《编译器构造实验》lab2报告

20331041　徐锶达　xusd3@mail2.sysu.edu.cn

1 实现功能

1.1 完成内容

- **所有必做内容**：检查并打印错误类型1-17
- **所有选做内容**：满足要求2.1，对应错误类型18、19；满足要求2.2、2.3，修改对应错误类型1-17
- 满足修改后的假设1-7，其中对于假设7中的"不同结构体中的域互不重名"，不满足的情况将打印错误15
- 个性化打印效果

1.2 符号表

- 基于**十字链表**和 **open hashing 散列表**的 **Imperative Style** 的**符号表**设计
- 在具体实现中，`global_table` 对应 Hash table，`scope_table` 对应 Stack，构成符号表的主体

1 global_table

```
struct Symbol_Stack_ global_head[SYMBOL_LEN];
```

- 插入时，将新元素插在对应位置的链表的表头
- 删除采用"虚拟节点+迭代"的方法
- 存放所有符号信息
- 实现了打印函数 `show_global_table()`

2 scope_table

```
Symbol_Stack scope_head = NULL;    Symbol_Stack scope_tail = NULL;
```

- 表头永远指向全局作用域；还维护一个表尾，方便添加和删除作用域
- 退出作用域时将每个节点从 global_table 删除，最后删除该作用域
- 实现了打印函数 `show_scope_table()`

3 struct_table

```
struct Symbol_Stack_ struct_head[SYMBOL_LEN];
```

- 结构体除了会插入global_table，还会额外插入struct_table，方便一些查询操作
- 实现了打印函数 `show_struct_table()`

4 func_dec_list

`Func_List func_dec_head = NULL;    Func_List func_dec_tail = NULL;`

- 函数声明链表，用于检查每个声明过的函数是否定义了

1.3 细节

- **多维数组**和**结构体嵌套**采用**链表**表示
- 所有结构体及其域的定义默认都在全局（depth为0），保证不同结构体不能重名，不同结构体的域不能重名
- 结构体本身是结构体类型，结构体变量是变量类型
- 在词法分析部分新增终结符的行号信息，方便语义分析打印错误
- 在语法分析部分新增"函数声明"的产生式：`ExtDef -> Specifier FunDec SEMI`
- 类型比较：`type_check()` 函数
  - 结构体等价：采用**结构等价**
  - 数组比较：采用**弱等价**，即当维数相等即等价。但在结构体等价的判断中，如果结构体内定义了数组，则采用**强等价**，即数组的元素类型、每一维的大小、维数都必须相同

1.4 语义分析

- 维护深度信息和全局作用域
- 针对部分产生式特点书写递归函数
- 将错误信息的打印封装于函数 `semantic_error_handler()`

1 多维数组处理

- `VarDec_check()`：在`vardec`中的遍历顺序和要链接的链表顺序是相反的，具体而言对于`a[10][3][2]`访问的顺序是`2->3->10->ID`，但是要组成的是`ID->10->3->2->INT`这样的链表结构
- 首先进行一次遍历获得深度，然后根据这个深度`malloc`一个`type_list`数组，然后再遍历一次填写数组，最后根据这个数组组装成链表

2 结构体嵌套处理

- `Specifier_check()`：在匹配产生式`StructSpecifier -> STRUCT OptTag LC DefList RC`后，在`DefList`可能还会有新的结构体定义，因此额外书写递归式函数`DefList_struct_check()`、`Def_struct_check()`、`DecList_struct_check()`，和最后的`Dec_struct_check()`
- 总体而言，是通过函数的返回值将`FieldList`链表串起来
- 在`Def_struct_check()`中，通过再次调用`Specifier_check()`得到类型`Type`，可以递归地处理结构体嵌套的情况。因为在遇到子结构体时，会先去封装其类型`Type`，返回后再继续进行原结构体的域链表的串联

2 Makefile编译

- 进入`Code`文件夹：运行：`make`
- 清理编译生成文件，运行：`make clean`

3 测试

1️⃣ **单个文件手动测试**，运行：`./parser test.cmm`

若`test.cmm`与`parser`不在同一文件夹下，需自行添加路径。

2️⃣ **多个文件脚本测试**，运行：`./test.sh`

将多个测试文件（**只分析后缀为.cmm的文件**）统一放在`Test`文件夹下，运行**测试脚本**`test.sh`。

注意默认情况是`parser`在`Code`文件夹下，`Code`文件夹与`Test`文件夹同级。不同则自行修改脚本中的`path`（测试文件所在文件夹相对路径）。