

类型检查 & 中间代码生成

杨科迪 费迪

朱璟钰 杨科迪 徐文斌 张书睿

华志远 李帅东

2020 年 11 月 - 2024 年 10 月

目录

1 实验描述	3
2 实验要求	3
3 实验流程	4
3.1 类型检查	4
3.2 中间代码生成	5
3.3 表达式的翻译	6
3.4 控制流的翻译	6
3.5 一个完整的例子	8
3.6 CPP-ARM 代码框架	12
3.6.1 关键函数与关键类的介绍	12
3.6.2 实验效果	13
3.6.3 自动化测试脚本	15
3.7 CPP-RISCV 代码框架	16
3.7.1 关键函数与类的介绍	16
3.7.2 类型检查实验效果	17
3.7.3 中间代码生成实验效果	18
3.7.4 自动测试脚本	18
3.8 线下检查提问示例	20
4 评分标准	21
4.1 类型检查（满分 2 分）	21
4.2 中间代码生成（满分 6 分）	21
4.2.1 基本要求（满分 4 分）	21
4.2.2 进阶要求（满分 2 分）	21

1 实验描述

欢迎大家来到了编译器构建的最关键模块，也是代码量最大的模块，在本次实验中，我们需要在之前构造好的语法树的基础上，进行类型检查，检测出代码中的一些错误并进行错误信息的打印，之后需要进行中间代码的生成，在中间代码的基础上，大家就可以进行一系列代码的优化工作。

2 实验要求

1. 在语法分析实验的基础上，遍历语法树，进行简单的类型检查，对于语法错误的情况简单打印出提示信息。
2. 完成中间代码生成工作，输出中间代码。
3. 无需撰写完整研究报告，但需要在雨课堂上提交本次实验的 `gitlab` 链接。
4. 上机课时，以小组为单位，向助教讲解程序。

3 实验流程

3.1 类型检查

类型检查是编译过程的重要一步，以确保操作对象与操作符相匹配。每一个表达式都有相关联的类型，如关系运算表达式的类型为布尔型，而计算表达式的类型一般为整型或浮点型等。类型检查的目的在于找出源代码中不符合类型表达式规定的代码，在最终的代码生成之前报错，使得程序员根据错误信息对源代码进行修正。

在本学期的实验中，由于处理所有语义错误情况较为复杂，且重复性工作较多，所以我们对实验进行了简化，仅需要针对以下几种情况进行处理（**完成基础要求即可获得类型检查实验满分**）：

语义错误检查的实验要求

基础要求：

- 检查 main 函数是否存在 (根据 SysY 定义，如果不存在 main 函数应当报错)；
- 检查未声明变量，及在同一作用域下重复声明的变量；
- 条件判断和运算表达式：int 和 bool 隐式类型转换（请思考 int a=5, return a+!a 应当如何处理）；
- 数值运算表达式：运算数类型是否正确 (如返回值为 void 的函数调用结果是否参与了其他表达式的计算)；
- 检查未声明函数，及函数形参是否与实参类型及数目匹配（需要考虑对 SysY 运行时库函数的调用是否合法）；
- 检查是否存在整型变量除以整型常量 0 的情况 (如对于表达式 $a/(5-4-1)$ ，编译器应当给出警告或者直接报错)；

进阶要求：

- 实现了数组的同学，还可以对数组维度进行相应的类型检查；
- 实现了浮点的同学，还可以对浮点进行相应的类型匹配、隐式类型转换等检查。

类型检查最简单的实现方式是在建立语法树的过程中进行相应的识别和处理，也可以在建树完成后，自底向上遍历语法树进行类型检查。类型检查过程中，父结点需要检查孩子结点的类型，并根据孩子结点类型确定自身类型。有一些表达式可以在语法制导翻译时就确定类型，比如整数就是整型，这些表达式通常是语法树的叶结点。而有些表达式则依赖其子表达式确定类型，这些表达式则是语法树中的内部结点。以表达式结点的类型检查为例（由于框架选择较多，下面的代码推荐当作伪代码进行阅读）：

```
1 Type *type1 = expr1->getSymPtr()->getType();
2 Type *type2 = expr2->getSymPtr()->getType();
3 if(type1 != type2)
4 {
```

```

5     fprintf(stderr, "type %s and %s mismatch in line xx",
6           type1->toStr().c_str(), type2->toStr().c_str());
7     exit(EXIT_FAILURE);
8 }
9
10 symbolEntry->setType(type1);

```

首先得到两个子表达式结点类型，判断两个类型是否相同，如果相同，设置结点类型为该类型，如果不相同输出错误信息。我们只是输出报错信息并退出，你还可以输出信息后插入类型转换结点，继续进行后续编译过程。

3.2 中间代码生成

中间代码生成是本次实验的重头戏，旨在前继词法分析、语法分析实验的基础上，将 SysY 源代码翻译为中间代码。中间代码生成主要包含对数据流和控制流两种类型语句的翻译，数据流包括表达式运算、变量声明与赋值等，控制流包括 if、while、break、continue 等语句。

中间代码是什么？

顾名思义，词法分析和语法分析是编译器的前端，那么中间代码是编译器的中端，相应地，目标代码就是编译器的后端。

中间代码有什么意义？

中间代码位于源代码和目标代码之间，它是一种抽象的、中间层次的编程语言表示，通常比源代码更接近底层的机器代码，但又比目标代码更抽象。自然的，你可能会感到困惑：为什么不能直接将源码转换为目标代码，而是要大费周章地引入中间代码呢？实际上，这主要有两点好处：

1. 通过将不同源语言翻译成同一中间代码，再基于中间代码生成不同架构的目标代码，有利于各模块的独立实现，并降低更换编译器的前端/后端的成本。
2. 在抽象出来的中间代码上进行优化更加简便。

一个具体的例子：

<pre> 1 // Origin Code 2 // Omit preceding ⇨ declarations int ⇨ a,b,c; 3 c = a+b; </pre>	<pre> 1 ; IR 2 ; ignore align here 3 %0 = load i32, ptr %a 4 %1 = load i32, ptr %b 5 %add = add i32 %0, %1 6 store i32 %add, ptr %c </pre>	<pre> 1 /* ARM */ 2 ldr r2, [fp, #-8] 3 ldr r3, [fp, #-12] 4 add r3, r2, r3 5 str r3, [fp, #-16] 6 /* Risc-V */ 7 lw a4,-20(s0) 8 lw a5,-24(s0) 9 add a5,a4,a5 10 sw a5,-28(s0) </pre>
--	--	--

在上述实例中，我们将源码中的加法翻译为中间代码中的 2 条 load，1 条 add 以及 1 条 store 指令。以 load 指令为例，在后续生成目标代码时，就可以根据目标平台的不同，选择性地对应到 ARM

中的 `ldr` 或 Risc-V 中的 `lw` 指令，而在优化中间代码时则不必考虑具体的指令。

中间代码生成的总体思路是对抽象语法树作一次遍历，遍历的过程中需要根据综合属性和继承属性来生成各结点的中间代码，在生成完根结点的中间代码后便得到了最终结果。

3.3 表达式的翻译

（由于框架选择较多，下面的代码推荐当作伪代码进行阅读）

```
1 BasicBlock *bb = builder->getInsertBB();
2 expr1->genCode();
3 expr2->genCode();
4 Operand *src1 = expr1->getOperand();
5 Operand *src2 = expr2->getOperand();
6 int opcode;
7 switch (op)
8 {
9 case ADD:
10     opcode = BinaryInstruction::ADD;
11     break;
12 case SUB:
13     opcode = BinaryInstruction::SUB;
14     break;
15 }
16 new BinaryInstruction(opcode, dst, src1, src2, bb);
```

`builder` 是 `IRBuilder` 类对象，用于传递继承属性，如新生成的指令要插入的基本块，辅助我们进行中间代码生成。在上面的例子中，我们首先通过 `builder` 得到后续生成的指令要插入的基本块 `bb`，然后生成子表达式的中间代码，通过 `getOperand` 函数得到子表达式的目的操作数，设置指令的操作码，最后生成相应的二元运算指令并插入到基本块 `bb` 中。

3.4 控制流的翻译

控制流的翻译是本次实验的难点，我们通过回填技术¹来完成控制流的翻译。我们为每个结点设置两个综合属性 `true_list` 和 `false_list`，它们是跳转目标未确定的基本块的列表，`true_list` 中的基本块为无条件跳转指令跳转到的目标基本块与条件跳转指令条件为真时跳转到的目标基本块，`false_list` 中的基本块为条件跳转指令条件为假时跳转到的目标基本块，这些目标基本块在翻译当前结点时尚不能确定，等到翻译其祖先结点能确定这些目标基本块时进行回填。我们以布尔表达式中的逻辑与和控制流语句中的 `if` 语句为例进行介绍，其他布尔表达式和控制流语句的翻译需要同学们自行实现。（由于框架选择较多，下面的代码推荐当作伪代码进行阅读）

1. 布尔表达式的翻译

¹参考龙书 p263-p268

```

1 BasicBlock *bb = builder->getInsertBB();
2 Function *func = bb->getParent();
3 BasicBlock *trueBB = new BasicBlock(func);
4 expr1->genCode();
5 backPatch(expr1->trueList(), trueBB);
6 builder->setInsertBB(trueBB);
7 expr2->genCode();
8 true_list = expr2->trueList();
9 false_list = merge(expr1->>falseList(), expr2->>falseList());

```

此处的逻辑与具有短路的特性，当第一个子表达式的值为假时，整个布尔表达式的值为假，第二个子表达式不会执行；当第一个子表达式的值为真时，根据第二个子表达式的值得到整个布尔表达式的值。

短路求值

虽然在一般情况下，短路求值的特性并不会影响程序的正确性，但在一些特殊情况下仍然会对程序的运行结果产生干扰。比如：

```

1 int a = 0;
2 int func() {
3     a = a + 1;
4     return a;
5 }
6

```

```

1 int main() {
2     if (1 == 1 || 1 == func()) {
3         return a;
4     }
5     return 0;
6 }

```

此处的`1==func()`是否被短路将影响全局变量 `a` 的数值。

在代码中，我们首先创建一个基本块 `trueBB`，它是第二个子表达式生成的指令需要插入的位置，然后生成第一个子表达式的中间代码，在第一个子表达式生成中间代码的过程中，生成的跳转指令的目标基本块尚不能确定，因此会将其插入到子表达式结点的 `true_list` 和 `false_list` 中。在翻译当前布尔表达式时，我们已经能确定 `true_list` 中跳转指令的目的基本块为 `trueBB`，因此进行回填。我们再设置第二个子表达式的插入点为 `trueBB`，然后生成其中间代码。最后，因为当前仍不能确定子表达式二的 `true_list` 的目的基本块，因此我们将其插入到当前结点的 `true_list` 中，我们也不能知道两个子表达式的 `false_list` 的跳转基本块，便只能将其插入到当前结点的 `false_list` 中，让父结点回填当前结点的 `true_list` 和 `false_list`。

2. 控制流语句的翻译

```

1 Function *func;
2 BasicBlock *then_bb, *end_bb;
3
4 func = builder->getInsertBB()->getParent();

```

```

5  then_bb = new BasicBlock(func);
6  end_bb = new BasicBlock(func);
7
8  cond->genCode();
9  backPatch(cond->>trueList(), then_bb);
10 backPatch(cond->>falseList(), end_bb);
11
12 builder->setInsertBB(then_bb);
13 thenStmt->genCode();
14 then_bb = builder->getInsertBB();
15 new UncondBrInstruction(end_bb, then_bb);
16
17 builder->setInsertBB(end_bb);

```

我们创建出 then_bb 和 end_bb 两个基本块，then_bb 是 thenStmt 结点生成的指令的插入位置，end_bb 为 if 语句后续的结点生成的中间代码的插入位置。第 8 行生成 cond 结点的中间代码，cond 为真时将跳转到基本块 then_bb，cond 为假时将跳转到基本块 end_bb，我们进行回填。第 12 行设置插入点为基本块 then_bb，然后生成 thenStmt 结点的中间代码。因为生成 thenStmt 结点中间代码的过程中可能改变指令的插入点，因此第 14 行更新插入点，然后生成无条件跳转指令跳转到 end_bb。最后设置后续指令的插入点为 end_bb。

实际上，不使用回填技术也是可以实现控制流翻译的，我们可以在生成这个 bool 运算表达式之前，从 Cond(SysY 文法定义中的非终结符，SysY 的短路求值运算符只会出现于条件判断中，不会出现类似 `int b = a || c` 这种情况) 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。不过该思路本质上和回填技术并没有太大的差别。

3.5 一个完整的例子

图3.1是将以下 SysY 语言翻译成中间代码的过程，同学们可以仔细体会指令回填的过程，在第③步中，条件跳转指令的目标基本块 b1 和 b2 不能确定，我们将其放入该结点的 true_list 和 false_list 中，在第④步中，条件为真跳转到的目标基本块 b1 已经能确定了，我们将其回填为 true_bb，在第⑤步中，b3 和 b4 不能确定，我们将其放入回填列表，在第⑥步中，b2、b3 和 b4 仍无法确定，我们将其放入当前结点的回填列表，等到第⑦步中，我们已经能确定 b3 基本块为 then_bb，b2、b4 基本块为 end_bb，因此进行回填，在第⑧步翻译完根结点后，我们已经得到了一个完整的流图，流图中基本块的前驱和后继关系已经能确定，基本块中的中间代码也已经得到了。

```

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

```

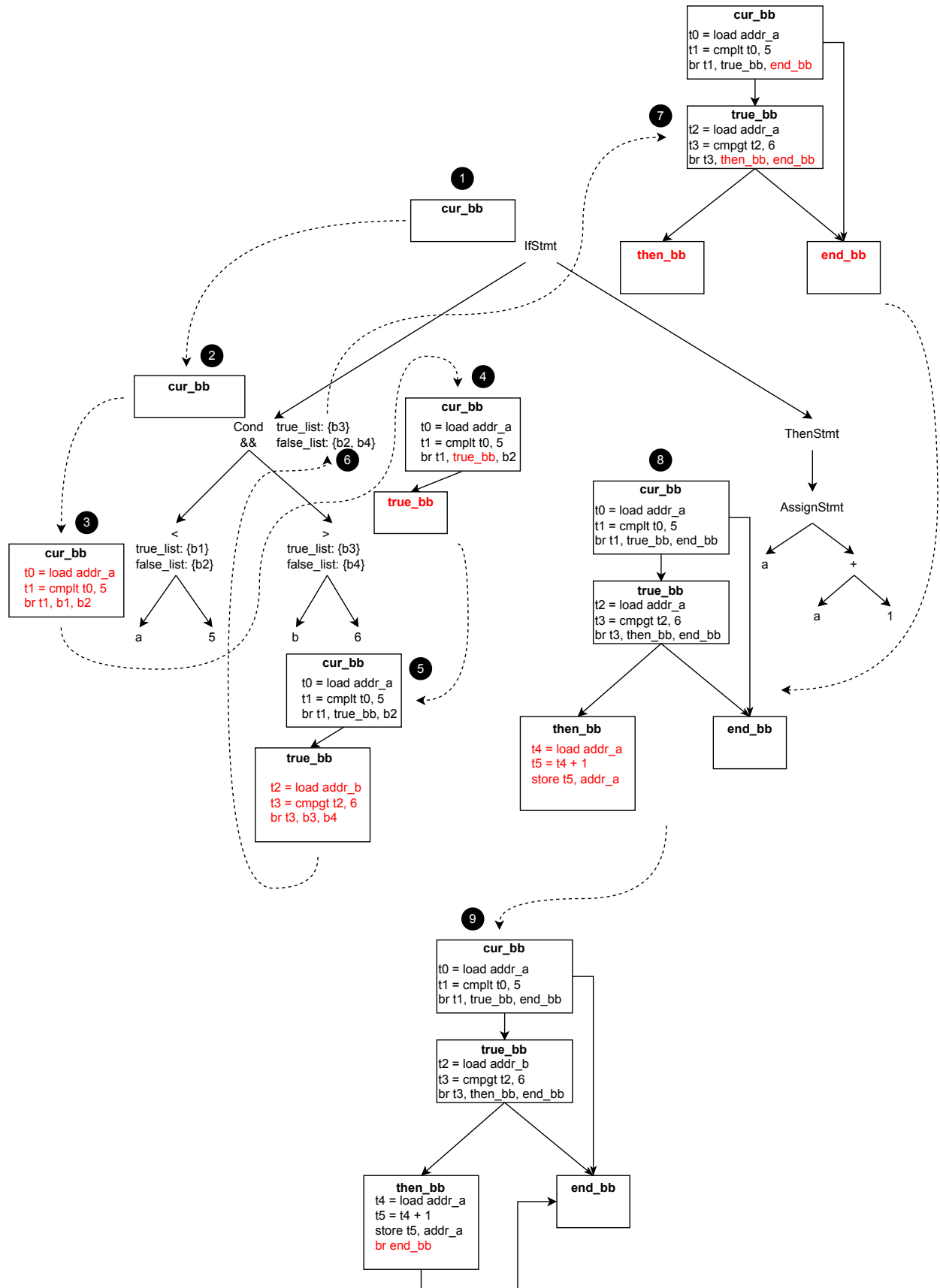



图 3.1: 中间代码生成的一个例子

如果你不使用回填技术，下面是一个可供参考的简易算法流程（仍是上面的代码例子，**注意为了实现下面的算法，你可能需要自行在框架中的语法树节点中添加成员变量**）

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。

```

1  define i32 @main() {
2  ; 根据上文描述，0 号基本块为上文的 x4
3      %1 = alloca i32
4      %2 = alloca i32
5      %3 = alloca i32
6      store i32 0, ptr %1 ; useless code
7      store i32 1, ptr %2 ; int a = 1
8      store i32 10, ptr %3 ; int b = 10

```

```
9      %4 = load i32, ptr %2
10      ; 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
11      ; 下面紧接着就是 exp1->codeIR(), 我们设置 exp1 的真值出口为 6
12      %5 = icmp slt i32 %4, 5 ; a < 5
13      ; 根据上文描述, %5 寄存器为上文的 r1
14      br i1 %5, label %6, label %12
15 6:    ; 根据上文描述, 6 号基本块为上文的 x3, x5
16      ; 这里是 exp2->codeIR() 生成的结果
17      %7 = load i32, ptr %3
18      %8 = icmp sgt i32 %7, 6 ; b > 6
19      ; 根据上文描述, %8 寄存器为上文的 r2
20      br i1 %8, label %9, label %12
21 9:    ; 根据上文描述, 9 号基本块为上文的 x1, x6
22      ; 这里是 stmt->codeIR() 生成的结果
23      %10 = load i32, ptr %2
24      %11 = add i32 %10, 1
25      store i32 %11, ptr %2 ; a = a + 1
26      br label %12
27 12:   ; 根据上文描述, 12 号基本块为上文的 x2
28      ret i32 0
29  }
```

3.6 CPP-ARM 代码框架

3.6.1 关键函数与关键类的介绍

本次实验框架代码的目录结构如下：

```
./
├── include
│   ├── common.h
│   ├── Ast.h
│   ├── SymbolTable.h
│   ├── Type.h
│   ├── IRBuilder.h ..... 中间代码构造辅助类
│   ├── Unit.h ..... 编译单元
│   ├── Function.h ..... 函数
│   ├── BasicBlock.h ..... 基本块
│   ├── Instruction.h ..... 指令
│   ├── Operand.h ..... 指令操作数
│   ├── IRBlockMerge.h ..... 基本块合并优化
│   └── IRComSubExprElim.h ..... 公共子表达式消除优化
├── src
│   ├── Ast.cpp
│   ├── BasicBlock.cpp
│   ├── Function.cpp
│   ├── Instruction.cpp
│   ├── lexer.l
│   ├── main.cpp
│   ├── Operand.cpp
│   ├── parser.y
│   ├── SymbolTable.cpp
│   ├── Type.cpp
│   ├── Unit.cpp
│   ├── IRBlockMerge.cpp
│   └── IRComSubExprElim.cpp
├── sysruntime library
├── test
├── .gitignore
├── example.sy
└── Makefile
```

- Unit 为编译单元，是我们中间代码的顶层模块，包含我们中间代码生成时创建的函数。
- Function 是函数模块。函数由多个基本块构成，每个函数都有一个 entry 基本块，它是函数的入口结点。²

²框架中并未设置 exit 基本块，但为了方便的实现优化，建议设置 exit 基本块，作为函数的出口结点。

- **BasicBlock** 为基本块。基本块包含有中间代码的指令列表，因为我们可能频繁地向基本块中插入和删除指令，还有可能反向遍历指令列表，因此基本块中的指令列表适合用双向循环链表来表示。基本块中的指令是顺序执行的，也就是说，跳转指令只能跳转到基本块中的第一条指令，基本块中的最后一条指令只能是跳转指令或者函数返回指令，基本块中间不含有控制流指令。基本块之间形成了流图，对于基本块 A 来说，如果基本块 A 跳转到基本块 B，我们说基本块 A 是基本块 B 的前驱，基本块 B 是基本块 A 的后继。如果基本块 A 的最后一条指令是条件跳转指令，那么基本块 A 含有两个后继结点，分别是条件为真和为假时跳转到的基本块；如果基本块 A 的最后一条指令是无条件跳转指令，那么基本块 A 含有一个后继结点；最后一条指令是函数返回指令，则基本块 A 不含有后继结点。我们使用邻接链表来表示流图，每个基本块都有前驱基本块列表 `pred` 和后继基本块列表 `succ`。
- **Instruction** 是我们中间代码的指令基类。指令包含有操作码 `opcode` 和操作数 `operands`。指令列表由双向循环链表来表示，因此每条指令都有指向前一条及后一条指令的指针 `prev` 和 `next`。已经派生出的指令包含：

LoadInstruction	从内存地址中加载值到中间变量中。
StoreInstruction	将值存储到内存地址中。
BinaryInstruction	二元运算指令, 包含一个目的操作数和两个源操作数。
CmpInstruction	关系运算指令。
CondBrInstruction	条件跳转指令，分支为真和为假时分别跳转到基本块 <code>true_branch</code> 和 <code>false_branch</code> 。
UncondBrInstruction	无条件跳转指令，直接跳转到基本块 <code>branch</code> 。
RetInstruction	函数返回指令。
AllocaInstruction	在内存中分配空间。

- **Operand** 为指令的操作数, **Operand** 类中包含一条定义-引用链, `def` 为定义该操作数的指令, `uses` 为使用该操作数的指令。
- **Type** 为函数或操作数的类型，框架中已经实现的类型包含：

IntType	整数类型，我们规定 <code>int(i32)</code> 类型的 <code>size</code> 为 32, <code>bool(i1)</code> 类型的 <code>size</code> 为 1
VoidType	仅用于函数的返回类型
FunctionType	函数类型，包含函数的返回值类型和形参类型
PointerType	指针类型， <code>valueType</code> 为所指向的值的类型

更多的类型、指令大家可以照猫画虎地进行构造，完成实验要求。

3.6.2 实验效果

以如下 SysY 语言为例：

```

1 int main()
2 {

```

```

3      int a;
4      int b;
5      int min;
6      a = 1 + 2 + 3;
7      b = 2 + 3 + 4;
8      if (a < b)
9          min = a;
10     else
11         min = b;
12     return min;
13 }

```

执行 make run 命令会生成对应的中间代码:

```

1  define i32 @main() {
2  B17:
3      %t20 = alloca i32, align 4
4      %t19 = alloca i32, align 4
5      %t18 = alloca i32, align 4
6      %t4 = add i32 1, 2
7      %t5 = add i32 %t4, 3
8      store i32 %t5, i32* %t18, align 4
9      %t7 = add i32 2, 3
10     %t8 = add i32 %t7, 4
11     store i32 %t8, i32* %t19, align 4
12     %t9 = load i32, i32* %t18, align 4
13     %t10 = load i32, i32* %t19, align 4
14     %t11 = icmp slt i32 %t9, %t10
15     br i1 %t11, label %B21, label %B24
16 B21:                                     ; preds = %B17
17     %t13 = load i32, i32* %t18, align 4
18     store i32 %t13, i32* %t20, align 4
19     br label %B23
20 B24:                                     ; preds = %B17
21     br label %B22
22 B23:                                     ; preds = %B21, %B22
23     %t16 = load i32, i32* %t20, align 4
24     ret i32 %t16
25 B22:                                     ; preds = %B24
26     %t15 = load i32, i32* %t19, align 4
27     store i32 %t15, i32* %t20, align 4
28     br label %B23

```

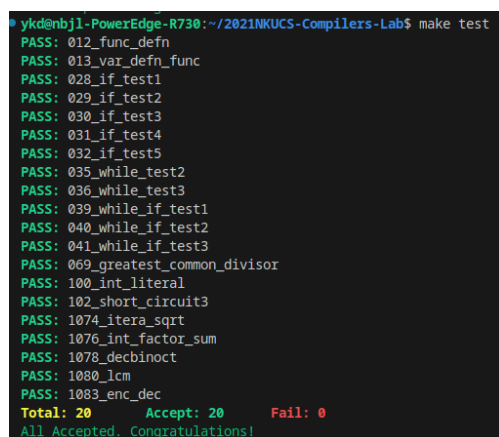
29 }

我们可以使用 `llvm` 编译器将该中间代码编译成可执行文件，验证我们实现的正确性：

```
1 clang example.ll -o example.out
2 ./example.out
3 echo $?
4 6
```

3.6.3 自动化测试脚本

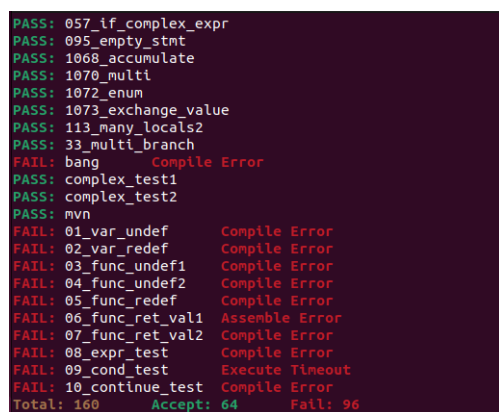
你也可以通过运行 `make test`，进行批量测试，图3.4为测试结果。



```
ykdenbjl-PowerEdge-R730:~/2021NKUCS-Compilers-Lab$ make test
PASS: 012_func_defn
PASS: 013_var_defn_func
PASS: 028_if_test1
PASS: 029_if_test2
PASS: 030_if_test3
PASS: 031_if_test4
PASS: 032_if_test5
PASS: 035_while_test2
PASS: 036_while_test3
PASS: 039_while_if_test1
PASS: 040_while_if_test2
PASS: 041_while_if_test3
PASS: 069_greatest_common_divisor
PASS: 100_int_literal
PASS: 102_short_circuit3
PASS: 1074_itera_sqrt
PASS: 1076_int_factor_sum
PASS: 1078_decbinoct
PASS: 1080_lcm
PASS: 1083_enc_dec
Total: 20      Accept: 20      Fail: 0
All Accepted. Congratulations!
```

图 3.2: 测试结果

如图3.3，会直接显示报错信息。我们也可以进入 `test` 测试样例文件夹当中，查找对应的测试样例报错，如 `01_var_undef`，可以找到名为 `01_var_undef.log` 的文件，查看具体的报错信息。



```
PASS: 057_if_complex_expr
PASS: 095_empty_stmt
PASS: 1068_accumulate
PASS: 1070_multi
PASS: 1072_enum
PASS: 1073_exchange_value
PASS: 113_many_locals2
PASS: 33_multi_branch
FAIL: bang      Compile Error
PASS: complex_test1
PASS: complex_test2
PASS: mvn
FAIL: 01_var_undef      Compile Error
FAIL: 02_var_redef      Compile Error
FAIL: 03_func_undef1      Compile Error
FAIL: 04_func_undef2      Compile Error
FAIL: 05_func_redef      Compile Error
FAIL: 06_func_ret_val1      Assemble Error
FAIL: 07_func_ret_val2      Compile Error
FAIL: 08_expr_test      Compile Error
FAIL: 09_cond_test      Execute Timeout
FAIL: 10_continue_test      Compile Error
Total: 160      Accept: 64      Fail: 96
```

图 3.3: 测试报错

3.7 CPP-RISCV 代码框架

和之前实验一样，你需要先阅读 README.md，然后阅读相关文件的注释

3.7.1 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，每个函数在框架的 LLVMIR 类中是一个 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。(你需要关注一下浮点数是怎么输出的, 线下检查可能会提问相关内容)
RegOperand	寄存器操作数, 变量 <code>reg_no</code> 表示寄存器编号, 输出前缀为 <code>r</code> , 即假设 <code>reg_no</code> 为 9961, 该操作数的输出为 <code>%r9961</code> (加上前缀 <code>r</code> 是因为如果为纯数字, <code>llvm</code> 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 <code>L</code> 。

- 框架提供的类型有 `I32`, `FLOAT32`, `VOID`, `I8`, `I1`, `I64`, `DOUBLE`, `PTR`。对于 `PTR` 类型, 请参考预备实验 1 的实验指导书中关于 `opaque` 指针的介绍。

3.7.2 类型检查实验效果

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 -semant -o example.out.ll example.sy
```

一个测试的 `SysY` 程序例子如下:

```
int b = 7;
int a = 5;
int a = 13;
int c = 55;
```

编译器应该输出类似下面的信息:

```
redefinition of 'a' in line 3
main function does not exist
```

3.7.3 中间代码生成实验效果

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
```

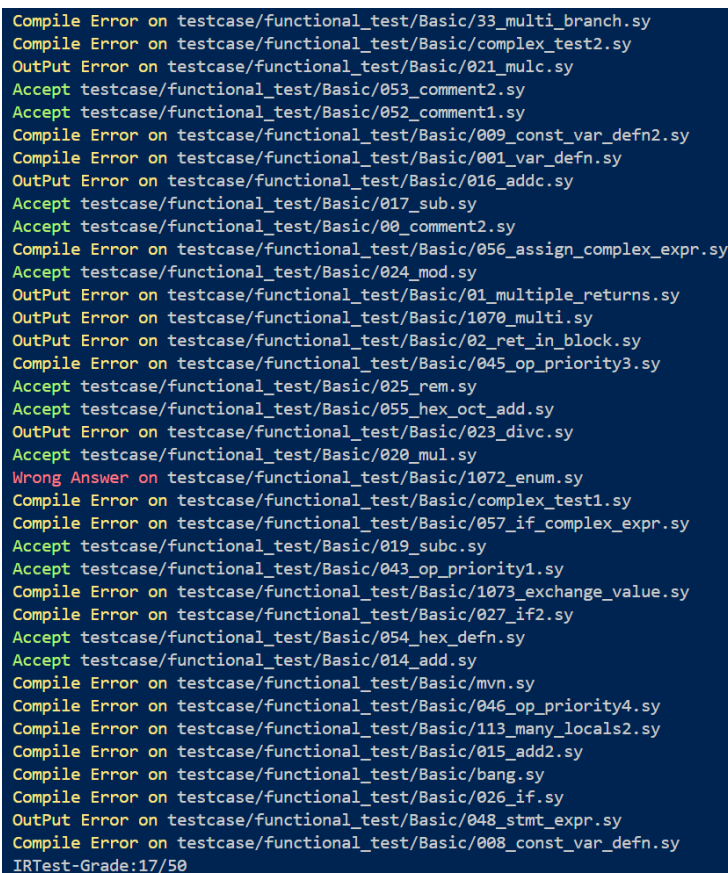
假设你成功编译后想要查看你的编译器对项目根目录下的 example.sy 的编译结果, 使用如下命令:

```
./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
```

3.7.4 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
```

一个可能的测试结果如下图所示:



```
Compile Error on testcase/functional_test/Basic/33_multi_branch.sy
Compile Error on testcase/functional_test/Basic/complex_test2.sy
OutPut Error on testcase/functional_test/Basic/021_mulc.sy
Accept testcase/functional_test/Basic/053_comment2.sy
Accept testcase/functional_test/Basic/052_comment1.sy
Compile Error on testcase/functional_test/Basic/009_const_var_defn2.sy
Compile Error on testcase/functional_test/Basic/001_var_defn.sy
OutPut Error on testcase/functional_test/Basic/016_addc.sy
Accept testcase/functional_test/Basic/017_sub.sy
Accept testcase/functional_test/Basic/00_comment2.sy
Compile Error on testcase/functional_test/Basic/056_assign_complex_expr.sy
Accept testcase/functional_test/Basic/024_mod.sy
OutPut Error on testcase/functional_test/Basic/01_multiple_returns.sy
OutPut Error on testcase/functional_test/Basic/1070_multi.sy
OutPut Error on testcase/functional_test/Basic/02_ret_in_block.sy
Compile Error on testcase/functional_test/Basic/045_op_priority3.sy
Accept testcase/functional_test/Basic/025_rem.sy
Accept testcase/functional_test/Basic/055_hex_oct_add.sy
OutPut Error on testcase/functional_test/Basic/023_divc.sy
Accept testcase/functional_test/Basic/020_mul.sy
Wrong Answer on testcase/functional_test/Basic/1072_enum.sy
Compile Error on testcase/functional_test/Basic/complex_test1.sy
Compile Error on testcase/functional_test/Basic/057_if_complex_expr.sy
Accept testcase/functional_test/Basic/019_subc.sy
Accept testcase/functional_test/Basic/043_op_priority1.sy
Compile Error on testcase/functional_test/Basic/1073_exchange_value.sy
Compile Error on testcase/functional_test/Basic/027_if2.sy
Accept testcase/functional_test/Basic/054_hex_defn.sy
Accept testcase/functional_test/Basic/014_add.sy
Compile Error on testcase/functional_test/Basic/mvn.sy
Compile Error on testcase/functional_test/Basic/046_op_priority4.sy
Compile Error on testcase/functional_test/Basic/113_many_locals2.sy
Compile Error on testcase/functional_test/Basic/015_add2.sy
Compile Error on testcase/functional_test/Basic/bang.sy
Compile Error on testcase/functional_test/Basic/026_if.sy
OutPut Error on testcase/functional_test/Basic/048_stmt_expr.sy
Compile Error on testcase/functional_test/Basic/008_const_var_defn.sy
IRTest-Grade:17/50
```

图 3.4: 测试结果示例

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

3.8 线下检查提问示例

如果下面的提问示例中的语法你没有实现，那么不会问你相关的问题。需要结合你的代码实现进行回答。

1. 请以 `int a[5][4][3] = {{{2,3},6,7},7,8,11};` 为例来说明你是如何处理全局变量数组的初始化语法的。
2. `int a[5][4][3] = {{{2,3},6,7,5,4,3,2,11,2,4,5},7,8,11};` 该初始化是否合法，请说明理由。
3. 请描述框架中的符号表是如何设计的，使用到了什么数据结构。如果你没使用框架中的符号表，你是如何设计一个可以支持作用域的符号表的。
4. 请说明你是如何处理 `int/float/bool` 类型的隐式转换的，简单说明 `int a = 5;int b = a + 3.5 + !a` 的语法树结构，并说明各节点上的类型。
5. 请说明对于局部变量 `int a[5][4][3] = {{{2,3},1}};` 应当如何生成 `llvm-ir`。
6. 请说明编译器是如何处理除数为 0 的情况的。
7. 说明你是如何在语义分析步骤中检查 `SysY` 库函数调用是否合法的。
8. 说明你是如何实现短路求值的控制流翻译的。

4 评分标准

4.1 类型检查 (满分 2 分)

你需要对前文中提到的情况进行相应的处理，打印出对应的提示信息。其中对数组部分的类型检查为选做项，我们会为大家提供包含类型错误的代码，来供大家测试，当然我们在作业检查过程中会随机改动错误源代码，以验证类型检查的正确性。

4.2 中间代码生成 (满分 6 分)

4.2.1 基本要求 (满分 4 分)

基本要求得到满分分数需正确实现如下 SysY 特性：

1. 数据类型：int
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值（=）、表达式语句、语句块、if、while、return
4. 表达式：算术运算（+、-、*、/、%，其中 +、- 都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（&&（与）、||（或）、！（非））
5. 注释
6. main 函数和输入输出函数调用（实现链接 SysY 运行时库，参见文档《SysY 运行时库》）

具体评分标准：基本要求一共有 50 个测试用例，其中 50 个测试用例平分 3 分，即每个测试用例的分值为 0.06 分，对于该项分数我们会以 0.1 为精度向下取整（即如果你的得分为 2.94 分，我们会将你的分数记为 2.9 分）。**如果你通过了所有的基本要求测试用例，额外再获得 1 分。**如果你线下检查时未能成功回答出助教的问题，我们会根据你的回答情况在测试用例得分的基础上扣除一定分数。

4.2.2 进阶要求 (满分 2 分)

进阶要求得到满分分数需正确实现如下 SysY 特性：

1. 函数声明、函数调用
2. 变量、常量作用域，在函数中、语句块（嵌套）中包含变量、常量声明的处理，break、continue 语句
3. 数组（一维、二维、…）的声明和数组元素访问
4. 浮点数常量识别、变量声明、存储、运算

具体评分标准：进阶要求一共有 100 个测试用例，其中 100 个测试用例平分 1 分，即每个测试用例的分值为 0.01 分，对于该项分数我们会以 0.1 为精度向下取整（即如果你的得分为 0.99 分，我们会将你的分数记为 0.9 分）。**如果你通过了 90% 以上的进阶要求测试用例，额外获得 0.5 分，如果你通过了所有的进阶要求测试用例，额外再获得 0.5 分。**如果你线下检查时未能成功回答出助教的问题，我们会根据你的回答情况在测试用例得分的基础上扣除一定分数。

你需要注意的是目标代码生成的语法进阶要求同样占 2 分，并且实现目标代码生成的进阶要求的前置条件是完成中间代码生成的进阶要求，请注意提前计划好你想要实现的进阶要求。**同时提醒大家，进阶要求难度较大，在两人合作的情况下，需要花费的时间很可能在 50 小时以上，如果想实现进阶要求，请尽早开始你的工作。**