实验 2：语义分析

数据结构如讲义中所述，并带有适当的扩展。 更详细的解释反映在评论中。其中，z 号表以 FieldList 为基本数据单元，我们选择将所有符号组织成一张表，这里选择哈希表作为符号表的数据结构，哈希函数的算法由 提出。算法。 并使用开放哈希方法作为处理哈希表冲突的一种方式。另外，由于实验需要支持多个作用域，这里我们使用 Imperative Style 作为表示法表维护方式，并设计了基于交叉链表和 openhashing 改进的 hash 表，与 handout 的相比主要区别在于我们关于层的深度的 FieldList 与 Hash 表无关，而是单独维护这个深度。符号表。简单地说，它维护了一个全局变量 curFuncDep。每当遇到 CompSt 结构的语句块时，输入在进入 curFuncDep+=1 之前，构造这个深度的 FieldList 头，当在这个语句块中遇到新变量时定义时，将变量插入符号表并使用 FieldList 标头将其字符串化。当任何表达式出现在语句块中时，使用变量直接使用哈希函数查询。退出 CompSt 语句块后，当前深度的 FieldList 表头连接的所有符号都从表中删除，然后设置 curFuncDep-=1。

本实验室使用的实验一中用到的语法结构，首先对语法树进行了深入探索。语义分析类
类型检查和错误检查本实验室使用的实验 1 的数据结构。附录 A 中函数的语法规则。并在插入哈希表之前应用于 Type 和 FieldList。执行搜索，然后使用 type_check 函数执行类型检查。共执行了 17 次错误检查。修改后的符号数据使用以下数据结构存储。

语义分析借助实验一建立的语法树，我们可以深度优先遍历语法树。在此过程中执行语义分析和错误检查。 更详细的解释反映在评论中。实验的主要代码在 semantic.h 和 semantic.c 中。该结构被提取以单独形成文件 lexical.l。之后，根据附录 A 中的语法规则，我们基本上将每条语法规则封装成一个函数，Type 和 FieldList 是主界面。

每次在插入符号表之前执行查找操作，并在通过 typeCheck 函数进行类型检查之后，
通过考虑不同的情况。 更详细的解释反映在评论中。对于一些未定义的错误和与假设相矛盾的输入，我们做了尽可能多的兼容处理，基本原则这是一个未定义的错误，不会随机报告。另外，考虑到程序必须能够同时检查多个错误，在一些不可逆的情况下发现错误我们将返回 NULL，然后按照我们的方式处理接受 NULL 的语法。

实现及结果分析

1. Makefile 用于编译一个解析器程序，测试输入文件的词法和句法分析
    a) flex lexical.l
    b) bison -d -v syntax.y
    c) gcc main.c syntax.tab.c semetic.c -lfl -o parser
2. 只需调用以下命令即可调用所有命令制作
3. 可以使用以下任何命令使用测试文件测试解析器
    a) 使用解析器 ./parser Test/test1.cmm

```
┌──(kali㉿kali)-[~/Desktop/lab2]
└─$ cat Test/test2.cmm
int main(){
        float a[10][2];
        int i;
        a[5,3] = 1.5;
        if(a[1][2] == 0) i = 1 else i = 0;
}
```

```
┌──(kali㉿kali)-[~/Desktop/lab2]
└─$ ./parser Test/test2.cmm
Error type B at Line 4: syntax error.
Error type B at Line 5: syntax error.
```