编译原理第三次实验报告

小组信息	181860087	唐业
	181860102	王印可
任务信息	编号3	必做内容 + 选做3.2（数组处理）

1.实现的功能

1.1.实现功能

- 必做内容——在前两次实验的基础上，将C--源代码翻译为中间代码
- 选做内容——数组处理
  - 源代码中一维数组类型的变量可以作为函数参数
  - 可以处理高维数组类型的变量（高维数组类型的变量不会作为函数的参数或返回值）

1.2.数据结构

在实现中，我们采用的是线性IR的双向Linked list结构，结构定义如下

`struct InterCode_` (表示一条中间代码的结构)

```
struct InterCode_
{
    enum {
MYFUNCTION,MYPARAM,MYRETURN,MYLABEL,MYGOTO,MYREAD,MYWRITE,MYARG,MYASSIGN, MYDEC,
MYCALL,MYADD,MYSUB,MYMUL,MYDIV,MYIFGOTO } kind;
    union {
        struct{ Operand op; } op_single;
        struct { Operand left, right; } op_assign;
        struct { Operand result, op1, op2; char _operator; } op_binary;
        struct { Operand x, y, label; char relop[CHARMAXSIZE]; } op_triple;
        struct { Operand op; int size; } op_dec;
    } u;
    InterCode prev;
    InterCode next;
};
```

其中kind代表当前code的类型，针对不同类型我们在联合体u中使用了不同种的结构来表示它。

`struct Operand_` (表示一个操作数的结构)

```
struct Operand_ {
    enum {
        VARIABLE,CONSTANT,ADDRESS,STAR__,COSNTVAR,TEMPVAR,NOTHING,LABEL,
        FUNCTION__
    } kind;
    union {
        int var_no; char value[CHARMAXSIZE];
    } u;
};
```

同样，我们根据枚举类型变量kind来表示不同种的操作数，同时在联合体u中定义了 var_no 和 value（例如当操作数类型为常量时 var_no 代表其整形值，当操作数类型为变量等类型时 value 代表其变量名）

1.3.实现方法

- 和前一次的语义分析相同，我们为每个语法单元定义了一个翻译函数。例如，对于语法单元Exp，我们定义了如下函数

  ```
  void translateExp (node root, Operand op)
  ```

  其中，root代表指向当前语法单元的指针，op为调用者传入的用于生成中间代码的一个操作数指针。

  我们根据不同语法单元来定义不同的翻译函数，这样，通过遍历整棵语法树便可生成最终的中间代码。

- 在翻译时，我们并不会一边翻译一边输出代码内容，而是将解析好的中间代码插入到我们先前定义好的 InterCode_ 数据结构中。等完全解析完毕后再一并输出，从而增加了代码的可优化性。

  ```
  void insertCode (InterCode code)
  //我们封装了插入函数，将当前code插入到解析好的codelist尾部
  ```

- 在翻译时需要注意数组的赋值处理

  1. 对于数组赋值 a = b ，我们首先得到数组a和数组b的首元素地址 i，j
  2. 在计算出 b 数组的大小 size_of_b 后，令 b0 加上 size_of_b 从而得到需要赋值的最后一个元素地址的后一个地址 k
  3. 这样，我们创造如下中间代码

     ```
     LABEL label1:
     IF j >= k GOTO label2
     *i := *j
     i := i + 4
     j := j + 4
     GOTO label1
     LABEL label2:
     ```

     即可通过循环来进行数组元素的赋值（当 j >= k 时，说明数组b最后一个需要赋值的元素已经完成了赋值，用 GOTO label2 结束循环）

     - *完整代码可见 `translateExp` 函数中的 "ASSIGNOP" 分支处理*

- 程序优化

  1. **常量折叠**

     常量折叠即把 `int a = 3 + 1 - 1 * 5;` 优化为 `int a = -1;`，实现方法是在 `Operand_` 结构体中加入 `COSNTVAR` 这一类型，如果在当处理 `Exp -> Exp1 +-*/ Exp2` 的时候，如果Exp1和Exp2的类型均为 `CONSTANT` 或 `CONSNTVAR`，那么将Exp的类型设置为 `CONSNTVAR`，并计算Exp的数值 `var_no`。需要注意的是除法的情况，遇到除数为0的情况，直接翻译成 `Exp := #1 / #0`，因为此时无法进行常量折叠。

  2. **删除多余的GOTO**

     删除多余的GOTO，即对RELOP取反，将

```
IF op1 RELOP op2 GOTO label_true
GOTO label_false
LABEL label_true:
xxx...
LABEL label_false:
yyy...
```

优化为：

```
IF op1 !RELOP op2 GOTO label_false
xxx...(原label_true的处理部分)
LABEL label_false:
yyy...(原label_false的处理部分)
```

这样可以删掉原翻译模式中冗余的GOTO 和部分LABEL，测试发现此方法的效果显著。

3. **删除多余的表达式**

   删除多余的无效表达式，比如`i + i;`、`1 + 1 + 1`，实现方法是在在`Operand_`结构体中加入`NOTHING`这一类型，此类无效表达式会翻译成`nothing := i + i`，最后遍历删除所有的NOTHING中间代码。

4. **合并LABEL**

   合并LABEL即当遇到多个LABEL重合的情况，`LABEL label1`

```
LABEL label1
LABEL label3
LABEL label5
...
```

此时只需要保留第一个label即可，并把需要删除的label全替换为保留的label即可。

2. 编译运行

- 如何编译?
  1. 进入目录`Compiler/Lab/Code`
  2. 在控制台运行指令`make parser`
- 如何运行?
  - 命令行输入`make test`或`./parser  xxx.cmm  yy.ir`(`xxx.cmm`表示自己编写的测试输入文件，`yy.ir`表示生成的中间代码文件，如果没有第三个参数`yy.ir`，默认输出文件为`out.ir`)
  - 对.ir文件使用irsim小程序即可

以上便为实验三报告的全部内容