构建MIDL语言的编译器MIDLCompiler

实验环境

IDE：IntelliJ IDEA 2023.2.3 (Ultimate Edition)

编程语言：Java

ANTLR版本：4.13.0

操作系统：Windows 10 22H2 19045.4412

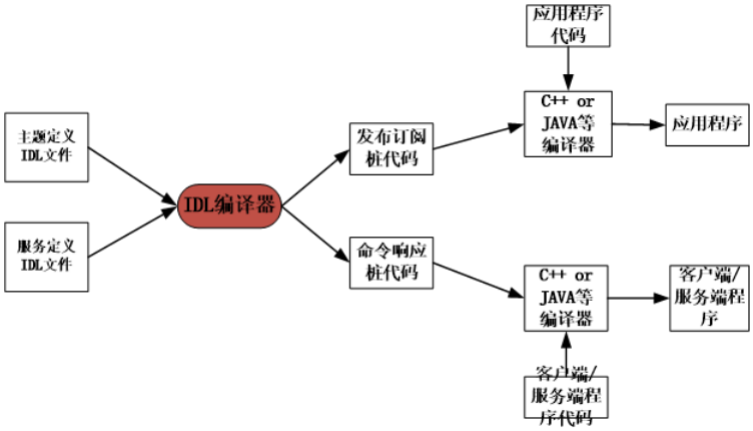
实验内容

IDL（Interface Definition Language）是一种平台无关的接口描述语言。IDL主要用于描述分布式通信的数据接口，它的语法结构简单、功能强大、跨平台，是分布式数据通信应用开发的最佳选择。IDL编译技术完成两项工作：

1）生成发布订阅需要的数据结构和桩文件；

2）生成命令响应过程中的桩代码。

IDL语言使得嵌入式中间件通信所需的数据接口和具体实现语言分离，通过IDL编译器代码生成工具，生成相应语言的接口代码，可大大提高应用程序开发效率。



本实验项目是通过裁剪IDL语言，形成MIDL（Mini Interface Definition Language），MIDL是IDL语言的一个语法子集，作为编译原理课程的实验对象。本实验项目要求研制MIDL语言的一款编译器MIDLCompiler，编译器MIDLCompiler实现从MIDL源代码转换为C++源代码的功能，被称为源到源编译器（source-to-source compiler）。在该实验项目中，要求实现MIDLCompiler源码的词法分析和语法分析，并生成抽象语法树，在此基础上进行语义分析与C++代码生成。

实验流程

基于Antlr4的MIDL词法语法规则的构建

ANTLR v4是一款功能强大的语法分析器生成器，可以用来读取、处理、执行和转换结构化文本或二进制文件。它被广泛应用于学术界和工业界构建各种语言、工具和框架。

从称为文法的一种形式化的语言描述中，ANTLR生成该语言的语法分析器。生成的语法分析器可以自动构建语法分析树——表示文法如何匹配输入的数据结构。ANTLR还可以自动生成树遍历器，你可以用它来访问那些树的节点，以执行特定的代码。

ANTLR v4的语法分析器使用一种新的称为`Adaptive LL(*)`或`ALL(*)`的语法分析技术，它可以在生成的语法分析器执行前在运行时动态地而不是静态地执行文法分析。

ANTLR v4极大地简化了匹配句法结构（如算术表达式）的文法规则。

词法分析器处理字符序列并将生成的词法符号提供给语法分析器，**语法分析器**随即根据这些信息来检查语法的正确性并建造一颗语法分析树。

这个过程对应ANTLR类是CharStream、Lexer、Token、Parser，以及ParseTree。

学习ANTLR书籍：《ANTLR4权威指南》

通过上面的学习，最开始是声明了简单计算器的词法和语法规则，然后知晓G4文件基本的定义规则。假如给出以下MIDL的语法规则：

```
specification -> definition { definition }
definiton -> type_decl“;”| module “;”
module -> “module”ID “{” definition { 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”| “uint16”)
   | (“unsigned”“long”| “uint32”)
   | (“unsigned” “long” “long” | “uint64”)
   | “uint8”
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
```

然后就根据MIDL词法语法规则编写了G4文件，并且可以在IDEA的Antlr4插件中可视化语法树来看是否有那里定义错误。

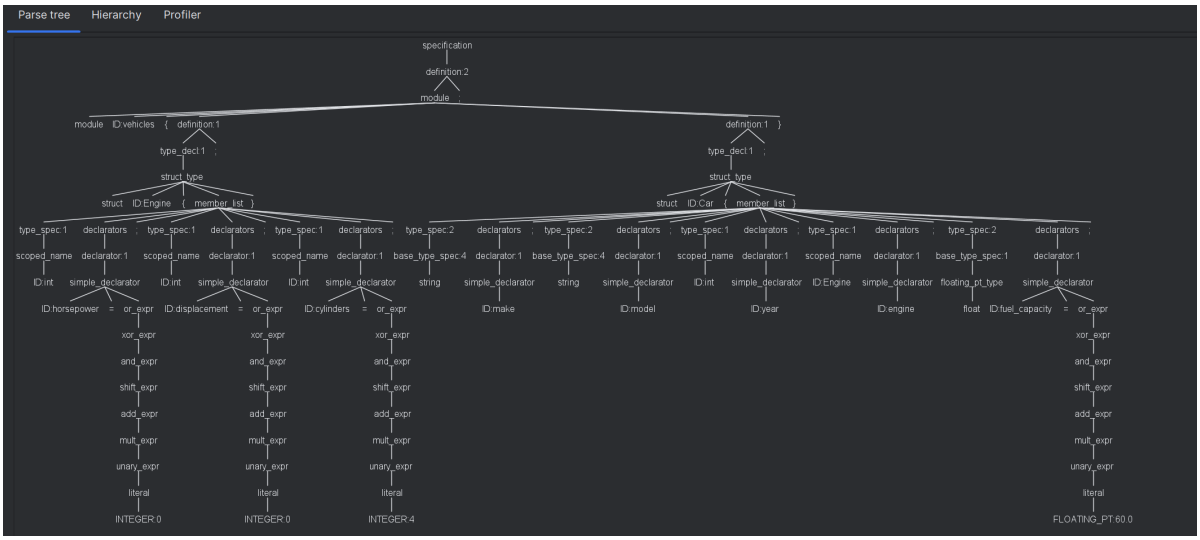
```
/*
 * Parser rules
 */
grammar MIDL;
```

```
import MIDL_Lexer;

specification : (definition)+; // 序列模式，{ }表示括起来的内容可以重复0至n次

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 ';' )* ; // 带终止符的序列模式，{ }表示括起来的内容可以重复0至n次

type_spec : scoped_name
            | base_type_spec
            | struct_type;

scoped_name : ('::')?ID ('::' ID )*; // 可选

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'| 'uint16')
            | ('unsigned long'| 'uint32')
            | ('unsigned long long' | 'uint64')
            | 'uint8';

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 )* ']';
```

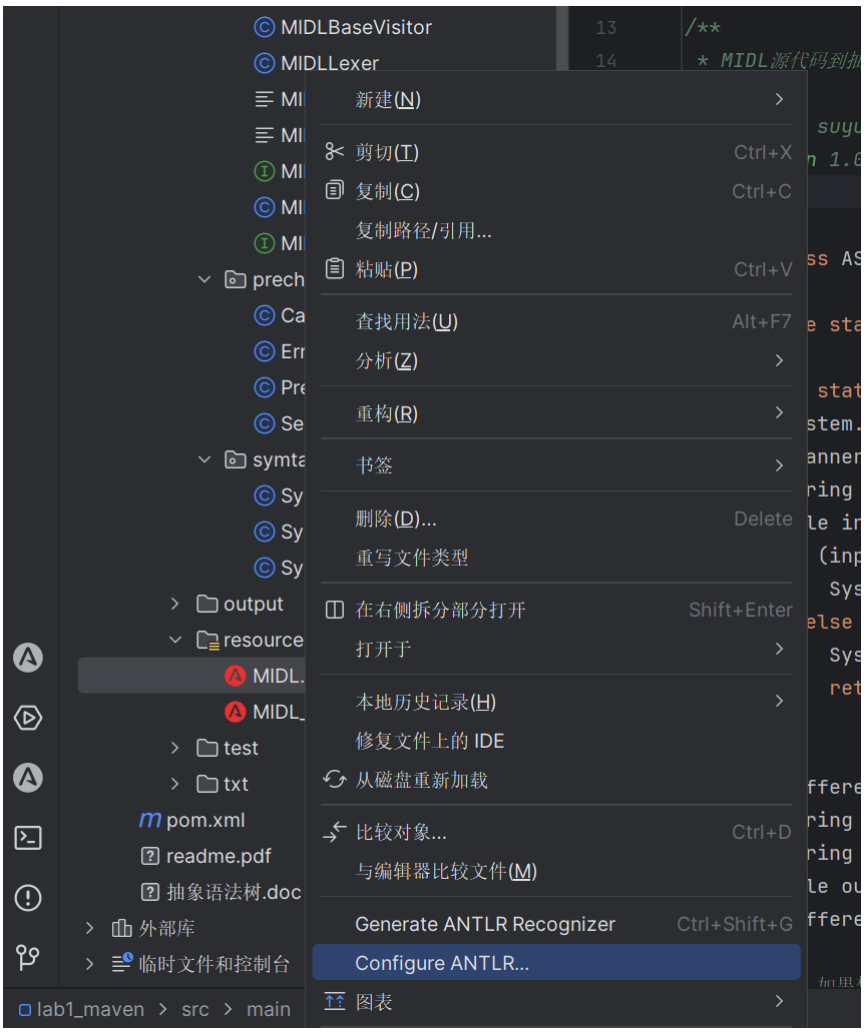
```
or_expr : xor_expr ('|' xor_expr )*;

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;
```

例如：输入以下结构体

```
module vehicles {
    struct Engine {
        int horsepower = 0;
        int displacement = 0;
        int cylinders = 4;
    };

    struct Car {
        string make;
        string model;
        int year;
        Engine engine;
        float fuel_capacity = 60.0;
    };
};
```

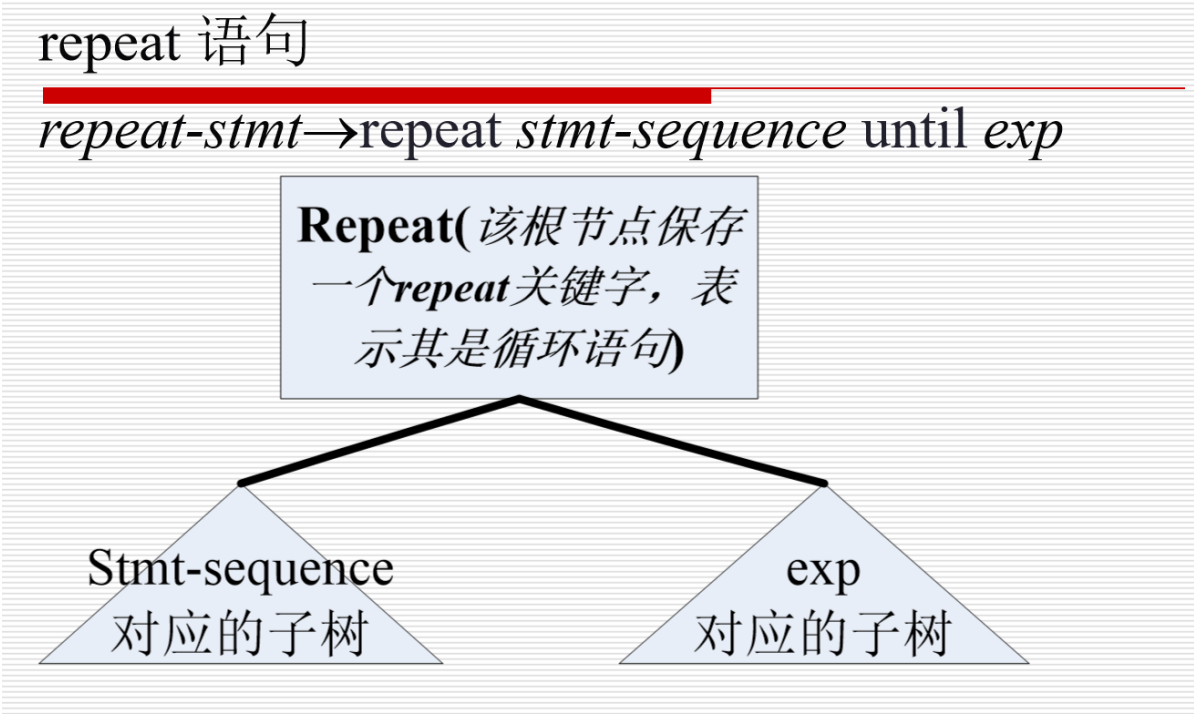


之后便是使用Antlr工具生成G4文件对应的词法语法分析程序源码，插件的话直接选中g4文件然后右键生成即可。



基于Antlr4的MIDL编译器前端实现与测试

首先是根据MIDL语法规则的文法给出相应的抽象语法树结构，可以参考ppt例子。



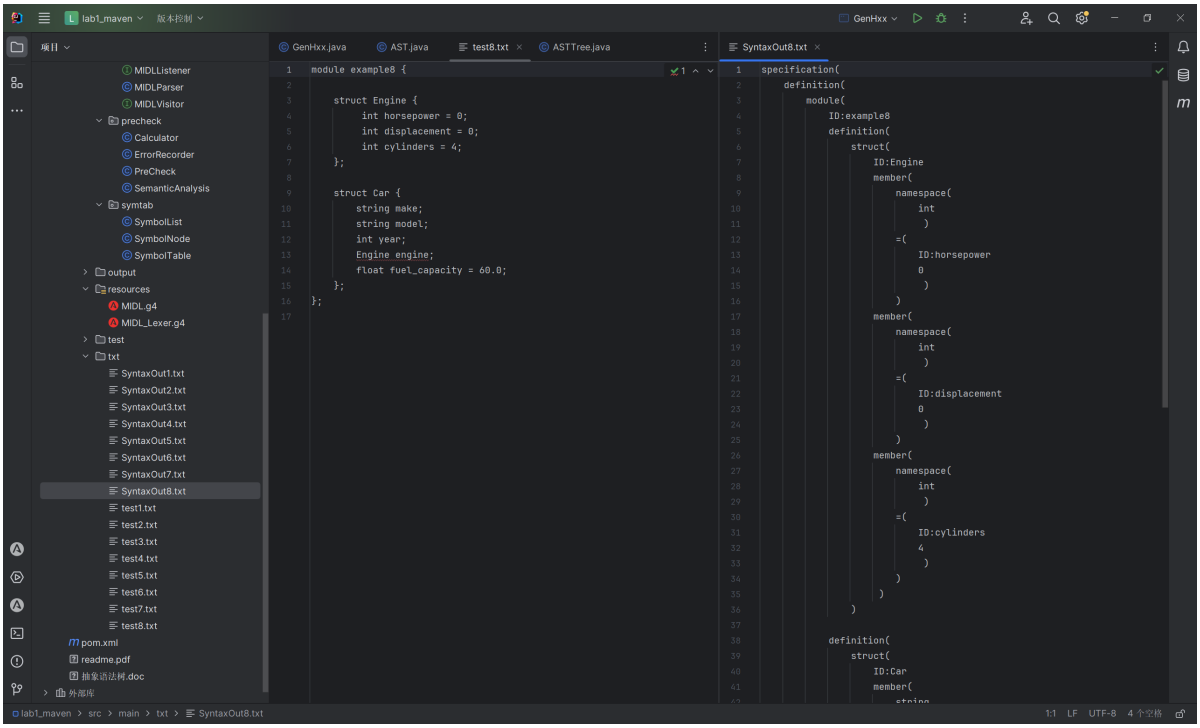
然后是利用Antlr4生成的词法语法分析程序来构建ast，通过继承MIDLBaseVisitor，调用visit方法去访问子树节点，最终将遍历后的树格式化输出。

为制作语言应用，我们必须为每个输入短语或子短语执行一些适当的代码，那样做最简单的方法是操作由语法分析器自动创建的语法分析树。

早些时候我们已经学习了词法分析器处理字符和把记号传递给语法分析器，然后语法分析器分析语法和创建语法分析树的相关知识。对应的ANTLR类分别是CharStream、Lexer、Token、Parser和ParseTree。连接词法分析器和语法分析器的管道被称为TokenStream。

这些ANTLR数据结构分享尽可能多的数据以便节省内存的需要。上图显示在语法分析树中的叶子（记号）节点含有在记号流中记号的点。记号记录开始和结束字符在CharStream中的索引，而不是复制子串。这里没有与空格字符有关的记号，因为我们假设我们的词法分析器扔掉了空格。

给出了具体类型的描述，我们可以手工写代码去执行树的深度优先遍历。当我们发现和完成节点时我们可以执行任何我们想要的动作。典型的操作是诸如计算结果，更新数据结构，或者生成输出。相比每次为每个应用写同样的树遍历样板代码，我们可以使用ANTLR自动生成的树遍历机制。

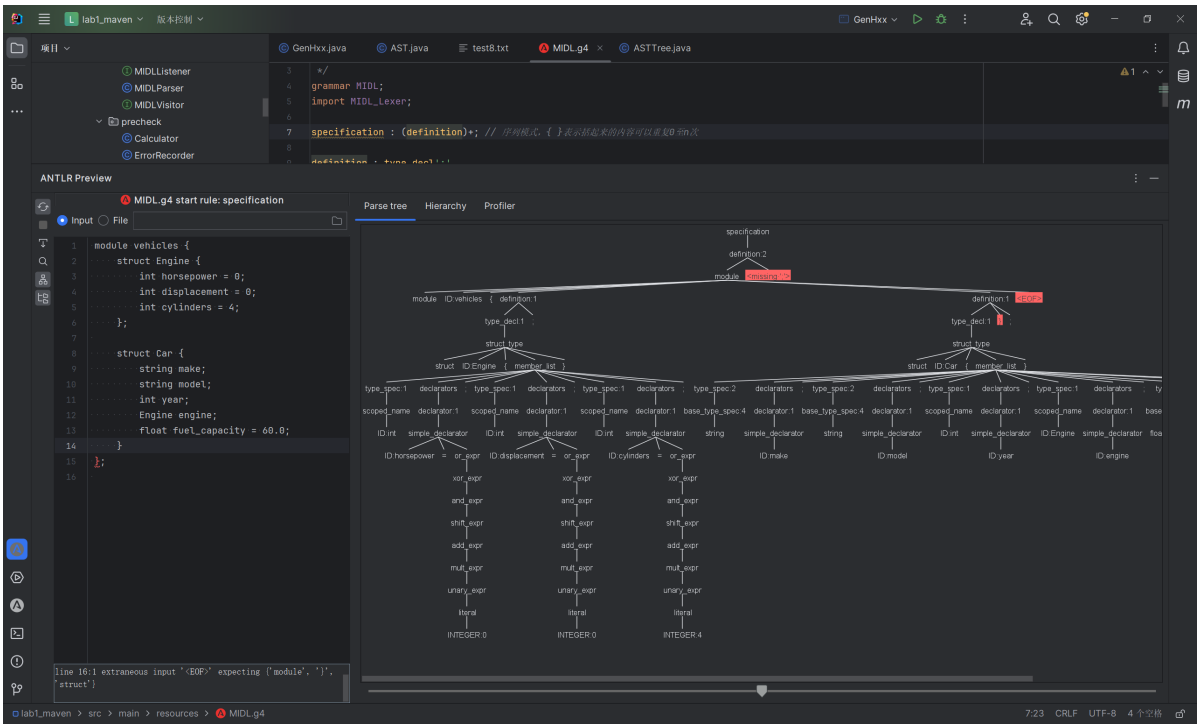


测试说明

根据实验一的要求，测试用例需要自己设计构建，测试的目标有两个：

1. 测试G4中词法，文法是否正确定义。

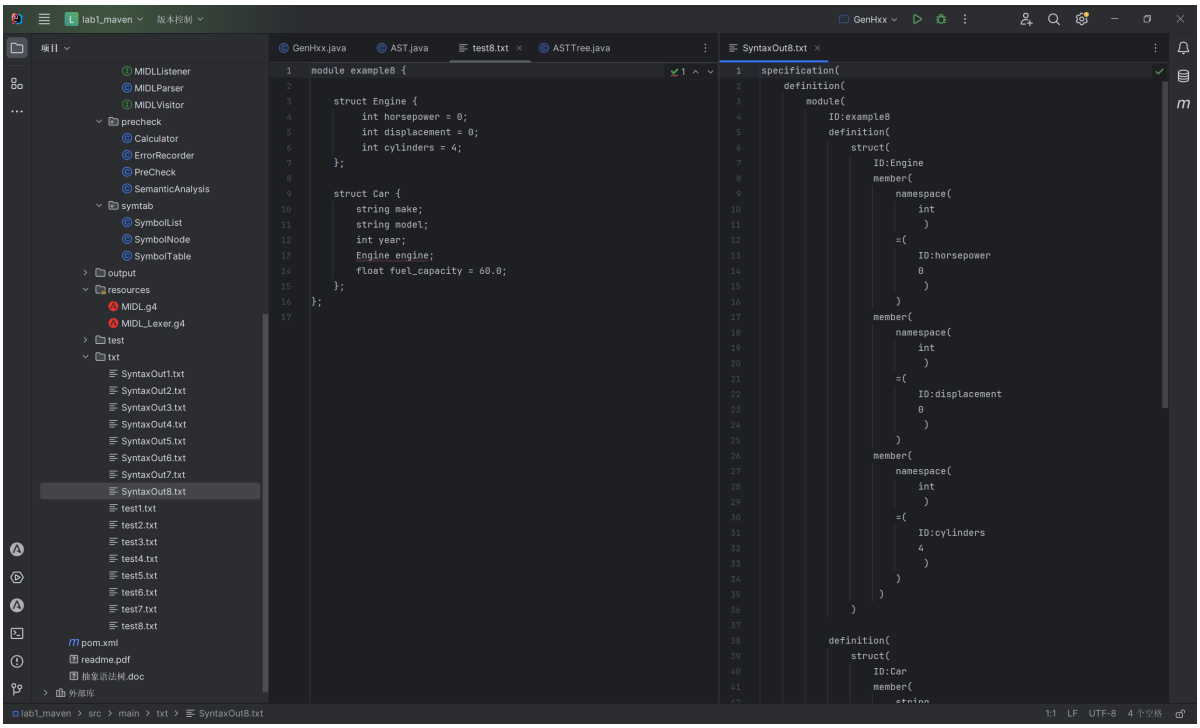
借助IDEA中的ANTLR 4插件来进行测试：在本次测试中，正确测试结果以在上文给出。若将倒数第二个分号删除，插件会进行报错。类似的测试用例还有很多，不再一一展示，因为最后的测试结果都是正确的。



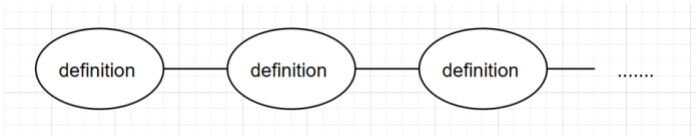
2. 测试设计的抽象语法树是否正确构建，每个测试用例对应一个SyntaxOut.txt。

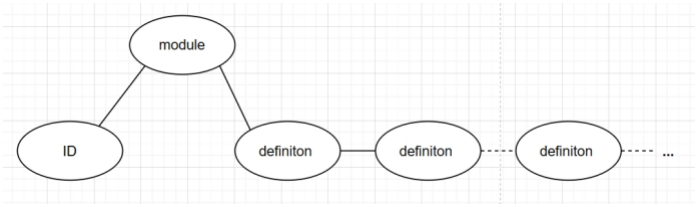
用例编号	用例说明
test1	使用模块和结构体
test2	定义带初始化值的结构体成员
test3	定义多个结构体成员
test4	定义结构体中包含数组
test5	定义std::space命名空间的结构体成员
test6	定义位运算
test7	定义带有位移操作的字段
test8	使用模块和结构体

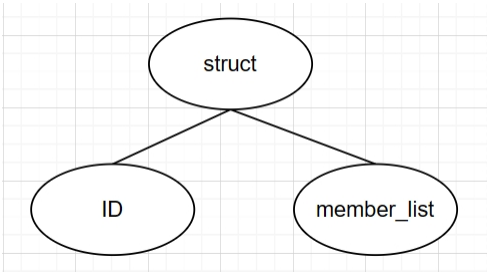
以上测试样例及其输出全保存于./src/main/txt/*，以下以test8为例说明具体测试结果：使用模块和结构体



其中，对应ast文档中的







基于词法语法分析的语义分析

完成了基本的词法语法分析程序后，还需要实现对3条语义错误进行检查和报错。报错提示要给出错误位置，以及错误类型。

这部分主要是根据PPT中提到的构建符号表、扩展符号表功能来完成这三种语义错误的检查的。

测试说明

运行 src/main/java/precheck/PreCheck.java 文件，输入文件地址，比如./src/main/test/test1.txt，即可得到结果。相应的测试文件在 src/main/test 路径下。

命名冲突

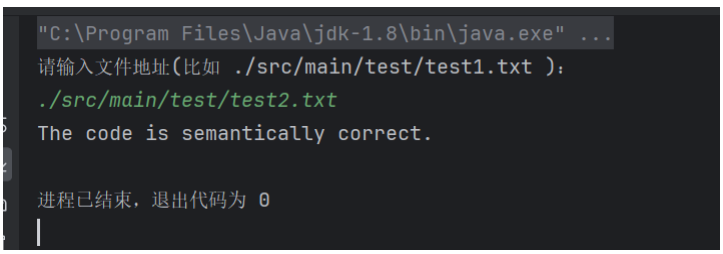
要求同一命名空间内，不能出现相同名字的接口定义。

这一部分主要是利用符号表存储命名空间和其中的结构体、变量等信息。在遍历的时候，如果遇到结构体或变量，那么就根据命名空间和名称构建相应的SymNode对象，插入到符号表中。

```java
    /**
     * 检查命名冲突
     *
     * @param sn sn
     * @return {@link SymbolNode}
     */
    public SymbolNode lookupSt(SymbolNode sn) {
        int hashCode = hash(sn.getName());
        SymbolList sl = symTable[hashCode];
        if (sl == null) {
            return null;
        } else {
            return sl.findNode(sn);
        }
    }
```

如果当前命名空间内已经存在同名的结构体或变量，那么会返回相应的SymNode对象，否则返回null。如果 SymTab.lookup() 返回的结果不为 null ，则表示存在命名冲突， PreCheck 会报告命名冲突的错误。

示例	语义错误
struct A{short num;long num;};	同一个struct空间下，不能有同名变量。
module space{struct A{ short a; };struct B{ short a;};};	同一个module下，不同的struct可以有同名变量。
module space{struct A{ short a; };struct A{ short a;};};	同一个module下，不能出现同名的struct。



```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
请输入文件地址(比如 ./src/main/test/test1.txt ):
./src/main/test/test3.txt
Errors TraceBack:
line 5:4 exception:struct A 命名冲突，重复定义
line 6:14 exception:short a 命名冲突，重复定义


进程已结束，退出代码为 0
```

未定义即使用

要求struct结构需要先定义才能使用。这一部分通过lookupAll( )方法，获取到符号表中与该节点匹配的所有成员变量节点。

```java
/**
 * 查找所有节点，检查未定义即使用的情况
 *
 * @param sn sn
 * @return {@link ArrayList}<{@link SymbolNode}>
 */
public ArrayList<SymbolNode> lookupAll(SymbolNode sn) {
    ArrayList<SymbolNode> ans = new ArrayList<SymbolNode>();
    SymbolList sl;
    for (int i = 0; i < SIZE; i++) {
        ArrayList<SymbolNode> tp;
        sl = symTable[i];
        if (sl != null) {
            tp = sl.findNodes(sn);
            ans.addAll(tp);
        }
    }
    return ans;
}
```

示例	语义错误
struct A{short a;B b;};	B结构应该先定义才能引用类型
module space1{struct B{ ...};};module space2{ struct A{short a;B b;};};	虽然B结构定义了，但是命名空间的引用不对。应该是 space1::B

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
请输入文件地址(比如 ./src/main/test/test1.txt ):
./src/main/test/test4.txt
Errors TraceBack:
line 3:1 exception:struct B 未定义即使用


进程已结束，退出代码为 0
```

字面量类型检查

要求字面量的数据类型需要和变量类型相同或兼容。针对类型进行检查：

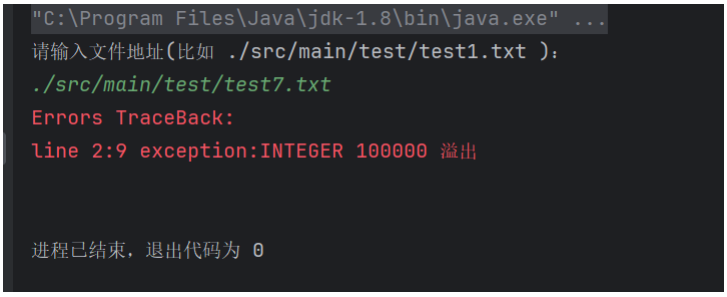
- 如果为整型（ctx.INTEGER() 不为空），且为数组，则检查是否为负数或者是否发生溢出，
- 如果当前节点不是INTEGER，则类型不匹配，如果都没有问题，则语义检测通过；
- 如果是浮点型（ctx.FLOATING_PT() 不为空），进行类型匹配检查；
- 如果是字符型（ctx.CHAR() 不为空），进行类型匹配检查；
- 如果是字符串（ctx.STRING() 不为空），进行类型匹配检查；
- 如果是布尔型（ctx.BOOLEAN() 不为空），进行类型匹配检查。

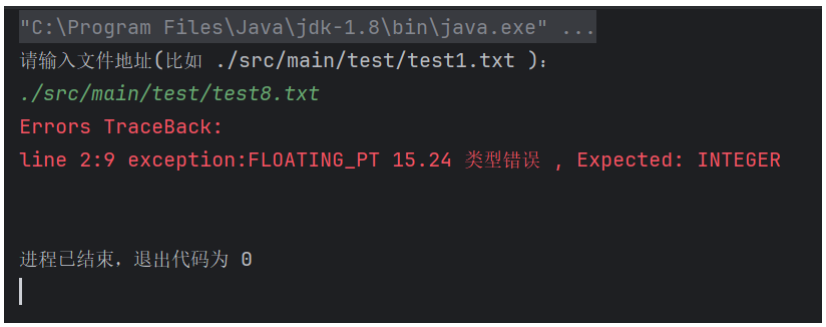
```
    /**
     * literal -> INTEGER | FLOATING_PT | CHAR | STRING | BOOLEAN
     *
     * @param ctx ctx
     * @return {@link String}
     */
    @Override
    public String visitLiteral(MIDLParser.LiteralContext ctx) {

        String checkType = tempNodes.get(tempNodes.size() - 1).getType();
        boolean typeTrue = false;
        if (ctx.INTEGER() != null) {
            //看看是不是在赋值数组下标
            if (tempNodes.get(tempNodes.size() -
1).getType().startsWith("Array")) {
                if (!isPos) {
                    er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "INTEGER", "-" +
ctx.getChild(0).getText(), overflow, null);
                } else if
(Calculator.checkOverFlow(ctx.getChild(0).getText(), isPos))
                    er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "INTEGER",
ctx.getChild(0).getText(), overflow, null);
                else typeTrue = true;
            } else if (!typeLoc(checkType).equals("INTEGER"))
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "INTEGER",
ctx.getChild(0).getText(), typeError, typeLoc(checkType));
            else if (Calculator.checkOverFlow(ctx.getChild(0).getText(),
isPos)) {
                String prefix = "";
                if (!isPos) prefix += "-";
```

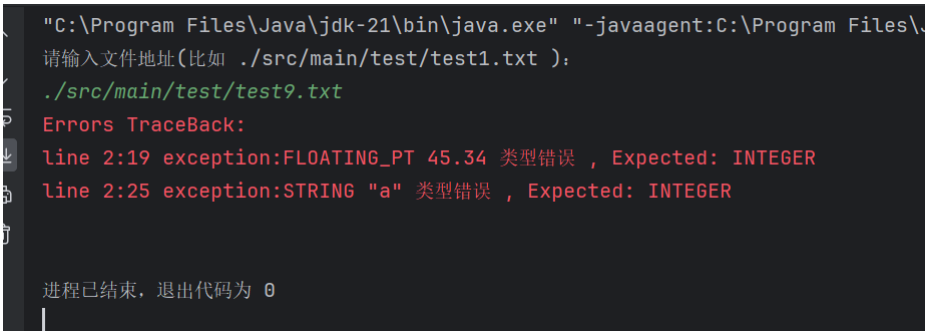
```
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "INTEGER", prefix +
ctx.getChild(0).getText(), overflow, null);
            } else typeTrue = true;
        } else if (ctx.FLOATING_PT() != null) {
            if (!typeLoc(checkType).equals("FLOATING_PT"))
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "FLOATING_PT",
ctx.getChild(0).getText(), typeError, typeLoc(checkType));
            else typeTrue = true;
        } else if (ctx.CHAR() != null) {
            if (!typeLoc(checkType).equals("CHAR"))
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "CHAR", ctx.getChild(0).getText(),
typeError, typeLoc(checkType));
            else typeTrue = true;
        } else if (ctx.STRING() != null) {
            if (!typeLoc(checkType).equals("STRING"))
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "STRING", ctx.getChild(0).getText(),
typeError, typeLoc(checkType));
            else typeTrue = true;
        } else if (ctx.BOOLEAN() != null) {
            if (!typeLoc(checkType).equals("BOOLEAN"))
                er.addError(ctx.getStart().getLine() + ":" +
ctx.getStart().getCharPositionInLine(), "BOOLEAN",
ctx.getChild(0).getText(), typeError, typeLoc(checkType));
            else typeTrue = true;
        }

        //标记这个declaration不要插入符号表
        if (!typeTrue)
            abandoned = true;
        if (typeTrue) {
            //存在~1
            String preVal = tempNodes.get(tempNodes.size() - 1).getVal();
            if (preVal == null)
                tempNodes.get(tempNodes.size() -
1).setVal(ctx.getChild(0).getText());
            else {
                tempNodes.get(tempNodes.size() - 1).setVal(preVal +
ctx.getChild(0).getText());
            }
        }
        return null;
    }
```

示例	语义错误
struct A{short a='a';};	a是整型变量，字面量却是字符类型
struct A{short a=100000;};	short为有符号短整型，最大值不超过 $2^{15}-1$
struct A{short a=15.24;};	a是整型变量，字面量却是浮点类型

示例	语义错误
struct A{short a[4]=[10,12,45.34,'a'];};	a是整型数组，数组字面量里必须保证数据类型的统一







基于词法语法分析的代码生成

实验内容是从抽象语法树生成对应的C++代码，测试用例参考“代码生成用例集.zip”，依据用例集中的输入输出，完成代码生成模块。

这部分主要是看了实验文件夹下的 JAVA中调用ST模板引擎演示视频 和 StringTemplate模板语法介绍 这两部分内容，学习了StringTemplate的C++模板的定义。根据给出的五个case生成的hxx文件，定义了如下的模板：

```
headerTemplate(fileName)
::= <<
/*
WARNING: THIS FILE IS AUTO-GENERATED. DO NOT MODIFY.

This file was generated from <fileName>.idl using "idltoc".
```

```
The idltoc tool is part of the RTI Data Distribution Service distribution.
For more information, type 'idltoc -help' at a command shell
or consult the RTI Data Distribution Service manual.
*/

#ifndef <fileName>_h
#define <fileName>_h

#ifndef rti_me_cpp_hxx
#include "rti_me_cpp.hxx"
#endif

#ifdef NDDS_USER_DLL_EXPORT
#if (defined(RTI_WIN32) || defined(RTI_WINCE))
/* If the code is building on Windows, start exporting symbols. */
#undef NDDSUSERDllExport
#define NDDSUSERDllExport __declspec(dllexport)
#endif
#else
#undef NDDSUSERDllExport
#define NDDSUSERDllExport
#endif
>>

bodyTemplate(structs) ::= <<
<structs:{item|<structTemplate(item)>};separator="\n">
>>

structTemplate(struct) ::= <<

struct <struct.name>Seq;
class <struct.name>TypeSupport;
class <struct.name>DataWriter;
class <struct.name>DataReader;

class <struct.name>
{
public:
    typedef struct <struct.name>Seq Seq;
    typedef <struct.name>TypeSupport TypeSupport;
    typedef <struct.name>DataWriter DataWriter;
    typedef <struct.name>DataReader DataReader;

    <struct.members:{item|CDR_<item.type> <item.name><if(item.isArray)>
[<item.valNum>]<endif><if((!struct.hasScoped)&&(item.hasVal))> = <item.val>
<endif>;};separator="\n">
    <if(struct.hasScoped)><struct.scopeMembers:{item|<item.type>
<item.name>;};separator="\n"><endif>
};

extern const char *<struct.name>TYPENAME;

REDA_DEFINE_SEQUENCE_STRUCT(<struct.name>Seq, <struct.name>);

REDA_DEFINE_SEQUENCE_IN_C(<struct.name>Seq, <struct.name>);

```

```
<if(struct.hasScoped)>

NDDSUSERDllExport extern <struct.name>()
{
    <struct.members:{item|this-\><item.name><if(item.hasVal)> = <item.val>
<endif>;};separator="\n">
    <struct.scopeMembers:{item|
<scopedmemberTemplate(item)>};separator=";\n">
}

<endif>

NDDSUSERDllExport extern RTI_BOOL
<struct.name>_initialize(<struct.name> *sample)
{
    <struct.members:{item|
    <if((!item.isArray)&&(!item.isString))>
CDR_Primitive_init_<item.type>(&sample-\><item.name>);
    <elseif((item.isString)&&(!item.isArray))>
    if (!CDR_String_initialize(&sample-\><item.name>, (255)))
    {
        return RTI_FALSE;
    \}
    <elseif((item.isArray)&&(!item.isString))>
CDR_Primitive_init_Array(
            sample-\><item.name>, ((<item.valNum>) *
CDR_<item.upperType>_SIZE));
    <elseif((item.isArray)&&(item.isString))>
CDR_Primitive_init_Array(
                sample-\><item.name>, ((<item.valNum>) *
CDR_<item.upperType>_SIZE));
    {
        for(RTI_UINT32 i = 0; i \< (<item.valNum>); i++)
        {
            if (!CDR_String_initialize(&sample-\><item.name>[i], (255)))
            {
                return RTI_FALSE;
            \}
        \}
    \}
    <endif>
    };separator="\n">
    <if(struct.hasScoped)>
    <struct.scopeMembers:{item|
    if (!<item.type>_initialize(&sample-\><item.name>))
    {
        return RTI_FALSE;
    \}
    };separator="\n">
    <endif>
    return RTI_TRUE;
}

NDDSUSERDllExport extern RTI_BOOL
<struct.name>_finalize(<struct.name> *sample)
{
```

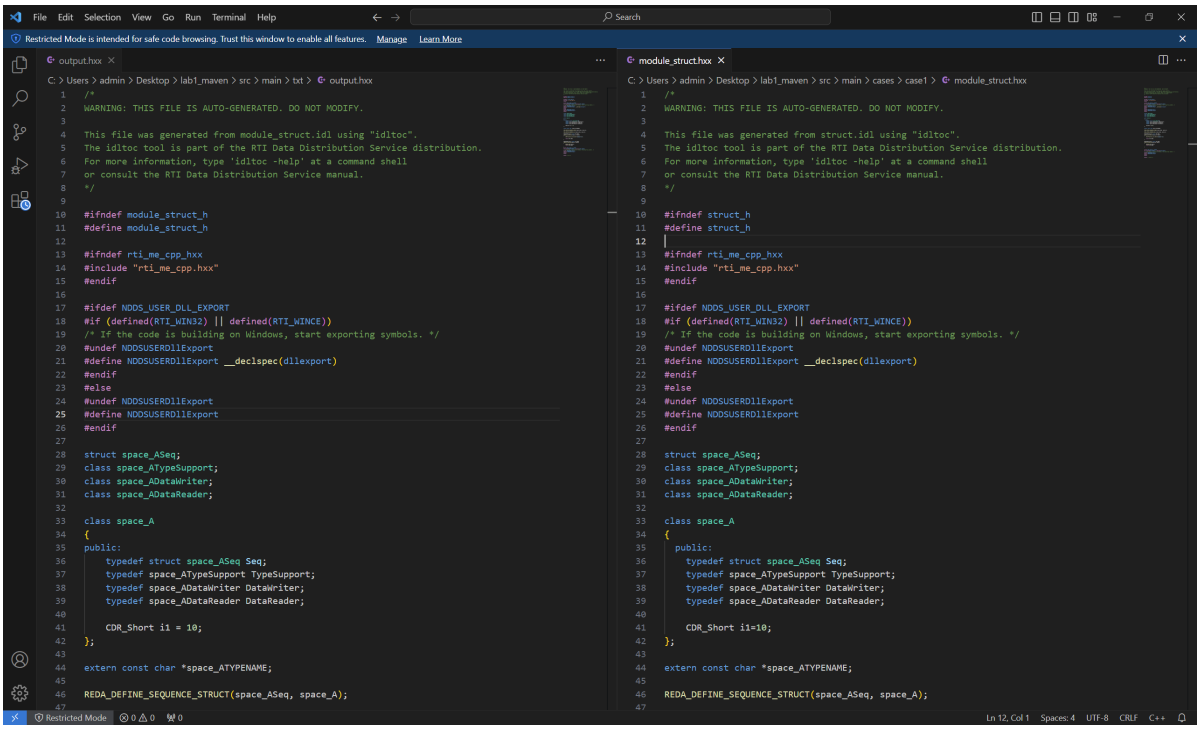
```
    UNUSED_ARG(sample);
    <if(struct.hasScoped)>
    <struct.scopeMembers:{item|
<item.type>_finalize(&sample-\><item.name>);
    };separator="\n">
    <endif>

    <struct.members:{item|
    <if((item.isString)&&(!item.isArray))>
    CDR_String_finalize(&sample-\><item.name>);
    <elseif((item.isArray)&&(!item.isString))>
    {
            RTI_UINT32 i;

            for (i = 0; i \< (<item.valNum>); i++)
            {
                if (!CDR_<item.type>_copy(&dst-\><item.name>[i],
                                &src-\><item.name>[i]))
                {
                    return RTI_FALSE;
                \}
            \}
    \}
    <elseif((item.isArray)&&(item.isString))>
    {
            RTI_UINT32 i;

            for (i = 0; i \< (<item.valNum>); i++)
            {
                CDR_String_finalize(&sample-\><item.name>[i]);
                if (!CDR_String_copy(&dst-\><item.name>[i], &src-\>
<item.name>[i]))
                {
                    return RTI_FALSE;
                \}
            \}
    \}
    <endif>
    };separator="\n">
    return RTI_TRUE;

\}
>>

footerTemplate(fileName)
::= <<
#ifdef NDDS_USER_DLL_EXPORT
#if (defined(RTI_WIN32) || defined(RTI_WINCE))
/* If the code is building on Windows, stop exporting symbols. */
#undef NDDSUSERDllExport
#define NDDSUSERDllExport
#endif
#endif

#endif /* <fileName> */
>>
```

```
167
168 scopedmemberTemplate(scopedmember)
169 ::= <<
170     <scopedmember.members:{item|this-\><scopedmember.name>.<item.name>
    <if(item.hasVal)> = <item.val><endif>;};separator="\n">
171 >>
```

测试说明

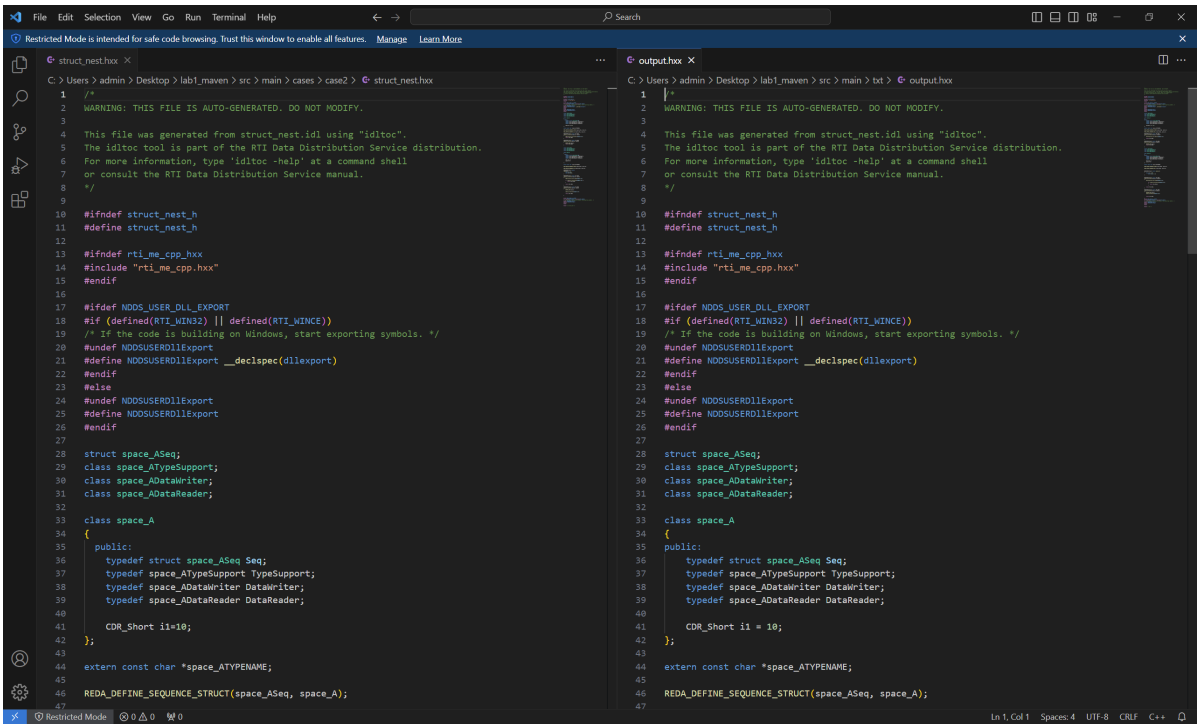
运行 src/main/java/backGen/GenHxx.java 文件，输入文件地址，比如./src/main/cases/case1/module_struct.idl，即可得到结果。相应的测试文件在 src/main/cases 路径下：

```
module space{
    struct A{
        short i1=10;
    };
};
```



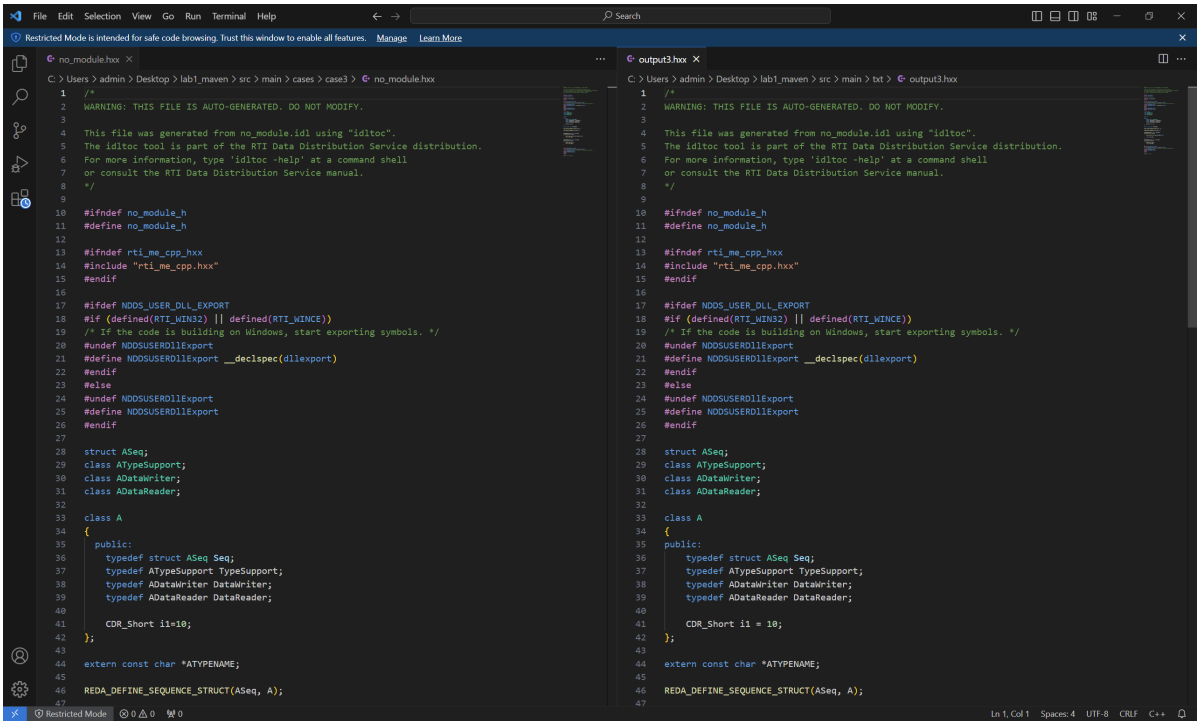
case2:

```
module space{
    struct A{
        short i1=10;
    };
    struct B{
        long i2=100;
        A i3;
    };
};
```



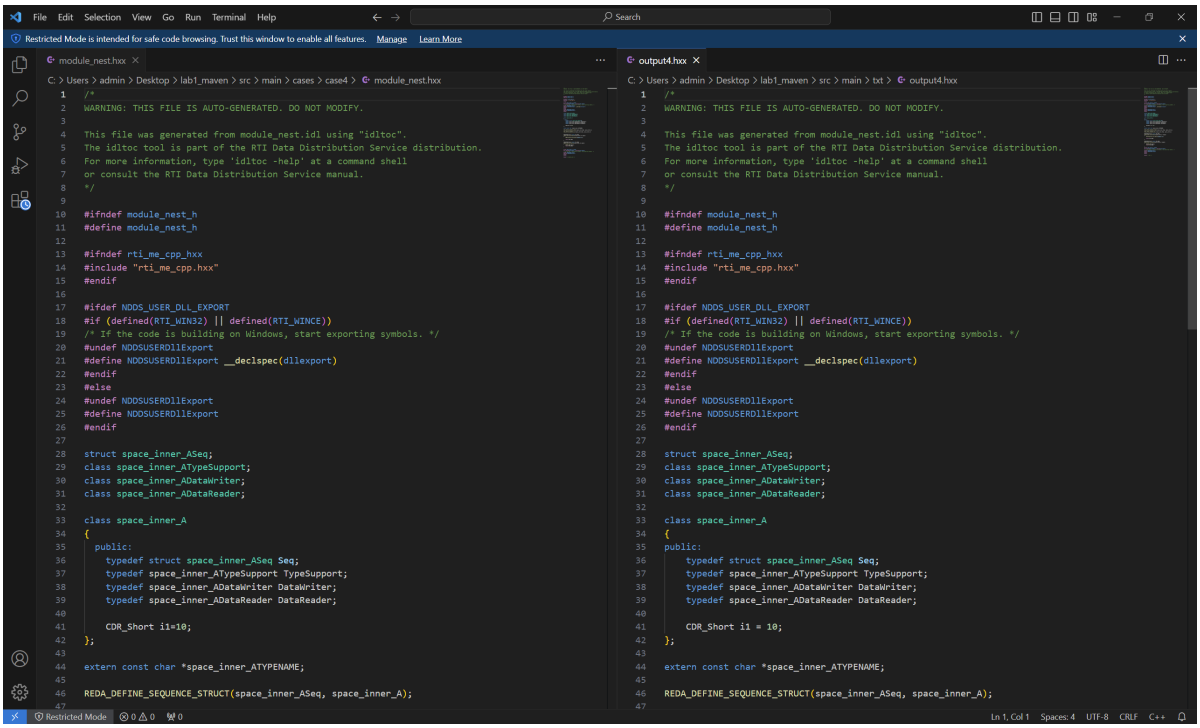
case3:

```
struct A{
    short i1=10;
};
```



case4:

```
module space{
    module inner{
        struct A{
            short i1=10;
        };
    };
};
```



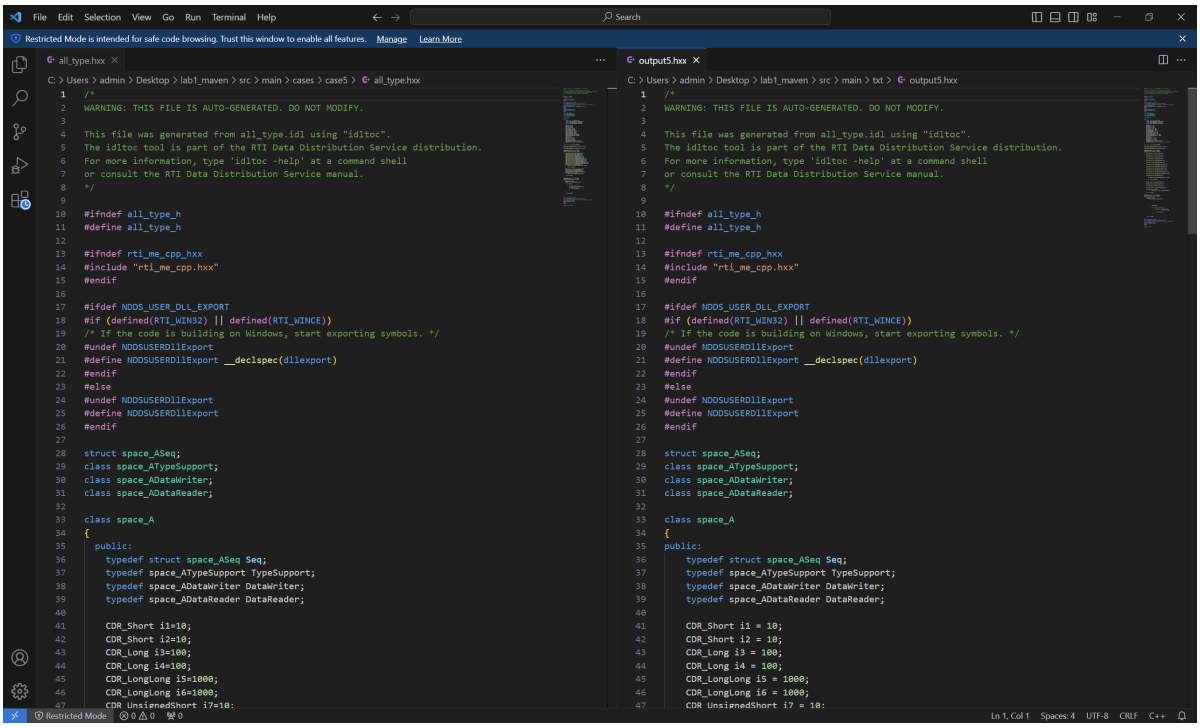
case5:

```
module space{
    struct A{
        short i1=10;
        int16 i2=10;
        long i3=100;
        int32 i4=100;
        long long i5=1000;
        int64 i6=1000;
        unsigned short i7=10;
        uint16 i8=10;
        unsigned long i9=100;
        uint32 i10=100;
        unsigned long long i11=1000;
        uint64 i12=1000;
        char c0='a';
        string c1="abc";
        boolean c2=true;
        float c3=10.901f;
        double c4=23.234d;
        long double c5=12.23456432235d;
        short arr[10]=[0,1,2,3,4,5,6,7,8,9];
    };
};
```

其中，这里出现的一个问题是case5并不能从idl正确生成hxx文件，会报如下错误：

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
请输入文件地址(比如 ./src/main/cases/case1/module_struct.idl ):
./src/main/cases/case5/all_type.idl
line 19:20 missing '}' at ';'
line 20:2 extraneous input 'long double' expecting {'module', '}', 'struct'}
The code is semantically correct.
输出结果已保存到.\src\main\txt\output.hxx

进程已结束，退出代码为 0
```

因为在idl文件的声明中就有错误，这里c4 c5后面有两个;;，同时arr数组的声明应该是[]而不是{}，修改过后可以正常生成。



实验总结

1. 词法分析和语法分析

在实验中，我使用Antlr4开源工具进行词法分析和语法分析。首先，创建了一个`.g4`文件，定义了MIDL语言的词法和语法规则。通过Antlr4工具生成了相应的词法分析器和语法分析器。这部分工作帮助我深入理解了编译原理中的词法分析和语法分析过程，尤其是如何通过正则表达式定义词法规则，如何通过BNF（巴科斯-瑙尔范式）定义语法规则。

2. 抽象语法树的生成

生成词法分析器和语法分析器后，我编写了一个Java程序，调用生成的词法和语法分析器，从而生成MIDL源代码的抽象语法树（AST）。这一步让我学会了如何使用Antlr4工具解析代码并生成语法树，理解了语法树的结构及其在编译过程中的作用。

3. 语义分析

在生成抽象语法树后，进行语义分析是下一步的重要工作。语义分析主要包括符号表的构建和类型检查。通过这一步，我学会了如何设计符号表以及如何在语法树遍历过程中填充符号表，并进行类型检查以确保源代码的语义正确性。

4. C++代码生成

在语义分析的基础上，进行了C++代码生成。C++代码生成器遍历语法树，根据语法树节点生成对应的C++代码。这一步让我理解了源到源编译器的工作原理，即如何将一种语言的源代码转换为另一种语言的源代码。

实验心得

1. Antlr4工具的学习与应用

通过本次实验，我系统地学习了Antlr4工具的使用方法。Antlr4作为一个功能强大的解析器生成工具，其丰富的特性和简洁的语法让我能够快速定义词法和语法规则，并生成对应的解析器。此外，Antlr4生成的Java代码让我能够方便地集成到我的项目中，极大地提高了开发效率。

2. 编译原理知识的实践应用

在实验中，我将编译原理课程中学到的词法分析、语法分析、语义分析等知识应用到实际项目中。通过实践，我对这些理论知识有了更深入的理解，并且学会了如何在实际项目中应用这些知识。

3. 调试与问题解决

在实验过程中，我遇到了一些问题，如词法规则和语法规则冲突、语法树生成错误等。通过调试和查阅资料，我解决了这些问题。这让我学会了如何在复杂的项目中进行调试和问题解决，增强了我的编程能力和解决问题的能力。

通过本次实验，我不仅掌握了Antlr4工具的使用方法，还深入理解了编译器的各个阶段及其实现方法。从MIDL语言的词法分析、语法分析到抽象语法树生成，再到语义分析和C++代码生成，每一个环节都让我对编译原理有了更深的认识。这次实验不仅巩固了我的理论知识，还提升了我的实际编程能力和项目开发能力。总的来说，这是一次非常有意义的实践经历，为我今后的学习和工作打下了坚实的基础。