编译原理Lab2实验报告

匡亚明学院
181240035 刘春旭
181240035@smail.nju.edu.cn

实验目标和结果

在词法和语法分析的基础上自己定义符号表结构，编写程序，对C--的源代码进行语义分析和类型检查。结果完成了所有基础要求和附加任务，并且在学长学姐们编写的测试程序中能匹配出（几乎）所有样例（除了过分长的嵌套类型检查）。

> 编译方式和原 `Makefile` 无区别，并在README中有说明。

代码框架

本次新增的代码文件有两个，分别为 `common.h` 负责数据结构的定义，以及 `hash.c` 负责所有符号表的填写和类型检查，以下将分开两部分进行说明：

1. `common.h` 文件：
    1. 符号表 `Table`：由表项指针数组组成的符号表和栈组成：（`Table.hashTable` 以及 `Table.stack`）
        - 表项：

```
struct TableEntry_{
  int sym_depth;
  char *sym_name;
  Type *sym_type;
  int is_struct_def;
  TableEntry *next_hash; // hash collision
  TableEntry *next_stack; // same depth
};
```

其中的is_struct_def稍显不协调，这是由于在C--源代码中出现的所有函数定义、结构体定义、结构体变量、变量都会插入同一张表。在这种情况下是必须要有手段将结构体定义和结构体变量区分开的。因为虽然结构体定义和变量不能重名，但是结构体定义和结构体变量的类型是相同的。为了防止检查类型的函数将结构体定义和变量误认为类型相同，需要在表项中加入一个新的指示变量，将结构体定义和结构体变量区分开。

2. FieldList和Type基本如书中例子，但是需要对Type类型进行小小改造：

```
struct Type_{
  enum{ BASIC, ARRAY, STRUCTURE, FUNCTION } kind;
  union{
    int basic;
    struct {Type *elem; int size; } array;
    struct {FieldList *member; int is_defined; }structure;
    struct {int param_cnt; FieldList *param; Type *return_type; int is_defined; int first_line;}
func;
  };
};
```

- BASIC变量：用 `basic` 记录其基本类型（ `int` / `float` ）
- ARRAY变量：用 `array` 记录数组的大小和类型信息，和讲义相同。
- STRUTURE变量/定义：用 `structure` 来记录其成员；其中的 `is_defined` 是为了防止嵌套结构体定义中的结构体名称复用（Error 16）. 即在看到结构体定义时，不管它其中成员有什么，先把它插到符号表里。由于表中内容存的全都是指针，所以先插表后修改和先修改后插表没有任何区别。
- FUNCTION定义：用 `func` 来记录它的参数个数和类型信息以及返回值。最后的 `is_defined` 和 `first_line` 是用来检查错误类型18，函数声明却未被定义。

3. `#define Assert(cond) ...`

这个在ics PA中救我多次的 `Assert` 在本次实验中也防止了太多次指针飘飞导致的Segmentation Fault，我认为有必要在实验报告中给它一个credit

4. `#define IS(a, b) strcmp(a->token_name, b)==0`

可以更方便的判断节点，想到这个宏之后非常后悔，后悔定义晚了。因为这个实验中需要判断节点的情况实在是太多了，有了这个宏之后非常方便。

2. `hash.c` 文件：

有了上次实验语法树的铺垫，我们可以顺着树根一层层地递归分析，和打印树的顺序相同。

只要把`ExtDef`里涉及到的所有查表、用表操作全部检查一遍即可。只要按照产生式逐层对每个节点进行分析，就能分辨出是定义、声明结构体、函数、变量还是使用函数、变量。按照产生时分析还有一个好处就是可以编写模块化很强的、复用程度很高的函数。

函数的返回类型主要是`Type *`，`TableEntry *`，`int`，`void`

比如`Type *Specifier(struct Node *spec)`返回的永远是某个类型的指针，并且还可以做这些：

- 如果是BASIC类型的变量，则返回其类型；
- 如果是结构体变量的定义，将表项整理好之后插入符号表，并且返回结构体变量的类型；
- 如果是结构体变量的使用，检查对应结构体是否定义，返回结构体变量类型；
- 如果是函数变量/数组的类型，会将其类型中的成员整理好之后返回，等待插入/使用。

再比如`TableEntry *VarDec(struct Node *vardec)`就可以返回一个准备好了的表项，再决定是插入还是不插入。如果出现Error选择不插入，那么就立刻将返回的地址`free()`掉，这样就不会造成内存的浪费。

`Int`是指示型函数，比如`compare_type(Type * t1, Type* t2)`用来比较两个符号的类型。由于`Specifier`对于任何有类型的符号都能返回其正确的`Type *`，则所有符号的类型都是可以进行比较的。这个函数的返回值有三种：0，1，2；0代表类型不同，1代表类型相同，2代表至少其中一方是空指针，说明肯定在进行类型比较之前的某个地方就报了错，所以不需要再报类型不匹配的错误了。

`void`是功能型函数，比如`void Stmt(struct Node * stmt)`只负责调用其他类型的函数进行变量的注册和使用（查询）操作。

另外本次实验中可以利用很多`do{...}while( );`来简化代码结构，在大部分情况下要比`while( ){...}`的结构简洁一些。

测试结果

非常感谢学长学姐们的测试代码，帮助我找到了很多bug。最终的结果是基本全部可以通过。这里的“基本”是指嵌套的深度太深的时候可能会导致TLE，比如说学长学姐测试代码中的ZZ01和ZZ03，包含多个结构体类型的嵌套比较。即使假设每个结构体内部只有n个结构体定义，嵌套m层，那么分析的时间就是$O(n^m)$，目前没有解决掉这个问题。