编译原理实验报告

实验四 中间代码生成

学号：202011998088　　姓名：臧祝利　　日期：2022年12月10日

实验要求

在词法分析、语法分析和语义分析程序的基础上，将 `C--` 源代码翻译为中间代码。

要求将中间代码输出成**线性结构（三地址代码）**，使用虚拟机小程序来测试中间代码的运行结果。

基本要求

- 对于正确的测试样例，输出可以在虚拟机上运行的中间代码文件。✅
- 在程序可以生成正确的中间代码（“正确”是指该中间代码在虚拟机小程序上运行结果正确）的前提下，效率也是最终评分的关键因素。✅

附加要求

- 修改前面对C--源代码的假设2和3，使源代码中：
  - 可以出现结构体类型的变量（但不会有结构体变量之间直接赋值）。✅
  - 结构体类型的变量可以作为函数的参数（但函数不会返回结构体类型的值）。✅
- 修改前面对C--源代码的假设2和3，使源代码中：
  - 一维数组类型的变量可以作为函数参数（但函数不会返回一维数组类型的值）。✅
  - 可以出现高维数组类型的变量（但高维数组类型的变量不会作为函数的参数或返回类值）。✅

实验分工

小组内的实验分工如下：

> 臧祝利：代码框架+改造文件+计算优化
>
> 陈金利：附加要求-数组+条件优化
>
> 孙泽林：负数优化+附加要求-数组、附加-要求结构体+小程序
>
> 赵建锟：附加要求-结构体

实验环境

- Linux：Ubuntu 20.04 LTS
- Flex： V2.6.4
- GCC： V9.3.0
- Bison：V3.5.1
- python：V2.7.18
- IR Simulator：V1.0.2
- Qt：V4.8.7

由于Ubuntu 20.04除了对Qt库的支持且不自带python2，因此无法直接使用课本上的安装代码，如果需要使用小程序，需要额外进行环境配置

详情配置过程请查看文件 `IRSIM.md` 【经过外组某同学测试，实测可行】

实验设计

代码框架

使用 双向链表 进行中间代码生成，其本质就是借助语法树+符号表时生成链表结点，再借助一定的规则遍历生成中间代码；

Operand.h

进行了双向链表的定义；

包含两个数据结构，

运算对象Operand

```
struct Operand_ //运算对象
{
    enum
    {
        OP_VARIABLE, //变量 v_i
        OP_CONSTANT, //常量 # int
        OP_FUNCTION, //函数
        OP_TEMPVAR,  //临时变量 t_i
        OP_LABEL     //标签 label_i
    } kind;

    enum
    {
        OP_ADDRESS,
        OP_VAR
    } ifaddress; //地址类型还是变量类型

    char *varname;  //变量名
    int varno;      //变量编号
    int value;      // int值
    char *funcname; //函数名称
    int depth;
};
```

双向链表结点InterCode

```
struct InterCode
{
    enum
    {
        // one
        IN_FUNCTION,
        IN_PARAM,
        IN_RETURN,
        IN_LABEL,
        IN_GOTO,
        IN_WRITE,
        IN_READ,
        IN_ARG,
        // two
        IN_ASSIGN,
        IN_DEC,
        IN_CALL,
        // three
        IN_ADD,
        IN_SUB,
        IN_MUL,
        IN_DIV,
```

```
        // four
        IN_IFGOTO
    } kind;

    union
    {
        struct
        {
            Operand op0;
        } one;

        struct
        {
            Operand left;
            Operand right;
        } two;

        struct
        {
            Operand result;
            Operand op1;
            Operand op2;
        } three;

        struct
        { //实际上是op1,relop1,op2,op3
            Operand op1;
            Operand op2;
            Operand op3;
            char *relop;
        } four;

    } u;
};
```

Operand.c

主要的算法思想沿用 `lab3-yyfx` 的思路，整体分析使用深度优先搜索，对每一个树结点进行中间代码分析；

限于篇幅，这里只介绍最为关键的为两个函数，分别是 `new_op()` 函数和 `new_intercode()函数`

new_op()

得到一个运算对象，根据不同的类别进行赋值；

```
Operand new_op(int kind, int ifaddress, ...)
{
    va_list args;
    va_start(args, ifaddress);
    Operand op = (Operand)malloc(sizeof(struct Operand_));
    op->kind = kind;
    op->ifaddress = ifaddress;
    switch (kind)
    {
    case OP_VARIABLE:
    {
        op->varname = va_arg(args, char *); //获取变量名
        varcnt++;                           //变量个数++
        op->varno = varcnt;                 //赋值
        break;
```

```
    }
    case OP_FUNCTION:
    {
        op->funcname = va_arg(args, char *); //获取函数名
        op->varname = NULL;
        break;
    }
    case OP_CONSTANT:
    {
        op->value = va_arg(args, int); //获取int值
        op->varname = NULL;
        break;
    }
    case OP_LABEL:
    {
        labelcnt++; //标签个数++
        op->varno = labelcnt;
        op->varname = NULL;
        break;
    }
    case OP_TEMPVAR:
    {
        tmpcnt++;
        op->varno = tmpcnt;
        op->varname = NULL;
        break;
    }
    default:
        break;
    }
    va_end(args);
    return op;
}
```

new_intercode()

根据种类创建链表结点，并将对应的运算对象匹配到节点之中，然后插入双向链表中；

```
void new_intercode(int kind, ...)
{
    va_list args;
    va_start(args, kind);
    struct InterCodes *tmp = (struct InterCodes *)malloc(sizeof(struct InterCodes));
    tmp->code.kind = kind;
    tmp->next = NULL;
    switch (kind)
    {
    case IN_FUNCTION:
    case IN_PARAM:
    case IN_RETURN:
    case IN_LABEL:
    case IN_GOTO:
    case IN_WRITE:
    case IN_READ:
    case IN_ARG:
        tmp->code.u.one.op0 = va_arg(args, Operand);
        break;
    case IN_ASSIGN:
    case IN_DEC:
    case IN_CALL:
```

```
    {
        tmp->code.u.two.left = va_arg(args, Operand);
        tmp->code.u.two.right = va_arg(args, Operand);
        break;
    }
    case IN_ADD:
    case IN_SUB:
    case IN_MUL:
    case IN_DIV:
    {
        tmp->code.u.three.result = va_arg(args, Operand);
        tmp->code.u.three.op1 = va_arg(args, Operand);
        tmp->code.u.three.op2 = va_arg(args, Operand);
        break;
    }
    case IN_IFGOTO:
    {
        tmp->code.u.four.op1 = va_arg(args, Operand);
        tmp->code.u.four.relop = va_arg(args, char *);
        tmp->code.u.four.op2 = va_arg(args, Operand);
        tmp->code.u.four.op3 = va_arg(args, Operand);
        break;
    }
    default:
        break;
    }
    add_list(tmp);
    va_end(args);
}
```

改造文件

主要对 `lab3-yyfx` 中的以下地方进行了修改：

SymbolTable.h

```
struct SymbolNode
{
    enum
    {
        VARIABLE_NAME, //变量
        STRUCT_NAME,   //结构体
        FUNCTION_NAME  //函数
    } kind;
    struct SymbolNode *next;      //同一hash值的下一个结点
    struct SymbolNode *areanext; //
    struct FieldList_ field;      //存符号结点的类型和名称
    char *structsymbolname;       //结构体的名称
    int isdef;                    // 0→声明；1→定义
    int depth;                    //所在的深度
    //lab4
    int var_no;  //变量编号
    int ifaddress;  //是否是地址
    int offset;  //偏移量，仅对结构体有用
    char *belongtostructname;  //结构体名称
};
```

SymbolTable.c

新增函数`querySymbolNodeZJDM(char *name,int *success)`，用途是根据名称查询符号表中结点；

更改了`InsertStruct()`的部分代码，新增函数`CreateSymbolNodeZJDM()`,在插入结构体时使用，主要是进行初始化；

semantic.c

新增`write(),read()`函数，修改`DefStruct()`函数和`SpecifierAnalyse()`函数，用于修正struct可能引发的问题；

Write()

写入`write`函数，防止其遇到时出现语义错误

```
void Write(){
    char *functionname = (char*)malloc(sizeof(char)*32);
    strcpy(functionname,"write");

    Type functiontype = (Type)malloc(sizeof(struct Type_));

    FieldList params = (FieldList)malloc(sizeof(struct FieldList_));
    strcpy(params->name,"function write n");
    params->type = (Type)malloc(sizeof(struct Type_));
    params->type->kind = BASIC;
    params->type->u.basic = 0;

    Type returntype = (Type)malloc(sizeof(struct Type_));
    returntype->kind = BASIC;
    returntype->u.basic = 0;

    functiontype->kind = FUNCTION;
    functiontype->u.fuction.num=1;
    functiontype->u.fuction.ReturnParameter =returntype;
    functiontype->u.fuction.parameters=params;

    int isdef = 1;
    int depthtmp = 0;
    struct SymbolNode* insert = CreateSymbolNode(functiontype,functionname,isdef,depthtmp,FUNCTION_NAME);
    InsertSymbolNode(insert,GlobalScope);
}
```

Read()

```
void Read(){
    char *functionname = (char*)malloc(sizeof(char)*32);
    strcpy(functionname,"read");

    Type functiontype = (Type)malloc(sizeof(struct Type_));
    Type returntype = (Type)malloc(sizeof(struct Type_));
    returntype->kind = BASIC;
    returntype->u.basic = 0;

    functiontype->kind = FUNCTION;
    functiontype->u.fuction.num = 0;
    functiontype->u.fuction.ReturnParameter = returntype;
    functiontype->u.fuction.parameters = NULL;

    int isdef = 1;
    int depthtmp = 0;
    struct SymbolNode* insert = CreateSymbolNode(functiontype,functionname,isdef,depthtmp,FUNCTION_NAME);
```

```
    InsertSymbolNode(insert,GlobalScope);
}
```

计算优化

对数组来说，在计算偏移量的过程中，源代码如下：

```
            Node *tmp3 = GetChild(cur, 2);
            Operand ExpOp2 = ExpZJDM(tmp3);

            Operand tmpop1 = new_op(OP_TEMPVAR, OP_VAR);
            Operand constantop = new_op(OP_CONSTANT, OP_VAR, offset);

            new_intercode(IN_MUL, tmpop1, ExpOp2, constantop);
```

忽视了`ExpOp2`可能是常数的情况，对于`test4.cmm`生成代码时就会巨蠢无比的计算出如下结果：

```
FUNCTION add :
PARAM v0
t1 := #0 * #4
t2 := v0 + t1
t3 := #1 * #4
t4 := v0 + t3
t0 := *t2 + *t4
RETURN t0
```

即把每一步运算结果都显示出来了(这里特指常数之间的运算，尤其是`#0 * #4`属实是让人高血压的操作)；

因此优化的目的很简单——把立即数之间的运算优化掉，且对于取当前地址来说，不要出现`0`去乘某个数甚至`0`去加某个数的操作；

修改如下：

```
            Node *tmp3 = GetChild(cur, 2);
            Operand ExpOp2 = ExpZJDM(tmp3);

            Operand tmpop1 = new_op(OP_TEMPVAR, OP_VAR);

            Operand constantop = new_op(OP_CONSTANT, OP_VAR, offset);
            /*————————————————————————————————Changes————————————————————————————————*/
            int value = 0;
            if (ExpOp2->kind == OP_CONSTANT){
                value = ExpOp2->value * constantop->value;   //为常数时，把值记录下来
            }
            /*———————————————————————————————————————————————————————————————————————*/
            else new_intercode(IN_MUL, tmpop1, ExpOp2, constantop);   //正常进行乘法

            Operand tmpop2 = new_op(OP_TEMPVAR, OP_VAR);
            tmpop2->varname = ExpOp1->varname;
            tmpop2->depth = depth + 1;
            if (depth == 0 && ExpOp1->ifaddress == OP_VAR)
            {
                ExpOp1->ifaddress = OP_ADDRESS;
            }
            else
            {
                ExpOp1->ifaddress = OP_VAR;
            }
            /*————————————————————————————————Changes————————————————————————————————*/
            if (ExpOp2->kind == OP_CONSTANT){   //如果是立即数的话，和计算结果进行加法运算即可
                Operand valueop = new_op(OP_CONSTANT,OP_VAR,value);
```

```
            if (value == 0) new_intercode(IN_ASSIGN,tmpop2,ExpOp1);  //如果是0，不会写作+ #0，而是直接赋值
            else new_intercode(IN_ADD,tmpop2,ExpOp1,valueop);
        }
        /*——————————————————————————————————————————————————————————————————————————————————————*/
        else new_intercode(IN_ADD, tmpop2, ExpOp1, tmpop1);
```

修改后生成的 `out4.ir` 的 `add` 函数如下：

```
FUNCTION add :
PARAM v1
t3 := v1
t5 := v1 + #4
t1 := *t3 + *t5
RETURN t1
```

实验结果

使用以下命令进行编译：

```
sh cmd.sh
```

使用以下命令生成 `.ir` 文件

```
./parser [filename.cmm] [outname.ir]
```

test1.cmm

```
FUNCTION main :
READ t1
v1 := t1
IF v1 <= #0 GOTO label1
WRITE #1
t2 := #0
GOTO label2
LABEL label1 :
IF v1 >= #0 GOTO label3
WRITE #-1
t3 := #0
GOTO label4
LABEL label3 :
WRITE #0
t4 := #0
LABEL label4 :
LABEL label2 :
RETURN #0
```

test2.cmm

```
FUNCTION fact :
PARAM v1
IF v1 != #1 GOTO label1
RETURN v1
GOTO label2
LABEL label1 :
t3 := v1 - #1
ARG t3
t2 := CALL fact
t1 := v1 * t2
RETURN t1
```

```
LABEL label2 :
FUNCTION main :
READ t4
v2 := t4
IF v2 ≤ #1 GOTO label3
ARG v2
t5 := CALL fact
v3 := t5
GOTO label4
LABEL label3 :
v3 := #1
LABEL label4 :
WRITE v3
t6 := #0
RETURN #0
```

test3.cmm

```
FUNCTION add :
PARAM v1
t2 := v1
t3 := v1 + #4
t1 := *t2 + *t3
RETURN t1
FUNCTION main :
DEC v3 8
t4 := &v3
*t4 := #1
t5 := &v3 + #4
*t5 := #2
ARG &v3
t6 := CALL add
v2 := t6
WRITE v2
t7 := #0
RETURN #0
```

test4.cmm

```
FUNCTION add :
PARAM v1
t3 := v1
t5 := v1 + #4
t1 := *t3 + *t5
RETURN t1
FUNCTION main :
DEC v2 8
DEC v3 8
v4 := #0
v5 := #0
LABEL label1 :
IF v4 ≥ #2 GOTO label2
LABEL label3 :
IF v5 ≥ #2 GOTO label4
t6 := v5 * #4
t7 := &v2 + t6
t8 := v4 + v5
*t7 := t8
t9 := v5 + #1
```

```
v5 := t9
GOTO label3
LABEL label4 :
t11 := &v3
t12 := v4 * #4
t13 := t11 + t12
ARG &v2
t14 := CALL add
*t13 := t14
t16 := &v3
t17 := v4 * #4
t18 := t16 + t17
WRITE *t18
t19 := #0
t20 := v4 + #1
v4 := t20
v5 := #0
GOTO label1
LABEL label2 :
RETURN #0
```

实验反思

中间代码应该没有做到最优；

仅以 `test4.cmm` 而言，如果使用师兄语雀上的代码，使用小程序，可以得知$Total \quad instructions = 81$

在进行 计算优化 前，使用小程序，$Total \quad instructions = 94$

优化后，有$Total \quad instructions = 86$，只能说是在能力之内进行了一定程度的优化，对于最优还是没有达到（如果以给定为最优的话）

私以为所谓“最优”没有一定的标准，实验教材上只说 `Total instructions` 越少效率越好，但是没有 `baseline` 作为标准，还是有些难受的。