词法分析详细设计

## 一、词法分析概述：

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词或符号。并且在识别过程中跳过不必要的空格和注释。

利用lex工具，我们对不同的字符和类型做操作，通过建立token类型建立识别到的结点，通过yylloc值记录所在行列号。lex通过正则表达式的匹配，遵循最长匹配原则，执行相应的动作。而我们需要输出记号序列。词法分析程序在识别出一个记号后，要把与之有关的信息作为它的属性保留下来，记号影响语法分析的决策，属性影响记号的翻译。

## 二、接口分析

### 1．与语法分析的接口

在词法分析的过程中通过token来传递识别到的字符内容。在token中记录识别到的字符类型、字符串、所在行号。如果每一个关键字、运算符或标点符号作为单独的一类，则记号所代表的单词是唯一的，所以属性域可以是空的。若记号所代表的单词不唯一，则需要给出属性。

class Token

{

public:

    string type;              //类型

    string value;             //值

    int lineNo;               //所在行号

    vector<Token \*> children; //语法树子节点

    Token() {}

    Token(string val) : type(val) {}

    Token(string tp, string val, int ln) : type(tp), value(val), lineNo(ln) {}

};

同时，需要和语法分析统一我们需要识别的字符串种类，明确共有哪一些类型。将关键字和类型定义如下：

 enum yytokentype

  {

    PROGRAM = 258,

    CONST = 259,

    VAR = 260,

    ARRAY = 261,

    OF = 262,

    PROCEDURE = 263,

    FUNCTION = 264,

    \_BEGIN = 265,

    END = 266,

    IF = 267,

    THEN = 268,

    CASE = 269,

    FOR = 270,

    TO = 271,

    DOWNTO = 272,

    DO = 273,

    ELSE = 274,

    REPEAT = 275,

    UNTIL = 276,

    WHILE = 277,

    IDENTIFIER = 278,

    UINUM = 279,

    UFNUM = 280,

    CHAR = 281,

    LETTER = 282,

    INTEGER = 283,

    TYPE = 284,

    ASSIGNOP = 285,

    RELOP = 286,

    EQUAL = 287,

    ADDOP = 288,

    ADD = 289,

    MINUS = 290,

    MULOP = 291,

    NOT = 292,

    RECORD = 293,

    REAL = 294,

    BOOLEAN = 295,

    BOOL\_CONSTANT = 296,

    MUL = 297,

    UMINUS = 298

  };

## 三、各部分功能描述

实现词法分析的lex.l文件如下结构如下：

|  |
| --- |
| %{  Declarations  %}  Definitions  %%  Translation rules  %%  Function segment |

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词或符号。并且在识别过程中跳过不必要的空格和注释。

其中，Declarations是定义、声明部分。这部分通常进行一些头文件声明，变量（全局，外部）、常量的定义，这部分内容会被拷贝到lex.yy.c中。

Definitions是正规定义和状态定义。正规定义是为了简化后面的词法规则而给部分正规式定义了名字。上面正规定义中出现的小括号表示分组，而不是被匹配的字符。而大括号括起的部分表示正规定义名。

Translation rules是词法规则段列出的是词法分析器需要匹配的正规式，以及匹配该正规式后需要进行的相关动作。每行都是一条规则，该规则的前一部分是正规式，需要顶行首写，后一部分是匹配该正规式后需要进行的动作，这个动作是用C语法来写的，被包裹在{}之内，被Lex翻译器翻译后会被直接拷贝进lex.yy.c。正规式和语义动作之间要有空白隔开。其中用{}扩住的正规式表示正规定义的名字。也可以若干个正规式匹配同一条语义动作，此时正规式之间要用 | 分隔。

Function segment是辅助函数段，辅助函数段用C语言语法来写，辅助函数一般是在词法规则段中用到的函数。这一部分一般会被直接拷贝到lex.yy.c中。

## 四、数据结构说明

### 1、正则规则识别字符串分类

我们设计的词法分析中的状态和类型分类如下

%option yylineno

%x CH

%x SCMT

%x MCMT

line \n

digit [0-9]

bool (true|false)

letter [a-zA-Z]

identifier {letter}({letter}|{digit})\*

integer ({digit}+)

floater {digit}+\.{digit}+

expint ({integer})E[+|-]?{integer}

expfloa ({floater})E[+|-]?{integer}

blank\_chars [ \f\r\t\v]+

assginop (:=)

relop (>=|>|<=|<|<>)

addop (or)

add +

minus -

mulop (\\*|\/|div|mod|and)

divop (\(|\)|\[|\]|:|,|;|\.)

%x开头的是我们设计的字符串识别会遇到的模式，黄色标识符是我们识别到该类型的时候给字符串类型进行的取名。

### 2、错误处理相关数据结构

我们对词法分析进行错误处理和记录，将错误处理的信息存储在数组lexicalErrorInformation中。在main中我们可以调用该数组查看词法分析的错误信息。

void addLexicalErrorInformation(char\* word, string info){

    string errorInformation = "[" + info + "] " + itos(yylineno) + ":" + string(word);

    //cout << errorInformation << endl;

    lexicalErrorInformation.push\_back(errorInformation);

}

## 五、算法描述

### 1、对直接识别的字符串进行词法分析

在面对可以直接字符串或者类型的时候，直接用双引号表示字符串，花括号表示识别到的类型，在识别到对应字符串时，直接新建token类型并赋上初值，并对yylloc做相应操作即可。

识别到字符例子如下：

"array" {

    yylval=new Token;

    yylval->value=yytext;

    yylval->type="ARRAY";

    yylval->lineNo = yylineno;

    yylloc.first\_line=yylloc.last\_line=yylineno;

    yylloc.first\_column=yycolumn;

    yylloc.last\_column=yycolumn+yyleng-1;

#ifdef LEXDEBUG

    cout << "keyword: " << yylval->value << endl;

#endif

    return ARRAY;

}

识别到类型例子如下：

 {relop} {

    yylval=new Token;

    yylval->value=yytext;

    yylval->type="RELOP";

    yylval->lineNo = yylineno;

    yylloc.first\_line=yylloc.last\_line=yylineno;

    yylloc.first\_column=yycolumn;

    yylloc.last\_column=yycolumn+yyleng-1;

 #ifdef LEXDEBUG

    cout << "relop:" << yylval->value << endl;

 #endif

    return RELOP;

 }

### 2、对字符类型进行词法分析

在对字符串做词法分析时，因为PASCAL-S语言仅支持字符串中只有一个字符，因此需要对多种情况进行讨论。当遇到’时，则转到CH识别字符类型状态，同时初始化记录的字符串。

"'" {//进入字符常量识别

    BEGIN CH;

    charTemp="";

}

当在CH状态遇到文件结束符时则直接报错。

<CH><<EOF>> {

    addLexicalErrorInformation(yytext, "Unexpected end of file when reading a char constant");

    return 0;

}

当遇到’或者\n时，进行讨论。若此时识别到的字符为空时，则做报错处理，并返回’\0’；若此时识别到的字符串长度不止一个时，则做报错处理，并且错误处理将识别到的第一个字符当作内容返回；如果此时直接遇到换行符，则做报错处理，并根据识别到的字符串长度进行类似上述的讨论；如果识别到单引号并且长度恰为1，则正确返回识别到的内容。

<CH>("'"|"\n") {

    int len = charTemp.size();

    if(yytext[0] == '\'' && len == 0){  //双引号之间无空格,错误处理为\0

        addLexicalErrorInformation(yytext,"Missing character!");

        yylval = new Token;

        yylval -> value = "\0";

        yylval -> type = "LETTER";

        yylval -> lineNo = yylineno;

#ifdef LEXDEBUG

        cout << "letter: " << yylval->value << endl;

#endif

        BEGIN INITIAL;

        return LETTER;

    }

    else if(yytext[0] == '\'' && len == 1){ //正确识别字符

        yylval = new Token;

        yylval -> value = charTemp;

        yylval -> type = "LETTER";

        yylval -> lineNo = yylineno;

#ifdef LEXDEBUG

        cout << "letter: " << yylval->value << endl;

#endif

        BEGIN INITIAL;

        return LETTER;

    }

    else if(yytext[0] == '\''){ //引号之间有多个字符，错误处理为第一个字符

        addLexicalErrorInformation(yytext,"Too many characters!");

        yylval = new Token;

        yylval -> value = charTemp[0];

        yylval -> type = "LETTER";

        yylval -> lineNo = yylineno;

#ifdef LEXDEBUG

        cout << "letter: " << yylval->value << endl;

#endif

        BEGIN INITIAL;

        return LETTER;

    }

    else if(yytext[0] == '\n'){ //缺失单引号

        yyless(0);

        addLexicalErrorInformation(yytext,"Right quote missing!");

        yylval = new Token;

        if(len == 0){

            yylval -> value = "\0";

        }

        else{

            yylval -> value = charTemp[0];

        }

        yylval -> type = "LETTER";

        yylval -> lineNo = yylineno;

#ifdef LEXDEBUG

        cout << "letter: " << yylval->value << endl;

#endif

        BEGIN INITIAL;

        return LETTER;

    }

}

### 3、对注释进行词法分析

在对注释进行语义分析时会比较复杂。因为注释涉及到单行注释和多行注释，我们需要分别处理，创建SCMT和MCMT两个状态，在识别到注释开头的{和(\*的情况时，进行状态跳转。

"{" {

    BEGIN SCMT;

}

"(\*" {

    BEGIN MCMT;

}

如果正确识别情况下，该注释会识别到其注释结尾符}和\*)，转回INITIAL状态。

<SCMT>"}" {

    BEGIN INITIAL;

}

<MCMT>"\*)" {

    BEGIN INITIAL;

}

如果单行注释遇到\n则会做错误处理，如果注释时遇到文件结束符也会做错误处理。

<SCMT>"}" {

    BEGIN INITIAL;

}

<MCMT>"\*)" {

    BEGIN INITIAL;

}

<SCMT><<EOF>> {

    addLexicalErrorInformation(yytext, "Unexpected EOF before a singleline comment ends");

    return 0;

}

<SCMT>"\n" {

    yyless(0);

    addLexicalErrorInformation(yytext, "Unexpected '\\n' before a singleline comment ends");

    BEGIN INITIAL;

}

<SCMT>. {/\*do nothing\*/}

<MCMT><<EOF>> {

    addLexicalErrorInformation(yytext, "Unexpected EOF before a multiline comment ends");

    return 0;

}

<MCMT>{line} {

    // yyless(0);

}

<MCMT>. {}