# 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　3　月　28　日

# 信息与软件工程学院上机实验报告

# （第 1 次）

## 一、实验名称

实验一：词法分析

## 二、实验目的及要求

本次实验的目标是完成一个基础的词法分析程序，使其能够对源程序进行逐字符扫描，并把识别结果表示为“内码 + 属性值”的二元式，为后续语法分析阶段提供输入。通过本次上机，我希望把课堂上关于词法分析的概念真正落实到代码里，理解 token 的划分方法、错误定位方式以及标识符表在编译前端中的作用。

结合实验任务，程序需要完成以下要求：

1. 从源程序文件中读入字符。
2. 统计行数和列数用于错误单词的定位。
3. 删除空格类字符，包括回车、制表符空格。
4. 按拼写单词，并用（内码，属性）二元式表示。(属性值——token 的机内表示)
5. 如果发现错误则报告出错 7
6. 根据需要是否填写标识符表供以后各阶段使用。

## 三、实验环境

1. 操作系统：Linux
2. 编程语言：Python 3.9.6
3. 开发工具：Visual Studio Code

# 四、实验设计

## 1. 实验步骤

本实验采用“单指针线性扫描”的方式实现词法分析。程序只维护一个当前扫描位置，并同步记录当前行号和列号，然后根据当前字符的类型进入不同的识别流程。为了尽量贴合实验要求，同时保证代码简洁易懂，我把各类识别逻辑拆成多个职责单一的小函数，这样阅读和调试都会轻松很多。

1.1 设计思想

程序核心为 `Lexer` 类。这个类接收源程序全文，内部维护当前扫描位置，并在分析过程中持续生成三个结果：token 列表、错误列表和标识符表。为了避免把所有逻辑都堆在一个长函数里，我按单词类型把识别过程拆分成几个小步骤：

1. 先判断并跳过空白字符和注释。
2. 如果当前字符是字母或下划线，则识别关键字或标识符。
3. 如果当前字符是数字，则识别整数或实数，并检查非法数字常量。
4. 如果当前字符是单引号或双引号，则分别识别字符常量和字符串常量。
5. 如果当前字符属于运算符或界符，则优先匹配双字符运算符，再匹配单字符。
6. 如果不属于以上任何情况，则记录词法错误，并给出具体行列位置。

这种实现方式的优点是结构清楚、定位问题方便，也更符合词法分析“从左到右、逐类判断”的基本思路。

1.2 实现流程

为了更直观地说明程序执行过程，实验流程采用流程图表示如下。

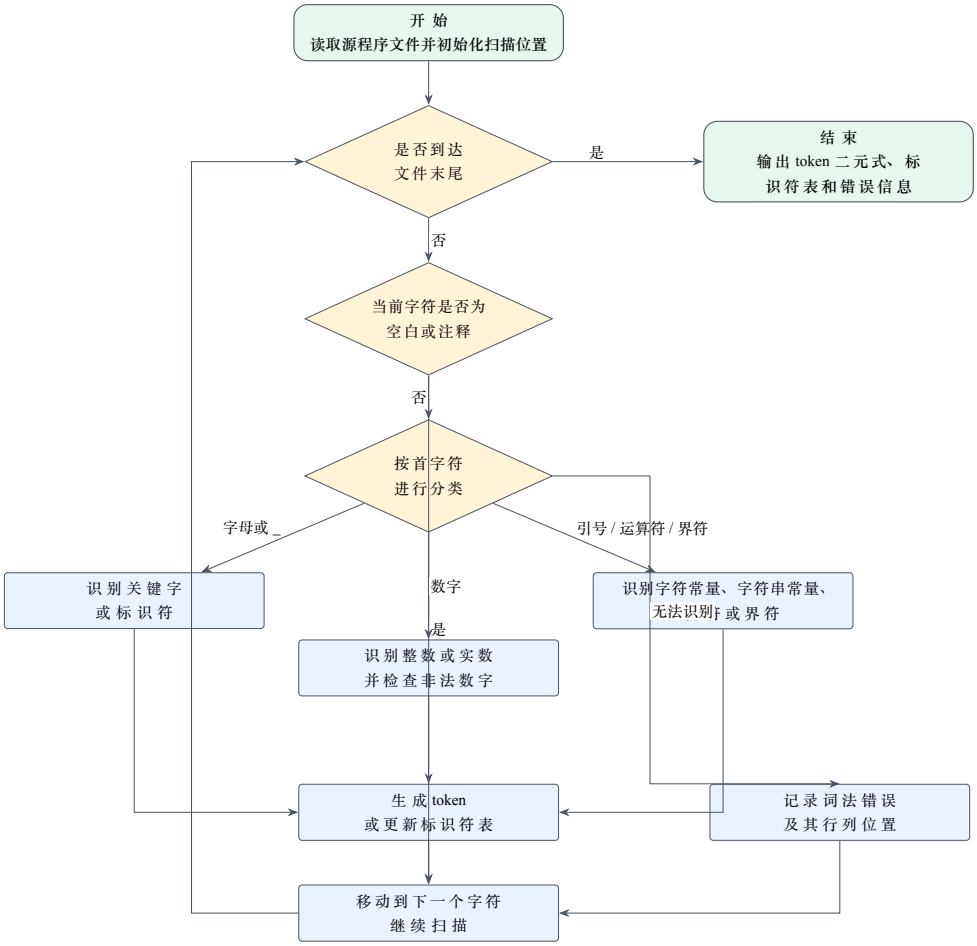



图 1: 词法分析程序实现流程图

从流程图可以看出，程序始终围绕“读取字符、分类判断、生成结果、继续扫描”这条主线运行。这样不仅能够保证识别顺序清楚，也便于在出现错误时快速定位问题所在。

1.3 内码与数据结构设计

为了让输出结果更便于观察和整理，程序对不同种别的单词设计了统一的内码规则：关键字使用 `1--11`，运算符使用 `101--122`，界符使用 `201--209`，标识符、整数、实数、字符常量、字符串常量分别使用 `300--304`。其中，关键字和运算符的属性值记为 `-`，标识符的属性值是它在标识符表中的编号，常量的属性值记录其实际内容或数值。

此外，程序使用列表保存 token 和错误，使用“列表 + 字典”的方式维护标识符表。列表负责按加入顺序输出，字典负责快速判断某个标识符是否已经出现过。这样既能保证结果稳定，也能避免重复插入。

## 2. 调试过程及实验结果

在调试阶段，我主要围绕“常见单词能否正确识别”和“非法输入能否准确报错”这两个方向进行测试。开始时，程序已经能够完成一般关键字、标识符和整数的识别，但在一些边界情况上还不够完善，例如非法数字常量、未闭合字符常量以及非法字符的处理。后续通过针对性样例逐项排查，程序逐渐稳定下来。

### 2.1 测试样例选择

为了覆盖主要功能，本次实验选择了两个测试文件：

1. 正常样例 `sample.c`：用于验证关键字、标识符、整数、实数、运算符、界符和注释的识别效果。
2. 错误样例 `error_sample.c`：用于验证非法数字常量、未结束字符常量和非法字符的错误定位能力。

### 2.2 实验结果

表 1: 测试样例与运行结果汇总

| 测试编号 | 输入文件 | 测试目的 | 实际结果 |
| --- | --- | --- | --- |
| 1 | `sample.c` | 验证常见 token 的正常识别，包括关键字、标识符、常量、运算符、界符和注释。 | 成功识别 33 个 token，建立 3 个标识符，无词法错误。 |
| 2 | `error_sample.c` | 验证错误定位功能，包括非法数字、未结束字符常量和非法字符。 | 成功识别 21 个 token，建立 3 个标识符，报告 3 个词法错误，错误位置分别为第 2 行第 9 列、第 3 行第 14 列和第 4 行第 16 列。 |

### 2.3 运行截图

为了更直观地展示程序运行效果，我分别保留了正常样例和错误样例的终端输出截图。正常样例主要用于说明程序能够正确生成 token 二元式和标识符表；错误样例则用于展示程序对非法输入的定位和报错能力。

```
Code/Compiler/Lab01 [📦 v0.1.0][🐍 v3.13.12]
❯ uv run python main.py examples/sample.c
实验一：词法分析结果
源文件：examples/sample.c

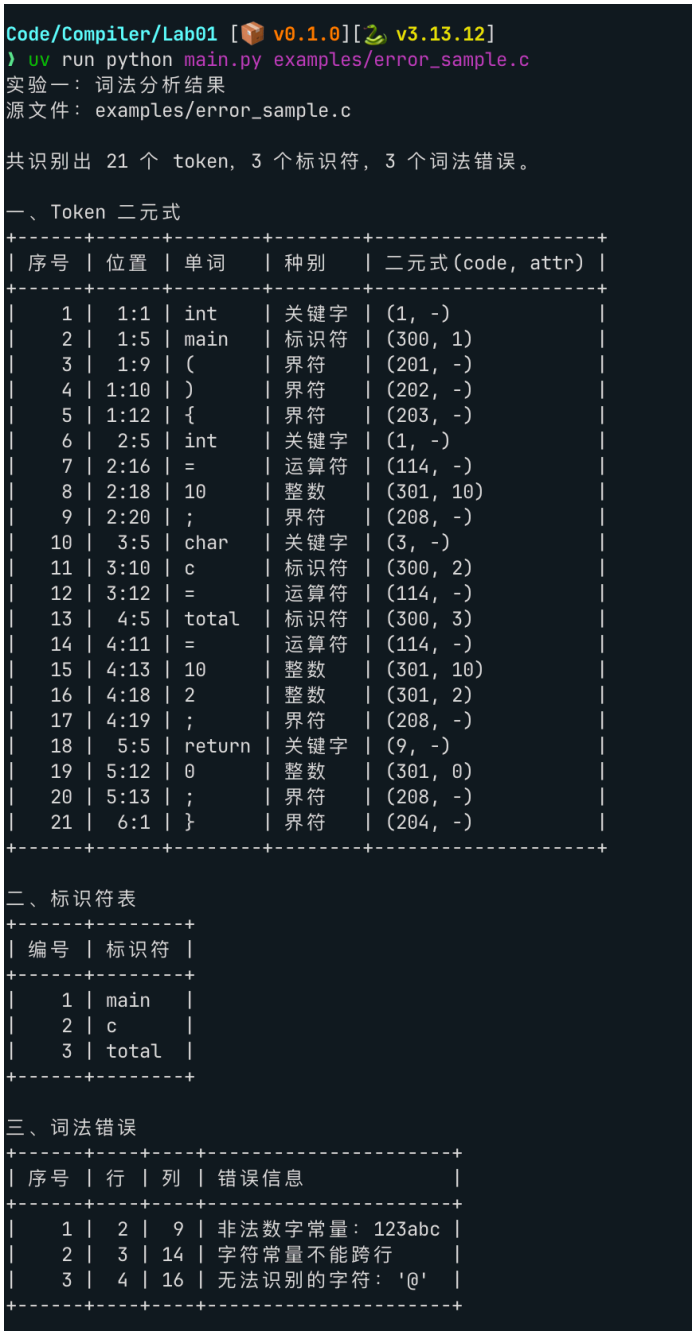
共识别出 33 个 token，3 个标识符，0 个词法错误。

一、Token 二元式
+------+-------+--------+--------+--------------------+
| 序号 | 位置  | 单词   | 种别   | 二元式(code, attr) |
+------+-------+--------+--------+--------------------+
|    1 |   1:1 | int    | 关键字 | (1, -)             |
|    2 |   1:5 | main   | 标识符 | (300, 1)           |
|    3 |   1:9 | (      | 界符   | (201, -)           |
|    4 |  1:10 | )      | 界符   | (202, -)           |
|    5 |  1:12 | {      | 界符   | (203, -)           |
|    6 |   2:5 | int    | 关键字 | (1, -)             |
|    7 |   2:9 | total  | 标识符 | (300, 2)           |
|    8 |  2:15 | =      | 运算符 | (114, -)           |
|    9 |  2:17 | 12     | 整数   | (301, 12)          |
|   10 |  2:20 | +      | 运算符 | (115, -)           |
|   11 |  2:22 | 8      | 整数   | (301, 8)           |
|   12 |  2:23 | ;      | 界符   | (208, -)           |
|   13 |   3:5 | float  | 关键字 | (2, -)             |
|   14 |  3:11 | rate   | 标识符 | (300, 3)           |
|   15 |  3:16 | =      | 运算符 | (114, -)           |
|   16 |  3:18 | 3.5    | 实数   | (302, 3.5)         |
|   17 |  3:21 | ;      | 界符   | (208, -)           |
|   18 |   6:5 | if     | 关键字 | (5, -)             |
|   19 |   6:8 | (      | 界符   | (201, -)           |
|   20 |   6:9 | total  | 标识符 | (300, 2)           |
|   21 |  6:15 | >=     | 运算符 | (104, -)           |
|   22 |  6:18 | 20     | 整数   | (301, 20)          |
|   23 |  6:20 | )      | 界符   | (202, -)           |
|   24 |  6:22 | {      | 界符   | (203, -)           |
|   25 |   7:9 | total  | 标识符 | (300, 2)           |
|   26 |  7:15 | +=     | 运算符 | (109, -)           |
|   27 |  7:18 | 1      | 整数   | (301, 1)           |
|   28 |  7:19 | ;      | 界符   | (208, -)           |
|   29 |   8:5 | }      | 界符   | (204, -)           |
|   30 |  10:5 | return | 关键字 | (9, -)             |
|   31 | 10:12 | total  | 标识符 | (300, 2)           |
|   32 | 10:17 | ;      | 界符   | (208, -)           |
|   33 |  11:1 | }      | 界符   | (204, -)           |
+------+-------+--------+--------+--------------------+

二、标识符表
+------+--------+
| 编号 | 标识符 |
+------+--------+
|    1 | main   |
|    2 | total  |
|    3 | rate   |
+------+--------+

三、词法错误
未发现词法错误。
```

图 2: 正常样例的词法分析输出结果



```
Code/Compiler/Lab01 [📦 v0.1.0][🐍 v3.13.12]
❯ uv run python main.py examples/error_sample.c
实验一：词法分析结果
源文件：examples/error_sample.c

共识别出 21 个 token，3 个标识符，3 个词法错误。

一、Token 二元式
+------+------+--------+--------+--------------------+
| 序号 | 位置 | 单词   | 种别   | 二元式(code, attr) |
+------+------+--------+--------+--------------------+
|    1 |  1:1 | int    | 关键字 | (1, -)             |
|    2 |  1:5 | main   | 标识符 | (300, 1)           |
|    3 |  1:9 | (      | 界符   | (201, -)           |
|    4 | 1:10 | )      | 界符   | (202, -)           |
|    5 | 1:12 | {      | 界符   | (203, -)           |
|    6 |  2:5 | int    | 关键字 | (1, -)             |
|    7 | 2:16 | =      | 运算符 | (114, -)           |
|    8 | 2:18 | 10     | 整数   | (301, 10)          |
|    9 | 2:20 | ;      | 界符   | (208, -)           |
|   10 |  3:5 | char   | 关键字 | (3, -)             |
|   11 | 3:10 | c      | 标识符 | (300, 2)           |
|   12 | 3:12 | =      | 运算符 | (114, -)           |
|   13 |  4:5 | total  | 标识符 | (300, 3)           |
|   14 | 4:11 | =      | 运算符 | (114, -)           |
|   15 | 4:13 | 10     | 整数   | (301, 10)          |
|   16 | 4:18 | 2      | 整数   | (301, 2)           |
|   17 | 4:19 | ;      | 界符   | (208, -)           |
|   18 |  5:5 | return | 关键字 | (9, -)             |
|   19 | 5:12 | 0      | 整数   | (301, 0)           |
|   20 | 5:13 | ;      | 界符   | (208, -)           |
|   21 |  6:1 | }      | 界符   | (204, -)           |
+------+------+--------+--------+--------------------+

二、标识符表
+------+--------+
| 编号 | 标识符 |
+------+--------+
|    1 | main   |
|    2 | c      |
|    3 | total  |
+------+--------+

三、词法错误
+------+----+----+----------------------+
| 序号 | 行 | 列 | 错误信息             |
+------+----+----+----------------------+
|    1 |  2 |  9 | 非法数字常量：123abc |
|    2 |  3 | 14 | 字符常量不能跨行     |
|    3 |  4 | 16 | 无法识别的字符：'@'  |
+------+----+----+----------------------+
```

图 3: 错误样例的词法分析输出结果

### 2.4 结果分析

从测试结果和运行截图可以看出，程序已经能够满足实验要求中的核心功能。对于正常输入，程序可以稳定输出二元式，并生成标识符表；对于错误输入，程序能够在不影响整体扫描流程的前提下给出清楚的中文报错信息和准确的行列位置。除此之外，输出层还额外做了一个小优化：通过自动计算列宽，把token、标识符和错误信息渲染成更整齐的纯文本表格，使终端结果更清晰，也

更方便直接整理到实验记录中。

## 3. 总结

本次实验完成了一个面向教学场景的词法分析程序，整体实现覆盖了实验要求中的核心任务，包括源文件读取、空白字符跳过、关键字与标识符识别、整数与实数识别、字符常量与字符串常量识别、运算符与界符识别，以及词法错误的行列定位与输出。程序最终能够以“内码 + 属性值”的二元式形式输出 token 序列，并维护标识符表，为后续语法分析阶段提供结构清晰的输入结果。

从实现角度看，本实验进一步验证了词法分析的基本原理，即按照“逐字符读取、按类别分支、生成单词结果、继续向前扫描”的方式对源程序进行线性处理。为了保证分析过程稳定可靠，程序在识别时采用了若干关键策略，例如优先匹配双字符运算符、对非法数字常量进行整体报错、对未闭合字符常量和字符串常量进行及时终止与定位。这些处理使程序不仅能够识别正常输入，也能够较准确地应对常见错误输入。

综合来看，本实验已经较好地实现了词法分析器的基础功能，也让我对编译前端中 token 生成、属性表示和错误恢复这些关键问题形成了更系统的认识。若后续继续扩展，可以进一步补充更多关键字类型、支持更多数值形式，并将分析结果导出为文件，以增强程序的完整性与实用性。

### 3.1 上机感悟

通过这次上机实践，我更加深刻地体会到了“动手做一遍”和“只在课堂上理解概念”之间的差别。词法分析在课本里看起来只是把源程序拆分成若干个 token，但真正写成程序以后，我才发现其中包含了很多细致而严谨的判断过程，例如如何逐字符扫描、如何区分关键字和标识符、如何维护行列号、以及在遇到错误时怎样做到准确定位。正是在不断调试和修正这些细节的过程中，我对编译原理这门课程的理解变得更加具体，也更加扎实。这次实验让我认识到，上机并不是对课堂内容的简单重复，而是把理论知识真正转化为工程能力的重要环节。通过亲手完成词法分析器，我不仅学会了如何把抽象规则落实到代码中，也增强了自己分析问题、定位问题和改进程序的信心。我觉得这样的上机实践

非常有价值，它让我看到了编译原理的实际应用意义，也激发了我继续深入学习后续语法分析和语义分析内容的兴趣。

## 4. 附录

### 4.1 词法分析核心程序 lexer.py

```python
from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path
import unicodedata

# 关键字的内码。
KEYWORD_CODES = {
    "int": 1,
    "float": 2,
    "char": 3,
    "void": 4,
    "if": 5,
    "else": 6,
    "while": 7,
    "for": 8,
    "return": 9,
    "break": 10,
    "continue": 11,
}

# 运算符的内码。
OPERATOR_CODES = {
    "==": 101,
    "!=": 102,
    "<=": 103,
    ">=": 104,
    "&&": 105,
    "||": 106,
    "++": 107,
    "--": 108,
    "+=": 109,
    "-=": 110,
    "*=": 111,
    "/=": 112,
    "%=": 113,
    "=": 114,
    "+": 115,
```

```
    "-": 116,
    "*": 117,
    "/": 118,
    "%": 119,
    "<": 120,
    ">": 121,
    "!": 122,
}

# 界符的内码。
DELIMITER_CODES = {
    "(": 201,
    ")": 202,
    "{": 203,
    "}": 204,
    "[": 205,
    "]": 206,
    ",": 207,
    ";": 208,
    ".": 209,
}

# 其他单词种别的内码。
SPECIAL_CODES = {
    "IDENTIFIER": 300,
    "INTEGER": 301,
    "REAL": 302,
    "CHAR": 303,
    "STRING": 304,
}

# 为了优先识别双字符运算符，需要按长度从长到短匹配。
SORTED_OPERATORS = sorted(OPERATOR_CODES, key=len, reverse=True)


@dataclass
class Token:
    """一个 token 就是词法分析得到的最小语法单元。"""

    kind: str
    lexeme: str
    code: int
    attribute: object
    line: int
    column: int
```

```
84
85
86  @dataclass
87  class LexicalError:
88      """记录词法错误，方便最后统一输出。"""
89
90      message: str
91      line: int
92      column: int
93
94
95  class Lexer:
96      """一个尽量小而清晰的词法分析器。"""
97
98      def __init__(self, text: str) -> None:
99          self.text = text
100         self.index = 0
101         self.line = 1
102         self.column = 1
103
104         self.tokens: list[Token] = []
105         self.errors: list[LexicalError] = []
106
107         # 标识符表使用“列表 + 映射”保存。
108         # 列表用于按顺序输出，映射用于快速查重。
109         self.symbol_table: list[str] = []
110         self.symbol_index: dict[str, int] = {}
111
112     def analyze(self) -> tuple[list[Token], list[LexicalError],
        list[str]]:
113         """循环扫描整个源程序，直到读到文件末尾。"""
114         while (current := self.current_char()) is not None:
115             if current.isspace():
116                 self.skip_whitespace()
117                 continue
118
119             if current == "/" and self.peek() == "/":
120                 self.skip_single_line_comment()
121                 continue
122
123             if current == "/" and self.peek() == "*":
124                 self.skip_multi_line_comment()
125                 continue
126
127             if current.isalpha() or current == "_":
```

```
                self.scan_identifier_or_keyword()
                continue

            if current.isdigit():
                self.scan_number()
                continue

            if current == "'":
                self.scan_char_literal()
                continue

            if current == '"':
                self.scan_string_literal()
                continue

            if self.scan_operator_or_delimiter():
                continue

            # 走到这里，说明当前字符无法被识别，属于词法错误。
            self.add_error(f"无法识别的字符：{current!r}", self.line
, self.column)
            self.advance()

        return self.tokens, self.errors, self.symbol_table

    def current_char(self) -> str | None:
        """返回当前字符；如果越界则返回 None。"""
        if self.index >= len(self.text):
            return None
        return self.text[self.index]

    def peek(self, offset: int = 1) -> str | None:
        """向前偷看若干个字符，但不真正移动位置。"""
        target = self.index + offset
        if target >= len(self.text):
            return None
        return self.text[target]

    def advance(self) -> str | None:
        """读取当前字符，并把扫描位置移动到下一个字符。"""
        current = self.current_char()
        if current is None:
            return None

        self.index += 1
```

```
172             if current == "\n":
173                 self.line += 1
174                 self.column = 1
175             else:
176                 self.column += 1
177             return current
178 
179         def add_error(self, message: str, line: int, column: int) ->
    None:
180             """添加一条词法错误。"""
181             self.errors.append(LexicalError(message, line, column))
182 
183         def add_token(
184             self,
185             kind: str,
186             lexeme: str,
187             code: int,
188             attribute: object,
189             line: int,
190             column: int,
191         ) -> None:
192             """添加一条 token。"""
193             self.tokens.append(Token(kind, lexeme, code, attribute,
    line, column))
194 
195         def skip_whitespace(self) -> None:
196             """跳过空格、换行、制表符等空白字符。"""
197             while (current := self.current_char()) is not None and
    current.isspace():
198                 self.advance()
199 
200         def skip_single_line_comment(self) -> None:
201             """跳过 // 注释。"""
202             self.advance()
203             self.advance()
204             while (current := self.current_char()) is not None and
    current != "\n":
205                 self.advance()
206 
207         def skip_multi_line_comment(self) -> None:
208             """跳过 /* ... */ 注释；如果注释没有结束则报错。"""
209             start_line, start_column = self.line, self.column
210             self.advance()
211             self.advance()
212 
```

```
213         while (current := self.current_char()) is not None:
214             if current == "*" and self.peek() == "/":
215                 self.advance()
216                 self.advance()
217                 return
218             self.advance()
219 
220         self.add_error("多行注释没有正常结束", start_line,
    start_column)
221 
222     def scan_identifier_or_keyword(self) -> None:
223         """识别关键字或标识符。"""
224         start_line, start_column = self.line, self.column
225         chars: list[str] = []
226 
227         while (current := self.current_char()) is not None and (
228             current.isalnum() or current == "_"
229         ):
230             chars.append(self.advance() or "")
231 
232         lexeme = "".join(chars)
233 
234         if lexeme in KEYWORD_CODES:
235             self.add_token(
236                 kind="关键字",
237                 lexeme=lexeme,
238                 code=KEYWORD_CODES[lexeme],
239                 attribute="-",
240                 line=start_line,
241                 column=start_column,
242             )
243             return
244 
245         attribute = self.intern_identifier(lexeme)
246         self.add_token(
247             kind="标识符",
248             lexeme=lexeme,
249             code=SPECIAL_CODES["IDENTIFIER"],
250             attribute=attribute,
251             line=start_line,
252             column=start_column,
253         )
254 
255     def scan_number(self) -> None:
256         """识别整数或实数。"""
```

```
257         start_line, start_column = self.line, self.column
258         chars: list[str] = []
259         is_real = False
260 
261         while (current := self.current_char()) is not None and
    current.isdigit():
262             chars.append(self.advance() or "")
263 
264         # 形如 12.34 的单词识别为实数。
265         if self.current_char() == "." and (next_char := self.peek
    ()) is not None and next_char.isdigit():
266             is_real = True
267             chars.append(self.advance() or "")
268             while (current := self.current_char()) is not None
    and current.isdigit():
269                 chars.append(self.advance() or "")
270 
271         # 形如 123abc 属于非法数字。
272         if (current := self.current_char()) is not None and (
273             current.isalpha() or current == "_"
274         ):
275             while (current := self.current_char()) is not None
    and (
276                 current.isalnum() or current == "_"
277             ):
278                 chars.append(self.advance() or "")
279             bad_lexeme = "".join(chars)
280             self.add_error(f"非法数字常量：{bad_lexeme}",
    start_line, start_column)
281             return
282 
283         lexeme = "".join(chars)
284         if is_real:
285             self.add_token(
286                 kind="实数",
287                 lexeme=lexeme,
288                 code=SPECIAL_CODES["REAL"],
289                 attribute=float(lexeme),
290                 line=start_line,
291                 column=start_column,
292             )
293         else:
294             self.add_token(
295                 kind="整数",
296                 lexeme=lexeme,
```

```
                code=SPECIAL_CODES["INTEGER"],
                attribute=int(lexeme),
                line=start_line,
                column=start_column,
            )

    def scan_char_literal(self) -> None:
        """识别字符常量，例如 'a' 或 '\\n'。"""
        start_line, start_column = self.line, self.column
        self.advance()
        chars: list[str] = []

        while (current := self.current_char()) is not None:
            if current == "\n":
                self.add_error("字符常量不能跨行", start_line,
start_column)
                return

            if current == "\\":
                chars.append(self.advance() or "")
                if self.current_char() is None:
                    break
                chars.append(self.advance() or "")
                continue

            if current == "'":
                self.advance()
                lexeme = "'" + "".join(chars) + "'"
                content = "".join(chars)
                if not content:
                    self.add_error("空字符常量不合法", start_line,
start_column)
                    return
                self.add_token(
                    kind="字符常量",
                    lexeme=lexeme,
                    code=SPECIAL_CODES["CHAR"],
                    attribute=content,
                    line=start_line,
                    column=start_column,
                )
                return

            chars.append(self.advance() or "")

```

```
340             self.add_error("字符常量没有正常结束", start_line,
    start_column)
341
342         def scan_string_literal(self) -> None:
343             """识别字符串常量，例如 \"hello\"。"""
344             start_line, start_column = self.line, self.column
345             self.advance()
346             chars: list[str] = []
347
348             while (current := self.current_char()) is not None:
349                 if current == "\n":
350                     self.add_error("字符串常量不能跨行", start_line,
    start_column)
351                     return
352
353                 if current == "\\":
354                     chars.append(self.advance() or "")
355                     if self.current_char() is None:
356                         break
357                     chars.append(self.advance() or "")
358                     continue
359
360                 if current == '"':
361                     self.advance()
362                     lexeme = '"' + "".join(chars) + '"'
363                     self.add_token(
364                         kind="字符串常量",
365                         lexeme=lexeme,
366                         code=SPECIAL_CODES["STRING"],
367                         attribute="".join(chars),
368                         line=start_line,
369                         column=start_column,
370                     )
371                     return
372
373                 chars.append(self.advance() or "")
374
375             self.add_error("字符串常量没有正常结束", start_line,
    start_column)
376
377         def scan_operator_or_delimiter(self) -> bool:
378             """识别运算符或界符。成功返回 True，失败返回 False。"""
379             start_line, start_column = self.line, self.column
380
381             for operator in SORTED_OPERATORS:
```

```
382                 if self.text.startswith(operator, self.index):
383                     for _ in operator:
384                         self.advance()
385                     self.add_token(
386                         kind="运算符",
387                         lexeme=operator,
388                         code=OPERATOR_CODES[operator],
389                         attribute="-",
390                         line=start_line,
391                         column=start_column,
392                     )
393                     return True
394 
395             current = self.current_char()
396             if current in DELIMITER_CODES:
397                 self.advance()
398                 self.add_token(
399                     kind="界符",
400                     lexeme=current,
401                     code=DELIMITER_CODES[current],
402                     attribute="-",
403                     line=start_line,
404                     column=start_column,
405                 )
406                 return True
407 
408             return False
409 
410         def intern_identifier(self, name: str) -> int:
411             """把标识符放进标识符表，并返回它的编号。"""
412             if name not in self.symbol_index:
413                 self.symbol_table.append(name)
414                 self.symbol_index[name] = len(self.symbol_table)
415             return self.symbol_index[name]
416 
417 
418 def build_report(
419     source_path: Path,
420     tokens: list[Token],
421     errors: list[LexicalError],
422     symbol_table: list[str],
423 ) -> str:
424     """把分析结果整理成适合直接展示或粘贴到报告里的文本。"""
425     lines: list[str] = []
426     lines.append("实验一：词法分析结果")
```

```
    lines.append(f"源文件：{source_path}")
    lines.append("")
    lines.append(
        f"共识别出 {len(tokens)} 个 token，{len(symbol_table)} 个标识符，{len(errors)} 个词法错误。"
    )
    lines.append("")
    lines.append("一、Token 二元式")

    if not tokens:
        lines.append("（没有识别到 token）")
    else:
        token_rows = [
            [
                str(index),
                f"{token.line}:{token.column}",
                token.lexeme,
                token.kind,
                f"({token.code}, {token.attribute})",
            ]
            for index, token in enumerate(tokens, start=1)
        ]
        lines.extend(
            render_table(
                headers=["序号", "位置", "单词", "种别", "二元式(code, attr)"],
                rows=token_rows,
                aligns=["right", "right", "left", "left", "left"],
            )
        )

    lines.append("")
    lines.append("二、标识符表")
    if not symbol_table:
        lines.append("（没有标识符）")
    else:
        symbol_rows = [[str(index), name] for index, name in enumerate(symbol_table, start=1)]
        lines.extend(
            render_table(
                headers=["编号", "标识符"],
                rows=symbol_rows,
                aligns=["right", "left"],
            )
```

```
        )

    lines.append("")
    lines.append("三、词法错误")
    if not errors:
        lines.append("未发现词法错误。")
    else:
        error_rows = [
            [str(index), str(error.line), str(error.column), error.message]
            for index, error in enumerate(errors, start=1)
        ]
        lines.extend(
            render_table(
                headers=["序号", "行", "列", "错误信息"],
                rows=error_rows,
                aligns=["right", "right", "right", "left"],
            )
        )

    return "\n".join(lines)


def render_table(
    headers: list[str],
    rows: list[list[str]],
    aligns: list[str] | None = None,
) -> list[str]:
    """把二维数据渲染成更整齐的纯文本表格。"""
    if aligns is None:
        aligns = ["left"] * len(headers)

    widths = [display_width(header) for header in headers]
    for row in rows:
        for index, cell in enumerate(row):
            widths[index] = max(widths[index], display_width(cell))

    border = "+" + "+".join("-" * (width + 2) for width in widths) + "+"
    header_line = "| " + " | ".join(
        pad_text(header, widths[index], aligns[index]) for index, header in enumerate(headers)
    ) + " |"

```

```
    lines = [border, header_line, border]
    for row in rows:
        row_line = "| " + " | ".join(
            pad_text(cell, widths[index], aligns[index]) for index, cell in enumerate(row)
        ) + " |"
        lines.append(row_line)
    lines.append(border)
    return lines


def pad_text(text: str, width: int, align: str = "left") -> str:
    """按显示宽度补空格，避免中文列名或内容错位。"""
    padding = max(width - display_width(text), 0)
    if align == "right":
        return " " * padding + text
    return text + " " * padding


def display_width(text: str) -> int:
    """计算字符串在终端中的显示宽度。"""
    width = 0
    for char in text:
        if unicodedata.combining(char):
            continue
        if unicodedata.east_asian_width(char) in {"W", "F"}:
            width += 2
        else:
            width += 1
    return width
```

4.2 正常测试输入文件 sample.c

```c
int main() {
    int total = 12 + 8;
    float rate = 3.5;

    // 这是一行注释
    if (total >= 20) {
        total += 1;
    }

    return total;
}
```

### 4.3 错误测试输入文件 error_sample.c

```c
int main() {
    int 123abc = 10;
    char c = 'x;
    total = 10 @ 2;
    return 0;
}
```