编译原理大作业报告

PL/0 语言词法分析程序

学号：517021911065　姓名：刘浩文　班级：F1703603
联系方式：IssacLiewX@sjtu.edu.cn

2019 年 11 月 25 日

摘要

在这次大作业实践中，我完成了大作业的全部要求，包括：填补程序中所有缺失部分，添加程序对特定符号计数并输出功能，添加程序处理注释的功能。此外，我还对程序进行了进一步完善，使其输出更合理清晰。之后，使用完成后的词法分析程序对提供的 PL/0 源程序进行了编译，得到了很好的输出结果。此外，我还使用 PL/0 语言编写了一个计算正整数 a 和 b 的 $Bezout$ 等式的程序（在 $Bezout.txt$ 文件中），并使用词法分析程序对其进行了编译，也得到了很好的输出结果。本报告先概述了词法分析的原理和作用，然后对任务要求的完成情况及最后的输出结果进行了展示，之后记录并分析了实践过程中出现的一些问题，最后表达了对本次作业的感想与一些建议。

关键词：　编译原理　　词法分析程序　　PL/0 语言

1 词法分析概述

词法分析是编译过程的第一步。词法分析的任务是对字符串表示的源程序从左到右地进行扫描和分解，根据语言的词法规则识别出一个一个具有独立意义的单词符号。

执行词法分析的程序称为词法分析程序，或称词法分析器或扫描器。

2 词法分析程序的补全与改进

2.1 调试环境

以下列出本次实践的基本调试环境：

- **机器**：$MacBook\ Pro\ (13-inch,\ 2016,\ Four\ Thunderbolt\ 3\ Ports)$
- **系统**：$macOS\ Catalina\ Version 10.15.1$
- **调试软件**：$Xcode\ Version\ 11.2\ (11B52)$

2.2 词法分析程序的补全

Step 1: $getch()$ 函数 $/*\ TO\ BE\ MODIFIED\ */$ 位置填空。这一段 $while$ 循环是为了读取一整行源码存入符号数组 $line$ 中，同时输出该行源码并计算出该行源码长度保存在变量 ll 中。

```
while (!feof(infile) && (ch = getc(infile))!='\n')
{
    printf("%c", ch);
    /* TO BE MODIFIED */
        line[++ll] = ch;
} // while 这里读入了源代码符号串中的一行字符串
```

Step 2: 在 pl0.h 中的 $/*\ TO\ BE\ MODIFIED\ */$ 位置，仿照 $word[]$ 和 $wsym[]$，根据已给出的 $csym[]$，完成 $ssym[]$ 的定义。阅读 pl0.h 与 pl0.c 文件可知，$ssym$ 保存了 10 个运算符与界符的类别，用于之后进行符号匹配。

```
int ssym[NSYM+1] =
{
    /* TO BE MODIFIED */
    SYM_NULL, SYM_PLUS, SYM_MINUS, SYM_TIMES, SYM_SLASH, SYM_LPAREN, SYM_RPAREN, SYM_EQU, SYM_COMMA,
        SYM_PERIOD, SYM_SEMICOLON
};
```

Step 3: 在 pl0.c 中 $getsym()$ 函数的 $/*\ TO\ BE\ MODIFIED\ */$ 位置，根据 PL/0 语言的要求，仿照对“>”的处理，完成与“<”符号相关的符号的处理。处理依据的 FA 如图 1。

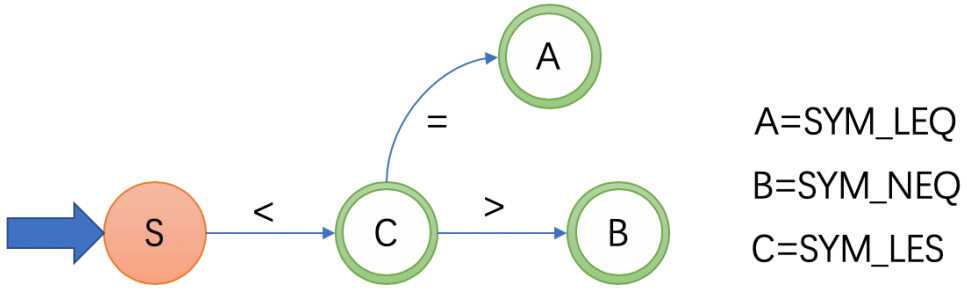

图 1: 处理与“<”符号相关的符号的 FA

```
else if (ch == '<')
    {
        /* TO BE MODIFIED */
        g_etch();
        if (ch == '=')
        {
            sym = SYM_LEQ;
            g_etch();
        }
        else if(ch == '>')
        {
            sym = SYM_NEQ;
            /* TO BE MODIFIED */
            C_all++;
            C_neq++;
            g_etch();
        }
        else
        {
            sym = SYM_LES;
            /* TO BE MODIFIED */
            C_all++;
            C_les++;// <
        }
```

Step 4: 在 pl0.c 中添加 /* *TO BE MODIFIED* */ 位置，添加对“:=”，“>”，“<”，“<>”符号计数的处理功能：

定义了全局变量来存储各符号的个数。具体计数见 **Step 3**，识别出相应符号后直接将对应全局变量加一。

```
/* TO BE MODIFIED */
int   addr = 0;
int   C_all = 0;
int   C_becomes = 0;
int   C_gtr = 0;
int   C_les = 0;
int   C_neq = 0;
```

Step 5: 完成 *main* 函数中/* *Please output types of words in every line* */位置的代码，输出每一行识别到符号的类别。由于全局变量 *sym* 中保存了当前识别到的符号的类别，因此直接将 *sym* 输出即可。另外，由于全局变量 *cc* 存储了当前所处的源码行的具体位置，而 *ll* 保存了该源码行的长度，因此可以使用 *cc* 与 *ll* 进行比较来达到识别完一行源码后换行输出的目的。

```
while(ch != '.')
{
    getsym();

    /* TO BE MODIFIED */
    /* Please output types of words in every line */
        printf("%d ",sym);
        if (cc >= ll) printf("\n");
}
```

Step 6: 在原有函数功能的基础上加入对注释的处理，其中注释由 (* 和 *) 包含，不允许嵌套。(具体位置在 pl0.c 的 getsym 函数中/*Skip Notes*/位置)。

为了能够发现注释只有开始符没有结束符的错误，并告知错误开始（注释开始符）的位置，定义了 *addr* 全局变量来指示注释开始符的位置：

```
/* TO BE MODIFIED */
int  addr = 0;
```

接下来完成识别并跳过注释的程序，流程图如图 2。需要说明的是，若注释只有注释开始符 '(*' 而没有结束符 '*)'，那从注释开始符后到程序结束符 '.' 都将被跳过，并报错：在第 *addr* 行注释没有结束符。这是借鉴了 *Xcode* 对 C 语言注释无结束符的处理机制。

```
    /* TO BE MODIFIED */
    /* Skip  Notes*/
    else if (ch == '(')
    {
        g_etch();
        if (ch == '*')
        {
            addr = cx - 1;//记录注释起始位置，方便报错
            g_etch();
            char ch1 = ch;//ch1与ch保存最近经过的两个符号
            while((ch1 != '*') || (ch != ')') )//检查最近经过的两个符号是否组成注释结束符
            {
                ch1 = ch;
                g_etch();
                if(ch == '.')
                    printf("ERROR: Line %d Unterminated Notes.",addr);
                    //若直到程序结束都没有注释结束符，则报错，指出注释起始位置
            }//这里跳过所有注释开始符之后的字符串，直到遇到注释结束符
            g_etch();
            getsym();//直接跳过注释，读取下一个符号
        }
        else sym = SYM_LPAREN;
    }
```

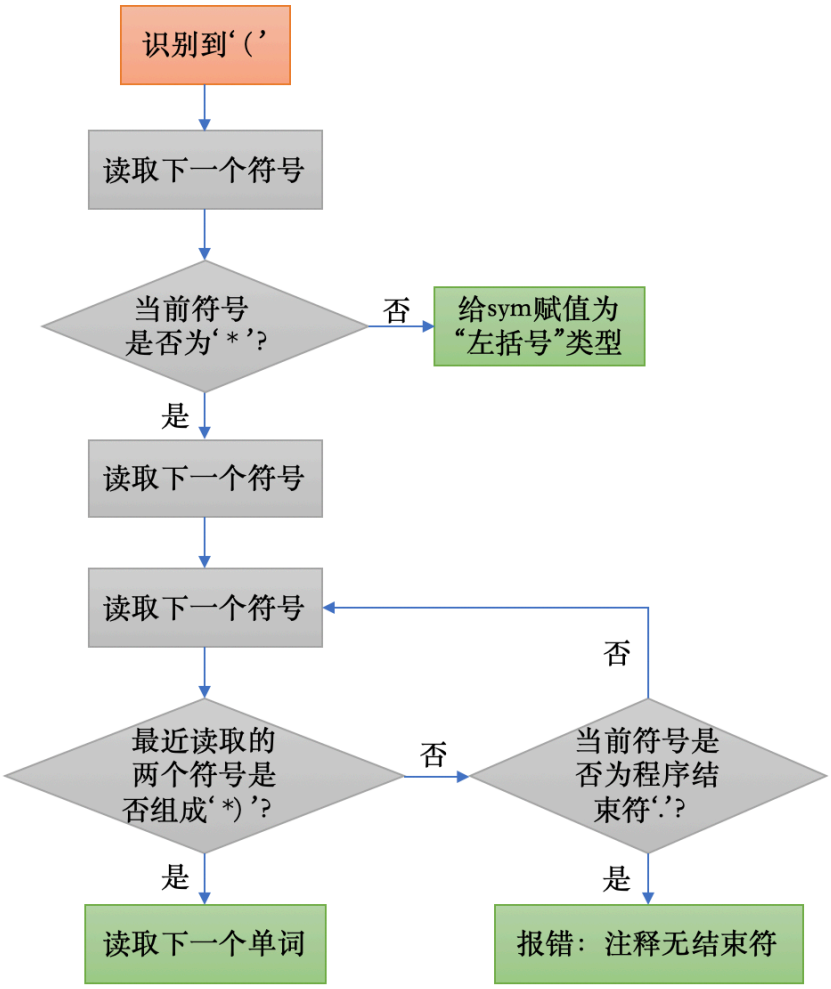

图 2: 识别并跳过注释的流程图

Step 7: 增加文件输出 (将 **Step 4** 中的计数功能的结果输出到文件)，输出的文件名同输入文件名，后缀为.out，即如果为 test.txt，输出文件命名为 test.out。

```
/* TO BE MODIFIED */
/* Please output the number of ":=", ">", "<", "<>"  to file "test.out"*/
outfile = fopen("/Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/报告/test.out", "w");
fprintf( outfile, "All ':=','>','<','<>' : %d\n':=' : %d\n'>' : %d\n'<' : %d\n'<>' : %d\n", C_all,
    C_becomes, C_gtr, C_les, C_neq);
```

2.3 词法分析程序的改进

1. *g_etch*() 函数中修改了对 *cx* 的操作，使其能指示代码的行数；且补全为五位数，便于与输出中的符号类别区分：

```
////////////////////////////////////////////////////////////////////
void g_etch(void) //由于笔者的Xcode的C环境的curses.h文件中已有getch()函数的定义，因此这个project中的原
    getch()函数的命名都改为了g_etch()
```

```
{
    if (cc == ll)                       //若一行已结束，这里读入下一行代码
    {
        if (feof(infile))               //若文件结束。feof():文件结束，返回非0值，文件未结束，返回0值
        {
            printf("\nPROGRAM INCOMPLETE\n");
            exit(1);
        }
        ll = cc = 0;                    //若文件未结束，ll和cc（代码读入指针）重新计数
        printf("%05d  ", cx);           //输出的整型宽度为n位，右对齐，%05d即宽度为5位，位数小于5则前补零
        while (!feof(infile) && (ch = getc(infile))!='\n')
        {
            printf("%c", ch);
            /* TO BE MODIFIED */
            line[++ll] = ch;

        }                               // while 这里读入了源代码符号串中的一行字符串
        printf("\n");
        line[++ll] = ' ';               //最后ll等于该行代码长度
        /* TO BE MODIFIED */
        cx++;                           //指示代码当前行数
    }
    ////////////////////////////////////////////
    ch = line[++cc];                     //若一行还没有结束，这里读入该行下一个字符
} // getch
```

2. 原 *getsym*() 函数中的 [*while* (*ch* == ' ' || *ch* ==′*t*′) *g_etch*();] 会跳过所有空格和制表符。如果在源代码的某行末尾有空格，那 *getsym*() 会跳过换行符直接到下一行继续读，这样 *main* 函数中的输出便会失去换行的机会，虽然对词法分析本身并无影响，但会使输出不易阅读。如图 3。

```
Please input source file name: /Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/报告/test.txt
00000  const m = 7,n = 85;
27 1 8 2 16 1 8 2 17 00001  var x,y,z,q,r;
28 1 16 1 16 1 16 1 16 1 17 00002
00003  procedure multiply;
29 1 17 00004  var a,b;
28 1 16 1 17
00005  begin
20 00006    a :=x;b :=y;z :=0;
1 19 1 17 1 19 1 17 1 19 2 17 00007     while b > 0 do
24 1 12 2 25
00008   begin
20 00009        if odd b then z := z + a;
22 7 1 23 1 19 1 3 1 17
00010       a := 2 * a; b := b / 2;
1 19 2 5 1 17 1 19 1 6 2 17
```

图 3: 原来的输出可能会变得较为混乱

将该段代码改为如下形式可以解决这个问题，使输出变得整洁美观，如图 4。

```
void getsym(void)
{
    int i, k;
    char a[MAXIDLEN + 1];

    while (ch == ' ' || ch=='\t')
    {
        g_etch();
```

```
        /* TO BE MODIFIED */
        if (cc >= ll) printf("\n");      //该行是增加用来使输出更整齐美观
    }
```

```
Please input source file name: /Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/报告/test.txt
00000  const m = 7,n = 85;
27 1 8 2 16 1 8 2 17
00001  var x,y,z,q,r;
28 1 16 1 16 1 16 1 16 1 17
00002
00003  procedure multiply;
29 1 17
00004  var a,b;
28 1 16 1 17
00005  begin
20
00006    a :=x;b :=y;z :=0;
1 19 1 17 1 19 1 17 1 19 2 17
00007    while b > 0 do
24 1 12 2 25
00008    begin
20
00009        if odd b then z := z + a;
22 7 1 23 1 19 1 3 1 17
00010        a := 2 * a; b := b / 2;
1 19 2 5 1 17 1 19 1 6 2 17
```

图 4: 加一行代码后使输出整齐美观，不受行尾空格影响

3 程序输出结果展示

3.1 对提供的 test.txt 中源程序进行词法分析

源程序展示如下（**注意第 28 行加了一行进行测试的注释**）。

```
const m = 7,n = 85;
var x,y,z,q,r;

procedure multiply;
var a,b;
begin
    a :=x;b :=y;z :=0;
    while b > 0 do
    begin
        if odd b then z := z + a;
        a := 2 * a; b := b / 2;
    end
end;

procedure divide;
var w;
begin
    r := x; q := 0; w := y;
    while w > y do
    begin
        q := 2 * q; w := w / 2;
        if w <= r then
        begin
            r := r - w;
```

```
            q := q + 1;
        end;
    end
end;
(*注(释*测)试*)

procedure gcd;
var f, g;
begin
    f := x;
    g := y;
    while f <> g do
    begin
        if f < g then g := g - f;
        if g < f then f := f - g;
    end
end;

procedure max;
var i, j, k;
begin
    i := x;
    j := y;
    k := 0;
    if i < j then k := j;
    if j < i then k := i;
end;


begin
    x := m; y := n; call multiply;
    x := 25; y := 3; call divide;
    x := 34; y := 36; call gcd;
end
.
```

使用完成的词法分析程序进行编译后输出结果如图 5、6、7 (**注意第 28 行加了一行进行测试的注释**)。可以看到每行行标后输出源代码，并在下一行输出各单词的类别。且注释能正确地完全跳过。

```
Please input source file name: /Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/报告/test.txt
00000  const m = 7,n = 85;
27 1 8 2 16 1 8 2 17
00001  var x,y,z,q,r;
28 1 16 1 16 1 16 1 16 1 17
00002
00003  procedure multiply;
29 1 17
00004  var a,b;
28 1 16 1 17
00005  begin
20
00006    a :=x;b :=y;z :=0;
1 19 1 17 1 19 1 17 1 19 2 17
00007    while b > 0 do
24 1 12 2 25
00008    begin
20
00009        if odd b then z := z + a;
22 7 1 23 1 19 1 3 1 17
00010        a := 2 * a; b := b / 2;
1 19 2 5 1 17 1 19 1 6 2 17
00011    end
21
00012  end;
21 17
00013
00014  procedure divide;
29 1 17
00015  var w;
28 1 17
00016  begin
20
00017    r := x; q := 0; w := y;
1 19 1 17 1 19 2 17 1 19 1 17
00018    while w > y do
24 1 12 1 25
00019    begin
20
00020        q := 2 * q; w := w / 2;
1 19 2 5 1 17 1 19 1 6 2 17
00021        if w <= r then
22 1 11 1 23
00022        begin
20
00023            r := r - w;
1 19 1 4 1 17
00024            q := q + 1;
1 19 1 3 2 17
00025        end;
21 17
00026    end
21
00027  end;
```

图 5: 词法分析 *test.txt* 输出结果

```
00028  (*注(释*测)试*)
00029

00030  procedure gcd;
29 1 17
00031  var f, g;
28 1 16 1 17
00032  begin
20
00033   f := x;
1 19 1 17
00034   g := y;
1 19 1 17
00035   while f <> g do
24 1 9 1 25
00036   begin
20
00037       if f < g then g := g - f;
22 1 10 1 23 1 19 1 4 1 17
00038       if g < f then f := f - g;
22 1 10 1 23 1 19 1 4 1 17
00039   end
21
00040  end;
21 17
00041
```

图 6: 词法分析 *test.txt* 输出结果

```
00042  procedure max;
29 1 17
00043  var i, j, k;
28 1 16 1 16 1 17
00044  begin
20
00045   i := x;
1 19 1 17
00046   j := y;
1 19 1 17
00047   k := 0;
1 19 2 17
00048   if i < j then k := j;
22 1 10 1 23 1 19 1 17
00049   if j < i then k := i;
22 1 10 1 23 1 19 1 17
00050  end;
21 17
00051

00052

00053  begin
20
00054   x := m; y := n; call multiply;
1 19 1 17 1 19 1 17 26 1 17
00055   x := 25; y := 3; call divide;
1 19 2 17 1 19 2 17 26 1 17
00056   x := 34; y := 36; call gcd;
1 19 2 17 1 19 2 17 26 1 17
00057  end
21
00058  .\377
18
Compile Success!
Program ended with exit code: 0
```

图 7: 词法分析 *test.txt* 输出结果

为了测试注释的结束符缺失报错是否有效，将第 28 行的注释结束符去掉，只保留注释开始符，即：

```
(*注(释*测)试
```

```
Please input source file name: /Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/报告/test.txt
00000  const m = 7,n = 85;
27 1 8 2 16 1 8 2 17
00001  var x,y,z,q,r;
28 1 16 1 16 1 16 1 16 1 17
00002
00003  procedure multiply;
29 1 17
00004  var a,b;
28 1 16 1 17
00005  begin
20
00006   a :=x;b :=y;z :=0;
1 19 1 17 1 19 1 17 1 19 2 17
00007   while b > 0 do
24 1 12 2 25
00008   begin
20
00009       if odd b then z := z + a;
22 7 1 23 1 19 1 3 1 17
00010       a := 2 * a; b := b / 2;
1 19 2 5 1 17 1 19 1 6 2 17
00011   end
21
00012  end;
21 17
00013
```

图 8: 词法分析 *test.txt* 输出结果（注释出错）

```
00026   end
21
00027  end;
21 17
00028  (*注(释*测)试
00029
00030  procedure gcd;
00031  var f, g;
00032  begin
00033   f := x;
00034   g := y;
00035   while f <> g do
00036   begin
00037       if f < g then g := g - f;
00038       if g < f then f := f - g;
00039   end
00040  end;
00041
00042  procedure max;
00043  var i, j, k;
00044  begin
00045   i := x;
00046   j := y;
00047   k := 0;
00048   if i < j then k := j;
00049   if j < i then k := i;
00050  end;
00051
00052
00053  begin
00054   x := m; y := n; call multiply;
00055   x := 25; y := 3; call divide;
00056   x := 34; y := 36; call gcd;
00057  end
00058  .\377
ERROR: Line 28 Unterminated Notes.
PROGRAM INCOMPLETE
Program ended with exit code: 1
```

图 9: 词法分析 *test.txt* 输出结果（注释出错）

程序输出结果如图 8、9。可以看到注释之前的源程序都能被正常编译（这里只截取开头一部分）；而从注释开始符后到程序结束符 . 都被跳过，并报错：**ERROR: Line 28 Unterminated Notes.**。这是借鉴了

Xcode 对 C 语言注释无结束符的处理机制。

3.2 对自己撰写的的 Bezout.txt 中源程序进行词法分析

源程序展示如下：

```
(*--UTF-8--*)
(*这是使用pl0语言编写的一个计算正整数a和b的Bezout等式的程序*)
(*Created by 刘浩文 517021911065 on 2019/11/21*)

var a, b, q, c, r, R, s, S, t, T;

procedure initial    (*该过程将变量初始化*)

begin

    q = 0, c = 0;
    r = a, R = b;
    s = 1, S = 0;
    t = 0, T = 1;
end;

procedure LOOP       (*该过程计算出a与b的最大公因数并保存在r中, Bezout等式的s值保存在S中, t值保存在T中*)
begin
    while R <> 0 do
    begin
        c = S;
        S = -q * S + s;
        s = c;
        c = T;
        T = -q * T + t;
        t = c;
        q = r / R;
        c = R;
        R = -q * R + r;
        r = c;
    end
end;

begin            (*主函数, 输入a,b的数值得到其Bezout等式*)
    a = 1847, b = 9832;
    call initial;
    q = r / R;
    c = R;
    R = -q * R + r;
    r = c;
    call LOOP;
end
.
```

使用完成的词法分析程序进行编译后输出结果如图 10、11、12。可见，词法分析程序能正确对自己撰写的 PL/0 源程序进行词法分析。

```
Please input source file name:
    /Users/hongxing/Desktop/大三上学期/编译原理/大作业/词法分析大作业/517021911065_刘浩文_词法分析/Compil_prin/Bezout.txt
00000  (*--UTF-8--*)
00001  (*这是使用pl0语言编写的一个计算正整数a和b的Bezout等式的程序*)

00002  (*Created by 刘浩文 517021911065 on 2019/11/21*)

00003

00004  var a, b, q, c, r, R, s, S, t, T;
28 1 16 1 16 1 16 1 16 1 16 1 16 1 16 1 16 1 16 1 17
00005

00006  procedure initial    (*该过程将变量初始化*)
29 1
00007

00008  begin
20
00009

00010   q = 0, c = 0;
1 8 2 16 1 8 2 17
00011   r = a, R = b;
1 8 1 16 1 8 1 17
00012   s = 1, S = 0;
1 8 2 16 1 8 2 17
00013   t = 0, T = 1;
1 8 2 16 1 8 2 17
00014  end;
21 17
00015
```

图 10: 词法分析 *Bezout.txt* 输出结果

```
00016  procedure LOOP       (*该过程计算出a与b的最大公因数并保存在r中，Bezout等式的s值保存在S中，t值保存在T中*)
29 1 00017  begin
20
00018   while R <> 0 do
24 1 9 2 25
00019   begin
20
00020       c = S;
1 8 1 17
00021       S = -q * S + s;
1 8 4 1 5 1 3 1 17
00022       s = c;
1 8 1 17
00023       c = T;
1 8 1 17
00024       T = -q * T + t;
1 8 4 1 5 1 3 1 17
00025       t = c;
1 8 1 17
00026       q = r / R;
1 8 1 6 1 17
00027       c = R;
1 8 1 17
00028       R = -q * R + r;
1 8 4 1 5 1 3 1 17
00029       r = c;
1 8 1 17
00030   end
21
00031  end;
21 17
```

图 11: 词法分析 *Bezout.txt* 输出结果

```
00032

00033  begin            (*主函数，输入a,b的数值得到其Bezout等式*)
20 00034    a = 1847, b = 9832;
1 8 2 16 1 8 2 17
00035   call initial;
26 1 17
00036   q = r / R;
1 8 1 6 1 17
00037   c = R;
1 8 1 17
00038   R = -q * R + r;
1 8 4 1 5 1 3 1 17
00039   r = c;
1 8 1 17
00040   call LOOP;
26 1 17
00041  end
21
00042  .\377
18
Compile Success!
Program ended with exit code: 0
```

图 12: 词法分析 *Bezout.txt* 输出结果

4 实验中的问题与思考

由于提供的源码已经比较完整，需要改动与添加的地方较少，且清晰易懂，所以完成此次大作业实践没有遇到什么大的问题。一个需要注意的问题就是对注释没有结束符的处理。这里我借鉴了当时正在使用的 Xcode 对 C 语言注释无结束符的处理方法：注释仍生效，即注释开始符之后的所有源代码全部跳过，然后标示出该注释开始符的位置并报错：Unterminated /* comment.，如图 13。我认为这是一种非常友好而方便的机制，因此对此进行了效仿。

图 13: *Xcode* 对 *C* 语言注释无结束符的处理

另一个问题是程序一开始的输出可读性很差，一行的单词类别输出与下一行的源程序输出混杂在一起。在 *main* 函数中进行改进后发现仍有某些行不能正确换行。对源程序进行研究发现不能正确换行的都是行尾有多余的空格的行。所以我对词法分析程序进行了排查，最后分析出问题在 *getsym*() 函数中并进行了改进，如第 2.3 节所述。最后程序的输出整齐美观，达到了我的期望。

5 感想与建议

5.1 感想

通过这次大作业，我综合了课堂所学的理论知识，完成了一个词法分析程序，对词法分析形成了一个感性认识。之前学习理论知识的时候，只单单了解词法分析的原理与过程，对实际操作并没有成型的概念。而这次使用 C 语言写一个词法分析程序，让我对词法分析有了新的认识。特别是头文件中符号类型的定义方式让我打开了新思路。这次的实践定将让我受益匪浅。

5.2 建议

这次的大作业比较简单且工作量较少，让人有一种刚到兴头上却不得不戛然而止的感觉。希望提供的源程序只是为我们搭好框架，比如说只定义头文件，定义函数名和少量处理提示，具体的内容由我们自己完成，这样可能会让我们充分发挥。另外个人觉得这次大作业与编译原理的关系其实不是很紧密，并没有使用到多少课堂上的理论知识，而更偏重于 C 语言编程，希望这点之后能改变。