词法分析

Hollow Man

实验原理分析

定义：

词法分析器的功能输入源程序代码，按照构词规则分解成一系列单词符号。单词是语言中具有独立意义的最小单位，包括关键字、标识符、运算符、界符和常量等。

输出：

词法分析器所输出单词符号输出成如下的二元式：

（单词内容，单词种别）

词法分析器作为一个独立子程序：

词法分析是编译过程中的一个阶段，在语法分析前进行。词法分析作为一遍，可以简化设计，改进编译效率，增加编译系统的可移植性。也可以和语法分析结合在一起作为一遍，由语法分析程序调用词法分析程序来获得当前单词供语法分析使用。

算法流程分析

（流程图及相关解释分析）

程序关键部分分析

1. **定义**

本编译器使用 python3 来实现。使用了如下单词种类

(1) 关键字(keywords) 是由程序语言定义的具有固定意义的标识符。

(2) 标识符(identifier) 用来表示各种名字，如变量名，数组名，过程名等等。

(3) 常数(integer) 常数的类型一般有整型，浮点型。

(4) 运算符(operators) 如+、-、*、/等等。

(5) 界符(symbols) 如逗号、分号、括号、等等。

(6) 字符串(string)

(7) 文件结束(EOF)

输出：

输出为元组形式：(单词内容，单词种别)，其中单词种别我们这里采用字符串表示形式，分别为上 7 种种类形式：["Identifier","Integer","Symbol","String","Keyword","Operator","EOF"]

2. **关键部分分析**

我使用嵌套 if...elif...的语句来实现词法分析的有限自动机逻辑。

我创建了一个独立的 python 源文件，名为 lexer.py 因此，如果您想使用该词法分析器，只需将其作为库导入。在该模块中，其包括了 GetNextToken（）和 PeekNextToken（）两个方法。输入的源代码以列表的形式存储在变量 code 中，这样的列表由字符串组成，每个元素是源代码文件中一行的内容。例如，code[line]表示源代码文件中的第 line 行内容字符串，而 code[line][pointer]表示指定行号中的第 pointer 个字符。此模块还有一个变量 line 参数用于存储已使用的行号，还有一个变量 pointer 用于存储在已使用的行中指定的字符列号。

当第一次进入 GetNextToken()方法时，它将处理源文件是空的还是指针或行参数超出实际大小的情况，以避免这些问题。然后它开始去除 tab、空格和注释，直到出现下一个有效单词。如果源代码中使用了/*而不使用*/，lexer 将停止并显示错误消息。最后，我们开始识别这些标记，如果一个字符串在一行中没有以"结束，或者有一个不允许的符号或一个错误的标识符，就会报错。如果成功，函数将首先增加指针，然后返回一个包含单词内容和单词种别的元组。

PeekNextToken()方法首先记录当前行和指针编号，然后执行 GetNextToken（），最后恢复行和指针编号，从而赋予程序预知下一个单词的能力。

主程序为 myc.py，其接受一个文件夹路径参数。当程序运行时，其会遍历该文件夹下所有以.c 为扩展名的文件，并将其结果输出至与件同名的 txt 文件。

3. **结果分析**

然后我使用一个空文件，一个只有空行的文件，一个缺少*/表示注释文件，放置在当前程序目录下的 test 文件夹，随后运行 ./myc.py test 来测试我的程序，程序都给出了空结果并能正常运行并给出提示。随后，我使用了以下程序测试文件：

```
int main(){
  //sdfdsfdfsdsfsdf


    int a;
    int b;
```

```
    char k[3]="a\"bc";


   /*
   sdfsdfdfs
   */


       a = 1;
    a++;
       b = a + 2 ;


       return b;
}
```

程序能够正常输出其运行结果：

```
('int', 'Keyword')
('main', 'Identifier')
('(', 'Symbol')
(')', 'Symbol')
('{', 'Symbol')
('int', 'Keyword')
('a', 'Identifier')
(';', 'Symbol')
('int', 'Keyword')
('b', 'Identifier')
(';', 'Symbol')
('char', 'Identifier')
('k', 'Identifier')
```

('[', 'Symbol')

(3, 'Integer')

(']', 'Symbol')

('=', 'Operator')

('a\\"bc', 'String')

(';', 'Symbol')

('a', 'Identifier')

('=', 'Operator')

(1, 'Integer')

(';', 'Symbol')

('a', 'Identifier')

('++', 'Operator')

(';', 'Symbol')

('b', 'Identifier')

('=', 'Operator')

('a', 'Identifier')

('+', 'Operator')

(2, 'Integer')

(';', 'Symbol')

('return', 'Keyword')

('b', 'Identifier')

(';', 'Symbol')

('}', 'Symbol')

4. **正规集的表示工具和识别工具**

使用 **Python re** 库完成。

实验总结

1. **编译环境**

Fedora 33

2. **语言环境**

Python 3.9

3. 自我分析（遇到的问题，如何解决等）

实验中，一开始我并未设计 PeekNextToken()方法，所以程序无法预知下一个字符是什么，在遇到如”++”这种运算符时有些棘手，随后我封装了这一方法，程序变得清晰起来。

最后，实验过程中还遇到了一些逻辑和细节问题，这些都被我一一解决了，并且还解决了字符串中引号的转义问题，最终形成了这一较完美的程序。