哈尔滨工业大学

编译原理 2024 春

实验三

学院：	计算学部
姓名：	徐柯炎
学号：	2021110683
指导教师：	单丽莉

一、实验目的

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

二、实验环境

- 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。

三、实验内容

（一）实现功能

在实验一和实验二已经完成词法分析、语法分析和语义分析的基础上，本实验我们将完成中间代码的生成。由于实验一构造了语法树，实验二构建了符号表，我们可以在前两个实验的基础上进行中间代码的生成。中间代码生成的整体流程如下：

(1) 从已经构造完成的语法分析树的根节点开始分析，采用深度优先搜索的方法调用每个根节点类型的 translate 函数，以此来递归翻译实现的所有中间代码的生成；
(2) 将每个 translate 函数分析出的中间代码存放在 InterCodes 类型的链表中；
(3) 最后调用 intergenerate 函数将链表中的内容翻译为中间代码，并将中间代码输出到 out 文件中。

（二）数据结构

本次实验主要用到了两个自己定义的结构体，分别是操作数结构体和中间代码结构体。

首先是操作数结构体，具体结构如下图所示。这个结构体包括两个部分，kind 表示这个操作数的类型，u 为此操作数的值，一共有如下几类：

- VARIABLE 表示操作数是一个变量，其值存储在 u.name 字段中。
- CONSTANT 表示操作数是一个常量，其值存储在 u.value 字段中。
- ADDRESS 表示操作数是一个地址，其值存储在 u.name 字段中。

- LABEL_OP 表示操作数是一个标签，其值存储在 u.name 字段中。
- RELOP1 表示关系操作符，其值存储在 u.value 字段中。
- DER 表示间接寻址操作符，通常用于指针操作，表示操作数是通过另一个地址间接获取的值，其值存储在 u.name 字段中。

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

接着是中间代码结构体 InterCode，同样包含两个部分，kind 表示这个操作数的类型，u 为此操作数的值，总的来说分为以下几类：

- oneop 用于表示一元操作符，比如一元运算或单个操作数的指令。
- assign 用于表示赋值操作，包括左操作数、右操作数。
- binop 用于表示二元操作符，包括结果操作数、两个操作数。
- ifgoto 用于表示条件跳转指令，包括条件判断的操作数和跳转的目标地址等信息。
- dec 用于表示分配空间的操作，包括操作数和所需的空间大小。

```
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;
};
```

然后在基于 InterCode 结构体的基础上实现了 InterCode 双向链表 InterCodes，为的是存储遍历和访问 InterCode 节点方便。

（三）翻译过程

以 translate_Exp 中的 Exp1 PLUS Exp2 为例。

(1) 一开始通过 translate_Program 函数来递归翻译中间代码，在一级一级的调用中进入 translate_Exp 函数，于是比较 Exp 节点的第二个子节点是否为 PLUS，如果是，则进入该分支。

(2) 首先通过 new_temp 函数生成两个中间变量 t1 和 t2，然后递归调用 translate_Exp 函数获取计算 t1 和 t2 的中间代码，最后通过 gen_intercode1 函数来生成当前节点的中间代码，并将上述中间代码存放到 InterCodes 双向链表中。

(3) 这样我们就完成了这个节点上的中间代码生成。

(4) 最后当所有中间代码都存储到 InterCodes 链表中后，调用 intergenerate 函数将链表中的内容翻译为中间代码，并将中间代码输出到 out 文件中。这样就完成了中间代码的翻译过程。

```
if (strcmp(Exp->children[1]->name, "PLUS") == 0){
    Operand t1 = new_temp();
    Operand t2 = new_temp();
    translate_Exp(Exp->children[0], sym_table, t1);
    translate_Exp(Exp->children[2], sym_table, t2);
    char* op;
    if(strcmp(Exp->children[1]->name, "PLUS") == 0) op = "ADD1";
    gen_intercode1(op, place, t1, t2);
}
```

（四）编译过程和测试过程

本实验采用 makefile 进行编译，采用 test.sh 脚本来测试结果，如下图所示。

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

```
#!bin/bash

testpath=./test
parserpath=./parser
outpath=./out

testfiles=$(find $testpath -name "*.cmm")
# find $testpath -name "*.cmm"

for file in $testfiles
do
    echo ========================= Testing $file =========================
    $parserpath $file $outpath/$(basename $file .cmm).out
    cat $outpath/$(basename $file .cmm).out
    echo
done
```

四、实验结果

运行 test.sh 得到实验结果，并通过 IR 虚拟机来验证结果正确性，如下图所示。

```
1   FUNCTION fact :
2   PARAM n
3   temp1 := #1
4   IF n == temp1 GOTO label1
5   GOTO label2
6   LABEL label1 :
7   RETURN n
8   GOTO label3
9   LABEL label2 :
10  temp11 := #1
11  temp8 := n - temp11
12  ARG temp8
13  temp12 := CALL fact
14  temp5 := n * temp12
15  RETURN temp5
16  LABEL label3 :
17  FUNCTION main :
18  READ temp14
19  m := temp14
20  temp17 := #1
21  IF m > temp17 GOTO label4
22  GOTO label5
23  LABEL label4 :
24  ARG m
25  temp20 := CALL fact
26  result := temp20
27  GOTO label6
28  LABEL label5 :
29  temp22 := #1
30  result := temp22
31  LABEL label6 :
32  WRITE result
33  temp24 := #0
34  RETURN temp24
```

```
运行  单步  重置  清屏
>请输入temp14的值：4
24
程序执行结束，返回值为0。
总执行步数：45；总执行耗时：3,322ms
>
```

五、实验总结

1) 了解了中间代码生成的全过程，学习了如何生成简单的中间代码。
2) 增强了编程能力、分析问题的能力和动手能力；
3) 对中间代码的生成有了更深层次的认识。