

# 类型检查 & 中间代码生成

杨科迪 费迪

2021 年 11 月 14 日

# 目录

<b>1 实验描述</b>	<b>3</b>
<b>2 实验要求</b>	<b>3</b>
<b>3 实验流程</b>	<b>4</b>
3.1 代码框架 . . . . .	4
3.2 类型检查 . . . . .	6
3.3 中间代码生成 . . . . .	7
3.3.1 表达式的翻译 . . . . .	7
3.3.2 控制流的翻译 . . . . .	7
3.3.3 实验效果 . . . . .	9
<b>4 评分标准</b>	<b>11</b>
4.1 类型检查（满分 2 分） . . . . .	11
4.2 中间代码生成（满分 4 分） . . . . .	11
4.2.1 基本要求 . . . . .	11
4.2.2 进阶要求 . . . . .	11

## 1 实验描述

欢迎大家来到了编译器构建的全新模块，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 代码框架

本次实验框架代码的目录结构如下：

```
./
├── include
│   ├── 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
├── src
│   ├── Ast.cpp
│   ├── BasicBlock.cpp
│   ├── Function.cpp
│   ├── Instruction.cpp
│   ├── lexer.l
│   ├── main.cpp
│   ├── Operand.cpp
│   ├── parser.y
│   ├── SymbolTable.cpp
│   ├── Type.cpp
│   └── Unit.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。
- 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`。我们派生出的指令包含:

<b>LoadInstruction</b>	从内存地址中加载值到中间变量中。
<b>StoreInstruction</b>	将值存储到内存地址中。
<b>BinaryInstruction</b>	二元运算指令, 包含一个目的操作数和两个源操作数。
<b>CmpInstruction</b>	关系运算指令。
<b>CondBrInstruction</b>	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> 。
<b>UncondBrInstruction</b>	无条件跳转指令, 直接跳转到基本块 <code>branch</code> 。
<b>RetInstruction</b>	函数返回指令。
<b>AllocaInstruction</b>	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, 我们实现的类型包含:

<b>IntType</b>	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
<b>VoidType</b>	仅用于函数的返回类型
<b>FunctionType</b>	函数类型, 包含函数的返回值类型和形参类型
<b>PointerType</b>	指针类型, <code>valueType</code> 为所指向的值的类型

## 3.2 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，由于我们基于 SysY 语言进行了语法设计，Identifier 的基本类型仅包括 int，这就减轻了类型检查的工作任务。我们仅需要针对以下几种情况进行处理。

- 变量未声明，及在同一作用域下重复声明
- 条件判断表达式 int 至 bool 类型隐式转换
- 数值运算表达式运算数类型是否正确（可能导致此错误的情况有返回值为 void 的函数调用结果参与了某表达式计算）
- 函数未声明，及不符合重载要求的函数重复声明
- 函数调用时形参及实参类型或数目的不一致
- return 语句操作数和函数声明的返回值类型是否匹配
- 实现了数组要求的同学，还可以对数组维度进行相应的类型检查
- 实现了 break、continue 语句要求的同学，还可以对 break、continue 语句进行静态检查，判断是否仅出现在 while 语句中

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：

---

```
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。

### 3.3 中间代码生成

词法分析和语法分析是编译器的前端，中间代码是编译器的中端，目标代码是编译器的后端，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，我们可以极大的简化编译器的构造。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。

#### 3.3.1 表达式的翻译

---

```
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，辅助我们进行中间代码生成。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。

#### 3.3.2 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跳转指令的列表，回填是指当跳转的目标基本块确定时，设置列表中跳转指令的跳转目标为该基本块。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。

##### 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();
```

---

<sup>1</sup>参考龙书 p263-p268

---

```

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());

```

---

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。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`。



### 3.3.3 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
```

---

## 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，我们也为大家提供了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 4 分）

#### 4.2.1 基本要求

1. 实现指令双向循环链表的插入操作。
2. 级别一（基本要求）中所有语法特性的中间代码生成：
  - 变量、常量的声明和初始化
  - 算数表达式的翻译
  - 关系表达式的翻译
  - 布尔表达式的翻译
  - 一般语句的翻译，块语句、赋值语句等
  - 控制流语句的翻译，if-else 语句、while 语句、return 语句等
  - 输入输出函数、函数调用等
3. 根据基本块的前驱、后继关系进行流图的构造。

只要实现了以上基本要求即可得到满分。

#### 4.2.2 进阶要求

1. 完成你的编译器所支持的其他语法特性的中间代码生成工作，如 break、continue 语句、数组等。
2. 我们目前生成的中间代码是静态单赋值形式（Static Single Assignment）的，SSA 形式中间代码的性质是每一个变量只能在一条指令中定义，可以在多条指令中使用，你可以理解为 SSA 形式的中间代码只有 data dependence, 不存在 name dependence。SSA 为优化带来了方便，比如对于死代码删除来说：我们可以遍历指令列表，如果一条指令定义了一个目的操作数，使用它的指令数为 0，那么这条指令就是死代码，可以将其从指令列表中删除。

对于临时变量，我们通过标号不断递增的方式，确保每个临时变量只被定义一次，从而可以很容易的维护临时变量的 SSA 性质。对于局部变量，我们的处理方式是使用 alloca 指令为其在内存中分配空间，alloca 指令定义了一个临时变量 t，t 为局部变量所对应的内存地址，使用该局部变量前从 t 中加载（使用 t），定义该局部变量向内存地址中存储值（使用 t），通过该方式，我们的局部变量也是符合 SSA 形式的。你很容易想到，局部变量也是可以放在寄存器中的（把局部变量视作临时变量），但是源程序可能多次定义一个局部变量，你又灵机一动，可以通过类似寄存器重命名的方式，每定义一次局部变量，便创建一个该局部变量的不同实例，对局部变量的使用与相应的实例进行对应，但是问题又来了，如图4.1所示，局部变量 a 在基本块 B1 中被定义，我们将其重命名为  $a_1$ ，a 在 B2 中也被定义了一次，我们将其重命名为  $a_2$ ，那么在基本块 B3 中对局部

变量  $a$  的使用是  $a_1$  还是  $a_2$  呢, 我们不能确定, 因为我们无法知道程序执行时是从 B1 还是 B2 跳转到 B3 的。

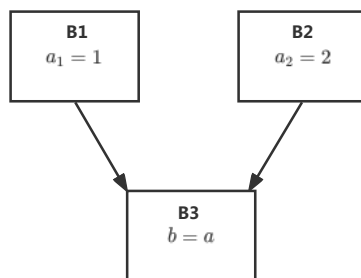


图 4.1: 重命名产生的问题

SSA 使用  $\phi$  函数来解决这一问题, 如图4.2所示, 我们在 B3 的开头插入了  $\phi$  函数  $a_3 = \phi(a_1, a_2)$ , 如果从 B1 跳转到了 B3, 那么  $\phi(a_1, a_2) = a_1$ , 如果从 B2 跳转到了 B3, 那么  $\phi(a_1, a_2) = a_2$

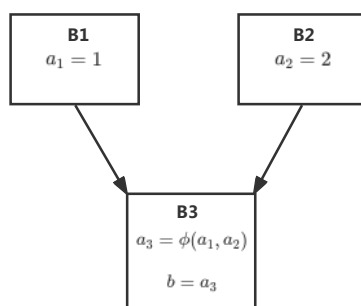


图 4.2: 静态单赋值形式

SSA 形式的转化需要进行支配边界的计算、 $\phi$  函数插入、变量重命名等步骤, 查阅相关资料进行实现。

- 考虑无用控制流消除的实现, 在上一小节生成的中间代码中, B17 跳转到了 B24, B24 跳转到了 B22, B22 只有 B24 一个前驱结点, 我们可以将 B24 和 B22 合并, B17 直接跳转到 B24 和 B22 合并后的基本块。
- 实现不可达代码消除优化, 从 entry 基本块开始对流图进行深度或广度优先遍历, 保留 entry 基本块所在的连通分支, 将不在 entry 连通分支的其他基本块删除。
- 考虑代数化简和强度削弱优化, 比如将乘法指令转化为移位指令、乘以 0 和 1 的优化等。
- 实现常量折叠优化, 常量折叠是指当算数指令的两个操作数都为常量时, 我们直接在编译时计算出运算结果。
- 查阅相关资料, 学习并实现稀疏条件常量传播优化 (SCCP)。
- 考虑寄存器分配算法的实现。通过寄存器分配, 将临时变量 (虚拟寄存器) 映射到物理寄存器, 生成最终的目标代码。