词法分析器项目文档

李东洋 16307130226
蔡彦麓 16307130222
杨俊逸 16307130239
吴钟立 15307130213

一、flex概览及部分接口的使用方法

flex是一个用来生成扫描器的工具。所谓扫描器就是可以从文本流中探测出词法模式的程序，亦即词法分析器。flex以由一系列规则构成的文件作为输入，生成一个C源文件，默认的输出文件名是`lex.yy.c`，其中定义了最主要的例程`yylex()`。接下来编译、链接该C源文件，得到的可执行文件可以分析输入的字符串中正则表达式的匹配情况，当它发现匹配时，就执行规则中对应的C代码。

flex生成的词法分析器中定义了一系列有用的接口和全局变量，本次实验中我们使用的接口和全局变量有：`int yylex()`, `FILE* yyin`, `char* yytext`和`int yyleng`。其中，`yylex()`是词法分析器的主程序。当它被调用时，它从输入文件对应的文件指针`yyin`处开始，按照规则中的正则表达式匹配token。当匹配成功或遇到文件结尾EOF时该函数返回。若为文件结尾，则`yylex()`返回0；若找到一处匹配，则该函数执行规则中匹配的正则表达式对应的C代码作为返回值。

当用flex生成的扫描器运行时，其主要过程就是检查输入是否匹配给定的模式，如果发现超过一个匹配，它将选择最长匹配；如果存在长度相同的匹配，则选择第一个列出的规则。一旦匹配成功，则全局指针`yytext`就指向导致匹配发生的字符串，其长度存储在整型的全局变量`yyleng`中。

根据官方文档，`yytext`有两种声明方式：字符指针(char*)或字符数组(char [])。使用指针的好处是可以较大程度上加快扫描速度，而且不会有缓冲区溢出的问题；缺点是将不能自由的修改`yytext`指向的内存。

二、项目框架

项目总体框架如下：

```
lexical_analysis/
|
|---include/
|   |---lexAnalyzer.h       //使用的flex接口及全局变量的声明、Token和LexAnalyzer的类定义
|   |---lexer.h             //可以被匹配的Token类型以及错误的定义(枚举类型)
|
|---src/
|   |---lexAnalyzer.cpp     //LexAnalyzer类的接口实现
```

```
|   |---lexer.lex           //规则定义，用于flex生成扫描器
|
|---CMakeLists.txt          //cmake和make的命令
|---main.cpp            //词法分析器的主函数，依次调用LexAnalyzer类的三个接口进行词法分析

LexAnalyzer类是词法分析器的抽象，定义了三个接口：
LexAnalyzer::analyze()      //调用yylex()函数进行词法分析，将提取到的token保存在vector中，并
//根据token类型记录行号、列号等信息
LexAnalyzer::sanityCheck()  //遍历保存的token，检查是否存在非法字符串、整数溢出、非法变量名、
//未终结的注释等错误，若存在则将该token替换为error类型及相应的                    //错误信息
LexAnalyzer::display()      //将保存的token按照格式输出
```

三、实现细节

1、正则表达式部分

本次实验的目标是实现面向PCAT语言的词法分析器。根据PCAT语言的定义(参考手册)以及目标实现的token类型，我们定义正则表达式如下：

```
//lexer.lex

DIGIT       [0-9]
INTEGER     {DIGIT}+
REAL        {DIGIT}+"."{DIGIT}*
COMMENT     \(\*(([^*])|(\**[^*)]))*\**\*\)
UNTERMINATED_COMMENT    \(\*(([^*])|(\**[^*)]))*\**$
UNTERMINATED_STRING     \"[^\"\n]*(\n|$)
WS          [ \t]+
NEWLINE     \n
UNKNOWN     .|\n
LETTER      [a-zA-Z]
STRING      \"[^\"\n]*\"
ID          {LETTER}({LETTER}|{DIGIT})*

ASSIGN      :=
PLUS        \+
MINUS       -
STAR        \*
SLASH       \/
LT          <
LE          <=
GT          >
GE          >=
EQ          =
NEQ         <>

COLON       :
SEMICOLON   ;
COMMA       ,
DOT         \.
LPAREN      \(
RPAREN      \)
LBRACKET    \[
```

```
RBRACKET    \]
LBRACE      \{
RBRACE      \}
LABRACKET   \[<
RABRACKET   >\]
BACKSLASH   \\
```

这里就几条比较复杂的正则表达式进行说明。

COMMENT

注释的正则表达式定义为`\(\*(([^*])|(\**[^*)]))*\**\*\)`。其中开头的`\(\*`和末尾的`\*\)`是注释的开始和结束符；中间部分`(([^*])|(\**[^*)]))*`表示，一个非*的字符，或者*字符重复0或多次后，后接一个非*且非)的字符——这两种模式的并集的星闭包，即每次从这两种模式中任选一种，连接0次或多次。这样可以保证注释中间不提前出现注释结束符；但上述模式没有覆盖到以*结尾的注释正文，故紧随其后的`\**`处理了这种情况。

UNTERMINATED_COMMENT

未终结的注释的正则表达式定义为`\(\*(([^*])|(\**[^*)]))*\**$`。该表达式即为合法注释的正则表达式删去匹配注释结束符后的部分加上匹配文件末尾的符号，且由于该表达式匹配的字符串，或者以*结尾，或者以非*字符结尾且前一个字符不为*，或者以非*且非)的字符结尾，故它不会匹配合法注释。

UNTERMINATED_STRING

未终结的字符串的正则表达式定义为`\"[^\"\n]*(\n|$)`。这里不允许字符串内部存在换行符，故若在匹配字符串的过程中，在匹配到右引号之前匹配到换行符，或者匹配到文件末尾仍未出现引号，则视为出现了字符串未终结的错误。

2、规则定义

根据上述定义的正则表达式匹配到token后，相应的行为定义如下：

```
//lexer.lex

{ASSIGN}    |
{PLUS}      |
{MINUS}     |
{STAR}      |
{SLASH}     |
{LT}        |
{LE}        |
{GT}        |
{GE}        |
{EQ}        |
{NEQ}       return OPERATOR;

{COLON}     |
{SEMICOLON} |
{COMMA}     |
{DOT}       |
{LPAREN}    |
{RPAREN}    |
{LBRACKET}  |
{RBRACKET}  |
{LBRACE}    |
{RBRACE}    |
{LABRACKET} |
```

```
{RABRACKET} |
{BACKSLASH} return DELIMITER;

AND         |
ARRAY       |
BEGIN       |
BY          |
DIV         |
DO          |
ELSE        |
ELSIF       |
END         |
EXIT        |
FOR         |
IF          |
IN          |
IS          |
LOOP        |
MOD         |
NOT         |
OF          |
OR          |
OUT         |
PROCEDURE   |
PROGRAM     |
READ        |
RECORD      |
RETURN      |
THEN        |
TO          |
TYPE        |
VAR         |
WHILE       |
WRITE       return RESERVED_KEY;

{WS}        return WS; // skip blanks and tabs
{NEWLINE}   return NEWLINE;
{UNTERMINATED_STRING} return UNTERMINATED_STRING;
{UNTERMINATED_COMMENT} return UNTERMINATED_COMMENT;
{ID}        return ID;
<<EOF>>     return T_EOF;
{INTEGER}   return INTEGER;
{COMMENT}   return COMMENT;
{REAL}      return REAL;
{STRING}    return STRING;
{UNKNOWN}   return UNKNOWN;
```

其中，各个return语句均返回一个整数，其值定义在include/lexer.h中的枚举类型变量type中。OPERATOR表示运算符，DELIMITER表示分隔符和界定符，RESERVED_KEY表示PCAT语言的保留字。

3、LexAnalyzer类

LexAnalyzer类定义如下：

```
//include/lexAnalyzer.h
```

```
2
3  class LexAnalyzer {
4  public:
5      ~LexAnalyzer() {}
6      // static void init(FILE* file=nullptr);
7      static LexAnalyzer* GetInstance() {
8          if (m_instance == nullptr) {
9              m_instance = new LexAnalyzer();
10         }
11         return m_instance;
12     }
13
14     void analyze();
15     void sanityCheck();
16     void display();
17
18 private:
19     LexAnalyzer() : errorCount(0) {}
20     static LexAnalyzer* m_instance;
21     std::vector<Token> tokens;
22     int errorCount;
23 };
```

其中，m_instance是指向该类的唯一实例的指针。我们将类的构造函数设为private，故无法通过新建对象的方式创建类的实例，而只能通过public接口GetInstance()访问m_instance指针指向的实例，从而保证整个程序只存在一个该类的实例。另外三个public接口analyze()，sanityCheck()和display()的定义将在下文叙述。

LexAnalyzer类的私有成员包括构造函数(将errorCount初始化为0)，m_instance指针，Token类型的vector容器tokens(用于保存提取的token信息)，以及errorCount(统计出现的错误个数)。

4、Token的定义

```
1  //include/lexAnalyzer.h
2
3  struct Token {
4      int type;
5      std::string token;
6      int row;
7      int col;
8      Token() {}
9      explicit Token(int type, std::string token, int row, int col) : type(type), token(token),
   row(row), col(col) {}
10 };
```

Token的定义较为简单。type对应匹配到的token的类型，token保存匹配到的字符串，row和col记录该token出现的起始位置。

5、LexAnalyzer接口定义

analyze()

```
1  //src/lexAnalyzer.cpp
2
3  void LexAnalyzer::analyze() {
```

```
    int row = 1;
    int col = 1;

    while (true) {
        int type = yylex();
        // std::cout << yytext << ' ' << type2str[type] << std::endl;
        if (type == T_EOF) break;
        std::string token(yytext);
        int newlineCount = std::count(token.begin(), token.end(), '\n');
        if (type == NEWLINE) {
            row ++;
            col = 1;
            continue;
        }
        if (type == COMMENT) {
            int p = token.find_last_of('\n');
            row += newlineCount;
            col = token.size() - p;
            continue;
        }

        if (type == WS) {
            col += yyleng;
            continue;
        }

        tokens.push_back(Token(type, token, row, col));
        col += yyleng;
        if (type == UNTERMINATED_STRING) {
            row++;
            col = 1;
        }
        if (type == UNTERMINATED_COMMENT) break;
    }
}
```

该函数的逻辑较为简单：每次循环调用yylex()函数试图匹配token。yylex()函数的返回值保存在type中，即匹配到的模式的类型。若遇到文件末尾或未终结的注释则结束匹配。这里为了与官方手册中yylex()的行为相匹配，T_EOF的值设为0。对于一次成功的匹配，yytext指针指向被匹配的字符串的起始位置，同时该字符串的长度保存在yyleng变量中。使用yytext指针初始化字符串token，然后根据匹配的类型相应地调整行号和列号。如遇到换行符(NEWLINE)则行号加1，列号置为1；遇到非换行符的空白符(空格和制表符)则简单将列号加上yyleng。对注释的处理稍微复杂一些：首先要统计token中换行符的个数newlineCount，然后对于多行的注释，新的行号为注释所占的行数(单行注释行号不变)，新的列号为注释的总长度减去最后一个换行符出现的位置。注释、换行符和空白符不被保存在vector中，而只是修改起始行号和列号；其他类型的token将先被保存在vector中，然后修改行号和列号。其中对于未终结字符串，根据前面的定义，对行号和列号的处理为行号加1，列号置为1；对于其他类型的的token，由于不会跨行，因此只需简单将列号加上token的长度yyleng即可。

sanityCheck()

```
void LexAnalyzer::sanityCheck() {
    for (Token& t : tokens) {
        switch (t.type) {
            case STRING:
                if (t.token.find_first_of("\t\n") != std::string::npos) {
                    t.type = ERROR;
```

```
                    t.token = "Error: String contains tabs or newlines.";
                }
                else if (t.token.size() > 257) {
                    t.type = ERROR;
                    t.token = "Error: String length out of range.";
                }
                break;
            case INTEGER:
                if (t.token.size() > 10 || (t.token.size()==10 && t.token > "2147483647")) {
                    t.type = ERROR;
                    t.token = "Error: Integer out of range.";
                }
                break;
            case ID:
                if (t.token.size() > 255) {
                    t.type = ERROR;
                    t.token = "Error: Identifier length out of range.";
                }
                break;
            case UNTERMINATED_COMMENT:
                t.type = ERROR;
                t.token = "Error: unterminated comment.";
                break;
            case UNTERMINATED_STRING:
                t.type = ERROR;
                t.token = "Error: unterminated string.";
                break;
            case UNKNOWN:
                t.type = ERROR;
                t.token = "Error: unknown token.";
                break;
            default:
                break;
        }
        if (t.type == ERROR) errorCount++;
    }
}
```

如前所述，该函数遍历提取到的token，根据其类型检查是否存在相应的错误，如字符串长度溢出、整数值溢出、ID长度溢出等。若存在这些错误，或者token的类型是`UNTERMINATED_COMMENT`、`UNTERMINATED_STRING`或`UNKNOWN`，则将token的`type`修改为`ERROR`，并将token的内容(字符串)修改为相应的错误信息，最后将`errorCount`加1。

display()

```cpp
void LexAnalyzer::display() {
    std::cout << std::setw(5) << std::left << "Row" << std::setw(5) << std::left \
        << "Col" << std::setw(20) << std::left << "Type" << "Token/Error" << std::endl;
    for (Token t : tokens) {
        int row = t.row;
        int col = t.col;
        std::string type = type2str[t.type];
        std::string token = t.token;
        std::cout << std::setw(5) << std::left << row << std::setw(5) << std::left \
            << col << std::setw(20) << std::left << type << token << std::endl;
    }
    std::cout << errorCount << " errors, " << tokens.size() - errorCount << " tokens." <<
std::endl;
}
```

该函数按格式要求依次输出匹配到的token，此处不再赘述。

四、实验结果

以一段测试代码为例：

```
(* test01:               *)
(*   test var decls.     *)
(*                       *)
PROGRAM IS
    VAR i, j : INTEGER := 1;
    VAR x : REAL := 2.0;
    VAR y : REAL := 3.0;
BEGIN
    WRITE ("i = ", i, ", j = ", j);
    WRITE ("x = ", x, ", y = ", y);
END;
```

对测试代码文件进行词法分析，输出结果如下：

```
ldy@ubuntu:~/code/CompilerPJ/PJ/build$ ./lexAnalyzer ../tests/test01.pcat
Row   Col   Type                Token/Error
4     1     reserved keyword    PROGRAM
4     9     reserved keyword    IS
5     5     reserved keyword    VAR
5     9     identifier          i
5     10    delimiter           ,
5     12    identifier          j
5     14    delimiter           :
5     16    identifier          INTEGER
5     24    operator            :=
5     27    integer             1
5     28    delimiter           ;
6     5     reserved keyword    VAR
6     9     identifier          x
6     11    delimiter           :
6     13    identifier          REAL
6     18    operator            :=
6     21    real                2.0
6     24    delimiter           ;
7     5     reserved keyword    VAR
7     9     identifier          y
7     11    delimiter           :
7     13    identifier          REAL
7     18    operator            :=
7     21    real                3.0
7     24    delimiter           ;
8     1     reserved keyword    BEGIN
9     5     reserved keyword    WRITE
9     11    delimiter           (
9     12    string              "i = "
9     18    delimiter           ,
9     20    identifier          i
9     21    delimiter           ,
9     23    string              ", j = "
9     31    delimiter           ,
9     33    identifier          j
9     34    delimiter           )
9     35    delimiter           ;
10    5     reserved keyword    WRITE
10    11    delimiter           (
10    12    string              "x = "
10    18    delimiter           ,
10    20    identifier          x
10    21    delimiter           ,
10    23    string              ", y = "
10    31    delimiter           ,
10    33    identifier          y
10    34    delimiter           )
10    35    delimiter           ;
11    1     reserved keyword    END
11    4     delimiter           ;
0 errors, 50 tokens.
```

输出结果符合预期，输出格式符合要求。

五、分析与总结

本次实验，我们借助flex工具，基于正则表达式匹配，设计完成了一个面向PCAT语言的词法分析器。该词法分析器可以完成对注释、字符串、整数、实数、ID、保留字、运算符、分隔符等基本代码构件的提取和识别，并且能正确识别注释未终结、字符串未终结、字符串长度溢出、整数数值溢出、ID长度溢出等语法错误，显示相应的错误信息。测试结果表明实现符合预期。

六、小组成员分工

代码：李东洋(25%)、蔡彦麓(25%)

报告：吴钟立(25%)、杨俊逸(25%)