编译原理实验二 实验报告

分工

小组由三位同学组成,分工如下:

1.初始构造阶段：符号表symbol构建：丁楚阳类型type处理：王文昊

2.语义逻辑semantic:付安胜后续代码调试：丁楚阳，王文昊文档撰写：付安胜

协作方式

小组建立了GitHub仓库,成员将本地测试成功的文件上传并与主分支merge,让每个成员都能时刻同步最新的代码,保证成员之间的有效交流,GitHub地址如下:

https://github.com/KaoriMiyazonoApril/NJU_Compiler_Design

编译方法

小组沿用了助教团队提供的makefile 文件,只需要在makefile文件目录下make即可得到parser可执行文件

功能和实现方法

小组采用分工协作，分别完成将符号表构建，类型处理，语义分析代码构建，通过对实验1生成的语法树进行遍历，完成对应的错误检查

符号表构建

采用栈加链表的形式储存symbol符号以及作用域的实现。

相关接口，功能在下面的注释中：

```
typedef struct var {
    Type *type;
} var;
typedef struct Symbol Symbol; // 提前告诉编译器 "有一个类型 Symbol 存在"
typedef struct func {
    Type *return_type;
    bool isDefined;//是否已经被定义了,不能重复定义
    int argNum;
    Symbol **arg_list;
} func;
typedef struct struc {
    int symbolNum;
    Symbol **symbol_list;
} struc;
typedef enum { VAR_KIND,
               FUNC_KIND,
               STRUCT_KIND } SymbolKind;
typedef struct Symbol {
    SymbolKind kind;
    char *name;
    Symbol *next;
```

```
    int lineno;
    union {
        var var_info;
        func func_info;
        struc struct_info;
    } info;
} Symbol; // 所有符号定义形成一张表
void enterScope();
void exitScope();
Symbol *createVariableSymbol(char *name, Type *type, int line);
Symbol *createFunctionSymbol(char *name, Type *return_type, int line, int argNum,
Symbol **argList);
Symbol *createStructSymbol(char *name, int line, int symbolNum, Symbol
**symbolList);
void freeSymbolStack(int tp);
void freeSymbol(Symbol *tofree);
Symbol *findSymbol(char *name);
void insertSymbol(Symbol *sym);
bool isInStruct(Symbol *sym, char *name); // a.x的x是否在a对应的类里,请传入a对应的类指
针
#endif
```

类型部分

类型部分定义类型结构体，实现类型定义、类型比较与结构体等复杂类型的处理

```
typedef struct Type {
    TypeKind kind;
    Type *base;//基类
    int arrayDim;
    int *arraySizes;//分别记录每一维的大小,之后可能有用
    Symbol* structType;//如果是结构体,指向结构体符号表
}Type;
bool TypeEqual(Type *a, Type *b);
bool isLValue(Node *exp);          // stub, 语义分析时定义
Type *getType(Node *specifierNode); // stub, 从 Specifier 节点中解析类型
bool StructEqual(Symbol *a, Symbol *b); // 比较两个结构体符号是否相等
```

semantic

遍历语法树，完成对应的类型检查，相应的函数作用如下注释。

核心思路，semanticAnalysis()为入口函数，各traverse*函数根据语法规则处理对应的语法结构，chekExp()处理表达式的类型检查等核心错误的识别。

语义错误输出遵循实验要求格式：

Error type X at Line Y: <错误描述>.

```
void semanticAnalysis(Node *root);//主程序入口，进入语法树分析
/**
 * 释放语义分析的所有资源
 */
void freeSemanticResource(void);
// 前向
```

```
// Program → ExtDefList
void traverseProgram(Node *node);
// ExtDefList → ExtDef ExtDefList | ϵ
void traverseExtDefList(Node *node);
/// ExtDef → Specifier ExtDecList SEMI
///        | Specifier SEMI
///        | Specifier FunDec CompSt
void traverseExtDef(Node *node);
/// ExtDecList → VarDec | VarDec COMMA ExtDecList
void traverseExtDecList(Node *node, Type *baseType);
//函数定义
void handleFuncDef(Node *funDec, Type *retType, Node *compSt);
//复合语句块处理：进入作用域->处理->推出作用域
void traverseCompSt(Node *node, Type *retType);
// DefList → Def DefList | ϵ
void traverseDefList(Node *node);
// Def → Specifier DecList SEMI
void traverseDef(Node *node);
// DecList → Dec | Dec COMMA DecList
void traverseDecList(Node *node, Type *baseType);
// StmtList → Stmt StmtList | ϵ
void traverseStmtList(Node *node, Type *retType);
// Stmt → Exp SEMI
void traverseStmt(Node *node, Type *retType);
//表达式检查，遍历语法树错误检查的最后部分
Type *checkExp(Node *exp);
//处理结构体类型
Type *handleStructSpecifier(Node *node);
//插入函数参数
void insertFuncParams(Node *funDec);
//检查函数调用
bool checkFuncCall(Symbol *funcSym, Node *argsNode, int line);
//获取结构体域
Symbol *getStructField(Symbol *structSym, char *fieldName);
```