中间代码生成说明

1 代码框架使用说明

1.1 关键函数与类的介绍

这里只大致描述一下整体代码结构，变量的具体含义见框架代码注释。

- LLVMIR 为描述整个 LLVMIR 的类，即中间代码生成的顶层模块。
- BasicBlock 是基本块。LLVMBlock 是对 BasicBlock* 的别名（即 BasicBlock 的指针）。成员变量包含一个 deque 来存储每个基本块中的指令，以及 block_id 表示该基本块的编号。
- BasicInstruction 是中间代码的指令基类。Instruction 是对 BasicInstruction* 的别名（即 BasicInstruction 的指针）。框架提供的派生的指令如下，未说明的指令含义请参考 llvm 文档以及框架代码注释llvm 参考手册:

  LoadInstruction

  StoreInstruction

  ArithmeticInstruction 基本算术指令 (理论上所有的二元运算指令都可以使用该类)，基类的 opcode 变量表示具体操作，可以为 add，sub，mod，fadd，fmul，xor 等。

  ICmpInstruction

  FCmpInstruction

  BrCondInstruction

  BrUncondInstruction

  RetInstruction

  AllocaInstruction

  PhiInstruction

  CallInstruction

  GetElementptrInstruction

  FptosiInstruction

  SitpfpInstruction

  ZextInstruction

  GetElementptrInstruction

  FptosiInstruction

SitpfpInstruction

ZextInstruction

FunctionDeclareInstruction 函数声明指令，主要用于声明 SysY 库函数。

FunctionDefineInstruction 函数定义指令，每个函数在 LLVMIR 是一个 key-value 对，其中 key 是指向函数定义指令的指针，value 是一个嵌套的 key2-value2 对，用于表示该函数对应的基本块。其中 key2 为基本块编号，value2 为指向基本块的指针。注意 FuncDefInstruction 是对 FunctionDefineInstruction* 的别名。

GlobalVarDefineInstruction 全局变量定义指令。

- BasicOperand 是操作数基类。Operand 是对 BasicOperand* 的别名（即 BasicOperand 的指针）。框架提供的派生的操作数如下：

  GlobalOperand 全局变量操作数，用一个字符串表示名称。

  IMMI32Operand 32 位整型操作数。

  IMMI64Operand 64 位整型操作数。

  IMMF32Operand 32 位浮点操作数。

  RegOperand 寄存器操作数，变量 reg_no 表示寄存器编号，输出前缀为 r，即假设 reg_no 为 9961，该操作数的输出为%r9961(加上前缀 r 是因为如果为纯数字，llvm 对数字的顺序有严格的要求，但是字符串并没有类似的要求)。

  LabelOperand 标签操作数，输出前缀为 L。

- 框架提供的 LLVMIR 类型有 I32，FLOAT32，VOID，I8，I1，I64，DOUBLE，PTR。

1.2 运行编译器

使用如下命令进行编译

```
make -j   # 如果你更新的不是 SysY_parser.y 和 SysY_lexer.l 文件，直接 make -j 即可
```

假设你成功编译后想要查看你的编译器对项目根目录下的 example.sy 的编译结果，使用如下命令：

```
./bin/SysYc -llvm -o example.out.ll example.sy
clang-15 example.out.ll  -c -o tmp.o
clang-15 -static tmp.o lib/libsysy_x86.a
# 请注意，如果你的电脑架构不为 x86，你需要自行制作链接库用于链接，测试脚本中也需要进行修改。
./a.out  # 运行程序
echo $?  # 查看程序 main 函数的返回值，你可以自行比较是否符合预期
```

1.3 自动测试脚本

```
python3 grade.py 3 0 # 测试基本要求
python3 grade.py 3 1 # 测试进阶要求
```

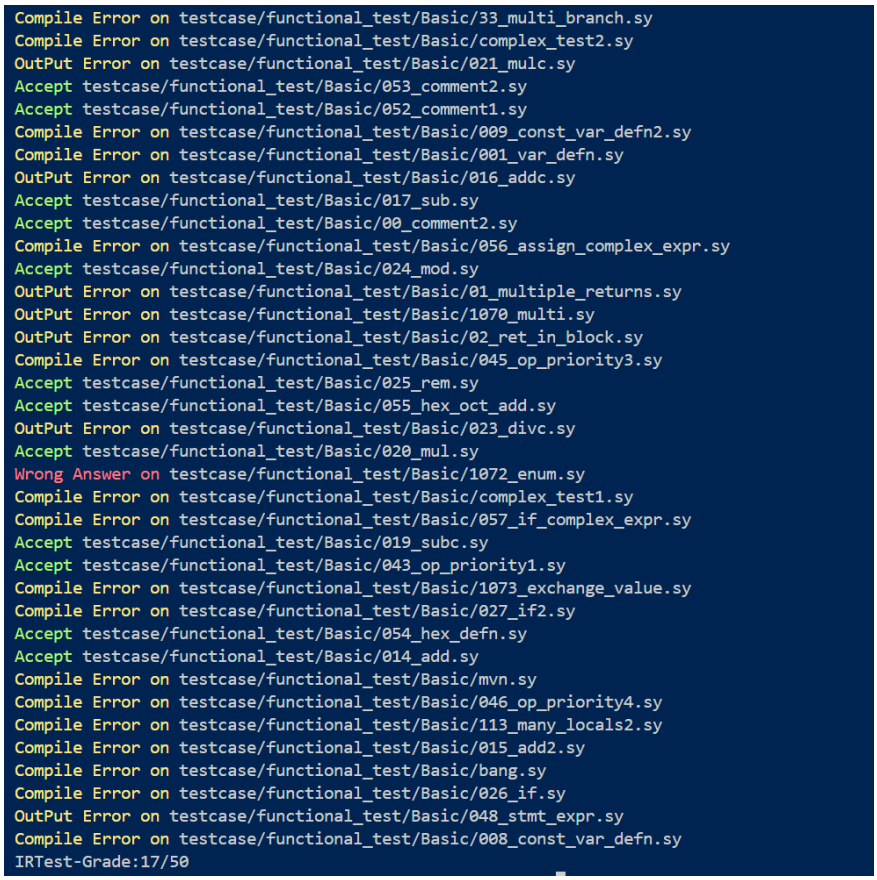

图 1.1: 测试结果示例

一个可能的测试结果如下图所示:

测试结果解析:

测试结果	原因
Compile Error	你写的编译器在运行过程中发生错误，例如运行超时，段错误等
Output Error	生成的文件在编译为.o 目标文件时出错，可能是你的输出不符合 LLVMIR 语法
Link Error	链接时发生错误，可能是你的输出中使用了未定义的函数或变量
Time Limit Exceed	可执行文件运行超时，可能是死循环或者性能过低等原因导致
RunTime Error	可执行文件运行时错误，可能是数组访问越界，栈溢出等原因导致
Wrong Answer	可执行文件输出错误
Accept	成功通过测试

2 如何翻译控制流语句

控制流的翻译会涉及到 bool 运算表达式 (例如 SysY 文法中的 Cond)，而对于 bool 运算表达式，我们可以在生成这个 bool 运算表达式之前，从 Cond 表达式处自上而下预先设定好每个 bool 运算表达式的真值出口和假值出口，之后生成跳转指令时，直接根据预先设置好的出口进行跳转即可。(SysY 的短路求值运算符只会出现在条件判断中，不会出现类似 int b = a || c 这种情况)

我们以下面的代码作为一个示例。

```
int a = 1;
int b = 10;
if (a < 5 && b > 6) {
    a = a + 1;
}
```

1. 在语法树上遍历到 IfStmt 时，设置 IfStmt 的 Cond 表达式的真值出口为 x1 号基本块，假值出口为 x2 号基本块，并新建这两个基本块。
2. 继续往下递归 (即调用 Cond->codeIR())，我们发现 Cond 表达式为 exp1 && exp2 的形式，此时根据短路求值的定义，我们可以知道 exp1 的假值出口为 x2，exp2 的真值出口为 x1，假值出口为 x2。但是我们此时无法知道 exp1 的真值出口。
3. 新建基本块 x3，并设置 exp1 的真值出口为 x3。
4. 调用函数 exp1->codeIR()，生成 exp1 的中间代码
5. 假设 exp1 的中间代码结果保存在 r1 寄存器，且我们此时应当在基本块 x4 处继续生成中间代码 (在哪个基本块生成应当在 exp1 的 codeIR 函数返回前设置好，这里我们假设 exp1 的 codeIR 函数设置成了 x4，后续 exp2 同理)。
6. 检查 r1 的类型是 bool 还是 int, 如果是 int, 在基本块 x4 处生成 int 到 bool 的隐式转换代码。(后续流程不会再提示隐式转换，同学们可以自行考虑什么时候需要进行转换)
7. 以转换后的 r1 为条件，在基本块 x4 生成一条条件跳转语句，如果为真，跳转到之前设置好的真值出口；如果为假，跳转到之前设置好的假值出口。
8. 设置我们现在应该在 x3 处继续生成中间代码，调用函数 exp2->codeIR()，生成 exp2 的中间代码
9. 假设 exp2 的中间代码结果保存在 r2 寄存器，且此时我们应当在基本块 x5 处继续生成中间代码。
10. 此时我们回到了 IfStmt 中，且能知道 cond 结果保存的寄存器编号。在基本块 x5 生成一条条件跳转语句，如果为真跳转到 x1，如果为假跳转到 x2。
11. 设置当前我们应当在基本块 x1 处继续生成中间代码，调用函数 stmt->codeIR() 生成 if 整体语句块的中间代码。
12. 假设生成完后我们应当在基本块 x6 处继续生成中间代码，在 x6 末尾生成一条无条件跳转语句，跳转到 x2。并设置后续我们应当在 x2 基本块处继续插入代码。此时即完成了上述示例的整个 if 语句块的生成。

如果你只看文字无法理解，可以对着下面的 LLVMIR 进行阅读。

```
define i32 @main() {
; 根据上文描述，0 号基本块为上文的 x4
    %1 = alloca i32
    %2 = alloca i32
    %3 = alloca i32
    store i32 0, ptr %1 ; useless code
    store i32 1, ptr %2 ; int a = 1
    store i32 10, ptr %3  ; int b = 10
    %4 = load i32, ptr %2
    ; 这里开始 Cond->codeIR()，我们在此处设置 Cond 的真值出口为 9，假值出口为 12
    ; 下面紧接着就是 exp1->codeIR()，我们设置 exp1 的真值出口为 6
    %5 = icmp slt i32 %4, 5 ;a < 5
    ; 根据上文描述，%5 寄存器为上文的 r1
    br i1 %5, label %6, label %12
6:  ; 根据上文描述，6 号基本块为上文的 x3，x5
    ; 这里是 exp2->codeIR() 生成的结果
    %7 = load i32, ptr %3
    %8 = icmp sgt i32 %7, 6 ; b > 6
    ; 根据上文描述，%8 寄存器为上文的 r2
    br i1 %8, label %9, label %12
9:  ; 根据上文描述，9 号基本块为上文的 x1，x6
    ; 这里是 stmt->codeIR() 生成的结果
    %10 = load i32, ptr %2
    %11 = add i32 %10, 1
    store i32 %11, ptr %2 ;a = a + 1
    br label %12
12: ; 根据上文描述，12 号基本块为上文的 x2
    ret i32 0
}

```