编译原理 实验一

命名约定

为了区分枚举和 flex 与 bison 定义的一些单元，有如下约定

- `common.h` 中定义了 terminal 和 non-terminal，以下划线开头
- flex 中定义的正则表达式的别名为小写
- bison 中定义的 terminal 和 non-terminal 与 `common.h` 一致，只不过没有下划线

另外在 `common.h` 中使用 X-Macros 快速生成单元名以供打印

```
#define SYMBOL_NAME(name) [_##name] = #name,
static const char *symbol_names[] = {SYMBOL_KEYS(SYMBOL_NAME)};
```

语法树结构

目前设计如下

```
struct Ast {
  int lineno;
  int symbol_index;
  int type;
  size_t children_count;
  struct Ast *children[MAX_CHILDREN];
};
```

- lineno → 行号
- symbol_index → 在符号表中的下标，若无则为 `-1`
- type → 对应的终结符或非终结符的枚举值
- children_count 和 children 形成多叉树结构，由于最大分支数预先可知，使用 MAX_CHILDREN 宏控制

符号表为一个联合体

```
typedef union {
  int _attr;         // for TYPE or RELOP
  unsigned _int;     // for INT
  float _float;      // for FLOAT
  char _string[64]; // for ID
} Symbol;
```

`lexical.l` 的 `install` 函数会填充符号表，对终结符的属性进行赋值

而在 `syntax.y` 中产生式的语义部分使用如下函数

```
extern void make_root(struct Ast **root);
extern void make_node(struct Ast **node, int type);
extern void make_children(struct Ast **root, int count, ...);
```

使用**变长参数**以统一处理

内存泄露

使用 valgrind 诊断

```
$ valgrind --leak-check=full --show-leak-kinds=all --error-exitcode=1 ./parser ../Test/temp.c
```

三个方面

- 语法树结构

使用 `clear_tree` 函数

依赖于记录了每次动态内存分配的节点表

```
void *node_table[MAX_NODE];
size_t node_table_index = 0;
```

还有 log_malloc 函数

```
void *log_malloc(size_t size) {
  void *res = malloc(size);
  node_table[node_table_index] = res;
  ++node_table_index;
  return res;
}
```

由于出现错误后语法树的结构可能无法正确保持，所以引入了节点表

在出现错误后，有两个选择，一是继续分配内存，二是停止分配

但是两个选择目前似乎都没有问题

- lex 和 bison 的内部结构

使用 `yylex_destroy` 函数

- 打开的文件

使用 `fclose` 函数

不使用 `yyin`

一键回归测试

额外写了一个 shell 脚本进行一键回归测试，依赖于 bat 命令行工具

```
TEST=$(find ../Test/$1 -name "*.c" | sort)
TESTARR=(${TEST})

make clean
make parser

for i in "${!TESTARR[@]}"
do
  echo -e "\e[1;35m./parser ${TESTARR[i]}\e[0m"
  bat ${TESTARR[i]}
  ./parser ${TESTARR[i]}
  echo -e "\n"
done
```

可以这样使用

```
$ ./test.sh
$ ./test.sh err
$ ./test.sh err/err7.c
```