实验报告

1 程序功能

实验四实现了使用 LLVM 对于 main 函数和表达式的翻译 reuturn 语句中的表达式，通过 LLVM IR 工具将语义部分的代码进行中间代码翻译，最后能正确输出函数内容和表达式结果的值。

2 程序实现

①首先创建一个 Visitor 类，继承自 basevisitor。在 Visitor 中初始化 LLVM。
②其次创建 module，初始化 IRBuilder，定义一个 Int 基本类型。

③在遍历语法树时，在 funcDef 节点生成返回值类型，生成函数参数类型，接着生成函数并向 module 中添加该函数。然后在当前节点中加入一个基本块。

④接着遍历子节点，在 returnStmt 节点中判断表达式属于哪种类型：
如果是数字表达式则将数字转换为 LLVMValueRef，在用 LLVMBuildRet 将返回语句插入到 module 中；

```
else if (ctx.exp() instanceof SysYParser.NumberExpContext) {
    String numStr = ctx.exp().getText();
    numStr = getTenNumber(numStr);
    number += Integer.parseInt(numStr);
}

LLVMValueRef result = LLVMConstInt(i32Type, number, /* signExtend */ 0);
```

如果不是数字表达式，则访问更深一层的节点，例如在 plusExp 中，递归得到两侧的值，再进行加减法，最后将局部变量返回到 returnStmt 节点中

```
if (ctx.exp() instanceof SysYParser.PlusExpContext) {
            number += LLVMConstIntGetSExtValue(visitPlusExp((SysYParser.PlusExpContext) ctx.exp()))
;
}
```

⑤最后在 returnStmt 节点中将得到的局部变量插入到 ret 语句中，

```
LLVMBuildRet(builder, /*result:LLVMValueRef*/result);
```

最后在 main 函数中将 module 以文件的形式输出到指定文件夹下

其中，在 visitPlusExp 中得到两侧的值时，同样使用递归得到两侧的值

```
public int getNumber (SysYParser.ExpContext exp) {
        int number = 0;
        if (exp instanceof SysYParser.NumberExpContext) { // 数字表达式
            String numStr = exp.getText();
            numStr = getTenNumber(numStr);
```

```
            number += Integer.parseInt(numStr);
        }
        else if (exp instanceof SysYParser.PlusExpContext) { // 加法运算
            number += LLVMConstIntGetSExtValue(visitPlusExp((SysYParser.PlusExpContext) exp));
        }
        return number;
    }
```

3 精巧设计

在获得表达式值的时候，使用递归函数获得表达式的值，这样可以不用在整个表达式语句中判断运算符的优先级，同时也可以使代码结构更加简单。同时，由于取值是递归进行，并且表达式取值已经覆盖所有情况，所以在访问各种运算符节点的时候不需要 super.visitXXX，这样可以避免嵌套递归，只需要访问节点得到具体的值即可

4 印象深的地方

实验在理解 LLVM 何 IR 语句上花费较多时间，其次便是把 visit 函数的返回值类型从之前的 void 变成了 LLVMValueRef 类型，这样可以方便表达式值的传递。同时在表达式节点中不用 super.visitXXX，之前我在这里访问其他节点的时候会互相递归导致超出时间限制，删除访问子节点语句即可，并且访问表达式节点为自己主动调用，同时保证了能够完全访问所有节点。