**编译和测试方法**

1. cd Code/：进入源代码和Makefile文件所在的路径下
2. make：使用make命令对程序代码进行**编译构建**
3. ./parser ../Test/xxx.cmm ../result/temple.ir：对指定的 C--代码xxx.cmm进行中间代码的生成，将输出结果写入到result文件夹下名为temple.ir的文件中。 也可以修改Makefile文件中的test部分的内容，然后使用make test命令进行测试和中间代码的结果输出。本人的Makefile文件中已经将Test文件夹中需要测试的代码的路径均写入，可以同时输出Test文件夹中多个测试文件的测试结果。

本人已经使用虚拟机小程序测试过中间代码的运行结果，**均顺利执行**，保证了正确性。 为避免冗余，此处没有粘贴结果。由于生成中间代码的过程中做了较多优化，我认为生成的中间代码的效率也是很高的。**生成的中间代码已经输出到result文件夹中。**

以下**todo**中主要集中在**语句表达式的翻译和条件表达式的翻译**。主要的优化思路：计算的操作可以在编译时计算结果，从而避免运行时的计算，提高程序的执行效率。对于跳转指令，利用短路的思想，减少不必要的跳转过程。在翻译过程中将新生成的中间代码插入到已有的中间代码链中，确保生成的代码按正确的顺序执行。

**\_Todo 1 和 \_Todo 2**

加法优化（optimizePLUSIR） 参考减法优化的代码（optimizeSUBIR）进行实现， 乘法优化（optimizeMULIR） 参考除法优化的代码（optimizeDIVIR）进行实现，其思想是避免运行时的计算，实现较为简单。

对于加法优化代码 与减法优化代码非常相似，分为常量操作数的加法、含有常量0的加法和普通加法，本人不再赘述。下面简单叙述乘法优化。

* 常量操作数的乘法：两个源操作数都是常量（CONSTANT\_OP），创建常量并赋值。
* 含有常量0的乘法：只要两个操作数中存在一个为常量0，乘法的结果将是0。因此，可以直接给目标操作数赋值为0（operandCpy(dest, getValue(0))），并返回一个空中间代码（getNullInterCode()）。
* 含有常量1的乘法：其中一个操作数是常量1且另一个操作数不是取地址或解引用的操作数，乘法操作可以简化为赋值操作，**处理做法相当于加法优化情况下有一个操作数为常量0**，直接将非1操作数src复制到目标操作数dest（operandCpy(dest, src)）。
* 普通乘法：如果以上条件都不满足，则生成并返回一条正常的乘法指令MUL\_IR。

**\_Todo 3**

基本表达式的翻译——单变量作为左值 主要是**三地址码赋值如何实现**的问题，根据符号表条目创建新的临时变量之后，将右侧表达式的翻译结果进行赋值即可。

* 根据root->children[0]的内容判断出当前是单个变量作为左值的基本表达式的翻译，调用findSymbolAll**查找该变量在符号表中的条目**，获取到变量信息。然后创建两个临时变量 tmp1 和 tmp2，Operand tmp1 = getVar(sym->name)，即根据变量名**创建一个新的临时变量**tmp1。
* 右侧**表达式的翻译**：递归调用函数 translateExp 来翻译右侧表达式root->children[2]并将结果存储在 tmp2 中，同时将translateExp返回的翻译过程作为中间代码code1进行存储。
* 创建一个新的赋值指令code2，将表达式tmp2 的值赋给 临时变量tmp1，将指令进行拼接之后，将运算结果存回函数调用的place，即将临时变量tmp1 operandCpy到不空的place，完成单变量作为左值的基本表达式的翻译。

**\_Todo 4**

基本表达式的翻译——条件表达式 **参照PPT所给表格中的翻译过程进行代码编写**。但是依赖于**todo5**中的条件表达式的翻译函数translate\_Cond进行短路翻译。

* 创建标签： 通过 newLabel函数创建新标签label1 和 label2 ，用于条件跳转。
* 条件表达式初始化，设置**条件表达式的初始值为假**： 创建一个新的中间代码 code0，其类型为 ASSIGN\_IR（赋值操作），用于将 常量0 赋值给 place，对应表格中0place := #0。基于控制流的优化考虑，对条件表达式进行初始化时假设条件为假，设置默认执行路径，避免在条件为假时执行额外的跳转指令。
* 翻译条件表达式：调用 translateCond 函数，其根据条件表达式的真假，生成跳转到 label1 或 label2 的代码，翻译过程作为中间代码code1进行存储。对应表格中code1 = translate\_Cond(Exp, label1, label2, sym\_table)
* 优化标签：由于之前调用 translateCond 函数进行翻译并跳转，此处进行优化。 调用 optimizeLABELBeforeGOTO 函数优化中间代码code1。如果code1的最后一句是label语句，那么将code1中的所有GOTO语句中的该label替换为label1，即下面接着定义的label1。

|  |
| --- |
| InterCode code1 = translateCond(root, label1, label2); // 翻译条件表达式 optimizeLABELBeforeGOTO(code1, label1); // 中间代码优化\_短路思想 |

注：优化后面跟着 GOTO 或 LABEL 的 LABEL 语句 -> optimizeLABELBeforeGOTO函数。这种优化主要是为了**减少中间代码中不必要的跳转**。 通过更新无条件跳转和条件跳转指令，可以直接跳转到最终目标标签，从而减少代码长度和提高运行效率。

* 设置真值跳转标签：创建 code\_label1 作为 LABEL\_IR 类型的中间代码，表示 label1 的位置。 创建 code2 作为 ASSIGN\_IR 类型的中间代码，用于将 常量1 赋值给 place，表示条件为真的情况。将二者进行组合，即得到真值的跳转标签。
* 设置假值标签: 创建 code\_label2 作为 LABEL\_IR 类型的中间代码。

**\_Todo 5**

条件表达式的翻译模式 主要**参照PPT所给表格中的翻译过程进行代码编写**。但是不同于课本上的回填的内容，此处将跳转的两个目标labelTrue和labelFalse作为继承属性（函数参数）进行处理，每当在条件表达式内部需要跳转到外部时，跳转目标都已经从父节点通过参数进行获取，直接填入。

**处理逻辑非 （NOT）**

递归翻译： 递归调用 translateCond，交换 labelTrue 和 labelFalse 。

**处理关系表达式 （RELOP）**

* 翻译表达式： 创建临时变量 t1 和 t2 用于存储左右两个表达式的值。调用函数translateExp分别翻译 root->children[0] 和 root->children[2]，即关系运算符的两侧，将翻译结果存储到临时变量中，并将翻译过程作为中间代码code1 和 code2。
* 生成条件跳转指令： code\_ifgoto 为 IF\_GOTO\_IR 类型的指令，比较 t1 和 t2。 根据printInterCodes中case IF\_GOTO\_IR的实现，将中间代码code\_ifgoto的ops[0]置为左侧表达式的值t1，ops[1]置为右侧表达式的值t2，ops[2]置为GOTO到的位置labelTrue，relop置为当前比较的关系运算符root->children[1]->strVal。如果比较为真，则跳转到 labelTrue。

|  |
| --- |
| InterCode code\_ifgoto = (InterCode)malloc(sizeof(InterCode\_)); code\_ifgoto->kind = IF\_GOTO\_IR; code\_ifgoto->ops[0] = t1; code\_ifgoto->ops[1] = t2; code\_ifgoto->ops[2] = labelTrue; strcpy(code\_ifgoto->relop, root->children[1]->strVal); insertInterCode(code\_ifgoto, code1); |

* 生成无条件跳转指令： 中间代码code\_goto 为 GOTO\_IR 类型的指令，用于在条件为假时跳转到 labelFalse。

**处理逻辑与 （AND） 和逻辑或 （OR）**

* 短路逻辑递归翻译左子表达式root->children[0]：对于逻辑与（AND），如果左子表达式为假，则直接跳转到 labelFalse，否则继续计算右子表达式； 对于逻辑或（OR），如果左子表达式为真，则直接跳转到 labelTrue，否则继续计算右子表达式。这样实现了**短路翻译**，即确定整个表达式结果之前不需要完全计算所有子表达式。对应表格中 AND：code1 = translate\_Cond(Exp1, label1, label\_false, sym\_table) OR： code1 = translate\_Cond(Exp1, label\_true, label1, sym\_table)

|  |
| --- |
| else if (root->childNum >= 2 && strcmp(root->children[1]->name, "AND") == 0) {  code1 = translateCond(root->children[0], label, labelFalse); //...  else if (root->childNum >= 2 && strcmp(root->children[1]->name, "OR") == 0) {  code1 = translateCond(root->children[0], labelTrue, label); //... |

* 右子表达式root->children[2]的翻译： 创建 LABEL\_IR 类型的中间代码code\_label ，表示 label的位置。 其内容为**调用translateCond翻译右子表达式**，翻译过程生成了中间代码code2，code2的生成依赖于右子表达式的结果，根据该结果决定是跳转到labelTrue还是labelFalse。

**其他情况**

**视为处理单一（布尔）表达式**：根据表达式的真值，生成相应跳转指令。

* 条件为真的条件跳转： code\_ifgoto 为 IF\_GOTO\_IR 类型的指令，用于比较表达式 t1 和 常量0。 将中间代码code2的ops[0]置为表达式t1的值，ops[1]置为常量0，ops[2]置为GOTO到的位置labelTrue，relop置为!=，即条件为真、不等于常量0的时候跳转到 labelTrue。对应表格中code2 = [IF t1 != #0 GOTO label\_true]。
* 条件为假无条件跳转。code\_goto 指示在条件为假时跳转到labelFalse。

**总结**： 实验中实现了理论部分对应的一些内容，对于短路思想有了更深的体会。尤其是对于AND和OR条件表达式的短路翻译过程，根据PPT提示实现了巧妙的短路翻译过程。理解整个代码上下文之后，综合利用已有的代码模块完成**todo**的内容，比如利用符号表的查找操作、调用短路优化函数进行GOTO跳转优化，综合代码上下文和表格翻译过程提示来完成**todo**部分。