《编译原理》课程实验报告一

——一**个简易词法分析器的实现**

**殷迪 131250021**

**Motivation/Aim**

本实验的目的在于通过实现一个简单的词法分析器，进一步掌握和理解课堂所讲解的有关词法分析的相关理论，增强程序设计能力，从而为进一步学习《编译原理》课程的语法分析部分作准备。

**Content description**

利用Java语言实现一个C语言子集词法的词法分析器，程序的输入为一个待进行语法分析的C语言源程序，程序的输出为词法分析过程后的Token序列，用ArrayList<Token>返回，为进一步进行词法分析提供数据源。

**Ideas/Methods**

本词法分析器通过以下流程进行词法分析的具体过程，从而得出最后的词法分析结果：

1. 通过一个Scan类获取输入的C语言源程序文件，解析过程中滤掉所有的空格和注释。
2. 手工构建一个DFA，构建的程序写在LexicalAnalyzer类里面，通过简单的循环获取一个个的Token序列并返回。

**Assumptions**

本程序进行一个C语言子集的词法分析，所输入的C语言程序满足以下要求：

1. 包含C语言的保留字，不超过以下范围

**auto,double,int,struct,break,else,long,switch,case,enum,register,typedef,char,return,union,const,extern,float,short,unsigned,continue,for,signed,void,default,goto,sizeof,volatile,do,if,static,while**

1. 包含C语言的基本符号，不超过以下范围

**= == <= >= < > | || + ++ - -- ! != , ； { } [ ] ( ) #**

**+= -= \*= /=**

1. 其余均为变量或者数字

**Related FA descriptions**

自动机比较大，不方便计算机作图，此处给出所有状态的定义，源代码写在Type类中

**public** **static** **final** **int** ***AUTO*** = 0;

**public** **static** **final** **int** ***DOUBLE*** = 1;

**public** **static** **final** **int** ***INT*** = 2;

**public** **static** **final** **int** ***STRUCT*** = 3;

**public** **static** **final** **int** ***BREAK*** = 4;

**public** **static** **final** **int** ***ELSE*** = 5;

**public** **static** **final** **int** ***LONG*** = 6;

**public** **static** **final** **int** ***SWITCH*** = 7;

**public** **static** **final** **int** ***CASE*** = 8;

**public** **static** **final** **int** ***ENUM*** = 9;

**public** **static** **final** **int** ***REGISTER*** = 10;

**public** **static** **final** **int** ***TYPEDEF*** = 11;

**public** **static** **final** **int** ***CHAR*** = 12;

**public** **static** **final** **int** ***EXTERN*** = 13;

**public** **static** **final** **int** ***RETURN*** = 14;

**public** **static** **final** **int** ***UNION*** = 15;

**public** **static** **final** **int** ***CONST*** = 16;

**public** **static** **final** **int** ***FLOAT*** = 17;

**public** **static** **final** **int** ***SHORT*** = 18;

**public** **static** **final** **int** ***UNSIGNED*** = 19;

**public** **static** **final** **int** ***CONTINUE*** = 20;

**public** **static** **final** **int** ***FOR*** = 21;

**public** **static** **final** **int** ***SIGNED*** = 22;

**public** **static** **final** **int** ***VOID*** = 23;

**public** **static** **final** **int** ***DEFAULT*** = 24;

**public** **static** **final** **int** ***GOTO*** = 25;

**public** **static** **final** **int** ***SIZEOF*** = 26;

**public** **static** **final** **int** ***VOLATILE*** = 27;

**public** **static** **final** **int** ***DO*** = 28;

**public** **static** **final** **int** ***IF*** = 29;

**public** **static** **final** **int** ***STATIC*** = 30;

**public** **static** **final** **int** ***WHILE*** = 31;

/\*运算符和界符\*/

**public** **static** **final** **int** ***ASSIGN*** = 32;

**public** **static** **final** **int** ***ADD*** = 33;

**public** **static** **final** **int** ***SUB*** = 34;

**public** **static** **final** **int** ***DIV*** = 35;

**public** **static** **final** **int** ***LT*** = 36;//<

**public** **static** **final** **int** ***LE*** = 37;//<=

**public** **static** **final** **int** ***GT*** = 38;//>

**public** **static** **final** **int** ***GE*** = 39;//>=

**public** **static** **final** **int** ***NE*** = 40;//!=

**public** **static** **final** **int** ***EQUAL*** = 41;//==

**public** **static** **final** **int** ***OR\_1*** = 42;//|

**public** **static** **final** **int** ***OR\_2*** = 43;//||

**public** **static** **final** **int** ***AND\_1*** = 44;//=

**public** **static** **final** **int** ***AND\_2*** = 45;//==

**public** **static** **final** **int** ***NOT*** = 46;//!

**public** **static** **final** **int** ***XOR*** = 47;

**public** **static** **final** **int** ***INCREASE*** = 48;//++

**public** **static** **final** **int** ***DECREASE*** = 49;//--

**public** **static** **final** **int** ***COMMA*** = 50;//,

**public** **static** **final** **int** ***SEMICOLON*** = 51;//;

**public** **static** **final** **int** ***BRACE\_L*** = 52;//{

**public** **static** **final** **int** ***BRACE\_R*** = 53;//}

**public** **static** **final** **int** ***BRACKET\_L*** = 54;//[

**public** **static** **final** **int** ***BRACKET\_R*** = 55;//]

**public** **static** **final** **int** ***PARENTHESIS\_L*** = 56;//(

**public** **static** **final** **int** ***PARENTHESIS\_R*** = 57;//)

**public** **static** **final** **int** ***POUND*** = 58;//#

/\*标识符和数字\*/

**public** **static** **final** **int** ***ID*** = 59;

**public** **static** **final** **int** ***NUM*** = 60;

/\*补充\*/

**public** **static** **final** **int** ***INCREASEBY*** = 61;//+=

**public** **static** **final** **int** ***DECREASEBY*** = 62;//-=

**public** **static** **final** **int** ***MULBY*** = 63;//\*=

**public** **static** **final** **int** ***DIVBY*** = 64;///=

**public** **static** **final** **int** ***MUL*** = 65;//\*

**public** **static** **final** **int** ***INCLUDE*** = 66;

**public** **static** **final** **int** ***SINGLE\_QUOTAOTION*** = 67;

**public** **static** **final** **int** ***DOUBLE\_QUOTATION*** = 68;

**public** **static** **final** **int** ***TRANSFER*** = 69;

**Description of important Data Structures**

用于输出的Token类

**public** **class** Token {

**public** **int** type;

**public** String value;

**public** Token(**int** type,String value){

**this**.type = type;

**this**.value = value;

}

**public** String toString(){

**return** "<"+**this**.type+","+**this**.value+">";

}

}

用于获取输入源程序的Scan类的属性和接口（限于篇幅，本报告中略去所有的方法定义）

**public** **class** Scan {

**private** **static** String *inputPath* = "Input/";

**public** String input;//从文件中读取的略去所有空格和注释的输入

**public** **int** pointer;//指向当前读取的位置

**public** Scan(String filename);

**public** **char** getNextChar();

**public** **void** retract(**int** n);//指针回退n位

**public** **int** getIndex();

**public** **int** getLength();

**public** String getSubStr(**int** index,**int** length);

**public** String getTestString(**int** index);

**public** String getLeftStr(**int** index);

**public** **void** move(**int** n);

**public** String getStringInQuotation(**int** index);

}

**Description of core Algorithms**

**在Scan类的构造方法中实现读取源文件并滤掉所有的空白和注释**

**public** Scan(String filename){

File sourceFile = **new** File(Scan.*inputPath*+filename);

ArrayList<Character> trans = **new** ArrayList<Character>();

**try** {

FileInputStream in = **new** FileInputStream(sourceFile);

**char** ch1 = ' ';

**char** ch2 = ' ';//用于在验证是否为引号内结尾或者注释结尾

**while**(in.available()>0){

**if**(ch2 != ' ')

ch1 = ch2;

**else**

ch1 = (**char**) in.read();

**if**(ch1 == '\''){//避免删除空白时将‘’包含的空白字符剔除

trans.add(ch1);

trans.add((**char**)in.read());

trans.add((**char**)in.read());

} **else** **if** (ch1 == '\"'){//避免将字符串中的空白剔除

trans.add(ch1);

**while**(in.available()>0){

ch1 = (**char**)in.read();

trans.add(ch1);

**if**(ch1 == '\"')

**break**;

}

} **else** **if** (ch1 == '/'){//剔除字符串

ch2 = (**char**)in.read();

**if**(ch2 == '/'){

**while**(in.available() > 0){

ch2 = (**char**)in.read();

**if**(ch2 == '\n')

**break**;

}

ch2 = ' ';

} **else** **if** (ch2 == '\*') {

**while**(in.available() > 0){

ch1 = (**char**)in.read();

**if**(ch1 == '\*'){

ch2 = (**char**)in.read();

**if**(ch2 == '/')

**break**;

}

}

} **else** {

**if**(ch2 == ' '){

**while**(ch2 == ' '){

ch2 = (**char**)in.read();

}

}

trans.add(ch1);

trans.add(ch2);

ch2 = ' ';

}

} **else** **if**(ch1 == ' '){

**if**(trans.get(trans.size()-1) == ' '){

**continue**;

} **else** {

//trans.add(' ');

}

} **else** {

**if**((**int**)ch1 == 13 ||(**int**)ch1 == 10 ||(**int**)ch1 == 32){//去除换行

} **else** {

trans.add(ch1);

}

}

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**char**[] chStr = **new** **char**[trans.size()];

**for**(**int** i = 0;i < trans.size();i++){

chStr[i] = trans.get(i);

}

String result = **new** String(chStr);

**this**.input = result;

**this**.pointer = 0;

}

**通过自动机进行词法分析**

我的主要思路是利用一个while循环不断读取数据来进行分析，过程中对数字、变量、符号、运算符进行不同的分析，整体结构没什么特别大的难度。后面调试过程中修正所花的时间主要都是在==、>=、<=等由两个基本运算符字符的读取上，这里用到了前面Scan类里定义的retract方法，可以多读一位判断后进行回退操作并且跳出相应的while循环，返回token值。如果匹配，则直接返回一个token来跳出这个循环。另一个注意点是“”中的字符串需要单独作为一个token，如果有空格的话不能拆开，这里就需要在处理引号时进行一个配对处理。

**public** Token analyze(**int** index){

**int** length = scan.getLength();

**int** type = -1;

String value = "";

**while**(index < length){

**char** ch = scan.getNextChar();

//System.out.println(ch);

index++;

**char** ch1 = '\0';

**if**(isDigit(ch)){//判断是否为一个数字

**if**(Type.*isCalc*(type)){

scan.retract(1);

**break**;

}

**if**(value == ""){

value = **new** Character(ch).toString();

type = Type.***NUM***;

} **else** {

value += **new** Character(ch).toString();

}

} **else** **if** (isLetter(ch)){

**if**(Type.*isCalc*(type)){

scan.retract(1);

**break**;

}

**if**(flag){

value = scan.getStringInQuotation(index);

type = Type.***ID***;

scan.move(value.length()-1);

**return** **new** Token(type,value);

}

**if**(type == Type.***ID***){

value += **new** Character(ch).toString();

**continue**;

}

String str = scan.getTestString(index);

String val = **null**;

**if**(str.startsWith("include")){

val = "include";

type = Type.***INCLUDE***;

} **else** {

**for**(**int** i = 0;i < keyword.length;i++){

**if**(str.startsWith(keyword[i])){

val = keyword[i];

type = i;

**break**;

}

}

}

**if**(val == **null**){

type = Type.***ID***;

**if**(value == ""){

value = **new** Character(ch).toString();

} **else** {

value += **new** Character(ch).toString();

}

} **else** {

value = val;

scan.move(value.length()-1);

**return** **new** Token(type,value);

}

} **else** {

**if**(type == Type.***NUM*** || type == Type.***ID***){

scan.retract(1);

**return** **new** Token(type,value);

}

**switch**(ch){

**case** '='://==,=

**if**(type == -1){

type = Type.***ASSIGN***;

value = "=";

} **else** **if**(type == Type.***LT***){//<=

type = Type.***LE***;

value = "<=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***GT***){//>=

type = Type.***GE***;

value = ">=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***ASSIGN***){//==

type = Type.***EQUAL***;

value = "==";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***NOT***){//!=

type = Type.***NE***;

value = "!=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***ADD***){//+=

type = Type.***INCREASEBY***;

value = "+=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***SUB***){//-=

type = Type.***DECREASEBY***;

value = "-=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***DIV***){///=

type = Type.***DIVBY***;

value = "/=";

**return** **new** Token(type,value);

} **else** **if**(type == Type.***MUL***){//\*=

type = Type.***MULBY***;

value = "\*=";

**return** **new** Token(type,value);

}

**break**;

**case** '+':

**if**(type == -1){

type = Type.***ADD***;

value = "+";

} **else** **if**(type == Type.***ADD***){//++

type = Type.***INCREASE***;

value = "++";

**return** **new** Token(type,value);

}

**break**;

**case** '-':

**if**(type == -1){

type = Type.***SUB***;

value = "-";

} **else** **if**(type == Type.***SUB***){//--

type = Type.***DECREASEBY***;

value = "--";

**return** **new** Token(type,value);

}

**break**;

**case** '\*':

**if**(type == -1){

type = Type.***MUL***;

value = "\*";

}

**break**;

**case** '/':

**if**(type == -1){

type = Type.***DIV***;

value = "/";

}

**break**;

**case** '<':

**if**(type == -1){

type = Type.***LT***;

value = "<";

}

**break**;

**case** '>':

**if**(type == -1){

type = Type.***GT***;

value = ">";

}

**break**;

**case** '!':

**if**(type == -1){

type = Type.***NOT***;

value = "!";

}

**break**;

**case** '|':

**if**(type == -1){

type = Type.***OR\_1***;

value = "|";

} **else** **if**(type == Type.***OR\_1***){

type = Type.***OR\_2***;

value = "||";

**return** **new** Token(type,value);

}

**break**;

**case** '&':

**if**(type == -1){

type = Type.***AND\_1***;

value = "&";

} **else** **if**(type == Type.***AND\_1***){

type = Type.***AND\_2***;

value = "&&";

**return** **new** Token(type,value);

}

**break**;

**case** ';':

**if**(type == -1){

type = Type.***SEMICOLON***;

value = ";";

}

**break**;

**case** '{':

**if**(type == -1){

type = Type.***BRACE\_L***;

value = "{";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** '}':

**if**(type == -1){

type = Type.***BRACE\_R***;

value = "}";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** '[':

**if**(type == -1){

type = Type.***BRACKET\_L***;

value = "[";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** ']':

**if**(type == -1){

type = Type.***BRACKET\_R***;

value = "]";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** '(':

**if**(type == -1){

type = Type.***PARENTHESIS\_L***;

value = "(";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** ')':

**if**(type == -1){

type = Type.***PARENTHESIS\_R***;

value = ")";

} **else** **if**(Type.*isCalc*(type)){

scan.retract(1);

**return** **new** Token(type,value);

}

**break**;

**case** '#':

**if**(type == -1){

type = Type.***POUND***;

value = "#";

}

**break**;

**case** ',':

**if**(type == -1){

type = Type.***COMMA***;

value = ",";

}

**break**;

**case** '\'':

**if**(type == -1){

type = Type.***SINGLE\_QUOTAOTION***;

value = "\'";

}

**break**;

**case** '"':

**if**(flag == **false**){

flag = **true**;//表明这是配对的双引号中的第一个

} **else** {

flag = **false**;

}

**if**(type == -1){

type = Type.***DOUBLE\_QUOTATION***;

value = "\"";

}

**break**;

**default**:

**break**;

}

**if**(!Type.*isCalc*(type)){

**break**;

}

}

}

**if**(value.length()>1){

scan.move(value.length()-1);

}

Token token = **new** Token(type,value);

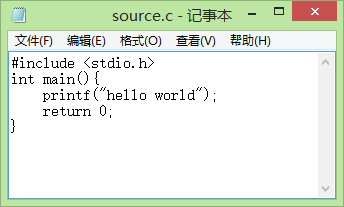
**return** token;

}

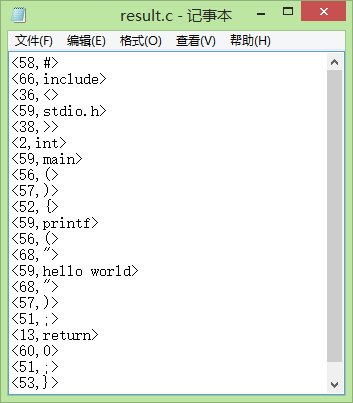
**Use cases on running**

第一个测试用例是大家所熟悉的hello world，源文件存在input里面，输出结果在output里，可以调用LexicalAnalyzer类里面的main方法进行运行测试。

**输入程序source.c**



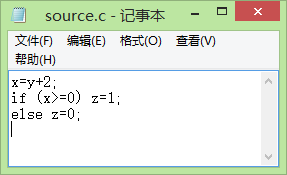
**输出程序result.txt**



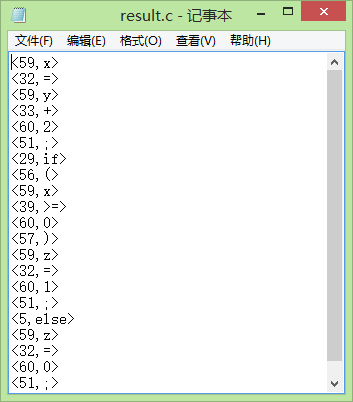
Nice！词法分析器可以分析hello world了，而且可以识别字符串中的内容并且把它单独独立出来了！

下面我们再来分析以下翟老师在群里给的那个语法分析的例子中的程序，正确的词法分析结果也是后面进行正确的语法分析的基础。

**输入程序source.c**



**输出程序result.txt**



仔细检查了一下，并没有什么问题，可以安心地做下一次实验而不用担心前面的错导致后面的一连串bug了！

**Problems occurred and related solutions**

1. 对于引号中的字符串包含空白字符导致读取时被滤掉，对应的解决方案是在Scan中进行配对，然后读取并存储。
2. 对于由两个字符组成的运算符，需要一个while循环进行状态匹配。

PS：由于我为了用更多的时间做好后面的语法分析器，这里的词法分析器就简单一点的，遇到的问题也不是什么大问题。

**Your feelings and comments**

此次实验，通过一个实际的词法分析器的实现，增强了程序设计能力，考虑了编译器的词法分析器部分进行构建时需要考虑的很多细节，加深了对编译原理中词法分析相关理论以及有限自动机理论的理解，为进一步学习语法分析等编译理论奠定了扎实的基础。