

哈尔滨工业大学

编译系统 2022 春

实验三

学院:	计算学部
姓名:	冯开来
学号:	1190201215
指导教师:	陈鄞

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们已经构造除了语法树和符号表，在这个基础上，我们完成了 `intercodegenerate.c` 的设计，即中间代码生成。整体的流程就是将源程序生成的中间代码内容存到 `InterCodes` 这个链表中，所以我们需要做的就是对一些语法单元编写 `translate` 函数，通过语法树根节点 `Program` 的翻译调用到子树，进行递归翻译实现所有中间代码的生成。

本次实验实现了以上所说的功能，对于实验指导书上的样例只完成了必做样例的中间代码生成。

2. 数据结构

本次实验主要用到的数据结构是两个自己定义的结构体。一个是操作数结构体，因为我们需要打印中间代码生成的每个操作数是什么类型和数值。这样我们在生成中间代码的过程中能明确操作数是什么，其中 `CONSTANT` 和 `RELOP1` 需要拥有 `value` 值，其他类型需要拥有 `name` 值。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;
};
```

还有一个结构体是就是用来保存生成结果的 `InterCode`。为了写入和读出方便，以及保存所有中间生成的代码，我定义了双向链表 `InterCodes` 来保存前后节点的信息。具体构造如下图：

```

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由对应的翻译函数。在定义操作的时候，我们需要定义一个操作数并赋予字段值，在生成一句中间代码是，我们需要调用对应的 `gen_intercode()` 函数将新生成的中间代码加入到 `InterCodes` 这个链表中，比如在 `translate_FunDec` 函数中就需要生成 Function f 代码，即调用 `gen_intercode3("FUNCTION", func)`；函数部分如果所示：

```

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);
    }
    else
    {

```

4. 个性内容

特别的，我在各个翻译函数生成对应中间代码之前，我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，后续在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。我们以四个参数的 `if_goto()` 语句为例，如下图：

```

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;
}

```

为了能够完好的连接所有的中间代码节点，我这里使用了 `geninterCodeList()` 生成一个中间代码链表的首节点，然后将它指向 `last`，表明也同时指向尾节点。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，将 `newcode->next` 赋值为空，这样新节点作为尾节点继续延伸下去。

5. 编译过程

本次实验采用 Makefile 进行编译。

```
CC = gcc
FLEX = flex
BISON = bison

parser: main.c lex.yy.c syntax.tab.c
$(CC) main.c syntax.tab.c -lfl -o parser -g

lex.yy.c: lexical.l
$(FLEX) lexical.l

syntax.tab.c: syntax.y
$(BISON) -d -t syntax.y

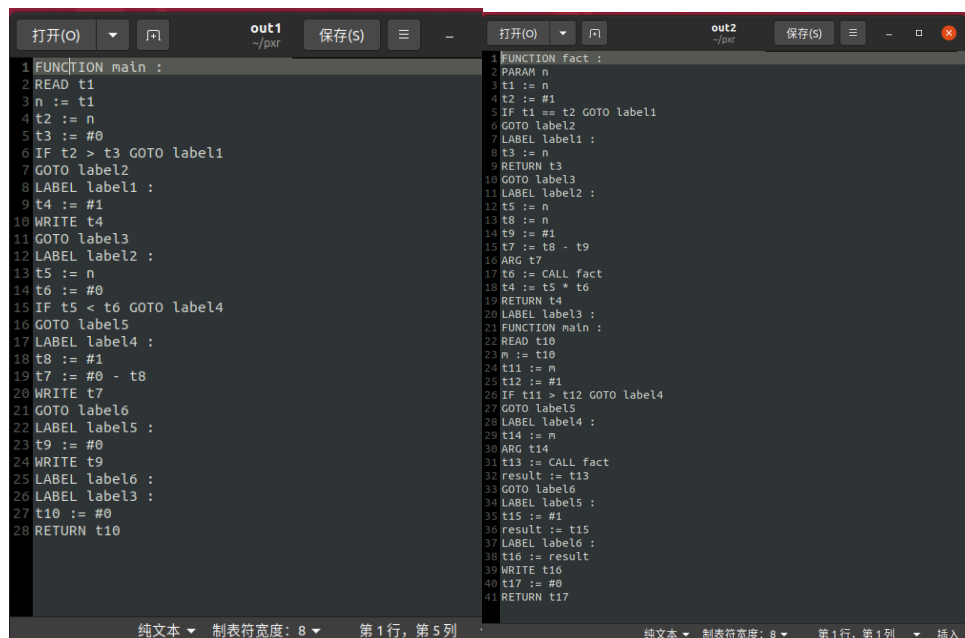
.PHONY: clean test

test:
./parser test1.c
./parser test2.c
./parser test3.c
./parser test4.c
./parser test5.c
./parser test6.c
./parser test7.c
./parser test8.c
./parser test9.c
./parser test10.c

clean:
rm -f parser lex.yy.c syntax.tab.c syntax.tab.h syntax.output
```

(二) 实验结果

在终端运行 `./parser test1.cmm out1` 和 `./parser test2.cmm out2`



```
1 FUNCTION main :
2 READ t1
3 n := t1
4 t2 := n
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 := n
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 :
26 LABEL label3 :
27 t10 := #0
28 RETURN t10

1 FUNCTION fact :
2 PARAM n
3 t1 := n
4 t2 := #1
5 IF t1 == t2 GOTO label1
6 GOTO label2
7 LABEL label1 :
8 t3 := n
9 RETURN t3
10 GOTO label3
11 LABEL label2 :
12 t5 := n
13 t8 := n
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 n := t10
24 t11 := n
25 t12 := #1
26 IF t11 > t12 GOTO label4
27 GOTO label5
28 LABEL label4 :
29 t14 := n
30 ARG t14
31 t13 := CALL fact
32 result := t13
33 GOTO label6
34 LABEL label5 :
35 t15 := #1
36 result := t15
37 LABEL label6 :
38 t16 := result
39 WRITE t16
40 t17 := #0
41 RETURN t17
```

三、实验总结

1. 学会了如何实现简单的中间代码生成。