编译原理实验一

姓名	刘凯锋
学号	2021111057
班级	2103601

一，实验目的

1. 理解词法分析与语法分析的原理
2. 实现词法分析与语法分析程序
3. 能够认识到处理过程中的错误导致原因

二，实验内容

（一）实验要求

能够识别词法类型错误(Type A)与语法类型错误(Type B)。除此之外，还应能识别八进制数和十六进制数，指数形式的浮点数，识别“//”和“/...//”形式的注释。

（二）实验环境

> Ubuntu 20.04 GCC 7.5.0 Flex 2.6.4 Bison 3.5.1

（三）实验过程

1. 数据结构定义

```c
struct AFT{
    int line; //number of line where the node is located
    char* name; // name of node
    enum TYPE type; // type of node(terminator or non-terminator)(for printer)
    union{ // value of node(if they have)
        int i;
        float f;
        char* id;
    }value;

    struct AFT* child; // the first child of the node
    struct AFT* brother; // the next brother of the node

typedef struct AFT *treeNode;
```

定义抽象语法树的每个节点，考虑到数的可扩展性，我这里采用基于节点的方式构建树，这里定义了一个枚举类型type用于定义节点的类型，再根据类型将其值存储到value中。
增加非叶节点：对应非终结符的分析过程。
增加叶节点：对应终结符的分析过程。
这里简单列一下定义：

```c
struct AFT* addNode(int line, char* name, enum TYPE type, int argc, ...);
// insert leaf(has not children)
struct AFT* addLeaf(int line, char* name, enum TYPE type, char* val);
// release memory
void release(struct AFT* node);
void preorder(struct AFT* node, int layer);
void yyerror(const char* msg);
```

考虑到构建语法分析树的时候，一个节点的子节点数目是不确定的。所以定义为变长参数，只需传入变长参数列表长度。
这里的relese()为释放节点与其子节点，preorder()为前序遍历用于输出程序语法分析结构，yyerror重新写了错误输出。另外，高度并没有定义，通过递归调用的过程中记录高度，并且输出。

2.词法分析

词法分析要求：

1. 识别八进制数，十进制数，十六进制数并且可以进行进制转换
2. 识别正确的浮点数，指数形式的浮点数
3. 识别出错误的八进制数，十六进制数，浮点数，指数形式数
4. 能够识别注释
5. 识别出ID,TYPE,INT,FLOAT并打印相应的信息
   前两点均为基本要求，这里比较特殊的是我对错误的八进制数，十六进制数，浮点数错误进

行了特殊的定义来保证可以输出实验测试用例中的结果。

```
 /*错误*/
AERROR .
INT8_ERROR 0[0-7]*[8-9]+[0-7]*
INT16_ERROR 0[Xx][a-fA-F0-9]*[g-zG-Z]+[a-fA-F0-9]*
FLOAT_ERROR [0]+(0|[1-9][0-9]*)\.[0-9]+|\.[0-9]+|[0-9]+\.|[0-9]+[Ee][+-]?[0-9]*|([0-9]+|(([0-9]+\.[0-9]*|[0-9]*\.[0-9]+)([Ee][+-]?[0-9]+)?))[fFlL]?
ID_ERROR [0-9]+[a-zA-Z_]+
```

```
{AERROR} {
    printf("error type A at line %d: mysterious character '%s'\n", yylineno, yytext);
}
{INT8_ERROR} {
    printf("error type A at line %d: Illegal octal number '%s'\n", yylineno, yytext);
}
{INT16_ERROR} {
    printf("error type A at line %d: Illegal hexadecimal number '%s'\n", yylineno, yytext);
}
{ID_ERROR} {
    printf("error type A at line %d: Illegal ID '%s'\n", yylineno, yytext);
}
```

我在实验过程中单独进行了测试，结果如下：

~/Desktop/complier/scanner_test

```
1 int
2 float
3 =
4 +
5 -
6 _hello
7 13
8 1.45
9 1.05e-4
10 10.e
11 .e-5
12 10.e3
13 0x13
14 07
15 09
16 0x1g
17 (
18 )
19 {
20 }
21 //int
22 /*hello*/
23 0hello
```

kfliu@kfliu-virtual-machine: ~/Desktop/complier/scanner_test

```
int
float
=
+
-
id:_hello
int:13
float:1.45
float:1.05e-4
error type A at line 10: Illegal float point number '10.'
id:e
.
id:e
int:-5
float:10.e3
int:0x13
int:07
error type A at line 15: Illegal octal number '09'
error type A at line 16: Illegal hexadecimal number '0x1g'
(
)
{
}
error type A at line 23: Illegal ID '0hello'
```

3.语法分析

语法分析要求：

1. 构建语法分析树，先序遍历打印节点信息
2. 语法单元：打印行号，若产生e，则不打印
3. 词法单元：打印词法单元名称，不打印行号，能输出词法单元的“高度”
4. ID,TYPE,NUMBER的打印要求
5. 能够判断程序错误，输出TYPE B类型错误
   这里我考虑到程序的扩展性，采用第一部分所述的数据结构，那么语法分析的过程就是一个建立抽象语法树的过程。
   我将语法树分为非叶节点和叶节点，分别对应非终结符和终结符。
   对于非终结符，只需要按照实验指导书附录，写它们的产生式即可。

```
Program: ExtDefList {$$=addNode(@$.first_line,"Program",NOT_A_TOKEN,1,$1);root=$$;}
    ;
ExtDefList:         ExtDef ExtDefList {$$ = addNode(@$.first_line,"ExtDefList",NOT_A_TOKEN,2,$1,$2);}
    |                                             {$$=NULL;}
    ;
ExtDef:             Specifier ExtDecList SEMI           {$$ = addNode(@$.first_line,"ExtDef",NOT_A_TOKEN,3,$1,$2,$3);}
    |               Specifier SEMI                       {$$ = addNode(@$.first_line,"ExtDef",NOT_A_TOKEN,2,$1,$2);}
    |               Specifier FunDec CompSt              {$$ = addNode(@$.first_line,"ExtDef",NOT_A_TOKEN,3,$1,$2,$3);}
    |               error SEMI                           {synError=1;}
    ;
```

这里比较特殊的是，我在建立非叶节点时采用的是输入变长参数的方法，这里的root是单独定义的一个指针，用于将其传给Program来做根节点，以`ExtDefList`为例，这里的`2`代表其传入两个变长参数，`$1`，`$2`也是语法树节点类型。用于传给addNode来建立子节点，其逻辑如下。

```c
if (argc > 0) {
    struct AFT* firstChild = va_arg(args, struct AFT*);
    newNode->child = firstChild;

    struct AFT* currentChild = firstChild;
    for (int i = 1; i < argc; i++) {
        struct AFT* nextChild = va_arg(args, struct AFT*);
        if (currentChild != NULL) {
            currentChild->brother = nextChild;
        }
        currentChild = nextChild;
    }
}
```

同样的逻辑进行叶结点的处理，由于叶节点没有子节点，所以其不需要传入变长参数。考虑根据节点的类型来输出对应的内容，其在先序遍历中核心逻辑如下：

```c
void preOrder(struct AFT* node, int layer) {
    // 根据节点类型打印值或其他信息
    switch (node->type) {
        case TOKEN_TYPE:
            printf(": %s", node->value.id);
            break;
        case TOKEN_ID:
            printf(": %s", node->value.id);
            break;
        case TOKEN_INT:
            printf(": %d", node->value.i);
            break;
        case TOKEN_FLOAT:
            printf(": %f", node->value.f); // 打印float类型的值
            break;
        case NOT_A_TOKEN:
            printf(" (%d)", node->line); // 打印行号信息
            break;
        default:
            break;
```

这样就可以实现打印正确的信息的功能。

（四）编译过程

单独词法分析测试

如果您想单独测试词法分析的过程，请进入`scanner_test`文件夹进行测试。这是我在实验前期单独写的一个程序，测试命令为：

```
./scanner sccaner_test.cmm
```

其对应的编译命令为：

```
flex lexical_test.l
gcc main.c lex.yy.c -lfl -o scanner
```

实验结果测试

编译命令：

```
bison -d syntax.y
flex lexical.l
gcc syntax.tab.c tree.c -lfl -ly -o parser
```

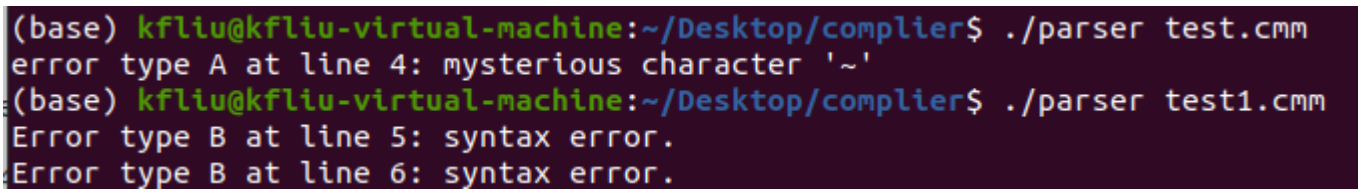
测试命令：

```
./parser test.cmm
```

需要注意的是，这里我将主函数合并在了tree.c中，而main.c是在前期单独词法分析测试中使用，双方没有关系。

（五）实验结果

可以通过所有的测试用例。
这里展示必做样例的几个结果。



实验总结

学会了flex,bison的写法，对词法分析和语法分析的过程有了深刻而直观的认识。