编译原理程序设计报告

程序设计 1：C 语言词法分析器

2021211110 杜抒泽

2023 年 10 月 21 日

目录

1. 实验要求

设计并实现 C 语言的词法分析程序，要求实现如下功能。

（1）可以识别出用 C 语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。

（2）可以识别并跳过源程序中的注释。

（3）可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。

（4）检查源程序中存在的词法错误，并报告错误所在的位置。

（5）对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。

2. 程序设计说明

2.1. 实现的基本功能

本次程序设计使用 C++ 完成 C 语言词法分析器的编写。未使用 lex 等词法分析工具。程序完成了：

- 对 ISO/IEC 9899:1999 标准（C99）中定义绝大多数词法的识别，并美观地输出每个词法符号的种类和内容。
- 对源程序中的语句行数、字符总数，以及所有词法标记数量的统计。
- 对错误的检查和报告、对多种错误的合理恢复。
- 对注释的跳过功能（也作为一类词法符号输出与统计）。

为了降低复杂性，我的程序未实现对预处理器以及一些生僻的、实际应用中几乎不会遇到的词法标记的处理。也即：

- 所有预处理器指令（`#` 或 `%:` 开头的行）整体被当做一类单独的词法标记处理。实际上，实现预处理器不是词法分析的职责。
- 假定程序中不存在位于行尾的反斜杠。

- 假定程序中不存在三标符（也即 `??/`、`??!`、`??<` 等记号）。
- 假定程序中不存在带 Unicode 转义序列的标识符（如 `int \u675c\u6292\u6cfd = 1;` 中的标识符）。

2.2. 开发环境

本次程序设计的项目名为 CLAP，采用的 C++ 标准为 C++20（GNU++20），在 ARM aarch 64 架构的 macOS Ventura 13.0 上编写、调试与测试，代码总行数为 1193 sloc，全部代码随附在作业中提交。若发现有任何问题请随时联系我。采用的编译工具链如下表：

工具	版本
`CMake`	3.26.4
`g++-13`	`(Homebrew GCC 13.2.0) 13.2.0`
`lldb`	16.0
`ld`	`ld64-820.1`

表 1: 工具链信息

2.3. 运行方式与 CLI 用户界面

CLAP 是一个命令行工具，带有命令行用户界面（CLI）。构建完成后，在命令行中键入 `./clap`，将收到如下回显。

```
$ ./clap
Usage: ./clap [OPTIONS] [FILE]

Arguments:
  [FILE]

Options:
  -h, --help       Print help
  -V, --version    Print version and exit
```

```
  -v, --verbosity  Set verbosity level [possible values: 0, 1,
2, 3, 4]
```

程序接收三个可选参数与一个位置参数。可选参数 `--help` 和 `--version` 指示是否打印帮助信息和版本信息，若指定则程序将打印信息并退出。可选参数 `--verbosity` 指示程序的详细程度，其值为 0、1、2、3、4 中的一个，分别对应输出错误信息、警告信息、提示信息、调试信息中的前若干者。若不指定 `--verbosity`，程序默认输出错误、警告和提示信息。位置参数 `FILE` 指示要分析的源文件路径，若不指定则程序打印帮助并退出。

例如，若要分析工作目录下的 `test.c`，可以键入 `./clap test.c`。本文后的测试部分将有大量运行示例，再次不再赘述。假设 `test.c` 的内容为 `int main(int argc, char **argv) {}`，输出的内容如下（不含颜色）：

```
Line:Column |            Token Type | Content
   1:1      |               Keyword | int
   1:5      |            Identifier | main
   1:9      |            Punctuator | (
   1:10     |               Keyword | int
   1:14     |            Identifier | argc
   1:18     |            Punctuator | ,
   1:20     |               Keyword | char
   1:25     |            Punctuator | *
   1:26     |            Punctuator | *
   1:27     |            Identifier | argv
   1:31     |            Punctuator | )
   1:33     |            Punctuator | {
   1:34     |            Punctuator | }

Statistic:
                  Line count: 1
```

```
            Character count: 34
          Total Token count: 13
              Keyword count: 3
           Identifier count: 3
     Integer constant count: 0
       Float constant count: 0
   Character constant count: 0
       String literal count: 0
           Punctuator count: 7
              Comment count: 0
Preprocessor command count: 0
                Error count: 0
```

2.4. 程序功能细节

2.4.1. Token 种类

CLAP 识别的 token 种类如下表：

种类	名称	备注
关键字	Keyword	提前硬编码，查红黑树
标识符	Identifier	
常量	Constant	包含整数常量、浮点数常量、字符常量
字符串字面量	String Literal	
符号	Punctuator	包含运算符、括号、双标替用记号等
预处理器命令	Preprocessor Command	
注释	Comment	包含行内和行间注释，抽象层面未区分
空白字符	Whitespace	不进行输出
词法错误	Lex Error	包含多种，见下节；以专门形式输出

表 2: Token 种类

2.4.2. 实现的词法错误种类

实现的以下的词法错误种类如下表。它们在词法分析中被抽象为类型 `LexError`，额外存储信息标记是何种词法错误。

种类	名称	错误恢复位置
非法字符	Illegal Character	下一个字符
非法浮点数常量	Illegal Float Constant	下一个字符
非法整数常量	Illegal Integer Constant	下一个字符
非法符号	Illegal Punctuator	下一个字符
块注释未闭合	Unclosed Block Comment	文件末尾（无法恢复）
字符常量未闭合	Unclosed Character Constant	下一行行首
字符串字面量未闭合	Unclosed String Literal	下一行行首

表 3: Token 种类

2.5. 源码编写思路：以 Lexer 为主体

程序是规范 C++ 项目。源码除 `main` 外还包含 `Cfg`、`Lexer`、`Util` 四个子模块，分别用于命令行与全局配置、词法分析器主体与常用工具，它们全部位于 `namespace clap` 内。此外还包含一个外部库 `argparse`，用于辅助命令行参数的解析。

`Cfg` 与 `Util` 模块与本次程序设计题目关系较弱，仅用于辅助实现功能，在此不过多介绍。以下主要介绍 `Lexer` 模块与 `main` 中对其的调用。

`Lexer` 模块包含 `FileReader`、`Lexer`、`Stat`、`Token` 四个部分。以下按照依赖关系自底向上依次对它们进行介绍。

2.5.1. 源码文件包装器 `FileReader`

`FileReader` 是词法分析器的一个核心组件，负责对输入的 C 语言源代码文件进行处理。这个类通过内部机制，不仅读取源文件内容，还解析了文件的结构，如行数、每行的起始位置等，为后续的词法分析阶段提供了方便。通过这种方式，当我们的词法分析器解析源代码生成 Token 时，能够快速准确地定位每个 Token 的具体位置（行号和列号），极大地方便了错误定位和报告，以及后续可能的语法分析阶段。

`FileReader` 的完整定义见附录相关部分。以下是部分成员说明：

1. `std::shared_ptr<std::string> content`
   - 源文件内容。由于源文件内容可能会被多个模块使用，因此使用 `std::shared_ptr` 进行生命周期管理。
2. `std::vector<usize> line_starts`
   - 源文件每行的起始位置。方便对每个 Token 的行号、列号进行定位。
3. 构造函数 `FileReader(const std::string &path)`
   - 给定一个文件路径，构造一个 `FileReader` 实例，读取文件内容并初始化行的起始位置信息。通过一次性读取文件避免了反复文件操作，提高了效率。同时，在构造阶段处理行信息，为后续操作提供了便利。
4. 禁用拷贝构造函数和拷贝赋值运算符，采用默认的移动构造函数和移动赋值运算符
   - 由于本类实例包含指向文件内容的指针和其他资源，禁用拷贝构造和拷贝赋值运算符是防止资源复制和潜在错误的重要措施。但支持移动语义，在对象被移动时确保资源的正确转移和原对象的状态更新，避免不必要的资源复制，提高效率。
5. `usize lookup_line(usize pos) const`
6. `std::pair<usize, usize> lookup_line_and_column(usize pos) const`
   - 这二者分别根据字符位置 `pos` 查找其所在的行号，或行号和列号。此功能依赖于存储的 `line_starts`，是其暴露给外部的接口。
7. `usize len() const`
8. `usize max_line_len() const`
9. `usize line_count() const`
   - 这三者分别为获取文件内容的总字符长度；计算所有行中最长的一行的字符数；返回文件的行数。同上，在设置缓冲区大小或进行格式化输出时有用。它们有助于设置缓冲区大小或进行格式化输出，确保资源的有效利用和输出的美观。

`FileReader` 利用了现代 C++ 的资源管理方式，不仅仅是一个简单的文件读取工具，更是词法分析过程中不可或缺的一部分，能够大大简化词法分析器的设计和实现。

2.5.2. 词法单元 Token

Token 是构成源代码的基本元素。本实验中 token 的种类在前文 2.4.1 节已经介绍过。这些组件是词法分析的基础，它们不仅帮助我们将源代码分解为标准化的元素，还存储了与这些元素相关的重要信息，如它们在源代码中的位置。

本项目在 token 的实现上采用了非常现代的实现方式：用类型而非简单的枚举表示 token 的种类，用 `std::variant` 实现 sum type（tagged union），用 `std::visit` 模拟模式匹配。这种方式的好处是，不仅能够保证类型安全，还能在编译期就检查出可能的错误。

Token 相关的完整定义见附录相关部分。

全局符号说明：

1. `enum class ConstantType` 和 `enum class LexErrorType`
   - 这些枚举类定义了常量的类型（如整数、浮点数、字符）以及可能的词法错误类型（如无效字符、未关闭的注释等），为 `Constant` 和 `LexError` 结构提供了一个明确的类型标签。
2. `Keyword`, `Identifier`, `Constant`, `StringLiteral`, `Punctuator`, `PreprocessorCommand`, `Comment`, `Whitespace` 和 `LexError`
   - `Constant` 和 `LexError` 分别包含 `ConstantType` 和 `LexErrorType` 类型的成员，用于标记常量的类型和词法错误的类型。其余类型是不同的空类型，用于标记不同的 token 种类。它们的存在是为了方便 `std::variant` 的使用，使得 `TokenBase` 类型成为一个 sum type，能够表示所有可能的 token 种类。

`Token` 部分成员说明：

1. `std::shared_ptr<std::string> src_`
   - 指向包含 Token 文本的字符串的智能指针。

2. `TokenBase inner() const noexcept`
   - 功能：返回存储在 `token_` 成员变量中的 `TokenBase` 对象。
3. `std::string get_name() const` 和 `std::string to_string() const`
   - `get_name()` 返回 Token 类型的名称，`to_string()` 依赖 `get_name()` 和 `src_`，返回 Token 的字符串表示形式，包括其位置和内容。这些方法提供了 Token 的可视化，这对于调试、错误报告和任何需要人类可读输出的场景都非常有用。

2.5.3. 词法分析器 `Lexer`

`Lexer` 类是本次词法分析程序设计的核心，负责将输入的源代码分解成一系列的 Token。该类通过一系列的成员函数实现了复杂的状态管理和字符处理逻辑，确保了词法分析的准确性和高效性。`Lexer` 相关的完整定义见附录相关部分。以下是部分成员说明：

1. `size pos_{}`
   - 这是一个内部使用的指示器，用于追踪 `Lexer` 当前的指针在源文件中的位置。它是控制读取和解析流程的关键变量。
2. `FileReader file`
   - 这是一个 `FileReader` 对象，它包含了要进行词法分析的源文件的内容和结构信息。`Lexer` 类通过这个成员变量读取字符和获取必要的文件信息。
3. `std::optional<Token> next_token()`
   - 从当前位置提取下一个 Token。该函数管理了 Token 识别的主流程，能够处理不同类型的词法元素，并在源文件结束时返回一个空的 `std::optional`，表示文件的终结。
4. `std::optional<char> peek(usize offset = 0) const` 和 `std::optional<char> consume()`

- `peek` 函数用于预览当前位置后的某个字符，而 `consume` 用于实际读取当前字符并将 `pos_` 前移。这两个函数是词法分析的基础，使得类能够根据当前的解析状态灵活地读取字符。

5. `bool reached_eof() const`
   - 判断是否到达文件末尾。提供了一个安全检查机制，防止对已结束的文件进行进一步操作，这对于控制解析流程和防止潜在的越界错误非常重要。
6. `consume_...` 系列函数
   - 这些函数处理特定的词法元素，如空白符、预处理命令、注释、字符串字面量、字符常量、标识符、数字等。每个函数都针对一种词法元素，实现了状态机的逻辑。这种模块化的设计使得 `Lexer` 类更容易扩展和维护。
7. `skip_...` 系列函数
   - 这些函数用于跳过特定类型的字符序列，如八进制数字、十进制数字、十六进制数字、指数部分等。在解析过程中，某些字符序列可能需要被忽略或仅用于确认 Token 的类型。这些函数提高了解析过程的灵活性和效率。

2.5.4. 统计信息 `Statistic`

`Statistic` 类用于统计信息，包含了一系列的成员变量，每个变量都对应于一个特定的统计指标。`Statistics` 相关的完整定义见附录相关部分。以下是部分成员说明：

1. `void update_by_file(const FileReader &file)`
   - 根据 `FileReader` 对象提供的源文件数据更新行数和字符数的统计信息。
2. `void update_by_token(TokenBase token)`
   - 根据传入的 Token 更新相关的统计计数。方法通过检查 Token 的类型来自动更新相应的计数器，展现了低耦合的设计。此外，它允许 `Statistic` 类能够实时更新，确保在任何时刻都能反映当前的分析状态。
3. `std::string to_string() const`

- 功能：生成当前统计数据的字符串表示，用于输出。这个函数提供了一种人类可读的输出格式，使得从调试和性能分析角度检查统计数据变得简单直观。它的设计考虑了易用性和可读性，是与用户交互不可或缺的一部分。

`Statistic` 类负责在词法分析阶段收集和管理统计数据。它对外部类和数据结构的依赖性低，显示了良好的封装性，这使得它在多种上下文中都能作为一个可靠的工具来使用。在未来，如果需要收集更多种类的统计信息，`Statistic` 类也可以很容易地进行扩展。

3. 测试、运行结果与分析说明

预先准备了若干个有强度的测试桩，其中包含应该被识别的各种 token，以及各种可能的错误。以下给出各个测试桩的运行结果的截图和分析。

1. `corr1.c`，内容为

```c
#include <stdio.h>

/* Multi-line comment
  that spans several lines */
int main() {
  int _validIdentifier123 = 456;
  double number = 3.14159;
  // Single line comment
  char c = 'a';
  if (c == 'a') {
    printf("Variable number has a value of: %f\n", number);
  }
  return 0;
}
```

```
# fa_555 @ fa555-kooBcaM in ~/Documents/repos/CLAP/cmake-build-release [20:17:26]
[$ ./clap ../testbench/corr1.c -v 0

Line:Column |           Token Type | Content
   1:1      | Preprocessor Command | #include <stdio.h>
   3:1      |              Comment | /* Multi-line comment
   that spans several lines */
   5:1      |              Keyword | int
   5:5      |           Identifier | main
   5:9      |           Punctuator | (
   5:10     |           Punctuator | )
   5:12     |           Punctuator | {
   6:3      |              Keyword | int
   6:7      |           Identifier | _validIdentifier123
   6:27     |           Punctuator | =
   6:29     |     Integer Constant | 456
   6:32     |           Punctuator | ;
   7:3      |              Keyword | double
   7:10     |           Identifier | number
   7:17     |           Punctuator | =
   7:19     |       Float Constant | 3.14159
   7:26     |           Punctuator | ;
   8:3      |              Comment | // Single line comment
   9:3      |              Keyword | char
   9:8      |           Identifier | c
   9:10     |           Punctuator | =
   9:12     |   Character Constant | 'a'
   9:15     |           Punctuator | ;
  10:3      |              Keyword | if
  10:6      |           Punctuator | (
  10:7      |           Identifier | c
  10:9      |           Punctuator | ==
  10:12     |   Character Constant | 'a'
  10:15     |           Punctuator | )
  10:17     |           Punctuator | {
  11:5      |           Identifier | printf
  11:11     |           Punctuator | (
  11:12     |       String Literal | "Variable number has a value of: %f\n"
  11:50     |           Punctuator | ,
  11:52     |           Identifier | number
  11:58     |           Punctuator | )
  11:59     |           Punctuator | ;
  12:3      |           Punctuator | }
  13:3      |              Keyword | return
  13:10     |     Integer Constant | 0
  13:11     |           Punctuator | ;
  14:1      |           Punctuator | }

Statistic:
                 Line count: 14
            Character count: 283
          Total Token count: 42
              Keyword count: 6
           Identifier count: 7
      Integer constant count: 2
        Float constant count: 1
    Character constant count: 2
        String literal count: 1
            Punctuator count: 20
               Comment count: 2
  Preprocessor command count: 1
                Error count: 0
```

这是一段比较简单的代码，所有词法成分都正确，且覆盖了所有种类的单词（当然除了错误）。

如左图，可以看到各个语法成分正确识别，统计也正常工作，没有错误。

2. `wrong1.c`，内容为

```c
#include <stdio.h>

int main() {
  int @wrong_identifier = 456;   // 错误：@ 字符
  double 3number = 3.14159;   // 错误：标识符以数字开头
```

```c
    if ('a' == 'b) { // 错误：字符常量未关闭
        printf("Unclosed string);  // 错误：字符串字面量未关闭
    }
    return 0;
}
/* unclosed comment  // 错误：未关闭的多行注释
```

```
# fa_555 @ fa555-kooBcaM in ~/Documents/repos/CLAP/cmake-build-release [20:27:26]
$ ./clap ../testbench/wrong1.c -v 0

Line:Column |           Token Type | Content
   1:1      | Preprocessor Command | #include <stdio.h>
   3:1      |              Keyword | int
   3:5      |           Identifier | main
   3:9      |           Punctuator | (
   3:10     |           Punctuator | )
   3:12     |           Punctuator | {
   4:3      |              Keyword | int
LexError: Invalid Punctuator | @ at 4:7
   4:8      |           Identifier | wrong_identifier
   4:25     |           Punctuator | =
   4:27     |     Integer Constant | 456
   4:30     |           Punctuator | ;
   4:32     |              Comment | // 错误：标识符中的 @ 字符
   5:3      |              Keyword | double
   5:10     |     Integer Constant | 3number
   5:18     |           Punctuator | =
   5:20     |       Float Constant | 3.14159
   5:27     |           Punctuator | ;
   5:32     |              Comment | // 错误：标识符以数字开头
   6:5      |              Keyword | if
   6:8      |           Punctuator | (
   6:9      |   Character Constant | 'a'
   6:13     |           Punctuator | ==
LexError: Unclosed Character Constant | 'b) {
 at 6:16
   7:7      |           Identifier | printf
   7:13     |           Punctuator | (
LexError: Unclosed String Literal | "Unclosed string);  // 错误：字符串常量未关闭
 at 7:14
   8:1      |           Punctuator | }
  10:1      |              Keyword | return
  10:8      |     Integer Constant | 0
  10:9      |           Punctuator | ;
  11:1      |           Punctuator | }
LexError: Unclosed Block Comment | /* unclosed comment  // 错误：未关闭的多行注释
 at 12:1

Statistic:
                 Line count: 12
            Character count: 333
          Total Token count: 33
              Keyword count: 5
           Identifier count: 3
     Integer constant count: 3
       Float constant count: 1
    Character constant count: 1
       String literal count: 0
            Punctuator count: 13
               Comment count: 2
  Preprocessor command count: 1
                Error count: 4
```

这段代码中覆盖了五类错误，分别是非法运算符号、非法标识符、字符常量未关闭、字符串字面量未关闭、多行注释未关闭。

如左图，可以看到，五类错误全部对正常诊断，且恢复的时机都与前文介绍相同。词法分析过程正确。

3. corr2.c，内容为

```c
int main() {
  int decimal = 255;
  int octal = 0377;
  int hex = 0×ff;
  int largeNumber = 1000000;
  double legal_float[] = {0.123456789, 1., .1, .2e3, 1.2e3,
1.2e-3, 0123.123};
}
```

```
# fa_555 @ fa555-kooBcaM in ~/Documents/repos/CLAP/cmake-build-release [20:45:04]
$ ./clap ../testbench/corr2.c -v 0

Line:Column |           Token Type | Content
   1:1      |              Keyword | int
   1:5      |           Identifier | main
   1:9      |           Punctuator | (
   1:10     |           Punctuator | )
   1:12     |           Punctuator | {
   2:5      |              Keyword | int
   2:9      |           Identifier | decimal
   2:17     |           Punctuator | =
   2:19     |     Integer Constant | 255
   2:22     |           Punctuator | ;
   3:5      |              Keyword | int
   3:9      |           Identifier | octal
   3:15     |           Punctuator | =
   3:17     |     Integer Constant | 0377
   3:21     |           Punctuator | ;
   4:5      |              Keyword | int
   4:9      |           Identifier | hex
   4:13     |           Punctuator | =
   4:15     |     Integer Constant | 0×ff
   4:19     |           Punctuator | ;
   5:5      |              Keyword | int
   5:9      |           Identifier | largeNumber
   5:21     |           Punctuator | =
   5:23     |     Integer Constant | 1000000
   5:30     |           Punctuator | ;
   6:5      |              Keyword | double
   6:12     |           Identifier | legal_float
   6:23     |           Punctuator | [
   6:24     |           Punctuator | ]
   6:26     |           Punctuator | =
   6:28     |           Punctuator | {
   6:29     |       Float Constant | 0.123456789
   6:40     |           Punctuator | ,
   6:42     |       Float Constant | 1.
   6:44     |           Punctuator | ,
   6:46     |       Float Constant | .1
   6:48     |           Punctuator | ,
   6:50     |       Float Constant | .2e3
   6:54     |           Punctuator | ,
   6:56     |       Float Constant | 1.2e3
   6:61     |           Punctuator | ,
   6:63     |       Float Constant | 1.2e-3
   6:69     |           Punctuator | ,
   6:71     |     Integer Constant | 0123.123
   6:79     |           Punctuator | }
   6:80     |           Punctuator | ;
   7:1      |           Punctuator | }

Statistic:
                Line count: 7
           Character count: 192
         Total Token count: 47
             Keyword count: 6
          Identifier count: 6
     Integer constant count: 5
       Float constant count: 6
   Character constant count: 0
       String literal count: 0
           Punctuator count: 24
              Comment count: 0
  Preprocessor command count: 0
               Error count: 0
```

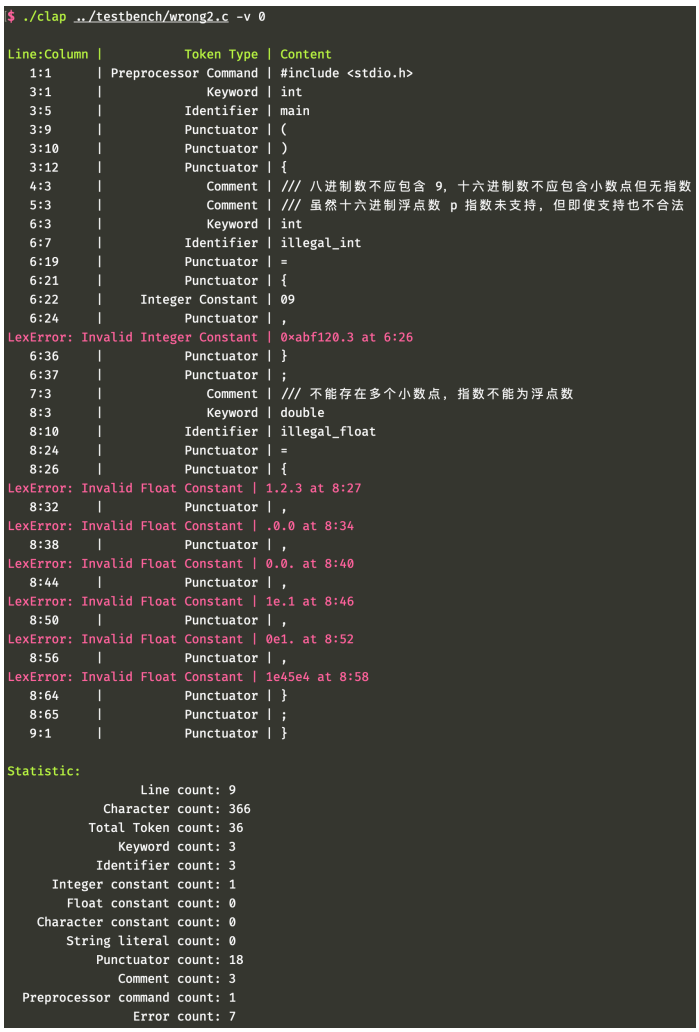
如左图，这段代码中测试了八进制、十进制、十六进制数字，以及浮点数常量的识别。

其中浮点数常量的例子比较刁钻。0.123456789 是很普通的浮点数。.1，1.，.2e3，1.2e3，1.2e-3 都是比较刁钻、难以词法分析，但容易理解的合法浮点数常量。0123.123 虽然以 0 开头，看似是八进制数字，但因其后有小数点，最终被认定为十进制浮点数。程序运行正确。

4. wrong2.c，内容为

```c
#include <stdio.h>

int main() {
  /// 八进制数不应包含 9，十六进制数不应包含小数点但无指数
  /// 虽然十六进制浮点数 p 指数未支持，但即使支持也不合法
  int illegal_int = {09, 0×abf120.3};
  /// 不能存在多个小数点，指数不能为浮点数
  double illegal_float = {1.2.3, .0.0, 0.0., 1e.1, 0e1.,
1e45e4};
}
```



```
$ ./clap ../testbench/wrong2.c -v 0

Line:Column |           Token Type | Content
   1:1      | Preprocessor Command | #include <stdio.h>
   3:1      |              Keyword | int
   3:5      |           Identifier | main
   3:9      |           Punctuator | (
   3:10     |           Punctuator | )
   3:12     |           Punctuator | {
   4:3      |              Comment | /// 八进制数不应包含 9，十六进制数不应包含小数点但无指数
   5:3      |              Comment | /// 虽然十六进制浮点数 p 指数未支持，但即使支持也不合法
   6:3      |              Keyword | int
   6:7      |           Identifier | illegal_int
   6:19     |           Punctuator | =
   6:21     |           Punctuator | {
   6:22     |      Integer Constant | 09
   6:24     |           Punctuator | ,
LexError: Invalid Integer Constant | 0×abf120.3 at 6:26
   6:36     |           Punctuator | }
   6:37     |           Punctuator | ;
   7:3      |              Comment | /// 不能存在多个小数点，指数不能为浮点数
   8:3      |              Keyword | double
   8:10     |           Identifier | illegal_float
   8:24     |           Punctuator | =
   8:26     |           Punctuator | {
LexError: Invalid Float Constant | 1.2.3 at 8:27
   8:32     |           Punctuator | ,
LexError: Invalid Float Constant | .0.0 at 8:34
   8:38     |           Punctuator | ,
LexError: Invalid Float Constant | 0.0. at 8:40
   8:44     |           Punctuator | ,
LexError: Invalid Float Constant | 1e.1 at 8:46
   8:50     |           Punctuator | ,
LexError: Invalid Float Constant | 0e1. at 8:52
   8:56     |           Punctuator | ,
LexError: Invalid Float Constant | 1e45e4 at 8:58
   8:64     |           Punctuator | }
   8:65     |           Punctuator | ;
   9:1      |           Punctuator | }

Statistic:
                Line count: 9
           Character count: 366
         Total Token count: 36
             Keyword count: 3
          Identifier count: 3
     Integer constant count: 1
       Float constant count: 0
   Character constant count: 0
      String literal count: 0
          Punctuator count: 18
             Comment count: 3
  Preprocessor command count: 1
               Error count: 7
```

如左图，这段代码中测试了不合法的八进制、十六进制数字，以及浮点数常量的识别。

整数常量 09 不合法，因为八进制整数不能包含数码 9。十六进制数 0×abf120.3 不合法，因为十六进制浮点数必须带有 p 引导的指数序列（虽然没有实现）。

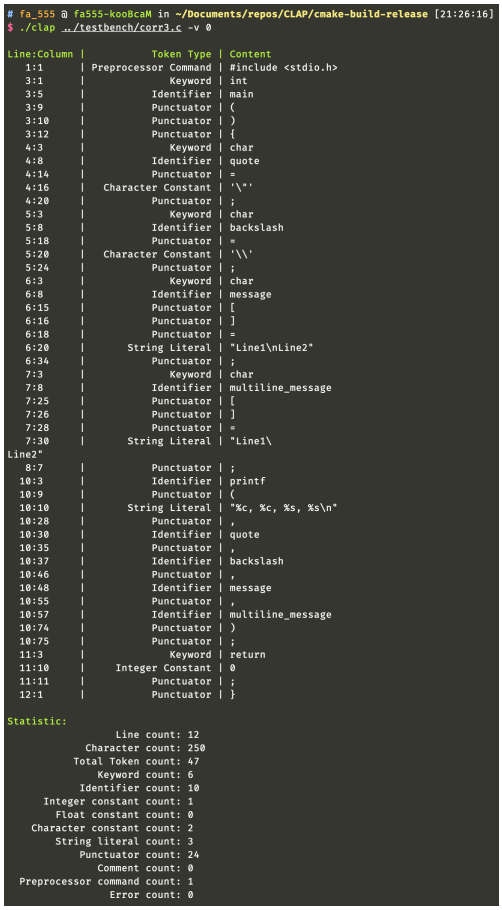
浮点数常量 1.2.3、.0.0、0.0. 是不合法的，因为含有多个 0。1e.1 和 0e1. 是不合法的，因为指数中不能含有小数点。1e45e4 是不合法的，因为不能嵌套指数。程序运行正确。

5. corr3.c，内容为

```c
#include <stdio.h>

int main() {
  char quote = '\"';
  char backslash = '\\';
  char message[] = "Line1\nLine2";
  char multiline_message[] = "Line1\
Line2";

  printf("%c, %c, %s, %s\n", quote, backslash, message,
multiline_message);
  return 0;
}
```



```
# fa_555 @ fa555-koo8caM in ~/Documents/repos/CLAP/cmake-build-release [21:26:16]
$ ./clap ../testbench/corr3.c -v 0

Line:Column |          Token Type | Content
   1:1      | Preprocessor Command | #include <stdio.h>
   3:1      |              Keyword | int
   3:5      |           Identifier | main
   3:9      |           Punctuator | (
   3:10     |           Punctuator | )
   3:12     |           Punctuator | {
   4:3      |              Keyword | char
   4:8      |           Identifier | quote
   4:14     |           Punctuator | =
   4:16     |   Character Constant | '\"'
   4:20     |           Punctuator | ;
   5:3      |              Keyword | char
   5:8      |           Identifier | backslash
   5:18     |           Punctuator | =
   5:20     |   Character Constant | '\\'
   5:24     |           Punctuator | ;
   6:3      |              Keyword | char
   6:8      |           Identifier | message
   6:15     |           Punctuator | [
   6:16     |           Punctuator | ]
   6:18     |           Punctuator | =
   6:20     |      String Literal | "Line1\nLine2"
   6:34     |           Punctuator | ;
   7:3      |              Keyword | char
   7:8      |           Identifier | multiline_message
   7:25     |           Punctuator | [
   7:26     |           Punctuator | ]
   7:28     |           Punctuator | =
   7:30     |      String Literal | "Line1\
Line2"
   8:7      |           Punctuator | ;
  10:3      |           Identifier | printf
  10:9      |           Punctuator | (
  10:10     |      String Literal | "%c, %c, %s, %s\n"
  10:28     |           Punctuator | ,
  10:30     |           Identifier | quote
  10:35     |           Punctuator | ,
  10:37     |           Identifier | backslash
  10:46     |           Punctuator | ,
  10:48     |           Identifier | message
  10:55     |           Punctuator | ,
  10:57     |           Identifier | multiline_message
  10:74     |           Punctuator | )
  10:75     |           Punctuator | ;
  11:3      |              Keyword | return
  11:10     |    Integer Constant | 0
  11:11     |           Punctuator | ;
  12:1      |           Punctuator | }

Statistic:
                Line count: 12
           Character count: 250
         Total Token count: 47
             Keyword count: 6
          Identifier count: 10
     Integer constant count: 1
       Float constant count: 0
   Character constant count: 2
       String literal count: 3
           Punctuator count: 24
              Comment count: 0
 Preprocessor command count: 1
               Error count: 0
```

如左图，这段代码中测试了字符常量和字符串字面量中的多种转义序列，包括反斜杠续行。

能够正确处理这段代码。程序运行正确。

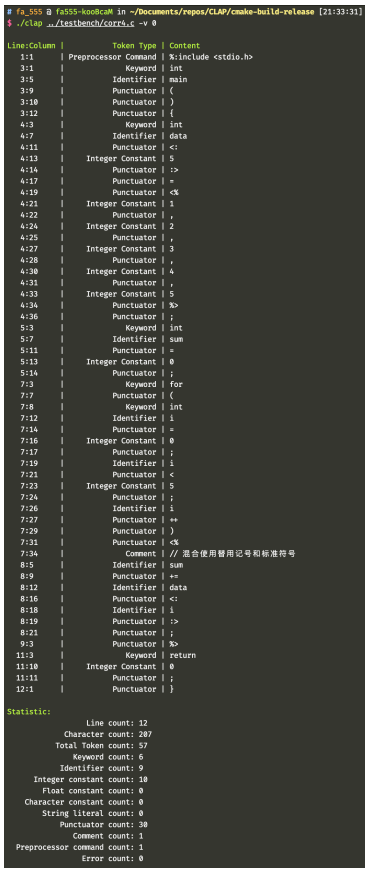
6. corr4.c，内容为

```c
%:include <stdio.h>

int main() {
  int data<:5:> = <%1, 2, 3, 4, 5%>;
  int sum = 0;

  for (int i = 0; i < 5; i++) <% // 混合使用替用记号和标准符号
    sum += data<:i:>;
  %>

  printf("Total sum: %d\n", sum);
  return 0;
}
```



```
# fa_555 @ fa555-kaoBca# in ~/Documents/repos/CLAP/cmake-build-release [21:33:31]
$ ./clap ../testbench/corr4.c -v 0

Line:Column |          Token Type | Content
   1:1      | Preprocessor Command | %:include <stdio.h>
   3:1      |              Keyword | int
   3:5      |           Identifier | main
   3:9      |           Punctuator | (
   3:10     |           Punctuator | )
   3:12     |           Punctuator | {
   4:3      |              Keyword | int
   4:7      |           Identifier | data
   4:11     |           Punctuator | <:
   4:13     |    Integer Constant | 5
   4:14     |           Punctuator | :>
   4:17     |           Punctuator | =
   4:19     |           Punctuator | <%
   4:21     |    Integer Constant | 1
   4:22     |           Punctuator | ,
   4:24     |    Integer Constant | 2
   4:25     |           Punctuator | ,
   4:27     |    Integer Constant | 3
   4:28     |           Punctuator | ,
   4:30     |    Integer Constant | 4
   4:31     |           Punctuator | ,
   4:33     |    Integer Constant | 5
   4:34     |           Punctuator | %>
   4:36     |           Punctuator | ;
   5:3      |              Keyword | int
   5:7      |           Identifier | sum
   5:11     |           Punctuator | =
   5:13     |    Integer Constant | 0
   5:14     |           Punctuator | ;
   7:3      |              Keyword | for
   7:7      |           Punctuator | (
   7:8      |              Keyword | int
   7:12     |           Identifier | i
   7:14     |           Punctuator | =
   7:16     |    Integer Constant | 0
   7:17     |           Punctuator | ;
   7:19     |           Identifier | i
   7:21     |           Punctuator | <
   7:23     |    Integer Constant | 5
   7:24     |           Punctuator | ;
   7:26     |           Identifier | i
   7:27     |           Punctuator | ++
   7:29     |           Punctuator | )
   7:31     |           Punctuator | <%
            |              Comment | // 混合使用替用记号和标准符号
   8:5      |           Identifier | sum
   8:9      |           Punctuator | +=
   8:12     |           Identifier | data
   8:16     |           Punctuator | <:
   8:18     |           Identifier | i
   8:19     |           Punctuator | :>
   8:21     |           Punctuator | ;
   9:3      |           Punctuator | %>
  11:3      |              Keyword | return
  11:10     |    Integer Constant | 0
  11:11     |           Punctuator | ;
  12:1      |           Punctuator | }

Statistic:
                Line count: 12
           Character count: 207
         Total Token count: 57
             Keyword count: 6
          Identifier count: 9
     Integer constant count: 10
       Float constant count: 0
  Character constant count: 0
      String literal count: 0
         Punctuator count: 30
           Comment count: 1
 Preprocessor command count: 1
               Error count: 0
```

如左图，这段代码中测试了双标替用记号（如 %: 代替 #、<: 代替 [）和混合使用双标替用记号与普通记号。

能够正确处理这段代码。程序运行正确。

7. huge_test.c。与前面不同，这是一个超级大的测试。首先在 huge_test.c 中写入如下代码：

```
#include <assert.h>
#include <complex.h>
#include <ctype.h>
#include <errno.h>
#include <fenv.h>
#include <float.h>
#include <inttypes.h>
#include <iso646.h>
#include <limits.h>
#include <locale.h>
#include <math.h>
#include <setjmp.h>
#include <signal.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tgmath.h>
#include <time.h>
#include <wchar.h>
#include <wctype.h>
```

这是 C 语言直到 C99 标准的所有标头。然后，我们在 testbench/ 目录下输入如下命令：

```
$ gcc -E huge_test.c -o huge_test_out.c
```

再回到 CLAP 所在的目录下，输入执行命令并获取最后 40 行输出如图：

```
# fa_555 @ fa555-kooBcaM in ~/Documents/repos/CLAP/cmake-build-release [21:45:54]
$ ./clap ../testbench/huge_test_out.c -v 0 | tail -n 40
 3305:8      |            Identifier | nextwctype
 3305:18     |            Punctuator | (
 3305:19     |            Identifier | wint_t
 3305:25     |            Punctuator | ,
 3305:27     |            Identifier | wctype_t
 3305:35     |            Punctuator | )
 3305:36     |            Punctuator | ;
 3307:1      |            Identifier | wint_t
 3307:8      |            Identifier | towctrans
 3307:17     |            Punctuator | (
 3307:18     |            Identifier | wint_t
 3307:24     |            Punctuator | ,
 3307:26     |            Identifier | wctrans_t
 3307:35     |            Punctuator | )
 3307:36     |            Punctuator | ;
 3308:1      |            Identifier | wctrans_t
 3309:2      |            Identifier | wctrans
 3309:9      |            Punctuator | (
 3309:10     |               Keyword | const
 3309:16     |               Keyword | char
 3309:21     |            Punctuator | *
 3309:22     |            Punctuator | )
 3309:23     |            Punctuator | ;
 3311:1      | Preprocessor Command | # 25 "huge_test.c" 2

Statistic:
                 Line count: 3311
            Character count: 109613
          Total Token count: 14555
              Keyword count: 3572
           Identifier count: 3617
     Integer constant count: 220
       Float constant count: 3
   Character constant count: 1
       String literal count: 74
           Punctuator count: 6615
              Comment count: 0
  Preprocessor command count: 453
                Error count: 0
```

gcc 编译器的 -E 选项会将源代码只经过预处理器处理，输出预处理器的结果。

可见，程序成功分析了全体 C99 标头的内容，总字符数和总 token 数分别达到了较大的 189613 和 14555，且词法错误计数（输出的最后一行）为 0。

由此，可以认为本次实验的词法分析器能够正确处理 C99 标准的所有常用词法，程序设计的正确性较好，能够满足作业要求。

4. 总结

本实验到此完全结束。在本次实验中，我成功实现了一个名为 CLAP 的 C 语言词法分析程序。为了降低复杂性，CLAP 并没有实现对预处理器以及一些较为生僻的词法标记的处理，但这并不影响 CLAP 在处理常规 C99 程序时的准确性和高效性。

CLAP 能够准确地识别和输出 C 语言源程序中的各类单词符号，处理代码中的注释，还具备了统计源程序中的语句行数、各类词法记号的数量以及字符总数的能力。此外，CLAP 还能够检测源程序中存在的词法错误，并准确无误地报告错误所在的具体位置。在遇到错误时，它能够进行适当的错误恢复，确保词法分析过程持续进行。

最终，通过在少于 0.01 秒内完成对包含 C99 标准全部头文件的庞大测试文件 `huge_test.c` 的测试，CLAP 证明了自己在处理大型源代码文件时依然能够保持高性能和准确性，成功地分析了文件的内容，准确统计了字符数和 token 数，并且未报告任何词法错误。这一结果充分展示了 CLAP 在词法分析方面的能力，标志着本次实验圆满完成，成功达到了实验要求和预期目标。通过这次实验，我加深了对 C 语言词法结构的理解，而且积累了在实际场景下设计和实现词法分析器的宝贵经验。

5. 附录：部分代码

5.1. 源码文件包装器 `FileReader`

```
class FileReader {
  std::string path_{};

public:
  std::shared_ptr<std::string> content{};
  std::vector<usize> line_starts{};

  explicit FileReader(const std::string &path);

  FileReader(const FileReader &rhs) = delete;

  FileReader(FileReader &&rhs) noexcept = default;
```

```
  FileReader &operator=(const FileReader &rhs) = delete;

  FileReader &operator=(FileReader &&rhs) noexcept = default;

  ~FileReader() = default;

  usize lookup_line(usize pos) const;

   std::pair<usize, usize> lookup_line_and_column(usize pos)
const;

  std::string get_line(usize line_number) const;

  usize len() const;

  usize max_line_len() const;

  usize line_count() const;
};
```

5.2. 词法单元 Token

```
enum class ConstantType : u8 {
  Integer,
  Float,
  Character,
};

enum class LexErrorType : u8 {
  IllegalCharacter,
  IllegalFloatConstant,
  IllegalIntegerConstant,
```

```
  IllegalPunctuator,
  UnclosedBlockComment,
  UnclosedCharacterConstant,
  UnclosedStringLiteral,
};

struct Keyword {
  const inline static std::set<std::string_view> KEYWORDS = {
    /* ... */
  };
};

struct Identifier {};

struct Constant {
  ConstantType type_{};

  explicit Constant(ConstantType type);
};

struct StringLiteral {};

struct Punctuator {};

struct PreprocessorCommand {};

struct Comment {};

struct Whitespace {};

struct LexError {
  LexErrorType type_{};
```

```
  explicit LexError(LexErrorType type);
};

using TokenBase = std::variant<
    Keyword, Identifier, Constant, StringLiteral, Punctuator,
    PreprocessorCommand, Comment, Whitespace, LexError>;

class Token {
  std::shared_ptr<std::string> src_{};
  usize start_{}, end_{};
  TokenBase token_{};

public:
  Token(
      TokenBase token, usize start, usize end,
      const std::shared_ptr<std::string> &src
  );

  TokenBase inner() const noexcept;

  usize start() const noexcept;

  usize end() const noexcept;

  usize len() const noexcept;

  std::string get_name() const;

  std::string to_string() const;
};
```

5.3. 词法分析器 Lexer

```
class Lexer {
  usize pos_{};

public:
  FileReader file;

  explicit Lexer(const std::string &file_path);

  Lexer(const FileReader &file) = delete; // no need to impl

  explicit Lexer(FileReader &&file) noexcept;

  Lexer(const Lexer &rhs) = delete;

  std::optional<Token> next_token();

  std::optional<char> peek(usize offset = 0) const;

  std::optional<char> consume();

  bool reached_eof() const;

  TokenBase consume_whitespace();

  TokenBase consume_preprocessor_command();

  TokenBase consume_inline_comment();

  TokenBase consume_block_comment();

  TokenBase consume_string_literal();
```

```
  TokenBase consume_character_constant();

  TokenBase consume_identifier_or_keyword();

  TokenBase consume_number(char first_char);

  TokenBase consume_punctuator(char first_char);

  TokenBase consume_octal_constant();

  TokenBase consume_decimal_constant();

  TokenBase consume_hexadecimal_constant();

   TokenBase consume_float_constant(bool has_decimal_point =
false);

  void skip_identifier_or_keyword();

  bool skip_octal_digits();

  bool skip_decimal_digits();

  bool skip_hexadecimal_digits();

  bool skip_integer_suffix(bool allow_floating_point);

  bool skip_float_suffix();

  bool skip_exponent_part();
```

```
  void skip_all_number_char();
};
```

5.4. 统计信息 Statistic

```
class Statistic {
  usize line_cnt_{};
  usize char_cnt_{};
  usize token_cnt_{};
  usize keyword_cnt_{};
  usize identifier_cnt_{};
  usize integer_cnt_{};
  usize float_cnt_{};
  usize character_cnt_{};
  usize string_cnt_{};
  usize punctuator_cnt_{};
  usize comment_cnt_{};
  usize preprocessor_command_cnt_{};
  usize error_cnt_{};

public:
  Statistic() = default;

  void update_by_file(const FileReader &file);

  void update_by_token(TokenBase token);

  std::string to_string() const;
};
```

点此回到附录开头。