姓名：刘权祥

学号：2019300414

编译原理实验一

Step 1

安装ANTLR 4：

在ubuntu18.04操作系统终端中输入如下命令进行安装。

```
$ cd /usr/local/lib
$ wget https://www.antlr.org/download/antlr-4.10.1-complete.jar
```

之后修改~/.bashrc文件，在最后面加上如下的内容即可。

这里第一行是定义了路径，第二行第三行相当于定义了一个别名，可以方便我们后续的一些操作。

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.10.1-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.10.1-complete.jar'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

学习ANTLR 4：

主要是根据antlr 4实现一个计算器--python语言熟悉了ANTLR 4的使用流程，之后根据ANTLR 4简明教程学习了ANTLR4的相关知识。

Step 2

构造词法规则：

```
lexer grammar MIDLLexerRules;
// 因为是从上到下匹配的，如果BOOLEAN 在 LITTER后面，会导致'TRUE'等都被识别成LETTER
BOOLEAN : 'TRUE' | 'true' | 'FALSE' | 'false' ;
FLOAT_TYPE_SUFFIX : 'f' | 'F' | 'd' | 'D';
UNDERLINE : '_';
INTEGER_TYPE_SUFFIX : 'l' | 'L';

INTEGER : ('0' | [1-9] [0-9]*) INTEGER_TYPE_SUFFIX?;
EXPONENT : ('e' | 'E') ('+' | '-')? [0-9]+;
```

```
FLOATING_PT:  [0-9]+ '.' [0-9]* EXPONENT? FLOAT_TYPE_SUFFIX?
        | '.' [0-9]+ EXPONENT? FLOAT_TYPE_SUFFIX?
        | [0-9]+ EXPONENT FLOAT_TYPE_SUFFIX?
        | [0-9]+ EXPONENT? FLOAT_TYPE_SUFFIX
        ;

ESCAPE_SEQUENCE : '\\'('b' | 't' | 'n' | 'f' | 'r' | '"' | '\'' | '\\' );
CHAR : '\''(ESCAPE_SEQUENCE | (~'\\'|~'\''))'\'';
STRING : '"'(ESCAPE_SEQUENCE | (~'\\'|~'"'))*'"';

LETTER : [a-z] | [A-Z];
DIGIT : [0-9];
ID : LETTER (UNDERLINE?(LETTER | DIGIT))*;

WS : [ \t\r\n]+ -> skip;
```

构造语法规则：

```
grammar MIDLGrammarRules;
import MIDLLexerRules;
//*号是0或多次，而+号是1或者多次，但是这里测试的结果是*是至少1次，+才是0到多次
specification : definition+;
definition : type_decl ';' | module ';';
module : 'module' ID '{' definition+ '}';
type_decl : struct_type | 'struct' ID;
struct_type : 'struct' ID '{' member_list '}';
member_list : (type_spec declarators ';')*;
type_spec : scoped_name | base_type_spec | struct_type;
scoped_name : '::'? ID ('::' ID)*;
base_type_spec : floating_pt_type | integer_type | 'char' | 'string' | 'boolean';
floating_pt_type : 'float' | 'double' | 'long double';
integer_type : signed_int | unsigned_int;
signed_int : ('short' | 'int16')
        | ('long' | 'int32')
        | ('long' 'long' | 'int64')
        | 'int8';
unsigned_int : ('unsigned' 'short' | 'unit16')
        | ('unsigned' 'long' | 'unit32')
        |('unsigned' 'long' 'long' | 'unit64')
        |'unit8';
declarators : declarator (',' declarator)*;
declarator : simple_declarator | array_declarator;
simple_declarator : ID ('=' or_expr)?;
array_declarator : ID '[' or_expr ']' ('=' exp_list)?;
exp_list : '[' or_expr (',' or_expr)* ']';
or_expr : xor_expr ('|' xor_expr)*;
xor_expr : and_expr ('^' and_expr)*;
and_expr : shift_expr ('&' shift_expr)*;
shift_expr : add_expr ('>>'|'<<') add_expr)*;
add_expr : mult_expr ('+'|'-') mult_expr)*;
mult_expr : unary_expr ('*'|'/'|'%') unary_expr)*;
unary_expr : ('-'|'+'|'~')? literal;
literal : INTEGER | FLOATING_PT | CHAR | STRING | BOOLEAN;
```

Step 3

使用如下的命令生成源代码。这里目标语言是python，并且没有生成listener，只生成了visitor，生成的源代码在MIDL文件夹下。

```
antlr4 -no-listener -visitor -Dlanguage=Python3 -o ../MIDL MIDLGrammarRules.g4
```

Step 4

抽象语法树定义文件在AST文件夹下的抽象语法树.pdf

Step 5

实现原理

这里我定义了一个数据结构来存储抽象语法树，并且使用缩进表示父子关系。

定义的抽象语法树如下所示：

```
class TreeNode:
    def __init__(self, type_: str, str_=""):
        self.type_ = type_ # 该节点对应的规则的名称
        self.str_ = str_ # 该节点对应的终结符（如果不是终结符就取""）

        self.level = 0 # 该节点在抽象语法树中的深度
        self.child = [] # 该节点的孩子节点们

    def add_child(self, child):
        """
        添加孩子节点
        :param child: 孩子节点
        """
        if child is None:
            return
        self.child.append(child)

    def add_end_child(self, child):
        """
        在该节点最底层的孩子节点中添加孩子节点
        这个函数只用于处理member_list-> { type_spec declarators ";" }规则
        :param child:孩子节点
        """
        if child is None:
            return
        if len(self.child) == 0:
            self.add_child(child)
        else:
            self.child[-1].add_end_child(child)

    def set_child_level(self):
        """
        设置每一个孩子节点的深度
        """
        for child in self.child:
            child.level = self.level + 1
            child.set_child_level()
```

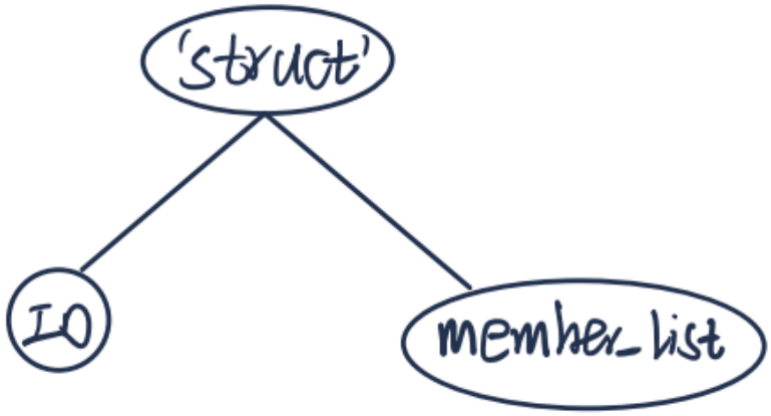
```
def get_structure(self):
    """
    获取抽象语法树的结构
    """
    structure = ""
    self.set_child_level()
    for i in range(self.level):
        structure += "\t"
    structure += str(self.type_) + "\n"
    for child in self.child:
        structure += child.get_structure()
    return structure

def get_AST(self):
    """
    获取抽象语法树的内容
    """
    AST = ""
    self.set_child_level()
    if self.type_.startswith("Terminator"):
        for i in range(self.level):
            AST += " "
        AST += str(self.str_) + "\n"
    for child in self.child:
        AST += child.get_AST()
    return AST
```

实现过程举例

继承 `MIDLGrammarRulesVisitor` 类，并且重写相关的visit函数。

比如：对于 `struct_type-> “struct” ID “{” member_list “}”`

它的抽象语法树为：



对应的代码为：

```
    # Visit a parse tree produced by MIDLGrammarRulesParser#struct_type.
    def visitStruct_type(self, ctx: MIDLGrammarRulesParser.Struct_typeContext):
        struct_type_node = TreeNode("Struct_type")
        node = TreeNode("Terminator struct", "struct")
        node.add_child(TreeNode("Terminator ID", ctx.ID()))
        node.add_child(self.visit(ctx.member_list()))
        struct_type_node.add_child(node)
        return struct_type_node
```

实现结果

在本文件夹下使用如下命令，将test/test_0.in（或者其他文件）中的MIDL源代码编译生成抽象语法树到test_0.out（注：为了方便对多个测试用例进行处理，这里没有统一输出到SyntaxOut.txt文件）。

```
python3 main.py test/test_0
```

运行截图：

如果MIDL源代码没有问题，则运行完毕后终端不会有输出，但是会保存格式化的抽象语法树。

如果MIDL源代码有问题，运行完毕后终端会提示源代码哪一行有什么问题，同样也会保存格式化的抽象语法树。

```
# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:41]
$ python3 main.py test/test_0

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:47]
$ python3 main.py test/test_1

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:57]
$ python3 main.py test/test_2

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:58]
$ python3 main.py test/test_3

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:59]
$ python3 main.py test/test_4

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:44:59]
$ python3 main.py test/test_5

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:45:00]
$ python3 main.py test/test_6

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:45:01]
$ python3 main.py test/test_7

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:45:02]
$ python3 main.py test/test_8
line 2:11 mismatched input 'int8' expecting ID
line 3:10 mismatched input '123' expecting ID
line 4:13 mismatched input '"test"' expecting ID
line 5:10 mismatched input '12' expecting ID
line 6:13 extraneous input '_' expecting ';'

# immortalqx @ Legion in ~/JetBrainsProjects/PycharmProjects/编译原理实验一 [15:45:03]
$ python3 main.py test/test_9
line 2:17 missing '[' at '0.1e+10'
line 2:24 mismatched input ';' expecting {',', ']'}
line 3:19 extraneous input '|' expecting {'+', '-', '~', BOOLEAN, INTEGER, FLOATING_PT, CHAR, STRING}
line 6:35 mismatched input ';' expecting {'+', '-', '~', BOOLEAN, INTEGER, FLOATING_PT, CHAR, STRING}
line 7:22 extraneous input '&' expecting {'+', '-', '~', BOOLEAN, INTEGER, FLOATING_PT, CHAR, STRING}
```

Step 6

这部分的说明在test文件夹下的测试说明.pdf中。