# Antlr使用的语法文件MIDL.g4

|  |
| --- |
| /\*\* Generate Shell:  $ cd ~/CompilePrinciple\_Ex1/CompilePrinciple\_Ex1  $ java -jar ./dependencies/antlr-4.13.1-complete.jar -Dlanguage=CSharp -visitor ./src/MIDL.g4  \*\*/  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  ; |

# 测试用例

|  |
| --- |
| module A  {  module inner  {  struct A  {  short i1=10;  int16 i2=10;  };  };  struct inner  {  long i3=100;  int32 i4=100;  };  };  module B  {  module middle  {  module inner  {  struct bottom  {  unsigned short i7=10;  uint16 i8=10;  uint32 i10=100;  long long i5=1000;  int64 i6=1000;  unsigned long i9=100;  };  };  };  };  struct B  {  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};  bool testBool1 = true | false;  bool testBool2 = true ^ false & false | true;  int testInt = 2 + 5 \* 2 / 3;  float testFloat = 1.2 \* 3 - 2 % 0;  }; |

可以看到，以上测试用例既包含了module和struct间的多层嵌套关系，同时也包含了各种不同类型变量声明的语句，还包含了表达式计算，可以比较完整的测试出词法分析和语法分析的正确性。

# 语法分析树构建

|  |
| --- |
| specification  definition  module  'module'  'A'  '{'  definition  module  'module'  'inner'  '{'  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'  ';'  type\_spec  base\_type\_spec  integer\_type  signed\_int  'int16'  declarators  declarator  simple\_declarator  'i2'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '10'  ';'  '}'  ';'  '}'  ';'  definition  type\_decl  struct\_type  'struct'  'inner'  '{'  member\_list  type\_spec  base\_type\_spec  integer\_type  signed\_int  'long'  declarators  declarator  simple\_declarator  'i3'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '100'  ';'  type\_spec  base\_type\_spec  integer\_type  signed\_int  'int32'  declarators  declarator  simple\_declarator  'i4'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '100'  ';'  '}'  ';'  '}'  ';'  definition  module  'module'  'B'  '{'  definition  module  'module'  'middle'  '{'  definition  module  'module'  'inner'  '{'  definition  type\_decl  struct\_type  'struct'  'bottom'  '{'  member\_list  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'unsigned'  'short'  declarators  declarator  simple\_declarator  'i7'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '10'  ';'  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'uint16'  declarators  declarator  simple\_declarator  'i8'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '10'  ';'  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'uint32'  declarators  declarator  simple\_declarator  'i10'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '100'  ';'  type\_spec  base\_type\_spec  integer\_type  signed\_int  'long'  'long'  declarators  declarator  simple\_declarator  'i5'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1000'  ';'  type\_spec  base\_type\_spec  integer\_type  signed\_int  'int64'  declarators  declarator  simple\_declarator  'i6'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1000'  ';'  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'unsigned'  'long'  declarators  declarator  simple\_declarator  'i9'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '100'  ';'  '}'  ';'  '}'  ';'  '}'  ';'  '}'  ';'  definition  type\_decl  struct\_type  'struct'  'B'  '{'  member\_list  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'unsigned'  'long'  'long'  declarators  declarator  simple\_declarator  'i11'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1000'  ';'  type\_spec  base\_type\_spec  integer\_type  unsigned\_int  'uint64'  declarators  declarator  simple\_declarator  'i12'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1000'  ';'  type\_spec  base\_type\_spec  'char'  declarators  declarator  simple\_declarator  'c0'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  ''a''  ';'  type\_spec  base\_type\_spec  'string'  declarators  declarator  simple\_declarator  'c1'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '"abc"'  ';'  type\_spec  base\_type\_spec  'boolean'  declarators  declarator  simple\_declarator  'c2'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'true'  ';'  type\_spec  base\_type\_spec  floating\_pt\_type  'float'  declarators  declarator  simple\_declarator  'c3'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '10.901f'  ';'  type\_spec  base\_type\_spec  floating\_pt\_type  'double'  declarators  declarator  simple\_declarator  'c4'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '23.234d'  ';'  type\_spec  base\_type\_spec  floating\_pt\_type  'long'  'double'  declarators  declarator  simple\_declarator  'c5'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '12.23456432235d'  ';'  type\_spec  base\_type\_spec  integer\_type  signed\_int  'short'  declarators  declarator  array\_declarator  'arr'  '['  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '10'  ']'  '='  exp\_list  '{'  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '0'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '2'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '3'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '4'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '5'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '6'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '7'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '8'  ','  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '9'  '}'  ';'  type\_spec  scoped\_name  'bool'  declarators  declarator  simple\_declarator  'testBool1'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'true'  '|'  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'false'  ';'  type\_spec  scoped\_name  'bool'  declarators  declarator  simple\_declarator  'testBool2'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'true'  '^'  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'false'  '&'  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'false'  '|'  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  'true'  ';'  type\_spec  scoped\_name  'int'  declarators  declarator  simple\_declarator  'testInt'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '2'  '+'  mult\_expr  unary\_expr  literal  '5'  '\*'  unary\_expr  literal  '2'  '/'  unary\_expr  literal  '3'  ';'  type\_spec  base\_type\_spec  floating\_pt\_type  'float'  declarators  declarator  simple\_declarator  'testFloat'  '='  or\_expr  xor\_expr  and\_expr  shift\_expr  add\_expr  mult\_expr  unary\_expr  literal  '1.2'  '\*'  unary\_expr  literal  '3'  '-'  mult\_expr  unary\_expr  literal  '2'  '%'  unary\_expr  literal  '0'  ';'  '}'  ';' |

以上是根据测试用例生成的语法分析树，可以看到，语法分析的结果没有错误。

# 抽象语法树构建

考虑到语法分析树中有很多冗余结点：例如空格、括号等符号，进行优化。

对每个产生式，删除固定的括号、“module”、“struct”等标识符。

优化了结构，删除了一些冗余的非终结符，使其不成为结点。

以下是AST树中仅仅含有的🧘几种结点类：

|  |
| --- |
| using System.Collections.Generic;  public abstract class ASTNode // 抽象AST树结点  {  public IList<ASTNode> Children { get; private set; } = new List<ASTNode>();  public abstract override string ToString();  // Specification结点，初始结点，孩子均为Struct结点或Module结点。  public class Specification : ASTNode  {  public override string ToString() => "Specification";  }  // Struct 结点，结构结点，孩子为若干个Member结点，表示该结构内声明的成员。  public class Struct : ASTNode  {  public string ID { get; set; }  public override string ToString() => $"Struct\_{ID}";  }  // Module结点，模块结点，孩子均为Struct结点或Module结点，表示该模块中声明的子模块和子结构。  public class Module : ASTNode  {  public string ID { get; set; }  public override string ToString() => $"Module\_{ID}";  }  // Member结点，结构成员结点。声明成员时，其保存成员的类型，其孩子均为Declarator结点，保存声明的多个变量。  public class Member : ASTNode  {  public enum TypeSpec  {  ScopeName,  BaseTypeSpec,  StructType  }  public TypeSpec Type { get; set; }  public string TypeText { get; set; }  public override string ToString() => $"Member\_Type({TypeText})";  }  // Declarator结点，每一个Declarator结点保存一个声明的变量。如果该变量有初始值，若是普通变量，则其第1个孩子是Expression结点，代表该变量的初始值；若是数组变量，则其第一个孩子是Expression结点，代表数组的长度，其余的孩子也是Expression结点，保存数组元素的初始化值。  public class Declarator : ASTNode  {  public string ID { get; set; }  public bool IsArray { get; set; }  public override string ToString() => $"{(!IsArray ? "Variable" : "Array")}\_{ID}";  }  // Expression结点，表达式结点，该结点的孩子也是Expression结点。通过先序遍历以Expression结点开始的子树，就可以得到Expression的值。  public class Expression : ASTNode  {  public string Operator { get; set; }  public override string ToString() => Operator;  }  // Literal结点，叶子结点，保存着初始结点数值。  public class Literal : Expression  {  public string Text { get; set; }  public override string ToString() => Text;  }  } |

以下是生成的AST树：

|  |
| --- |
| Specification  Module\_A  Module\_inner  Struct\_A  Member\_Type(short)  Variable\_i1  10  Member\_Type(int16)  Variable\_i2  10  Struct\_inner  Member\_Type(long)  Variable\_i3  100  Member\_Type(int32)  Variable\_i4  100  Module\_B  Module\_middle  Module\_inner  Struct\_bottom  Member\_Type(unsignedshort)  Variable\_i7  10  Member\_Type(uint16)  Variable\_i8  10  Member\_Type(uint32)  Variable\_i10  100  Member\_Type(longlong)  Variable\_i5  1000  Member\_Type(int64)  Variable\_i6  1000  Member\_Type(unsignedlong)  Variable\_i9  100  Struct\_B  Member\_Type(unsignedlonglong)  Variable\_i11  1000  Member\_Type(uint64)  Variable\_i12  1000  Member\_Type(char)  Variable\_c0  'a'  Member\_Type(string)  Variable\_c1  "abc"  Member\_Type(boolean)  Variable\_c2  true  Member\_Type(float)  Variable\_c3  10.901f  Member\_Type(double)  Variable\_c4  23.234d  Member\_Type(longdouble)  Variable\_c5  12.23456432235d  Member\_Type(short)  Array\_arr  10  0  1  2  3  4  5  6  7  8  9  Member\_Type(bool)  Variable\_testBool1  |  true  false  Member\_Type(bool)  Variable\_testBool2  |  ^  true  &  false  false  true  Member\_Type(int)  Variable\_testInt  +  2  /  \*  5  2  3  Member\_Type(float)  Variable\_testFloat  -  \*  1.2  3  %  2  0 |