2015-11-12

编译原理实验

词法分析器的简单实现

南京大学软件学院 王振聪 131250218

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分析如下：

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 序列的格式如下：

关键字：　< 值，关键字 >
常量：　< 值，××常量，在常量表中的位置 >
标识符：　< 值，标识符，在符号表的位置 >
操作符：　< 值，操作符 >
界符：　< 值，界符 >
特殊符号：　< 值，特殊符号 >

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

鉴于本次实验的要求是实现简单的词法分析器，并没有涉及后续的语法分析和语义分析，所以没有采用高级的数据结构来存储 Token 序列、符号表和常量表。程序中用到的数据结构如下所示：

1. StringBuilder result：用于存储 Token 序列，便于最后的控制台输出和文件写入。
2. List<String> symbolTable：用于存储符号表。
3. List<String> constTable：用于存储常量表。

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 序列。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：
< # , 特殊符号 >
```

< include , 标识符 , 0 >
< "stdio.h" , 字符串常量 , 0 >
< int , 关键字 >
< main , 标识符 , 1 >
< (, 界符 >
<) , 界符 >
< { , 界符 >
< int , 关键字 >
< a , 标识符 , 2 >
< = , 操作符 >
< 0 , 整数常量 , 1 >
< ; , 界符 >
< double , 关键字 >
< b , 标识符 , 3 >
< = , 操作符 >
< - , 操作符 >
< 1.23 , 浮点数常量 , 2 >
< ; , 界符 >
< a , 标识符 , 2 >
< = , 操作符 >
< a , 标识符 , 2 >
< + , 操作符 >
< 1 , 整数常量 , 3 >
< ; , 界符 >
< if , 关键字 >
< (, 界符 >
< a , 标识符 , 2 >
< > , 操作符 >
< 0 , 整数常量 , 1 >
< && , 操作符 >
< b , 标识符 , 3 >
< < , 操作符 >
< 1 , 整数常量 , 3 >
<) , 界符 >
< { , 界符 >
< a , 标识符 , 2 >
< ++ , 操作符 >
< ; , 界符 >
< } , 界符 >
< else , 关键字 >
< { , 界符 >
< a , 标识符 , 2 >
< /= , 操作符 >
< 5 , 整数常量 , 4 >

< ; , 界符 >
< } , 界符 >
< while , 关键字 >
< (, 界符 >
< a , 标识符 , 2 >
< > , 操作符 >
< 0 , 整数常量 , 1 >
<) , 界符 >
< { , 界符 >
< printf , 标识符 , 4 >
< (, 界符 >
< "1234567890" , 字符串常量 , 5 >
<) , 界符 >
< ; , 界符 >
< } , 界符 >
< switch , 关键字 >
< (, 界符 >
< a , 标识符 , 2 >
<) , 界符 >
< { , 界符 >
< case , 关键字 >
< 1 , 整数常量 , 3 >
< : , 界符 >
< break , 关键字 >
< ; , 界符 >
< default , 关键字 >
< : , 界符 >
< a , 标识符 , 2 >
< = , 操作符 >
< 1 , 整数常量 , 3 >
< ; , 界符 >
< } , 界符 >
< char , 关键字 >
< xyz , 标识符 , 5 >
< = , 操作符 >
< 'a' , 字符常量 , 6 >
< ; , 界符 >
< char , 关键字 >
< c , 标识符 , 6 >
< = , 操作符 >
< '\t' , 字符常量 , 7 >
< ; , 界符 >
< c , 标识符 , 6 >
< = , 操作符 >

```
Error: 非法的转义字符
< ; , 界符 >
< xyz , 标识符 , 5 >
< = , 操作符 >
Error: 字符常量长度大于 1
< ; , 界符 >
< b , 标识符 , 3 >
< = , 操作符 >
Error: 非法的浮点数常量
< ; , 界符 >
< a , 标识符 , 2 >
< = , 操作符 >
Error: 不能识别的字符
Error: 不能识别的字符
< ; , 界符 >
< } , 界符 >
< $ , 特殊符号 >
```

符号表如下:

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

常量表如下:

```
0    "stdio.h"
1    0
2    1.23
3    1
4    5
5    "1234567890"
6    'a'
7    '\t'
```

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。