编原算法汇总

计83 饶淙元 2020年1月5日

Last Update: 2021年1月6日15:49:03

以下为我个人对编原算法的一些总结和个人解读，本来想早点儿发出来供有需要的同学复习使用，但是进度滞后了，不知道有没有人看……如果发现有和讲义冲突之处欢迎指出，谢谢~

1. LL(1)

1.1 动机

Left（从左到右扫描输入串）Leftmost（非终结符最左推导）1（向前查看一个单词）

小tip：之所以是“单词”而非“字母”，是因为理论上我们讨论文法时，针对的都是词法处理后的 Token。对文法而言可以理解为终结符（但相比终结符，说“单词”时不算ε）。

算法诞生逻辑：

1. 考虑自顶而下分析推导，朴素方法下每一步推导无法确定对哪一个终结符用哪一个产生式，因此复杂度很高。
2. 人为约定LL，即使用最左推导，从左到右契合模式串，失败则回溯，从而确定了终结符，只是不确定产生式，复杂度大减。
3. 为了使得推导过程变成确定的，查看后续K个单词，确定应该使用哪一个产生式，则有LL(K)分析

我们重点学习的是最简单的 LL(1) 分析。如果不学习系统方法，用观察法来完成 LL(1) 分析，在产生式不多时也是可以完成的，因此考试时可以将得到的结果和观察法进行校验。

1.2 First & Follow

为了完成 LL(1) 分析，我们希望知道每个非终结符所推导出的所有串里可能的首单词，进而我们可以根据推导式的右边得知每个非终结符后面可能紧邻的单词，前者即 First 集合，后者即 Follow 集合。

例如，对如下文法

$(1)S \to AB$
$(2)A \to Da|\varepsilon$
$(3)B \to cC$
$(4)C \to aADC|\varepsilon$
$(5)D \to b|\varepsilon$

注意以下为我对标准算法做个人解读得出的观察法，思路上与标准算法略有不同

首先分析 First。以 S 为例，由 (1) 可见 S 的首单词取决于 A 的首单词，由 (2) 可见 A 的首单词是 ε 或 D 的首单词，从 (5) 可以看出 D 的首单词是 b 和 ε，但注意当 D 的首单词是 ε 时，(2) 会推导出 a，A 并不会利用 Da 得到首单词 ε，因此 A 的首单词即有 First(A) = {a,b,ε}。同理对 (1) 而言，A 取 ε 时 S 的首单词就成了 B 的首单词，而从 (3) 可以看出 B 的首单词一定是 c，所以 First(B)={c}, First(S) = (First(A) - {ε}) ∪ First(B) = {a,b,c}。进一步，**如果 $\varepsilon \in$ First(B)**，那么 ε 会真正出现在 S 中，这是因为此时推导式右边每一项都有 ε，因此 S 首单词真的可以是 ε。

然后分析 Follow。由于一些非终结符所推导出的串可能是结尾串，它们后面没有新的单词，这种情况下记为 #，即初始情况下就有 # ∈ Follow(S)。以 A 为例进行分析，我们只需要找 A 在所有产生式右边出现时，A右边的符号的 First 集合即可，在 (1) 中将 First(B) = {a,b,c} 并入 Follow(A) 中；同时注意到 B 后什么都没有，即 B 后的符号为 ε，而 First(ε)={ε}，这种情况下不能将 ε 放入 Follow(B) 中（这没有意义），而应该考虑到 B 的下一个单词和产生式左边的 S 的下一个单词相同，因此将 Follow(S) = {#} 并入 Follow(B) 中，这又是 B 的唯一来源，故有 Follow(B)={#}。从(4)可知要将 First(D)={b, ε} 并入 Follow(A) 中，同样 ε 不应该放入，改为放 First(C) = {a, ε}，这里的 ε 仍然不该放，而 C 已经是结尾，故讲产生式左边的 C 的 Follow(C) 并入 Follow(A)，而从 C 的唯一来源 (3) 同理有 Follow(C) =Follow(B)={#}，从而由 (4) 实际得到 {a, b, #} 并入 Follow(A)。最终 Follow(A) = {a,b,c,#}。

可以验证，讲义上的迭代法和上述观察法得出结果一样。

总的来说，注意 ε 的处理，对 First 来说当且仅当产生式右边整个能推导出 ε 时才会有 ε 在 First 集合中（当然，First(ε)={ε}）；对 Follow 来说永远不会出现 ε，当某个非终结符 B 的后续符号能推导出 ε 时，就该将产生式左边的非终结符 A 的 Follow(A) 并入 Follow(B)。

观察法end

讲义中的迭代法略微拗口一些，迭代法中将产生式右边的非终结符右边的后缀也作为一个整体分析了 First。如果选择使用讲义中的迭代法，请自行记忆~~（背书是不可能背书的，反正不会考算法填空）~~

1.3 PS

构造出 First 和 Follow 后，距离投入使用还差最后最后一个过渡：Predictive Set（预测集 PS），它代表选择一个产生式所对应的下一个单词。对任意产生式 $A \to \alpha$（这里的 α 代表产生式右边的所有符号），则

1. 如果 First(α) 中没有 ε，有 PS($A \to \alpha$)=First(α)
2. 否则， 有 PS($A \to \alpha$)= (First(α) - {ε}) ∪ Follow(A)

这个理解比较容易，在 LL(1) 分析中，我们需要知道从左到右看到一个单词、且已知最左推导的非终结符为 A 时，应该选择哪一个产生式，PS 则是看对每一个产生式，可以匹配到的下一个单词是什么，构造了产生式到终结符的映射。

定义 LL(1) 文法，这类文法一定能用 LL(1) 分析法进行分析，它的判定条件是，对于非终结符的任意两个产生式 $A \to \alpha|\beta$，有 PS($A \to \alpha$) ∩ PS($A \to \beta$)=ϕ，可能的终结符和产生式之间可以构成映射。

1.4 递归下降分析程序

分析程序即写（伪）代码进行递归分析，参考讲义即可，基本格式如下，每一个 case 内对非终结符递归调用其他 Parse，对终结符 x 用 `MatchToken(x)` 即可，注意老师提供的伪码里是允许是 `case b,c:` 这样的写法的。

```
void ParseA()
{
    switch (lookahead) {
        case PS(A→u1):
            /* code to recognize u1 */
            break;
        case PS(A→u2):
            /* code to recognize u2 */
            break;
            ...
        case PS(A→un):
            /* code to recognize un */
            break;
        default:
            printf("syntax error \n");
```

```
            exit(0);
    }
}
```

1.5 表驱动分析程序

我们使用一个固定的分析表（即 PS 所得）和一个动态的下推栈完成，分析表即表示在最左非终结符为A，下一个单词为a时，所使用的产生式。

算法：

1. 初始化下栈中push一个标志符号#和一个开始符号 S，开始扫描剩余串
2. 栈顶为非终结符时，根据剩余串首理应可以找到产生式，pop非终结符，push产生式右边的符号，注意入栈顺序从右到左
3. 栈顶为终结符时，输入串首理应和它相同，pop终结符并将将剩余串向右扫描一位
4. 栈顶为#时，剩余串理应也是#，完成推导
5. 如果上述2~4任何一步无法进行，则报错

1.6 文法变换

有些文法无法用 LL(1) 分析，但是经过变换可以解决：

1. 包含左递归（产生式右边的第一个符号和产生式左边符号相同，如 S->Sa，也包括间接左递归 S->B, B->Sb，即几个推导式联立可以得到一个左递归）的式子无论 K 取多大，都无法使用 LL(K) 完成确定性分析，因此要消除左递归
2. 包含左公因式（同一个非终结符的几个产生式右边有相同前缀，如 S-> abS, S->abc），此时需要一个大于1的K来完成 LL(K) 分析

但系统使用它们之前，需要先处理两个问题，可以用直觉分析验证：

1. 存在左递归（产生式右边的第一个符号和产生式左边符号相同，如 S->Sa，也包括间接左递归 S->B, B->Sb，即几个推导式联立可以得到一个左递归）的式子无论 K 取多大，都无法使用 LL(K) 完成确定性分析，因此要

   存在左公因子（两个推导式左侧符号相同，右侧有公共前缀，如 S->abB, S-> abC）会导致 LL(K) 的 K 增大，

消除直接左递归的方法是对于 $P \to Pa_1|Pa_2|\ldots|Pa_m|b_1|b_2|\ldots|b_n$ 可以改写为 $P \to b_1Q|b_2Q|\ldots|b_nQ$ 与 $Q \to a_1Q|a_2Q|\ldots|a_mQ|\varepsilon$。

消除间接左递归的方法：

1. 为非终结符任意排序 $A_1, A_2, \ldots, A_n$
2. 从前往后扫描每一个非终结符 A_i，每次扫描中找到所有“回头递归”的，即 $A_i \to A_j \ldots$，其中 i > j，此时将 A_j 代换为它在左边的所有产生式，由算法的先后顺序知代换结果的首符号一定不会是 $A_k (k \leq j)$，反复代换直到没有“回头递归”，然后再消除直接递归，结束本次扫描

左公因式直接提取即可，处理方法比较符合直觉，例如对 $P \to \alpha\beta|\alpha\gamma$ 可以变换为 $P \to \alpha Q$ 和 $Q \to \beta|\gamma$

1.7 问题探讨

不含左递归和左公因式的文法也不一定是 LL(1) 文法。

非 LL(1) 文法也可能能用 LL(1) 分析法，比如对于所有非确定的情况都手动规定（优先）使用哪一个产生式。

2. LR分析

2.1 动机

Left（从左到右扫描输入串）Rightmost（最右推导）

算法诞生逻辑：

1. 考虑自底而上分析，即将输入序列逐步规约得到开始符号 S，朴素算法无法确定该用哪个产生式处理哪段序列，复杂度高
2. 每次规约时都选择一段“可规约串”，能够降低试错成本
3. 如果能约定每次都用从左到右的第一个“一步可规约串”，则得到了确定性的算法

模糊定义（讲义没给定义，不见得准确，应该也不会考）：

- 句子：将开始符号 S 能推导出来的一个**终结符串**称为一个句子
- 句型：将开始符号 S 能推导出来的一个**串**称为一个句型
- 右句型：将开始符号 S 能**最右推导**出来的一个**串**称为一个右句型

定义：

- 短语：取一个句型 $\alpha A\beta$ 中的非终结符 A 能**推导**出来的串 w，称为句型 $\alpha w\beta$ 相对于 A 的短语（注意短语是句型的子串）

- 直接短语：取一个句型 $\alpha A\beta$ 中的非终结符 A 能**一步推导**出来的串 w，称为句型 $\alpha w\beta$ 相对于 A 的直接短语
- 句柄：取一个**右句型** $\alpha Aw'$ 中的非终结符 A 能**一步推导**出来的串 w，称为句型 $\alpha ww'$ 相对于 A 的句柄

显然，短语、直接短语、句柄都可以进行规约，其中句柄是当前句型从左到右最先出现的“一步可规约串”，且短语包含直接短语包含句柄。具体例子见讲义。

2.2 移进-规约分析

LR分析采用 移进-规约（shift-reduce）分析技术，我们需要一个下推栈，一个状态控制引擎（有限状态机？），引擎根据当前状态、下推栈内容、剩余输入序列来确定动作：

- Reduce：对栈顶短语规约
- Shift：将输入序列移进一个单词
- Error：报告/恢复错误
- Accept：完成分析

我们需要掌握四种具体分析法，分别是 LR(0), SLR(1), LR(1), LALR(1)。

2.3 冲突

移进-规约冲突：不确定该移进还是该规约

规约-规约冲突：有多个规约式可用

注意：接受项目不会产生冲突，因为具体到后文的冲突定义中指明了是“移进项目”与“规约项目”，其定义见后

2.4 表驱动方法

对于涉及到的四类LR分析法，可以共用一个 LR 分析表，它包括 ACTION 表和 GOTO 表两部分。

- ACTION 表：ACTION[k, a] = rj/si/acc/err 告诉引擎在栈顶状态为 k，输入符号为 a 时该做什么（四个动作之一）
- GOTO 表：GOTO[i,A]=j 告诉引擎在用产生式 $A \to \beta$ 规约，栈顶状态为 i 时，push 状态 j （规约时会将 $|\beta|$ 个状态 pop 出）

表的定义也包含了表的使用方法（算法）。

下推栈选择普通的状态分析栈与选择带符号栈的分析栈基本没有区别，只是下推栈中是否一并写上符号，可读性高低的差别例如普通栈中某个时刻的栈可能为0,1,7,10,8,11，而对应的带符号的栈可能为0#, 1E, 7+, 10T, 8*, 11F，做题时如果考查也应该是完善表格，根据已有部分照搬格式即可。

2.5 LR(0)

Left（从左到右）Rightmost（最右推导）0（向前查看0符号）

为了方便定义一些算法的初态，我们引入新的开始符号 S'->S 对文法 G 进行扩充得到增广文法 G'。

（此处涉及到活前缀概念，但我没看出其应用场景和考点，跳过）

构造 LR(0) FSM 即可得到 LR(0) 分析表，为了构造 FSM，首先需要明确 LSM 的状态含义：一个 LR(0) 的 items set（项目集），一个 LR(0) item 是一个右端有圆点（表示当前所处位置）的产生式，如 $A \to x.yz$ 表示对这个产生式而言已经匹配 x 了。

项目可以分为四种：

- 移进项目：如 $A \to \alpha.a\beta$，表示期望移进 a 以继续
- 待约项目：如 $A \to \alpha.B\beta$，表示希望能够规约出 B 以继续
- 规约项目：如 $A \to \alpha.$，表示可以进行规约
- 接受项目：如 $S' \to S.$，表示开始符号能够规约了，完成分析

闭包（CLOSURE）：对于一个项目集 I，将其中每一个项目产生式右边的 . 紧邻的非终结符代入其所有产生式（项目），可以得到若干新的项目，将它们迭代放入 CLOSURE(I) 中。

FSM 构造的初态即 CLOSURE({S'->S})，之后根据每个项目集期望的输入绘制状态转移图即可。考试的时候大概率不会让从零开始画图，而是补全部分项目集。

完成 FSM 的状态转移图后即可得到 LR 分析表，考试通常只要求补全 LR 表，注意以下关系即可：

- 移进项目对应 ACTION 表的 Shift
- 待约项目对应 GO 表
- 规约项目对应 ACTION 表 Reduce
- 接受项目对应 ACTION 表的 Accept
- 其余空白处全部是 Error

如果没有 2.3 中提到的两种冲突，则得到的LR分析表是一个 LR(0) 表，对应的文法是 LR(0) 文法：

- 无移进-规约冲突：任何一个状态中不会同时有移进项目和规约项目

- 无规约-规约冲突：任何一个状态中不会有多个规约项目

2.6 SLR(1)

Simple（简单）Left（从左到右）Rightmost（最右推导）1（向前查看1符号）

LR(0) 中一行如果有规约，则全部是同一条规约，显得冗余而浪费，我们在此基础上进行优化：遇到冲突时，考虑规约所得非终结符的 Follow 集合，如果几个规约所得结果的 Follow 集合交集为空，则可以解决规约-规约冲突；如果移进对象不在任何规约项的 Follow 集中，则可以解决移进-规约冲突。

所得到的 SLR(1) 表在 LR(0) 表的基础上对规约进行了约束，只有当前输入符号在规约结果的 Follow 集中才规约，这也使得报错出现得更早（LR(0) 在不满足 Follow 时的报错至少晚一步规约）。

SLR(1) 表的无冲突条件和 LR(0) 自然有所不同：

- 无移进-规约冲突：任何一个状态中如果有移进项目期待移进终结符 x，则x不在该状态任何一个规约结果的 Follow 集中
- 无规约-规约冲突：任何一个状态中的任何两个规约项目的规约结果的 Follow 集无交集

2.7 LR(1)

Left（从左到右）Rightmost（最右推导）1（向前查看1符号）

SLR(1) 仅仅考虑了规约所得终结符的 Follow，没考虑句柄的 Follow，如果对整个句柄考虑 Follow，则可以有更多信息来解决冲突。

为此，我们扩充 LR(0) 项目以得到 LR(1) 项目，LR(0) 项目引入了 . 表示已匹配部分，此处再引入句柄 Follow 集（称作“向前搜索符”）写到产生式右边，如 $A \to \alpha . \beta, a/b/c$ 表示这一产生式在规约后后面的输入必须是 {a, b, c} 之一。特别的，结束符号为 # 作为 S'->S 的后续输入（也即句柄 S 的 Follow）。

由于项目变了，因此闭包算法有所差别，需要在每次代入非终结符 A 产生式时，将原式中自 A 之后的后缀的 First 以及原式的向前搜索符（后缀为空时用得到）考虑在内。例如如果已有项目 $A \to Q . BCD, x$，产生式 $B \to b$，First(CD) = {c, ε}，则迭代后项目闭包中加入 $B \to . b, c/x$，其中 c 来自 First(CD)，x 是因为 First(CD) 有 ε 故继承了原项目的 x。

CLOSURE 改变后规约的限制增大了，其他和 SLR(1) 相同，从而解决了部分冲突。

LR(1) 表的无冲突条件和 SLR(1) 有所不同：

- 无移进-规约冲突：任何一个状态中如果有移进项目期待移进终结符 x，则x不在该状态任何一个规约项目的向前搜索符中
- 无规约-规约冲突：任何一个状态中任何两个规约项目的向前搜索符不相交

2.8 LALR(1)

LookAhead（向前搜索）Left（从左到右）Rightmost（最右推导）1（向前查看1符号）

定义：

- 芯：将 LR(1) 项目的向前搜索符去掉的结果（即为对应的 LR(0) 项目）称为它的芯
- 同芯状态：LR(1) FSM 的两个状态如果只考虑芯发现它们完全一样，那么它们是同芯状态

将 LR(1) 文法的 LR(1) FSM 的同芯状态合并，如果没有产生规约-规约冲突，则得到了一个 LALR(1) FSM，原文法是一个 LALR(1) 文法。

注：这不会带来新的移进-规约冲突，因为移进与否在合并中没有受到影响，如果有冲突则合并前已经冲突了。

LALR(1) 分析放弃了一些状态，弱于 LR(1) 分析，但是强于 SLR(1) 分析（当然LR(0)是最弱、适用范围最小的），因为它的向前搜索符仍然是规约结果的 Follow 集的子集，删除了 SLR(1) 里的一些规约项。

另外 LL(1) 的适用范围和上述方法适用范围均有交集，包含于 LR(1) 分析。

2.9 二义文法

二义文法可以通过约定优先级和结合性来得到 LR 分析器（考点），需要掌握某种规定是如何约定的（例如规定谁的优先级高，左结合还是右结合）

3. 语义计算

3.1 属性文法

属性文法即给上下文无关文法的每个符号加上一个属性，然后在每个生成式的后面附带上属性的计算与动作（通常是最后判断是否接受），写法上用大括号括起来，中间写用分号隔开、:= 赋值的伪码。

- 综合属性：产生式左边的符号的属性，由它的孩子的属性计算而来
- 继承属性：产生式右边的符号的属性（但不见得都是由父亲、兄长而来）

实现语义计算（即计算出所有属性值）有树遍历方法与单遍方法，前者非常复杂大概不会考，后者需要重点掌握。

单遍方法有一定限制，课程只要求掌握两类特殊文法的单遍遍历：

- S-属性文法：只包含综合属性
- L-属性文法：产生式右边的符号的继承属性只来自于父亲的继承属性与长兄的属性

3.1.1 自下而上

S-属性文法可以自下而上计算，采用 LR 分析即可，综合属性的计算恰好在 LR 分析规约时，因此将前文 LR 分析中的栈在状态栈、符号栈的基础上再加一个语义栈用于记录综合属性（注：一般只涉及一个综合属性，如果有多个可以有多个语义栈），掌握 LR 分析后加个语义计算不难。

3.1.2 自上而下

L-属性文法可以考虑自上而下（递归下降）计算，采用多叉树的后根遍历顺序，在从左到右访问完所有孩子后，再计算父亲的综合属性，访问孩子时计算其继承属性，由于L-属性文法继承属性一定来自父亲和长兄，因此这个朴素的算法正确性没有问题。这个过程本身的复杂度在于构造出语法生成树，可能需要用到 LL(1) 分析。

3.2 翻译模式

翻译模式形式和属性文法类似，但是用大括号括起来的语义规则可以放到产生式右端的任何空隙（如果右端有n个符号，则有n+1个空隙）。

为了保证正确性，翻译模式需要考虑大括号写得是否恰当，它也有类似于S-属性文法和L-属性文法的情况：

- 类似于S-属性文法，如果一个翻译模式只有综合属性，则直接将所有语义规则写到产生式最右边即可，此时它和S-属性文法没啥差别

- 仿照L-属性文法，如果翻译模式有继承属性，则生成继承属性的语义规则的位置必须放到拥有者之前，且其中不能出现拥有者之后的符号的属性

3.2.1 自上而下

考虑用自上而下的 LL(1) 递归下降处理翻译模式，这和自上而下处理L-属性文法其实没什么区别，只是属性文法部分默认 LL(1) 分析已经完成了，只需要计算属性，而翻译模式需要连带着 LL(1) 分析一并进行，因此给出的代码是在 LL(1) 分析的代码的基础上补充属性部分，对于非终结符 A，ParseA 的输入参数为 A 的继承属性，返回值为 A 的综合属性。

但是要做 LL(1) 分析就得考虑左递归的消除，左递归消除时得连带着语义规则进行改变，例如

$A \to A_1 Y \ \{A.a := g(A_1.a, Y.y)\}$
$A \to X \ \{A.a := f(X.x)\}$

可以消除左递归后变为

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

注意此处增加了新的非终结符 R，并给他准备了继承属性（输入）i 和综合属性（输出）s，用于完成计算与结果传递。

3.2.2 自下而上

自下而上和 L-属性文法的自下而上 LR 分析类似，但是需要解决几个问题

- 去掉嵌套在中间的语义动作（即中间只有赋值没有操作，末尾的操作可以在规约时完成，中间不好处理触发时机）
- 在分析栈中考虑继承属性的读写（不再是简单地和符号栈、状态栈保持一致）
- 用综合属性替代继承属性

方法分别如下：

- 新增一个生成 ε 的非终结符（下称致空非终结符），用它来承担动作
- （重要）借用 val 数组下标相对于栈顶 top 的偏移来定位，例如对 P->AB {P.s := A.i}，写入 A 的综合属性时，应该写到 val[top - 1]（因为符号栈要 pop 两下 push 一下），A 和 B 的继承属性不能直接获取，需要考虑它们源自于哪个综合属性，然后直接定位到该处，例如此前有 Q -> M N{A.i := M.s} A，那么 A 往前两个就是M，则前述综合属性的获取应该是 val[top - 1] := val[top - 3]。如果继承属性不是直接复写（赋值）自一个综合属性，则要加一个致空非终结符来承担；如果

继承属性有多个产生式来源，来源距离不同（比如一个间隔1，一个间隔2），则要么为间隔短的增加致空非终结符来缓冲，要么强行增加致空非终结符统一加到目标前面。另外特别注意，如果是形如 $A \to \varepsilon\{A.attr := 1\}$ 的规约，此时要赋值 A 的综合属性 attr，因为 ε 没有位置，规约后会多一个符号，故综合属性的获取是 val[top + 1] := 1。
- 综合属性替代继承属性主要是为了解决继承属性需要访问弟弟的属性的情况，为此需要改写文法，比较复杂，没有介绍系统方法，大概不考

4. 语义分析

语义分析在考试中主要是做类型检查，即表达式得出类型，语句得出 ok / type_error，这部分主要利用 L 翻译模式进行（引入 break 前 L 模式主要用于变量声明，多数语句的处理其实还是 S 模式够用，往年考试出现过 S 模式做类型检查，并要求同样用 S 模式检查 E?E:E），虽然看上去语义规则比较复杂，但是基本逻辑很简单：

- 表达式要类型匹配
- 需要布尔逻辑时类型确实是 bool
- 每条语句都 ok 了才是 ok
- 常量表达式的类型是自己，标识符的类型查符号表
- 有 break 时需要注意在循环里才能 break （所以加了个 break 继承属性标志是否在循环里）

这部分占的篇幅也不大，考试时可能会加入新的语句要求写语义规则，本质上还是考查 L 翻译模式的掌握。

5. 中间代码生成

考试主要考 TAC （三地址码），这里又分为语法制导翻译和拉链与代码回填两种技术。

5.1 语法制导翻译

这里采用 L 翻译模式，要求借助继承属性

1. 赋值与计算：这部分用 S 翻译模式即可，生成代码时用连接符号||把各个部分的代码先摆出来（理论上先后顺序可以调）然后再用 gen 生成它们之间的计算语句；需要注意计算结果一般是右值，需要用

newtemp 新建一个临时变量保存结构（但是小括号和标识符作为左值直接传递）
2. 说明语句：这里需要完成符号表，因此要弄清楚每个变量的类型、偏移，以及为了算偏移而增加的继承属性宽度
3. 数组说明：数组说明略微复杂一点，而且课上例子给的是非常反人类的（常用语言数组范围都是 [0, n) 或 [1, n]，我还没见过 [l, u) 的），似乎也没考过
4. 布尔表达式：如果直接用 1 和 0 完成布尔表达式计算则和赋值与计算差别不大，只需要主要逻辑表达式（id1 rop id2）需要用到 if id1 rop.op id2 goto 的分支来处理结果是 1 还是 0，另外nextstat 表示下一条 TAC 语句的位置，但由于在 gen 中用到它时，gen 的这条语句还没生成出来，所以此时 nextstat 就是指的要 gen 的这个语句
5. 布尔短路表达式：如果要考虑到布尔表达式的短路运算，则要用到 L 翻译模式处处考虑 true 标签和 false 标签，可以看看讲义里的图来辅助理解。下面的几种语句应该是基于这种模式设计的（否则布尔表达式里都没有跳转到 true 和 false）
6. 条件语句：if E then S 和 if E then S else S，参考讲义图片理解即可，别忘了在 else 之间加上 goto 到 S.next
7. 循环语句：while E then S语句，同样参考讲义图片理解，别忘了在 S 之后加上 goto 到 S.next (E 之前)
8. 顺序语句：即 S; S 语句，记得在中间打上 S1.next 标签即可
9. break语句：由于 break 的合法性已经在语义检查完成了，因此这里默认 break 是可行的，使用继承属性 break 传递了 break 的目标地址

后面几种涉及到标签的语句需要记住标签是“谁生成谁放置”，例如顺序语句中不能在 S2 后面放置 S.next（尽管老师给的示意图上有它），因为这个标签不是这个产生式生成的。

5.2 拉链与代码回填

这里是另一种中间代码生成技术，采用了完全的 S 翻译模式，即只有综合属性，产生式中间不会出现大括号（看上去舒服多了）。

这里用到了三种链，链表中的语句均是没有给出跳转目标的 goto 语句，语义上一条链中的 goto 希望跳到同一个悬而未决的地方：

- E.truelist：该链的目标是代表布尔表达式为真时的跳转对象
- E.falselist：该链的目标是代表布尔表达式为假时的跳转对象
- S.nextlist：该链的目标是 S 执行的出口（即执行完 S 后的下一行的标签）

本技术主要处理上一节 5~9 语句，核心思想即将有同样跳转目标的 goto 语句汇集起来，在明确了跳转位置后再用 backpatch 函数统一填入目标（行号）。汇集使用 merge 函数，例如对于或运算而言两个操作数任意一个为真都表示结果为真，因此有 E.truelist := merge(E1.truelist, E2.truelist)，但是如果前者为假就需要执行后者，因此中间会加入一个 M 用来记录后者的开始位置，填写到前者的 falselist 里。所以或运算的语句是：

$$
\begin{aligned}
E \to E_1 \vee M E_2 \{ & backpatch(E_1.falselist, M.gotostm); \\
& E.truelist := merge(E_1.truelist, E_2.truelist); \\
& E.falselist := E2.falselist\}
\end{aligned}
$$

$$
M \to \varepsilon \{M.gotostm := nextstm\}
$$

也需要注意 M.gotostm 的运行时机，它发生在 E1 的语义规则执行完毕，E2 的语义规则执行之前，此时 E1 对应的代码已经生成，因此 M 处所看到的 nextstm 恰好是将要到来的 E2 的代码。课件上有一页给出了 $a < b \vee c < d \vee e < f$ 的翻译示意，看懂了那一页应该就理解了这部分内容。

处理条件时需要注意 if E then M S N else M S 中所新增的非终结符，M 是为了记录跳转位置（nextstm）无需多数，这里的 N 比较特殊，它会跳转到 if else 结束的位置，并生成一条链，因此最终被 merge 到了综合属性 S.nextlist 中。

循环的逻辑也不麻烦，需要注意的是引入 breaklist 后，breaklist 总在 break 语句处生成，在 while 中被 merge 到 S.nextlist 中在其他地方传递（然后新的 breaklist 是空链）。讲义上在整个程序的入口生成式 $P \to D; SM$ 中将 S 的 breaklist backpatch 为 M.gotostm，这一举措我有所困惑，如果代码正确（通过了语义检查），那么传递到这里的 breaklist 必然是空链，不知道这么写是否是为了一种完备性。

6. 作用域

考试常考支持嵌套过程声明的静态作用域，这个可以参考 Python 函数内的作用域处理。

需要用到动态链和静态链，动态链即汇编中所学过的栈帧所形成的链（用于过程调用结束时回溯），静态链则是看函数在源代码中的位置，然后依次指向它的开作用域的每一层——静态链一定是动态链的子集（因此课件上画图时只需要话一个调用栈然后就直接标出了静态链），这里有一个隐含逻辑，过程调用时目标层级要么不大于自己，要么恰好比自己大1（我定义的嵌套过程的层级比我大1）。因此，如果用 `P::Q::R` 表示过程 P 中声明的过程 Q 中声明的过程 R（显然静态链是 R->Q->P，R 的层级为3），那么当它被调用时，调用者一定在 `P::Q` 过程中（可能是 `P::Q` 的嵌套过

程，甚至是嵌套过程的嵌套过程），也许调用者层级很深，但是我们可以不断溯源下去，由于函数调用最多只能加深一层，且第一个栈帧层级一定是1，因此最终总能找到某个栈帧层级为2（在那之前所有的过程都在 `P::Q` 中），且恰好是 `P::Q` 。

如果是动态作用域，则静态链没有意义，只需要动态链，此时寻找变量直接沿着动态链寻找即可，一般这种题目会直接问变量 x 在哪一行声明的。

7. 流图

7.1 基本概念

流图需要划分基本块，但是基本块的划分似乎没考过，一般都是给出了划分好的块并给出了流图，所以可以直接快进到流图算法。

首先给出几个定义，往年题也考过：

- 支配结点：如果从流图出发点到 n 必须经过 m，则 m 是 n 的支配结点（n 被 m 支配了），记作 m DOM n，特别的约定 n DOM n 恒成立
- 支配结点集：n 的所有支配结点的集合（由定义知至少有 n，所以非空），虽然流图是个有环图，不过从树的角度考虑可以认为里面的都是 n 的长辈
- 回边：如果 d 在 n 的支配结点集里（d 是 n 的长辈），但是存在一条有向边 n -> d（n 以下犯上），则称这条边为回边
- 自然循环：若 n->d 是回边，则从 d 出发、到 n 停止（中间不得再经过 d，但是可以多次经过 n）的所有路径上的点组成了 n->d 对应的自然循环。特别的，如果存在边 n->n，那么它对应的自然循环为 {n}

7.2 到达-定值方程

到达定值方程研究的是每一个定值语句（即给定变量一个指，近似理解成赋值语句，但不能划等号，比如 scanf("%d", &x); 也是定值语句）可能发挥作用的范围，研究对象称作“定值点”，可以近似理解为代码行号，但为了方便描述我下文直接称作定值语句，方程如下：

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$
$$IN[B] = \cup_{b \in P[B]} OUT[B]$$

其中 P[B] 指 B 的所有前驱。

- GEN[B]：B 中新生成的定值语句，且能作用到出口（即没有被新的定值语句覆盖）

- KILL[B]：B 之外能够到达 B 入口，又在 B 中被重新定值覆盖掉的语句（显然，它是 IN[B] 的子集，且和 GEN[B] 无交集）
- IN[B]：所有能到 B 入口处的定值语句
- OUT[B]：所有能到 B 出口处的定值语句

方程组包括数据流方程（规定了同一块内关系）和控制流方程（规定了不同块间关系），通常涉及到四项，求解以数据流方程中定义出的两项作为算法输入，以 IN 和 OUT 作为算法输出进行迭代计算，得到稳态解。对到达-定值方程而言，求解如下：

1. 计算出各个块 GEN 和 KILL，初始化 IN 为空，OUT 为 GEN
2. 逐一对各个块用控制流方程计算 IN（OUT 不会是全空，所以第一轮必然能算出新东西），如果 IN 可以更新，则用数据流方程尝试更新 OUT
3. 重复过程 2 直到没有每个块的 IN 不再更新

注意到这个过程需要反复计算控制流方程中的并集，手算求解时列表时可以标记一下 IN 的来源是哪些块的 OUT，一旦 OUT 有变马上级联更新所有 IN，加快速度，然后再逐个更新（IN 发生改变的） OUT，重复上述过程。

注：根据上述定义，被覆盖掉的语句（如同一块的 x = 2; x = 3; 中的 x = 2;）不会出现在方程的任何一项中，这与龙书上有所出入。

7.3 活跃变量方程

活跃变量方程研究的是每一个变量的活跃范围，研究对象相比到达-定值从语句变为变量（抽象来讲精度算是变低了）。活跃变量可以理解为“还有用的变量”，但是显然在它被用到之前我们无法知道它还有用，因此它的控制方向是从后到前，这是一个向后溜，而之前的到达-定值是一个向前流。其方程如下：

$$LiveIn[B] = LiveUse[B] \cup (LiveOut[B] - Def[B])$$
$$LiveOut[B] = \cup_{s \in S[B]} LiveIn[s]$$

其中 S[B] 指 B 的所有后继。

- LiveUse[B]：指 B 中在定值前要被引用的变量（即在 B 之前必须被定值过，至于在 B 中有没有定值无所谓）
- Def[B]：B 中定值且定值前未在 B 中被使用（注意后面的约束保证了 LiveUse 和 Def 无交集，且并集是 B 中所有变量）
- LiveIn[B]：B 入口处的活跃变量
- LiveOut[B]：B 出口处的活跃变量

它的迭代逻辑和我在前文提到的一般化方法相同，具体来说：

1. 计算出各个块的 LiveUse 和 Def，初始化 LiveIn 为 LiveUse，LiveOut 为空
2. 逐一对各个块用控制流方程求解 LiveOut，如果 LiveOut 可以更新，则用数据流方程尝试更新 LiveIn
3. 重复过程 2 直到没有每个块的 LiveOut 不再更新

这里的计算 trick 和到达-定值方程同理。注意初始化和之前有所不同，这个不用刻意去记，其实很自然的有数据流方程的计算结果可以通过部分已知变量初始化（未知的就考虑最没有信息量的情况），控制流方程的计算结果无需初始化，因为它将在每次迭代中被重新计算，初始化它是为了迎合前面说的“没有信息量”——例如求并集时初始化为空集，求交集时初始化为全集。

另外这里 Def 不见得是 LiveOut 的子集（从定义也可以看出来），和到达-定值不同。

7.4 UD 链和 DU 链

UD 链（Use-Definition Chain）记录了一个引用可能在何处被定值（可能不止一处，例如 $x := 1$; if $y > 0$ then $x := 2$; print(x)），计算需要借助到达-定值的 IN。考虑块 B 中变量 A 的引用点 u 的定值点：

- 如果块 B 中在 u 之前对 A 定值了，显然这就是 A 的定值点（注意，块的划分方法保证了块内不会有分支，因此如果此前有定值点则一定会经过）
- 否则，IN[B] 中所有对 A 定值的点构成了 A 在 u 处的 UB 链

DU 链（Definition-Use Chain） 记录了一个定值点可能在何处被使用，讲义上没有介绍具体计算方法，但是可以凭借观察法直接得出，也可以借用活跃变量方程沿着 B 的后继递归寻找 LiveUse 有该变量且块中没有新的定值点的块中的引用。

后面还有待用信息和活跃信息，但是讲得有些朦胧，也没考过，我跳过了。

7.5 DAG

考虑三类基本的 TAC 语句：赋值、单目运算、双目运算，记下它们各自的 DAG 画法（叶结点表示输入变量，写在结点下，非叶结点表示中间结果，对应的变量写在结点右），画的话的时候秉持着“能省则省”的原则即可，尤其是常量运算记得优化，包括间接常量（如 b=2;c=b+3; 可以直接得到 c = 5;）。

如果画出来不是连通图，也要保留每一个图，比如课件上最后没有用到的常量 T0 = 3.14 仍然保留了，因为可能后面的代码会用到。

根据 DAG 可以写出简单优化后的代码，但是讲义上没有给出具体算法，所幸图一般不复杂，如果有需要可以自行给出一个合理的顺序即可。往年题考过画出 DAG 并写求值表达式，如果是树则中序遍历，如果不是树则需要分解成一棵树（大概不会这么考）。

7.6 寄存器数量优化

DAG 产生语句序列启发式排序（对非叶节点排序，标号过程即排序从前往后的过程）：

1. 如果所有非叶结点都已经标号，则结束算法，否则选取一个父节点已经排序的非叶节点 n（第一次一定选的根节点）并为它标号
2. 如果 n 的最左孩子 m 不是叶节点且它所有父亲都已经标号，则跳3，否则跳1
3. 对 m 标号，取 n = m，跳 2

然后按照排序结果的逆序写 TAC 语句，可以减少寄存器使用。

DAG 中一个结点可能有多个父亲，所以情况略复杂，如果是表达式树求最小寄存器数（Ershov）则简单多了，对于二叉表达式树而言可以用如下算法：

- 叶节点标记为 1
- 自下而上逐个标记非叶节点，如果两个孩子标号一样则取孩子 + 1，如果两个孩子不同则取较大的

先算大的，存起来，再用剩下的reg算小的

计算时需要同等数量的reg，则先算的那个要用额外的1个保存值，后算的才能用同样多的reg计算

最终根节点标号即为 Ershov 数。

生成对应 TAC 代码的算法：

- 自上而下遍历每个结点，假设标号为 k，则根节点能用寄存器为 R_0 到 R_{k-1}
- 确定当前结点能用的寄存器R_b 到 R_{b+k-1}，其中 k 为当前结点的标号（即Ershov数，也即能用的寄存器）
- 如果两个孩子标号一样，则一定都是 k - 1，左孩子用前 k - 1个（结果会被存到能用的第 1 个，即 R_b），右孩子用后k - 1个（结果会被存到 R_{b+1}），然后计算 R_{b+1} 和 R_b 得到结果存到 R_b 中
- 如果两个孩子标号不同，则一定有一个大的标号为 k，小的标号为 m < k，则先让大的用所有寄存器完成计算并存结果到第一个即 R_b，小的使用后 m 个寄存器完成计算并存结果到第 k - m + 1 个即 R_{b+k-m}，再计算 R_b 和 R_{b+k-m} 并存结果到 R_b

当然不用去记后面那个 b+k-m 的下标，实际计算的时候对每个结点而言弄清楚要用到的寄存器是哪些，然后子树运算结果存到第一个寄存器就好。

寄存器相干图着色问题：画出程序每两行代码之间的活跃变量集合（它们是同时活跃的变量），为每个变量准备一个结点，在同时活跃的变量之间连线，得到相干图。对所得图进行着色，最小色数即最少寄存器数。

启发式着色算法：为了判断相干图是否能 k-着色，可以将图中度数小于 k 的点删除，如果新的图能则原来的也能；重复删点过程，如果最后得到了空图（没有边的图）则成功，否则失败。

注：实际上并不见得需要到空图，只要某个时刻图最大度小于 k 即可结束。