南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院

编译系统原理实验报告

编译器最终完成版

艾明旭　2111033

年级：2021 级

专业：信息安全

指导教师：王刚 李忠伟

2024 年 1 月 16 日

目录

一、 引言与综述

(一) 实验描述

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使我们能够更好地掌握编译原理的基本理论和编译程序构造的慕木方法和技巧，融会贯通本课程所学专业理讼知识：培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力：培养学生的创新能力及团队协作精神。

一个编译器所进行的工作一般可以划分为五个阶段：词法分析、语法分析、语义分析和中间代码产生、中间代码的优化、目标代码生成。

首先是词法分析，针对词法分析，我们设计了一个可以识别绝大部分标准 SysY 语言支特的词法符号，该词法分析器可以过滤空格、Tab 和回车，并且支持注释功能，即过滤掉注释符号后面的代码。词法分析器在识别到一个单词后，将该单词记录下来，如果是数据，则会在符号表的相应位置记录它的值，如果是标识符，则会先在符号表上进行查询，若没有则将其记录到符号表上，并将相应 TOKEN 的指针指向表中该位置。

接下来进行语法分析，在语法分析部分，会对所编写的代码的语法进行检验，看是否合乎我们所设定的语法规则，所设计的文法支特了函数、函数类型声明、变量类型声明、变量定义、表达式语句、if 条件语句和 while 循环语句等。在表达式语句方面，我们设计了支持所有算术运算、关系运算、逻辑运算和位运算功能的语法结构，并且语法上支持数组。

在类型检查（语义分析）和中间代码产生的阶段。我们在语法分析程序的相应部分加上了语义动作。在语义分析部分主要是对语法分析器的输出进行检查，并提供相应的错误，这是对程序员友好的。在中间代码部分根据正确的程序进行基本块的构建，流图的构建，并最终遍历语法树，输出中间代码。

最后一个阶段是目标代码生成。在这一步骤需要将中间代码提供的输出转换成真正的可以在环境中运行的汇编代码，需要分配真实存在的物理寄存器，在这一步骤中，我们完成了所有的提高要求，并对寄存器分配进行优化，最终通过了所有的 151 个测试样例。

1. 思考

如果不是人“手工编译”，而是要实现一个计算机程序 (编译器) 来将 SysY 程序转换为汇编程序，应该如何做？这个编译器程序的数据结构和算法设计是怎样的?

注意：编译器不能只会翻译一个源程序，而是要有能力翻译所有合法的 SysY 程序。而穷举所有 SysY 程序（无穷无尽）是不可能的，怎么办？搞定每个语言特性如何翻译即可！

2. 要求：

可学习 GCC 生成的其他 C 程序的汇编程序，仿照着编写自己 SysY 程序的汇编程序。

二、 定义你的编译器

这一部分作业的主要要求是了解你的编译器所支持的 SysY 语言特性，如支持何种数据类型 (int 等)，支持变量声明，赋值语句，复合语句，if 分支语句，以及 while/for 循环，支持算术运算（加减乘除、按位与或等）、逻辑运算（逻辑与或等）、关系运算（不等、等于、大于、小于等），支持函数、数组指针等等。从中选取你要实现的部分定义为你编译器功能，使用上下文无关文法描述你所选取的 SysY 语言子集。

(一) 上下文无关文法

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：

（1）一个终结符号集合 VT，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

（2）一个非终结符号集合 VN，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

（3）一个产生式集合 P，其中每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

（4）指定一个非终结符号为开始符号 S。因此，上下文无关文法可以通过 (VT ,VN ,P,S) 这个四元式定义。在描述文法时，我们将数位、符号和黑体字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

(二) SysY 语言特性

- 数据类型：int
- 变量声明、常量声明，常量、变量的初始化
- 语句：赋值（=）、表达式语句、语句块、if、while、return
- 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））
- 注释
- 输入输出
- 数组
- 变量、常量作用域——在语句块中包含变量、常量声明，break、continue 语句
- 函数

(三) CFG 描述 SysY 语言特性

CFG 是一个四元组 G =（N，T，P，S），其中

（1）N 是非终结符（Nonterminals）的有限集合；

（2）T 是终结符（Terminals）的有限集合，且 $N \cap T = u$；

（3）P 是产生式（Productions）的有限集合，$A \to a$，其中 $A \in N$（左部），$a \in (N \cup T)*$（右部），若 $a = \in$ ，则称 $A \to \in$ 为空产生式（也可以记为 A $\to$）；

（4）S 是文法的开始符号（Start symbol），$S \in N$

CFG 产生语言的基本方法：推导

推导的定义

推导的定义将产生式左部的非终结符替换为右部的文法符号序列 (展开产生式，用标记 => 表示)，直到得到一个终结符序列。

推导的符号：=>

推导的输入：产生式左部

推导的输出：一个终结符序列

1. 关键字

C++ 常用关键字如表1所示，每一个关键字在上下文无关文法中都会看作一个终结符，即语法树的叶结点。本实验将选取其中的一部分作为子集，构造 SysY 语言。

类型	关键字
数据类型相关	$int, bool, true, false, char, wchar\_t, int, double, float, short,$ $long, signed, unsigned$
控制语句相关	$switch, case, default, do, for, while, if, else,$ $break, continue, goto$
定义、初始化相关	$const, voltile, enum, export, extern, public, protected, private,$ $template, static, struct, class, union, mutable, virtual$
系统操作相关	$catch, throw, try, new, delete, friend, inline,$ $operator, retuster, typename$
命名相关	$using, namespace, typeof$
函数和返回值相关	$void, return, sizeof, typied$
其他	$this, asm, \_cast$

表 1: C++ 关键字

2. 变量

C 语言中规定，将一些程序运行中可变的值称之为变量，与常量相对。在程序运行期间，随时可能产生一些临时数据，应用程序会将这些数据保存在一些内存单元中，每个内存单元都用一个标识符来标识。这些内存单元我们称之为变量，定义的标识符就是变量名，内存单元中存储的数据就是变量的值。变量可以作左值，常量则只能作为右值。变量除了与常量相同的整型类型、实型类型、字符类型这三个基本类型之外，还有构造类型、指针类型、空类型。

三种基本类型

整型变量 整型常量为整数类型 *int* 的数据。可分别如下表示为八进制、十进制、十六进制

- 十进制整型变量：0，123，-1
- 八进制整型变量：0123，-01
- 十六进制整型变量：0x123，-0x88

实型变量 实型变量是实际中的小数，又称为浮点型变量。按照精度可以分为单精度浮点数（float）和双精度浮点数（double）。浮点数的表示有三种方式如下：

- 指明精度的表示：以 f 结尾为单精度浮点数，如：2.3f；以 d 结尾为双精度浮点数，如：3.6d
- 不加任何后缀的表示：11.1，5.5
- 指数形式的表示：5.022e+23f，0f

字符变量 *char* 用于表示一个字符，表示形式为 ′ 需要表示的字符常变量 ′。其中，所表示的内容可以是英文字母、数字、标点符号以及由转义序列来表示的特殊字符。如 ′a′ ′3′ ′,′ ′\n′。

变量的定义 用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。变量有三个基本要素：变量名，代表变量的符号；变量的数据类型，每一个变量都应具有一种数据类型且内存中占据一定的储存空间；变量的值，变量对应的存贮空间中所存放的内容。变量的定义可以以如下的形式：

```
type variable_list
```

在我们定义的 SysY 语言中，将支持整型变量（十进制）、字符型变量以及行主存储的整型一维数组类型。

3. 常量

C 语言中规定，将一些不可变的值称之为常量。常量可以分为整型常量、实型常量、字符常量这三种常量，其形式与整型变量、实型变量、字符变量基本相同，只不过在声明时必须初始化且在程序中不可以改变其值。**在我们定义的 SysY 语言中，将定义整型常量（十进制）和字符型常量。**

4. 运算符和表达式

在 C 语言中，运算符分为算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、杂项运算符六大类。其中，我们所设计的 SysY 语言将定义以下运算符：

运算符	描述
+	把两个操作数相加
-	从第一个操作数中减去第二个操作数
*	把两个操作数相乘
/	分子除以分母
%	取模运算符，整除后的余数

表 2: 算术运算符

运算符	描述
==	检查两个操作数的值是否相等，如果相等则条件为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

表 3: 关系运算符

表达式 由运算分量和运算符按一定规则组成。运算分量是运算符操作的对象，通常是各种类型的数据。运算符指明表达式的类型；表达式的运算结果是一个值——表达式的值。出现在赋值运算符左边的分量为左值，代表着一个可以存放数据的存储空间；左值只能是变量，不能是常量或表达式，因为只有变量才可以带表存放数据的存储空间。出现在赋值运算符右边的分量为右值，右值没有特殊要求。

运算符	描述
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。

表 4: 逻辑运算符

运算符	描述
=	简单的赋值运算符，把右边操作数的值赋给左边操作数。

表 5: 赋值运算符

运算符	描述
&	返回变量的地址。
*	指向一个变量。

表 6: 复杂运算符

运算符优先级 运算符中优先级确定了表达式中项的组合，这会极大地影响表达式的计算过程以及结果。运算的优先顺序为：括号优先运算 → 优先级高的运算符优先运算 → 优先级相同的运算参照运算符结合性依次进行。当表达式包含多个同级运算符时，运算的先后次序分为左结合规则和右结合规则。其中左结合规则是从左向右依次计算，包括的运算符有双目的算术运算符、关系运算符、逻辑运算符、位运算符、逗号运算符；右结合规则是从右向左依次计算，包括的运算符有可以连续运算的单目运算符、赋值运算符、条件运算符。运算符优先级由高到低排列：后缀 → 一元 → 乘除 → 加减 → 移位 → 关系 → 相等 → 位与 → 位异或 → 位或 → 逻辑与 → 逻辑或 → 条件 → 赋值 → 逗号

5. 语句

在 C 语言中，语句分为说明语句、表达式语句、控制语句、标签语句、复合语句和块语句。**在我们所定义的 SysY 语言中，我们将定义表达式语句和控制语句，其中控制语句分为分支语句、循环语句和转向语句。**

表达式语句 任意有效表达式都可以作为表达式语句，其形式为表达式后面加上“;”。

分支语句 if 语句和 if……else 语句，由关键字 if 和 else 组成。其基本形式如下

```
if(expr){
  stmts
}
else{
  stmts
}
```

循环语句 又称重复语句，用于重复执行某些语句。本实验的循环语句由 while 语句实现，基本形式如下

```
while{
  stmts
}
```

转向语句 用于从循环体跳出的 break 语句；用于立即结束本次循环而去继续下一次循环的 continue 语句；用于立即从某个函数中返回到调用该函数位置的 return 语句。

6. 函数

函数的定义 函数是一组一起执行一个任务的语句。程序中功能相同，结构相似的代码段可以用函数进行描述。函数的功能相对独立，用来解决某个问题，具有明显的入口和出口。函数也可以称为方法、子例程或程序等等。

函数说明 C 语言中，函数必须先说明后调用。函数的说明方式有两种，一种是函数原型，相当于“说明语句”，必须出现在调用函数之前；一种是函数定义，相当于“说明语句 + 初始化”，可以出现在程序的任何合适的地方。在函数声明中，参数的名称并不重要，只有参数的类型是必需的。函数的声明形式如下所示：

```
return_type function_name(parameter list);
```

形式化定义 C 语言中，函数的形式化定义如下所示：

```
return_type function_name(parameter list)
{
   body of the function
}
```

本实验定义的 SysY 语言的函数定义完全与此相同。

函数的参数 函数可以分为有参函数和无参函数。如果函数要使用参数，则必须声明接受参数值的变量，这些变量称为函数的形式参数。形式参数和函数中的局部变量一样，在函数创建时被赋予地址，在函数退出是被销毁。函数参数的调用分为传值调用和引用调用两种，传值调用是将实际的变量的值复制给形式参数，形式参数在函数体中的改变不会影响实际变量；引用调用是将形式参数作为指针调用指向实际变量的地址，当对在函数体中对形式参数的指向操作时，就相当于对实际参数本身进行的操作。

除此之外，函数还可以分为内联函数、外部函数等等，并还可以进行重载等操作。**本次实验的 SysY 语言，只对函数最基本的功能进行实现。**

(四) 形式化定义

接下来，我们将采用 CFG 即上下文无关文法对 SysY 语言进行形式化定义。上下文无关文由一个终结符号集合 V_T、一个非终结符号集合 V_N、一个产生式集合 P 和一个开始符号 S 四个元素组成。在接下来的定义中，数位、符号和黑体字符串将被看作终结符号，斜体字符串将被看

作非终结符号。若多个产生式以一个非终结符号为头部，则这些产生式的右部可以放在一起，并用 | 分割。

名称	符号	名称	符号
声明语句	*decl*	标识符	`id`
标识符列表	*idlist*	数据类型	*type*
表达式	*expr*	一元表达式	*unary_expr*
赋值表达式	*assign_expr*	逻辑表达式	*logical_expr*
算数表达式	*math_expr*	关系表达式	*relation_expr*
数字	`digit`	整数	*decimal*
符号和字母	`character`	常量定义	*const_init*
分配内存	*allocate*	回收内存	*recovery*
语句	*stmt*	循环语句	*loop_stmt*
分支语句	*selection_stmt*	跳转语句	*jmp_stmt*
函数定义	*funcdef*	函数参数	*para*
函数参数列表	*paralist*	函数名称	*funcname*
函数返回值	*re_type*		

表 7: 下文中各符号含义

1. 变量声明

变量可以声明分为仅声明变量和声明变量且赋初值。(整型变量只支持十进制) 数组由指针实现。

$$
\begin{aligned}
idlist &\rightarrow idlist, \texttt{id} \mid \texttt{id} \\
type &\rightarrow \texttt{int} \mid \texttt{char} \\
decl &\rightarrow type\ idlist \mid \\
&\quad type\ \texttt{id} = logical\_expr \mid \\
&\quad type\ \texttt{id} = unary\_expr \mid \\
&\quad type\ *\texttt{id} = \&\texttt{id}
\end{aligned}
$$

2. 常量声明

常量包括整型常量（十进制）和字符型常量。

$$const\_init \rightarrow \texttt{const}\ type\ \texttt{id} = unary\_expr$$

3. 表达式

表达式可以分为一元表达式、赋值表达式、逻辑表达式、算数表达式、关系表达式。

$$expr \rightarrow unary\_expr \mid assign\_expr \mid logical\_expr \mid math\_expr \mid relation\_expr$$

4. 赋值表达式

赋值表达式不能对常量进行赋值。

$$\begin{aligned}
\texttt{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{decimal} &\rightarrow \texttt{digit} \mid \mathit{decimal}\ \texttt{digit} \\
\texttt{character} &\rightarrow _ \mid \texttt{a}-\texttt{z} \mid \texttt{A}-\texttt{Z} \\
\mathit{unary_expr} &\rightarrow \mathit{decimal} \mid {'}\texttt{character}{'} \mid \texttt{id} \\
\mathit{assign_expr} &\rightarrow \texttt{id} = \mathit{unary_expr} \mid \\
&\quad\ \texttt{id} = \mathit{logical_expr} \mid \\
&\quad\ \texttt{id} = \mathit{funcnmae}(\mathit{paralist})
\end{aligned}$$

5. 逻辑表达式

逻辑表达式包括逻辑与、逻辑或、逻辑非运算。

$$\begin{aligned}
\mathit{logical_expr} \rightarrow\ & \mathit{unary_expr} \mid \\
& !(\mathit{logical_expr}) \mid \\
& \mathit{logical_expr} || \mathit{logical_expr} \mid \\
& \mathit{logical_expr} \&\& \mathit{logical_expr}
\end{aligned}$$

6. 关系表达式

关系表达式包括判断两个值是否相等或比较两值的大小。

$$\begin{aligned}
\mathit{relation_expr} \rightarrow\ & \mathit{unary_expr} == \mathit{unary_expr} \mid \\
& \mathit{unary_expr}\ != \mathit{unary_expr} \mid \\
& \mathit{unary_expr} > \mathit{unary_expr} \mid \\
& \mathit{unary_expr} < \mathit{unary_expr} \mid \\
& \mathit{unary_expr} >= \mathit{unary_expr} \mid \\
& \mathit{unary_expr} <= \mathit{unary_expr}
\end{aligned}$$

7. 算数表达式

算数表达式包括加、减、乘、除、取模、取负六种运算。

$$\begin{aligned}
\mathit{math_expr} \rightarrow\ & \mathit{unary_expr} \mid \\
& -\mathit{unary_expr} \mid \\
& \mathit{math_expr} + \mathit{math_expr} \mid \\
& \mathit{math_expr} - \mathit{math_expr} \mid \\
& \mathit{math_expr} * \mathit{math_expr} \mid \\
& \mathit{math_expr}\ /\ \mathit{math_expr} \mid \\
& \mathit{math_expr}\ \%\ \mathit{math_expr}
\end{aligned}$$

8. 函数

函数的返回值有整型、字符型、指针型三种，参数有整型、字符型、指针型、引用四种。(数组由指针实现)

$$
\begin{aligned}
funcdef &\rightarrow re\_type\ \texttt{funcname}(paralist)\ stmt \\
paralist &\rightarrow para, paralist \mid para \\
para &\rightarrow type\ \texttt{id} \mid \\
&\quad\ type*\ \texttt{id} \mid \\
&\quad\ type\&\ \texttt{id} \\
re\_type &\rightarrow type \mid type* \mid \texttt{void}
\end{aligned}
$$

9. 系统操作

系统操作包括分配内存和回收内存。

$$
\begin{aligned}
allocate &\rightarrow type\ *\texttt{id} = \texttt{new}\ type\ [\texttt{id}] \mid \\
&\quad\ type\ *\texttt{id} = \texttt{new}\ type\ [decimal] \\
recovery &\rightarrow \texttt{delete}\ []\ \texttt{id}
\end{aligned}
$$

10. 循环语句

循环语句利用 while 实现。

$$
loop\_stmt \rightarrow \texttt{while}(expr)\ \{stmt\}
$$

11. 分支语句

分支语句利用 if else 实现。

$$
\begin{aligned}
selection\_stmt \rightarrow &\texttt{if}(expr)\ \{stmt\} \mid \\
&\texttt{if}(expr)\ \{stmt\}\ \texttt{else}\ \{stmt\}
\end{aligned}
$$

12. 跳转语句

跳转语句包括继续执行循环、退出循环和返回值。

$$
\begin{aligned}
jmp\_stmt \rightarrow &\texttt{continue} \mid \\
&\texttt{break} \mid \\
&\texttt{return} \mid \\
&\texttt{return}\ expr
\end{aligned}
$$

三、 词法分析器

- 由选项和声明组成。%{ %} 围住的部分会被原封不动的复制到 Flex 生成的词法分析器代码靠近前面的地方。
- 由一系列的匹配规则 (Pattern) 和动作 (Action) 组成，每行一个匹配规则和一个动作。匹配采用了正则表达式，动作是纯 C 代码编写的，放在里面，可以有多行代码。动作描述了匹配到规则后执行的操作，上面的例子的动作是打印提示信息。
- C 语言代码，会直接被复制到生成的词法生成器代码中。一般都是 main 程序，是词法分析器的入口。

(一) 基础结构

1. 定义部分

在此次实验中，我们已经设计了符号表，但不过只是实现了一些简单的功能，比如保存词素等简单内容，但是符号表的数据结构，搜索算法，词素的保存，保留字的处理等问题将放到后续实验中进行。

定义部分包含选项、文字块、开始条件、转换状态、规则等。

在给出的样例中%option noyywrap 即为一个选项，控制 flex 的一些功能，具体来说，这里的选项功能为去掉默认的 yywrap 函数调用，这是一个早期 lex 遗留的鸡肋，设计用来对应多文件输入的情况，在每次 yylex 结束后调用，但一般来说用户往往不会用到这个特性。

而用%{%} 包围起来的部分为文字块，可以看到块内可以直接书写 C 代码，Flex 会把文字块内的内容原封不动的复制到编译好的 C 文件中，而%top{} 块也为文字块，只是 Flex 会将这部分内容放到编译文件的开头，一般用来引用额外的头文件，这里值得说明的是，如果观察 Flex 编译出的文件，可以发现它默认包含了以下内容:

```
/* begin standard C headers. */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

/* end standard C headers. */
```

也就是说这部分文件其实不需要额外的声明就可以直接使用。

最终的部分结果如下:

```
    //十进制
DECIMIAL ([1-9][0-9]*|0)
 //八进制
OCTAL (0[0-7]+)
 //十六进制
HEXADECIMAL (0[xX][0-9A-Fa-f]+)

ID [[:alpha:]_][[:alpha:][:digit:]_]*
 //回车换行符
EOL (\r\n|\n|\r)
 //制表符（空格）
```

```
WHITE [\t ]

%x BLOCKCOMMENT
 //块注释
BLOCKCOMMENTBEGIN "/*"
 //通配符+换行
BLOCKCOMMENTELEMENT .|\n
BLOCKCOMMENTEND "*/"
 //行注释：以双斜杠开头，后跟若干个非换行的字符
LINECOMMENT \/\/[^\n]*
```

2. 规则部分

规则部分包含模式行与 C 代码，这里的写法很好理解，需要说明的是当存在二义性问题时，Flex 采用两个简单的原则来处理矛盾:

1. 匹配尽可能长的字符串——最长前缀原则。
2. 如果两个模式都可以匹配的话，匹配在程序中更早出现的模式。

```
    "const" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("CONST\tconst");
    return CONST;
}

"int" {
    /*
    * Questions:
    *   Q1: Why we need to return INT in further labs?
    *   Q2: What is "INT" actually?
    */
    if(dump_tokens)
        DEBUG_FOR_LAB4("INT\tint");
    return INT;
}
"void" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("VOID\tvoid");
    return VOID;
}
"if" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("IF\tif");
    return IF;
};

"else" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("ELSE\telse");
```

```
31      return ELSE;
32  };
33  "return" {
34      if(dump_tokens)
35          DEBUG_FOR_LAB4("RETURN\treturn");
36      return RETURN;
37  }
38  "while" {
39      if(dump_tokens)
40          DEBUG_FOR_LAB4("WHILE\twhile");
41      return WHILE;
42  }
43  "break" {
44      if(dump_tokens)
45          DEBUG_FOR_LAB4("BREAK\tbreak");
46      return BREAK;
47  }
48  "continue" {
49      if(dump_tokens)
50          DEBUG_FOR_LAB4("CONTINUE\tcontinue");
51      return CONTINUE;
52  }
53  "&&" {
54      if(dump_tokens)
55          DEBUG_FOR_LAB4("AND\t=");
56      return AND;
57  }
58  "||" {
59      if(dump_tokens)
60          DEBUG_FOR_LAB4("OR\t||");
61      return OR;
62  }
63  "!" {
64      if(dump_tokens)
65          DEBUG_FOR_LAB4("NOT\t!");
66      return NOT;
67  }
68  "=" {
69      if(dump_tokens)
70          DEBUG_FOR_LAB4("ASSIGN\t=");
71      return ASSIGN;
72  }
73  "<" {
74      if(dump_tokens)
75          DEBUG_FOR_LAB4("LESS\t<");
76      return LESS;
77  }
78  "+" {
```

```
79      if(dump_tokens)
80          DEBUG_FOR_LAB4("ADD\t+");
81      return ADD;
82  }
83  "-" {
84      if(dump_tokens)
85          DEBUG_FOR_LAB4("SUB\t-");
86      return SUB;
87  }
88  "*" {
89      if(dump_tokens)
90          DEBUG_FOR_LAB4("MUL\t*");
91      return MUL;
92  }
93  "/" {
94      if(dump_tokens)
95          DEBUG_FOR_LAB4("DIV\t/");
96      return DIV;
97  }
98  "%" {
99      if(dump_tokens)
100         DEBUG_FOR_LAB4("MOD\t%");
101     return MOD;
102 }
103 ">" {
104     if(dump_tokens)
105         DEBUG_FOR_LAB4("GREATER\t>");
106     return GREATER;
107 }
108 ">=" {
109     if(dump_tokens)
110         DEBUG_FOR_LAB4("GREATEREQUAL\t>=");
111     return GREATEREQUAL;
112 }
113 "<=" {
114     if(dump_tokens)
115         DEBUG_FOR_LAB4("LESSEQUAL\t<=");
116     return LESSEQUAL;
117 }
118 "==" {
119     if(dump_tokens)
120         DEBUG_FOR_LAB4("EQUAL\t==");
121     return EQUAL;
122 }
123 "!=" {
124     if(dump_tokens)
125         DEBUG_FOR_LAB4("NOTEQUAL\t!=");
126     return NOTEQUAL;
```

```
}
";" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("SEMICOLON\t;");
    return SEMICOLON;
}

"," {
    if(dump_tokens)
        DEBUG_FOR_LAB4("COMMA\t}");
    return COMMA;
}
"[" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("LBRACKET\t[");
    return LBRACKET;
}
"]" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("RBRACKET\t}");
    return RBRACKET;
}
```

正则定义对于正则定义部分需要完成对例如如八进制数字、十六进制数字、行注释等内容的补全

进制转换对于进制转换来说 8 进制和 16 进制是相似的，所以在这里解释一下 16 进制的转换，代码如下：

```
    {HEXADECIMAL} {
    int num;
    sscanf(yytext, "%x", &num);
    if(dump_tokens)
        DEBUG_FOR_LAB4(string("NUMBER\t") + to_string(num));
    yylval.itype = num;
    return INTEGER;
}
```

接下来将有一部分定义输入输出的部分：

我们可以利用 get 函数定义输入输出进行实现

```
    "getint" {
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    char* str=new char[strlen(yytext)+1];
    strcpy(str,yytext);
    yylval.strtype=str;
    std::vector<Type*> vec;
    Type* funcType = new FunctionType(TypeSystem::intType, vec);
    SymbolTable* st = identifiers;
    while(st->getPrev())
        st = st->getPrev();
```

```
12      SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
            getLevel());
13      if(!(st->lookup(yytext)))
14      st->install(yytext, se);
15      return ID;
16  }
17  "getch" {
18      if(dump_tokens)
19          DEBUG_FOR_LAB4(yytext);
20      char* str=new char[strlen(yytext)+1];
21      strcpy(str,yytext);
22      yylval.strtype=str;
23      std::vector<Type*> vec;
24      Type* funcType = new FunctionType(TypeSystem::intType, vec);
25      SymbolTable* st = identifiers;
26      while(st->getPrev())
27          st = st->getPrev();
28      SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
            getLevel());
29      if(!(st->lookup(yytext)))
30      st->install(yytext, se);
31      return ID;
32  }
33  "putint" {
34      if(dump_tokens)
35          DEBUG_FOR_LAB4(yytext);
36      char* str=new char[strlen(yytext)+1];
37      strcpy(str,yytext);
38      yylval.strtype=str;
39      std::vector<Type*> vec;
40      Type *t=(Type*)(new IntType(32));
41      vec.push_back(t);
42      Type* funcType = new FunctionType(TypeSystem::voidType, vec);
43      SymbolTable* st = identifiers;
44      while(st->getPrev())
45          st = st->getPrev();
46      SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
            getLevel());
47      if(!(st->lookup(yytext)))
48          st->install(yytext, se);
49      return ID;
50  }
51  "putch" {
52      if(dump_tokens)
53          DEBUG_FOR_LAB4(yytext);
54      char* str=new char[strlen(yytext)+1];
55      strcpy(str,yytext);
56      yylval.strtype=str;
```

```
57      std::vector<Type*> vec;
58      Type *t=(Type*)(new IntType(32));
59      vec.push_back(t);
60      Type* funcType = new FunctionType(TypeSystem::voidType, vec);
61      SymbolTable* st = identifiers;
62      while(st->getPrev())
63          st = st->getPrev();
64      SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
            getLevel());
65      if(!(st->lookup(yytext)))
66          st->install(yytext, se);
67      return ID;
68  }
69  "putarray" {
70      if(dump_tokens)
71          DEBUG_FOR_LAB4(yytext);
72      char* str=new char[strlen(yytext)+1];
73      strcpy(str,yytext);
74      yylval.strtype=str;
75      std::vector<Type*> vec;
76      Type *t=(Type*)(new IntType(32));
77      vec.push_back(t);
78      Type* t2 = (Type*)(new ArrayType(TypeSystem::intType, -1));
79      vec.push_back(t2);
80      Type* funcType = new FunctionType(TypeSystem::voidType, vec);
81      SymbolTable* st = identifiers;
82      while(st->getPrev())
83          st = st->getPrev();
84      SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
            getLevel());
85      if(!(st->lookup(yytext)))
86          st->install(yytext, se);
87      return ID;
88  }
89  "getarray" {
90      if(dump_tokens)
91          DEBUG_FOR_LAB4(yytext);
92      char* str=new char[strlen(yytext)+1];
93      strcpy(str,yytext);
94      yylval.strtype=str;
95      std::vector<Type*> vec;
96      Type* t2 = (Type*)(new ArrayType(TypeSystem::intType, -1));
97      vec.push_back(t2);
98      Type* funcType = new FunctionType(TypeSystem::intType, vec);
99      SymbolTable* st = identifiers;
100     while(st->getPrev())
101         st = st->getPrev();
102     SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st->
```

```
        getLevel());
103     if(!(st->lookup(yytext)))
104         st->install(yytext, se);
105     return ID;
106 }
```

3. C++ 版本

如果我们想要调用一些 C++ 中的标准库，或者说运用 C++ 的语法，对应的 Flex 程序结构需要做出一些调整，但大同小异。

最终的部分代码如下;

```
1     yyerrorlab:
2   if (0)
3     YYERROR;
4   ++yynerrs;
5   YYPOPSTACK (yylen);
6   yylen = 0;
7   YY_STACK_PRINT (yyss, yyssp);
8   yystate = *yyssp;
9   goto yyerrlab1;
10
11 yyerrlab1:
12   yyerrstatus = 3;      /* Each real token shifted decrements this.  */
13
14   /* Pop stack until we find a state that shifts the error token.  */
15   for (;;)
16     {
17       yyn = yypact[yystate];
18       if (!yypact_value_is_default (yyn))
19         {
20           yyn += YYSYMBOL_YYerror;
21           if (0 <= yyn && yyn <= YYLAST && yycheck[yyn] == YYSYMBOL_YYerror)
22             {
23               yyn = yytable[yyn];
24               if (0 < yyn)
25                 break;
26             }
27         }
28
29       /* Pop the current state because it cannot handle the error token.  */
30       if (yyssp == yyss)
31         YYABORT;
32
33
34       yydestruct ("Error: popping",
35                   YY_ACCESSING_SYMBOL (yystate), yyvsp);
36       YYPOPSTACK (1);
```

```
37        yystate = *yyssp;
38        YY_STACK_PRINT (yyss, yyssp);
39      }
40
41    YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN
42    *++yyvsp = yylval;
43    YY_IGNORE_MAYBE_UNINITIALIZED_END
44
45
46    /* Shift the error token.  */
47    YY_SYMBOL_PRINT ("Shifting", YY_ACCESSING_SYMBOL (yyn), yyvsp, yylsp);
48
49    yystate = yyn;
50    goto yynewstate;
51
52      yyreturnlab:
53    if (yychar != YYEMPTY)
54      {
55        /* Make sure we have latest lookahead translation.  See comments at
56           user semantic actions for why this is necessary.  */
57        yytoken = YYTRANSLATE (yychar);
58        yydestruct ("Cleanup: discarding lookahead",
59                    yytoken, &yylval);
60      }
61    /* Do not reclaim the symbols of the rule whose action triggered
62       this YYABORT or YYACCEPT.  */
63    YYPOPSTACK (yylen);
64    YY_STACK_PRINT (yyss, yyssp);
65    while (yyssp != yyss)
66      {
67        yydestruct ("Cleanup: popping",
68                    YY_ACCESSING_SYMBOL (+*yyssp), yyvsp);
69        YYPOPSTACK (1);
70      }
71  #ifndef yyoverflow
72    if (yyss != yyssa)
73      YYSTACK_FREE (yyss);
74  #endif
75    return yyresult;
76  }
```

(二) 其他特性

标识符

符号表对于符号表来说本次实现的较为简单，声明一个结构体，存储它的值以及作用域，然后定义一个此结构体的容器，相当于一个简易的符号表。

作用域由于每个符号都有一定的作用域，在不同的作用域中的符号可使用相同的名字，在同

一个作用域就不能出现这种情况，也方便之后对于重定义的类型检查的判断，在这里判断作用域的方法主要是根据大括号：

```
    "(" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("LPAREN\t(");
    return LPAREN;
}
")" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("RPAREN\t)");
    return RPAREN;
}
"{" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("LBRACE\t{");
    return LBRACE;
}
"}" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("RBRACE\t}");
    return RBRACE;
}
```

定义当作用域和符号表实现之后，对于标识符的实现就变得简单了，具体如下：

1. 起始状态

在定义部分，我们可以声明一些起始状态，用来限制特定规则的作用范围。用它可以很方便地做一些事情，我们用识别注释段作为一个例子，因为在注释段中，同样会包含数字字母标识符等等元素，但我们不应将其作为正常的元素来识别，这时候通过声明额外的起始状态以及规则会很有帮助。

```
    {commentbegin} {BEGIN BLOCKCOMMENT;}
<BLOCKCOMMENT>{commentelement} {}
<BLOCKCOMMENT>{commentend} {BEGIN INITIAL;}
{LINECOMMENT}
```

在这之中，声明部分的%x 声明了一个新的起始状态，而在之后的规则使用中加入 < 状态名 > 的表明该规则只在当前状态下生效。而状态的切换可以看出通过在之后附加的语法块中通过定义好的宏 BEGIN 来切换，注意初始状态默认为 INITIAL，因此在结束该状态时我们实际写的是切换回初始状态。

2. 行号

如果你有需要了解当前处理到文件的第几行，通过添加%option yylineno，Flex 会定义全局变量 yylineno 来记录行号，遇到换行符后自动更新，但要注意 Flex 并不会帮你做初始化，需要自行初始化。

```
    <*>{EOL} {                  // EOL回车换行符
```

```
    yylineno++;
}
extern int yylineno;
int yylineno = 1;

extern char *yytext;
#ifdef yytext_ptr
#undef yytext_ptr
#endif
#define yytext_ptr yytext

static yy_state_type yy_get_previous_state ( void );
static yy_state_type yy_try_NUL_trans ( yy_state_type current_state  );
static int yy_get_next_buffer ( void );
static void yynoreturn yy_fatal_error ( const char* msg  );
```

注释实现多行注释的时候，重点是状态的切换和换行的处理

四、 语法分析器

(一) 类型系统

变量的类型，仿佛只是简单作为变量结点的一个属性而已，但仔细考虑会发现它可以极其复杂。直观上，我们有 struct、union 构造复合类型，函数本身作为变量，它也有其自身的特殊类型。类型系统是编程语言理论的一个重要一部分。很有趣的一点是，类型系统与数理逻辑紧密相关，类型的检查可以视为定理的证明，这一关系被称为 Curry-Howard Correspondence。比如 struct 可以视为合取，union 可以视为析取，函数的输入类型与输出类型可以视为蕴含 1 。为了进行静态类型检查，你需要根据你的目标语言设计好与你需要的类型系统有关的数据结构。你可能还要考虑如何插入“类型转换”。

在语法分析的实验中，我们借助 Yacc 工具实现了语法分析器，其使用词法分析器输出的 token 流作为输入，把 token 流转换成树状的中间表示，通常会转换成语法树，然后在语法树的基础上做一系列的处理。如果输入的 token 流不符合语法分析器的规定的语法，语法分析器会报语法错误。

设计目标在本次实验中，需要完善 SysY 语言的上下文无关文法并借助 Yacc 工具实现语法分析器:

- 设计语法树数据结构：结点类型的设计，不同类型的节点应保存的信息
- 扩展上下文无关文法，设计翻译模式
- 设计 Yacc 程序，实现能构造语法树的分析器
- 以文本方式输出语法树结构，验证语法分析器实现的正确性

(二) 符号表

符号表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。

符号表主要用于作用域的管理，我们为每个语句块创建一个符号表，块中声明的每一个变量都在该符号表中对应着一个符号表条目。在词法分析阶段，我们只能识别出标识符，不能区分这个标识符是用于声明还是使用。而在语法分析阶段，我们能清楚的知道一个程序的语法结构，如果该标识符用于声明，那么语法分析器将创建相应的符号表条目，并将该条目存入当前作用域对应的符号表中，如果是使用该标识符，将从当前作用域对应的符号表开始沿着符号表链搜索符号表项。

框架代码中，我们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。代码已经实现了符号表的插入函数，你需要在 SymbolTable.cpp 中实现符号表的查找函数。

```
1    SymbolEntry::SymbolEntry(Type *type, int kind)
2    {
3        this->type = type;
4        this->kind = kind;
5        this->link = nullptr;
6    }
7
8    void SymbolEntry::setLink(SymbolEntry *se)
```

```
{
    SymbolEntry *temp = this;
    int count = ((FunctionType *)(se->getType()))->getParamsType().size();
    while (temp)
    {
        if ((long unsigned int)count == ((FunctionType *)(temp->getType()))->
            getParamsType().size())
        {
            fprintf(stderr, "function \"%s\" is defined twice\n", se->toStr()
                .c_str());
            assert((long unsigned int)count != ((FunctionType *)(temp->
                getType()))->getParamsType().size());
        }
        temp = temp->getLink();
    }
    temp = this;
    while (temp->getLink())
        temp = temp->getLink();
    temp->link = se;
}
SymbolEntry *SymbolTable::lookup(std::string name)
{
    SymbolTable *current = identifiers;
    while (current != nullptr)
        // symbolTable为map类型的成员变量
        if (current->symbolTable.find(name) != current->symbolTable.end())
            return current->symbolTable[name];
        else
            // 向下一个SymbolTable去找
            current = current->prev;
    return nullptr;
}

// install the entry into current symbol table.
void SymbolTable::install(std::string name, SymbolEntry *entry)
{
    // 同时检查是否有函数的重定义,如果有同名的函数, 链入符号表项
    if (symbolTable.find(name) != symbolTable.end() && symbolTable[name]->
        getType()->isFunc())
    {
        symbolTable[name]->setLink(entry);
    }
    else
    {
        fprintf(stderr, "install成功");
        symbolTable[name] = entry;
    }
}
```

(三) 抽象语法树

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。我们可以简单用 struct 去涵盖所有需要的内容，也可以设计复杂的继承结构。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。

对于节点类的定义如下:

```
    class Node
{
private:
    static int counter;
    int seq;

protected:
    std::vector<Instruction *> true_list;
    std::vector<Instruction *> false_list;
    static IRBuilder *builder;
    void backPatch(std::vector<Instruction *> &list, BasicBlock *bb, bool
        branch);
    std::vector<Instruction *> merge(std::vector<Instruction *> &list1, std::
        vector<Instruction *> &list2);

public:
    // 结点又指向下一个结点，用以表示表达式声明列表，函数参数列表等若干个表达
        式拼接而成的表达式
    Node *link;
    Node();
    int getSeq() const { return seq; };
    static void setIRBuilder(IRBuilder *ib) { builder = ib; };
    virtual void output(int level) = 0; // 等待子类重载实现
    // 添加link结点
    void setLink(Node *node);
    // 获取当前结点的link结点
    Node *getLink();
    virtual void typeCheck() = 0;
    virtual void genCode() = 0;
    std::vector<Instruction *> &trueList() { return true_list; }
    std::vector<Instruction *> &falseList() { return false_list; }
};
```

二元表达式

```
   void BinaryExpr::output(int level)
{
    std::string op_str;
    switch (op)
    {
    case ADD:
```

```
7           op_str = "add";
8           break;
9       case SUB:
10          op_str = "sub";
11          break;
12      case MUL:
13          op_str = "mul";
14          break;
15      case DIV:
16          op_str = "div";
17          break;
18      case MOD:
19          op_str = "mod";
20          break;
21      case AND:
22          op_str = "and";
23          break;
24      case OR:
25          op_str = "or";
26          break;
27      case LESS:
28          op_str = "less";
29          break;
30      case LEQ:
31          op_str = "lessequal";
32          break;
33      case MORE:
34          op_str = "greater";
35          break;
36      case MEQ:
37          op_str = "greaterequal";
38          break;
39      case EQ:
40          op_str = "equal";
41          break;
42      case NE:
43          op_str = "notequal";
44          break;
45      }
46      fprintf(yyout, "%*cBinaryExpr\top: %s\n", level, ' ', op_str.c_str());
47      expr1->output(level + 4);
48      expr2->output(level + 4);
49  }
50
51  int BinaryExpr::getValue()
52  {
53      int value;
54      switch (op)
```

```
55      {
56      case ADD:
57          value = expr1->getValue() + expr2->getValue();
58          break;
59      case SUB:
60          value = expr1->getValue() - expr2->getValue();
61          break;
62      case MUL:
63          value = expr1->getValue() * expr2->getValue();
64          break;
65      case DIV:
66          value = expr1->getValue() / expr2->getValue();
67          break;
68      case MOD:
69          value = expr1->getValue() % expr2->getValue();
70          break;
71      case AND:
72          value = expr1->getValue() && expr2->getValue();
73          break;
74      case OR:
75          value = expr1->getValue() || expr2->getValue();
76          break;
77      case LESS:
78          value = expr1->getValue() < expr2->getValue();
79          break;
80      case LEQ:
81          value = expr1->getValue() <= expr2->getValue();
82          break;
83      case MORE:
84          value = expr1->getValue() > expr2->getValue();
85          break;
86      case MEQ:
87          value = expr1->getValue() >= expr2->getValue();
88          break;
89      case EQ:
90          value = expr1->getValue() == expr2->getValue();
91          break;
92      case NE:
93          value = expr1->getValue() != expr2->getValue();
94          break;
95      }
96      return value;
97  }
```

Node 为 AST 结点的抽象基类，ExprNode 为表达式结点的抽象基类，从 ExprNode 中派生出 Id。Node 类中声明了纯虚函数 output，用于输出语法树信息，派生出的具体子类均需要对其进行实现。

接下来还有对于语法树的定义：

```
    class BinaryExpr : public ExprNode
{
private:
    int op;
    ExprNode *expr1, *expr2;

public:
    enum
    {
        ADD,SUB, MUL, DIV, MOD,AND,OR,LESS, MORE,LEQ,  MEQ, EQ,  NE
    };
    BinaryExpr(SymbolEntry *se, int op, ExprNode *expr1, ExprNode *expr2) :
        ExprNode(se), op(op), expr1(expr1), expr2(expr2) { dst = new Operand(
        se); };
    void output(int level);
    int getValue();
    void typeCheck();
    void genCode();
    void setType(Type *type) { this->type = type; };
};

class UnaryExpr : public ExprNode
{
private:
    int op;
    ExprNode *expr;

public:
    enum
    {
        ADD,
        SUB,
        NOT
    };
    UnaryExpr(SymbolEntry *se, int op, ExprNode *expr) : ExprNode(se), op(op)
        , expr(expr) { dst = new Operand(se); };
    void output(int level);
    int getValue();
    void typeCheck();
    void genCode();
};

class FuncCallExpr : public ExprNode
{
private:
    ExprNode *param;

```

```
public:
    FuncCallExpr(SymbolEntry *se, ExprNode *param = nullptr);
    void output(int level);
    void typeCheck();
    void genCode();
    int getValue() { return -1; };
};
```

语句与**语法分析**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器创建。在自底向上构建语法树时（与预测分析法相对），我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，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。

对于 if、while、return 这种基本块变化的语句，在语法分析阶段需要将其的条件和对应条件所能跳到的基本块的内容都要传给语法树，这在将来基本块跳转的分析阶段十分有用

赋值和语句块对于赋值来说，需要传递的参数为其左值和对应的赋值的表达式，而对于一个语句块来说，由于已经进入到了一个新的作用域中，所以需要重新赋予其符号表，将之前的弹出，为后面标识符的分析作准备

```
void CompoundStmt::output(int level)
{
    fprintf(yyout, "%*cCompoundStmt\n", level, ' ');
    if (stmt)
        stmt->output(level + 4);
}

void SeqNode::output(int level)
{
    // fprintf(yyout, "%*cSequence\n", level, ' ');
    stmt1->output(level);
    stmt2->output(level);
}

void EmptyStmt::output(int level)
{
    fprintf(yyout, "%*cEmptyStmt\n", level, ' ');
}

void IfStmt::output(int level)
{
    fprintf(yyout, "%*cIfStmt\n", level, ' ');
    cond->output(level + 4);
    thenStmt->output(level + 4);
}

void IfElseStmt::output(int level)
{
    fprintf(yyout, "%*cIfElseStmt\n", level, ' ');
    cond->output(level + 4);
```

```
    thenStmt->output(level + 4);
    elseStmt->output(level + 4);
}

void WhileStmt::output(int level)
{
    fprintf(yyout, "%*cWhileStmt\n", level, ' ');
    cond->output(level + 4);
    stmt->output(level + 4);
}
void BreakStmt::output(int level)
{
    fprintf(yyout, "%*cBreakStmt\n", level, ' ');
}

void ContinueStmt::output(int level)
{
    fprintf(yyout, "%*cContinueStmt\n", level, ' ');
}
void ReturnStmt::output(int level)
{
    fprintf(yyout, "%*cReturnStmt\n", level, ' ');
    retValue->output(level + 4);
}
void AssignStmt::output(int level)
{
    fprintf(yyout, "%*cAssignStmt\n", level, ' ');
    lval->output(level + 4);
    expr->output(level + 4);
}
```

二元表达式二元表达式的实现需要考虑先后顺序，所以要使得单表达式（例如一元表达式）先计算，然后进行乘除法，然后进行加减法等，然后是关系运算，最后是逻辑运算，在这里用如下箭头的方式进行表达，位于前面的属于越靠近最后生成的表达式，越靠近后面的属于最开始的表达式

```
    AddExp  // 加减法(双目算数表达式)
    : MulExp {$$ = $1;}
    | AddExp ADD MulExp
    {
        SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::intType,
            SymbolTable::getLabel());
        $$ = new BinaryExpr(se, BinaryExpr::ADD, $1, $3);
    }
    | AddExp SUB MulExp
    {
        SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::intType,
            SymbolTable::getLabel());
        $$ = new BinaryExpr(se, BinaryExpr::SUB, $1, $3);
```

```
    }
    ;
```

(四) 语法分析与语法树的构造

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:

语法分析树 stmt 的声明与定义:

```

class ExprStmt : public StmtNode
{
private:
    ExprNode *expr;

public:
    ExprStmt(ExprNode *expr) : expr(expr){};
    void output(int level);
    void typeCheck();
    void genCode() { expr->genCode(); };
    bool checkRet(Type *retType) { return false; }
};

class DeclStmt : public StmtNode
{
private:
    Id *id;
    // 用于获取赋值给标识符的表达式,如果为空, 代表只声明, 不赋初值
    ExprNode *expr;

public:
    DeclStmt(Id *id, ExprNode *expr = nullptr) : id(id), expr(expr)
    {
        if (expr)
        {
            this->expr = expr;
            if (expr->isInitArr())
                ((InitArray *)(this->expr))->fill();
        }
    };
    void output(int level);
    Id *getId() { return id; }
    void typeCheck();
    void genCode();
    bool checkRet(Type *retType) { return false; }
};
```

```
38
39 class IfStmt : public StmtNode
40 {
41 private:
42     ExprNode *cond;
43     StmtNode *thenStmt;
44
45 public:
46     IfStmt(ExprNode *cond, StmtNode *thenStmt) : cond(cond), thenStmt(
       thenStmt){};
47     void output(int level);
48     void typeCheck();
49     void genCode();
50     bool checkRet(Type *retType)
51     {
52         if (thenStmt)
53         {
54             return thenStmt->checkRet(retType);
55         }
56         return false;
57     }
58 };
59
60 class IfElseStmt : public StmtNode
61 {
62 private:
63     ExprNode *cond;
64     StmtNode *thenStmt;
65     StmtNode *elseStmt;
66
67 public:
68     IfElseStmt(ExprNode *cond, StmtNode *thenStmt, StmtNode *elseStmt) : cond
       (cond), thenStmt(thenStmt), elseStmt(elseStmt){};
69     void output(int level);
70     void typeCheck();
71     void genCode();
72     bool checkRet(Type *retType)
73     {
74         bool result1 = false;
75         bool result2 = false;
76         if (thenStmt)
77         {
78             result1 = thenStmt->checkRet(retType);
79         }
80         if (elseStmt)
81         {
82             result2 = elseStmt->checkRet(retType);
83         }
```

```
84          return result1 || result2;
85      }
86  };
87
88  class WhileStmt : public StmtNode
89  {
90  private:
91      ExprNode *cond;
92      StmtNode *stmt;
93      BasicBlock *cond_bb;
94      BasicBlock *end_bb;
95
96  public:
97      WhileStmt(ExprNode *cond, StmtNode *stmt = nullptr) : cond(cond), stmt(
            stmt){};
98
99      void output(int level);
100     void typeCheck();
101     void genCode();
102     void setStmt(StmtNode *stmt) { this->stmt = stmt; };
103     BasicBlock *get_cond_bb() { return this->cond_bb; };
104     BasicBlock *get_end_bb() { return this->end_bb; };
105     bool checkRet(Type *retType)
106     {
107         if (stmt)
108         {
109             return stmt->checkRet(retType);
110         }
111         return false;
112     }
113 };
114
115 class BreakStmt : public StmtNode
116 {
117 private:
118     StmtNode *whileStmt;
119
120 public:
121     BreakStmt(StmtNode *whileStmt) { this->whileStmt = whileStmt; };
122     void output(int level);
123     void typeCheck();
124     void genCode();
125     bool checkRet(Type *retType) { return false; }
126 };
127
128 class ContinueStmt : public StmtNode
129 {
130 private:
```

```
131     StmtNode *whileStmt;
132
133 public:
134     ContinueStmt(StmtNode *whileStmt) { this->whileStmt = whileStmt; };
135     void output(int level);
136     void typeCheck();
137     void genCode();
138     bool checkRet(Type *retType) { return false; }
139 };
140
141 class ReturnStmt : public StmtNode
142 {
143 private:
144     ExprNode *retValue;
145
146 public:
147     ReturnStmt(ExprNode *retValue = nullptr) : retValue(retValue){};
148     void output(int level);
149     void typeCheck();
150     void genCode();
151     bool checkRet(Type *retType);
152 };
153
154 class AssignStmt : public StmtNode
155 {
156 private:
157     ExprNode *lval;
158     ExprNode *expr;
159
160 public:
161     AssignStmt(ExprNode *lval, ExprNode *expr) : lval(lval), expr(expr){};
162     void output(int level);
163     void typeCheck();
164     void genCode();
165     bool checkRet(Type *retType) { return false; }
166 };
```

SysY 语言中的 if 语句并没有终结符 then，在 yacc 中我们可以使用%prec 关键字，将终结符 then 的优先级赋给产生式。

五、 语义分析

编译器的语义分析是编译过程中的一个关键阶段，它主要负责理解源代码的语义结构，确保程序在执行时能够按照程序员的意图正确运行。语义分析包括以下几个主要方面：

语义检查：语义分析阶段对源代码进行检查，确保程序的语义是合法的。这包括检查变量的类型、函数参数的匹配、运算符的合法性等。

类型检查：确保变量和表达式的使用符合语言规定的类型系统。类型检查可以防止一些常见的编程错误，例如将字符串赋值给整数变量。

作用域分析：确定变量和标识符的作用域，以便在程序的不同部分正确引用它们。这涉及到识别变量的声明和确定其可见范围。

语法糖转换：有时，编译器会对源代码进行一些语法糖的转换，以简化语法或者将高级语法转换为更基础的形式。这有助于后续阶段的处理。

语义错误报告：如果在语义分析过程中发现错误，编译器会生成相应的错误消息，以便程序员能够及早发现并修复问题。

中间代码生成：在语义分析的同时，可能还会生成中间代码。这是一个抽象的表示，介于源代码和目标代码之间，为后续的优化和代码生成阶段提供基础。

语义分析通常紧跟在词法分析和语法分析之后，而在目标代码生成之前。通过确保程序的语义正确性，语义分析为编译器的后续阶段提供了正确的输入。

LLVM 是一个模块化的、可重用的编译器和工具链的集合，目的是提供一个现代的、基于 SSA 的、能够支持任意静态和动态编译的编程语言的编译策略。在最近几年已经成为表现上能够和 gcc 对标的项目。LLVM IR 是 LLVM 项目中通用的中间代码，作为源语言和体系架构的连接部分，中间代码可以连接不同的架构，为汇编代码的生成提供可移植性。

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。

中间代码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，可以极大的简化编译器的构造。我们设计的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。主要需要完成对流图构造，以及对构造完成后的翻译。

(一) 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。然后输出报错信息并退出即可。

定义相关的 type 类头文件首先增加了 bool 类型的变量，为之后的表达式的判断类型做铺垫：

```
    class Type
{
private:
    int kind;

protected:
    int size;

public:
    enum
    {
        INT,
        VOID,
        FUNC,
        PTR,
        ARRAY,
    };
    Type(int kind) : kind(kind){};
    // virtual ~Type() {};
    virtual std::string toStr() = 0;
    bool isInt() const { return kind == INT; };
    bool isVoid() const { return kind == VOID; };
    bool isFunc() const { return kind == FUNC; };
    bool isArray() const { return kind == ARRAY; };
    int getKind() { return kind; };
    virtual int getSize() = 0;
};

class IntType : public Type
{
private:
    bool constant;

public:
    IntType(int size, bool constant = false)
        : Type(Type::INT), constant(constant) { this->size = size; };
    std::string toStr();
    bool isConstant() { return constant; }
    int getSize() { return size; };
};

class VoidType : public Type
{
public:
    VoidType() : Type(Type::VOID){};
    std::string toStr();
    int getSize() { return size; };
```

```
};

class FunctionType : public Type
{
private:
    Type *returnType;
    std::vector<Type *> paramsType;
    std::vector<SymbolEntry *> paramsSe;

public:
    FunctionType(Type *returnType,
                 std::vector<Type *> paramsType,
                 std::vector<SymbolEntry *> paramsSe)
        : Type(Type::FUNC),
          returnType(returnType),
          paramsType(paramsType),
          paramsSe(paramsSe){};
    Type *getRetType() { return returnType; };
    std::vector<Type *> getParamsType() { return paramsType; };
    std::string toStr();
    std::vector<SymbolEntry *> getParamsSe() { return paramsSe; };
    int getSize() { return size; };
};

class ArrayType : public Type
{
private:
    Type *elementType;
    Type *arrayType = nullptr;
    int length;
    bool constant;

public:
    ArrayType(Type *elementType,
              int length,
              bool constant = false)
        : Type(Type::ARRAY),
          elementType(elementType),
          length(length),
          constant(constant)
    {
        size = elementType->getSize() * length;
    };
    std::string toStr();
    int getLength() const { return length; };
    Type *getEType() const { return elementType; };
    void setArrayType(Type *arrayType) { this->arrayType = arrayType; };
    Type *getArrayType() const { return arrayType; };
```

```
96      int getSize() { return size; };
97  };
98
99  class PointerType : public Type
100 {
101 private:
102     Type *valueType;
103
104 public:
105     PointerType(Type *valueType) : Type(Type::PTR) { this->valueType =
            valueType; };
106     std::string toStr();
107     int getSize() { return size; };
108     Type *getType() const { return valueType; };
109 };
110
111 class TypeSystem
112 {
113 private:
114     static IntType commonInt;
115     static IntType commonBool;
116     static VoidType commonVoid;
117     static IntType commonConstInt;
118
119 public:
120     static Type *intType;
121     static Type *voidType;
122     static Type *boolType;
123     static Type *constIntType;
124 };
```

在此基础上定义 type 类的参数:

```
1   std::string IntType::toStr()
2   {
3       std::ostringstream buffer;
4       buffer << "i" << size;
5       return buffer.str();
6   }
7   std::string FloatType::toStr() {
8       return "float";
9   }
10  std::string VoidType::toStr()
11  {
12      return "void";
13  }
14
15  std::string ArrayType::toStr() {
16      std::vector<std::string> vec;
```

```
17      Type* temp = this;
18      int count = 0;
19      bool flag = false;
20      while (temp && temp->isArray()) {
21          std::ostringstream buffer;
22          if (((ArrayType*)temp)->getLength() == -1) {
23              flag = true;
24          } else {
25              buffer << "[" << ((ArrayType*)temp)->getLength() << " x ";
26              count++;
27              vec.push_back(buffer.str());
28          }
29          temp = ((ArrayType*)temp)->getEType();
30      }
31      std::ostringstream buffer;
32      for (auto it = vec.begin(); it != vec.end(); it++)
33          buffer << *it;
34      if (temp->isInt()) {
35          buffer << "i32";
36      } else if (temp->isFloat()) {
37          buffer << "float";
38      } else {
39          assert(false);  // invalid type
40      }
41      while (count--)
42          buffer << ']';
43      if (flag)
44          buffer << '*';
45      return buffer.str();
46  }
47
48  std::string FunctionType::toStr()
49  {
50      std::ostringstream buffer;
51      buffer << returnType->toStr() << "(";
52      for (auto it = paramsType.begin(); it != paramsType.end(); it++)
53      {
54          buffer << (*it)->toStr();
55          if (it + 1 != paramsType.end())
56              buffer << ", ";
57      }
58      buffer << ')';
59      return buffer.str();
60  }
61
62  std::string PointerType::toStr()
63  {
64      std::ostringstream buffer;
```

```
    buffer << valueType->toStr() << "*";
    return buffer.str();
}
```

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。

语法树的补充

```
    class Node
{
private:
    static int counter;
    int seq;
     Node* next;

protected:
    std::vector<Instruction *> true_list;
    std::vector<Instruction *> false_list;
    static IRBuilder *builder;
    void backPatch(std::vector<Instruction *> &list, BasicBlock *bb, bool
        branch);
    std::vector<Instruction *> merge(std::vector<Instruction *> &list1, std::
        vector<Instruction *> &list2);
public:
    // 结点又指向下一个结点，用以表示表达式声明列表，函数参数列表等若干个表达
        式拼接而成的表达式
    Node *link;
    Node();
    int getSeq() const { return seq; };
    static void setIRBuilder(IRBuilder *ib) { builder = ib; };
    virtual void output(int level) = 0; // 等待子类重载实现
    // 添加link结点
    void setLink(Node *node);
    // 获取当前结点的link结点
    Node *getLink();
    virtual void typeCheck() = 0;
    virtual void genCode() = 0;
    std::vector<Instruction *> &trueList() { return true_list; }
    std::vector<Instruction *> &falseList() { return false_list; }
};
```

输入输出在主函数部分首先定义四个表项分别是输入输出字符和数字，这是按照 SysY 语言特性的输入输出函数来补充的，可以进行字符和数字的输入输出相当于预定义了四个函数，如下所示:

```
    std::vector<Type *> vec;
    std::vector<SymbolEntry *> vec1;
    Type *funcType = new FunctionType(TypeSystem::intType, vec, vec1);
    SymbolTable *st = identifiers;
    while (st->getPrev())
```

```
6           st = st->getPrev();
7       SymbolEntry *se = new IdentifierSymbolEntry(funcType, "getint", st->
            getLevel());
8       st->install("getint", se);
9
10      // getch
11      funcType = new FunctionType(TypeSystem::intType, vec, vec1);
12      st = identifiers;
13      while (st->getPrev())
14          st = st->getPrev();
15      se = new IdentifierSymbolEntry(funcType, "getch", st->getLevel());
16      st->install("getch", se);
17
18      // getarray
19      ArrayType *arr = new ArrayType(TypeSystem::intType, -1);
20      vec.push_back(arr);
21      funcType = new FunctionType(TypeSystem::intType, vec, vec1);
22      st = identifiers;
23      while (st->getPrev())
24          st = st->getPrev();
25      se = new IdentifierSymbolEntry(funcType, "getarray", st->getLevel());
26      st->install("getarray", se);
27
28      // putint
29      vec.clear();
30      vec1.clear();
31      vec.push_back(TypeSystem::intType);
32      funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
33      st = identifiers;
34      while (st->getPrev())
35          st = st->getPrev();
36      se = new IdentifierSymbolEntry(funcType, "putint", st->getLevel());
37      st->install("putint", se);
38
39      // putch
40      vec.clear();
41      vec1.clear();
42      vec.push_back(TypeSystem::intType);
43      funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
44      st = identifiers;
45      while (st->getPrev())
46          st = st->getPrev();
47      se = new IdentifierSymbolEntry(funcType, "putch", st->getLevel());
48      st->install("putch", se);
49
50      // putarray
51      vec.clear();
52      vec1.clear();
```

```
    vec.push_back(TypeSystem::intType);
    arr = new ArrayType(TypeSystem::intType, -1);
    vec.push_back(arr);
    funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
    st = identifiers;
    while (st->getPrev())
        st = st->getPrev();
    se = new IdentifierSymbolEntry(funcType, "putarray", st->getLevel());
    st->install("putarray", se);

    // starttime
    vec.clear();
    vec1.clear();
    funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
    st = identifiers;
    while (st->getPrev())
        st = st->getPrev();
    se = new IdentifierSymbolEntry(funcType, "starttime", st->getLevel());
    st->install("starttime", se);

    // stoptime
    vec.clear();
    vec1.clear();
    funcType = new FunctionType(TypeSystem::voidType, vec, vec1);
    st = identifiers;
    while (st->getPrev())
        st = st->getPrev();
    se = new IdentifierSymbolEntry(funcType, "stoptime", st->getLevel());
    st->install("stoptime", se);
```

函数的输出在函数的部分首先是其输出加上了参数列表，如果其参数不为空的话，需要在括号后输出，然后对函数进行遍历基本块的输出即可：

```
    void Function::output() const
{
    FunctionType *funcType = dynamic_cast<FunctionType *>(sym_ptr->getType())
        ;
    Type *retType = funcType->getRetType();

    std::vector<Type *> params = funcType->getParamsType();
    std::string paramstr;
    std::vector<Type *>::iterator it;
    int i = 0;
    for (it = params.begin(); it != params.end(); it++)
    {
        if (paramstr.size() != 0)
            paramstr = paramstr + ",";
        paramstr = paramstr + " " + (*it)->toStr() + " " + paramsOperand[i]->
            toStr();
```

```
        i++;
    }

    fprintf(yyout, "define %s %s(%s) {\n", retType->toStr().c_str(), sym_ptr
        ->toStr().c_str(), paramstr.c_str());

    // fprintf(yyout, "define %s %s() {\n", retType->toStr().c_str(), sym_ptr
        ->toStr().c_str());
    std::set<BasicBlock *> v;
    std::list<BasicBlock *> q;
    // enrty为函数的入口基本块
    q.push_back(entry);
    v.insert(entry);

    while (!q.empty())
    {
        // bb为这个函数最前面的基本块（第一次位entry）
        auto bb = q.front();
        q.pop_front();
        // 调用当前基本块的output函数
        bb->output();
        // succ为基本块bb的后续基本块构成的向量
        for (auto succ = bb->succ_begin(); succ != bb->succ_end(); succ++)
        {
            // v是一个set，所以会排除掉那些重复的基本块
            if (v.find(*succ) == v.end())
            {
                v.insert(*succ);
                q.push_back(*succ);
            }
        }
    }
    fprintf(yyout, "}\n");
}
```

二元指令输出

```
    void BinaryInstruction::output() const{
    std::string s1, s2, s3, op, type;
    s1 = operands[0]->toStr();
    s2 = operands[1]->toStr();
    s3 = operands[2]->toStr();
    type = operands[0]->getType()->toStr();
    switch (opcode) {
        case ADD:
            if (type == "float") {
                op = "fadd";
            } else {
                op = "add";
```

```
            }
            break;
        case SUB:
            if (type == "float") {
                op = "fsub";
            } else {
                op = "sub";
            }
            break;
        case MUL:
            if (type == "float") {
                op = "fmul";
            } else {
                op = "mul";
            }
            break;
        case DIV:
            if (type == "float") {
                op = "fdiv";
            } else {
                op = "sdiv";
            }
            break;
        case MOD:
            op = "srem";
            break;
        default:
            break;
    }
    fprintf(yyout, "  %s = %s %s %s, %s\n", s1.c_str(), op.c_str(),
            type.c_str(), s2.c_str(), s3.c_str());}
```

语法树的调整与重新输出

语法树的重新调整也是最为重点的部分，首先错误回填函数中需要增加指令的前驱和后继关系，仿照回填函数写的条件错误的回填也是类似的实现，在其后代码调用时逻辑更清楚。

在二元表达式上首先在原先基础上增加了前驱后继关系，然后仿照和运算写出或运算，其也具有短路的特性，当第一个表达式为真时，整个表达式的值为真，第二个表达式不会执行，然后将在这个函数下创建的 falseBB 基本块回填，这也是第二个表达式要插入的位置，然后生成表达式 2 的代码，当前不能确定表达式 2 的 falselist 和两个表达式的 truelist，所以都将其插入到当前节点中，让父节点回填；对于小于大于等指令与其类似，他们的错误和正确列表都由父节点进行回填；对于加减等指令没太大变化，增加了乘除法等。

```
void Node::backPatch(std::vector<Instruction *> &list, BasicBlock *bb,
    bool branch)
{
    if (branch)
        for (auto &inst : list)
        {
```

```
            if (inst->isCond())
                dynamic_cast<CondBrInstruction *>(inst)->setTrueBranch(bb);
            else if (inst->isUncond())
                dynamic_cast<UncondBrInstruction *>(inst)->setBranch(bb);
        }
    else
        for (auto &inst : list)
        {
            if (inst->isCond())
                dynamic_cast<CondBrInstruction *>(inst)->setFalseBranch(bb);
            else if (inst->isUncond())
                dynamic_cast<UncondBrInstruction *>(inst)->setBranch(bb);
        }
}

std::vector<Instruction *> Node::merge(std::vector<Instruction *> &list1, std
    ::vector<Instruction *> &list2)
{
    std::vector<Instruction *> res(list1);
    res.insert(res.end(), list2.begin(), list2.end());
    return res;
}
```

接下来的类型检查都融合在中间代码生成的过程当中，中间代码生成当中进行了相关内容的补充。于是接下来将详细讲述中间代码生成。

六、 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。

(一) 表达式的翻译

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。

对 basicblock 类的定义

```
    class BasicBlock
{
    typedef std::vector<BasicBlock *>::iterator bb_iterator;

private:
    std::vector<BasicBlock *> pred, succ;
    Instruction *head;
    Function *parent;
    int no;

public:
    BasicBlock(Function *);
    BasicBlock();
    ~BasicBlock();
    void insertFront(Instruction *);
    void insertBack(Instruction *);
    void insertBefore(Instruction *, Instruction *);
    void remove(Instruction *);
    bool empty() const { return head->getLink() == head; }
    void output() const;
    bool succEmpty() const { return succ.empty(); };
    bool predEmpty() const { return pred.empty(); };
    void addSucc(BasicBlock *);
    void removeSucc(BasicBlock *);
    void addPred(BasicBlock *);
    void removePred(BasicBlock *);
    int getNo() { return no; };
    Function *getParent() { return parent; };
    Instruction *begin() { return head->getLink(); };
    Instruction *end() { return head; };
    Instruction *rbegin() { return head->getPrev(); };
    Instruction *rend() { return head; };
```

```
33      bb_iterator succ_begin() { return succ.begin(); };
34      bb_iterator succ_end() { return succ.end(); };
35      bb_iterator pred_begin() { return pred.begin(); };
36      bb_iterator pred_end() { return pred.end(); };
37      int getNumOfPred() const { return pred.size(); };
38      int getNumOfSucc() const { return succ.size(); };
39      void genMachineCode(AsmBuilder *);
40  };
```

(二) 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 2 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。

```
1       void Ast::genCode(Unit *unit)
2   {
3       fprintf(stderr, "Ast::genCode\n");
4       // 这个builder应该是唯一的
5       IRBuilder *builder = new IRBuilder(unit);
6       // Node类的静态方法
7       Node::setIRBuilder(builder);
8       root->genCode();
9       }
10  void FunctionDef::genCode()
11  {
12      fprintf(stderr, "FunctionDef::genCode\n");
13      Unit *unit = builder->getUnit();
14      Function *func = new Function(unit, se);
15      BasicBlock *entry = func->getEntry();
16      // set the insert point to the entry basicblock of this function.
17
18      builder->setInsertBB(entry);
19
20      paramno = 0;
21      // 函数参数中间代码生成
22      if (FuncDefParams)
23          FuncDefParams->genCode();
24
25      // 函数体中间代码生成
26      if (stmt)
27          stmt->genCode();
28
29      // 根据基本块的前驱、后继关系进行流图的构造
30      for (auto block = func->begin(); block != func->end(); block++)
31      {
```

```
32          // 获取该块的第一条和最后一条指令
33          Instruction *i = (*block)->begin();
34          Instruction *last = (*block)->rbegin();
35          while (i != last)
36          {
37              if (i->isCond() || i->isUncond())
38              {
39                  // 移除块中跳转指令
40                  (*block)->remove(i);
41              }
42              i = i->getLink();
43          }
44          // 有条件跳转
45          if (last->isCond())
46          {
47              BasicBlock *truebranch, *falsebranch;
48              truebranch =
49                  dynamic_cast<CondBrInstruction *>(last)->getTrueBranch();
50              falsebranch =
51                  dynamic_cast<CondBrInstruction *>(last)->getFalseBranch();
52              if (truebranch->empty())
53              {
54                  new RetInstruction(nullptr, truebranch);
55              }
56              else if (falsebranch->empty())
57              {
58                  new RetInstruction(nullptr, falsebranch);
59              }
60              (*block)->addSucc(truebranch);
61              (*block)->addSucc(falsebranch);
62              truebranch->addPred(*block);
63              falsebranch->addPred(*block);
64          }
65          // 无条件跳转
66          else if (last->isUncond())
67          {
68              // 获取要跳转的基本块
69              BasicBlock *dst =
70                  dynamic_cast<UncondBrInstruction *>(last)->getBranch();
71              // 跳转块链接
72              (*block)->addSucc(dst);
73              dst->addPred(*block);
74
75              if (dst->empty())
76              {
77                  if (((FunctionType *)(se->getType()))->getRetType() ==
78                      TypeSystem::intType)
79                      new RetInstruction(new Operand(new ConstantSymbolEntry(
```

```
                                                TypeSystem::intType, 0)),
                                            dst);
                else if (((FunctionType *)(se->getType()))->getRetType() ==
                         TypeSystem::voidType)
                    new RetInstruction(nullptr, dst);
            }
        }
        // 最后一条语句不是返回以及跳转
        else if (!last->isRet())
        {
            if (((FunctionType *)(se->getType()))->getRetType() ==
                TypeSystem::voidType)
            {
                new RetInstruction(nullptr, *block);
            }
        }
    }
}
// 函数调用中间代码生成
void FuncCallExpr::genCode()
{
    fprintf(stderr, "FuncCallExpr::genCode\n");

    BasicBlock *bb = builder->getInsertBB();
    std::vector<Operand *> operands;
    ExprNode *temp = param;
    while (temp)
    {
        // 参数中间代码生成
        temp->genCode();
        operands.push_back(temp->getOperand());
        temp = ((ExprNode *)temp->getLink());
    }
    bb = builder->getInsertBB();
    // fprintf("sssssssss%s")
    new CallInstruction(dst, symbolEntry, operands, bb);
}
```

二元表达式运算中间代码生成

```
    // 二元运算表达式中间代码生成
void BinaryExpr::genCode()
{
    fprintf(stderr, "BinaryExpr::genCode\n");
    // 获取新的指令要插入的基本块insertBB
    BasicBlock *bb = builder->getInsertBB();
    // 获取当前所处的函数
    Function *func = bb->getParent();
    // fprintf(stderr, "op: %d", op);
```

```
    if (op == AND)
    {
        // fprintf(stderr, "Come in AND\n");
        expr1->genCode();
        if (expr1->getOperand() == nullptr)
        {
            fprintf(stderr, "BinaryExpr can't be void type\n");
            assert(expr1->getOperand() != nullptr);
        }
        Instruction *test = bb->rbegin();
        // 考虑if(2&&3),第一个表达式为算术表达式
        if (!test->isCond() && !test->isUncond())
        {
            int opcode = CmpInstruction::NE;
            Operand *src1 = expr1->getOperand();
            Operand *src2 = src0_const0;
            // dst比较指令目的寄存器
            SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
            Operand *dst = new Operand(tse);
            Operand *n1 = src1;
            // bool转int的类型转换
            if (src1->getType() == TypeSystem::boolType)
            {
                SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType
                    , SymbolTable::getLabel());
                n1 = new Operand(s);
                new Bool2IntInstruction(n1, src1, bb); // 零拓展
            }
            // 跟0比较，判断表达式真假
            new CmpInstruction(opcode, dst, n1, src2, bb);
            Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst,
                bb);
            // trueList和falseList均为目标地址未确定的跳转指令的向量
            expr1->trueList().push_back(temp);
            expr1->falseList().push_back(temp);
        }
        // new了一个新的基本块trueBB，如果第一个表达式的结果为真，那么跳到这
            个基本块
        BasicBlock *trueBB = new BasicBlock(func);

        backPatch(expr1->trueList(), trueBB, true);
        // 设置当前的新的指令（第二个表达式对应的指令）要插入到的基本块为
            trueBB
        builder->setInsertBB(trueBB); // set the insert point to the trueBB
            so that intructions generated by expr2 will be inserted into it.
        trueBB->addPred(bb);            // 没用
        bb->addSucc(trueBB);
```

```
52
53          expr2->genCode();
54          if (expr2->getOperand() == nullptr)
55          {
56              fprintf(stderr, "BinaryExpr can't be void type\n");
57              assert(expr2->getOperand() != nullptr);
58          }
59          bb = builder->getInsertBB();
60          test = bb->rbegin();
61          if (!test->isCond() && !test->isUncond())
62          {
63              int opcode = CmpInstruction::NE;
64              Operand *src1 = expr2->getOperand();
65              Operand *src2 = src0_const0;
66              SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
67              Operand *dst = new Operand(tse);
68              Operand *n1 = src1;
69              if (src1->getType() == TypeSystem::boolType)
70              {
71                  SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType
                        , SymbolTable::getLabel());
72                  n1 = new Operand(s);
73                  new Bool2IntInstruction(n1, src1, bb);
74              }
75              new CmpInstruction(opcode, dst, n1, src2, bb);
76              Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst,
                    bb);
77              expr2->trueList().push_back(temp);
78              expr2->falseList().push_back(temp);
79          }
80
81          true_list = expr2->trueList();
82          false_list = merge(expr1->falseList(), expr2->falseList());
83      }
84      else if (op == OR)
85      {
86          // fprintf(stderr, "Come in OR\n");
87          expr1->genCode();
88          if (expr1->getOperand() == nullptr)
89          {
90              fprintf(stderr, "BinaryExpr can't be void type\n");
91              assert(expr1->getOperand() != nullptr);
92          }
93          Instruction *test = bb->rbegin();
94          if (!test->isCond() && !test->isUncond())
95          {
96              int opcode = CmpInstruction::NE;
```

```
97              Operand *src1 = expr1->getOperand();
98              Operand *src2 = src0_const0;
99              SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
100             Operand *dst = new Operand(tse);
101             Operand *n1 = src1;
102             if (src1->getType() == TypeSystem::boolType)
103             {
104                 SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType
                        , SymbolTable::getLabel());
105                 n1 = new Operand(s);
106                 new Bool2IntInstruction(n1, src1, bb); // 零拓展
107             }
108             new CmpInstruction(opcode, dst, n1, src2, bb);
                                        // 比较
109             Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst,
                    bb); // 条件跳转
110             expr1->trueList().push_back(temp);
111             expr1->falseList().push_back(temp);
112         }
113
114         BasicBlock *ntrueBB = new BasicBlock(func); // if the result of lhs
                is true, jump to the trueBB.
115
116         // 与AND逻辑区别
117         backPatch(expr1->falseList(), ntrueBB, false);
118
119         builder->setInsertBB(ntrueBB); // set the insert point to the trueBB
                so that intructions generated by expr2 will be inserted into it.
120         ntrueBB->addPred(bb);           // 没用
121         bb->addSucc(ntrueBB);
122
123         expr2->genCode();
124         if (expr2->getOperand() == nullptr)
125         {
126             fprintf(stderr, "BinaryExpr can't be void type\n");
127             assert(expr2->getOperand() != nullptr);
128         }
129         bb = builder->getInsertBB();
130         test = bb->rbegin();
131         if (!test->isCond() && !test->isUncond())
132         {
133             int opcode = CmpInstruction::NE;
134             Operand *src1 = expr2->getOperand();
135             Operand *src2 = src0_const0;
136             SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                        SymbolTable::getLabel());
137             Operand *dst = new Operand(tse);
```

```
138             Operand *n1 = src1;
139             if (src1->getType() == TypeSystem::boolType)
140             {
141                 SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType
                        , SymbolTable::getLabel());
142                 n1 = new Operand(s);
143                 new Bool2IntInstruction(n1, src1, bb);
144             }
145             new CmpInstruction(opcode, dst, n1, src2, bb);
146             Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst,
                    bb);
147             expr2->trueList().push_back(temp);
148             expr2->falseList().push_back(temp);
149         }
150 
151         // 与AND逻辑区别
152         false_list = expr2->falseList();
153         true_list = merge(expr1->trueList(), expr2->trueList());
154     }
155     else if (op >= LESS && op <= NE) // LESS, MORE, LEQ, MEQ, EQ, NE
156     {
157         // fprintf(stderr, "Come in LESS~NE\n");
158         expr1->genCode(); // 生成子表达式中间代码
159         expr2->genCode();
160         // 类型检查
161         if (expr1->getOperand() == nullptr)
162         {
163             fprintf(stderr, "BinaryExpr can't be void type\n");
164             assert(expr1->getOperand() != nullptr);
165         }
166         if (expr2->getOperand() == nullptr)
167         {
168             fprintf(stderr, "BinaryExpr can't be void type\n");
169             assert(expr2->getOperand() != nullptr);
170         }
171         // 通过getOperand函数得到子表达式的目的操作数
172         Operand *src1 = expr1->getOperand();
173         Operand *src2 = expr2->getOperand();
174 
175         int opcode = -1;
176         switch (op)
177         {
178         case LESS:
179             opcode = CmpInstruction::L;
180             break;
181         case MORE:
182             opcode = CmpInstruction::G;
183             break;
```

```
184             case LEQ:
185                 opcode = CmpInstruction::LE;
186                 break;
187             case MEQ:
188                 opcode = CmpInstruction::GE;
189                 break;
190             case EQ:
191                 opcode = CmpInstruction::E;
192                 break;
193             case NE:
194                 opcode = CmpInstruction::NE;
195                 break;
196             }
197
198             Operand *n1 = src1, *n2 = src2;
199             // bool -> int 隐式类型转换
200             if (src1->getType() == TypeSystem::boolType)
201             {
202                 SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
203                 n1 = new Operand(s);
204                 new Bool2IntInstruction(n1, src1, bb);
205             }
206             if (src2->getType() == TypeSystem::boolType)
207             {
208                 SymbolEntry *s2 = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
209                 n2 = new Operand(s2);
210                 new Bool2IntInstruction(n2, src2, bb);
211             }
212
213             SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
214             dst = new Operand(tse); // 目的寄存器
215             new CmpInstruction(opcode, dst, n1, n2, bb);
216         }
217         else if (op >= ADD && op <= MOD)
218         {
219             // fprintf(stderr, "Come in ADD~MOD\n");
220             expr1->genCode();
221             expr2->genCode();
222             if (expr1->getOperand() == nullptr)
223             {
224                 fprintf(stderr, "BinaryExpr can't be void type\n");
225                 assert(expr1->getOperand() != nullptr);
226             }
227             if (expr2->getOperand() == nullptr)
228             {
```

```
229             fprintf(stderr, "BinaryExpr can't be void type\n");
230             assert(expr2->getOperand() != nullptr);
231         }
232         Operand *src1 = expr1->getOperand();
233         Operand *src2 = expr2->getOperand();
234         int opcode;
235         switch (op)
236         {
237         case ADD:
238             opcode = BinaryInstruction::ADD;
239             break;
240         case SUB:
241             opcode = BinaryInstruction::SUB;
242             break;
243         case MUL:
244             opcode = BinaryInstruction::MUL;
245             break;
246         case DIV:
247             opcode = BinaryInstruction::DIV;
248             break;
249         case MOD:
250             opcode = BinaryInstruction::MOD;
251             break;
252         }
253 
254         Operand *n1 = src1, *n2 = src2;
255         if (src2->getType() == TypeSystem::boolType)
256         {
257             SymbolEntry *s2 = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
258             n2 = new Operand(s2);
259             new Bool2IntInstruction(n2, src2, bb);
260         }
261         if (src1->getType() == TypeSystem::boolType)
262         {
263             SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
264             n1 = new Operand(s);
265             new Bool2IntInstruction(n1, src1, bb);
266         }
267         if (op == ADD || op == MUL)
268         {
269             new BinaryInstruction(opcode, dst, src2, src1, bb);
270         }
271         else
272         {
273             new BinaryInstruction(opcode, dst, src1, src2, bb);
274         }
```

```
275     }
276     // fprintf(stderr, "NOT Come in\n");
277 }
```

加载值中间代码生成

```
1   // ID中间代码生成（加载值）
2 void Id::genCode()
3 {
4     fprintf(stderr, "Id::genCode\n");
5     BasicBlock *bb = builder->getInsertBB();
6     Operand *addr = dynamic_cast<IdentifierSymbolEntry *>(symbolEntry)->
          getAddr();
7     Type *symboleType = this->getSymPtr()->getType();
8     if (this->getSymPtr()->getType()->isInt())
9         new LoadInstruction(dst, addr, bb);
10    else if (symboleType->isArray())
11    {
12        // b=2+a[2][3]
13        // arrIdx代表[2][3]这种，实际存在形式为2->3
14        if (arrIdx)
15        {
16            fprintf(stderr, "AAAAAAAAAAAAAAAAAAAAId::genCode->arrIdx\n");
17            // type1为数组类型[2][3],type为元素类型[3]
18            Type *type = ((ArrayType *)symboleType)->getEType();
19            Type *type1 = symboleType;
20            // tempSrc: 要加载的标识符地址, tempDst: 目的操作数
21            Operand *tempSrc = addr;
22            Operand *tempDst = dst;
23            // idx就是2->3
24            ExprNode *idx = arrIdx;
25            bool flag = false;
26            bool pointer = false; // 为true表示dst为指针类型->用于函数参数
27            bool firstFlag = true;
28            while (true)
29            {
30                fprintf(stderr, "%d\n", ((ArrayType *)type1)->getLength());
31                // 函数参数, 像a[2][],a[]
32                if (((ArrayType *)type1)->getLength() == -1)
33                {
34                    fprintf(stderr, "(ArrayType *)type1)->getLength() == -1\n
                          ");
35                    Operand *dst1 = new Operand(new TemporarySymbolEntry(new
                          PointerType(type), SymbolTable::getLabel()));
36                    tempSrc = dst1;
37                    new LoadInstruction(dst1, addr, bb);
38                    flag = true;
39                    firstFlag = false;
40                }
```

```
41                  // idx为空, 也就是表示数组是定义而非使用, 生成ArInstruction
42                  // idx=0, paramFirst = false跳出
43                  // a[0]
44                  if (!idx)
45                  {
46                      fprintf(stderr, "!idx\n");
47                      Operand *dst1 = new Operand(new TemporarySymbolEntry(new
                            PointerType(type), SymbolTable::getLabel()));
48                      Operand *idx = new Operand(new ConstantSymbolEntry(
                            TypeSystem::intType, 0));
49                      new ArInstruction(dst1, tempSrc, idx, bb);
50                      tempDst = dst1;
51                      pointer = true;
52                      break;
53                  }
54                  // 以下必定是idx有值, idx=2->3
55                  // 带有此处flag (Ar指令中paramFirst) 标志, 若为函数参数中的[]
                        维度(参数最后一维), 该标志为true, 否则为false
56                  idx->genCode();
57                  auto gep = new ArInstruction(tempDst, tempSrc, idx->
                        getOperand(), bb, flag);
58                  if (!flag && firstFlag)
59                  {
60                      gep->setFirst();
61                      firstFlag = false;
62                  }
63                  if (flag)
64                      flag = false;
65                  if (type == TypeSystem::intType ||
66                      type == TypeSystem::constIntType)
67                      break;
68                  type = ((ArrayType *)type)->getEType();
69                  type1 = ((ArrayType *)type1)->getEType();
70                  tempSrc = tempDst;
71                  tempDst = new Operand(new TemporarySymbolEntry(
72                      new PointerType(type), SymbolTable::getLabel()));
73                  idx = (ExprNode *)(idx->getLink());
74              }
75              dst = tempDst;
76              // 如果是右值还需要一条load
77              if (!left && !pointer)
78              {
79                  Operand *dst1 = new Operand(new TemporarySymbolEntry(
80                      TypeSystem::intType, SymbolTable::getLabel()));
81                  new LoadInstruction(dst1, dst, bb);
82                  dst = dst1;
83              }
84          }
```

```
85          else
86          {
87              if (((ArrayType *)symboleType)->getLength() == -1)
88              {
89                  Operand *dst1 = new Operand(new TemporarySymbolEntry(
90                      new PointerType(
91                          ((ArrayType *)symboleType)->getEType()),
92                      SymbolTable::getLabel()));
93                  new LoadInstruction(dst1, addr, bb);
94                  dst = dst1;
95              }
96              else
97              {
98                  Operand *idx = new Operand(
99                      new ConstantSymbolEntry(TypeSystem::intType, 0));
100                 auto gep = new ArInstruction(dst, addr, idx, bb);
101                 gep->setFirst();
102             }
103         }
104     }
105 }
```

在标识符定义处，由于有多个标识符，所以设置一个 for 循环，内部代码与原先类似，在全局变量处增加了查找，如果可以找到就直接生成代码，否则为其赋值一个常量，如果是局部变量的话就分配内存就可以了，后面的 for 循环是生成局部变量的代码：

```
1     void DeclStmt::genCode()
2 {
3     fprintf(stderr, "DeclStmt::genCode\n");
4     IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry *>(id->
          getSymPtr());
5     // 全局变量
6     if (se->isGlobal())
7     {
8         Operand *addr;
9         SymbolEntry *addr_se;
10        addr_se = new IdentifierSymbolEntry(*se); // 地址
11        addr_se->setType(new PointerType(se->getType()));
12        addr = new Operand(addr_se);
13        se->setAddr(addr);
14        Instruction *g;
15        if (expr != nullptr)
16        {
17            if (expr->isInitArr())
18            {
19                int n = se->getType()->getSize() / 32;
20                int *p = se->getArrayValue();
21                std::vector<Operand *> v;
22                BasicBlock *bb = builder->getUnit()->getGlobalBB();
```

```
                for (int i = 0; i < n; i++)
                {
                    ConstantSymbolEntry *temp = new ConstantSymbolEntry(
                        TypeSystem::intType, p[i]);
                    Operand *t = new Operand(temp);
                    v.push_back(t);
                }
                g = new GlobalInstruction(new Operand(id->getSymPtr()), v, se
                    , bb);
            }
            else
            {
                expr->genCode();
                // 通通化为const，让等号右侧返回一个定值
                // 为了应对单目表达式（例如-1），为了获取表达式的值直接转为
                    const类型，
                ConstantSymbolEntry *se_const = new ConstantSymbolEntry(
                    TypeSystem::intType, expr->getOperand()->getEntry()->
                    getValue());
                Operand *n = new Operand(se_const);
                BasicBlock *bb = builder->getUnit()->getGlobalBB();
                std::vector<Operand *> v;
                v.push_back(n);
                g = new GlobalInstruction(new Operand(id->getSymPtr()), v, se
                    , bb);
            }
        }
        else
        {
            fprintf(stderr, "DeclStmt::genCode_expr = nullptr");
            BasicBlock *bb = builder->getUnit()->getGlobalBB();
            Operand *src0 = src0_const0;
            // 全局变量默认赋值0
            std::vector<Operand *> v;
            v.push_back(src0);
            g = new GlobalInstruction(new Operand(id->getSymPtr()), v, se, bb
                );
        }
        global.push_back(g);
    }
    else if (se->isLocal() || se->isParam())
    {
        fprintf(stderr, "44444444444444444se->isLocal() || se->isParam()\n");
        Function *func = builder->getInsertBB()->getParent();
        BasicBlock *entry = func->getEntry();
        Instruction *alloca;
        Operand *addr;
        SymbolEntry *addr_se;
```

```
64          Type *type;
65          type = new PointerType(se->getType());
66          addr_se = new TemporarySymbolEntry(type, SymbolTable::getLabel());
67          addr = new Operand(addr_se);
68          alloca = new AllocaInstruction(addr, se); // allocate space for local
                id in function stack.
69          entry->insertFront(alloca);                 // allocate instructions
                should be inserted into the begin of the entry block.
70          se->setAddr(addr);                          // set the addr operand in
                symbol entry so that we can use it in subsequent code generation.
71
72          if (se->isParam())
73          {
74              fprintf(stderr, "%s: se->isParam()\n", se->toStr().c_str());
75              // store参数的值到刚刚分配的空间中
76              type = se->getType();
77              SymbolEntry *s = new TemporarySymbolEntry(type, SymbolTable::
                    getLabel());
78              Operand *src = new Operand(s);
79              BasicBlock *bb = builder->getInsertBB();
80              new StoreInstruction(addr, src, bb, paramno);
81              paramno++;
82              func->pushParamsOperand(src);
83          }
84
85          if (expr != nullptr)
86          {
87              fprintf(stderr, "44444444444444444expr != nullptr\n");
88              if (expr->isInitArr())
89              {
90                  fprintf(stderr, "444444444444444444expr->isInitArr()\n");
91                  Operand *init = nullptr;
92                  BasicBlock *bb = builder->getInsertBB();
93                  // temp遍历{}中每一个赋值
94                  ExprNode *temp = expr;
95                  std::stack<ExprNode *> Arr_stack;
96                  std::vector<int> idx;
97                  idx.push_back(0);
98                  //{{2,3},2,3+2,4}那么{2,3}->2->(3+2)->4
99                  //{2,3,4}
100                 // 循环遍历初始化表达式中的每个元素
101                 // 使用一个堆栈来跟踪初始化表达式中的当前位置，一个向量来跟踪
                        被初始化元素的索引
102                 while (temp)
103                 {
104                     fprintf(stderr, "4444444444444444444while\n");
105                     // if 语句检查当前元素是否为数组，如果是，则将其压入堆栈
                            并继续遍历
```

```
106                     // 第一次一定会进入到下面这个if，因为一定是一个{}在最外
                            面，每次遇到新的{}都会idx压入
107                     if (temp->isInitArr())
108                     {
109                         fprintf(stderr, " if (temp->isInitArr())\n");
110                         Arr_stack.push(temp);
111                         idx.push_back(0);
112                         temp = ((InitArray *)temp)->getExpr();
113                         continue;
114                     }
115                     else
116                     {
117                         fprintf(stderr, "4444444444444444444else\n");
118                         temp->genCode();
119                         // 获取数组的子元素的类型（如果是多维数组，type为
                                array，否则为int）
120                         /*
121                         a[4][2][3]
122                         arraytype(4,arraytype(2,arraytype(3,int)))
123                         */
124                         Type *type = ((ArrayType *)(se->getType()))->getEType
                                ();
125                         // tempSrc赋初值为标识符的地址
126                         Operand *tempSrc = addr;
127                         Operand *tempDst;
128                         Operand *index;
129                         bool flag = true;
130                         int i = 1;
131                         // 如果不是，则为当前元素生成代码，然后使用一系列
                                ArInstruction和StoreInstruction将元素存储到内存中
                                的相应位置
132                         // 像是a[4][2][3],我们一维一维地推进寻址，tempdst由
                                [2][3]数组指针类型->[3]数组指针类型->
133                         // int指针类型，每一维对应一条Ar指令
134                         // Ar指令中fist表示第一维，last表示最后一维,index表示
                                索引
135                         while (true)
136                         {
137                             tempDst = new Operand(new TemporarySymbolEntry(
                                    new PointerType(type), SymbolTable::getLabel
                                    ()));
138                             index = (new Constant(new ConstantSymbolEntry(
                                    TypeSystem::intType, idx[i++])))->getOperand
                                    ();
139                             // new一条数组指令,first表示是第一个
140                             // a[index1]->dst(二维数组);a[index1][index2]->
                                    dst(一维);a[index1][index2][index3]->dst(int)
141                             auto gep = new ArInstruction(tempDst, tempSrc,
```

```
                                index, bb);
142                         // init初始为空
143                         gep->setInit(init);
144                         // 第一维
145                         if (flag)
146                         {
147                             gep->setFirst();
148                             flag = false;
149                         }
150                         // 判断最后一维并设置last标志位
151                         if (type == TypeSystem::intType || type ==
                                TypeSystem::constIntType)
152                         {
153                             gep->setLast();
154                             init = tempDst;
155                             break;
156                         }
157                         type = ((ArrayType *)type)->getEType();
158                         tempSrc = tempDst;
159                     }
160                     new StoreInstruction(tempDst, temp->getOperand(), bb)
                            ;
161                 }
162                 while (true)
163                 {
164                     if (temp->getLink())
165                     {
166                         temp = (ExprNode *)(temp->getLink());
167                         idx[idx.size() - 1]++;
168                         break;
169                     }
170                     else
171                     {
172                         temp = Arr_stack.top();
173                         Arr_stack.pop();
174                         idx.pop_back();
175                         if (Arr_stack.empty())
176                             break;
177                     }
178                 }
179                 if (Arr_stack.empty())
180                     break;
181             }
182         }
183         else
184         {
185             fprintf(stderr, "444444444444444444else\n");
186             expr->genCode();
```

```
                BasicBlock *bb = builder->getInsertBB();
                Operand *src = expr->getOperand();
                new StoreInstruction(addr, src, bb);
            }
        }
    }
    // 串接后续函数参数/变量声明
    if (this->getLink())
        this->getLink()->genCode();
}
```

于 while 循环生成代码时，首先在当前函数下生成三个基本块和前驱后继关系，并将循环体填入到正确跳转的位置，将结束的位置填入到错误跳处，然后生成循环体，并设置后续插入节点；接下来常量的声明与刚才类似，只不过对于全局变量是一定能查找到的，就不用条件判断了，对于局部变量因为必须先赋值，所以就得将其储存，如下为 while 循环的代码：

```
void WhileStmt::genCode()
{
    fprintf(stderr, "WhileStmt::genCode\n");
    Function *func;
    BasicBlock *cond_bb, *loop_bb, *end_bb, *bb;

    bb = builder->getInsertBB();
    func = bb->getParent();
    cond_bb = new BasicBlock(func);
    loop_bb = new BasicBlock(func);
    end_bb = new BasicBlock(func);

    this->cond_bb = cond_bb;
    this->end_bb = end_bb;

    bb->addSucc(cond_bb);
    cond_bb->addPred(bb);

    builder->setInsertBB(cond_bb);
    cond->genCode();
    new UncondBrInstruction(cond_bb, bb);

    bb = builder->getInsertBB();
    Instruction *test = bb->rbegin();
    if (!test->isCond() && !test->isUncond())
    {
        Operand *src1 = cond->getOperand();
        // fprintf(stderr, "ttttttttttttt%s", src1->getType()->toStr().c_str
            ());
        Operand *n = src1;
        if (src1->getType() == TypeSystem::intType)
        {
            // fprintf(stderr, "hhhhhhhhhh");
```

```
            int opcode = CmpInstruction::NE;
            Operand *src2 = src0_const0;
            SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
            Operand *dst = new Operand(tse);
            new CmpInstruction(opcode, dst, src1, src2, bb);
            n = dst;
        }
        // fprintf(stderr, "ppppppppppppp%s", n->getType()->toStr().c_str())
            ;
        Instruction *temp = new CondBrInstruction(nullptr, nullptr, n, bb);
        cond->trueList().push_back(temp);
        cond->falseList().push_back(temp);
    }

    backPatch(cond->trueList(), loop_bb, true);
    backPatch(cond->falseList(), end_bb, false);
    bb = builder->getInsertBB();

    builder->setInsertBB(loop_bb);
    if (stmt != nullptr)
        stmt->genCode();
    loop_bb = builder->getInsertBB();
    new UncondBrInstruction(cond_bb, loop_bb);

    builder->setInsertBB(end_bb);
}
```

类型检查接下来是类型检查部分，对于根节点直接递归调用即可，对于二元表达式首先判断两个子式是否为 void，如果是 void 类型直接报错退出，接下来判断其是否相等，不等的话也直接报错退出。然后是条件判断的地方需要看其是否是布尔值，如果不是就报错退出，对于 if、ifelse、while 都是如此，其内容直接递归判断；对于 seqnode 为递归调用两个子式；对于返回值类型需要检查其是否为空；对于赋值语句需要判断左右两类型是否相同；单目运算其类型也不能是 void，在这里展示二元表达式的判断，其余代码已经提交到 gitlab 上，就不在这里展示了：

```
    void Id::typeCheck()
{
}
// 二元表达式的类型检查
void BinaryExpr::typeCheck()
{
    expr1->typeCheck();
    expr2->typeCheck();
}
// 一元表达式的类型检查
void UnaryExpr::typeCheck()
{
    expr->typeCheck();
}
```

```
15 void FuncCallExpr::typeCheck()
16 {
17 }
```

七、 中间代码优化

中间代码优化是编译器中的一个关键阶段，其目标是改进程序的性能、减小代码体积、或者在不改变程序语义的前提下使生成的目标代码更有效。中间代码优化通常在生成的中间代码上进行，因为中间代码提供了一个抽象的表示形式，更容易进行各种优化而不涉及目标机器的细节。优化的质量和效果取决于编译器的设计和实现。

(一) 公共子表达式消除

主要的思想在于基本块的合并

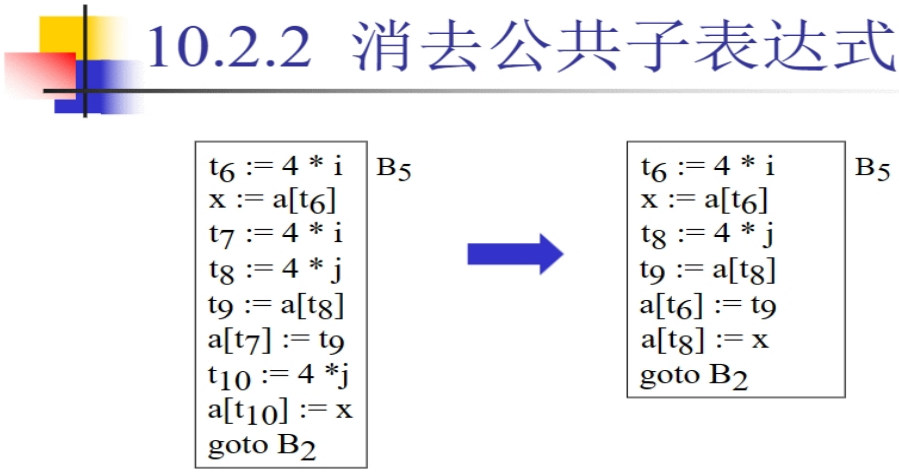

图 1: 公共子表达式删除

```
1  void BlockMerge::findBLocks(Function *func) {
2  for (auto bb : func->getBlockList()) {
3      /*
4       * 如果块内存在条件跳转指令，说明该块的后继块可能不止一个，这里不作合并。
5       * 不过此处如果实现了其他优化，比如常量传播，进一步导致条件判断恒真/假，
6       * 可能会发现有很多分支实际上并不可达，此时即使该块含有cond指令，实际上
7       * 仍然是可以考虑与其后继块进行合并的。
8       * 另外，此处如果实现了phi指令，同样需要根据phi指令的具体实现进一步考虑
```

```
9           * 其他情况。
10          */
11         // TODO: 1. 检查块内是否存在cond指令，后继块数目是否为1
12 BasicBlock *block = bb;
13         int succ_num = block->getNumOfSucc();
14         if(succ_num>1){
15             continue;
16         }
17 
18         mergeList.clear();
19 
20         // 依据控制流持续向后合并，直至存在块不可合并
21         while (true) {
22             // TODO: 2. 检查后继块是否可以合并，包括后继块的前驱块数目等
23             bool can_merge = 0;
24             // 获取block的后继块succ;
25             BasicBlock* succ;
26 if (succ->getNumOfPred() == 1) {
27         can_merge = true;
28         break;
29     }
30             if (can_merge) {
31                 mergeList.push_back(succ);
32                 block = succ;
33             } else {
34                 break;
35             }
36         }
37         // TODO: 3. 合并基本块
38 
39         if (mergeList.size() > 0)
40             merge(func, bb);
41     }
42 }
```

(二) 复制传播

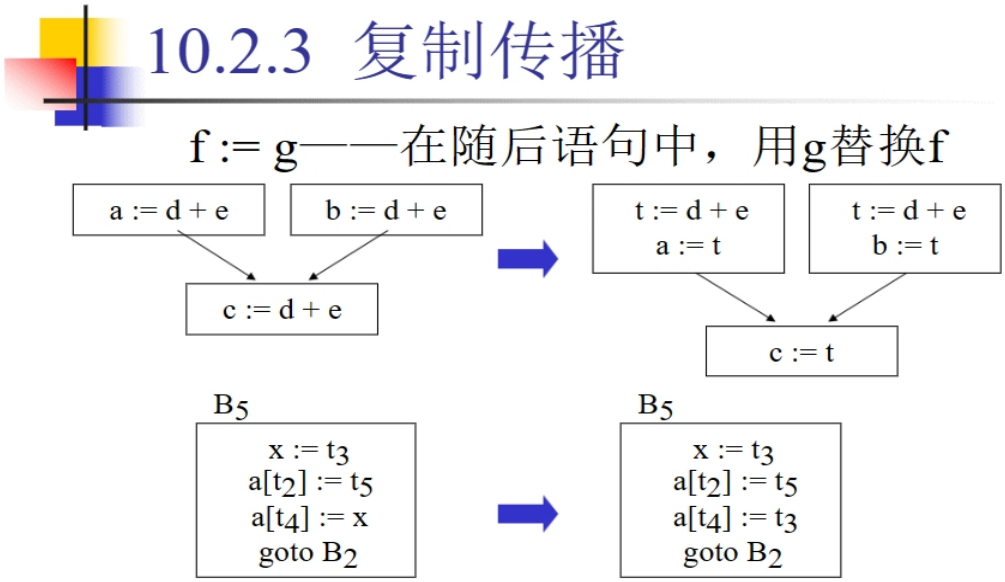

图 2: 复制传播

```
    bool IRComSubExprElim::removeGlobalCSE(Function *func)
{
    // 计算gen、kill、in和out
    calGenKill(func);
    calInOut(func);

    // 标记是否发生了变化，用于判断是否需要继续迭代
    bool changed = false;

    // 获取函数的基本块列表
    std::vector<BasicBlock*> blocks = func->getBlockList();
    // 遍历每个基本块
    for (BasicBlock *bb : blocks)
    {
        // 获取基本块内的指令列表
        std::vector<Instruction*> instructions = bb->getInstructionList();
        // 遍历每条指令
        for (Instruction *inst : instructions)
        {
            // TODO: 根据计算出的gen、kill、in和out进行全局公共子表达式消除的
                逻辑
            // 这可能涉及到比较gen、kill、in和out集合，判断是否可以进行子表达
                式消除

            // 示例：假设gen和in集合相同，则可以考虑消除子表达式
```

```
            if (genBB == inBB)
            {
                // 进行全局公共子表达式消除的操作
                // 根据实际情况修改
                inst->remove();
                changed = true;
            }
        }
    }
    // 如果发生了变化，可能需要继续迭代，根据具体情况决定是否返回true
    return changed;
}
```

(三) 无用代码删除

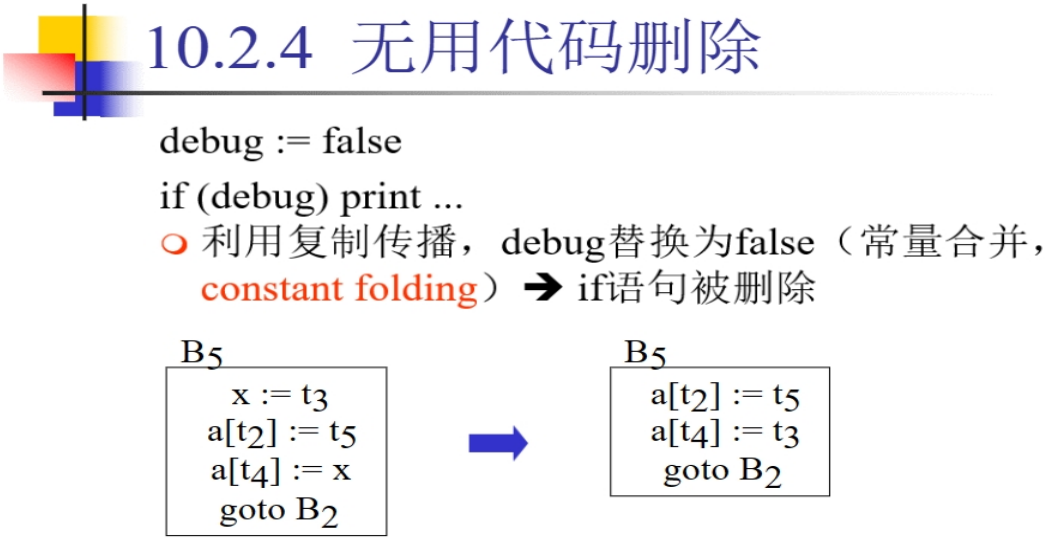

图 3: 无用代码删除

```
void BlockMerge::merge(Function *func, BasicBlock *start) {
// TODO: 1. 处理所有待合并块之间的联系，包括删除冗余的可合并块之间的跳转等;
for (auto bb : mergeList) {
    std::vector<Instruction *> mergeInst = {};
    auto head = bb->end();
    for (auto instr = head->getNext(); instr != head;
         instr = instr->getNext()) {
        // 此处需要补充对指令的判断与处理
        mergeInst.push_back(instr);
    }
```

```
        // TODO: 2. 在合并留下来的唯一块中插入所有被合并块中留下的指令;
for (auto mergedInstr : mergeInst) {
    // 插入指令到合并后的块中
    start->insertBack(mergedInstr);
     //start->getInstList().push_back(mergedInstr);
}
        // TODO: 3. 维护块之间的前驱后继关系, 将被合并的块从容器中移除。
        func->remove(bb);
    }
}
```

(四) 循环的外提

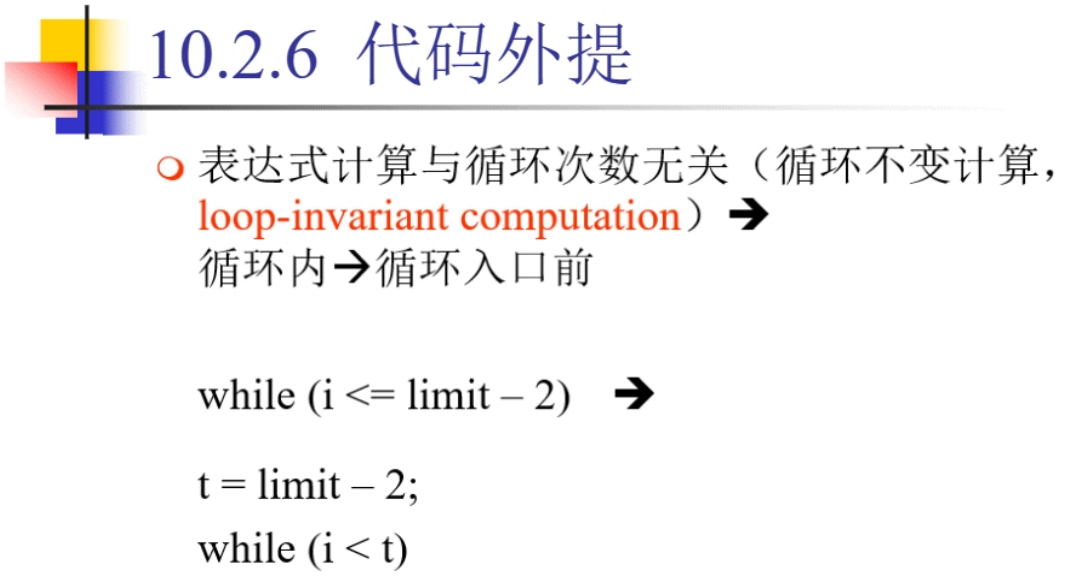

图 4: 循环代码外提

```
    bool IRComSubExprElim::localCSE(Function *func)
{
    bool result = true;
    std::vector<Expr> exprs;
    for (auto block = func->begin(); block != func->end(); block++)
    {
        exprs.clear();
        for (auto inst = (*block)->begin(); inst != (*block)->end(); inst =
            inst->getNext())
        {
            if (skip(inst))
                continue;
            auto preInstIt = std::find(exprs.begin(), exprs.end(), Expr(inst)
                );
```

```
13              if (preInstIt != exprs.end())
14              {
15                  //todo
16                  // 获取前一个指令
17      Instruction* preInst = inst->getPrev();
18
19      // 获取当前指令的def
20      Operand* currentDef = inst->getDef();
21
22      // 获取前一个指令的def
23      Operand* preDef = preInst->getDef();
24
25      // 替换对当前指令def的use成为对前一个指令def的use
26      inst->replaceUse(currentDef, preDef);
27
28      // 删除当前指令
29      inst->remove();
30
31              }
32
33          }
34      }
35      return result;
36  }
```

八、 目标代码生成

在中间代码生成之后，大家可以对中间代码进行自顶向下的遍历，从而生成使用虚拟寄存器的目标代码。整个目标代码的框架和中间代码的框架是比较类似的，只有在指令和操作数的设计上有所不同。

此次实验中我们还完成了对生成的目标代码进行寄存器的分配，从而使得其可以在真正的环境下运行。

(一) 访存指令

对于访存指令，框架代码中已经给出了对于 LoadInstruction 的翻译，具体可以分为以下三种情况：**数据访存指令**在数据访存指令部分，由于本次实现了浮点数，所以需要补充上浮点数，与整形变量大部分的操作都乐死，例如加载全局变量，先从标签处将其地址加载到寄存器中，然后从刚加载到寄存器的地址获取值即可，只不过要用 vldr（指令前加 v 代表是 NEON 双精度或者 VFP 单精度 s0-s31 的指令，使用的是扩展向量寄存器，用于浮点数的计算）

- 加载一个全局变量或者常量

```
1        LoadInstruction::LoadInstruction(Operand *dst, Operand *src_addr,
             BasicBlock *insert_bb) : Instruction(LOAD, insert_bb)
2  {
3      fprintf(stderr, "LoadInstruction\n");
4      fprintf(stderr, "dst: %s,src_addr :%s\n", dst->toStr().c_str(), src_addr
           ->toStr().c_str());
5      operands.push_back(dst);
6      operands.push_back(src_addr);
7      dst->setDef(this);
8      src_addr->addUse(this);
9  }
10
11 LoadInstruction::~LoadInstruction()
12 {
13     operands[0]->setDef(nullptr);
14     if (operands[0]->usersNum() == 0)
15         delete operands[0];
16     operands[1]->removeUse(this);
17 }
18
19 void LoadInstruction::output() const
20 {
21 }
```

- 加载一个栈中的临时变量
- 加载一个数组元素

机器码的翻译

```
1
2  StoreMInstruction::StoreMInstruction(MachineBlock *p,
```

```
                                        MachineOperand *src1, MachineOperand *
                                            src2, MachineOperand *src3,
                                        int cond)
{
    // TODO
    this->parent = p;
    this->type = MachineInstruction::STORE;
    this->op = -1;
    this->cond = cond;
    this->use_list.push_back(src1);
    this->use_list.push_back(src2);
    if (src3)
        this->use_list.push_back(src3);
    src1->setParent(this);
    src2->setParent(this);
    if (src3)
        src3->setParent(this);
}

void StoreMInstruction::output()
{
    fprintf(stderr, "StoreMInstruction::output\n");
    // TODO
    // eg1. str r0 [r1, #4]
    // eg2. str r0 [r1]
    fprintf(yyout, "\tstr ");
    this->use_list[0]->output();
    fprintf(yyout, ", ");

    if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
        fprintf(yyout, "[");
    this->use_list[1]->output();

    // if use_list[2] is existed
    if (this->use_list.size() > 2)
    {
        fprintf(yyout, ", ");
        this->use_list[2]->output();
    }

    if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
        fprintf(yyout, "]");

    fprintf(yyout, "\n");
}
```

生成目标代码

```
void StoreInstruction::genMachineCode(AsmBuilder *builder)
```

```
{
    // TODO
    auto cur_block = builder->getBlock();
    MachineInstruction *cur_inst = nullptr;
    // Store to a global var
    if (operands[0]->getEntry()->isVariable() && dynamic_cast<
        IdentifierSymbolEntry *>(operands[0]->getEntry())->isGlobal())
    {
        auto dst = genMachineOperand(operands[0]);
        auto src = genMachineOperand(operands[1]);
        // 把imm→reg(new load)
        if (src->isImm())
        {
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
            cur_block->InsertInst(cur_inst);
            src = new MachineOperand(*internal_reg);
        }
        auto internal_reg1 = genMachineVReg();
        auto internal_reg2 = new MachineOperand(*internal_reg1);
        // example: load r0, addr_a 加载全局变量地址
        cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
        cur_block->InsertInst(cur_inst);
        // example: store r1, [r0] 存值入全局变量
        cur_inst = new StoreMInstruction(cur_block, src, internal_reg2);
        cur_block->InsertInst(cur_inst);
    }
    // store to a param
    else if (paramno != -1)
    {
        auto dst1 = genMachineReg(11);
        int off = dynamic_cast<TemporarySymbolEntry *>(operands[0]->getEntry
            ())->getOffset();
        auto dst2 = genMachineImm(off);
        if (off > 255 || off < -255)
        {
            auto operand = genMachineVReg();
            if (Judge(off))
                cur_block->InsertInst((new LoadMInstruction(cur_block,
                    operand, dst2)));
            else
            {
                cur_block->InsertInst(new MovMInstruction(cur_block,
                    MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                    )));
                if (off & 0xffff00)
                    cur_block->InsertInst(new BinaryMInstruction(cur_block,
                        BinaryMInstruction::ADD, operand, operand,
```

```
                        genMachineImm(off & 0xff0000)));
44                  if (off & 0xff000000)
45                      cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff000000)));
46              }
47              // cur_block->InsertInst((new LoadMInstruction(cur_block, operand
                    , dst2)));
48              dst2 = operand;
49          }
50          // auto dst2 = genMachineImm(dynamic_cast<TemporarySymbolEntry *>(
                operands[0]->getEntry())->getOffset());
51          if (paramno > 3) // is in stack
52          {
53              // int off = dynamic_cast<TemporarySymbolEntry *>(operands[1]->
                    getEntry())->getOffset();
54              // auto src2 = genMachineImm(off);
55
56              auto reg1 = genMachineVReg();
57              // cur_inst = new LoadMInstruction(cur_block, reg1, genMachineReg
                    (11), src2);
58              cur_inst = new LoadMInstruction(cur_block, reg1, genMachineReg
                    (11), genMachineImm((paramno - 4) * 4), true);
59              cur_block->InsertInst(cur_inst);
60              cur_inst = new StoreMInstruction(cur_block, new MachineOperand(*
                    reg1), dst1, dst2);
61              cur_block->InsertInst(cur_inst);
62          }
63          else // is in regs
64          {
65              auto src = genMachineReg(paramno);
66              cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
67              cur_block->InsertInst(cur_inst);
68          }
69      }
70      // store to a local operand 存储一个栈中的临时变量
71      else if (operands[0]->getEntry()->isTemporary() && operands[0]->getDef()
            && operands[0]->getDef()->isAlloc())
72      {
73          // example: store r1, [r0, #4]
74          auto src = genMachineOperand(operands[1]);
75
76          if (src->isImm())
77          {
78              int off = src->getVal();
79              auto operand = genMachineVReg();
80              if (Judge(off))
81                  cur_block->InsertInst((new LoadMInstruction(cur_block,
```

```
                operand, src)));
82          else
83          {
84              cur_block->InsertInst(new MovMInstruction(cur_block,
                    MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                    )));
85              if (off & 0xffff00)
86                  cur_block->InsertInst(new BinaryMInstruction(cur_block,
                        BinaryMInstruction::ADD, operand, operand,
                        genMachineImm(off & 0xff0000)));
87              if (off & 0xff000000)
88                  cur_block->InsertInst(new BinaryMInstruction(cur_block,
                        BinaryMInstruction::ADD, operand, operand,
                        genMachineImm(off & 0xff000000)));
89          }
90          src = new MachineOperand(*operand);
91      }
92
93      auto dst1 = genMachineReg(11);
94      auto dst2 = genMachineImm(dynamic_cast<TemporarySymbolEntry *>(
            operands[0]->getEntry())->getOffset());
95      cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
96      cur_block->InsertInst(cur_inst);
97  }
98  // store to temporary variable 存储一个数组元素，数组元素的地址存放在一个
        临时变量中，只需生成一条存储指令即可
99  else
100 {
101     // example: store r1, [r0]
102     auto src = genMachineOperand(operands[1]);
103     if (src->isImm())
104     {
105         int off = src->getVal();
106         auto operand = genMachineVReg();
107         if (Judge(off))
108             cur_block->InsertInst((new LoadMInstruction(cur_block,
                    operand, src)));
109         else
110         {
111             cur_block->InsertInst(new MovMInstruction(cur_block,
                        MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                        )));
112             if (off & 0xffff00)
113                 cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff0000)));
114             if (off & 0xff000000)
115                 cur_block->InsertInst(new BinaryMInstruction(cur_block,
```

```
                        BinaryMInstruction::ADD, operand, operand,
                        genMachineImm(off & 0xff000000)));
            }
            src = new MachineOperand(*operand);
        }
        auto dst = genMachineOperand(operands[0]);
        cur_inst = new StoreMInstruction(cur_block, src, dst);
        cur_block->InsertInst(cur_inst);
    }
}
```

(二) 内存分配指令

```
AllocaInstruction::AllocaInstruction(Operand *dst, SymbolEntry *se,
    BasicBlock *insert_bb) : Instruction(ALLOCA, insert_bb)
{
    operands.push_back(dst);
    dst->setDef(this);
    this->se = se;
}

AllocaInstruction::~AllocaInstruction()
{
    operands[0]->setDef(nullptr);
    if (operands[0]->usersNum() == 0)
        delete operands[0];
}

void AllocaInstruction::output() const
{
}
```

(三) 二元运算指令

- 二元运算

```
void StoreInstruction::genMachineCode(AsmBuilder *builder)
{
    // TODO
    auto cur_block = builder->getBlock();
    MachineInstruction *cur_inst = nullptr;
    // Store to a global var
    if (operands[0]->getEntry()->isVariable() && dynamic_cast<
        IdentifierSymbolEntry *>(operands[0]->getEntry())->isGlobal())
    {
        auto dst = genMachineOperand(operands[0]);
        auto src = genMachineOperand(operands[1]);
        // 把imm→reg(new load)
```

```
12          if (src->isImm())
13          {
14              auto internal_reg = genMachineVReg();
15              cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
16              cur_block->InsertInst(cur_inst);
17              src = new MachineOperand(*internal_reg);
18          }
19          auto internal_reg1 = genMachineVReg();
20          auto internal_reg2 = new MachineOperand(*internal_reg1);
21          // example: load r0, addr_a 加载全局变量地址
22          cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
23          cur_block->InsertInst(cur_inst);
24          // example: store r1, [r0] 存值入全局变量
25          cur_inst = new StoreMInstruction(cur_block, src, internal_reg2);
26          cur_block->InsertInst(cur_inst);
27      }
28      // store to a param
29      else if (paramno != -1)
30      {
31          auto dst1 = genMachineReg(11);
32          int off = dynamic_cast<TemporarySymbolEntry *>(operands[0]->getEntry
                ())->getOffset();
33          auto dst2 = genMachineImm(off);
34          if (off > 255 || off < -255)
35          {
36              auto operand = genMachineVReg();
37              if (Judge(off))
38                  cur_block->InsertInst((new LoadMInstruction(cur_block,
                        operand, dst2)));
39              else
40              {
41                  cur_block->InsertInst(new MovMInstruction(cur_block,
                        MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                        )));
42                  if (off & 0xffff00)
43                      cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff0000)));
44                  if (off & 0xff000000)
45                      cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff000000)));
46              }
47              // cur_block->InsertInst((new LoadMInstruction(cur_block, operand
                    , dst2)));
48              dst2 = operand;
49          }
50          // auto dst2 = genMachineImm(dynamic_cast<TemporarySymbolEntry *>(
```

```
            operands[0]->getEntry())->getOffset());
51          if (paramno > 3) // is in stack
52          {
53              // int off = dynamic_cast<TemporarySymbolEntry *>(operands[1]->
                    getEntry())->getOffset();
54              // auto src2 = genMachineImm(off);
55
56              auto reg1 = genMachineVReg();
57              // cur_inst = new LoadMInstruction(cur_block, reg1, genMachineReg
                    (11), src2);
58              cur_inst = new LoadMInstruction(cur_block, reg1, genMachineReg
                    (11), genMachineImm((paramno - 4) * 4), true);
59              cur_block->InsertInst(cur_inst);
60              cur_inst = new StoreMInstruction(cur_block, new MachineOperand(*
                    reg1), dst1, dst2);
61              cur_block->InsertInst(cur_inst);
62          }
63          else // is in regs
64          {
65              auto src = genMachineReg(paramno);
66              cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
67              cur_block->InsertInst(cur_inst);
68          }
69      }
70      // store to a local operand 存储一个栈中的临时变量
71      else if (operands[0]->getEntry()->isTemporary() && operands[0]->getDef()
            && operands[0]->getDef()->isAlloc())
72      {
73          // example: store r1, [r0, #4]
74          auto src = genMachineOperand(operands[1]);
75
76          if (src->isImm())
77          {
78              int off = src->getVal();
79              auto operand = genMachineVReg();
80              if (Judge(off))
81                  cur_block->InsertInst((new LoadMInstruction(cur_block,
                        operand, src)));
82              else
83              {
84                  cur_block->InsertInst(new MovMInstruction(cur_block,
                        MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                        )));
85                  if (off & 0xffff00)
86                      cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff0000)));
87                  if (off & 0xff000000)
```

```
88                      cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff000000)));
89              }
90              src = new MachineOperand(*operand);
91          }
92
93          auto dst1 = genMachineReg(11);
94          auto dst2 = genMachineImm(dynamic_cast<TemporarySymbolEntry *>(
                operands[0]->getEntry())->getOffset());
95          cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
96          cur_block->InsertInst(cur_inst);
97      }
98      // store to temporary variable 存储一个数组元素，数组元素的地址存放在一个
            临时变量中，只需生成一条存储指令即可
99      else
100     {
101         // example: store r1, [r0]
102         auto src = genMachineOperand(operands[1]);
103         if (src->isImm())
104         {
105             int off = src->getVal();
106             auto operand = genMachineVReg();
107             if (Judge(off))
108                 cur_block->InsertInst((new LoadMInstruction(cur_block,
                        operand, src)));
109             else
110             {
111                 cur_block->InsertInst(new MovMInstruction(cur_block,
                        MovMInstruction::MOV, operand, genMachineImm(off & 0xffff
                        )));
112                 if (off & 0xffff00)
113                     cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff0000)));
114                 if (off & 0xff000000)
115                     cur_block->InsertInst(new BinaryMInstruction(cur_block,
                            BinaryMInstruction::ADD, operand, operand,
                            genMachineImm(off & 0xff000000)));
116             }
117             src = new MachineOperand(*operand);
118         }
119         auto dst = genMachineOperand(operands[0]);
120         cur_inst = new StoreMInstruction(cur_block, src, dst);
121         cur_block->InsertInst(cur_inst);
122     }
123 }
124
```

```
125 void BinaryInstruction::genMachineCode(AsmBuilder *builder)
126 {
127     // TODO:
128     // complete other instructions
129     auto cur_block = builder->getBlock();
130     auto dst = genMachineOperand(operands[0]);
131     auto src1 = genMachineOperand(operands[1]);
132     auto src2 = genMachineOperand(operands[2]);
133     /* HINT:
134      * The source operands of ADD instruction in ir code both can be
            immediate num.
135      * However, it's not allowed in assembly code.
136      * So you need to insert LOAD/MOV instrucrion to load immediate num into
            register.
137      * As to other instructions, such as MUL, CMP, you need to deal with this
            situation, too.*/
138     MachineInstruction *cur_inst = nullptr;
139     if (src1->isImm())
140     {
141         auto internal_reg = genMachineVReg();
142 
143         cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
144         cur_block->InsertInst(cur_inst);
145         src1 = new MachineOperand(*internal_reg);
146     }
147     switch (opcode)
148     {
149     case ADD:
150     {
151         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                dst, src1, src2);
152         break;
153     }
154     case SUB:
155     {
156         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB,
                dst, src1, src2);
157         break;
158     }
159     case MUL:
160         if (src2->isImm())
161         {
162             auto internal_reg = genMachineVReg();
163             cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
164             cur_block->InsertInst(cur_inst);
165             src2 = new MachineOperand(*internal_reg);
166         }
167         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL,
```

```
                dst, src1, src2);
168             break;
169         case DIV:
170             if (src2->isImm())
171             {
172                 auto internal_reg = genMachineVReg();
173                 cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
174                 cur_block->InsertInst(cur_inst);
175                 src2 = new MachineOperand(*internal_reg);
176             }
177             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV,
                    dst, src1, src2);
178             break;
179         case AND:
180             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::AND,
                    dst, src1, src2);
181             break;
182         case OR:
183             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::OR,
                   dst, src1, src2);
184             break;
185         case MOD:
186         {
187             // 运算转化: a%b <==> a-(a/b)*b
188             if (src2->isImm())
189             {
190                 fprintf(stderr, "%d", src2->getVal());
191                 auto internal_reg = genMachineVReg();
192                 cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
193                 cur_block->InsertInst(cur_inst);
194                 src2 = new MachineOperand(*internal_reg);
195             }
196             // DIV
197             auto dst_div = genMachineVReg();
198             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV,
                    dst_div, src1, src2);
199             cur_block->InsertInst(cur_inst);
200             // MUL
201             auto dst_mul = genMachineVReg();
202             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL,
                    dst_mul, dst_div, src2);
203             cur_block->InsertInst(cur_inst);
204             // SUB
205             cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB,
                    dst, src1, dst_mul);
206             break;
207         }
208         default:
```

```
209             break;
210         }
211         cur_block->InsertInst(cur_inst);
212     }
```

- 赋值指令

```
1           MovMInstruction::MovMInstruction(MachineBlock *p, int op,
2                                            MachineOperand *dst, MachineOperand *src,
3                                            int cond)
4   {
5       // TODO
6       this->parent = p;
7       this->op = op;
8       this->type = MachineInstruction::MOV;
9       this->cond = cond;
10      this->def_list.push_back(dst);
11      this->use_list.push_back(src);
12      dst->setParent(this);
13      src->setParent(this);
14  }
15
16  void MovMInstruction::output()
17  {
18      fprintf(stderr, "MovMInstruction::output\n");
19      // TODO
20      switch (this->op)
21      {
22      case MovMInstruction::MOV:
23          fprintf(yyout, "\tmov");
24          break;
25      case MovMInstruction::MVN:
26          fprintf(yyout, "\tmvn");
27          break;
28      }
29      this->PrintCond();
30      this->def_list[0]->output();
31      fprintf(yyout, ", ");
32      this->use_list[0]->output();
33      fprintf(yyout, "\n");
34  }
```

- 分支指令

```
1           BranchMInstruction::BranchMInstruction(MachineBlock *p, int op,
2                                                  MachineOperand *dst,
3                                                  int cond)
4   {
5       // TODO
```

```
6       this->parent = p;
7       this->type = MachineInstruction::BRANCH;
8       this->op = op;
9       this->cond = cond;
10      this->def_list.push_back(dst);
11      dst->setParent(this);
12  }
13
14  void BranchMInstruction::output()
15  {
16      fprintf(stderr, "BranchMInstruction::output\n");
17      // TODO
18      switch (this->op)
19      {
20      case BranchMInstruction::EQ:
21          fprintf(yyout, "\tbeq ");
22          break;
23      case BranchMInstruction::NE:
24          fprintf(yyout, "\tbne ");
25          break;
26      case BranchMInstruction::LT:
27          fprintf(yyout, "\tblt ");
28          break;
29      case BranchMInstruction::LE:
30          fprintf(yyout, "\tble ");
31          break;
32      case BranchMInstruction::GT:
33          fprintf(yyout, "\tbgt ");
34          break;
35      case BranchMInstruction::GE:
36          fprintf(yyout, "\tbge ");
37          break;
38      case BranchMInstruction::B:
39          fprintf(yyout, "\tb ");
40          break;
41      case BranchMInstruction::BL:
42          fprintf(yyout, "\tbl ");
43          break;
44      case BranchMInstruction::BX:
45          fprintf(yyout, "\tbx ");
46          break;
47      default:
48          break;
49      }
50      this->def_list[0]->output();
51      fprintf(yyout, "\n");
52  }
```

- **比较指令**对于比较指令，当其为立即数时按照二元运算指令先将其存储到一个寄存器，生成比较指令，为了之后的跳转指令做铺垫，对其操作进行判断，如果是大于，小于等操作，其为假的分支就是在 7 下互补的指令，如果是等于操作，则为假的分支是不等，不等的指令也类似；整型部分和浮点部分类似

```
void CmpInstruction::genMachineCode(AsmBuilder *builder)
{
    // TODO
    auto cur_block = builder->getBlock();
    // auto dst = genMachineOperand(operands[0]);
    auto src1 = genMachineOperand(operands[1]);
    auto src2 = genMachineOperand(operands[2]);
    MachineInstruction *cur_inst = nullptr;
    // cmp R,R  or  cmp R,#imm
    if (src1->isImm())
    {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
        cur_block->InsertInst(cur_inst);
        src1 = new MachineOperand(*internal_reg);
    }
    if (src2->isImm())
    {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
        cur_block->InsertInst(cur_inst);
        src2 = new MachineOperand(*internal_reg);
    }
    cur_inst = new CmpMInstruction(cur_block, src1, src2);
    cur_block->InsertInst(cur_inst);

    CondBrInstruction *nextinst = dynamic_cast<CondBrInstruction *>(getLink()
        );
    if (nextinst != 0)
    {
        switch (this->opcode)
        {
        case E:
            nextinst->setop(CondBrInstruction::E);
            break;
        case NE:
            nextinst->setop(CondBrInstruction::NE);
            break;
        case LE:
            nextinst->setop(CondBrInstruction::LE);
            break;
        case GE:
            nextinst->setop(CondBrInstruction::GE);
```

```
            break;
        case L:
            nextinst->setop(CondBrInstruction::L);
            break;
        case G:
            nextinst->setop(CondBrInstruction::G);
            break;
        default:
            break;
        }
        std::string label;
        label = ".L";
        std::stringstream s;
        auto bb = nextinst->getTrueBranch();
        s << bb->getNo();
        std::string temp;
        s >> temp;
        label += temp;
        // dst为真分支对应的基本块编号
        auto dst = new MachineOperand(label);
        int op;
        switch (opcode)
        {
        case E:
            op = BranchMInstruction::EQ;
            break;
        case NE:
            op = BranchMInstruction::NE;
            break;
        case LE:
            op = BranchMInstruction::LE;
            break;
        case GE:
            op = BranchMInstruction::GE;
            break;
        case L:
            op = BranchMInstruction::LT;
            break;
        case G:
            op = BranchMInstruction::GT;
            break;
        default:
            break;
        }
        cur_inst = new BranchMInstruction(cur_block, op, dst);
        cur_block->InsertInst(cur_inst);
    }
    // nextinst is not br ==> need to save the result of cmp
```

```
91      else
92      {
93          int op;
94          switch (opcode)
95          {
96          case E:
97              op = MachineInstruction::EQ;
98              break;
99          case NE:
100             op = MachineInstruction::NE;
101             break;
102         case LE:
103             op = MachineInstruction::LE;
104             break;
105         case GE:
106             op = MachineInstruction::GE;
107             break;
108         case L:
109             op = MachineInstruction::LT;
110             break;
111         case G:
112             op = MachineInstruction::GT;
113             break;
114         default:
115             break;
116         }
117
118         auto dst = genMachineOperand(operands[0]);
119         // 生成mov指令存储cmp结果
120         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                new MachineOperand(MachineOperand::IMM, 1), op);
121         cur_block->InsertInst(cur_inst);
122         switch (op)
123         {
124         case MachineInstruction::EQ:
125             op = MachineInstruction::NE;
126             break;
127         case MachineInstruction::NE:
128             op = MachineInstruction::EQ;
129             break;
130         case MachineInstruction::GE:
131             op = MachineInstruction::LT;
132             break;
133         case MachineInstruction::LE:
134             op = MachineInstruction::GT;
135             break;
136         case MachineInstruction::GT:
137             op = MachineInstruction::LE;
```

```
            break;
        case MachineInstruction::LT:
            op = MachineInstruction::GE;
            break;
        default:
            break;
        }
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            new MachineOperand(MachineOperand::IMM, 0), op);
        cur_block->InsertInst(cur_inst);
    }
}
```

机器码翻译

```

void CmpInstruction::output() const
{
    std::string s1, s2, s3, op, type;
    s1 = operands[0]->toStr();
    s2 = operands[1]->toStr();
    s3 = operands[2]->toStr();
    type = operands[1]->getType()->toStr();
    switch (opcode) {
        case E:
            op = "eq";
            break;
        case NE:
            op = "ne";
            break;
        case L:
            op = "slt";
            break;
        case LE:
            op = "sle";
            break;
        case G:
            op = "sgt";
            break;
        case GE:
            op = "sge";
            break;
        default:
            op = "";
            break;
    }
}
```

- 堆栈指令

```
StackMInstrcuton::StackMInstrcuton(MachineBlock *p, int op,
                                   MachineOperand *src,
                                   int cond)
{
    // TODO
    this->parent = p;
    this->op = op;
    this->type = MachineInstruction::STACK;
    this->cond = cond;
    this->use_list.push_back(src);
    src->setParent(this);
}

StackMInstrcuton::StackMInstrcuton(MachineBlock *p,
                                   int op,
                                   std::vector<MachineOperand *> srcs,
                                   MachineOperand *src,
                                   MachineOperand *src1,
                                   int cond)
{
    this->parent = p;
    this->type = MachineInstruction::STACK;
    this->op = op;
    this->cond = cond;
    if (srcs.size())
        for (auto it = srcs.begin(); it != srcs.end(); it++)
            this->use_list.push_back(*it);
    this->use_list.push_back(src);
    src->setParent(this);
    if (src1)
    {
        this->use_list.push_back(src1);
        src1->setParent(this);
    }
}

void StackMInstrcuton::output()
{
    fprintf(stderr, "StackMInstrcuton::output\n");
    switch (op)
    {
    case PUSH:
        fprintf(yyout, "\tpush ");
        break;
    case POP:
        fprintf(yyout, "\tpop ");
        break;
```

```
    }
    fprintf(yyout, "{");
    this->use_list[0]->output();
    for (long unsigned int i = 1; i < use_list.size(); i++)
    {
        fprintf(yyout, ", ");
        this->use_list[i]->output();
    }
    fprintf(yyout, "}\n");
}
```

- 全局指令

```
        void GlobalInstruction::genMachineCode(AsmBuilder *builder)
{
    // TODO
    auto cur_block = builder->getBlock();
    auto cur_unit = builder->getUnit();
    MachineInstruction *cur_inst = 0;
    auto dst = genMachineOperand(operands[0]);
    std::vector<MachineOperand *> src;
    for (unsigned int i = 1; i < operands.size(); i++)
    {
        src.push_back(genMachineOperand(operands[i]));
    }
    cur_inst = new GlobalMInstruction(cur_block, dst, src, se);
    // 交付给unit去实现
    cur_unit->insertGlobal(dynamic_cast<GlobalMInstruction *>(cur_inst));
}
```

(四) 控制流指令

控制流指令对于控制流指令,对于 UncondBrInstr 生成一条无条件跳转,仿照 genMchineCode 生成汇编代码部分的块号；对于 CondBrInstr，由于其一定在比较指令之后，在比较指令的部分已经有了铺垫，根据其 true 或 false 分支进行跳转而 RetInstr 如果有返回值，需要生成 MOV 指令，将返回值保存在 R0 寄存器，浮点数保存在 16 号寄存器中，由于整型变量大小有一定的限制，所以当其超过限制时需要对其重新进行加载；接下来生成 add 指令恢复栈帧，并生成跳转指令返回 caller。

```
RetInstruction::~RetInstruction()
{
    if (!operands.empty())
        operands[0]->removeUse(this);
}


void RetInstruction::output() const
{
```

```
    if (operands.empty()) {
        fprintf(yyout, "  ret void\n");
    } else {
        std::string ret, type;
        ret = operands[0]->toStr();
        type = operands[0]->getType()->toStr();
        fprintf(yyout, "  ret %s %s\n", type.c_str(), ret.c_str());
    }
}
```

关于控制流指令的机器码翻译将跳转的类型和跳转的地方输出，如果有条件则打印条件，控制流指令的构造的话先进行传参，如果为 BL 指令，为带链接的跳转，须保存 PC 当前内容，设置好整数和浮点数的寄存器，并将其添加到定义列表，然后根据跳转到的参数个数来决定是否将其添加到使用列表；如果为 BX 指令，是带状态切换的跳转，保存状态参数即可。

(五) 函数定义

(六) 函数调用指令

函数定义及函数调用在函数调用部分，先对参数的个数进行计算，根据参数个数判断是否进行 push 操作来传递参数后分别对整型和浮点型的变量进行判断其个数，对超过相应部分的参数进行压栈，并记录相应的压栈的个数，最后生成 bl 跳转指令，保存 pc 相应内容，为非叶函数做铺垫，根据记录的个数来为其分配栈空间，结果如果被用到须根据其类型在相应的寄存器中保存返回值，代码如下:

```
void CallInstruction::genMachineCode(AsmBuilder *builder)
{
    auto cur_block = builder->getBlock();
    MachineInstruction *cur_inst = 0;

    // pass params
    for (unsigned int i = 1; i < operands.size(); i++)
    {
        // 参数值传递按顺序存放在寄存器r0,r1,r2,r3里，超过4个参数值传递则放栈里
        // r0用于存放返回值
        if (i > 4)
        {
            auto reg1 = genMachineVReg();
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, reg1, genMachineOperand(operands[i]));
            cur_block->InsertInst(cur_inst);
            cur_inst = new StoreMInstruction(cur_block, reg1, genMachineReg(13), genMachineImm(-(operands.size() - i) * 4));
            cur_block->InsertInst(cur_inst);
            builder->getFunction()->AllocSpace(4);

            continue;
        }
```

```
        else
        {
            // auto reg1 = genMachineVReg();
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
                genMachineReg(i - 1), genMachineOperand(operands[i]));
            cur_block->InsertInst(cur_inst);
        }
    }
    if (operands.size() > 5)
    {
        cur_inst = new BinaryMInstruction(nullptr, BinaryMInstruction::SUB,
            genMachineReg(13), genMachineReg(13), genMachineImm((operands.
            size() - 5) * 4));
        cur_block->InsertInst(cur_inst);
    }
    // example:  bl func
    std::string label = dynamic_cast<IdentifierSymbolEntry *>(this->getEntry
        ())->getName();
    auto dst = new MachineOperand(label, 1);
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL, dst)
        ;
    cur_block->InsertInst(cur_inst);

    if (operands.size() > 5)
    {
        auto off = genMachineImm((operands.size() - 5) * 4);
        auto sp = new MachineOperand(MachineOperand::REG, 13);
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                                          sp, sp, off);
        cur_block->InsertInst(cur_inst);
    }

    // save the return value
    if (dynamic_cast<FunctionType *>(this->getEntry()->getType())->getRetType
        ()->toStr() != "void")
    {
        auto ret = genMachineOperand(operands[0]);
        // r0存储函数返回值
        auto r0 = genMachineReg(0);
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, ret,
            r0);
        cur_block->InsertInst(cur_inst);
    }
}
```

机器码翻译

```
    void CallInstruction::output() const
{
```

```
3       fprintf(yyout, "  ");
4       if (operands[0])
5           fprintf(yyout, "%s = ", operands[0]->toStr().c_str());
6       FunctionType* type = (FunctionType*)(func->getType());
7       fprintf(yyout, "call %s %s(", type->getRetType()->toStr().c_str(),
8               func->toStr().c_str());
9       for (long unsigned int i = 1; i < operands.size(); i++) {
10          if (i != 1)
11              fprintf(yyout, ", ");
12          fprintf(yyout, "%s %s", operands[i]->getType()->toStr().c_str(),
13                  operands[i]->toStr().c_str());
14      }
15      fprintf(yyout, ")\n");
16  }
```

九、 实现寄存器分配

寄存器分配是编译器的一个重要优化技术，通过将程序变量尽可能地分配到寄存器，从而提高程序执行速度。在本次实验中我们需要完成线性扫描寄存器分配算法，遍历每个活跃区间(Interval)，为其分配物理寄存器。其具体分为以下三个步骤。

(一) 活跃区间分析

在前一步的目标代码生成过程中，已经为所有临时变量分配了一个虚拟寄存器。在这一步需要为每个虚拟寄存器计算活跃区间，活跃区间相交的虚拟寄存器不能分配相同的物理寄存器。活跃区间的计算主要依赖活跃变量分析这一数据流分析方法，活跃变量分析的结果可以判断变量 x 在程序点 p 处是否活跃，通俗来讲，变量 x 在点 p 处活跃指的是变量 x 在点 p 处的值在点 p 或点 p 之后仍然会被用到。变量 x 编号最小和最大的两个活跃点便是其活跃区间的端点。这一步在课程中应该已经有所讲解，具体算法可参照龙书第二版 P391。

部分代码如下:

```
1  void LiveVariableAnalysis::pass(MachineUnit *unit)
2  {
3      for (auto &func : unit->getFuncs())
4      {
5          computeUsePos(func);
6          computeDefUse(func);
7          iterate(func);
8      }
9  }
10
11 void LiveVariableAnalysis::pass(MachineFunction *func)
12 {
13     computeUsePos(func);
14     computeDefUse(func);
15     iterate(func);
16 }
17
18 void LiveVariableAnalysis::computeDefUse(MachineFunction *func)
19 {
20     for (auto &block : func->getBlocks())
21     {
22         for (auto inst = block->getInsts().begin(); inst != block->getInsts()
               .end(); inst++)
23         {
24             auto user = (*inst)->getUse();
25             std::set<MachineOperand *> temp(user.begin(), user.end());
26             set_difference(temp.begin(), temp.end(),
27                            def[block].begin(), def[block].end(), inserter(use
                                   [block], use[block].end()));
28             auto defs = (*inst)->getDef();
29             for (auto &d : defs)
30                 def[block].insert(all_uses[*d].begin(), all_uses[*d].end());
```

```
        }
    }
}

void LiveVariableAnalysis::iterate(MachineFunction *func)
{
    for (auto &block : func->getBlocks())
        block->getLiveIn().clear();
    bool change;
    change = true;
    while (change)
    {
        change = false;
        for (auto &block : func->getBlocks())
        {
            block->getLiveOut().clear();
            auto old = block->getLiveIn();
            for (auto &succ : block->getSuccs())
                block->getLiveOut().insert(succ->getLiveIn().begin(), succ->
                    getLiveIn().end());
            block->getLiveIn() = use[block];
            std::vector<MachineOperand *> temp;
            set_difference(block->getLiveOut().begin(), block->getLiveOut().
                end(),
                           def[block].begin(), def[block].end(), inserter(
                               block->getLiveIn(), block->getLiveIn().end()))
                               ;
            if (old != block->getLiveIn())
                change = true;
        }
    }
}

void LiveVariableAnalysis::computeUsePos(MachineFunction *func)
{
    for (auto &block : func->getBlocks())
    {
        for (auto &inst : block->getInsts())
        {
            auto uses = inst->getUse();
            for (auto &use : uses)
                all_uses[*use].insert(use);
        }
    }
}
```

(二) 寄存器分配

算法主要涉及到了两个集合: intervals 表示还未分配寄存器的活跃区间, 其中所有的 interval 都按照开始位置进行递增排序; active 表示当前正在占用物理寄存器的活跃区间集合, 其中所有的 interval 都按照结束位置进行递增排序。算法遍历 intervals 列表, 对遍历到的每一个活跃区间 i 都进行如下的处理:

- 遍历 active 列表, 看该列表中是否存在结束时间早于区间 i 开始时间的 interval (即与活跃区间 i 不冲突), 若有, 则说明此时为其分配的物理寄存器可以回收, 可以用于后续的分配, 需要将其在 active 列表删除;
- 判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等,

  (a) 若相等, 则说明当前所有物理寄存器都被占用, 需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和活跃区间 i 中选择一个 interval 将其溢出到栈中, 选择策略就是看哪个活跃区间结束时间更晚, 如果是活跃区间 i 的结束时间更晚, 只需要置位其 spill 标志位即可, 如果是 active 列表中的活跃区间结束时间更晚, 需要置位其 spill 标志位, 并将其占用的寄存器分配给区间 i, 再将区间 i 插入到 active 列表中。

  (b) 若不相等, 则说明当前有可用于分配的物理寄存器, 为区间 i 分配物理寄存器之后, 再按照活跃区间结束位置, 将其插入到 active 列表中即可。

```
1   void LinearScan::allocateRegisters()
2   {
3       fprintf(stderr, "LinearScan::allocateRegisters()\n");
4       for (auto &f : unit->getFuncs())
5       {
6           fprintf(stderr, "auto &f : unit->getFuncs(),f:%s\n", f->getSymbol()->
                toStr().c_str());
7           func = f;
8           bool success;
9           success = false;
10          while (!success) // repeat until all vregs can be mapped
11          {
12              fprintf(stderr, "computeLiveIntervals in\n");
13              computeLiveIntervals();
14              fprintf(stderr, "computeLiveIntervals out\n");
15              success = linearScanRegisterAllocation();
16              fprintf(stderr, "linearScanRegisterAllocation\n");
17              if (success)
18              {
19                  fprintf(stderr, "success \n");
20                  // all vregs can be mapped to real regs
21                  modifyCode();
22                  fprintf(stderr, "modifyCode\n");
23              }
24              else
25              {
26                  fprintf(stderr, "success not\n");
27                  // spill vregs that can't be mapped to real regs
```

```
                genSpillCode();
                fprintf(stderr, "genSpillCode\n");
            }
        }
    }
}
bool LinearScan::linearScanRegisterAllocation()
{
    // Todo
    bool success = true;
    active.clear();
    regs.clear();
    for (int i = 4; i < 11; i++)
        regs.push_back(i);
    for (auto &i : intervals)
    {
        expireOldIntervals(i);
        if (regs.empty())
        {
            spillAtInterval(i);
            success = false;
        }
        else
        {
            i->rreg = regs.front();
            regs.erase(regs.begin());
            insertToActive(i);
        }
    }
    return success;
}
```


(三) 生成溢出代码

在上一步寄存器分配结束之后，如果没有临时变量被溢出到栈内，那寄存器分配的工作就结束了，所有的临时变量都被存在了寄存器中；若有，就需要在操作该临时变量时插入对应的 LoadMInstruction 和 StoreMInstruction，其起到的实际效果就是将该临时变量的活跃区间进行切分，以便重新进行寄存器分配。

- 为其在栈内分配空间，获取当前在栈内相对 FP 的偏移；
- 遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚拟寄存器中；
- 遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内；

```
void LinearScan::spillAtInterval(Interval *interval)
{
```

```
    auto spill = active.back();
    if (spill->end > interval->end)
    {
        spill->spill = true;
        interval->rreg = spill->rreg;
        insertToActive(interval);
    }
    else
    {
        interval->spill = true;
    }
}
```

十、 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。

(一) 窥孔优化

窥孔优化的概念较为简单，使用一个滑动窗口，对窗口中的一段指令序列进行分析，如果该指令序列存在一个性能更好的等价的指令序列，那么我们便可以用这个等价序列对原序列进行替换。、

1. Mem2Reg Mem2Reg 是 LLVM 采用的 SSA 转换算法。如果使用 LLVM 作为后端，编译器前端在生成 LLVM IR 时可以先生成 alloca/load/store 这样借助内存存储局部变量值的 SSA 形式（即可以先不生成 函数)。在 LLVM 拿到前端生成的代码后，Mem2Reg 会将这些指令删除，并插入合适的 函数。

示例:

```
int main () {
int x , cond = 1;
if ( cond > 0)
x = 1;
else
x = -1;
return x ;
}
```

中间代码为:

```
define dso_local i32 @main() {
%1 = alloca i32
%2 = alloca i32
store i32 1 , i32 * %1
%3 = load i32 , i32 * %1
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %8

5:
store i32 1 , i32 * %2
br label %6

6:
%7 = load i32 , i32 * %2
ret i32 %7

8:
%9 = sub i32 0 , 1
store i32 %9, i32 * %2
br label %6
}
```

Mem2Reg 转换后的代码:

```
    define dso_local i32 @main() {
 %1 = icmp sgt i32 1 , 0
 br i1 %1, l a b e l %2, l a b e l %5
 2:
 br l a b e l %3

 3:
%4 = phi i32 [ 1 , %2 ] , [ %6, %5 ]
 ret i32 %4

 5:
 %6 = sub i32 0 , 1
 br l a b e l %3
 }
```

图着色寄存器分配如果寄存器分配问题被抽象成图着色问题，那么图中的每个节点代表某个变量的活跃期或生存期（Live range)。活跃期定义是从变量第一次被定义（赋值）开始，到它下一次被赋值前的最后一次被使用为止。两个节点之间的边表示这两个变量活跃期因为生命期（lifetime）重叠导致互相冲突或干涉。一般说来，如果两个变量在函数的某一点是同时活跃（live）的，它们就相互冲突，不能占有同一个寄存器。

基于着色图分配寄存器的过程就是将不同颜色对应不同物理寄存器，颜色数量 k 对应物理寄存器数量。通过图着色方法可以为变量分配寄存器而不产生冲突。当然，寄存器分配比较复杂，不仅仅是图着色的问题。比如，当物理寄存器数目不足以分配给所有变量时，就必须将某些变量溢出到内存中，即 spill。最小化溢出代价的问题，也是一个 NP-complete 问题。

十一、 额外添加部分

- 浮点数与其他进制浮点数对于浮点类型，首先在词法和语法分析部分直接仿照整型变量对其进行添加，在之后操作中需要进行判断是整型还是浮点类型，根据类型进行操作。

  比如在二元指令的计算，数组各个维度类型的判断，函数的参数等，生成中间代码也需要根据浮点或者是整型来判断是 float 还是 i32；生成机器码时通过对 float 的判断来决定是 arm 指令还是 NEON/VFP 指令，相应的判断是否寄存到扩展寄存器中。

  如下所示的 call 指令代码当中就用到了大量的判断 float 类型的代码

```
void CallInstruction::genMachineCode(AsmBuilder* builder) {
auto cur_block = builder->getBlock();
MachineOperand* operand;  //, *num;
MachineInstruction* cur_inst;
// auto fp = new MachineOperand(MachineOperand::REG, 11);
// TODO
size_t idx;
auto funcSE = (IdentifierSymbolEntry*)func;
if (funcSE->getName() == "llvm.memset.p0.i32") {
    auto r0 = genMachineReg(0);
    auto r1 = genMachineReg(1);
    auto r2 = genMachineReg(2);
    auto int8Ptr = operands[1];
    auto bitcast = (BitcastInstruction*)(int8Ptr->getDef());
    if (!bitcast->getFlag()) {
        auto arraySE =
            (TemporarySymbolEntry*)(bitcast->getUse()[0]->getEntry());
        int offset = arraySE->getOffset();
        operand = genMachineVReg();
        auto fp = genMachineReg(11);
        if (offset > -255 && offset < 255) {
            cur_block->InsertInst(
                new BinaryMInstruction(cur_block, BinaryMInstruction::ADD
                    ,
                                       r0, fp, genMachineImm(offset)));
        } else {
            cur_inst =
                new LoadMInstruction(cur_block, LoadMInstruction::LDR,
                                     operand, genMachineImm(offset));
            operand = new MachineOperand(*operand);
            cur_block->InsertInst(cur_inst);
            cur_block->InsertInst(new BinaryMInstruction(
                cur_block, BinaryMInstruction::ADD, r0, fp, operand));
        }
    } else {
        cur_block->InsertInst(
            new MovMInstruction(cur_block, MovMInstruction::MOV, r0,
                                genMachineOperand(bitcast->getUse()[0])))
                                    ;
```

```
38          }
39          cur_block->InsertInst(new MovMInstruction(
40              cur_block, MovMInstruction::MOV, r1, genMachineImm(0)));
41          auto len = genMachineOperand(operands[3]);
42          if (len->isImm() && len->getVal() > 255) {
43              operand = genMachineVReg();
44              cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
45                                              operand, len);
46              operand = new MachineOperand(*operand);
47              cur_block->InsertInst(cur_inst);
48          } else
49              operand = len;
50          cur_block->InsertInst(
51              new MovMInstruction(cur_block, MovMInstruction::MOV, r2, operand)
                    );
52          cur_block->InsertInst(new BranchMInstruction(
53              cur_block, BranchMInstruction::BL, new MachineOperand("@memset"))
                    );
54          return;
55      }
56
57      int stk_cnt = 0;
58      std::vector<MachineOperand*> vec;
59
60      bool need_align = false;  // for alignment
61      int float_num = 0;
62      int int_num = 0;
63      for (size_t i = 1; i < operands.size(); i++) {
64          if (operands[i]->getType()->isFloat()) {
65              float_num++;
66          } else {
67              int_num++;
68          }
69      }
70
71      int push_num = 0;
72      if (float_num > 4) {
73          push_num += float_num - 4;
74      }
75      if (int_num > 4) {
76          push_num += int_num - 4;
77      }
78
79      if (push_num % 2 != 0) {
80          need_align = true;
81      }
82
83
```

```
84
85      size_t int_idx = idx;
86
87      int fpreg_cnt = 1;
88      for (idx = 1; idx < operands.size(); idx++) {
89          if (fpreg_cnt == 5 && !need_align)
90              break;
91          if (fpreg_cnt == 6 && need_align) {
92              break;
93          }
94          if (!operands[idx]->getType()->isFloat()) {
95              continue;
96          }
97          operand = genMachineFReg(fpreg_cnt - 1);
98          auto src = genMachineFloatOperand(operands[idx]);
99          if (src->isImm()) {
100             auto internal_reg = genMachineVReg();
101             cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
102                                             internal_reg, src);
103             cur_block->InsertInst(cur_inst);
104             internal_reg = new MachineOperand(*internal_reg);
105             cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
106                                            operand, internal_reg);
107         } else {
108             cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
109                                            operand, src);
110         }
111         cur_block->InsertInst(cur_inst);
112         fpreg_cnt++;
113     }
114
115     size_t float_idx = idx;
116
117
118     idx = std::min(float_idx, int_idx);
119
120     for (size_t i = operands.size() - 1; i >= idx; i--) {
121         if (operands[i]->getType()->isFloat() && i >= float_idx) {
122             operand = genMachineFloatOperand(operands[i]);
123             if (operand->isImm()) {
124                 auto dst = genMachineVReg(true);
125                 auto internal_reg = genMachineVReg();
126                 cur_inst = new LoadMInstruction(
127                     cur_block, LoadMInstruction::LDR, internal_reg, operand);
128                 cur_block->InsertInst(cur_inst);
129                 internal_reg = new MachineOperand(*internal_reg);
130                 cur_inst = new MovMInstruction(cur_block, MovMInstruction::
                    VMOV,
```

```
131                                                 dst, internal_reg);
132                 cur_block->InsertInst(cur_inst);
133                 operand = new MachineOperand(*dst);
134             }
135             cur_inst = new StackMInstruction(
136                 cur_block, StackMInstruction::VPUSH, vec, operand);
137             cur_block->InsertInst(cur_inst);
138             stk_cnt++;
139         } else if (!operands[i]->getType()->isFloat() && i >= int_idx) {
140             operand = genMachineOperand(operands[i]);
141             if (operand->isImm()) {
142                 auto dst = genMachineVReg();
143                 if (operand->getVal() < 256) {
144                     cur_inst = new MovMInstruction(
145                         cur_block, MovMInstruction::MOV, dst, operand);
146                 } else {
147                     cur_inst = new LoadMInstruction(
148                         cur_block, LoadMInstruction::LDR, dst, operand);
149                 }
150                 cur_block->InsertInst(cur_inst);
151                 operand = new MachineOperand(*dst);
152             }
153             cur_inst = new StackMInstruction(cur_block, StackMInstruction::
                    PUSH,
154                                              vec, operand);
155             cur_block->InsertInst(cur_inst);
156             stk_cnt++;
157         }
158     }
159
160     auto label = new MachineOperand(func->toStr().c_str());
161     cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL,
            label);
162     cur_block->InsertInst(cur_inst);
163     if ((gpreg_cnt >= 5 || fpreg_cnt >= 5) && stk_cnt != 0) {
164         auto off = genMachineImm(stk_cnt * 4);
165         auto sp = new MachineOperand(MachineOperand::REG, 13);
166         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
167                                           sp, sp, off);
168         cur_block->InsertInst(cur_inst);
169     }
170     if (dst) {
171         if (dst->getType()->isFloat()) {
172             operand = genMachineFloatOperand(dst);
173             auto s0 = new MachineOperand(MachineOperand::REG, 16, true);
174             cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
175                                            operand, s0);
176             cur_block->InsertInst(cur_inst);
```

```
177         } else {
178             operand = genMachineOperand(dst);
179             auto r0 = new MachineOperand(MachineOperand::REG, 0);
180             cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
181                                            operand, r0);
182             cur_block->InsertInst(cur_inst);
183         }
184     }
185 }
```

- 数组在语法分析部分，由 VarDeclStmt 和 ConstDeclStmt 来生成，主要是类型和名字列表，名字列表 VarDefList 又由每一个 VarDef 组成，VarDef 可以是单一变量或者是数组变量并加上赋值，对于数组变量需要考虑中括号 ArrayIndices, 其里面的值为常量，普通变量直接将其存入到符号表中。

  数组变量需要根据中括号的个数确定其维度并初始化为全零, 如果需要赋值则与之类似, 将相应的值从栈中取出赋予。

  在数组类型转换字符串中，根据数组的维度输出，并根据其类型将其赋到后面并输出中括号以及乘号，在指令构造时对其进行传参，输出 getelementptr inbounds（越界检查）, 然后输出初始指针类型，指针的基址的类型以及一组索引的类型等。

  在语法树中, 如果是多维数组的话, 则循环每一个维度, 如果没有全局的赋值则需要为其声明临时变量；如果到最后一个直接退出循环，否则改变目的指针并进行下一次的循环，当其为右值需要加载指令；如果是一维数组，则直接判断是否有提前声明，没有则为其声明临时变量，否则直接赋值

```
1  void GepInstruction::genMachineCode(AsmBuilder* builder) {
2    auto cur_block = builder->getBlock();
3    MachineInstruction* cur_inst;
4    auto dst = genMachineOperand(operands[0]);
5    auto idx = genMachineOperand(operands[2]);
6    if (init) {
7        if (last) {
8            auto base = genMachineOperand(init);
9            MachineOperand* imm = genMachineImm(off + 4);
10           int off = this->off + 4;
11           if (off > 255) {
12               MachineOperand* temp = genMachineVReg();
13               cur_block->InsertInst(new LoadMInstruction(
14                   cur_block, LoadMInstruction::LDR, temp, imm));
15               imm = temp;
16           }
17           cur_inst = new BinaryMInstruction(
18               cur_block, BinaryMInstruction::ADD, dst, base, imm);
19           cur_block->InsertInst(cur_inst);
20       } else {
21           noAsm = true;
22       }
23       return;
```

```
    }
    MachineOperand* base = nullptr;
    int size;
    auto idx1 = genMachineVReg();
    if (idx->isImm()) {
        if (idx->getVal() < 255) {
            cur_inst =
                new MovMInstruction(cur_block, MovMInstruction::MOV, idx1,
                    idx);
        } else {
            cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
                                            idx1, idx);
        }
        idx = new MachineOperand(*idx1);
        cur_block->InsertInst(cur_inst);
    }
    if (paramFirst) {
        size =
            ((PointerType*)(operands[1]->getType()))->getType()->getSize() /
                8;
    } else {
        if (first) {
            base = genMachineVReg();
            if (operands[1]->getEntry()->isVariable() &&
                ((IdentifierSymbolEntry*)(operands[1]->getEntry()))
                    ->isGlobal()) {
                auto src = genMachineOperand(operands[1]);
                cur_inst = new LoadMInstruction(
                    cur_block, LoadMInstruction::LDR, base, src);
            } else {
                int offset = ((TemporarySymbolEntry*)(operands[1]->getEntry()
                    ))
                                 ->getOffset();
                if (offset > -255 && offset < 255) {
                    cur_inst =
                        new MovMInstruction(cur_block, MovMInstruction::MOV,
                                            base, genMachineImm(offset));
                } else {
                    cur_inst =
                        new LoadMInstruction(cur_block, LoadMInstruction::LDR
                            ,
                                             base, genMachineImm(offset));
                }
            }
            cur_block->InsertInst(cur_inst);
        }

        ArrayType* type =
```

```
68              (ArrayType*)(((PointerType*)(operands[1]->getType()))->getType())
                    ;
69          Type* elementType = type->getElementType();
70          size = elementType->getSize() / 8;
71      }
72      auto size1 = genMachineVReg();
73      if (size > -255 && size < 255) {
74          cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, size1
                ,
75                                         genMachineImm(size));
76      } else {
77          cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
                size1,
78                                          genMachineImm(size));
79      }
80      cur_block->InsertInst(cur_inst);
81      size1 = new MachineOperand(*size1);
82      auto off = genMachineVReg();
83      cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL, off
            ,
84                                        idx, size1);
85      off = new MachineOperand(*off);
86      cur_block->InsertInst(cur_inst);
87      if (paramFirst || !first) {
88          auto arr = genMachineOperand(operands[1]);
89          auto in = operands[1]->getDef();
90          if (in && in->isGep()) {
91              auto gep = (GepInstruction*)in;
92              if (gep->hasNoAsm()) {
93                  gep->setInit(nullptr, 0);
94                  gep->genMachineCode(builder);
95              }
96          }
97          cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
98                                            dst, arr, off);
99          cur_block->InsertInst(cur_inst);
100     } else {
101         // auto addr = genMachineVReg();
102         // auto base1 = new MachineOperand(*base);
103         // cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::
                ADD,
104         //                                   addr, base1, off);
105         // cur_block->InsertInst(cur_inst);
106         // addr = new MachineOperand(*addr);
107         // if (operands[1]->getEntry()->isVariable() &&
108         //     ((IdentifierSymbolEntry*)(operands[1]->getEntry()))->isGlobal
                ()) {
109         //     cur_inst =
```

```
110         //          new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                addr);
111         // } else {
112         //     auto fp = genMachineReg(11);
113         //     cur_inst = new BinaryMInstruction(
114         //         cur_block, BinaryMInstruction::ADD, dst, fp, addr);
115         // }
116         // cur_block->InsertInst(cur_inst);
117         auto addr = genMachineVReg();
118         auto base1 = new MachineOperand(*base);
119         if (operands[1]->getEntry()->isVariable() &&
120             ((IdentifierSymbolEntry*)(operands[1]->getEntry()))->isGlobal())
                {
121             cur_inst = new BinaryMInstruction(
122                 cur_block, BinaryMInstruction::ADD, addr, base1, off);
123             cur_block->InsertInst(cur_inst);
124             addr = new MachineOperand(*addr);
125             cur_inst =
126                 new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                        addr);
127         } else {
128             auto fp = genMachineReg(11);
129             cur_inst = new BinaryMInstruction(
130                 cur_block, BinaryMInstruction::ADD, addr, fp, base1);
131             cur_block->InsertInst(cur_inst);
132             addr = new MachineOperand(*addr);
133             cur_inst = new BinaryMInstruction(
134                 cur_block, BinaryMInstruction::ADD, dst, addr, off);
135         }
136         cur_block->InsertInst(cur_inst);
137     }
138 }
```

- 三维数组 Ast.cpp

  在 ‘Ast.cpp‘ 文件中，找到需要使用三维数组的 ‘BinaryExpr::output‘ 函数。在该函数中，添加代码来处理三维数组的输出。

```
1  void BinaryExpr::output(int level)
2  {// 输出语法树
3      std::string op_str;
4      switch (op)
5      {
6      case ADD:
7          op_str = "add";
8          break;
9      case SUB:
10         op_str = "sub";
11         break;
```

```
        case MUL:
            op_str = "mul";
            break;
        case DIV:
            op_str = "div";
            break;
        case MOD:
            op_str = "mod";
            break;
        case AND:
            op_str = "and";
            break;
        case OR:
            op_str = "or";
            break;
        case LESS:
            op_str = "less";
            break;
        case LEQ:
            op_str = "lessequal";
            break;
        case MORE:
            op_str = "greater";
            break;
        case MEQ:
            op_str = "greaterequal";
            break;
        case EQ:
            op_str = "equal";
            break;
        case NE:
            op_str = "notequal";
            break;
        // 添加处理三维数组的代码
        case ARRAY_ACCESS:
            op_str = "array_access";
            // 假设数组名为 array3D, 索引变量为 i、j、k
            fprintf(stderr, "array3D[%d][%d][%d]\n", i, j, k);
            break;
        }
        fprintf(yyout, "%*cBinaryExpr\top: %s\n", level, ' ', op_str.c_str());
        expr1->output(level + 4);
        expr2->output(level + 4);
}
```

parser.y

```
%{
%union {
```

```
    Type* type;
}

%token T_INT
%token T_IDENTIFIER
%token T_LEFT_BRACKET
%token T_RIGHT_BRACKET
%token T_SEMICOLON

%type <type> type

%%

program: declaration_list

declaration_list: declaration
                | declaration_list declaration

declaration: type declarator_list T_SEMICOLON

type: T_INT
    | T_INT T_LEFT_BRACKET T_RIGHT_BRACKET
    | T_INT T_LEFT_BRACKET T_RIGHT_BRACKET T_LEFT_BRACKET T_RIGHT_BRACKET
      T_LEFT_BRACKET T_RIGHT_BRACKET // 三维数组

declarator_list: declarator
                | declarator_list T_IDENTIFIER

declarator: T_IDENTIFIER
          | T_IDENTIFIER T_LEFT_BRACKET T_RIGHT_BRACKET
          | T_IDENTIFIER T_LEFT_BRACKET expression T_RIGHT_BRACKET

%%
```

十二、 总结

gitlab 代码库地址:

https://gitlab.eduxiji.net/nku2023-az/compilation.git

在本学期的编译原理实验中，从总体上来说，我们设计实现了一个较为完整的拥有前端和后端、能够生成可执行文件并且检测能够程序运行结果的简单编译器，该编译器的最终设计目的得到了实现。

我们的编译器能够识别标准 SysY 所支持的绝大部分词法符号，支持了变量的定义、数组的定义、函数的声明、各类表达式语句、if-else 条件语句和 while 循环语句，在语义动作上可以生成相应语句动作的四也可以对相应的错误进行检测，从而实现了编译前端。

在中间代码部分我们可以成功将其生成 LLVM 体系下的中间代码，并在对应实验中取得了满分的成绩，最后对于目标代码我们不仅通过了本学期所有的基本和提升要求的 151 个样例，并对寄存器的分配和 SSA 语言进行了优化，整体下来获益匪浅，我们高质量完成了一个相对工程量大的项目，这也是属于程序员的浪漫。

参考文献

[1] 杨俣哲, 李煦阳, 杨科迪, 费迪, 周辰霏, 谢子涵, and 杨科迪. 编译器开发环境部署. 2023.

[2] 杨俣哲, 李煦阳, 孙一丁, 李世阳, 杨科迪, 周辰霏, 尧泽斌, 时浩铭, 贺祎昕, 张书睿. 预备工作 1——了解编译器及 llvm ir 编程. 2023.

[3] 杨俣哲, 李煦阳, 费迪, 张书睿, 贺祎昕, 预备工作 2——定义你的编译器 & 汇编编程

[4] 杨俣哲, 李煦阳, 杨科迪, 孙一丁, 韩佳迅, 朱璟钰, 实现语法分析器

[5] 杨俣哲, 李煦阳, 杨科迪, 孙一丁, 韩佳迅, 朱璟钰, 类型检查与中间代码生成

[6] 杨俣哲, 李煦阳, 杨科迪, 孙一丁, 韩佳迅, 朱璟钰, 完成编译器