

1、实验目的与要求

一、实验目的：

本实验旨在通过设计和实现一个程序，对给定的文法文件进行读取和分类，加深对高级语言文法结构的理解，并掌握如何使用编程语言来实现文法的定义和分类。通过本实验，学生将能够：

- ①、理解和掌握文法四元组 (V, T, P, S) 的定义和文法文件的组织形式
- ②、设计和实现一个程序，能够读取文法文件并解析其中的文法规则；
- ③、理解和掌握 Chomsky 文法体系，并能够编写程序自动判断文法的类型；

二、实验要求：

- ①、使用 C、C++、或 Java 完成任务一、二的程序编写；

2、实验内容

文法（Grammar）是描述高级语言语法结构的重要工具。定义任意的文法 G ，需要完成对其四元组 (V, T, P, S) 的定义（课本 P33）。在该实验中，请制定文法文件的具体组织形式、编程完成对文法文件的读取、并完成对文法的分类。该实验具体包含以下两个任务：

• 任务一：文法的定义及读取

现规定文法由 Grammar.txt 文件保存，请制定文法文件的具体存储格式。如文法 $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S \}$ 在 Grammar.txt 文件中可由以下方式描述并存储：

```
-----  
S,A,B,C  
a,b,c  
S->ABC,A->a,B->b,C->c  
S  
-----
```

文法的文本形式可根据自己需要自由定义，在此基础上，编程实现对任意文法文件的读取。

• 任务二：文法的分类

根据 Chomsky 的文法体系分类（课本 P40），文法分为四大种类。请在任务一的基础上，编程实现对 Grammar.txt 中存储的文法进行分类，自动判断其所属类别。例如任务一中所给出的文法 G 应被判定为 2 型文法，即上下文无关文法。请设计分类方法，并设计四类不同的测试文法测试分类结果的正确性。

3、实验过程步骤及说明

本次实验使用的是 Java 语言完成任务一、二的程序编写。

一、任务一：文法的定义及读取

$G = \{\{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S\}$ 文法由 Grammar.txt 文件保存。如下图所示：

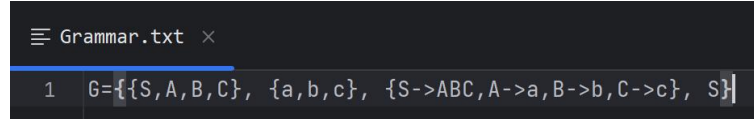
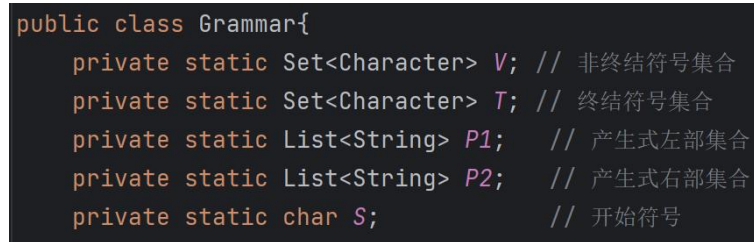
A screenshot of a text editor window titled 'Grammar.txt'. The file contains a single line of text: `G={{S,A,B,C}, {a,b,c}, {S->ABC,A->a,B->b,C->c}, S}`.

图 1 文法的初始保存格式

1、Grammar 类的设计

首先编写一个 Grammar 类，包含五个类成员变量，用于表示文法的四个结构 V、T、P、S，即文法的非终结符号集合、终结符号集合、产生式集合和开始符号。

如下图所示，其中产生式用两个集合存储，一个存储产生式左部、一个存储产生式右部，以便实现任务二中对文法的分类。

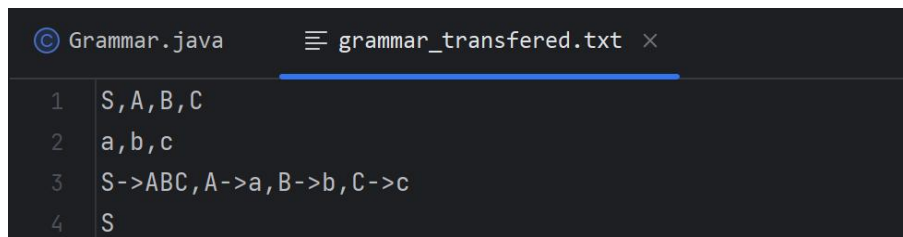
A screenshot of a Java code editor showing the Grammar class definition. The code is as follows:

```
public class Grammar{
    private static Set<Character> V; // 非终结符号集合
    private static Set<Character> T; // 终结符号集合
    private static List<String> P1; // 产生式左部集合
    private static List<String> P2; // 产生式右部集合
    private static char S; // 开始符号
```

图 2 Grammar 类的成员变量

2、文法格式转换

为了更好地读取文法各部分内容，先设计一个 convertGrammarToFile 函数，读取原始文法文件的每一行，并将每一行的内容转换成适合程序处理的格式。转换后的文法如下图所示：

A screenshot of a text editor window titled 'grammar_transferred.txt'. The file contains four lines of text:

```
1 S,A,B,C
2 a,b,c
3 S->ABC,A->a,B->b,C->c
4 S
```

图 3 便于读取的文法存储格式

下面介绍具体的实现过程：

①、根据源文件的路径读取文法数据并存储在 data 列表中

convertGrammarToFile 函数包含两个参数，inputFilePath——原始文法文件的路径，outputFilePath——输出文件的路径。首先根据源文件的路径读取文法数据并存储在 data 列表中，如下图所示。

```
// 转换文法格式
public static void convertGrammarToFile(String inputFilePath, String outputFilePath) {
    List<String> data = new ArrayList<>();
    String s;
    int lineNum = 0;

    // 首先根据源文件的路径读取文法数据并存储在data列表中
    try (BufferedReader br = new BufferedReader(new FileReader(inputFilePath))) {
        while ((s = br.readLine()) != null) {
            data.add(s);
            lineNum++;
        }
    } catch (IOException e) {
        System.err.println("Error: Unable to open file!");
        System.exit(status: 1);
    }
}
```

图 4 根据源文件的路径读取文法数据并存储在 data 列表中

②、将转换后的文法内容写入新文件

A、截取最外层大括号内的内容

使用循环遍历文法数据列表 data 中的每一行文法规则。对于每一行文法规则，创建一个长度为 4 的字符串数组 Grammar，用于存储文法规则的四个部分。然后获取当前行的文法规则，并找到左花括号的位置 pos，接着使用 substring 方法截取左右花括号之间的内容，并赋值给 center。

以 $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S \}$ 为例子，截取的内容为 $\{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S$ 。

```
// 转换文法格式并写入新文件
try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFilePath))) {
    for (int i = 0; i < lineNum; i++) {
        String[] Grammar = new String[4]; // 用于存储文法规则的数组，每个规则最多包含四部分
        s = data.get(i); // 获取当前行的文法规则
        int pos = s.indexOf('{'); // 查找左花括号的位置
        int length = s.length(); // 获取当前行的长度
        String center = s.substring(pos + 1, length - 1); // 截取左右花括号之间的内容
    }
}
```

图 5 截取大括号内的内容

B、分割 V、T、P、S 四个部分内容

对于当前行的文法规则，使用循环遍历其中的四个部分。

在每一次循环中，查找右花括号的位置 pos，接着进行判断：

- 如果找不到右花括号（即开始符号），则将剩余部分作为最后一个部分，并加上换行符，然后将该部分写入文件。
- 如果找到了右花括号，则截取当前部分并加上换行符，并更新 center，去掉已经处理的部分及其后的两个字符，然后将该部分写入文件。

对应代码如下所示：

```

for (int j = 0; j < 4; j++) { // 遍历文法规则的四个部分
    pos = center.indexOf('}'); // 查找右花括号的位置
    if (pos < 0) { // 如果没有找到右花括号，即开始符号部分
        // 则将剩余部分作为最后一个部分，并加上换行符
        Grammar[j] = center.substring(0, 1) + "\n";
        writer.write(Grammar[j]); // 将当前部分写入文件
        //System.out.println(Grammar[j]);
        continue; // 继续处理下一个文法规则
    }
    // 截取当前部分并加上换行符
    Grammar[j] = center.substring(1, pos) + "\n";
    // 更新中心部分，去掉已经处理的部分及其后的两个字符
    center = center.substring(beginIndex: pos + 3);
    writer.write(Grammar[j]); // 将当前部分写入文件
    //System.out.println(Grammar[j]);
}
}

```

图 6 分割 V、T、P、S 四个部分内容

C、文法格式转换测试

在主方法中，创建一个 Grammar 实例，调用 convertGrammarToFile 函数进行文法格式的转换，代码如下所示：

```

public static void main(String[] args) {
    // 创建 Grammar 实例
    Grammar grammar = new Grammar();
    // 转换文法格式
    convertGrammarToFile(inputFilePath: "Grammar.txt", outputFilePath: "grammar_transferred.txt");
}

```

图 7 文法格式转换测试

```

1 G={S,A,B,C}, {a,b,c}, {S->ABC,A->a,B->b,C->c}, S

```

图 8 源文件

```

1 S,A,B,C
2 a,b,c
3 S->ABC,A->a,B->b,C->c
4 S

```

图 9 转换后文件

3、文法内容读取

对转换后的文件内容进行读取，进一步分割出文法四个部分的内容。

①、读取转换后文件的内容

创建一个空的字符串列表 myFile，用于存储从文件中读取的文本行。通过 BufferedReader 逐行读取文件内容，直到文件末尾，每读取一行文本内容，将其添加到 myFile 列表中。最后调用 extractGrammarData(myFile)方法，将读取的文法数据传递给该方法进行进一步处理。

代码如下所示：

```
// 读取转换后的文法数据并返回列表
private static void readGrammarFromFile(String filePath) {
    List<String> myFile = new ArrayList<>();
    try (BufferedReader infile = new BufferedReader(new FileReader(filePath))) {
        String line;
        while ((line = infile.readLine()) != null) {
            myFile.add(line);
        }
    } catch (IOException e) {
        System.err.println("Error: Unable to open file!");
        e.printStackTrace();
        return;
    }
    extractGrammarData(myFile);
}
```

图 10 读取转换后文件的内容到 myFile 列表

②、具体分割文法各个部分的内容

根据转换后的文法格式，可知 V、T、P、S 分别对应文件中第一、二、三、四行的内容。下面介绍具体的分割方式：

A、提取 V（非终结符号集合）、T（终结符号集合）

提取 V、T 元素的思路类似，因为他们的元素之间都用逗号分割开，因此只需要遍历该行内容，逐个字符判断是否为逗号，如果不是逗号，则将其添加到对应集合中。其中非终结符号对应第一行的内容，终结符号对应第二行的内容。

代码如下图所示：

```
// 提取文法数据中的V、T、P、S
private static void extractGrammarData(List<String> myFile) {
    String s;
    // 提取V，获取第一行的内容并去掉逗号即可
    s = myFile.get(0);
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == ',') continue;
        V.add(s.charAt(i));
    }
    // 提取T，获取第二行的内容并去掉逗号即可
    s = myFile.get(1);
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == ',') continue;
        T.add(s.charAt(i));
    }
}
```

图 11 提取 V（非终结符号集合）、T（终结符号集合）的代码

B、提取产生式的左部集合（P1）和右部集合(P2)

为了便于后续对文法的分类判断，这里需要将产生式的左右部分离开来，即考虑右部有候选式的情况。

首先从 myFile 列表中获取第三行文本内容，即产生式集合。然后使用循环处理该行内容，每次处理一个产生式，直至该行为空。

在每次循环中，首先查找逗号首次出现的位置，以分割当前需要处理的产生式。对

于当前产生式，检查是否存在箭头符号 ("→")，如果存在，则将产生式左部和右部分别提取出来。如果右部存在多个候选项（用竖线"|"分隔），则将其分割并分别存储。

最后，将每个左部和右部对应的产生式添加到 P1 和 P2 列表中。

代码如下图所示

```
// 提取P1和P2，获取第三行的内容
s = myFile.get(2);
while (!s.isEmpty()) {
    // 查找逗号的位置，以分割产生式
    int pos = s.indexOf(','); // 查找逗号的位置
    String term = pos >= 0 ? s.substring(0, pos) : s; // 截取产生式
    s = pos >= 0 ? s.substring(beginIndex: pos + 1) : ""; // 更新s
    // 提取产生式左部和右部
    int index_of_arrow = term.indexOf("→"); // 查找箭头的位置
    if (index_of_arrow != -1) { // 如果找到了箭头
        String alpha = term.substring(0, index_of_arrow); // 提取产生式左部
        String beta = term.substring(beginIndex: index_of_arrow + 2); // 提取产生式右部
        String[] betas = beta.split(regex: "\\|"); // 以竖线分割候选式
        for (String beta1 : betas) { // 将左部和右部的候选式添加到P1和P2中
            P1.add(alpha);
            P2.add(beta1);
        }
    }
}
```

图 12 提取产生式的左部集合（P1）和右部集合(P2)

C、提取开始符号 S

从 myFile 列表中获取第四行文本内容，即开始符号。因为开始符号只有一个，所以提取第一个字符即可。代码如下图所示：

```
// 提取开始符号
S = myFile.get(3).charAt(0);
```

图 13 提取开始符号 S

D、打印文法的各个部分内容

编写一个 printGrammar 方法，输出文法各个部分内容。其中产生式需要用 “→” 符号将左右部组合起来。对应的代码如下图所示：

```
// 输出文法各个部分内容
public void printGrammar() {
    System.out.println("Non-terminal symbols (V): \n" + V);
    System.out.println("Terminal symbols (T): \n" + T);
    System.out.println("Production rules (P): ");
    for (int i = 0; i < P1.size(); i++) {
        System.out.println(P1.get(i) + "→" + P2.get(i));
    }
    System.out.println("Start symbol (S): \n" + S);
}
```

图 14 打印文法的各个部分内容

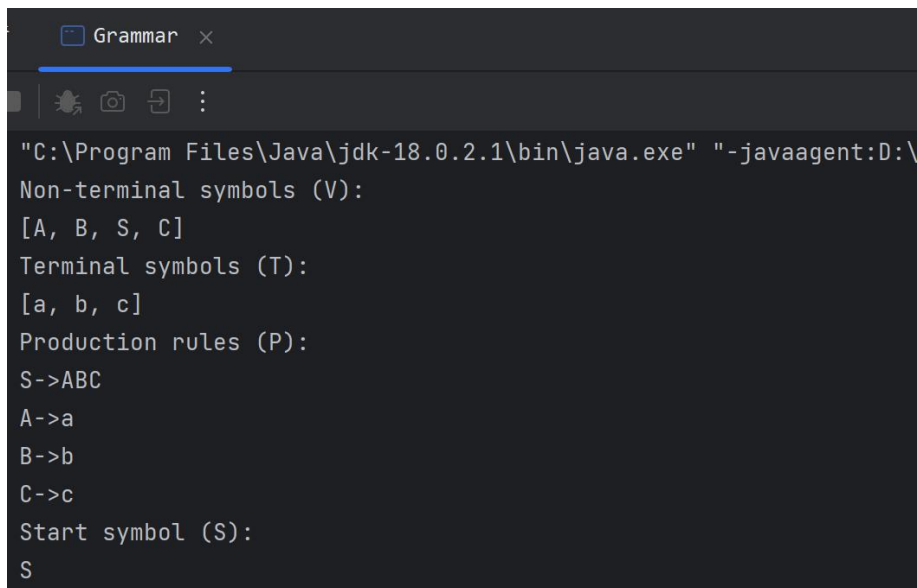
E、读取文法内容测试：

主方法在转换文法格式的基础上，调用 `readGrammarFromFile` 函数读取转换后的文法数据，再调用 `printGrammar` 函数打印文法各个部分的内容，代码如下图所示：

```
// 读取转换后的文法数据
readGrammarFromFile( filePath: "grammar_transferred.txt");
// 输出文法内容
grammar.printGrammar();
```

图 15 读取文法内容测试代码

输出结果如下图所示，可以正确实现对文法文件的读取。



```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" "-javaagent:D:\
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->ABC
A->a
B->b
C->c
Start symbol (S):
S
```

图 16 读取文法后的输出结果

综上，通过文法格式转换、读取转换后文件并提取文法各个部分内容，成功实现对文法文件的读取。同时，如果文法直接以转换后的形式存储，那么可以直接调用读取方法进行读取，不需要进行转换。因此通过上述程序，实际上实现了对两种存储方式的文法的读取。

二、任务二：文法的分类

1、编写判断文法类型的代码：

我编写了一个 `checkGrammarType` 方法用于确认某个文法具体属于哪种类型。

```
// 判断文法类型
private static int checkGrammarType() {
    int one_of_type = 4; // 初始值为4，代表未确定类型
    int type = 4; // 文法类型，默认为4，表示未确定类型
    int is_left = 0; // 左线性文法标记，0表示不是左线性文法，1表示是左线性文法
    int is_right = 0; // 右线性文法标记，0表示不是右线性文法，1表示是右线性文法
```

图 17 `checkGrammarType` 函数代码片段

如上图所示，首先定义四个变量，其中 `one_of_type` 用于表示当前产生式符合的文法类型，`type` 表示文法类型，这两个变量都初始为 4 表示未确认。`is_left` 和 `is_right` 变量

用于 3 型文法的判断，检测是否出现左右线性文法混用的情况。

然后循环遍历每个产生式，进行文法的判断。其中 P1 存储产生式左部，P2 存储产生式右部。每次循环初始需要更新文法类型，其中如果 one_of_type 小于 type，则更新 type。

```
// 遍历产生式列表
for (int it = 0; it < P1.size(); it++) {
    if (one_of_type < type) { // 更新文法类型
        type = one_of_type;
    }
    String alpha = P1.get(it); // 获取产生式左部
    String beta = P2.get(it); // 获取产生式右部
}
```

图 18 使用循环遍历产生式列表

、0 型文法：

定义：设 $G=(V,T,P,S)$ ，如果它的每个产生式 $\alpha \rightarrow \beta$ 是这样一种结构： $\alpha \in (V \cup T)^+$ 且至少含有一个非终结符，而 $\beta \in (V \cup T)^*$ ，则 G 是一个 0 型文法。

0 型文法也称短语文法。一个非常重要的理论结果是：0 型文法的能力相当于图灵机(Turing)。或者说，任何 0 型文语言都是递归可枚举的，反之，递归可枚举集必定是一个 0 型语言。0 型文法是这几类文法中，限制最少的一个。

根据 0 型文法的定义，首先检查左部 α 是否只包含终结符和非终结符，若有非法符号则标记为无效。代码如下图所示：

```
// 判断0型文法，短语结构文法
// 检查左部α是否属于VUT的集合
boolean validAlpha = true; // 标记左部α是否有效
for (int i = 0; i < alpha.length(); i++) {
    char symbol = alpha.charAt(i);
    if (!V.contains(symbol) && !T.contains(symbol)) {
        validAlpha = false; // 有非法符号，标记为无效
        break;
    }
}
```

图 19 检查左部 α 是否只包含终结符和非终结符

接着判断左部 α 是否至少含有一个非终结符，若有则更新标记符号 has_Vp 为 true。代码如下图所示：

```
//检查左部α是否至少含有一个非终结符
boolean has_Vp = false;
for (int i = 0; i < alpha.length(); i++) {
    if (V.contains(alpha.charAt(i))) {
        has_Vp = true; // 含有非终结符
        break;
    }
}
```

图 20 判断左部 α 是否至少含有一个非终结符

检查右部 β 是否只包含终结符和非终结符，若有非法符号则标记为无效。这里需要考虑右部含有空串的情况，空串不属于非法符号。

```
// 检查右部 $\beta$ 是否属于VUT的集合
boolean validBeta = true; // 标记右部 $\beta$ 是否有效
for (int i = 0; i < beta.length(); i++) {
    char symbol = beta.charAt(i);
    if (!V.contains(symbol) && !T.contains(symbol)) {
        if (symbol == '\epsilon'){ continue; } // 空串
        validBeta = false; // 有非法符号，标记为无效
        break;
    }
}
```

图 21 检查右部符号是否有效

最后根据以上三次判断，如果有一个不符合 0 型文法的定义，则不符合 0 型文法，退出整个循环，更新 type 为-1。如果符合 0 型文法的定义，则将 one_of_type 更新为 0。

```
// 如果不符合0型文法定义
if (!has_Vp || !validAlpha || !validBeta) {
    type = -1;
    break;
}
one_of_type = 0;
```

图 22 综合完成 0 型文法的判断

- 1 型文法:

1 型文法也叫上下文有关文法，此文法对应于线性有界自动机。

它是在 0 型文法的基础上每一个 $\alpha \rightarrow \beta$, 都有 $|\beta| \geq |\alpha|$ ($\alpha \rightarrow \epsilon$ 除外)。这里的 $|\beta|$ 表示的是 β 的长度。

如有 $A \rightarrow Ba$ 则 $|\beta|=2, |\alpha|=1$ 符合 1 型文法要求。反之,如 $aA \rightarrow a$, 则不符合 1 型文法。

根据 1 型文法的定义，首先计算右部除去空串 ϵ 的长度，如果左部 α 的长度小于等于右部 β 的长度或者 β 为空串，则将 one_of_type 设置为 1 型文法。如果不符合要求，则结束本次循环，不进行 2 型、3 型文法的判断，直接对下一产生式进行判断。

```
// 判断1型文法，上下文有关文法，即右部的字符数要大于或等于左部的字符数
int betaLength = 0; // 右部字符数(除去空串)
for (int i = 0; i < beta.length(); i++) {
    char symbol = beta.charAt(i);
    if (symbol != '\epsilon') { // 如果字符不是 $\epsilon$ 
        betaLength++;
    }
}
if (alpha.length() <= betaLength || betaLength == 0){
    one_of_type = 1;
} else { continue; }
```

图 23 1 型文法的判断

- 2 型文法:

2 型文法也叫上下文无关文法，它对应于下推自动机。

如果对于任意 $\alpha \rightarrow \beta \in P$ ，均有 $|\beta| \geq |\alpha|$ ，并且 $\alpha \in V$ 成立，则称 G 为 2 型文法

即 2 型文法是在 1 型文法的基础上,再满足：每一个 $\alpha \rightarrow \beta$ 都有 α 是非终结符。

如 $A \rightarrow Ba$,符合 2 型文法要求。而 $Ab \rightarrow Bab$ 虽然符合 1 型文法要求,但不符合 2 型文法要求，因为其 $\alpha = Ab$ ，而 Ab 不是一个非终结符。

判断 2 型文法的代码如下图所示，只需要在 1 型文法的基础上，检查左部 α 是否只有一个非终结符，若符合则将 `one_of_type` 设置为 2 型文法。

```
// 判断2型文法，上下文无关文法，即左部只有一个非终结符
if (alpha.length() == 1 && V.contains(alpha.charAt(0))) {
    one_of_type = 2;
} else { continue; }
```

图 24 2 型文法的的判断

- 3 型文法

3 型文法也叫正规文法，它对应于有限状态自动机。

它是在 2 型文法的基础上满足: $A \rightarrow \alpha | \alpha B$ （右线性）或 $A \rightarrow \alpha | B \alpha$ （左线性），其中 $A, B \in V$ ， $\alpha \in T$ 或为空串。

这里需要特别注意的是左、右线性文法不可混用。

根据 3 型文法的定义，判断：

若右部 β 只有一个非终结符或者只有空串，则将 `one_of_type` 设置为 3 型文法。

若右部 β 有两个符号，并且一个是非终结符，一个是终结符或空串，则判断是左线性文法还是右线性文法，并标记对应的标志位。

其中如果循环过程中出现左右线性标志位同时为 1 的情况，说明出现了左右线性文法混用，不符合 3 型文法的定义。所以只有当其中一个线性标志位为 1 另一个线性标志位为 0 的情况才符合 3 型文法的定义。代码如下图所示：

```
// 判断3型文法，正规文法，符合左线性文法或右线性文法即可
// 右部只有一个非终结符或空串
if (beta.length() == 1 && (T.contains(beta.charAt(0)) || beta.charAt(0) == 'ε')) {
    one_of_type = 3;
}
// 右部有两个符号，并且一个是非终结符，一个是终结符或空串
else if (beta.length() == 2){
    // 右线性文法
    if((T.contains(beta.charAt(0)) || beta.charAt(0) == 'ε') && V.contains(beta.charAt(1))) {
        is_right = 1;
        if(is_left == 0){ // 左、右线性文法不可混用
            one_of_type = 3;
        }
    }
    // 左线性文法
    else if (V.contains(beta.charAt(0)) && (T.contains(beta.charAt(1)) || beta.charAt(1) == 'ε')) {
        is_left = 1;
        if(is_right == 0){ // 左、右线性文法不可混用
            one_of_type = 3;
        }
    }
}
```

图 25 3 型文法的判断

循环遍历每个产生式结束后，再次根据 `one_of_type` 和 `type` 的大小关系更新 `type` 并返回。

```
if (one_of_type < type) { // 最后再一次更新文法类型
    type = one_of_type;
}
return type;
```

图 26 最后更新并返回文法类型

最后编写一个 `analyzeGrammarType` 方法输出文法类型。它调用 `checkGrammarType` 方法获取文法的类型 `type`，然后根据 `type` 输出文法的类型。其中如果 `type` 是 -1，则表明文法不是正规文法。

```
// 分析文法类型并输出结果
private static void analyzeGrammarType() {
    int grammarType = checkGrammarType();
    if (grammarType == -1) {
        System.out.println("该文法不是正规文法");
    } else {
        System.out.println("该文法为 " + grammarType + " 型文法");
    }
}
```

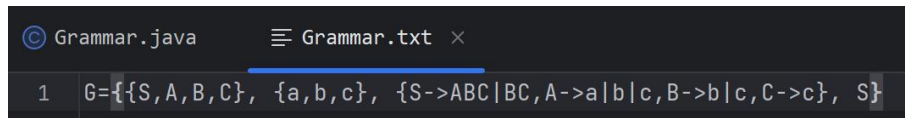
图 27 输出文法的类型

2、测试程序能否正确判断文法类型

①、特殊情况测试（右部有多项和空产生式）

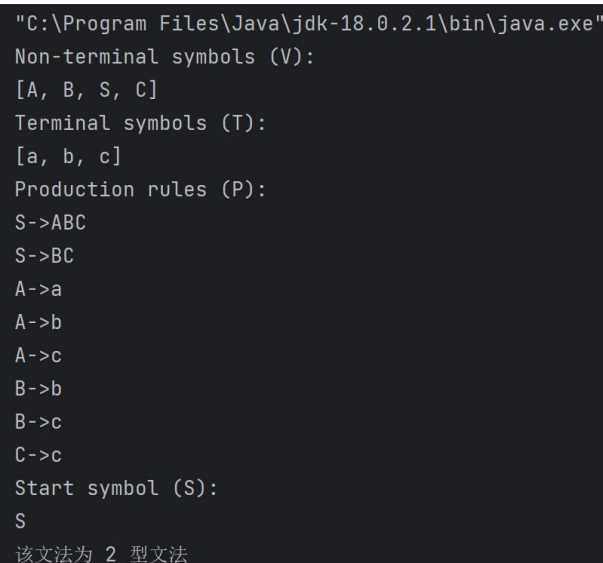
A、右部为候选式

如下图所示，测试样例右部为候选式，测试对该文件的读取和文法类型判断。



```
Grammar.java  Grammar.txt x
1  G = { {S, A, B, C}, {a, b, c}, {S -> ABC | BC, A -> a | b | c, B -> b | c, C -> c}, S }
```

图 28 右部为候选式测试



```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->ABC
S->BC
A->a
A->b
A->c
B->b
B->c
C->c
Start symbol (S):
S
该文法为 2 型文法
```

图 29 右部为候选式测试结果

结果上图所示，可以正确提取所有产生式的左部和右部，对于候选式，将拆分为多个产生式，如 $B \rightarrow b|c$ 拆分为 $B \rightarrow b$ 和 $B \rightarrow c$ 。同时可以正确判断文法的类型。

B、测试空产生式

如下图所示，测试样例中有空产生式，测试对该文件的读取和文法类型判断。

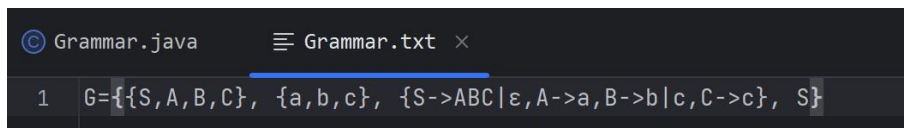


图 30 测试空产生式

结果如下图所示，可以正确提取所有产生式的左部和右部并正确判断文法类型。

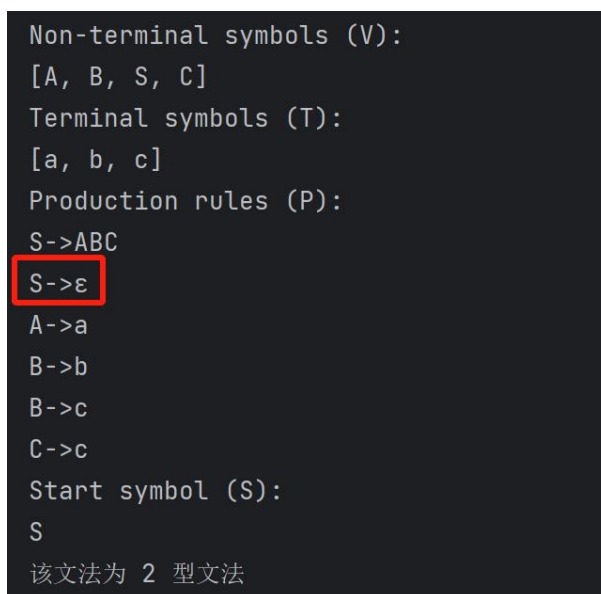


图 31 测试空产生式结果

②、测试 0 型文法

如下图所示， $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC | \epsilon, AB \rightarrow a, B \rightarrow b | c, C \rightarrow c\}, S \}$ 为一个 0 型文法，因为 $AB \rightarrow a$ 不符合 1 型文法的定义。

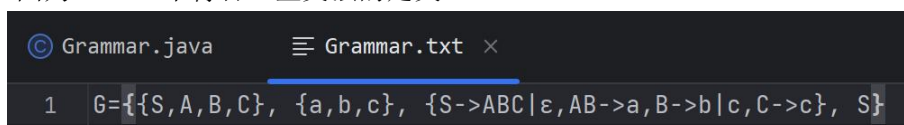


图 32 0 型文法样例

测试结果如下，可以正确提取所有产生式的左部和右部并正确判断文法类型。

```

"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->ABC
S->ε
AB->a
B->b
B->c
C->c
Start symbol (S):
S
该文法为 0 型文法

```

图 33 0 型文法样例测试结果

③、测试 1 型文法

如下图所示， $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC | \epsilon, A \rightarrow a | AB, AB \rightarrow ab, B \rightarrow b | c, C \rightarrow c\}, S \}$ ， S 为一个 1 型文法，因为 $AB \rightarrow ab$ 不符合 2 型文法的定义。

```

Grammar.java  Grammar.txt x
1  G = { {S, A, B, C}, {a, b, c}, {S -> ABC | ε, A -> a | AB, AB -> ab, B -> b | c, C -> c}, S }

```

图 34 1 型文法样例

测试结果如下，可以正确提取所有产生式的左部和右部并正确判断文法类型。

```

"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->ABC
S->ε
A->a
A->AB
AB->ab
B->b
B->c
C->c
Start symbol (S):
S
该文法为 1 型文法

```

图 35 1 型文法样例测试结果

④、测试 2 型文法

2 型文法使用的样例是 $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow ABC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S \}$ ，该文法符合 2 型文法的定义，但是 $S \rightarrow ABC$ 不符合 3 型文法的定义。

```
Grammar.java Grammar.txt x
1 G={{S,A,B,C}, {a,b,c}, {S->ABC,A->a,B->b,C->c}, S}
```

图 36 2 型文法测试样例

测试结果如下，可以正确提取所有产生式的左部和右部并正确判断文法类型。

```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->ABC
A->a
B->b
C->c
Start symbol (S):
S
该文法为 2 型文法
```

图 37 2 型文法样例测试结果

⑤、测试 3 型文法

3 型文法测试样例为 $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow aA \mid bB \mid cC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S \}$ ，它符合 3 型文法的所有要求。

```
Grammar.java Grammar.txt x
1 G={{S,A,B,C}, {a,b,c}, {S->aA|bB|cC,A->a,B->b,C->c}, S}
```

图 38 3 型文法测试样例

测试结果如下，可以正确提取所有产生式的左部和右部并正确判断文法类型。

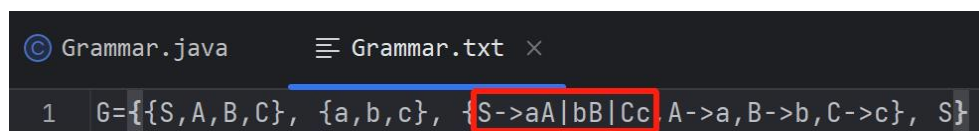
```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->aA
S->bB
S->cC
A->a
B->b
C->c
Start symbol (S):
S
该文法为 3 型文法
```

图 39 3 型文法样例测试结果

⑥、左右线性文法混用情况测试

根据 3 型文法的定义，不可以混用左、右线性文法。混用时只符合 2 型文法的定义。例如 $G = \{ \{S, A, B, C\}, \{a, b, c\}, \{S \rightarrow aA \mid bB \mid cC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}, S \}$ ，其中 $S \rightarrow aA$ 和 $S \rightarrow bB$ 属

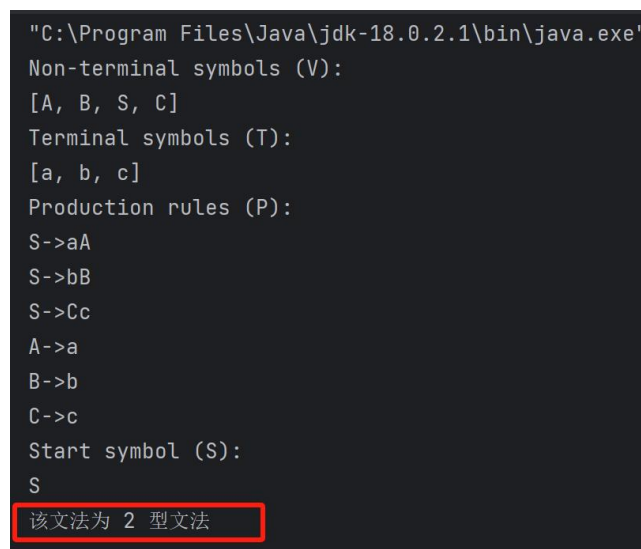
于右线性文法，而 $S \rightarrow Cc$ 属于左线性文法。



```
Grammar.java Grammar.txt x
1 G = ({S, A, B, C}, {a, b, c}, {S -> aA | bB | Cc}, A -> a, B -> b, C -> c}, S)
```

图 40 左右线性文法混用样例测试

结果如下图所示，可以正确判断左、右线性文法混用的情况。



```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe"
Non-terminal symbols (V):
[A, B, S, C]
Terminal symbols (T):
[a, b, c]
Production rules (P):
S->aA
S->bB
S->Cc
A->a
B->b
C->c
Start symbol (S):
S
该文法为 2 型文法
```

图 41 左右线性文法混用样例测试结果

4、实验总结：

在本次实验中，我设计并实现了一个程序，用于读取给定的文法文件并对其进行分类。首先定义文法的四元组 (V, T, P, S) ，然后编写了程序来解析给定的文法文件，提取其中的文法规则。接着，我还实现了文法分类的功能，根据 Chomsky 的文法体系将文法分为四种类型：0 型、1 型、2 型和 3 型。通过逐个检查文法规则的特征，我设计的程序能够准确地确定给定文法的类型。其中，我还考虑到了空产生式的情况以产生式右部为候选式的情况，充分考虑了各种可能性，成功实现对任意文法的读取和分类。

最后，我进行了一系列的测试，充分验证了程序的正确性和可靠性。综上，我很好地完成了任务一和任务二的要求。

5、实验心得体会

通过本次实验，我收获了以下几点经验和体会：

①、**对文法结构的理解加深：**通过设计和实现程序，我更深入地理解了文法的结构和分类，包括非终结符、终结符、产生式等概念，以及 Chomsky 的文法体系。

②、**编程能力的提升：**通过编写程序来解析文法文件并进行分类判断，我加强了编程能力，特别是对文件操作、字符串处理、逻辑判断等方面的掌握。

③、**对文法分类算法的理解：**通过实现文法分类算法，我深入了解了不同类型文法的特征和区别，进一步提高了对编译原理相关知识的理解和掌握。

④、**本次实验遇到的困难与解决：**进行编程实现文法分类过程中，如何处理空产生式是一个重难点，很考验逻辑性，在不断调试思考后，我成功完成了对各种情况的处理，使得程序能够准确识别各种类型的文法。

⑤、**巩固文法基础知识：**在上学期我已经学习了《形式语言与自动机》这个课程，课程中已经学习过 Chomsky 的文法体系等基本概念，在本次实验中我编程实现了文法的识别和分类程序，在这个过程中，我巩固了已学知识，对文法相关知识概念有了更加深入地理解和掌握。