Lab1-实验报告：2021111161-李睿博

1）实现功能：

- **基本部分：**
  - 识别词法错误（错误类型 `A` ）与 语法错误（错误类型 `B` ）
  - 打印语法树与错误恢复
- **选作部分：**
  - 识别八进制和十六进制数与浮点数，进行进制转换
  - 识别错误的八进制、十六进制与浮点数输入，归为`A`类型错误给出对应提示信息
  - 识别`//`与"`/*..*/`格式的注释，并识别嵌套注释错误与注释未闭合错误

2）实现功能所用结构与方法：

- **语法树数据结构与构建：**

  语法树采用多叉树的结构，使用链表实现，定义如下：

```
    struct tree_node{
        char* name;                     //节点符号名称
        int line;                       //行号
        struct tree_node** child;       //孩子结点数组指针
        int child_num;                  //孩子个数
        void* literal;                  //终结符字面值
    };
```

  语法树的构建在`syntax.y`文件的动作部分进行，实际上是一个`SDT`。以`ExtDef`的产生式为例：

```
ExtDef : Specifier FunDec CompSt
    { $$= create_node("ExtDef", 0, 3, create_child_list(3, (node[]){$1, $2, $3})); }
```

  归约时，为当前归约符号`ExtDef`创建一个节点，对于所有非终结符的行号，我们先设为`0`，之后再进行处理；其孩子数量为`3`，孩子数组的指针则通过`create_child_list`函数进行创建，这个函数定义如下：

```
node* create_child_list(int num_children, node children[]);
```

  它接受待归约的几个符号作为一个数组，为其开辟内存空间并返回指向这个数组的指针。

  值得一提的是，`void* literal`用于记录终结符带有的字面值，如`INT`、`FLOAT`与`ID`的属性值，非终结符则无需设置。不用联合体而是用`void*`类型可以节省内存空间，毕竟只有少数的终结符需要设置这一项属性。终结符在`lexical.y`进行词法分析时，将创建非终结符节点，并对`INT`、`FLOAT`、`ID`调用`set_literal`函数，设置`literal`属性，定义如下：

```
void set_literal(node node,ValueType type, void* value);
```

  `ValueType`是一个枚举属性，根据传入符号不同，开辟不同大小空间并设置对应属性值，赋给`node`节点的`literal`指针。

  综上，语法树的构建是自底向上的，先在词法分析器中产生非终结符节点，然后返回语法分析器，进行逐步归约，途中添加终结符节点，最后形成一颗完整的语法树

- **识别错误的八进制数和十六进制数与浮点数**

  识别正确的八进制数、十六进制数是基本内容，不在此赘述了。

  正确的浮点数包含两种，带指数和不带指数的，我觉得值得注意的是带指数的浮点数的小数点可以出现在数字串的任意位置，包括首尾，但不带指数的浮点数不可以以`.`开头或结尾，所以其定义如下：

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

  对于错误的八进制数，我认为以`0`开头，但在之后出现了`8`和`9`的数字，应该算为错误的八进制数，正则表达式定义如下：

```
ERROR_INT8 0[0-7]*[8-9]+[0-7]*
```

  对于错误的十六进制数，我认为应该以`0x`或`0X`开头，但在其之后出现了`g-z`或`G-Z`的数，定义如下：

```
ERROR_INT16 0[Xx][0-9a-fA-F]*[g-zG-Z]+[0-9a-fA-F]*
```

```
Enter test cmm_file number:6
Error type A at Line 3: illegal INT8 number '09'
Error type A at Line 4: illegal INT16 number '0x3G'
```

图1 错误八进制数与十六进制数识别结果

对于错误的浮点数，单独的`.`被视为一次浮点数的错误输入尝试我认为是不太合理的，所以我只识别了错误的带指数浮点数，思路是在`e`或`E`后没有指数数字，或`+-`符号出现在了`e`或`E`之前，定义如下：

```
ERROR_FLOAT (([0-9]*\.[0-9]*)[eE])|(([0-9]*\.[0-9]*)[+-][eE][0-9]*)
```

```
Enter test cmm_file number:8
Error type A at Line 3: illegal FLOAT number '1.05e'
```

图2 错误浮点数识别结果

- **识别注释与嵌套、未闭合错误注释**

  行注释的识别比较简单，只需在读取到`\\`后，消耗之后的输入符号，直到遇到了换行符或`EOF`停止。

  对于块注释，我使用`switch`语句与循环模拟了一个自动机，其思路大概如下：

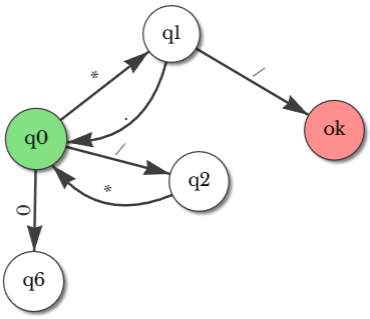

图3 识别注释的自动机（只表示思路，为美观少画了几条转移边）

  状态`0`为读取到`/*`后的开始状态，继续往后读取字符，遇到连续的`*/`则识别成功，其他字符忽略掉。不过，当遇到新的`/*`则再次进入状态`0`，如果后面再遇到`*/`就直接认为块注释结束，如果有嵌套的块注释，在这之后的`*/`会被语法分析器识别出语法错误。如果在识别块注释的时候遇到了`EOF`，则直接抛出`A`类型错误，表示注释未闭合。

- **符号所在行号**

打印语法树时需要给出每个符号所在的行号，一开始我使用的是`yylineno`作为符号，但发现一些在语法树顶层的符号的行号不正确，比如`Program`应该为第一行，但会被设置为最后一行。我猜想的原因是只有读取了全部的文本才能归约出`Program`，此时为`Program`创建节点并用`yylineno`设置行号，因为已经到文件末尾了，`yylineno`会为最后一行。

于是我的方法是只为终结符使用`yylineno`设置行号，这是在词法分析的时候进行的，所以行号都是正确的，对于非终结符，建立节点时都设为`0`，我写了一个`get_line`函数来处理行号，思路就是遍历子孙节点，当遇到一个行号非`0`的孩子时，说明这是一个终结符，所以它的行号就是该非终结符的行号。

- **进制转换**

  对于八进制、十六进制、浮点数的进制转换，我实现了自己的`stoi`和`stof`函数，思路基本上类似正则表达式匹配，然后乘上不同进制的系数累加返回，浮点数则需要进行乘方操作。

- **错误恢复**

  我在可能会出现错误的产生式位置，添加了许多产生式使得可以移入`error`符号，并找到同步单元进行恐慌模式的错误恢复，这样可以识别文件中的多个错误。

3）如何编译：

直接运行`./run.sh`进行编译。

同时也可以用以下命令来编译，并测试其他文件。

```
flex lexical.l
bison -d syntax.y
gcc syntax.tab.c NodeUtil.c -lfl -ly -lm -o parser
./parser "test_file_name"
```

4）实验结果：

篇幅所限，只展示了几个选做测试文件的结果：

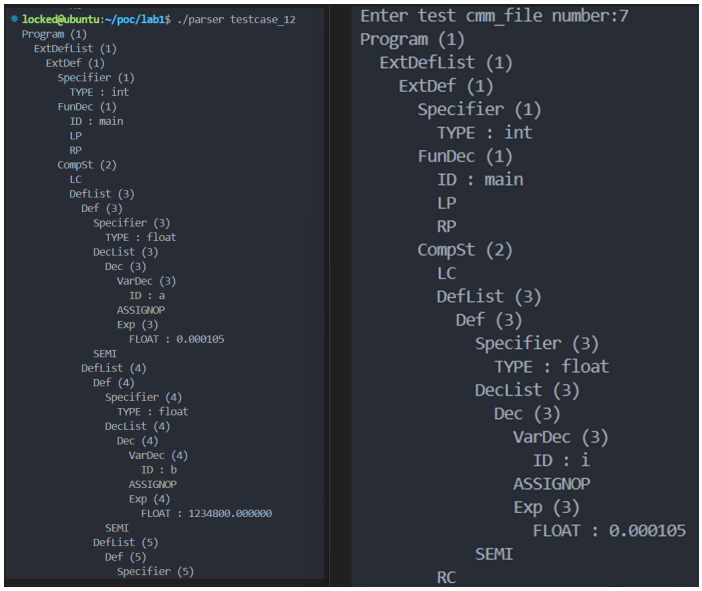

图4 测试结果

附我的实验代码仓库地址：`https://gitee.com/Lockede/poc`