实验 3　　191220154　　张涵之

a) 实现功能
   a) 必做内容（能处理假设 1-7 的中间代码生成）；
   b) 选做内容（要求 3.2）；

b) 实现思路
   a) 在 lib.h 和 lib.c 中定义了新的数据结构 Operand，InterCode 等。

```
struct Operand_ {
    enum { VARIABLE, TEMPVAR, CONSTANT, ADDRESS, POINTER, LABEL, FUNC, OTHER } kind;
    union {
        int var;
        char value[32];
    } u;
};

struct InterCode_ {
    enum { _LABEL, _FUNC, _ASSIGN, _ADD, _SUB, _MUL, _DIV,
        _GOTO, _IFGOTO, _RETURN, _DEC, _ARG, _CALL, _PARAM, _READ, _WRITE
    } kind;
    union {
        struct { Operand right, left; } assign;
        struct { Operand op; } sinop;
        struct { Operand result, op1, op2; } binop;
        struct { Operand x, y, z; char relop[5]; } ifgoto;
        struct { Operand op; int size; } dec;
    } u;
    InterCode prev;
    InterCode next;
};

struct ArgNode_ {
    Operand op;
    ArgNode next;
};

struct ArgAddr_ {
    char name[32];
    ArgAddr next;
};
```

   其中 Operand 表示操作数，有变量（对应程序定义有名字的变量，形如 vi），临时变量（过程中产生的其他变量，形如 ti），常数，地址，指针，标号（形如 labeli）和函数几种类型，“其他”用于处理过程中表示类型尚未确定。其中对常数 u.var 取常数的值，对其他类型的操作数，u.name 为变量/函数或标号的名字。
   InterCode 表示中间代码，含有类型、参数和前后指针，其中赋值语句、单目操作符、双目操作符、ifgoto 语句和 dec 语句各自拥有不同种类的变量。
   ArgNode 用于表示函数的参数列表，ArgAddr 标记函数定义时有参数是以地址形式传入的，在函数参数含有数组时起到区别作用，后面将进一步进行说明。
   b) 全局变量 variableIndex，tempvarIndex，labelIndex 用于确定新生成变量、临时变量或标号的下标，interCodeHead 和 Tail 标记中间代码链表，argAddrList 为全局范围所有被定义成以地址形式传入的函数参数及其名称（形如 vi）。

```
int variableIndex;
int tempvarIndex;
int labelIndex;

InterCode interCodeHead;
InterCode interCodeTail;
ArgAddr argAddrList;

int calcSize(Type type);
Operand newOperand(int kind, ...);
InterCode newInterCode(int kind, ...);
void insertCode(InterCode code);
```

函数 calcSize 可递归计算一维和多维数组的大小，从而生成 dec 语句。
newOperand，newInterCode 根据传入的类型读取参数，生成对应的操作数或中间代码，insertCode 将生成的中间代码插入到中间代码的链表中。

c) 新建 intermediate.h 和 intermediate.c 文件实现进行中间代码的生成。
对应 semantic 模块中的函数，对每个语法单元 X 写对应的函数 translateX，注意这里也有一些不需要写的，如 Specifiers 部分在 semantic 中已经完成了，与中间代码生成无关。另外此处 FunDec -> VarList -> ParamDec -> VarDec 所对应的 VarDec 有 translateParamVarDec，具体功能是根据函数参数的 id 查找符号表，生成对应的变量 vi 和 param 语句（这里在符号表中加入 int index 并初始化为-1，结合上面的全局变量 variableIndex 可保证对符号表中每一个 id 生成唯一对应的变量 vi）。另有函数 translateVarDec 在其他所有产生语法单元 VarDec 处调用，只在查表得变量类型为数组时生成对应的 dec 语句，其他情况已经在 semantic 模块处理完毕。
对 translateExp 增加辅助函数 translateExpBool（处理关系运算符及整数 01 上的与或非操作），translateExpNum（处理整数的加减乘除运算），translateExpArray（递归处理一维和多维数组的取值）和 translateExpFunc（处理函数调用返回值），从而避免直接在 translateExp 一个函数中进行所有的操作，代码更清晰易读。

```
void translateCond(Node* cur, Operand trueLabel, Operand falseLabel);
void printOprand(FILE* p, Operand op);
void printInterCode(FILE* p, InterCode code);
void writeCode2File(char* filename);
```

另有函数 translateCond（在处理 if、while 语句时和 translateExpBool 中调用），根据传入节点的布尔值和 true false 标号生成代码，printOperand、printInterCode 和 writeCode2File 函数用于向指定的文件输出操作数和中间代码。

c) 反思与总结

a) 各种跳转（if，if else 和 while）代码的生成：
手册中的表写得比较令人费解，自己画了一下，逻辑就很清楚了：

if b then c1 else c2	while b do c
if b goto truelabel (else) goto falselabel truelabel c1 goto endlabel falselabel c2 endlabel	backlabel if b goto truelabel (else) goto endlabel truelabel c goto backlabel endlabel

可见两者唯一的区别是 if 执行完 c1 后，需跳过 c2 直接转到 endlabel，而 while 执行完 c 后，则是跳转回到 backlabel，从而再次进行对 b 的判断。
我在“程序设计语言的形式语义”课学到 operational semantics 的 induction rule：

$$\frac{[\![b]\!]_{boolexp}\,\sigma = true}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = false}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = true}{(\textbf{while } b \textbf{ do } c, \sigma) \longrightarrow (c\,;\textbf{while } b \textbf{ do } c, \sigma)}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = false}{(\textbf{while } b \textbf{ do } c, \sigma) \longrightarrow (\textbf{skip}, \sigma)}$$

这些规则和 label、goto 代码的生成形成巧妙对应，让人觉得很有意思。实际上这里可以立即想出一个优化方案，即对 b 取反，if !b goto falselabel，这样就可以省去一个 goto 和一个 label 语句，然而写起来发现“对 b 取反”实际上并不容易，假如碰巧 b 是 x relop y 也就罢了，可以直接对 relop 进行取反，否则还要处理各种与或非的运算操作问题，递归地去取反，很是麻烦，因此尝试了一下就放弃了。

b) 高维数组和数组传参的实现：

假设高维数组定义的时候只分配空间，不能整体赋值，那么此后无论根据下标对其中的某一位取值或者赋值，都是很容易的，只要递归地调用 translateExp，间接调用 translateExpArray，对返回的地址加上数据宽度乘以偏移量就可以了。唯一需要注意的是，translateExpArray 最外层返回的操作数类型应该是一个 pointer，这样它用得到的地址去进行取值和赋值的时候，会自动加上*，但递归过程中内层调用的返回值还不能取内存单元，仍然作为变量运算，因此需要特别判断。

另外就是数组作为参数传递。作为 arg 传入的地址已经是&vi 或&ti 格式了，所以在函数体内，所有以地址格式定义的参数 vi 都不能再当作地址进行二次取&，而是必须直接作为变量运算，因此在函数的 param 声明中，对每个类型为数组的参数都加入 argAddrList。在函数体内碰到每一个变量首先查符号表，如果是数组类型则继续查 argAddrList，如果它对应的 vi 在这个 list 中，说明这个变量是作为地址类型的参数传入的，则将 vi 当作变量而非地址去进行后续的运算。

c) 有关中间代码优化的思考：

由于时间（和能力）有限，没有写中间代码的优化模块。对于跳转代码的生成上面有一些初步思考，但没有具体实现。最后只在中间代码生成过程中加入一些尽量减少不必要的赋值语句和临时变量的代码，即每次递归调用 translateExp 时都先生成类型为 OTHER 的操作数，如果翻译表达式得到的不是常数、变量或临时变量中的任何一种，才生成新的临时变量并插入赋值语句，否则直接返回该常数或变量。

```
Operand ot = newOperand(OTHER);
translateExp(p, ot);
InterCode code;
Operand op;
if (ot->kind != CONSTANT && ot->kind != TEMPVAR && ot->kind != VARIABLE) {
    op = newOperand(TEMPVAR);
    code = newInterCode(_ASSIGN, op, ot);
    insertCode(code);
}
else
    op = ot;
```

d) 编译运行

使用 makefile 进行编译。直接 cd 到 Code 目录下 make 即可。

编译完成后，输入./parser test output 对 test 进行分析，中间代码输出到 output 中。