# 哈尔滨工业大学

## 编译系统 2023 春

## 实验三

| | |
|---|---|
| 学院： | 未来技术学院 |
| 姓名： | 刘天瑞 |
| 学号： | 7203610121 |
| 指导教师： | 朱庆福 |

# 一、 实验目的

a) 巩固对中间代码生成的基本功能和原理的认识；
b) 能够基于语法指导翻译的知识进行中间代码生成；
c) 掌握类高级语言中基本语句所对应的语义动作。

# 二、 实验内容

## (一) 实验过程

### 1. 实现功能

根据之前实验的词法分析和语法分析我已经构造出了语法分析树和各类符号表，在此基础上，我又完成了 interim.c 的设计，即中间代码生成。整体的处理流程就是将源程序所生成的中间代码的内容存放到 InterCodes 这个链表中，所以我需要做的就是对一些语法单元编写一个 translate()函数，通过语法分析树的根节点 Program 的翻译调用到子树，进行递归翻译来实现所有中间代码的生成。

本次实验实现了以上所说的所有功能，对于实验指导书上的样例即完成了必做样例的中间代码生成，也完成了选做的样例。

### 2. 数据结构

本次实验我主要用到的数据结构还是两个自己定义的结构体。一个是操作数结构体 Operand，因为我需要打印中间代码生成的每个操作数是什么类型和数值，如此一来便可以在生成中间代码的过程中明确操作数是什么，其中 CONSTANT 和 RELOP1 需要拥有 value 值，其他类型需要拥有 name 值。其具体构造如下图所示：

```
struct Operand_ {
    enum { VARIABLE, CONSTANT, ADDRESS, LABEL_OP, RELOP1, DER } kind;
    union {
        char* name;
        int value;
    } u;
};
```

另外一个结构体是用来保存生成结果的 InterCode。考虑到写入和读出的方便，以及为了保存所有中间生成的代码，我还定义了一个双向链表 InterCodes 来保存前后节点的信息。其具体构造如下图所示：

```
struct InterCode_
{
    enum { LABEL_IR, FUNCTION, ASSIGN, ADD1, SUB1, MUL1, DIV1, GOTO, IF_GOTO, RETURN1, DEC, ARG, CALL, PARAM, READ, WRITE } kind;
    union {
        struct { Operand op; } oneop;
        struct { Operand right, left; } assign;
        struct { Operand result, op1, op2; } binop;
        struct { Operand x, relop, y, z; } ifgoto;
        struct { Operand op;  int size; } dec;
    } u;
};
```

### 3. 翻译过程

本来想面向测试用例来进行编程翻译，但是有的翻译函数需要调用子节点的翻译函数，在编写每个翻译函数中，我发现一直有问题，所以最后还是保证了所有的语法单元都需要有对应的翻译函数。在定义操作的时候需要定义一个操作数并赋予其字段值，而在生成一句中间代码时需要调用对应的 gen_intercode()函数将新生成的中间代码加入到 InterCodes 链表中，例如在 translate_FunDec()函数中就需要生成 Function 代码，即调用了 gen_intercode3(“FUNCTION”, func)函数；该具体运行部分如下图所示：

```
void translate_FunDec(node* FunDec, tablenode* sym_table)
{
    if(FunDec->numchild == 3)
    {
        Operand func = (Operand)malloc(sizeof(struct Operand_));
        func->kind = VARIABLE;
        func->u.name = FunDec->children[0]->IDvalue;
        gen_intercode3("FUNCTION", func);
    }
```

### 4. 个性内容

除上述基本实验内容之外，我在各个翻译函数生成对应中间代码之前还编写了四个不同的中间代码生成函数，以此先根据操作数个数先统一进行函数调用，后续再通过对应的语法单元执行对应的生成函数。这些函数的功能在于能够根据传入的中间代码种类和操作数生成一个新的中间代码节点加入到 InterCodes 中。以四个参数的 if_goto()条件跳转语句为例，其具体运行部分如下图所示：

```
void gen_intercode0(char* kind, Operand op1, Operand op2, Operand op3, Operand op4)
{
    InterCodes newcode = (InterCodes)malloc(sizeof(struct InterCodes_));
    newcode->code = (InterCode)malloc(sizeof(struct InterCode_));
    if(strcmp(kind, "IF_GOTO") == 0)
    {
        newcode->code->kind = 8;
        newcode->code->u.ifgoto.x = op1;
        newcode->code->u.ifgoto.relop = op2;
        newcode->code->u.ifgoto.y = op3;
        newcode->code->u.ifgoto.z = op4;
    }
    last->next = newcode;
    newcode->prev = last;
    newcode->next = NULL;
    last = newcode;
}
```

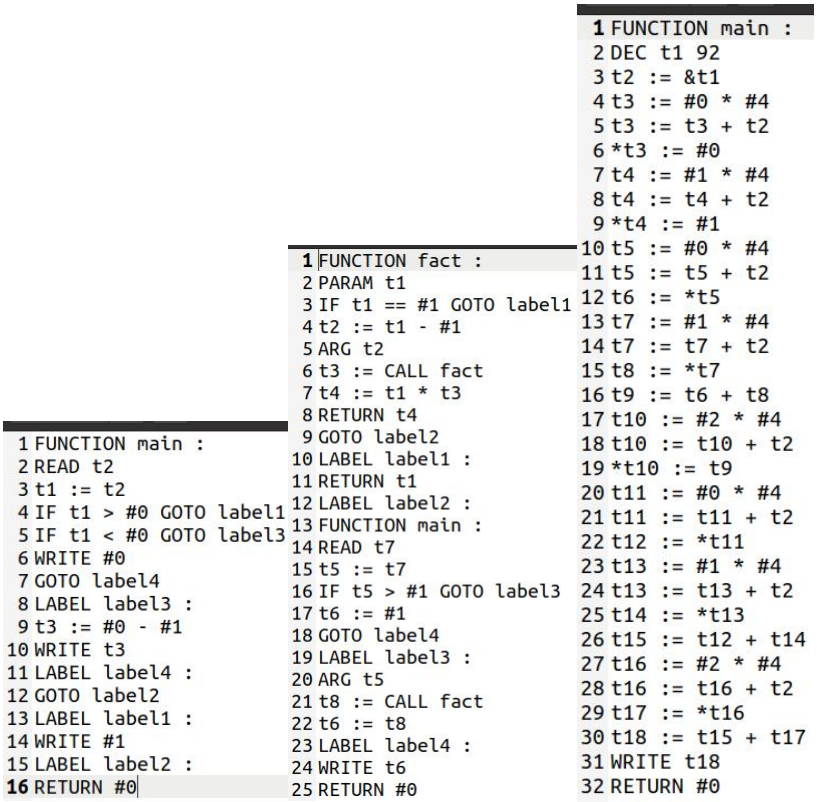
为了能够完整连接所有中间代码节点，我此处还使用了 geninterCodeList()函数生成了一个中间代码链表的前端节点，然后将它指向 last，表明也同时指向尾端节点。然后在所有生成中间代码函数中将新节点赋给 last->next 字段，last 赋给 newcode->prev 字段，随后再将 newcode->next 字段赋值为空，这样一来新节点会作为尾端节点继续延伸下去。

### 5. 编译过程

本次实验我并没有采用 Makefile 进行编译，而是直接在命令行进行输入：

flex lexical.l；bison syntax.y；gcc main.c syntax.tab.c -lfl -ly -o parser

在终端运行./parser test1.cmm 1t.ir，./parser test2.cmm 2t.ir 以及./parser test3.cmm 3t.ir 中间代码内容如下图所示（提供的虚拟机小程序执行文件 irsim 均可通过）：



```
1 FUNCTION main :
2 READ t2
3 t1 := t2
4 IF t1 > #0 GOTO label1
5 IF t1 < #0 GOTO label3
6 WRITE #0
7 GOTO label4
8 LABEL label3 :
9 t3 := #0 - #1
10 WRITE t3
11 LABEL label4 :
12 GOTO label2
13 LABEL label1 :
14 WRITE #1
15 LABEL label2 :
16 RETURN #0
```

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

```
1 FUNCTION main :
2 DEC t1 92
3 t2 := &t1
4 t3 := #0 * #4
5 t3 := t3 + t2
6 *t3 := #0
7 t4 := #1 * #4
8 t4 := t4 + t2
9 *t4 := #1
10 t5 := #0 * #4
11 t5 := t5 + t2
12 t6 := *t5
13 t7 := #1 * #4
14 t7 := t7 + t2
15 t8 := *t7
16 t9 := t6 + t8
17 t10 := #2 * #4
18 t10 := t10 + t2
19 *t10 := t9
20 t11 := #0 * #4
21 t11 := t11 + t2
22 t12 := *t11
23 t13 := #1 * #4
24 t13 := t13 + t2
25 t14 := *t13
26 t15 := t12 + t14
27 t16 := #2 * #4
28 t16 := t16 + t2
29 t17 := *t16
30 t18 := t15 + t17
31 WRITE t18
32 RETURN #0
```

1t.ir　　2t.ir　　3t.ir

## 三、 实验总结

中间代码是指在源代码和目标代码之间的一种代码表示形式。它可以是一种抽象的高级语言表示形式，也可以是一种类似于汇编语言的低级语言表示形式。中间代码的生成是编译器的一个重要步骤，它将源代码翻译成中间代码，然后再将中间代码转换成目标代码。

生成中间代码：在语法分析和符号表处理之后，编译器将使用语法树和符号表来生成中间代码。中间代码可以是三地址码、四元式或类似汇编语言的低级代码。每个中间代码指令表示一个简单的操作，如赋值、加法、乘法或函数调用。

优化中间代码：编译器可以对生成的中间代码进行优化。中间代码优化的目的是生成更有效率的目标代码。优化可以包括删除无用代码、代数化简、常量折叠和代码移动等技术。

总的来说，中间代码的生成是编译器的一个重要步骤。通过生成中间代码，编译器可以将源代码翻译成一种易于优化和转换成目标代码的形式。

引入中间代码有两个主要的好处。一方面，中间代码将编译器自然地分为前端和后端两个部分。当我们需要改变编译器的源语言或目标语言时，如果采用了中间代码，我们只需要替换原编译器的前端或后端，而不需要重写整个编译器。另一方面，即使源语言和目标语言是固定的，采用中间代码也有利于编译器的模块化。人们将编译器设计中那些复杂但相关性不大的任务分别放在前端和后端的各个模块中，这既简化了模块内部的处理，又使我们能单独对每个模块进行调试与修改而不影响其它模块。