编译原理作业 (1)

姓名： 魏恒峰　　**学号：** hfwei@nju.edu.cn

评分： _____　　**评阅：** _____

2024 年 3 月 31 日

请独立完成作业，不得抄袭。
若得到他人帮助，请致谢。
若参考了其它资料，请给出引用。
鼓励讨论，但需独立书写解题过程。
允许并鼓励使用 ChatGPT 等工具，但需明确说明使用方式。

1　作业（必做部分）

题目 1 (C 语言中的 ANTLR 4 词法规约)
阅读 C 语言 .g4 文件，完成以下任务 (均可用图示辅助解释)。

(1) 定位并解释其中的“字符串” (`StringLiteral`) 词法规则。

(2) 定义并解释其中的“常量” (`IntegerConstant`、`FloatingConstant`、`CharacterConstant`) 词法规则。

(3) 使用 `lexer grammar` [1] 在 ANTLR 4 工具中测试 `C.g4` 中的词法单元
- 考虑如何设计测试用例覆盖尽可能多的情况?
- 检查你对词法规则的理解是否与 ANTLR 4 的输出一致。

(4) 其它: 请自行挖掘有趣的内容。

[1] 关于 `lexer grammar` 的用法:
- 见《ANTLR 4 权威指南》第 4.1 节
- 注意: Gradle ANTLR 插件在需要将 lexer grammar 导入到更大的 grammar 文件中时有一个尚未修复的“幺蛾子” (bug), 参见 build.gradle 文件。

解答：

(1) “字符串”(`StringLiteral`) 词法规则由由四个部分组成，分别是可选的编码前缀 `EncodingPrefix`、双引号、`SCharSequence` 和另一个双引号。整体上，它表示了一个字符串常量的结构。接下来，我们将依次解释 `EncodingPrefix` 和 `SCharSequence` 这两个部分。

- `EncodingPrefix` 规则定义了字符串的编码前缀。它可以是 ‘u8’、‘u’、‘U’ 或 ‘L’ 之一。这些前缀用于指示字符串的字符编码方式，比如 UTF-8 (u8)、UTF-16 (u 和 U) 或宽字符 (L)。

- SCharSequence 规则定义了定义了 SChar 的序列, 即字符串中的字符序列。它由一个或多个 SChar 组成。而 SChar 定义了字符串中的单个字符。它可以是除了双引号、反斜杠和换行符之外的任何字符。如果字符是反斜杠, 则可能是转义序列 EscapeSequence。此外, 该规则还允许处理转义序列 \n（表示换行）和 \r\n（表示回车换行）。接下来, 我们将解释 EscapeSequence。EscapeSequence 定义了转义序列的形式。转义序列允许在字符串中插入一些特殊字符，如换行符、制表符等。它可以是简单转义序列 SimpleEscapeSequence、八进制转义序列 OctalEscapeSequence、十六进制转义序列 HexadecimalEscapeSequence 或通用字符名称 UniversalCharacterName 之一：
  - SimpleEscapeSequence 定义了简单转义序列的形式，比如 \n（换行符）、\t（制表符）等。
  - OctalEscapeSequence 定义了八进制转义序列的形式，比如 \0、\123 等。
  - HexadecimalEscapeSequence 定义了十六进制转义序列的形式，比如 \x1F 等。
- EscapeSequence 在 '\\\n' 前面可能是 "\\\n' （也包括下一条规则'\\\r\n'）是为了处理 C 语言中的多行字符串字面量与多行宏中的换行的。

(2)
- 整数常量 IntegerConstant 由四个部分组成,分别是十进制常量(DecimalConstant)、八进制常量（OctalConstant）、十六进制常量（HexadecimalConstant）和二进制常量（BinaryConstant）。除了二进制常量外，每个部分都可以跟随一个整数后缀（IntegerSuffix）。
  - BinaryConstant：以 '0b' 或 0B 为前缀，后面跟着一个由 0 和 1 组成的二进制数字序列。
  - DecimalConstant：以 1 到 9 之间的非零数字开头，后面可以跟着任意数量的数字（0-9）。
  - OctalConstant：以数字 0 开头，后面跟着零个或多个八进制数字（0-7）。
  - HexadecimalConstant：以 '0x' 或 '0X' 为前缀，后面跟着至少一个十六进制数字（0-9，a-f，A-F）。
  - IntegerSuffix：整数后缀由可选的无符号后缀（UnsignedSuffix）和可选的长整数后缀（LongSuffix 或 LongLongSuffix）组成。无符号后缀可以是 'u' 或 'U'。长整数后缀可以是 'l'、'L'、'll' 或 'LL'，分别表示长整数和长长整数。。
- 浮点数常量 FloatingConstant 可以是十进制浮点常量(DecimalFloatingConstant）或十六进制浮点常量（HexadecimalFloatingConstant）之一。
  - 十进制浮点常量 DecimalFloatingConstant 由两个部分组成：小数部分（FractionalConstant）、指数部分（ExponentPart）和浮点数后缀(FloatingSuffix)以及数字序列 (DigitSequence)、指数部分(ExponentPart）和浮点数后缀（FloatingSuffix）。小数部分可以是小数点前的数字序列，后跟小数点，再跟小数点后的数字序列；或者是仅有小数点后的数字序列。指数部分以 'e' 或 'E' 开头，可以带正负号，后面跟着一个数字序列。
  - 十六进制浮点常量 HexadecimalFloatingConstant 由四个部分组成：十六进制前缀(HexadecimalPrefix)、十六进制小数部分(HexadecimalFractionalConstant 或 HexadecimalDigitSequence)、二进制指数部分 (BinaryExponentPart）和浮点数后缀（FloatingSuffix）。十六进制前缀以 '0x' 或 '0X' 开头。十六进制小数部分可以是可选的十六进制数字序列，后跟小数点，再跟着另一个十六进制数字序列；或者仅有十六进制数字序列。二进制指数部分以 'p' 或 'P' 开头，可以带正负号，后面跟着一个数字序列。
  - FloatingSuffix 可以是 'f'、'F'、'l' 或 'L'，表示浮点数的类型（单精度浮点数、双精度浮点数、长浮点数等）。

- `CharacterConstant` 字符常量由一个单引号、`CCharSequence` 和另一个单引号组成。其中 `CCharSequence` 是一个或多个 `CChar` 组成的序列。字符常量可以具有不同的前缀：'L'、'u' 或 'U'，用于表示不同的字符类型。`CChar` 定义了字符常量中的单个字符。它可以是除了单引号、反斜杠和换行符之外的任何字符。如果字符是反斜杠，则可能是转义序列（`EscapeSequence`）。`EscapeSequence` 定义了转义序列的形式。转义序列允许在字符常量中插入一些特殊字符，如换行符、制表符等。它也可以是简单转义序列（`SimpleEscapeSequence`）、八进制转义序列（`OctalEscapeSequence`）、十六进制转义序列（`HexadecimalEscapeSequence`）或通用字符名称（`UniversalCharacterName`）之一。这里只介绍通用字符名称（`UniversalCharacterName`）。
  - `UniversalCharacterName` 定义了通用字符名称的形式，以 \u 或 \U 开头，后跟四个或八个十六进制数字。`HexQuad` 定义了四个连续的十六进制数字，用于表示通用字符名称中的码点。

(3) 为了尽可能覆盖 C 语言的词法单元，可以考虑设计以下几种情况：

- 测试基本词法单元：添加各种基本数据类型的常量，如整数常量、浮点数常量、字符常量和各种进制的常量；添加标识符和关键字。
- 测试边界情况：可以设计整数和浮点数常量的最小值、最大值以及边界情况；设计包含特殊字符的标识符。
- 测试转义序列和通用字符：识别输出各种转义字符；识别通用字符。
- 测试不合法字符识别：添加错误的标识符或者未定义的字符；添加错误的转义序列和通用字符。

题目 2（词法分析器代码分析）
查看 Clang Lexer 文档, 阅读 Clang 词法分析器源码 `Lexer.cpp`, 完成以下任务 (均可用图示辅助解释)。

- 整理函数 `Lexer::Lex()` 的主要逻辑。
- 定义到处理 `StringLiteral` 词法单元的代码, 并分析代码的主要逻辑。
- 定义到处理 `IntegerConstant` 与 `FloatingConstant` 词法单元的代码, 并分析代码的主要逻辑。
- 其它: 请自行挖掘有趣的内容。

解答：

- `Lexer::Lex()` 函数主要逻辑如下：
  - assert(!isDependencyDirectivesLexer());：这是一个断言语句，用于确保词法分析器不处于依赖指令的词法分析模式下。如果该条件为真，将导致断言失败，表示出现了意料之外的情况。
  - Result.startToken();：开始一个新的标记。
  - 设置各种标记的标志，这些标志指示标记的一些属性，如是否在行的开头（Token::StartOfLine）、是否具有前导空格（Token::LeadingSpace）、是否具有前导空的宏（Token::LeadingEmptyMacro）等。
  - 'bool atPhysicalStartOfLine = IsAtPhysicalStartOfLine;'将变量 IsAtPhysicalStartOfLine 的当前值复制给 atPhysicalStartOfLine，用于记录是否在物理行的起始位置。这是为了在调用 LexTokenInternal 函数之前保留 IsAtPhysicalStartOfLine 的值，以便在需要时了解当前位置是否在物理行的起始位置。

- ‘IsAtPhysicalStartOfLine = false;’ 将 IsAtPhysicalStartOfLine 的值设置为 false，表示当前不在物理行的起始位置。这是因为在开始处理新的标记之前，词法分析器不再处于物理行的起始位置。
- ‘bool isRawLex = isLexingRawMode();’ 调用 isLexingRawMode() 函数，用于检查词法分析器是否处于原始词法分析模式（即以原始模式进行词法分析，不执行任何预处理）。返回值存储在 isRawLex 变量中。
- ‘(void) isRawLex;’ 这是一个类型转换语句，将 isRawLex 变量的值强制转换为 void 类型，从而避免编译器产生“未使用变量”的警告。在这里，它的目的是告诉编译器，我们有意不使用 isRawLex 变量，但是调用了 isLexingRawMode() 函数来检查词法分析器是否处于原始词法分析模式。
- 接下来调用 LexTokenInternal 函数，这是词法分析器内部用于实际识别和解析标记的函数。它将填充 Result 中的标记对象，并返回一个布尔值，指示是否成功识别了标记。
- returnedToken 变量接收 LexTokenInternal 函数的返回值，用于检查是否成功识别了标记。
- 最后，如果词法分析处于原始词法分析模式（isRawLex 为真），则断言确保词法分析成功。否则，返回 returnedToken，指示词法分析是否成功。

- StringLiteral 词法单元处理函数主要有两个 LexStringLiteral 和 LexRawStringLiteral。
  - ‘LexStringLiteral’ 函数处理处理的是普通的字符串常量，逻辑如下：
    - 扫描字符串常量：从指定的字符指针 CurPtr 开始扫描源代码字符流，直到遇到字符串常量的结束引号（”）。
    - 处理转义字符：在扫描过程中，处理转义字符，包括反斜杠后的转义序列，如 \n、\t 等。转义字符会被逐个处理，以确保正确解析字符串内容。
    - 处理特殊情况：在处理过程中，如果遇到换行符（\n、\r）、文件结束符（EOF），或者文件未正确结束，则发出相应的诊断信息，并生成一个未知的标记。
    - 检查是否存在空字符：如果字符串中存在空字符（’\0’），则发出警告。
    - 识别可选的用户定义后缀（UD-suffix）：如果在 C++11 模式下，识别并处理可选的用户定义后缀。
    - 构建标记：构建识别到的字符串常量为标记，并设置其类型为 tok::string_literal。设置标记的文本范围，并更新 BufferPtr 的位置。
  - LexRawStringLiteral 函数处理逻辑如下：
    - 扫描原始字符串常量：从指定的字符指针 CurPtr 开始扫描源代码字符流，直到找到与原始字符串的结束定界符匹配的位置。
    - 检查原始字符串定界符：检查原始字符串的定界符是否合法，即是否以 ‘R’ 或 ‘LR’ 或 ‘u8R’ 或 ‘uR’ 或 ‘UR’ 开头，后接（。如果不合法，则发出相应的诊断信息，并尝试寻找下一个（”）。
    - 识别可选的用户定义后缀（UD-suffix）：如果在 C++11 模式下，识别并处理可选的用户定义后缀。
    - 构建标记：构建识别到的原始字符串常量为标记，并设置其类型为 tok::string_literal。设置标记的文本范围，并更新 BufferPtr 的位置。

- IntegerConstant 和 FloatConstant 词法单元处理的函数是 LexNumericConstant。逻辑如下：
  - 扫描数字常量：从指定的字符指针 CurPtr 开始扫描源代码字符流，识别整数或浮点数常量的字符序列。
  - 识别数字常量：通过 ‘isPreprocessingNumberBody’ 函数判断当前字符是否属于整数或浮点数的字符序列，循环扫描直到不再是数字常量的一部分。在扫描过程中，通过 ‘ConsumeChar’ 函数逐步移动字符指针，并存储识别到的字符。

- 处理特殊情况：对于特殊情况，如指数表示的浮点数常量（如 1e+12）、十六进制浮点数常量以及数字分隔符（C++14/C23 新增），进行额外处理；对于指数表示的浮点数常量，检查是否后续还有符号，并在 Microsoft 模式下特殊处理；对于十六进制浮点数常量，判断是否处于不支持的模式下，例如不支持的 C99 模式，或者存在下划线的 C++17 模式；对于数字分隔符，检查其是否符合 C++14 或 C23 的要求，如果符合则发出警告。
- 处理转移序列和 UTF-8 字符：对于可能存在于数字常量中的转义序列和 UTF-8 字符进行处理，包括 Unicode 字符名称（UCN）和 UTF-8 字符。
- 构建标记：将识别到的数字常量构建为标记(Token)，设置其类型为 'tok::numeric_constant'。设置标记的文本范围，并更新 CurPtr 和 BufferPtr 的位置。
- 返回结果：返回 true 表示成功识别数字常量，并存储到传入的 Token 对象中。

---

2 作业 (选做部分)

题目 1 (手写词法分析器)

- 为 `FloatingConstant` 词法单元手写词法分析器，通过与 ANTLR 4 的输出进行对比检查正确性。建议画出状态转移图。

解答：

下面给出一个简单的 Java 实现的浮点数识别的代码：

```
public class Main {
public static String lexFloatingConstant(String inputString) {
    int position = 0;
    char currentChar = inputString.charAt(position);

    StringBuilder floatingConstant = new StringBuilder();

    // Initial state
    String state = "start";

    while (currentChar != '\0') {
        if (state.equals("start")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
                state = "digit_sequence";
            } else if (currentChar == '.') {
                floatingConstant.append(currentChar);
                state = "fractional_constant";
            } else {
                break;
            }
        } else if (state.equals("digit_sequence")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
            } else if (currentChar == 'e' || currentChar == 'E' || currentChar == 'p'
                floatingConstant.append(currentChar);
                state = "exponent_part";
            } else if (currentChar == '.') {
                floatingConstant.append(currentChar);
```

```
                state = "fractional_part";
            } else {
                break;
            }
        } else if (state.equals("fractional_constant")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
                state = "fractional_part";
            } else {
                break;
            }
        } else if (state.equals("fractional_part")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
            } else if (currentChar == 'e' || currentChar == 'E' || currentChar == 'p'
                floatingConstant.append(currentChar);
                state = "exponent_part";
            } else {
                break;
            }
        } else if (state.equals("exponent_part")) {
            if (currentChar == '+' || currentChar == '-') {
                floatingConstant.append(currentChar);
                state = "exponent_sign";
            } else if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
                state = "exponent_digit_sequence";
            } else {
                break;
            }
        } else if (state.equals("exponent_sign")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
                state = "exponent_digit_sequence";
            } else {
                break;
            }
        } else if (state.equals("exponent_digit_sequence")) {
            if (Character.isDigit(currentChar)) {
                floatingConstant.append(currentChar);
            } else {
                break;
            }
        }

        position++;
        if (position < inputString.length()) {
            currentChar = inputString.charAt(position);
        } else {
            currentChar = '\0';
        }
    }
```

```
        return floatingConstant.toString().trim();
    }

    public static void main(String[] args) {
        String inputString = "3.14E+12";
        String floatingConstant = lexFloatingConstant(inputString);
        System.out.println(floatingConstant);
    }
}
```

状态转移图的文字描述如下，首先给出状态含义：

- start：初始状态，开始识别浮点数常量。
- digit_sequence：识别数字序列部分。
- fractional_constant：识别小数部分。
- fractional_part：继续识别小数部分。
- exponent_part：识别指数部分。
- exponent_sign：识别指数部分的正负号。
- exponent_digit_sequence：识别指数部分的数字序列。

接下来，我们给出状态转移描述：

(1) 从初始状态 'start' 开始。

(2) 如果当前字符是数字（[0-9]），转移到状态"digit_sequence"。

(3) 如果当前字符是小数点（'.'），转移到状态"fractional_constant"。

(4) 如果当前字符是'e' 或'E' 或'p' 或'P'，转移到状态"exponent_part"。

(5) 如果当前状态是"digit_sequence"，可能转移到状态"digit_sequence"、"fractional_part" 或"exponent_part"，取决于下一个字符。

(6) 如果当前状态是"fractional_constant"，可能转移到状态"fractional_part"，取决于下一个字符。

(7) 如果当前状态是"fractional_part"，可能继续识别数字序列，也可能转移到状态"exponent_part"，取决于下一个字符。

(8) 如果当前状态是"exponent_part"，可能转移到状态"exponent_sign"，取决于下一个字符。

(9) 如果当前状态是"exponent_sign"，可能转移到状态"exponent_digit_sequence"，取决于下一个字符。

(10) 如果当前状态是"exponent_digit_sequence"，可能继续识别数字序列。

---

3 反馈

请在 Zulip 上讨论对作业或者课程的意见。