编译原理 实验三

IR 模块

```
MODULE(ir) {
  void (*ir_translate)(struct Ast *);
  void (*ir_generate)(FILE *);
};
```

和语义分析类似，仍采取遍历语法树的方案，只不过侧重点不同

在已有符号表和没有全局变量的前提下，只需考虑

```
ExtDef -> Specifier FunDec CompSt
```

注意到

```
CompSt -> LC DefList StmtList RC
```

对于 `DefList`，只需考虑有赋值的情形即可

其余没有赋值的情形，在**使用**时才会进行初始化

> *lazy load*

操作数表示

```
typedef struct OperandItem *Operand;
struct OperandItem {
  enum {
    OP_FUNC,
    OP_ADDRESS,
    OP_ADDRESS_ORI,
    OP_ADDRESS_DEREF,
    OP_VARIABLE,
    OP_CONSTANT,
    OP_LABEL, // for label
  } kind;
  union {
    size_t placeno;
    unsigned value; // for constant, note unsigned
  } u;
  Type type;
};


```

对操作数定义了多种类型，主要有两类

- constant
- non-constant

其中 non-constant 对应的字符串使用 placemap 存储
为了在生成中间代码时能够正确的反映出 `*` 或 `&` 等前缀，进一步将 non-constant 细分

- `OP_FUNC / OP_VARIABLE / OP_LABEL / OP_ADDRESS` - 无前缀
- `OP_ADDRESS_ORI` - 前缀为 `&`
- `OP_ADDRESS_DEREF` - 前缀为 `*`

错误处理

使用 setjmp 和 longjmp 完成非本地跳转，在 IR 模块翻译入口函数处建立快照

```
1  static void translate(struct Ast *root) {
2    if (!setjmp(buf)) {
3      init_ir();
4      build_int_type();
5      translate_program(root);
6    } else {
7      printf("IR Error: Translation fails.\n");
8      ir_errors++;
9    }
10 }
```

主模块使用 `ir_errors` 判断中间代码是否正确生成

placemap

本质上是个字符串常量池
将所有的字符串分为 5 类，分别为

- 临时变量名 `ir_tmp_no`
- 标号名 `ir_label_no`
- 函数名 `xxx`
- 变量名 `ir_var_xxx`
- 形参地址变量名 `ir_addr_xxx`

对于临时变量名和标号名，其后缀为不重复的数字，使用 `next_anonymous_no` 记录
其余则显式传入后缀，并且通过 `load_placeno` 复用