实验一：使用Antlr4开源工具构建MIDL语言编译器前端

> ## 实验内容
>
> 1. 根据Antlr4开源工具简介内容，自行搜索**学习Antlr4的使用方法**，主要涉及1）如何再G4文件中定义词法语法的规则。2）如何使用Antlr4工具解析G4生成相应的词法，语法分析程序。3）如何调用词法，语法分析程序中的方法生成对应MIDL源代码的抽象语法树。
> 2. 完成MIDL词法语法规则的**G4文件定义**。
> 3. 使用Antlr4编译G4文件**生成MIDL语言词法，语法分析程序的源代码**。
> 4. 根据MIDL语法规则的文法**给出相应的抽象语法树结构**，提交相应的语法树定义的文件,语法树的定义尽量简洁易懂，必要时提供说明。
> 5. 使用生成的词法，语法分析程序构建一个从MIDL源代码到抽象语法树的分析程序，并输出格式化的抽象语法树到SyntaxOut.txt文件中。
> 6. 给出测试方法描述,提交readme.doc,如果你还有其它需要说明的问题须写在readme.doc中。
> 7. 编程语言不限制，目前Antlr4支持的target有Java, C#, Python2|3, JavaScript, Go, C++, Swift, Dart, PHP，只要在这些语言范围之内都可以。

实验结果

1.g4文件定义

MIDLg4文件见MIDL.g4

Grammer Structure

```
/** Optional javadoc style comment */
grammar Name; ①
options {...}
import ... ;

tokens {...}
channels {...} // lexer only
@actionName {...}

rule1 // parser and lexer rules, possibly intermingled
...
ruleN
```

grammar的名字必须和g4文件名字相同. 一个g4文件必须有一个header ①和至少一个词法或语法规则. 其余的options, tokens,channels等可选.

rule:

```
ruleName : alternative1 | ... | alternativeN ;
```

如果语法声明前没有前缀,则改语法包含词法和语法规则,也可以加上parser 或 lexer前缀 表明该语法中只有语法规则或此法规则.可以用import继承,和面向对象中的继承类似,继承父类所有的规则,可复写.

Token rules大写字母开头,Parser rules 小写字母开头.

2.antlr生成词法语法分析程序

使用命令,因为我的格式化语法树遍历输出是采用visitor接口实现的,所以需要加上-visitor参数.

```
$antlr4 MIDL.g4 -visitor
```

3.抽象语法树结构定义文件

定义结构详见 **抽象语法树设计.doc**

4.遍历抽象语法树并输出(使用ANTLR接口)

首先尝试使用listener模式,被visitor接口的范式形式吓到了,但是listener太依赖于遍历顺序,输出并不好控制,所以采用visitor. 采用直接打印的方式输出到文件.

实现思路为从语法树顶端开始vist节点,然后根据每一条文法规则的子结构,继续向下遍历或输出叶节点.具体代码见`ASTGeneratorToTXT.java`.输出的格式化树结构在`SyntaxOut.txt`文件中.

-visitor参数会让antlr为每一个文法规则生成一个visit函数. 在第4部把每一个visit函数都覆写了,通过该条文法规则的访问逻辑覆写该函数.例如member_list函数,通过判断当前子结构的个数来写对改文法的访问或打印操作.

```
/**
 * member_list: (type_spec declarators ';')*;
 *
 * @param ctx
 * @return
 */
@Override
public String visitMember_list(MIDLParser.Member_listContext ctx) {
    out.print("[member_list ");
    int n = ctx.getChildCount();
    if (n == 0) {
        out.print("]");
        return null;
    } else {
        for (int i = 0; i < n / 3; i++) {
            visit(ctx.getChild(3 * i));
            visit(ctx.getChild(3 * i + 1));
            out.print(ctx.getChild(3 * i + 2).getText());
        }
    }
    out.print("]");
    return null;
}
```

5.测试说明

> 测试g4是否正确
>
> 测试生成的抽象语法树是否设计的语法树.

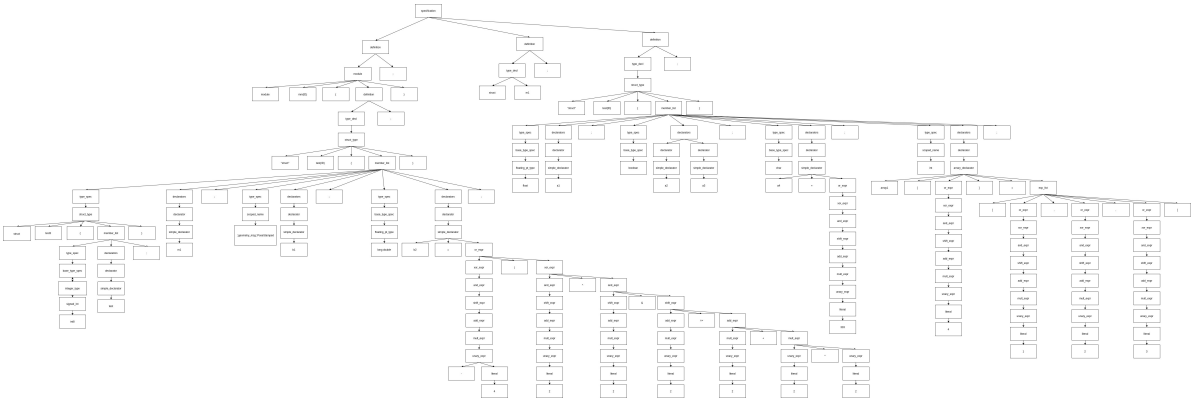
g4文法

测试代码详见`test.java`文件.测试用例test.txt文件

经过测试和debug,修改前期g4文法出现的错误,最终测试用例均无报错,且visit函数覆盖率为100%.

ASTGenerator	100% (1/1)	100% (26/...	100% (176...
ASTGeneratorToTXT	100% (1/1)	100% (27/...	100% (180...

然后,下图是部分测试用例的完整分析树.该分析树格式化输出与antlr生成分析树相同. 证明g4文法没有出现错误.



格式化AST输出和设计是否相同

将测试样例分为以下几个类别进行分析,每个类别都有多个不同变量类型的测试用例.

1.module和struct结构体 + 变量声明

测试样例

```
module mm{
struct test2{
int m1;
};};
```

测试输出

```
module mm
    struct test2
        member
            type_spec
                int
            declarators
                m1
```

推导:module->module ID {difinition}->module ID {type_decl}->module ID { struct_type}->module ID {struct ID {member_list}}->module ID {struct ID {member}}->module ID {struct ID {member{type_spec, declarators}}}... type_spec->int declarators->m1

设计时省略了{}等符号,也省略了definition, type_decl等父节点,只留下了member,type_spec,declarators等关键字分别表示一行声明,变量类型和变量名称.所以输出与设计相符合.

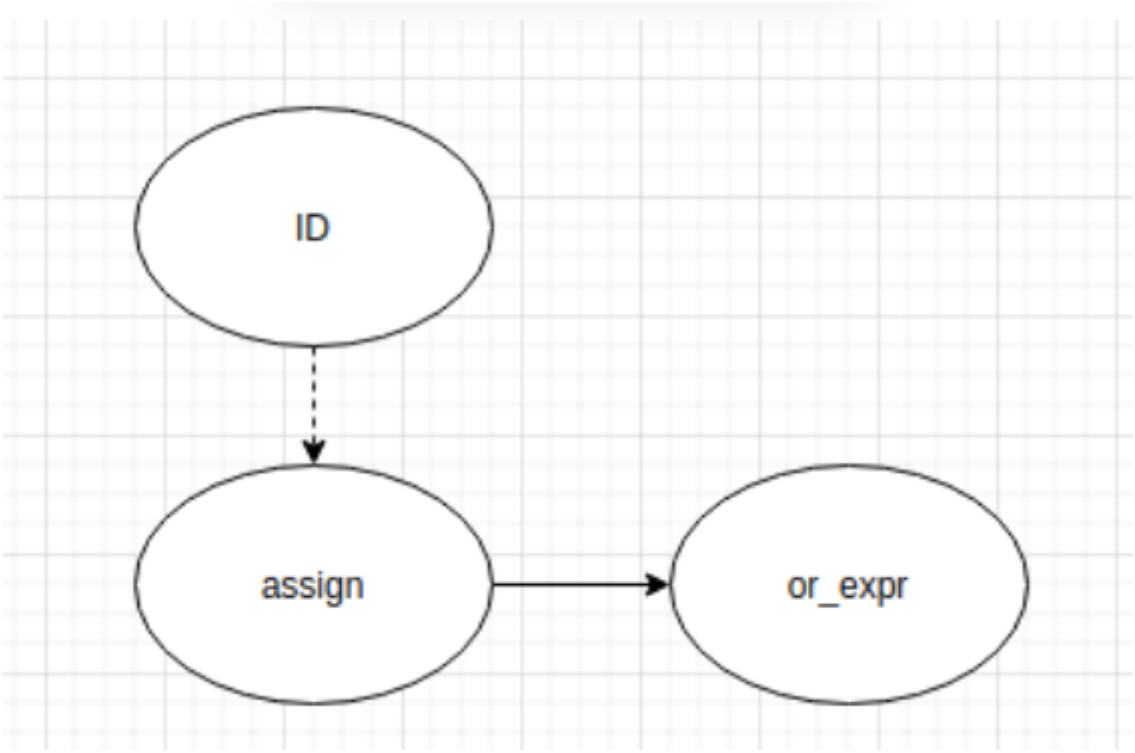
2.声明变量时赋值

在结构体中输入`string s1="hello world!";`

测试输出(此处省略前面结构体的输出,只留下声明赋值的ast输出)

```
        member
            type_spec
                string
            declarators
                s1
                    assign
                    "hello world!"
```

分析

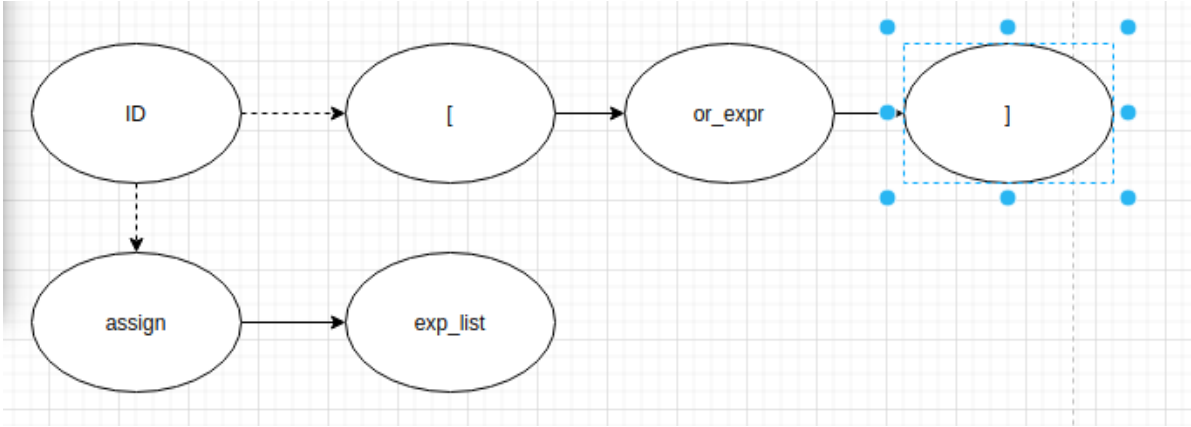


declarators->declarator->simple_declarator->ID = or_expr 按照AST设计这里应该为declaratos{s1{assign,"hello world"}}.所以输出与设计相符合

3.数组类型变量及其赋值

输出

```
        member
            type_spec
                int
            declarators
                array1 [ 4 ]
                    assign
                    [ 1, 2, 3]
```



declarators->declarator->arrary_declarator->ID '[' or_expr ']' ('=' exp_list) ast输出应为ID [or_expr] {assign,exp_list}.输出与设计相符合

4.结构体作为变量类型

输入

```
module mm{
struct test2{
struct test3{
int8 test;
} m1;
...
}
```

输出

```
module mm
    struct test2
        member
            type_spec
                struct test3
                member
                    type_spec
                        int8
                    declarators
                        test

            declarators
                m1
    ...
```

这里struct test3作为一个type_spec表示m1的变量类型.输出和设计相符

5.赋值时存在计算符号

输出

```
        member
            type_spec
                long double
            declarators
                b2
                    assign
                    -4|2^2&2>>2+2*2
```

因为在处理or_expr : xor_expr ('|' xor_expr)*;xor_expr : and_expr ('^' and_expr)*;and_expr : shift_expr ('&' shift_expr)*;等文法规则时没有保留父节点的关键字,所以会一直递推下去只留下计算过程中的计算符号.输出和设计相符合.

附件

`ast.jpg`基于visitor遍历文法规则得到的分析树

`TreeOut.txt`基于visitor遍历文法规则得到的分析树输出

`ASTGenerator.java` visitor遍历文法规则代码实现分析树(打印)

`SyntaxOut.txt`基于visitor遍历文法规则得到的**抽象语法树**的格式化输出

`ASTGeneratorToTXT.java` visitor遍历文法规则代码并将**抽象语法树**格式化输出到syntaxout.txt文件中

`MIDL.g4` MIDL的定义文件

`test.txt` 测试用例

`test.java` 测试函数