实验报告

-- 编译原理 实验二

姓名：梁宇方 学号：171860695

邮箱：leungjyufong2019@outlook.com

a) 我的程序实现了所有的必做功能和所有的选做功能

b) 直接使用默认的 makefile 进行编译，即键入 make，即可完成编译

c) 实现细节与亮点

1. 使用多态的符号

使用枚举类型 skind 来标识表中的符号，然后用一个 union 存放数据区的指针。因此在进行符号表的插入时只需要拷贝数组区的指针，能够显著地提升符号插入的效率

```
struct Symbol {
    enum { S_UNDEFINED, S_VARIABLE, S_STRUCTNAME, S_FUNCTIONNAME } skind;
    union {
        Var* pvar;
        StructName* pstruct;
        FuncName* pfunc;
    };
    char sbname[MAX_NAME_LEN];
    int dec_lineno;
};
```

2. 使用二维符号表

第一维：使用函数栈帧来实现语句块的局部作用域；第二维：处理结构体嵌套的情况；初始状态为全局符号表(0,0)。遇到语句块左括号的时候纵坐标+1，遇到右括号的时候纵坐标-1；遇到结构体定义时横坐标+1，结构体定义结束时横坐标-1。在二维符号表中只需要检查横坐标是否等于零就可以区分普通变量和域变量。

3. 递归改迭代

对于语义分析中的简单递归，均可以改写为迭代形式。过程 StmtList 中，需要对右儿子递归地调用过程 StmtList；直接将表示状态的参数 node 赋值为它的右儿子即可；将简单的递归改写为迭代可以增加执行速度、节省栈空间

```
void StmtList(struct ast* node, const Type* ret_type) {
  // 右递归改迭代
```

```
    while (node->num != -1) {
        // StmtList -> Stmt StmtList
        Stmt(node->children[0], ret_type);
        node = node->children[1];
    }
    // StmtList -> empty
}
```

4. 容错处理

在对 Exp（即表达式）进行语义分析时，常常会遇到类型不匹配的问题，如果不进行处理就会使同一个表达式的其他部分也报错，从而出现多报的现象。正如下面的代码所列，对于 NOT 运算符的 Exp，无论前面计算的结果是否为 INT 类型，均会返回一个 INT 类型给上层。这样上层就能够假设这个 Exp 返回了正确的结果继续分析。

同理，当一个整型常量与一个不可辨认的标识符进行四则运算时，Exp 会返回一个 INT 类型给上层，即能够正确分析的子式的类型会作为整个表达式的类型返回。

```
if (!strcmp(node->children[0]->name, "NOT")) {
    // Exp -> NOT
    const Type* type = Exp(node->children[1]);
    if (!type || type->tkind != T_INT) semantic_error(7, node->lineno, NULL);
    return &INT;
}
```