编译原理 实验二

符号表

更改实验一中的符号表为属性表

设计符号表如下

```
typedef struct SymbolItem *Symbol;
typedef struct SymbolInfoItem *SymbolInfo;

struct SymbolInfoItem {
  enum { TypeDef, VariableInfo, FunctionInfo } kind;
  char name[64];
  union {
    Type type;
    struct {
      SymbolInfo arguments[MAX_ARGUMENT];
      size_t argument_count;
      Type return_type;
    } function;
  } info;
};

struct SymbolItem {
  SymbolInfo symbol_info;
  Symbol tail;
};
```

符号有三种类型

- 变量
- 结构体定义
- 函数

对于变量和结构体而言，只需要存储 Type 即可

对于函数而言，需要存储参数和返回值的 Type，实际上参数存储的是符号类型

内存泄露

使用 `log_malloc` 记录整个编译程序中的动态内存分配

```
void *log_malloc(size_t size) {
  if (malloc_table_size == 0) {
    malloc_table = (void **)malloc(INIT_MALLOC_SIZE * sizeof(void *));
    malloc_table_size = INIT_MALLOC_SIZE;
  }

  if (malloc_table_index == malloc_table_size) {
    malloc_table =
        (void **)realloc(malloc_table, malloc_table_size * 2 * sizeof(void *));
    malloc_table_size *= 2;
  }
```

```
12
13    void *res = malloc(size);
14    malloc_table[malloc_table_index] = res;
15    ++malloc_table_index;
16    return res;
17  }
```

并且 `malloc_table` 也可以动态调整大小

另外在声明的过程中常常会使用同一个 Type 或 Symbol

为此使用 `memcpy` 的方式完整复制整个结构体，防止多个指针指向同一个结构体，导致 double free

> 较大的内存开销

模块化

仿照 OS Lab 里面的 `MODULE` 和 `MODULE_DEF`，设计模块如下

```
1   typedef union Attribute attribute_t;
2   typedef struct Ast ast_t;
3   MODULE(parser) {
4     attribute_t (*get_attribute)(size_t);
5     void (*print_ast_tree)();
6     ast_t *(*get_ast_root)();
7     void (*restart)(FILE *);
8     int (*parse)();
9     int (*lex_destroy)();
10  };
11
12  MODULE(analyzer) {
13    void (*semantic_analysis)(ast_t *);
14    void (*print_symbol_table)();
15  };
16
17  MODULE(mm) {
18    void *(*log_malloc)(size_t);
19    void (*clear_malloc)();
20  };
```

范式

通过遍历语法树的方式进行语义分析

其范式如下

```
1   void foo(struct Ast *node) {
2     action("hit %s\n", unit_names[node->type]);
3     assert(node->type == _UNIT);
4
5     if (check_node(node, count, ...) {
6       ...
7       return;
8     }
9
10    assert(0);
11    return;
12  }
```

使用 `check_node` 检测结点结构

`Specifier` 的设计

主要体现在解析 specifier 和 struct_specifier 中

一些约定

```
int -> VariableInfo
struct A -> VariableInfo
struct {...} -> VariableInfo
struct A {...} -> TypeDef
```

这样就可以在使用 specifier 的类型时，根据不同情形进行处理

典型的例子是 `external_definition`