哈尔滨工业大学

编译系统 2023 春

实验一

学院：	未来技术学院
姓名：	刘天瑞
学号：	7203610121
指导教师：	朱庆福

一、 实验目的

1．巩固对词法分析与语法分析的基本功能和原理的认识；
2．能够应用自动机的知识进行词法分析与语法分析；
3．理解并处理词法分析与语法分析中的异常和错误。

二、 实验内容

(一)实验的运行环境

虚拟机 Ubuntu 20.04.1 kernel version 5.15.0-67-generic；GCC version 9.4.0；GNU Flex version 2.6.4-6.2；GUN Bison version 2:3.5.1

(二)程序功能

1. 本次实验所使用到的数据结构

本次实验使用到的数据结构较为简单：1 个结构体数组，里面维护节点名称（包括语法单元与词法单元名称）、行号（语法单元），Num（子节点数量），Child_List（子节点列表），1 个联合体 union（TYPE，ID，INT，FLOAT etc.存放相应数据）。除此之外还有 1 个数组用于存放这棵语法分析树 TreeNode，同时并且设置了 flag 标志来判断是否为子节点（输出打印时用到）。

2. 词法分析

词法分析属于整个该实验甚至语法分析中的基础部分，我先进行词法分析的编写。单独对于词法分析的部分来说，我成功地实现了以下所列举的实验所要求的功能：

（1） 能够正确识别十进制数、八进制数以及十六进制数并能进行进制转换；
（2） 能够正确识别浮点数、指数形式的浮点数；
（3） 可以判断出错误的八进制数、十六进制数、浮点数、指数形式数；
（4） 能够识别所有的注释符号，并判断出错误的注释符号；
（5） 能够识别所有的词法单元的标识符 ID；
（6） 能够特别地识别出标识符 ID 中的 TYPE 类型（即区分 int 和 float 类型）。

因为其中正确识别十进制数、八进制数、浮点数等都是实验基本要求，故省略。本实验较为特别的内容是实现判断出词法分析中的错误类型。我所尝试使用的方法是对于错误的八进制数、十六进制数再进行重新定义。例如：八进制数首位为 0 但后面的数中不能出现 9，所以对首位为 0 且后面出现 9 的数将其定义为 INT8_ERROR。同理，对于 INT16_ERROR 我认定在 0x 后面出现[g-zG-Z]都为错误类型。

```
(0[xX]([0-9a-fA-F]*[g-zG-Z][0-9a-zA-Z]*)?) {
```
非法的十六进制数

```
0[0-7]*[8-9][0-9]* {
```
非法的八进制数

而对于指数形式的浮点数则略有些复杂，因为形如.5E03 与 10.e3 都是对的，但是对于形如 10.e 与.e 和

10.与.05 都是错误的。我的定义如下图所示：

```
FLOAT_ERROR [0]+(0|[1-9][0-9]*)\.[0-9]+\.
SCIENCE_ERROR [0-9]+[Ee][+-]?[0-9]*|([+-]?([0-9]*\.[0-9]+\.[0-9]*)[Ee][+-]?[0-9]+\.[0-9]+)
```

```
((([0-9])+\.([0-9])+)|(((([0-9])+\.([0-9])*)|(([0-9])*\.([0-9])+))[Ee]([+-]?)([0-9])+)) {
```

同样对于 ID_ERROR，定义为数字在前字母在后即可。对于包括以上的这些错误，匹配 pattern 和 action，便可以打印出错误信息了（即利用 scanner 来测试，这边为方便直接使用 parser）。由此一来词法分析器便可以识别出源程序中到底是哪部分出现了哪种错误了：

```
hiterltr@ubuntu:~/byxt_lab1_3$ ./parser test.cmm
Error type B at Line 1: syntax error, near'sturct'
Error type A at Line 2: Illegal ID '36d'
Error type A at Line 3: Illegal ID '35e'
Error type A at Line 8: Illegal float number '.56'
Error type A at Line 9: Illegal ID '004e'
Error type A at Line 15: Illegal ID '0x45'
Error type B at Line 16: syntax error, near'='
```

打开(O) test.cmm ~/byxt_lab1_3 保存(S)

```
1 sturct
2 36d
3 35e
4 0;
5 123
6 0067.6900
7 135.34
8 .56
9 004e+3
10 __int64
11 A3
12 a5_56
13 abc_
14 int a = 012389,
15 int b = 0x45;
16 float c = 34.23
17 float c_4 = 002.
```

除了能识别出特定的词法错误以外，我的词法分析程序还能够识别出不同的注释符号，但是在刚开始编写词法分析程序时，注释符号是通过正则表达式文法写的，但是这样会在后面更为深入的语法分析中出现报错，因为程序虽然会认为该符号合法，但是类似”//”这种符号，程序很难识别其后面到底是注释内容还是程序内容，所以我最后还是选择了额外编用写代码来实现，即碰到”//”和”/*”后一直读入字符，直到读到”\n”和”*/”停止。

3. 语法分析

在接下来的语法分析部分，我所做出的主要成果在于递归遍历语法分析树的建立实现与一部分错误恢复。该分析树程序主要实现的功能包括如下列所示：

（1） 构造语法树并按照先序编批的方式打印每 1 个节点信息；

（2） 语法单元：打印输入文件中的行号。若产生 error，则不打印；

（3） 词法单元：打印词法单元的名称，不打印该词法单元的行号；

（4） ID：打印标识符词素；TYPE：打印 int 或者 float；NUMBER：打印该十进制数；

（5） 若程序错误，则打印行号，并且判断该错误类型。同时能进行错误恢复，使得程序得以继续成功运行。

在写该部分时我曾经尝试过直接打印输出语法分析的错误类型，但是这样一来出现的错误可能性过多了，所以后来我并没有能够全部写完，最后只能就写了 1 个缺少’]’，而为了保证整个程序语法较为统一起来，也就没能使用该部分代码和内容。

首先为语法分析树的构建：对于 1 个语法单元或者词法单元，声明 1 个结构体（数据结构中有提到）。建立节点的过程就是将 pattern 模式中匹配到的语法单元和词法单元作为节点内容来进行创建，然后再通过压栈操作将其放置到数组中：

```
/* 高级定义 */
Program : ExtDefList {$$ = NewTreeNode(@$.first_line, "Program", 1, $1); root = $$;} /* 定义根节点 */
    ;
ExtDefList : ExtDef ExtDefList {$$ = NewTreeNode(@$.first_line, "ExtDefList", 2, $1, $2);}
    | /* 空 */ {$$ = EmptyNode();}
    ;
ExtDef : Specifier ExtDecList SEMI {$$ = NewTreeNode(@$.first_line, "ExtDef", 3, $1, $2, $3);}
    | Specifier SEMI {$$ = NewTreeNode(@$.first_line, "ExtDef", 2, $1, $2);}
    | Specifier FunDec CompSt {$$ = NewTreeNode(@$.first_line, "ExtDef", 3, $1, $2, $3);}
    | error SEMI {hasFault = TRUE;}
    ;
ExtDecList : VarDec {$$ = NewTreeNode(@$.first_line, "ExtDecList", 1, $1);}
    | VarDec COMMA ExtDecList {$$ = NewTreeNode(@$.first_line, "ExtDecList", 3, $1, $2, $3);}
    ;
```

特别的，接着还需要判断子节点是否为终结符，若不是则将子节点数组导入到该节点，若是终结符则又需要判断该节点是否为 ID，TYPE 或者 INT、FLOAT。如果是这些字符，则将行号设置为-1（后续打印输出节点时要用到），附上部分关键代码如下图所示：

```
//递归遍历语法分析树，语法分析树的高度Height从0开始
static void Traverse_Print(P_Node tree, int Height)
{
    if (tree != NULL && tree->Token != NULL)
    {
        printf("%*s", 2 * Height, ""); // 注意先缩进两个空格
        printf("%s", tree->Token);  // 其次打印其名称
        if ((!strcmp(tree->Token, "ID"))||(!strcmp(tree->Token, "TYPE")))
        {
            printf(": %s", tree->Id_Type);
        }
        else if(!strcmp(tree->Token, "INT"))
        {
            printf(": %d", tree->intVal);
        }
        else if(!strcmp(tree->Token, "FLOAT"))
        {
            printf(": %f", tree->floatVal);
        }
        else if (!tree->Is_leaf) // 为非终结符
        {
            printf(" (%d)", tree->line);
        }
        printf("\n");
        Traverse_Print(tree->first_child, Height + 1);
        Traverse_Print(tree->next_brother, Height);
    }
}
```

最后我在创建节点的时候判断是否为终结符（因为若是终结符则必为子节点，但是在其父节点时已经被打印输出了，于是后续便无需重复再打印输出）。将该节点进行入栈操作（以便后续的先序遍历输出），这里的栈结构我是用数组来实现的，最后打印输出时也只打印出非终结符节点的内容以及其所有的子节点（此时并不会造成重复）。除此之外，我打印输出时就只需要打印出行号不为-1 的节点并且如果该节点是非终结符的话，还需要打印出该行的行号。以上便是总体大概的语法分析树实现过程。

(三)编译过程

1. 在编译语法分析源代码时，先利用 bison 进行编译 syntax.y 这个源代码生成语法分析器；
2. 编译好的结果会保存在 syntax.tab.c 和 syntax.tab.h 这两个文件中，.h 格式文件中包含词法单元的类型定义，但是同时也需要在词法分析的源代码中加入对该.h 格式文件的引用，然后加上该引用后再利用 flex 进行编译 lexical.l 这个源代码生成词法分析器；
3. 最后便得到了 lex.yy.c 这一词法分析器，当然此时还需要 NodeTree.c（主函数）/NodeTree.h(头文件)，syntax.tax.h（函数定义和声明）和 syntax.tab.c（bison 所编译的），最后将以上提及的这些文件用 GCC 链接起来，再加入定义好的库函数-lfl 和-ly；
4. 最后进行语法分析，生成最终的合成词法语法分析器 parser：

```
hiterltr@ubuntu:~/byxt_lab1_1$ bison -d syntax.y
hiterltr@ubuntu:~/byxt_lab1_1$ flex lexical.l
hiterltr@ubuntu:~/byxt_lab1_1$ gcc syntax.tab.c lex.yy.c TreeNode.c -lfl -ly -o parser
```

(四)实验结果

这里我就仅仅展示一下必做内容和选做内容的几个标准分析结果如下图所示：

```
hiterltr@ubuntu:~/byxt_lab1_3$ ./parser test1.cmm
Error type A at Line 4: Mysterious characters '~'
hiterltr@ubuntu:~/byxt_lab1_3$ ./parser test5.cmm
Error type B at Line 3: syntax error, near'123'
Error type A at Line 4: Illegal ID '0x3F'
```

```
hiterltr@ubuntu:~/byxt_lab1_3$ ./parser test3.cmm
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: inc
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
            SEMI
        StmtList (4)
          Stmt (4)
            Exp (4)
              Exp (4)
                ID: i
              ASSIGNOP
              Exp (4)
                Exp (4)
                  ID: i
                PLUS
                Exp (4)
                  INT: 1
            SEMI
        RC
```

三、 实验总结与收获

1. 我学会了如何使用 flex，bison 等编译工具分别进行词法分析和语法分析；
2. 我学会了将有穷状态自动机原理进行良好应用，对源程序的语法、词法错误恢复以及准确的错误判断有较多的改进；
3. 我学会了在摸索中实现了词法分析里对于十进制整数、八进制数、十六进制数以及浮点数的错误判断和注释符号的识别；
4. 我学会了如何顺利地进行语法分析树的表示与构建，并且进行遍历来打印输出相关信息；
5. 我还学会了如何保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口来保证整个实验代码的健壮性和可读性。