中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

编译器构造实验

Lab 3 - 中间代码生成

2023年秋季学期

Teaching Team: 王焱林、伍嘉栋、黄炎贤、王岩立、张俊鹏

特别说明

- 本课程需要和理论课《编译原理》配套学习
- 作业迟交：每天扣10%的作业分

3.1 实验内容

3.1.1 实验要求

- 在本次实验中，对C−−语言做如下假设
  - 假设1：不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量
  - 假设2：不会出现类型为结构体或高维数组（高于1维的数组）的变量。
  - 假设3：任何函数参数都只能为简单变量，也就是说，结构体和数组都不会作为参数传
  - 入函数中。
  - 假设4：没有全局变量的使用，并且所有变量均不重名。
  - 假设5：函数只会进行一次定义（没有函数声明）。
  - 假设6：函数无法进行嵌套定义。
  - 假设7：输入文件中不包含任何词法、语法或语义错误（函数也必有return语句）。

3.1.1 实验要求

- 你的程序需要将符合以上假设的C--源代码翻译为中间代码

表1. 中间代码的形式及操作规范。

语法	描述
LABEL x :	定义标号x。
FUNCTION f :	定义函数f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取y的地址赋给x。
x := *y	取以y值为地址的内存单元的内容赋给x。
*x := y	取y值赋给以x值为地址的内存单元。
GOTO x	无条件跳转至标号x。
IF x [relop] y GOTO z	如果x与y满足[relop]关系则跳转至标号z。
RETURN x	退出当前函数并返回x值。
DEC x [size]	内存空间申请，大小为4的倍数。
ARG x	传实参x。
x := CALL f	调用函数，并将其返回值赋给x。
PARAM x	函数参数声明。
READ x	从控制台读取x的值。
WRITE x	向控制台打印x的值。

3.1.1 实验要求

- 表中的操作大致可以分为如下几类：
  - 赋值语句可以对变量进行赋值操作。赋值号左边的x一定是一个变量或者临时变量，而赋值号右边的y既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将常数5赋给临时变量t1，可以写成t1 := #5。
  - 跳转语句分为无条件跳转和有条件跳转两种。无条件跳转语句GOTO x会直接将控制转移到标号为x的那一行，而有条件跳转语句（注意语句中变量、关系操作符前后都应该被空格或制表符分开）则会先确定两个操作数x和y之间的关系（相等、不等、小于、大于、小于等
  - 变量声明语句DEC用于为一个函数体内的局部变量声明其所需要的空间，该空间的大小以字节为单位。

3.1.1 实验要求

- 表中的操作大致可以分为如下几类：
  - 与函数调用有关的语句包括CALL、PARAM和ARG三种。
    - 其中PARAM语句在每个函数开头使用，对于函数中形参的数目和名称进行声明。
      - 若一个函数func有三个形参a、b、c，则该函数的函数体内前三条语句为：PARAM a、PARAM b和PARAM c。
    - CALL和ARG语句负责进行函数调用。
      - 仍以函数func为例，如果我们需要依次传入三个实参x、y、z，并将返回值保存到临时变量t1中，则可分别表述为：ARG z、ARG y、ARG x和t1 := CALL func。注意ARG传入参数的顺序和PARAM声明参数的顺序正好相反。
      - 当函数参数是结构体或数组时，ARG语句的参数为结构体或数组的地址

3.1.1 实验要求

- 完成以下部分或全部的要求：
  - 要求1：修改前面对C−−源代码的假设2和3，使源代码中：
    - 可以出现结构体类型的变量（但不会有结构体变量之间直接赋值）。
    - 结构体类型的变量可以作为函数的参数（但函数不会返回结构体类型的值）。
  - 要求2：修改前面对C−−源代码的假设2和3，使源代码中：
    - 一维数组类型的变量可以作为函数参数（但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（但高维数组类型的变量不会作为函数的参数或返回类值）。
  - 此外，实验三还会考察你的程序输出的中间代码的执行效率，因此你需要考虑如何优化中间代码的生成。

3.1.2 输入格式

- 输入是一个包含C--源代码的文本文件
- 接收输入文件名和输出文件名作为参数。假设你的程序名为cc、输入文件名为test1、输出文件名为out1.ir，程序和输入文件都位于当前目录下，那么在Linux命令行下运行./cc test1 out1.ir即可将输出结果写入当前目录下名为out1.ir的文件中。

3.1.3 输出格式

- 要求你的程序将运行结果输出到文件，输出文件要求每行一条中间代码，每条中间代码的含义如前文所述。
- 对每个特定的输入，并不存在唯一正确的输出。将使用虚拟机小程序对你的中间代码的正确性进行测试。任何能被虚拟机小程序顺利执行并得到正确结果的输出都将被接受。
- 此外，虚拟机小程序还会统计你的中间代码所执行过的各种操作的次数，以此来估计你的程序生成的中间代码的效率。

3.1.4 测试环境

- 你的程序将在如下环境中被编译并运行（同实验一）：
  - GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29
  - GCC version 4.6.3
  - GNU Flex version 2.5.35
  - GNU Bison version 2.5
  - 可以使用其它版本的Linux或者GCC等

3.1.5 提交要求

- 实验二要求提交如下内容（同实验一）：
  - 提交链接：https://send2me.cn/wPaU7_oL/RImHQjWxzFYBww
  - 截止时间：2023.11.29 23:59
  - Flex、Bison以及C语言的可被正确编译运行的源程序。
  - 结果输出文件放在result文件下，如result/test_case1.ir
  - 一份PDF格式的实验报告，内容包括：
    - 程序实现了哪些功能？简要说明如何实现这些功能
    - 程序应该如何被编译？可以使用脚本、makefile或逐条输入命令进行编译，请详细说明应该如何编译程序

3.1.6 样例

- 样例1：
  - 输入：

```
1  int main()
2  {
3    int n;
4    n = read();
5    if (n > 0) write(1);
6    else if (n < 0) write (-1);
7    else write(0);
8    return 0;
9  }
```

  - 输出：

```
 1  FUNCTION main :
 2  READ t1
 3  v1 := t1
 4  t2 := #0
 5  IF v1 > t2 GOTO label1
 6  GOTO label2
 7  LABEL label1 :
 8  t3 := #1
 9  WRITE t3
10  GOTO label3
11  LABEL label2 :
12  t4 := #0
13  IF v1 < t4 GOTO label4
14  GOTO label5
15  LABEL label4 :
16  t5 := #1
17  t6 := #0 - t5
18  WRITE t6
19  GOTO label6
20  LABEL label5 :
21  t7 := #0
22  WRITE t7
23  LABEL label6 :
24  LABEL label3 :
25  t8 := #0
26  RETURN t8
```

3.1.6 样例

➢ 样例1：

- 可以发现，这段中间代码中存在很多可以优化的地方。首先，0这个常数我们将其赋给了t2、t4、t7、t8这四个临时变量，实际上赋值一次就可以了。
- 其次，对于t6的赋值我们可以直接写成t6 := #−1而不必多进行一次减法运算。另外，程序中的标号也有些冗余。

```
 1  FUNCTION main :
 2  READ t1
 3  v1 := t1
 4  t2 := #0
 5  IF v1 > t2 GOTO label1
 6  IF v1 < t2 GOTO label2
 7  WRITE t2
 8  GOTO label3
 9  LABEL label1 :
10  t3 := #1
11  WRITE t3
12  GOTO label3
13  LABEL label2 :
14  t6 := #-1
15  WRITE t6
16  LABEL label3 :
17  RETURN t2
```

3.1.6 样例

- 样例2：
  - 输入

```
1  int fact(int n)
2  {
3    if (n == 1)
4      return n;
5    else
6      return (n * fact(n - 1));
7  }
8  int main()
9  {
10   int m, result;
11   m = read();
12   if (m > 1)
13     result = fact(m);
14   else
15     result = 1;
16   write(result);
17   return 0;
18 }
```

  - 输出：

```
1  FUNCTION fact :
2  PARAM v1
3  IF v1 == #1 GOTO label1
4  GOTO label2
5  LABEL label1 :
6  RETURN v1
7  LABEL label2 :
8  t1 := v1 - #1
9  ARG t1
10 t2 := CALL fact
11 t3 := v1 * t2
12 RETURN t3
13
14 FUNCTION main :
15 READ t4
16 v2 := t4
17 IF v2 > #1 GOTO label3
18 GOTO label4
19 LABEL label3 :
20 ARG v2
21 t5 := CALL fact
22 v3 := t5
23 GOTO label5
24 LABEL label4 :
25 v3 := #1
26 LABEL label5 :
27 WRITE v3
28 RETURN #0
```

3.1.6 样例

➢ 样例3

- 输入

```
1  struct Operands
2  {
3    int o1;
4    int o2;
5  };
6
7  int add(struct Operands temp)
8  {
9    return (temp.o1 + temp.o2);
10 }
11
12 int main()
13 {
14   int n;
15   struct Operands op;
16   op.o1 = 1;
17   op.o2 = 2;
18   n = add(op);
19   write(n);
20   return 0;
21 }
```

- 输出：

  - 样例输入中出现了结构体类型的变量，若完成要求1:

```
1  FUNCTION add :
2  PARAM v1
3  t2 := *v1
4  t7 := v1 + #4
5  t3 := *t7
6  t1 := t2 + t3
7  RETURN t1
8  FUNCTION main :
9  DEC v3 8
10 t9 := &v3
11 *t9 := #1
12 t12 := &v3 + #4
13 *t12 := #2
14 ARG &v3
15 t14 := CALL add
16 v2 := t14
17 WRITE v2
18 RETURN #0
```

3.1.6 样例

➢ 样例4

- 输入

```
1  int add(int temp[2])
2  {
3    return (temp[0] + temp[1]);
4  }
5
6  int main()
7  {
8    int op[2];
9    int r[1][2];
10   int i = 0, j = 0;
11   while (i < 2)
12   {
13     while (j < 2)
14     {
15       op[j] = i + j;
16       j = j + 1;
17     }
18     r[0][i] = add(op);
19     write(r[0][i]);
20     i = i + 1;
21     j = 0;
22   }
23   return 0;
24 }
```

- 输出:

```
1  FUNCTION add :
2  PARAM v1
3  t2 := *v1
4  t11 := v1 + #4
5  t3 := *t11
6  t1 := t2 + t3
7  RETURN t1
8  FUNCTION main :
9  DEC v2 8
10 DEC v3 8
11 v4 := #0
12 v5 := #0
13 LABEL label1 :
14 IF v4 < #2 GOTO label2
15 GOTO label3
16 LABEL label2 :
17 LABEL label4 :
18 IF v5 < #2 GOTO label5
19 GOTO label6
20 LABEL label5 :
21 t18 := v5 * #4
22 t19 := &v2 + t18
23 t20 := v4 + v5
24 *t19 := t20
25 v5 := v5 + #1
26 GOTO label4
27 LABEL label6 :
28 t31 := v4 * #4
29 t32 := &v3 + t31
30 ARG &v2
31 t33 := CALL add
32 *t32 := t33
33 t41 := v4 * #4
34 t42 := &v3 + t41
35 t35 := *t42
36 WRITE t35
37 v4 := v4 + #1
38 v5 := #0
39 GOTO label1
40 LABEL label3 :
41 RETURN #0
```

3.1.7 参考代码（挖空）

- 一共挖了五个空，集中在语句表达式的翻译和条件表达式的翻译，对应理论知识中比较重要的两块内容，具体：
  - 1）加法操作，可参考减法，比较简单

```
// *** todo ***
/*
You need to finish Plus operation. You can refer to other subtraction and similar operations
*/
InterCode optimizePLUSIR(Operand dest, Operand src1, Operand src2) {
    // todo
}
```

  - 2）乘法操作，可参考除法，比较简单

```
// *** todo ***
/*
You need to finish Mul operation. You can refer to other Div and similar operations
*/
InterCode optimizeMULIR(Operand dest, Operand src1, Operand src2) {
    // todo
}
```

3.1.7 参考代码（挖空）

- 一共挖了五个空，主要集中在语句表达式的翻译和条件表达式的翻译，对应理论知识中比较重要的两块内容，具体：
  - 3）表达式翻译，左边是单变量，主要是三地址码赋值如何实现的问题

```
// *** todo ***
/*
You need to finish expression translation if the exp is just one variable
You need to add a temporary variable,
then assign the expression to the temporary variable, and finally assign the temporary variable.
*/
if (root->children[0]->childNum == 1 &&
    strcmp(root->children[0]->children[0]->name, "ID") == 0) {
    // todo
}
```

3.1.7 参考代码（挖空）

- 一共挖了五个空，主要集中在语句表达式的翻译和条件表达式的翻译，对应理论知识中比较重要的两块内容，具体：
  - 4）条件表达式 And OR 等
  - 5）短路翻译对应理论部分的短路翻译和条件表达式

```
// *** todo ***
/*
You need to finish expression translation if the exp is an conditional expression
"You need to make appropriate use of the translateCond function.
*/
else if (root->childNum >= 2 && (
        strcmp(root->children[0]->name, "NOT") == 0 ||
        strcmp(root->children[1]->name, "RELOP") == 0 ||
        strcmp(root->children[1]->name, "AND") == 0 ||
        strcmp(root->children[1]->name, "OR") == 0)) {
    // todo
}
```

```
// *** todo ***
/*
You need to translate conditional expressions.
You can refer to the lab manual and textbooks.
Pay attention to how to implement short-circuit translation.
*/
InterCode translateCond(Node* root, Operand labelTrue, Operand labelFalse) {
    // todo
}
```

3.2 实验指导

3.2.1中间代码的表示（线形）

- 实验三会将所生成的中间代码先保存到内存中，等全部翻译完毕，优化也都做完后再使用一个专门的打印函数把在内存中的中间代码打印出来。
- 线形IR是一种保存中间代码的数据结构
- 实现起来最简单，而且打印起来最方便的

3.2.1中间代码的表示（线形）

- 图5(a)为一个大的静态数组
  - 数组中的每个元素（图中的一行）就是一条中间代码
  - 实现简单；但是中间代码的最大行数受限，而且代码的插入、删除以及调换位置的代价较大

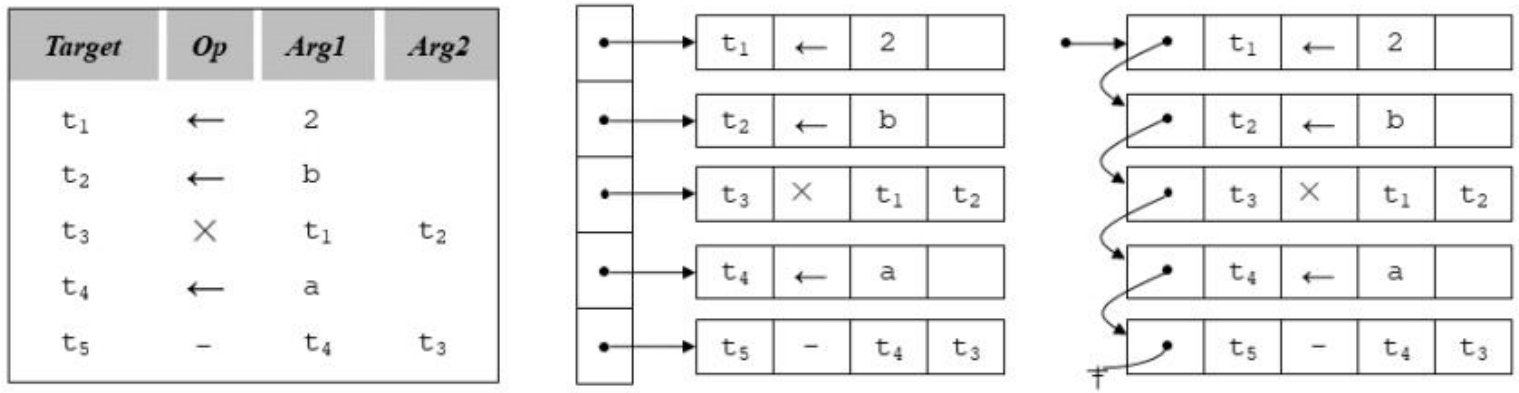

图5. 表示线性IR的三种基本数据结构。

3.2.1中间代码的表示（线形）

- 图5(b)同样为一个大数组
  - 数组中的每个元素并不是一条中间代码，而是一个指向中间代码指针
- 图5(c)是一个纯链表的实现
  - 建议使用双向循环链表。链表以增加实现的复杂性为代价换得了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且几乎不存在代码最大行数的限制。

3.2.1中间代码的表示（线形）

➢ 单条中间代码的数据结构定义

```
1  typedef struct Operand_* Operand;
2  struct Operand_ {
3    enum { VARIABLE, CONSTANT, ADDRESS, … } kind;
4    union {
5      int var_no;
6      int value;
7      …
8    } u;
9  };
10
11 struct InterCode
12 {
13   enum { ASSIGN, ADD, SUB, MUL, … } kind;
14   union {
15     struct { Operand right, left; } assign;
16     struct { Operand result, op1, op2; } binop;
17     …
18   } u;
19 }
```

- 图5(a)中的实现可以写成：

```
InterCode codes[MAX_LINE];
```

- 图5(b)中的实现可以写成：

```
InterCode* codes[MAX_LINE];
```

- 图5(c)中的实现可以写成：

```
struct InterCodes { InterCode code; struct InterCodes *prev, *next; };
```

3.2.2 中间代码的表示（树形）

- 树形IR只需在原有语法树基础上稍加修改即可
- 对树形IR进行（深度优先）遍历，根据当前结点的类型递归地对其各个子结点进行打印。

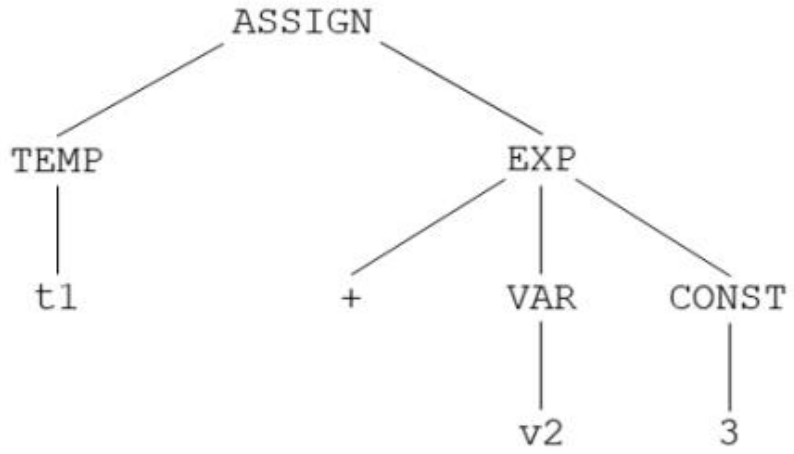

图6. 中间代码t1 := v2 + #3的树形结构表示。

3.2.3翻译模式（基本表达式）

- 如何将语法树变成中间代码？
  - 遍历语法树中的每一个结点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码。
  - 借助实验二的语法制导翻译（SDT）。具体到代码上，我们可以为每个主要的语法单元“X”都设计相应的翻译函数“translate_X”，对语法树的遍历过程也就是这些函数之间互相调用的过程。

3.3.3翻译模式（基本表达式）

- 表达式的翻译
  - 函数translate_Exp()，它接受三个参数：语法树的结点Exp、符号表sym_table以及一个变量名place
  - 并返回一段语法树当前结点及其子孙结点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。

表2. 基本表达式的翻译模式。

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value][2]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[3] (Exp_1 → ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] +[4] [place := variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp_1 RELOP Exp_2	label1 = new_label() label2 = new_label() code0 = [place := #0] code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]
NOT Exp_1	
Exp_1 AND Exp_2	
Exp_1 OR Exp_2	

3.2.3翻译模式（基本表达式）

- 表达式的翻译
  - 如果Exp产生了一个整数INT，那么我们只需要为传入的place变量赋值成前面加上一个“#”的相应数值即可。
  - 如果Exp产生了一个标识符ID，那么我们只需要为传入的place变量赋值成ID对应的变量名（或该变量对应的中间代码中的名字）即可。
  - 如果Exp产生了赋值表达式Exp1 ASSIGNOP Exp2：
    - 这里仅列出当Exp1 → ID时应该如何进行翻译。我们需要通过查表找到ID对应的变量，然后对Exp2进行翻译（运算结果储存在临时变量t1中），再将t1中的值赋于ID所对应的变量并将结果再存回place，最后把刚翻译好的这两段代码合并随后返回即可。

3.2.3翻译模式（基本表达式）

- 表达式的翻译
  - 如果Exp产生了算术运算表达式Exp1 PLUS Exp2：
    - 先对Exp1进行翻译（运算结果储存在临时变量t1中），再对Exp2进行翻译（运算结果储存在临时变量t2中），最后生成一句中间代码place := t1 + t2，并将刚翻译好的这三段代码合并后返回即可。
  - 如果Exp产生了取负表达式MINUS Exp1：
    - 则先对Exp1进行翻译（运算结果储存在临时变量t1中），再生成一句中间代码place := #0 − t1从而实现对t1取负，最后将翻译好的这两段代码合并后返回。

3.2.4 翻译模式（语句）

- 语句的翻译
  - 表达式语句、复合语句、返回语句、跳转语句和循环语句。

表3. 语句的翻译模式。

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_CompSt(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1 + code2
IF LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP $Stmt_1$ ELSE $Stmt_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) code3 = translate_Stmt($Stmt_2$, sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

3.2.4 翻译模式（语句）

- 条件表达式的翻译

表4. 条件表达式的翻译模式。

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp_1	return translate_Cond(Exp_1, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label1, label_false, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label_true, label1, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

3.2.5 翻译模式（函数调用）

- 函数调用的翻译

表5. 函数调用的翻译模式。

translate_Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	function = lookup(sym_table, ID) if (function.name == "read") return [READ place] return [place := CALL function.name]
ID LP Args RP	function = lookup(sym_table, ID) arg_list = NULL code1 = translate_Args(Args, sym_table, arg_list) if (function.name == "write") return code1 + [WRITE arg_list[1]] + [place := #0] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name]

3.2.5 翻译模式（函数调用）

- 函数参数的翻译

表6. 函数参数的翻译模式。

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA $Args_1$	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args($Args_1$, sym_table, arg_list) return code1 + code2

3.2.6 翻译模式（数组与结构体）

- 数组或结构体的跟变量处理不同地方在于内存地址的运算
  - 以三维数组为例，假设有数组int array[7][8][9]，为了访问数组元素array[3][4][5]：
    - 首先需要找到三维数组array的首地址（直接对变量array取地址即可）
    - 然后找到二维数组array[3]的首地址(array的地址加上3 乘以二维数组的大小（8*9 ）再乘以int类型的宽度4)
    - 然后找到一维数组array[3][4]的首地址（array[3]的地址加上4乘以一维数组的大小（9）再乘以int类型的宽度4），
    - 最后找到整array[3][4][5]的地址（array[3][4]的地址加上5乘以int类型的宽度4）。整个运算过程可以表示为：

$$\mathrm{ADDR}(\mathrm{array}[i][j][k]) = \mathrm{ADDR}(\mathrm{array}) + \sum_{t=0}^{i-1} \mathrm{SIZEOF}(\mathrm{array}[t]) + \sum_{t=0}^{j-1} \mathrm{SIZEOF}(\mathrm{array}[i][t]) + \sum_{t=0}^{k-1} \mathrm{SIZEOF}(\mathrm{array}[i][j][])。$$

3.2.6 翻译模式（数组与结构体）

➢ 结构体内存访问

- 假设要访问结构体struct { int x[10]; int y, z; } st中的域z，
  - 我们首先找到变量st的首地址，然后找到st中域z的首地址（st的地址加上数组x的大小（4*10）再加上整数y的宽度4）。

$$\mathrm{ADDR}(\mathrm{st}.field_n) = \mathrm{ADDR}(\mathrm{st}) + \sum_{t=0}^{n-1} \mathrm{SIZEOF}(\mathrm{st}.field_t)。$$

3.2.7 中间代码生成提示

- 可以在实验二的语义分析部分添加中间代码生成的内容，使编译器可以一边进行语义检查一边生成中间代码；
- 也可以将关于中间代码生成的所有内容写到一个单独的文件中，等到语义检查全部完成并通过之后再生成中间代码。
- 前者会让你的编译器效率高一些，后者会让你的编译器模块性更好一些。

3.2.7 中间代码生成提示

- 确定了在哪里进行中间代码生成之后，下一步就要实现中间代码的数据结构，然后按照输出格式的要求自己编写函数将你的中间代码打印出来。
- 接下来的任务是根据前面介绍的翻译模式完成一系列的translate函数。我们已经给出了Exp和Stmt的翻译模式，你还需要考虑包括数组、结构体、数组与结构体定义、变量初始化、语法单元CompSt、语法单元StmtList在内的翻译模式。
- 最后，虚拟机小程序将以总共执行过的中间代码条数为标准来衡量你的编译器所输出的中间代码的运行效率。