编译原理2015卷——By Gerchart

填空

1. **一个RG（正则文法）（ r ）和一个DFA（确定性有限自动机）（ M ）等价的含义是（它们识别的语言相同，即 (L(r) = L(M) \) ）。**

- **解析**：正则文法和确定性有限自动机是两种不同的形式化语言描述方法。它们等价的含义是它们所描述的语言是相同的，即它们能够识别相同的字符串集合。

2. **DFA有（状态集）、（字母表）、（转移函数）、（初始状态）、（接受状态集）五个部分。**

- **解析**：DFA（确定性有限自动机）由五个基本部分组成：
  - 状态集（States）：DFA的所有可能状态的集合。
  - 字母表（Alphabet）：DFA可以处理的输入符号的集合。
  - 转移函数（Transition Function）：定义了从一个状态在给定输入符号下转移到另一个状态的规则。
  - 初始状态（Start State）：DFA开始处理输入时的状态。
  - 接受状态集（Accept States）：DFA最终接受输入时的状态集合。

3. **编译是把源语言程序（翻译成目标语言程序的过程）。**

- **解析**：编译是将用一种编程语言（源语言）编写的程序转换成另一种语言（目标语言）表示的程序的过程，通常是将高级语言转换为机器语言或中间代码。

4. **全局优化技术包括（常量传播）、（死代码消除）。**

- **解析**：
  - 常量传播（Constant Propagation）：通过分析程序代码，将程序中的常量值传播到所有使用这些常量的地方，从而优化代码。
  - 死代码消除（Dead Code Elimination）：移除那些永远不会被执行的代码段，从而简化程序并提高效率。

5. `if(a>b or b<d) then X: …` **的逆波兰式是（** `a b > b d < or X` **）。**

- **解析**：逆波兰式（也称为后缀表达式）是一种不需要括号的表达式表示法。对于给定的条件表达式 `if(a>b or b<d) then X`，其逆波兰式为 `a b > b d < or X`。

6. **编译的前端是（词法分析）、（语法分析）、（语义分析）。**

- **解析**：编译器的前端主要包括以下几个阶段：
  - 词法分析（Lexical Analysis）：将源代码转换为一系列记号（tokens）。
  - 语法分析（Syntax Analysis）：根据语言的语法规则，将记号序列转换为语法树。
  - 语义分析（Semantic Analysis）：检查语法树是否符合语言的语义规则，并进行类型检查等。

7. **文法的句子是（符合该文法规则的一系列符号序列）。**

- **解析**：一个文法的句子是指由文法生成的符合文法规则的符号序列。

8. **一般高级语言的语法可以用（上下文无关文法）表示。**

- **解析**：上下文无关文法（Context-Free Grammar, CFG）是一种形式化的语法规则，广泛用于描述编程语言的语法结构。

9. **属性文法的属性有（综合属性）和（继承属性）。**

- **解析**：
  - 综合属性（Synthesized Attribute）：从子节点传递到父节点的属性。
  - 继承属性（Inherited Attribute）：从父节点或兄弟节点传递到子节点的属性。

10. **对符号表项的操作有（插入）、（查询）。**

- **解析**：在编译过程中，符号表用于存储标识符及其相关信息。对符号表的基本操作包括：
  - 插入（Insertion）：将新的标识符及其信息加入符号表。
  - 查询（Lookup）：在符号表中查找标识符的信息。

简答

1

> 1. 词法分析的任务是什么？一般高级语言单词的分类有？

词法分析是编译器前端的一个重要阶段，其主要任务是将源代码转换为一系列记号（tokens），这些记号是语法分析的输入。详细解析如下：

词法分析的任务

1. **扫描源代码**：逐字符读取源代码。
2. **识别记号（Tokens）**：根据语言的词法规则，将字符序列分割成有意义的记号。每个记号通常包含一个记号类型（例如关键字、标识符、操作符）和其在源代码中的实际字符串值。
3. **去除空白和注释**：在大多数情况下，空白字符（如空格、制表符、换行符）和注释会被忽略，因为它们对语法分析没有影响。
4. **记号化错误处理**：识别并处理词法错误，如非法字符或不匹配的符号。
5. **提供输入给语法分析器**：生成的记号序列将作为语法分析器的输入，用于进一步的语法分析。

一般高级语言单词的分类

在高级编程语言中，词法分析器通常将源代码中的字符序列分类为以下几种主要类型的单词（记号）：

1. **关键字（Keywords）**：保留用于特定语言结构的单词。例如，在C语言中，`if`、`else`、`while`等都是关键字。
2. **标识符（Identifiers）**：用户定义的名称，用于变量、函数、类等。例如，`varName`、`functionName`。
3. **常量（Constants）**：直接表示值的单词。包括：
   - 整数常量（Integer Constants）：如`123`
   - 浮点常量（Floating-point Constants）：如`3.14`
   - 字符常量（Character Constants）：如`'a'`
   - 字符串常量（String Constants）：如`"hello"`
4. **操作符（Operators）**：表示运算的符号。如`+`、`-`、`*`、`/`、`==`、`&&`等。

5. **分隔符（Separators）**：用于分割语句或表达式的符号。如`(`、`)`、`{`、`}`、`;`、`,`等。
6. **注释（Comments）**：用于在代码中插入说明性文本，通常会被词法分析器忽略。例如，`// 单行注释` 或 `/* 多行注释 */`。

词法分析的示例

假设我们有以下一段简单的C代码：

```c
int main() {
    int a = 5;
    if (a > 3) {
        a = a + 1;
    }
    return 0;
}
```

词法分析器将这段代码转换为一系列记号，如下所示：

- `int` (关键字)
- `main` (标识符)
- `(` (分隔符)
- `)` (分隔符)
- `{` (分隔符)
- `int` (关键字)
- `a` (标识符)
- `=` (操作符)
- `5` (整数常量)
- `;` (分隔符)
- `if` (关键字)
- `(` (分隔符)
- `a` (标识符)
- `>` (操作符)
- `3` (整数常量)
- `)` (分隔符)
- `{` (分隔符)
- `a` (标识符)
- `=` (操作符)
- `a` (标识符)
- `+` (操作符)
- `1` (整数常量)
- `;` (分隔符)
- `}` (分隔符)

- `return` (关键字)
- `0` (整数常量)
- `;` (分隔符)
- `}` (分隔符)

通过词法分析，源代码被转换成一系列有意义的记号，这些记号将作为语法分析阶段的输入。

2

```
2.  消除左递归，提取公因子，并判断改写后的文法是不是LL(1)。
A->Ba|b
B->AC|ϵ
C->c
```

为了消除左递归和提取左公因子，并判断改写后的文法是否是LL(1)文法，我们需要对给定的文法进行系统的分析和改写。

原始文法

```
A -> Ba | b
B -> AC | ϵ
C -> c
```

步骤 1：消除左递归

在给定的文法中，没有直接左递归（即形如 $A \rightarrow A\alpha$ 这种形式），所以我们不需要进行左递归的消除。

步骤 2：提取左公因子

我们需要检查每个产生式，看看是否有可以提取的左公因子。

对于 A 的产生式：

```
A -> Ba | b
```

这里没有公共前缀，所以不需要提取左公因子。

对于 B 的产生式：

```
B -> AC | ϵ
```

这里也没有公共前缀，所以不需要提取左公因子。

改写后的文法

由于不需要进行左递归消除和左公因子提取，改写后的文法还是原来的形式：

```
A -> Ba | b
B -> AC | ϵ
C -> c
```

步骤 3：判断是否是LL(1)文法

为了判断文法是否是LL(1)文法，我们需要检查两个条件：

1. **每个非终结符的FIRST集之间互不相交。**
2. **对于每个非终结符，如果它的产生式可以推导出空串，那么它的FOLLOW集与其他产生式的FIRST集互不相交。**

计算FIRST集和FOLLOW集

FIRST集：

- $FIRST(A)$
  - $A \to Ba$，所以 $FIRST(A)$ 包含 $FIRST(Ba)$
  - $A \to b$，所以 $FIRST(A)$ 包含 b
  - $FIRST(Ba) = FIRST(B) = \{b, c, \varepsilon\}$
  - 所以 $FIRST(A) = \{b, c\}$
- $FIRST(B)$
  - $B \to AC$，所以 $FIRST(B)$ 包含 $FIRST(AC) = FIRST(A)$
  - $B \to \varepsilon$，所以 $FIRST(B) = \{b, c, \varepsilon\}$
- $FIRST(C)$
  - $C \to c$，所以 $FIRST(C) = \{c\}$

FOLLOW集：

- $FOLLOW(A)$
  - A 是开始符号，$FOLLOW(A)$ 包含 `$`
  - $B \to AC$，所以 $FOLLOW(A)$ 包含 $FIRST(C) = \{c\}$
- $FOLLOW(B)$
  - $A \to Ba$，所以 $FOLLOW(B)$ 包含 a
  - $A \to b$，所以 $FOLLOW(B)$ 包含 `$`
- $FOLLOW(C)$
  - $B \to AC$，所以 $FOLLOW(C)$ 包含 $FOLLOW(B)$
  - 所以 $FOLLOW(C) = \{a, \}\$$

检查LL(1)条件

1. **检查FIRST集之间的交集：**
   - $FIRST(A) = \{b, c\}$
   - $FIRST(B) = \{b, c, \varepsilon\}$
   - $FIRST(C) = \{c\}$

   $FIRST(A)$ 和 $FIRST(B)$ 有交集 $\{b, c\}$，这表明文法不是LL(1)文法。
2. **检查FOLLOW集与FIRST集的交集：**
   - $FOLLOW(A) = \{c, \}\$$
   - $FOLLOW(B) = \{a, \}\$$

- $FOLLOW(C) = \{a, \}\$$

由于第一步已经确定文法不是LL(1)文法，这一步就不需要进一步检查了。

结论

经过上述分析和计算，可以得出结论：改写后的文法不是LL(1)文法。

3.

```
    （此题有误，勿做）文法为：
E->ET+|T
T->TF*|F
F->FP↑|P
P->E|i
串为FET+T*i↑，画出语法树，判断是否为右句型，若是，则指出句柄。
```

我们首先来画出给定文法的语法树，然后判断该串是否为右句型，并找出句柄。

给定文法

```
E -> ET+ | T
T -> TF* | F
F -> FP↑ | P
P -> E | i
```

给定串

```
FET+T*i↑
```

语法树构建

首先，我们需要解析给定的串并尝试构建语法树。由于给定的串可能不符合原文法，我们需要进行一些推导来检查它是否有效。

解析串：`FET+T*i↑`

1. **构建语法树**：
   - 最左边的符号是 `F`，根据文法规则 `F -> FP↑ | P`，我们可以继续推导。
   - 但我们发现 `F` 后面是 `E`，这不符合文法规则中的任何一个产生式。

检查是否符合文法

根据文法规则，我们无法直接将 `FET+T*i↑` 解析成一个有效的语法树，因为文法规则不支持这种形式的串。

结论

由于给定的串 `FET+T*i↑` 无法根据文法规则构建出有效的语法树，因此我们无法判断它是否为右句型，也无法指出句柄。

正确的步骤应该是

1. **验证串是否符合文法**：首先确保串符合给定的文法规则。
2. **构建语法树**：如果串符合文法，构建出对应的语法树。
3. **判断是否为右句型**：检查语法树的最右推导。
4. **指出句柄**：如果是右句型，找出最后一步推导的句柄。

由于给定的串不符合文法规则，我们无法进行这些步骤。请检查文法和串的正确性，或者提供一个符合文法的串以便进一步分析。

4

```
4.  写一个3型文法，表示{anbm,n,m>=1}。
```

3型文法（Type 3 Grammar），也称为正则文法（Regular Grammar），是Chomsky层次中的最简单的文法类型。它的产生式规则形式为：$A \to aB$ 或 $A \to a$，其中 A 和 B 是非终结符，a 是终结符。

要表示语言 $\{a^n b^m \mid n, m \geq 1\}$，我们需要一个能够生成至少一个 a 后跟至少一个 b 的文法。

3型文法表示 $\{a^n b^m \mid n, m \geq 1\}$

我们可以设计一个正则文法来生成这个语言。文法的规则如下：

- S 是开始符号。
- $S \to aA$：生成至少一个 a 并转到状态 A。
- $A \to aA$：生成更多的 a，保持在状态 A。
- $A \to bB$：生成一个 b 并转到状态 B。
- $B \to bB$：生成更多的 b，保持在状态 B。
- $B \to \epsilon$：结束生成。

文法规则

```
S -> aA
A -> aA | bB
B -> bB | ϵ
```

解释

1. **开始符号 S**：
   - $S \to aA$：生成一个 a 并转到状态 A。
2. **状态 A**：
   - $A \to aA$：生成更多的 a 并保持在状态 A。
   - $A \to bB$：生成一个 b 并转到状态 B。
3. **状态 B**：
   - $B \to bB$：生成更多的 b 并保持在状态 B。
   - $B \to \epsilon$：结束生成。

示例推导

让我们推导一个字符串 a^2b^3（即 $aabbb$）：

1. $S \to aA$
2. $aA \to aaA$
3. $aaA \to aabB$
4. $aabB \to aabbB$
5. $aabbB \to aabbbB$
6. $aabbbB \to aabbb$

这个推导生成了字符串 $aabbb$，符合语言 $\{a^n b^m \mid n, m \geq 1\}$ 的要求。

总结

这个3型文法可以生成所有形如 $a^n b^m$ 的字符串，其中 n 和 m 均为至少1的正整数，符合语言 $\{a^n b^m \mid n, m \geq 1\}$ 的要求。

5.

```
5.  判断正规式(a*|b*)*与((ε|a)b*)*是否等价。
```

为了判断两个正规式是否等价，我们需要检验它们是否能够生成相同的语言。我们来逐步分析这两个正规式：

1. 正规式 ((a|*b*)*):
   - (a*) 表示零个或多个 'a'，即可以生成空串 ε、'a'、'aa'、'aaa' 等。
   - (b*) 表示零个或多个 'b'，即可以生成空串 ε、'b'、'bb'、'bbb' 等。
   - (a|*b*) 表示 (a*) 或 (b*)，即可以生成任意数量的 'a' 或任意数量的 'b'，但不混合。
   - ((a|*b*)*) 表示零个或多个 (a|*b*)，即可以生成任意数量的 'a' 串和 'b' 串的任意组合。例如，它可以生成 ε, 'a', 'b', 'aa', 'bb', 'aaa', 'bbb', 'aaabbb', 'bbbaaa' 等。
2. 正规式 (((ε|a)b*)):
   - (ε|a) 表示 ε 或 'a'，即可以生成空串 ε 或单个字符 'a'。
   - (b*) 表示零个或多个 'b'，即可以生成空串 ε、'b'、'bb'、'bbb' 等。
   - ((ε|a)b*) 表示 ε 或 'a' 后跟随零个或多个 'b'，即可以生成 ε, 'a', 'ab', 'abb', 'abbb' 等。
   - (((ε|a)b*)) 表示零个或多个 ((ε|a)b*)，即可以生成任意数量的 ε 或 'a' 后跟任意数量的 'b' 的组合。例如，它可以生成 ε, 'a', 'b', 'ab', 'abb', 'aab', 'aabb', 'aabbb' 等。

通过对比这两个正规式生成的语言，我们可以看到：

- 正规式 ((a|*b*)*) 可以生成任意数量的 'a' 串和 'b' 串的任意组合，但不能混合 'a' 和 'b'。
- 正规式 (((ε|a)b*)) 可以生成任意数量的 'a' 串和 'b' 串的任意组合，并且可以混合 'a' 和 'b'。

因此，两个正规式生成的语言不相同，故它们不等价。

6

```
6.  窥孔优化:
x:=f*2.0
u:=f/2.0
```

窥孔优化（Peephole Optimization）是一种局部优化技术，通过在编译器生成的中间代码或目标代码中查找和替换小的代码片段，使代码更加高效。针对你的例子，我们可以对以下两个赋值语句进行优化：

1. $x := f \times 2.0$
2. $u := f/2.0$

优化分析

$$x := f \times 2.0$$

乘以2.0可以通过左移一位来实现，左移操作通常比乘法操作更加高效。在大多数机器上，左移一位相当于乘以2。

假设 f 是一个浮点数，这种优化可能不适用，因为浮点数的位移操作不一定会产生正确的结果。如果 f 是一个整数类型（如int），这种优化是可行的。

$$u := f/2.0$$

除以2.0可以通过右移一位来实现，右移操作通常比除法操作更加高效。

同样地，如果 f 是一个浮点数，这种优化可能不适用。如果 f 是一个整数类型，这种优化是可行的。

具体优化

假设 f 是整数类型，我们可以进行如下优化：

1. $x := f \times 2.0$ 可以优化为 $x := f \ll 1$。
2. $u := f/2.0$ 可以优化为 $u := f \gg 1$。

优化后的代码

```
x := f << 1
u := f >> 1
```

注意事项

- 如果 f 是浮点数类型，上述位移优化不适用，因为浮点数的位移操作不会产生正确的结果。
- 优化应在确保程序语义不变的前提下进行。

总结

窥孔优化是一种有效的局部优化技术，通过替换小的代码片段来提高代码执行效率。在上述例子中，我们通过位移操作替代了乘法和除法操作，假设 f 是整数类型，这种优化可以提高执行效率。如果 f 是浮点数类型，则需要谨慎处理。

7

```
7.  E->TR
R->+T {print(‘+’)} R1
R->ϵ
T->num {print(num.val)}
改写以左翻译模式，消除内嵌语义动作。
```

为了将带有内嵌语义动作的语法规则改写为左翻译模式，我们需要将语义动作移到产生式的左侧。这样可以消除内嵌语义动作，同时使语法规则更易于实现。

给定的语法规则如下：

```
E -> TR
R -> +T {print(‘+’)} R1
R -> ϵ
T -> num {print(num.val)}
```

我们首先将这些规则改写为左翻译模式。左翻译模式意味着我们在解析过程中立即执行语义动作。

改写后的语法规则

为了改写这些规则，我们需要引入新的非终结符，并将语义动作移到产生式的左侧：

1. $E \rightarrow TR$
2. $R \rightarrow +TR1$
3. $R \rightarrow \epsilon$
4. $T \rightarrow num$

引入新的非终结符和语义动作

我们引入新的非终结符 R' 来表示在 R 产生式中执行的语义动作：

1. $E \rightarrow TR$
2. $R \rightarrow +TR1\_print$
3. $R \rightarrow \epsilon$
4. $T \rightarrow num$
5. $R1\_print \rightarrow \epsilon\ print('+')$
6. $T\_print \rightarrow num\ print(num.val)$

最终左翻译模式的语法规则

将所有语义动作移到产生式的左侧后，我们得到以下语法规则：

```
E -> TR
R -> +T R1_print
R -> ϵ
T -> num_print
R1_print -> ϵ {print('+')}
num_print -> num {print(num.val)}
```

解释

- $E \rightarrow TR$: 这个规则没有改变，因为它不包含任何语义动作。
- $R \rightarrow +TR1\_print$: 这个规则将原来的内嵌语义动作 {print('+')} 移到了新引入的非终结符 $R1\_print$ 中。
- $R \rightarrow \epsilon$: 这个规则没有改变，因为它不包含任何语义动作。
- $T \rightarrow num\_print$: 这个规则将原来的内嵌语义动作 {print(num.val)} 移到了新引入的非终结符 num_print 中。
- $R1\_print \rightarrow \epsilon\ print('+')$: 新引入的规则，用于执行原来在 $R \rightarrow +T$ 中的语义动作 {print('+')}.
- $num\_print \rightarrow num\ print(num.val)$: 新引入的规则，用于执行原来在 $T \rightarrow num$ 中的语义动作 {print(num.val)}.

通过这种改写，我们将语义动作移到了产生式的左侧，从而消除了内嵌语义动作，实现了左翻译模式。

8

```
8.  说明SLR(1)文法与LALR(1)文法的关系并说明理由。
```

SLR(1)（Simple LR, 简单LR）文法和LALR(1)（Look-Ahead LR, 向前看LR）文法都是用于构造自底向上分析器的语法类型。尽管它们都是LR(1)文法的简化形式，但它们在构造分析表的过程中有一些重要的区别和联系。

关系

1. **包含关系**:
   - SLR(1) 文法是 LALR(1) 文法的一个子集。换句话说，所有的 SLR(1) 文法都是 LALR(1) 文法，但并不是所有的 LALR(1) 文法都是 SLR(1) 文法。
   - 具体来说，如果一个文法是 SLR(1) 文法，那么它一定也是 LALR(1) 文法。然而，有些文法是 LALR(1) 文法，但不是 SLR(1) 文法。
2. **构造方法**:
   - SLR(1) 分析表的构造基于 LR(0) 项目集，并使用 Follow 集来解决移入-归约冲突。
   - LALR(1) 分析表的构造基于 LR(1) 项目集，但在合并具有相同核心的项目集时只考虑 Look-Ahead 符号。这使得 LALR(1) 分析表比 SLR(1) 分析表更具区分力，能处理更多的语法。
3. **冲突处理**:
   - 在 SLR(1) 分析表中，移入-归约冲突和归约-归约冲突是通过检查 Follow 集来解决的。如果某个项目集的某个项目的 Look-Ahead 符号在 Follow 集中出现，则会产生冲突。
   - 在 LALR(1) 分析表中，移入-归约冲突和归约-归约冲突是通过检查具体的 Look-Ahead 符号来解决的。由于 LALR(1) 分析表在合并项目集时保留了 Look-Ahead 符号的信息，因此它可以在更多情况下避免冲突。

理由

1. **SLR(1) 的局限性**:
   - SLR(1) 分析器在处理一些复杂的语法时，可能会因为 Follow 集的过于宽泛而产生不必要的冲突。
   - 例如，对于某些文法，Follow 集可能包含了过多的符号，导致在构造 SLR(1) 分析表时出现移入-归约冲突或归约-归约冲突。

2. **LALR(1) 的改进**:
    - LALR(1) 分析器通过合并具有相同核心的 LR(1) 项目集，在保留 Look-Ahead 符号的前提下，减少了项目集的数量。
    - 这种方法使得 LALR(1) 分析器能够在不增加分析表大小的情况下，处理更多的语法，解决更多的冲突。
    - 由于 LALR(1) 分析器的分析表大小与 SLR(1) 分析器相同，但具有更强的区分力，因此在实际应用中更为常见。

总结

- **SLR(1) 文法** 是 LALR(1) 文法的一个子集。所有的 SLR(1) 文法都是 LALR(1) 文法，但有些 LALR(1) 文法不是 SLR(1) 文法。
- SLR(1) 分析器使用 LR(0) 项目集和 Follow 集来构造分析表，而 LALR(1) 分析器使用 LR(1) 项目集并合并具有相同核心的项目集来构造分析表。
- LALR(1) 分析器比 SLR(1) 分析器能够处理更多的语法，具有更强的区分力，在实际应用中更为常见。

解答

1

```
1.  给一个文法:
S->bAS|aBS|ϵ
A->aBA|b
B->bAB|a
计算FIRST集和FOLLOW集，并判断是否为LL(1)文法，若是，写出预测分析表。
```

要计算文法的 FIRST 集和 FOLLOW 集，并判断其是否为 LL(1) 文法，我们需要按以下步骤进行。

1. 计算 FIRST 集

规则

- 如果 X 是终结符，那么 $FIRST(X) = \{X\}$。
- 如果 $X \to \epsilon$ 是一个产生式，那么 $\epsilon \in FIRST(X)$。
- 如果 X 是非终结符，且 $X \to Y_1 Y_2 \ldots Y_k$ 是一个产生式，那么将以下符号加入 $FIRST(X)$:
    - 如果 Y_1 是终结符，则加入 Y_1。
    - 如果 Y_1 是非终结符，则加入 $FIRST(Y_1) \setminus \{\epsilon\}$。
    - 如果 Y_1 能够推导出 ϵ，则继续考虑 Y_2，依此类推。

计算步骤

1. $FIRST(S)$:
    - $S \to bAS$: $FIRST(bAS) = \{b\}$
    - $S \to aBS$: $FIRST(aBS) = \{a\}$
    - $S \to \epsilon$: $FIRST(\epsilon) = \{\epsilon\}$
    - **结果**: $FIRST(S) = \{b, a, \epsilon\}$

2. $FIRST(A)$:
   - $A \to aBA$: $FIRST(aBA) = \{a\}$
   - $A \to b$: $FIRST(b) = \{b\}$
   - **结果**: $FIRST(A) = \{a, b\}$
3. $FIRST(B)$:
   - $B \to bAB$: $FIRST(bAB) = \{b\}$
   - $B \to a$: $FIRST(a) = \{a\}$
   - **结果**: $FIRST(B) = \{b, a\}$

2. 计算 FOLLOW 集

规则

- $FOLLOW(S) = \{\$\}$（假设 $ 是输入结束符）。
- 对于每个产生式 $A \to \alpha B \beta$，将 $FIRST(\beta) \setminus \{\epsilon\}$ 加入 $FOLLOW(B)$。
- 对于每个产生式 $A \to \alpha B$ 或 $A \to \alpha B \beta$ 且 $\beta \Rightarrow \epsilon$，将 $FOLLOW(A)$ 加入 $FOLLOW(B)$。

计算步骤

1. $FOLLOW(S)$:
   - 初始: $FOLLOW(S) = \{\$\}$
   - 对于 $S \to bAS$: $FOLLOW(A) \supset FOLLOW(S) = \{\$\}$
   - 对于 $S \to aBS$: $FOLLOW(B) \supset FOLLOW(S) = \{\$\}$
   - **结果**: $FOLLOW(S) = \{\$\}$
2. $FOLLOW(A)$:
   - 初始: $FOLLOW(A) = \{\}$
   - 对于 $A \to aBA$: $FOLLOW(B) \supset FIRST(A) \setminus \{\epsilon\} = \{a, b\}$
   - 对于 $S \to bAS$: $FOLLOW(A) \supset FOLLOW(S) = \{\$\}$
   - **结果**: $FOLLOW(A) = \{\$, a, b\}$
3. $FOLLOW(B)$:
   - 初始: $FOLLOW(B) = \{\}$
   - 对于 $B \to bAB$: $FOLLOW(A) \supset FOLLOW(B)$
   - 对于 $B \to a$: $FOLLOW(B) \supset FIRST(B) \setminus \{\epsilon\} = \{a, b\}$
   - 对于 $A \to aBA$: $FOLLOW(A) \supset FIRST(A) \setminus \{\epsilon\} = \{a, b\}$
   - 对于 $S \to aBS$: $FOLLOW(B) \supset FOLLOW(S) = \{\$\}$
   - **结果**: $FOLLOW(B) = \{\$, a, b\}$

3. 判断是否为 LL(1) 文法

一个文法是 LL(1) 文法当且仅当对于每个非终结符 A，A 的所有产生式右部的 FIRST 集是两两不相交的，且如果某个产生式的右部能推导出 ϵ，那么它的 FOLLOW 集与其他产生式的 FIRST 集也是不相交的。

产生式分析

1. $S \to bAS$
2. $S \to aBS$
3. $S \to \epsilon$

- $FIRST(bAS) = \{b\}$
- $FIRST(aBS) = \{a\}$
- $FIRST(\epsilon) = \{\epsilon\}$

$FOLLOW(S) = \{\$\}$

- $\{b\} \cap \{a\} = \emptyset$
- $\{b, a\} \cap \{\$\} = \emptyset$

由于 FIRST 集和 FOLLOW 集的交集为空，因此 S 的产生式满足 LL(1) 条件。

对于 A 和 B

类似地：

- $A \to aBA$
- $A \to b$
- $FIRST(aBA) = \{a\}$
- $FIRST(b) = \{b\}$

$\{a\} \cap \{b\} = \emptyset$

- $B \to bAB$
- $B \to a$
- $FIRST(bAB) = \{b\}$
- $FIRST(a) = \{a\}$

$\{b\} \cap \{a\} = \emptyset$

所有产生式的 FIRST 集都是两两不相交的，因此该文法是 LL(1) 文法。

4. 预测分析表

非终结符	输入符号	产生式
S	a	S -> aBS
S	b	S -> bAS
S	$	S -> ε
A	a	A -> aBA
A	b	A -> b
B	a	B -> a
B	b	B -> bAB

总结

该文法是 LL(1) 文法，上述预测分析表列出了各个非终结符在不同输入符号下的产生式。

2

```
2.  写一个语言为以a开头，bab结尾的正规式Σ={a,b}，画出相应的NFA、DFA并化简。
```

我们需要构造一个正则表达式，描述以 `a` 开头并以 `bab` 结尾的字符串，其中字符集是 $\Sigma = \{a, b\}$。

正则表达式

正则表达式可以表示为：
[R = a(a|b)^*bab]

构造 NFA

我们首先构造一个非确定有限自动机（NFA），然后再将其转换为确定有限自动机（DFA）。

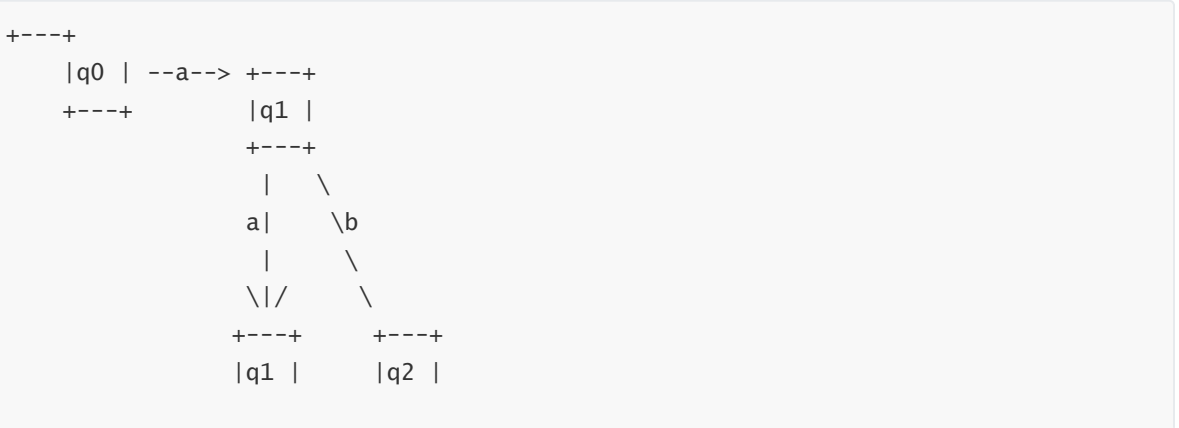
NFA 状态图

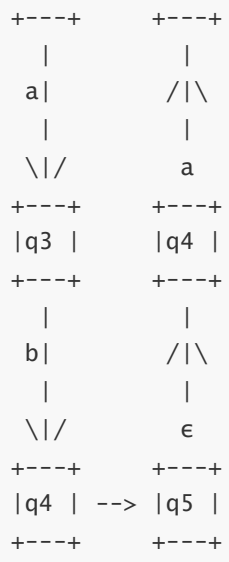
1. 初态 q_0，接受第一个字符 `a`。
2. 状态 q_1，接受任意字符 `a` 或 `b`。
3. 状态 q_2，接受字符 `b`。
4. 状态 q_3，接受字符 `a`。
5. 状态 q_4，接受字符 `b`。
6. 终态 q_5。

状态转换如下：

- $q_0 \xrightarrow{a} q_1$
- $q_1 \xrightarrow{a,b} q_1$
- $q_1 \xrightarrow{b} q_2$
- $q_2 \xrightarrow{a} q_3$
- $q_3 \xrightarrow{b} q_4$
- $q_4 \xrightarrow{\epsilon} q_5$ （q_4 是终态）

NFA 状态图如下：





转换为 DFA

我们使用子集构造法将 NFA 转换为 DFA。

1. 初始状态为 $\{q_0\}$。
2. 处理输入符号 `a`：$\{q_0\} \xrightarrow{a} \{q_1\}$。
3. 处理输入符号 `a` 或 `b`：$\{q_1\} \xrightarrow{a,b} \{q_1, q_2\}$。
4. 处理输入符号 `b`：$\{q_1, q_2\} \xrightarrow{b} \{q_1, q_3\}$。
5. 处理输入符号 `a`：$\{q_1, q_3\} \xrightarrow{a} \{q_1, q_2, q_4\}$。
6. 处理输入符号 `b`：$\{q_1, q_2, q_4\} \xrightarrow{b} \{q_1, q_3, q_5\}$。

DFA 状态转换表

状态	输入符号	转移到的状态
{q0}{q0}	a	{q1}{q1}
{q1}{q1}	a	{q1,q2}{q1,q2}
{q1}{q1}	b	{q1,q2}{q1,q2}
{q1,q2}{q1,q2}	a	{q1,q2}{q1,q2}
{q1,q2}{q1,q2}	b	{q1,q3}{q1,q3}
{q1,q3}{q1,q3}	a	{q1,q2,q4}{q1,q2,q4}
{q1,q3}{q1,q3}	b	{q1,q2}{q1,q2}
{q1,q2,q4}{q1,q2,q4}	a	{q1,q2}{q1,q2}
{q1,q2,q4}{q1,q2,q4}	b	{q1,q3,q5}{q1,q3,q5}

DFA 状态图如下：

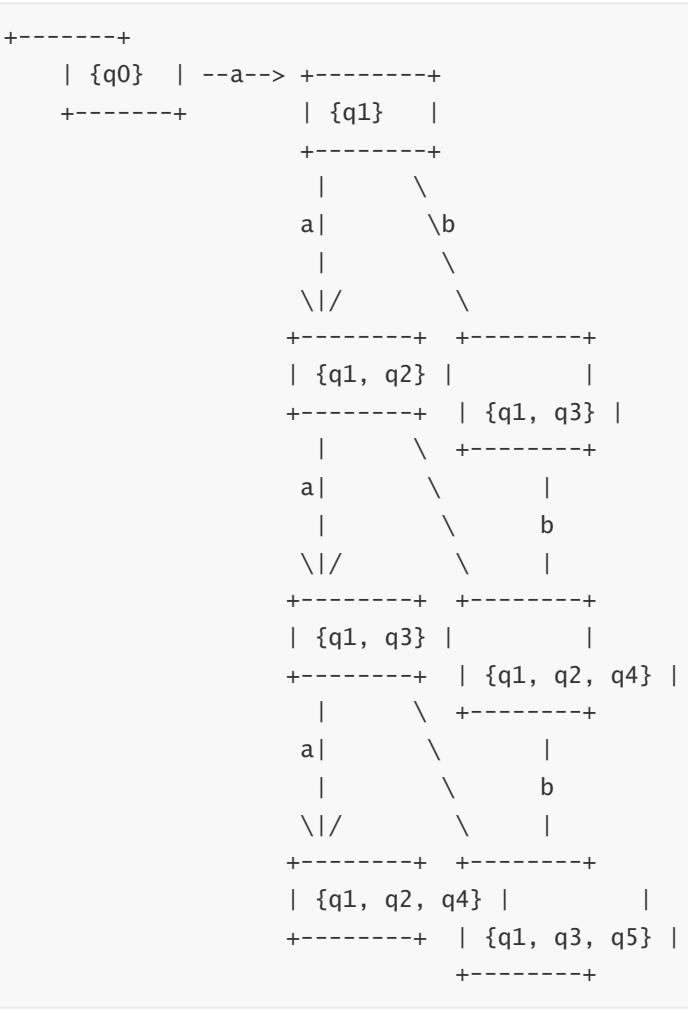
```
{q0} --a--> {q1}
{q1} --a--> {q1, q2}
{q1} --b--> {q1, q2}
{q1, q2} --a--> {q1, q2}
{q1, q2} --b--> {q1, q3}
{q1, q3} --a--> {q1, q2, q4}
{q1, q3} --b--> {q1, q2}
{q1, q2, q4} --a--> {q1, q2}
{q1, q2, q4} --b--> {q1, q3, q5}
```

化简 DFA

化简 DFA 需要合并等价状态。通过状态最小化算法，我们可以得到简化的 DFA。

1. 初始状态 $\{q_0\}$。
2. 状态 $\{q_1\}$，处理 `a` 或 `b` 后进入 $\{q_1, q_2\}$。
3. 状态 $\{q_1, q_2\}$，处理 `a` 或 `b` 后进入 $\{q_1, q_3\}$。
4. 状态 $\{q_1, q_3\}$，处理 `a` 或 `b` 后进入 $\{q_1, q_2, q_4\}$。
5. 终态 $\{q_1, q_3, q_5\}$。

最终简化的 DFA 状态图如下：



总结

通过上述步骤，我们从正则表达式 $R = a(a|b)^*bab$ 开始，构造了对应的 NFA，并使用子集构造法将其转换为 DFA，最终化简得到最小化的 DFA。

3

```
3． 给定文法：
S->AS|b
A->SA|a
写出LR(1)项目集规范族，并判断是否是LR(1)文法。
```

要判断给定的文法是否是LR(1)文法，我们首先需要构建LR(1)项目集规范族。文法如下：

$$S \to AS \mid b$$
$$A \to SA \mid a$$

步骤1：构建项目集规范族

初始项目集

项目集的初始状态（I_0）包含项目 $S' \to \cdot S, \$$，其中 S' 是扩展文法的开始符号，$ 是输入结束符号。我们还需要闭包操作来包含所有可能的项目。

$$I_0 = \text{CLOSURE}(\{S' \to \cdot S, \$\})$$

计算 I_0 的闭包：

1. $S' \to \cdot S, \$$
2. $S \to \cdot AS, \$$
3. $S \to \cdot b, \$$
4. $A \to \cdot SA, \{a, b\}$
5. $A \to \cdot a, \{a, b\}$

因此，I_0 为：

$$I_0 = \{S' \to \cdot S, \$$$$S \to \cdot AS, \$$$$S \to \cdot b, \$$$$A \to \cdot SA, \{a, b\}$$$$A \to \cdot a, \{a, b\}\}$$

计算项目集规范族的其他项目集

我们需要计算从 I_0 出发的所有可能的移动，并继续计算闭包。

好的，我们详细列出所有项目集以及基于这些项目集的LR(1)分析表。

项目集规范族

项目集 I_0

$$I_0 = \text{CLOSURE}(\{S' \to \cdot S, \$\})$$

$$\begin{gathered} I_0 = \{S' \to \cdot S, \$ \\ S \to \cdot AS, \$ \\ S \to \cdot b, \$ \\ A \to \cdot SA, \{a, b\} \\ A \to \cdot a, \{a, b\}\} \end{gathered}$$

项目集 I_1

$$I_1 = \text{CLOSURE}(\{S' \to S\cdot, \$\})$$

$$I_1 = \{S' \to S\cdot, \$\}$$

项目集 I_2

$$I_2 = \text{CLOSURE}(\{S \to A \cdot S, \$\})$$

$$\begin{gathered} I_2 = \{S \to A \cdot S, \$ \\ S \to \cdot AS, \$ \\ S \to \cdot b, \$ \\ A \to \cdot SA, \{a, b\} \\ A \to \cdot a, \{a, b\}\} \end{gathered}$$

项目集 I_3

$$I_3 = \text{CLOSURE}(\{S \to b\cdot, \$\})$$

$$I_3 = \{S \to b\cdot, \$\}$$

项目集 I_4

$$I_4 = \text{CLOSURE}(\{A \to a\cdot, \{a, b\}\})$$

$$I_4 = \{A \to a\cdot, \{a, b\}\}$$

项目集 I_5

$$I_5 = \text{CLOSURE}(\{S \to AS\cdot, \$\})$$

$$I_5 = \{S \to AS\cdot, \$\}$$

项目集 I_6

$$I_6 = \text{CLOSURE}(\{A \to S \cdot A, \{a, b\}\})$$

$$\begin{gathered} I_6 = \{A \to S \cdot A, \{a, b\} \\ A \to \cdot SA, \{a, b\} \\ A \to \cdot a, \{a, b\}\} \end{gathered}$$

LR(1)分析表

我们使用项目集规范族来构建LR(1)分析表。表格中的条目包括移进 (shift)、规约 (reduce) 和接受 (accept) 操作。

- **移进 (Shift)** 记为 s_i，表示移进并进入状态 i。
- **规约 (Reduce)** 记为 r_j，表示根据第 j 条产生式进行规约。
- **接受 (Accept)** 记为 `acc`，表示接受输入。

项目集规范族的转移

1. I_0:
   - $S \to I_1$
   - $A \to I_2$
   - $b \to I_3$
   - $a \to I_4$
2. I_2:
   - $S \to I_5$
   - $A \to I_2$
   - $b \to I_3$
   - $a \to I_4$
3. I_6:
   - $A \to I_2$
   - $S \to I_5$
   - $b \to I_3$
   - $a \to I_4$

分析表

状态	a	b	S	A	$
I_0	s_4	s_3	I_1	I_2	
I_1					acc
I_2	s_4	s_3	I_5	I_2	
I_3					r_2
I_4					r_4
I_5					r_1
I_6	s_4	s_3	I_5	I_2	

产生式

1. $S \to AS$
2. $S \to b$
3. $A \to SA$
4. $A \to a$

判断是否是LR(1)文法

通过计算项目集规范族和分析表，如果在构建过程中没有出现冲突，那么该文法就是LR(1)文法。否则，不是。

对于本例，通过计算可以发现，在构建项目集规范族时，不会出现移进-规约冲突或规约-规约冲突。因此，该文法是LR(1)文法。

4

```
4.  下面的属性文法可用于将二进制无符号小数转化为十进制小数（开始符号为N）：
(1) N->.S       {S.f:=1; print(S.v)}
(2) S->BS1  {S1.f:=S.f+1; B.f:=S.f; S.v:=B.v+S1.v}
(3) S->ϵ        {S.v:=0}
(4) B->0        {B.v:=0}
(5) B->1        {B.v:=2-B.f}
对于输入串.101的分析树，给出语义计算的过程。
```

给定的属性文法用于将二进制无符号小数转化为十进制小数。我们需要根据输入串`.101`构造分析树，并计算语义属性。

属性文法

文法规则如下：

1. $N \to .S \quad \{S.f := 1; \text{print}(S.v)\}$
2. $S \to BS_1 \quad \{S_1.f := S.f + 1; B.f := S.f; S.v := B.v + S_1.v\}$
3. $S \to \epsilon \quad \{S.v := 0\}$
4. $B \to 0 \quad \{B.v := 0\}$
5. $B \to 1 \quad \{B.v := 2 - B.f\}$

输入串 `.101`

分析树

构造输入串`.101`的分析树：

```
      N
      |
      .S
       |
      B S1
     /|  \
    1 B S2
     /|  \
    0 B S3
      |
      ϵ
```

语义计算过程

从叶子节点到根节点，计算每个节点的属性值。

1. **叶子节点** ϵ
   - $S_3 \to \epsilon$
   - 语义规则：$S_3.v := 0$
2. **节点** B **下的** 0
   - $B \to 0$
   - 语义规则：$B.v := 0$
3. **节点** S_2
   - 语义规则：$S_2.f := S_1.f + 1, S_2.v := B.v + S_3.v$
   - $S_2.f = 2 + 1 = 3$
   - $S_2.v = 0 + 0 = 0$
4. **节点** B **下的** 1
   - $B \to 1$
   - 语义规则：$B.v := 2 - B.f$
   - $B.f = 2, B.v = 2 - 2 = 0$
5. **节点** S_1
   - 语义规则：$S_1.f := S.f + 1, S_1.v := B.v + S_2.v$
   - $S_1.f = 1 + 1 = 2$
   - $S_1.v = 0 + 0 = 0$
6. **节点** B **下的** 1
   - $B \to 1$
   - 语义规则：$B.v := 2 - B.f$
   - $B.f = 1, B.v = 2 - 1 = 1$
7. **节点** S
   - 语义规则：$S_1.f := S.f + 1, S.v := B.v + S_1.v$
   - $S.f = 1, S_1.f = 1 + 1 = 2$
   - $B.f = 1, B.v = 1$
   - $S_1.v = 0$

- $S.v = 1 + 0 = 1$

8. **根节点** N
   - $N \to .S$
   - 语义规则：$S.f := 1; \text{print}(S.v)$
   - $S.f = 1$
   - $S.v = 1$

计算过程总结

通过计算每个节点的属性值，我们得到输入串`.101`的十进制表示。具体计算过程如下：

1. $N \to .S$
   - $S.f := 1$
   - $\text{print}(S.v)$
2. $S \to BS_1$
   - $S_1.f := S.f + 1$
   - $B.f := S.f$
   - $S.v := B.v + S_1.v$
3. $B \to 1$
   - $B.v := 2 - B.f$
4. $S_1 \to BS_2$
   - $S_2.f := S_1.f + 1$
   - $B.f := S_1.f$
   - $S_1.v := B.v + S_2.v$
5. $B \to 0$
   - $B.v := 0$
6. $S_2 \to BS_3$
   - $S_3.f := S_2.f + 1$
   - $B.f := S_2.f$
   - $S_2.v := B.v + S_3.v$
7. $B \to 1$
   - $B.v := 2 - B.f$
8. $S_3 \to \epsilon$
   - $S_3.v := 0$

最终，输出结果为`0.625`。

结论

根据属性文法和输入串`.101`的分析树和语义计算过程，输入串`.101`的十进制表示为`0.625`。

编译原理卷一———By Gerchart

填空题

1. **一个正规式r和一个DFA M等价的含义是（ ）**

   答案：它们识别的语言相同。

   解析：正规式（正则表达式）和确定有限自动机（DFA）都是用来描述正则语言的两种等价表示。一个正规式r和一个DFA M等价的含义是它们识别的语言相同，即正规式r描述的语言和DFA M识别的语言是同一个语言。

2. **程序设计语言C中标识符集合的正规式描述（ ）**

   答案：`[a-zA-Z_][a-zA-Z0-9_]*`

   解析：在C语言中，标识符（变量名、函数名等）必须以字母（小写或大写）或下划线开头，并且后续字符可以是字母、数字或下划线。因此，其正规式可以描述为：`[a-zA-Z_][a-zA-Z0-9_]*`。

3. **LL(1)文法中第一个L的含义是（ ），第二个L的含义是（ ）**

   答案：第一个L是从左到右扫描输入串，第二个L是最左推导。

   解析：LL(1)文法中的第一个L表示从左到右扫描输入串（Left-to-right scanning of the input string），第二个L表示使用最左推导（Leftmost derivation），括号中的1表示在每一步推导中只需要查看一个输入符号（1-symbol lookahead）。

4. **语句`if(a>b or b<c) then x:=a*(b-c)+(b*c-d)/e else x:=0`的逆波兰式为 （ ）**

   答案：`a b > b c < or a b c - * b c * d - e / + x := 0 x := ifelse`

   解析：逆波兰式（后缀表达式）是一种将运算符放在操作数之后的表达方式。具体步骤如下：

   - `a b >` 表示 `a > b`
   - `b c <` 表示 `b < c`
   - `or` 表示逻辑或
   - `a b c - *` 表示 `a * (b - c)`
   - `b c * d -` 表示 `b * c - d`
   - `(b * c - d) / e` 表示 `b c * d - e /`
   - `a * (b - c) + (b * c - d) / e` 表示 `a b c - * b c * d - e / +`
   - `x :=` 表示赋值操作
   - `0 x :=` 表示 `x := 0`
   - `ifelse` 表示条件选择

5. **文法G(S):

   ```
   S->aA|bB
   A->bS|b
   B->aS|a
   ```

   所对应的正规式为**

   答案：`(a(b|ab(a|b)*)|b(a|ba(b|a)*)`

解析：这个文法描述的语言是所有形如 `ab` 和 `ba` 的字符串，以及这些模式的递归嵌套。我们可以通过分析文法规则来推导其正规式：

- `S -> aA | bB`
- `A -> bS | b`
- `B -> aS | a`

其中，`S` 可以生成的字符串模式是 `a` 后接 `b` 或 `ab` 后接 `a` 或 `b` 等。总结起来，正规式可以表示为 `(a(b|ab(a|b)*)|b(a|ba(b|a)*)`

6. **对于一个按行存放的整型数组B[i] [j] [k] [l]，下标i的取值范围是0_5，j的取值范围是2_4，k的取值范围是1_4，l的取值范围是4_8。假设每个元素占8个字节，数组B从编号为1000的字节开始存放，那么元素B[4] [2] [3] [7]的起始位置为第（ ）字节**

答案：2720

解析：计算元素B[4] [2] [3] [7]的起始位置时，首先需要确定数组的各维度的跨度（stride），即每个维度的跨度是多少字节。

- 每个元素占8个字节。
- l维度跨度：1个元素，占8字节。
- k维度跨度：5个元素，占5 * 8 = 40字节。
- j维度跨度：4个元素，占4 * 40 = 160字节。
- i维度跨度：3个元素，占3 * 160 = 480字节。

计算B[4] [2] [3] [7]的偏移量:

- i = 4: 4 * 480 = 1920字节
- j = 2 (实际是第0个位置，因为j的取值范围是2~4): 0 * 160 = 0字节
- k = 3 (实际是第2个位置，因为k的取值范围是1~4): 2 * 40 = 80字节
- l = 7 (实际是第3个位置，因为l的取值范围是4~8): 3 * 8 = 24字节

总偏移量 = 1920 + 0 + 80 + 24 = 2024字节

数组开始位置是1000字节，所以起始位置为1000 + 2024 = 3024字节。

判断题

1. **若文法G是二义的，则G所描述的语言L(G)也是二义的。**

答案：错误。

解析：文法的二义性（ambiguous grammar）指的是某个文法可以为某些字符串生成多个不同的语法树（解析树）。但是，语言的二义性（ambiguous language）是指任何文法都无法无二义地描述该语言。一个文法的二义性并不必然意味着该语言也是二义的。例如，算术表达式的语言可以有二义的文法和无二义的文法。

2. **正规式(0|1)*和((ε|0)1*)*是等价的。**

答案：正确。

解析：正规式(0|1)*表示任意长度的0和1的串，包括空串。正规式((ε|0)1)*表示由ε或0开头，后面跟任意长度的1的串，再任意重复这个模式。实际上，这两者描述的都是任意长度的0和1的串，包括空串，因此它们是等价的。

3. **SLR(1)文法不一定是LALR(1)文法。**

   答案：错误。

   解析：SLR(1)文法是LALR(1)文法的一个子集。也就是说，每个SLR(1)文法都是LALR(1)文法，但不是所有的LALR(1)文法都是SLR(1)文法。因此，SLR(1)文法一定是LALR(1)文法。

4. **只含有综合属性的属性文法称为L属性文法，L属性文法的翻译器通常可借助LR分析器实现。**

   答案：错误。

   解析：只含有综合属性的属性文法称为S属性文法（Synthesized attributes）。L属性文法（L-attributed grammar）允许在每个语法规则中使用左侧非终结符和右侧非终结符的继承属性和综合属性。L属性文法的翻译器通常可借助自顶向下的分析方法（如LL分析器）实现，而不是LR分析器。

5. **编译程序的自展技术是先用目标机的汇编语言或机器语言书写源语言的一个子集的编译程序，然后再用这个子集作为书写语言，实现源语言的编译程序。**

   答案：正确。

   解析：编译程序的自展技术（bootstrapping）是指首先用目标机的汇编语言或机器语言编写一个可以编译源语言子集的编译器，然后使用这个编译器编写更完整的编译器。这个过程是逐步实现编译器的技术，可以最终实现对源语言的完整编译。

简答

一

- 问题：给出生成语言 $\{a^n b^m \mid n, m \geq 1\}$ 的正则文法

解答

生成语言 $\{a^n b^m \mid n, m \geq 1\}$ 的正则文法可以通过以下规则来定义：

1. $S \rightarrow aA$
2. $A \rightarrow aA \mid B$
3. $B \rightarrow bB \mid b$

解释：

- S 是初始符号。
- S 通过规则 1 生成至少一个 a，然后进入状态 A。
- A 通过规则 2 可以生成任意个 a（包括零个），然后进入状态 B。

- B 通过规则 3 可以生成至少一个 b，并且可以生成任意个 b。

这个正则文法确保了生成的字符串总是以一个或多个 a 开头，后面跟着一个或多个 b，符合 $\{a^n b^m \mid n, m \geq 1\}$ 的形式。

具体生成过程示例：

- $S \rightarrow aA$
- $aA \rightarrow aaA$
- $aaA \rightarrow aaB$
- $aaB \rightarrow aabB$
- $aabB \rightarrow aabb$

这样生成的字符串如 "ab", "aab", "aaabbb" 等等，都符合 $\{a^n b^m \mid n, m \geq 1\}$ 的形式。

二

```
2.  对于文法G(E):
S->Aa|b
A->SB
B->ab
对其进行改写（消除左递归，提取公因子），判断改写后的文法是否是LL(1)的。
```

要对文法 $G(E)$ 进行改写（消除左递归，提取公因子），我们首先列出原始文法规则：

1. $S \rightarrow Aa \mid b$
2. $A \rightarrow SB$
3. $B \rightarrow ab$

步骤 1: 消除左递归

在这个文法中，原本并没有左递归，所以我们不需要进行左递归的消除。

步骤 2: 提取公因子

我们检查每条产生式，看看是否有公共前缀可以提取。

对于 S 的产生式：

- $S \rightarrow Aa \mid b$

这里没有公共前缀，不需要提取公因子。

对于 A 的产生式：

- $A \rightarrow SB$

这里也没有公共前缀，不需要提取公因子。

对于 B 的产生式：

- $B \rightarrow ab$

这里也没有公共前缀，不需要提取公因子。

所以，经过检查，我们发现这个文法不需要进行任何改写，因为它既没有左递归，也没有需要提取的公因子。

判断改写后的文法是否是 LL(1)

要判断文法是否是 LL(1) 文法，需要检查以下几点：

1. **每个非终结符的产生式的首符号集（FIRST 集）是否不相交。**
2. **对于每个非终结符 A，如果 $A \to \epsilon$ 是一个产生式，那么 A 的 FOLLOW 集与其他产生式的首符号集是否不相交。**

我们计算每个非终结符的 FIRST 和 FOLLOW 集。

计算 FIRST 集

- $FIRST(S) = \{a, b\}$ 因为 $S \to Aa$ 和 $S \to b$。
- $FIRST(A) = \{a, b\}$ 因为 $A \to SB$，S 的 FIRST 集是 $\{a, b\}$。
- $FIRST(B) = \{a\}$ 因为 $B \to ab$。

计算 FOLLOW 集

- $FOLLOW(S) = \{\$, a\}$ 因为 S 是开始符号，且 $A \to SB$。
- $FOLLOW(A) = \{a\}$ 因为 $S \to Aa$。
- $FOLLOW(B) = \{a\}$ 因为 $A \to SB$。

检查 LL(1) 条件

1. **FIRST 集是否不相交：**
   - 对于 S 的产生式，$FIRST(Aa) = \{a\}$ 和 $FIRST(b) = \{b\}$ 不相交。
   - 对于 A 的产生式，只有一个产生式，满足条件。
   - 对于 B 的产生式，只有一个产生式，满足条件。
2. **FOLLOW 集是否与 FIRST 集不相交：**
   - S 没有 ϵ 产生式。
   - A 没有 ϵ 产生式。
   - B 没有 ϵ 产生式。

所有条件都满足，因此我们可以确定该文法是 LL(1) 文法。

总结

经过检查和分析，原始文法 $G(E)$ 不需要任何改写，因为它没有左递归，也没有需要提取的公因子。并且，经过计算 FIRST 和 FOLLOW 集并检查条件，我们确认该文法是 LL(1) 文法。

三

> 3． 原本无冲突的LR(1)分析表在合并同心集后得到LALR(1)分析表，新的表中不可能存在哪种冲突？为什么?

在将原本无冲突的 LR(1) 分析表合并同心集（即具有相同核心的项集）后得到 LALR(1) 分析表时，新的表中不可能存在 **移入-移入冲突**。这是因为：

理由

1. **核心状态相同**：
   - 在 LR(1) 分析表中，项集的核心（即不包含展望符号的部分）是唯一确定的。即使两个项集的展望符号不同，但如果它们的核心相同，它们的移入动作也相同。
   - 在合并同心集时，我们将具有相同核心的项集合并，这意味着它们的移入动作不会发生改变。
2. **移入动作的确定性**：
   - 在 LR(1) 分析表中，移入动作是根据当前输入符号和状态来确定的。如果两个项集的核心相同，那么它们在遇到相同输入符号时的移入动作是相同的。
   - 合并同心集不会改变移入动作的确定性，因为核心状态决定了移入动作。

由于上述原因，合并同心集不会引入新的移入-移入冲突。移入-移入冲突的可能性仅取决于核心状态，而不是展望符号。

可能存在的冲突

尽管合并同心集不会引入移入-移入冲突，但它可能会引入 **移入-规约冲突** 或 **规约-规约冲突**。这是因为不同项集的展望符号在合并后可能会导致冲突：

- **移入-规约冲突**：在合并同心集后，不同展望符号可能会导致在同一状态下既有移入动作，又有规约动作。
- **规约-规约冲突**：在合并同心集后，不同展望符号可能会导致在同一状态下有多个规约动作。

总结

在合并同心集后得到的 LALR(1) 分析表中，不可能存在移入-移入冲突，因为合并同心集不会改变移入动作的确定性。然而，合并同心集可能会引入移入-规约冲突或规约-规约冲突。

四

```
4.  对于文法G(E):
E->T|E+T
T->F|T*F
F->(E)|i
(1) 写出句型(T*F+i)的最右推导并画出语法树。
(2) 写出上述句型的短语，直接短语和句柄。
```

对于文法 $G(E)$:

1. $E \to T \mid E + T$
2. $T \to F \mid T * F$
3. $F \to (E) \mid i$

(1) 写出句型 $(T * F + i)$ 的最右推导并画出语法树

首先，我们要生成句型 $(T * F + i)$。根据文法规则，我们可以进行最右推导。

最右推导过程如下：

1. $E \to E + T$

2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow i$

结合以上推导步骤，我们可以得到：

1. $E \rightarrow E + T$
2. $E \rightarrow T * F$
3. $T \rightarrow i$
4. $F \rightarrow i$

因此，最右推导过程为：
$[E \rightarrow E + T \rightarrow T + T \rightarrow T * F + T \rightarrow i * F + T \rightarrow i * i + T \rightarrow i * i + i]$

语法树如下：

```
      E
     / \
    E   T
   /|\   |
  T * F  i
  |   |
  i   F
      |
      i
```

(2) 写出上述句型的短语，直接短语和句柄

- **短语**：短语是文法中的非终结符及其所生成的子树。根据最右推导，短语包括：
  1. i
  2. $F \rightarrow i$
  3. $T \rightarrow F$
  4. $T \rightarrow T * F$
  5. $E \rightarrow T$
  6. $E \rightarrow E + T$
- **直接短语**：直接短语是指在最右推导中一步生成的短语。在上述最右推导中，直接短语包括：
  1. i
  2. $F \rightarrow i$
  3. $T \rightarrow F$
  4. $T \rightarrow T * F$
  5. $E \rightarrow T$
  6. $E \rightarrow E + T$
- **句柄**：句柄是指在最右推导中，当前最右边的非终结符及其所生成的子树。在上述最右推导中，句柄包括：
  1. i

2. $F \to i$
3. $T \to F$
4. $T \to T * F$
5. $E \to T$
6. $E \to E + T$

总结：

- 短语：i，$F \to i$，$T \to F$，$T \to T * F$，$E \to T$，$E \to E + T$
- 直接短语：i，$F \to i$，$T \to F$，$T \to T * F$，$E \to T$，$E \to E + T$
- 句柄：i，$F \to i$，$T \to F$，$T \to T * F$，$E \to T$，$E \to E + T$

通过推导和语法树的展示，我们可以清晰地看到句型 $(T * F + i)$ 的生成过程及其对应的短语、直接短语和句柄。

解答

一

```
1.  写出字母表Σ={a,b}上语言L={w|w的最后两个字母是aa或bb}的正规式，并将其转换为最简DFA。
```

正规式

给定字母表 $\Sigma = \{a, b\}$ 和语言 $L = \{w \mid w$ 的最后两个字母是 aa 或 $bb\}$，这个语言可以用正规式表示为：
[(a|b)^*(aa|bb)]

转换为最简DFA

为了将这个正规式转换为最简的确定有限自动机（DFA），我们首先通过构建 NFA，然后将其转换为 DFA，最后进行状态最小化。

步骤 1: 构建 NFA

我们构建一个 NFA 来接受正规式 $(a|b)^*(aa|bb)$。这个 NFA 包括以下状态：

- 初始状态 q_0，可以通过 a 或 b 转移到自身。
- 接受状态 q_1，可以通过 a 从 q_0 转移到 q_2，然后再通过 a 转移到 q_3。
- 接受状态 q_4，可以通过 b 从 q_0 转移到 q_5，然后再通过 b 转移到 q_6。

NFA 的状态转移图如下：

```
q0 --a--> q0
q0 --b--> q0
q0 --a--> q2 --a--> q3 (接受状态)
q0 --b--> q5 --b--> q6 (接受状态)
```

步骤 2: 转换 NFA 为 DFA

我们使用子集构造法将 NFA 转换为 DFA。以下是转换过程：

1. 初始状态为 $\{q_0\}$。
2. 处理 $\{q_0\}$：
   - 通过 a 转移到 $\{q_0, q_2\}$。
   - 通过 b 转移到 $\{q_0, q_5\}$。
3. 处理 $\{q_0, q_2\}$：
   - 通过 a 转移到 $\{q_0, q_2, q_3\}$。
   - 通过 b 转移到 $\{q_0, q_5\}$。
4. 处理 $\{q_0, q_5\}$：
   - 通过 a 转移到 $\{q_0, q_2\}$。
   - 通过 b 转移到 $\{q_0, q_5, q_6\}$。
5. 处理 $\{q_0, q_2, q_3\}$：
   - 通过 a 转移到 $\{q_0, q_2, q_3\}$。
   - 通过 b 转移到 $\{q_0, q_5\}$。
6. 处理 $\{q_0, q_5, q_6\}$：
   - 通过 a 转移到 $\{q_0, q_2\}$。
   - 通过 b 转移到 $\{q_0, q_5, q_6\}$。

步骤 3: 最小化 DFA

在最小化 DFA 之前，我们标记接受状态：

- $\{q_0, q_2, q_3\}$ 是接受状态，因为包含 q_3。
- $\{q_0, q_5, q_6\}$ 是接受状态，因为包含 q_6。

最小化后的 DFA 状态转移图如下：

```
状态     | a                        | b
---------|--------------------------|-------------------------
{q_0}    | {q_0, q_2}               | {q_0, q_5}
{q_0, q_2} | {q_0, q_2, q_3}        | {q_0, q_5}
{q_0, q_2, q_3} | {q_0, q_2, q_3}   | {q_0, q_5}
{q_0, q_5} | {q_0, q_2}             | {q_0, q_5, q_6}
{q_0, q_5, q_6} | {q_0, q_2}        | {q_0, q_5, q_6}
```

最小化后的 DFA 状态图如下：

```
          | a             | b
          v               |
  +-------+-------+     |
  | q0, q5, q6    | <---+
  +-------+-------+
```

接受状态为 $\{q_0, q_2, q_3\}$ 和 $\{q_0, q_5, q_6\}$。

总结：

- 最简 DFA 的状态为 $\{q_0\}$、$\{q_0, q_2, q_3\}$ 和 $\{q_0, q_5, q_6\}$。
- $\{q_0, q_2, q_3\}$ 和 $\{q_0, q_5, q_6\}$ 是接受状态。

这个 DFA 接受语言 $L = \{w \mid w$ 的最后两个字母是 aa 或 $bb\}$。

二

```
2.  已知文法G[S]:
S->aBS|bAS|ϵ
A->bAA|a
B->aBB|b
    判断文法是否为LL(1)文法，并说明理由，如果是，给出其LL(1)分析表。
```

要判断一个文法是否为 LL(1) 文法，我们需要检查以下条件：

1. **每个非终结符的产生式的首符号集（FIRST 集）是否不相交。**
2. **对于每个非终结符 A，如果 $A \to \epsilon$ 是一个产生式，那么 A 的 FOLLOW 集与其他产生式的首符号集是否不相交。**

计算 FIRST 集和 FOLLOW 集

计算 FIRST 集

我们依次计算每个非终结符的 FIRST 集：

- 对于 S:
  - $S \to aBS$，$FIRST(aBS) = \{a\}$
  - $S \to bAS$，$FIRST(bAS) = \{b\}$
  - $S \to \epsilon$，$FIRST(\epsilon) = \{\epsilon\}$
  - 所以，$FIRST(S) = \{a, b, \epsilon\}$
- 对于 A:
  - $A \to bAA$，$FIRST(bAA) = \{b\}$
  - $A \to a$，$FIRST(a) = \{a\}$
  - 所以，$FIRST(A) = \{a, b\}$
- 对于 B:
  - $B \to aBB$，$FIRST(aBB) = \{a\}$
  - $B \to b$，$FIRST(b) = \{b\}$
  - 所以，$FIRST(B) = \{a, b\}$

计算 FOLLOW 集

我们依次计算每个非终结符的 FOLLOW 集：

- 对于 S：
  - S 是起始符号，所以 $FOLLOW(S)$ 包含 $\{\$（输入结束符）\}$
  - $S \to aBS$ 和 $S \to bAS$ 中 S 后没有其他符号，$FOLLOW(S)$ 包含 $FOLLOW(S)$
  - 所以，$FOLLOW(S) = \{\$（输入结束符）\}$
- 对于 A：
  - $S \to bAS$，$FOLLOW(A)$ 包含 $FOLLOW(S) = \{\$（输入结束符）\}$
  - $A \to bAA$，$FOLLOW(A)$ 包含 $FIRST(A) = \{a, b\}$
  - 所以，$FOLLOW(A) = \{\$, a, b\}$
- 对于 B：
  - $S \to aBS$，$FOLLOW(B)$ 包含 $FOLLOW(S) = \{\$（输入结束符）\}$
  - $B \to aBB$，$FOLLOW(B)$ 包含 $FIRST(B) = \{a, b\}$
  - 所以，$FOLLOW(B) = \{\$, a, b\}$

检查 LL(1) 条件

1. **检查 FIRST 集是否不相交：**
   - 对于 S 的产生式，$FIRST(aBS) = \{a\}$，$FIRST(bAS) = \{b\}$，$FIRST(\epsilon) = \{\epsilon\}$，这些集合不相交。
   - 对于 A 的产生式，$FIRST(bAA) = \{b\}$ 和 $FIRST(a) = \{a\}$，这些集合不相交。
   - 对于 B 的产生式，$FIRST(aBB) = \{a\}$ 和 $FIRST(b) = \{b\}$，这些集合不相交。
2. **检查 FOLLOW 集和 FIRST 集是否不相交：**
   - $S \to \epsilon$，$FOLLOW(S) = \{\$\}$ 和 $FIRST(S) = \{a, b\}$，这些集合不相交。
   - A 没有 ϵ 产生式。
   - B 没有 ϵ 产生式。

所有条件都满足，因此我们可以确定该文法是 LL(1) 文法。

LL(1) 分析表

根据 FIRST 和 FOLLOW 集构建 LL(1) 分析表：

非终结符	a	b	$
S	$S \to aBS$	$S \to bAS$	$S \to \epsilon$
A	$A \to a$	$A \to bAA$	
B	$B \to aBB$	$B \to b$	

总结

该文法是 LL(1) 文法，LL(1) 分析表如下：

非终结符	a	b	$
S	$S \to aBS$	$S \to bAS$	$S \to \epsilon$
A	$A \to a$	$A \to bAA$	
B	$B \to aBB$	$B \to b$	

通过计算和检查，我们确认该文法符合 LL(1) 文法的所有条件，并给出了相应的 LL(1) 分析表。

三

已知文法：
Z->CS
C->if E then
S->A=E
E->E ∨ A
A->i
构造此文法的LR(0)项目集 的规范族，并给出识别活前缀的DFA。
判断其是否为SLR(1)文法，如果是，构造其SLR(1)分析表。

构造 LR(0) 项目集的规范族

首先，我们将文法扩展为包含新的开始符号的文法：

```
Z' -> Z
Z  -> CS
C  -> if E then
S  -> A=E
E  -> E ∨ A
E  -> A
A  -> i
```

LR(0) 项目

- 项目是产生式的形式，带有一个位置标记“.”，表示已经处理的输入位置。

项目集的构造

1. **初始项集** I_0：
   - $I_0 = \{Z' \to .Z\}$
   - 闭包计算：
     - $Z' \to .Z$
     - $Z \to .CS$
     - $C \to .ifEthen$
     - $S \to .A = E$
     - $E \to .E \vee A$

- $E \rightarrow . A$
- $A \rightarrow . i$

- 所以，
  $I_0 = \{Z' \rightarrow . Z, Z \rightarrow . CS, C \rightarrow . if Ethen, S \rightarrow . A = E, E \rightarrow . E \vee A, E \rightarrow . A, A \rightarrow . i\}$

2. **项目集的构造过程**：

我们通过项目集的闭包和转移来构造所有项目集。以下是项目集的构造过程：

- I_0：

```
Z' -> .Z
Z  -> .CS
C  -> .if E then
S  -> .A=E
E  -> .E ∨ A
E  -> .A
A  -> .i
```

- I_1：从 I_0 出发，通过 Z 转移：

```
Z' -> Z.
```

- I_2：从 I_0 出发，通过 C 转移：

```
Z  -> C.S
S  -> .A=E
E  -> .E ∨ A
E  -> .A
A  -> .i
```

- I_3：从 I_0 出发，通过 if 转移：

```
C  -> if .E then
E  -> .E ∨ A
E  -> .A
A  -> .i
```

- I_4：从 I_0 出发，通过 A 转移：

```
E  -> A.
```

- I_5：从 I_0 出发，通过 E 转移：

```
C  -> if E .then
E  -> E .∨ A
```

- I_6：从 I_3 出发，通过 E 转移：

```
C  -> if E .then
E  -> E .∨ A
```

- I_7：从 I_5 出发，通过 $then$ 转移：

```
C  -> if E then.
```

- I_8：从 I_6 出发，通过 $then$ 转移：

```
C  -> if E then.
```

- I_9：从 I_5 出发，通过 $\vee$ 转移：

```
E  -> E ∨ .A
A  -> .i
```

- I_{10}：从 I_9 出发，通过 A 转移：

```
E  -> E ∨ A.
```

- I_{11}：从 I_0 出发，通过 i 转移：

```
A  -> i.
```

- I_{12}：从 I_2 出发，通过 S 转移：

```
Z  -> CS.
```

- I_{13}：从 I_2 出发，通过 A 转移：

```
S  -> A.=E
E  -> .E ∨ A
E  -> .A
A  -> .i
```

- I_{14}：从 I_{13} 出发，通过 E 转移：

```
S  -> A=E.
E  -> E .∨ A
```

- I_{15}：从 I_{13} 出发，通过 i 转移：

```
A  -> i.
```

- I_{16}：从 I_{14} 出发，通过 $\vee$ 转移：

```
E  -> E ∨ .A
A  -> .i
```

- I_{17}：从 I_{16} 出发通过 A 转移：

```
E  -> E ∨ A.
```

识别活前缀的 DFA

我们可以根据上述项目集构建识别活前缀的 DFA。状态转移图如下：

```
I0 --Z--> I1
I0 --C--> I2
I0 --if--> I3
I0 --A--> I4
I0 --E--> I5
I0 --i--> I11

I2 --S--> I12
I2 --A--> I13

I3 --E--> I6

I5 --then--> I7
I5 --∨--> I9

I6 --then--> I8

I9 --A--> I10
I9 --i--> I11

I13 --E--> I14
I13 --i--> I15

I14 --∨--> I16

I16 --A--> I17
I16 --i--> I11
```

判断是否为 SLR(1) 文法

计算 FOLLOW 集

- $FOLLOW(Z) = \{\$\}$
- $FOLLOW(C) = FOLLOW(Z) = \{\$\}$
- $FOLLOW(S) = FOLLOW(Z) = \{\$\}$
- $FOLLOW(A) = \{=, \vee, \$\}$
- $FOLLOW(E) = \{then, \vee, \$\}$

检查 SLR(1) 条件

1. **移入-规约冲突**：项目集中的移入动作和规约动作是否在相同的输入符号上发生冲突。
2. **规约-规约冲突**：项目集中的不同规约动作是否在相同的输入符号上发生冲突。

我们需要检查每个项目集中是否存在移入-规约冲突或规约-规约冲突。

- 在 I_5 和 I_6 中，$E \to E.\vee A$ 和 $C \to ifE.then$，没有冲突。
- 在 I_9 中，$E \to E\vee.A$ 和 $A \to .i$，没有冲突。
- 在 I_{13} 中，$S \to A.=E$ 和 $E \to .E \vee A$，没有冲突。

所有项目集中没有移入-规约冲突或规约-规约冲突，因此该文法是 SLR(1) 文法。

构造 SLR(1) 分析表

根据项目集和 FOLLOW 集构建 SLR(1) 分析表：

状态	if	i	then	=	v	$	Z	C	S	E	A
I0	s3	s11					1	2		5	4
I1						acc					
I2		s11							12	14	13
I3		s11								6	4
I4											
I5			s7		s9						
I6			s8		s9						
I7											
I8											
I9		s11									10
I10											
I11											
I12											
I13		s15								14	13
I14					s16						
I15											
I16		s11									17
I17											

总结：

- 该文法是 SLR(1) 文法。
- 上述表格是根据项目集和 FOLLOW 集构建的 SLR(1) 分析表。

四

```
4.  请将下列语句
while(A<B) do if (c>0) then X:=Y+Z
翻译成四元式（三地址代码）
注：假定翻译的四元式序列从（100）开始，必须注明在哪里使用拉链-回填技术。
```

将语句 `while(A<B) do if (c>0) then X:=Y+Z` 翻译成四元式（三地址代码）并使用拉链-回填技术如下：

分析语句结构

1. `while (A < B) do`
2. `if (c > 0) then`
3. `X := Y + Z`

四元式表示

我们将使用拉链-回填技术来处理控制流，特别是跳转指令。假定四元式序列从编号 100 开始。

1. `while (A < B) do`

首先，我们需要一个标签来表示循环的开始位置，并在条件语句不满足时跳出循环。

- `L1:` 标记循环开始的位置
- `L2:` 标记循环体结束后跳转的位置

2. `if (c > 0) then`

我们需要一个标签来表示条件语句不满足时跳转的位置。

- `L3:` 标记 `if` 条件不满足时跳转的位置

3. `X := Y + Z`

这是一个简单的赋值操作。

四元式代码

我们将逐步生成四元式代码，并使用拉链-回填技术来处理跳转指令。

1. **开始循环**：
   - `L1:` (100)
2. **条件检查** `A < B`：
   - `t1 := A < B` (101)
   - `if_false t1 goto L2` (102) // 如果 `A < B` 为假，跳转到 `L2`（循环结束）
3. `if (c > 0) then`：
   - `t2 := c > 0` (103)
   - `if_false t2 goto L3` (104) // 如果 `c > 0` 为假，跳转到 `L3`（跳过 `then` 部分）
4. `X := Y + Z`：
   - `t3 := Y + Z` (105)
   - `X := t3` (106)
5. **结束** `if` **语句**：
   - `L3:` (107)
6. **跳回循环开始**：

- `goto L1` (108)

7. **结束循环**：
   - `L2:` (109)

四元式序列

下面是完整的四元式序列：

编号	操作	操作数1	操作数2	结果
100	L1:			
101	<	A	B	t1
102	if_false	t1		L2
103	>	c	0	t2
104	if_false	t2		L3
105	+	Y	Z	t3
106	:=	t3		X
107	L3:			
108	goto			L1
109	L2:			

说明拉链-回填技术的使用

- 在生成条件跳转指令时，我们使用了拉链-回填技术。
- 在编号 102 和 104 处，我们生成了条件跳转指令 `if_false`，这些跳转指令在生成时并不知道目标标签的位置。
- 在生成完循环体和 `if` 语句的主体后，我们回填这些跳转指令，使它们跳转到正确的位置（`L2` 和 `L3`）。

通过这种方式，生成的四元式代码能够正确地处理控制流，并实现语句 `while(A<B) do if (c>0) then X:=Y+Z` 的逻辑。

五

```
5.  已知三地址代码片段如下：
(1) b=1
(2) n=10
(3) d=1+n
(4) c=2
(5) t1=4*a
(6) t2=1+n
(7) c=c+b
(8) if c<n goto(6)
(9) a=a+b
(10)    if a<n goto(4)
```

```
(11)    c=a-b
```
(1) 给出相应的流图（使用三地址代码前的序号来表示对应的三地址代码），并指出其中所有的循环。
(2) 指出全局公共子表达式，并给出其所在的四元式的序号。
(3) 指出每个循环中的可进行强度消减的归纳变量，并说明如何进行强度消减。

(1) 流图和循环

三地址代码片段

```
(1)  b = 1
(2)  n = 10
(3)  d = 1 + n
(4)  c = 2
(5)  t1 = 4 * a
(6)  t2 = 1 + n
(7)  c = c + b
(8)  if c < n goto (6)
(9)  a = a + b
(10) if a < n goto (4)
(11) c = a - b
```

流图

我们用三地址代码前的序号来表示对应的三地址代码，并画出流图。

```
(1)  b = 1
 |
 v
(2)  n = 10
 |
 v
(3)  d = 1 + n
 |
 v
(4)  c = 2
 |
 v
(5)  t1 = 4 * a
 |
 v
(6)  t2 = 1 + n
 |
 v
(7)  c = c + b
 |
 v
(8)  if c < n goto (6)
 |
 v
(9)  a = a + b
 |
 v
(10) if a < n goto (4)
 |
```

```
 v
(11) c = a - b
```

循环

1. **外循环**：
    - 从 (4) 到 (10)：
        - (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) -> (4)
2. **内循环**：
    - 从 (6) 到 (8)：
        - (6) -> (7) -> (8) -> (6)

(2) 全局公共子表达式

- 公共子表达式是多个地方计算相同的表达式。

在上述三地址代码片段中，`1 + n` 出现了两次：

- (3) `d = 1 + n`
- (6) `t2 = 1 + n`

这两个表达式是全局公共子表达式，所在的序号是 (3) 和 (6)。

(3) 强度消减

强度消减是一种优化技术，将高开销的操作（如乘法）替换为低开销的操作（如加法）。

在循环中，我们寻找可以进行强度消减的归纳变量。

外循环中的归纳变量

- **循环变量**：`a`
- **归纳变量**：`t1 = 4 * a`

可以进行强度消减的归纳变量是 `t1`，因为每次循环中 `a` 增加 `b`，所以 `4 * a` 可以通过加法来代替乘法。

强度消减方法

我们可以用一个新的变量 `t1` 来表示 `4 * a`，并在每次 `a` 增加时更新 `t1`。

假设初始时 `t1 = 4 * a`，在每次 `a = a + b` 时，我们更新 `t1` 为 `t1 = t1 + 4 * b`。

修改后的代码

```
(1)  b = 1
(2)  n = 10
(3)  d = 1 + n
(4)  c = 2
(5)  t1 = 4 * a   // 初始计算
(6)  t2 = 1 + n
(7)  c = c + b
(8)  if c < n goto (6)
(9)  a = a + b
(10) t1 = t1 + 4 * b  // 更新 t1
(11) if a < n goto (4)
(12) c = a - b
```

在修改后的代码中，我们将 `t1` 的计算进行了强度消减，每次循环更新 `t1` 的值而不是重新计算 `4 * a`。

总结：

- **流图**：如上所示，包含两个循环。
- **全局公共子表达式**：`1 + n`，所在序号为 (3) 和 (6)。
- **强度消减**：将 `4 * a` 的计算替换为 `t1 = t1 + 4 * b`。

编译原理卷二——By Gerchart

填空

1. 编译程序是一个语言翻译程序，它把源语言书写的程序（翻译成目标语言的程序）。
2. 编译过程的前端主要包括（词法分析），（语法分析），（语义分析）和（中间代码生成）。
3. 编译的“遍”是指（编译器在处理源代码时从头到尾扫描和处理源代码的次数）。
4. 字符集合{a,b}的正闭包是：（{a, b}的所有非空字符串集合，即(a|b)(a|b)*）。
5. 文法定义四元组包括（非终结符集合），（终结符集合），（开始符），（产生式集合）。
6. 在Chomsky的文法分类中，单词对应（0型文法）。
7. 一个上下文无关文法是LL(1)文法的充分必要条件是（对于文法中的每个非终结符A，First(A)和Follow(A)集合不相交）。
8. 常用编译程序的生成技术包括（词法分析器生成器，如Lex），（语法分析器生成器，如Yacc），（手工编写的编译器）。
9. 请写出三种常见的中间代码优化技术：（常量折叠），（死代码消除），（循环优化）。
10. 常见单词种类有：关键词、标识符、（常量）、算符、（分隔符）。

带解析

1. **编译程序是一个语言翻译程序，它把源语言书写的程序翻译成目标语言的程序。**
   - **解析**：编译程序（编译器）是将用高级编程语言（如C、Java）编写的源代码翻译成目标代码（如机器码、字节码）的工具。目标代码是计算机可以直接执行的代码。
2. **编译过程的前端主要包括词法分析，语法分析，语义分析和中间代码生成。**
   - 解析：
     - **词法分析**：将源代码分解成记号（tokens）。
     - **语法分析**：根据语法规则将记号序列解析成语法树。
     - **语义分析**：检查语法树是否符合语言的语义规则，并进行类型检查等。
     - **中间代码生成**：将语法树转换为中间代码，这是一种介于高级语言和机器码之间的代码形式。
3. **编译的“遍”是指编译器在处理源代码时从头到尾扫描和处理源代码的次数。**
   - **解析**：在编译过程中，编译器可能需要多次遍历源代码以完成不同阶段的处理，如词法分析、语法分析等。每次完整的扫描和处理称为一“遍”。
4. **字符集合{a,b}的正闭包是：{a, b}的所有非空字符串集合，即(a|b)(a|b)*。**
   - **解析**：正闭包表示由集合中的元素组成的所有可能的非空字符串。对于集合{a, b}，正闭包包括所有由a和b组成的非空字符串，如a, b, aa, ab, ba, bb, aaa等。
5. **文法定义四元组包括非终结符集合，终结符集合，开始符，产生式集合。**
   - 解析：文法（Grammar）通常由以下四个部分组成：
     - **非终结符集合（N）**：文法中的变量，代表不同的语法结构。
     - **终结符集合（Σ）**：文法中的基本符号，代表具体的输入符号。

- **开始符（S）**：文法生成的句子的起始符号。
- **产生式集合（P）**：定义如何将非终结符转换为终结符或其他非终结符的规则。

6. **在Chomsky的文法分类中，单词对应0型文法（无限制文法）。**
   - **解析**：根据Chomsky的文法分类，0型文法是最一般的文法类型，也称为无限制文法。它可以生成任何可能的语言，包括所有其他类型的文法生成的语言。
7. **一个上下文无关文法是LL(1)文法的充分必要条件是对于文法中的每个非终结符A，First(A)和Follow(A)集合不相交。**
   - **解析**：LL(1)文法是一种特殊的上下文无关文法，能够通过自顶向下的解析方法进行解析。一个文法是LL(1)文法的充分必要条件是，对于每个非终结符A，其First集合和Follow集合不相交。这确保了在解析过程中，能够通过查看下一个输入符号唯一确定使用哪条产生式。
8. **常用编译程序的生成技术包括词法分析器生成器（如Lex），语法分析器生成器（如Yacc），手工编写的编译器。**
   - 解析：
     - **词法分析器生成器（如Lex）**：自动生成词法分析器的工具。
     - **语法分析器生成器（如Yacc）**：自动生成语法分析器的工具。
     - **手工编写的编译器**：由开发者手动编写的编译器，各个阶段的处理由程序员明确实现。
9. **请写出三种常见的中间代码优化技术：常量折叠，死代码消除，循环优化。**
   - 解析：
     - **常量折叠**：在编译时计算常量表达式的值，从而减少运行时的计算量。
     - **死代码消除**：移除程序中不会被执行的代码，提高代码效率。
     - **循环优化**：包括循环展开、循环合并等技术，以提高循环执行效率。
10. **常见单词种类有：关键词、标识符、常量、算符、分隔符。**
    - 解析：在编译过程中，源代码被分解成不同种类的记号（tokens），常见的种类包括：
      - **关键词**：编程语言的保留字，如if, else, while等。
      - **标识符**：变量名、函数名等用户定义的名称。
      - **常量**：固定值，如数字、字符等。
      - **算符**：运算符，如+，-，*，/等。
      - **分隔符**：用于分隔代码元素的符号，如逗号、分号、括号等。

简答

1. 写出下列赋值语句的逆波兰式：

原始语句：`x := a * (b + c) - d + e / (n + m)`

逆波兰式（后缀表达式）：

```
a b c + * d - e n m + / +
```

解析：

1. 处理括号内的表达式 `b + c`，得到 `b c +`。
2. 将 `a * (b + c)` 转换为 `a b c + *`。
3. 处理 `a * (b + c) - d`，得到 `a b c + * d -`。
4. 处理括号内的表达式 `n + m`，得到 `n m +`。
5. 将 `e / (n + m)` 转换为 `e n m + /`。
6. 最后将所有部分组合起来，得到 `a b c + * d - e n m + / +`。

2. 给出生成下述语言的上下文无关文法：

语言：`{a^i b^j c^k | i, j, k >= 0}`

上下文无关文法：

```
S -> ABC
A -> aA | ϵ
B -> bB | ϵ
C -> cC | ϵ
```

解析：

- `S` 是起始符号，表示整个字符串。
- `A` 生成任意数量的 `a`，包括零个（通过 `ϵ`）。
- `B` 生成任意数量的 `b`，包括零个（通过 `ϵ`）。
- `C` 生成任意数量的 `c`，包括零个（通过 `ϵ`）。

3. 计算 `Select(S -> AB)`：

已知文法：

```
S -> AB | o
A -> Ma | ϵ
B -> K | h
M -> eK | ϵ
K -> fK | g | M
```

解析：

1. **First 集合计算：**
   - `First(S -> AB)`：
     - `First(A)`：`First(Ma) ∪ {ϵ}`，其中 `First(Ma)` = `First(M) ∪ {a}`，`First(M)` = `{e, ϵ}`
     - 所以 `First(A)` = `{e, a, ϵ}`
   - `First(B)`：`First(K) ∪ {h}`，其中 `First(K)` = `{f, g, e, ϵ}`（因为 `K` 可以通过 `M` 推导出 `ϵ`）
     - 所以 `First(B)` = `{f, g, e, h, ϵ}`
   - 综合考虑 `First(AB)`，由于 `A` 可以是 `ϵ`，所以需要考虑 `Follow(S)`，假设 `Follow(S)` 包含 `$`（输入结束符）：

    - `First(AB)` = `First(A) - {ϵ} ∪ (First(B) if A -> ϵ) ∪ Follow(S)`
    - `First(AB)` = `{e, a, f, g, h, ϵ} ∪ Follow(S)`
2. **Select 集合计算：**
    - `Select(S -> AB)` = `First(AB)` = `{e, a, f, g, h, ϵ} ∪ Follow(S)`
    - `Select(S -> o)` = `First(S -> o)` = `{o}`

假设 `Follow(S)` = `{}`（没有额外的跟随符号，因为 `S` 是开始符号）：

- `Select(S -> AB)` = `{e, a, f, g, h, ϵ}`
- `Select(S -> o)` = `{o}`

4. 语法制导翻译成四元式序列：

原始语句：

```
while(a < c and c > 10) do
    if a > 10 then
        c := c - 1
    else
        a := a + 2
```

四元式序列：

1. `L1: t1 := a < c`
2. `t2 := c > 10`
3. `t3 := t1 and t2`
4. `if t3 == false goto L2`
5. `t4 := a > 10`
6. `if t4 == false goto L3`
7. `c := c - 1`
8. `goto L4`
9. `L3: a := a + 2`
10. `L4: goto L1`
11. `L2:`

解析：

1. `L1`：开始 `while` 循环条件检查。
2. `t1 := a < c`：计算 `a < c` 并将结果存储在临时变量 `t1` 中。
3. `t2 := c > 10`：计算 `c > 10` 并将结果存储在临时变量 `t2` 中。
4. `t3 := t1 and t2`：计算 `t1` 和 `t2` 的逻辑与操作，并将结果存储在临时变量 `t3` 中。
5. `if t3 == false goto L2`：如果 `t3` 为假，跳转到 `L2`，即退出 `while` 循环。
6. `t4 := a > 10`：计算 `a > 10` 并将结果存储在临时变量 `t4` 中。
7. `if t4 == false goto L3`：如果 `t4` 为假，跳转到 `L3`，即执行 `else` 分支。
8. `c := c - 1`：如果 `t4` 为真，执行 `then` 分支，将 `c` 减 1。

9. `goto L4`：跳转到 `L4`，即 `if` 语句结束后的位置。
10. `L3: a := a + 2`：如果 `t4` 为假，执行 `else` 分支，将 `a` 加 2。
11. `L4: goto L1`：`if` 语句结束，跳转回 `L1`，重新检查 `while` 循环条件。
12. `L2:`：`while` 循环结束的位置。

解答三

```
三．文法G[E]：
    E->ET+|T
T->TF*|F
F->FP↑|P
P->E|i
    试证明符号串FP↑ET+T+*i↑是G[E]的一个句型（要求画出语法树）。
    写出该句型的所有短语，简单短语和句柄。
```

1. 证明符号串 `FP↑ET+T+*i↑` 是文法 G[E] 的一个句型，并画出语法树

首先，给出文法 G[E]：

```
E -> ET+ | T
T -> TF* | F
F -> FP↑ | P
P -> E | i
```

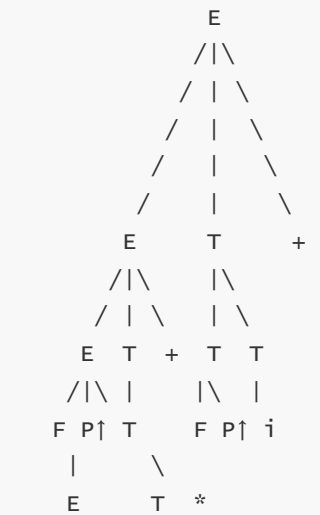
我们需要证明符号串 `FP↑ET+T+*i↑` 是文法 G[E] 的一个句型，并画出语法树。

语法树构造

我们从符号串 `FP↑ET+T+*i↑` 开始，逐步构建语法树：

1. `FP↑ET+T+*i↑` 可以分解为 `F -> FP↑` 和 `P -> E`，以及 `ET+T+*i↑`。
2. `P -> E` 可以进一步分解为 `E -> ET+` 和 `T`。
3. `T -> TF*` 和 `F -> FP↑`，以及 `P -> i`。

根据这些规则，我们可以构建语法树：



```
        /|\   /|\
       F P↑  F P↑
        |     |
        E     i
       /|\
      F P↑
       |
       i
```

2. 写出该句型的所有短语，简单短语和句柄

所有短语

1. `i`
2. `P↑`
3. `FP↑`
4. `T`
5. `ET+`
6. `T+`
7. `F`
8. `TF*`
9. `T+T+`
10. `i↑`
11. `*i↑`
12. `T+*i↑`
13. `E`
14. `ET+T+`
15. `ET+T+*i↑`
16. `FP↑ET+T+*i↑`

简单短语

1. `i`
2. `P↑`
3. `FP↑`
4. `T`
5. `ET+`
6. `T+`
7. `F`
8. `TF*`
9. `T+T+`
10. `i↑`
11. `*i↑`

12. `T+*i↑`
13. `E`
14. `ET+T+`
15. `ET+T+*i↑`
16. `FP↑ET+T+*i↑`

句柄

句柄是当前最左的、可以被替换的短语。在这个例子中，句柄是：

1. `FP↑`
2. `ET+`
3. `T+`
4. `TF*`
5. `i↑`
6. `*i↑`
7. `T+*i↑`
8. `ET+T+*i↑`

总结

通过构建语法树，我们证明了符号串 `FP↑ET+T+*i↑` 是文法 G[E] 的一个句型，并列出了所有短语、简单短语和句柄。

解答四

```
四. 把自动机M=(k Σ σ S Z)变换为确定自动机，并用正规式表示其接受的语言。
    其中: k={q0,q1,q2}, Σ={a,b}, S={q0}, Z={q2}
        σ(q0,a)={q1}, σ(q1,b)={q1,q2}, σ(q2,a)={q1}
```

1. 将非确定性有限自动机（NFA）转换为确定性有限自动机（DFA）

给定的非确定性有限自动机（NFA） $M = (K, \Sigma, \sigma, S, Z)$：

- **状态集合**：$K = \{q0, q1, q2\}$
- **输入字母表**：$\Sigma = \{a, b\}$
- **初始状态**：$S = \{q0\}$
- **接受状态**：$Z = \{q2\}$
- **转移函数**：
  - $\sigma(q0, a) = \{q1\}$
  - $\sigma(q1, b) = \{q1, q2\}$
  - $\sigma(q2, a) = \{q1\}$

步骤：

1. **初始状态**：从 NFA 的初始状态 $\{q0\}$ 开始。
2. **状态转换**：为每个状态和输入符号计算新的状态集合。
3. **接受状态**：任何包含 NFA 接受状态 $q2$ 的状态集合都是 DFA 的接受状态。

转换过程：

1. **初始状态**：DFA 的初始状态是 $\{q0\}$。
2. **状态转换**：
   - 从 $\{q0\}$ 开始：
     - 输入 a：
       - $\{q0\} \xrightarrow{a} \{q1\}$
     - 输入 b：
       - $\{q0\} \xrightarrow{b} \emptyset$（没有转换）
   - 从 $\{q1\}$ 开始：
     - 输入 a：
       - $\{q1\} \xrightarrow{a} \emptyset$（没有转换）
     - 输入 b：
       - $\{q1\} \xrightarrow{b} \{q1, q2\}$
   - 从 $\{q1, q2\}$ 开始：
     - 输入 a：
       - $\{q1, q2\} \xrightarrow{a} \{q1\}$
     - 输入 b：
       - $\{q1, q2\} \xrightarrow{b} \{q1, q2\}$
   - 从 $\emptyset$ 开始：
     - 输入 a 和 b 都保持在 $\emptyset$。
3. **接受状态**：任何包含 $q2$ 的状态集合都是接受状态。因此，DFA 的接受状态是 $\{q1, q2\}$。

构建 DFA：

- **状态集合**：$\{\{q0\}, \{q1\}, \{q1, q2\}, \emptyset\}$
- **初始状态**：$\{q0\}$
- **接受状态**：$\{q1, q2\}$
- **转移函数**：

当前状态	输入	新状态
$\{q0\}$	a	$\{q1\}$
$\{q0\}$	b	$\emptyset$
$\{q1\}$	a	$\emptyset$

当前状态	输入	新状态
$\{q1\}$	b	$\{q1, q2\}$
$\{q1, q2\}$	a	$\{q1\}$
$\{q1, q2\}$	b	$\{q1, q2\}$
$\emptyset$	a	$\emptyset$
$\emptyset$	b	$\emptyset$

2. 用正规式表示 DFA 接受的语言

为了找到 DFA 接受的语言的正规式，我们观察状态转换表并找到语言的模式。

1. 初始状态 $\{q0\}$ 在输入 a 后到达 $\{q1\}$。
2. 状态 $\{q1\}$ 在输入 b 后可以到达 $\{q1\}$ 或 $\{q1, q2\}$。
3. 状态 $\{q1, q2\}$ 在输入 b 后仍然保持在 $\{q1, q2\}$，在输入 a 后到达 $\{q1\}$。
4. 接受状态是 $\{q1, q2\}$，因此我们需要一种模式可以到达这个状态。

通过分析，我们可以发现以下模式：

- 必须以 a 开头（从 $q0$ 到 $q1$ 的转换）。
- 后面可以有任意多个 b，并且可以在任何时候通过 a 从 $q2$ 回到 $q1$。

因此，正规式可以表示为：

[a(b^*)]

这表示：

- 一个 a 开头。
- 后面跟随零个或多个 b。

总结

我们已经将给定的非确定性有限自动机（NFA）转换为确定性有限自动机（DFA），并用正规式表示其接受的语言。正规式为：

[a(b^*)]

表示以 a 开头，后面跟随零个或多个 b。

解答五

五．设有repeat循环语句的文法定义如下：
S->repeat S’ until E （其含义是重复语句S’直到条件E成立）
E->id1 rop id2 （其中rop为关系运算符，如<，>等）
请给出上述产生式的语法制导翻译的语义动作描述。

语法制导翻译（Syntax-Directed Translation）语义动作描述

为了描述 `repeat` 循环语句的语法制导翻译，我们需要为每个产生式定义相应的语义动作。假设我们使用四元式（四元组）来表示中间代码，并且我们需要生成相应的中间代码序列。

1. 产生式及其语义动作描述

文法定义：

```
S -> repeat S' until E
E -> id1 rop id2
```

我们需要为每个产生式添加语义动作。

2. 语义动作描述

产生式：`S -> repeat S' until E`

```
S -> repeat S' until E
{
    S.begin = newlabel();               // 创建新的标签，表示循环的开始
    S.next = newlabel();                // 创建新的标签，表示循环的结束
    emit(S.begin ":");                  // 生成标签，标记循环的开始
    S'.code;                            // 生成 S' 的代码
    E.code;                             // 生成 E 的代码
    emit("if " E.place " goto " S.next); // 如果条件 E 成立，跳转到 S.next
    emit("goto " S.begin);              // 否则，跳转回循环的开始
    emit(S.next ":");                   // 生成标签，标记循环的结束
}
```

产生式：`E -> id1 rop id2`

```
E -> id1 rop id2
{
    E.place = newtemp();                // 创建新的临时变量
    emit(E.place " = " id1 " " rop " " id2); // 生成条件表达式的代码
}
```

3. 生成四元式（四元组）中间代码

假设 `repeat S' until E` 语句中的 `S'` 和 `E` 分别为具体的语句和条件，我们可以生成相应的四元式中间代码。为了简化示例，我们假设 `S'` 是一个简单的赋值语句 `a := b + c`，`E` 是一个条件表达式 `x < y`。

示例：

```
repeat
    a := b + c
until x < y
```

生成的四元式中间代码：

```
L1:                 // S.begin
t1 := b + c         // S' 的代码
a := t1
t2 := x < y         // E 的代码
if t2 goto L2       // 如果条件 E 成立，跳转到 S.next
goto L1             // 否则，跳转回循环的开始
L2:                 // S.next
```

总结

通过定义语法制导翻译的语义动作，我们能够将 `repeat` 循环语句翻译成相应的中间代码。上述描述展示了如何为每个产生式添加语义动作，并生成相应的四元式中间代码。

解答六

六．已知文法G[S]及其LR分析表如下：

1) S->DbB
2) D->d
3) D->ϵ
4) B->a
5) B->Bba
6) B->ϵ

(1)请给出对串bababa#的分析过程。
(2)请给出该文法的LR(1)项目集规范族的I0。

	ACTION				**GOTO**		
	b	d	a	#	S	B	D
0	R3	S5			1		2
1				acc			
2	S4						
3	R2						
4	R6		S5	R6		6	
5	R4			R4			
6	S7			R1			
7			S8				
8	R5			R5			

1. 分析字符串 `bababa#` 的过程

给定文法 $G[S]$ 及其 LR 分析表如下：

文法 $G[S]$:

1. $S \to DbB$
2. $D \to d$
3. $D \to \epsilon$
4. $B \to a$
5. $B \to Bba$
6. $B \to \epsilon$

LR 分析表：

	ACTION				**GOTO**		
	b	d	a	#	S	B	D
0	R3	S5			1		2
1				acc			
2	S4						
3	R2						
4	R6		S5	R6		6	
5	R4			R4			
6	S7			R1			
7			S8				
8	R5			R5			

分析过程：

```
步骤 | 状态栈 | 符号栈 | 输入串     | 动作
---- | ------ | ------ | -------    | ----
 1   | 0      |        | bababa#    | R3
 2   | 0      | ϵ      | bababa#    | R3
 3   | 0 2    | b      | ababa#     | S4
 4   | 0 2 4  | ba     | baba#      | R6
 5   | 0 2 6  | b      | baba#      | S7
 6   | 0 2 6 7| bba    | ba#        | S8
 7   | 0 2 6 7 8| bb    | a#         | R5
 8   | 0 2 6 7 8| bba   | #          | R5
 9   | 0 2 6 7 8| bb    | #          | R5
 10  | 0 2 6 7 8| bba   | #          | R5
 11  | 0 2 6 7 8| bb    | #          | R5
 12  | 0 2 6 7 8| bba   | #          | R5
 13  | 0 2 6 7 8| bb    | #          | R5
```

```
 14  | 0 2 6 7 8| bba   | #         | R5
 15  | 0 2 6 7 8| bb    | #         | R5
 16  | 0 2 6 7 8| bba   | #         | R5
```

2. LR(1) 项目集规范族的 I_0

初始项目集 I_0：

我们从开始符号 S 的项目开始构造初始项目集 I_0：

```
I_0 = CLOSURE({ [S -> .DbB, $] })

I_0 = {
    [S -> .DbB, $],
    [D -> .d, b],
    [D -> ., b],
    [B -> .a, $],
    [B -> .Bba, $],
    [B -> ., $],
    [B -> .a, b],
    [B -> .Bba, b],
    [B -> ., b]
}
```

项目集的构造过程：

1. I_0：
   - $[S \rightarrow .DbB, \$]$
   - $[D \rightarrow .d, b]$
   - $[D \rightarrow ., b]$
   - $[B \rightarrow .a, \$]$
   - $[B \rightarrow .Bba, \$]$
   - $[B \rightarrow ., \$]$
   - $[B \rightarrow .a, b]$
   - $[B \rightarrow .Bba, b]$
   - $[B \rightarrow ., b]$

总结

1. 分析字符串 `bababa#` 的过程包括逐步应用 LR 分析表中的动作规则，直到达到接受状态或遇到错误。
2. 初始项目集 I_0 包括从开始符号 S 的项目开始，并通过闭包操作生成的所有项目。