实验报告

1. Antlr环境安装

1. 下载Antlr的jar包。它作为一个Java程序，需要正确配置Java环境后才能运行。
2. 由于使用的是.Net，前往Nuget包管理器下载C#的Antlr运行时依赖包。

2. 语法文件.g4编写

根据语法文件.g4的语法，写出产生式。

```
grammar MIDL;

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

fragment LETTER
    : [a-z] | [A-Z]
    ;

fragment DIGIT
    : [0-9]
    ;

fragment UNDERLINE
    : '_'
    ;

fragment INTEGER_TYPE_SUFFIX
    : 'l'
    | 'L'
    ;

fragment ESCAPE_SEQUENCE
    :  '\\' [btnfr"'\\]
    ;

fragment EXPONENT
    : ( 'e' | 'E' ) ( '+' | '-' )? [0-9]+
    ;

fragment FLOAT_TYPE_SUFFIX
    : 'f'
    | 'F'
    | 'd'
    | 'D' ;

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
    ;

INTEGER
    : ('0' | [1-9] [0-9]* ) INTEGER_TYPE_SUFFIX?
    ;

BOOLEAN
    : 'TRUE'
    | 'true'
    | 'FALSE'
    | 'false'
    ;

CHAR
    : '\'' (ESCAPE_SEQUENCE |  (~'\\' | ~'\'') ) '\''
    ;

STRING
    : '"' (ESCAPE_SEQUENCE |  (~'\\' | ~'"') )*? '"'
    ;

ID
    : LETTER ( UNDERLINE? ( LETTER | DIGIT ) ) *
    ;

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'
    | '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
    ;
```

其中，**Specification**之前的是词法匹配，而之后的是语法匹配。

通过这样一份既符合.g4文件语法，也符合目标语法要求的文件，可以使用Antlr工具，生成对应的词法分析程序和语法分析程序。

```
    $ cd ~/CompilePrinciple_Ex1/CompilePrinciple_Ex1
    $ java -jar ./dependencies/antlr-4.13.1-complete.jar -Dlanguage=CSharp -visitor
./src/MIDL.g4
```

执行以上命令后，就会生成程序代码。这里，**-Dlanguage**指明了要生成的目标语言。在这里我选用了C#。**-visitor**则说明产生Visitor类，来方便的对语法分析树进行访问。

以下是生成的文件列表：

3. 语法分析树生成

通过命名空间**Antlr4.Runtime**和**Antlr4.Runtime.Tree**，可以很快的由给定的文本，生成一棵语法分析树。

```
        static void Main(string[] args)
        {
            string content = File.ReadAllText(MIDLInputFilePath);
            ICharStream stream = CharStreams.fromString(content);
            ITokenSource lexer = new MIDLLexer(stream);
            ITokenStream tokens = new CommonTokenStream(lexer);
            MIDLParser parser = new MIDLParser(tokens);
            IParseTree tree = parser.specification();
        }
```

这里生成的**IParseTree**类型的变量**tree**即保存着语法分析树。

例如，对于以下IDL文件：

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

会生成以下的语法分析树（缩进代表树的层次）：

```
specification
  definition
    module
      'module'
      'space'
      '{'
      definition
        type_decl
          struct_type
            'struct'
            'A'
            '{'
            member_list
              type_spec
                base_type_spec
                  integer_type
                    signed_int
                      'short'
              declarators
                declarator
                  simple_declarator
                    'i1'
                    '='
                    or_expr
                      xor_expr
                        and_expr
                          shift_expr
                            add_expr
                              mult_expr
                                unary_expr
                                  literal
                                    '10'
              ';'
            '}'
        ';'
      '}'
    ';'
```

4. 抽象语法树生成

由于语法分析树含有很多冗余信息，因此对其简化，删除不必要的结点。

在抽象语法树中，只有7种结点：

- **Specification**：起始结点，代表抽象语法树的根。它的孩子结点是**Struct**结点或者**Module**结点，代表顶级的声明域。
- **Struct**：结构体结点。一个这种结点声明了一个结构体。它的孩子结点**Member**结点，是该结构体结点内声明的所有成员。
- **Module**：模块结点。一个这种结点声明了一个模块。它的孩子结点是**Struct**结点或者**Module**结点，代表在该模块结点内的声明域。
- **Member**：成员结点。一个这种结点声明了一次变量。它只对变量的类型作出规定。它的孩子结点是**Declarator**结点，在**Declarator**结点中，会记录声明的变量名。
- **Declarator**：变量结点。一个这种结点，代表实际声明的一个变量。它记录了变量的名字，以及是否为一个数组。如果这个变量是普通变量，当该变量没有初始值的时候，它没有孩子结点；否则它的第一个孩子为**Expression**结点，该结点代表变量的初始值。如果这个变量是一个数组变量，它的第一个孩子是**Expression**结点，该结点代表数组的长度；余下的孩子也均为**Expreesion**结点，依次为数组内元素从0到N的初始化值。
- **Expression**：表达式结点。一个这种结点，代表一个表达式的值。它的孩子都是**Expression**结点。它保存着运算符的类型，如果该运算符是一元运算符，则第一个孩子结点就代表着，将此结点的运算符运用到第一个孩子结点的值上。如果该运算符是二元运算符，则第一、二个孩子结点就代表着，将此节点的运算符运用在第一个和第二个孩子结点的值之间。
- **Literal**：值结点。它是特殊的表达式结点。它不会有孩子，它保存一个常量值，即是其作为表达式结点时的值。通过作为**Expression**结点的孩子，可以使其进入计算。

以以下的IDL文件为例：

```
module space{
  struct A{
    short i1=10;
    short i2=10+5*2;
  };
};
```

最后生成的抽象语法树为：

```
Specification                 // Specification结点
  Module_space                // Module结点
    Struct_A                  // Struct结点
      Member_Type(short)      // Member结点
        Variable_i1           // Declarator结点
          (Integer):10        // Literal结点
      Member_Type(short)      // Member结点
        Variable_i2           // Declarator结点
          +                   // Expression结点
            (Integer):10      // Literal结点
            *                 // Expression结点
              (Integer):5     // Literal结点
              (Integer):2     // Literal结点
```

5. 语义分析

5.1 符号表数据结构

使用了一个树状的符号表，可以得知在不同声明域声明的不同符号信息。

符号表之间树状连接，模拟了变量和类型声明在不同声明域的情况。每一个符号表，就代表一个声明域。

一个符号表中，有一个哈希表（Identifiers），用于存储在本声明域中声明的所有标识符，并为其标上声明符的类型（模块、结构体、变量声明）。

还有一个哈希表（ChildScopes），用于存储子声明域。

例如，在Module A中声明了Module B的话，在模块A这个声明域的符号表中：

- Identifier中有一个键值对<B，Module>
- ChildScopes中有一个键值对<B, MIDLScope>，值是模块B这个声明域的符号表。

同时，这个符号表还支持了寻找标识符的功能。给定一个域名，该域名之间的不同层级的声明域由“::”隔开，便可查找在当前域中是否能找到该标识符。

```
using System.Collections.Generic;

public enum IdentifierType
{
    All,
    Module,
    Struct,
    Declaration,
}

public class MIDLScope
{
    public string Name { get; set; } = "<default_scope>";
    public IDictionary<string, IdentifierType> Identifiers { get; private set; } = new
Dictionary<string,IdentifierType>();
    public MIDLScope Parent { get; set; }
    public IDictionary<string, MIDLScope> ChildScopes { get; private set; } = new
Dictionary<string, MIDLScope>();
    public string GetScopePrefix()
    {
        return Parent == null ? "" : Parent.GetScopePrefix() + $"{Name}::";
    }

    private bool SearchScopeNameInternal(string[] scopes, int startIndex, IdentifierType
type)
    {
        if (startIndex == scopes.Length - 1)
        {
            string identifier = scopes[startIndex];
            if (!Identifiers.ContainsKey(identifier))
                return false;
```

```
            return type == IdentifierType.All ? true : Identifiers[identifier] == type;
        }
        string scope = scopes[startIndex];
        if (ChildScopes.ContainsKey(scope))
            return ChildScopes[scope].SearchScopeNameInternal(scopes, startIndex + 1,
type);
        else
            return false;
    }

    public bool SearchScopeName(string scopeName, IdentifierType type =
IdentifierType.All)
    {
        string[] scopes = scopeName.Split("::");
        return SearchScopeNameInternal(scopes, 0, type)
            || Parent == null ? false : Parent.SearchScopeNameInternal(scopes, 0, type);
    }

    public override string ToString() => $"Scope:{Name}";
}
```

通过这样的符号表，就可以快速检查在声明变量时，变量类型是否已有定义。同时，在声明一个新的变量或作用域时，可以查看当前声明域是否已有同名的变量。

5.2 变量初始化值类型匹配

对于变量的初始化值，在抽象语法树中，只需要获得**Declarator**结点的孩子结点中的**Expreesion**结点，计算该结点的值的变量类型是否与**Declarator**的父亲结点——**Member**结点相匹配。

对于**Expression**结点的值的变量类型，可以递归地，根据运算符以及操作数的值的变量类型，进行更细致的变量类型运算。在本次语义分析中，为了简便，仅允许相同类型的变量可以互相运算，且运算结果等于原来的变量类型。

5.3 变量初始化值值匹配

对于不同的变量类型，允许的初始化值也不同。

而由于只有基本类型（非结构体）的变量能初始化，所以对于这些基本类型的变量都实现了一个转换函数，判断给定的文本值转换为基本类型后，会不会超出初始化的值。

5.4 拼装在一起

首先，对抽象语法树进行扫描，生成一个树状的符号表结构。

第一次扫描仅对声明域进行扫描，建立符号表之间的层级结构。此时，该符号表内仅含有**Struct**类型和**Module**类型的标识符。借此，该符号表来寻找所有可用的自定义结构体类型。

```
    private MIDLScope SearchScope(ASTNode tree)
    {
        Stack<(MIDLScope ParentScope, ASTNode Node)> stack = new Stack<(MIDLScope
ParentScope, ASTNode Node)>();
```

```
        MIDLScope rootScope = new MIDLScope();
        for (int i = tree.Childs.Count - 1; i >= 0; i--)
            stack.Push((rootScope, tree.Childs[i]));
        while (stack.Count > 0)
        {
            (MIDLScope parentScope, ASTNode node) = stack.Pop();
            if (node is ASTNode.Module module)
            {
                if (parentScope.Identifiers.ContainsKey(module.ID))
                {
                    Print(module.Start.Line, module.Start.Column,
                        $"The identifier \"{module.ID}\" is already defined in scope \"
{parentScope.Name}\"."
                        );
                    continue;
                }
                parentScope.Identifiers.Add(module.ID, IdentifierType.Module);
                var moduleScope = new MIDLScope() { Name = module.ID };
                parentScope.ChildScopes.Add(module.ID, moduleScope);
                moduleScope.Parent = parentScope;

                for (int i = node.Childs.Count - 1; i >= 0; i--)
                    stack.Push((moduleScope, node.Childs[i]));
            }
            else if (node is ASTNode.Struct @struct)
            {
                if (parentScope.Identifiers.ContainsKey(@struct.ID))
                {
                    Print(@struct.Start.Line, @struct.Start.Column,
                        $"The identifier \"{@struct.ID}\" is already defined in scope \"
{parentScope.Name}\"."
                        );
                    continue;
                }
                parentScope.Identifiers.Add(@struct.ID, IdentifierType.Struct);
                var typeName = parentScope.GetScopePrefix() + @struct.ID;
                Types.Add(new SystemType.Custom(typeName));
                var structScope = new MIDLScope() { Name = @struct.ID };
                parentScope.ChildScopes.Add(@struct.ID, structScope);
                structScope.Parent = parentScope;

                for (int i = node.Childs.Count - 1; i >= 0; i--)
                    stack.Push((structScope, node.Childs[i]));
            }
        }
        return rootScope;
    }
```

随后，再对抽象语法树进行第二次扫描。此次扫描关注的是变量的声明，并将声明的变量标识符添加到已有的符号表中。在扫描时，对其初始化值的变量类型以及初始化值的值进行分析。

```
  private void SearchDeclaration(MIDLScope root, ASTNode tree)
    {
        Stack<(MIDLScope ParentScope, ASTNode Node)> stack = new Stack<(MIDLScope
ParentScope, ASTNode Node)>();
        for (int i = tree.Childs.Count - 1; i >= 0; i--)
            stack.Push((root, tree.Childs[i]));
        while (stack.Count > 0)
        {
            (MIDLScope parentScope, ASTNode node) = stack.Pop();
            if (node is ASTNode.Scope scope)
            {
                var tmp = parentScope.ChildScopes[scope.ID];
                for (int i = node.Childs.Count - 1; i >= 0; i--)
                    stack.Push((tmp, node.Childs[i]));
            }
            else if (node is ASTNode.Member member)
            {
                var type = GetType(member.TypeText);
                if(type == null && !parentScope.SearchScopeName(member.TypeText,
IdentifierType.Struct))
                {
                    Print(member.Start.Line, member.Start.Column,
                        $"Type \"{member.TypeText}\" is not defined yet."
                        );
                    continue;
                }
                foreach(var child in member.Childs)
                {
                    var declarator = (ASTNode.Declarator)child;
                    if (parentScope.Identifiers.ContainsKey(declarator.ID))
                    {
                        Print(declarator.Start.Line, declarator.Start.Column,
                            $"The identifier \"{declarator.ID}\" is already defined in
scope \"{parentScope.Name}\"."
                            );
                        continue;
                    }
                    parentScope.Identifiers.Add(declarator.ID,
IdentifierType.Declaration);


                    if (!declarator.IsArray)
                    {
                        if (declarator.Childs.Count == 0)
                            continue;

                        ASTNode.Expression expression =
(ASTNode.Expression)declarator.Childs[0];
                        if (!type.Accept(expression.Type))
                        {
                            Print(expression.Start.Line, expression.Start.Column,
```

```
                                $"Constant type \"{expression.Type}\" cannot be assigned
to type \"{member.TypeText}\"."
                                );
                            continue;
                        }
                        else if(expression is ASTNode.Literal literal &&
!type.Accept(literal.Text))
                        {
                            Print(expression.Start.Line, expression.Start.Column,
                                $"Value \"{literal.Text}\" cannot be assigned to type \"
{member.TypeText}\". ");
                            continue;
                        }
                    }
                    else
                    {
                        ASTNode.Expression expression =
(ASTNode.Expression)declarator.Childs[0];
                        if (expression.Type != ConstantType.Integer)
                        {
                            Print(expression.Start.Line, expression.Start.Column,
                                $"The array length must be an integer number. \"
{expression.Type}\" is provided."
                                );
                            continue;
                        }
                        for (int i = 1; i < declarator.Childs.Count; i++)
                        {
                            expression = (ASTNode.Expression)declarator.Childs[i];
                            if (!type.Accept(expression.Type))
                            {
                                Print(expression.Start.Line, expression.Start.Column,
                                    $"Constant type \"{expression.Type}\" cannot be
assigned to type \"{member.TypeText}\"."
                                    );
                                continue;
                            }
                            else if (expression is ASTNode.Literal literal &&
!type.Accept(literal.Text))
                            {
                                Print(expression.Start.Line, expression.Start.Column,
                                    $"Value \"{literal.Text}\" cannot be assigned to type
\"{member.TypeText}\".");
                                continue;
                            }
                        }
                    }
                }
            }
        }
    }
```

6. C++ 代码生成

对于抽象语法树的每个结点，都实现一个函数`ToCppCode(int level)`。其中`level`代表的是自顶向下声明域的层级，以方便控制缩进。递归的调用该函数，并注意IDL类型到C++的类型转换，就能写出下面的代码。

```
    public static string Indent(int level)
    {
        string ans = "";
        for (int i = 0; i < level; i++)
            ans += '\t';
        return ans;
    }
    public class Specification : ASTNode
    {
        public override string ToCppCode(int level)
        {
            string ans = "";
            ans += "typedef const char* string;\n";
            foreach (var child in Childs)
                ans += child.ToCppCode(level) + "\n";
            return ans;
        }
    }

    public abstract class Scope : ASTNode
    {
        public string ID { get; set; }
    }

    public class Struct : Scope
    {
        public override string ToCppCode(int level)
        {
            string indent = Indent(level);
            string ans = $"{indent}typedef struct {ID}\n{indent}{{\n";
            foreach (var child in Childs)
                ans += child.ToCppCode(level + 1);
            ans += $"{indent}}}{ID};\n";
            return ans;
        }
    }

    public class Module : Scope
    {
        public override string ToString() => $"Module_{ID}[{Start.Line}:{Start.Column}~
{Stop.Line}:{Stop.Column}]";
        public override string ToCppCode(int level)
        {
            string indent = Indent(level);
            string ans = $"{indent}namespace {ID}\n{indent}{{\n";
            foreach (var child in Childs)
```

```
                ans += child.ToCppCode(level + 1);
            ans += $"{indent}}}\n";
            return ans;
        }
    }

    public class Member : ASTNode
    {
        public string TypeText { get; set; }
        public override string ToString() => $"Member_Type({TypeText})[{Start.Line}:
{Start.Column}~{Stop.Line}:{Stop.Column}]";
        public override string ToCppCode(int level)
        {
            string ans = $"{Indent(level)}{CppTypeText()} {Childs[0].ToCppCode(level)}";
            for (int i = 1; i < Childs.Count; i++)
                ans += "," + Childs[i].ToCppCode(level);
            ans += ";\n";
            return ans;
        }
        private string CppTypeText()
        {
            switch(TypeText)
            {
                case "int16":
                    return "short";
                case "long":
                case "int32":
                    return "int";
                case "longlong":
                case "int64":
                    return "long long";
                case "unsignedshort":
                case "uint16":
                    return "unsigned short";
                case "unsignedlong":
                case "uint32":
                    return "unsigned int";
                case "unsignedlonglong":
                case "uint64":
                    return "unsigned long";
                case "longdouble":
                    return "long double";
                case "boolean":
                    return "bool";
            }
            return TypeText;
        }
    }

    public class Declarator : ASTNode
    {
        public string ID { get; set; }
```

```
        public bool IsArray { get; set; }
        public override string ToCppCode(int level)
        {
            if (!IsArray)
            {
                if (Childs.Count == 0)
                    return ID;
                else
                    return $"{ID} = {Childs[0].ToCppCode(level)}";
            }

            string ans = $"{ID}[{Childs[0].ToCppCode(level)}]";
            if (Childs.Count > 1)
                ans += " = [";
            for (int i = 1; i < Childs.Count - 1; i++)
                ans += Childs[i].ToCppCode(level) + ", ";
            ans += Childs[Childs.Count - 1].ToCppCode(level) + "]";
            return ans;

        }
    }

    public class Expression : ASTNode
    {
        public enum Op
        {
            Undefined,
            Or,
            Xor,
            And,
            LeftShift,
            RightShift,
            Add,
            Sub,
            Multi,
            Div,
            Mod,
            Positive,
            Negtive,
            Invert
        };
        public Op Operator { get; set; }
        public override string ToString()
        {
            switch (Operator)
            {
                case Op.Or:
                    return "|";
                case Op.Xor:
                    return "^";
                case Op.And:
                    return "&";
```

```
                case Op.LeftShift:
                    return "<<";
                case Op.RightShift:
                    return ">>";
                case Op.Add:
                    return "+";
                case Op.Sub:
                    return "-";
                case Op.Multi:
                    return "*";
                case Op.Div:
                    return "/";
                case Op.Mod:
                    return "%";
                case Op.Positive:
                    return "+";
                case Op.Negtive:
                    return "-";
                case Op.Invert:
                    return "~";
                case Op.Undefined:
                default:
                    return "Undefined";
            }
        }
        public override string ToCppCode(int level)
        {
            switch (Operator)
            {
                case Op.Or:
                case Op.Xor:
                case Op.And:
                case Op.LeftShift:
                case Op.RightShift:
                case Op.Add:
                case Op.Sub:
                case Op.Multi:
                case Op.Div:
                case Op.Mod:
                    return $"{Childs[0].ToCppCode(level)} {ToString()}
{Childs[1].ToCppCode(level)}";
                case Op.Positive:
                case Op.Negtive:
                case Op.Invert:
                    return $"{ToString()}{Childs[0].ToCppCode(level)}";
                case Op.Undefined:
                default:
                    return "";
            }
        }
    }
```

```
    public class Literal : Expression
    {
        public string Text { get; set; }
        public override string ToCppCode(int level) => Text;
    }
```

对于**4.抽象语法树生成**中给定的IDL文件例子，以下是生成的C++代码：

```
typedef const char* string;
namespace space
{
  typedef struct A
  {
    short i1 = 10;
    short i2 = 10 + 5 * 2;
  }A;
}
```