语法分析器使用说明

`JAVA` `LL1` 自顶向下

1 语法分析器的完整设计

核心为 **class LL1**

> 基于LL1文法分析，包括读取文法，识别文法中的终结符和非终结符。再基于文法求出某个字符串first集合、字符串组的first集合、某一个字符串follow集合。根据follow集和first集合制作语法分析预测表。最后读取待分析的内容，先使用词法分析获得字符串和相应类型，根据类型进行自定向下的词法分析。
>
> `PS` LL1 分析难以解决 if else 语句冲突

`注` 默认采用 $ 表示**空字符串**

代码已上传到github

1.1 读取文法并获得所有表达式

按行读取文法，每一行是一个产生式，结构为X1 -> Y1 Y2 ... Yn。每个产生式用空格分割

使用map来记录文法。

- key为非终结符
- value为这个终结符对应的所有产生式数组。

```
private TreeMap<String, ArrayList<String>> expression = new TreeMap<>(); //
记录文法

// 获得所有表达式
    void initExpress() {
        for (String line : grammer) { // 按行处理
            String[] strs = line.split(" -> ");
```

```
            ArrayList<String> exp;
            if (expression.get(strs[0]) == null) {
                // 没有存过这个非终结符的表达式
                exp = new ArrayList<String>() {
                    {
                        add(strs[1]);
                    }
                };
            } else { // 存过就需要把之前的拿出来
                exp = expression.get(strs[0]);
                exp.add(strs[1]);
            }
            expression.put(strs[0], exp);
        }
    }
```

1.2 first集合

1. `void firstAll()` 最外层遍历 expression，对每一个非终结符求 firs
2. `private Set<String> firstExps(String left)` 针对每个非终结符，遍历以其为开头的所有产生式，按如下规则求 first 集合

> 对每一个文法符号X∈VT UVN 构造FIRST(X): 应用下列规则,直到每 个集合FIRST不再增大为止.
>
> (1)如果X∈VT,则FIRST(X)={X}. (2)如果X∈VN,且有产生式X→a...,则把a加入到FIRST(X)中;若X→$也是一个
>
> 产生式, 则把$加入到FIRST(X)中. (3)如果X→Y...是一个产生式且Y∈VN, 则把FIRST(Y)\{$}加到FIRST(X)中;
>
> 如果X→Y Y ...Y 是一个产生式, Y ,...,Y ∈V ,而且对任何j, j∈[1, i-1], 12k * 1i-1N
>
> $∈FIRST (Yj), (即Y1Y2...Yi-1=>$) , 则把FIRST(Yj)\{$}加到FIRST(X)中; 特别 是,若所有的FIRST(Yj)均含有$, j=1,2,...,k, 则把$加到FIRST(X)中.

3. `private void firstOneExp(String left, String exp, int i)` 针对每个产生式分析求解非终结符仔这个产生式下的 first 集合
4. 遇到依赖情况，即这个非终结符号的first集合依赖于另外一个非终结符，就采用递归调用2的函数
5. 使用 hasrecord 记录一个非终结符是否已经完整被求出 first 集合

```
//  获得非终结符的所有first集合
    private Set<String> firstExps(String left) {
        // 从文法右边的第一个字符开始看
        ArrayList<String> strs = expression.get(left); // 用空格分割
        for (String exp : strs) {
            firstOneExp(left, exp, 0);
        }
        hasrecord.put(left, true);
```

```
        return first.get(left);
    }

    // 获得右边的产生式第一个字符串为终结符,left左边的非终结符
    private void firstOneExp(String left, String exp, int i) {

        String[] items = exp.split(" ");

        if (i >= items.length) { // 超过范围，不能再后看
            System.out.println("error");
            new HashSet<String>() {
            };
            return;
        }

        String right = items[i];   // 右边的产生式第一个字符串

        Set<String> firstList; // 非终结符号对应的first集合
        if (first.get(left) == null) { // 没有记录过这个非终结符
            firstList = new HashSet<String>();
        } else { // 记录过这个终结符
            firstList = first.get(left);
        }

        if (terminators.contains(right)) {  // 文法右边的产生式第一个字符串
是终结符
            firstList.add(right);
        } else { // 文法右边的产生式第一个字符串是非终结符
            Set<String> temps;
            if (hasrecord.get(right) == null) {
                temps = new HashSet<>(firstExps(right));
            } else { // 已经找过非终结符，就不用再找
                temps = first.get(right);
            }
            boolean hasnull = false; // first 集合是否存在空字符串 $
            Iterator<String> it = temps.iterator();
            while (it.hasNext()) {
                String temp = it.next();
                if (temp.equals(nullString)) {
                    hasnull = true;
                    it.remove(); // 要除开这个空字符串
                }
            }
            // 将右边第一个非终结符的fisrt集合的，除开开始符号加入
            firstList.addAll(temps);
            if (hasnull) {
                if (i + 1 < items.length) {
                    // 存在空字符串则要向下看下一个字符串
                    firstOneExp(left, exp, i + 1);
```

```
                } else {
                    // 表达式中最后一个字符串仍然有空字符串，则将空字符串加入
                    firstList.add(nullString);
                }
            }
        }
        first.put(left, firstList);
    }
```

1.3 firstOfStrs

为了便于后面的follow集合计算和获得预测分析表，使用 firstOfStrs 用来求一组字符串的first集合，并返回这个字符串组是否能推出空字符串。算法如下：

> 对文法G的任何符号串α=X1X2...Xn构造集合FIRST(α)
>
> (1)首先置FIRST(α)= FIRST(X1){□}. (2)如果对任何j, j∈[1, i-1], $ ∈FIRST (Xj), 则把FIRST(Xj)\{$}加入到FIRST(α)
>
> 中.特别是,若所有的FIRST(Xj)均含有$, j=1,2,...,n, 则把$加到FIRST(α)中.

```
private Boolean firstofStrs(String line, int i, Set<String> sets, Boolean
hasA) {
        //Set<String> sets = new HashSet<>();
        // 若A→αBβ是一个产生式, 则把FIRST(β)\{ε}加至FOLLOW(B)中
        String[] items = line.split(" ");
        int j;
        for (j = i; j < items.length; j++) {
            // B有下一位
            String next = items[j];
            if (next.equals(nullString)) { // 如果是空字符串
                sets.add(next);
                return true;
            } else if (terminators.contains(next)) { // 终结符的 first 集合是
自己
                sets.add(next);
                return hasA;// 遇到终结符不用再往下看
            } else { // 非终结符的 first 集合通过之前求的 first 集合获得

                Set<String> temps = new HashSet<>(first.get(next));
                boolean hasnull = false; // first 集合是否存在空字符串 $
                Iterator<String> it = temps.iterator();
                while (it.hasNext()) {
                    String temp = it.next();
                    if (temp.equals(nullString)) {
                        hasnull = true;
                        it.remove(); // 要除开这个空字符串
                    }
                }
                // 首先置FIRST(α)= FIRST(X1)\{$}.
```

```
                sets.addAll(temps);
                if (hasnull) { // 如果对任何j, j∈[1, i-1], $ ∈FIRST (Xj), 则把FIRST(Xj)\{$}加入到FIRST(α)
                    if (j + 1 == items.length) {
                        // 表达式中最后一个字符串仍然有空字符串，则将follow(A)加入
                        hasA = true;
                    }
                } else {
                    // 没有空，就不用看下一个字符串
                    break;
                }
            }
        }
        return hasA;
    }
```

1.4 follow 集合

结构同first 集合类似，外层遍历加中间分解。但是follow由于存在**成环依赖**，并且环与环之间存在**交集**，所以采用递归会造成**死循环**。因此不再使用递归，而是外面套一个while循环，直到某一次循环所有的follow集合不再变化时，循环停止。

```
    // 获得所有的 follow 集合
    void followAll() {
        Set<String> lefts = expression.keySet();

        while (true) {
            for (String left : lefts) { // 遍历
                followExps(left);
            }
            if (!change) {
                break;
            } else { //发生了改变，将change置位于未改变
                change = false;
            }
        }
    }

    /**
     * 找到非终结符 str 的 follow 集合
     *
     * @param str 非终结符
     * @return str 的 follow 集合
     */
    private void followExps(String str) {
        Set<String> sets;
```

```
        if (follow.get(str) == null) {
            sets = new HashSet<>();
        } else {
            sets = follow.get(str);
        }
        Set<String> setcmp = new HashSet<>(sets);// 使用setcmp用来比较前后是否
变化。

        for (String left : expression.keySet()) { // 根据非终结符遍历所有表达式
            ArrayList<String> exps = expression.get(left); // 获得表达式
            if (left.equals(start)) {// 对于文法的开始符号S, 置#于FOLLOW(S)中;
                sets.add("#");
                follow.put(str, sets);
            }

            for (String exp : exps) { // 表达式
                String[] items = exp.split(" ");
                for (int i = 0; i < items.length; i++) {
                    String item = items[i];
                    if (item.equals(str)) { // 表达式的右边有这个非终结符B
                        Boolean hasA = Boolean.FALSE;
                        if (i + 1 == items.length) {
                            hasA = true;
                        } else { // 若A→αBβ是一个产生式, 则把FIRST(β)\{ε}加至
FOLLOW(B)中;
                            hasA = firstofStrs(exp, i + 1, sets, hasA);
                        }
                        if (hasA && !str.equals(left)) { // follow 不是它自
己
                            // 若A→αB是一个产生式, 或A→αBβ是一个产生式而β => ε
(即 ε∈FIRST(β) ), 则把FOLLOW(A)加至FOLLOW(B)中.
                            if (follow.get(left) != null) {
                                sets.addAll(follow.get(left));// 记录下依赖集
                            }
                        }
                        // 匹配上非终结符则不用再往后遍历
                        break;
                    }
                }
            }
        }
        if (!equals(sets, setcmp)) {
            change = true; // 发生了改变
            follow.put(str, sets);
        }
    }
```

1.5 table 预测分析表

使用map结构存储预测分析表

`private Map<String, Map<String, String>> table = new HashMap<>(); // 记录预测分析表`

- key 非终结符
- value 为一个 map，记录下这个非终结符在遇到终结符（key1）时使用的产生式（value1）

不在这个表内的即为错误。

```
    /**
     * 获得某一个非终结符的预测分析表的那一行
     * @param left 非终结符
     */
    private void getOneline(String left) {
        ArrayList<String> exps = expression.get(left); // 获得所有表达式
        Map<String, String> line = new HashMap<>();
        for (String exp : exps) { // 对于每一个表达式
            Set<String> sets = new HashSet<>();
            Boolean hasnull = firstofStrs(exp, 0, sets, false);
            if (hasnull || exp.equals(nullString)) { // 空字符串，则看左边字符串的follow集合
                sets.addAll(follow.get(left));
            }
            for (String key : sets) {
                if (line.containsKey(key)) { // 已经存过这个终结符，冲突
                    System.out.println(left + "|" + key + "|" + exp + "|" + line.get(key));
                } else { // 没有存过，存进去
                    line.put(key, exp);
                }
            }
        }
        table.put(left, line);
    }
```

1.6 analyze ll1 自顶向下分析

根据预测分析表进行文法分析，并输出每一步分析的情况。每一步有三种情况：

> 假设要用非终结符A进行匹配，面临的输入符号为a，A的所有产生式为
>
> A→α1 |α2 | ... |αn
>
> - 若a∈FIRST(αi)，则指派αi去执行匹配任务。
> - 若a不属于任何一个候选首字符集，则:
>   - 若ε属于某个FIRST(αi)，且a∈FOLLOW(A)，则让A与ε自动匹配;
>   - 否则，a的出现是一种语法错误。

```
 /**
     * 根据预测分析表进行文法分析
     */
    void analyze() {
        Stack<String> signs = new Stack<>();
        signs.push("#");
        signs.push(start);
        String[] strs = text.split(" |\r\n");
        for (int i = 0; i < strs.length; i++) {
            String str = strs[i];
            String top = signs.peek(); // 获得栈顶元素
            if (top.isEmpty()) {
                System.out.println("栈已空");
            }
            System.out.print("\33[29;4m符号栈：\33[34;4m" +
stacktoStr(signs) + "     ");
            if (top.equals(str)) {
                if (top.equals("#")) {
                    System.out.println("\33[28;4m大公告成，匹配成功辽～");
                    if (i < strs.length - 1) {
                        System.out.println("但是后面还有内容");
                        break;
                    }
                }
                System.out.println("\33[33;4m移进：" + top);
                signs.pop(); // 推出栈顶
            } else if (table.get(top) == null) { // 语法错误
                System.out.println("\33[31;4m错误：栈顶：" + top + " 输入串的
第一个字符串：" + str);
                break;
            } else if (table.get(top).get(str) == null) { // 获得空项
                System.out.println("\33[31;4m错误：栈顶：" + top + " 输入串的
第一个字符串：" + str);
                break;
            } else {
                String exp = table.get(top).get(str); // 获得产生式
                System.out.println("\33[28;4m所用产生式：\33[35;4m" + top + "
-> " + exp);
                signs.pop(); // 先将栈顶推出
                String[] items = exp.split(" ");
                for (int j = items.length - 1; j >= 0; j--) {
                    String item = items[j]; // 逆序推入
                    if (!item.equals(nullString)) {
                        // 不是空字符串才入栈
                        signs.push(item);
                    }
                }
                --i; // 并没有移动
            }
```

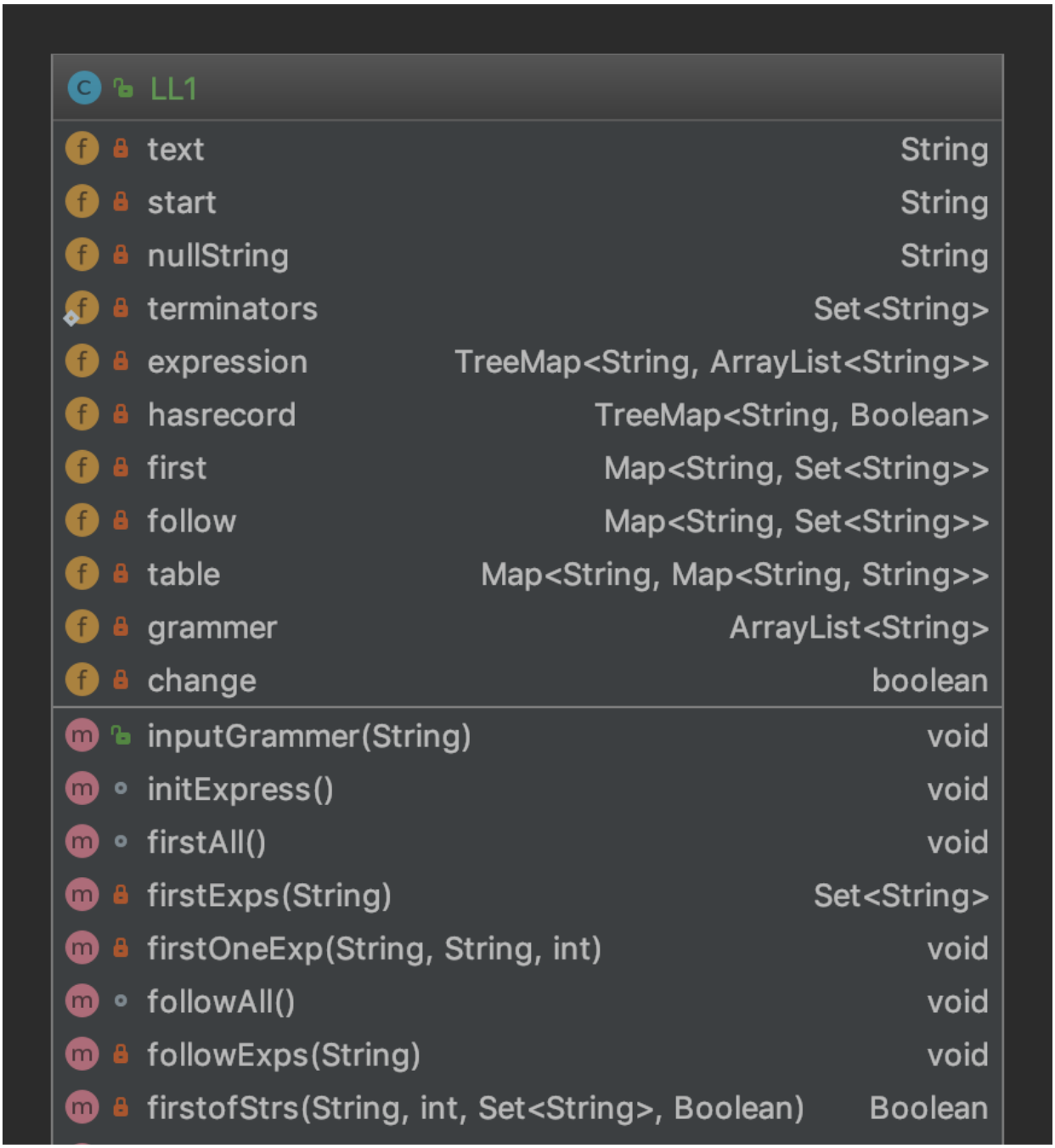
```
        }
    }
```

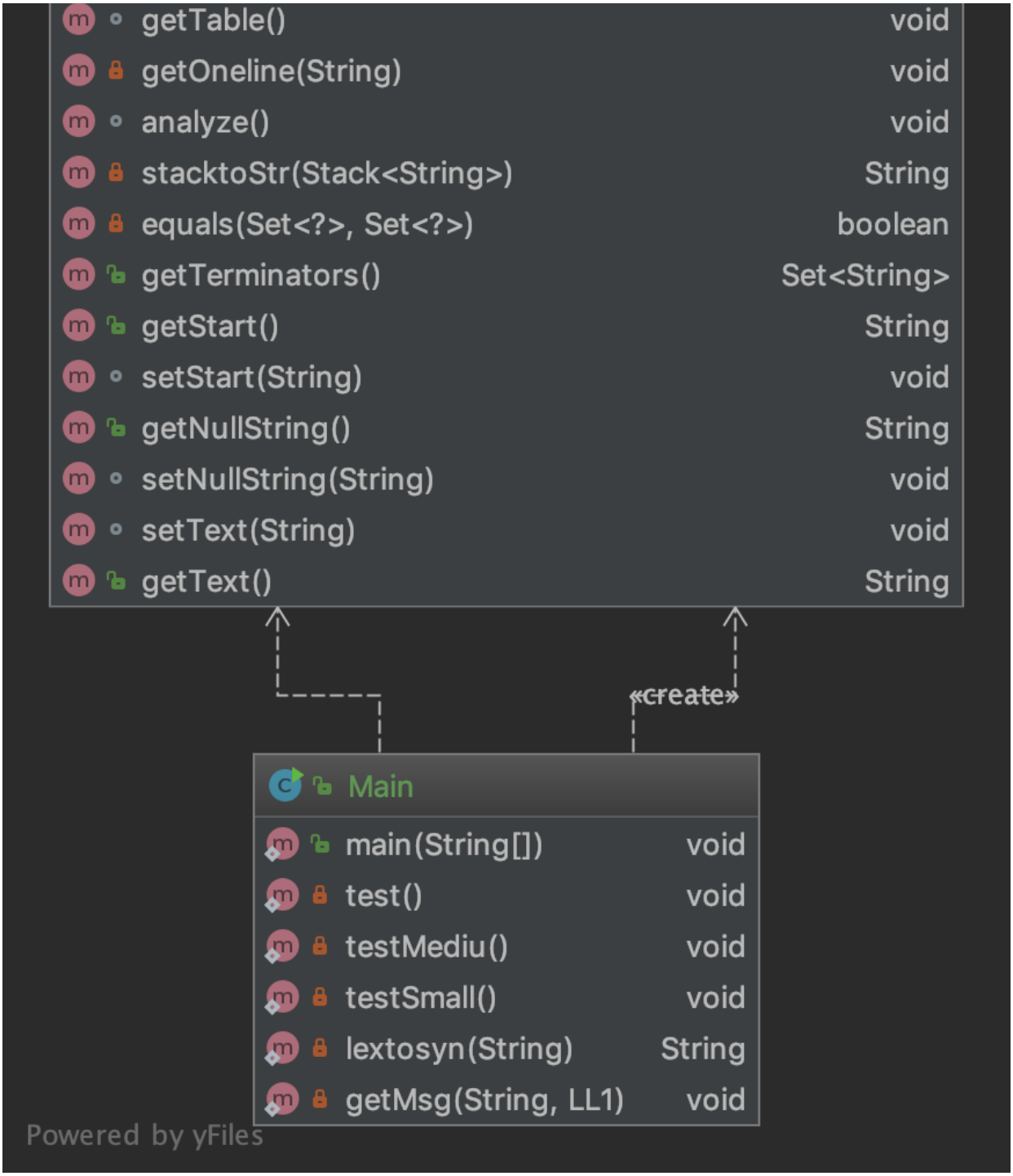
2 LL1 文法分析设计总结

2.1 基于LL1文法分析：

- 读取文法，识别文法中的终结符和非终结符的方法。
- 基于文法求出某个字符串first集合、字符串组的first集合、某一个字符串follow集合的方法。
- 根据follow集和first集合制作语法分析预测表。
- 读取待分析的内容，先使用词法分析获得字符串和相应类型，根据类型进行自定向下的词法分析的方法。
- 使用时需要注意给出文法的开始符号和空字符串表示方法。

LL1 的UML图如下所示





2.2 LL1 文法缺陷

if-else具有固有的二义属性，使用LL1文法并不能解决；可以看到在构造预测分析表时，会输出如下冲突：

```
else_result|else|$|else elseif_stmt
elseif_stmt|if|block_stmt|if_control_stmts
```