编译原理第三次实验 实验报告

151220152 殷乔逸

程序功能

- [x] 代码中**已实现**了实验一和实验二的所有需求。(**包括所有必做和选做要求**)
- [x] 代码中**已实现**对正确通过词法、语法及语义分析的 C 语言代码进行中间代码的生成。(**完成所有必做和选做要求**)
- [x] 代码中改进或实现了实验三的以下假设或者功能：**(亮点)**
    1. 改进假设 1，支持八进制和十六进制数(转为十进制数进行计算)。
    2. 改进假设 4，支持变量作用域。
    3. 改进假设 6，支持函数声明。
    4. 改进要求 3.2，高维数组也允许作为参数。此外，结构体和数组可以互相嵌套。
    5. 优化控制流语句，尽可能少生成 GOTO 和 LABEL 而使用 fall 特性。
    6. 优化常数表达式，可计算的常数表达式会被直接计算好。
    7. 优化临时变量，避免生成不必要的中间变量，减少不必要的赋值表达式。
    8. 语句块优化暂时未实现。

数据结构

1. 操作数的数据结构如下，`OP_TYPE` 表示操作数类型，是一个枚举类型，`MODIFIER_TYPE` 表示操作数前的修饰符，同样也是一个枚举类型。最后的匿名联合体用于存储变量或临时变量的标号或者立即数的值。

```
typedef struct IROperand_ {
    /* The type of operands */
    OP_TYPE kind;
    /* Whether the operand is only a varibale or an
     * address or the dereferenced
     */
    MODIFIER_TYPE modifier;
    /* Identify different temporary variables */
    union {
        int var_label_num;
        int value_int;
        float value_float;
        char* var_func_name;
    };
} IROperand;
```

2. 中间代码的数据结构如下，采用了线性 IR 的结构，并且使用了双向链表。其中 IR_TYPE 用于表示中间代码的类型，是一个枚举类型。紧接着的一个大的匿名联合体用于存储不同类型中间代码所需要的操作数。最后的匿名联合体专门用于 IR_RELOP 和 IR_DEC 两种类型的中间代码表示。

```
typedef struct IRCode_ {
    /* The type of a piece of intermediate
     * representation
     */
    IR_TYPE kind;
    /* For result or temporary variables */
    union {
        struct {
            IROperand* dst;
            union {
                struct {
                    IROperand* src1;
                    IROperand* src2;
                };
            IROperand* merged_src;
            };
        };
        struct {
            // 1-operand
            IROperand* src;
        };
        struct {
            // for RETURN only
            IROperand* rtn;
            IROperand* func;
        };
    };
    /* The type of relop operation */
    union {
        RELOP_TYPE relop;
        int declared_size;
        bool none_flag;
    };
    /* A bi-direction list */
    struct IRCode_* prev;
    struct IRCode_* next;
} IRCode;
```

实现思路

1. 首先是将 Semantic 与 Translate 分开来处理，重新构造了一个专门用于生成中间代码的模块，其中两者都调用 SymbolTable 模块，在两者调用中间加入了重置 SymbolTable 模块的方法，以避免两者相互影响。

2. 主要实现思路是完成对不同表达式和语句的翻译，尤其是对于表达式中出现的函数、结构体和数组的翻译，还有就是对条件表达式的翻译。采用的方法是实现了下述的两个方法。

   将所有的数组、函数和结构体单独放入 `Translate_DFS_Expression_Address()` 的方法中进行处理，在这个函数中首先查询符号表得到相应的符号数据记录，然后根据符号类型的不同采用不同方法处理。

   譬如针对数组则递归调用该函数，将 `a[i][j][k]`这样的高维数组拆开，第一次在函数中将 `a[i][j]`作为实参进入下层递归，第二次将 `a[i]`作为实参再进入下层递归，以此类推，最终得到 `a` 可以查表知道 `a` 是一个数组类型，然后返回给上层该条数据记录，然后在上层针对 `a[i]`则获取第 2、3 维度的声明大小再乘以 `i` 可以得到一次寻址的结果，然后将结果继续返回，以此类推最终得到数组 `a[i][j][k]`的位置，最后在加上修饰符*解引用得到数组在该位置上的值。

   除此之外，所有的赋值语句（左值）都以将值写入左值语句的地址中的方法来实现，这样统一了平凡变量与函数、数组和结构体的赋值方法。对于条件表达式则交给 `Translate_DFS_Expression_Condition()`的方法来处理，传入 `label_false` 和 `label_true` 用于标记语句的转到位置。

实验难点

1. 难点 1 是针对条件表达式和含结构体、数组或函数的表达式的翻译，在上面已经大致描述了，具体参考代码，基本就是按照 `Project_3.pdf` 上的实验指导和翻译模式来做。

2. 难点 2 是针对生成的中间代码进行优化，基本的优化思路包括**控制流语句的优化**，**常数表达式的优化**，**临时变量的优化**和**语句块的优化**，优化方法在解决方案中大致介绍。

解决方案 (亮点)

1. **控制流语句的优化：** 利用中间代码执行完本条自动就是下一条的特性(fall 特性)，尽可能减少 GOTO、LABEL 语句。这一部分的代码在函数 `Translate_DFS_Expression_Condition()`中。函数中如果 `label` 是 `NULL` 则认为是 fall，可以直接省略该标号。

2. **常数表达式的优化：** 增加新的 IR 语句时，如果该 IR 是加减乘除，检查 `src1` 和 `src2` 是否是立即数，如果某一个是立即数则直接可以计算出 `dst`，并删掉这条语句，改为直接赋值。 譬如：

   ```
   t1 = #7 + #10
   t2 = t1 + #0
   t3 = t1 * #1
   ```

则会被优化成如下形式：

```
t1 = #17
t2 = t1
t3 = t1
```

3. **临时变量的优化：** 在每次产生临时变量后，调用函数 `temp_var = Clean_Temp_Var(temp_var)`，如果刚生成的语句是一个赋值语句，那么临时变量即可以去掉。所以在函数中即判断该操作数是否是临时变量，如果是临时变量且中间代码的类型是赋值，那么就记录赋值右边的量，即 `merged_src`，释放临时变量返回被记录的量；如果非临时变量则不做处理。 譬如：

```
...
t1 = v1
t2 = t1
t3 = t2 + #7
WRITE t3
```

则会被优化成如下形式：

```
t3 = v1 + #7
WRITE t3
```

4. **语句块 DAG 的优化：** 未实现

编译及运行方法

首先压缩包解压后得到的文件夹 `Lab3`,`Test` 文件夹中存储了 5 个测试用例，`Code` 文件夹存储了所有代码文件。

- 如果想要直接编译运行，在 shell 命令行执行`./run.sh`，则会按默认模式(仅生成中间代码)直接编译并且对 `Test` 文件夹下的用例进行测试，输出的结果以`.ir` 的文件格式按序存储在 `Test_IR` 文件夹中。
- 如果想要删除中间文件、可执行文件和默认的结果存储文件夹，在 shell 命令行执行`./run.sh --clean` 即可。
- 如果想要采用不同的模式或者不同结果路径进行测试，在 shell 命令行执行`./run.sh [-p] [-s] [-dag] [-ir] -[a] [result path]`即可，默认的资源文件夹是 `Test`，并且脚本默认批量处理。
- 如果仅仅想要编译，则进入 `Code` 文件夹执行 `make parser`，则会重新编译并在上层目录 `Lab3` 得到可执行文件 `parser`。可以执行`./parser` 来查看可用的参数选项。
- 如果想要**对某一个测试用例生成中间代码**，则在 `Lab3` 目录下执行`./parse -ir source [result]`，其中 `source` 代表测试用例的路径，`result` 代表结果文件路径，如果该参数为空则打印至屏幕。