编译原理 Lab3

202220013 徐简

项目环境:

1. GNU LINUX Realease: Ubuntu
2. GCC version: 7.5.0
3. GNU Flex version:2.6.4
4. GNU Bison version: 3.0.4

实现的功能

中间代码生成

- 完成了必做
- 完成了选做1，即可以出现结构体变量（未完成选做2）

代码实现

- 首先添加read和write函数的信息(添加READ和WRITE枚举类型即可)

```
struct InterCode {
    enum { LABEL, FUNCTION, ASSIGN, ADD, SUB, MUL, DIV, GOTO, IF, RETURN, DEC, ARG,
CALL, PARAM, READ, WRITE } kind;
 //...
};
```

- 按照实验手册中推荐的方法，采用双向链表来记录中间代码

```
struct InterCode {
  //...
};

struct InterCodes {
    InterCode code;
    bool isDelete;
    InterCodes *prev, *next;
    //双向链表的头尾结点
};
```

- 根据手册中的翻译模式，实现各翻译函数translate_X

```
void translate_Program(Node* node);
void translate_ExtDefList(Node* node);
void translate_ExtDef(Node* node);
//...

//根据lab2的到的分析树，进行翻译
void translate_Program(Node* node)
{
    translate_ExtDefList(node->child);
}
```

- 一系列函数，实现双向链表的建立，主要的操作是设置InterCode的属性，并加入到双向链表中

```
void createAssign(unsigned type, Operand* left, Operand* right);
void createBinaryOp(unsigned kind, unsigned type, Operand* res, Operand* op1, Operand*
op2);
void createSingleOp(unsigned kind, Operand* res, Operand* op);
void createSingle(unsigned kind, Operand* op);
void createCond(Operand* op1, Operand* op2, Operand* target, char* re);
void createDec(Operand* op, unsigned size);
```

- 将得到的中间代码，输出到文件中

```
void writeInterCodes(const char* fielname)
{
    InterCodes* p = head->next;
    FILE* f = fopen(fielname, "w");
    while (p) {
      //fprintf 写入文件
    }
    fclose(f);
}
```

- 结构体实现，用count Size函数来计算每个Type的大小，对于结构体，则递归地将结果相加返回，从而确定要申请的空间的大小。这样可以实现对结构体中每个成员的表示。测试用例3就说明了这一原理（结构体中含有两个int型的变量）。

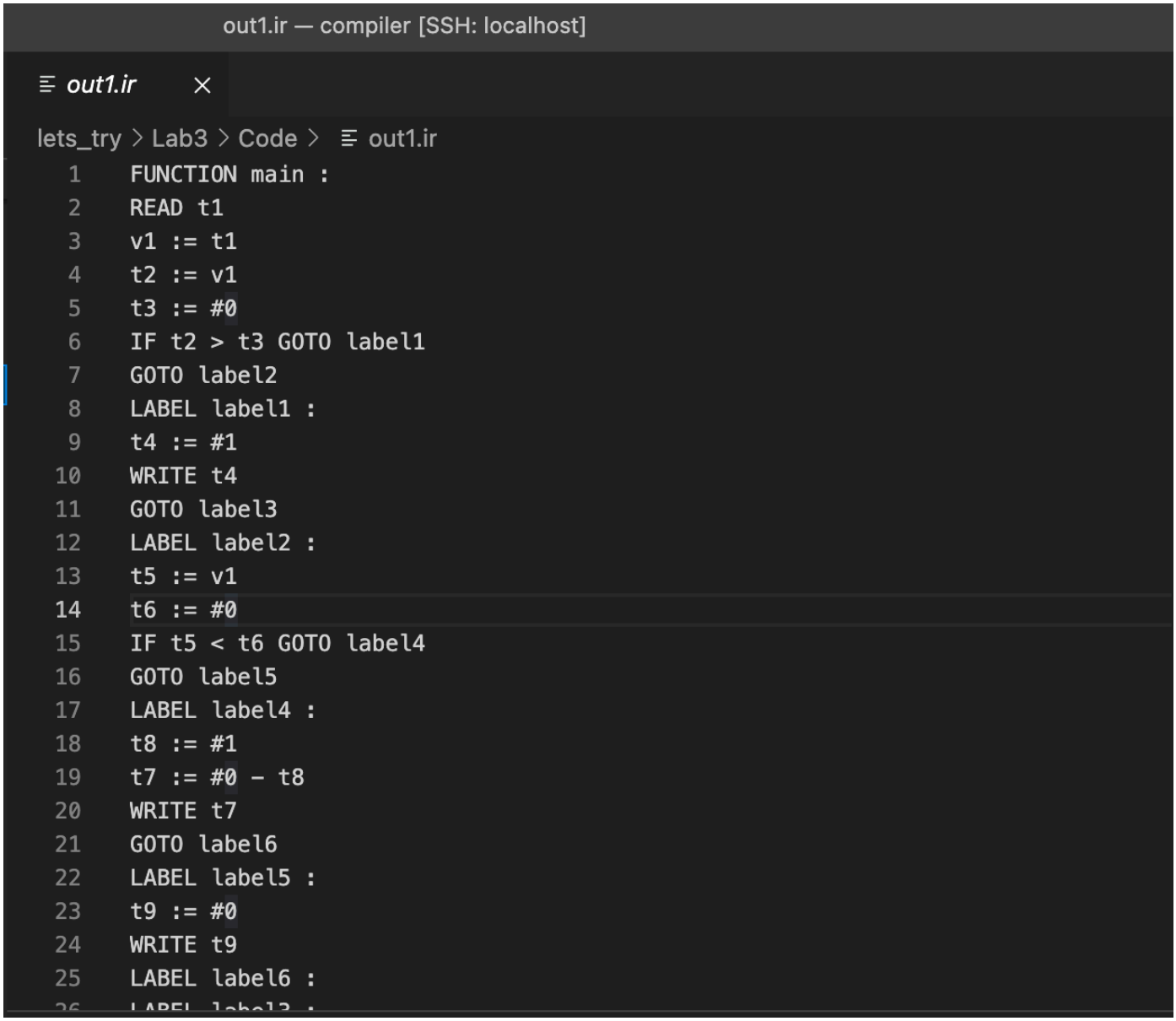
```
unsigned countSize(Type* type);
```

```
FUNCTION add :
PARAM v5
t4 := v5 + #0
t2 := *t4
t5 := v5 + #4
t3 := *t5
t1 := t2 + t3
RETURN t1
```

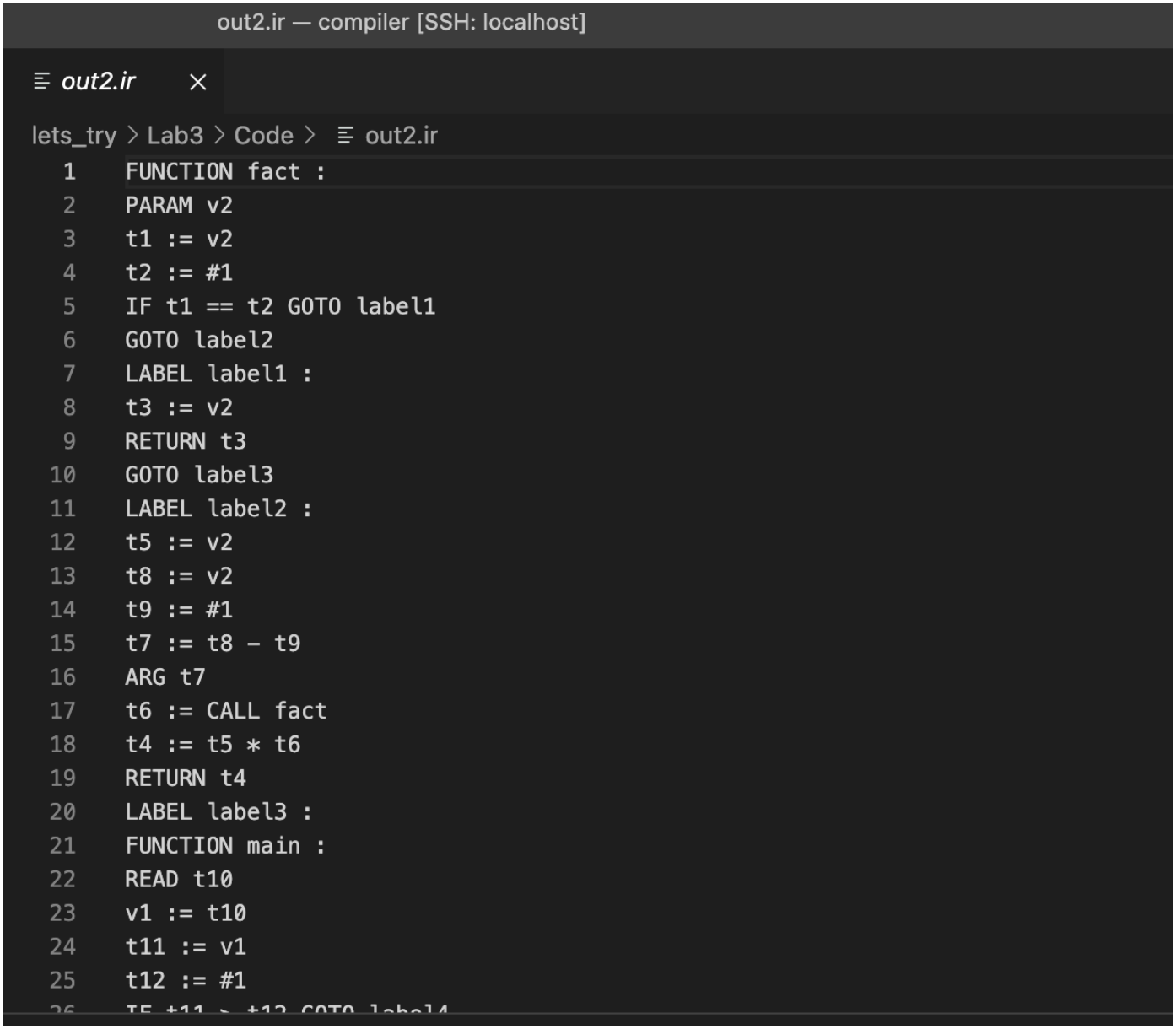
测试

- 手册测试样例运行截图如下

test1


out1.ir — compiler [SSH: localhost]
out1.ir
lets_try > Lab3 > Code > out1.ir
1 FUNCTION main :
2 READ t1
3 v1 := t1
4 t2 := v1
5 t3 := #0
6 IF t2 > t3 GOTO label1
7 GOTO label2
8 LABEL label1 :
9 t4 := #1
10 WRITE t4
11 GOTO label3
12 LABEL label2 :
13 t5 := v1
14 t6 := #0
15 IF t5 < t6 GOTO label4
16 GOTO label5
17 LABEL label4 :
18 t8 := #1
19 t7 := #0 - t8
20 WRITE t7
21 GOTO label6
22 LABEL label5 :
23 t9 := #0
24 WRITE t9
25 LABEL label6 :

test2


out2.ir — compiler [SSH: localhost]
out2.ir
lets_try > Lab3 > Code > out2.ir
1 FUNCTION fact :
2 PARAM v2
3 t1 := v2
4 t2 := #1
5 IF t1 == t2 GOTO label1
6 GOTO label2
7 LABEL label1 :
8 t3 := v2
9 RETURN t3
10 GOTO label3
11 LABEL label2 :
12 t5 := v2
13 t8 := v2
14 t9 := #1
15 t7 := t8 - t9
16 ARG t7
17 t6 := CALL fact
18 t4 := t5 * t6
19 RETURN t4
20 LABEL label3 :
21 FUNCTION main :
22 READ t10
23 v1 := t10
24 t11 := v1
25 t12 := #1

test3

out3.ir — compiler [SSH: localhost]

out3.ir

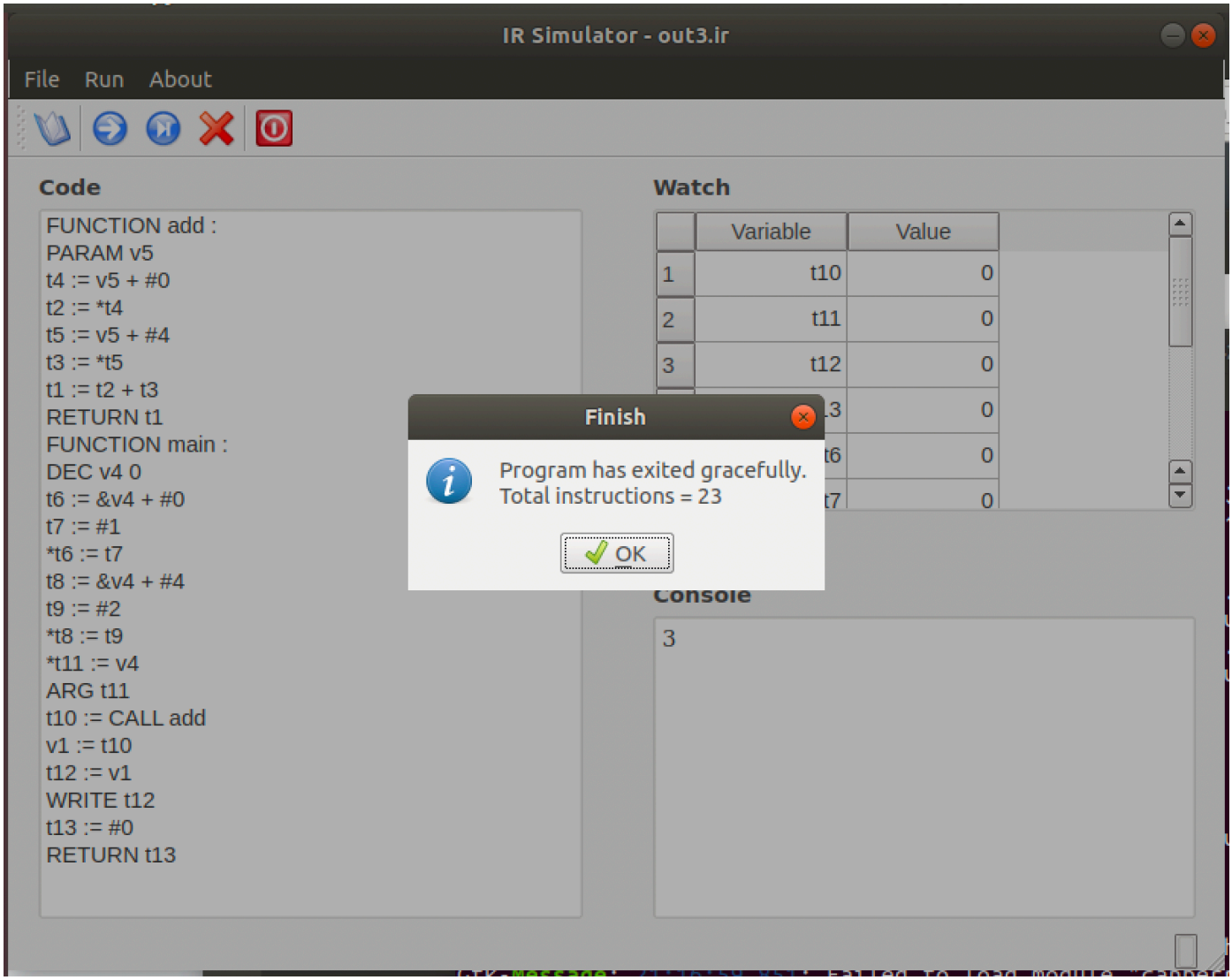
lets_try > Lab3 > Code > out3.ir

```
1   FUNCTION add :
2   PARAM v5
3   t4 := v5 + #0
4   t2 := *t4
5   t5 := v5 + #4
6   t3 := *t5
7   t1 := t2 + t3
8   RETURN t1
9   FUNCTION main :
10  DEC v4 0
11  t6 := &v4 + #0
12  t7 := #1
13  *t6 := t7
14  t8 := &v4 + #4
15  t9 := #2
16  *t8 := t9
17  *t11 := v4
18  ARG t11
19  t10 := CALL add
20  v1 := t10
21  t12 := v1
22  WRITE t12
23  t13 := #0
24  RETURN t13
25
```

test4(未实现数组)

```
njucs@njucs-VirtualBox:~/compiler/lets_try/Lab3/Code$ ./parser ../test/test4.cmm out4.ir
Cannot translate_: Code contains variables of multi-dimensional array type or parameters of array type.
```

- 在IR Simulator中运行截图如下



使用方法

使用makefile完成编译、运行：

- `make`，`make clean`
- 测试使用命令`./parser ../Test/test1.cmm out1.ir`。中间代码会被保存到当前目录的`outxxx.ir`中

实验思考

- debug花了很多时间，但是测试用例的强度还是太低了。
- 时间原因，也没有做中间代码生成的优化，实验四有机会再尝试。