程序的执行从类 Main 的方法 main 开始。方法 main 创建了一个词法分析器，然后调用词法分析器的 scan 方法，
逐个字符分析，每当出现新的词素时，便返回到 main 函数，并把它输出到终端。

```
package main;
import java.io.IOException;import lexer.Lexer;import lexer.Token;
public class Main {
    public static void main(String[] args) throws IOException {
        Lexer lexer = new Lexer();
        do {
            Token token=lexer.scan();
            switch (token.tag) {
            case 270:
            case 272:
                System.out.println("(NUM , "+token.toString()+")");break;
            case 267:
            case 268:
            case 269:
                System.out.println("(SYM , "+token.toString()+")");break;
            default:
                System.out.println("("+token.tag+" , "+token.toString()+")");break;
            }
        } while (lexer.getPeek()!='n');
    }
}
```

A.3 词法分析器

包 lexer 是 2.6.5 节中的词法分析器的代码的扩展。类 Tag 定义了各个词法单元对应的常量：

```
1) package lexer;                    // 文件 Tag.java
2) public class Tag {
3)    public final static int
4)       AND   = 256,  BASIC = 257,  BREAK = 258,  DO   = 259, ELSE  = 260,
5)       EQ    = 261,  FALSE = 262,  GE    = 263,  ID   = 264, IF    = 265,
6)       INDEX = 266,  LE    = 267,  MINUS = 268,  NE   = 269, NUM   = 270,
7)       OR    = 271,  REAL  = 272,  TEMP  = 273,  TRUE = 274, WHILE = 275;
8) }
```

其中的三个常量 INDEX、MINUS 和 TEMP 不是词法单元，它们将在抽象语法树中使用。

类 Token 和 Num 和 2.6.5 节的相同，但是增加了方法 toString：

```
1) package lexer;                    // 文件 Token.java
2) public class Token {
3)    public final int tag;
4)    public Token(int t) { tag = t; }
5)    public String toString() {return "" + (char)tag;}
6) }
```

```
1) package lexer;                    //文件 Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5)    public String toString() { return "" + value; }
6) }
```

类 Word 用于管理保留字、标识符和像 && 这样的复合词法单元的词素。它也可以用来管理在中间代码中运算符的书写形式；比如单目减号。例如，源文本中的 -2 的中间形式是 minus 2。

```
1) package lexer;                    // 文件 Word.java
2) public class Word extends Token {
3)    public String lexeme = "";
4)    public Word(String s, int tag) { super(tag); lexeme = s; }
5)    public String toString() { return lexeme; }
6)    public static final Word
7)       and = new Word( "&&", Tag.AND ),  or = new Word( "||", Tag.OR ),
8)       eq  = new Word( "==", Tag.EQ  ),  ne = new Word( "!=", Tag.NE ),
9)       le  = new Word( "<=", Tag.LE  ),  ge = new Word( ">=", Tag.GE ),
10)      minus  = new Word( "minus", Tag.MINUS ),
11)      True   = new Word( "true",  Tag.TRUE  ),
12)      False  = new Word( "false", Tag.FALSE ),
13)      temp   = new Word( "t",     Tag.TEMP  );
14) }
```

类 Real 用于处理浮点数：

```
1) package lexer;                    // 文件 Real.java
2) public class Real extends Token {
3)    public final float value;
4)    public Real(float v) { super(Tag.REAL); value = v; }
5)    public String toString() { return "" + value; }
6) }
```

如我们在 2.6.5 节中讨论的，类 Lexer 的主方法，即函数 scan，识别数字、标识符和保留字。

类 Lexer 中的第 9 ~ 13 行保留了选定的关键字。第 14 ~ 16 行保留了在其他地方定义的对象的词素。对象 Word.True 和 Word.False 在类 Word 中定义。对应于基本类型 int、char、bool 和 float 的对象在类 Type 中定义。类 Type 是 Word 的一个子类。类 Type 来自包 symbols。

```
1) package lexer;                    //文件 Lexer.java
2) import java.io.*; import java.util.*; import symbols.*;
3) public class Lexer {
4)    public static int line = 1;
5)    char peek = ' ';
6)    Hashtable words = new Hashtable();
7)    void reserve(Word w) { words.put(w.lexeme, w); }
8)    public Lexer() {
9)       reserve( new Word("if",    Tag.IF)    );
10)      reserve( new Word("else",  Tag.ELSE)  );
11)      reserve( new Word("while", Tag.WHILE) );
12)      reserve( new Word("do",    Tag.DO)    );
13)      reserve( new Word("break", Tag.BREAK) );
14)      reserve( Word.True );  reserve( Word.False );
15)      reserve( Type.Int  );  reserve( Type.Char  );
16)      reserve( Type.Bool );  reserve( Type.Float );
17)   }
```

函数 readch( )(第 18 行)用于把下一个输入字符读到变量 peek 中。名字 readch 被复用或重载，(第 19 ~ 24 行)，以便帮助识别复合的词法单元。比如，一看到输入字符 <，调用 readch("=")就会把下一个字符读入peek，并检查它是否为 =。

```
18)   void readch() throws IOException { peek = (char)System.in.read(); }
19)   boolean readch(char c) throws IOException {
20)      readch();
21)      if( peek != c ) return false;
22)      peek = ' ';
23)      return true;
24)   }
```

函数 scan 一开始首先略过所有的空白字符(第 26 ~ 30 行)。它首先试图识别像 < = 这样的复合词法单元(第 31 ~ 34 行)和像 365 及 3.14 这样的数字(第 45 ~ 58 行)。如果不成功，它就试图读入一个字符串(第 59 ~ 70 行)。

```
25)    public Token scan() throws IOException {
26)       for( ; ; readch() ) {
27)          if( peek == ' ' || peek == '\t' ) continue;
28)          else if( peek == '\n' ) line = line + 1;
29)          else break;
30)       }
31)       switch( peek ) {
32)       case '&':
33)          if( readch('&') ) return Word.and;  else return new Token('&');
34)       case '|':
35)          if( readch('|') ) return Word.or;   else return new Token('|');
36)       case '=':
37)          if( readch('=') ) return Word.eq;   else return new Token('=');
38)       case '!':
39)          if( readch('=') ) return Word.ne;   else return new Token('!');
40)       case '<':
41)          if( readch('=') ) return Word.le;   else return new Token('<');
42)       case '>':
43)          if( readch('=') ) return Word.ge;   else return new Token('>');
44)       }
45)       if( Character.isDigit(peek) ) {
46)          int v = 0;
47)          do {
48)             v = 10*v + Character.digit(peek, 10); readch();
49)          } while( Character.isDigit(peek) );
50)          if( peek != '.' ) return new Num(v);
51)          float x = v; float d = 10;
52)          for(;;) {
53)             readch();
54)             if( ! Character.isDigit(peek) ) break;
55)             x = x + Character.digit(peek, 10) / d; d = d*10;
56)          }
57)          return new Real(x);
58)       }
59)       if( Character.isLetter(peek) ) {
60)          StringBuffer b = new StringBuffer();
61)          do {
62)             b.append(peek); readch();
63)          } while( Character.isLetterOrDigit(peek) );
64)          String s = b.toString();
65)          Word w = (Word)words.get(s);
66)          if( w != null ) return w;
67)          w = new Word(s, Tag.ID);
68)          words.put(s, w);
69)          return w;
70)       }
```

最后，peek 中的任意字符都被作为词法单元返回(第 71 ~ 72 行)。

```
71)       Token tok = new Token(peek); peek = ' ';
72)       return tok;
73)    }
74) }
```