# 哈尔滨工业大学 编译系统 2023 春

## 实验三 中间代码生成

学院 计算学部 | 姓名 王炳轩 | 学号 120L022115 | 指导教师 陈鄞

### 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

### 实验内容

### (一) 实验过程

在词法分析和语法分析中, 我们已经构造了语法树 和符号表。讲行中间代码生成,即需要在语法分析之后 根据语法分析的结果生成中间代码。本次实验对于实验 指导书上的样例完成了必做的所有要求。

#### 1. 数据结构

本次实验主要用到的数据结构如图所 示。

为了记录中间代码生成中每个操作数 的类型和数值,我们需要 Operand 操作数 结构体。其中 CONSTANT 和 RELOP1 需要 value 值,其他类型只需要 name 值。

为了保存生成结果和方便回填, 我们 需要 InterCode。同时,还有一个双向链 表 InterCodes 来保存前后节点的信息。

```
struct InterCodes
   InterCode code:
   struct InterCodes *prev, *next;
```

#### 2. 翻译过程

在经过语义分析后,我们保留 root 节点方便访问语法树, 然后,我们传入 root 节点、0 级符号表和输出文件指针来开 始我们的中间代码生成。

我们为每一个文法符号和产生式都设计了一个翻译函数, 如图所示。

下面,我们以 translate Exp 为例子,来简单讲解翻译过 程。

```
truct InterCode
                    LABEL IR.
                    FUNCTION.
                    ASSIGN,
                    ADD1,
                    SUB1,
                    IF_GOTO,
                    RETURN1,
                    ARG,
                    CALL,
                    PARAM.
                    READ,
                } kind;
                        Operand op:
                    } oneop;
                        Operand right, left:
                    } assign;
char *name;
                        Operand result, op1, op2;
                    } binop;
                        Operand x, relop, y, z;
                    } ifgoto;
                         Operand op;
                         int size:
                    } dec:
                      translate Exp(node *, tablenode *, Op
```

LABEL OP,

RELOP1,

} kind;

translate\_Args(node \*, tablenode \*, C translate\_Cond(node \*, Operand, Ope translate\_Stmt(node \*, tablenode \*). translate StmtList(node \*, tablenode translate\_VarDec(node \*, tablenode \* translate Dec(node \*, tablenode translate\_DecList(node \*, tablenode translate\_Def(node \*, tablenode translate Defl ist(node \*, tablenode \* translate\_CompSt(node \*, tablenode translate\_ParamDec(node \*, tablenode translate VarList(node \*, tablenode \* translate\_FunDec(node \*, tablenode translate ExtDef(node \*, tablenode \*) translate ExtDefList(node \*, tablenod translate\_Program(node \*, tablenode

在这个翻译函数中,我们先判断子结点数,分为4、3、2、1分别考虑。例如,当子结点为3的时候,可能为二元运算,然后我们对第二个儿子节点的名称进行判断,例如,第二个儿子节点的名字为"PLUS",则是加法运算,我们申请两个临时变量,然后将两侧的运算符再进行分析,并存入t1、t2中,最后,再生成一条ADD指令。

```
(Exp->numchild == 4)

if (strcmp(Exp->children[2]->name, "Args") == 0)
{
    // tablenode* function = lookup(sym_table, Exp->children[0]);
    Operand *arg_list = (Operand *)malloc(sizeof(Operand) * 10);
    memset(arg_list, 0, 10 * sizeof(Operand));
    if (strcmp(Exp->children[0]->IDvalue, "write") == 0)
    {
        int num = 0;
        translate_Args(Exp->children[2], sym_table, arg_list, num);
        gen_intercode3("WRITE", arg_list[0]);
    }
    else
    {
        int num = 0;
        translate_Args(Exp->children[2], sym_table, arg_list, num);
        int i = 0;
        for (int j = i; j >= 0; j--)
        {
              gen_intercode3("ARG", arg_list[j]);
        }
        Operand val = (Operand)malloc(sizeof(struct Operand_));
        val->kind = VARIABLE;
        val->u.name = Exp->children[0]->IDvalue;
        gen_intercode2("CALL", place, val);
    }
}
```

```
(Exp->numchild == 3)
    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);
        gen_intercode1("ADD1", place, t1, t2);
    if (strcmp(Exp->children[1]->name, "MINUS") == 0)...
    if (strcmp(Exp->children[1]->name, "STAR") == 0) ...
    if (strcmp(Exp->children[1]->name, "DIV1") == 0)...
    if (strcmp(Exp->children[1]->name, "AND") == 0)
    if (strcmp(Exp->children[1]->name, "OR") == 0)
    if (strcmp(Exp->children[1]->name, "RELOP") == 0).
    if (strcmp(Exp->children[1]->name, "ASSIGNOP") == 0)
    if (strcmp(Exp->children[1]->name, "Exp") == 0) ··
    if (strcmp(Exp->children[0]->name, "ID") == 0)
if (Exp->numchild == 2)
    if (strcmp(Exp->children[0]->name, "MINUS") == 0)...
    if (strcmp(Exp->children[0]->name, "NOT") == 0)
   (Exp->numchild == 1)
    if (strcmp(Exp->children[0]->name, "ID") == 0)
    if (strcmp(Exp->children[0]->name, "INT") == 0)...
   (Exp->numchild == 4)
    if (strcmp(Exp->children[2]->name, "Args") == 0)...
if (strcmp(Exp->children[2]->name, "Exp") == 0)...
```

再例如,如果有 4 个儿子且第三个儿子是 "Args"则对应于 Fun (Args)。即这是一个函数 调用语句。

我们先判断是否是 write 语句,如果是,则翻译第一个参数并生成 write 调用。

如果不是 write,则是用户函数的调用,我们先翻译参数列表,然后循环生成 arg 中间代码,最后生成一条 call 指令。

与之相比的是,因为 read()没有参数,因此是在子结点数量为 3 的分析块中。

调用 gen\_intercode 后,将会新建intercode 节点插入到 intercodes 链表中。

#### 3. 编译过程

本次实验采用 Makefile 进行编译。

## (二) 实验结果

下载安装虚拟机小程序。

sudo apt-get install python3-pyqt5;

sudo pip install sip PyQt5 PyQt5-tools -i https://pypi.tuna.tsinghua.edu.cn/simple --verbose:

https://cs.nju.edu.cn/changxu/2compiler/index.html

```
# 衛祥器设定和确详能项

CC = gcc

FLEX = flex

BISON = bison

CFLAGS = -std=c99

# 爾祥目标: src目录下的所有.c文件

CFILES = $(shell find ./ -name "*.c")

OBJS = $(CFILES:.c=.o)

LFILE = $(shell find ./ -name "*.y")

LFC = $(shell find ./ -name "*.y") | sed s/[^/]*\\.1/lex.yy.c/)

YFC = $(shell find ./ -name "*.y" | sed s/[^/]*\\.y/syntax.tab.c/)

LFO = $(LFC:.c=.o)

YFO = $(YFC:.c=.o)

parser: syntax $(filter-out $(LFO),$(OBJS))

$(CC) -o parser $(filter-out $(LFO),$(OBJS)) -1f1

syntax: lexical syntax-c

$(CC) -c $(YFC) -o $(YFO)

lexical: $(LFILE)

$(FLEX) -o $(LFC) $(LFILE)

$(BISON) -o $(YFC) -d -v $(YFILE)

-include $(patsubst %.o, %.d, $(OBJS))
```

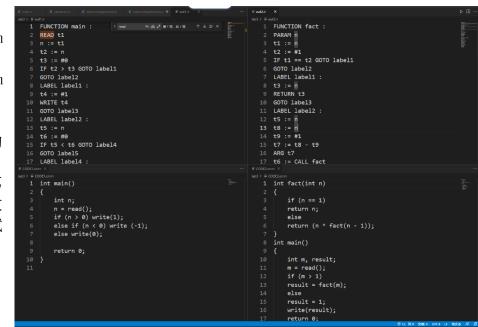
在终端运行

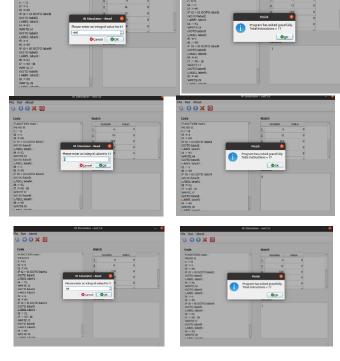
./parser CODE1.cmm out1.ir

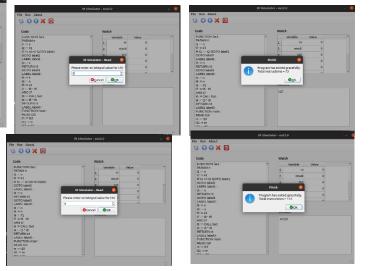
和 ./parser CODE2.cmm out2.ir

得到了正确的结果。

打开小程序后,运行我们的 ir 中间代码,输入测试数值,发现均得到了正确的结果。第一个测试用例是返回输入整数的符号(1,0,-1),第二个测试用例是求阶乘。







对测试用例 1,输入-99,返回-1,正确输入 0,返回 0,正确输入 99,返回 1,正确对测试用例 2,输入 5,返回 120,正确输入 8,返回 40320,正确。

## 三、实验总结

1. 学会了如何实现简单的中间代码生成。