编译原理

实验（二）词法分析器

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2023年3月26日
学　　院	信息学院
课程名称	编译原理

实验（二）词法分析器

目录

1 实验目的

掌握词法分析器的构造原理，掌握手工编程或LEX编程方法之一。

2 实验内容

编写一个LEX源程序，使之生成一个词法分析器，能够把输入的源程序转换为词法单元序列输出。

3 实现思路

我既完成了手工编程，也完成了使用LEX编程实现。

使用手工编程，可以参考关键字、表达式等词法单元的定义，手工处理歧义等问题，对源代码识别。

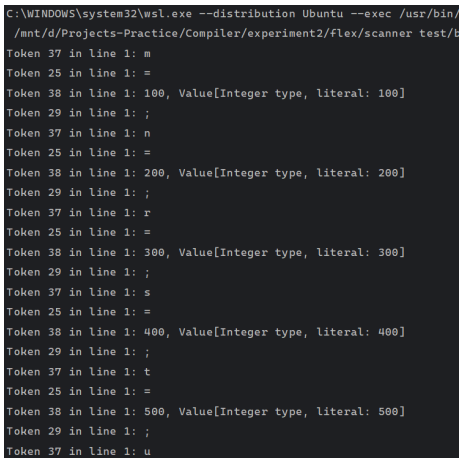
使用LEX编程，需要写出准确的正则表达式，对源代码识别。在识别的过程中，可以执行指定的动作，在执行动作时可以将识别的词法单元及其附加信息保存在结构体中构成一列表。再进行输出。定义的词法单元正则表达式如代码 1。

代码 1 词法单元正则表达式

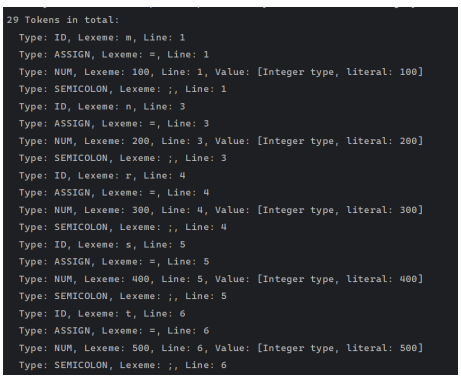
```
delim     [ \t\r\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?(f|lf|sz)?
eoc       \*\/
```

运行效果如图 1。

图 1: 运行结果



(a) 使用lex



(b) 手工编写

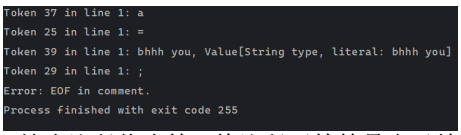
4 问题一、如何良好地处理嵌套块注释？

良好地处理嵌套注释不仅能正确地忽略注释内容，还能对不匹配的错误代码正常地报错。

嵌套块注释是一种常见的现象。在C语言中/* */是块注释，这一对符号可以嵌套使用，处理时应当以最外层的块注释为准。

在识别块注释时，我们不但应当正确识别正确的块注释，还应当正确地识别错误的块注释，并且进行报错，例如块注释开始符号少于结束符号、多于结束符号等状况。例如图 2展示了本次LEX实现中对不匹配块注释的两种报错。 可见该分析器能正确识别错误并给出报错。

图 2: 错误注释的情况



(a) 缺少注释终止符（块注释开始符号多于结束符号）

(b) 缺少注释开始符号（块注释开始符号少于结束符号）

4.1 正则表达式不可能良好地处理嵌套注释

良好地处理嵌套注释不仅能正确地忽略注释内容，还能对不匹配的错误代码正常地报错。

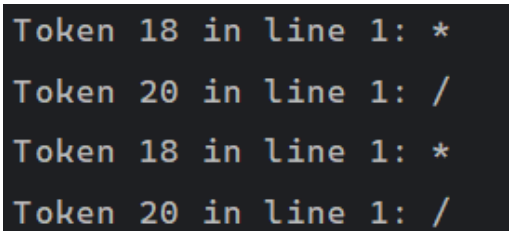
最简单的处理嵌套注释的方式是使用正则表达式，如代码 **??**所示。 然而，这种方式对错误的注释处理不

代码 2 处理嵌套注释正则表达式

```
[\/][\*]([^\*])*[\*]([\*]|[^\*\/](([^\*])*)[\*])*(\/)
```

够完善，例如处理串`/*aa/*bb*/*/*/`时会有多余输出，如图 3。

图 3: 错误注释的情况



这种现象产生的原因是源串中只有被/**/匹配对正确包含的bb会被作为注释匹配到，然后其它的元素则会被其他规则匹配到，因而不能够正确识别这是一种错误，然后输出相应的、类似我的实现的、如图 2中的错误消息。而在真实世界的编译器中，这一错误的Token序列会进入到后续的工作中，最终因为不符合语法或者语义规则而报错。然而一旦将报错推迟到后续阶段，由于被部分删除的注释和不完整的Token序列已经没有办法提供充足的信息来生成合理的、易读的、用户友好的错误报告，因此这种方式有致命的缺陷。

在手工编写的Lexer中，可以使用了一个栈来处理注释开始符号的嵌套，当遇到注释开始符号时，将其入栈，遇到注释结束符号时，将其出栈，当栈为空时，说明注释结束，否则说明注释未结束。如果缺少注释终结符号，当文件到达EOF时，栈仍不为空，则可以针对性报错，如果缺少注释开始符号，当遇到注释结束符号时，栈为空，则也可以针对性报错。仅仅使用正则表达式在这种情况下是无法处理的。那么使用flex如何生成一个能够良好处理嵌套注释的词法分析器呢？

4.2 使用Start Condition良好地处理嵌套块注释

Start condition是LEX提供的一种功能。它允许用户在词法分析器中定义多个状态，每个状态都有一个名字，当使用`BEGIN` 动作时可以激活一个start condition。此时，只有针对这一start condition的规则是有

效的。为了给规则限定start condition，可以在规则的前方进行声明。默认的、一进入LEX中的start condition是INITIAL。

为了处理嵌套注释，可以使用一个start condition，再定义特殊的规则，在这些规则的动作中维护栈的状态。这样就可以使用上述栈的方法处理注释的嵌套。当嵌套注释处理完毕时，回到INITIAL环境，这样就能处理正确的嵌套注释。

而对于错误的嵌套注释，上述切换的过程给予我们足够的信息进行具有针对性的报错：

1. 如果处在注释的start condition中，而且读到了EOF，则说明缺少注释终止符号。
2. 如果不在注释的start condition中，而且读到了*/，则说明缺少注释开始符号。

因此，借助start condition，我们可以很容易地实现良好地处理嵌套注释及其错误情况的报错。

此外，由于我们只需要判断是否栈空，因此进出栈的操作可以简化为变量的加减操作，这样可以减少内存访问的开销。

使用这一思路处理嵌套注释的LEX定义如代码 3所示。

代码 3 处理嵌套注释正则表达式

```
%x COMMENT
%%
%{
   int _yycmtnest = 0;
%}
"/*"            { BEGIN(COMMENT); ++_yycmtnest;}
"//".*          /* // comments to end of line */
<COMMENT>[^*/]* /* Eat non-comment delimiters */
<COMMENT>"/*"   {++_yycmtnest;}
<COMMENT>"*/"   {if (--_yycmtnest == 0) BEGIN(INITIAL);}
<COMMENT>[*/]   /* Eat a / or * if it doesn't match comment sequence */
<COMMENT><<EOF>> { yyerror("EOF in comment."); }
{eoc}            { yyerror("Stray */ symbol in code."); }
%%
```

以上代码实现了上述分析得出的所有必要的处理，能够实现如图 2中的错误报告。

注释这一在词法分析过程中需要被移除的内容，可以在词法分析阶段完成所有的错误处理，并且有能力给出易读、准确的错误报告，这样对于嵌套注释的处理才是正确、良好的。

5 问题二、如何处理字符串常量

处理字符串常量和处理注释类似，虽然不需要考虑嵌套的情况，但需要考虑当引号不匹配时如何正确地报告错误。

引号的错误也相对比较简单。由于使用相同的符号标记字符串的开始和结束，因此实质上的错误只有一种：缺乏字符串的结束符。因为若多出一个引号，那么这个引号会被当作下一个字符串的开始，而不会被当作多出了字符串的结束符，这样处理的合理性在于，在许多语言中两个相邻的字符串可以被合并为一个字符串，因此如此处理多出的引号更符合实践中的操作。

因此，仿照处理注释的方法，可以使用start condition来处理字符串常量。当遇到字符串常量的开始符号时，进入字符串常量的start condition，如果在其中读到了EOF，则可以认为缺乏字符串的结束符。如代码 4。

代码 4 问题二、如何处理字符串常量

```
%x STRING
%%
%{
   int _yycmtnest = 0;
%}
["]              { BEGIN(STRING);}
<STRING>[^"]*    { EMIT_TOKEN(TK_STRING); }
<STRING>["]      { BEGIN(INITIAL);}
<STRING><<EOF>> { yyerror("EOF in string."); }
%%
```

6 问题三、如何良好区分不同类型的字面量

不同类别的字面量通常在语法分析阶段需要进行不同的处理，而在词法分析阶段有足够的信息可以对不同类型的字面量进行区分。

以上代码中出现了EMIT_TOKEN宏，该宏定义进一步调用另外实现的 create_token函数，对LEX处理出的词法单元文本进一步处理，在这一过程中，可以对常量进行进一步的处理，以便于对不同类型的字面量进行不同的处理。

代码 5 区分不同类型的字面量

```
extern "C" void *create_token(token_type t, const char *s, int l)
{
        string lexeme{s};
        if (t != TK_NUM && t != TK_STRING){
                return new token{t, lexeme, l, {}};
        }else{
                if (t == TK_NUM){
                        bool floating = false;
                        for (char p: lexeme){
                                if (p == '.'){
                                        floating = true;
                                        break;
                                }
                        }
                        if (floating){
                                return new token{t, lexeme, l, {stod(lexeme)}};
                        }else{
                                return new token{t, lexeme, l, {stoi(lexeme)}};
                        }
                }else{
                        return new token{t, lexeme, l, {lexeme}};
                }
        }
}
```

对于TK_STRING类型，可以不做特殊处理，也可以做特殊处理：将其中符合转义规则的部分进行替换。在这里不做处理。对于TK_NUM类型的Token，它可能携带整数，也可能携带浮点数。因此，可以进一步扫描字面量的文本。例如科学计数法、小数点、类型后缀的部分均可以提示其类型。然后，可以使用共用体在存储Token信息的结构体中，进一步保存经过正确转换的数值。

这一部分的实现如代码 5所示。这里使用了C++11引入的std::variant来代替传统C语言的共用体。它可以记录类型信息，并根据其类型获取值，使得程序编写更为便利。

7 对手工编写和LEX生成优劣的讨论

在本次实验中，我分别使用手工编写与LEX生成的方式对一种语言的词法分析器进行编写。我认为，手工编写和LEX生成分别有如下优点：

手工生成

- 更灵活自由
- 能够更方便地实现复杂嵌套块的分析
- 更容易提供良好的错误信息

LEX生成

- 简洁直观
- 更容易保证正确性

从实践的视角来看，在工业界得以广泛应用的编译器，通常具有手工编写的词法分析器和语法分析器，因为真实世界的使用要求效率，就要求给出较好的错误信息，并且往往面临着复杂的语法结构，因此需要更大的灵活性和自由度。而学术性更强的语言，例如Scala、OCaml等语言通常使用LEX生成的词法分析器，因为这样一来可以忽略实现细节、直接使用规则生成对应的词法分析器。因而能够更方便地对各种特性进行研究和讨论，而不受限于具体的实现细节。

从历史的视角来看，随着一门编程语言从实验室走到日常使用的环境中去，其词法分析器和语法分析器通常都会从工具生成的方式走向手工编写。例如GCC，早期版本的GCC使用生成的词法分析和语法分析器。随着实际的使用中暴露的问题，例如不够友好地错误信息，日渐复杂的语法定义和对性能的要求；以及竞争对手Clang使用手工编写的词法分析和语法分析所带来的良好体验，从3.x版本开始[1]，GCC的词法分析和语法分析被替换成了手工编写的版本。这一过程持续了十多年，直到GCC 4.0版本，这一过程才正式完成。

手工编写的优势不仅体现在错误处理和恢复上，更体现在对复杂语言特性的处理上，这是由于其灵活自由的特点决定的。使用工具生成则缺乏这一特点，例如工具必须严格地使用正则描述词法规则，使用上下文无关文法描述语法规则。这一过程中，词法分析和语法分析严格分开。但是，面对复杂的语言特性，比如C++中的模板特性，当嵌套模板发生时，例如`std::vector<std::vector<int>>`，这时候“如何判断>>是一个Token还是两个Token”就成了一个问题。而解决这一问题，再复杂的正则、表达能力再强的语法都不如使语法分析和词法分析共同进行、互相使用对方的中间状态来的更为可行。然而，一旦使用了工具生成，就很难做到这一点。

当然，在面对实际的问题时，常常比起“哪一种方案更有学术上的正确性”，更需要确定“哪一种方案的正外部性和负外部性综合而言更能令人接受”。即使手工编写可能不如正则表达式直接定义来得正确、精准，从这一点出发，我认为手工编写和LEX生成的优劣并不是一个绝对的问题，而应当具有同等重要的地位、应当面向实际的需求做出正确的trade-off，选择最合适的方案。

[1] `https://gcc.gnu.org/wiki/New_C_Parser`

A 附录A、完整的LEX定义

图 4: 完整的LEX定义

```
%{
#include <stdio.h>
#include <stdlib.h>

#include "lexer.h"

#define EMIT_TOKEN(type) yylval=create_token(type,yytext,yylineno); return (type);

struct token* yylval;
int yylines=0;

static void yyerror(char *msg)
{
    fprintf(stderr, "Error: %s", msg);
    exit(-1);
}

%}

delim    [ \t\r\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?(f|lf|sz)?
eoc      \*\/

%x COMMENT
%x STRING

%%
%{
   int _yycmtnest = 0;
%}
"/*"            { BEGIN(COMMENT); ++_yycmtnest;}
"//".*          /* // comments to end of line */
<COMMENT>[^*/]* /* Eat non-comment delimiters */
<COMMENT>"/*"   {++_yycmtnest;}
<COMMENT>"*/"   {if (--_yycmtnest == 0) BEGIN(INITIAL);}
<COMMENT>[*/]   /* Eat a / or * if it doesn't match comment sequence */
<COMMENT><<EOF>> { yyerror("EOF in comment."); }
{eoc}            { yyerror("Stray */ symbol in code."); }

["]             { BEGIN(STRING);}
<STRING>[^"]*   { EMIT_TOKEN(TK_STRING); }
<STRING>["]     { BEGIN(INITIAL);}
```

```
<STRING><<EOF>> { yyerror("EOF in string."); }

{ws}        { /*no action and no return */ }
if          { EMIT_TOKEN(TK_IF); }
while       { EMIT_TOKEN(TK_WHILE);}
do          { EMIT_TOKEN(TK_DO);}
break       { EMIT_TOKEN(TK_BREAK);}
true        { EMIT_TOKEN(TK_TRUE); }
false       { EMIT_TOKEN(TK_FALSE);}
int         { EMIT_TOKEN(TK_INT);  }
char        { EMIT_TOKEN(TK_CHAR);  }
bool        { EMIT_TOKEN(TK_BOOL); }
float       { EMIT_TOKEN(TK_FLOAT); }

{id}        { EMIT_TOKEN(TK_ID); }
{number}    { EMIT_TOKEN(TK_NUM); }

\(          { EMIT_TOKEN(TK_LPAREN);}
\)          { EMIT_TOKEN(TK_RPAREN);}
\]          { EMIT_TOKEN(TK_LBRACKET); }
\[          { EMIT_TOKEN(TK_RBRACKET); }
\{          { EMIT_TOKEN(TK_LBRACE); }
\}          { EMIT_TOKEN(TK_RBRACE); }


";"         { EMIT_TOKEN(TK_SEMICOLON); }
","         { EMIT_TOKEN(TK_COMMA); }
"+"         { EMIT_TOKEN(TK_PLUS); }
"-"         { EMIT_TOKEN(TK_MINUS); }
"*"         { EMIT_TOKEN(TK_MULTIPLY); }
"/"         { EMIT_TOKEN(TK_DIVIDE); }
"<"         { EMIT_TOKEN(TK_LT); }
"<="        { EMIT_TOKEN(TK_LTE); }
"=="        { EMIT_TOKEN(TK_ASSIGN); }
"="         { EMIT_TOKEN(TK_EQ); }
"!="        { EMIT_TOKEN(TK_NEQ); }
">"         { EMIT_TOKEN(TK_GT); }
">="        { EMIT_TOKEN(TK_GTE); }

.           { yyerror("Unexpected character.");}
%%

int yywrap()
{
    return 1;
}
```

B 附录B、完整的手工实现

图 5: 完整的手工实现

```
#include <format>
#include <iostream>

#include "lexer.h"

using namespace std;

std::vector<token> scanner::scan()
{
        while (!is_end())
        {
                start_ = current_;
                scan_next();
        }

        tokens_.emplace_back(token_type::EOF_TOKEN, "", line_);
        return tokens_;
}

void scanner::scan_next()
{
        char c = advance();
        switch (c)
        {
        case '(':
                add_token(token_type::LPAREN);
                break;
        case ')':
                add_token(token_type::RPAREN);
                break;
        case '[':
                add_token(token_type::LBRACKET);
                break;
        case ']':
                add_token(token_type::RBRACKET);
                break;
        case '{':
                add_token(token_type::LBRACE);
                break;
        case '}':
                add_token(token_type::RBRACE);
                break;
        case ',':
                add_token(token_type::COMMA);
                break;
        case '-':
```

```
        if (match('-'))
        {
                add_token(token_type::DMINUS);
        }
        else
        {
                add_token(token_type::MINUS);
        }
        break;
case '+':
        if (match('+'))
        {
                add_token(token_type::DPLUS);
        }
        else
        {
                add_token(token_type::PLUS);
        }
        break;
case ';':
        add_token(token_type::SEMICOLON);
        break;
case '*':
        if (match('*'))
        {
                add_token(token_type::POWER);
        }
        else
        {
                add_token(token_type::MULTIPLY);
        }
        break;
case '!':
        add_token(match('=') ? token_type::NEQ : token_type::NOT);
        break;
case '=':
        add_token(match('=') ? token_type::EQ : token_type::ASSIGN);
        break;
case '<':
        add_token(match('=') ? token_type::LTE : token_type::LT);
        break;
case '>':
        add_token(match('=') ? token_type::GTE : token_type::GT);
        break;

case '/':
        if (match('/')) // this is a line comment
        {
                consume_line_comment();
        }
```

```
                else if (match('*')) // this is a block comment
                {
                        consume_block_comment();
                }
                else
                {
                        add_token(token_type::DIVIDE);
                }
                break;

        case ' ':
        case '\r':
        case '\t':
                // Do nothing to ignore whitespaces.
                break;

        case '\n':
                line_++;
                break;

        case '"':
                scan_string();
                break;

        default:
                if (is_number_literal_component(c))
                {
                        scan_number_literal();
                }
                else if (is_identifier_component(c))
                {
                        scan_identifier();
                }
                else
                {
                        cerr << format("Unexpected character {}.", c) << endl;
                }

                break;
        }
}

void scanner::scan_string()
{
        while (peek() != '"' && !is_end())
        {
                if (peek() == '\n')
                {
                        line_++;
                }
```

```
                advance();
        }

        if (is_end())
        {
                cerr << "Unterminated string." << endl;
                return;
        }

        advance(); // eat the closing "

        auto lexeme = whole_lexeme();
        add_token(token_type::STRING, lexeme.substr(1, lexeme.size() - 2));
}

void scanner::scan_number_literal()
{
        while (is_number_literal_component(peek()))
                advance();

        bool floating{false};
        if (peek() == '.' && is_number_literal_component(peek(1)))
        {
                floating = true;
                advance();
                while (is_number_literal_component(peek()))
                        advance();
        }

        if (floating)
        {
                add_token(token_type::NUM, stod(whole_lexeme()));
        }
        else
        {
                add_token(token_type::NUM, stoi(whole_lexeme()));
        }
}

void scanner::scan_identifier()
{
        while (is_identifier_component(peek()))
                advance();

        auto text = whole_lexeme();
        auto keyword = keywords_to_type_.find(text);

        if (keyword != keywords_to_type_.end())
        {
                add_token(keywords_to_type_[text]);
```

```
        }
        else
        {
                add_token(token_type::ID);
        }
}

void scanner::consume_block_comment()
{
        while (!is_end())
        {
                auto c = advance();
                if (c == '\n')
                {
                        line_++;
                }
                else if (c == '*' && peek() == '/')
                {
                        [[maybe_unused]]auto _ = advance(); // eat "/"
                        return;
                }
                else if (c == '/' && peek() == '*') // nested block comment
                {
                        [[maybe_unused]]auto _ = advance(); // eat "*"
                        consume_block_comment();
                }
        }

        // not enough code to find next close sign */, so it's an error
        if (is_end())
        {
                cerr << "Unterminated block comment." << endl;
        }
}

void scanner::consume_line_comment()
{
        while (peek() != '\n' && !is_end())
                advance();
}

void scanner::add_token(token_type t)
{
        tokens_.emplace_back(t, whole_lexeme(), line_);
}

void scanner::add_token(token_type t, const literal_type &lit)
{
        tokens_.emplace_back(t, whole_lexeme(), line_, lit);
}
```