

编译原理总结

Last Edit Time: 2023-12-20

一、简答与计算

1.1 必考

- 1. 编译过程
- 2. 消除左递归
- 3. 消除回溯
- 4. 后缀表达式

1.2 选考

- 1. 编译、翻译和解释
- 2. 高级语言的分类
- 3. 中间代码
- 4. 目标代码
- 5. 构造正规式
- 6. 有限自动机 (FA)
- 7. 确定有限自动机 (DFA)
- 8. 非确定有限自动机 (NFA)
- 9. 构造 NFA
- 10. 上下文无关文法
- 11. 二义文法
- 12. 属性文法
- 13. 构造文法
- 14. 句型、句子、短语
- 15. 规范规约
- 16. 符号表
- 17. 运行时空间组织
- 18. 优化手段
- 19. 待用/活跃信息
- 20. LL(1)分析
- 21. LR(1)分析
- 22. display 表
- 23. 语法制导翻译法

二、综合题

- 2.1 词法分析
- 2.2 自顶向下分析：LL(1) 分析
- 2.3 自底向上分析：LR(0) 分析
- 2.4 自底向上分析：SLR 分析
- 2.5 自底向上分析：LR(1) 分析

2.4 中间代码生成

三、历年考试回忆

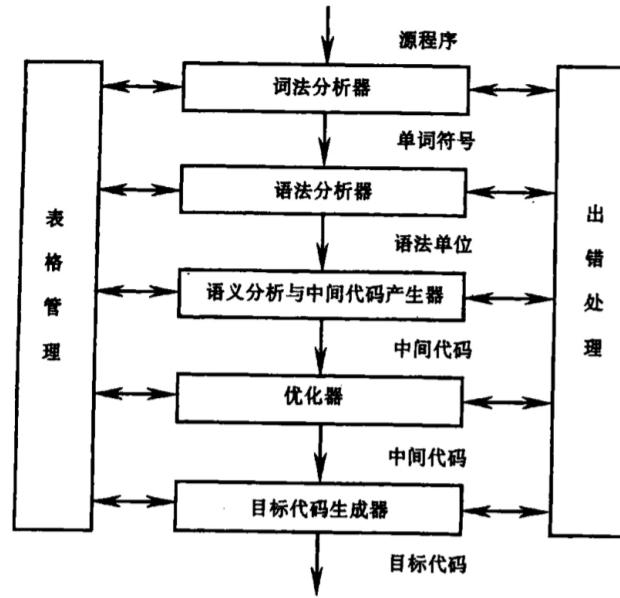
- 2004-2005
- 2015-2016
- 2016-2017
- 2017-2018
- 2019-2020
- 2020-2021
- 2021 Fall
- 2022 Spring
- 2022 Fall
- 2023 Spring
- 2023 Fall

一、简答与计算

1.1 必考

1. 编译过程

- 画图表示编译过程的各阶段，并简要说明各阶段的功能：



- 词法分析器：输入源程序，进行词法分析，输出单词符号；
- 语法分析器：根据文法构建分析表，对单词符号进行语法分析，检查程序是否符合语法规则；
- 语义分析与中间代码生成器：按照文法翻译规则对语法分析器归约出的语法单位进行语义分析，并把它们翻译成一定形式的中间代码；
- 优化器：对中间代码进行优化处理；
- 目标代码生成器：把中间代码翻译成目标代码。

2. 消除左递归

- 对于左递归文法 $P \rightarrow P\alpha | \beta$ (其中 β 的第一个符号不是 P)，可以直接利用下列规则将其转为右递归文法：

$$\begin{aligned} P &\rightarrow \beta P' \\ P' &\rightarrow \alpha P' | \epsilon \end{aligned}$$

- 同样地，如果 P 对应多个产生式，例如： $P \rightarrow P\alpha_1 | P\alpha_2 | \dots | P\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$ (其中 β_i 的第一个符号不是 P)，可以首先利用结合律将其转为一个产生式：

$$P \rightarrow P(\alpha_1 | \dots | \alpha_n) | (\beta_1 | \dots | \beta_m)$$

然后利用上面的规则将其转为右递归文法：

$$\begin{aligned} P &\rightarrow (\beta_1 | \dots | \beta_m)P' \\ P' &\rightarrow (\alpha_1 | \dots | \alpha_n)P' | \epsilon \end{aligned}$$

- 有些文法会有隐式的左递归，例如：

$$\begin{aligned} S &\rightarrow Qc | c \\ Q &\rightarrow Rb | b \\ R &\rightarrow Sa | a \end{aligned}$$

其中一种隐式的左递归为：

$$S \rightarrow Qc \rightarrow Rbc \rightarrow Sabc$$

消除隐式左递归的步骤为：

- 将非终结符进行排序（不同的顺序会有不同的结果）：S, Q, R
- 根据顺序重构产生式，确保每个非终结符所推出的产生式体中，不能包含其排序之前的非终结符（例如 R 推出的产生式不能包含 S 和 Q，若有则按顺序逐步推导，直到不包含 S 和 Q）：

$$\begin{aligned}
 S &\rightarrow Qc \mid c \\
 Q &\rightarrow Rb \mid b \\
 R &\rightarrow Sa \mid a \rightarrow Qca \mid ca \mid a \rightarrow Rbca \mid bca \mid ca \mid a
 \end{aligned}$$

- 对重构后的产生式应用上面的规则，最终的产生式为：

$$\begin{aligned}
 S &\rightarrow Qc \mid c \\
 Q &\rightarrow Rb \mid b \\
 R &\rightarrow bcaR' \mid caR' \mid aR' \\
 R' &\rightarrow bcaR' \mid \epsilon
 \end{aligned}$$

3. 消除回溯

在构造文法时，消除回溯可以提高编译器在解析代码时的效率和准确性，避免在遇到解析歧义时重复尝试多个解析路径，从而减少资源消耗和提高错误定位的准确性。

如下是一个简单的有回溯文法：

$$\begin{aligned}
 A &\rightarrow ab \\
 A &\rightarrow ac
 \end{aligned}$$

在不消除回溯的解析器中，如果输入是 "ac"，解析器首先尝试使用产生式 1。它匹配了 'a'，但在尝试匹配 'b' 时失败，因为下一个字符是 'c'。于是，解析器回溯到选择点，放弃已经做出的选择，然后尝试产生式 2。这次它成功匹配了 'a' 和 'c'。

这个过程中，解析器不得不回到选择点并重新尝试，这在复杂的语法结构中会导致效率低下。如果我们通过重构产生式来消除回溯，比如改为：

$$\begin{aligned}
 A &\rightarrow aB \\
 B &\rightarrow b \\
 B &\rightarrow c
 \end{aligned}$$

这样，解析器首先匹配 'a'，然后根据接下来的字符是 'b' 还是 'c' 来决定是使用产生式 2 还是 3。这种方法避免了回溯，因为每个步骤的选择都是基于当前和后续的输入明确的，从而提高了解析效率。

一个无回溯文法 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ 的每对候选式应该具有以下条件：

$$First(\alpha_i) \cap First(\alpha_j) = \emptyset \quad (i \neq j)$$

解决回溯的办法是反复提左公因子，例如对于如下的有回溯文法：

$$P \rightarrow \alpha A_1 \mid \alpha A_2 \mid \dots \mid \alpha A_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

通过提左公因子 α 可以得到下列消除回溯的文法：

$$\begin{aligned}
 P &\rightarrow \alpha A \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \\
 A &\rightarrow A_1 \mid A_2 \mid \dots \mid A_n
 \end{aligned}$$

4. 后缀表达式

自然语言描述数学表达式使用的是中缀表达式，计算机执行数学运算需要使用后缀表达式，在编译器处理数学表达式时自然要考虑表达式的转换。此处有可能出现一个小题，给你一个中缀形式的数学表达式，要求你转换为后缀表达式。首先介绍一下这三种表达式：

- 中缀表达式：这是最常见的表达式形式，其操作符位于操作数之间，例如表达式 $A + B$ 。中缀表达式的主要特点是直观易懂，但在计算时可能需要括号来明确操作的优先级。
- 后缀表达式（逆波兰表示法）：在这种表达式中，操作符位于操作数之后。例如，中缀表达式 $A + B$ 在后缀表达式中可以表示为 $AB+$ 。后缀表达式的优点是不需要括号来指定运算优先顺序，计算机执行起来更为直接和高效。
- 前缀表达式（波兰表示法）：在这种表达式中，操作符位于操作数之前。例如，中缀表达式 $A + B$ 在前缀表达式中表示为 $+AB$ 。前缀表达式也不需要括号来指定运算的优先级，便于计算机解析和计算。

举个例子，对于一个简单的表达式 $3 \times (3 + 5 \div (2 \times 2))$ ，其三种不同的表达方式如下：

- 中缀表达式：原始表达式就是中缀表达式，即 $3 \times (3 + 5 \div (2 \times 2))$ 。

- 后缀表达式：在后缀表达式中，操作符位于操作数之后。转换步骤如下：

- 2×2 转换为 $2\ 2\ \times$
- $5 \div (2 \times 2)$ 转换为 $5\ 2\ 2\ \times\ \div$
- $3 + 5 \div (2 \times 2)$ 转换为 $3\ 5\ 2\ 2\ \times\ \div\ +$
- 最后整个表达式转换为 $3\ 3\ 5\ 2\ 2\ \times\ \div\ +\ \times$

- 前缀表达式：在前缀表达式中，操作符位于操作数之前。转换步骤如下：

- 2×2 转换为 $\times\ 2\ 2$
- $5 \div (2 \times 2)$ 转换为 $\div\ 5\ \times\ 2\ 2$
- $3 + 5 \div (2 \times 2)$ 转换为 $+ 3 \div 5 \times 2\ 2$
- 最后整个表达式转换为 $\times\ 3 + 3 \div 5 \times 2\ 2$

中缀转后缀算法：

- 在算法开始时，你需要初始化一个符号栈，然后顺序读取中缀表达式，根据读取到的不同符号来执行相应的策略：
 - (将其入栈；
 -) 依次弹出栈顶的元素并附加到结果字符串中，直到遇到下一个 (出现在栈顶，将该 (出栈但不附加到结果字符串中；
 - + - 弹出栈顶的 + - */ 并附加到结果字符串中，然后自己进栈；
 - */ 先让栈顶的 */ 出栈，附加到结果字符串中，然后自己进栈；
 - 操作数 直接附加到结果字符串中；

最后，当输入串被扫面完毕，但符号中还有操作符时，将符号栈栈顶元素依次附加到结果字符串中。下面是一个 C++ 实现的函数：

```
// 中缀转后缀
vector<string> interToPost(vector<string> vec)
{
    vector<string> res;
    stack<string> op_stack;
    for (int i = 0; i < vec.size(); i++) {
        if (vec[i] == "(")
            op_stack.push(vec[i]);
        else if (vec[i] == ")") {
            while (op_stack.top() != "(") {
                res.push_back(op_stack.top());
                op_stack.pop();
            }
            // 左括号出栈
            op_stack.pop();
        }
        // 遇到+-，先让栈中的+-*/出栈
        else if (vec[i] == "+" || vec[i] == "-") {
            while (!op_stack.empty() && op_stack.top() != "(") {
                res.push_back(op_stack.top());
                op_stack.pop();
            }
            op_stack.push(vec[i]);
        }
        // 遇到*/，先让栈中的*/出栈
        else if (vec[i] == "*" || vec[i] == "/") {
            while (!op_stack.empty() && (op_stack.top() == "*" || op_stack.top() == "/")) {
                res.push_back(op_stack.top());
                op_stack.pop();
            }
            op_stack.push(vec[i]);
        }
    }
}
```

```

    }
    else
        res.push_back(vec[i]);
}
while (!op_stack.empty()) {
    res.push_back(op_stack.top());
    op_stack.pop();
}
return res;
}

```

计算后缀表达式：

- 计算后缀表达式的过程相对直观，主要是通过一个数据栈来实现。下面是计算后缀表达式的步骤：
 - **创建一个空栈**：这个栈将用于暂存操作数。
 - **从左到右扫描后缀表达式**：逐个检查后缀表达式中的元素。
 - **处理遇到的元素**：
 - 当遇到一个**操作数**（数字）时，将其压入栈中。
 - 当遇到一个**操作符**（比如 $+$ $-$ $*$ $/$ ）时，从栈中弹出所需数量的操作数（对于大多数二元操作符，需要弹出两个操作数）。接着，使用这些操作数执行相应的运算（比如加法、减法、乘法、除法），需要注意要将栈上的第二个元素作为左操作符，栈顶元素作为右操作符。
 - **将运算结果压回栈中**：计算完操作符指定的运算后，将结果压入栈中。
 - **重复以上步骤**：继续处理表达式中的下一个元素，直到整个表达式被完全处理完毕。
 - **得到最终结果**：当表达式结束时，栈顶的元素就是整个后缀表达式的计算结果。

下面是一个 C++ 实现的函数：

```

// 计算后缀表达式
double solvePostPrefix(vector<string> post_prefix) {
    stack<double> dataStack;
    double num1, num2, result;
    for (auto x : post_prefix) {
        if (x == "+" || x == "-" || x == "*" || x == "/") {
            num1 = dataStack.top();
            dataStack.pop();
            num2 = dataStack.top();
            dataStack.pop();
            if (x == "+")
                result = num2 + num1;
            if (x == "-")
                result = num2 - num1;
            if (x == "*")
                result = num2 * num1;
            if (x == "/")
                result = num2 / num1;
            dataStack.push(result);
        }
        else {
            double num = strtoDouble(x);
            dataStack.push(num);
        }
    }
    return dataStack.top();
}

```

1.2 选考

1. 编译、翻译和解释

- **翻译**：把一种语言程序（源语言）转换成另一种语言程序（目标语言），例如将Python代码翻译为Java代码。这种转换使得原本只能在特定语言环境中运行的程序能够在其他环境或平台上运行。
- **编译**：编译程序是翻译程序的一种特殊形式，它专门将高级编程语言（如C++或Java）编写的源代码转换为低级语言（如汇编语言或机器代码），从而可以在计算机硬件上直接运行。例如，GCC（GNU Compiler Collection）可以将C语言代码编译成适用于不同操作系统的机器代码。
- **解释**：解释程序接受用源语言编写的代码作为输入，但不生成独立的目标程序，而是在程序运行时实时翻译和执行源代码。这种方式允许更快的开发迭代，但可能会牺牲运行速度。例如，Python解释器可以直接执行Python代码，而无需先将其编译成机器代码。

2. 高级语言的分类

- **强制式语言**：这种语言注重底层细节和具体的操作指令。在强制式语言中，程序由一系列命令组成，每条命令具体指示计算机改变某些存储单元中的值。这种语言通常更关注如何执行操作，而不仅仅是执行什么操作。
例如：C语言是一种典型的强制式语言。在C语言中，程序员编写一系列具体的命令来告诉计算机如何操作。比如，使用循环和分支语句来控制程序流程，直接对内存进行读写等。C语言允许程序员以非常细粒度的方式控制程序的每一个方面。
- **应用式语言**：相比于强制式语言的操作细节，应用式语言更关注于程序的功能和目标。在这类语言中，每条语句都表达了一个较高层次的操作或结果，而不是具体的执行步骤。应用式语言的语句通常封装了更复杂的功能和逻辑。
例如：SQL专注于数据的查询和操作，而不是具体的操作步骤。当使用SQL时，程序员描述他们想要查询或修改什么数据，而不需要指定如何进行这些操作。例如，一个SQL查询可以非常简洁地表达对数据库的复杂查询请求。
- **基于规则的语言**：这类语言基于一套特定的规则来执行程序。程序运行时会检查一定的条件，当这些条件满足特定值时，就会触发相应的动作或规则。基于规则的语言常用于专家系统和逻辑编程。
例如：Prolog是一种基于规则的逻辑编程语言，它使用事实和规则来表达逻辑。在Prolog中，程序是一系列的规则，形式上类似于“当满足这些条件时，则执行这些动作”。Prolog广泛用于人工智能和计算机语言理解。
- **面向对象的语言**：面向对象的语言以对象为核心，其主要特点包括封装性、继承性和多态性。封装性允许隐藏内部状态和复杂性；继承性支持新对象基于现有对象的属性和行为构建；多态性允许以统一的方式处理不同类型的对象。
例如：Java是一种广泛使用的面向对象编程语言，它通过类和对象的概念来封装数据和操作。Java中的程序设计包括创建对象、通过继承机制共享行为以及利用接口实现多态性。这种语言风格便于构建模块化、可扩展和易于维护的代码。

3. 中间代码

中间代码是一种清晰且易于操作的符号系统，通常与具体的硬件无关，但在一定程度上接近于指令格式，或可以较为轻松地转换成机器指令。它在编程语言的编译过程中扮演着桥梁的角色，平衡了源代码的高级特性和目标代码的低级细节。中间代码的常见形式包括：

- **三元式**：一种紧凑的表示方法，用于描述运算符及其操作数，例如表达式 $a = b + c$ 的三元式表示为 $(+, b, c)$ 。
- **间接三元式**：类似于三元式，但增加了间接性，以支持更复杂的编译器优化技术。假设上述表达式在三元式序列中的位置是 4，那么其间接三元式表示为 (4) 。
- **四元式**：提供了更详细的操作描述，每个元素代表一个操作码或操作数。同样，上述表达式的四元式表示为 $(+, b, c, a)$ ，四元式在三元式的基础上增加了一个字段来明确指出结果的存储位置（此处为 a ）。
- **逆波兰式**：一种无需括号即可表示运算顺序的方法，利于快速解析和执行。例如上述表达式的逆波兰式为 $b\ c\ +$ 。在逆波兰表示法中，操作符跟在操作数之后，这种格式消除了对括号的需求，使得表达式的计算更加直接。
- **树形表示**：以树的形式展示程序结构，便于进行结构化的分析和优化。上述表达式的树形表示为构建一个节点为 $+$ 的树，其左右子节点分别是 b 和 c ，这种表示法以树的形式体现了表达式的结构，使得对程序的结构化分析和优化变得更加直观。

4. 目标代码

将中间代码转换为特定机器上的低级语言代码，并生成能充分利用硬件性能的目标代码，是一个颇具挑战的任务。这一过程涉及到多种目标代码的形式，包括：

- **汇编指令代码**：这类代码需要通过汇编程序转换才能执行。它是一种更接近机器语言的代码形式，提供了对硬件的细致控制，但同时要求更多的手动管理。

- **绝对指令代码**：这种代码可以直接在机器上执行，不需要任何额外的转换步骤。绝对指令代码直接映射到机器的内存地址，因此效率很高，但它缺乏灵活性，因为代码一旦编写，其运行位置就固定了。
- **可重定位指令代码**：这类代码在运行前需要借助链接装配程序。链接装配程序的任务是将各个目标模块（包括系统提供的库模块）连接在一起，并确定程序在内存中的起始地址。这一过程使得各个模块能够形成一个完整的、可运行的绝对指令代码程序，提供了更高的灵活性和模块化能力。

5. 构造正规式

正则表达式 (Regular Expression, RE)，也称正规式，是一种强大的文本处理工具，用于在字符串中进行搜索、匹配和替换操作。它通过定义一个特定的模式 (pattern)，来描述一系列符合某个句法规则的字符串。正则表达式通常用于文本搜索、数据验证、数据提取等领域。

正则表达式中包含许多特殊符号，每个都有其独特的用途和含义。以下是一些常见的正则表达式符号及其用途：

- . (点)：匹配任何单个字符（除了换行符）；
- *：表示前面的字符可以出现零次或多次；
- +：表示前面的字符至少出现一次；
- ?：表示前面的字符最多出现一次（即该字符是可选的）；
- {n}：表示前面的字符恰好出现n次；
- {n,}：表示前面的字符至少出现n次；
- {n, m}：表示前面的字符至少出现n次，但不超过m次；
- [abc]：表示匹配括号内的任意一个字符（在这个例子中是 a、b 或 c）；
- [^abc]：表示匹配不在括号内的任何字符；
- (abc)：表示匹配括号内的精确序列 abc；
- |：表示逻辑或 (OR)，匹配前后的表达式之一。

这些符号可以组合使用，创建复杂的匹配模式，以满足各种文本处理需求。

- 例1：令 $\Sigma = \{0, 1\}$ ，构造正规式，使其包含偶数个0和偶数个1的字。

分析：一个包含偶数个0和偶数个1的字由若干个已满足要求的短字组成，有以下三种情形：

- 00，这种以满足要求；
- 11，这种也已满足要求；
- 10或01开头，则中间经历任意多个00或11，最后必须以10或01结尾，即： $(10|01)(00|11)^*(10|01)$ 。

由以上三种任意组合，即可满足要求：

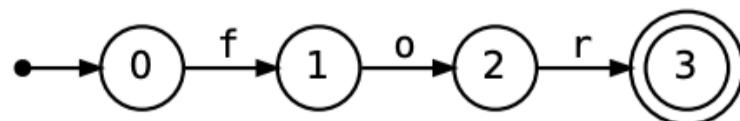
$$((10|01)(00|11) * (10|01)|00|11)*$$

6. 有限自动机 (FA)

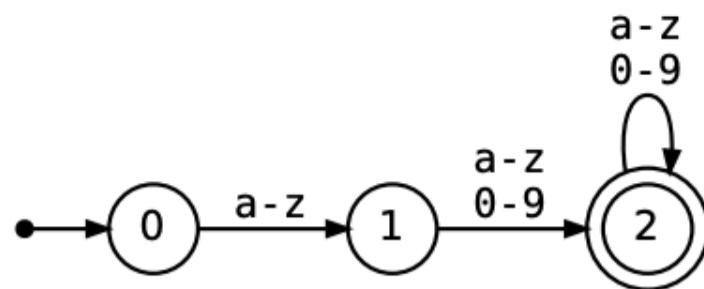
有限自动机 (Finite Automata, FA) 是一种抽象的状态图，它可以用来表示某些计算形式。从图形上来看，一个有限自动机由若干状态（用编号的圆圈表示）和这些状态之间的若干边（用标记的箭头表示）组成。每条边上标记有一个或多个来自字母表 Σ 的符号。

这台机器以一个起始状态 S_0 开始。对于每一个呈现给 FA 的输入符号，它会移动到与该输入符号标签相同的边所指示的状态。FA 的某些状态被称为接受状态，用双层圆圈表示。如果在所有输入被消耗后，FA 处于一个接受状态，那么我们说 FA 接受这个输入。如果 FA 以一个非接受状态结束，或者当前输入符号没有对应的边，我们则说 FA 拒绝这个输入字符串。

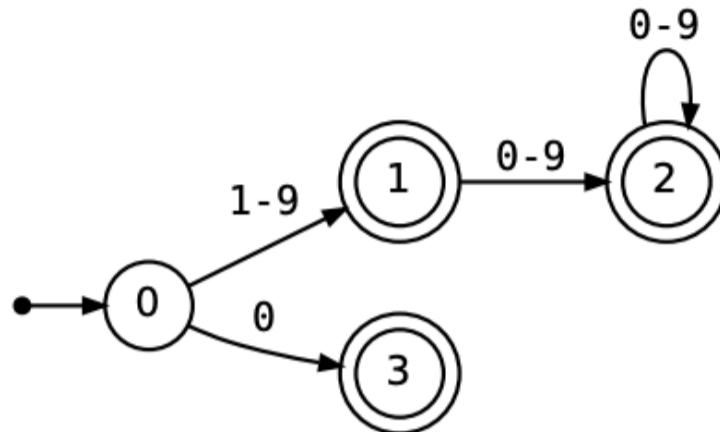
每一个正则表达式 (RE) 都可以写成一个 FA，反之亦然。对于一个简单的正则表达式，人们可以手动构造一个 FA。例如，以下是一个用于关键字 for 的 FA：



下面是用来表示正规式 $[a - z][a - z0 - 9]^+$ 的 FA：



下面是用来表示正规式 $([1-9][0-9]^*)|0$ 的 FA：



7. 确定有限自动机 (DFA)

上述三个例子中的每一个都是一个确定性有限自动机 (Deterministic Finite Automata, DFA)。DFA 是 FA 的一个特殊情况，其中每个状态对于给定的符号最多只有一个出边。换句话说，DFA 没有歧义：对于每一种状态和输入符号的组合，都有且仅有一种选择来决定下一步该怎么做。

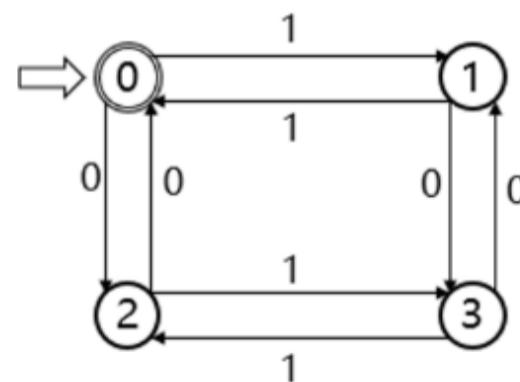
由于这个特性，DFA 在软件或硬件中非常容易实现。只需要一个整数 c 来跟踪当前状态。状态之间的转换由一个矩阵 $M[s, i]$ 表示，该矩阵编码了给定当前状态和输入符号的下一个状态。（如果不允许转换，我们用 E 来标记，表示错误）对于每一个符号，我们计算 $c = M[s, i]$ ，直到所有输入被消耗完毕，或达到错误状态。

- 例1：设计一个 DFA，使其识别包含偶数个0和偶数个1的句子（包含空句子）。

分析：DFA 可以包含下面4种状态，输入一个字符后从一个状态转换到另一个状态：

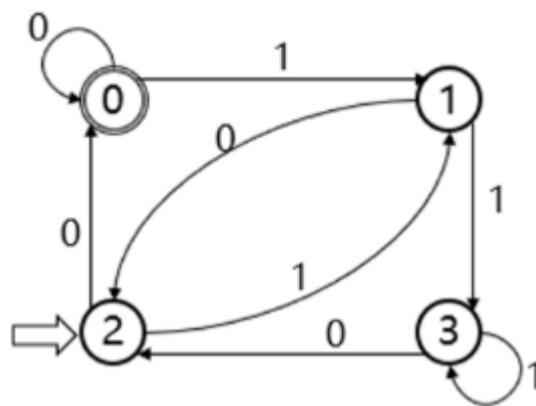
- 0：偶数个0偶数个1；
- 1：偶数个0奇数个1；
- 2：奇数个0偶数个1；
- 3：奇数个0奇数个1。

于是 DFA 可以设计为：



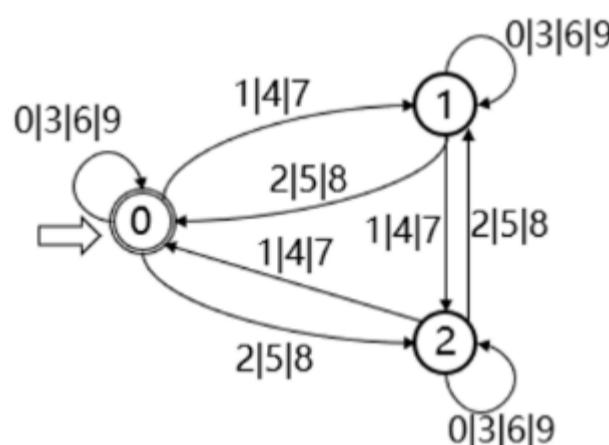
- 例2：设计一个 DFA，使其接受 $\Sigma = \{0, 1\}$ 上能被4整除的大于1的二进制数。

分析：1. 任意二进制数除以四，只有余数为0、1、2、3四种情况，因此需要四个状态；2. 当一个二进制数的后面增加一个0，该二进制数变为原来的2倍，如果后面增加一个1，则变为原来的2倍加1；3. 大于1的第一个二进制数是10，其被4整除之后是2，因此将起始状态设置为2。通过以上分析，设计出来的 DFA 如下所示：



- 例3：设计一个 DFA，使其接受 $\Sigma = \{0, 1, \dots, 9\}$ 上能被3整除的十进制数。

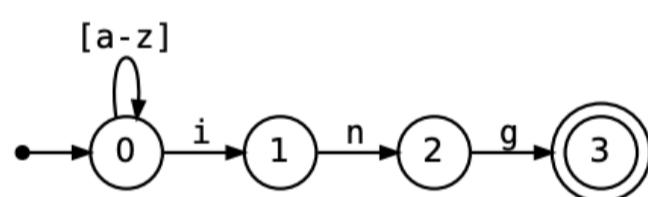
分析：1. 任意十进制数除以3，只有余数为0、1、2三种情况，因此需要三个状态；2. 一个十进制数 n 后面加 i ，变为 $10n + i$ ；3. 初态只能为0，但可以接受空字。通过以上分析，设计出来的 DFA 如下所示：



8. 非确定有限自动机 (NFA)

DFA 的替代选择是非确定有限自动机（Nondeterministic Finite Automata, NFA）。NFA 是一种有效的有限自动机，但其内在的不确定性使得它在处理上相对更为复杂。

以正则表达式 `[a-z]*ing` 为例，该表达式代表所有以 `ing` 结尾的小写单词。这可以用以下自动机表示：



现在考虑这个自动机如何处理单词 `sing`。它可以有两种不同的处理方式：

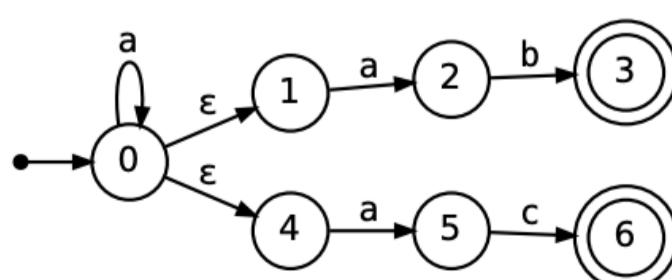
- 在 `s` 上转移到状态0，`i` 上转移到状态1，`n` 上转移到状态2，`g` 上转移到状态3；
- 整个过程中都停留在状态0，将每个字母与 `[a-z]` 转换匹配。

这两种方式都遵守转换规则，但一种导致接受，另一种导致拒绝。这里的问题在于状态0在符号 `i` 上允许两种不同的转换。一种是留在状态0，匹配 `[a-z]`，另一种是转移到状态1，匹配 `i`。

此外，没有简单的规则来选择其中一条路径。如果输入是 `sing`，正确的解决方案是在 `i` 上立即从状态0转移到状态1。但如果输入是 `singing`，那么我们应该在第一个 `ing` 时留在状态0，然后在第二个 `ing` 时转移到状态1。

为了解决这个问题，我们可以在状态转换图中引入空字 ϵ ，使得所有可能的路径都会被同时考虑。如果任何一条路径能够成功地处理整个输入字符串并且到达接受状态，那么状态转换图就会接受该字符串。这种构造方法叫做 NFA。

例如，为正规式 `a*(ab|ac)` 我们可以构造下列的 NFA：



这样，当输入句子为 `aab` 的时候，它会同时考虑下面的匹配规则：

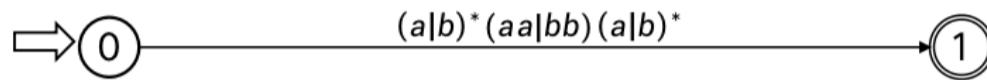
- 在 `a` 上转移到状态0，下一个 `a` 上再次转移到状态0，`b` 上没有可行的转移规则；
- 通过 ϵ 转移到状态1，`a` 上转移到状态2，下一个 `a` 上没有可行的转移规则；
- 在 `a` 上转移到状态0，通过 ϵ 转移到状态1，下一个 `a` 上转移到状态2，`b` 上转移到状态3。

NFA 会对输入的句子执行所有可行的尝试，直到有一种转移方法到达接受状态为止。

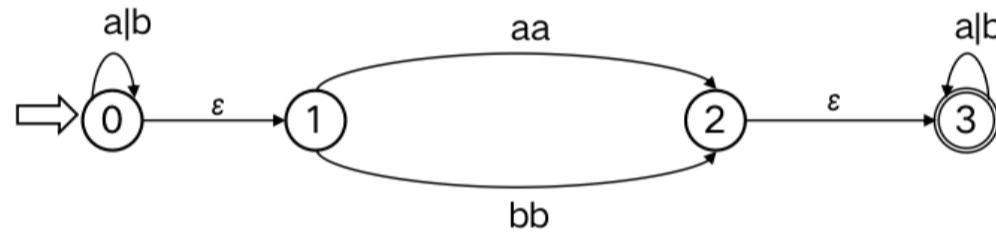
正则表达式和有限自动机在能力上是等价的。对于每一个正则表达式 (RE)，都存在一个相应的有限自动机 (FA)，反之亦然。然而，在三者中，确定性有限自动机 (DFA) 是最直接实现于软件中的，但非确定性有限自动机 (NFA) 是最便于编程者构造的。一般而言，在接受了一个 RE 后，我们会将其转换为 NFA，然后再确定化为 DFA。

9. 构造 NFA

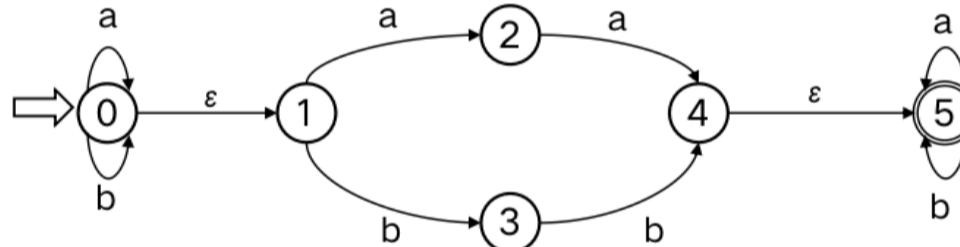
假设要将正规式 `(a|b)*(aa|bb)(a|b)*` 转换为 NFA，第一步是构造初态和终态两个状态，将它们之间用箭头连接起来，并将正规式放在箭头上面：



接下来，将正规式扩展为 $(a|b)^*$ 、 $(aa|bb)$ 、 $(a|b)^*$ 三个部分：



最后，进一步扩展深层的表达式：



10. 上下文无关文法

上下文无关文法 (Context-Free Grammar, 简称CFG) 是一种用来描述形式语言的文法类型。在计算机科学和语言学中，它被广泛用于描述编程语言的语法和自然语言的结构。一个上下文无关文法 G 可以表示为一个四元组：

$$G = (V_N, V_T, P, S)$$

- V_N 是一个非空有限集合，它的每个元素被称为非终结符号 (non-terminal)，非终结符是用来表示语法结构的符号，它们是产生式规则中的变量，代表了更大的构造块或模式；
- V_T 是一个非空有限集合，它的每个元素被称为终结符号 (terminal)，终结符是文法的基本符号，不能被进一步分解。在编程语言中，终结符通常是关键字、运算符、数字等最基本的元素，并有 $V_T \cap V_N = \emptyset$ ；
- P 是一个有限集合，包含了一系列产生式，产生式规则定义了如何从非终结符生成字符串（可以包含非终结符和终结符），每个产生式的形式是 $P \rightarrow \alpha$ ，其中 $P \in V_N$ 且 $\alpha \in (V_T \cup V_N)$ ；
- S 是一个非终结符，称为开始符号，它是整个文法的起点。

例如下列文法：

$$\begin{aligned} T &\rightarrow T + F \\ F &\rightarrow F * N \\ T &\rightarrow \text{int} \\ F &\rightarrow \text{int} \\ N &\rightarrow \text{int} \end{aligned}$$

非终结符号 $V_N = \{T, F, N\}$ ，终结符号 $V_T = \{int\}$ ，产生式为这五条形如 $P \rightarrow \alpha$ 的式子的集合，开始符号为 $S = T$ 。

11. 二义文法

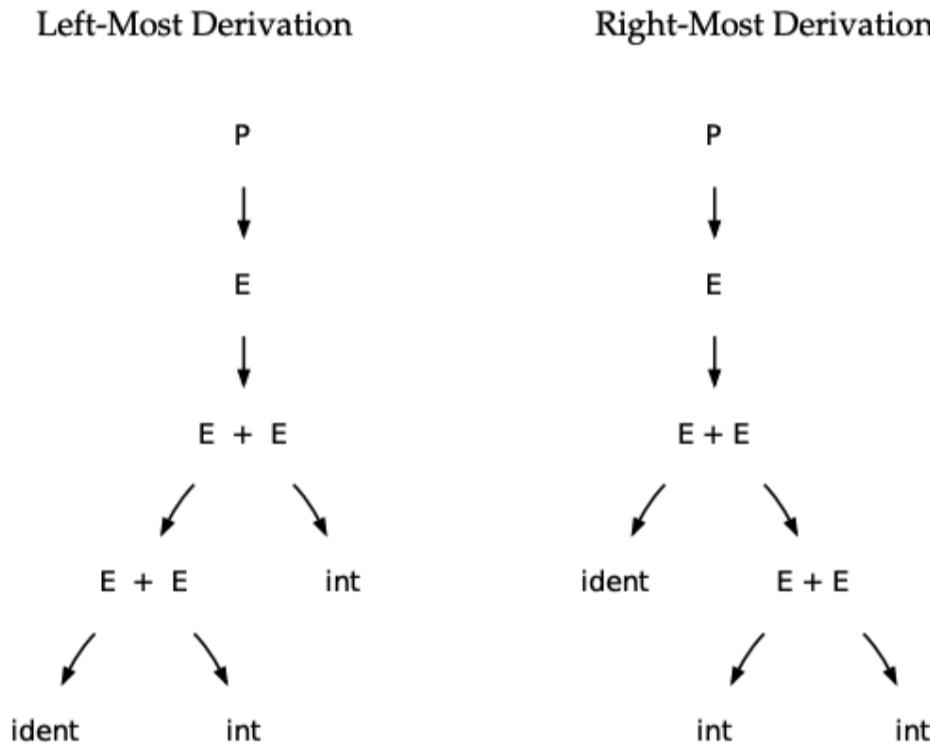
如果一个文法的某个句子对应两棵不同的语法树，即其最左（最右）推导不唯一，称该文法为二义文法。对于程序设计语言而言，通常需要其语法是无二义的。这是因为在编程中，每个语句的含义必须是清晰且明确的，以保证程序的一致性和可预测性。如果一个编程语言的文法是二义的，同一个语句可能会被编译器或解释器以不同的方式解释，导致程序行为的不确定性，这在实际应用中是不可接受的。

然而，证明一个文法是否是二义的通常是非常困难的。一个常见的方法是找到一个具体的句子，并展示它可以对应至少两棵不同的语法树。这种方法可以证明文法是二义的。但是，如果无法找到这样的句子，我们通常不能简单地断定该文法是无二义的。这是因为不存在一个通用的算法可以穷举所有可能的句子和它们的推导树，以证明一个文法的无二义性。

下面是一个二义文法的例子：

$$\begin{aligned} P &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow ident \\ E &\rightarrow int \end{aligned}$$

由于找到句子 $ident + int + int$ 可以对应下面两种推导方式，因此可以证明这个文法是二义的：



要想消除这个文法的二义性，我们可以修改上述文法，使得非终结符 E 不能既推出表达式 $E + E$ 又能推出终结符 $ident$ 和 int ，可以做下列修改：

$$\begin{aligned} P &\rightarrow E \\ E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow ident \\ T &\rightarrow int \end{aligned}$$

这样，通过保留左递归和消除右递归，上述文法的二义性就被消除了。现在假设我们想要为该文法增加更多的运算符，例如 $*$ ，如果将该规则定义为 $E \rightarrow E * E$ 的形式，二义文法还是会再次出现。然而，我们还是可以采用上述相同的方式（保留左递归消除右递归）来消除这种二义性，文法整体就变成了：

$$\begin{aligned} P &\rightarrow E \\ E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T * F \\ T &\rightarrow F \\ F &\rightarrow ident \\ F &\rightarrow int \end{aligned}$$

一般来说，优先级越低的运算符在更高层（例如 +、-），而优先级越高的运算符在更底层（例如 *、÷），这是因为计算数学表达式的时候是自底向上的，总是计算优先级高的表达式再执行优先级低的表达式。

悬空 *else* 是一个更经典的二义文法的例子：

$$\begin{aligned} P &\rightarrow S \\ S &\rightarrow \text{if } E \text{ then } S \\ S &\rightarrow \text{if } E \text{ then } S \text{ else } S \\ S &\rightarrow \text{other} \end{aligned}$$

句子 *if E then if E then other else other* 对应下面两种推导，因此文法是二义的：

```
if E then
  if E then
    other
  else
  other
```

```
if E then
  if E then
    other
  else
  other
```

句子末尾的 *else other* 可以属于两个 *if* 中的任意一个，于是产生了二义性。为了消除这个文法的二义性，我们的核心目标是确保每个 *else* 分支都能与一个明确的 *if* 语句对应。通过下列改写可以消除这种二义性：

$$\begin{aligned} P &\rightarrow S \\ S &\rightarrow \text{if } E \text{ then } S \\ S &\rightarrow \text{if } E \text{ then } L \text{ else } S \\ L &\rightarrow \text{if } E \text{ then } L \text{ else } L \\ S &\rightarrow \text{other} \\ L &\rightarrow \text{other} \end{aligned}$$

当一个 *if* 语句包含一个嵌套的 *if – else* 结构时，它将产生一个 *L* 非终结符，每个 *L* 表示一个完整的 *if – else* 语句，并在需要时递归扩展。通过这种方式，文法确保了每个 *else* 语句都紧随其最近的 *if* 语句。举个例子，加入我们在句子中遇到一个 *else*，现在有两种选择：

- 使用规则 $L \rightarrow \text{if } E \text{ then } L \text{ else } L$ 是将其规约为 *L*；
- 使用规则 $S \rightarrow \text{if } E \text{ then } L \text{ else } S$ 是将其规约为 *S*；

如果将其规约为 *L*，那么其一定是一个完整的 *if – else* 结构的前半部分；如果将其规约为 *S*，那么其一定是一个完整的 *if – else* 结构的后半部分，或 *if* 结构的后半部分。通过这种方法，每个 *else* 都会与它最近的 *if* 成功匹配，从而避免了悬空 *else* 问题。

12. 属性文法

属性文法是一种用于描述语言语法及其语义属性的强大工具，主要分为两种类型：

- **S-属性文法**：这种文法仅包含综合属性。综合属性是那些从语法树的子节点计算并传递到父节点的属性，它们通常用于构建自底向上的解析过程。在S-属性文法中，每个语法结构的语义由其组成部分的语义直接决定，没有外部依赖。

假设我们有一个简单的算术表达式文法，用于处理加法和乘法，如 $3 + 2 * 4$ 。在S-属性文法中，我们可能有如下的规则：

$$\begin{aligned} T &\rightarrow T + T \\ T &\rightarrow T * T \\ T &\rightarrow \text{int} \end{aligned}$$

在这种情况下，每个表达式的值（综合属性）可以由其子表达式的值计算得出。例如 $T \rightarrow T + T$ 的值是两个子表达式值的总和。

- **L-属性文法**：这类文法既包含综合属性，也包含继承属性。不同于综合属性，继承属性是从父节点或相邻兄弟节点传递到当前节点的属性。在L-属性文法中，一个节点的继承属性可能依赖于：

- 产生式体中该符号左侧的属性：这些属性可以是其他节点的综合属性或继承属性；
- 产生式头部的继承属性：这意味着子节点的属性可以受到父节点属性的影响。

考虑一个用于处理变量声明和赋值的语法。在这种情况下，变量的类型（继承属性）可能需要从声明传递到使用的地方。考虑一个用于标记表达式中每个数字的深度的例子。这里的“深度”是指数字在语法树中的层级，根节点的深度为0，每向下一层深度增加

1。在这个例子中，我们只使用继承属性来传递深度信息，不计算表达式的值，也不使用综合属性。假设我们的文法如下：

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T * F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow \text{number} \end{aligned}$$

在这个文法中， E 、 T 和 F 分别代表表达式、项和因子。我们定义一个继承属性 $depth$ 来表示当前节点的深度。处理过程如下：

- 在解析开始时，最顶层的 E （即整个表达式）的 $depth$ 设置为0。
- 对于每个规则，当我们向下移动到子节点时（如 $E \rightarrow E + T$ 中的第二个 E 或 T ）， $depth$ 增加1。
- 当遇到数字时（即 $F \rightarrow \text{number}$ ），我们记录该数字的 $depth$ 。

例如，对于表达式 $(3 + (4 * 5))$ ：

- 最外层的表达式 $(3 + (4 * 5))$ 的 $depth$ 是 0。
- 数字 3 直接位于这个表达式中，因此它的 $depth$ 也是 0。
- 对于内层表达式 $(4 * 5)$ ，其 $depth$ 为 1。
- 因此，数字 4 和 5 的 $depth$ 都是 2，因为它们位于括号内的子表达式中。

通过这种方式，我们可以使用继承属性来追踪每个数字在表达式中的深度，而不需要任何综合属性。总得来说，L-属性文法提供了更大的灵活性，允许属性值在语法树中更广泛地传递，但同时也增加了设计和实现的复杂性。通过使用这两种属性文法，可以更精确地定义和解析程序语言的语法结构及其相关语义。

- **终结符的属性**：在属性文法中，终结符通常只有综合属性。这是因为终结符在解析树中是叶子结点，它们没有子结点，因此不能从子结点继承属性。非终结符既可以有综合属性，也可以有继承属性。
- **依赖图**：是一个表示语法树中结点间相互依赖关系的有向图。构造依赖图的步骤为：

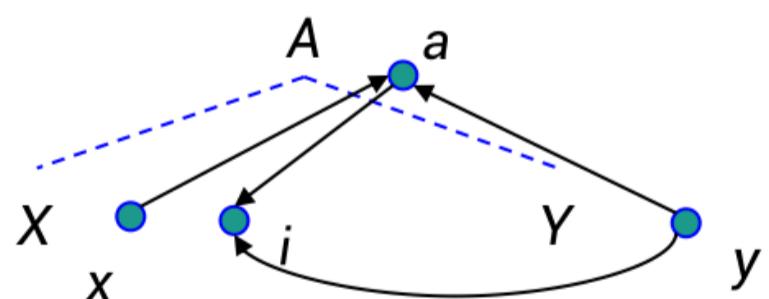
```
for node n in 语法树:
    for attribute a in node n:
        create a related node in 依赖图
for node n in 语法树:
    for rule b=f(c1,c2,...,ck) in node n:
        for i = 1,2,...,k:
            create a directed edge from ci to b
```

若有依赖关系 $b = f(c_1, c_2, \dots, c_k)$ ，则属性 b 依赖于属性 $c_i (i = 1, \dots, k)$ ，从 $c_i (i = 1, \dots, k)$ 向 b 画有向边。例如下列语法及其依赖图可以表示为：

$$A \rightarrow XY$$

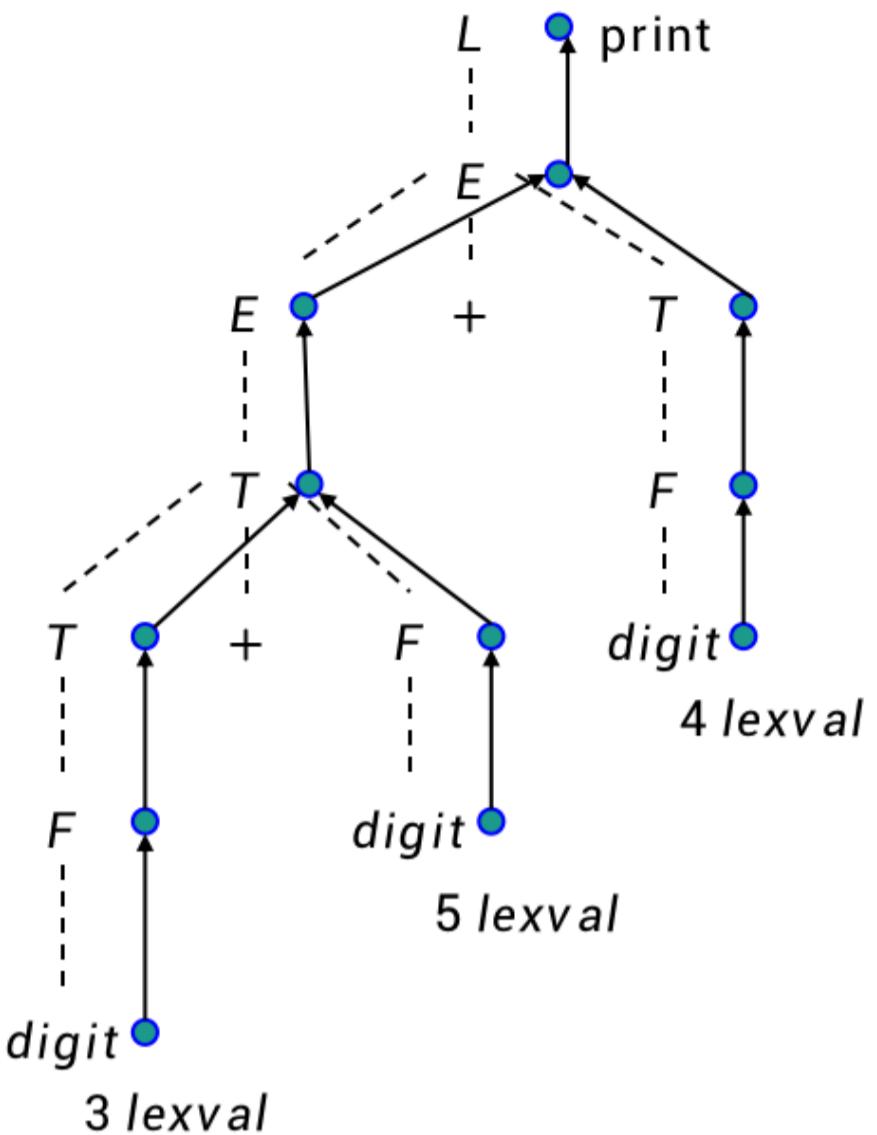
$$A. a = f(X. x, Y. y)$$

$$X. i = g(A. a, Y. y)$$



对下列文法计算 $3 * 5 + 4$ 的属性值：

$L \rightarrow E$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$



13. 构造文法

构造文法需要先找出文法的开始符号，然后定义非终结符号和终结符号的集合，再定义出符合规则的产生式。

- 例1：构造文法 G ，使其描述的语言为正奇数集合。

分析：正奇数要求要麼是一位奇数数字，要麼是以奇数数字结尾的十进制数字。

- 终结符 V_T ：数字 $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ，因为奇数的个位只能是这些数字；
- 非终结符 V_N ： $\{odd_number, digit, odd_digit\}$ ，分别表示一个奇数、一位数字、一位奇数数字；
- 开始符号 S ： odd_number ；
- 产生式 P ：

$$\begin{aligned} odd_number &\rightarrow odd_digit \\ odd_number &\rightarrow digit \ odd_digit \\ digit &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\ odd_digit &\rightarrow 1|3|5|7|9 \end{aligned}$$

这种文法允许生成任意的正奇数。它通过允许数字序列中的最后一位是奇数来确保生成的数是奇数，而序列中的其他位可以是任意数字。

- 例2：给出下面语言的相应文法：

$$\begin{aligned} L_1 &= \{a^n b^n c^i \mid n \geq 1, i \geq 1\} \\ L_2 &= \{a^i b^n c^n \mid n \geq 1, i \geq 0\} \\ L_3 &= \{a^n b^n a^m b^m \mid m \geq 0, n \geq 0\} \\ L_4 &= \{1^n 0^m 1^m 0^n \mid m \geq 0, n \geq 0\} \end{aligned}$$

解：

$$\begin{aligned} G_1[S] : S &\rightarrow AB, A \rightarrow aAb \mid ab, B \rightarrow Bc \mid c \\ G_2[S] : S &\rightarrow AB, A \rightarrow Aa \mid \epsilon, B \rightarrow bBc \mid bc \\ G_3[S] : S &\rightarrow AB, A \rightarrow aAb \mid \epsilon, B \rightarrow aBc \mid \epsilon \\ G_4[S] : S &\rightarrow 1S0 \mid A, A \rightarrow 0A1 \mid \epsilon \end{aligned}$$

- 例3：已知语言 $L(G) = ab^n c^n$ ，构造文法 G 。

$$\begin{aligned} S &\rightarrow aBC \\ B &\rightarrow bB \mid \epsilon \\ C &\rightarrow cC \mid \epsilon \end{aligned}$$

14. 句型、句子、短语

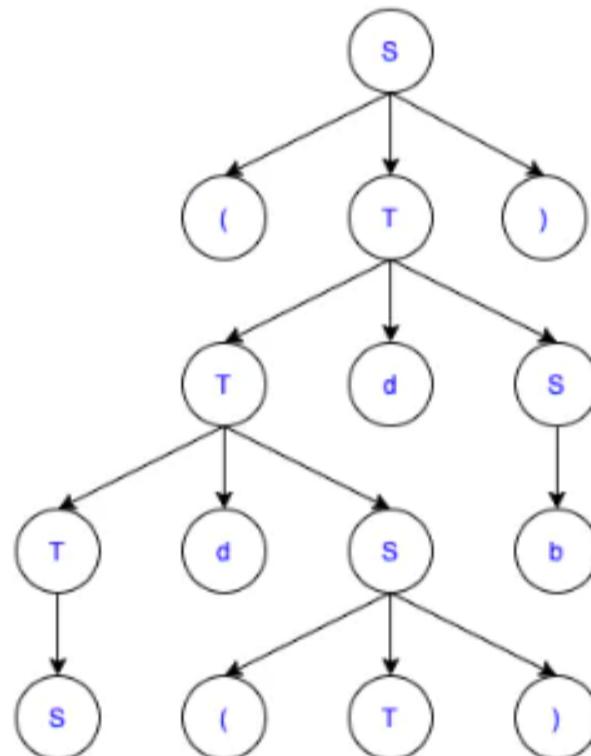
句型：句型是由文法的开始符号通过一系列的推导规则所生成的符号串。这些符号串可能包含非终结符号和终结符号。例如，在一个简单的算术表达式文法中，句型可能是 $E + T$ 或者 $num * (E)$ 这样的符号串，其中 E 和 T 是非终结符号， num 是终结符号；

句子：句子是一种特殊的句型，它完全由终结符号组成。换句话说，句子是从开始符号仅通过终结符号推导出来的符号串。在上述算术表达式的例子中，一个句子可能是 $3 + 5$ 或者 $4 * (2 + 3)$ 。这些都是不再含有任何非终结符号的表达式，代表了该文法所能描述的具体语言实例；

短语：短语是句型中相对于某个非终结符号的一部分。如果在某个句型的推导过程中，非终结符号 A 被替换为了一系列符号（可以是终结符号、非终结符号或二者的混合），那么这一系列符号就构成了一个短语。短语反映了语法树中非终结符号所代表的子树的结构。例如，在句型 $E + T$ 中，如果 E 被推导为 num ，那么 num 就是这个句型相对于非终结符号 E 的短语。

- 对一个抽象语法树来说，子树的边缘是相对于子孩子树根节点的短语。

例如如下的语法树，对于所有以 S 为根的子树，其边缘自底向上可以是： (T) 、 b 、 $(Sd(T)db)$ ；对于所有以 T 为根的子树，其边缘自底向上可以是： S 、 $Sd(T)$ 、 $Sd(T)db$ 。



直接短语：对文法 G ，如果存在一个推导 $S \Rightarrow \alpha A \delta$ 且 $A \Rightarrow \beta$ ，则称 β 是句型 $\alpha A \delta$ 相对于非终结符号 A 的直接短语。直接短语对应于语法树中从某个非终结符号直接推导出来的所有符号，它们构成了语法树中高度为2的子树，即二层子树的边缘。换句话说，直接短语是所有二层子树的边缘。在上图中有三个二层子树：

- S 是 T 的直接短语；
- (T) 是 S 的直接短语；
- b 是 S 的直接短语。

句柄：在一个句型中，最左边的直接短语被称为句柄。句柄是语法分析中特别重要的概念，因为它代表了最左边的二层子树的边缘。在某些语法分析算法中，如移进-归约分析法，句柄的识别是进行归约操作的关键。最左直接短语。句柄是最左二层子树的边缘，例如上图中的句柄是 S ；

- 例1：证明 $E + T * F$ 是下列文法的一个句型，并指出这个句型的所有短语、直接短语和句柄。

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid i \end{aligned}$$

证明：上述句型可以由 $E \rightarrow E + T \rightarrow E + T * F$ 得出，因此该句型属于此文法。同时，画出该句型的抽象语法树可以得到其直接短语（所有二层子树的边缘）是 $T * F$ ，句柄（最左二层子树的边缘）是 $T * F$ 。

活前缀：活前缀是指在语法分析过程中，那些能够被进一步扩展成为合法句型的前缀部分，简单来说就是一个句型的前缀。

- 例1：对于下列文法，写出 abc 和 abA 的活前缀：

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow aA \\ A &\rightarrow bA \\ A &\rightarrow c \end{aligned}$$

abc 和 abA 都是合法的句型，因此它们的活前缀分别是：

abc 的活前缀：

ϵ
 a
 ab
 abc

abA 的活前缀：

ϵ
 a
 ab
 abA

▼ 素短语和最左素短语的定义非常不明确，目前翻遍了中文互联网没找到能说清楚的帖子，实际写编译器的时候也没什么用，深究可能会造成混淆。有兴趣可以查看一篇英文文章：[Bottom-Up Parsing \(Compiler Writing\) Part 2 \(what-when-how.com\)](#)

15. 规范规约

规范规约，也即“最右规约”，是“最左推导”的逆过程。“推导”即自顶向下分析句子，“规约”即自底向上分析句子。为了解释什么是规范规约，首先需要知道什么是“最左推导”。假设有下列文法：

$$\begin{aligned} S &\rightarrow aAcBe \\ A &\rightarrow b \\ A &\rightarrow Ab \\ B &\rightarrow d \end{aligned}$$

对于句子 $abbcde$ ，如果对开始符合的产生式 $S \rightarrow aAcBe$ 左到右进行推导，即最左推导，其推导过程可以为：

$$S \Rightarrow aAcBe \Rightarrow aAbcBe \Rightarrow abbcBe \Rightarrow abbcde$$

如果对该产生式从右到左进行推导，则也可以为：

$$S \Rightarrow aAcBe \Rightarrow aAcde \Rightarrow aAbcde \Rightarrow abbcde$$

由于最左推导比较直观，对编译器的设计也比较友好，因此我们约定“最左推导”为“规范推导”，而“最左推导”的逆过程，即“最右规约”为“规范规约”。例如上述句子的规范规约为：

$$abbcde \leq abbcBe \leq aAbcBe \leq aAcBe \leq S$$

16. 符号表

用于登记源程序的各类信息，如变量名、常量名、过程名等，以及编译各阶段的进展状况。当扫描器识别出一个标识符后，把该名字填入符号表，在语义分析阶段回填类型，在目标代码生成阶段回填地址。

符号表的作用和地位：（重点）

- 收集符号属性；
- 语义合法性检查的依据，如重复变量定义；
- 作为目标代码生成阶段地址分配的依据。

符号表的主要属性：

- 符号名：变量、过程、类的名称；
- 符号数据类型：整型、实型、布尔型；
- 符号声明类别：`static`、`const` 等；
- 符号存储方式：堆区存储还是栈区存储等；
- 符号作用域：全局变量与局部变量；

符号表的组织方式：

- 构造多个符号表，具有相同属性种类的符号组织在一起；
- 把所有符号项都组织在一张大的符号表中；
- 折中了上述两种方案，根据符号属性的相似程度分类成若干张表；
- 使用对象组织，需要编译器的支持，但非常方便管理；

符号表项的排列：

- 数组：线性组织；
- 链表：`Hash` 表，跳表；
- 树形：二叉树，平衡二叉树。

17. 运行时空间组织

运行时存储器的划分：

- 代码区：编译生成的目标代码；
- 静态区：编译时就可以完全确定的数据；
- 栈区：栈式内存分配；
- 堆区：堆式内存分配；

存储分配策略：

- **静态分配策略**：在编译时对所有数据对象分配固定的存储单元，且在运行时始终保持不变；
- **栈式动态分配策略**：在运行时把存储器作为一个栈进行管理，运行时，每当调用一个过程，它所需要的存储空间就动态地分配于栈顶，一旦退出，它所占空间就予以释放；
- **堆式动态分配策略**：在运行时把存储器组织成堆结构，以便用户关于存储空间的申请与归还（回收），凡申请者从堆中分给一块，凡释放者退回给堆；

活动记录：

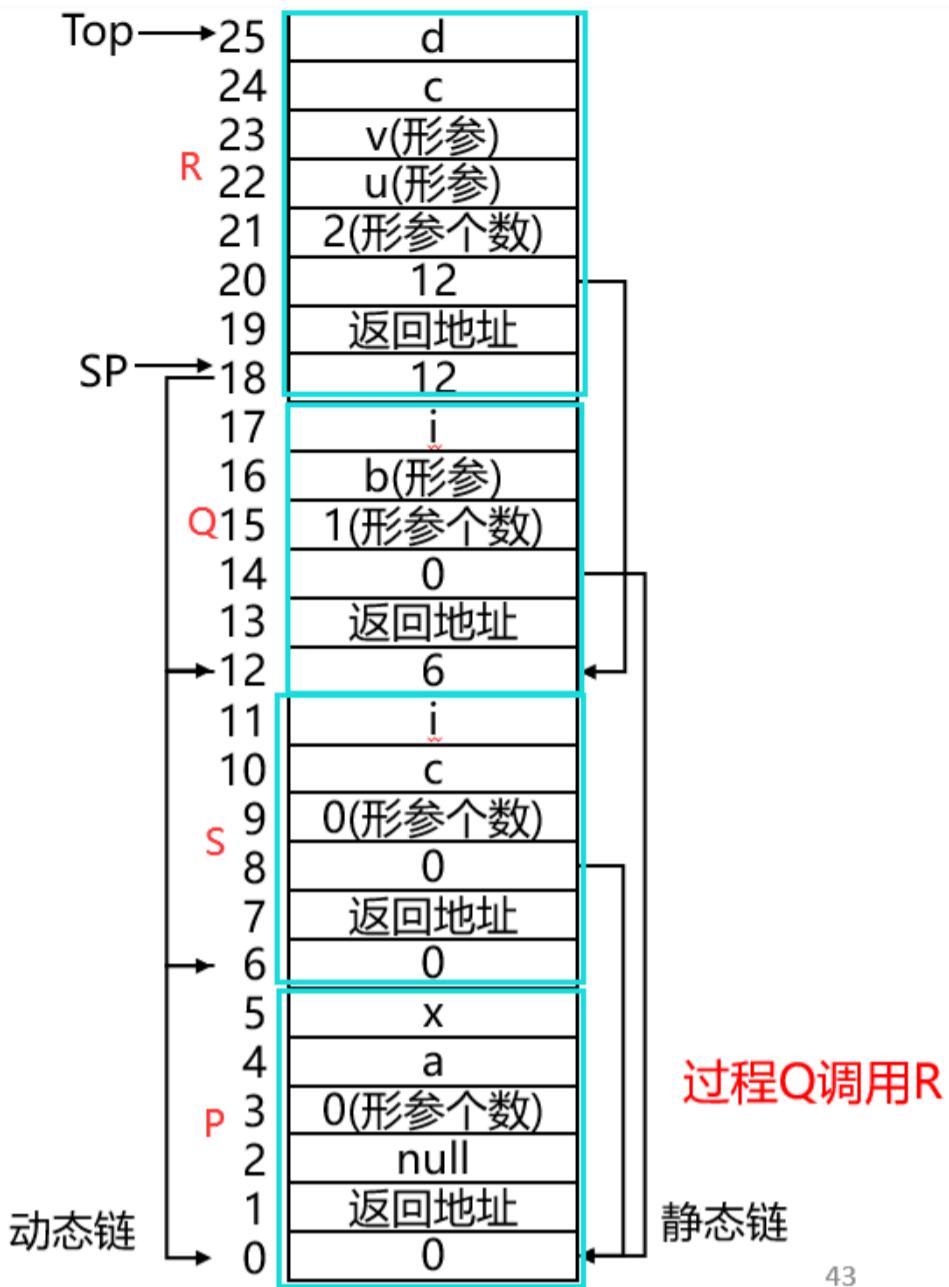
为了管理过程在一次执行中所需要的信息，使用一个连续的存储块，这样的一个连续存储块称为活动记录，一般包括：

- 局部变量：源代码中在过程体中声明的变量；
- 临时单元：为了满足表达式计算要求而不得不预留的临时变量；
- 内情向量：内情向量是静态数组的特征信息，用来描述数组属性信息的一些常量，包括数组类型、维数、各维的上下界及数组首地址；
- 返回地址：调用位置的地址；
- 动态链：`SP` 指向当前过程的动态链地址（也是帧起始地址），它又指向调用该过程的上一过程的帧起始地址，用于过程结束后回收分配的帧；它和函数的嵌套定义关系无关，只与调用顺序有关；
- 静态链：指向当前过程的直接父过程的帧起始地址，用于访问非局部数据，它只与函数嵌套定义关系有关。

```

program P;
var a, x: integer;
procedure Q (b: integer);
var i: integer;
procedure R (u, v: integer);
var c, d: integer;
begin
... if u = 1 then R(u+1, v); ...
v := (a + c) * (b - d); ...
end {R}
begin
... R(1, x); ...
end {Q}
procedure S;
var c, i: integer;
begin
a := 1; Q(c); ...
end {S}
begin
a := 0; S; ...
end {P}

```



18. 优化手段

- 源代码级别：选择适当的算法，例如快排优于插排；
- 语义动作级别：生成高效的中间代码，例如在词法分析阶段加入错误检查；
- 中间代码级别：安排专门的优化阶段，例如 DAG 优化；
 - 局部优化：例如 DAG 优化；
 - 循环优化：包含代码外提、强度削弱、删除归纳变量、复写传播等；
- 目标代码级别：考虑如何有效地利用寄存器，例如窥孔优化。

19. 待用/活跃信息

当翻译 `A = B op C` 时：

- 待用信息：变量在哪些中间代码中还会被引用；
- 活跃信息：`A`、`B`、`C` 是否还会在基本块内被引用；

【例11.2】 考察基本块，其中 W 是出口活跃变量，计算待用信息和活跃信息。

$$(1) T = A - B \quad (2) U = A - C \quad (3) V = T + U \quad (4) W = V + U$$

变量名	待用信息及活跃信息		
T	(-, -)	$\rightarrow (3, Y)$	$\rightarrow (-, -)$
A	(-, -)	$\rightarrow (2, Y)$	$\rightarrow (1, Y)$
B	(-, -)	$\rightarrow (1, Y)$	
C	(-, -)	$\rightarrow (2, Y)$	
U	(-, -)	$\rightarrow (4, Y)$	$\rightarrow (3, Y)$ $\rightarrow (-, -)$
V	(-, -)	$\rightarrow (4, Y)$	$\rightarrow (-, -)$
W	$(-, Y)$	$\rightarrow (-, -)$	

序号	中间代码	左值	左操作数	右操作数
1	$T = A - B$	$(3, Y)$	$(2, Y)$	$(-, -)$
2	$U = A - C$	$(3, Y)$	$(-, -)$	$(-, -)$
3	$V = T + U$	$(4, Y)$	$(-, -)$	$(4, Y)$
4	$W = V + U$	$(-, Y)$	$(-, -)$	$(-, -)$

18

20. LL(1)分析

对于一个不含回溯和左递归的文法，LL(1)方法从左到右扫描输入串，维护一个状态栈和一个符号栈，每一步只向右查看一个符号，根据状态栈顶、符号栈顶和分析表的内容来确定下一步的动作，最终分析出整个句子。

21. LR(1)分析

LR(1)分析适合大多数上下文无关文法，它从左到右扫描符号串，能记住移进和规约出的整个符号串，即“记住历史”，还可以根据所用的产生式推测未来可能碰到的输入符号，即“展望未来”，根据“历史”、“展望”和分析表的内容来确定下一步的动作，最终分析出整个句子。

22. display 表

为提高访问非局部变量的速度，引入指针数组指向本过程的所有外层，成为嵌套层次显示表，display 表是一个栈，自顶向下依次指向当前层、直接外层、直接外层的直接外层，直到最外层。

23. 语法制导翻译法

对单词符号串进行语法分析，构造语法分析树，然后根据需要遍历语法树并在语法树的各结点处按语义规则进行计算。这种有源程序的语法结构驱动的处理办法就是语法制导翻译法。

二、综合题

2.1 词法分析

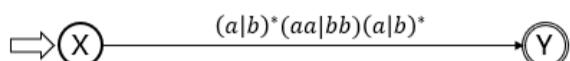
词法分析题的一般步骤为：

- 给定正规式，构造NFA；
- 将 NFA 确定化为 DFA；
- 最小化该 DFA。

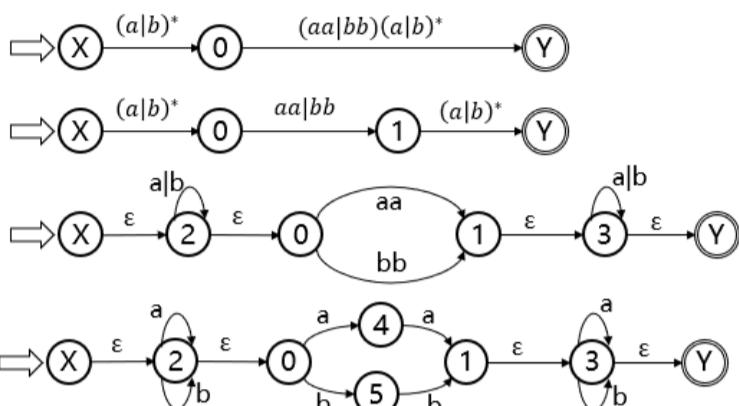
例题：

【例3-14】构造正规式 $(a|b)^*(aa|bb)(a|b)^*$ 的DFA

(1) 构造NFA，使初态、终态唯一



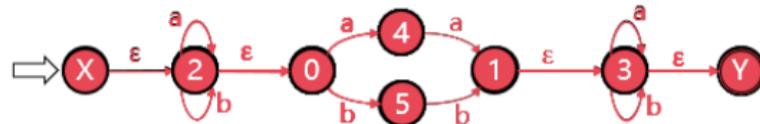
(2) 分裂，直至每条箭弧上或为 ϵ ，或为单个字符



(4) 状态合并

	I	I_a	I_b
0	0	1	2
1	1	3	2
2	2	1	4
3	3	3	5
4	4	6	4
5	5	6	4
6	6	3	5

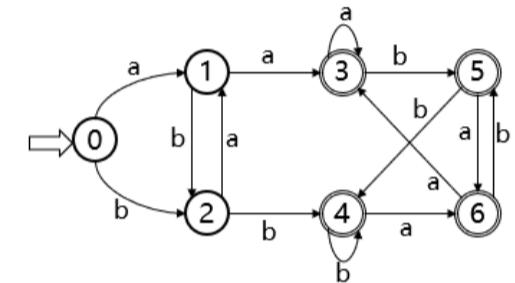
(3) 寻找可合并状态



I	I_a	I_b
{X,0,2}	{0,2,4}	{0,2,5}
{0,2,4}	{0,1,2,3,4,Y}	{0,2,5}
{0,2,5}	{0,2,4}	{0,1,2,3,5,Y}
{0,1,2,3,4,Y}	{0,1,2,3,4,Y}	{0,2,3,5,Y}
{0,1,2,3,5,Y}	{0,2,3,4,Y}	{0,1,2,3,5,Y}
{0,2,3,5,Y}	{0,2,3,4,Y}	{0,1,2,3,5,Y}
{0,2,3,4,Y}	{0,1,2,3,4,Y}	{0,2,3,5,Y}

(4) 状态合并

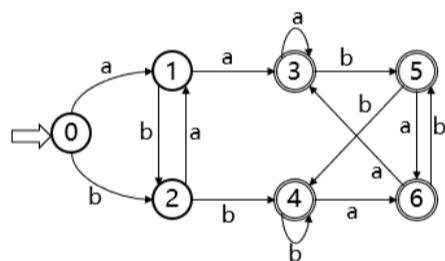
	I	I_a	I_b
{X,0,2}	0	1	2
{0,2,4}	1	3	2
{0,2,5}	2	1	4
{0,1,2,3,4,Y}	3	3	5
{0,1,2,3,5,Y}	4	6	4
{0,2,3,5,Y}	5	6	4
{0,2,3,4,Y}	6	3	5



(3)

【例3-17】化简如下DFA M

□ 初次划分： $\Pi_0 = \{\{0,1,2\}, \{3,4,5,6\}\}$



□ 考察子集 $\{0,1,2\}$

➢ $\delta(0, a) = 1 \in \{0,1,2\}$

➢ $\delta(1, a) = 3 \in \{3,4,5,6\}$

➢ $\delta(2, a) = 1 \in \{0,1,2\}$

□ $\Pi_1 = \{\{0\}, \{2\}, \{1\}, \{3,4,5,6\}\}$

□ 考察子集 $\{0,2\}$

➢ $\delta(0, b) = 2 \in \{0,2\}$, $\delta(2, b) = 4 \in \{3,4,5,6\}$

□ $\Pi_2 = \{\{0\}, \{2\}, \{1\}, \{3,4,5,6\}\}$

□ 考察子集 $\{3,4,5,6\}$

➢ $\delta(3, a) = 3 \in \{3,4,5,6\}$

➢ $\delta(4, a) = 6 \in \{3,4,5,6\}$

➢ $\delta(5, a) = 6 \in \{3,4,5,6\}$

➢ $\delta(6, a) = 3 \in \{3,4,5,6\}$

➢ $\delta(3, b) = 5 \in \{3,4,5,6\}$

➢ $\delta(4, b) = 4 \in \{3,4,5,6\}$

➢ $\delta(5, b) = 4 \in \{3,4,5,6\}$

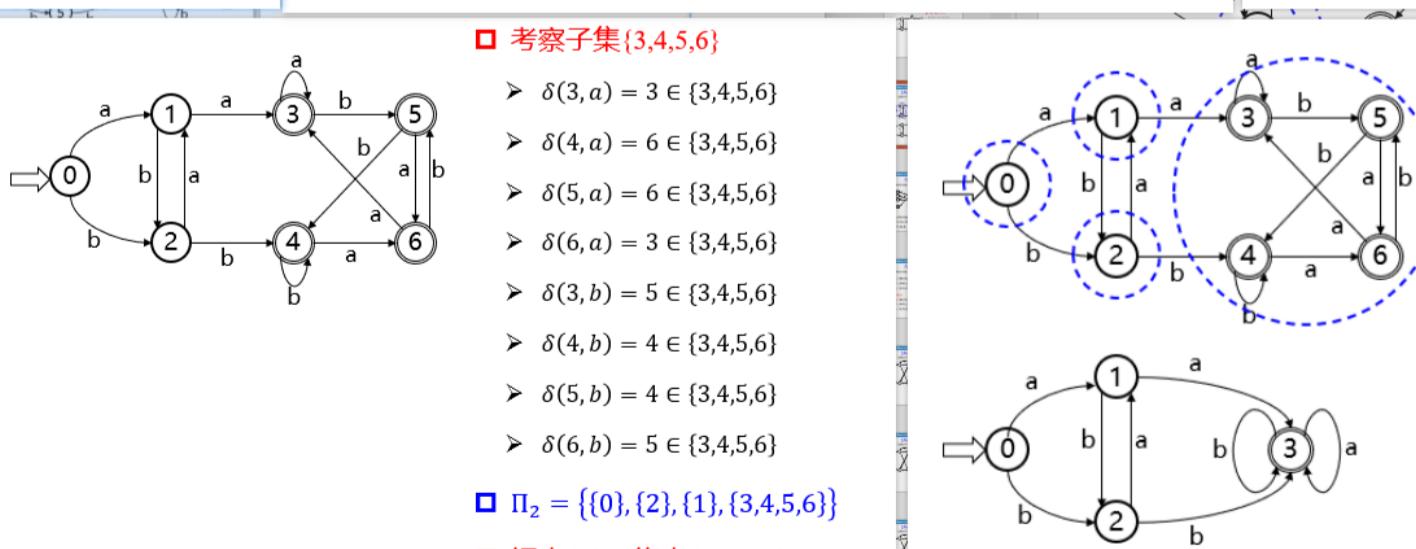
➢ $\delta(6, b) = 5 \in \{3,4,5,6\}$

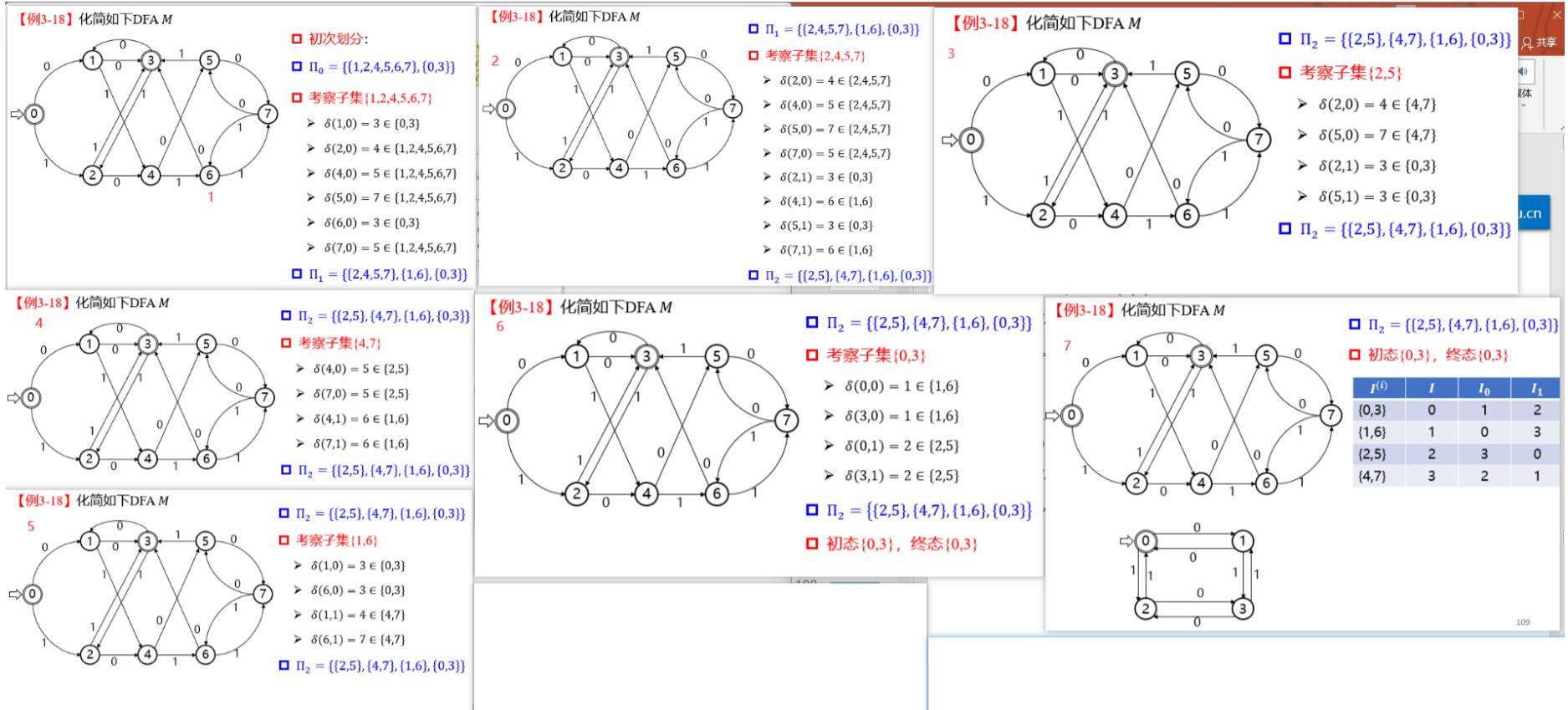
□ $\Pi_2 = \{\{0\}, \{2\}, \{1\}, \{3,4,5,6\}\}$

□ 初态 $\{0\}$ ，终态 $\{3,4,5,6\}$

□ $\Pi_2 = \{\{0\}, \{2\}, \{1\}, \{3,4,5,6\}\}$

□ 初态 $\{0\}$ ，终态 $\{3,4,5,6\}$





2.2 自顶向下分析：LL(1) 分析

自顶向下分析只包含 LL(1) 分析，这种题的考点有：

- 对于给定的文法，首先判断文法是否是 LL(1) 的，如果不是，需要将其转换为 LL(1) 文法。

一个文法是 LL(1) 的，当且仅当下列条件成立：

- 文法不含左递归（包含隐式左递归）；
- 产生式无公共子前缀：对于 $A \rightarrow \alpha_1 | \dots | \alpha_n$ 的每对候选式，有 $First(\alpha_i) \cap First(\alpha_j), i \neq j$ 成立；
- 文法无二义性：同一个句子不能由两棵不同的语法树推导而来。

如果一个文法符合上述条件，则它是 LL(1) 的，否则就需要采取下列措施来构造 LL(1) 文法：

- 消除左递归；
- 消除回溯（提左公因子）；
- 消除二义性。

- 对这个 LL(1) 文法构造 First 和 Follow 集合；

对任意非终结符 X ，构造 $First(X) = \{\}$ 作为初始状态，重复以下步骤，直到 $First(X)$ 不再增大为止：

- 若 $X \in V_T$ ，则 $First(X) = First(X) \cup \{X\}$ ；
- 若 $X \in V_N$ ，对于产生式 $X \rightarrow Y_1 Y_2 \dots Y_k$ ，则 $First(X) = First(X) \cup (First(Y_1) - \{\epsilon\})$ ，当且仅当 $Y_1 Y_2 \dots Y_k \Rightarrow \epsilon$ 时， $First(X) = First(X) \cup \{\epsilon\}$ 。

换句话说， $First(X)$ 是非终结符 X 能推导出的所有句子的首字符。

构造 Follow 集合时，对任意非终结符 A ，构造 $Follow(A) = \{\}$ 作为初始状态，对开始符号 S 构造 $Follow(S) = \{\#\}$ 。重复以下步骤，直到 $Follow(A)$ 不再增大为止：

- 对于产生式 $A \rightarrow \alpha B \beta$, $B \in V_N$, $\epsilon \notin First(\beta)$ ，则 $Follow(B) = Follow(B) \cup (First(\beta) - \{\epsilon\})$ ；
- 对于产生式 $A \rightarrow \alpha B \beta$, $B \in V_N$, $\epsilon \in First(\beta)$ ，则 $Follow(B) = Follow(B) \cup Follow(A)$ 。

换句话说， $Follow(A)$ 是非终结符 A 之后能跟的第一个字符的集合，开始符号 S 之后要跟结束符号 #，任意非终结符之后都不能跟 ϵ 。

- 根据构造出的 First 和 Follow 集合构造 LL(1) 分析表；

构造 LL(1) 分析表时，列为 $a \in V_T$ ，行为 $A \in V_N$ ， A 与 a 对应的元素记为 $M[A, a]$ 。对于每一个产生式 $A \rightarrow \alpha$ ，执行以下操作：

- 对于 $First(\alpha)$ 中的每一个元素 α_i ：
 - 若 $\alpha_i \neq \epsilon$ ，置 $M[A, \alpha_i] = A \rightarrow \alpha$ ；

- 若 $\alpha_i = \epsilon$, 对所有 $\beta_i \in Follow(A)$, 置 $M[A, \beta_i] = A \rightarrow \alpha$ 。
- 根据 LL(1) 分析表识别句子。

通过上面的构造, 我们已经有一个 LL(1) 分析表 M , 它是用于指导我们按照文法规则解析输入串的重要工具。在这个过程中, 我们使用开始符号作为解析的起点, 代表文法的初始状态。此外, 我们还使用一个特殊的结束符号 \$ 或 # 来标识输入串的结束。

我们的目标是分析输入串, 判断其是否符合给定的文法规则。为了实现这一目标, 我们采用以下步骤:

- 初始化文法符号栈: 在栈中先放入结束符号 \$ 或 # 和开始符号, 设定当前字符 c 为输入串的第一个字符;
- 开始解析过程, 重复以下步骤, 直到文法符号栈和输入串都为空表示解析成功, 或者遇到特定情况表示解析失败:

- 查看文法符号栈顶元素 E , 如果 $E \in V_T$:
 - E 与当前输入串字符 c 为同一个字符, 则从栈中移除 E , 并将 c 更新为输入串的下一个字符;
 - E 与当前输入串字符 c 为不同字符, 表示出现错误, 应该停止解析。

- 如果 $E \in V_N$:
 - 查找 LL(1) 分析表, 如果 $M[E, c] = E \rightarrow \alpha$, 从栈中移除 E , 将候选式 α 的每一个元素按照逆序压入栈中 (例如 $E \rightarrow TE'$ 的话就弹出 E 并依次压入 E' 和 T)。
 - 如果 $M[E, c]$ 为空, 表示没有合适的规则进行匹配, 应停止解析并报错。

【例4.11】构造文法 $G[E]$ 的 LL(1) 分析表

$E \rightarrow TE'$	$First(TE') = \{\(\), i\}$				$Follow(E) = \{#,)\}$
$E' \rightarrow +TE' \epsilon$	$First(+TE') = \{+\}$	$First(\epsilon) = \{\epsilon\}$			
$T \rightarrow FT'$	$First(FT') = \{\(\), i\}$				$Follow(T) = \{+, \#,)\}$
$T' \rightarrow *FT' \epsilon$	$First(*FT') = \{* \}$	$First(\epsilon) = \{\epsilon\}$			
$F \rightarrow (E) i$	$First((E)) = \{\(\)\}$	$First(i) = \{i\}$			

	i	+	*	()	#
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

④分析过程

序号	文法符号栈	输入串	所用产生式
1	# E	$i * (i + i) + i \#$	
2	# $E'T$	$i * (i + i) + i \#$	$E \rightarrow TE'$
3	# $E'T'F$	$i * (i + i) + i \#$	$T \rightarrow FT'$
4	# $E'T'i$	$i * (i + i) + i \#$	$F \rightarrow i$
5	# $E'T'$	$* (i + i) + i \#$	
6	# $E'T'F *$	$* (i + i) + i \#$	$T' \rightarrow *FT'$
7	# $E'T'F$	$(i + i) + i \#$	
8	# $E'T')E($	$(i + i) + i \#$	$F \rightarrow (E)$
9	# $E'T')E$	$i + i) + i \#$	
10	# $E'T')E'T$	$i + i) + i \#$	$E \rightarrow TE'$
11	# $E'T')E'T'F$	$i + i) + i \#$	$T \rightarrow FT'$

序号	文法符号栈	输入串	所用产生式
12	# $E'T')E'T'i$	$i + i) + i \#$	$F \rightarrow i$
13	# $E'T')E'T'$	$+i) + i \#$	
14	# $E'T')E'$	$+i) + i \#$	$T' \rightarrow \epsilon$
15	# $E'T')E'T +$	$+i) + i \#$	$E' \rightarrow +TE'$
16	# $E'T')E'T$	$i) + i \#$	
17	# $E'T')E'T'F$	$i) + i \#$	$T \rightarrow FT'$
18	# $E'T')E'T'i$	$i) + i \#$	$F \rightarrow i$
19	# $E'T')E'T'$	$) + i \#$	
20	# $E'T')E'$	$) + i \#$	$T' \rightarrow \epsilon$
21	# $E'T')$	$) + i \#$	$E' \rightarrow \epsilon$
22	# $E'T'$	$+i \#$	

序号	文法符号栈	输入串	所用产生式
22	#E'T'	+i#	
23	#E'	+i#	$T' \rightarrow \epsilon$
24	#E'T +	+i#	$E' \rightarrow +TE'$
25	#E'T	i#	
26	#E'T'F	i#	$T \rightarrow FT'$
27	#E'T'i	i#	$F \rightarrow i$
28	#E'T'	#	
29	#E'	#	$T' \rightarrow \epsilon$
30	#	#	$E' \rightarrow \epsilon$
31	#	#	Success

..

2.3 自底向上分析：LR(0) 分析

LL(1) 分析方法，虽然直观且易于实现，但它不能处理所有编程语言结构。它的主要局限性在于无法处理左递归和有共同左因子的语法，这在许多编程语言中是常见的。此外，LL(1) 分析要求必须能够通过向前看一个符号就确定如何继续解析，这在复杂语法结构中并不总是可能的。

为了解决这些问题，我们引入 LR(0) 自动机。LR(0) 自动机是一种更强大的解析工具，能够处理更复杂的语法结构，包括左递归和共同左因子。这种自动机基于移入-归约策略，可以有效地处理更广泛的语言特性。

要想使用 LR(0) 方法进行分析，对于给定的文法 $G_1 = \{V_N, V_T, P, S\}$ ，需要构造文法 $G_2 = \{V_N \cup \{S'\}, V_T, P \cup \{S' \rightarrow S\}, S'\}$ ，显然 $L(G_2) = L(G_1)$ ，我们称 G_2 为 G_1 的拓广文法。

每个 LR(0) 自动机的状态都由多个项组成，这些项是由规则和一个点 (.) 构成的，这个点标识解析器在该规则中的当前位置。例如，配置 $E \rightarrow E. + T$ 表示解析器当前已经处理了 E ，接下来可能处理的令牌序列是 $+T$ 。

构造 LR(0) 自动机的过程是这样的：

- 首先创建状态 0，方法是取开始符号的产生式（例如 $S' \rightarrow S$ ）并在右侧的开头添加一个点得到 $S' \rightarrow .S$ 。这个点表示我们期待看到一个完整的程序，但尚未开始处理任何符号。这被称为该状态的核心 (kernel)。
- 接下来，我们计算状态的闭包 (closure)。对于状态中点右侧紧接着的每个非终结符 X ，我们添加所有以 X 作为左侧的语法规则。新添加的项在右侧开始处有一个点。例如我们已经有了 $S' \rightarrow .S$ ，点右侧的符号是 S ，那么我们就对该状态添加所有以 S 为左侧的产生式，并在在每个产生式的第一个符号之前添加一个点，现在该状态可能变成：

$$\begin{aligned} S' &\rightarrow .S \\ S &\rightarrow .E + T \\ S &\rightarrow .a \end{aligned}$$

这个过程持续进行，直到不能添加新的项目为止，例如，上述状态的闭包可能包括：

$$\begin{aligned} S' &\rightarrow .S \\ S &\rightarrow .E + T \\ S &\rightarrow .a \\ E &\rightarrow .b \end{aligned}$$

- 每一个状态都包括一个核心 (kernel) 和一个闭包 (closure)，一旦构建出状态 I_0 后，我们应检查是否有产生式中的点可以右移。如果可以，将点右移跨过一个终结符或非终结符 X ，形成一个新状态的核心。然后，根据这个新核心再构建新状态的闭包，得到一个完整的新状态 I_1
- 当我们从状态 I_0 转移到新状态 I_1 时，这个转移可以在自动机的图形表示中用一条线来表示。在这条连线上，我们标记上符号 X ，这个符号表示导致状态转移的那个终结符或非终结符。状态转换可以用 $I_1 = Go(I_0, X)$ 来表示。
- 对于每个状态的每个产生式都采取同样的步骤，通过这种方式，我们逐步构建出自动机的所有状态，每个状态都基于其核心的产生式和由此产生的闭包。这个过程持续进行，直到无法生成更多新状态为止。这样，LR(0) 自动机就能够捕捉到解析过程中所有可能的状态，为自底向上的解析提供必要的信息。所有状态的集合及其转换关系称为项目集规范族。
- 一旦构建出项目集规范族，可以画出状态转换表，根据状态转换表可以构造出 LR(0) 分析表，LR(0) 分析表由两部分组成：ACTION 表和 GOTO 表，他们分别的构造步骤为：
 - ACTION 表：ACTION 表用于决定在遇到终结符时应采取的动作（移入、归约、接受或错误）。对于每个状态和终结符的组合，你需要决定以下动作之一：

- **移入 (Shift)** : 如果在某个状态下, 一个终结符后面有一个点, 并且这个点可以右移, 则在 ACTION 表中为该状态和终结符标记为移入, 并指明下一个状态。例如, 如果从状态 I_0 可以通过非终结符 a 到达状态 I_1 , 则置 $ACTION[I_0, a] = S_1$ 。
 - **归约 (Reduce)** : 如果某个状态包含一个点在最右端的产生式 (例如 $A \rightarrow \alpha.$) , 则在 ACTION 表中为该状态下的所有终结符标记为归约 $A \rightarrow \alpha$ (或用 R 加上产生式的序号 i 来表示)。
 - **接受 (Accept)** : 如果某个状态包含点在最右端的起始产生式 (例如 $S' \rightarrow S.$) , 则在 ACTION 表中为该状态和输入符号 \$ 或 # 标记为接受。
 - **错误 (Error)** : 其他情况通常标记为错误。
 - **GOTO 表** : GOTO 表用于在遇到非终结符时指导状态转移。对于每个状态和非终结符的组合, 如果从该状态出发, 存在一个以该非终结符开始的点右移动作, 则在 GOTO 表中记录下这种移动后达到的新状态。
- 使用 LR(0) 分析表来识别句子 : 使用 LR(0) 分析表来识别 (解析) 句子涉及一系列的移入 (shift) 和归约 (reduce) 操作, 直到整个句子被成功解析或者遇到错误。这个过程通常涉及一个分析栈和输入缓冲区。下面是使用 LR(0) 分析表进行句子识别的步骤 :
- 初始化 : 创建一个状态栈, 并将起始状态 (通常是状态 0) 压入栈中 ; 初始化一个符号栈, 将结束符号压入栈中 (通常是 \$ 或 #) ; 将待解析的句子放入输入缓冲区, 并在末尾添加一个结束符号。
 - 查看当前状态和输入 : 根据状态栈栈顶元素得到当前状态 s , 根据输入串的第一个项目得到终结符 c , 查找 $ACTION[s, c] = k$, 若 :
 - k 表示移入 : 将 k 指示的下一个状态压入状态栈中, 将 c 压入符号栈中, 然后从输入缓冲区移除这个符号 ;
 - k 表示规约 : 根据产生式 $A \rightarrow \beta$ 的右侧 β 的长度, 从状态栈和符号栈中弹出相应数量的元素。例如, 如果 β 的长度是 3, 则需要从状态栈和符号栈中各连续弹出3个元素。
 - 将产生式左侧的非终结符和通过 GOTO 表查找到的下一个状态压入栈中 ; 在弹出相应的符号和状态之后, 查看现在的栈顶状态, 记为 s' 。查找 $GOTO[A, s'] = n$, 表示要转到的新状态, 将 n 压入状态栈中, 同时将 A 压入符号栈中 ;
 - k 表示接受 : 表示整个句子已经成功解析 ;
 - k 为空 : 表示解析错误, 当前句子不能由当前的文法规则所解析。

例题 : 给定下列文法 G , 使用 LR(0) 分析法来识别句子 abc。

$$\begin{aligned} S &\rightarrow aBC \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

分析 : 首先构造拓广文法 G' :

$$\begin{aligned} 1. S' &\rightarrow S \\ 2. S &\rightarrow aBC \\ 3. B &\rightarrow b \\ 4. C &\rightarrow c \end{aligned}$$

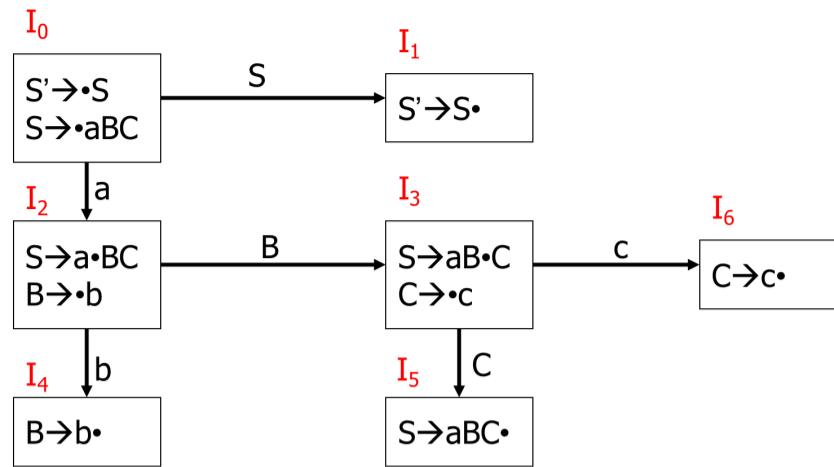
然后构造 G' 的项目集规范族 :

```

I0:           { S'->.S, S->.aBC }
I1 = Go(I0, S): { S'->S. }
I2 = Go(I0, a): { S->a.BC, B->.b }
I3 = Go(I2, B): { S->aB.C, C->.c }
I4 = Go(I2, b): { B->b. }
I5 = Go(I3, C): { S->aBC. }
I6 = Go(I3, c): { C->c. }

```

根据项目集规范族画出状态转换表 :



根据状态转换表构造 LR(0) 分析表：

状态	Action				Goto		
	a	b	c	#	S	B	C
0	S_2				1		
1				acc			
2		S_4				3	
3			S_6				5
4	r_3	r_3	r_3	r_3			
5	r_2	r_2	r_2	r_2			
6	r_4	r_4	r_4	r_4			

接着识别句子 abc：

序号	状态栈	符号栈	输入串
0	0	#	abc#
1	02	#a	bc#
2	024	#ab	c#
3	023	#aB	c#
4	0236	#aBc	#
5	0235	#aBC	#
6	0	#	#
	acc		

2.4 自底向上分析：SLR 分析

LR(0) 分析法的局限性在于：

- 移入-规约冲突：如果一个项目集中既有移入项目又有规约项目，则 $ACTION[i, a]$ 不唯一。例如，如果一个状态含有的项目集为 $\{A \rightarrow a.b, B \rightarrow b.\}$ ，则此时 $ACTION[i, a]$ 既可以置为移入，也可以置为规约，出现了冲突。
- 规约-规约冲突：如果一个项目集中含有两个或两个以上的规约项目，则 $ACTION[i, a]$ 不唯一。例如，如果一个状态含有的项目集为 $\{A \rightarrow a., B \rightarrow b.\}$ ，则此时 $ACTION[i, a]$ 有两个可选的规约项，出现了冲突。

为了解决这一局限性并扩展可处理的文法范围，SLR 分析器被提出。SLR 引入了向前看符号的概念，在遇到一个符号时，它不仅考虑当前的符号，还考虑接下来的一个符号，从而更准确地判断是进行移入操作还是归约操作。这种方法有效地解决了 LR(0) 分析器中遇到的大部分移入-归约冲突，使得 SLR 分析器能够处理更广泛的文法。

SLR 分析与 LR(0) 分析的不同点在于项目集规范族、状态转换表和 SLR 分析表构造的不同。

遵循 LR(0) 分析的方法，在构造出拓广文法 G' 和项目集规范族后，SLR 分析法要求对 G' 构造 First 和 Follow 集合，并根据 Follow 集合的元素来构建 SLR 分析表。

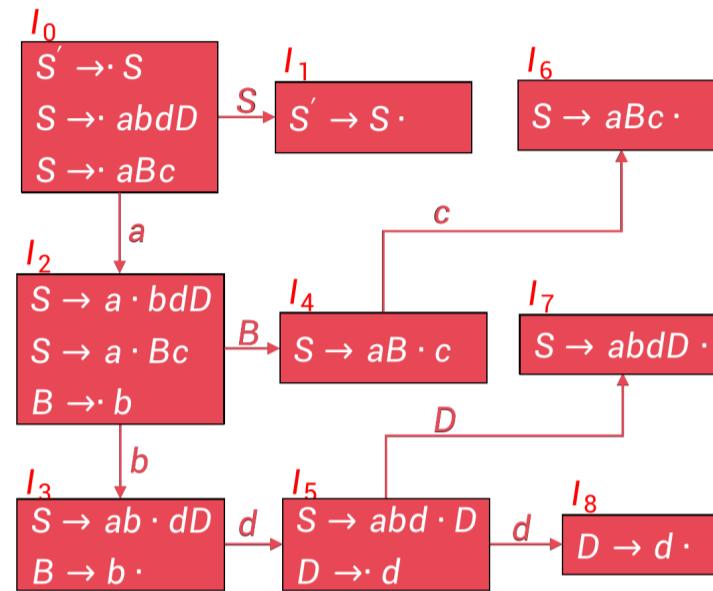
- 填充 ACTION 表：如果存在形如 $A \rightarrow \alpha.$ 的项目，则对于 $FOLLOW(A)$ 中的每个符号 b ，在 ACTION 表的相应单元格中标记为 归约操作 $A \rightarrow \alpha$ 。这意味着只有当输入中的下一个符号是 $FOLLOW(A)$ 中的符号时，才执行归约操作。**这是 SLR 分析和 LR(0) 分析的主要区别**。其余操作和 LR(0) 相同。
- 填充 GOTO 表：和 LR(0) 的方法相同。

通过这种方式，SLR 分析法利用 FOLLOW 集合有效地解决了 LR(0) 分析法中常见的移入-归约冲突，使得 SLR 分析表能够适用于更广泛的文法。然而，它依然难以解决规约-规约冲突。

例题：构造下列拓广文法 G' 的 SLR 分析表：

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow abdD \\ S &\rightarrow aBc \\ B &\rightarrow b \\ D &\rightarrow d \end{aligned}$$

分析：首先，使用 LR(0) 同样的方法画出其状态转换图：



为了构造 SLR 分析表，构造文法 G' 的 First 和 Follow 集合：

First(S') = { a }
First(S) = { a }
First(B) = { b }
First(D) = { d }

Follow(S') = { # }
Follow(S) = { # }
Follow(B) = { c }
Follow(D) = { # }

然后根据 SLR 的规则构造出状态转换表：

状态	Action					Goto		
	a	b	c	d	#	S	B	D
0	S_2					1		
1					acc			
2		S_3					4	
3			r_4	S_5				
4			S_6					
5				S_8				7
6					r_3			
7					r_2			
8					r_5			

2.5 自底向上分析：LR(1) 分析

SLR 分析器在构建分析表时，利用非终结符的 FOLLOW 集合来帮助决定归约操作。这种方法虽然简化了分析表的构建，但并不能处理规约-规约冲突，特别是在复杂文法中。

LR(1) 分析器的核心改进在于为每个项目引入一个向前看符号。这个向前看符号代表紧随当前项目之后的输入符号，例如， $A \rightarrow \alpha.\beta, a$ 表示在解析 $A \rightarrow \alpha\beta$ 时，下一个输入符号是 a 。通过这个向前看符号来决定下一步应该采取的操作，可以有效解决规约-规约冲突。

LR(1) 分析与 LR(0) 分析在构建项目集规范族时的主要区别在于，LR(1) 对每个产生式都添加了一个向前看符号集合 L ，规则为：

- 对于起始状态的核心（kernel）项，其向前看符号集合通常设定为 {#}，表示输入的结束；

- 当闭包项 (closure) 中包含形如 $A \rightarrow \alpha.B\beta, L$ 的项时, 其中 B 是非终结符, β 是一个可能为空的符号串, L 是当前项的向前看符号集合。你需要为 B 的每个产生式 $B \rightarrow .\gamma$ 添加新的项到闭包中;
- 新添加的项 $B \rightarrow .\gamma$ 的向前看符号集合取决于 β 和 L :
 - 如果 β 不为空且不能推导出空串 ϵ , 则新项的向前看符号集合是 $FIRST(\beta)$;
 - 如果 β 能推导出空串或为空, 则新项的向前看符号集合是 $FIRST(\beta) \cup L$ 。

在构造出项目集规范族和状态转移表后, 可以构造 LR(1) 分析表:

- 填充 ACTION 表: 如果项目集中包含形如 $A \rightarrow \alpha., a$ 的项, 表示在当前状态 i 下, 当向前看符号是 a 时 (即 $ACTION[i, a]$), 应使用产生式 $A \rightarrow \alpha$ 进行归约。这是 LR(1) 分析和 LR(0) 分析的主要区别。其余操作和 LR(0) 分析一样。
- 填充 GOTO 表: 和 LR(0) 的方法相同。

例1: 构造下列文法 G 的 LR(1) 分析表, 并识别句子 `aaaab` 和 `aab`:

$$\begin{aligned} S &\rightarrow aCaCb \\ S &\rightarrow aDb \\ C &\rightarrow a \\ D &\rightarrow a \end{aligned}$$

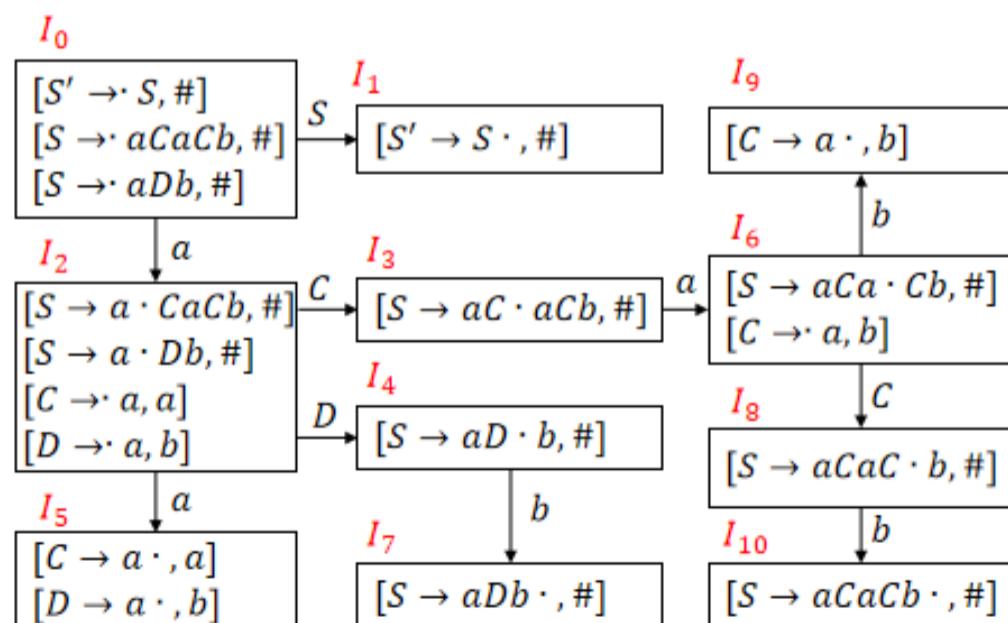
分析: 首先构造拓广文法 G' :

$$\begin{array}{l} 1. S' \rightarrow S \\ 2. S \rightarrow aCaCb \\ 3. S \rightarrow aDb \\ 4. C \rightarrow a \\ 5. D \rightarrow a \end{array}$$

然后对上述文法求 First 集合:

$First(S') = \{a\}$
$First(S) = \{a\}$
$First(C) = \{a\}$
$First(D) = \{a\}$

根据 LR(1) 分析法的规则, 构造项目集规范族:



根据项目集规范族构造 LR(1) 分析表:

状态	Action			Goto		
	a	b	#	S	C	D
0	S_2			1		
1			acc			
2	S_5				3	4
3	S_6					
4		S_7				
5	r_4	r_5				
6	S_9				8	
7			r_3			
8		S_{10}				
9		r_4				
10			r_2			

识别句子 `aaaab` 和 `aab` :

序号	状态栈	符号栈	输入串
1	0	#	<code>aaaab#</code>
2	02	#a	<code>aaab#</code>
3	025	#aa	<code>aab#</code>
4	023	#aC	<code>aab#</code>
5	0236	#aCa	<code>ab#</code>
6	02369	#aCa a	<code>b#</code>
7	02368	#aCa C	<code>b#</code>
8	023681 <u>0</u>	#aCa C b	<code>#</code>
9	01	#S	<code>#</code>
10	acc		

序号	状态栈	符号栈	输入串
1	0	#	<code>aab#</code>
2	02	#a	<code>ab#</code>
3	025	#aa	<code>b#</code>
3	024	#aD	<code>b#</code>
4	0247	#aDb	<code>#</code>
5	01	#S	<code>#</code>
6	acc		

2.4 中间代码生成

给出翻译模式和高级语言程序，翻译句子，一般涉及多种类型句子的综合，也可能涉及声明语句填写符号表。

1. 过程中的说明语句

7.2.1 过程中的说明语句

在 C、Pascal 及 FORTRAN 等语言的语法中,允许在一个过程中的所有说明语句作为一个组来处理,把它们安排在一所数据区中。从而我们需要一个全程变量如 offset 来跟踪下一个可用的相对地址的位置。

在图 7.6 关于说明语句的翻译模式中,非终结符号 P 产生一系列形如 id:T 的说明语句。在处理第一条说明语句之前,先置 offset 为 0,以后每次遇到一个新的名字,便将该名字填入符号表中并置相对地址为当前 offset 之值,然后使 offset 加上该名字所表示的数据对象的域宽。

$P \rightarrow D$	{ offset := 0)
$D \rightarrow D; D$	
$D \rightarrow id: T$	{ enter(id.name, T.type, offset); offset := offset + T.width }
$T \rightarrow integer$	{ T.type := integer; T.width := 4 }
$T \rightarrow real$	{ T.type := real; T.width := 8 }
$T \rightarrow array[num] of T_1$	{ T.type := array(num.val, T_1.type); T.width := num.val * T_1.width }
$T \rightarrow \uparrow T_1$	{ T.type := pointer(T_1.type); T.width := 4 }

图 7.6 计算说明语句中名字的类型和相对地址

过程 enter(name, type, offset) 用来把名字 name 填入到符号表中,并给出此名字的类型 type 及在过程数据区中的相对地址 offset。非终结符号 T 有两个综合属性 T.type 和 T.width,分别表示名字的类型和名字的域宽(即该类型名字所占用的存储单元个数)。在图 7.6 中,假定整数类型域宽为 4;实数域宽为 8;一个数组的域宽可以通过把数组元素数目与一个元素的域宽相乘获得;每个指针类型的域宽假定为 4。

如果把图 7.6 中的第一条产生式及其语义动作写在一行,则对 offset 赋初值更明显,如下式所示:

$$P \rightarrow \{ offset := 0 \} D \quad (7.1)$$

在 6.5 节曾谈到产生 ϵ 的标记非终结符号,可以用它来重新改写上述产生式以便语义动作均出现在整个产生式的右边。我们可采用标记非终结符号 M 来重写式(7.1):

$$\begin{aligned} P &\rightarrow M D \\ M &\rightarrow \epsilon \quad \{ offset := 0 \} . \end{aligned}$$

□ 【例7.8】分析句子: $p, q, r: real$

$D \rightarrow id L$	$\{enter(id.name, L.type, offset);$
	$offset = offset + L.width\}$
	$L.type = L_1.type; L.width = L_1.width\}$

步骤	文法符号栈	输入串
1	#	$p, q, r: real\#$
2	#M	$p, q, r: real\#$
3	#Mp, q, r: real	#
4	#Mp, q, r: T	#
5	#Mp, q, rL	#
6	#Mp, qL	#
7	#MpL	#
8	#MD	#
9	#P	#
10	acc	#

id	type	offset	offset=24
r	real	0	
q	real	8	
p	real	16	

$L.type = real; L.width = 8$

2. 算术表达式的翻译

□ 简单算术表达式及赋值语句的操作:

- $tblptr$: 是一个栈, 栈顶为当前过程的符号表, 所以可以取到符号信息。
- $lookup(name)$: 从 $top(tblptr)$ 符号表寻找名字, 找到即返回, 找不到则转到外层 (上层) 符号表继续查找, 直到找到或者所有外围过程都找不到为止。
- $gen(op, arg1, arg2, result)$: 生成三地址代码。
- $newtemp$: 是一个方法, 生成一个临时变量。
- $place$: 是一个属性, 存放文法符号的值 (变量) 的名字。

□ 简单算术表达式及赋值语句:

$S \rightarrow id = E$	$\{p = lookup(id.name);$
	$if p \neq null \text{ then } gen(=, E.place, -, p); \text{ else error; }\}$
$E \rightarrow E_1 + E_2$	$\{E.place = newtemp; gen(+, E_1.place, E_2.place, E.place)\}$
$E \rightarrow E_1 * E_2$	$\{E.place = newtemp; gen(*, E_1.place, E_2.place, E.place)\}$
$E \rightarrow - E_1$	$\{E.place = newtemp; gen(@, E_1.place, -, E.place)\}$
$E \rightarrow (E_1)$	$\{E.place = E_1.place\}$
$E \rightarrow id$	$\{p = lookup(id.name);$
	$if p \neq null \text{ then } E.place = p; \text{ else error; }\}$

$S \rightarrow id = E \quad \{p = lookup(id.name);$ $\text{if } p \neq \text{null} \text{ then } gen(=, E.place, -, p); \text{ else error;}\}$	$E.place\})$
---	--------------

步骤	文法符号栈	输入串	动作
1	#	$x = (a + b) * -c \#$	初始
2	$\#x = (a$	$+b) * -c \#$	移进
3	$\#x = (E$	$+b) * -c \#$	归约
4	$\#x = (E + b$	$) * -c \#$	移进
5	$\#x = (E + E$	$) * -c \#$	归约
6	$\#x = (E$	$) * -c \#$	归约
7	$\#x = (E)$	$* -c \#$	移进
8	$\#x = E$	$* -c \#$	归约
9	$\#x = E * -c$	#	移进
10	$\#x = E * -E$	#	归约
11	$\#x = E * E$	#	归约
12	$\#x = E$	#	归约
13	$\#S$	#	归约
14	$\#S$	#	成功

三地址码

(+, a, b, T1)
(@, c, -, T2)
(*, T1, T2, T3)
(=, T3, -, x)

 $E.place = T2$ $E.place = T3$ 栈属性
(给人看的)

38

3. 布尔表达式的翻译

重点：回填。

- (1) $E \rightarrow E_1 \vee ME_2 \quad \{backpatch(E_1.falselist, M.quad);$
 $E.truelist = merge(E_1.truelist, E_2.truelist);$
 $E.falselist = E_2.falselist;\}$
- (2) $E \rightarrow E_1 \wedge ME_2 \quad \{backpatch(E_1.truelist, M.quad);$
 $E.falselist = merge(E_1.falselist, E_2.falselist);$
 $E.truelist = E_2.truelist;\}$
- (3) $M \rightarrow \epsilon \quad \{M.quad = nxq;\} // 在 E_2 之前把它记下来， E_2 之后使用。$
- (4) $E \rightarrow \neg E_1 \quad \{E.truelist = E_1.falselist;$
 $E.falselist = E_2.truelist;\}$
- (5) $E \rightarrow (E_1) \quad \{E.truelist = E_1.truelist;$
 $E.falselist = E_2.falselist;\}$
-
- (6) $E \rightarrow id_1 \theta id_2 \quad \{E.truelist = mklist(nxq);$
 $E.falselist = mklist(nxq + 1);$
 $gen(j\theta, id_1.place, id_2.place, 0);$
 $gen(j, -, -, 0);\}$
- (7) $E \rightarrow id \quad \{E.truelist = mklist(nxq);$
 $E.falselist = mklist(nxq + 1);$
 $gen(jnz, id.place, -, 0);$
 $gen(j, -, -, 0);\}$

【例7.15】 布尔表达式: $a < b \vee c \leq d \wedge e$, 假设 $nxq = 100$

$E \rightarrow id \quad \{E.\text{truelist} = \text{mklist}(nxq);$

$E \rightarrow E_1 \vee ME_2 \quad \{backpatch(E_1.\text{falselist}, M.\text{quad});$
 $E.\text{truelist} = \text{merge}(E_1.\text{truelist}, E_2.\text{truelist});$
 $E.\text{falselist} = E_2.\text{falselist};\}$

步骤	文法符号栈	输入串	动作
1	#	$a < b \vee c \leq d \wedge e \#$	初始
2	# $a < b$	$\vee c \leq d \wedge e \#$	移进
3	# E	$\vee c \leq d \wedge e \#$	归约
4	# $E \vee$	$c \leq d \wedge e \#$	移进
5	# $E \vee M$	$c \leq d \wedge e \#$	归约
6	# $E \vee Mc \leq d$	$\wedge e \#$	移进
7	# $E \vee ME$	$\wedge e \#$	归约
8	# $E \vee ME \wedge$	$e \#$	移进
9	# $E \vee ME \wedge M$	$e \#$	归约
10	# $E \vee ME \wedge Me$	$\#$	移进
11	# $E \vee ME \wedge ME$	$\#$	归约
12	# $E \vee ME$	$\#$	归约
13	# E	$\#$	归约
14	# E	$\#$	成功

三地址码

100: ($j <, a, b, 0$)
101: ($j, -, -, 102$)
102: ($j \leq, c, d, 104$)
103: ($j, -, -, 0$)
104: ($jnz, e, -, 100$)
105: ($j, -, -, 103$)

两个未填充四元式链，需要等到确定布尔式为真做什么、为假做什么时才能回填。

$E.\text{truelist} = 104, E.\text{falselist} = 105$
$M.\text{quad} = 104$
$E.\text{truelist} = 104, E.\text{falselist} = 105$
$M.\text{quad} = 102$
$E.\text{truelist} = 104, E.\text{falselist} = 105$

4. 控制流语句的翻译

重点：if 和 while。

- (1) $S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$
- ```
{backpatch(E.truelist, M1.quad);
 backpatch(E.falselist, M2.quad);
 S.nextlist = merge(S1.nextlist, N.nextlist, S2.nextlist); }
```
- (2)  $M \rightarrow \epsilon$  { $M.\text{quad} = nxq;$ } // 在 $S$ 之前把它记下来， $S$ 之后使用。
- (3)  $N \rightarrow \epsilon$  { $N.\text{nextlist} = \text{mklist}(nxq);$ }
- ```
gen(j, -, -, 0); } // 跳到 $S_2$ 之后，也就是整个语句之后
```
- (4) $S \rightarrow \text{if } E \text{ then } MS_1$
- ```
{backpatch(E.truelist, M.quad);
 S.nextlist = merge(E.falselist, S1.nextlist); }
```
- 
- (5)  $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$
- ```
{backpatch(S1.nextlist, M1.quad);
 backpatch(E.truelist, M2.quad);
 S.nextlist = E.falselist;
 gen(j, -, -, M1.quad); }
```
- (6) $S \rightarrow \text{begin } L \text{ end}$ { $S.\text{nextlist} = L.\text{nextlist};$ }
- (7) $S \rightarrow A$ { $S.\text{nextlist} = \text{mklist}();$ } // 初始化为空表
- (8) $L \rightarrow L_1; MS$ { $\text{backpatch}(L_1.\text{nextlist}, M.\text{quad});$ } // L_1 的结束是 S 的开始
- ```
L.nextlist = S.nextlist; }
```
- (9)  $L \rightarrow S$  { $L.\text{nextlist} = S.\text{nextlist};$ }

**【例7.16】续例 【7.15】** if  $a < b \vee c \leq d \wedge e$  then  $x = y + z$  else  $x = 0$

$S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

```
{backpatch(E.truelist, M1.quad);
 backpatch(E.falselist, M2.quad);
 S.nextlist = merge(S1.nextlist, N.nextlist, S2.nextlist); }
```

|    |                | x | y | + | z | c | x           | = | 0         |
|----|----------------|---|---|---|---|---|-------------|---|-----------|
| 3  | #iEtM          |   |   |   |   |   | $x = y + z$ | e | $x = 0\#$ |
| 4  | #iEtMx = y     |   |   |   |   |   | $+z$        | e | $x = 0\#$ |
| 5  | #iEtMx = E     |   |   |   |   |   | $+z$        | e | $x = 0\#$ |
| 6  | #iEtMx = E + z |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 7  | #iEtMx = E + E |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 8  | #iEtMx = E     |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 9  | #iEtMA         |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 10 | #iEtMS         |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 11 | #iEtMSN        |   |   |   |   |   | $e$         | x | $= 0\#$   |
| 12 | #iEtMSNe       |   |   |   |   |   | $x$         | = | $0\#$     |
| 13 | #iEtMSNeM      |   |   |   |   |   | $x$         | = | $0\#$     |
| 14 | #iEtMSNeMx = 0 |   |   |   |   |   | #           |   |           |
| 15 | #iEtMSNeMx = E |   |   |   |   |   | #           |   |           |
| 16 | #iEtMSNeMA     |   |   |   |   |   | #           |   |           |
| 17 | #iEtMSNeMS     |   |   |   |   |   | #           |   |           |
| 18 | #S             |   |   |   |   |   | #           |   |           |

现在剩下最后一个链 $S.\text{nextlist}$ ，  
此处我们手工使  
用 $nxq$ 填充。

| 三地址码                         |
|------------------------------|
| 100: ( $j <, a, b, 106$ )    |
| 101: ( $j, -, -, 102$ )      |
| 102: ( $j \leq, c, d, 104$ ) |
| 103: ( $j, -, -, 109$ )      |
| 104: ( $jnz, e, -, 106$ )    |
| 105: ( $j, -, -, 109$ )      |
| 106: ( $+, y, z, T1$ )       |
| 107: ( $=, T1, -, x$ )       |
| 108: ( $j, -, -, 110$ )      |
| 109: ( $=, 0, -, x$ )        |

|                                   |
|-----------------------------------|
| $S.\text{nextlist} = \text{null}$ |
| $M.\text{quad} = 109$             |
| $N.\text{nextlist} = 108$         |
| $S.\text{nextlist} = \text{null}$ |
| $M.\text{quad} = 106$             |
| $S.\text{nextlist} = 108$         |

**【例7.17】** while  $a < b$  do if  $c < d$  then  $x = y + z$ , 假设  $nxq = 100$

|                                                                                                                                                                                   |                          |     |                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|-----|-----------------------|
| <i>S → while M<sub>1</sub>E do M<sub>2</sub>S<sub>1</sub></i>                                                                                                                     |                          | ; } | 三地址码                  |
| {backpatch(S <sub>1</sub> .nextlist, M <sub>1</sub> .quad);<br>backpatch(E.truelist, M <sub>2</sub> .quad);<br>S.nextlist = E.falselist;<br>gen(j, -, -, M <sub>1</sub> .quad); } |                          |     |                       |
| 1 #                                                                                                                                                                               | wa < bdic < dtx = y + z# |     | 100: (j <, a, b, 102) |
| 2 #w                                                                                                                                                                              | a < bdic < dtx = y + z#  |     | 101: (j, -, -, 107)   |
| 3 #wM                                                                                                                                                                             | a < bdic < dtx = y + z#  |     | 102: (j <, c, d, 104) |
| 4 #wMa < b                                                                                                                                                                        | dic < dtx = y + z#       |     | 103: (j, -, -, 100)   |
| 5 #wME                                                                                                                                                                            | dic < dtx = y + z#       |     | 104: (+, y, z, T1)    |
| 6 #wMED                                                                                                                                                                           | ic < dtx = y + z#        |     | 105: (=, T1, -, x)    |
| 7 #wMEDM                                                                                                                                                                          | ic < dtx = y + z#        |     | 106: (j, -, -, 100)   |
| 8 #wMEDMic < d                                                                                                                                                                    | tx = y + z#              |     |                       |
| 9 #wMEDMiE                                                                                                                                                                        | tx = y + z#              |     |                       |
| 10 #wMEDMiEt                                                                                                                                                                      | x = y + z#               |     |                       |
| 11 #wMEDMiEtM                                                                                                                                                                     | x = y + z#               |     |                       |
| 12 #wMEDMiEtMx = y + z                                                                                                                                                            | #                        |     |                       |
| 13 #wMEDMiEtMA                                                                                                                                                                    | #                        |     |                       |
| 14 #wMEDMiEtMS                                                                                                                                                                    | #                        |     |                       |
| 15 #wMEDMS                                                                                                                                                                        | #                        |     |                       |
| 16 #S                                                                                                                                                                             | #                        |     |                       |

此处如果用  $nxq$  填充  $S.nextlist$ , 应为:  
103: (j, -, -, 106)

此处用  $nxq$  手工填充  $S.nextlist$

|                                       |
|---------------------------------------|
| $S.nextlist = null$                   |
| $M.quad = 104$                        |
| $S.nextlist = 103$                    |
| $M.quad = 102$                        |
| $E.truelist = 100, E.falselist = 101$ |
| $S.nextlist = 101$                    |

## 2.5 目标代码生成

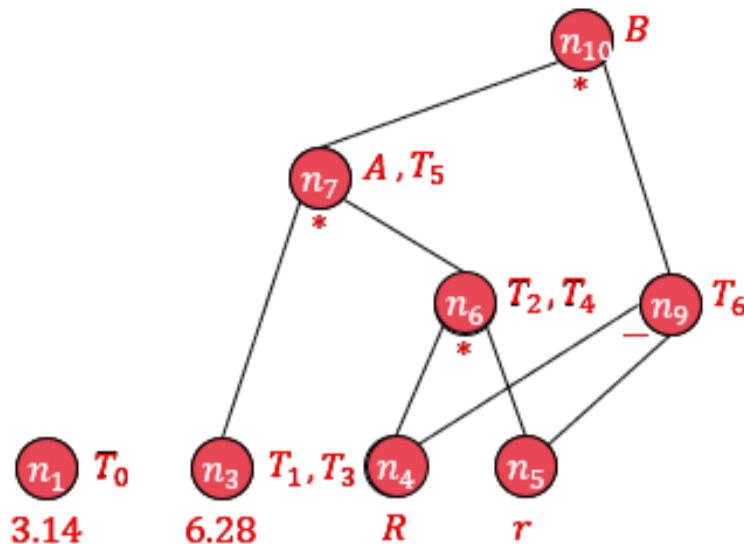
给出基本块代码：①构造DAG ②写出优化后的中间代码 ③写出DAG目标优化后的中间代码 ④根据变量活跃性和寄存器信息，写出目标代码。

### 例题1

给出基本块代码为：

- (1)  $T_0 = 3.14$
- (2)  $T_1 = 2 * T_0$
- (3)  $T_2 = R * r$
- (4)  $A = T_1 * T_2$
- (5)  $B = A$
- (6)  $T_3 = 2 * T_0$
- (7)  $T_4 = R * r$
- (8)  $T_5 = T_3 * T_4$
- (9)  $T_6 = R - r$
- (10)  $B = T_5 * T_6$

### ①构造DAG



②写出优化后的中间代码

|                      |
|----------------------|
| (1) $T_0 = 3.14$     |
| (2) $T_1 = 6.28$     |
| (3) $T_3 = 6.28$     |
| (4) $T_2 = R * r$    |
| (5) $T_4 = T_2$      |
| (6) $A = 6.28 * T_2$ |
| (7) $T_5 = A$        |
| (8) $T_6 = R - r$    |
| (9) $B = A * T_6$    |

③写出DAG目标优化后的中间代码

```

(1) T6 = R - r
(2) T2 = R * r
(3) T4 = T2
(4) T1 = 6.28
(5) T3 = 6.28
(6) A = 6.28 * T2
(7) T5 = A
(8) B = A * T6
(9) T0 = 3.14

```

④根据变量活跃性和寄存器信息，写出目标代码

假定  $B$  是基本块出口之后活跃的，有寄存器  $R0$  和  $R1$  可用，目标代码为：

DAG 优化中，不活跃变量，目标代码依然要生成计算其值的代码，只是不生成存储到主存的代码。计算代码被优化是后续优化完成的，不是 DAG 完成的。

```

LD R0, R
SUB R0, r R0: T6 R1: \
LD R1, R
MUL R1, r R0: T6 R1: T2
ST R0, T6

```

```

LD R0, 6.28 R0: 6.28 R1: T2
MUL R0, R1 R0: A R1: T2
MUL R0, T6 R0: B R1: T2
ST R0, B

```

### 三、历年考试回忆

2004-2005

一、(30) 问答

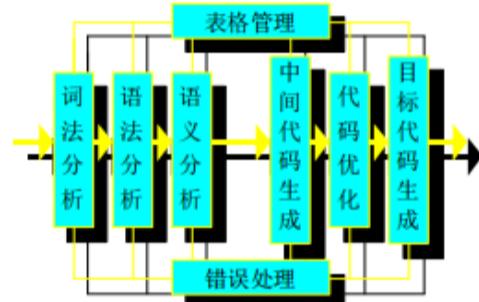
1、解释 LL(1)文法

答：一个文法满足下面条件：

- (1) 文法不含左递归
- (2) 对于文法中每一个非终结符 A 的各个产生式的候选首符集量不相交。
- (3) 对于文法中的每个非终结符 A，若它存在某个候选首符集包含  $\epsilon$ ，那么  $\text{First}(A) \cap \text{Follow}(A) = \emptyset$

则称该文法为 LL(1)文法。

2、出编译程序总框图。



3、写出活跃变量数据流方程及在编译中的应用。

答：数据流方程：

$$\text{Liveout}(B) = \bigcup_{i \in S(B)} \text{Livein}(i)$$

$$\text{Livein}(B) = \text{Liveuse}(B) \cup (\text{Liveout}(B) - \text{Def}(B))$$

在编译中的应用：(1) 优化 (2) 目标代码生成 (3) 检查变量定值之前被引用

4、写出语言  $L = \{a^n b^m c^m d^n | n \geq 1 \text{ 且 } m \geq 1\}$  的上下文无关文法。

答： $S \rightarrow aSdjaAd$   
 $A \rightarrow bAc \mid bc$

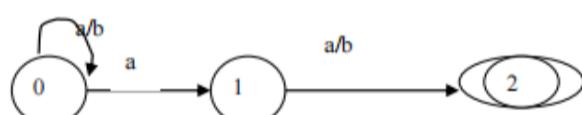
5、解释语法制导翻译法。

答：从概念上讲，基于属性文法的处理过程如下：对单词符号串进行语法分析，构造语法分析树，然后根据需要遍历语法树并在语法树的各结点处按语义规则进行计算。这种有源程序的语法结构驱动的处理办法就是语法制导翻译法

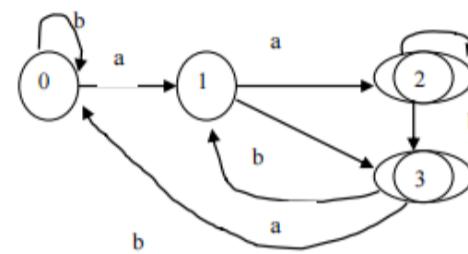
二、(15) 构造下列正规式相应的最简 DFA

$$(ab)^* a(ab)$$

解：1、由正规式到 NFA



|         | a       | b     |
|---------|---------|-------|
| {0}     | {0,1}   | {0}   |
| {0,1}   | {0,1,2} | {0,2} |
| {0,1,2} | {0,1,2} | {0,2} |
| {0,2}   | {0,1}   | {0}   |



3、DFA 最小化

初始分划：{0,1} {2,3}

因为 {0,1} ≠ {1,2} 不属于上面任意一个子集，所以分开：{0}, {1}

因为 {2,3} ≠ {1,2} 不属于上面任意一个子集，所以分开：{2}, {3}

已经是简

三、(20) 文法 G:

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow i$$

$$R \rightarrow L$$

文法 G 是 LALR(1) 文法吗？如果是构造出分析表。

解答：1、 $S \rightarrow L = R$

$$2、S \rightarrow R$$

$$3、L \rightarrow *R$$

$$4、L \rightarrow i$$

$$5、R \rightarrow L$$

首先拓广文法

$$S' \rightarrow S$$

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow i$$

$$R \rightarrow L$$

构造 LR(1)项目集规范组

I0:  $S' \rightarrow \bullet S \#$     $S \rightarrow \bullet L = R \#$     $S \rightarrow \bullet R \#$     $L \rightarrow \bullet^* R = / \#$     $L \rightarrow \bullet i = / \#$     $R \rightarrow \bullet L \#$   
I1:  $S' \rightarrow S \bullet \#$   
I2:  $S \rightarrow L \bullet = R \#$     $R \rightarrow L \bullet \#$   
I3:  $S \rightarrow R \bullet \#$   
I4:  $L \rightarrow \bullet^* R = / \#$     $R \rightarrow \bullet L = / \#$     $L \rightarrow \bullet^* R = / \#$     $L \rightarrow \bullet i = / \#$   
I5:  $L \rightarrow i \bullet = / \#$   
I6:  $S \rightarrow L = \bullet R \#$     $L \rightarrow \bullet i \#$     $R \rightarrow \bullet L \#$     $L \rightarrow \bullet^* R \#$   
I7:  $L \rightarrow \bullet^* R \bullet = / \#$   
I8:  $R \rightarrow L \bullet = / \#$   
I9:  $S \rightarrow L = R \bullet \#$   
I10:  $L \rightarrow i \bullet \#$   
I11:  $L \rightarrow \bullet^* R \#$     $R \rightarrow \bullet L \#$     $L \rightarrow \bullet^* R \#$     $L \rightarrow \bullet i \#$   
I12:  $R \rightarrow L \bullet \#$   
I13:  $L \rightarrow \bullet^* R \bullet \#$

因为每一个项目集都不存在冲突，所以该文法为 LR(1) 文法。合并同心集没有冲突，所以是 LALR(1) 文法。

I0:  $S' \rightarrow \bullet S \#$     $S \rightarrow \bullet L = R \#$     $S \rightarrow \bullet R \#$     $L \rightarrow \bullet^* R = / \#$     $L \rightarrow \bullet i = / \#$     $R \rightarrow \bullet L \#$   
I1:  $S' \rightarrow S \bullet \#$   
I2:  $S \rightarrow L \bullet = R \#$     $R \rightarrow L \bullet \#$   
I3:  $S \rightarrow R \bullet \#$   
I4:  $L \rightarrow \bullet^* R = / \#$     $R \rightarrow \bullet L = / \#$     $L \rightarrow \bullet^* R = / \#$     $L \rightarrow \bullet i = / \#$   
I5:  $L \rightarrow i \bullet = / \#$   
I6:  $S \rightarrow L = \bullet R \#$     $L \rightarrow \bullet i \#$     $R \rightarrow \bullet L \#$     $L \rightarrow \bullet^* R \#$   
I7:  $L \rightarrow \bullet^* R \bullet = / \#$   
I8:  $R \rightarrow L \bullet = / \#$   
I9:  $S \rightarrow L = R \bullet \#$

分析表为

| 状态 | Action | *  | =   | I  | # | S | Goto L | R |
|----|--------|----|-----|----|---|---|--------|---|
| 0  | s4     |    | s5  |    |   | 1 | 2      | 3 |
| 1  |        |    | acc |    |   |   |        |   |
| 2  |        | s6 |     | r5 |   |   |        |   |
| 3  |        |    |     | r2 |   |   |        |   |
| 4  | s4     |    | s5  |    |   | 8 | 7      |   |
| 5  |        |    | r4  | r4 |   |   |        |   |
| 6  | s4     |    | s5  |    |   | 8 | 9      |   |
| 7  | r3     |    |     | r3 |   |   |        |   |
| 8  | r5     |    |     | r5 |   |   |        |   |
| 9  |        |    |     | r1 |   |   |        |   |

四、(15) 用语法制导翻译的思想，把下面的语句翻译成四元式序列。

```

If a=1 or c<5 then
 While a<c and b<d do
 c=c+1
 else c=c+2

```

解：

| 分析栈                        | 输入栈    | 动作                    |
|----------------------------|--------|-----------------------|
| If a=1 or m1 c<5 then m2   | if a=1 | E11.t=100             |
| While m3 a<c and m4 b<d do | if E11 | E11.f=101             |
| m5 c=c+1 N                 |        | 100: (j=a,1,0) → 104  |
|                            |        | 101: (j,-,0) → 102    |
|                            |        | m1.q=102              |
|                            |        | E12.t=102             |
|                            |        | E12.f=103             |
|                            |        | 102: (j<,c,5,0) → 104 |
|                            |        | 103: (j,-,-,0) → 112  |
|                            |        | Backpatch(E11.f,m1.q) |
|                            |        | E11.t={100,102}       |
|                            |        | E11.f=103             |
|                            |        | m2.q=104              |
|                            |        | m3.q=104              |
|                            |        | E21.t=104             |
|                            |        | E21.f=105             |
|                            |        | 104: (j<,a,c,0) → 106 |
|                            |        | 105: (j,-,-,0)        |
|                            |        | m4.q=106              |
|                            |        | E22.t=106             |
|                            |        | E22.f=107             |
|                            |        | 106: (j<,b,d,0) → 108 |
|                            |        | 107: (j,-,-,0)        |
|                            |        | Backpatch(E21.t,m4.q) |
|                            |        | E21.t={106}           |
|                            |        | E21.f={105,107}       |

## 2015-2016

### 一、简答 (30 分)

1. 给了一个文法（具体忘了），让证明二义性。
2. 写出文法，表示{0、1}集上的所有正规式
3. 解释 S-属性文法
4. 说出传地址和传质这两种参数传递方式的异同
5. 结合具体例子谈一谈回填思想

### 二、( $b^*abbb^*$ )\* 画 NFA 转 DFA，最小化 (15 分)

### 三、(1) 一个文法，判断是否为 LL (1) 文法 (10 分)

- (2) 上题中的文法是否为 SLR 文法，给出证明 (15 分)  
(与课本上例子不同之处在于有  $B \rightarrow \epsilon$ )

### 四、利用语法指导思想写四元式 (15 分)

(比较简单，就是一个 while-do 语句)

### 五、给出一段中间代码，画 DAG 图；只有 R 在代码块外是活跃的，做优化；如果有寄存器 R0 和 R1，写出目标代码。 (15 分)

## 2016-2017

### 一、简答题

1. 编译流程图及各部分的作用。2. 举例说明文法二义性。
3. 什么是 L 属性文法？
4. 写出允许过程嵌套的 C 活动记录结构。5. 中间代码优化有哪几种？
6. 符号表作用。

### 二、词法分析

RE NFA DFA 最小化 DFA 整个流程走一遍

### 三、自上而下文法分析

给了一个文法，证明文法是LL1的，画表。

### 四、自下而上文法分析

给了一个文法，证明文法是LR1的，画表。

### 五、中间代码生成

一段代码，四元式翻译。涉及循环和判断的嵌套。

### 六、代码优化和目标代码生成。

给了一个代码段，

先画DAG图，然后优化，最后输出汇编代码。

## 2017-2018

### 山东大学 2017-2018 学年 1 学期 编译原理与技术 课程试卷 B

|                                                                                                                                                              |     |                                                                                                                                                                                          |   |   |   |   |   |   |   |   |    |     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---|---|---|---|---|---|---|----|-----|
| 题号                                                                                                                                                           | 一   | 二                                                                                                                                                                                        | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 | 总分 | 阅卷人 |
| 得分                                                                                                                                                           |     |                                                                                                                                                                                          |   |   |   |   |   |   |   |   |    |     |
| 得分                                                                                                                                                           | 阅卷人 | 四、(15分)<br>用语法制导翻译的思想，把下面的语句翻译成三地址码序列。<br>While a<c and b<d do if a=1 then c=c+1 else c=c+2                                                                                              |   |   |   |   |   |   |   |   |    |     |
| 一、简答题(40分)<br>1. 编译程序分哪几个主要部分？各部分的主要功能是什么？<br>2. 描述 LR 语法分析算法。<br>3. 解释语法制导翻译。<br>4. $L(G) = \{a^n b^n \mid n \geq 1\}$ ，试求上下文无关文法 G。<br>5. C 语言活动记录的结构是怎样的？ |     |                                                                                                                                                                                          |   |   |   |   |   |   |   |   |    |     |
| 得分                                                                                                                                                           | 阅卷人 | 二、(15分) 已知正规式 $(a b)^*(aa bb)(a b)^*$<br>构造其 NFA，并将其确定化，最小化。                                                                                                                             |   |   |   |   |   |   |   |   |    |     |
| 得分                                                                                                                                                           | 阅卷人 | 五、(15分) 对基本块 B:<br>$T_0 = 2$<br>$T_1 = 2*T_0$<br>$T_2 = A+B$<br>$T_3 = C*D$<br>$T_4 = T_1/2$<br>$T_5 = E+T_3$<br>$T_6 = C*D$<br>$R = T_3$<br>$X = T_2 - T_5$<br>$Y = T_1*T_6$<br>$R = X$ |   |   |   |   |   |   |   |   |    |     |
| 得分                                                                                                                                                           | 阅卷人 | 三、(15分)<br>证明下面文法消除左递归后是 LL(1) 文法。<br>$E \rightarrow E + T \mid T$<br>$T \rightarrow T * F \mid F$<br>$F \rightarrow (E) \mid i$                                                         |   |   |   |   |   |   |   |   |    |     |

第 1 页 共 1 页

## 2019-2020

### 一、简答题 (25分)

1. 判断一个文法是否二义
2. 编译的前端，后端，什么是一遍扫描
3. 什么是S属性
4. 什么是语法制导翻译
5. 在语法制导翻译中，空返产生式的作用 ( $M \rightarrow e$ )

### 二、计算题 (75分)

1. 一个单词表由 a, b 组成，请写出代表偶数个 a 的正规式，NFA，并确定化、最小化
2. 判断一个文法是不是 LL(1) 的，如果是就写出预测分析表，不是就说明原因 (15分)

3. 判断一个文法是不是SLR (1) 的，如果是就写出预测分析表，不是就说明原因（15分）

4. 中间代码生成程序（15分）

while a<c and b<d do if c==1 then c:=c+1 else c:=c+2;

5. 代码优化（15分）

DAG优化，最后写出四元式的形式（这个是一个坑，四元式是目标代码，也就是此时要做目标代码生成），同时目标代码生成要列表（Rvalue 寄存器描述，Avalue地址描述）。

## 2020-2021

### 《编译原理》试题C卷

简答

1. 编辑程序分为哪几个主要部分？简述各部分的主要功能。

2. 什么是综合属性和继承属性？

3. 解释语法制导翻译。

二. 证明下面文法是LALR (1) 文法。

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow i$

$R \rightarrow L$

三. 构造下列正规式相应的最简DFA

$1 (0 \mid 1)^* 101$

四. 用语法制导翻译的思想，把下面的语句翻译成三地址码序列。

While a < b do  
If c < d then x:=y+z

五. 对基本块B：

T0= 2

T1= 2\*T0

T2= A+B

T3= C\*D

T4= T1/2

T5= E+T3

T6= C\*D

R= T3

X= T2 - T5

Y= T1\*T6

R= X

1.构造B的DAG

2.若只有R在B出口之后是活跃的，写出优化之后的中间代码。若有寄存器R0,R1可用，将优化之后的中间代码生成目标代码。

## 2021 Fall

### 一、简答题

1. 画编译流程图；

2. 判断一个文法是不是二义文法；

3. 给一个句型，找它的句柄；

4. 中缀表达式转后缀，比较长，最好使用算法一步步推导；

5. 消除循环左递归；

6. 给一个 DFA，把它转换为正规式

### 二、计算题

1. 词法分析：给定正规式 ①构造NFA ②确定化 ③最小化

2. LL(1)分析，给出文法 ①构造First集合 ②构造Follow集合 ③构造LL(1)分析表（可能涉及消除二义文法冲突）④识别句子

3. LR(1)分析，给出文法 ①构造拓广文法 ②构造拓广文法的LR(1)项目集规范族 ③构造LR(1)分析表（及消除二义文法冲突）④识别句子

4. 给出基本块代码 ①构造DAG ②写出优化后的中间代码 ③写出DAG目标优化后的中间代码 ④根据变量活跃性和寄存器信息，写出目标代码

5.给出翻译模式和高级语言程序，翻译句子

`while a < b do if c > d then x = y * z`，会给出翻译规则。

## 2022 Spring

一、概念题 5分×5

- 1.画编译流程图
- 2.给出有穷自动机的概念，说明 NFA DFA的区别
- 3.简述推导和归约的概念
- 4.说明法制导定义的概念。S-SDD、L-SDD的概念
- 5.基本块划分方法

二、 $a((b(a|b)^*)|\emptyset)ab$  画NFA转DFA并最小化

三、说明下列文法是LL(1)的，给出语法分析表，并分析ccccc

```
S -> CC
C -> cC
C -> d
```

四、说明下列文法是LR(0)的，给出分析表并分析accd

```
S -> aA|bB
A -> cA|d
B -> cB|
```

五、说明语法制导翻译的思想 (7分)

六、代码优化列举四种并说明 (8分)

## 2022 Fall

1、考试时间：2023/5/26 14:00-16:00

2、考试科目：编译原理 (老师：LiuHong)

3、考后感悟：

本次考试题目近80%都是2023年初开学考试的题目，真的一模一样，符号都不带变的。提醒一下最好带个尺子、铅笔和橡皮，这样更方便画图和表格。

### 一、简答题 (5\*5'=25')

1.画出编译原理的程序框图。

2.什么是文法的二义性？为什么要消除二义性？如何消除二义性？

1、二义性：给定文法，若存在某个句子，有多个最左/右推导，即可以生成多棵解析树，则这个文法就是二义的。 2、通常要求程序设计语言的文法的无二义性的，否则会导致一个程序有多“正确”的解释。即使文法允许二义性，但仍需要在文法之外加以说明，来剔除不要的语法分析树。总之，必须保证文法消除了二义性使得最后的语法解析树只有一棵。 3、①改写原文法 ②引入消除二义性的规则。

3.简述推导和归约的概念。

推导：将终结符替换为它的某个产生式的体。归约：将一个与某个产生式的体相匹配的特定子串替换为该产生式的头。

4.简述递归下降语法分析技术的基本思想。

对于LL(1)文法，不必实际构建解析树，而且可以借助系统栈来实现预测分析，这就是递归下降算法。

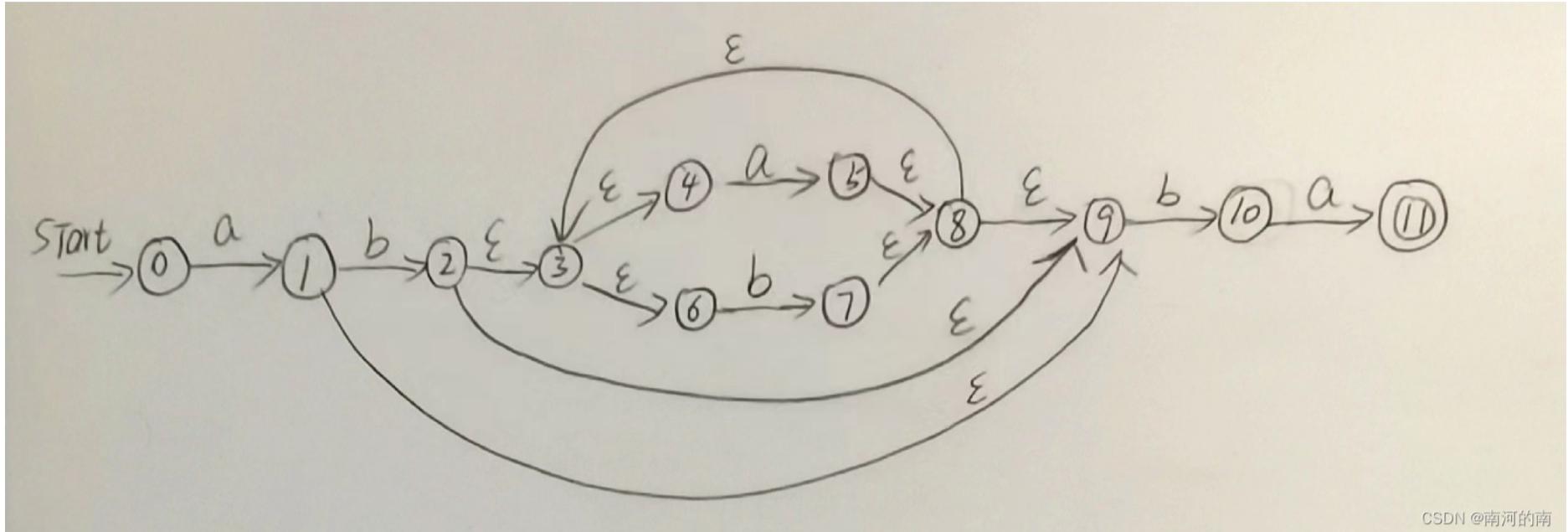
5.简述划分基本块的算法。

①确定首指令：第一个三地址指令；任意一个转移指令的目标指令；转移指令后的一个指令。②确定基本块：从一个首指令开始到下一个首指令之间的部分为一个基本块。

## 二、词法分析 (20')

根据正则式： $a((b(a|b)^*)|\$)\varepsilon ba$ , 写出NFA, 确定化, 最小化。

①NFA:



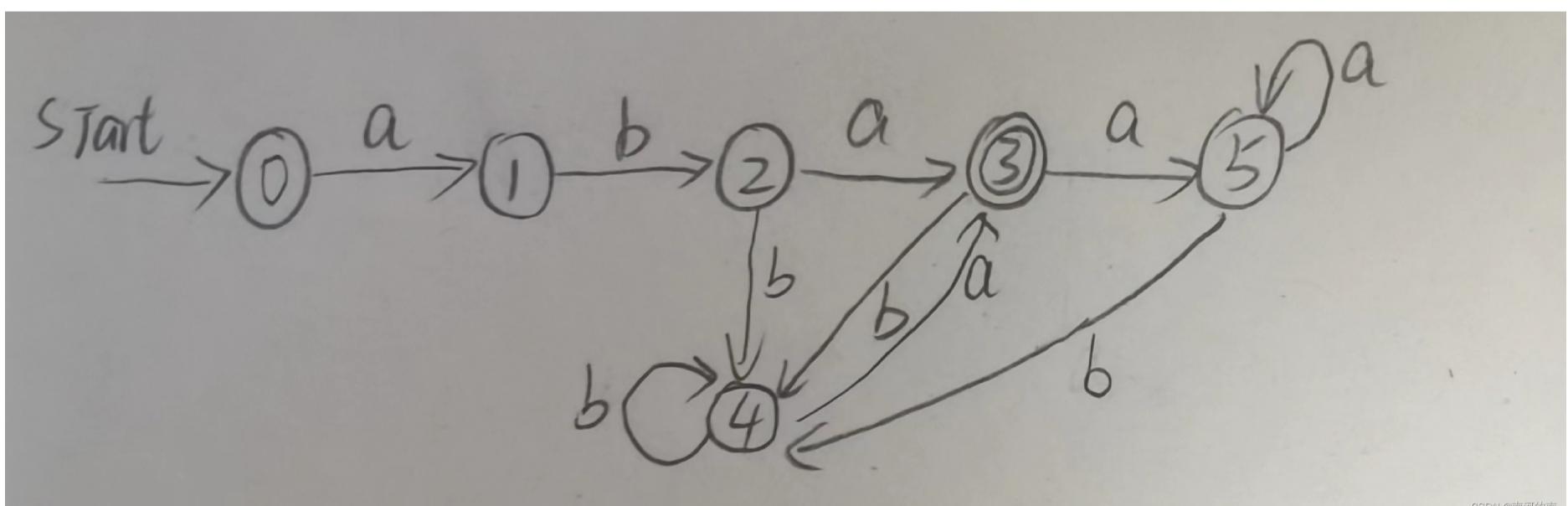
CSDN @南河的南

②确定化：

| DFA | NFA                    | a                      | b                      |
|-----|------------------------|------------------------|------------------------|
| 0   | {0}                    | {1, 9}                 | $\emptyset$            |
| 1   | {1, 9}                 | $\emptyset$            | {2, 3, 4, 6, 9, 10}    |
| 2   | {2, 3, 4, 6, 9, 10}    | {3, 4, 5, 6, 8, 9, 11} | {3, 4, 6, 7, 8, 9, 10} |
| 3   | {3, 4, 5, 6, 8, 9, 11} | {3, 4, 5, 6, 8, 9}     | {3, 4, 6, 7, 8, 9, 10} |
| 4   | {3, 4, 6, 7, 8, 9, 10} | {3, 4, 5, 6, 8, 9, 11} | {3, 4, 6, 7, 8, 9, 10} |
| 5   | {3, 4, 5, 6, 8, 9}     | {3, 4, 5, 6, 8, 9}     | {3, 4, 6, 7, 8, 9, 10} |

CSDN @南河的南

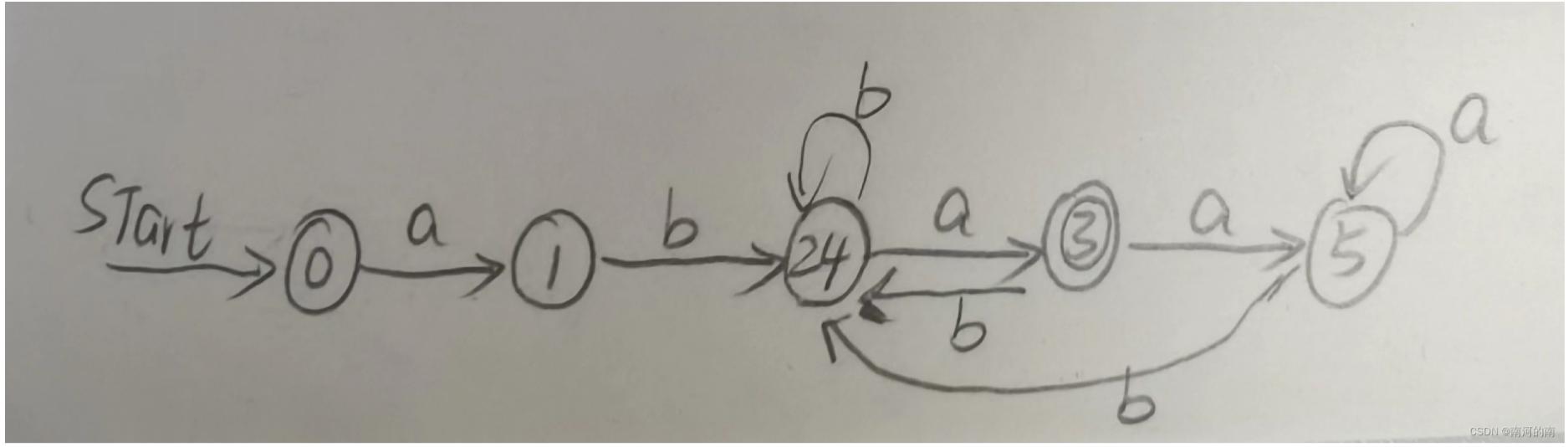
DFA:



CSDN @南河的南

③最小化：

初步划分为{0, 1, 2, 4, 5}、{3}；  
根据a将{0, 1, 2, 4, 5}划分为{0, 1, 5}、{2, 4}；  
根据a将{0, 1, 5}划分为{0}、{1}、{5}；  
{2, 4}不用划分，归为一个节点；  
最终节点划分为{0}、{1}、{2, 4}、{3}、{5}。



CSDN @南河的雨

### 三、语法分析 (20')

$G(S)$ 文法如下：

$$\begin{aligned} S &\rightarrow CC \\ C &\rightarrow cC \\ C &\rightarrow d \end{aligned}$$

3.1  $G(S)$ 对应的First和Follow是什么？证明是 $G(S)$ 是LL(1)。

```
First(S) = {c, d}
First(C) = {c, d}
Follow(S) = {$}
Follow(C) = {c, d, $}
```

证明：①  $G(S)$ 不含左递归。

② 对于 $C \rightarrow cC, C \rightarrow d$ 中， $First(cC) \cap First(d) = \emptyset$

③ 对于 $A, C$ ，它们的首终结符都不含  $\epsilon$

所以 $G(S)$ 是LL(1)

3.2写出 $G(S)$ 的预测分析表。

```
Start(S → CC) = {c, d} Start(C → CC) = {c} Start(C → d) = {d}
```

|   | c                  | d                  |
|---|--------------------|--------------------|
| S | $S \rightarrow CC$ | $S \rightarrow CC$ |
| C | $C \rightarrow cC$ | $C \rightarrow d$  |

3.3根据预测分析表，写出cdcccd的TOP-DOWN的推导过程。

|       |          |
|-------|----------|
| S\$   | cdcccd\$ |
| CC\$  | cdcccd\$ |
| cCC\$ | cdcccd\$ |
| CC\$  | dccccd\$ |
| dC\$  | dccccd\$ |
| C\$   | ccccd\$  |
| cC\$  | ccccd\$  |
| C\$   | cccd\$   |
| cC\$  | cccd\$   |
| C\$   | ccd\$    |
| cC\$  | ccd\$    |
| C\$   | cd\$     |

|      |      |
|------|------|
| cC\$ | cd\$ |
| C\$  | d\$  |
| d\$  | d\$  |
| \$   | \$   |

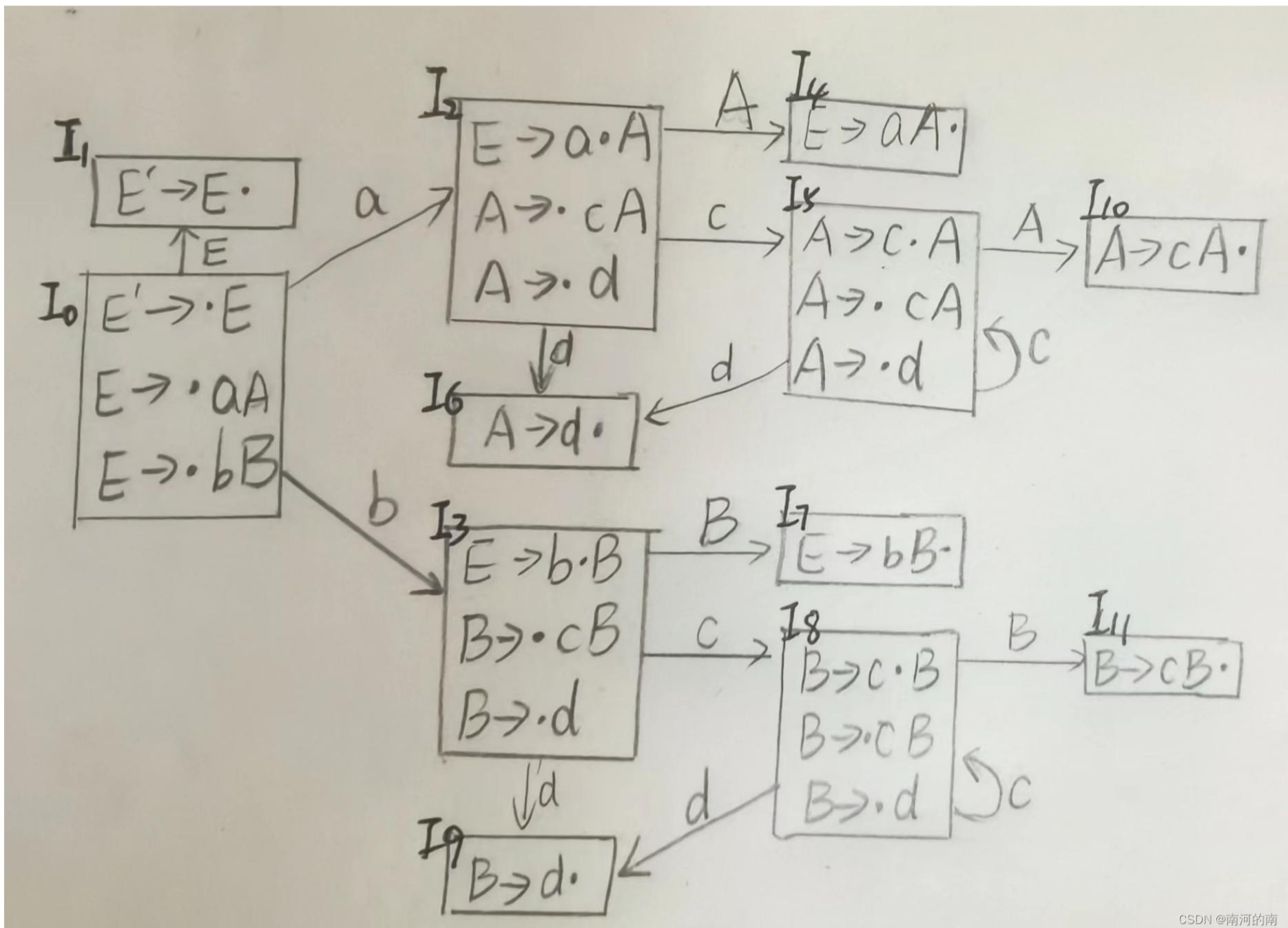
#### 四、语义分析 (20')

给出以下文法：

$$\begin{aligned} E &\rightarrow aA \mid bB \\ A &\rightarrow cA \mid d \\ B &\rightarrow cB \mid d \end{aligned}$$

4.1 证明是LR(0)。

证明：对文法进行拓广：(1)  $E' \rightarrow E$  (2)  $E \rightarrow aB$  (3)  $E \rightarrow bB$  (4)  $A \rightarrow cA$  (5)  $A \rightarrow d$  (6)  $B \rightarrow cB$  (7)  $B \rightarrow d$  画图如下：



从上图可以看出该文法没有移入-归约冲突，也没有归约-归约冲突，所以是LR(0)文法。

4.2 写出预测分析表。

| STATE | ACTION |    |    |    |     | GOTO |    |   |
|-------|--------|----|----|----|-----|------|----|---|
|       | a      | b  | c  | d  | \$  | A    | B  | E |
| 0     | S2     | S3 |    |    |     |      |    | 1 |
| 1     |        |    |    |    | acc |      |    |   |
| 2     |        |    | S5 | S6 |     | 4    |    |   |
| 3     |        |    | S8 | S9 |     |      | 7  |   |
| 4     | r2     | r2 | r2 | r2 | r2  |      |    |   |
| 5     |        |    | S5 | S6 |     | 10   |    |   |
| 6     | r5     | r5 | r5 | r5 | r5  |      |    |   |
| 7     | r3     | r3 | r3 | r3 | r3  |      |    |   |
| 8     |        |    | S8 | S9 |     |      | 11 |   |
| 9     | r7     | r7 | r7 | r7 | r7  |      |    |   |
| 10    | r4     | r4 | r4 | r4 | r4  |      |    |   |
| 11    | r6     | r6 | r6 | r6 | r6  |      |    |   |

注意归约E' —> E

## 五、语法制导翻译 (7')

写出语法制导翻译的基本思想。并说明抽象语法树在语法制导翻译中的角色。

1、基本思想：对字符串进行语法分析，构建语法分析树，然后根据需要遍历语法树并在语法书的各结点处按语义规则进行计算。这种有源程序的语法结构驱动的处理方法就是语法制导翻译。  
2、抽象语法树中，每个结点代表一个语法结构，比如对应某个运算符； 结点的每个子结点代表其子结构，比如对应运算分量，表示这些子结构按照特定的方式组成了较大的结构，可以忽略掉一些标点符号等非本质的东西。 抽象语法树是将源代码转换为目标代码的中间表示形式，可以帮助我们更好地理解源代码的结构和语义。在语法制导翻译中，我们可以通过遍历抽象语法树来执行语义动作，生成目标代码。

## 六、代码优化 (8')

说明局部优化和全局优化的不同。写出至少四个优化方法，并简述其算法。

1、局部优化是指单个基本块范围内的优化；全局优化是指面向多个基本块的优化。 2、优化方法：  
 ①删除公共子表达式：如果表达式  $x \ op \ y$  先前已被计算过，并且从先前的计算到现在， $x \ op \ y$  中变量的值没有改变。那么可以删除公共子表达式。  
 ②删除无用代码：在复制语句  $x = y$  的后面尽可能地用  $y$  替代  $x$ 。  
 ③常量合并：如果在编译时刻推导出一个表达式的值是常量，就可以 使用该常量来替代这个表达式。  
 ④代码移动：对于那些不管循环执行多少次都得到相同结果的表达式，在进入循环之前就对它们求值。  
 ⑤强度削弱：用较快的操作代替较慢的操作。

# 2023 Spring

## 一、简答题

1. 画图表示编译过程的各阶段
2. 给一个句子判断是不是二义文法
3. 给一个句子，写出短语和句柄
4. 提左公因子
5. 中缀转后缀表达式（只有加法和括号和乘法）  
给一个FA，给出对应的正则表达式

二、词法分析：给定正规式，类似作业题那个00|11闭包那个题，然后构造NFA、确定化和最小化。

三、LL(1)分析，给出文法，消除左递归、构造First、Follow集合、构造LL(1)分析表（可能涉及消除二义文法冲突）、识别句子。

四、LR分析，给出文法，构造拓广文法，构造拓广文法的LR(1)项目集规范族（这里好像设计了同心集合并，因为他给了10种状态，正常推肯定大于10种），构造LR(1)分析表，识别句子（一共规约9~10行，很快，我是强行ACC了最后）。

五、给出翻译模式和高级语言程序，翻译句子。翻译模式很长，给了一个while+if+then的句子最后好像是then a;b (a和b都是句子) 然后让填符号表，让填10行的中间代码。

六、给出基本块代码（和最后的作业题很像）（比较简单），构造DAG，写出优化后的中间代码，写出DAG目标优化后的中间代码，根据变量活跃性和寄存器信息，写出目标代码，整个题量很大，很多人都没做完，题也比较难，主要是LR1分析大部分人都画时间很多。只有看到一个题，一眼就知道咋做，然后立刻写，中间不停顿才差不多做完。

## 2023 Fall

### 一、简答题：

1. 什么是编译？典型编译系统的组成部分。
2. 简述什么是FA，并说明DFA和NFA的区别。
3. 证明  $S \rightarrow SaS \mid \epsilon$  是二义文法。
4. 已知语言  $L(G) = ab^n c^n (n \geq 0)$ ，构造文法  $G$ 。
5. 简述推导和规约的概念。

### 二、分析题

#### 1、词法分析

- (1)  $\Sigma = \{0, 1\}$ ，写出倒数第二位为1的字符串的正规式。
- (2) 做出NFA、确定化得到DFA、DFA最小化。

#### 2、给定一个文法，进行语法分析

- (1) 求 FIRST 集、FOLLOW 集，并证明是 LL(1) 文法；
- (2) 画出预测分析表；
- (3) 自顶向下分析一个给定字符串。

#### 3、给定一个文法，进行语法分析：

$$\begin{aligned} S &\rightarrow aB \\ S &\rightarrow A \\ A &\rightarrow a \\ B &\rightarrow aAb \end{aligned}$$

- (1) 证明这个文法不是 LR(0)，是 LR(1)；写出 LR(1) 项目簇、画出预测分析表；
- (2) 自底向上分析字符串  $aaab$ 。

### 三、综合题

#### 1、属性文法（7分）

- (1) 什么是综合属性、继承属性？说明终结符的属性。
- (2) 什么是依赖图？给定下列文法，画出句子  $6 * 8 + 9$  的解析树。

|                                  |                                           |
|----------------------------------|-------------------------------------------|
| (1) $L \rightarrow E\$$          | $L.val = E.val$                           |
| (2) $E \rightarrow E_1 + T$      | $E.val = E_1.val + T.val$ $E.val = T.val$ |
| (3) $E \rightarrow T$            | $T.val = T_1.val \times F.val$            |
| (4) $T \rightarrow T_1 * F$      | $T.val = F.val$                           |
| (5) $T \rightarrow F$            | $F.val = E.val$                           |
| (6) $F \rightarrow ( E )$        | $F.val = \text{digit}.lexval$             |
| (7) $F \rightarrow \text{digit}$ |                                           |

## 2、代码优化 (8分)

请列举出四个代码优化方法，并阐述其思想。

(整体来说比较简单，题型比较基础，后面的目标代码生成和中间代码生成有的班好像没有讲，也没考。)