# Lab02-Semantic Analysis

> 实验名称：NJU-编译原理(2022秋)-Lab02-语义分析
> 实验人员：201220096 钟亚晨
> 完成日期：2022/10/31
> 更多实现细节可见个人博客文章：(pawx2's Blog)【密码：`chcp65001`】

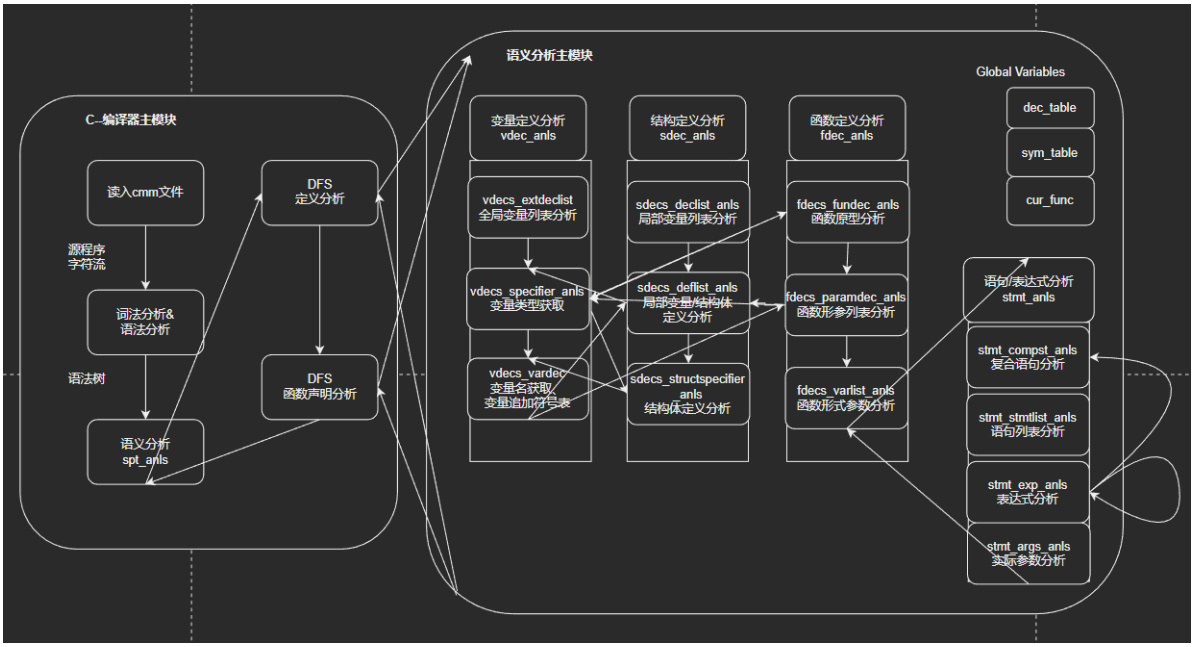
## 功能实现

- 实现了`实验二_C--语义分析.pdf`中要求的**17种基本错误类型**的检测；
- 实现了额外的**3类拓展机制**，包括：作用域、函数声明、结构体结构等价；
- 通过了OJ平台上的所有测试用例；

## 实现思路

- 核心实现思路为：依照语法树结构转入不同的函数进行分析，在分析过程当中打印语义错误。
- 三种拓展功能的实现：
  - 作用域：使用`哈希表+栈->十字链表`进行支持，一个栈帧对应一个纵向链表，为一个作用域符号表；
  - 函数声明：补充文法，使用二次DFS语义分析，开辟专用符号表记录函数声明。第一次仅分析定义部分，待符号表填充完整分析声明部分；
  - 结构体等价：通过定义类型节点`TypeNode`、编写函数`equal_type_nodes`进行递归比较来进行实现；

### 程序架构图




## 实现亮点

### 数据结构方面：

符号表：使用 `哈希表 + 栈` 构成的十字链表：

- 每个哈希表项对应的**横向链表**为哈希表聚集时开散列存储的表项，该部分主要用于哈希查找；
- 每个栈帧对应的**纵向链表**为当前作用域下的哈希表项集合，通过此可完成作用域机制；

数据类型：

- 类型采用自定义类型 `TypeNode` 使用链表进行实现，每个符号表项都具备一个 `TypeNode` 指针：
  - 对于 int、float型符号表项：该指针为空，直接使用符号表项类型 `SymNode` 的字符串类型名进行指定即可；
  - 对于结构体、函数符号表项：其分别具备对应的链表成员（由其它变量的符号表项副本构成）表示结构体成员、函数形式参数列表；
- 对于结构体类型变量的符号表项，其具备 `TypeNode*` 类型成员，指向一个结构体符号节点，用于指示该变量的结构体类型信息。

### 语义分析实现方面：

**Trick01：函数定义的分析过程中：**

首先我们需要对函数的形式参数列表进行分析，而后才是函数体部分。而该部分在我的实现中被划分为两个部分：1. 函数原型的分析、2. 函数体的分析，因此此时产生问题：参数列表中的形式参数如何传递给函数体的作用域？这里的解决办法为：

- 在进入函数形式参数列表分析时，使用函数 `stack_push` 追加符号表栈帧，新建作用域符号表，然后对函数形参列表分析、填表，但在退出函数形参分析时，不使用函数

stack_pop清空当前作用域符号表，而是直接移动符号表的作用域栈栈顶指针，此时形式参数仍被保留，但并不可用。

- 在进入函数体分析时，使用函数`stack_push`追加符号表栈帧，此时符号表作用域栈栈顶指针移动到新的一层，而这里正好是之前遗留的形参作用域符号表，至此，形参符号作用域符号表的内容就传递给了函数体作用域符号表。

**Trick02：结构体类型的数据，其数据类型如何表示?**

这里的核心思路为：所有同类型的变量，其TypeNode指向同一个结构体符号表项。而在删除作用域符号表时，如果该表项的类型是一个结构体，那么就仅仅只是将其移出符号表而仍然保留，否则进行删除。每一个结构体类型的变量都具有一个`SymNode*`类型成员，指向一个结构体符号节点（可能在符号表中，也可能不在），该节点保存了结构体的类型信息。

**Trick03：函数声明的实现**

- 开辟一个新的`dec_table`符号表，专门用于存储函数声明；
- 第一遍DFS分析仅分析各类定义，填充符号表，如遇到函数定义，则将其符号表节点副本添加到`dec_table`符号表。第二遍DFS分析则仅针对函数声明：
  - 查询符号表以检查是否存在定义，查询`dec_table`以检查其是否与其它声明/定义相冲突。

# 实验感想

- 总体设计方面：事实上，在开发过程中遇到的一个大问题是数据类型的表示，最开始对于结构体类型是使用`结构体名`而后查询符号表来实现，但这样带来的问题是多层嵌套定义的结构体，在有作用域约束的条件下自然而然就引发了段错误（内部结构体已被删除，但在外部引用了其相关变量），这是一个偏向于设计底层的问题，因此修改该部分耗费了较多的时间。这也启示了我：**在一个模块/项目最初设计时，并不仅仅是数据结构、模块划分的选型决定了方向就可以开始进行项目开发的，最好的情况是能够进行“思想实验”，想想功能从顶层到底层，边想边构建出一棵“树/图”，不断修正最底层的数据结构。在设计之初通过思考修正错误的成本要远低于在开发后进行修正的成本。**
- Dubug方面：本次实验过程中出现了众多的**段错误**，在最初进行Debug时通过"插桩法"不断进行终端输出来找出错误代码，这样做效率低、难度大。因此后续抽时间学习了**gdb的基本调试方法**和Linux的**coredump**，从而能够较快的获取错误信息，尤其是能够快速地进行**段错误定位**。在感受到调试工具带来的便利高效后，不禁感慨：程序调试的工具得要学起来、用起来（诸如vscode调试配置、gdb等），但同时在最初的编程设计时也需要尽可能考虑周到。