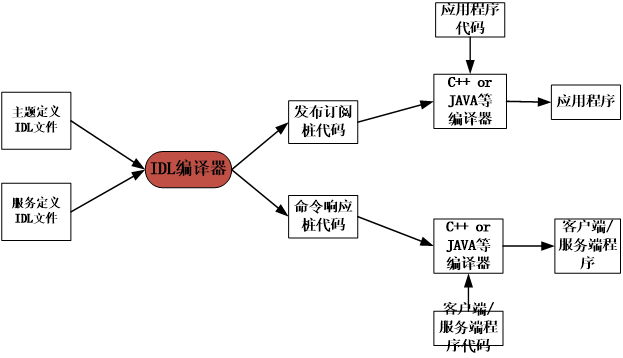
## 实验二：MIDL语言的语义分析与代码生成

### MIDL简介

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。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。



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。

### MIDL To C++ 编译器

一个完整的编译器的逻辑结构如下图所示。



但是对于MIDL To C++ 编译器来说，我们只需要从一种源代码转换为另一种源代码即可，而无需转换为特定机器上的二进制机器码，这种编译器被称为源到源编译器（source-to-source compiler）。因此，我们只需要考虑编译器前端部分即可完成一个源到源编译器。在实验一中，我们实现了词法分析，语法分析，并生成了抽象语法树。在本次实验中，我们要完成剩下两个模块，语义分析与代码生成模块。

### MIDL 语义分析

本次实验语义分析的内容有以下3条：

1. 命名冲突。同一命名空间内，不能出现相同名字的接口定义。

|  |  |
| --- | --- |
| 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。 |

1. 未定义即使用。struct结构需要先定义才能使用。

|  |  |
| --- | --- |
| 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 |

1. 字面量类型检查。字面量的数据类型需要和变量类型相同或兼容。

|  |  |
| --- | --- |
| 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 |
| struct A{  short a=15.24;  }; | a是整型变量，字面量却是浮点类型 |
| 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 |

### MIDL To C++ 代码生成

IDL作为接口定义语言，具有跨平台的好处。在分布式应用中，我们可以使用IDL编译器将IDL接口定义转换为特定编程语言实现的客户端或服务端桩程序。例如，商业IDL编译器rtiddsgen工具，针对特定的IDL（struct.idl）会生成6个实现文件，分别是实现文件（struct.h/struct.cxx），Plugin文件（structPlugin.h/structPlugin.cxx），Support文件（structSupport.h/structSupport.cxx）。

本次实验代码生成部分为了降低难度，仅需要生成一份实现文件即可。输入为MIDL源码，输出为一份.hxx实现文件。相关测试用例详见“**代码生成用例集.zip**”，可对照测试用例进行代码生成的逻辑编写。

### 实验二任务要求

1. 在实验一内容的基础上，完成语义分析，实现能够对上文所提的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。并给出测试说明文档。
2. 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，总结抽象语法节点的生成规则，并编写代码生成模块。（代码生成可使用ppt中第6章代码生成的方法，也可以采用附录A中介绍的StringTemplate模板方式生成目标代码，并配有学习视频“StringTemplate模板语法介绍.mp4”和“StringTemplate模板语法介绍.mp4”）。
3. 针对语义分析和代码生成设计测试用例，给出测试方法描述，提交readme.doc，如果你还有其他需要说明的问题须写在readme.doc中。
4. 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。

附录A:

### StringTemplate模板引擎

StringTemplate（简称ST）是一个基于Java的模板引擎库（同时支持C#、Python、Ruby），可以用于生成源代码、Web页面、电子邮件等各种有格式的文本。StringTemplate不同于其他类似的模板引擎的显著特点是严格执行模型视图分离，尤其擅长多目标代码生成、多种网站风格管理以及网站多语言版本生成等。目前支持的语言有Java,C#,JavaScript(not stable),Python和Objective-C。

在我们提供的“代码生成用例集.zip”中，可以发现生成的hxx文件是以特定的格式进行组织的，且有大量固定的字符串。如果我们使用完全的字符串拼接，有两个坏处：

（1）可读性差。因为是在源码中进行字符串拼接，因此代码的整洁度和可读性都不好。

（2）可维护性差。同样因为是在源码中进行字符串拼接，所以在字符串格式控制上难以维护，可能稍有不慎就导致比较大的格式问题。

StringTemplate模板引擎可以很好的解决这个问题。用户可以通过定义STG文件，定义模板函数来将视图与模型分离（view-model)，进而直接控制字符文本的格式，所见即所得。而模板中变化的部分可以通过函数参数的形式传入模板中进行渲染。相关文档如下链接所示：

<https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>

[](StringTemplate模板语法介绍.mp4)

[](JAVA中调用ST模板引擎演示视频.mp4)