信息与软件工程学院上机实验报告

（第 3 次）

一、实验名称

实验三：SLR(1) 语法分析

二、实验目的及要求

1. 实验目的

本次实验的目的如下：

1. 掌握 SLR(1) 分析法的基本原理。
2. 掌握 SLR(1) 分析表的构造方法。
3. 掌握 SLR(1) 表驱动程序的构造方法。

2. 实验内容

构造 SLR(1) 分析程序，利用它进行语法分析，判断对于给定的输入符号串是否为该文法识别的句子。了解 SLR(1) 分析方法是严格的从左向右扫描，以便对任意的输入符号串进行分析。本次实验的主要目的是加深对 SLR(1) 分析法的理解。

3. 实验要求

1. 能够判断某一给定文法是否是 SLR(1) 文法。
2. 能够根据给定的文法构造其对应的 SLR(1) 分析表。
3. 能够根据构造的 SLR(1) 分析表进行语法分析。

三、实验环境

1. 操作系统：Linux

2. 编程语言：Python 3.9 及以上

3. 开发工具：Visual Studio Code

四、实验设计

1. 实验步骤

程序按照 SLR(1) 分析的一般流程实现：先把文法写成程序内部的数据结构，再通过迭代算法计算 FIRST 集和 FOLLOW 集；随后构造扩广文法，自动求出 LR(0) 项目集规范族以及各状态之间的 GOTO 转移；在此基础上根据 FOLLOW 集填写归约动作，构造 SLR(1) 分析表，并检查是否出现移进/归约或归约/归约冲突；最后使用“状态栈 + 符号栈 + 输入串 + 分析表”模拟整个分析过程。

由于任务书要求实现的是通用的 SLR(1) 分析程序，但没有给出唯一固定的具体文法，因此程序在演示和测试时选用了经典表达式文法作为示例文法。

1.1 设计思想

程序把文法表示成 Python 字典：键为非终结符，值为对应产生式右部列表；右部使用元组表示。整个实现以 `SLRGrammar` 类为核心，它把文法相关的核心算法集中在同一个类中，主要负责以下五件事：

1. 计算单个符号串的 FIRST 集，并通过不动点迭代法求所有非终结符的 FIRST 集和 FOLLOW 集。
2. 构造扩广文法和产生式编号表，为后续项目集和归约动作提供统一依据。
3. 实现 `closure` 与 `goto` 运算，自动构造 LR(0) 项目集规范族。
4. 根据项目集规范族和 FOLLOW 集构造 SLR(1) 分析表，并检测冲突。
5. 使用分析表模拟输入串的移进、归约与接受过程，输出逐步分析结果。

这种设计的优点在于职责清晰：文法、项目集和分析表都在同一个算法主体里维护，而命令行入口和终端输出只是外层包装，便于阅读、调试和写实验报告。

1.2 实现流程

程序的整体执行流程如图所示。

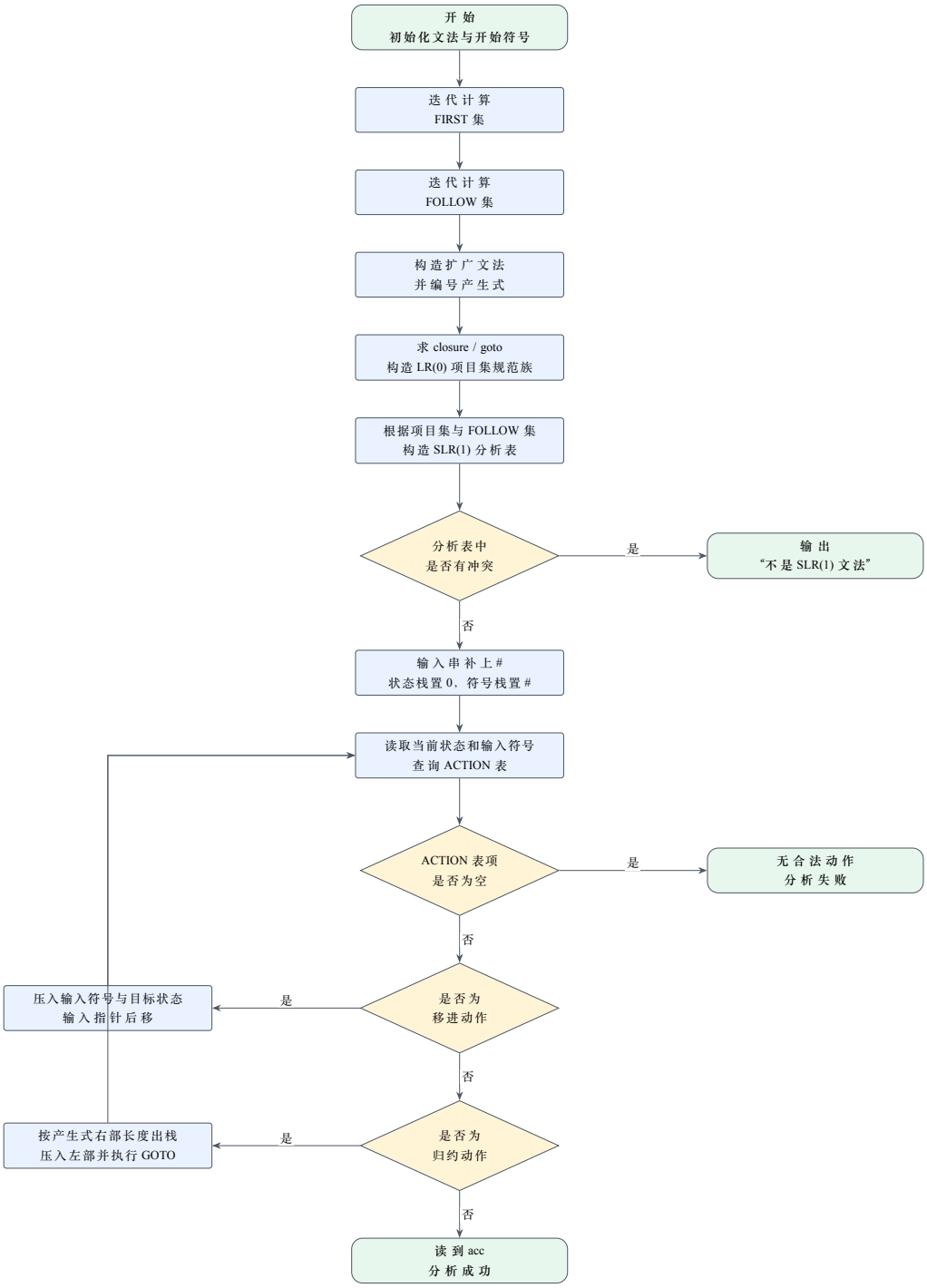


图 1: SLR(1) 语法分析程序实现流程图

从流程图可以看出，程序先处理文法本身，再处理输入串；而在分析输入串时，始终围绕“当前状态 + 当前输入 + 分析表动作”这一主线运行。这正是 LR 类自底向上分析器的典型特征。

1.3 核心算法说明

程序中最关键的五个算法如下：

1. **FIRST 集计算**：对每条产生式右部从左到右扫描。如果遇到终结符，就把它加入结果；如果遇到非终结符，就把该非终结符的 FIRST 集除去 ε 后加入结果；若该非终结符可推出 ε，则继续向后扫描。

2. **FOLLOW 集计算**：先把开始符号的 FOLLOW 集初始化为 `#`。之后遍历每条产生式，把某非终结符后继串的 FIRST 集加入它的 FOLLOW 集；若后继串可推出 ε，则把左部非终结符的 FOLLOW 集继续传递给它。

3. **LR(0) 项目集构造**：先由扩广文法的初始项目出发求闭包，得到初始状态；再对每个状态上点号后可能出现的符号执行 `goto` 运算，从而逐步生成完整的项目集规范族和状态转移关系。

4. **SLR(1) 分析表构造**：若状态在某终结符上存在转移，则填写移进动作；若状态中存在形如 $A \to \alpha\cdot$ 的完成项目，则在 $FOLLOW(A)$ 中的符号上填写归约动作；若存在扩广产生式 $E' \to E\cdot$，则在 `#` 上填写接受动作 `acc`。如果同一表项被填入不同动作，则说明文法不是 SLR(1) 文法。

5. **表驱动分析过程**：初始状态栈为 `0`，符号栈为 `#`。每一步只看栈顶状态和当前输入符号：若 ACTION 表给出移进动作，就压入输入符号和新状态；若给出归约动作，就按产生式右部长度出栈，再依据 GOTO 表转入新状态；若读到 `acc`，则分析成功；若查不到表项，则分析失败。

为了降低理解难度，程序把算法主体、数据结构和纯文本输出表格分开实现，这样在阅读代码时可以先抓住分析原理，再去看结果展示部分。

2. 调试过程及实验结果

调试时，我主要关注四类问题：第一，FIRST 集和 FOLLOW 集是否计算正确；第二，LR(0) 项目集是否完整且状态编号稳定；第三，SLR(1) 分析表是否出现冲突；第四，输入串分析过程是否能够正确体现移进、归约和接受动作。由

于 SLR(1) 比 LL(1) 多了“项目集”和“状态栈”这两个关键概念，因此本次调试重点放在状态转移和归约时机上。

2.1 FIRST 集与 FOLLOW 集计算结果

程序针对所选示例文法自动计算得到的集合如下表所示。

表 1: 表达式文法的 FIRST 集与 FOLLOW 集

非终结符	FIRST 集	FOLLOW 集
E	$\{i, (\}$	$\{+,), \#\}$
T	$\{i, (\}$	$\{+, *,), \#\}$
F	$\{i, (\}$	$\{+, *,), \#\}$

从结果可以看出，该文法中三个非终结符都可以从 `i` 或左括号开始，因此 FIRST 集完全一致；而 FOLLOW 集体现了不同层次的运算符约束，例如 * 只会出现在 `T` 和 `F` 的 FOLLOW 集中，这为后续构造归约动作提供了依据。

2.2 LR(0) 项目集规范族与 SLR(1) 判定

程序自动构造出 12 个 LR(0) 项目集状态，从 `I0` 到 `I11`。其中几个具有代表性的状态如下表所示。

表 2: 代表性 LR(0) 项目集状态说明

状态	代表项目	说明
`I0`	$E' \to \cdot E, E \to \cdot E{+}T, E \to \cdot T, T \to \cdot T{*}F,$ $T \to \cdot F,\ F \to \cdot(E),\ F \to \cdot i$	初始状态，包含扩广文法起始项目及其闭包。
`I3`	$E' \to E\cdot,\ E \to E \cdot {+}T$	一方面已经具备接受条件，另一方面在读到 + 时仍可继续移进。
`I10`	$E \to E + T\cdot,\ T \to T \cdot {*}F$	既可以把 $E + T$ 归约为 E，又可以在读到 * 时继续扩展乘法，从而体现乘法优先于加法。

根据项目集和 FOLLOW 集构造分析表时，程序没有发现任何移进/归约或归约/归约冲突，因此该文法可以判定为 SLR(1) 文法。

表 3: 扩广文法产生式编号

编号	产生式
0	$E' \to E$
1	$E \to E + T$
2	$E \to T$
3	$T \to T * F$
4	$T \to F$
5	$F \to (E)$
6	$F \to i$

表 4: 示例文法的 SLR(1) 分析表

状态	i	$+$	$*$	$($	$)$	#	E	T	F
0	s1	/	/	s2	/	/	3	4	5
1	/	r6	r6	/	r6	r6	/	/	/
2	s1	/	/	s2	/	/	6	4	5
3	/	s7	/	/	/	acc	/	/	/
4	/	r2	s8	/	r2	r2	/	/	/
5	/	r4	r4	/	r4	r4	/	/	/
6	/	s7	/	/	s9	/	/	/	/
7	s1	/	/	s2	/	/	/	10	5
8	s1	/	/	s2	/	/	/	/	11
9	/	r5	r5	/	r5	r5	/	/	/
10	/	r1	s8	/	r1	r1	/	/	/
11	/	r3	r3	/	r3	r3	/	/	/

从分析表可以看到，状态 4 在读到 * 时选择移进、而在读到 +、）或 # 时选择按 r2 归约，这说明程序正确地区分了运算符优先级；而状态 3 在 # 上给出 acc，表示整个句子已经分析完成。

2.3 测试样例与分析过程

为了验证程序功能，本实验选取了若干正确样例和错误样例进行测试。正确样例用于覆盖标识符、加法、乘法和括号等不同语法结构；错误样例用于验证程序能否在缺少操作数或括号不匹配时及时停止并给出结论。

表 5: 测试样例与运行结果汇总

测试编号	输入串	实际结果	说明
1	`i#`	接受	验证最基本的标识符表达式可被归约为开始符号。
2	`i+i#`	接受	验证加法表达式能够正确完成多次移进与归约。
3	`i+i*i#`	接受	验证乘法高于加法，程序会先处理 `T * F` 再归约成 `E`。
4	`(i+i)*i#`	接受	验证括号表达式与乘法组合时的状态转移过程。
5	`i+#`	拒绝	在读入 `+` 后缺少右操作数，分析表中不存在合法动作。
6	`(i*i#`	拒绝	缺少右括号，分析到末尾时仍无法满足状态 6 的合法输入条件。

下面选取一个正确样例 `i+i*i#` 和一个错误样例 `i+#` 展示具体分析过程。

表 6: 输入串 i+i*i# 的 SLR(1) 分析过程

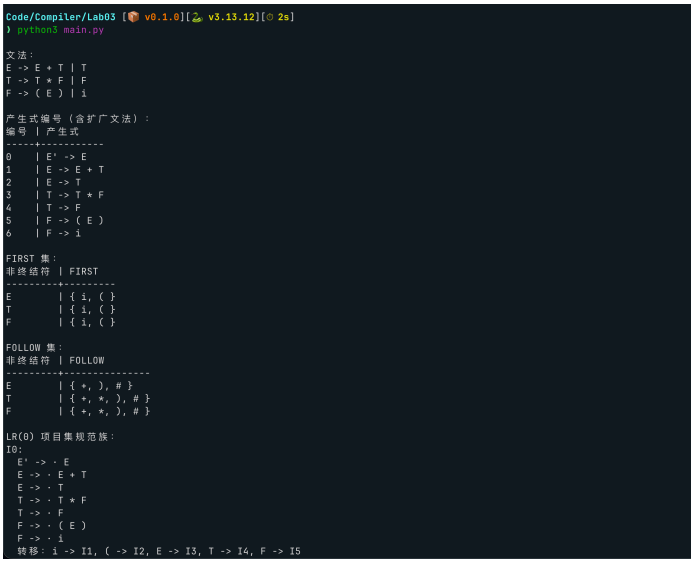
步骤	状态栈	符号栈	剩余输入	动作
1	0	#	i+i*i#	按 ACTION[0, i] = s1 移进符号 i。
2	0 1	# i	+i*i#	按 ACTION[1, +] = r6，用 $F \to i$ 归约，再转到状态 5。
3	0 5	# F	+i*i#	按 ACTION[5, +] = r4，用 $T \to F$ 归约，再转到状态 4。
4	0 4	# T	+i*i#	按 ACTION[4, +] = r2，用 $E \to T$ 归约，再转到状态 3。
5	0 3	# E	+i*i#	按 ACTION[3, +] = s7 移进符号 +。
6	0 3 7	# E +	i*i#	按 ACTION[7, i] = s1 移进符号 i。
7	0 3 7 1	# E + i	*i#	按 ACTION[1, *] = r6，用 $F \to i$ 归约，再转到状态 5。
8	0 3 7 5	# E + F	*i#	按 ACTION[5, *] = r4，用 $T \to F$ 归约，再转到状态 10。
9	0 3 7 10	# E + T	*i#	按 ACTION[10, *] = s8 移进符号 *。
10	0 3 7 10 8	# E + T *	i#	按 ACTION[8, i] = s1 移进符号 i。
11	0 3 7 10 8 1	# E + T * i	#	按 ACTION[1, #] = r6，用 $F \to i$ 归约，再转到状态 11。
12	0 3 7 10 8 11	# E + T * F	#	按 ACTION[11, #] = r3，用 $T \to T * F$ 归约，再转到状态 10。
13	0 3 7 10	# E + T	#	按 ACTION[10, #] = r1，用 $E \to E + T$ 归约，再转到状态 3。
14	0 3	# E	#	按 ACTION[3, #] = acc，分析成功。

表 7: 输入串 i+# 的 SLR(1) 分析过程

步骤	状态栈	符号栈	剩余输入	动作
1	0	#	i+#	按 ACTION[0, i] = s1 移进符号 i。
2	0 1	# i	+#	按 ACTION[1, +] = r6，用 $F \to i$ 归约，再转到状态 5。
3	0 5	# F	+#	按 ACTION[5, +] = r4，用 $T \to F$ 归约，再转到状态 4。
4	0 4	# T	+#	按 ACTION[4, +] = r2，用 $E \to T$ 归约，再转到状态 3。
5	0 3	# E	+#	按 ACTION[3, +] = s7 移进符号 +。
6	0 3 7	# E +	#	ACTION[7, #] 为空，分析失败。

2.4 运行截图

为了更直观地展示程序运行效果，报告中补充了两张终端截图。第一张展示程序输出文法、产生式编号、FIRST 集、FOLLOW 集、LR(0) 项目集规范族以及 SLR(1) 分析表的结果；第二张展示程序对多个测试样例进行表驱动分析时的具体过程。



```
Code/Compiler/Lab03 [ v0.1.0][ v3.13.12][ 2s]
❯ python3 main.py

文法：
E -> E + T | T
T -> T * F | F
F -> ( E ) | i

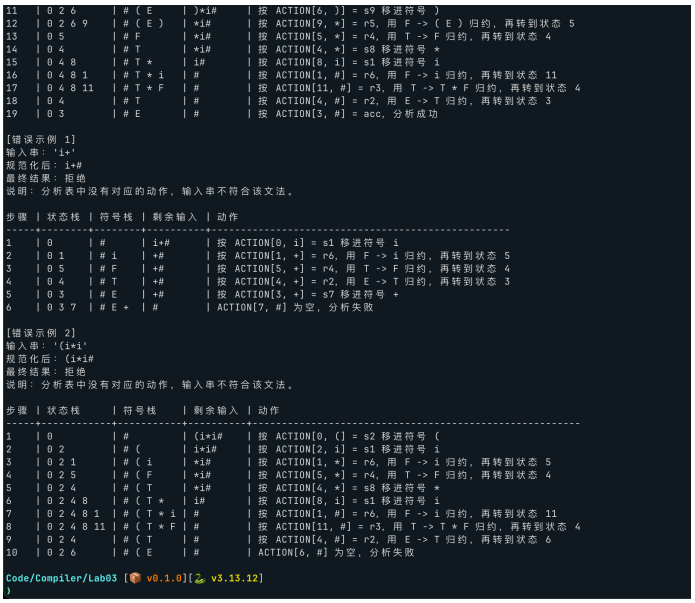
产生式编号（含扩广文法）：
编号 | 产生式
-----+----------
0    | E' -> E
1    | E -> E + T
2    | E -> T
3    | T -> T * F
4    | T -> F
5    | F -> ( E )
6    | F -> i

FIRST 集：
非终结符 | FIRST
---------+--------
E        | { i, ( }
T        | { i, ( }
F        | { i, ( }

FOLLOW 集：
非终结符 | FOLLOW
---------+--------------
E        | { +, ), # }
T        | { +, *, ), # }
F        | { +, *, ), # }

LR(0) 项目集规范族：
I0:
  E' -> · E
  E -> · E + T
  E -> · T
  T -> · T * F
  T -> · F
  F -> · ( E )
  F -> · i
  转移：i -> I1, ( -> I2, E -> I3, T -> I4, F -> I5
```

图 2: 程序输出 FIRST 集、FOLLOW 集、LR(0) 项目集与 SLR(1) 分析表的运行结果



```
11   | 0 2 6      | # ( E      | )*i#     | 按 ACTION[6, )] = s9 移进符号 )
12   | 0 2 6 9    | # ( E )    | *i#      | 按 ACTION[9, *] = r5, 用 F -> ( E ) 归约，再转到状态 5
13   | 0 5        | # F        | *i#      | 按 ACTION[5, *] = r4, 用 T -> F 归约，再转到状态 4
14   | 0 4        | # T        | *i#      | 按 ACTION[4, *] = s8 移进符号 *
15   | 0 4 8      | # T *      | i#       | 按 ACTION[8, i] = s1 移进符号 i
16   | 0 4 8 1    | # T * i    | #        | 按 ACTION[1, #] = r6, 用 F -> i 归约，再转到状态 11
17   | 0 4 8 11   | # T * F    | #        | 按 ACTION[11, #] = r3, 用 T -> T * F 归约，再转到状态 4
18   | 0 4        | # T        | #        | 按 ACTION[4, #] = r2, 用 E -> T 归约，再转到状态 3
19   | 0 3        | # E        | #        | 按 ACTION[3, #] = acc，分析成功

[错误示例 1]
输入串：'i+'
规范化后：i+#
最终结果：拒绝
说明：分析表中没有对应的动作，输入串不符合该文法。

步骤 | 状态栈 | 符号栈 | 剩余输入 | 动作
-----+--------+--------+----------+-----------------------------------------------------
1    | 0      | #      | i+#      | 按 ACTION[0, i] = s1 移进符号 i
2    | 0 1    | # i    | +#       | 按 ACTION[1, +] = r6, 用 F -> i 归约，再转到状态 5
3    | 0 5    | # F    | +#       | 按 ACTION[5, +] = r4, 用 T -> F 归约，再转到状态 4
4    | 0 4    | # T    | +#       | 按 ACTION[4, +] = r2, 用 E -> T 归约，再转到状态 3
5    | 0 3    | # E    | +#       | 按 ACTION[3, +] = s7 移进符号 +
6    | 0 3 7  | # E +  | #        | ACTION[7, #] 为空，分析失败

[错误示例 2]
输入串：'(i+i'
规范化后：(i+i#
最终结果：拒绝
说明：分析表中没有对应的动作，输入串不符合该文法。

步骤 | 状态栈     | 符号栈     | 剩余输入 | 动作
-----+------------+------------+----------+-----------------------------------------------------
1    | 0          | #          | (i+i#    | 按 ACTION[0, (] = s2 移进符号 (
2    | 0 2        | # (        | i+i#     | 按 ACTION[2, i] = s1 移进符号 i
3    | 0 2 1      | # ( i      | +i#      | 按 ACTION[1, +] = r6, 用 F -> i 归约，再转到状态 5
4    | 0 2 5      | # ( F      | +i#      | 按 ACTION[5, +] = r4, 用 T -> F 归约，再转到状态 4
5    | 0 2 4      | # ( T      | +i#      | 按 ACTION[4, +] = s8 移进符号 *
6    | 0 2 4 8    | # ( T *    | i#       | 按 ACTION[8, i] = s1 移进符号 i
7    | 0 2 4 8 1  | # ( T * i  | #        | 按 ACTION[1, #] = r6, 用 F -> i 归约，再转到状态 11
8    | 0 2 4 8 11 | # ( T * F  | #        | 按 ACTION[11, #] = r3, 用 T -> T * F 归约，再转到状态 4
9    | 0 2 4      | # ( T      | #        | 按 ACTION[4, #] = r2, 用 E -> T 归约，再转到状态 6
10   | 0 2 6      | # ( E      | #        | ACTION[6, #] 为空，分析失败

Code/Compiler/Lab03 [ v0.1.0][ v3.13.12]
❯
```

图 3: 程序对测试输入串进行 SLR(1) 分析的运行结果

2.5 结果分析

从集合计算、项目集构造、分析表构造和测试样例四个方面来看，程序已经完整覆盖实验要求。首先，FIRST 集和 FOLLOW 集全部由程序自动计算，避免

了手工推导时容易出现的遗漏。其次，程序能够从扩广文法自动生成 LR(0) 项目集规范族，并进一步构造出无冲突的 SLR(1) 分析表，从而说明程序对示例文法的 SLR(1) 判定与分析表构造是正确的。再次，在输入串分析阶段，程序能够清楚展示每一步的状态栈、符号栈、剩余输入和具体动作，因此不仅能得到“接受或拒绝”的最终结论，还能完整追踪移进与归约发生的原因。

此外，测试样例还体现了 SLR(1) 分析的两个重要特点：一是通过状态和 FOLLOW 集共同决定归约时机，从而正确体现乘法高于加法的优先级；二是通过状态栈精确识别括号结构和错误位置，在缺少右操作数或右括号时及时停止分析。这说明程序不仅实现了功能，而且能够较好地反映 SLR(1) 分析法的工作机制。

3. **总结**

本次实验完成了一个面向教学场景的 SLR(1) 语法分析程序，整体实现覆盖了实验要求中的核心任务，包括 FIRST 集计算、FOLLOW 集计算、LR(0) 项目集规范族构造、SLR(1) 判定、SLR(1) 分析表构造以及输入串分析过程模拟。程序规模不大，但算法链条完整、层次清楚，适合用于理解自底向上语法分析器的工作方式。

从实现角度看，这次实验让我更清楚地认识到：SLR(1) 分析并不是简单地把“移进”和“归约”写成两个动作，而是要先通过扩广文法、项目集和 FOLLOW 集把这些动作的触发条件严格确定下来。只有项目集构造正确，状态转移关系清楚，归约动作才能在合适的输入符号上发生；也只有分析表无冲突，表驱动分析过程才会稳定运行。因此，本实验很好地把课堂上的抽象概念转化成了可观察、可验证的程序行为。

3.1 上机感悟

通过这次上机，我对 SLR(1) 分析的理解比单纯看书时清晰了很多。以前更多是从公式和定义上理解“项目”“闭包”“GOTO”“移进归约冲突”等概念，真正写成程序后才体会到它们之间是如何层层衔接的：先由文法生成 FIRST 集和 FOLLOW 集，再由扩广文法生成项目集规范族，最后把这些信息综合到分析表

中，驱动整个分析过程。尤其是在调试状态栈和归约步骤时，我更加体会到编译原理中的每一个规则最终都会体现在程序的一步动作里。通过自己实现并验证这些过程，我不仅掌握了 SLR(1) 分析法的基本原理，也增强了把抽象算法整理成清晰代码的能力。

4. 附录

附录中只保留核心代码和测试样例。为了突出算法主体，终端输出格式化、结果拼接和报告展示相关函数不再展开。

4.1 SLR(1) 分析核心程序节选 `slr1.py`

```python
@dataclass(frozen=True)
class Production:
    """一条产生式，同时记录它在分析表中的编号。"""

    index: int
    left: str
    right: tuple[str, ...]

    def display(self) -> str:
        """把产生式格式化成便于展示的字符串。"""
        right_text = " ".join(self.right) if self.right else
    EPSILON
        return f"{self.left} -> {right_text}"

    def numbered_display(self) -> str:
        """带编号输出，方便和 r1、r2 之类的归约动作对应。"""
        return f"{self.index}: {self.display()}"


@dataclass(frozen=True)
class Item:
    """LR(0) 项目，例如 E -> E · + T。"""

    production_index: int
    dot: int


@dataclass(frozen=True)
class Action:
    """SLR(1) 分析表中的一个 ACTION 表项。"""

```

```
    kind: str
    target: int | None = None

    def short(self) -> str:
        """输出表中常见的简写形式，例如 s5、r3、acc。"""
        if self.kind == "shift":
            return f"s{self.target}"
        if self.kind == "reduce":
            return f"r{self.target}"
        return "acc"


@dataclass
class AnalysisStep:
    """记录 SLR(1) 表驱动分析过程中的一步。"""

    step_no: int
    state_stack: str
    symbol_stack: str
    remaining_input: str
    action: str


@dataclass
class ParseResult:
    """保存一次输入串分析的完整结果。"""

    original_sentence: str
    normalized_sentence: str
    accepted: bool
    message: str
    steps: list[AnalysisStep]


class SLRGrammar:
    """一个尽量小而清晰的 SLR(1) 文法工具类。"""

    def __init__(
        self,
        productions: dict[str, list[tuple[str, ...]]],
        start_symbol: str,
        terminal_order: list[str] | None = None,
    ) -> None:
        self.productions = productions
        self.start_symbol = start_symbol
```

```
        self.nonterminals = list(productions.keys())
        self.nonterminal_set = set(self.nonterminals)

        self.augmented_start_symbol = self._build_augmented_start_symbol()
        self.all_nonterminals = [self.augmented_start_symbol] + self.nonterminals
        self.all_nonterminal_set = set(self.all_nonterminals)

        auto_terminals: list[str] = []
        for alternatives in productions.values():
            for right in alternatives:
                for symbol in right:
                    if symbol in self.nonterminal_set or symbol == EPSILON:
                        continue
                    if symbol not in auto_terminals:
                        auto_terminals.append(symbol)

        self.terminals = terminal_order or auto_terminals
        self.production_list = self._build_production_list()
        self.production_indexes_by_left = self._build_production_indexes_by_left()

    def _build_augmented_start_symbol(self) -> str:
        """构造一个不会和原文法冲突的扩广开始符号。"""
        candidate = f"{self.start_symbol}'"
        while candidate in self.productions:
            candidate += "'"
        return candidate

    def _normalize_right(self, right: tuple[str, ...]) -> tuple[str, ...]:
        """允许用户把 epsilon 写成空元组或显式的 ε。"""
        if right == (EPSILON,):
            return ()
        return right

    def _build_production_list(self) -> list[Production]:
        """
        构造产生式编号表。

        约定:
        - 0 号产生式是扩广文法: S' -> S
        - 1 号及以后才是原文法中的产生式
```

```
        """
        production_list = [
            Production(0, self.augmented_start_symbol, (self.
    start_symbol,))
        ]

        next_index = 1
        for left, alternatives in self.productions.items():
            for right in alternatives:
                production_list.append(
                    Production(next_index, left, self.
    _normalize_right(right))
                )
                next_index += 1

        return production_list

    def _build_production_indexes_by_left(self) -> dict[str, list
    [int]]:
        """按左部归类产生式，便于 closure 直接查找。"""
        mapping = {left: [] for left in self.all_nonterminals}
        for production in self.production_list:
            mapping.setdefault(production.left, []).append(
    production.index)
        return mapping

    def is_nonterminal(self, symbol: str) -> bool:
        """判断一个符号是否是非终结符。"""
        return symbol in self.all_nonterminal_set

    def sort_symbol_key(self, symbol: str) -> tuple[int, int]:
        """给终结符和非终结符一个稳定的展示顺序。"""
        if symbol in self.terminals:
            return (0, self.terminals.index(symbol))
        if symbol in self.nonterminals:
            return (1, self.nonterminals.index(symbol))
        if symbol == self.augmented_start_symbol:
            return (2, 0)
        return (3, 0)

    def sort_items(self, items: frozenset[Item]) -> list[Item]:
        """让项目输出顺序稳定，方便和实验报告对照。"""
        return sorted(items, key=lambda item: (item.
    production_index, item.dot))

```

```
156     def production_at(self, index: int) -> Production:
157         """按编号取产生式。"""
158         return self.production_list[index]
159 
160     def symbol_after_dot(self, item: Item) -> str | None:
161         """查看项目中点号后面的符号。"""
162         production = self.production_at(item.production_index)
163         if item.dot >= len(production.right):
164             return None
165         return production.right[item.dot]
166 
167     def format_item(self, item: Item) -> str:
168         """把 LR(0) 项目格式化成字符串。"""
169         production = self.production_at(item.production_index)
170 
171         if not production.right:
172             return f"{production.left} -> {ITEM_DOT}"
173 
174         parts: list[str] = []
175         for index, symbol in enumerate(production.right):
176             if index == item.dot:
177                 parts.append(ITEM_DOT)
178             parts.append(symbol)
179         if item.dot == len(production.right):
180             parts.append(ITEM_DOT)
181 
182         return f"{production.left} -> {' '.join(parts)}"
183 
184     def first_of_sequence(
185         self,
186         symbols: tuple[str, ...],
187         first_sets: dict[str, set[str]],
188     ) -> set[str]:
189         """计算一个符号串的 FIRST 集。"""
190         if not symbols:
191             return {EPSILON}
192 
193         result: set[str] = set()
194         for symbol in symbols:
195             if symbol not in self.nonterminal_set:
196                 result.add(symbol)
197                 return result
198 
199             result.update(first_sets[symbol] - {EPSILON})
200             if EPSILON not in first_sets[symbol]:
```

```
                return result

        result.add(EPSILON)
        return result

    def compute_first_sets(self) -> dict[str, set[str]]:
        """使用不动点迭代法计算 FIRST 集。"""
        first_sets = {nonterminal: set() for nonterminal in self.
    nonterminals}

        changed = True
        while changed:
            changed = False

            for left, alternatives in self.productions.items():
                for right in alternatives:
                    normalized_right = self._normalize_right(
    right)
                    before_size = len(first_sets[left])
                    first_sets[left].update(
                        self.first_of_sequence(normalized_right,
    first_sets)
                    )
                    if len(first_sets[left]) != before_size:
                        changed = True

        return first_sets

    def compute_follow_sets(
        self,
        first_sets: dict[str, set[str]],
    ) -> dict[str, set[str]]:
        """使用不动点迭代法计算 FOLLOW 集。"""
        follow_sets = {nonterminal: set() for nonterminal in self
    .nonterminals}
        follow_sets[self.start_symbol].add(END_MARKER)

        changed = True
        while changed:
            changed = False

            for left, alternatives in self.productions.items():
                for right in alternatives:
                    normalized_right = self._normalize_right(
    right)
```

```
                    for index, symbol in enumerate(normalized_right):
                        if symbol not in self.nonterminal_set:
                            continue

                        beta = normalized_right[index + 1 :]
                        first_beta = self.first_of_sequence(beta, first_sets)

                        before_size = len(follow_sets[symbol])
                        follow_sets[symbol].update(first_beta - {EPSILON})

                        if EPSILON in first_beta:
                            follow_sets[symbol].update(follow_sets[left])

                        if len(follow_sets[symbol]) != before_size:
                            changed = True

        return follow_sets

    def closure(self, items: frozenset[Item]) -> frozenset[Item]:
        """计算一个项目集的闭包。"""
        closure_set = set(items)
        queue = deque(items)

        while queue:
            item = queue.popleft()
            symbol = self.symbol_after_dot(item)

            if symbol is None or symbol not in self.nonterminal_set:
                continue

            for production_index in self.production_indexes_by_left[symbol]:
                new_item = Item(production_index, 0)
                if new_item in closure_set:
                    continue
                closure_set.add(new_item)
                queue.append(new_item)

        return frozenset(closure_set)
```

```
279
280     def goto(self, items: frozenset[Item], symbol: str) ->
    frozenset[Item]:
281         """计算 GOTO(I, X)。"""
282         moved_items = {
283             Item(item.production_index, item.dot + 1)
284             for item in items
285             if self.symbol_after_dot(item) == symbol
286         }
287
288         if not moved_items:
289             return frozenset()
290
291         return self.closure(frozenset(moved_items))
292
293     def build_canonical_collection(
294         self,
295     ) -> tuple[list[frozenset[Item]], dict[tuple[int, str], int
    ]]:
296         """构造 LR(0) 项目集规范族及状态转移。"""
297         start_state = self.closure(frozenset({Item(0, 0)}))
298         states = [start_state]
299         state_ids = {start_state: 0}
300         transitions: dict[tuple[int, str], int] = {}
301
302         queue = deque([0])
303         while queue:
304             state_id = queue.popleft()
305             state = states[state_id]
306
307             next_symbols: list[str] = []
308             for item in self.sort_items(state):
309                 symbol = self.symbol_after_dot(item)
310                 if symbol is not None and symbol not in
    next_symbols:
311                     next_symbols.append(symbol)
312
313             next_symbols.sort(key=self.sort_symbol_key)
314
315             for symbol in next_symbols:
316                 next_state = self.goto(state, symbol)
317                 if not next_state:
318                     continue
319
320                 if next_state not in state_ids:
```

```
                    state_ids[next_state] = len(states)
                    states.append(next_state)
                    queue.append(state_ids[next_state])

                transitions[(state_id, symbol)] = state_ids[next_state]

        return states, transitions

    def describe_action(self, action: Action) -> str:
        """把 ACTION 表项转成更完整的中文说明。"""
        if action.kind == "shift":
            return f"{action.short()}（移进并转到状态 {action.target}）"
        if action.kind == "reduce":
            production = self.production_at(action.target or 0)
            return f"{action.short()}（按 {production.display()} 归约）"
        return "acc（接受）"

    def _put_action_entry(
        self,
        action_table: dict[int, dict[str, Action | None]],
        state_id: int,
        terminal: str,
        action: Action,
        conflicts: list[str],
    ) -> None:
        """向 ACTION 表填入一个表项，并检测冲突。"""
        current = action_table[state_id][terminal]

        if current is None:
            action_table[state_id][terminal] = action
            return

        if current != action:
            message = (
                f"ACTION[{state_id}, {terminal}] 同时出现 "
                f"{self.describe_action(current)} 和 {self.describe_action(action)}"
            )
            if message not in conflicts:
                conflicts.append(message)

    def build_slr_table(
```

```
        self,
        follow_sets: dict[str, set[str]],
        states: list[frozenset[Item]],
        transitions: dict[tuple[int, str], int],
    ) -> tuple[
        dict[int, dict[str, Action | None]],
        dict[int, dict[str, int | None]],
        list[str],
    ]:
        """根据 LR(0) 项目集和 FOLLOW 集构造 SLR(1) 分析表。"""
        action_table = {
            state_id: {terminal: None for terminal in self.terminals + [END_MARKER]}
            for state_id in range(len(states))
        }
        goto_table = {
            state_id: {nonterminal: None for nonterminal in self.nonterminals}
            for state_id in range(len(states))
        }
        conflicts: list[str] = []

        for (state_id, symbol), target_state in transitions.items():
            if symbol in self.terminals:
                self._put_action_entry(
                    action_table,
                    state_id,
                    symbol,
                    Action("shift", target_state),
                    conflicts,
                )
                continue

            if symbol in self.nonterminal_set:
                goto_table[state_id][symbol] = target_state

        for state_id, state in enumerate(states):
            for item in self.sort_items(state):
                production = self.production_at(item.production_index)

                if item.dot != len(production.right):
                    continue

```

```
                if production.left == self.augmented_start_symbol
    :
                    self._put_action_entry(
                        action_table,
                        state_id,
                        END_MARKER,
                        Action("accept"),
                        conflicts,
                    )
                    continue

                for terminal in follow_sets[production.left]:
                    self._put_action_entry(
                        action_table,
                        state_id,
                        terminal,
                        Action("reduce", production.index),
                        conflicts,
                    )

        return action_table, goto_table, conflicts

    def analyze_sentence(
        self,
        sentence: str,
        action_table: dict[int, dict[str, Action | None]],
        goto_table: dict[int, dict[str, int | None]],
    ) -> ParseResult:
        """根据 SLR(1) 分析表模拟输入串的表驱动分析过程。"""
        normalized_sentence, tokens = normalize_sentence(sentence
    , self)
        allowed_symbols = set(self.terminals + [END_MARKER])

        for token in tokens:
            if token not in allowed_symbols:
                return ParseResult(
                    original_sentence=sentence,
                    normalized_sentence=normalized_sentence,
                    accepted=False,
                    message=f"输入串中出现了文法未定义的符号：{token}",
                    steps=[],
                )

        state_stack = [0]
        symbol_stack = [END_MARKER]
```

```
446         pointer = 0
447         step_no = 1
448         steps: list[AnalysisStep] = []
449 
450         while True:
451             current_state = state_stack[-1]
452             current_input = tokens[pointer]
453             action = action_table[current_state].get(
    current_input)
454 
455             state_snapshot = format_state_stack(state_stack)
456             symbol_snapshot = format_symbol_stack(symbol_stack)
457             remaining_input = format_token_sequence(tokens[
    pointer:])
458 
459             if action is None:
460                 steps.append(
461                     AnalysisStep(
462                         step_no,
463                         state_snapshot,
464                         symbol_snapshot,
465                         remaining_input,
466                         f"ACTION[{current_state}, {current_input
    }] 为空，分析失败",
467                     )
468                 )
469                 return ParseResult(
470                     original_sentence=sentence,
471                     normalized_sentence=normalized_sentence,
472                     accepted=False,
473                     message="分析表中没有对应的动作，输入串不符合该文法。
    ",
474                     steps=steps,
475                 )
476 
477             if action.kind == "shift":
478                 steps.append(
479                     AnalysisStep(
480                         step_no,
481                         state_snapshot,
482                         symbol_snapshot,
483                         remaining_input,
484                         f"按 ACTION[{current_state}, {
    current_input}] = {action.short()} 移进符号 {current_input}",
485                     )
```

```
                )
                symbol_stack.append(current_input)
                state_stack.append(action.target or 0)
                pointer += 1
                step_no += 1
                continue

            if action.kind == "reduce":
                production = self.production_at(action.target or 0)

                if production.right:
                    for _ in production.right:
                        symbol_stack.pop()
                        state_stack.pop()

                goto_source_state = state_stack[-1]
                goto_state = goto_table[goto_source_state].get(production.left)
                if goto_state is None:
                    steps.append(
                        AnalysisStep(
                            step_no,
                            state_snapshot,
                            symbol_snapshot,
                            remaining_input,
                            f"归约后 GOTO[{goto_source_state}, {production.left}] 为空，分析失败",
                        )
                    )
                    return ParseResult(
                        original_sentence=sentence,
                        normalized_sentence=normalized_sentence,
                        accepted=False,
                        message="归约后找不到合法的 GOTO 状态。",
                        steps=steps,
                    )

                steps.append(
                    AnalysisStep(
                        step_no,
                        state_snapshot,
                        symbol_snapshot,
                        remaining_input,
                        (
```

```
                                f"按 ACTION[{current_state}, {current_input}] = {action.short()}，"
                                f"用 {production.display()} 归约，再转到状态 {goto_state}"
                            ),
                        )
                    )
                    symbol_stack.append(production.left)
                    state_stack.append(goto_state)
                    step_no += 1
                    continue

            steps.append(
                AnalysisStep(
                    step_no,
                    state_snapshot,
                    symbol_snapshot,
                    remaining_input,
                    f"按 ACTION[{current_state}, {current_input}] = acc，分析成功",
                )
            )
            return ParseResult(
                original_sentence=sentence,
                normalized_sentence=normalized_sentence,
                accepted=True,
                message="输入串符合该 SLR(1) 文法。",
                steps=steps,
            )


def build_default_grammar() -> SLRGrammar:
    """构造实验三默认使用的表达式文法。"""
    productions = {
        "E": [("E", "+", "T"), ("T",)],
        "T": [("T", "*", "F"), ("F",)],
        "F": [("(", "E", ")"), ("i",)],
    }

    # 这里使用更常见的列顺序，输出表格时更容易阅读。
    terminal_order = ["i", "+", "*", "(", ")"]
    return SLRGrammar(
        productions=productions,
        start_symbol="E",
        terminal_order=terminal_order,
```

```
    )


DEFAULT_CASES = [
    ("单个标识符", "i"),
    ("简单加法", "i+i"),
    ("乘法优先级", "i+i*i"),
    ("括号表达式", "(i+i)*i"),
    ("错误示例 1", "i+"),
    ("错误示例 2", "(i*i"),
]


def split_sentence(sentence: str, grammar: SLRGrammar) -> list[str]:
    """把输入串拆成符号序列。"""
    stripped = sentence.strip()

    if not stripped:
        return []

    cleaned = "".join(sentence.split())

    # 默认文法中的终结符都是单字符，因此无论用户有没有写空格，
    # 都可以先去掉空白，再逐字符切分。
    if all(len(symbol) == 1 for symbol in grammar.terminals + [END_MARKER]):
        return list(cleaned)

    # 如果你以后把文法改成多字符终结符（例如 id、num），
    # 那么建议输入时使用空格分隔。
    if any(char.isspace() for char in stripped):
        return stripped.split()

    return [cleaned]


def format_token_sequence(tokens: list[str]) -> str:
    """把符号序列还原成适合终端显示的文本。"""
    if all(len(token) == 1 for token in tokens):
        return "".join(tokens)
    return " ".join(tokens)


def normalize_sentence(sentence: str, grammar: SLRGrammar) ->
```

```
        tuple[str, list[str]]:
    """规范化输入串，并自动在末尾补上结束符 #。"""
    tokens = split_sentence(sentence, grammar)
    if not tokens or tokens[-1] != END_MARKER:
        tokens.append(END_MARKER)
    return format_token_sequence(tokens), tokens
```

4.2 正确测试样例 accepted_cases.txt

```
# 下面每一行都是一个应当被接受的输入串。
# 这里的 i 表示一个标识符。
#
i#
i+i#
i*i#
(i+i)*i#
```

4.3 错误测试样例 rejected_cases.txt

```
# 下面每一行都是一个应当被拒绝的输入串。
# 这里的 i 表示一个标识符。
#
i+#
+i#
(i*i#
i*)#
```