编译原理实验 2 报告

姓名：黄奕诚　　**学号**：161220049

1　开发环境

- 操作系统：macOS Mojave 10.14.4
- 编辑器：Visual Studio Code 1.32.3
- C 编译器：Apple LLVM version 10.0.1 (clang-1001.0.46.3)
- Flex 版本：flex 2.5.35 Apple(flex-31)
- Bison 版本：bison (GNU Bison) 2.3

注：开发完成后已部署到 Ubuntu 18.04 上进行再测试

2　功能介绍

- 完成基本要求
- 完成函数声明、作用域嵌套、结构等价三个选做要求，并能判定相应的语义错误

3　编译运行方法

- 编译：在 `Code/`目录下运行 `$ make` 即可，随后在项目根目录生成可执行文件 `parser`. **由于我在 macOS 上进行开发，将 `Makefile` 中编译选项`-lfl` 改为`-ll`，如果编译遇到问题不妨检查一下这里.**
- 运行：在项目根目录执行 `$ ./parser` [测试文件名] 即可

4　实现中重点细节

4.1　数据结构

我维护了两个哈希表，一个用来存储结构体和变量信息，另一个用来存储函数信息。哈希表的每个元素为一个链表，维护重名的那些变量/结构/函数。

1. 变量/结构体的结构体如下：

```
struct Field_List {
  char name[MAX_NAME_LEN];
  struct Type *type;
  struct Field_List *next;
  int wrapped_layer;
  int is_structure;
  int line_num;
};
```

其中,name 是名字;type 是类型;next 是挂载在同名链表的下一个结点;wrapped_layer 是该变量/结构体所属的作用域层数; is_structure 表明它是一个结构体类型还是变量; line_num 记录了它出现的行号，便于报错。

2. 函数结构体如下:

```
struct Func {
  char name[MAX_NAME_LEN];
  struct Type *return_type;
  int defined;
  int param_size;
  struct Field_List *first_param;
  int line_num;
};
```

其中，name 是名字; type 是函数的返回值类型; defined 标识该函数是否已经被定义（或只是被声明）; param_size 表示函数参数的数目; first_param 指向第一个参数（可为空），维护参数链表; line_num 维护函数的行号，便于报错。

3. 类型结构体如下:

```
struct Type {
  enum {
    BASIC, ARRAY, STRUCTURE
  } kind;
  union {
    int basic;
    struct {
      struct Type *elem;
      int size;
    } array;
    struct Field_List *structure;
  } u;
  struct Type *next_ret_type; // multiple return type in a
CompSt
  struct Type *next_actual_param; // next actual param
  struct Type *next_flat; // struct type to non-struct flat type
 list
  int line_num; // multiple return type in a CompSt
};
```

其中，kind 维护类型（基本类型、数组、结构体），数组类型进行递归维护，结构体的 u.structure 指向结构体中第一个域（可为空）。next_ret_type 用于函数体中维护的 return type list 链表的结点，在函数体结束后可以逐一检查返回值类型（因为函数体中可能有多个返回值）。next_actual_param 用于实参列表中下一个实参类型。next_flat 用于比较结构体的类型等价时，指向的下一个非结构体类型。line_num 维护类型的行号，便于报错。

4. 语义错误链表结构体如下：

```
struct Sem_Error_List {
  int type;
  int line_num;
  char info[MAX_ERROR_INFO_LEN];
  struct Sem_Error_List *next;
};
```

其中，type 表示错误类型，line_num 记录错误行号，info 保存报错信息，next 指向下一个语义错误。

4.2 讲义中未 well-defined 的语义规约

1. 我规定当函数返回值为结构体且该结构体有类型名时，这个类型名的作用域在外围。如

```
int a;
struct a {int b;} func() {int a; }
```

此时 struct a {int b;} 与全局变量 a 名称冲突，而与函数体内的局部变量 a 不冲突。

2. 当一个表达式有多个类型不匹配错误时，如表达式

```
int main(){
  int a = (15 * 5.4) + (4 - 0.9);
}
```

此时我规定只报最先触发的类型不匹配错误，这里即为 15 * 5.4。因为其类型不匹配错误而像多米诺骨牌一样发生的其它类型不匹配错误不再报出。也就是说，在一个大表达式中，类型不匹配错误至多报出一个。

4.3 留下深刻印象的 bug

1. 各种空指针访问导致的段错误。有了 GDB 的 bt 可以快速定位到错误现场。好久没写 C，真的难受（瘫

2. 当一个结构体身处多个链表时，它的几个 next 指针一定要好好考虑，写昏头了就会把几个链表弄乱。就像我实现作用域嵌套的时候，发现如下测试代码中的问题

```
1       int main(){
2         int a;
3         {
4           float a;
5         }
6         a = 1;
7       }
```

报出了第 6 行中 a 未定义，经过打了一堆 Log 筛查，发现 push 和 pop 操作没有任何问题，最后定位到了一个神奇的地方把 next 指针修改了，导致第二个结点开始的内容全丢了。

5 建议与吐槽

1. 在实验发布后第二天花了一天写完了，脑子一热一口气写了六七百行，一口气编译居然没报错，然后测讲义上 23 个样例竟然只错了一个。事实证明是讲义的样例太弱了。hyf 大佬看见我写完了，编了一堆反人类的测试样例 hack 我，一堆 bug 浮出水面。寻思着写个自动生成语法正确的测试样例的程序，但好像有个师兄在做这事了 23333

2. 一个编译器实验要做到 well-defined 真的太难了。这个实验很难，不是难在实现或是思路上面，而是难在各种复杂情形的完整考量。我觉得我考虑得还不够完整，当然实验讲义也没做到……

3. 我还是觉得“错误所在行号要求正确”这个要求很奇怪。比如说

```
1     int main(){
2       int
3       a
4       =
5       1
6       *
7       1.6
8       -
9       0.7
10      /
11      5
12      ;
13    }
14
```

这里应该报多少错，以及报在第几行。我的实现是报出了第 6 行的运算符两边类型不匹配。当然，助教们进行人工测试的话，这类错只要能找到根源，类型正确，不论连锁地报了多少错都应该不影响得分。但是这个要求很难做到自动测试和自动评分（可能是我太抠了）。