



南开大学  
Nankai University

南开大学

计算机学院和网络空间安全学院

## 编译系统原理实验报告

---

### SysY 编译器设计与实现

---

刘修铭 2112492 信息安全

年级：2021 级

指导教师：王刚

[HTTPS://GITLAB.EDUXIJI.NET/NKU2023/NKU2023\\_COMPILER](https://gitlab.eduxiji.net/NKU2023/NKU2023_COMPILER)

2024 年 1 月 16 日

## 摘要

本次编译系统原理工程作业完成了从词法分析、语法分析、类型检查、中间代码生成再到目标代码生成及优化的全过程，实现了自己定义的 SysY 语言特性中的全部内容，完成了对变量作用域的区分、break 和 continue 等语句、非叶函数和数组声明及初始化等功能。

在给定的模板的基础上，我们调整了整体框架，使之能够更好的实现相关功能，最终完整构建了 SysY 语言编译器。

**关键字：** SysY 语言，ARM，词法分析，语法分析，类型检查，中间代码生成，目标代码生成及优化，编译系统原理

# 目录

<b>一、 分工</b>	<b>1</b>
(一) 词法分析	1
(二) 语法分析	1
(三) 语义分析 (类型检查)	1
(四) 中间代码生成	1
(五) 目标代码生成	2
(六) 代码优化	2
<b>二、 总体设计</b>	<b>3</b>
(一) 词法分析	3
(二) 语法分析	3
(三) 语义分析 (类型检查)	4
(四) 中间代码生成	4
(五) 目标代码生成	5
(六) 代码优化	5
<b>三、 词法分析</b>	<b>6</b>
(一) 运行时库函数的识别与连接	6
(二) 注释	6
1. 单行注释	6
(三) 多行注释	7
(四) ID 与字符串	7
<b>四、 语法分析</b>	<b>9</b>
(一) 类型系统	9
(二) 翻译模式	9
1. 变量定义	10
2. 数组定义	12
(三) 抽象语法树	12
<b>五、 语义分析 (类型检查)</b>	<b>14</b>
(一) 数组的检查	14
(二) 运算对象的检查	15
<b>六、 中间代码生成</b>	<b>17</b>
(一) AST::genCode	17
(二) 二元运算	17
1. 逻辑短路机制	17
2. 类型转换	17
(三) 声明语句	19
1. 全局变量	19
2. 局部变量	20

<b>七、 目标代码生成</b>	<b>23</b>
(一) 基本汇编指令 . . . . .	23
(二) 数组 . . . . .	27
(三) 全局数据 . . . . .	32
<b>八、 代码优化</b>	<b>34</b>
1. delAEB . . . . .	34
2. vec_intersection . . . . .	35
3. vec_union . . . . .	35
4. vintersection . . . . .	36
5. local_elim_cse . . . . .	36
6. elim_cse . . . . .	39
<b>九、 遇到的困难及解决方案</b>	<b>41</b>
<b>十、 总结与感想</b>	<b>42</b>

## 一、 分工

从 lab1-lab6, 不知道熬了多少夜。分工比较粗略, 更多的是一边写, 一边讨论; 一边说, 一边骂自己笨。相互扶持, 相互补充。舍友的关系, 让讨论变得更加常态化。

一台电脑, 两个人,  $n$  个夜晚, 一个奇迹。

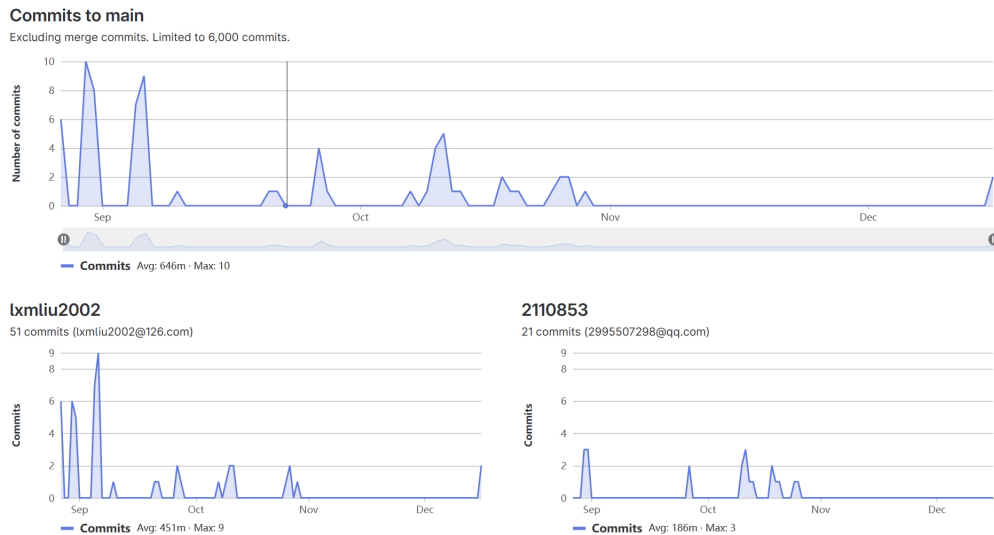


图 1: gitlab 提交记录

上图是我们 gitlab 仓库的提交记录。下面是我们的大致分工情况:

### (一) 词法分析

在词法分析阶段, 我主要完成了对运行时库函数的识别与连接、单行注释与多行注释的处理、ID、字符串、空白串等的处理; 朱子轩同学则主要负责处理不同进制的数、条件表达式、括号、整型变量、浮点型变量等的识别。

### (二) 语法分析

在语法分析阶段, 我主要对变量和常量等的声明、算数表达式、数组等进行分析处理, 并定义基本的语法树; 朱子轩同学则主要完成符号表、函数等相关内容以及浮点数、条件表达式等的分析处理工作。

### (三) 语义分析 (类型检查)

在语义分析 (类型检查) 部分, 我主要完成对数组及运算对象等部分的检查; 朱子轩同学则主要负责完成对常量及变量的使用、函数的相关部分等进行检查。

### (四) 中间代码生成

在中间代码生成部分, 我主要负责运算语句、条件语句的指令生成, 以及类型转换、数组的相关处理; 朱子轩同学则主要负责浮点部分的中间代码生成, 以及对函数及控制流的相关处理工作。

### (五) 目标代码生成

在目标代码生成部分，我主要负责基本汇编指令、数组及全局变量等的处理；朱子轩同学则主要负责函数调用、浮点类型及控制流相关处理。

### (六) 代码优化

在最后的优化部分，我主要实现了公共子表达式删除操作；朱子轩同学负责实现了无用代码删除工作。

说明：二人原本以为以小组为单位交实验报告，故而前期由二人共同撰写，总体设计部分内容会有重复性。特此说明！

NIJU

## 二、 总体设计

### (一) 词法分析

词法分析在编译过程中扮演着非常重要的角色，它是编译器中的第一个阶段，负责将源代码转换为一系列的记号或标记（tokens），并且为这些标记附加词法信息。其任务是：对源程序进行从左至右的扫描，产生一个个单词符号，以将源代码转化为标记序列，并保存各个单词的属性。主要有以下作用：

- 将源代码转化为标记序列
- 去除空白字符和注释
- 识别关键字和标识符
- 进行错误处理
- 附加词法信息

利用 Flex 工具完成程序设计，输入一个 SysY 语言程序，输出每一个文法单元类别、词素、行号、列号以及其属性值（DECIMAL 的属性为数值，ID 的属性为符号表项指针），从而实现词法分析的功能。在 lexer.l 中完成对 Flex 程序的编写，对于无需考虑属性值的终结符，如“int”，“return”等，通过设置保留字，在 Flex 程序的规则部分直接返回响应的单词；对于需要考虑属性值的终结符，需要对词素进行处理保存其值到 yylval 中；除此之外，对于构成较复杂的单词，我们需要定义其正规式。

### (二) 语法分析

语法分析（Syntax Analysis），是编译器中的重要步骤之一，它在编译过程中扮演着关键的角色。它的将在词法分析的基础上对 tokens 进行组合，形成各种的语法短语，其主要作用是将源代码转换为抽象语法树（Abstract Syntax Tree, AST），以便进一步的编译和优化步骤。

- 类型系统（Type）：类型系统定义了编程语言中的数据类型以及如何类型检查。它确保在程序中的操作和赋值是类型安全的。本项目主要实现了 int、float、void、常量、数组、函数等类型。
- 符号表（SymbolTable）：符号表主要用于作用域的管理，是编译器用于保存源程序符号信息的数据结构，这些信息在词法分析、语法分析阶段被存入符号表中，最终用于生成中间代码和目标代码。符号表条目可以包含标识符的词素、类型、作用域、行号等信息。框架代码中，定义了三种类型的符号表项：用于保存字面值常量属性值的符号表项、用于保存编译器生成的中间变量信息的符号表项以及保存源程序中标识符相关信息的符号表项。
- 抽象语法树（Ast）：语法分析的目的是构建出一棵抽象语法树（AST），因此我们需要设计语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。结点的类型大体上可以分为表达式和语句，每种类型又可以分为许多子类型，如表达式结点可以分为词法分析得到的叶结点、二元运算表达式的结点等；语句还可以分为 if 语句、while 语句和块语句等。在 AST.cpp 中 Node 为 AST 结点的抽象基类，从 Node 中派生出 ExprNode 类；ExprNode 为表达式结点的抽象基类，从 ExprNode 中派生出 BinaryExp、UnaryExpr、FuncCallExp 等类；StmtNode 为语句结点的抽象基类，从 StmtNode 中派生出 CompoundStmt、IfStmt、WhileStmt 等类。

- 语法分析与语法树的创建 (parser): 该部分是语法分析部分的重点内容。词法分析得到的, 实质是语法树的叶子结点的属性值, 语法树所有结点均由语法分析器创建。在自底向上构建语法树时 (与预测分析法相对), 我们使用孩子结点构造父结点。在 yacc 每次确定一个产生式发生归约时, 我们会创建出父结点、根据子结点正确设置父结点的属性、记录继承关系。
- SysY 运行时库连接: 将给定手册中的函数添加到符号表中, 以便在后续的编译阶段中进行引用和处理, 实现了对 SysY 运行时库的连接。

### (三) 语义分析 (类型检查)

语义分析是编译器中的一个关键阶段, 负责确保源代码的语义符合语言规范, 并生成中间代码以供后续阶段使用。类型检查是语义分析的一个重要方面, 其目标是验证变量、表达式和操作符之间的类型关系是否符合语言规范。

语义分析 (类型检查) 主要完成了常量检查, 表达式类型确定, 函数调用参数匹配, 函数返回值匹配, 数组下标维度匹配, 常量计算和变量数组初始化赋值等相关操作。我主要负责对数组及运算对象等部分的检查。在实现中, 部分类型检查已在语法分析阶段完成, 如控制语句不匹配、变量未定义等。其余相关操作均在语义分析阶段完成。

语法分析得到抽象语法树后, 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。

### (四) 中间代码生成

中间代码生成阶段, 是整个编译器的“中端”, 串联起了前端源代码与后端机器码。在本次实验的中间代码生成阶段, 我们将 SysY 源语言翻译成中间代码 LLVM。完成了基本 IR 的构建, 完成了运算语句、条件语句、类型转换、浮点类型、数组的相关处理工作。

- 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, 其中的最后一条指令只能是跳转指令或者函数返回指令, 中间不含有控制流指令。每个基本块都有前驱基本块列表 pred 和后继基本块列表 succ, 基本块相互连接就形成了流图。
- 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。
- 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。在本次实验中, 实现的类型有 IntType、VoidType、FunctionType 和 PointerType 四种类型。



## (五) 目标代码生成

目标代码生成阶段主要完成了对基本汇编指令的生成，完成对整型、浮点型汇编指令及其类型间的相互转换。除此之外，还需要对函数及控制流语句进行翻译。在完成虚拟寄存器的分配后，还需要使用寄存器分配算法（此处为线性扫描分配算法）对虚拟寄存器进行寄存器的分配。

- `AsmBuilder.h` 为汇编代码构造辅助类。类似于中间代码生成中的 `IRBuilder`，其主要作用就是在中间代码向目标代码进行自顶向下的转换过程中，记录当前正在翻译的函数、基本块，以便于函数、基本块及指令的插入。
- `MachineCode.h` 为汇编代码构造相关的框架，大体的结构和中间代码是类似的，只有具体到汇编指令和对应操作数时有不同之处。
- `LiveVariableAnalysis.h` 为活跃变量分析，用于寄存器分配过程。
- `LinearScan.h` 为线性扫描寄存器分配算法相关类，为虚拟寄存器分配物理寄存器。

## (六) 代码优化

在对中间代码进行优化后，从中间代码到目标代码的翻译过程中仍然有可能会引入新的冗余。我们可以通过一些优化方法来对目标代码进行精简。此外，我们可以针对目标代码以及目标平台的一些特点有针对性地进一步优化目标代码。在本次实验中，我们主要针对于“循环”语句进行了针对性优化，实现了公共子表达式删除及无用代码删除的优化操作。

### 三、词法分析

该部分的输入是一个 SysY 语言程序，输出每一个文法单元类别、词素、行号、列号以及其属性值（DECIMAL 的属性为数值，ID 的属性为符号表项指针）。我主要完成了对运行时库函数的识别与连接、单行注释与多行注释的处理、ID、字符串、空白串等的处理，以下是该部分各个功能的实现示例：

#### （一）运行时库函数的识别与连接

此处以 `putint` 函数为例进行说明。这段代码的主要作用是在遇到标识符“`putint`”时执行一些操作，包括将相关信息存储到符号表（`SymbolTable`）中，并返回标识符类型（ID）。

运行时库函数的识别与连接

```

1  "putint" {
2      if (dump_tokens) {DUMP_TOKEN(yytext);}
3      char *lexeme;
4      lexeme = new char[strlen(yytext) + 1];
5      strcpy(lexeme, yytext);
6      yylval.strtype = lexeme;
7      std::vector<Type*> vec;
8      std::vector<SymbolEntry*> vec1;
9      vec.push_back(TypeSystem::intType);
10     Type* funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
11     SymbolTable* st = identifiers;
12     while(st->getPrev()) {st = st->getPrev();}
13     SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
14         getLevel(), -1, true);
15     st->install(yytext, se);
16     return ID;
17 }
```

其他库函数同理即可。完成上述工作，该编译器即可实现对 SysY 运行时库函数的识别与连接。

#### （二）注释

##### 1. 单行注释

对于单行注释，SysY 语言的定义为：以序列“`//`”开始，直到换行符结束，不包括换行符。基于此，我们使用以下代码实现了其功能。

单行注释

```

1  "//" {
2      dump_tokens("SCOMMENT\t%s\t%d\t%d\n", yytext, yylineno - 1, column);
3      column += yyleng;
4      BEGIN SCOMMENT;
5  }
6  <SCOMMENT>.* {
7      dump_tokens("COMMENT\t%s\t%d\t%d\n", yytext, yylineno - 1, column);
8      column += yyleng;
9  }
```

```

9  }
10 <SCOMMENT>\n {
11     column = 0;
12     BEGIN INITIAL;
13 }

```

### (三) 多行注释

对于多行注释, SysY 语言的定义为: 以序列 ‘/\*’ 开始, 直到第一次出现 ‘\*/’ 时结束, 包括结束处。基于此, 我们使用以下代码实现了其功能。当 Flex 匹配到多行注释时, 它会输出适当的标记, 并在注释内捕获注释文本, 实现了多行注释的匹配功能。

#### 多行注释

```

1  "/*" {
2      dump_tokens("MCOMMENTBEGIN\t%s\t%d\t%d\n", yytext, yylineno - 1, column);
3      column += yyleng;
4      BEGIN MCOMMENT;
5  }
6  <MCOMMENT>"*/" {
7      dump_tokens("MCOMMENTEND\t%s\t%d\t%d\n", yytext, yylineno - 1, column);
8      column += yyleng;
9      BEGIN INITIAL;
10 }
11 <MCOMMENT>. {
12     dump_tokens("MCOMMENT\t%s\t%d\t%d\n", yytext, yylineno - 1, column);
13     column += yyleng;
14 }
15 <MCOMMENT>\r\n|\n|\r {
16     column = 0;
17 }

```

### (四) ID 与字符串

参考框架中的定义, 我们给出了 ID 与字符串的词法分析定义。

#### ID 与字符串

```

1  {ID} {
2      if(dump_tokens)
3          DUMP_TOKEN(yytext);
4      char *lexeme;
5      lexeme = new char[strlen(yytext) + 1];
6      strcpy(lexeme, yytext);
7      yylval.strtype = lexeme;
8      return ID;
9  }
10 {STRING} {
11     if(dump_tokens)

```

```

12     DUMP_TOKEN(yytext);
13     char* lexeme;
14     lexeme = new char[strlen(yytext) + 1];
15     strcpy(lexeme, yytext);
16     yylval.strtype = lexeme;
17     return STRING;
18 }

```

借助编写的 `lexer.l` 程序，我们可以实现对给定 SysY 编程语言的词法分析。以给定示例代码为例，我们可以得到下面的分析结果，每个单词的类别、词素、行号、列号及其属性都能够正确输出。

```

example.sy
1 int a;
2 int main()
3 {
4     int a;
5     a = 1 + 2;
6     if(a < 5)
7         return 1;
8     return 0;
9 }

example.toks
1 INT int 0 0
2 ID a 0 4 0x55b1d13e0f60
3 SEMICOLON ; 0 5
4 INT int 1 0
5 FUNCID main 1 4 0x55b1d13e0f60
6 LPAREN ( 1 8
7 RPAREN ) 1 9
8 LBRACE { 2 0
9 INT int 3 4
10 ID a 3 8 0x55b1d13e74c0
11 SEMICOLON ; 3 9
12 ID a 4 4 0x55b1d13e74c0
13 ASSIGN = 4 6
14 DECIMAL 1 4 8 1
15 ADD + 4 10
16 DECIMAL 2 4 12 2
17 SEMICOLON ; 4 13
18 IF if 5 4
19 LPAREN ( 5 6
20 ID a 5 7 0x55b1d13e74c0
21 LESS < 5 9
22 DECIMAL 5 5 11 5
23 RPAREN ) 5 12
24 RETURN return 6 8
25 DECIMAL 1 6 15 1
26 SEMICOLON ; 6 16
27 RETURN return 7 4
28 DECIMAL 0 7 11 0
29 SEMICOLON ; 7 12
30 RBRACE } 8 0

```

图 2: 词法分析结果

## 四、 语法分析

在语法分析阶段，我主要对变量和常量等的声明、算数表达式、数组等进行分析处理，并定义基本的语法树。

### (一) 类型系统

本项目中，我们实现了 int、float、void、常量、数组、函数等多个类型。此处以个人完成的数组类型为例进行说明。

首先创建一个空的字符串向量 vec 以存储数组的维度信息，并初始化一个指针 temp 指向当前数组类型。然后，进入循环，只要 temp 不为空且为数组类型，就会继续循环，逐一处理数组的维度。在循环中，它构建每个维度的字符串表示并将其添加到 vec 中。接着，它将 temp 移动到下一个维度的数组元素类型，并确保元素类型为整数。最后，它创建一个字符串流 buffer，构建最终的类型字符串表示，包括维度信息、是否为常量以及数组元素的类型，然后将这个字符串返回。

数组类型

```
1 string ArrayType::toStr()
2 {
3     vector<string> vec;
4     Type *temp = this;
5     while (temp && temp->isArray())
6     {
7         ostringstream buffer;
8         if (temp == this && length == 0) buffer << '[' << ' ';
9         else buffer << '[' << ((ArrayType *)temp)->getLength() << ' ';
10        vec.push_back(buffer.str());
11        temp = ((ArrayType *)temp)->getElementType();
12        ;
13    }
14    assert(temp->isInt());
15    ostringstream buffer;
16    if (constant) buffer << "const ";
17    buffer << "int";
18    for (auto it = vec.begin(); it != vec.end(); it++) buffer << *it;
19    return buffer.str();
20 }
```

### (二) 翻译模式

该部分是语法分析部分内容的重中之重，需设计各种语法块的翻译模式。项目中使用 Bison 和 Flex 编写了一个语法分析器。为了避免报告冗长，此处仅作部分展示。

在框架代码之上，为了完善我们的编译器功能，我们又添加了许多终结符与非终结符，如 FLOAT、CONTINUE、FuncDefParams 等。

## 1. 变量定义

此部分主要用于解析声明语句，其有两个主要的产生式。第一个产生式用于解析变量，第二个用于常量的声明解析。其通过 Node 类中的 next 指针将所有的声明串联到一起。而对于常量声明，借助 setConst() 函数为其设置标志位。

此部分主要用于变量定义的分析，其主要有四个产生式。第一个产生式用于解析变量，第二个用于处理数组相关内容，第三个主要处理带初始化的变量声明。第四个则主要处理带数组索引和初始化值的变量声明。

以第一个产生式为例进行简要说明。当遇到一个简单的标识符(ID),创建一个新的标识符符号表条目(IdentifierSymbolEntry);设置该符号表条目的类型为声明时指定的类型(declType);利用符号表(identifiers)安装这个标识符,检查是否已经定义过;最后创建对应的声明语句(DeclStmt),将其返回。

### 变量定义

```

1 VarDef
2   : ID {
3       SymbolEntry* se;
4       se = new IdentifierSymbolEntry(declType, $1, identifiers->getLevel())
5       ;
6       if (!identifiers->install($1, se))
7           fprintf(stderr, "identifier \"%s\" is already defined\n", (char*)
8               $1);
9       $$ = new DeclStmt(new Id(se));
10      delete [] $1;
11  }
12  | ID ArrayIndices {
13      SymbolEntry* se;
14      std::vector<int> vec;
15      ExprNode* temp = $2;
16      while (temp) {
17          vec.push_back(temp->getValue());
18          temp = (ExprNode*)(temp->getNext());
19      }
20      Type* type = declType;
21      Type* temp1;
22      while (!vec.empty()) {
23          temp1 = new ArrayType(type, vec.back());
24          if (type->isArray())
25              ((ArrayType*)type)->setArrayType(temp1);
26          type = temp1;
27          vec.pop_back();
28      }
29      arrayType = (ArrayType*)type;
30      se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
31      ((IdentifierSymbolEntry*)se)->setAllZero();
32      if (!identifiers->install($1, se))
33          fprintf(stderr, "identifier \"%s\" is already defined\n", (char*)
34              $1);

```

```

32     $$ = new DeclStmt(new Id(se));
33     delete [] $1;
34 }
35 | ID ASSIGN InitVal {
36     SymbolEntry* se;
37     se = new IdentifierSymbolEntry(declType, $1, identifiers->getLevel())
38     ;
39     if (!identifiers->install($1, se))
40         fprintf(stderr, "identifier \"%s\" is already defined\n", (char*)
41             $1);
42     double val = $3->getValue();
43     if (declType->isInt() && $3->getType()->isFloat()) {
44         float temp = (float)val;
45         int temp1 = (int)temp;
46         val = (double)temp1;
47     }
48     ((IdentifierSymbolEntry*)se)->setValue(val);
49     $$ = new DeclStmt(new Id(se), $3);
50     delete [] $1;
51 }
52 | ID ArrayIndices ASSIGN {
53     SymbolEntry* se;
54     std::vector<int> vec;
55     ExprNode* temp = $2;
56     while (temp) {
57         vec.push_back(temp->getValue());
58         temp = (ExprNode*)(temp->getNext());
59     }
60     Type* type = declType;
61     Type* temp1;
62     for(auto it = vec.rbegin(); it != vec.rend(); it++) {
63         temp1 = new ArrayType(type, *it);
64         if (type->isArray())
65             ((ArrayType*)type)->setArrayType(temp1);
66         type = temp1;
67     }
68     arrayType = (ArrayType*)type;
69     idx = 0;
70     std::stack<InitValueListExpr*>().swap(stk);
71     se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
72     $<se>$ = se;
73     arrayValue = new double[arrayType->getSize()];
74 }
75 InitVal {
76     ((IdentifierSymbolEntry*)$<se>4)->setArrayValue(arrayValue);
77     if (((InitValueListExpr*)$5)->isEmpty())
78         ((IdentifierSymbolEntry*)$<se>4)->setAllZero();
79     if (!identifiers->install($1, $<se>4))

```

```

78         fprintf(stderr, "identifier \"%s\" is already defined\n", (char*)
           $1);
79     $$ = new DeclStmt(new Id($<se>4), $5);
80     delete [] $1;
81 }
82 ;

```

## 2. 数组定义

该部分用于处理数组索引的产生式规则，用于解析数组的维度信息等。产生式较为简单，分为一维数组与多维数组两种情况进行处理。对于多维数组，则使用链表将其串联在一起，即代码中的 \$1->setNext(\$3) 部分。

### 数组定义

```

1 ArrayIndices
2 : LBRACKET ConstExp RBRACKET {
3     $$ = $2;
4 }
5 | ArrayIndices LBRACKET ConstExp RBRACKET {
6     $$ = $1;
7     $1->setNext($3);
8 }
9 ;

```

## (三) 抽象语法树

语法分析程序实现的本质是得到语法树的叶节点的相关属性。在构建抽象语法树时，使用自底向上构造法，根据子节点的属性确定父节点。程序从 Node 派生出若干继承子类，比如 ExprNode、StmtNode 等，并又在此基础上二次派生出如 ExprStmt、SeqNode 等类，用于准备标定叶节点属性。在每次派生时，都会根据其子类的相关情况添加新的成员变量或是成员函数，并重写虚函数，用于最终语法树的构建。

### FuncCallExp 类定义

```

1 class FuncCallExp : public ExprNode
2 {
3     private:
4         ExprNode *param;
5
6     public:
7         FuncCallExp(SymbolEntry *se, ExprNode *param = nullptr): ExprNode(se),
           param(param) {};
8         void output(int level);
9         void typeCheck();
10        void genCode();
11 };

```



利用编写的相关程序，对给定的 SysY 编程语言进行语法分析。以给定示例代码为例，我们得到下面的语法树，可以看到各项数据都能够正确输出。

```
1  program
2    DeclStmt
3      Id name: a scope: 0    type: int
4    FunctionDefine function name: main type: int()
5      CompoundStmt
6        DeclStmt
7          Id name: a scope: 2    type: int
8        AssignStmt
9          Id name: a scope: 2    type: int
10         BinaryExpr op: add
11           IntegerLiteral value: 1    type: int
12           IntegerLiteral value: 2    type: int
13        IfStmt
14          BinaryExpr op: less
15            Id name: a scope: 2    type: int
16            IntegerLiteral value: 5    type: int
17          ReturnStmt
18            IntegerLiteral value: 1    type: int
19        ReturnStmt
20          IntegerLiteral value: 0    type: int
21
```

图 3: 语法分析结果

## 五、 语义分析（类型检查）

在语义分析（类型检查）部分，我主要完成对数组及运算对象等部分的检查。该部分主要对前面生成的语法树进行遍历，从开始符号开始递归调用其 `typecheck()` 函数，对整个语法树进行类型检查，如果检查到错误则输出相应的日志以帮助程序员进行代码检查。

### （一） 数组的检查

在赋值运算中，我们要求等号左边不能是数组，即数组类型不支持赋值操作，因为数组通常被视为一组元素，而不是一个可直接赋值的单一值。

#### 数组的检查

```

1 AssignStmt::AssignStmt(ExprNode *lval, ExprNode *expr) : lval(lval), expr(
    expr)
2 {
3     Type *type = ((Id *)lval)->getType();
4     Type *exprType = expr->getType();
5     SymbolEntry *se = lval->getSymbolEntry();
6     bool flag = true;
7
8     ...
9
10    else if (type->isArray())
11    {
12        fprintf(stderr, "array type '%s' is not assignable\n", type->toStr
            ().c_str());
13        flag = false;
14    }
15
16    ...

```

在获取标识符类型时，如果标识符为数组类型，则需对其进行索引的类型检查，如果索引表达式为空，则说明其不为数组。

#### 数组的检查

```

1 Type *Id::getType()
2 {
3     SymbolEntry *se = this->getSymbolEntry();
4     if (!se)
5     {
6         return TypeSystem::voidType;
7     }
8     Type *type = se->getType();
9     if (!arrIdx)
10    {
11        return type;
12    }
13    else if (!type->isArray())
14    {

```

```

15     fprintf(stderr, "subscripted value is not an array\n");
16     return TypeSystem::voidType;
17 }
18 else
19 {
20     ArrayType *temp1 = (ArrayType *)type;
21     ExprNode *temp2 = arrIdx;
22     while (!temp1->getElementType()->isInt() && !temp1->getElementType()
23            ->isFloat())
24     {
25         if (!temp2)
26         {
27             return temp1;
28         }
29         temp2 = (ExprNode *) (temp2->getNext());
30         temp1 = (ArrayType *) (temp1->getElementType());
31     }
32     if (!temp2)
33     {
34         fprintf(stderr, "subscripted value is not an array\n");
35         return temp1;
36     }
37     else if (temp2->getNext())
38     {
39         fprintf(stderr, "subscripted value is not an array\n");
40         return TypeSystem::voidType;
41     }
42     return temp1->getElementType();
43 }
44 }

```

## (二) 运算对象的检查

对于运算表达式而言, 我们要求其数据类型不为 void, 因此我们在进行表达式翻译时, 需检查运算表达式的运算对象的数据类型, 确保其有意义。此处以一元运算为例给出代码实现:

### 运算对象的检查

```

1 UnaryExpr::UnaryExpr(SymbolEntry *se, int op, ExprNode *expr) : ExprNode(se,
  UNARYEXPR), op(op), expr(expr)
2 {
3     std::string op_str = op == UnaryExpr::NOT ? "!" : "-";
4     if (expr->getType()->isVoid())
5     {
6         fprintf(stderr, "invalid operand of type \'void\' to unary \'opeartor
  %s\' \n", op_str.c_str());
7     }
8     if (op == UnaryExpr::NOT)

```

```

9      {
10         type = TypeSystem::intType;
11         dst = new Operand(se);
12         if (expr->isUnaryExpr())
13         {
14             UnaryExpr *ue = (UnaryExpr *)expr;
15             if (ue->getOp() == UnaryExpr::NOT)
16             {
17                 if (ue->getType() == TypeSystem::intType)
18                 {
19                     ue->setType(TypeSystem::boolType);
20                 }
21             }
22         }
23     }
24     else if (op == UnaryExpr::SUB)
25     {
26         type = expr->getType();
27         dst = new Operand(se);
28         if (expr->isUnaryExpr())
29         {
30             UnaryExpr *ue = (UnaryExpr *)expr;
31             if (ue->getOp() == UnaryExpr::NOT)
32             {
33                 if (ue->getType() == TypeSystem::intType)
34                 {
35                     ue->setType(TypeSystem::boolType);
36                 }
37             }
38         }
39     }
40 };

```

利用编写的相关程序，对给定的 SysY 编程语言进行语义分析。以给定示例代码为例，可以看到其提示的错误信息。

```

1  identifier "sum" is undefined
2  cannot initialize a variable of type 'void' with an rvalue of type 'i32'
3  identifier "sum" is undefined
4  cannot initialize a variable of type 'void' with an rvalue of type 'i32'
5  identifier "sum" is undefined
6  retValue or its symbol entry error

```

图 4: 语义分析结果

## 六、 中间代码生成

在中间代码生成部分，我主要负责运算语句、条件语句的指令生成，以及类型转换、数组的相关处理。该部分主要思路即为对语法树进行遍历（与类型检查同时进行），遍历时根据结点的属性信息生成各个结点的中间代码，遍历完成之后即可得到整个程序的中间代码。

### （一） AST::genCode

首先对于整个 Ast 语法树定义一个 genCode 函数，用于根节点中间代码的生成，进而递归调用其他节点的 genCode 函数，从而生成整个程序的中间代码。

Ast::genCode

```

1 void Ast::genCode(Unit *unit)
2 {
3     IRBuilder *builder = new IRBuilder(unit);
4     Node::setIRBuilder(builder);
5     root->genCode();
6 }
```

### （二） 二元运算

运算表达式的中间代码生成由我负责，此处我以二元运算为例进行简要说明。

#### 1. 逻辑短路机制

在本次实验中，我们实现了逻辑短路机制的处理，此处以与运算为例进行说明。对于与运算而言，只有当两个运算数都为真时，运算结果才为真；当第一个运算数为假时，第二个运算数不予分析，直接将整个表达式判定为假。

首先创建一个新的基本块 trueBB 用于存放逻辑 AND 操作的结果，然后生成左侧表达式 expr1 的中间代码，对左侧表达式的 true 列表进行回填，将其跳转指令的目标基本块设置为 trueBB；将当前插入基本块设置为 trueBB，即接下来生成的中间代码将插入到 trueBB 中。生成右侧表达式 expr2 的中间代码，将 true 列表设为右侧表达式 expr2 的 true 列表，最后将 false 列表设为左侧表达式 expr1 的 false 列表和右侧表达式 expr2 的 false 列表的合并。这样通过回填操作，我们就实现了逻辑短路的效果，避免不必要的计算，以此提高程序执行效率。

逻辑短路机制

```

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

#### 2. 类型转换

在本次实验中，难免会遇到运算时两边运算对象的类型不相同的问题，此处就以二元运算为例进行说明。

如果 `expr1` 的类型是整数且大小为 32 位, 那么创建一个新的 `ImplicitCastExpr` 对象 `temp`, 以将 `expr1` 进行整数类型的隐式转换。否则, 如果 `expr1` 的类型是浮点数, 创建一个浮点数常量 0, 然后生成一个临时的布尔类型变量 `temp`, 接着使用 `BinaryExpr` 创建一个比较表达式, 检查 `expr1` 是否不等于 0 (`BinaryExpr::NOTEQUAL`)。最后, 将新创建的比较表达式赋值给 `expr1`。

#### 类型转换

```

1  if (expr1->getType()->isInt() && expr1->getType()->getSize() == 32)
2  {
3      ImplicitCastExpr *temp = new ImplicitCastExpr(expr1);
4      this->expr1 = temp;
5  }
6  else if (expr1->getType()->isFloat())
7  {
8      SymbolEntry *zero = new ConstantSymbolEntry(TypeSystem::floatType, 0);
9      SymbolEntry *temp = new TemporarySymbolEntry(TypeSystem::boolType,
10             SymbolTable::getLabel());
11      BinaryExpr *cmpZero = new BinaryExpr(temp, BinaryExpr::NOTEQUAL, expr1,
12             new Constant(zero));
13      this->expr1 = cmpZero;
14  }
15  if (expr2->getType()->isInt() && expr2->getType()->getSize() == 32)
16  {
17      ImplicitCastExpr *temp = new ImplicitCastExpr(expr2);
18      this->expr2 = temp;
19  }
20  else if (expr2->getType()->isFloat())
21  {
22      SymbolEntry *zero = new ConstantSymbolEntry(TypeSystem::floatType, 0);
23      SymbolEntry *temp = new TemporarySymbolEntry(TypeSystem::boolType,
24             SymbolTable::getLabel());
25      BinaryExpr *cmpZero = new BinaryExpr(temp, BinaryExpr::NOTEQUAL, expr2,
26             new Constant(zero));
27      this->expr2 = cmpZero;
28  }

```

为了方便进行类型转换, 我们定义了 `ImplicitCastExpr` 类。根据 `ImplicitCastExpr` 节点的类型, 生成适当的中间代码, 执行类型转换的操作。

如果目标类型是 `boolType`, 则创建三个基本块 `trueBB`、`tempbb` 和 `falseBB`。通过比较 `expr` 是否不等于零, 生成条件跳转指令。根据条件跳转的结果, 更新 `trueList` 和 `falseList`; 如果目标类型是整数类型, 生成 `FptosiInstruction` 指令, 将浮点数转换为整数; 如果目标类型是浮点数类型, 生成 `SitofpInstruction` 指令, 将整数转换为浮点数。

#### `ImplicitCastExpr`

```

1  void ImplicitCastExpr::genCode()
2  {
3      expr->genCode();
4      BasicBlock *bb = builder->getInsertBB();

```

```

5
6     if (type == TypeSystem::boolType)
7     {
8         Function *func = bb->getParent();
9         BasicBlock *trueBB = new BasicBlock(func);
10        BasicBlock *tempbb = new BasicBlock(func);
11        BasicBlock *falseBB = new BasicBlock(func);
12
13        new CmpInstruction(CmpInstruction::NE, this->dst, this->expr->
            getOperand(), new Operand(new ConstantSymbolEntry(TypeSystem::
            intType, 0)), bb);
14        this->trueList().push_back(new CondBrInstruction(trueBB, tempbb, this
            ->dst, bb));
15        this->falseList().push_back(new UncondBrInstruction(falseBB, tempbb))
            ;
16    }
17    else if (type->isInt())
18    {
19        new FptosiInstruction(dst, this->expr->getOperand(), bb);
20    }
21    else if (type->isFloat())
22    {
23        new SitofpInstruction(dst, this->expr->getOperand(), bb);
24    }
25    else
26    {
27        assert(false);
28    }
29 }

```

### (三) 声明语句

在本次实验中，我们实现了对声明语句的中间代码生成，此处以全局变量、局部变量的声明为例进行说明。

#### 1. 全局变量

在处理全局变量声明时，为该全局变量创建一个符号表项，表示其地址。这个地址符号表项被插入到全局符号表和模块符号表中，以便在后续代码生成过程中能够正确地引用全局变量的地址。

##### 全局变量

```

1 if (se->isGlobal())
2 {
3     Operand *addr;
4     SymbolEntry *addr_se;
5     addr_se = new IdentifierSymbolEntry(*se);
6     addr_se->setType(new PointerType(se->getType()));

```

```

7     addr = new Operand(addr_se);
8     se->setAddr(addr);
9     unit.insertGlobal(se);
10    mUnit.insertGlobal(se);
11 }

```

## 2. 局部变量

在处理局部变量时，为局部变量分配地址空间，并根据初始化表达式生成相应的代码，确保局部变量在运行时能够正确地访问。最后递归执行，处理下一个语句的中间代码。

对于初始化列表表达式，则递归地予以处理。如果当前节点是初始化列表表达式 (InitValueListExpr)，将其入栈，更新索引，然后移动到下一层的初始化列表；如果当前节点不是初始化列表表达式，表示处理到了基本类型的元素，调用 temp->genCode() 生成相应的中间代码。使用 while (true) 循环生成 GEP (GetElementPtr) 指令，计算数组元素的地址。每次迭代处理一个维度，生成一个 GEP 指令。使用 new StoreInstruction 生成 Store 指令，将初始化的值存储到数组元素的地址。

### 局部变量

```

1  else if (se->isLocal())
2  {
3      Function *func = builder->getInsertBB()->getParent();
4      BasicBlock *entry = func->getEntry();
5      Instruction *alloca;
6      Operand *addr;
7      SymbolEntry *addr_se;
8      Type *type;
9      type = new PointerType(se->getType());
10     addr_se = new TemporarySymbolEntry(type, SymbolTable::getLabel());
11     addr = new Operand(addr_se);
12     alloca = new AllocaInstruction(addr, se);
13     entry->insertFront(alloca);
14     Operand *temp = nullptr;
15     if (se->isParam())
16     {
17         temp = se->getAddr();
18     }
19     se->setAddr(addr);
20     if (expr)
21     {
22         if (expr->isInitValueListExpr())
23         {
24             Operand *init = nullptr;
25             BasicBlock *bb = builder->getInsertBB();
26             ExprNode *temp = expr;
27             std::stack<ExprNode*> stk;
28             std::vector<int> idx;
29             idx.push_back(0);

```



```

30     while (temp)
31     {
32         if (temp->isInitValueListExpr())
33         {
34             stk.push(temp);
35             idx.push_back(0);
36             temp = ((InitValueListExpr *)temp)->getExpr();
37             continue;
38         }
39         else
40         {
41             temp->genCode();
42             Type *type = ((ArrayType *) (se->getType()))->
43                 getElementType();
44             Operand *tempSrc = addr;
45             Operand *tempDst;
46             Operand *index;
47             bool flag = true;
48             int i = 1;
49             while (true)
50             {
51                 tempDst = new Operand(new TemporarySymbolEntry(new
52                     PointerType(type), SymbolTable::getLabel()));
53                 index = (new Constant(new ConstantSymbolEntry(
54                     TypeSystem::intType, idx[i++])))>getOperand();
55                 auto gep = new GepInstruction(tempDst, tempSrc, index
56                     , bb);
57                 gep->setInit(init);
58                 if (flag)
59                 {
60                     gep->setFirst();
61                     flag = false;
62                 }
63                 if (type == TypeSystem::intType || type == TypeSystem
64                     ::constIntType || type == TypeSystem::floatType
65                     || type == TypeSystem::constFloatType)
66                 {
67                     gep->setLast();
68                     init = tempDst;
69                     break;
70                 }
71                 type = ((ArrayType *)type)->getElementType();
72                 tempSrc = tempDst;
73             }
74             new StoreInstruction(tempDst, temp->getOperand(), bb);
75         }
76     }
77     while (true)
78     {

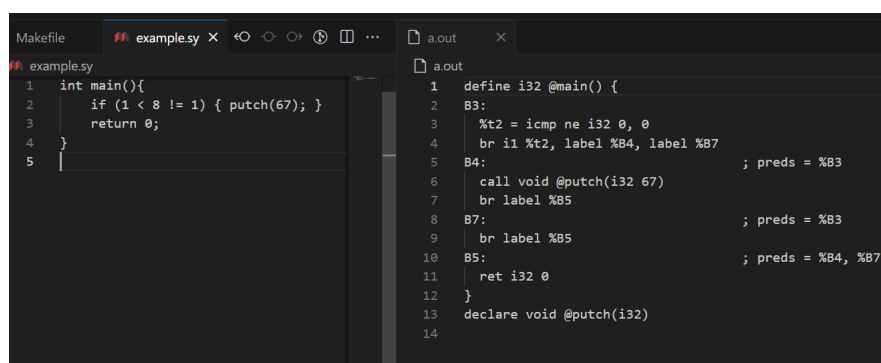
```

```

72         if (temp->getNext())
73         {
74             temp = (ExprNode *) (temp->getNext());
75             idx[idx.size() - 1]++;
76             break;
77         }
78         else
79         {
80             temp = stk.top();
81             stk.pop();
82             idx.pop_back();
83             if (stk.empty())
84             {
85                 break;
86             }
87         }
88     }
89     if (stk.empty())
90     {
91         break;
92     }
93 }
94 }
95 else
96 {
97     BasicBlock *bb = builder->getInsertBB();
98     expr->genCode();
99     Operand *src = expr->getOperand();
100    new StoreInstruction(addr, src, bb);
101 }
102 }
103 }

```

利用编写的相关程序，对给定的 SysY 编程语言生成中间代码。以给定示例代码为例，可以看到其生成的中间代码。



```

Makefile  example.sy  a.out
1  int main(){
2      if (1 < 8 != 1) { putch(67); }
3      return 0;
4  }
5

1  define i32 @main() {
2      B3:
3          %t2 = icmp ne i32 0, 0
4          br i1 %t2, label %B4, label %B7
5      B4:
6          call void @putch(i32 67)
7          br label %B5
8      B7:
9          br label %B5
10     B5:
11         ret i32 0
12 }
13 declare void @putch(i32)
14

```

图 5: 中间代码生成结果

## 七、 目标代码生成

在目标代码生成部分，我主要负责基本汇编指令、数组及全局变量等的处理。下面将选取几个示例对该部分的工作进行解释。

### (一) 基本汇编指令

此处以关系运算指令为例进行说明。

定义了一个 CmpMInstruction 类，用于表示关系运算指令相关内容。CmpMInstruction 构造函数第一个参数 p 为指令所插入基本块并赋给成员函数 parent，第二、三个参数为比较指令的两个操作数 src1、src2，将这两个操作数依次 push\_back 到 vector 类型的 use\_list 中

#### CmpMInstruction

```

1  class CmpMInstruction : public MachineInstruction
2  {
3  public:
4      enum opType
5      {
6          CMP
7      };
8      CmpMInstruction(MachineBlock *p, MachineOperand *src1, MachineOperand *
          src2, int cond = MachineInstruction::NONE);
9      void output();
10 };
11
12 CmpMInstruction::CmpMInstruction(MachineBlock *p, MachineOperand *src1,
    MachineOperand *src2, int cond)
13 {
14     this->parent = p;
15     this->type = MachineInstruction::CMP;
16     this->op = -1;
17     this->cond = cond;
18     p->setCmpCond(cond);
19     this->use_list.push_back(src1);
20     this->use_list.push_back(src2);
21     src1->setParent(this);
22     src2->setParent(this);
23 }
```

CmpMInstruction 输出函数 output() 先输出 cmp，再输出两个比较操作数 def\_list[0] 和 def\_list[1]

#### CmpMInstruction::output()

```

1  void CmpMInstruction::output()
2  {
3      fprintf(yyout, "\\tmp ");
4      this->use_list[0]->output();
5      fprintf(yyout, ", ");
```

```

6     this->use_list[1]->output();
7     fprintf(yyout, "\n");
8 }

```

定义了一个 CmpInstruction 类，表示并执行比较操作，例如，检查两个操作数是否相等、大小关系等。构造函数接收操作码 (opcode)、目标操作数 (dst)、源操作数 1 (src1)、源操作数 2 (src2) 和插入基本块 (insert\_bb) 作为参数。

#### CmpInstruction

```

1 class CmpInstruction : public Instruction
2 {
3 public:
4     CmpInstruction(unsigned opcode, Operand *dst, Operand *src1, Operand *
5         src2, BasicBlock *insert_bb = nullptr);
6     ~CmpInstruction();
7     void output() const;
8     void genMachineCode(AsmBuilder *);
9     enum
10    {
11        E,
12        NE,
13        L,
14        LE,
15        G,
16        GE
17    };
18 };

```

其 genMachineCode 函数用于将中间表示转换成目标代码表示的比较指令。函数通过调用 AsmBuilder 对象的方法生成目标机器码。

如果比较的操作数是浮点数类型，那么首先生成浮点数操作数的机器码。这涉及到加载浮点数值或者将寄存器中的值移动到目标寄存器。然后，根据比较操作符的类型（如 <, <=, >, >=, ==, !=），选择相应的浮点数比较函数（如 @\_\_aeabi\_fcmplt、@\_\_aeabi\_fcmlpe 等）。使用 BranchMInstruction 生成无条件分支到比较函数，然后使用 CmpMInstruction 生成对比较结果的处理，设置目标寄存器的值。

如果比较的操作数是整数类型，那么首先生成整数操作数的机器码。这包括加载整数值或者将寄存器中的值移动到目标寄存器。然后，使用 CmpMInstruction 生成整数比较的机器码。根据比较操作符的类型，使用 MovMInstruction 生成条件移动指令，将对比较结果映射为 0 或 1，并设置目标寄存器的值。

#### CmpInstruction::genMachineCode()

```

1 void CmpInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     auto src1 = genMachineOperand(operands[1]);
5     auto src2 = genMachineOperand(operands[2]);
6
7     if (operands[1]->getType()->isFloat())

```

```

8 {
9     MachineInstruction *cur_inst;
10
11     auto operand1 = genMachineReg(0);
12     auto operand2 = genMachineReg(1);
13
14     if (src1->isImm())
15     {
16         cur_inst = new LoadMInstruction(cur_block, operand1, src1);
17     }
18     else
19     {
20         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
21             operand1, src1);
22     }
23     cur_block->InsertInst(cur_inst);
24
25     if (src2->isImm())
26     {
27         cur_inst = new LoadMInstruction(cur_block, operand2, src2);
28     }
29     else
30     {
31         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
32             operand2, src2);
33     }
34     cur_block->InsertInst(cur_inst);
35
36     std::string abi_str;
37
38     int cmp_imm = 1;
39
40     switch (opcode)
41     {
42     case CmpInstruction::L:
43         abi_str = "@__aeabi_fcmplt";
44         break;
45     case CmpInstruction::LE:
46         abi_str = "@__aeabi_fcmple";
47         break;
48     case CmpInstruction::G:
49         abi_str = "@__aeabi_fcmpgt";
50         break;
51     case CmpInstruction::GE:
52         abi_str = "@__aeabi_fcmpge";
53         break;
54     case CmpInstruction::E:
55         abi_str = "@__aeabi_fcmpeq";
56     }
57 }

```

```

54         break;
55     case CmpInstruction::NE:
56         abi_str = "@__aeabi_fcmpeq";
57         cmp_imm = 0;
58         break;
59     default:
60         // error
61         break;
62 }
63
64 auto label = new MachineOperand(abi_str);
65 cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL,
66     label);
67 cur_block->InsertInst(cur_inst);
68
69 auto r0 = new MachineOperand(MachineOperand::REG, 0);
70 auto dst = genMachineOperand(operands[0]);
71
72 auto bool_operand = genMachineImm(cmp_imm);
73 cur_inst = new CmpMInstruction(cur_block, r0, bool_operand,
74     CmpMInstruction::EQ);
75 cur_block->InsertInst(cur_inst);
76
77 cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
78     genMachineImm(0), CmpInstruction::NE);
79 cur_block->InsertInst(cur_inst);
80 cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
81     genMachineImm(1), CmpInstruction::E);
82 cur_block->InsertInst(cur_inst);
83
84 return;
85 }
86
87 MachineInstruction *cur_inst = nullptr;
88 if (src1->isImm())
89 {
90     auto internal_reg = genMachineVReg();
91     cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
92     cur_block->InsertInst(cur_inst);
93     src1 = new MachineOperand(*internal_reg);
94 }
95 if (src2->isImm())
96 {
97     auto internal_reg = genMachineVReg();
98     cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
99     cur_block->InsertInst(cur_inst);
100     src2 = new MachineOperand(*internal_reg);
101 }

```

```

98     cur_inst = new CmpMInstruction(cur_block, src1, src2, opcode);
99     cur_block->InsertInst(cur_inst);
100     if (opcode >= CmpInstruction::L && opcode <= CmpInstruction::GE)
101     {
102         auto dst = genMachineOperand(operands[0]);
103         auto trueOperand = genMachineImm(1);
104         auto falseOperand = genMachineImm(0);
105         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
106             trueOperand, opcode);
107         cur_block->InsertInst(cur_inst);
108         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
109             falseOperand, 7 - opcode);
110         cur_block->InsertInst(cur_inst);
111     }
112 }

```

## (二) 数组

数组的目标代码生成函数较为复杂。idx 存储的是行偏移，size1 存储的是行维度，通过 MUL 指令使得 idx 与 size1 相乘得出的值为该索引相对于数组首地址的行偏移，将 operands[1] 操作数（即 arr 变量）和 off 变量相加的结果存储在 dst 变量中，相加结果即为索引在栈中偏移，将 fp 寄存器和 addr 变量相加的结果存储在 dst 变量中，相加结果为该索引在栈中地址。其余部分细节已在代码中注释部分予以添加。

```

// GepInstruction
1 // 第一维；最后一维；中间若干维；参数
2 void GepInstruction::genMachineCode(AsmBuilder *builder)
3 {
4     auto cur_block = builder->getBlock();
5     MachineInstruction *cur_inst;
6     auto dst = genMachineOperand(operands[0]); // 目的操作数
7     auto idx = genMachineOperand(operands[2]); // 维度
8     // 是否初始化,比如a[3],init就表示a的地址
9     if (init)
10     {
11         // 最后一个维度a[2][3], 数组等价于数组的数组, 不断拆解下去肯定有一个
12         // 维度是int类型
13         // 如果是最后一个维度的话, 我们已经有初地址, 那么不断+4+4就可以寻址
14         // 了
15         if (last)
16         {
17             // 插入一个新的 BinaryMInstruction 指令,
18             // 该指令将 dst 操作数加上 genMachineOperand(init) 操作数再加上
19             // genMachineImm(4) 操作数。并且返回。
20             auto base = genMachineOperand(init);
21             cur_inst = new BinaryMInstruction(
22                 cur_block, BinaryMInstruction::ADD, dst, base, genMachineImm
23                 (4));

```

```

20         cur_block->InsertInst(cur_inst);
21     }
22     return;
23 }
24 MachineOperand *base = nullptr;
25 int size;
26 // idx->索引
27 // 若idx为常数, 那么加载它的值, 将其转为MachineOperand*类型
28 auto idx1 = genMachineVReg();
29 if (idx->isImm())
30 {
31     if (idx->getVal() < 255)
32     {
33         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
34             idx1, idx);
35         cur_block->InsertInst(cur_inst);
36     }
37     else
38     {
39         int off = idx->getVal();
40         if (Judge(off))
41             cur_block->InsertInst((new LoadMInstruction(cur_block, idx1,
42                 idx)));
43         else
44         {
45             cur_block->InsertInst(new MovMInstruction(cur_block,
46                 MovMInstruction::MOV, idx1, genMachineImm(off & 0xffff)));
47             ;
48             if (off & 0xffff00)
49                 cur_block->InsertInst(new BinaryMInstruction(cur_block,
50                     BinaryMInstruction::ADD, idx1, idx1, genMachineImm(
51                         off & 0xff0000)));
52             if (off & 0xff000000)
53                 cur_block->InsertInst(new BinaryMInstruction(cur_block,
54                     BinaryMInstruction::ADD, idx1, idx1, genMachineImm(
55                         off & 0xff000000)));
56         }
57     }
58     idx = new MachineOperand(*idx1);
59 }
60 if (paramFirst) // 该数组是否为参数[]
61 {
62     // 按字节算
63     size = ((PointerType *) (operands[1]->getType()))->getType()->getSize() / 8;
64 }
65 else // 不是参数 -> 变量
66 {

```



```

59 // 加载base
60 if (first) // 如果是数组的第一维的话
61 {
62     // 创建一个新的机器操作数base
63     base = genMachineVReg();
64     // 检查 operands[1] 操作数（变量的地址）的符号表条目是否是变量。
        如果是,
65     // 并且该符号表条目是全局变量, 则在当前块中插入一个新的
        LoadMInstruction 指令, 该指令将 operands[1] 操作数的值加载到
        base 变量中。
66     if (operands[1]->getEntry()->isVariable() &&
67         ((IdentifierSymbolEntry *) (operands[1]->getEntry()))->
            isGlobal()) // 全局变量
68     {
69         auto src = genMachineOperand(operands[1]);
70         cur_inst = new LoadMInstruction(cur_block, base, src);
71         cur_block->InsertInst(cur_inst);
72     }
73     else
74     {
75         // offset 首地址偏移（局部变量栈中偏移）->将偏移保存到base之中
76         int offset = ((TemporarySymbolEntry *) (operands[1]->getEntry
            ()))->getOffset();
77         if (offset > -255 && offset < 255)
78         {
79             cur_inst = new MovMInstruction(cur_block, MovMInstruction
                ::MOV, base, genMachineImm(offset));
80             cur_block->InsertInst(cur_inst);
81         }
82         else
83         {
84             // 把 offset 的值传到base里面去
85             if (Judge(offset))
86             {
87                 cur_inst = new LoadMInstruction(cur_block, base,
                    genMachineImm(offset));
88                 cur_block->InsertInst(cur_inst);
89             }
90             else
91             {
92                 // 先用mov把它的低16位放到寄存器里, 然后用add把它高16
                    位的两个字节加到寄存器里
93                 cur_inst = new MovMInstruction(cur_block,
                    MovMInstruction::MOV, base, genMachineImm(offset
                        & 0xffff));
94                 cur_block->InsertInst(cur_inst);
95                 // cur_inst = new LoadMInstruction(cur_block, base,
                    genMachineImm(offset));

```

```

96         if (offset & 0xffff00)
97         {
98             cur_inst = new BinaryMInstruction(cur_block,
100             BinaryMInstruction::ADD, base, base,
101             genMachineImm(offset & 0xff0000));
102             cur_block->InsertInst(cur_inst);
103         }
104         if (offset & 0xff000000)
105         {
106             cur_inst = new BinaryMInstruction(cur_block,
107             BinaryMInstruction::ADD, base, base,
108             genMachineImm(offset & 0xff000000));
109             cur_block->InsertInst(cur_inst);
110         }
111     }
112     }
113     }
114     ArrayType *type = (ArrayType *)(((PointerType *) (operands[1]->getType
115     ()))>getType());
116     size = type->getEType()->getSize() / 8;
117 }
118 // 使用常量size的值创建一个寄存器操作数size1, size保存一个维度大小
119 auto size1 = genMachineVReg();
120 if (size > -255 && size < 255)
121 {
122     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, size1
123     , genMachineImm(size));
124     cur_block->InsertInst(cur_inst);
125 }
126 else
127 {
128     if (Judge(size))
129     {
130         cur_inst = new LoadMInstruction(cur_block, size1, genMachineImm(
131         size));
132         cur_block->InsertInst(cur_inst);
133     }
134     else
135     {
136         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
137         size1, genMachineImm(size & 0xffff));
138         cur_block->InsertInst(cur_inst);
139         if (size & 0xffff00)
140         {
141             cur_block->InsertInst(new BinaryMInstruction(cur_block,
142             BinaryMInstruction::ADD, size1, size1, genMachineImm(size
143             & 0xff0000)));
144         }
145     }
146 }

```

```

134         cur_block->InsertInst(cur_inst);
135     }
136
137     if (size & 0xff000000)
138     {
139         cur_inst = new BinaryMInstruction(cur_block,
140             BinaryMInstruction::ADD, size1, size1, genMachineImm(size
141                 & 0xff000000));
142         cur_block->InsertInst(cur_inst);
143     }
144 }
145
146 // a[3][4], 如果访问a[2][3]->*(a+(2*4)+3)
147 auto off = genMachineVReg();
148 cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL, off
149     , idx, size1);
150 off = new MachineOperand(*off);
151 cur_block->InsertInst(cur_inst);
152
153 // 如果不是第一维或者参数的话
154 if (paramFirst || !first)
155 {
156     // arr等价于operands[1]等价于变量的地址
157     auto arr = genMachineOperand(operands[1]);
158     // 将 operands[1] 操作数 (即 arr 变量) 和 off 变量相加的结果存储在
159     dst 变量中。
160     cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
161         dst, arr, off);
162     cur_block->InsertInst(cur_inst);
163 }
164 // 第一维并且不是参数
165 else
166 {
167     // 创建一个新的机器操作数 addr, 在当前块中插入一个新的
168     // BinaryMInstruction 指令, 该指令将 base 变量和 off 变量相加的结果存
169     储在 addr变量中。
170     auto addr = genMachineVReg();
171     auto base1 = new MachineOperand(*base);
172     cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
173         addr, base1, off);
174     cur_block->InsertInst(cur_inst);
175     addr = new MachineOperand(*addr);
176     // 检查 operands[1] 操作数的符号表条目是否是变量且是全局变量,
177     // 如果是, 在当前块中插入一个新的 MovMInstruction 指令, 该指令将 addr
178     变量的值存储在 dst 变量中。
179     if (operands[1]->getEntry()->isVariable() &&((IdentifierSymbolEntry
180         *) (operands[1]->getEntry()))->isGlobal())

```

```

173     {
174         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
175                                     dst, addr);
176     }
177     else
178     {
179         // 如果不是, 则在当前块中插入一个新的 BinaryMInstruction 指令,
180         // 该指令将 fp 寄存器和 addr 变量相加的结果存储在 dst 变量中
181         auto fp = genMachineReg(11);
182         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::
183             ADD, dst, fp, addr);
184     }
185     cur_block->InsertInst(cur_inst);
186 }
187 }

```

### (三) 全局数据

对于全局数据, 主要分为全局变量与全局常量两种。在类型检查阶段, 我们已经对非数组的全局常量进行了常量替换操作, 因此对于非数组的全局常量已经不需要进行处理了。而对于数组的全局常量, 需要将其写入到 rdata 段中。对于全局变量需要将其写入到 data 段中。对于有初始值的变量, 还需要在全局声明的时候带上初始值, 这里需要注意的是, 浮点数初始值在全局声明的时候, 需要按照 32 位整数进行输出。对于没有初始值的全局数组, 需要给其加上 .comm 标签, 加上这个标签之后, 该数组的初始值被置为 0。

访问全局变量时, 首先生成目的寄存器 dst, 用于存储加载全局变量的值。生成两个虚拟寄存器 internal\_reg1 和 internal\_reg2, 其中 internal\_reg2 的初始值与 internal\_reg1 相同, 生成源操作数 src, 该操作数对应于 operands[1]。生成一个 LOAD 指令, 将全局变量的值加载到 internal\_reg1 中。生成另一个 LOAD 指令, 将 internal\_reg2 中的值加载到目的寄存器 dst 中。这样, 通过这两个 LOAD 指令, 成功将全局变量的值加载到目的寄存器中, 以便后续对该值的使用。

#### LoadInstruction

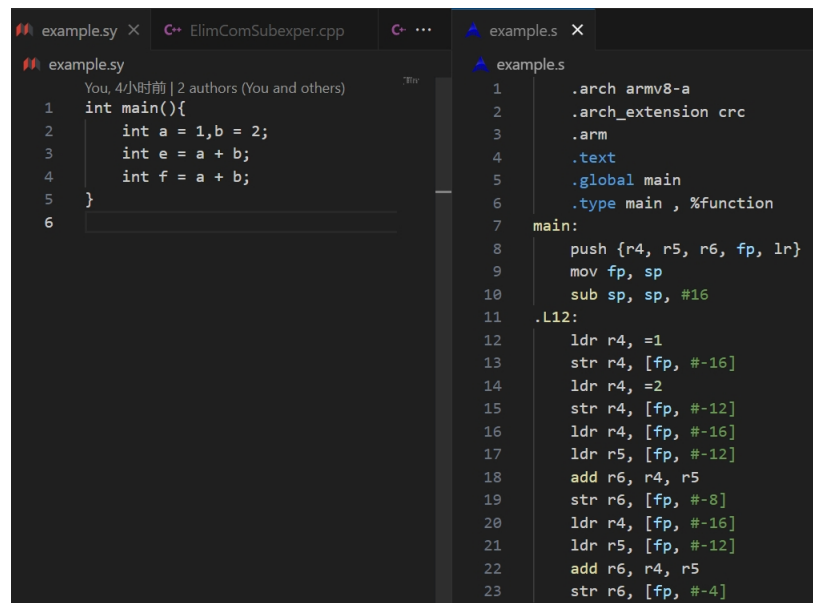
```

1 void LoadInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     MachineInstruction *cur_inst = nullptr;
5     if (operands[1]->getEntry()->isVariable() && dynamic_cast<
6         IdentifierSymbolEntry *>(operands[1]->getEntry()->isGlobal())// 全局
7         变量
8     {
9         auto dst = genMachineOperand(operands[0]);
10        auto internal_reg1 = genMachineVReg();
11        auto internal_reg2 = new MachineOperand(*internal_reg1);
12        auto src = genMachineOperand(operands[1]);
13        cur_inst = new LoadMInstruction(cur_block, internal_reg1, src);
14        cur_block->InsertInst(cur_inst);
15        cur_inst = new LoadMInstruction(cur_block, dst, internal_reg2);

```

```
14     cur_block->InsertInst ( cur_inst );
15 }
16
17 ...
18
19 }
```

利用编写的相关程序，对给定的 SysY 编程语言生成目标代码。以给定示例代码为例，可以看到生成的中间代码。



The image shows a code editor with two panels. The left panel displays a SysY program in `example.sy`, and the right panel displays the generated ARM assembly code in `example.s`.

**example.sy:**

```
1 int main(){
2     int a = 1, b = 2;
3     int e = a + b;
4     int f = a + b;
5 }
6
```

**example.s:**

```
1 .arch armv8-a
2 .arch_extension crc
3 .arm
4 .text
5 .global main
6 .type main , %function
7 main:
8     push {r4, r5, r6, fp, lr}
9     mov fp, sp
10    sub sp, sp, #16
11 .L12:
12    ldr r4, =1
13    str r4, [fp, #-16]
14    ldr r4, =2
15    str r4, [fp, #-12]
16    ldr r4, [fp, #-16]
17    ldr r5, [fp, #-12]
18    add r6, r4, r5
19    str r6, [fp, #-8]
20    ldr r4, [fp, #-16]
21    ldr r5, [fp, #-12]
22    add r6, r4, r5
23    str r6, [fp, #-4]
```

图 6: 目标代码生成结果

## 八、 代码优化

在本次实验中，我们实现了死代码删除以及公共子表达式删除两种优化方式，在此以我主要负责的公共子表达式删除为例进行说明。

首先在头文件中定义了两个数据结构：aeb 与 ElimComSubexpr，分别用于对指令的删改与对整个编译结果的优化。

数据结构

```

1 struct aeb
2 {
3     BinaryInstruction *inst;
4     SymbolEntry *opd1;
5     int opr;
6     SymbolEntry *opd2;
7     Operand *tmp = nullptr;
8     bool operator<(aeb a) const
9     {
10         return opr < a.opr;
11     }
12     bool operator==(aeb a) const
13     {
14         return inst == a.inst && opr == a.opr && opd1 == a.opd1 && opd2 == a.
            opd2;
15     }
16 };
17
18 class ElimComSubexpr
19 {
20     Unit *unit;
21
22 public:
23     ElimComSubexpr(Unit *unit) : unit(unit) {};
24     ~ElimComSubexpr();
25     void elim_cse();
26     void local_elim_cse(BasicBlock *bb, vector<struct aeb> AEB);
27 };

```

接着编写了 delAEB、vec\_intersection、vec\_union、vintersection、local\_elim\_cse 几个辅助函数对基本块进行处理。

### 1. delAEB

该函数用于从值编号表达式列表 AEB 中删除与给定符号 sym 相关的项。

delAEB

```

1 void delAEB(SymbolEntry *sym, VAEB AEB)
2 {
3     int len = AEB.size();
4     for (int i = 0; i < len; i++)

```

```

5   {
6       if (sym == AEB[i].opd1 || sym == AEB[i].opd2)
7       {
8           for (int j = i; j < len - 1; j++)
9           {
10              AEB[j] = AEB[j + 1];
11          }
12          len--;
13      }
14  }
15  }

```

## 2. vec\_intersection

公共子表达式是指在控制流图中不同的基本块中具有相同操作符、相同操作数的二元指令。为了进行局部删除，需要维护每个基本块的值编号表达式列表。这就涉及到对这些列表进行交集、并集等操作。而该函数计算两个值编号表达式列表 `v1` 和 `v2` 的交集，并返回结果。

```

                                vec_intersection
1  VAEB vec_intersection(VAEB v1, VAEB v2)
2  {
3      VAEB v11, v22, res;
4      int len = min(v1.size(), v2.size());
5      if (v1.size() > v2.size())
6      {
7          v11 = v2;
8          v22 = v1;
9      }
10     else
11     {
12         v11 = v1;
13         v22 = v2;
14     }
15     VAEB::iterator it;
16     for (int i = 0; i < len; i++)
17     {
18         it = find(v22.begin(), v22.end(), v11[i]);
19         if (it != v22.end())
20         {
21             res.push_back(v11[i]);
22         }
23     }
24     return res;
25 }

```

## 3. vec\_union

该函数计算两个值编号表达式列表 `v1` 和 `v2` 的并集，并返回结果。

## vec\_union

```

1 VAEB vec_union(VAEB v1, VAEB v2)
2 {
3     VAEB res(v1);
4     VAEB::iterator it;
5     for (unsigned i = 0; i < v2.size(); i++)
6     {
7         it = find(v1.begin(), v1.end(), v2[i]);
8         if (it == v1.end())
9         {
10             res.push_back(v2[i]);
11         }
12     }
13     return res;
14 }

```

## 4. vintersection

该函数用于计算基本块 bb 的入口值编号表达式列表的交集，其目的是通过迭代基本块的前驱列表，检查每个前驱基本块是否已经被访问过（ste[no] 表示是否已经访问），如果已经访问过，则将当前前驱基本块的出口值编号表达式列表与结果列表进行交集运算。最终，函数返回了基本块的入口值编号表达式列表的交集，从而帮助处理控制流图中的信息流。

## vintersection

```

1 VAEB vintersection(BasicBlock *bb)
2 {
3     VAEB res;
4     bb_iterator iter = bb->pred_begin();
5     bb_iterator end = bb->pred_end();
6     while (iter != end)
7     {
8         int no = (*iter)->getNo();
9         if (ste[no])
10        {
11            res = vec_intersection(res, AEout[no]);
12        }
13        iter++;
14    }
15    return res;
16 }

```

## 5. local\_elim\_cse

该函数实现了在一个基本块中进行局部公共子表达式删除的操作，其主要目的是在一个基本块中进行局部的公共子表达式删除。根据基本块中的每个二元指令，检查是否存在相同的子表达式，如果存在，则进行替换操作，否则将当前子表达式添加到列表中。最后，更新当前基本块的出口值编号表达式列表 AEout 和清空当前基本块的局部值编号表达式列表 AEB。



## local\_elim\_cse

```

1 void ElimComSubexpr::local_elim_cse(BasicBlock *bb, VAEB AEB)
2 {
3     int no = bb->getNo();
4     for (auto iter = bb->begin(); iter != bb->end(); iter = iter->getNext())
5         //遍历
6     {
7         vector<Operand *> operands(iter->getOperands());
8         if (iter->isBin())
9         {
10             int op = ((BinaryInstruction *)iter)->getOp();
11             Instruction *p = iter;
12             bool found = false;
13
14             SymbolEntry *sym1 = operands[1]->getEntry(), *sym2 = operands
15                 [2]->getEntry();
16             if (((IdentifierSymbolEntry *)sym1)->isSysy() || ((
17                 IdentifierSymbolEntry *)sym2)->isSysy())
18             {
19                 continue;
20             }
21             if (!sym1->isConstant())
22             {
23                 auto iter_def = operands[1]->getDef();
24                 if (iter_def)
25                 {
26                     vector<Operand *> def_operands(iter_def->getOperands());
27                     sym1 = def_operands[1]->getEntry();
28                 }
29             }
30             if (!sym2->isConstant())
31             {
32                 auto iter_def = operands[2]->getDef();
33                 if (iter_def)
34                 {
35                     vector<Operand *> def_operands(iter_def->getOperands());
36                     sym2 = def_operands[1]->getEntry();
37                 }
38             }
39             int len = AEB.size();
40             int i = 0;
41             for (; i < len; i++)
42             {
43                 if (op == AEB[i].opr && sym1->toStr() == AEB[i].opd1->toStr()
44                     && sym2->toStr() == AEB[i].opd2->toStr())
45                 {
46                     found = true;
47                     break;
48                 }
49             }
50         }
51     }
52 }

```

```

44         }
45     }
46     if (found)
47     {
48         p = AEB[i].inst;
49         Operand *dst = AEB[i].tmp;
50         if (dst == nullptr)
51         {
52             dst = new Operand(new TemporarySymbolEntry(TypeSystem::
53                 intType, SymbolTable::getLabel()));
54             vector<Operand*> pOperands(p->getOperands());
55             Instruction *inst = new BinaryInstruction(op, dst,
56                 pOperands[1], pOperands[2], nullptr);
57             AEB[i].tmp = dst;
58             bb->insertBefore(inst, p);
59             Instruction *inst1 = new LoadInstruction(pOperands[0],
60                 dst, nullptr);
61             bb->insertBefore(inst1, p);
62             bb->remove(p);
63         }
64         Instruction *inst2 = new LoadInstruction(operands[0], dst,
65             nullptr);
66         bb->insertBefore(inst2, iter);
67         bb->remove(iter);
68     }
69     else
70     {
71         struct aeb tmp;
72         tmp.inst = (BinaryInstruction *)iter, tmp.opd1 = sym1, tmp.
73             opr = op, tmp.opd2 = sym2;
74         AEB.push_back(tmp);
75         SymbolEntry *sym = operands[0]->getEntry();
76         auto iter_def = operands[0]->getDef();
77         if (iter_def)
78         {
79             vector<Operand*> def_operands(iter_def->getOperands());
80             sym = def_operands[1]->getEntry();
81         }
82         delAEB(sym, AEB);
83         delAEB(sym, AEin[no]);
84     }
85 }
86 AEout[no] = vec_union(AEB, AEin[no]);
87 AEB.clear();
88 }

```

## 6. elim\_cse

该函数实现了全局的公共子表达式删除。其遍历了函数的每个基本块，使用宽度优先搜索(BFS)算法访问基本块，进行全局的公共子表达式删除。对于每个基本块，首先标记为已访问，并更新当前基本块的入口值编号表达式列表 AEIn。然后，调用 local\_elim\_cse 函数进行局部的公共子表达式删除。接着，将当前基本块的后继基本块加入队列，以便后续遍历。这个过程一直重复，直到所有基本块都被访问完成。

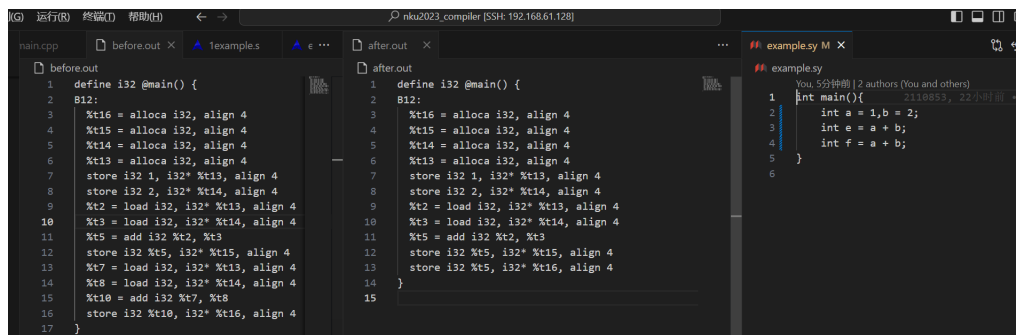
elim\_cse

```

1 void ElimComSubexpr::elim_cse()
2 {
3     auto iter = unit->begin();
4     VAEB AEB;
5     while (iter != unit->end())
6     {
7         vector<BasicBlock *> block_list = (*iter)->getBlockList();
8         BasicBlock *bb = (*iter)->getEntry();
9         queue<BasicBlock *> q;
10        q.push(bb);
11        bool first = true;
12        while (!q.empty())
13        {
14            BasicBlock *bb = q.front();
15            q.pop();
16            int no = bb->getNo();
17            ste[no] = true;
18            if (first)
19            {
20                AEIn[no];
21                first = false;
22            }
23            else
24            {
25                AEIn[no] = vintersection(bb);
26            }
27            local_elim_cse(bb, AEB);
28            for (auto succ = bb->succ_begin(); succ != bb->succ_end(); succ++)
29            {
30                if (!ste[(*succ)->getNo()])
31                {
32                    q.push(*succ);
33                }
34            }
35        }
36        iter++;
37    }
38 }

```

利用编写的相关程序，对给定的 SysY 编程语言代码优化。以给定示例代码为例，可以看到优化前后的代码差别。其中  $a + b$  部分为公共子表达式，在优化后，对该部分进行删除合并，使得生成的中间代码更加简洁，从而使得生成的目标代码也能够更加简练。



```
main.cpp before.out 1example.s A e ... after.out example.sy
1 define i32 @main() {
2 B12:
3 %t16 = alloca i32, align 4
4 %t15 = alloca i32, align 4
5 %t14 = alloca i32, align 4
6 %t13 = alloca i32, align 4
7 store i32 1, i32* %t13, align 4
8 store i32 2, i32* %t14, align 4
9 %t2 = load i32, i32* %t13, align 4
10 %t3 = load i32, i32* %t14, align 4
11 %t5 = add i32 %t2, %t3
12 store i32 %t5, i32* %t15, align 4
13 %t7 = load i32, i32* %t13, align 4
14 %t8 = load i32, i32* %t14, align 4
15 %t10 = add i32 %t7, %t8
16 store i32 %t10, i32* %t16, align 4
17 }

1 define i32 @main() {
2 B12:
3 %t16 = alloca i32, align 4
4 %t15 = alloca i32, align 4
5 %t14 = alloca i32, align 4
6 %t13 = alloca i32, align 4
7 store i32 1, i32* %t13, align 4
8 store i32 2, i32* %t14, align 4
9 %t2 = load i32, i32* %t13, align 4
10 %t3 = load i32, i32* %t14, align 4
11 %t5 = add i32 %t2, %t3
12 store i32 %t5, i32* %t15, align 4
13 store i32 %t5, i32* %t16, align 4
14 }
15 }

1 int main(){
2     int a = 1, b = 2;
3     int e = a + b;
4     int f = a + b;
5 }
6 }
```

图 7: 代码优化结果

## 九、 遇到的困难及解决方案

本次作业可谓“难度巨大”，遇到了数不胜数的困难，在此，感谢一直耐心解答我那些愚蠢问题的队友朱子轩同学，以及我的好朋友李威远、邓薇薇等人，他们是最牛的解决方案。

NIKU

## 十、 总结与感想

写到这，本学期的编译系统原理课程就要画上句号了，大三上也就要结束了。

本次实验，与其称之为实验，不如称之为一个小的工程。让我对整个编译器的前后端架构、不同模块间的分工协作等有了更加深刻的理解、认识。其代码量在现在这个阶段可谓十分巨大，其难度可谓十分巨大...

最终，我们的编译器完成了两种优化算法，通过了 130 个测试样例，在平台跑分 77 分（我们被平台算计了，平台上好多 WA 本地都是可以 PASS 的（气鼓鼓））。

	通过样例	总数	满分	得分		优化算法	满分	得分
	level1-1	38	58	5	5.00	寄存器分配优化	1.5	0
	level1-2	20				优化算法	1.5	1.5
	level2-1	3	14	1	0.93	(《循环的三种优化	2	0
	level2-2	5				自由发挥		
	level2-3	5				(可实现其他优化算		
	level2-4	39	42	1	0.93			
	level2-5	17	26	0.5	0.33			
	level2-6	3	11	1.5	0.41			
						满分	14	
						总分	9.09	

更改上方红色部分为自己的情况

图 8: 通过情况

从最开始的懵懵懂懂，到现在能够完成一个完整的 SysY 语言编译器工程作业；从学期初到学期末...bug 越写越多，难度越来越高，但一路走来，真的收获满满。非常感谢老师以及助教学长学姐们对我们的指导，你们辛苦了！

最后，祝自己完工大吉，祝老师及助教学长学姐们身体健康，工作顺意，提前祝大家新春愉快！