编译原理实验三报告

学号	1190202401
姓名	陈豪

一、实验环境

Ubuntu20.04TLS

gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0

flex 2.6.4

bison (GNU Bison) 3.5.1

二、程序实现的功能

本程序实现了对C--文法的中间代码生成，并完成了选做3.2（可以出现高维数组类型的变量并且一维数组类型的变量可以作为函数参数）

必做内容

使用的中间代码的表示为双链表线形IR，并且使用的是实验指导书中推荐的运算分量和中间代码数据结构。当然进行相应的类型扩充到19种中间代码。为了打印与管理方便另建立了一个结构体如下

```
struct InterCodesWrap_
{
    pInterCodes head; //第一条中间代码的头
    pInterCodes tail; //末尾的中间代码的尾
    int labelNum;     //符号数,用于给符号命名，符号用于跳转
    int tempVarNum;   //临时变量数，用于给临时变量命令
};
```

生成的过程是在实验二语义分析的基础上带着生成好的符号表进行中间代码生成，并且当完成所有中间代码的生成后再进行打印。

具体细节是首先在之前的语义分析中手动创建write和read函数，然后再中间代码生成里自顶向下进行翻译，这一点的流程同实验二差不多。

对于基本表达式，语句，条件表达式的翻译模式已经在实验指导书中给出，因此这里直接照着实验指导书翻译即可。

在完成本次实验的过程中我遇到了一个bug是在生成的中间代码由于我是直接将运算分量的指针给对应的中间代码去用，因此这导致未来释放相应中间代码的时候出现了重复释放或者找不到释放的内容的情况。因此我最后修改为每次生成中间代码的时候我就复制一遍运算分量，然后传递给中间代码依次解决。（当然对于前面新建过的运算分量在运行完一段代码后将在结尾释放）。

然后因为write函数和其他需要传递参数的函数调用不同，因此我的实现方式是在函数调用的时候，检查接下来需要传递参数的函数是否是write，如果是write就将前面的中间代码的ARG x类型的一条代码释放掉，然后在插入相应的write中间代码。细节如下：

```
// 因为write传递函数参数的方式不一样因此需要如下修改
pOperand temp = copyOperand(interCodesWrap->tail->code->u.oneOp.op);
pInterCodes prevCode = interCodesWrap->tail->prev;
freeInterCodes(interCodesWrap->tail);
interCodesWrap->tail = prevCode;
addInterCodesToWrap(interCodesWrap, newInterCodes(newInterCode(IR_WRITE, 1,
temp)));
```

在本次实验中由于时间紧张，我并没有做特别多的优化，一个小的优化是在函数返回的时候直接返回的是立即数。

选做内容

我完成的是多维数组和一维数组做参数的选做：

首先是对于数组的使用方面，每一次涉及exp的时候，会去检查这个exp是否是数组变量构成的，如果是的话就会进行一次读地址中内容或者写地址中的内容的操作具体细节如下

```
if (exp->child->brother && !strcmp(exp->child->brother->name, "LB"))
{
    addInterCodesToWrap(interCodesWrap, newInterCodes(newInterCode(IR_READ_ADDR,
2, temp, temp)));
    //或者是下面这样的，根据代码分析的阶段而判定。
    //addInterCodesToWrap(interCodesWrap,
newInterCodes(newInterCode(IR_WRITE_ADDR, 2, t1, t2)));
}
```

上面的代码在程序中多次出现。这让我实现多维数组和一维数组的参数方便了很多。因为我只需要确保在对数组的分析的时候，返回的会是正确的地址值即可。

对于多维数组计算偏移地址，我的方式就是做一个双重循环来计算偏移地址，然后将其和base相加即可。

对于一维数组做参数的情况，在语义分析中的fieldList结构中添加一个变量isParam来辨别它是否是参数，如果是参数那么这个一维数组在分析时作为运算分量将被打上一个标签。然后在计算一维数组地址时，根据这个标签判断是否需要取地址。细节如下：

```
//如果不是数组参数，那么不需要进行取地址操作
if (base->kind == OPERAND_VARIABLE)
{
    target = newTemp();
    addInterCodesToWrap(interCodesWrap, newInterCodes(newInterCode(IR_GET_ADDR,
2, target, base)));
}
else
{
    target = copyOperand(base);c
}
```

三、程序编译方式

本次程序同实验二一样使用了makefile配合python文件进行编译。编译命令如下

```
~/compilerLab/Lab3/code$ make
```

编译后，测试必做样例的命令如下

```
~/compilerLab/Lab3/code$ make havetodotest
```

编译后，测试选做样例的命令如下

```
~/compilerLab/Lab3/code$  make nothavetodotest
```

输出的文件组织如下

```
.
├── havetodo
│   ├── test1
│   ├── test1.ir
│   ├── test2
│   └── test2.ir
└── nothavetodo
    ├── test1
    ├── test2
    └── test2.ir
```

其中.ir是中间代码文件。nothavetodo中的test1没有做所以没有相应的中间代码文件。

如果需要运行小程序检查则需要额外的实验环境如下

Python	3.8.10
PyQt5	5.15.6
PyQt5-Qt5	5.15.2

```
~/compilerLab/Lab3/irsim$ python3 irsim.py
```

四、参考资料

- [1] 编译原理实践与指导教程
- [2] github