

A dark blue vertical bar on the left side of the slide. A blue arrow points to the right from the bar, containing the date.

2015-11-12

# 编译原理实验

词法分析器的简单实现

## 1. 实验目的

通过手工构造的最小化 DFA 来构建简单的词法分析器，进一步熟悉词法分析的过程以及加深对“RE→NFA→DFA→DFA(o)→Program”这个过程的了解。

## 2. 实验描述

本次实验使用 Java 语言编写，简单实现了对 C 语言程序的词法分析。程序的输入是 program.c 文件，内含一段 C 语言代码，程序的输出是控制台和 output.txt 文件，内容是 Token 序列、符号表和常量表。

该词法分析器的分析能力如下：

1. 忽略注释：单行注释//、多行注释/\* \*/。
2. 关键字(ANSI C定义的32个)："auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register", "return", "short", "signed", "static", "sizeof", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"
3. 字符常量（含转义字符）
4. 字符串常量（含转义字符）
5. 整数常量
6. 浮点数常量（不支持指数）
7. 标识符
8. 操作符：+, -, \*, /, %, =, ++, --, +=, -=, \*=, /=, %=, ==, >, <, >=, <=, >>, <<, &, &&, &=, |, ||, |=, ^, ^=
9. 界符：';', ':', '(', ')', '{', '}', '[', ']', ',', '.', ':'
10. 特殊符号：#, \$

此外，该语法分析器支持的错误识别如下：

1. 不能识别的字符
2. 非法的字符常量
3. 字符常量的长度大于1
4. 非法的转义字符
5. 非法的浮点数常量

Token 序列的格式为<value, type, location>：

关键字：< value , \_KEYWORD, \_ >  
数字：< value , \_NUM , \_ >  
字符：< value , \_CHAR , \_ >  
字符串：< value , \_STRING , \_ >  
标识符：< value , \_ID , location >  
操作符：< value , \_OP , \_ >  
界符：< value , \_DELIMITER , \_ >  
特殊符号：< value , \_SPECIAL, \_ >

### 3. 实验方法

本次实验选用的方法是 PPT 中给出的第一种方法，即手动构造最小化 DFA，然后根据最小化 DFA 编写程序，实现词法分析器。该方法的步骤如下：

1. 定义一些 RE
2. 将 RE 转化为 NFA
3. 将得到的这些 NFA 转化为单一的 NFA
4. 将 NFA 转化为最小化 DFA
5. 根据得到的最小化 DFA 编写程序

### 4. 实验设想

实验设想：词法分析器的能力如实验描述所示。

### 5. 相关的自动机描述

在实验中，我首先根据实验描述中所示的词法分析器的能力完成了 RE 的定义，然后根据实验方法中的步骤最终得到了最小化 DFA，所有的工作都是手工在草稿纸上完成。

### 6. 数据结构的定义

程序中用到的数据结构如下所示：

1. 定义了一个 Token 类，它属性包括 type、value 和 location。其中 location 是指标识符在符号表中的位置，因此只有当 type=" \_ID" 时 location 才是有效的。
2. List<String> symbolTable：符号表。
3. List<Token> tokenList：Token 序列。

### 7. 核心算法描述

程序的核心算法主要集中在 scanner 和 tokenizer 方法中。首先 scanner 扫描程序源文件，然后一个一个字符地依次传递给 tokenizer 方法来获取 Token。在处理一个字符的时候，会遇到两种情况：第一种是该字符是 buffer 的一部分，下次无需再用；第二种情况是该字符只是用来判定之前的情况，下一次仍需要用到。这两种不同的情况在 tokenizer 方法中有不同的实现，代码如下：

```
private void tokenizer(char c) {
    while (true) {
        switch (state) {
            case 0:
                // 第一种情况
                buffer += c;
                state = ?;
                return;
            case 1:
                // 第二种情况
```

```

        ...
        state = ?
        continue;
    }
}
}

```

第一种情况最后是 return，即跳出该方法，获取并处理下一个字符。第二种情况最后是 continue，即继续处理当前的字符直至 return。

程序实现的关键就是不同状态之间的转换，具体见代码。

## 8. 测试用例

C 语言程序源代码：(program.c)

```

#include "stdio.h"
/**
 * 这是多行注释
 */
int main() {    // 这是注释
    int a = 0;
    double b = -1.23;
    a = a + 1;
    if (a > 0 && b < 1) {
        a++;
    } else {
        a /= 5;
    }
    while (a > 0) {
        printf("1234567890");
    }
    switch (a) {
    case 1:
        break;
    default:
        a = 1;
    }
    char xyz = 'a';
    char c = '\t';
    c = '\g';    // 错误
    xyz = 'a123';    // 错误
    b = 12.R;    // 错误
    a = ~`;    // 错误
}

```

输出的结果如下所示：(output.txt)

**Token 序列如下：**

```
< # , _SPECIAL , _ >
< include , _ID , 0 >
< "stdio.h" , _STRING , _ >
< int , _KEYWORD , _ >
< main , _ID , 1 >
< ( , _DELIMITER , _ >
< ) , _DELIMITER , _ >
< { , _DELIMITER , _ >
< int , _KEYWORD , _ >
< a , _ID , 2 >
< = , _OP , _ >
< 0 , _NUM , _ >
< ; , _DELIMITER , _ >
< double , _KEYWORD , _ >
< b , _ID , 3 >
< = , _OP , _ >
< - , _OP , _ >
< 1.23 , _NUM , _ >
< ; , _DELIMITER , _ >
< a , _ID , 2 >
< = , _OP , _ >
< a , _ID , 2 >
< + , _OP , _ >
< 1 , _NUM , _ >
< ; , _DELIMITER , _ >
< if , _KEYWORD , _ >
< ( , _DELIMITER , _ >
< a , _ID , 2 >
< > , _OP , _ >
< 0 , _NUM , _ >
< && , _OP , _ >
< b , _ID , 3 >
< < , _OP , _ >
< 1 , _NUM , _ >
< ) , _DELIMITER , _ >
< { , _DELIMITER , _ >
< a , _ID , 2 >
< ++ , _OP , _ >
< ; , _DELIMITER , _ >
< } , _DELIMITER , _ >
< else , _KEYWORD , _ >
< { , _DELIMITER , _ >
< a , _ID , 2 >
```

```

< /= ,_OP ,_ >
< 5 ,_NUM ,_ >
< ; ,_DELIMITER ,_ >
< } ,_DELIMITER ,_ >
< while ,_KEYWORD ,_ >
< ( ,_DELIMITER ,_ >
< a ,_ID , 2 >
< > ,_OP ,_ >
< 0 ,_NUM ,_ >
< ) ,_DELIMITER ,_ >
< { ,_DELIMITER ,_ >
< printf ,_ID , 4 >
< ( ,_DELIMITER ,_ >
< "1234567890" ,_STRING ,_ >
< ) ,_DELIMITER ,_ >
< ; ,_DELIMITER ,_ >
< } ,_DELIMITER ,_ >
< switch ,_KEYWORD ,_ >
< ( ,_DELIMITER ,_ >
< a ,_ID , 2 >
< ) ,_DELIMITER ,_ >
< { ,_DELIMITER ,_ >
< case ,_KEYWORD ,_ >
< 1 ,_NUM ,_ >
< : ,_DELIMITER ,_ >
< break ,_KEYWORD ,_ >
< ; ,_DELIMITER ,_ >
< default ,_KEYWORD ,_ >
< : ,_DELIMITER ,_ >
< a ,_ID , 2 >
< = ,_OP ,_ >
< 1 ,_NUM ,_ >
< ; ,_DELIMITER ,_ >
< } ,_DELIMITER ,_ >
< char ,_KEYWORD ,_ >
< xyz ,_ID , 5 >
< = ,_OP ,_ >
< 'a' ,_CHAR ,_ >
< ; ,_DELIMITER ,_ >
< char ,_KEYWORD ,_ >
< c ,_ID , 6 >
< = ,_OP ,_ >
< '\t' ,_CHAR ,_ >
< ; ,_DELIMITER ,_ >

```

```

< c, _ID, 6 >
< =, _OP, _ >
< 非法的转义字符, _ERROR, _ >
< ;, _DELIMITER, _ >
< xyz, _ID, 5 >
< =, _OP, _ >
< 字符常量长度大于 1, _ERROR, _ >
< ;, _DELIMITER, _ >
< b, _ID, 3 >
< =, _OP, _ >
< 非法的浮点数常量, _ERROR, _ >
< ;, _DELIMITER, _ >
< a, _ID, 2 >
< =, _OP, _ >
< 不能识别的字符, _ERROR, _ >
< 不能识别的字符, _ERROR, _ >
< ;, _DELIMITER, _ >
< }, _DELIMITER, _ >
< $, _SPECIAL, _ >

```

符号表如下:

```

0   include
1   main
2   a
3   b
4   printf
5   xyz
6   c

```

## 9. 问题和解决方案

在完成实验的过程中,遇到的最大的困难就是对词法分析整个过程不够理解,一开始也不知道具体要做些什么。之后就把龙书的相关内容过了一遍,然后看了几篇关于词法分析器的实现的博文,阅读了 github 上一些别人开源的词法分析程序的代码之后,最终对词法分析的原理和过程有了更深的了解,之后手工构造最小化 DFA 和写程序也就轻松了许多。

## 10. 实验感受和评价

通过实现了一个简单的词法分析器,让我对词法分析有了更深的了解,同时加深了我对编译原理这门课的兴趣,让我感觉到编译原理这门课十分有趣而富有挑战性。对于这次实验,虽然完成了,但仍然存在着很大的不足,实现的只是 C 语言词法分析器的简单版本,很多情况还没有考虑,比如指数类型的浮点数,对#include <stdio.h>的不支持等等。由于时间比较紧,所以只能完成这种程度,以后有时间我会来完善甚至重写词法分析器。