编译原理实验三：中间代码生成

组长 161130118 尹浚宇
组员 171850524 刘扬

一、实现功能

完成必做内容与选做 3.2，实现如下功能:

1. 在实验二的基础上，将 C--源代码翻译为中间代码.
2. 一维数组类型的变量可以作为函数参数(但是函数不会返回一维数组类型的值)，可以出现高维数组类型的变量，即选做 3.2.
3. 对生成的中间代码进行了一定程度的优化.

二、实现方法

1. 相关数据结构

我们生成中间代码所用的数据结构包括操作数，中间代码以及储存中间代码的双向链表.

操作数的结构体定义如下:

```
struct Operand_ {
    enum { VARIABLE, CONSTANT, ADDRESS, TMPVAR, LABEL, FUNC, ARRAY_OP, ARRAY_PARAM } kind;
    union {
        int var_no; // tmpvar label
        int val; // constant address
        char *name;//func variable array_op arrar_param
    } u;
};
```

其中，VARIABLE 代表变量，CONSTANT 代表常数，ADDRESS 代表地址，TMPVAR 代表临时变量，LABLE 代表标签，FUNC 代表函数名.

特别地，ARRAY_OP 和 ARRAY_PARAM 都代表数组，区别在于前者是局部变量，后者是函数参数. 这里对这两者进行区分的原因是: 数组作为参数出现，应当为指针类型，不应该对其进行取地址的操作，而作为局部变量的数组需要取地址操作.

中间代码的结构体定义如下:

```
struct interCode_ {
    enum {
        LABEL_IC, FUNCTION_IC, RETURN_IC, GOTO, PARAM, ARG, READ, WRITE,
        ASSIGN,
        ADD_IC, SUB_IC, MUL_IC, DIV_IC,
        IFGOTO,
        DEC,
        CALL
    } kind;
    union
    {
        struct { Operand op; } single_op;//LABEL FUNCTION GOTO RETURN ARG PARAM READ WRITE
        struct { Operand left, right; } assign;//ASSIGN
        struct { Operand res, op1, op2; } double_op;//ADD SUB MUL DIV
        struct { Operand label, op1, op2; char* relop; } triple_op;//IFGOTO
        struct { Operand op; int size; } dec;//DEC
        struct { Operand op, fun; } call;//CALL
    } u;
    interCode pre;
    interCode next;
};
```

其中 kind 指明中间代码类型，用于产生正确的中间代码. 下方联合类型 u 则将中间代码按操作数格式类型分为六类. 最下方的 pre 和 next 两个指针将存储中间代码的节点串连成一个双向链表.

2. 中间代码的翻译

中间代码的翻译主要借鉴教程上给出的翻译模式. 对于文法中的每一个产生式，我们都实现了相应的 translate 函数. 特别地，有些产生式并不会产生中间代码，我们在实现过程中将其对应的 translate 函数删除.

注意到有一些产生式只需要根据不同情况调用其他翻译函数，其本身不会生成中间代码. 这部分函数实现较为简单，只需根据相应产生式进行判断即可.

剩下的函数都是本身会产生中间代码的函数，需要小心实现. 其中:
translate_VarDec 函数生成了为数组类型申请空间的中间代码: Dec x [size];
translate_FunDec 函数产生了和函数有关的代码: FUNCTION f 和 PARAM x;
translate_Dec 函数除了调用 translate_VarDec 外，还需要为可能存在的初始化语句生成代码.

translate_cond, translate_Stmt 和 translate_Exp 函数参考教程的翻译模式实现. 前两个函数的实现较为简单，只需要正确理解教程的翻译模式即可正确实现. translate_Exp 的实现最为复杂，除去大部分可照搬翻译模式的情况，由于要区分数组类型作为左值和右值的情况，我们为此函数多添加一个参数 isLeft. 对于左值，最后一条中间代码为 place := &array + offset; 对于右值，继续生成一条中间代码对 place 进行解引用.

对于数组寻址的计算，我们都采用基址加偏移量的方式，故数组维度没有太大的影响. 只需查找符号表获得数组类型每一维度的大小，进而就可以递归计算出数组变量的地址.

对于函数调用中参数列表的处理，我们直接在 translate_Exp 中通过迭代的方式处理每个参数，并在最后生成形如 ARG x 的中间代码.

3. 中间代码的优化

由于在第一次生成中间代码时，我们只考虑了正确性，所以生成出的代码存在大量的优化空间. 考虑到基于 DAG 的代码优化在实现上的复杂性，我们这里实现的优化均不涉及 DAG 的实现，所以其优化目标局限于连续的两到三条语句和一些冗余逻辑的消减.

具体而言，我们实现的优化大致分为以下几种:

(1) 优化 IFGOTO 的逻辑: 将形如

    if op1 [relop] op2 goto [labeln]
    goto [labelm]
    [labeln]

的代码片段变为:

    if op1 ![relop] op2 goto [labelm]
    [labeln]

(2) 消除常数之间的运算: 若算术运算的两个分量均为常数，则提前计算出结果赋给结果分量，并将结果分量的类型改为常数.

(3) 赋值表达式的合并: 将形如 t1 = x op y; v = t1 的代码合并为 v = x op y.

(4) 去除不必要的临时变量: 将形如

t1 = a

t2 = b

t3 = t1 op t2 或 if t1 relop t2 goto labeln

的代码片段变为:

t3 = a op b 或 if a relop b goto labeln

三、编译方式

使用自带的 makefile 进行编译