姓名：刘权祥

学号：2019300414

MIDL To C++源到源编译器实验报告

实验环境

- 操作系统：Ubuntu 18.04
- 编程语言：Python3.6.9

实验内容

IDL（Interface Definition Language）是一种平台无关的接口定义（或描述）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。MIDL（Mini Interface Definition Language）是IDL语言的一个语法子集，作为编译原理课程的实验对象。

实验的内容为：

- 使用Antlr4开源工具构建MIDL语言编译器前端
- 在MIDL语言编译器前端的基础上实现MIDL语言的语义分析和代码生成
- 最终实现MIDL To C++源到源编译器

实验过程与实现

> 整合实验一实验二内容，构建一个完整的MIDL To C++的源到源的编译器，并依据实践内容撰写一篇详细实验报告，报告内容包括但不限于实验环境，实验内容，实验流程，测试说明，将这两次实验课中的设计，实现，测试等内容讲述清楚即可。

学习Antlr4开源工具

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。

构建MIDL的词法规则与语法规则

这里我参考了网上的教程，分别构建词法规则与语法规则，这样主要是方便后续进行测试，并且也更加规范。

构建词法规则`MIDLLexerRules.g4`：

antrl4生成的词法分析程序是从上往下匹配的，所以如果词法规则中有的顺序不对，会导致词法分析出问题。

所以这里构建词法规则的时候，必须要注意相关的顺序，并且在构建的时候可以对已经写好的词法规则做测试。

```
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 | (~'\\'|~'"'))*'"';

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

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

构造语法规则 MIDLGrammarRules.g4：

下面是根据实验文档写出来的语法规则。

```
grammar MIDLGrammarRules;
import MIDLLexerRules;
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;
```

生成词法、语法分析程序

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
```

设计抽象语法树，实现并测试

在实验一中，我对抽象语法树的认识不够，写的抽象语法树很简单，就是用list保存了每一个节点的信息，这样也不方便后续的处理，但是格式化输出抽象语法树的任务达到了。

但是经过实验二，我充分认识到了抽象语法树的重要，如果没有定义好抽象语法树的话，语义分析和代码生成基本就无法进行，所以后面是在`ASTree.py`中定义了抽象语法树类。

定义抽象语法树类

抽象语法树定义文件`2_相关文档`在文件夹下的抽象语法树.pdf。

下面是我定义的类，相关的介绍可以看代码注释：

```
# 抽象语法树
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.children = []  # 该节点的孩子节点们

        self.dType_ = None

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

    # 这个函数最后没有使用!
    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.children) == 0:
            self.add_child(child)
        else:
            self.children[-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.children:
            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"
        if self.dType_ is not None:
            structure += str(self.type_) + " " + str(self.dType_) + "\n"
        else:
            structure += str(self.type_) + "\n"
        for child in self.children:
            structure += child.get_structure()
        return structure

    def get_AST(self, last_level=0):
        """
        获取抽象语法树的内容
        """
        AST = ""
        self.set_child_level()
        if self.type_.startswith("Terminator"):
            if self.level > last_level:
                last_level += 1
            elif self.level < last_level:
                last_level -= 1
            for i in range(last_level + 1):
                AST += "\t"
            AST += str(self.str_) + "\n"
        for child in self.children:
            AST += child.get_AST(last_level)
        return AST
```

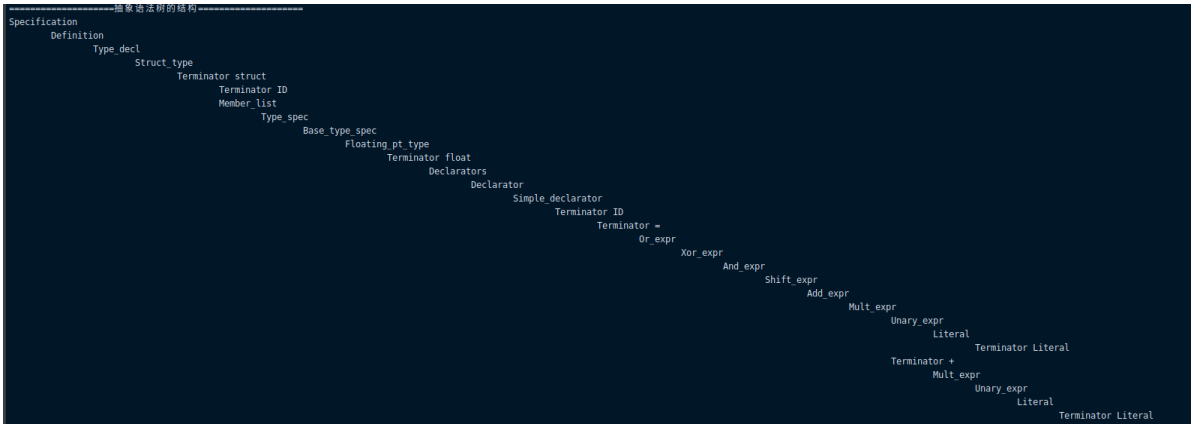
其中 get_structure 函数输出的是抽象语法树的结构，它的每一个节点都会被输出。

而 get_AST 函数，我代码里面写的是输出抽象语法树的内容，这样的表述可能不准确，我这里是希望它能够只把抽象语法树中包括了源代的内容输出。

比如对于：

```
struct test{
    float a1=1+2;
};
```

这是 get_structure 函数输出的结果



这是`get_AST`函数输出的结果

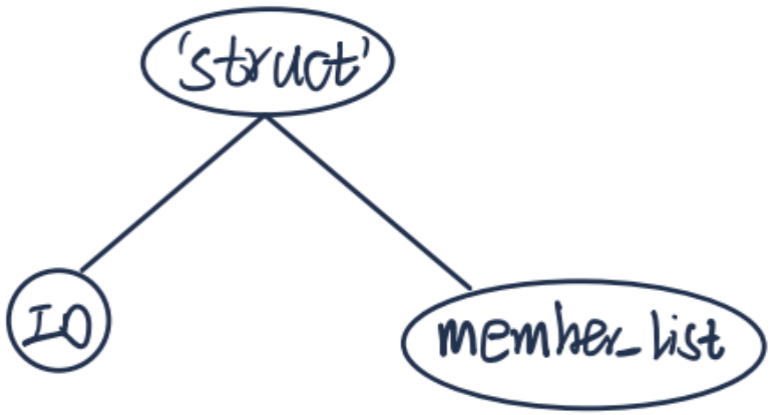
```
====================抽象语法树的内容====================
    struct
      test
      float
        a1
          =
            1
            +
              2
```

在Visitor中构建抽象语法树

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：



对应的代码如下所示：

```
# Visit a parse tree produced by MIDLGrammarRulesParser#struct_type.
def visitStruct_type(self, ctx: MIDLGrammarRulesParser.Struct_typeContext):
    # 构建当前语法规则的节点，表示执行了这条规则
    struct_type_node = TreeNode("Struct_type")
    # 根据抽象语法树，第一个节点是终结符struct，这里构建它的节点
    node = TreeNode("Terminator struct", "struct")
    # 它的左孩子是ID节点
    node.add_child(TreeNode("Terminator ID", ctx.ID()))
    # 它的右孩子是member_list节点，通过visit函数迭代返回得到
    node.add_child(self.visit(ctx.member_list()))
    # 最后把终结符struct节点加入到当前语法规则的孩子节点中
    struct_type_node.add_child(node)
    return struct_type_node
```

实现MIDL语义分析

为了能够实现MIDL的语义分析，这里我的思路是：

- 对于命名冲突和未定义即使用，构建符号表来进行判断。
  - 为每一个作用域构建一个符号表；
  - 符号表的孩子节点指向子作用域的符号表；
- 对于字面量类型检查，扩展抽象语法树，加入属性文法来解决。

下面是实现过程的介绍：

构建符号表

符号表的定义

下面是我定义的符号表，这个符号表的功能有：

- `add_child`：添加子作用域符号表。
- `add_symbol`：添加新的符号，在添加新的符号的同时进行检查，判断是否存在命名冲突。
- `lookup`：检查命名冲突。如果在当前作用域找到，就报错；如果在上级的作用域找到，就发出警告；如果找不到，就返回没有找到。
- `check_namespace`与`lookup_namespace`：检查是否存在未定义即使用的情况，检查过程是自顶向下的。
- `display`：格式化输出符号表，便于检查分析。

```
# 一个自定义的符号表
class SymbolTable:
  def __init__(self, namespace="root"):
    self.namespace = namespace  # 定义一个名字，方便后面debug

    self.children = []  # 每一个子作用域的符号表
    self.symbols = []  # 构建一个id_name和type的list
    self.parent = None  # 符号表的父节点

  def add_child(self, child):
    child.parent = self
    self.children.append(child)

  def add_symbol(self, symbol):
    look_result = self.lookup(symbol)
    if look_result == "NOT FOUND":
      self.symbols.append(symbol)
      return
    elif look_result == "FOUND":
      print("错误：\tID \"" + symbol + "\" 在当前作用域多次定义！")
      exit(-1)
    elif look_result == "FOUND IN PARENT":
      print("警告：\tID \"" + symbol + "\" 在上级作用域已经被定义！")

  def lookup(self, symbol):
    """
    检查命名冲突，只会往上一级作用域查找！
    """
    if symbol in self.symbols:
      return "FOUND"  # 表示在当前作用域
    elif self.parent is not None:
      if self.parent.lookup(symbol) == "FOUND": # 表示在上一级作用域
```

```
            return "FOUND IN PARENT"
        return "NOT FOUND"  # 表示找不到

    def check_namespace(self, type_list: list, type_name: str):
        if not self.lookup_namespace(type_list):
            print(type_name + " 未定义即使用！ ")
            exit(-1)

    def lookup_namespace(self, type_name: list):
        """
        检查未定义即使用，以及引用是否正确。自顶向下查找，只能由root节点进行！
        """
        if len(type_name) == 0:
            return True

        if self.namespace.endswith("root"):
            for child in self.children:
                if child.lookup_namespace(type_name):
                    return True
            return False

        else:
            if self.namespace.endswith(type_name[0]):
                type_name.pop(0)
                for child in self.children:
                    if child.lookup_namespace(type_name):
                        return True
                return False

    def display(self):
        print(self.namespace, end="\n\t")
        for symbol in self.symbols:
            print(symbol, end=" ")
        print()
        for child in self.children:
            child.display()
```

在visitor中构建符号表

思路：

- 在visitor中保存了两份符号表，一个是root_symbol_table（根符号表），一个是current_symbol_table（当前作用域符号表）。
- 在一开始就构建root_symbol_table，并且让current_symbol_table指向root_symbol_table。
- 在遍历的过程中：
    - 如果遇到了module或者struct，就表示进入了新的作用域。此时构建新的符号表，加入到当前符号表的孩子节点中，并且将current_symbol_table指向新建立的符号表。
    - 如果从module或者struct中退出，就将current_symbol_table指向进入之前的符号表。
    - 如果遇到了ID，就加入到current_symbol_table中，加入的同时会进行命名冲突的检查。
    - 如果遇到了自定义类型（ID::ID::ID类型的），就从根符号表root_symbol_table开始从上往下进行检查，查找自定义类型是否存在。

下面是一个例子：

```
    # Visit a parse tree produced by MIDLGrammarRulesParser#struct_type.
```

```
    def visitStruct_type(self, ctx: MIDLGrammarRulesParser.Struct_typeContext):
        # 添加抽象语法树的节点
        struct_type_node = TreeNode("Struct_type")
        node = TreeNode("Terminator struct", "struct")
        node.add_child(TreeNode("Terminator ID", ctx.ID()))
        # 添加ID到当前的符号表
        self.add_symbol(ctx.ID())
        # 创建新的作用域
        symbol_table = SymbolTable("struct " + str(ctx.ID()))
        self.current_symbol_table.add_child(symbol_table)
        # 进入作用域
        self.current_symbol_table = symbol_table
        node.add_child(self.visit(ctx.member_list()))
        struct_type_node.add_child(node)
        # 退出作用域
        self.current_symbol_table = self.current_symbol_table.parent

        return struct_type_node
```

扩展抽象语法树，编写属性文法

本来我是打算继续扩展符号表，在符号表中做字面量类型检查的，但是我觉得抽象语法树保存的信息更多，编写属性文法要简单很多。

构建DType类

在扩展语法树之前，我觉得MIDL中涉及的类型比较多，所以我定义了一个静态类，即DType类来保存MIDL中的类型，并且构建了几个简单的函数来判断是有符号整形、无符号整形还是浮点数类型。

下面是具体实现的代码截图：

```
    # float pt
    FLOAT = "float"
    DOUBLE = "double"
    LONG_DOUBLE = "long double"
    # text
    CHAR = "char"
    STRING = "string"
    # boolean
    BOOLEAN = "boolean"

    @staticmethod
    def is_signed_int(str_: str):
        if str_ == DType.SHORT or \
                str_ == DType.INT8 or \
                str_ == DType.INT16 or \
                str_ == DType.INT32 or \
                str_ == DType.INT64 or \
                str_ == DType.LONG or \
                str_ == DType.LONG_LONG:
            return True
        return False

    @staticmethod
    def is_unsigned_int(str_: str):
        if str_ == DType.UNSIGNED_SHORT or \
                str_ == DType.UINT8 or \
                str_ == DType.UINT16 or \
                str_ == DType.UINT32 or \
                str_ == DType.UINT64 or \
                str_ == DType.UNSIGNED_LONG or \
                str_ == DType.UNSIGNED_LONG_LONG:
            return True
        return False
```

扩展抽象语法树

这里我把和类型检查相关的节点都进行了扩展，放在了`ASTree.py`中，具体包括了：LiteralNode、Unary_exprNode、Mult_exprNode、Add_exprNode、Shift_exprNode、And_exprNode、Xor_exprNode、Or_exprNode、Exp_listNode、Array_declaratorNode、Simple_declaratorNode、DeclaratorsNode、Type_specNode。

由于几乎每一个节点的情况都不一样，属性文法也不太一样，并且属性文法写的比较多，所以下面就挑几个来进行介绍：

对于Unary_expr节点：

unary_expr的语法规则为：unary_expr -> [“-”| “+” | “~”] literal

这部分属性文法的思路为：

- 符号是可选的，所以如果没有符号，它的dtype就和literal保持一致。
- 如果有符号，就要考虑正确性以及是否转成signed类型
    - 对于char、string类型，它们不支持这种运算，所以需要报错；
    - 对于“~”，我设计的是它只支持bool类型的运算，其他情况都要报错；
    - 对于“+”，这个对整数或者浮点数都不影响，所以就保持原样，而其他类型就报错；
    - 对于“-”，只有是无符号整型才会被改成有符号整型，如果不是浮点数或者整数，这里就需要报错。

```
class Unary_exprNode(TreeNode):
  def __init__(self, type_: str, str_=""):
    super().__init__(type_, str_)
```

```
    self.dType_ = None

  def check_type(self):
    if len(self.children) == 1: # 没有符号就保持一致
      self.dType_ = self.children[0].dType_
    else: # 考虑正确性以及是否转成signed类型
      dType_ = self.children[1].dType_
      if self.children[0].str_ == "~": # 单独处理~运算
        if DType.is_BOOLEAN(dType_): # bool类型支持运算
          self.dType_ = DType.BOOLEAN
        else:
          print(self.children[1].str_)
          print("错误：" + dType_ + "类型不支持~运算！")
          exit(-1)
      # 对于+、-运算
      elif DType.is_text(dType_) or DType.is_BOOLEAN(dType_): # CHAR、STRING、BOOLEAN都不支持这
种运算
        print(self.children[1].str_)
        print(dType_ + "类型不支持-、+运算！")
        exit(-1)
      elif self.children[0].str_ == "-" and DType.is_unsigned_int(dType_): # 如果-遇上了无符号数
        self.dType_ = DType.to_signed_int(dType_)
      else: # 其他情况都应该是和子节点一致的类型！
        self.dType_ = dType_
```

对于Exp_list节点：

它的语法规则为：exp_list -> “[” or_expr { “,”or_expr } “]”

这部分属性文法的思路为：

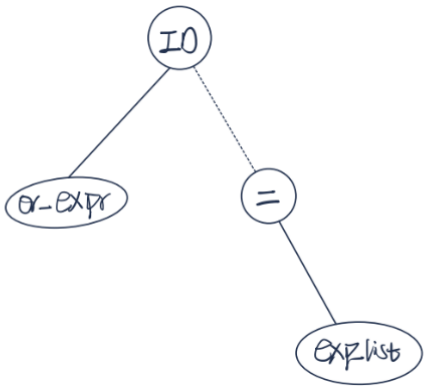
- 如果只有一个孩子节点，那么说明这部分不是数组类型的，类型就跟孩子节点保持一致；
- 如果存在多个孩子节点，说明这部分是一个数组类型，那么：
  - exp_list.dType等于任意一个孩子节点的dType
  - 如果存在孩子节点之间dType不统一的情况，就报错

```
class Exp_listNode(TreeNode):
  def __init__(self, type_: str, str_=""):
    super().__init__(type_, str_)

    self.dType_ = None

  def check_type(self):
    if len(self.children) == 1: # 只有一个or_expr，那么就不是数组类型
      self.dType_ = self.children[0].dType_
    else: # 否则就是多个or_exper，导致是数组类型
      temp = None # 一个临时变量
      for child in self.children:
        if temp is None:
          temp = child.dType_
        else:
          if temp != child.dType_:
            print("数组类型不统一")
            exit(-1)
      self.dType_ = temp
```

对于Array_declaratorNode节点：

它的语法规则为：array_declarator -> ID “[” or_expr “]” [“=” exp_list]

我定义的抽象语法树为：



属性文法的思路：

- 首先需要检查下标，即检查抽象语法树中or_expr节点的情况：
  - 如果下标是浮点数、STRING、负数，就报错；
  - 如果下标是CHAR、BOOLEAN，就报警，因为这里可以进行类型转换，转换之后的程序可以正常运行；
  - 其他情况（非负整数）则通过。
- 之后需要根据孩子节点的数目判断有没有进行赋值：
  - 如果孩子节点数目为1，就没有进行赋值，不需要继续进行检查；
  - 如果孩子节点数目大于1，说明存在给数组变量赋值的情况，需要进行检查，看数组长度是否正确：

    （因为孩子节点是Exp_list，在构建Exp_list节点的时候类型检查已经做过了，所以只需要检查数组长度）
    - 如果数组的长度小于定义的长度，就发出警告；
    - 如果数组的长度大于定义的长度，就报错。

```
class Array_declaratorNode(TreeNode):
    def __init__(self, type_: str, str_=""):
        super().__init__(type_, str_)
        self.dType_ = None

    def check_type(self):
        # 找到ID节点：
        node = self.children[0]
        # 首先，检查下标
        dType_ = node.children[0].dType_
        if DType.is_float_pt(dType_):
            print(node.children[0].str_)
            print("错误：" + dType_ + "类型不可以做索引！")
            exit(-1)
        elif dType_ == DType.CHAR or dType_ == DType.BOOLEAN:
            print("警告：" + dType_ + "类型被用作索引！")
        elif dType_ == DType.STRING:
            print(node.children[0].str_)
            print("错误：STRING类型不可以做索引！")
            exit(-1)
        elif DType.is_signed_int(dType_):
```

```
        print(node.children[0].str_)
        print("错误：下标不可以为负数！")
        exit(-1)

    if len(node.children) == 1:
        return
        # 获取数组的长度
    temp = node
    while len(temp.children) != 0:
        temp = temp.children[0]
    array_len = int(temp.str_)
    # 如果有赋值，给类型赋值，否则不复制，后续程序会忽略None类型
    node = node.children[1] # 目前是等号节点
    node = node.children[0] # 目前是Exp_list节点
    exp_len = len(node.children)
    if exp_len > 1: # 如果有多个孩子节点
        if exp_len > array_len:
            print("错误：数组长度超过定义的长度！")
            exit(-1)
        elif exp_len < array_len:
            print("警告：数组长度小于定义的长度！")
```

通过StringTemplate生成C++代码

StringTemplate的原理

string.Template，将一个string设置为模板，通过替换变量的方法，最终得到想要的string。

```
>>> from string import Template
>>> template_string = '$who likes $what'
>>> s = Template(template_string)
>>> d = {'who': 'Tim', 'what': 'kung pao'}
>>> s.substitute(d)
'Tim likes kung pao'
```

设计StringTemplate的C++模板

设计开始部分的代码模板：

基本每一个hxx文件开头部分都是一样的，所以这里可以单独设计一个开始部分的代码模板。

```
class CppStringTemplate():
    START = """
#ifndef ${name}_h
#define ${name}_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
    """
```

设计结束部分的代码模板:

基本每一个hxx文件开头部分是一样的，这里也单独设计一个开始部分的代码模板。

```
    END = """
#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
    """
```

设计变量声明和初始化的代码模板:

这里主要是处理这种类型的代码生成：`CDR_Short i1=10;` 和 `CDR_Primitive_init_Short(&sample->i1)`

因为一个module或者struct不一定只有一个变量，所以这里需要单独设计相关的模板，然后再整合到整个的代码模板中去。

```
    # 有声明
    CDR_INIT_A = """
CDR_${type} ${ID}=${exp};

"""
    # 无声明
    CDR_INIT_B = """
    CDR_${type} ${ID};

    """

    CDR_SENTENCE = """
CDR_Primitive_init_${type}(&sample->${ID});

"""
```

设计中间部分代码的模板：

这部分就是参照着hxx文件里面的形式构建的模板。

```
    CODE = """
struct ${classname}Seq;
class ${classname}TypeSupport;
class ${classname}DataWriter;
class ${classname}DataReader;

class ${classname}
{
  public:
    typedef struct ${classname}Seq Seq;
    typedef ${classname}TypeSupport TypeSupport;
    typedef ${classname}DataWriter DataWriter;
    typedef ${classname}DataReader DataReader;

    ${CDR_INIT}
};

extern const char *${classname}TYPENAME;

REDA_DEFINE_SEQUENCE_STRUCT(${classname}Seq, ${classname});

REDA_DEFINE_SEQUENCE_IN_C(${classname}Seq, ${classname});

NDDSUSERDllExport extern RTI_BOOL
${classname}_initialize(${classname}* sample)
{
    ${CDR_SENTENCES}

    return RTI_TRUE;
}

NDDSUSERDllExport extern RTI_BOOL
${classname}_finalize(${classname}* sample)
```

通过C++模板生成C++代码

在类的初始化过程中，使用传入的抽象语法树节点、符号表节点、文件名来生成c++代码

```
def __init__(self, node: TreeNode, table: SymbolTable, filename: str):
    # 开始部分,只有一个
    self.start_tmp = Template(CppStringTemplate.START)
    # 代码部分，要注意有可能有多个并列的struct或者module
    self.code_tmp = Template(CppStringTemplate.START)
    # 声明部分A，可能有多个声明
    self.cdr_init_a_tmp = Template(CppStringTemplate.CDR_INIT_A)
    # 声明部分B，可能有多个声明
    self.cdr_init_b_tmp = Template(CppStringTemplate.CDR_INIT_B)
    # 初始化部分，每个声明对应一个初始化
    self.cdr_sentence_tmp = Template(CppStringTemplate.CDR_SENTENCE)
    # 结束部分,也只有一个
    self.end_tmp = Template(CppStringTemplate.END)
    # 最后生成的代码
    self.code_string = ""

    # 生成代码的开头部分
    self.__init_start__(filename)
    # 生成代码的中间部分
    self.__init_code__(node, table)
    # 生成代码的结尾部分
    self.__init_end__()
```

其中开头部分的代码和结尾部分的代码比较基本，这里介绍生成代码的思路：

```
# 根据符号表和抽象语法树来进行代码生成
def __init_code__(self, node: TreeNode, table: SymbolTable):
    # 对于每一个struct或module
    for child in table.children:
        # 声明
        cdr_init_code = ""
        # 初始化
        cdr_init_sentence = ""
        # 遍历符号表
        for symbol in child.symbols:
            # 生成声明变量的代码
            cdr_init_code += self.__sub_init_code__(node, child)
            # 生成初始化的代码
            cdr_init_sentence += self.__sub_init_sentence__(node, child)
        # 生成中间部分的所有代码
        self.code_string += self.__sub_code_string__(node, child, cdr_init_code, cdr_init_sentence)
```

测试说明

对抽象语法树进行测试

这部分的说明在`2_相关文档`文件夹下的实验1_测试说明.pdf中。

对语义规则进行测试

这部分的说明在`2_相关文档`文件夹下的实验2_语义错误测试说明文档.pdf中。

对C++代码生成进行测试

这部分的说明在`2_相关文档`文件夹下的实验2_代码生成测试说明文档.pdf中。

总结

这个实验做的并不是很顺利，但很大程度的锻炼了我的编程能力，并且加深了我对编译原理知识的理解。

一开始做实验一的时候，找不到比较好的教程，前前后后耽误了比较长的时间才把g4文件给写出来，后面写抽象语法树就写的很简单，用list直接保存了起来；这就导致做实验二的时候，发现自己的抽象语法树完全没法用，所以做实验二的前几天都是在好好地修改实验一的代码，完善自己定义的抽象语法树并且实现抽象语法树的类。

而做实验二的时候，一开始没有思路，折腾了很久之后才意识到应该按老师课上讲的一样写一个符号表。符号表我定义的没有问题，不过在visitor里面用的时候出了比较多的状况。我感觉visitor遍历语法树是迭代的形式，这样方便编写代码，但是不方便理解，这导致我的符号表一开始一直没有应有的效果。

另外，写代码的时候最痛苦的就是发现自己抽象语法树相关的部分出问题了，这一般就需要修改visitor里面的函数。但因为MIDL的比较多，visitor里面要修改的有26个函数，每个规则对应的函数要修改的部分可能还不一样，这导致每次修改花的时间都挺多的。

不过最后还是把MIDL To C++源到源编译器给做出来了，C++代码生成的效果不是那么好，但还是挺有成就感的。