信息与软件工程学院

上机实验报告册

专　　业　计算机科学与技术

班　　级

学　　号

姓　　名

课程名称　　编译原理

教　　师　　邹长军

学　　期　2025—2026 学年第二学期

2026 年　4　月　1　1　日

信息与软件工程学院上机实验报告

（第 2 次）

一、实验名称

实验二：LL(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. 能够判定一个文法是否为 LL(1) 文法。
2. 能够构造 LL(1) 语法分析表。
3. 能够以程序实现 LL(1) 的语法分析过程。

2. 实验内容

已知文法 $G[S]$ 为：

$$
\begin{aligned}
S &\rightarrow MH \mid a \\
H &\rightarrow LSo \mid \varepsilon \\
K &\rightarrow dML \mid \varepsilon \\
L &\rightarrow eHf \\
M &\rightarrow K \mid bLM
\end{aligned}
$$

3. 实验要求

结合题目要求，程序需要完成以下内容：

1. 给出该文法的 FIRST 集和 FOLLOW 集，并由程序实现计算过程。
2. 构造该文法的语法分析表。
3. 根据语法分析表模拟 LL(1) 语法分析过程。

4. 设计相应测试样例，并输出所给句子的语法分析过程。

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 LL(1) 分析的一般流程实现：先把题目给定文法表示成程序内部的数据结构，再迭代计算 FIRST 集和 FOLLOW 集，随后根据这两个集合构造预测分析表，最后使用分析栈和输入指针对测试串进行逐步分析，并输出每一步所使用的产生式或匹配动作。

1.1 设计思想

程序首先把文法表示成 Python 字典：键为非终结符，值为该非终结符对应的产生式右部列表；右部再表示成元组。为了让空产生式更容易处理，程序使用“空元组”表示 ε 产生式。整个实现以 `LL1Grammar` 类为核心，它负责完成以下四件事：

1. 计算单个符号串的 FIRST 集。

2. 通过不动点迭代法求所有非终结符的 FIRST 集和 FOLLOW 集。

3. 根据 FIRST 集、FOLLOW 集构造预测分析表，并检测是否出现冲突。

4. 使用预测分析表模拟输入串的 LL(1) 分析过程。

这种设计的好处是职责清楚：文法相关的算法都集中在一个类里，入口程序只负责接收输入并展示结果，不会把核心算法和终端输出混在一起。

1.2 实现流程

程序的整体执行流程如图所示。

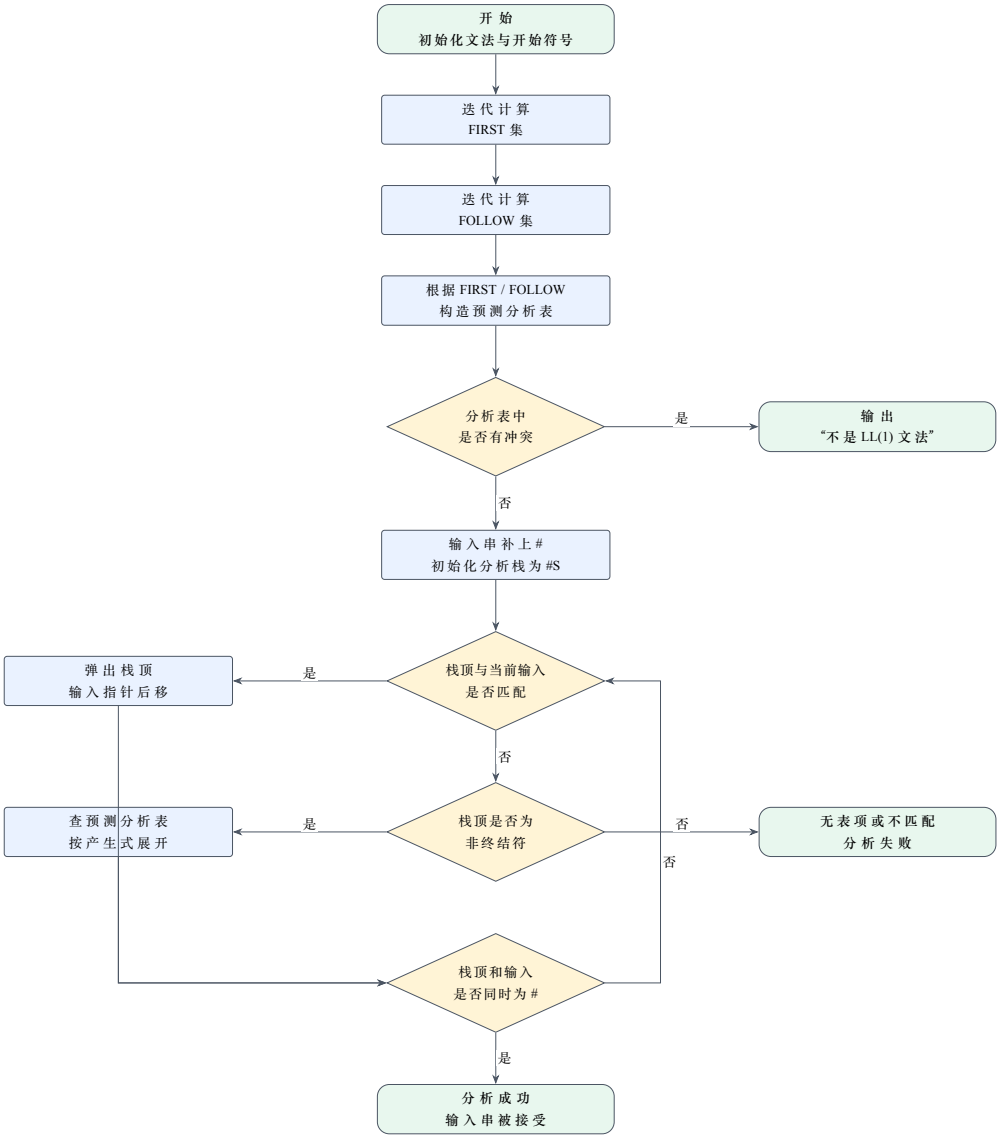


图 1: LL(1) 语法分析程序实现流程图

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，又始终围绕“查看栈顶、查看当前输入、决定匹配或展开”这条主线运行。这正是 LL(1) 自顶向下分析的核心思路。

1.3 核心算法说明

程序中最关键的四个算法如下：

1. **FIRST 集计算**：对每个产生式右部从左到右扫描。如果当前符号是终结符，就把它加入 FIRST 集；如果是非终结符，就把该非终结符的 FIRST 集（除去 ε）加入结果；若该非终结符可推出 ε，则继续看后继符号。

2. **FOLLOW 集计算**：把开始符号的 FOLLOW 集初始化为 #。随后遍历每条产生式，如果某个非终结符后面还有符号，就把后继符号串的 FIRST 集（除去 ε）加入它的 FOLLOW 集；若后继符号串可推出 ε，则把左部非终结符的 FOLLOW 集也传递给它。

3. **预测分析表构造**：对于产生式 $A \to \alpha$，把 $FIRST(\alpha) - \{\varepsilon\}$ 中的终结符全部填入表项 $M[A, a]$；如果 α 可以推出 ε，则把 $FOLLOW(A)$ 中的符号填入对应表项。若同一表项被填入不同产生式，则说明发生冲突。

4. **分析过程模拟**：初始时分析栈为 `#S`。每一步只看栈顶符号和当前输入符号：若二者相同，则匹配并同时前进；若栈顶是非终结符，则查询预测分析表并按产生式展开；若既不能匹配也没有表项，则分析失败。

为了降低理解难度，代码中的输出辅助函数和终端排版逻辑都与算法主体分离，附录里也只保留了核心代码片段。

2. 调试过程及实验结果

调试时，我主要关注三类问题：第一，FIRST 集和 FOLLOW 集是否计算正确；第二，预测分析表是否存在冲突；第三，输入串分析过程是否与人工推导一致。由于 LL(1) 实验比词法分析更依赖“集合传递”和“状态推进”的正确性，所以本次调试重点放在集合计算和步骤追踪上。

2.1 FIRST 集与 FOLLOW 集计算结果

程序自动计算得到的集合如下表所示。

表 1: 给定文法的 FIRST 集与 FOLLOW 集

非终结符	FIRST 集	FOLLOW 集
S	$\{a, b, d, e, \varepsilon\}$	$\{o, \#\}$
H	$\{e, \varepsilon\}$	$\{o, f, \#\}$
K	$\{d, \varepsilon\}$	$\{e, o, \#\}$
L	$\{e\}$	$\{a, b, d, e, o, \#\}$
M	$\{b, d, \varepsilon\}$	$\{e, o, \#\}$

这里有一个容易忽略的细节：程序计算出的 `FOLLOW(L)` 中包含 #。这是

因为 L 出现在产生式 $K \to dML$ 的末尾，而 K 又可以出现在整个句子的末尾，所以 # 会沿着 FOLLOW 集的传递关系进入 FOLLOW(L)。这个结果符合标准 FOLLOW 集算法。

2.2 LL(1) 判定与预测分析表

程序在构造预测分析表时，没有发现任何一个表项被不同产生式重复填写，因此该文法可以判定为 LL(1) 文法。构造出的预测分析表如表所示。

表 2: 该文法的 LL(1) 预测分析表

非终结符	a	b	d	e	o	f	#
S	$S \to a$	$S \to MH$	$S \to MH$	$S \to MH$	$S \to MH$	/	$S \to MH$
H	/	/	/	$H \to LSo$	$H \to \varepsilon$	$H \to \varepsilon$	$H \to \varepsilon$
K	/	/	$K \to dML$	$K \to \varepsilon$	$K \to \varepsilon$	/	$K \to \varepsilon$
L	/	/	/	$L \to eHf$	/	/	/
M	/	$M \to bLM$	$M \to K$	$M \to K$	$M \to K$	/	$M \to K$

从分析表可以直观看出：每个可用表项都唯一对应一条产生式，因此程序在进行预测时不会出现“同一步不知道该选哪条产生式”的情况，这也是该文法满足 LL(1) 条件的直接体现。

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。正确样例用于覆盖不同产生式分支，错误样例用于验证程序能否在无匹配表项或终结符不匹配时及时停止并给出结论。

表 3: 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	#	接受	验证开始符号经 $S \Rightarrow MH \Rightarrow \varepsilon$ 时，空串可以被接受。
2	a#	接受	验证产生式 $S \to a$ 的直接匹配过程。
3	bef#	接受	验证 $M \to bLM$、$L \to eHf$ 以及多个空产生式的组合情况。
4	def#	接受	验证 $K \to dML$ 分支，以及由预测分析表驱动的多步展开过程。
5	efao#	接受	验证 $H \to LSo$ 分支能够正确参与分析。
6	ao#	拒绝	先匹配到 a，但后续仍残留 o，导致栈顶与输入不匹配。
7	df#	拒绝	在分析到栈顶为 M、当前输入为 f 时，分析表中不存在表项。

下面选取一个正确样例 def# 和一个错误样例 ao# 展示具体分析过程。

表 4: 输入串 def# 的 LL(1) 分析过程

步骤	分析栈	剩余输入	动作
1	#S	def#	使用产生式 $S \to MH$
2	#HM	def#	使用产生式 $M \to K$
3	#HK	def#	使用产生式 $K \to dML$
4	#HLMd	def#	匹配终结符 d
5	#HLM	ef#	使用产生式 $M \to K$
6	#HLK	ef#	使用产生式 $K \to \varepsilon$
7	#HL	ef#	使用产生式 $L \to eHf$
8	#HfHe	ef#	匹配终结符 e
9	#HfH	f#	使用产生式 $H \to \varepsilon$
10	#Hf	f#	匹配终结符 f
11	#H	#	使用产生式 $H \to \varepsilon$
12	#	#	分析成功，输入串被接受

表 5: 输入串 ao# 的 LL(1) 分析过程

步骤	分析栈	剩余输入	动作
1	#S	ao#	使用产生式 $S \to a$
2	#a	ao#	匹配终结符 a
3	#	o#	栈顶符号 # 与当前输入 o 不匹配，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 FIRST 集、FOLLOW 集、LL(1) 判定结果以及预测分析表；第二张展示测试样例在预测分析表驱动下的具体分析过程。

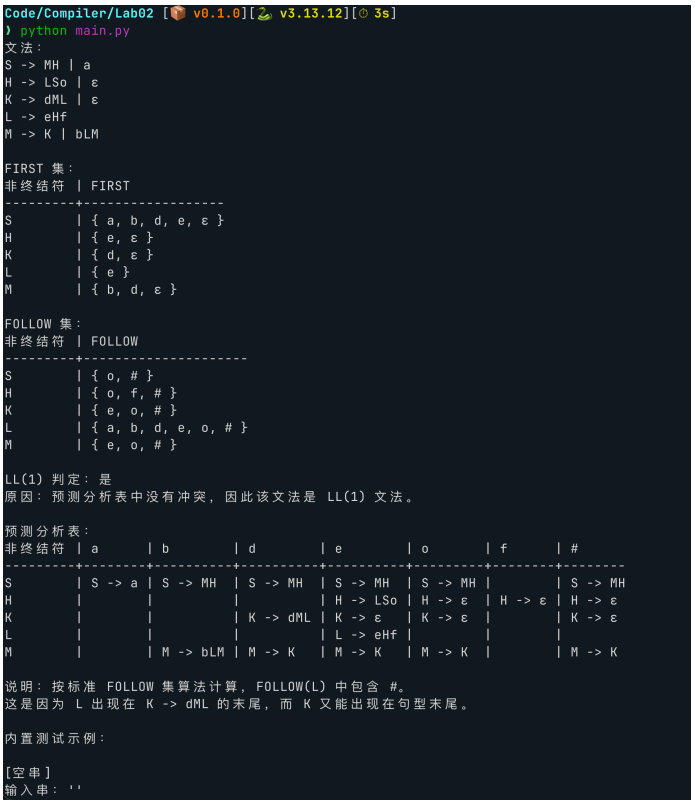
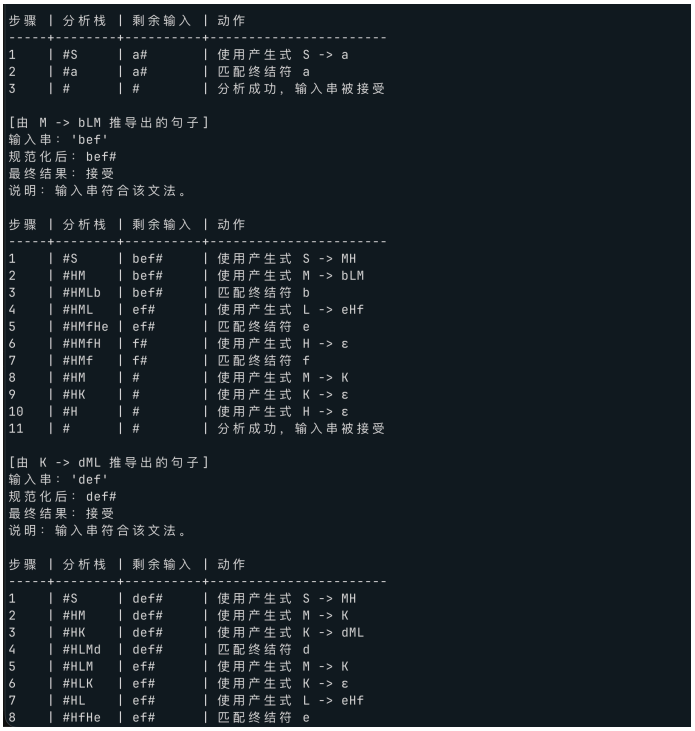


图 2: 程序输出 FIRST 集、FOLLOW 集与预测分析表的运行结果



```
步骤 | 分析栈 | 剩余输入 | 动作
-----+--------+----------+------------------------
1    | #S     | a#       | 使用产生式 S -> a
2    | #a     | a#       | 匹配终结符 a
3    | #      | #        | 分析成功，输入串被接受

[由 M -> bLM 推导出的句子]
输入串： 'bef'
规范化后： bef#
最终结果： 接受
说明： 输入串符合该文法。

步骤 | 分析栈 | 剩余输入 | 动作
-----+--------+----------+------------------------
1    | #S     | bef#     | 使用产生式 S -> MH
2    | #HM    | bef#     | 使用产生式 M -> bLM
3    | #HMLb  | bef#     | 匹配终结符 b
4    | #HML   | ef#      | 使用产生式 L -> eHf
5    | #HMfHe | ef#      | 匹配终结符 e
6    | #HMfH  | f#       | 使用产生式 H -> ε
7    | #HMf   | f#       | 匹配终结符 f
8    | #HM    | #        | 使用产生式 M -> K
9    | #HK    | #        | 使用产生式 K -> ε
10   | #H     | #        | 使用产生式 H -> ε
11   | #      | #        | 分析成功，输入串被接受

[由 K -> dML 推导出的句子]
输入串： 'def'
规范化后： def#
最终结果： 接受
说明： 输入串符合该文法。

步骤 | 分析栈 | 剩余输入 | 动作
-----+--------+----------+------------------------
1    | #S     | def#     | 使用产生式 S -> MH
2    | #HM    | def#     | 使用产生式 M -> K
3    | #HK    | def#     | 使用产生式 K -> dML
4    | #HLMd  | def#     | 匹配终结符 d
5    | #HLM   | ef#      | 使用产生式 M -> K
6    | #HLK   | ef#      | 使用产生式 K -> ε
7    | #HL    | ef#      | 使用产生式 L -> eHf
8    | #HfHe  | ef#      | 匹配终结符 e
```

图 3: 程序对测试输入串进行 LL(1) 分析的运行结果

2.5 结果分析

从集合计算、预测分析表和测试样例三方面来看，程序已经完整覆盖实验要求。首先，FIRST 集与 FOLLOW 集全部由程序自动计算，避免了手工推导容易出现的遗漏。其次，通过“表项是否冲突”这一标准判断方式，程序明确得出该文法是 LL(1) 文法。最后，在输入串分析阶段，程序能够清楚展示每一步的分析栈、剩余输入和所使用的产生式，因此不仅能得到“接受或拒绝”的最终结论，也能完整追踪分析过程。

3. 总结

本次实验完成了一个面向教学场景的 LL(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、LL(1) 判定、预测分析表构造以及输入串分析过程模拟。程序虽然规模不大，但结构完整、逻辑清楚，适合用于理解 LL(1) 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：LL(1) 分析的关键并不在“代码写得多复杂”，而在于是否严格按照文法规则、集合传播规则和分析表驱动规

则一步一步执行。只要 FIRST 集和 FOLLOW 集计算正确，预测分析表就能自然构造出来；只要分析栈和输入指针推进正确，整个语法分析过程就能稳定运行。因此，本实验很好地把课堂上的抽象概念转化成了可验证的程序行为。

3.1 上机感悟

通过这次上机，我对 LL(1) 分析的理解比单纯看书时清晰了很多。以前只是在纸上推导 FIRST 集、FOLLOW 集和预测分析表，感觉这些内容彼此独立；真正写成程序后才发现，它们其实是一条非常连贯的逻辑链：先有 FIRST 集和 FOLLOW 集，才能构造分析表；有了分析表，才能驱动后续的语法分析过程。尤其是在调试过程中，我更加体会到编译原理里的每一条规则都不是孤立存在的，它们最终都会落实到程序的某一步动作中。通过自己实现并验证这些过程，我不仅掌握了 LL(1) 分析的基本方法，也增强了把抽象算法翻译成清晰代码的能力。

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、结果拼接等辅助函数不再展开。

4.1 LL(1) 分析核心程序节选 `ll1.py`

```python
class LL1Grammar:
    """一个尽量小而清晰的 LL(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())

        auto_terminals: list[str] = []
        for alternatives in productions.values():
            for right in alternatives:
                for symbol in right:
                    if symbol in productions or symbol == EPSILON
```

```
    :
                        continue
                    if symbol not in auto_terminals:
                        auto_terminals.append(symbol)

        self.terminals = terminal_order or auto_terminals

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

    def first_of_sequence(
        self,
        symbols: tuple[str, ...],
        first_sets: dict[str, set[str]],
    ) -> set[str]:
        """
        计算一个符号串的 FIRST 集。

        例如:
        - FIRST(MH)
        - FIRST(LSo)
        - FIRST(ε)
        """
        if not symbols:
            return {EPSILON}

        result: set[str] = set()

        for symbol in symbols:
            if not self.is_nonterminal(symbol):
                result.add(symbol)
                return result

            result.update(first_sets[symbol] - {EPSILON})

            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所有非终结符的 FIRST 集。"""
        first_sets = {nonterminal: set() for nonterminal in self.
```

```
    nonterminals}
63
64          changed = True
65          while changed:
66              changed = False
67
68              for left, alternatives in self.productions.items():
69                  for right in alternatives:
70                      before_size = len(first_sets[left])
71                      first_sets[left].update(self.
    first_of_sequence(right, first_sets))
72                      if len(first_sets[left]) != before_size:
73                          changed = True
74
75          return first_sets
76
77      def compute_follow_sets(
78          self,
79          first_sets: dict[str, set[str]],
80      ) -> dict[str, set[str]]:
81          """使用不动点迭代法计算所有非终结符的 FOLLOW 集。"""
82          follow_sets = {nonterminal: set() for nonterminal in self
    .nonterminals}
83          follow_sets[self.start_symbol].add(END_MARKER)
84
85          changed = True
86          while changed:
87              changed = False
88
89              for left, alternatives in self.productions.items():
90                  for right in alternatives:
91                      for index, symbol in enumerate(right):
92                          if not self.is_nonterminal(symbol):
93                              continue
94
95                          beta = right[index + 1 :]
96                          first_beta = self.first_of_sequence(beta,
     first_sets)
97
98                          before_size = len(follow_sets[symbol])
99                          follow_sets[symbol].update(first_beta - {
    EPSILON})
100
101                         if EPSILON in first_beta:
102                             follow_sets[symbol].update(
```

```
        follow_sets[left])
103
104                         if len(follow_sets[symbol]) !=
        before_size:
105                             changed = True
106
107             return follow_sets
108
109         def build_parsing_table(
110             self,
111             first_sets: dict[str, set[str]],
112             follow_sets: dict[str, set[str]],
113         ) -> tuple[dict[str, dict[str, Production | None]], list[str
        ]]:
114             """
115             构造预测分析表。
116
117             返回值包含两部分：
118             1. 预测分析表本身
119             2. 冲突说明列表，若为空则说明该文法可视为 LL(1)
120             """
121             table = {
122                 nonterminal: {terminal: None for terminal in self.
        terminals + [END_MARKER]}
123                 for nonterminal in self.nonterminals
124             }
125             conflicts: list[str] = []
126
127             for left, alternatives in self.productions.items():
128                 for right in alternatives:
129                     production = Production(left, right)
130                     first_alpha = self.first_of_sequence(right,
        first_sets)
131
132                     for terminal in first_alpha - {EPSILON}:
133                         self._put_table_entry(table, terminal,
        production, conflicts)
134
135                     if EPSILON in first_alpha:
136                         for terminal in follow_sets[left]:
137                             self._put_table_entry(table, terminal,
        production, conflicts)
138
139             return table, conflicts
140
```

```
    def _put_table_entry(
        self,
        table: dict[str, dict[str, Production | None]],
        terminal: str,
        production: Production,
        conflicts: list[str],
    ) -> None:
        """向预测分析表中放入一条表项，并检查是否冲突。"""
        current = table[production.left][terminal]

        if current is None:
            table[production.left][terminal] = production
            return

        if current != production:
            conflicts.append(
                f"表项 M[{production.left}, {terminal}] 同时出现 "
                f"{current.display()} 和 {production.display()}"
            )

    def analyze_sentence(
        self,
        sentence: str,
        table: dict[str, dict[str, Production | None]],
    ) -> ParseResult:
        """根据预测分析表，模拟一个输入串的 LL(1) 分析过程。"""
        normalized_sentence = normalize_sentence(sentence)
        symbols = list(normalized_sentence)
        allowed_symbols = set(self.terminals + [END_MARKER])

        for symbol in symbols:
            if symbol 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：{symbol}"
,
                    steps=[],
                )

        stack = [END_MARKER, self.start_symbol]
        pointer = 0
        steps: list[AnalysisStep] = []
        step_no = 1
```

```
185
186         while True:
187             stack_snapshot = "".join(stack)
188             remaining_input = "".join(symbols[pointer:])
189             top = stack[-1]
190             current = symbols[pointer]
191
192             if top == current == END_MARKER:
193                 steps.append(
194                     AnalysisStep(step_no, stack_snapshot,
    remaining_input, "分析成功，输入串被接受")
195                 )
196                 return ParseResult(
197                     original_sentence=sentence,
198                     normalized_sentence=normalized_sentence,
199                     accepted=True,
200                     message="输入串符合该文法。",
201                     steps=steps,
202                 )
203
204             if not self.is_nonterminal(top):
205                 if top == current:
206                     steps.append(
207                         AnalysisStep(
208                             step_no,
209                             stack_snapshot,
210                             remaining_input,
211                             f"匹配终结符 {current}",
212                         )
213                     )
214                     stack.pop()
215                     pointer += 1
216                     step_no += 1
217                     continue
218
219                 steps.append(
220                     AnalysisStep(
221                         step_no,
222                         stack_snapshot,
223                         remaining_input,
224                         f"栈顶符号 {top} 与当前输入 {current} 不匹配，
    分析失败",
225                     )
226                 )
227                 return ParseResult(
```

```
                    original_sentence=sentence,
                    normalized_sentence=normalized_sentence,
                    accepted=False,
                    message="栈顶终结符与当前输入不匹配。",
                    steps=steps,
                )

            production = table[top].get(current)
            if production is None:
                steps.append(
                    AnalysisStep(
                        step_no,
                        stack_snapshot,
                        remaining_input,
                        f"预测分析表中不存在 M[{top}, {current}]，分析失败",
                    )
                )
                return ParseResult(
                    original_sentence=sentence,
                    normalized_sentence=normalized_sentence,
                    accepted=False,
                    message="预测分析表中没有对应表项。",
                    steps=steps,
                )

            steps.append(
                AnalysisStep(
                    step_no,
                    stack_snapshot,
                    remaining_input,
                    f"使用产生式 {production.display()}",
                )
            )

            stack.pop()
            for symbol in reversed(production.right):
                stack.append(symbol)

            step_no += 1


def build_default_grammar() -> LL1Grammar:
    """构造实验题目中给定的文法。"""
    productions = {
```

```
        "S": [("M", "H"), ("a",)],
        "H": [("L", "S", "o"), ()],
        "K": [("d", "M", "L"), ()],
        "L": [("e", "H", "f")],
        "M": [("K",), ("b", "L", "M")],
    }

    # 这里使用题目示例中更常见的列顺序，方便和实验报告对照。
    terminal_order = ["a", "b", "d", "e", "o", "f"]
    return LL1Grammar(productions=productions, start_symbol="S",
    terminal_order=terminal_order)


DEFAULT_CASES = [
    ("空串", ""),
    ("最简单的句子", "a"),
    ("由 M -> bLM 推导出的句子", "bef"),
    ("由 K -> dML 推导出的句子", "def"),
    ("包含 H -> LSo 的句子", "efao"),
    ("错误示例 1", "ao"),
    ("错误示例 2", "df"),
]


def normalize_sentence(sentence: str) -> str:
    """去掉输入中的空白字符，并在末尾补上结束符 #。"""
    cleaned = "".join(sentence.split())
    if not cleaned.endswith(END_MARKER):
        cleaned += END_MARKER
    return cleaned
```

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。
# 如果只写一个 #，表示空串。
#
a#
bef#
def#
efao#
```

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。
ao#
df#
bo#
```