实验三：中间代码生成 实验报告

成员	学号	院系	邮箱
张路远	221220144	计算机学院	craly199@foxmail.com
姜帅	221220115	计算机学院	jsissosix1221@gmail.com

实现方法

操作数的表示

操作数是中间代码生成的核心元素，用于表示变量、常量、地址等。为此，设计了 `Operand_` 结构体，结构如下：

```
struct Operand_ {
    enum { VARIABLE, TEMP_VAR, CONSTANT, REFERR, DEREFERR, LABEL, FUNC

    } kind;
    union {
        int no;          /**< Temporary variable or label number */
        int value;       /**< Constant value */
        char name[32];   /**< Variable or function name */
        Operand opr;     /**< Nested operand (e.g., for address/value
ops) */
    };
    Type type;           /**< Type of the operand */
    bool is_addr;        /**< Whether this operand represents an address
*/
};
```

`Operand_` 包括①操作数的种类(如变量,临时变量,取地址&x, 解引用*p等等) ②使用联合体存储不同类型操作数的数据(如临时变量或标签的标号 `no`, 常量的值 `value` 等等) ③记录操作数的类型信息的字段 `type` 和地址标记字段 `is_addr`

中间代码的表示

中间代码采用三地址码形式，在本次实验中我们使用线性IR, 使用**双向循环链表**将一条条IR组织起来, 在中间代码生成的过程中, 每解析出一条IR, 就将其添加到链表的尾部在内存中暂存, 而不是直接打印出来, 这种策略的目的在于对IR的可控性, 我们可以对最终生成的IR进行人为的控制, 以便于后续的优化例如不可达变量分析和死代码消除等.

通过 `CodeContent_` 和 `InterCode_` 两个结构体实现指令内容的表示与链表组织, 结构如下:

```
struct CodeContent_ {
    enum {
        LABEL_,    /**< LABEL x : */
        FUNC_,     /**< FUNCTION f : */
```

```
        ASSIGN_,  /**< x := y */
        ...
    } kind;
    Operand ops[3]; /**< Operands used by the instruction (up to 3) */
    union {
        char relop[4]; /**< Relational operator (used in IF_GOTO) */
        int size;      /**< Memory size (used in DEC) */
    };
};
struct InterCode_
{
    CodeContent content; /**< Content of the instruction */
    InterCode prev;      /**< Previous instruction (linked list) */
    InterCode next;      /**< Next instruction (linked list) */
};
```

`CodeContent_`结构体表示了本条IR的内容, 包括 ①指令种类, ②操作数数组(至多三个,故使用Operand[3]表示), ③元数据(关系运算符和内存分配大小)

`InterCode_`结构体代表指令链表, 内容字段为`CodeContent_`, 链表指针(`prev, next`)实现双向链表，支持指令序列的插入、删除和遍历。

中间代码生成流程

参考实验二，接收语法树的根节点，以递归调用的方式创建代码，每个`translate`函数的返回值为该语法树节点对应的中间代码段的**首行代码**对应的链表节点(`InterCode`类型), 通过`insertCode`函数进行拼接. 具体每个`translate`函数的翻译方案参照实验手册.

选作任务思路

我们的选作任务为4.1: 可以出现结构体类型的变量, 结构体类型的变量可以作为函数的参数, 由于实验二中我们的实现支持这种语法, 我们延续实验二的设计，在`Type->kind`中特别设置了`structure`，方便我们判断结构体，从而进行处理。同时注意到DOT现在只会用于结构体域访问，也可以用于条件判断. 对于不应该出现的情况,即一维数组不可做函数参数, 不可以出现高维数组变量, 由于我们的实验二依旧支持这类语义, 我们采用的方案是不回滚实验二已经完成的部分，而是使用一个全局标志位, 一旦检测到这类语法,该标志位就可以提示main函数不生成中间代码。该解法仅针对实验三而设计, 是一种面向实验要求的方法, 存在优化的空间.

其他

我们在完成代码后使用大模型为代码附加了注释, 以增强代码的可读性和规范性.