变量 x 在深度为 nq 的过程中声明，那么从当前活动记录出发，沿访问链前进 np-nq 次找到的活动记录中的 x 就是要找的变量位置
        - 如下，想解释的就是 exchange 嵌套深度 2，partition 嵌套深度 3，因此从 partition 前进 np-nq+1=3-2+1=2 次达到的 sort 的活动记录就是 exchange 直接指向的对象。其余比较好理

解，就是直接嵌套的往嵌套深度-1 的最近的那个指就行。



- 显示表：数组 d 为每个嵌套深度保留一个指针
  - 指针 d[i]指向栈中最高的、嵌套深度为 i 的活动记录。
  - 如果程序 p 中访问嵌套深度为 i 的过程 q 中声明的变量 x，那么 d[i]直接指向相应的（必然是 q 的）活动记录
  - 维护很好理解，就是多一个同层的就指最下的，返回就沿访问链往上一个

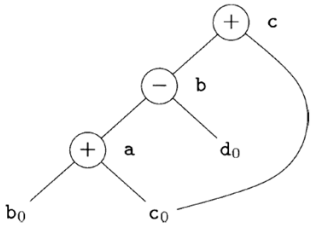
- 堆管理
  - 评价存储管理器的特性：空间效率、程序效率、低开销
- 垃圾回收
  - 引用计数：每个动态分配的对象附上一个计数：记录有多少个指针指向这个对象，当一个对象的引用计数为 0 时，该对象就会被删除
    - 对象分配：引用计数=1
    - 参数传递：引用计数+1
    - 引用赋值：u=v：u 指向的对象引用减 1、v 指向的对象引用加 1
    - 过程返回：局部变量指向对象的引用计数减 1
    - 如果一个对象的引用计数为 0，在删除对象之前，此对象中各个指针所指对象的引用计数减 1
  - 垃圾回收器的设计目标：总体运行时间、空间使用、停顿时间、程序局部性
  - 改变可达对象集合的四种基本操作：对象分配、参数传递和返回值、引用复制、过程返回
  - 引用计数垃圾回收的缺点：**不能回收不可达的循环数据结构、并且它的开销较大**
  - 标记清扫：先标记，从设置根集引用的每个对象 reached=1 开始，遍历所有可达对象，可达便设 reached=1 并放入 Unscanned 列表，循环直到 Unscanned 列表为空；再清扫，遍历堆区释放所有 reached 位为 0 的内存块。
    - 优化：用一个列表记录所有已经分配的对象，不可达对象等于已分配对象减去可达对象

```
1)  Scanned = ∅;
2)  Unscanned = 在根集中引用的对象的集合；
3)  while (Unscanned ≠ ∅) {
4)          将对象从 Unscanned 移动到 Scanned;
5)          for (在 o 中引用的每个对象 o′) {
6)                  if (o′ 在 Unreached 中)
7)                          将o′从Unreached 移动到Unscanned中;
            }
    }
8)  Free = Free ∪ Unreached;
9)  Unreached = Scanned;
```

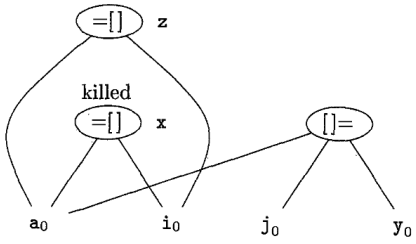
第八章 代码生成

- 寻址模式
  - 变量 x：指向分配 x 的内存位置
  - a(r)：地址是 a 的左值加上 r 中的值
  - constant(r)：寄存器中内容加上前面的常数即其地址
  - *r：寄存器 r 的内容为其地址
  - *constant(r)：r 中内容加上常量所指地址中存放的值为其地址
  - 常量#constant
- 实例：要懂数组、指针这些怎么写
  - b=a[i]
    - LD R1, i
    - MUL R1, R1, 8
    - LD R2, a(R1)
    - ST b, R2
  - a[j]=c
    - LD R1, c
    - LD R2, j
    - MUL R2, R2, 8
    - ST a(R2), R1
  - x=*p
    - LD R1, p
    - LD R2, 0(R1)
    - ST x, R2
  - *p=y
    - LD R1, p
    - LD R2, y
    - ST 0(R1), y
  - if x<y goto L
    - LD R1, x
    - LD R2, y
    - SUB R1, R1, R2
    - BLTZ R1, M
- 指令代价：1+寄存器分量寻址（0）/内存分量寻址（1），就是 1 或 2
- 静态分配使语言受到什么限制？递归的函数调用无法支持，例如 p 调用 p，则返回地址和数据无法区分

- 基本块和流图
  - 划分基本块算法：以下指令之间的指令从前到后一个首指令的前一个指令是一个基本块
    - 第一个三地址指令
    - 任意一个条件或无条件转移指令的目标指令
    - 紧跟在一个条件/无条件转移指令之后的指令
  - 确定活跃性（注意 2、3 顺序!!! 假设 x=x+z，此时 x 应活跃）
    - 从 B 的最后一个语句开始反向扫描
    - 对于每个语句 i：x=y+z
      - 1 令语句 i 和 x、y、z 的当前活跃性信息/使用信息关联
      - 2 设置 x 为不活跃、无后续使用
      - 3 设置 y 和 z 为活跃，并指明它们的下一次使用为语句 i
  - 流图
  - **基本块的 DAG 表示（重要）**
    - 例子



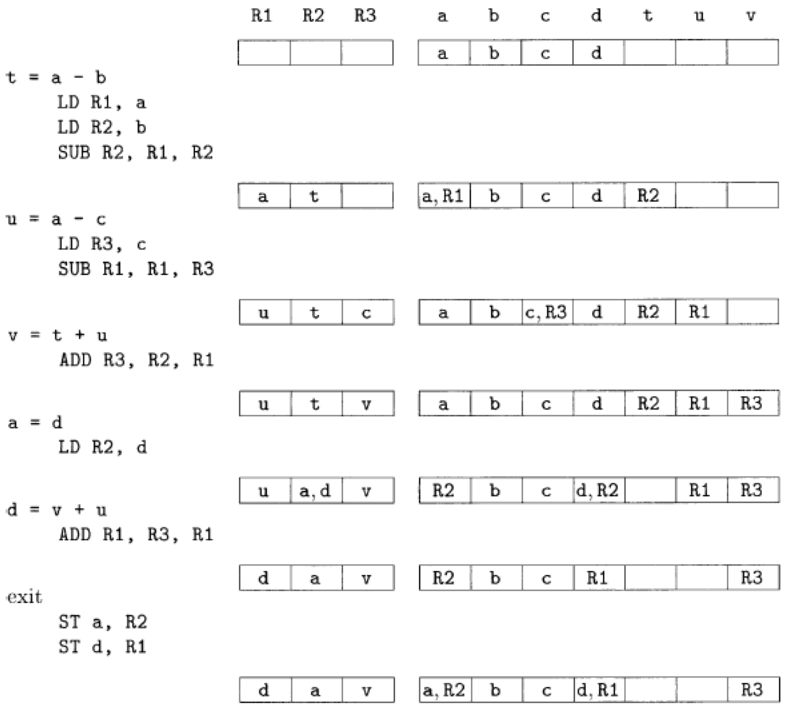
    - DAG 作用？
      - 消除局部公共子表达式（建立结点 M 前看是否有 N 与 M 具有相同的运算符和子结点和顺序）
      - 消除死代码（递归消除没有附加活跃变量的根结点）
      - 重组基本块等
      - 代数恒等式优化：消除计算步骤（x+0=0+x=x），强度消减（x^2=x*x），常量合并（2*3.14 用 6.28 替换）
      - 数组引用的表示：
        - x=a[i]的“=[]”和 a[j]=y 的“[]=”符号
        - 例如：
          - x = a[i]
          - a[j] = y
          - z= a[i]
          - a[j]=y 杀死所有当前已经建立的，值依赖于 a0 的结点
            一定要标出“被杀死”!



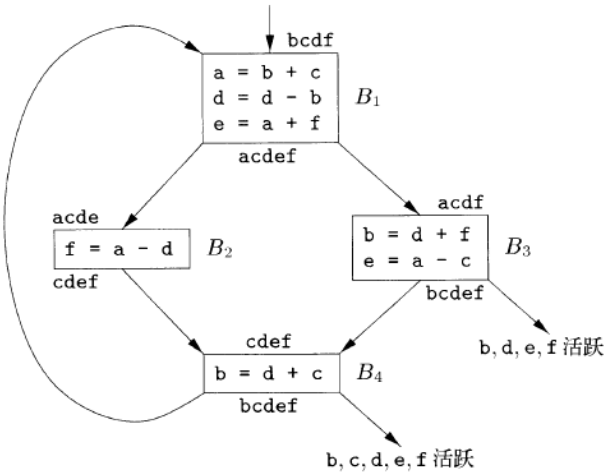
      - 指针，同数组引用考虑“被杀死”
    - 死代码和不可达代码：

- 死代码：计算得到的值不会被使用的指令
- 不可达代码：永远都不会达到的代码，即根本不会计算

- 一个简单的代码生成器
  - 基本块代码生成的隐含假设
    - 变量使用前要先 Load
    - 活跃变量在出口处要 Store
  - 寄存器描述符：跟踪各个寄存器都存放了哪些变量的当前值
  - 地址描述符：某个变量的当前值存放在哪个或哪些位置(包括内存位置和寄存器）上
  - 注意：基本块的收尾：如果变量 x 在出口处活跃，且 x 现在不在内存，那么生成指令 ST x, Rx
  - 例子：注意处理 x=y 时，我们总是让 Rx=Ry；注意最后要保存出口活跃的变量 a 和 b

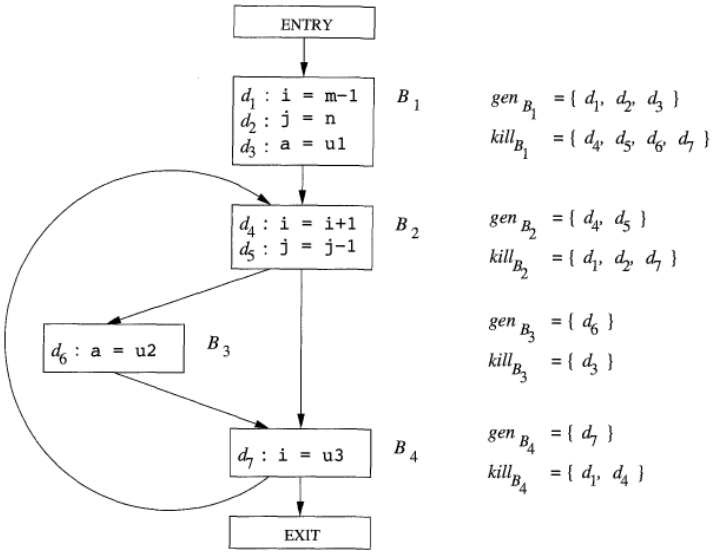


- 窥孔优化：使用一个滑动窗口（窥孔）来检查目标指令，在窥孔内实现优化
- 寄存器分配与指派：全局寄存器分配与优化
  - $\sum_{L\text{中的全部基本块}B} use(x,B) + 2 * live(x,B)$
  - x 在 B 中被定值之前就被引用的次数+2*出口处活跃并在 B 中被赋予一个值，则 live(x,B)=1，否则 0
  - 例如：
    - 下图中 a/b/c/d/e/f，分别可以节约 1+1+2=4/1+2+2=5/1+1+1=3/1+1+1+1+2=6/2+2=4/1+1+2=4 个单位成本
    - 所以为 a b d 指派三个全局寄存器 R0 R1 R2



第九章 机器无关优化——综合题，切基本块构造流图+数据流分析

- 优化的来源：全局公共子表达式；复制传播（复制语句 u=v 后尽可能用 v 来替代 u）；死代码消除（消除计算结果不会被使用的语句）；代码移动（把循环不变表达式移动到循环入口之前计算）；归纳变量（每次对 x 的赋值都使得 x 的值增加 c，那么 x 就是归纳变量，可以将对 x 的赋值操作改为增量操作）和强度消减（对循环变量的测试可以改为对别的临时变量的测试）
- do not 做激进的优化，因为用可能改变程序含义的代价来加快代码速度是不可接受的
- 数据流分析：到达定值分析；活跃变量分析；可用表达式分析；
  - 域：所有可能的数据流值的集合
  - 基本块内：前向：$IN[B]=f_B(OUT[B])$，逆向：$OUT[B]=f_B(IN[B])$
  - 到达定值：是否存在任何路径有信息==>并集∪
    - 如果 d 到达 p，那么在 p 点使用的值就可能是由 d 定值的
    - 对于可能定值的情况，如何做安全？
      - 确定变量 x 在某个程序点是否常量——假设所有可能定值都能到达
      - 确定变量是否先使用后定值（未初始化就使用）——假设所有可能定值都不能到达
    - $f_B(x)=gen_B \cup (x-kill_B)$
    - gen 和 kill 的例子



- 解法：
  - OUT[ENTRY]=空集
  - OUT[B]=$gen_B \cup (IN[B]-kill_B)$（先算 kill 再算 gen）
  - IN[B] = $\cup_{P\text{ 是 B 的前驱基本块}}$OUT[P]

■ 活跃变量分析（是否存在任何路径有信息==>并集∪）
- use_B：B 中使用先于定值（GEN）
- def_B：B 中定值先于使用（KILL）
- 解法
  - IN[EXIT]=空集
  - IN[B]=$use_B \cup$(OUT[B]-def_B)
  - OUT[B] = $\cup_{S\text{ 是 B 的后继基本块}}$IN[S]

■ 可用表达式（结论是否在所有的路径上都成立/一个表达式在一个基本块的**所有**前驱结尾可用，它才在这个基本块的开头可用==>交集∩）
- 主要用途：寻找全局公共子表达式
- 生成-杀死
- 解法：
  - OUT[ENTRY]={}
  - OUT[B]=$e\_gen_B \cup$(IN[B]-e_kill_B)
  - IN[B] = $\cap_{P\text{ 是 B 的前驱基本块}}$OUT[P]

● 三种数据流方程的总结

	到达定值	活跃变量	可用表达式
域	Sets of definitions	Sets of variables	Sets of expressions
方向	Forwards	Backwards	Forwards
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
边界条件	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
交汇运算(∧)	∪	∪	∩
方程组	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
初始值	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

- 部分冗余消除
  - 允许进行两种操作：在关键边上增加基本块；进行代码复制
- 循环的优化
  - 支配结点：如果每一条从入口结点到达 n 的路径都经过 d，我们就说 d 支配(dominate)n
  - 支配节点数据流方程：注意支配节点要在所有的路径上都成立==>交集

	支配结点
域	The power set of N
方向	Forwards
传递函数	$f_B(x) = x \cup \{B\}$
边界条件	$\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
交汇运算($\wedge$)	$\cap$
方程式	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
初始化设置	$\text{OUT}[B] = N$

  - 例子：一遍一遍计算直到这些都收敛

$$
\begin{aligned}
D(4) &= \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\} \\
D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\} \\
D(6) &= \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\} \\
D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\
&= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\} \\
D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\} \\
D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\} \\
D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}
\end{aligned}
$$

  - 回边和可归约性
    - 回边：若边 a->b，则头 b 支配了尾 a
    - 可归约性：所有后退边都是回边-可归约的
    - 深度：后退边数目的最大值
    - 自然循环：易错！——优化最内层的循环即可
      - 例子：
        - 回边：10->7：{7,8,10}
        - 回边：7->4：{4,5,6,7,8,10}
          - 包含了前面的循环
        - 回边 4->3,8->3：因为同样的头，所以同样的结点集合 {3,4,5,6,7,8,10}
        - 回边 9->1：整个流图