编译原理实验——中间代码生成
实验报告

171860667 袁想　　171860664 谢鹏飞　　yxnanda@sina.com

一、实现功能

输入 C--源码文件，输出对应的三地址代码文件；C--源代码中可以出现结构体变量，并且结构体变量可以作为函数参数(选做 3.1)；实现一维数组赋值以及一些中间代码优化。

二、主要数据结构

(1) 操作数结构 Operand,定义在 intercode.h 中

```
typedef enum {TEMP_VARIABLE, VARIABLE, CONSTANT, TEMP_ADDRESS, ADDRESS, STAR,
              LABEL_OP, FUNCTION_OP, VARIABLE_CONSTANT, DEFAULT} OperandKind;


typedef struct Operand
{
    OperandKind kind;
    struct {
        int constant_value;
        char contents[33];
    } opinfo;
} Operand;
```

用 Operand 类型定义操作数，操作数有不同类型 kind，例如变量 VARIABLE，常量 CONSTANT；常变量 VARIABLE_CONSTANT（用于常量折叠，它既有变量名称 contents 也有常量值 constant_value，因为常变量可能会变回变量，所以它需要有上述两方面信息，因此将 opinfo 设为 struct 类型，方便访问两个信息）；地址 ADDRESS，其内容 contents 存放的是这个变量的名字，需要完成取地址的操作，即在输出中间代码的时候加上取地址符&；指针 STAR，其内容 contents 是一个地址，需要完成的操作是取该地址指向的值，即在输出中间代码的时候加上取指针值*。等等。根据不同类型选择调用 opinfo 中的信息。

(2) 中间代码结构 InterCode,定义在 intercode.h 中

```
typedef enum {LABEL_IC, FUNCTION_IC, ASSIGN, PLUS, SUB, MUL, DIV, GETADDR, RIGHTSTAR, LEFTSTAR,
              GOTO, RELOPGOTO, RETURN, DEC, ARG, CALL, PARAM, READ, WRITE} InterCodeKind;


typedef struct InterCode
{
    InterCodeKind kind;
    union {
        struct {
            struct Operand *op;
        } singleop;

        struct {
            struct Operand *op;
            struct Operand *op1;
            struct Operand *op2;
        } tripleop; // op = op1 + op2
        struct {
            struct Operand *x;
            struct Operand *y;
            struct Operand *z;
            char relop[33];
        } relopgoto; // if x relop y goto z
        struct {
            struct Operand *op;
            int size;
        } dec;
    } codeinfo;

    struct InterCode *previous;
    struct InterCode *next;

} InterCode;
```

每一条中间代码对应一个 InterCode，中间代码有不同类型 kind，根据每条代码的操作数个数，调用内部联合体 codeinfo 中的不同结构。中间代码使用双向链表存储，提供函数 void insertintercode(InterCode *intercode) 插入中间代码，以及函数 void deleteintercode(InterCode *intercode)用于优化时删除一条中间代码。

三、实验设计思路与实现

（1）基本翻译模式：与实验二类似，为每一个产生式定义一个处理函数，在需要产生中间代码的产生式函数中通过调用进一步处理函数或者生成中间代码结构体变量，填写信息将其插入双向链表实现中间代码生成，对于基本表达式和条件表达式均采用实验指导的方法。

对于访问数组的翻译，即在处理 Exp 产生式时规约到 Exp LB Exp RB。由于我们选做 3.1 翻译的是一维数组，因此处理流程如下：首先创建两个临时变量，将其作为参数分别处理第一个 Exp 和第二个 Exp。如果处理完后 Exp1 处理的 Op1 类型为 STAR，说明 Op1 代表一个地址，这种情况代表着这个数组是结构体中的一个域，那么我们将该 Op1 类型改为 VARIABLE，因为后面是要进行地址偏移计算。否则 Op1 就是一个变量名，因此需要将 Op1 类型改为 ADDRESS，将其取地址获得地址的值才能进行后面的偏移计算。之后我们将第一个 Exp 作为参数调用实验二的 Exp 将返回 TypeInfo，通过实现的计算类型的大小的函数 calculatesize 计算出数组中每个元素的大小，之后判断如果放入第二个 Exp 处理的 Op2 类型为 CONSTANT，直接可以计算出偏移量；否则生成一条 MUL 中间代码，新建一个临时变量 Op3，形式为 Op3 = Op2*数组中元素大小计算偏移量。最后统一生成 PLUS 指令，结果存放在传入数组处理函数的 Op 中，Op 类型为临时变量，加法的两个数为 Op1 和偏移量。最后将传入数组处理函数的 Op 的类型改为 STAR，表明处理完是一个地址，加*才能取指向的值。

对于访问结构体中域的翻译，即在处理 Exp 产生式时规约到 Exp DOT ID。首先创建临时变量，将其作为参数处理 Exp1。之后我们将 Exp1 作为参数调用实验二的 Exp 将返回 TypeInfo，即返回结构体的组成，将 ID 与其与结构体中域逐个比对，直到找到，同时前面遍历到的域将计算出所占大小，累加形成偏移量。之后如果这个域是一个数组，并且其元素还是数组，那么就是高维数组，报错。否则判断如果 Op1 类型为 STAR，说明 Op1 代表一个地址，这种情况代表在访问一个结构体数组的一个元素中的域，那么我们将该 Op1 类型改为 VARIABLE，因为后面是要进行地址偏移计算。如果不是 STAR，那么就是一个变量名，此时我们进一步判断这个变量是不是作为了函数的参数（对结构体 a，如果它是函数参数，采用地址传递，a 代表一个地址；如果不是函数参数，那么 a 就是变量名，&a 是取结构体首地址。在处理每一个函数时，在前面函数参数声明 VarDec 中，搜索符号表判断当前参数是不是结构体，是则将其插入一个参数结构体列表），如果是，那么 Op1 类型改为 VARIABLE，因为后面是要进行地址偏移计算。其他情况，将 Op1 类型改为 ADDRESS。之后生成中间代码，类型为 PLUS，结果放在函数传入的 Op 中，Op 类型设为临时变量，加法两个操作数为 Op1 和偏移量。最后将传入结构体处理函数的 Op 类型改为 STAR，表明处理完是一个地址。

（2）数组赋值：对于选做 3.1，考虑一维数组赋值，不考虑结构体数组赋值。两个数组的赋值出现在表达式 Exp->Exp ASSIGNOP Exp 和函数体中一开始变量定义中的 Dec->VarDec ASSIGNOP Exp 中。对于 Exp->Exp ASSIGNOP Exp 考虑如下 4 种情况：1.两数组赋值，如 a=b；2.左为结构体中数组，右为单纯数组，如 aaa.a=b；3.左为单纯数组，右为结构体中数组，如 a=bbb.b；4.左右均为结构体中数组，如 aaa.a=bbb.b。以 a=b 为例，首先调用实验二中的函数 Exp，返回类型结构 TypeInfo 变量，当两个操作数的类型 type 中的 kind 均为 array 时，执行数组赋值操作，如果两个操作数类型 Operand 中的 kind 均为 VARIABLE 时，意味着这是两个单纯数组赋值，将两个操作数的 kind 改为 ADDRESS，即取数组地址，计算出每个元素的大小 elementsize，取两数组大小的最小值进行 for 循环，每次循环实现三条赋值中间代码，第一条中间代码为计算左数组的对应元素的地址，第二条同，第三条为对前两条中间代码计算出的地址取指针，将右数组的对应元素值赋给左数组。赋值完后，将两个

操作数的 kind 改回 VARIABLE 即可。如果不是单纯数组，则说明操作数的 kind 是 STAR，值为地址，则计算元素地址时将 kind 改为 VARIABLE 计算，赋值完后改回 STAR 即可。在数组定义的时候也有可能进行赋值初始化，即对于函数中 Dec->VarDec ASSIGNOP Exp 的情况，但只有一种可能，即函数参数为结构体，利用结构体中的数组赋值，其处理类似于 a=bbb.b，即右数组类型为 STAR，参见上述操作。我们对实现进行了测试，结果均是正确的。

（3）中间代码优化：我们实现了以下五种优化：

```
LABEL LABEL1 :
LABEL LABEL2 :
LABEL LABEL3 :
GOTO LABEL4
```

1、如图：…

删除冗余 LABEL。即对于上图情况，我们只需要保留第一个 LABEL，LABEL2 和 LABEL3 都可以用 LABEL1 替代。实现方法为循环，每次只处理一个 LABEL 块，记录第一个 LABEL，将后面紧接着的 LABEL 放入冗余 LABEL 链表（里面存放 LABEL 名字），之后遍历中间代码，如果是 IF-GOTO 语句或者是 GOTO 语句，且其 LABEL 是冗余的，那么替换为第一个 LABEL，如果是 LABEL 声明，并且这个 LABEL 是冗余的，那么直接把这个 LABEL 中间代码删除。

2、优化条件跳转的中间代码：正常的条件跳转中间代码由三条组成，即 IF x relop y GOTO label_true，GOTO label_false，LABEL label_true，这三条可以优化成两条，对条件 relop 取反，改为 IF x !relop y GOTO label_false，LABEL label_true。具体实现：遍历中间代码链表，判断当前节点 kind 是否为 RELOPGOTO，如果是接着判断之后两个节点的 kind 是否分别为 GOTO 和 LABEL_IC，且第三个节点内容是否与当前节点 GOTO 之后的内容一致，如果均满足，则将第二节点 GOTO 的内容赋给当前节点的跳转内容，删除 GOTO 节点。

3、精简 IF-GOTO 逻辑：如果中间代码中一个 IF-GOTO 语句中 RELOP 两边的操作数是常数或常变量，那么可以直接获取其常数值，那么直接判断是否满足 RELOP 条件，如果满足，直接将这个语句改为 GOTO，如果不满足，那么这条 IF-GOTO 语句没有意义，直接删除这条指令。例如 IF #2 == #3 GOTO LABEL1，显然永远不可能满足，直接删除这条指令。

4、去除不是目标 LABEL 的 LABEL：遍历中间代码，存放所有 IF-GOTO 语句和 GOTO 语句的 LABEL，可以模仿实验二中符号表的哈希表来存储这些 LABEL（因为 LABEL 可能很多）。之后再次扫描所有中间代码，对于所有 LABEL 定义，如果这个 LABEL 不在前面构造的 LABEL 表中，即这个 LABEL 不是任何一个跳转语句的跳转目标，那么将这个 LABEL 指令删除。

5、常量折叠：设定 Operand 一种类型为常变量，当表达式可以直接计算出来时，可以直接计算出其值，然后在用到这个表达式的时候直接用这个常数来替换。在函数体中变量定义且赋初值处理中，如果右边 Exp 返回一个常数或常变量，将定义的变量插入常量表，常量表记录变量名，常量值，是否有效（在 while 或者 if-else 语句中，没有跳出这些语句前，一些常变量要变成变量，否则会出现逻辑错误）。处理 Exp→ID 时，如果该 ID 在常量表中并且此时不在 while 或者 if-else 语句中，那么将传入的 operand 类型设为常变量，设定其常数值。如果在 while 语句中或者在 if-else 的 stmt 逻辑中，需要将其在常量表中的有效位设为 0 表示失效，operand 设为变量类型。在处理取负以及加减乘除四种运算时，如果操作数是常数或者常变量，那么直接计算出结果放入传入的 operand。如果是赋值语句，如果左边 Exp 返回类型为常变量，先改为变量，ASSIGNOP 中间代码生成后如果不在 while 或者 if-else 逻辑中，且右边是常数或者常变量，那么将左边插入常数表，否则如果左边在常数表中，将其有效位设为 0。在最后打印中间代码的时候，如果是常变量，直接作为常数输出数值。

四、编译说明：利用给定的 Makefile 文件，将其置于源码的相同路径下，在终端执行 make。