编译原理课程实验报告

实验 1：词法分析

姓名	王少博	院系	计算机科学与技术学院	学号	181110315
任课教师	韩希先		指导教师	韩希先	
实验地点	研究院中 507		实验时间	2020.10.29	
实验课表现	出勤、表现得分		实验报告得分		实验总分	
	操作结果得分					

一、实验目的

要求：需分析本次实验的基本目的，并综述你是如何实现这些目的的？

基本目的：

1. **巩固对词法分析的基本功能以及原理的认识**
   词法分析器的功能是读入表示源程序的字符流，按照程序功能等价的要求，将其转换为对应的单词序列，并且剔除空格、注解等不影响程序语义的字符。
2. **掌握并运用自动机的知识进行词法分析**
   通过词法规则的描述，画出相关的有穷状态自动机，然后根据状态转换图写出状态转移函数，最后逐个字符读出预处理后的源代码。
3. **理解并处理词法分析中的异常和错误**
   根据读取结果转换状态，输出识别的单词结果或者错误，从而完成词法分析。具体来说，对于词法分析过程中出现的例如：非法字符、单词拼写错误、注解或字符常数不封闭、变量重复说明等错误，我们通常采用非法字符检查、关键字拼写错误检查、不封闭错误检查、重复说明检查、以及紧急恢复的方式来进行错误的发现与处理。

三次实验所有代码已部署于 github，https://github.com/HITWH18SE/Compiler-Principles

二、实验内容

(1) 语言的词法规则描述

1. 我们采用十进制计数。描述文法产生式如下。

<字母> → A | B | … | Z | a | b | … | z

<数字> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<标识符> → <字母> (<字母> | <数字>)*

<关键字> →auto | break | case | char | const | continue | default | do | double | else | float | for | goto | if | int | long | register | return | short | signed | sizeof | static | struct | switch | typedef | unsigned | void | volatile | while

<整常数> → <数字> (<数字>)*

<字符常数> → “(<字母> | <数字>)*”

<浮点常数> → <数字> (<数字>)* . <数字> (<数字>)

<算术运算符> → + | - | * | /

<逻辑运算符> → < | <= | > | >= | == | !=

<单界符> → + | - | * | / | = | > | < | ! | (|) | [|] | { | } | . | , | ; | & | ~

<双界符> → | != | >= | <= | == | || | &&

<界符> → <单界符> | <双界符>

<注释> → /* (<字母> | <数字> | <界符>)* */ | // (<字母> | <数字> | <界符>)* //

2. 对于一般的类别，我们采用种别码和属性值标记单词，即一符一类。但是对于含有多个单词的类别，如标识符，我们采用其值作为属性值的标记方法。为了便于编写程序，我们把关键字作为保留字。对于非法字符，我们定义的种别码为-1.

表 1 种别码-属性值表

单词类型	种别码	属性值
标识符	1	标识符值
常数	2	常数值
算数运算符	3	
关系运算符	4	
界符	5	
保留字	6	
非法字符	-1	

3. 为了便于实现，我们把赋值符号”=”作为关系运算符。

(2) 针对这种单词的状态转换图和程序框图

- 状态转换图：其中双界符的第一个字符我们记为**双界符头**，第二个字符记为**双界符**。

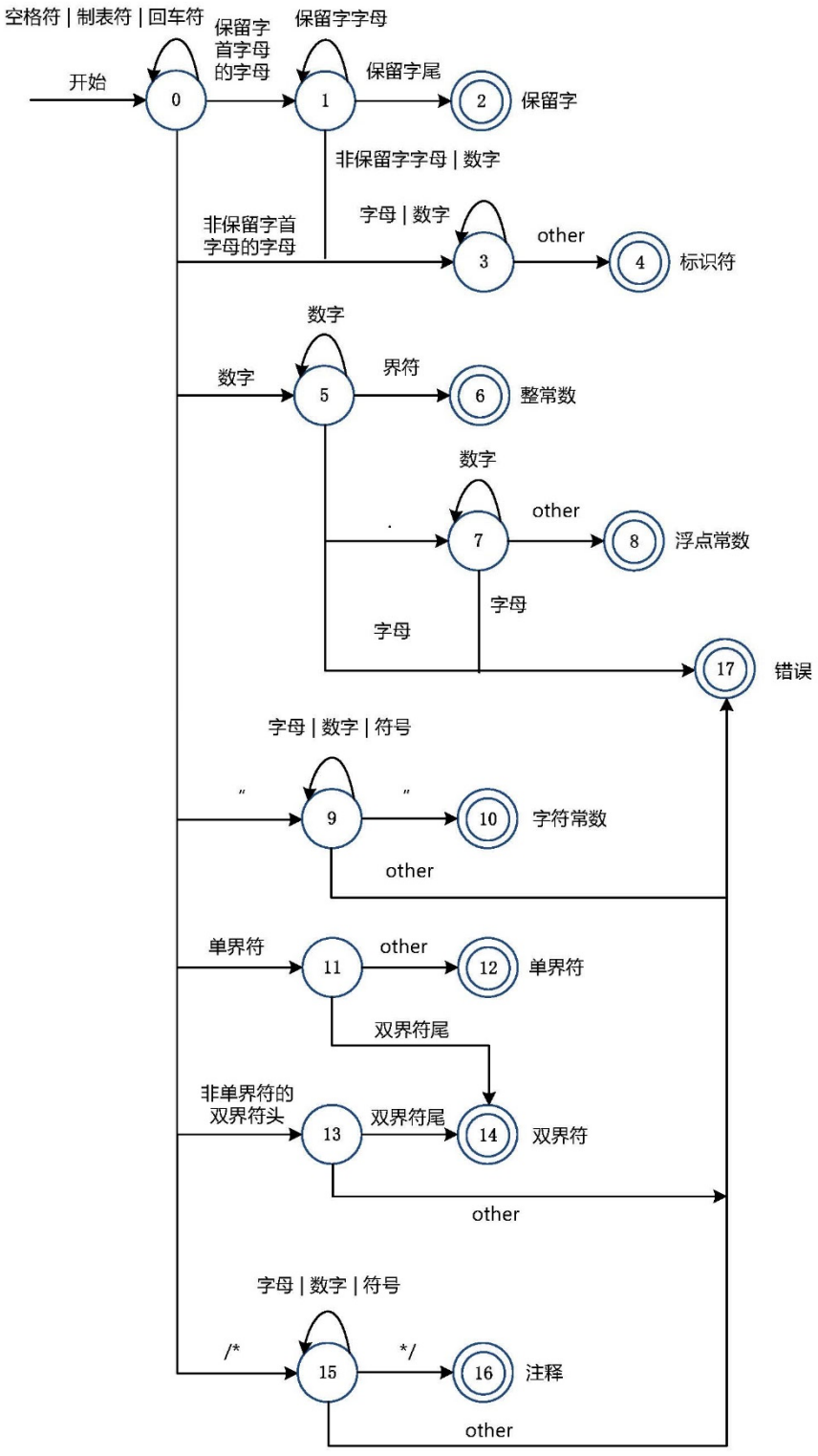
空格符 | 制表符 | 回车符
保留字首字母的字母
保留字字母
开始
0
1
保留字尾
2
保留字
非保留字字母 | 数字
字母 | 数字
非保留字首字母的字母
other
3
4
标识符
数字
数字
界符
5
6
整常数
数字
other
.
7
8
浮点常数
字母
字母
17
错误
字母 | 数字 | 符号
"
"
9
10
字符常数
other
单界符
11
other
12
单界符
双界符尾
非单界符的双界符头
13
双界符尾
14
双界符
other
字母 | 数字 | 符号
/*
*/
15
16
注释
other

图 1 状态转换图

- 程序框图：

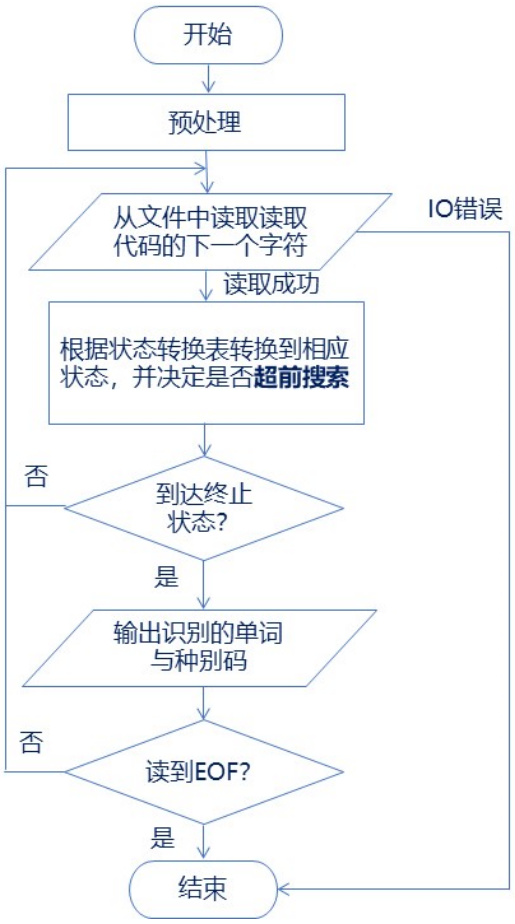

图 2 程序框图

(3) 核心数据结构的设计

- 符号表：如前所述，我们对于标识符和常数使用种别码和属性值的方法进行定义，对于其他类别采用一类一码的形式存储。我们假设标识符表和常数表中元素的个数不超过 1000，且每个不超过 40 个字节。关键字：词法规在 C 语言基础上进行了一定程度的精简，只保留基本的分支与循环结构、整数和浮点数对应的类型以及布尔型。这里的关键字作为了**保留字**。

  <关键字> → auto | break | case | char | const | continue | default | do | double | else | float | for | goto | if | int | long | register | return | short | signed| sizeof | static | struct | switch | typedef | unsigned | void | volatile | while

```
//标识符表
char IdentifierTable[1000][40] = {};

//常数表
char DigitBTable[1000][40] = {};

int Inum = 0;

int Dnum = 0;
```

图 3 标识符表、常数表的定义

```
//保留字表
static char ReserveWord[32][20] = {
        "auto", "break", "case", "char", "const", "continue",
        "default", "do", "double", "else", "enum", "extern",
        "float", "for", "goto", "if", "int", "long",
        "register", "return", "short", "signed", "sizeof", "static",
        "struct", "switch", "typedef", "union", "unsigned", "void",
        "volatile", "while"
};

//算术运算符表
static char ArithmeticOperator[4][4] = {
        "+", "-", "*", "/"
};
//逻辑运算符表
static char RelationalOperator[7][4] = {
        "<", "<=", ">", ">=", "=", "==", "!="
};
//界符表(12)
static char Boundary[36][4] = {
        ";", "(", ")", "^", ",", "#", "%", "[", "]", "{", "}", "."
};
```

图 4 保留字、算数运算符、逻辑运算符、界符表的定义

(4) 错误处理

扫描源代码并根据词法规则识别单词的过程中，如果发现有一串字符无法识别成任何单词或者不满足词法规则，则也将其存储在一个 token 中，种类记为错误，种别码为-1。但此时并不中止扫描，而是继续从下一个字符开始识别，直到源代码全部识别完毕。

三、实验结果

(1) 测试样例 1：

```
int main(void) {
  int min=2, mid=3, max=1, t;
  if (min > mid) {
    t = min;
    min = mid;
    mid = t;
  }
  if (min > max) {
    t = min;
    min = max;
    max = t;
  }
  if (mid > max) {
    t = mid;
    mid = max;
    max = t;
  }
}
```

图 5 测试样例 1

测试样例 1 为词法规则完全正确的一个函数，其词法分析的预期输出结果中也不应该存在种类为错误的 token。**为了方便查看，我在种别码之后加了这个种别码的含义作为参考。**

词法分析结果如下：

```
E:\programming\C++\lab1111\c
(int,6 ReservedWord)
(main,1 Identifier)
((,5 Boundary)
(void,6 ReservedWord)
(),5 Boundary)
({,5 Boundary)
(int,6 ReservedWord)
(min,2 Identifier)
(=,3 ArithmeticOperator)
(2,1 Digit)
(,,5 Boundary)
(mid,3 Identifier)
(=,3 ArithmeticOperator)
(3,2 Digit)
(,,5 Boundary)
(max,4 Identifier)
(=,3 ArithmeticOperator)
(1,3 Digit)
(,,5 Boundary)
(t,5 Identifier)
(;,5 Boundary)
(if,6 ReservedWord)
((,5 Boundary)
(min,2 Identifier)
(>,3 ArithmeticOperator)
(mid,3 Identifier)
(),5 Boundary)
({,5 Boundary)
(t,5 Identifier)
(=,3 ArithmeticOperator)
(min,2 Identifier)
(;,5 Boundary)
(min,2 Identifier)
(=,3 ArithmeticOperator)
(mid,3 Identifier)
(;,5 Boundary)
(mid,3 Identifier)
(=,3 ArithmeticOperator)
(t,5 Identifier)
(;,5 Boundary)
(},5 Boundary)
(if,6 ReservedWord)
((,5 Boundary)
(min,2 Identifier)
(>,3 ArithmeticOperator)
(max,4 Identifier)
(),5 Boundary)
({,5 Boundary)
(t,5 Identifier)
(=,3 ArithmeticOperator)
(min,2 Identifier)
(;,5 Boundary)
(min,2 Identifier)
(=,3 ArithmeticOperator)
(max,4 Identifier)
(;,5 Boundary)
(max,4 Identifier)
(=,3 ArithmeticOperator)
(t,5 Identifier)
(;,5 Boundary)
(},5 Boundary)
(if,6 ReservedWord)
((,5 Boundary)
(mid,3 Identifier)
(>,3 ArithmeticOperator)
(max,4 Identifier)
(),5 Boundary)
({,5 Boundary)
(t,5 Identifier)
(=,3 ArithmeticOperator)
(mid,3 Identifier)
(;,5 Boundary)
(mid,3 Identifier)
(=,3 ArithmeticOperator)
(max,4 Identifier)
(;,5 Boundary)
(max,4 Identifier)
(=,3 ArithmeticOperator)
(t,5 Identifier)
(;,5 Boundary)
(},5 Boundary)
(},5 Boundary)
```

图 6 测试样例 1 词法分析结果

如图所示为测试样例 1 的词法分析结果，可以看到 token 序列中不存在种类为错误的字符串，而且识别出的单词都是按照源代码中的出现顺序排列，符合预期的输出结果。

(2) 测试样例 2：

```
double test(double x){
    int sb;
    x = x @ x;
}
```

图 7 测试样例 2

如图，该测试样例中特殊符号@不是运算符，其词法分析的预期输出结果中应该有错误提示，并且能够指出相应错误的位置。

词法分析结果：

```
(double,6 ReservedWord)
(test,1 Identifier)
((,5 Boundary)
(double,6 ReservedWord)
(x,2 Identifier)
(),5 Boundary)
({,5 Boundary)
(int,6 ReservedWord)
(sb,3 Identifier)
(;,5 Boundary)
(x,2 Identifier)
(=,3 ArithmeticOperator)
(x,2 Identifier)
Illegal character   line 3 col 11  @
(x,2 Identifier)
(;,5 Boundary)
(},5 Boundary)
```

图 8 测试样例 2 词法分析结果

错误的运算符是@，符合预期的输出结果。

(3) 测试样例 3

```
double test(double x){
    int sb;
    x = x + x;
    //hello kitty!
}
```

图 9 测试样例 3

这个程序中没有注释，词法分析地结果中也不应该出现注释的结果.

```
(double,6 ReservedWord)
(test,1 Identifier)
((,5 Boundary)
(double,6 ReservedWord)
(x,2 Identifier)
(),5 Boundary)
({,5 Boundary)
(int,6 ReservedWord)
(sb,3 Identifier)
(;,5 Boundary)
(x,2 Identifier)
(=,3 ArithmeticOperator)
(x,2 Identifier)
(+,3 ArithmeticOperator)
(x,2 Identifier)
(;,5 Boundary)
(},5 Boundary)
```

图 10 测试样例 3 词法分析结果

四、实验中遇到的问题总结

（一）实验过程中遇到的问题以及解决方法：

问题 1：对于标识符和常量的处理

由于标识符和常量的个数可以是无限的，所以这个存储上用数组实现必须对于其长度大小和数量做一定的限制。在实际操作中，我假设一个标识符和一个常量不会超过 40 个字节，且一共不会超过 1000 个，从而解决了存储的问题。同时，一般情况下我们对于其他种类都是采用一类一码制，而对于这两个类中的每一个单词我们都额外设置一个属性值，从而实现存储。另外，由于我们假设了标识符和常数的长度不超过 1000，且种类码分别为 1，2。因此实际存储的时候，我们用(1000, 2000)中的数表示标识符，用(2000, 3000)中的数表示常数。之后进行判断类别时，我们只需要根据范围相应地减去 1000 或者 2000，从而达到判断类别地目的。同时，这两个类中有 1000 个位置可以存放相应地属性值，达到了类内存储不互斥的目的。

问题 2：对于注释的处理

在对注释进行识别的时候，容易将单行注释符号和整除搞混，同样是“//”，可以通过判断“//”之后的字符串的内容来确定“//”的实际含义。使用超前读取的机制可以解决这个问题。多行注释和单行注释的实现类似，具体代码如下：

```
    //首字符是 / 有可能是除号 也有可能是注释
else if (ch == '/') {
    col++;
    word += ch;
    ch = fgetc(fp);
    //这种情况是除号
    if (ch != '*' && ch != '/') {
        seekresult = fseek(fp, _Offset: -1, SEEK_CUR);
        //3:算数运算符
        return 3;
    }
    //注释符//：这一行剩下的全被注释了
    if (ch == '/') {
        word.clear();
        while ((ch = fgetc(fp)) && ch != '\n' && !feof(fp)) {}
        if (feof(fp)) {
            flag = 0;
            return 0;
        } else {
            seekresult = fseek(fp, _Offset: -1, SEEK_CUR);
        }
        line++;
        col = 1;
        return 0;
    }
```

```
if (ch == '*') {
    bool flag5 = 1;
    while (flag5) {
        word.clear();
        ch = fgetc(fp);
        col++;
        if (ch == '\n') {
            line++;
            col = 1;
        }
        if (ch != '*')
            continue;
        else {
            ch = fgetc(fp);
            col++;
            if (ch == '\n') {
                line++;
                col = 1;
            }
            if (ch == '/') {
                flag5 = 0;
            } else continue;
        }
        if (feof(fp)) {
            flag = 0;
```

图 11 注释的处理

（二）思考题的思考与分析

思考题 1：

自动机的初始态为 0，每次根据自动机的当前状态，词法分析器分析当前得到的单词，再依据根据状态转换图，进入下一个状态。就这样不断读入，不断处理，不断进行状态转移，直到到达状态转换图的终止态。程序首先通过预处理函数将源代码中的空格、tab 与换行符统一替换成单个空格，然后通过 for 循环按顺序一一读取源代码中的字符，每读到一个字符，就去查找事先设定好的 DFA 函数表，转换成相应的状态，如果出现了结束状态，就意味着单词已经被识别或者出现错误，就可以生成相应的 token。同时，在某些情况下，我们需要进行超前搜索。实现定义的 DFA 让我们清晰地知道每一步地转换。

思考题 2：

1. 我的实现：符号表是通过 C++的数组实现的，如前所述，我们对于标识符和常数规定只有 1000 个且不超过 40 字节的存储空间，这样做的好处是我们可以进行常数时间内的查找，而且存储起来十分方便。缺点就是限制了存储空间和数量。
2. 改进方法：建立一张 *hash table*，通过哈希表存储符号表，这样可以极大地减少符号表的插入和查找的时间复杂性，提升符号表插入和查找操作的性能。根据分析，长度为 n 的线性表被分割成 m 个长度为 m/n 的线性表，此时的时间复杂度是

$\mathrm{T}(n,e,m)=O(n(n+e)/m)$。如果 m 取 n 时，$\mathrm{T}(n,e,m)=O(n+e)$。相比于线性表存储符号表，时间复杂度从平方级别降到一次方级别。

五、实验体会

通过本次实验，我对编译原理的词法分析器有了更加深刻的了解。

词法分析是编译的第一个阶段，将构成源程序的字符串转换成等价的单词序列的程序。词法分析器从文件中读入表示源程序的字符流，按照程序功能等价的要求，将其转换成对应的单词序列，并提出其中的空格、注解等不影响程序语义的字符。本次实验实现的词法分析部分是语法分析与语义分析的基础，所以要尽可能地精简和正确，这样一来如果之后的部分出现问题，就不需要再回过头来排查问题的根源是否出在词法分析部分。重点还是在 DFA 的定义上。同时我的 C++代码能力得到了锻炼。

指导教师评语：

日期：