# 编译原理实验 2 实验报告

鄢振宇（基础班）　　171240518

2020 年 4 月 12 日

## I　代码框架

```
/Lab
    /Code #存放代码的文件夹
        Makefile    #用于编译的文件
        common.h    #常用的宏/函数的声明/定义
        error.c     #用于报错的相关函数的定义
        error.h     #用于报错的相关函数的声明
        handlers.c #包含若干接受node*的函数的定义
        handlers.h #包含若干接受node*的函数的声明
        lexical.l   #词法处理工具flex的源代码
        main.c      #包含main函数
        syntax.y    #语法处理工具bison的源代码
        table.c     #符号表相关函数的定义
        table.h     #符号表相关函数的声明
        test.c      #单元测试相关函数的定义
        type.c      #类型相关函数的定义
        type.h      #类型相关函数的声明
    README #框架包含的一个自述文件
    /Test #存放测试文件的文件夹，.cmm后缀表示C—源代码
          #.out后缀表示预期的输出（用于本地测试用）
        ...
        lab2-*.cmm #实验二相关测试
    parser      #预先编译好的二进制文件
    report.pdf #报告
```

## II 我的程序应该如何被编译

使用给定的 Makefile

## III 我的程序实现了哪些功能

实现了语义错误的检测。尝试并解决了大部分内存回收。（但是提交的版本中为了安全起见，将内存回收部分注释了，可以在 main.c 的 32-36 行重新开启）

## IV 自认为的亮点

### 函数指针实现"OOP"

在实验 2 中，我们需要根据节点类型/产生式的不同，进行不同的操作。纵使我们可以通过在节点里记录当前节点的类型，然后通过 if-else 来进行处理。但是我觉得这种写法的可读性较差，且会导致一个函数非常非常冗长，也会导致一些跨函数的“协作”难以进行

我的实现方式是,在 node 结构体中加入一个 func 成员,它是一个 void(*handler)(node*) 类型的函数指针。也即，它接受一个 node* 作为参数，返回一个 void*（返回 void* 的目的是 C 语言可以隐式地将 void* 转换成其他类型。并且我的架构设计可以保证，调用者是可以确定应该如何处理/解读这个 void* 的，不会发生运行时错误。由于实验 2 的特性，实际上返回的类型大都是 Type_*）

func 成员的赋值在语法分析阶段完成，以数组访问为例（func 由产生式而非节点类型决定，不同节点也可能用同一个 func）

```
| Exp LB Exp x_RB     {$$ = Node4("Exp");$$ -> func = array_access_handler;}
```

在数组访问中，我们只需要求出 0 号和 2 号后继（我使用的是多叉树结构）的 Type 即可进行类型的判断。由于我的架构的设计，我们只需要用 exp1 -> func(exp1) 就可以得到 exp1 对应的类型

```
make_handler(array_access) {// cur : Exp LB Exp RB
    node* exp1 = cur -> siblings[0];
    Type base = exp1 -> func(exp1);
    if(base -> kind != ARRAY) {
        semantic_error(exp1 -> lineno, NOT_ARRAY);
        return NULL;
    }
    node* exp2 = cur -> siblings[2];
    Type index = exp2 -> func(exp2);
    return type_check(type_int, index, base -> array.elem, exp2 -> lineno, NOT_INT);
}
```

**实现 typecmp 和 type_check 提升可读性、减少 copy-paste**

在实验 2 中，我们需要经常判断类型是否符合期望，并给出相应的报错。所以我定义了 typecmp，用于递归地比较两个类型是否可以被认为等价。type_check 通过调用 typecmp 比较两个类型是否等价。不等价则进行报错。

```
Type type_check(CType lhs, CType rhs, CType ret, int lineno, semantic_errors err, ...);
```

为了适应各个不同地方的需求，type_check 的参数列表较长，但是**几乎所有**类型判定都由 type_check 进行。type_check 会比较 lhs 和 rhs 两个类型，如果不等价，则根据 lineno, err 和可变参数表进行相应的报错。如果等价且 ret 为 NULL，则返回 lhs（此时 lhs 和 rhs 等价，返回哪个都无妨）。如果等价且 ret 不为 NULL，则返回 ret。

```
return type_check(lhs, rhs, NULL, cur -> siblings[0] -> lineno, ASSIGN_MISMATCH);
```

赋值运算的 type_check

```
return type_check(lhs, rhs, type_int, cur -> siblings[0] -> lineno, OPERAND_MISMATCH);
```

关系运算的 type_check，由于返回类型一定是 INT，所以参数三不为 NULL

```
return type_check(type_int, index, base -> array.elem, exp2 -> lineno, NOT_INT);
```

数组访问的 type_check

## 对内存回收的尝试

实验 2 中，我觉得内存回收的难点是 char* 的归属权，Type_* 的归属权

我的处理方式是，向符号表插入 char* 和 Type_* 后，Type_* 的归属权改为符号表，char* 的归属权不变（并且向符号表插入的 char* 都属于语法树）

因此，对于变量，我们在删除时并不需要 free 它们的 Type_* 或者 char*。由符号表和语法树进行对应的 clear。

但是还存在一个问题，如果有 struct 嵌套的情况，外层的 struct 一定要先 free，否则在遍历它的结构时，就会访问到已经被 free 的成员。这个问题可以通过对 struct 作拓扑排序（或者直接按照行号逆序清理），但由于和实验 2 关系不大，我就没有继续处理下去了。

我使用 mtrace 和 muntrace 进行 malloc 内存的统计。除去 printf, yyparse 申请的若干个 0x400 大小的缓存外，如果没有 struct 互相嵌套的情况，可以做到完全无内存泄漏

## 其他小亮点

- 缓存了基本类型 type_int 和 type_float，需要时可以直接使用，无需重新分配动态内存

- 通过 typedef const struct Type_* const CType 定义了不可修改的 Type，使代码更加安全（囿于个人水平，有三四处用了强制类型转换转回 Type）
- 定义宏 new，使得申请动态内存时更安全（防止 malloc 返回的 void* 隐式转换产生隐患）

```
#define new(type) (type*)malloc(sizeof(type));
//avoid code like
Type t = malloc(sizeof(Type));
```

- 定义了宏 remove_access，在遍历链表的同时完成内存清理

```
static inline void* free_first(void* ptr1, void* ptr2) {
    free(ptr1);
    return ptr2;
}
#define remove_access(vari, field) \
    free_first(vari, vari -> field)
```

for 循环只需写成

```
for(struct LNode* cur = cur_list -> dec;
    cur;
    cur = remove_access(cur, next))
```

- 在符号表中，将 struct 的深度设为 UINT_MAX。则变量如果与结构体重名会报错（insert 前会检查是否有 depth>= 当前深度且同名的表项）
- 使用 make_handler 宏定义函数 handler.c 里的大多数函数（受 ics 课程的 PA 代码影响）

```
#define make_handler(name) \
    void* name##_handler(node* cur)
```

- 如果发生类型错误，type_check 亦或是 handler 都会返回 NULL，调用者可以通过该信息减少过多的报错