

# 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

唐显达 卢艺晗

2020 年 11 月 - 2025 年 11 月

# 目录

<b>1 实验描述</b>	<b>3</b>
<b>2 实验要求</b>	<b>3</b>
<b>3 类型检查</b>	<b>4</b>
3.1 实验要求 . . . . .	4
3.2 符号表管理 . . . . .	4
3.3 数组初始化规则 . . . . .	5
3.4 实现示例 . . . . .	7
<b>4 中间代码生成</b>	<b>8</b>
4.1 表达式的翻译 . . . . .	9
4.2 控制流的翻译 . . . . .	11
4.3 一个完整的例子 . . . . .	13
<b>5 实验代码框架</b>	<b>16</b>
<b>6 测试方式与评分标准</b>	<b>18</b>
6.1 类型检查（满分 2 分） . . . . .	18
6.2 中间代码生成（满分 8 分） . . . . .	18
6.2.1 基本要求（满分 6 分） . . . . .	19
6.2.2 进阶要求（满分 2 分） . . . . .	19
<b>7 线下检查提问示例</b>	<b>20</b>

## 1 实验描述

欢迎大家来到了编译器构建的关键模块，也是代码量进入新高的一个模块，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，但需要在小雅平台上提交本次实验的 gitlab 链接。
4. 上机课时，以小组为单位，向助教讲解程序。

### 3 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。

#### 3.1 实验要求

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)；
- 如果你采用、或参考了语法实验指导书中提供的示例文法，需额外检查全局作用域内是否有不应出现的语句。

特别的，由于部分测试样例中存在函数内定义参数同名变量的行为（这一行为在 C 语言中是不合法的），在我们的编译器设计中，可以忽略该情况的检查，允许定义参数同名变量。

###### 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。

#### 3.2 符号表管理

为了完成包括未声明与重定义变量等检查，以及传递 const 修饰的变量信息，我们在此次实验中还需要完成符号表的创建与维护。在之前的语法分析实验中，我们只要求大家在语法分析的过程中创建了符号表项，但并没有要求大家实现一个符号表来管理这些符号表项。符号表的主要作用是管理程

序中的各类标识符，在 SysY 编译器中及其属性（如类型、作用域、常量值）。一个支持嵌套作用域的符号表通常需要实现以下几个核心功能：

- **enterScope**: 进入一个新的作用域。当编译器遍历 AST 进入到一个新的作用域（例如函数体、if/while 的语句块）时调用。
- **exitScope**: 退出当前作用域。当编译器完成对一个作用域的遍历时调用。当退出一个作用域时，其内持有的资源外部不需要再使用，可直接清理。
- **addSymbol**: 在当前作用域中添加一个新的符号。在添加时，可以检查该符号是否在当前作用域中已被定义，从而发现重定义错误。
- **getSymbol**: 查找一个符号。查找过程应当从当前作用域开始，如果找不到，则逐级向父作用域查找，直到全局作用域。如果最终仍未找到，则说明该符号未定义。

为了实现对作用域的支持，一种常见的实现方式是采用**栈式符号表**。具体来说，可以用一个类似栈的链式结构来组织各个作用域，每个作用域节点包含一个用于存储当前层级符号的映射结构以及一个指向其父作用域的指针。编译器维护一个指向当前作用域的指针。当进入新作用域时，创建一个新的作用域节点，将其父作用域指针指向当前作用域，然后将当前作用域指针更新为这个新节点。当退出作用域时，只需将当前作用域指针移回其父作用域即可。

### 3.3 数组初始化规则

数组的初始化规则大体上与 C 语言保持一致，只不过由于 SysY 语言没有显式的指针类型，因此初始化器的嵌套层级显然不能超过被初始化数组本身的维度。例如，用一个三层嵌套的列表 `{{{1}}}` 去初始化一个二维数组 `int a[2][2]` 是不合法的。编译器应能检查并报告此类维度不匹配的错误。下面简要说明一下数组的初始化规则：

**元素填充与默认初始化** 数组的初始化过程，本质上是使用初始化列表中的值，按照**行主序**依次填充数组的各个元素。如果初始化列表提供的元素数量少于数组元素的总量，那么数组中所有未被显式初始化的元素都将被默认初始化为 0。

**初始化列表的嵌套与展平** SysY 支持嵌套和展平两种形式的初始化列表，二者在填充规则上逻辑一致。

- **嵌套初始化**: 使用嵌套的花括号对应数组的维度结构。内层的花括号用于初始化对应的子数组。

```
1 int b[2][3] = {{1, 2}, {3}};
2 // {1, 2} 用于初始化第一行 b[0]。b[0][0] = 1, b[0][1] = 2。
3 //      该行剩余的 b[0][2] 被默认初始化为 0。
4 // {3} 用于初始化第二行 b[1]。b[1][0] = 3。
5 //      该行剩余的 b[1][1] 和 b[1][2] 被默认初始化为 0。
6 // 最终 b 的内容为 {{1, 2, 0}, {3, 0, 0}}。
7
```

- **展平初始化**: 如果省略了内层的花括号，初始化列表中的元素会依次、逐个地填充到数组的存储空间中，仍然遵循行主序。

```

1  int c[2][3] = {1, 2, 3, 4};
2  // 列表中的 1, 2, 3 依次填充第一行 c[0]。
3  //      c[0][0] = 1, c[0][1] = 2, c[0][2] = 3。
4  // 列表中的 4 填充第二行 c[1] 的第一个元素。
5  //      c[1][0] = 4。此时初始化列表已用尽。
6  //      第二行剩余的 c[1][1] 和 c[1][2] 被默认初始化为 0。
7  // 最终 c 的内容为 {{1, 2, 3}, {4, 0, 0}}。
8

```

- **混合与嵌套初始化：**初始化规则可以灵活组合。一个高维数组的初始化列表，其每个元素本身就是对一个低维子数组的初始化，因此可以独立应用嵌套或展平规则。例如：

```

1  int e[4][2][3] = {{1, 2, 3, 4}, {5, 6, 7}, {8}};
2  // 这是一个对三维数组 e[4][2][3] 的部分初始化
3
4  // e[0] (一个 int[2][3] 数组) 使用了展平规则进行初始化
5  // 初始化为 {1, 2, 3, 4}
6  // 结果: e[0] 的内容为 {{1, 2, 3}, {4, 0, 0}}
7
8  // e[1] (一个 int[2][3] 数组) 使用了嵌套规则进行初始化
9  // 初始化为 {{5, 6, 7}, {8}}
10 //   e[1][0] (一个 int[3] 数组) 的初始化为 {5, 6, 7}
11 //   e[1][1] (一个 int[3] 数组) 的初始化为 {8}
12 // 结果: e[1] 的内容为 {{5, 6, 7}, {8, 0, 0}}
13
14 // e[2] 和 e[3] 没有对应的初始化器, 因此被完全默认初始化为 0
15
16 /*
17     最终, e 的内容为
18     {
19         {{1, 2, 3}, {4, 0, 0}}, // e[0]
20         {{5, 6, 7}, {8, 0, 0}}, // e[1]
21         {{0, 0, 0}, {0, 0, 0}}, // e[2]
22         {{0, 0, 0}, {0, 0, 0}} // e[3]
23     }
24 */
25

```

**过量初始化** 任何情况下，初始化列表中元素的总数都不能超过数组元素的总数。如果超出了，编译器必须报告一个错误。

```

1  int d[2][2] = {1, 2, 3, 4, 5}; // 错误: 初始化列表过长

```

### 3.4 实现示例

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。同时，在类型检查过程中我们也可以做一些编译期的运算，如将字面量与常量的运算在检查时直接完成，这也是要检查出除 0 错误必须实现的。这里以字面量表达式与 If 语句的类型检查为例：

---

```

1  bool ASTChecker::visit(LiteralExpr& node)
2  {
3      // 字面量表达式一定是常量表达式，且我们可以确定它不会产生语义错误（若词法分析结果无误）
4      node.attr.val.isConstexpr = true;
5      node.attr.val.value       = node.literal;
6      return true;
7  }
8  bool ASTChecker::visit(IfStmt& node)
9  {
10     bool res = apply(*this, *node.cond);
11     if (node.cond->attr.val.value.type == voidType)
12     {
13         // If 的条件是不可参与运算的 void 类型，应当报告该错误
14         return false;
15     }
16     // 依此检查可能存在的 then 语句与 else 语句
17     if (node.thenStmt) res &= apply(*this, *node.thenStmt);
18     if (node.elseStmt) res &= apply(*this, *node.elseStmt);
19     return res;
20 }
```

---

对于 if (0512) s1; else s2; 这一语句，在语法分析阶段，我们可以对它构造出这样的语法树节点：

```

IfStmt
├── cond: LiteralExpr: 0512
├── thenStmt: s1
└── elseStmt: s2

```

通过上面定义的类型检查规则，在看到 If 语句时，就会先去对它的 cond 表达式递归应用检查，并在表达式的类型检查过程中完成节点属性的准备。完成 cond 的检查后再查看其属性类型是否为 void，如果是，那么这就是一个语义上错误的 If 语句；如果不是，我们继续往后检查其它内容。对其它语法树节点的检查流程也大致如此，只不过检查过程中是否需要记录什么内容，需要准备什么数据就需要同学们自行考虑了。

在本学期提供的实验框架中，interfaces/frontend/ast/ast\_defs.h 文件内定义了下面几个类型，你可能会在类型检查的实验中使用到它们：

- **VarValue**: 组合类型指针与持有的值，用于在类型检查中传递表达式的值；
- **ExprValue**: 持有 VarValue 与一个 bool，bool 用于表示该表达式值是否能在编译期内确定；
- **VarAttr**: 记录定义的变量属性，详见成员变量；
- **NodeAttr**: 用于记录某一节点的具体操作类型与类型检查后设置的属性值。

## 4 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：

```
1 // Origin Code
2 // Omit preceding
  ⇔ declarations int
  ⇔ a,b,c;
3 c = a+b;
```

```
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
```

```
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)
```

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。

## 4.1 表达式的翻译

此处以一个二元运算表达式的翻译流程作为示例，其核心流程是：递归生成左右子表达式、进行类型转换（让参与运算的操作数均为同一类型）、再依据结果类型发出算术指令，并插入到当前基本块中。LLVM IR 的层级结构相信同学们在之前的实验中已经了解了，此处不再赘述。

---

```

1  class BinaryExpr : public ExprNode
2  {
3      ExprNode* lhs;
4      ExprNode* rhs;
5      OpCode   op;
6
7      Operand* codeGen() override;
8  };
9
10 Operand* BinaryExpr::codegen()
11 {
12     // 处理左右的子表达式，并准备它们的运算结果
13     Operand* lhsOp = lhs->codegen();
14     Operand* rhsOp = rhs->codegen();
15
16     // 类型提升与隐式转换
17     DataType lhsType = getType(lhsOp);
18     DataType rhsType = getType(rhsOp);
19     DataType pType   = promoteType(lhsType, rhsType);
20     if (lhsType != pType) lhsOp = createConvertInst(lhsType, lhsOp, pType);
21     if (rhsType != pType) rhsOp = createConvertInst(rhsType, rhsOp, pType);
22
23     // 生成二元运算指令
24     Operand* resOp = getNewOperand();
25     insert(new ArithInst(resOp, op, lhsOp, rhsOp));
26
27     return resOp;
28 }

```

---

处理流程即为先分别完成左右操作数的运算，并获取到它的运算结果。在前面的语义分析中，我们设置的结点属性此时就可以发挥作用了，我们需要通过左操作数与右操作数的结点属性来获取其类型，再根据类型做对应的类型提升。完成类型提升后，再为当前的二元表达式分配一个新的操作数用于存储它的运算结果，再创建对应的运算语句即可。此处以这样一个式子来进行更详细的说明，对于表达式  $9.0 * (1 + f(3))$ ，此处认为  $f$  函数返回类型为 `int`，那么我们可以得到下面的语法树：

```

BinaryExpr *
├── lhs: literal float: 9.0
└── rhs: BinaryExpr +

```

```

├ lhs: literal int: 1
└ rhs: Call f(3)

```

接下来, 从最顶层进入, 会依次产生下面的语句:

1. 首先处理最外层的  $*$  表达式。先处理左子表达式  $9.0$ 。这是一个浮点数字面量, 我们为其分配操作数寄存器 `%reg_1` 并生成加载该常量的指令。

---

```
%reg_1 = fadd float 0x4022000000000000, 0x0
```

---

2. 接着, 处理右子表达式  $1 + f(3)$ 。这又是一个二元运算表达式, 因此会再次进入 `handleBinaryCalc` 逻辑。

(a) 处理  $+$  的左操作数  $1$ 。这是一个整数字面量, 我们为其分配寄存器 `%reg_2` 并加载该常量。

---

```
%reg_2 = add i32 1, 0
```

---

(b) 处理  $+$  的右操作数。

---

```
%reg_3 = call i32 @f(i32 3)
```

---

(c) 此时  $+$  的两个操作数都已处理完毕, 分别为 `i32` 类型的 `%reg_2` 和 `i32` 类型的 `%reg_3`。它们的类型相同, 无需进行类型提升。直接生成整数加法指令, 并将结果存入新寄存器 `%reg_4`。

---

```
%reg_4 = add i32 %reg_2, %reg_3
```

---

3. 至此, 内层  $+$  表达式处理完毕, 返回到外层的  $*$  表达式。此时  $*$  的左操作数是 `%reg_1` (`float` 类型), 右操作数是 `%reg_4` (`i32` 类型)。
4. 根据类型提升规则, 结果类型 `pType` 应为 `float`。由于右操作数 `%reg_4` 的类型 `i32` 与 `pType` 不同, 需要插入一条类型转换指令, 将其从 `i32` 转换为 `float`, 并存入新寄存器 `%reg_5`。

---

```
%reg_5 = sitofp i32 %reg_4 to float
```

---

5. 现在两个操作数 `%reg_1` 和 `%reg_5` 都是 `float` 类型了。生成浮点数乘法指令, 并将最终结果存入新寄存器 `%reg_6`。

---

```
%reg_6 = fmul float %reg_1, %reg_5
```

---

通过以上步骤, 我们就完成了对表达式  $9.0 * (1 + f(3))$  的 IR 生成, 当前的表达式计算结果放在寄存器 6 中, 且会返回给此函数的调用者。若该表达式是另一更大表达式的子部分, 或其它可能有表达式的地方, 那么调用者就可以使用返回的寄存器操作数作为计算结果, 继续生成其它部分的程序。

## 4.2 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<sup>1</sup>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 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

<sup>1</sup>参考龙书 p263-p268

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，回填技术的应用场景主要针对于自底向上的分析过程中，如理论课上提到的语法制导翻译过程。而对于我们目前的实验内容来说，由于我们已经构建出了一棵完整的抽象语法树，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。该思路本质上和回填技术并没有太大的差别，但对于已经构建好 AST 这样的，便于我们进行自顶向下遍历的结构而言，使用该方法可能显得更为自然。例如说，对于逻辑与的短路运算，使用这一思路，我们就可以写成下面这样的形式：

---

```

1  Block* rhsEvalBlock = createBlock();
2  lhs.trueTar  = rhsEvalBlock->label;      // 与运算时，左表达式为真则看右表达式
3  lhs.falseTar = node.falseTar;           // 左表达式为假则直接退出即可
4  rhs.trueTar  = node.trueTar;
5  rhs.falseTar = node.falseTar;
6
7  Operand* lhsOp = lhs->codeGen();
8  lhsOp = transferToI1(lhsOp);
9  insert(createBranchInst(cond, lhs.trueTar, lhs.falseTar));
10
11 enterBlock(lhs.trueTar);
12 rhsOp = rhs->codeGen();
13 rhsOp = transferToI1(rhsOp);
14
15 return res;

```

---

额外的，虽然 SysY 语言特性不要求实现逻辑表达式被赋值给一个左值，但如果有同学想实现这一点，可以使用 LLVM IR 的 PHI 结点来实现。PHI 结点用于在 SSA 形式程序中，根据来源的不同选择具体的操作数，其用法为 `<res> = phi <ty> [<val>, <label>], ...`。比如说，这里给出一个具体的 PHI 语句：`%r1 = phi i32 [0, %b0], [%r2, %b1], [1, %b2]`，意即从 %b0 跳转而来时就取将 0 赋值给 %r1，否则赋值 %r2/1 给 %r1。提供的实验框架中也给出了 PHI 语句的定义，至于怎么使用，以及如何收集来源值就请自行考虑了。

### 4.3 一个完整的例子

图4.1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  }

```

---

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

```

---

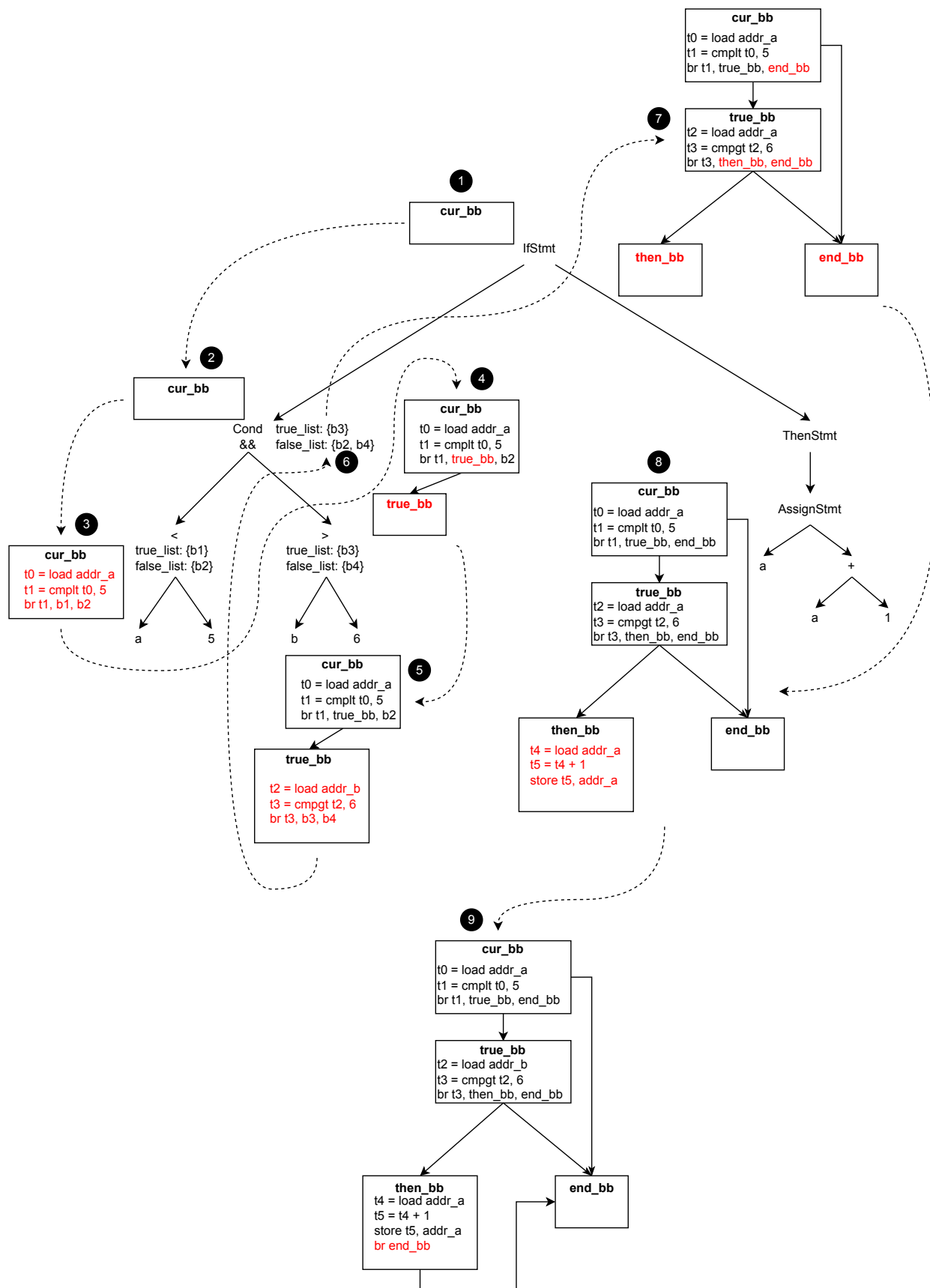


图 4.1: 中间代码生成的一个例子

```

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 }
30

```

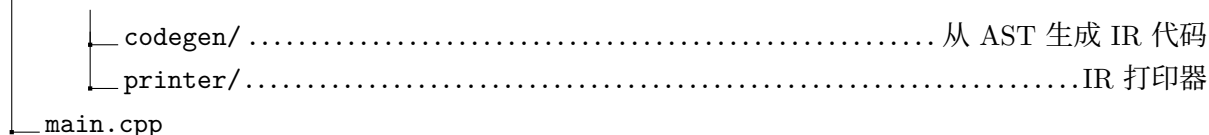
## 5 实验代码框架

本次实验可能涉及到的代码如下

```

./
├── frontend/
│   ├── ast/
│   │   └── visitor/
│   │       └── semantic_check/.....AST 上类型检查
│   └── symbol/.....符号表
├── interfaces/.....基础定义
├── middleend/
│   ├── module/.....IR 模块定义
│   └── visitor/

```



其中，你主要需要完成 **semantic\_check** 与 **codegen** 两个目录下的访问者类访问到各 AST 结点时的处理逻辑。对中端的 LLVM IR 代码的打印工具已经在 **printer** 下给出，如你实现的符合预期则应当能打印出正确的 LLVM IR 代码。当然，视你实际需求，你可以自由修改任意组件的代码逻辑，不需要再反复向助教确认能否修改。

前端的抽象语法树类在语法实验中已经涉及，这里就不再介绍。这里主要说明下实验过程中需要使用到的一些中端类型定义：

- **Module** 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全局变量、函数声明与函数定义。
- **Function** 是函数模块。函数由多个基本块构成，每个函数都有一个入口基本块（方便起见，可统一为块 0 作为入口块）。它同时管理着函数内虚拟寄存器和基本块标签的分配。
- **Block** 为基本块。基本块包含一个中间代码指令的列表。基本块中的指令顺序执行，并以一个终结指令（跳转或返回指令）结束。基本块之间通过跳转指令形成控制流图。
- **Instruction** 是我们中间代码的指令基类。所有具体的 IR 指令都继承自该类。框架中已经派生出了多种指令，例如：

<b>LoadInst</b>	从内存地址中加载值到虚拟寄存器中。
<b>StoreInst</b>	将值从虚拟寄存器存储到内存地址中。
<b>ArithmeticInst</b>	二元运算指令，包含一个目的操作数和两个源操作数。
<b>IcmpInst/FcmpInst</b>	整数/浮点数比较指令。
<b>BrCondInst</b>	条件跳转指令。
<b>BrUncondInst</b>	无条件跳转指令。
<b>RetInst</b>	函数返回指令。
<b>AllocaInst</b>	在栈上分配空间。
<b>CallInst</b>	函数调用指令。

- **Operand** 为指令的操作数基类。框架中派生出的操作数类型包括：

<b>RegOperand</b>	虚拟寄存器操作数，代表一个 SSA 形式的值。
<b>ImmeI32Operand/ImmeF32Operand</b>	32 位整型/浮点型立即数操作数。
<b>GlobalOperand</b>	全局变量操作数。
<b>LabelOperand</b>	基本块标签操作数，用于跳转指令。

框架通过 **OperandFactory** 来创建并管理操作数实例，以确保操作数的唯一性。这样我们就可以直接通过比较指针地址来对两个操作数判等。

- **ASTCodeGen**（定义于 `middleend/visitor/codegen/ast_codegen.h`）是一个核心的访问者类，它负责遍历经过语义分析的 AST，并将其翻译成由 **Module**, **Function**, **Block**, **Instruction** 构成的中间表示。

在实验框架中并未给类型检查专门划分出一个阶段，而是在语法分析后检查语义分析结果是否无误，见：

---

```
1 FE::AST::ASTChecker checker;
2 bool          accept = apply(checker, *ast);
3 if (!accept)
4 {
5     cerr << "Semantic check failed with " << checker.errors.size() << " errors." << endl;
6     for (const auto& err : checker.errors) cerr << "Error: " << err << endl;
7     ret = 1;
8     goto cleanup_ast;
9 }
```

---

在你完成实验后，使用

```
bin/compiler -llvm test.sy -o test.ll
```

来使用编译器生成中间代码。如果该样例实际上是存在语义错误的，那么在上面的检查中应当被报告出来，并让程序提前退出。

## 6 测试方式与评分标准

### 6.1 类型检查（满分 2 分）

这一部分仍然没有自动测评脚本，你还是需要根据给出的样例文件，运行它并报告出错误。助教会单独进行确认。

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。在随实验框架一起发布的测试样例文件夹下也有 `semant` 子目录，其中包含了一些类型错误的代码，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。

### 6.2 中间代码生成（满分 8 分）

从这一部分开始，我们提供了自动测评脚本。即使你选择不采用提供的代码框架来完成实验，你也需要去获取其中的 `testcase` 目录与 `test.py` 测评脚本，我们会根据你的编译器对 `testcase` 目录下的测试文件的通过情况来进行给分。需要注意的是，如无特殊情况，不允许擅自修改测评脚本；任意情况下，不允许擅自修改测试文件（除非你向助教报告了某一测试文件是错误的）。测试脚本的使用方法为：

---

```
1 python test.py --stage=llvm --group={Basic, Advanced} --opt={0, 1, 2}
```

---

其中 `group` 缺省为 `Basic`，`opt` 缺省为 `0`。

如果你想对单个 `ll` 文件进行测试，可以参考下面的指令：

---

```
1 clang test.ll -c -o test.o -w
2 clang -static test.o -L./lib -lsys_x86 -o test.bin
```

---

### 6.2.1 基本要求 (满分 6 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. 函数的声明与调用
7. 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
8. 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 100 个测试用例, 其中 100 个测试用例平分 5 分, 即每个测试用例的分值为 0.05 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。

### 6.2.2 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. 数组 (一维、二维、...) 的声明和数组元素访问
2. 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 0.05 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 0.95 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。**

## 7 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. 请描述符号表在语义检查中发挥的作用，以及你是如何实现符号表的。
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 `const 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。
9. 说明你是如何在 `if while` 等语句的处理中为不同基本块确定跳转目标的。