



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

编译原理实验报告

实现 SysY 编译器

周延霖 2013921

年级：2020 级

专业：信息安全

指导教师：王刚

2023 年 1 月 8 日

摘要

在本学期编译原理课程中完整实现了一个 SysY 语言的编译器，在本篇文章中将对其中从开始的词法分析、语法分析到后来的中间代码、目标代码生成进行逐一介绍，并在最后对完成所有任务后的优化进行简介。

关键字：SysY 语言、编译原理、指令翻译

目录

一、 引言	1
二、 词法分析	2
(一) 模块设计	2
1. 功能简述	2
2. 设计目标	2
(二) 个人贡献	2
1. 负责任务	2
2. 任务实现	3
3. 结果展示	6
三、 语法分析	11
(一) 模块设计	11
1. 功能简述	11
2. 设计目标	11
(二) 个人贡献	11
1. 负责任务	11
2. 任务实现	12
3. 结果展示	17
四、 类型检查 & 中间代码生成	19
(一) 模块设计	19
1. 功能简述	19
2. 设计目标	19
(二) 个人贡献	19
1. 负责任务	19
2. 任务实现	20
3. 结果展示	34
五、 目标代码生成	37
(一) 模块设计	37
1. 功能简述	37
2. 设计目标	37
(二) 个人贡献	37
1. 负责任务	37
2. 任务实现	38

3. 结果展示	60
六、 代码优化	65
(一) 思想简介	65
1. Mem2Reg	65
2. 图着色寄存器分配	66
(二) 个人贡献	66
七、 总结与展望	67
(一) 学期总结	67
(二) 未来展望	67

一、 引言

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基本方法和技巧，融会贯通本课程所学专业理论知识：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 语言支持的词法符号，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，所设计的文法支持了函数、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。

本次的所有代码已经上传于 gitlab，链接如下：

SSH: [git@gitlab.eduxiji.net:nku-hlast/lab2.git](ssh://git@gitlab.eduxiji.net:nku-hlast/lab2.git)

HTTP: <https://gitlab.eduxiji.net/nku-hlast/lab2.git>

二、词法分析

(一) 模块设计

1. 功能简述

在词法分析的实验中，我们利用 Flex 工具实现了词法分析器，识别输入的 SysY 语言程序中的所有单词，并将其转化为单词流，输出为每一个文法单元类别、词素，以及必要的属性。

例如，对于 NUMBER 会有属于它的“数值”属性；对于 ID 会有它在符号表的“序号”，有些标识符会有相同的“序号”。

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。

2. 设计目标

首先，Flex 的词法规则主要由 3 部分组成，部分之间使用 %% 分隔

- 第一部分：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。
- 第二部分：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。
- 第三部分：是 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。

在本次实验中，主要利用已有框架，在第二部分完成对 8 进制、16 进制等整型的翻译，并识别相应的符号将其存入到符号表中，最后通过分析 .sy 文件将其中的重要信息输出。

(二) 个人贡献

1. 负责任务

由于本次实验整体上不复杂，所以我们首先分别对自己的词法分析器进行实现，然后进行整合，每个部分选出实现更简单完成较好的代码，最后提交到 gitlab 上。

主要任务如下：

- 补全缺失的正则定义
- 参考“int”的规则，完成其他大部分终结符的规则定义
- 完成八进制数字、十六进制数字的规则定义，将其保存为十进制输出
- 完成标识符相关规则定义以及符号表
- 完成注释的正则定义和规则定义
- 完成对程序的输出

2. 任务实现

正则定义 对于正则定义部分需要完成对例如八进制数字、十六进制数字、行注释等内容的补全，补全后的正则定义部分如下：

词法分析部分的正则定义

```

1 HEX (0x[1-9|A-F|a-f][0-9|A-F|a-f]*|0x0)
2 OCT (0[1-7][0-7]*|00)
3 DECIMAL ([1-9][0-9]*|0)
4 ID ([:alpha:][:digit:])*
5 char ( ' ' )
6 EOL ( \r\n | \n | \r )
7 WHITE [ \t ]
8 linecomment \/\/*. *
9 commentbegin "/*"
10 commentelement . | \n
11 commentend "*/"
12
13 %x BLOCKCOMMENT

```

如上所示，仿照 10 进制对 8 进制和 16 进制进行补充定义，比如像 8 进制当识别到 0 开始的数字串的时候就可以确定是 8 进制串了，并加上了识别符号（由字母、数字和符号组成）以及注释等正则定义，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。

终结符定义 对于大部分的终结符，其实“int”已经写的非常明确了，只需要将 int 改个名字，其余部分照做即可，比如说 void 的定义如下：

仿照 int 定义 void

```

1 "void" {
2     #ifdef ONLY_FOR_LEX
3         char content[100];
4         sprintf(content,"%-10s","VOID");
5         AppendChar(content,"void");
6         // strcpy(content,"VOID\tvoid");
7         AddLineNo(content,yylineno);
8         AddOffset(content,offset);
9         offset += 4;
10        DEBUG_FOR_LAB4(content);
11    #else
12        return VOID;
13    #endif
14 }

```

可以看到只需将相应位置改名即可，但要注意的是 return 后面的返回值是需要为语法分析做铺垫的，所以此次命名的返回值在将来语法分析器部分也是要用相同的名字，所以按照关键词来设计就不会造成冲突。

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：

16 进制转换

```
1 {HEX} {
2     #ifdef ONLY_FOR_LEX
3         string a = yytext;
4         int len = a.length() - 2;
5         int all = 0;
6         for(int i = 0; i < len; i++)
7         {
8             int temp1 = (pow(16, (len - i - 1)));
9             int temp2;
10            if(a[i + 2] >= '0' && a[i + 2] <= '9'){
11                temp2 = (a[i + 2] - '0');
12            }
13            else if(a[i + 2] >= 'A' && a[i + 2] <= 'F'){
14                temp2 = a[i + 2] - 'A' + 10;
15            }
16            else if(a[i + 2] >= 'a' && a[i + 2] <= 'f'){
17                temp2 = a[i + 2] - 'a' + 10;
18            }
19            all = all + temp1 * temp2;
20        }
21        string temp = "HEX\t" + to_string(all) + "(DEC)" + "\tlinecount:" +
22            to_string(yylineno);
23        DEBUG_FOR_LAB4(temp);
24    #else
25        return HEX;
26    #endif
27 }
```

逻辑思路是循环输入字符的每一位，根据时字符还是数字来对其进行对应的加减操作，并乘以当前所在的位置的 16 的次方，然后将其加到总和中，最后返回即可。

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：

作用域的判断

```
1 "}" {
2     #ifdef ONLY_FOR_LEX
3         global_count_of_id++;
```

```

4      a.push_back(global_count_of_id);
5      //cout << "{" << a.back() << endl;
6      leftNum++;
7      string temp = "LBRACE\t{\tlinecount:" + to_string(yylineno);
8      DEBUG_FOR_LAB4(temp);
9  #else
10     return LBRACE;
11 #endif
12 }
13 "}" {
14     #ifdef ONLY_FOR_LEX
15         a.pop_back();
16         //cout << "}" << a.back() << endl;
17         rightNum++;
18         string temp = "RBRACE\t}\tlinecount:" + to_string(yylineno);
19         DEBUG_FOR_LAB4(temp);
20     #else
21         return RBRACE;
22     #endif
23 }

```

事先定义一个全局的变量 `global_count_of_id`，在每次出现左大括号时将其加一并加入容器，在每次出现右大括号是将其弹出，这样在任何时候出现标识符都可以知道当前对应的数字，来区分不同的作用域。

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：

标识符的实现

```

1 {ID} {
2     #ifdef ONLY_FOR_LEX
3         string m = yytext;
4         //cout << yytext << ":" << a.size() << endl;
5         int idnum = 0;
6         bool flag = 0;
7         for(int i = 0; i < id_count; i++)
8         {
9             if(table[i].name_of_id == yytext && table[i].global_count_of_id
10                == a.back())
11             {
12                 idnum = i + 1;
13                 flag = 1;
14                 break;
15             }
16         }
17         if(flag == 0)
18         {
19             table[id_count].name_of_id = yytext;
20             if(a.size() != 0){

```



```

20         table[id_count].global_count_of_id = a.back();
21     }
22     id_count++;
23     idnum = id_count;
24 }
25 string temp = "ID\t" + m + "\tlinecount:" + to_string(yylineno) + "\tIDcount:" + to_string(idnum);
26 DEBUG_FOR_LAB4(temp);
27 #else
28     return ID;
29 #endif
30 }

```

在每次出现标识符时在符号表中查找，如果有就直接输出，如果没有就根据当前的作用域和它的值将其加入符号表中即可

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，在这里解释一下对结束注释的处理，其余类似：

结束注释的处理

```

1 <BLOCKCOMMENT>{commentend} {
2     #ifdef ONLY_FOR_LEX
3         string a = yytext;
4         if(a.find("\n") != a.npos)
5         {
6             yylineno++;
7             a.replace(a.find("\n"),1,"");
8         }
9         string temp = "commentend\t" + a + "\tlinecount:" + to_string(
10             yylineno);
11         DEBUG_FOR_LAB4(temp);
12         BEGIN INITIAL;
13     #else
14         return commentend;
15     #endif
16 }

```

当出现换行时，需要将全局变量 yylineno 进行加 1 操作，并在结束完成后切换到初始状态。

程序输出 程序的输出其实主要是看是否成功捕获到相应的关系，所以将每个识别出来的变量赋予一个字符串，然后调用输出函数即可完成。

3. 结果展示

词法分析部分手写的样例代码如下：

词法分析样例代码

```

1 // i am a function: this function has many syntax errors.
2

```

```

3  int f(){
4      int a;
5      a = 0;
6      while(a<10){
7          a = a * 2345;
8      }
9      return a;
10 }
11
12 int main(){
13     int _testid;
14     _testid = 0;
15     if(_testid==0){
16         int _testid;
17         int _testid2;
18         _testid1=_testid1+1;
19     }
20     /*
21         Comment lines;
22     */
23     {
24         int a;
25     }
26     {
27         int a;
28     }
29     int b;
30     return 0;
31 }
32
33 /*
34     Comment lines.
35 */

```

词法分析所对应的元素分析如下：

词法分析结果

1	[DEBUG LAB4]:	linecomment	// i am a function: this function has many syntax errors.	linecount:1	
2	[DEBUG LAB4]:	INT	int	linecount:3	
3	[DEBUG LAB4]:	ID	f	linecount:3	IDcount:1
4	[DEBUG LAB4]:	LPAREN	(linecount:3	
5	[DEBUG LAB4]:	RPAREN)	linecount:3	
6	[DEBUG LAB4]:	LBRACE	{	linecount:3	
7	[DEBUG LAB4]:	INT	int	linecount:4	
8	[DEBUG LAB4]:	ID	a	linecount:4	IDcount:2
9	[DEBUG LAB4]:	SEMICOLON	;	linecount:4	
10	[DEBUG LAB4]:	ID	a	linecount:5	IDcount:2
11	[DEBUG LAB4]:	ASSIGN	=	linecount:5	

```

12 [DEBUG LAB4]:  DECIMAL      0      linecount:5
13 [DEBUG LAB4]:  SEMICOLON    ;      linecount:5
14 [DEBUG LAB4]:  WHILE   while  linecount:6
15 [DEBUG LAB4]:  LPAREN   (      linecount:6
16 [DEBUG LAB4]:  ID       a      linecount:6      IDcount:2
17 [DEBUG LAB4]:  LESS    <      linecount:6
18 [DEBUG LAB4]:  DECIMAL    10     linecount:6
19 [DEBUG LAB4]:  RPAREN   )      linecount:6
20 [DEBUG LAB4]:  LBRACE   {      linecount:6
21 [DEBUG LAB4]:  ID       a      linecount:7      IDcount:3
22 [DEBUG LAB4]:  ASSIGN   =      linecount:7
23 [DEBUG LAB4]:  ID       a      linecount:7      IDcount:3
24 [DEBUG LAB4]:  STAR     *      linecount:7
25 [DEBUG LAB4]:  DECIMAL    2345   linecount:7
26 [DEBUG LAB4]:  SEMICOLON    ;      linecount:7
27 [DEBUG LAB4]:  RBRACE   }      linecount:8
28 [DEBUG LAB4]:  RETURN  return  linecount:9
29 [DEBUG LAB4]:  ID       a      linecount:9      IDcount:2
30 [DEBUG LAB4]:  SEMICOLON    ;      linecount:9
31 [DEBUG LAB4]:  RBRACE   }      linecount:10
32 [DEBUG LAB4]:  INT      int    linecount:12
33 [DEBUG LAB4]:  ID       main   linecount:12     IDcount:4
34 [DEBUG LAB4]:  LPAREN   (      linecount:12
35 [DEBUG LAB4]:  RPAREN   )      linecount:12
36 [DEBUG LAB4]:  LBRACE   {      linecount:12
37 [DEBUG LAB4]:  INT      int    linecount:13
38 [DEBUG LAB4]:  ID       _testid linecount:13     IDcount:5
39 [DEBUG LAB4]:  SEMICOLON    ;      linecount:13
40 [DEBUG LAB4]:  ID       _testid linecount:14     IDcount:5
41 [DEBUG LAB4]:  ASSIGN   =      linecount:14
42 [DEBUG LAB4]:  DECIMAL    0      linecount:14
43 [DEBUG LAB4]:  SEMICOLON    ;      linecount:14
44 [DEBUG LAB4]:  IF       if      linecount:15
45 [DEBUG LAB4]:  LPAREN   (      linecount:15
46 [DEBUG LAB4]:  ID       _testid linecount:15     IDcount:5
47 [DEBUG LAB4]:  EQUAL    ==     linecount:15
48 [DEBUG LAB4]:  DECIMAL    0      linecount:15
49 [DEBUG LAB4]:  RPAREN   )      linecount:15
50 [DEBUG LAB4]:  LBRACE   {      linecount:15
51 [DEBUG LAB4]:  INT      int    linecount:16
52 [DEBUG LAB4]:  ID       _testid linecount:16     IDcount:6
53 [DEBUG LAB4]:  SEMICOLON    ;      linecount:16
54 [DEBUG LAB4]:  INT      int    linecount:17
55 [DEBUG LAB4]:  ID       _testid2 linecount:17     IDcount:7
56 [DEBUG LAB4]:  SEMICOLON    ;      linecount:17
57 [DEBUG LAB4]:  ID       _testid1 linecount:18     IDcount:8
58 [DEBUG LAB4]:  ASSIGN   =      linecount:18
59 [DEBUG LAB4]:  ID       _testid1 linecount:18     IDcount:8

```

```

60 [DEBUG LAB4]: ADD      +      linecount:18
61 [DEBUG LAB4]: DECIMAL  1      linecount:18
62 [DEBUG LAB4]: SEMICOLON ;      linecount:18
63 [DEBUG LAB4]: RBRACE   }      linecount:19
64 [DEBUG LAB4]: commentbegin /*    linecount:20
65 [DEBUG LAB4]: commentelement      linecount:21
66 [DEBUG LAB4]: commentelement      linecount:21
67 [DEBUG LAB4]: commentelement      linecount:21
68 [DEBUG LAB4]: commentelement C    linecount:21
69 [DEBUG LAB4]: commentelement o    linecount:21
70 [DEBUG LAB4]: commentelement m    linecount:21
71 [DEBUG LAB4]: commentelement m    linecount:21
72 [DEBUG LAB4]: commentelement e    linecount:21
73 [DEBUG LAB4]: commentelement n    linecount:21
74 [DEBUG LAB4]: commentelement t    linecount:21
75 [DEBUG LAB4]: commentelement      linecount:21
76 [DEBUG LAB4]: commentelement l    linecount:21
77 [DEBUG LAB4]: commentelement i    linecount:21
78 [DEBUG LAB4]: commentelement n    linecount:21
79 [DEBUG LAB4]: commentelement e    linecount:21
80 [DEBUG LAB4]: commentelement s    linecount:21
81 [DEBUG LAB4]: commentelement ;    linecount:21
82 [DEBUG LAB4]: commentelement      linecount:22
83 [DEBUG LAB4]: commentelement      linecount:22
84 [DEBUG LAB4]: commentend  */     linecount:22
85 [DEBUG LAB4]: LBRACE   {      linecount:23
86 [DEBUG LAB4]: INT      int    linecount:24
87 [DEBUG LAB4]: ID       a      linecount:24   IDcount:9
88 [DEBUG LAB4]: SEMICOLON ;      linecount:24
89 [DEBUG LAB4]: RBRACE   }      linecount:25
90 [DEBUG LAB4]: LBRACE   {      linecount:26
91 [DEBUG LAB4]: INT      int    linecount:27
92 [DEBUG LAB4]: ID       a      linecount:27   IDcount:10
93 [DEBUG LAB4]: SEMICOLON ;      linecount:27
94 [DEBUG LAB4]: RBRACE   }      linecount:28
95 [DEBUG LAB4]: INT      int    linecount:29
96 [DEBUG LAB4]: ID       b      linecount:29   IDcount:11
97 [DEBUG LAB4]: SEMICOLON ;      linecount:29
98 [DEBUG LAB4]: RETURN   return  linecount:30
99 [DEBUG LAB4]: DECIMAL  0      linecount:30
100 [DEBUG LAB4]: SEMICOLON ;      linecount:30
101 [DEBUG LAB4]: RBRACE   }      linecount:31
102 [DEBUG LAB4]: commentbegin /*    linecount:33
103 [DEBUG LAB4]: commentelement      linecount:34
104 [DEBUG LAB4]: commentelement      linecount:34
105 [DEBUG LAB4]: commentelement C    linecount:34
106 [DEBUG LAB4]: commentelement o    linecount:34
107 [DEBUG LAB4]: commentelement m    linecount:34

```

108	[DEBUG LAB4]:	commentelement	m	linecount:34
109	[DEBUG LAB4]:	commentelement	e	linecount:34
110	[DEBUG LAB4]:	commentelement	n	linecount:34
111	[DEBUG LAB4]:	commentelement	t	linecount:34
112	[DEBUG LAB4]:	commentelement		linecount:34
113	[DEBUG LAB4]:	commentelement	l	linecount:34
114	[DEBUG LAB4]:	commentelement	i	linecount:34
115	[DEBUG LAB4]:	commentelement	n	linecount:34
116	[DEBUG LAB4]:	commentelement	e	linecount:34
117	[DEBUG LAB4]:	commentelement	s	linecount:34
118	[DEBUG LAB4]:	commentelement	.	linecount:34
119	[DEBUG LAB4]:	commentelement		linecount:35
120	[DEBUG LAB4]:	commentend	*/	linecount:35

NOT

三、 语法分析

(一) 模块设计

1. 功能简述

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。

之后的所有实验，树上各结点信息的获取与流动、类型检查、翻译至中间代码，都可以看作对该树进行一次遍历。若一些操作需要考虑语法，比如构建作用域树，那么通过语法树上一次遍历，便可以很容易完成。

2. 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) 个人贡献

1. 负责任务

这一次的任务较上一次复杂，所以我们进行了明确的分工，我主要是负责运算符、循环等操作，我的队友主要负责函数等操作，我的具体任务如下：

1. 变量、常量的声明和初始化。
2. 语句：
 - 赋值 (=)
 - 表达式语句
 - 语句块
 - if、while、return
3. 表达式：
 - 算术运算 (+、-、*、/、%，其中 +、- 都可以是单目运算符)
 - 关系运算 (==, >, <, >=, <=, !=)
 - 逻辑运算 (&&, ||, !)

2. 任务实现

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。本次实现的符号表条目包含了标识符的词素、类型、作用域、行号等信息。

本次实现的符号表主要用于作用域的管理，为每个语句块都创建一个符号表，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。当该标识符用于声明，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。

框架代码中已经有插入函数的实现，`SymbolTable.cpp`中符号表的查找函数的实现如下：

查找函数

```
1 SymbolEntry* SymbolTable::lookup(std::string name)
2 {
3     // Todo
4     SymbolTable *t = this;
5     while(t != nullptr)
6     {
7         if(t -> symbolTable[name] != 0)
8         {
9             return t -> symbolTable[name];
10        }
11        t = t -> prev;
12    }
13    return nullptr;
14 }
```

查找函数实现的思路为遍历符号表，查找到与其名字相同值所储存的值，并将其返回即可。

抽象语法树 语法分析的目的是构建出一棵抽象语法树（AST），因此我们设计了语法树的结点。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。

二元表达式 对于二元表达式来说，具体实现如下：

二元表达式

```
1 void BinaryExpr::output(int level)
2 {
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;
12        case AND:
```

```
13         op_str = "and";
14         break;
15     case OR:
16         op_str = "or";
17         break;
18     case LESS:
19         op_str = "less";
20         break;
21     case MORE:
22         op_str = "more";
23         break;
24     case MOREEQUAL:
25         op_str = "moreequal";
26         break;
27     case LESSEQUAL:
28         op_str = "lessequal";
29         break;
30     case EQUAL:
31         op_str = "equal";
32         break;
33     case NOEQUAL:
34         op_str = "noequal";
35         break;
36     case DIV:
37         op_str = "div";
38         break;
39     case MUL:
40         op_str = "mul";
41         break;
42     case PERC:
43         op_str = "mod";
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 }
```

实现起来并不复杂，只需要按照相应传入的操作，将其赋给对应的字符串，然后递归输出两个子表达式即可。

语句 对于语句的输出并不复杂：

语句

```
1 void CompoundStmt::output(int level)
2 {
3     fprintf(yyout, "%*cCompoundStmt\n", level, ' ');
4     stmt->output(level + 4);
}
```



```

5 }
6
7 void SeqNode::output(int level)
8 {
9     fprintf(yyout, "%*cSequence\n", level, ' ');
10    stmt1->output(level + 4);
11    stmt2->output(level + 4);
12 }
13
14 void IfElseStmt::output(int level)
15 {
16     fprintf(yyout, "%*cIfElseStmt\n", level, ' ');
17     cond->output(level + 4);
18     thenStmt->output(level + 4);
19     elseStmt->output(level + 4);
20 }
21
22 void AssignStmt::output(int level)
23 {
24     fprintf(yyout, "%*cAssignStmt\n", level, ' ');
25     lval->output(level + 4);
26     expr->output(level + 4);
27 }

```

以上展示了像赋值、条件判断、语句块等操作，要么直接输出，要么递归输出即可，因为此次知识语法分析，并没有具体的功能，所以输出到抽象语法树即可。

声明和初始化 由于变量和常量在进行初始化的时候进行的操作类似，所以在此分析一下变量的操作：

声明和初始化

```

1 void Id::output(int level)
2 {
3     std::string name, type;
4     int scope;
5     name = symbolEntry->toStr();
6     type = symbolEntry->getType()->toStr();
7     scope = dynamic_cast<IdentifierSymbolEntry*>(symbolEntry)->getScope();
8     fprintf(yyout, "%*cId\tname: %s\tscope: %d\ttype: %s\n", level, ' ',
9             name.c_str(), scope, type.c_str());
10 }

```

在变量输出到抽象语法树时，需要获得其类型、作用域、名字等，然后将其输出即可。

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。

if、while、return 对于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，具体实现如下：

if、while、return

```

1 IfStmt
2   : IF LPAREN Cond RPAREN Stmt %prec THEN {
3       $$ = new IfStmt($3, $5);
4   }
5   | IF LPAREN Cond RPAREN LBRACE RBRACE{
6       $$ = new IfStmt($3, new Empty());
7   }
8   | IF LPAREN Cond RPAREN Stmt ELSE Stmt {
9       $$ = new IfElseStmt($3, $5, $7);
10  }
11  ;
12 WhileStmt
13   : WHILE LPAREN Cond RPAREN Stmt {
14       $$ = new WhileStmt($3, $5);
15   }
16  ;
17 ReturnStmt
18   :
19   RETURN Exp SEMICOLON{
20       $$ = new ReturnStmt($2);
21   }
22  ;

```

为了解决悬空-else 文法的二义性问题，使用%precTHEN 来避免，相当于给终结符声明优先级，这样终结符 else 的优先级高于终结符 then。产生式的优先级和右部最后一个终结符的优先级相同，在发生移入/归约冲突时，通过比较向前看符号和产生式的优先级来解决冲突，若向前看符号的优先级更高，则进行移入，若产生式的优先级更高，则进行归约。这里 else 的优先级更高，因此会将 else 移入。

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

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（在标识符的定义处需要根据其是否在当前的符号表中判断是否要为其重新生成一个 ID 并将其储存）具体实现如下：

赋值和语句块

```

1 AssignStmt
2   :
3   LVal ASSIGN Exp SEMICOLON {
4       $$ = new AssignStmt($1, $3);
5   }
6   ;

```

```

7 BlockStmt
8   : LBRACE
9     {identifiers = new SymbolTable(identifiers);}
10   Stmts RBRACE
11   {
12     $$ = new CompoundStmt($3);
13     SymbolTable *top = identifiers;
14     identifiers = identifiers->getPrev();
15     delete top;
16   }
17   ;
18 LVal
19   : ID {
20     SymbolEntry *se;
21     se = identifiers->lookup($1);
22     if(se == nullptr)
23     {
24       fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
25       delete [] (char*)$1;
26       assert(se != nullptr);
27     }
28     $$ = new Id(se);
29     delete [] $1;
30   }
31   ;

```

二元表达式 对于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， $\text{Cond} \rightarrow \text{LOrExp} \rightarrow \text{LAndExp} \rightarrow \text{RelExp} \rightarrow \text{AddExp} \rightarrow \text{MulExp} \rightarrow \text{UnaryExp} \rightarrow \text{PrimaryExp}$ ，之间的生成方式用其中一种来举例，比如由乘法表达式生成加法表达式如下：

乘法到加法

```

1 AddExp
2   :
3     MulExp {$$ = $1;}
4     |
5     AddExp ADD MulExp
6     {
7       SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::intType,
8         SymbolTable::getLabel());
9       $$ = new BinaryExpr(se, BinaryExpr::ADD, $1, $3);
10    }
11    |
12    AddExp SUB MulExp
13    {
14      SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::intType,

```

```

14         SymbolTable::getLabel());
15     $$ = new BinaryExpr(se, BinaryExpr::SUB, $1, $3);
16 }
;

```

首先其可以为单一的乘除法不用变，或者由自己加减子节点乘除法表达式即可生成，其余的表达式生成与之类似，按照箭头的顺序来一一生成即可，由于代码量太大，在这里就不一一列举了（已经提交到 gitlab 上了）

3. 结果展示

语法分析部分手写的样例代码如下：

语法分析样例代码

```

1  int a;
2  int main()
3  {
4      if(!a){
5          int z;
6      }
7      int a;
8      a = 1 + 2;
9      if(a < 5)
10         return 1;
11     return 0;
12 }

```

语法分析生成的语法树如下：

语法分析结果

```

1  program
2      Sequence
3          DeclStmt
4              IdList
5                  Id      name: a scope: 0      type: int
6          FunctionDefine function name: main, type: int()
7              CompoundStmt
8                  Sequence
9                      Sequence
10                         Sequence
11                             Sequence
12                                 IfStmt
13                                     SignleExpr op: anti
14                                         Id      name: a scope: 0      type:
15                                             int
16                                     CompoundStmt
17                                         DeclStmt
18                                             IdList

```

```
18                                     Id      name: z scope: 3
19                                     type: int
20                               DeclStmt
21                               IdList
22                               Id      name: a scope: 2      type:
23                               int
24                               AssignStmt
25                               Id      name: a scope: 2      type: int
26                               BinaryExpr  op: add
27                               IntegerLiteral  value: 1      type:
28                               int
29                               IntegerLiteral  value: 2      type:
30                               int
31                               IfStmt
32                               BinaryExpr  op: less
33                               Id      name: a scope: 2      type: int
34                               IntegerLiteral  value: 5      type: int
35                               ReturnStmt
36                               IntegerLiteral  value: 1      type: int
37                               ReturnStmt
38                               IntegerLiteral  value: 0      type: int
```

四、 类型检查 & 中间代码生成

(一) 模块设计

1. 功能简述

在本次实验中，需要检查语法分析生成的抽象语法树的正确性，即进行语义分析，并在遇到不符合表达式规定的代码进行退出报错（比如重定义和匹配等错误），在没有语法问题的基础上生成LLVM中间代码。

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。

2. 设计目标

类型检查 类型检查是编译过程的重要一步，需要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。在类型检查的实验中须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。

我通过在建立语法树的过程中进行相应的识别和处理。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。

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。主要需要完成对流程图构造，以及对构造完成后的翻译。

整体的框架是 Unit→Function→BasicBlock→Instruction→Operand→Type 来构造，这样从顶向下，分层实现功能。

(二) 个人贡献

1. 负责任务

这一次的任务也是比较复杂的，所以我们采用分工进行实现，由于上一次的分工大致是我负责循环，变量，基本块等构造，我的队友负责函数等的实现，所以这一次也是沿用上次的分工，我的具体任务如下：

类型检查 负责检查的类型：

- 变量未声明，及在同一作用域下重复声明
- 条件判断表达式 int 至 bool 类型隐式转换
- 数值运算表达式运算数类型是否正确
- 判断 break、continue 语句是否仅出现在 while 语句中

中间代码 中间代码生成部分：

1. 中间代码生成

- 变量、常量的声明和初始化
- 算术表达式的翻译
- 关系表达式的翻译
- 布尔表达式的翻译
- 控制流语句的翻译, if-else 语句、while 语句

2. 根据基本块的前驱、后继关系进行流图的构造

2. 任务实现**中间代码**

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:

bool 类型变量

```

1  class Type
2  {
3  protected:
4      enum {INT, VOID, FUNC, PTR, BOOL};
5  public:
6      int kind;
7      Type(int kind) : kind(kind) {};
8      virtual ~Type() {};
9      virtual std::string toStr() = 0;
10     bool isInt() const {return kind == INT;};
11     bool isVoid() const {return kind == VOID;};
12     bool isFunc() const {return kind == FUNC;};
13     bool isBool() const {return kind == BOOL;};
14 };
15
16 class BoolType : public Type
17 {
18 public:
19     BoolType() : Type(Type::BOOL) {};
20     std::string toStr();
21     Type* getrettype(){return NULL;}
22 };

```

在指令方面增加了函数调用、零扩展等指令, 零扩展主要是为了将 bool 值与 int 值进行匹配, AllocInstruction2 指令是为了分配全局变量定义的, 需要与局部变量区分开来。

指令扩展

```

1  class Instruction
2  {

```

```

3 public:
4     Instruction(unsigned instType, BasicBlock *insert_bb = nullptr);
5     virtual ~Instruction();
6     BasicBlock *getParent();
7     bool isUncond() const {return instType == UNCOND;};
8     bool isCond() const {return instType == COND;};
9     void setParent(BasicBlock *);
10    void setNext(Instruction *);
11    void setPrev(Instruction *);
12    Instruction *getNext();
13    Instruction *getPrev();
14    virtual void output() const = 0;
15 protected:
16     unsigned instType;
17     unsigned opcode;
18     Instruction *prev;
19     Instruction *next;
20     BasicBlock *parent;
21     std::vector<Operand*> operands;
22     enum {BINARY, COND, UNCOND, RET, LOAD, STORE, CMP, ALLOCA, CALL, XOR,
23           ZEXT};
24 };
25 class AllocaInstruction2 : public Instruction
26 {
27 public:
28     Operand *src;
29     AllocaInstruction2(Operand *src, Operand *dst, SymbolEntry *se,
30                       BasicBlock *insert_bb = nullptr);
31     ~AllocaInstruction2();
32     void output() const;
33 private:
34     SymbolEntry *se;
35 };
36 class ZextInstruction : public Instruction
37 {
38 public:
39     ZextInstruction(Operand *dst, Operand *src, BasicBlock *insert_bb =
40                   nullptr);
41     void output() const;
42 };

```

在建立语法树中增加了回填错误列表, 这个回填错误列表与原本的回填实现类似, 只不过是在判断条件的时候看起来逻辑更加清楚; 并在二元表达式中增加乘除、大于等于等操作符, 并上次相同增加一些例如函数元素声明, while 循环、函数调用等新状态。

语法树补充


```

1  class Node
2  {
3  private:
4      static int counter;
5      int seq;
6  protected:
7      std::vector<Instruction*> true_list;
8      std::vector<Instruction*> false_list;
9      static IRBuilder *builder;
10     void backPatch(std::vector<Instruction*> &list, BasicBlock*bb);
11     void backPatchFalse(std::vector<Instruction*> &list, BasicBlock*bb);
12     std::vector<Instruction*> merge(std::vector<Instruction*> &list1, std::
        vector<Instruction*> &list2);
13
14 public:
15     Node();
16     int getSeq() const {return seq;};
17     static void setIRBuilder(IRBuilder*ib) {builder = ib;};
18     virtual void output(int level) = 0;
19     virtual void typeCheck() = 0;
20     virtual void genCode() = 0;
21     std::vector<Instruction*>& trueList() {return true_list;}
22     std::vector<Instruction*>& falseList() {return false_list;}
23 };
24
25 class BinaryExpr : public ExprNode
26 {
27 private:
28     int op;
29     ExprNode *expr1, *expr2;
30 public:
31     enum {ADD, MUL, DIV, PERC, SUB, AND, OR, LESS, MOREEQUAL, LESSEQUAL,
        EQUAL, NOEQUAL, MORE};
32     BinaryExpr(SymbolEntry *se, int op, ExprNode*expr1, ExprNode*expr2) :
        ExprNode(se), op(op), expr1(expr1), expr2(expr2){dst = new Operand(se
        );};
33     void output(int level);
34     void typeCheck();
35     void genCode();
36 };

```

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：

输入输出

```

1  FunctionType *type1 = new FunctionType(TypeSystem::intType, {});

```

```

2   FunctionType *type2 = new FunctionType(TypeSystem::voidType, {});
3   SymbolEntry *se1 = new IdentifierSymbolEntry(type1, "getint", identifiers
    →getLevel());
4   SymbolEntry *se2 = new IdentifierSymbolEntry(type1, "getch", identifiers
    →getLevel());
5   SymbolEntry *se3 = new IdentifierSymbolEntry(type2, "putint", identifiers
    →getLevel());
6   SymbolEntry *se4 = new IdentifierSymbolEntry(type2, "putch", identifiers
    →getLevel());
7   identifiers→install("getint", se1);
8   identifiers→install("getch", se2);
9   identifiers→install("putint", se3);
10  identifiers→install("putch", se4);

```

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：

函数的输出

```

1  void Function::output() const
2  {
3      FunctionType* funcType = dynamic_cast<FunctionType*>(sym_ptr→getType());
4      Type *retType = funcType→getRetType();
5      fprintf(yyout, "define %s %s(", retType→toStr().c_str(), sym_ptr→toStr
        ().c_str());
6      for(long unsigned int i = 0; i < params.size(); i++)
7      {
8          if(params[i] != nullptr) //std::cout << "fuck" << std::endl;
9          fprintf(yyout, "i32 %s", (params[i]→toStr().c_str());
10         //std::cout << "1" << std::endl;
11         if(i != params.size() - 1)
12         {
13             fprintf(yyout, ", ");
14         }
15     }
16     fprintf(yyout, "){\n");
17
18     //for(long unsigned int i = 0; i < params.size(); i++)
19
20     // std::cout << "执行了Function的define" << std::endl;
21     std::set<BasicBlock*> v;
22     std::list<BasicBlock*> q;
23     q.push_back(entry);
24     v.insert(entry);
25     while (!q.empty())
26     {
27         // std::cout << "进入了Function的while循环" << std::endl;
28         auto bb = q.front();
29         q.pop_front();

```

```

30     bb->output();
31     for (auto succ = bb->succ_begin(); succ != bb->succ_end(); succ++)
32     {
33         //         std::cout << "进入了function的for循环" << std::endl;
34         if (v.find(*succ) == v.end())
35         {
36             v.insert(*succ);
37             q.push_back(*succ);
38         //         std::cout << "function的if判断完毕" << std::endl;
39         }
40     }
41 }
42 fprintf(yyout, "}\n");
43 }

```

指令输出 在指令定义方面，首先把二元指令增加的乘除模等运算补充上：

二元指令输出

```

1 void BinaryInstruction::output() const
2 {
3     std::string s1, s2, s3, op, type;
4     s1 = operands[0]->toStr();
5     s2 = operands[1]->toStr();
6     s3 = operands[2]->toStr();
7     type = operands[0]->getType()->toStr();
8     switch (opcode)
9     {
10    case ADD:
11        op = "add";
12        break;
13    case SUB:
14        op = "sub";
15        break;
16    case MUL:
17        op = "mul";
18        break;
19    case DIV:
20        op = "sdiv";
21        break;
22    case MOD:
23        op = "srem";
24        break;
25    default:
26        break;
27    }
28    fprintf(yyout, "    %s = %s %s %s, %s\n", s1.c_str(), op.c_str(), type.
29        c_str(), s2.c_str(), s3.c_str());

```

然后仿照局部变量的分配方式来写全局变量的分配，主要是输出必须更改为全局变量的输出方式：

全局变量输出

```

1  AllocaInstruction2::AllocaInstruction2(Operand *src, Operand *dst,
   SymbolEntry *se, BasicBlock *insert_bb) : Instruction(ALLOCA, insert_bb)
2  {
3      operands.push_back(dst);
4      dst->setDef(this);
5      this->se = se;
6      this->src = src;
7  }
8
9  AllocaInstruction2::~AllocaInstruction2()
10 {
11     operands[0]->setDef(nullptr);
12     if(operands[0]->usersNum() == 0)
13         delete operands[0];
14 }
15
16 void AllocaInstruction2::output() const
17 {
18     std::string dst, type;
19     dst = operands[0]->toStr();
20     type = se->getType()->toStr();
21     fprintf(yyout, "    %s = global %s %s, align 4\n", dst.c_str(), type.c_str
        (), (src -> toStr()).c_str());
22 }

```

接下来对于函数调用方面，首先将操作数储存，然后放入各个参数，输出时也是仿照函数定义，只不过这次换成了 call，如下所示：

函数调用输出

```

1  void FunctioncallInstruction::output() const
2  {
3      fprintf(yyout, "    ");
4      FunctionType* type = (FunctionType*)(func -> getType());
5      if(operands[0] && type -> getRetType() != TypeSystem::voidType)
6      {
7          fprintf(yyout, "%s = ", operands[0] -> toStr().c_str());
8      }
9      fprintf(yyout, "call %s %s(", type -> getRetType() -> toStr().c_str(),
        func -> toStr().c_str());
10     for(long unsigned int i = 1; i < operands.size(); i++)
11     {
12         if(i != 1)
13         {

```

```

14         fprintf(yyout, ", ");
15     }
16     fprintf(yyout, "%s %s", operands[i] -> getType() -> toStr().c_str(),
17             operands[i] -> toStr().c_str());
18 }
19 fprintf(yyout, "\n");

```

零扩展指令先将操作数存入并将 bool 的一位扩展至 32 位，余位补零，输出按照标准文件里进行输出，将目的数赋值为其零扩展；异或指令与其类似，先建立操作数，并将其付给其相应的关系，输出也是标准文件的输出。

零扩展和异或指令

```

1 void ZextInstruction ::output() const
2 {
3     Operand* dst=operands[0];
4     Operand* src=operands[1];
5     fprintf(yyout, " %s = zext i1 %s to i32\n",dst->toStr().c_str(),src->
6         toStr().c_str() );
7 }
8 void XorInstruction ::output() const
9 {
10     Operand* dst=operands[0];
11     Operand* src=operands[1];
12     fprintf(yyout, " %s = xor i1 %s, true\n",dst->toStr().c_str(),src->toStr
13         ().c_str() );

```

语法树 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。

二元表达式与错误回填

```

1 void Node::backPatchFalse(std::vector<Instruction*> &list, BasicBlock*bb)
2 {
3     for(auto &inst: list)
4     {
5         if(inst->isCond())
6         {
7             bb->addPred(dynamic_cast<CondBrInstruction*>(inst)->getParent());
8             dynamic_cast<CondBrInstruction*>(inst)->getParent()->addSucc(bb);

```

```

9         dynamic_cast<CondBrInstruction*>(inst)->setFalseBranch(bb);
10    }
11    else if(inst->isUncond())
12    {
13        bb->addPred(dynamic_cast<CondBrInstruction*>(inst)->getParent());
14        dynamic_cast<CondBrInstruction*>(inst)->getParent()->addSucc(bb);
15        dynamic_cast<UncondBrInstruction*>(inst)->setBranch(bb);
16    }
17 }
18 }
19
20 void BinaryExpr::genCode()
21 {
22     BasicBlock *bb = builder->getInsertBB();
23     Function *func = bb->getParent();
24     if (op == AND)
25     {
26         BasicBlock *trueBB = new BasicBlock(func); // if the result of lhs
27             is true, jump to the trueBB. 第二个子表达式生成的指令需要插入的位
28             置
29         trueBB->addPred(bb); // 放置这个基本块make
30         bb->addSucc(trueBB); // 放置这个基本块
31         expr1->genCode();
32         backPatch(expr1->>trueList(), trueBB);
33         builder->setInsertBB(trueBB); // set the insert point
34             to the trueBB so that intructions generated by expr2 will be
35             inserted into it
36         expr2->genCode();
37         true_list = expr2->>trueList();
38         false_list = merge(expr1->>falseList(), expr2->>falseList());
39         dst -> getType() -> kind = 4;
40     }
41     else if(op == OR)
42     {
43         // Todo
44         BasicBlock *falseBB = new BasicBlock(func);
45         expr1 -> genCode();
46         backPatchFalse(expr1->>falseList(), falseBB);
47         builder->setInsertBB(falseBB);
48         expr2->genCode();
49         false_list=expr2->>falseList();
50         true_list=merge(expr1->>trueList(), expr2->>trueList());
51         dst -> getType() -> kind = 4;
52     }
53     else if(op >= LESS && op <= MORE)
54     {
55         // Todo
56         expr1->genCode();

```

```

53     expr2->genCode();
54     Operand *src1 = expr1->getOperand();
55     Operand *src2 = expr2->getOperand();
56     int opcode;
57     switch (op)
58     {
59     case MORE:
60         opcode = CmpInstruction::G;
61         break;
62     case MOREEQUAL:
63         opcode = CmpInstruction::GE;
64         break;
65     case LESS:
66         opcode = CmpInstruction::L;
67         break;
68     case LESSEQUAL:
69         opcode = CmpInstruction::LE;
70         break;
71     case EQUAL:
72         opcode = CmpInstruction::E;
73         break;
74     case NOEQUAL:
75         opcode = CmpInstruction::NE;
76         break;
77     default:
78         break;
79     }
80
81     new CmpInstruction(opcode, dst, src1, src2, bb);
82
83     //自行添加的正确错误列表合并
84     true_list = merge(expr1->trueList(), expr2->trueList());
85     false_list = merge(expr1->falseList(), expr2->falseList());
86     Instruction* temp = new CondBrInstruction(nullptr, nullptr, dst, bb);
87     this->trueList().push_back(temp);
88     this->falseList().push_back(temp);
89     dst -> getType() -> kind = 4;
90
91 }
92 else if (op >= ADD && op <= SUB)
93 {
94     expr1->genCode();
95     expr2->genCode();
96     Operand *src1 = expr1->getOperand();
97     Operand *src2 = expr2->getOperand();
98     int opcode;
99     switch (op)
100    {

```

```

101     case ADD:
102         opcode = BinaryInstruction::ADD;
103         break;
104     case SUB:
105         opcode = BinaryInstruction::SUB;
106         break;
107     case MUL:
108         opcode = BinaryInstruction::MUL;
109         break;
110     case DIV:
111         opcode = BinaryInstruction::DIV;
112         break;
113     case PERC:
114         opcode = BinaryInstruction::MOD;
115         break;
116     }
117     new BinaryInstruction(opcode, dst, src1, src2, bb);
118 }
119 }

```

在 if 生成代码时，增加了前驱后继付给关系的语句，ifelse 仿照 if，在此函数下先声明三个基本块，并为其增加前驱后继关系，将 then 填充到正确列表，else 填充到错误列表，然后生成 then 和 else 的代码，最后设置 end 为插入点为后续进行插入，seqnode 就是让其两个状态先后生成代码即可，由于 ifelse 和 if 代码类似且篇幅较长，在此只展示 ifelse 代码：

ifelse 代码展示

```

1 void IfElseStmt::genCode()
2 {
3     // Todo完成
4     Function *func;
5     BasicBlock *then_bb, *else_bb, *end_bb;
6
7     func = builder->getInsertBB()->getParent();
8     then_bb = new BasicBlock(func);
9     end_bb = new BasicBlock(func);
10    else_bb = new BasicBlock(func);
11
12
13    //设置各种前驱后继
14    then_bb->addPred(builder->getInsertBB());
15    builder->getInsertBB()->addSucc(then_bb);
16
17    else_bb->addPred(builder->getInsertBB());
18    builder->getInsertBB()->addSucc(else_bb);
19
20    end_bb->addPred(then_bb);
21    then_bb->addSucc(end_bb);
22    end_bb->addPred(else_bb);

```



```

23     else_bb -> addSucc(end_bb);
24
25     //代码回填
26     cond -> genCode();
27     if(!cond -> getOperand() -> getType() -> isBool())
28     {
29         BasicBlock* bb=cond->builder->getInsertBB();
30         Operand *src = cond->getOperand();
31         SymbolEntry *se = new ConstantSymbolEntry(TypeSystem::intType, 0);
32         Constant* digit = new Constant(se);
33         Operand* t = new Operand(new TemporarySymbolEntry(TypeSystem::
34             boolType, SymbolTable::getLabel()));
35         CmpInstruction* temp = new CmpInstruction(CmpInstruction::EXCLAMATION
36             , t, src, digit->getOperand(), bb);
37         src=t;
38         cond->trueList().push_back(temp);
39         cond->>falseList().push_back(temp);
40         Instruction* m = new CondBrInstruction(nullptr, nullptr, t, bb);
41         cond->trueList().push_back(m);
42         cond->>falseList().push_back(m);
43     }
44     backPatch(cond -> trueList(), then_bb);
45     backPatchFalse(cond -> falseList(), else_bb);
46
47     //下一个
48     builder -> setInsertBB(then_bb);
49     thenStmt -> genCode();
50     then_bb = builder -> getInsertBB();
51     // builder->setInsertBB(then_bb);
52     new UncondBrInstruction(end_bb, then_bb);
53
54     builder -> setInsertBB(else_bb);
55     elseStmt->genCode();
56     else_bb = builder->getInsertBB();
57     // builder->setInsertBB(else_bb);
58     new UncondBrInstruction(end_bb, else_bb);
59
60     builder->setInsertBB(end_bb);
61 }

```

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：

标识符定义

```

1 void DeclStmt::genCode()
2 {
3     //std::cout << "start8" << std::endl;

```

```

4   for(auto iter = ids->Ids.rbegin(); iter != ids->Ids.rend(); iter++)
5   {
6       IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry *>((*
7           iter)-> getSymPtr());
8       if(se->isGlobal())
9       {
10          Operand *addr;
11          SymbolEntry *addr_se;
12          addr_se = new IdentifierSymbolEntry(*se);
13          addr_se->setType(new PointerType(se->getType()));
14          addr = new Operand(addr_se);
15          se->setAddr(addr);
16          bool temp = false;
17          Operand *src;
18          for(long unsigned int i = 0; i < ids -> Assigns.size(); i++)
19          {
20              if(ids -> Assigns[i] -> lval -> symbolEntry == se)
21              {
22                  ids -> Assigns[i] -> genCode();
23                  src = ids -> Assigns[i] -> expr -> getOperand();
24                  temp = true;
25                  break;
26              }
27          }
28          if(temp == false)
29          {
30              SymbolEntry *se = new ConstantSymbolEntry(TypeSystem::intType
31                  , 0);
32              Constant* digit = new Constant(se);
33              src = digit -> getOperand();
34          }
35          Instruction *alloca = new AllocaInstruction2(src, addr, se);
36          alloca -> output();
37      }
38      else if(se->isLocal())
39      {
40          Function *func = builder->getInsertBB()->getParent();
41          BasicBlock *entry = func->getEntry();
42          Instruction *alloca;
43          Operand *addr;
44          SymbolEntry *addr_se;
45          Type *type;
46          type = new PointerType(se->getType());
47          addr_se = new TemporarySymbolEntry(type, SymbolTable::getLabel())
            ;
48          addr = new Operand(addr_se);
49          alloca = new AllocaInstruction(addr, se);
50          //
51          allocate space for local id in function stack.

```

```

48         entry->insertFront(alloca); //
           allocate instructions should be inserted into the begin of
           the entry block.
49         se->setAddr(addr); //
           set the addr operand in symbol entry so that we can use it in
           subsequent code generation.
50     }
51 }
52 for(long unsigned int i = 0; i < ids -> Assigns.size(); i++)
53 {
54     IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry*>(ids
           -> Assigns[i] -> lval -> getSymPtr());
55     if(se -> isGlobal())
56     {
57         continue;
58     }
59     else if(se -> isLocal())
60     {
61         Operand *addr = dynamic_cast<IdentifierSymbolEntry*>(ids ->
           Assigns[i] -> lval -> getSymPtr())->getAddr();
62         se->setAddr(addr);
63         ids -> Assigns[i] -> genCode();
64     }
65 }
66 //std::cout << "end8" << std::endl;
67 }

```

对于 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：

while 循环

```

1 void WhileStmt::genCode()
2 {
3     Function *func;
4     BasicBlock *loop_bb, *end_bb, *cond_bb;
5
6
7     func = builder -> getInsertBB() -> getParent();
8     loop_bb = new BasicBlock(func);
9     end_bb = new BasicBlock(func);
10    cond_bb = new BasicBlock(func);
11
12    UncondBrInstruction *temp = new UncondBrInstruction(cond_bb, builder ->
           getInsertBB());
13    temp -> output();
14    //设置前后

```

```

15     cond_bb -> addPred(builder -> getInsertBB());
16     builder -> getInsertBB() -> addSucc(cond_bb);
17     loop_bb -> addPred(cond_bb);
18     cond_bb -> addSucc(loop_bb);
19
20     //builder -> getInsertBB() -> addSucc(loop_bb);
21     end_bb -> addPred(loop_bb);
22     loop_bb -> addSucc(end_bb);
23
24     end_bb -> addPred(cond_bb);
25     cond_bb -> addSucc(end_bb);
26
27     builder->setInsertBB(cond_bb);
28
29     cond -> genCode();
30     if(!cond -> getOperand() -> getType() -> isBool())
31     {
32         BasicBlock* bb=cond->builder->getInsertBB();
33         Operand *src = cond->getOperand();
34         SymbolEntry *se = new ConstantSymbolEntry(TypeSystem::intType, 0);
35         Constant* digit = new Constant(se);
36         Operand* t = new Operand(new TemporarySymbolEntry(TypeSystem::
            boolType, SymbolTable::getLabel()));
37         CmpInstruction* temp = new CmpInstruction(CmpInstruction::EXCLAMATION
            , t, src, digit->getOperand(), bb);
38         src=t;
39         cond->trueList().push_back(temp);
40         cond->>falseList().push_back(temp);
41         Instruction* m = new CondBrInstruction(nullptr, nullptr, t, bb);
42         cond->trueList().push_back(m);
43         cond->>falseList().push_back(m);
44     }
45     backPatch(cond -> trueList(), loop_bb);
46     backPatchFalse(cond -> falseList(), end_bb);
47
48     builder -> setInsertBB(loop_bb);
49     loop -> genCode();
50     loop_bb = builder -> getInsertBB();
51     new CondBrInstruction(cond_bb, end_bb, cond->getOperand(), loop_bb);
52
53     builder->setInsertBB(end_bb);
54 }

```

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：

二元表达式类型检查

```

1 void BinaryExpr::typeCheck()
2 {
3     // Todo
4     Type *type1 = expr1 -> getSymPtr() -> getType();
5     Type *type2 = expr2 -> getSymPtr() -> getType();
6     if(type1 != type2){
7         fprintf(stderr, "type %s and %s mismatch",
8             type1 -> toString().c_str(), type2 -> toString().c_str());
9         exit(EXIT_FAILURE);
10    }
11    fprintf(yyout, ";BinaryExpr TypeCheck OK!\n");
12    symbolEntry -> setType(type1);
13    expr1 -> typeCheck();
14    expr2 -> typeCheck();
15 }

```

3. 结果展示

在生成中间代码部分，本次样例代码非常多，选其中一个进行展示，样例代码如下：

中间代码样例代码

```

1 int fun(int m, int n){
2     int rem;
3     while(n > 0){
4         rem = m % n;
5         m = n;
6         n = rem;
7     }
8     return m;
9 }
10 int main(){
11     int n,m;
12     int num;
13     m=getint();
14     n=getint();
15     num=fun(m,n);
16     putint(num);
17
18     return 0;
19 }

```

中间代码的测试结果如下：

中间代码结果

```

1 ;TypeCheck Begin!
2 declare i32 @getint()
3 declare void @putint(i32)
4 declare i32 @getch()
5 declare void @putch(i32)
6 define i32 @fun(i32 %t24, i32 %t27){
7 B23:
8   %t30 = alloca i32, align 4
9   %t26 = alloca i32, align 4
10  store i32 %t24, i32* %t26, align 4
11  %t29 = alloca i32, align 4
12  store i32 %t27, i32* %t29, align 4
13  br label %B33
14 B33:                                ; preds = %B23
15  %t3 = load i32, i32* %t29, align 4
16  %t4 = icmp sgt i32 %t3, 0
17  br i1 %t4, label %B31, label %B32
18 B31:                                ; preds = %B33, %B33
19  %t6 = load i32, i32* %t26, align 4
20  %t7 = load i32, i32* %t29, align 4
21  %t8 = srem i32 %t6, %t7
22  store i32 %t8, i32* %t30, align 4
23  %t10 = load i32, i32* %t29, align 4
24  store i32 %t10, i32* %t26, align 4
25  %t12 = load i32, i32* %t30, align 4
26  store i32 %t12, i32* %t29, align 4
27  br i1 %t4, label %B33, label %B32
28 B32:                                ; preds = %B31, %B33, %B33
29  %t13 = load i32, i32* %t26, align 4
30  ret i32 %t13
31 }
32 define i32 @main(){
33 B34:
34  %t37 = alloca i32, align 4
35  %t36 = alloca i32, align 4
36  %t35 = alloca i32, align 4
37  %t39 = call i32 @getint()
38  store i32 %t39, i32* %t35, align 4
39  %t41 = call i32 @getint()
40  store i32 %t41, i32* %t36, align 4
41  %t20 = load i32, i32* %t35, align 4
42  %t21 = load i32, i32* %t36, align 4
43  %t43 = call i32 @fun(i32 %t20, i32 %t21)
44  store i32 %t43, i32* %t37, align 4
45  %t22 = load i32, i32* %t37, align 4
46  call void @putint(i32 %t22)
47  ret i32 0
48 }

```

针对类型检查展示一个错误样例，为调用参数的错误，样例如下：

类型检查样例代码

```
1 int func(int a, int b, int c)
2 {
3     return a + b + c;
4 }
5 int main()
6 {
7     return func(1);
8 }
```

针对此样例的检查结果如下：

类型检查结果

```
1 test/lab6/03_func_undef.ll:23:19: error: '@func' defined with type 'i32 (i32,
   i32, i32)*' but expected 'i32 (i32)*'
2 %t20 = call i32 @func(i32 1)
3           ^
```

五、 目标代码生成

(一) 模块设计

1. 功能简述

在本次实验中，我们完成了通过语义检查生成的中间代码，来生成相应的汇编代码，以对中间代码进行自顶向下的遍历，从而生成目标代码。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。

对于目标代码生成来看，其整体的框架是由 MachineUnit→ MachineFunction→ MachineBasicBlock→ MachineInstruction→ MachineOperand→ MachineType 来构成，可以看出与中间代码十分类似。

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。

2. 设计目标

在本次实验的目标为，根据现有的框架，在其基础上实现对中间代码到目标代码的转化，并对转换完成的目标代码进行寄存器的分配 [1]，主要需要对各种指令进行翻译以及对活跃区间进行分析，并对溢出进行处理。

但这些只是基础目标，我们小组的进阶目标是完成所有的进阶要求，数组、浮点数、非叶函数等，并完成部分的代码优化（即高级要求），比如 Mem2Reg、图着色寄存器分配等，高质量的完成我们最后的编译器。

(二) 个人贡献

1. 负责任务

由于本次实验难度系数大大增加，像之前循环、变量、函数这样拆解的分配任务的话很浪费时间，并且效率极低，所以我们小组更换分工思路，按一个一个大 part 来进行分工，最后我主要是基本要求的目标代码生成部分和所有的进阶要求，我的队友是寄存器分配部分以及代码优化，我的具体任务如下：

1. 实现基本的 IR 指令到汇编指令的翻译，完善 genMachineCode() 函数（基本要求）

- 数据访存指令的翻译，主要只需要完成 StoreInstruction
- 二元运算指令的翻译， BinaryInstrction
- 比较指令的翻译
- 控制流指令的翻译， UncondBrInstr 语句、 CondBrInstr 语句、 RetInstru 语句等
- 函数定义及函数调用的翻译

2. 进阶要求

- 实现数组的翻译
- 实现浮点类型的翻译
- 实现 break、continue 语句的翻译
- 实现非叶函数的翻译

2. 任务实现

基本要求

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）

对于存储指令，判断是整型还是浮点类型，生成对应的操作，如果是存储到一个标签地址中，则需要先将此地址加载到寄存器中，为后边的存储做准备；如果是局部变量，先获得其偏移，以基址寄存器 FP 加上偏移，如果有溢出需要重新加载其地址，并根据对应的类型进行存储；对于全局变量，加载标签对应的地址，根据此地址进行存储；对于指针变量来说即数组元素，直接进行存储即可，具体代码如下所示：

生成目标代码

```

1 void StoreInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     MachineInstruction *cur_inst = nullptr;
5
6     // bool dst_float = operands[0]->getType()->isFloat();
7     bool src_float = operands[1]->getType()->isFloat();
8
9     MachineOperand *dst = nullptr;
10    MachineOperand *src = nullptr;
11
12    dst = genMachineOperand(operands[0]);
13
14    if (src_float)
15    {
16        src = genMachineFloatOperand(operands[1]);
17    }
18    else
19    {
20        src = genMachineOperand(operands[1]);
21    }
22
23    // store immediate
24    if (operands[1]->getEntry()->isConstant())
25    {
26        auto dst1 = genMachineVReg(src_float);
27        if (src_float)
28        {
29            auto internal_reg = genMachineVReg();
30            cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
31                                           internal_reg, src);
32            cur_block->InsertInst(cur_inst);
33            internal_reg = new MachineOperand(*internal_reg);

```

```

34         cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
35                                         dst1, internal_reg);
36     }
37     else
38     {
39         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
40                                         dst1, src);
41     }
42     cur_block->InsertInst(cur_inst);
43     // src = dst1;
44     src = new MachineOperand(*dst1);
45 }
46 // store to local
47 if (operands[0]->getEntry()->isTemporary() && operands[0]->getDef() &&
48     operands[0]->getDef()->isAlloc())
49 {
50     auto src1 = genMachineReg(11);
51     int off = dynamic_cast<TemporarySymbolEntry*>(operands[0]->getEntry
52         ())->getOffset();
53     auto src2 = genMachineImm(off);
54     if (off > 255 || off < -255)
55     {
56         auto operand = genMachineVReg();
57         cur_block->InsertInst((new LoadMInstruction(
58             cur_block, LoadMInstruction::LDR, operand, src2)));
59         src2 = operand;
60     }
61     if (src_float)
62     {
63         if (off > 255 || off < -255)
64         {
65             auto reg = genMachineVReg();
66             cur_inst = new BinaryMInstruction(
67                 cur_block, BinaryMInstruction::ADD, reg, src1, src2);
68             cur_block->InsertInst(cur_inst);
69             cur_inst =
70                 new StoreMInstruction(cur_block, StoreMInstruction::VSTR,
71                                     src, new MachineOperand(*reg));
72         }
73         else
74         {
75             cur_inst = new StoreMInstruction(
76                 cur_block, StoreMInstruction::VSTR, src, src1, src2);
77         }
78     }
79     else
80     {

```

```

79         cur_inst = new StoreMInstruction(cur_block, StoreMInstruction::
80             STR,
81             src, src1, src2);
82     }
83     cur_block->InsertInst(cur_inst);
84 }
85 // store to global
86 else if (operands[0]->getEntry()->isVariable() && dynamic_cast<
87     IdentifierSymbolEntry *>(operands[0]->getEntry()->isGlobal()))
88 {
89     auto internal_reg1 = genMachineVReg();
90     // example: load r0, addr_a
91     if (src_float)
92     {
93         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
94             internal_reg1, dst);
95         cur_block->InsertInst(cur_inst);
96         cur_inst = new StoreMInstruction(cur_block, StoreMInstruction::
97             VSTR,
98             src, internal_reg1);
99         cur_block->InsertInst(cur_inst);
100     }
101     else
102     {
103         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
104             internal_reg1, dst);
105         cur_block->InsertInst(cur_inst);
106         // example: store r1, [r0]
107         cur_inst = new StoreMInstruction(cur_block, StoreMInstruction::
108             STR,
109             src, internal_reg1);
110         cur_block->InsertInst(cur_inst);
111     }
112 }
113 // store to pointer
114 else if (operands[0]->getType()->isPtr())
115 {
116     if (src_float)
117     {
118         cur_inst = new StoreMInstruction(cur_block, StoreMInstruction::
119             VSTR,
120             src, dst);
121         cur_block->InsertInst(cur_inst);
122     }
123     else
124     {
125         cur_inst = new StoreMInstruction(cur_block, StoreMInstruction::
126             STR,

```

```

121         src, dst);
122     cur_block->InsertInst(cur_inst);
123 }
124 }
125 }

```

在存储指令机器码的部分也是仿照加载指令，构造时直接将传参赋值，输出时不论是 str 指令还是 vstr 指令都是先输出存储的寄存器，然后输出其基地址，如果有偏移将偏移输出。

机器码翻译

```

1 void StoreMInstruction::output()
2 {
3     if (op == StoreMInstruction::STR)
4     {
5         fprintf(yyout, "\tstr ");
6         this->use_list[0]->output();
7         fprintf(yyout, ", ");
8         // store address
9         if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
10            fprintf(yyout, "[");
11        this->use_list[1]->output();
12        if (this->use_list.size() > 2)
13        {
14            fprintf(yyout, ", ");
15            this->use_list[2]->output();
16        }
17        if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
18            fprintf(yyout, "]");
19        fprintf(yyout, "\n");
20    }
21    else if (op == StoreMInstruction::VSTR)
22    {
23        // TODO
24        fprintf(yyout, "\tvstr.32 ");
25        this->use_list[0]->output();
26        fprintf(yyout, ", ");
27        // store address
28        if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
29            fprintf(yyout, "[");
30        this->use_list[1]->output();
31        if (this->use_list.size() > 2)
32        {
33            fprintf(yyout, ", ");
34            this->use_list[2]->output();
35        }
36        if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
37            fprintf(yyout, "]");
38        fprintf(yyout, "\n");
39    }

```

40 }

二元运算指令 对于二元指令生成机器码，由于浮点类型和整型部分相似，生成三个部分的操作数，中间代码允许两个操作数都可以是立即数，但汇编代码中不被允许，本次代码中不去判断有几个立即数，出现立即数，就将其加载到寄存器中，赋予一个临时寄存器，接下来根据相应的类型生成对应的指令即可

生成目标代码

```

1 void BinaryInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     if (operands[0]->getType()->isFloat())
4     {
5         auto flag = false;
6         auto dst = genMachineFloatOperand(operands[0]);
7         auto src1 = genMachineFloatOperand(operands[1]);
8         auto src2 = genMachineFloatOperand(operands[2]);
9         MachineInstruction *cur_inst = nullptr;
10        if (src1->isImm())
11        {
12            auto tmp_reg = genMachineVReg(true);
13            auto internal_reg = genMachineVReg();
14            cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
15                internal_reg, src1);
16            cur_block->InsertInst(cur_inst);
17            internal_reg = new MachineOperand(*internal_reg);
18            cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
19                tmp_reg, internal_reg);
20            cur_block->InsertInst(cur_inst);
21            src1 = new MachineOperand(*tmp_reg);
22        }
23        if (src2->isImm())
24        {
25            if (src2->getFVal() == 0 && opcode == ADD)
26                flag = true;
27            else
28            {
29                auto tmp_reg = genMachineVReg(true);
30                auto internal_reg = genMachineVReg();
31                cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::
32                    LDR, internal_reg, src2);
33                cur_block->InsertInst(cur_inst);
34                internal_reg = new MachineOperand(*internal_reg);
35                cur_inst = new MovMInstruction(cur_block, MovMInstruction::
36                    VMOV,
37                        tmp_reg, internal_reg);
38                cur_block->InsertInst(cur_inst);
39                src2 = new MachineOperand(*tmp_reg);

```

```

37     }
38 }
39
40 switch (opcode)
41 {
42     case ADD:
43         if (flag)
44             cur_inst = new MovMInstruction(
45                 cur_block, MovMInstruction::VMOVF32, dst, src1);
46         else
47             cur_inst = new BinaryMInstruction(
48                 cur_block, BinaryMInstruction::VADD, dst, src1, src2);
49         break;
50     case SUB:
51         cur_inst = new BinaryMInstruction(
52             cur_block, BinaryMInstruction::VSUB, dst, src1, src2);
53         break;
54     case MUL:
55         cur_inst = new BinaryMInstruction(
56             cur_block, BinaryMInstruction::VMUL, dst, src1, src2);
57         break;
58     case DIV:
59         cur_inst = new BinaryMInstruction(
60             cur_block, BinaryMInstruction::VDIV, dst, src1, src2);
61         break;
62     case MOD:
63         // error
64         break;
65     default:
66         break;
67 }
68 cur_block->InsertInst(cur_inst);
69 }
70 else
71 {
72     auto dst = genMachineOperand(operands[0]);
73     auto src1 = genMachineOperand(operands[1]);
74     auto src2 = genMachineOperand(operands[2]);
75     MachineInstruction *cur_inst = nullptr;
76     if (src1->isImm() && src2->isImm() && src2->getVal() == 0 &&
77         opcode == ADD)
78     {
79         if (!(src1->getVal() < 256 && src1->getVal() > -255))
80         {
81             auto internal_reg = genMachineVReg();
82             cur_inst = new LoadMInstruction(
83                 cur_block, LoadMInstruction::LDR, internal_reg, src1);
84             cur_block->InsertInst(cur_inst);

```

```

85         src1 = new MachineOperand(*internal_reg);
86     }
87     cur_inst =
88         new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
89                             src1);
89     cur_block->InsertInst(cur_inst);
90     return;
91 }
92 if (src1->isImm())
93 {
94     auto internal_reg = genMachineVReg();
95     cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
96                                     internal_reg, src1);
97     cur_block->InsertInst(cur_inst);
98     src1 = new MachineOperand(*internal_reg);
99 }
100 if (src2->isImm())
101 {
102     if ((opcode <= BinaryInstruction::OR &&
103         ((ConstantSymbolEntry *) (operands[2]->getEntry()))->getValue
104         () > 255) ||
105         opcode >= BinaryInstruction::MUL)
106     {
107         auto internal_reg = genMachineVReg();
108         cur_inst = new LoadMInstruction(
109             cur_block, LoadMInstruction::LDR, internal_reg, src2);
110         cur_block->InsertInst(cur_inst);
111         src2 = new MachineOperand(*internal_reg);
112     }
113 }
114 switch (opcode)
115 {
116 case ADD:
117     cur_inst = new BinaryMInstruction(
118         cur_block, BinaryMInstruction::ADD, dst, src1, src2);
119     break;
120 case SUB:
121     cur_inst = new BinaryMInstruction(
122         cur_block, BinaryMInstruction::SUB, dst, src1, src2);
123     break;
124 case AND:
125     cur_inst = new BinaryMInstruction(
126         cur_block, BinaryMInstruction::AND, dst, src1, src2);
127     break;
128 case OR:
129     cur_inst = new BinaryMInstruction(
130         cur_block, BinaryMInstruction::OR, dst, src1, src2);
131     break;

```

```

131     case MUL:
132         cur_inst = new BinaryMInstruction(
133             cur_block, BinaryMInstruction::MUL, dst, src1, src2);
134         break;
135     case DIV:
136         cur_inst = new BinaryMInstruction(
137             cur_block, BinaryMInstruction::DIV, dst, src1, src2);
138         break;
139     case MOD:
140     {
141         auto dst1 = genMachineVReg();
142         cur_inst = new BinaryMInstruction(
143             cur_block, BinaryMInstruction::DIV, dst1, src1, src2);
144         src1 = new MachineOperand(*src1);
145         src2 = new MachineOperand(*src2);
146         auto temp = new MachineOperand(*dst1);
147         cur_block->InsertInst(cur_inst);
148         // c2 = c1 * b
149         auto dst2 = genMachineVReg();
150         cur_inst = new BinaryMInstruction(
151             cur_block, BinaryMInstruction::MUL, dst2, temp, src2);
152         cur_block->InsertInst(cur_inst);
153         dst2 = new MachineOperand(*dst2);
154         // c = a - c2
155         cur_inst = new BinaryMInstruction(
156             cur_block, BinaryMInstruction::SUB, dst, src1, dst2);
157         break;
158     }
159     default:
160         break;
161 }
162 cur_block->InsertInst(cur_inst);
163 }
164 }

```

二元运算指令的机器码，将传参赋值并加入使用列表和设置父节点，输出函数将其对应的操作输出，比如加减乘除等，输出存储到的寄存器以及操作的两个数

机器码翻译

```

1 void BinaryMInstruction::output()
2 {
3     switch (this->op)
4     {
5     case BinaryMInstruction::ADD:
6         fprintf(yyout, "\tadd ");
7         break;
8     case BinaryMInstruction::SUB:
9         fprintf(yyout, "\tsub ");
10        break;

```



```

11     case BinaryMInstruction::AND:
12         fprintf(yyout, "\\tand ");
13         break;
14     case BinaryMInstruction::OR:
15         fprintf(yyout, "\\torr ");
16         break;
17     case BinaryMInstruction::MUL:
18         fprintf(yyout, "\\tmul ");
19         break;
20     case BinaryMInstruction::DIV:
21         fprintf(yyout, "\\tsdiv ");
22         break;
23     case BinaryMInstruction::VADD:
24         fprintf(yyout, "\\tvadd.f32 ");
25         break;
26     case BinaryMInstruction::VSUB:
27         fprintf(yyout, "\\tvsb.f32 ");
28         break;
29     case BinaryMInstruction::VMUL:
30         fprintf(yyout, "\\tvmul.f32 ");
31         break;
32     case BinaryMInstruction::VDIV:
33         fprintf(yyout, "\\tvddiv.f32 ");
34         break;
35     default:
36         break;
37 }
38
39 this->PrintCond();
40 this->def_list[0]->output();
41 fprintf(yyout, ", ");
42 this->use_list[0]->output();
43 fprintf(yyout, ", ");
44 this->use_list[1]->output();
45 fprintf(yyout, "\\n");
46 }

```

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

生成目标代码

```

1 void CmpInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4

```

```

5   if (operands[1]->getType()->isFloat())
6   {
7       auto src1 = genMachineFloatOperand(operands[1]);
8       auto src2 = genMachineFloatOperand(operands[2]);
9       MachineInstruction *cur_inst = nullptr;
10      if (src1->isImm())
11      {
12          auto tmp_reg = genMachineVReg(true);
13          auto internal_reg = genMachineVReg();
14          cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
15                                          internal_reg, src1);
16          cur_block->InsertInst(cur_inst);
17          internal_reg = new MachineOperand(*internal_reg);
18          cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
19                                          tmp_reg, internal_reg);
20          cur_block->InsertInst(cur_inst);
21          src1 = new MachineOperand(*tmp_reg);
22      }
23      if (src2->isImm())
24      {
25          auto tmp_reg = genMachineVReg(true);
26          auto internal_reg = genMachineVReg();
27          cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
28                                          internal_reg, src2);
29          cur_block->InsertInst(cur_inst);
30          internal_reg = new MachineOperand(*internal_reg);
31          cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
32                                          tmp_reg, internal_reg);
33          cur_block->InsertInst(cur_inst);
34          src2 = new MachineOperand(*tmp_reg);
35      }
36      cur_inst = new CmpMInstruction(cur_block, CmpMInstruction::VCMP, src1
37                                   ,
38                                   src2, opcode);
39      cur_block->InsertInst(cur_inst);
40      cur_inst = new VmrsMInstruction(cur_block);
41      cur_block->InsertInst(cur_inst);
42
43      if (opcode >= CmpInstruction::L && opcode <= CmpInstruction::GE)
44      {
45          auto dst = genMachineOperand(operands[0]);
46          auto trueOperand = genMachineImm(1);
47          auto falseOperand = genMachineImm(0);
48          cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
49                                          dst,
50                                          trueOperand, opcode);
51          cur_block->InsertInst(cur_inst);
52          cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,

```

```

        dst,
        falseOperand, 7 - opcode);
51
    cur_block->InsertInst(cur_inst);
52
53 }
54 else if (opcode == CmpInstruction::E)
55 {
56     auto dst = genMachineOperand(operands[0]);
57     auto trueOperand = genMachineImm(1);
58     auto falseOperand = genMachineImm(0);
59     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
        dst,
60                                     trueOperand, E);
61     cur_block->InsertInst(cur_inst);
62     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
        dst,
63                                     falseOperand, NE);
64     cur_block->InsertInst(cur_inst);
65 }
66 else if (opcode == CmpInstruction::NE)
67 {
68     auto dst = genMachineOperand(operands[0]);
69     auto trueOperand = genMachineImm(1);
70     auto falseOperand = genMachineImm(0);
71     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
        dst,
72                                     trueOperand, NE);
73     cur_block->InsertInst(cur_inst);
74     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
        dst,
75                                     falseOperand, E);
76     cur_block->InsertInst(cur_inst);
77 }
78 }
79 else
80 {
81     auto src1 = genMachineOperand(operands[1]);
82     auto src2 = genMachineOperand(operands[2]);
83     MachineInstruction *cur_inst = nullptr;
84     if (src1->isImm())
85     {
86         auto internal_reg = genMachineVReg();
87         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
88                                         internal_reg, src1);
89         cur_block->InsertInst(cur_inst);
90         src1 = new MachineOperand(*internal_reg);
91     }
92     if (src2->isImm() &&
93         ((ConstantSymbolEntry *) (operands[2]->getEntry()))->getValue() >

```

```

94         255)
95     {
96         auto internal_reg = genMachineVReg();
97         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
98                                         internal_reg, src2);
99         cur_block->InsertInst(cur_inst);
100        src2 = new MachineOperand(*internal_reg);
101    }
102    cur_inst = new CmpMInstruction(cur_block, CmpMInstruction::CMP, src1,
103                                    src2, opcode);
104    cur_block->InsertInst(cur_inst);
105    if (opcode >= CmpInstruction::L && opcode <= CmpInstruction::GE)
106    {
107        auto dst = genMachineOperand(operands[0]);
108        auto trueOperand = genMachineImm(1);
109        auto falseOperand = genMachineImm(0);
110        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
111                                        dst,
112                                        trueOperand, opcode);
113        cur_block->InsertInst(cur_inst);
114        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
115                                        dst,
116                                        falseOperand, 7 - opcode);
117        cur_block->InsertInst(cur_inst);
118    }
119    else if (opcode == CmpInstruction::E)
120    {
121        auto dst = genMachineOperand(operands[0]);
122        auto trueOperand = genMachineImm(1);
123        auto falseOperand = genMachineImm(0);
124        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
125                                        dst,
126                                        trueOperand, E);
127        cur_block->InsertInst(cur_inst);
128        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
129                                        dst,
130                                        falseOperand, NE);
131        cur_block->InsertInst(cur_inst);
132    }
133    else if (opcode == CmpInstruction::NE)
134    {
135        auto dst = genMachineOperand(operands[0]);
136        auto trueOperand = genMachineImm(1);
137        auto falseOperand = genMachineImm(0);
138        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
139                                        dst,
140                                        trueOperand, NE);
141        cur_block->InsertInst(cur_inst);

```

```

137         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
138                                     dst,
139                                     falseOperand, E);
140     cur_block->InsertInst(cur_inst);
141 }
142 }

```

对于比较指令的翻译，初始化的操作直接传参，然后判断比较的类型，并将比较的两个数输出即可实现，具体代码如下：

机器码翻译

```

1 void CmpMInstruction::output()
2 {
3     switch (this->op)
4     {
5         case CmpMInstruction::CMP:
6             fprintf(yyout, "\tcmp ");
7             break;
8         case CmpMInstruction::VCMP:
9             fprintf(yyout, "\tvcmp.f32 ");
10            break;
11        default:
12            break;
13    }
14    this->use_list[0]->output();
15    fprintf(yyout, ", ");
16    this->use_list[1]->output();
17    fprintf(yyout, "\n");
18 }

```

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 genMachineCode 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。

由于无条件跳转和有条件跳转较为简单，就不展示代码，如下为返回值的目标代码生成：

生成目标代码

```

1 void RetInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     if (!operands.empty())
5     {
6         MachineOperand *dst;
7         MachineOperand *src;

```

```

8      MachineInstruction *cur_inst;
9
10     if (operands[0]->getType()->isFloat())
11     {
12         dst = new MachineOperand(MachineOperand::REG, 16, true);
13         src = genMachineFloatOperand(operands[0]);
14         if (src->isImm())
15         {
16             auto internal_reg = genMachineVReg();
17             cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::
                LDR, internal_reg, src);
18             cur_block->InsertInst(cur_inst);
19             src = internal_reg;
20         }
21         cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
                dst, src);
22     }
23     else
24     {
25         dst = new MachineOperand(MachineOperand::REG, 0);
26         src = genMachineOperand(operands[0]);
27         if (operands[0]->isConst())
28         {
29             auto val = operands[0]->getConstVal();
30             if (val > 255 || val <= -255)
31             {
32                 auto r0 = new MachineOperand(MachineOperand::REG, 0);
33                 cur_block->InsertInst(new LoadMInstruction(cur_block,
                    LoadMInstruction::LDR, r0, src));
34                 src = r0;
35             }
36         }
37         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
                dst, src);
38     }
39     cur_block->InsertInst(cur_inst);
40 }
41 auto cur_func = builder->getFunction();
42 auto sp = new MachineOperand(MachineOperand::REG, 13);
43 auto size = new MachineOperand(MachineOperand::IMM, cur_func->AllocSpace
    (0));
44 auto cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD
    , sp, sp, size);
45 cur_block->InsertInst(cur_inst);
46
47 auto lr = new MachineOperand(MachineOperand::REG, 14);
48 auto cur_inst2 = new BranchMInstruction(cur_block, BranchMInstruction::BX
    , lr);

```

```

49     cur_block->InsertInst(cur_inst2);
50 }

```

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。

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：

生成目标代码

```

1 void CallInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     MachineOperand *operand; //, *num;
5     MachineInstruction *cur_inst;
6
7     size_t idx;
8     auto funcSE = (IdentifierSymbolEntry *)func;
9
10    int stk_cnt = 0;
11    std::vector<MachineOperand *> vec;
12
13    bool need_align = false; // for alignment
14    int float_num = 0;
15    int int_num = 0;
16    for (size_t i = 1; i < operands.size(); i++)
17    {
18        if (operands[i]->getType()->isFloat())
19        {
20            float_num++;
21        }
22        else
23        {
24            int_num++;
25        }
26    }
27
28    int push_num = 0;
29    if (float_num > 4)
30    {
31        push_num += float_num - 4;
32    }
33    if (int_num > 4)

```

```
34 {
35     push_num += int_num - 4;
36 }
37
38 if (push_num % 2 != 0)
39 {
40     need_align = true;
41 }
42
43 int gpreg_cnt = 1;
44 for (idx = 1; idx < operands.size(); idx++)
45 {
46     if (gpreg_cnt == 5)
47         break;
48     if (operands[idx]->getType()->isFloat())
49     {
50         continue;
51     }
52     operand = genMachineReg(gpreg_cnt - 1);
53     auto src = genMachineOperand(operands[idx]);
54     if (src->isImm() && src->getVal() > 255)
55     {
56         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
57                                         operand, src);
58     }
59     else
60     {
61         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
62                                       operand, src);
63     }
64     cur_block->InsertInst(cur_inst);
65     gpreg_cnt++;
66 }
67
68 size_t int_idx = idx;
69
70 int fpreg_cnt = 1;
71 for (idx = 1; idx < operands.size(); idx++)
72 {
73     if (fpreg_cnt == 5 && !need_align)
74         break;
75     if (fpreg_cnt == 6 && need_align)
76     {
77         break;
78     }
79     if (!operands[idx]->getType()->isFloat())
80     {
81         continue;
```



```

82     }
83     operand = genMachineFReg(fpreg_cnt - 1);
84     auto src = genMachineFloatOperand(operands[idx]);
85     if (src->isImm())
86     {
87         auto internal_reg = genMachineVReg();
88         cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
89                                         internal_reg, src);
90         cur_block->InsertInst(cur_inst);
91         internal_reg = new MachineOperand(*internal_reg);
92         cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
93                                       operand, internal_reg);
94     }
95     else
96     {
97         cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
98                                       operand, src);
99     }
100    cur_block->InsertInst(cur_inst);
101    fpreg_cnt++;
102 }
103
104 size_t float_idx = idx;
105
106 idx = std::min(float_idx, int_idx);
107
108 for (size_t i = operands.size() - 1; i >= idx; i--)
109 {
110     if (operands[i]->getType()->isFloat() && i >= float_idx)
111     {
112         operand = genMachineFloatOperand(operands[i]);
113         if (operand->isImm())
114         {
115             auto dst = genMachineVReg(true);
116             auto internal_reg = genMachineVReg();
117             cur_inst = new LoadMInstruction(
118                 cur_block, LoadMInstruction::LDR, internal_reg, operand);
119             cur_block->InsertInst(cur_inst);
120             internal_reg = new MachineOperand(*internal_reg);
121             cur_inst = new MovMInstruction(cur_block, MovMInstruction::
122                 VMOV,
123                                         dst, internal_reg);
124             cur_block->InsertInst(cur_inst);
125             operand = new MachineOperand(*dst);
126         }
127         cur_inst = new StackMInstruction(
128             cur_block, StackMInstruction::VPUSH, vec, operand);
129         cur_block->InsertInst(cur_inst);

```

```

129         stk_cnt++;
130     }
131     else if (!operands[i] -> getType() -> isFloat() && i >= int_idx)
132     {
133         operand = genMachineOperand(operands[i]);
134         if (operand -> isImm())
135         {
136             auto dst = genMachineVReg();
137             if (operand -> getVal() < 256)
138             {
139                 cur_inst = new MovMInstruction(
140                     cur_block, MovMInstruction::MOV, dst, operand);
141             }
142             else
143             {
144                 cur_inst = new LoadMInstruction(
145                     cur_block, LoadMInstruction::LDR, dst, operand);
146             }
147             cur_block -> InsertInst(cur_inst);
148             operand = new MachineOperand(*dst);
149         }
150         cur_inst = new StackMInstruction(cur_block, StackMInstruction::
            PUSH,
151             vec, operand);
152         cur_block -> InsertInst(cur_inst);
153         stk_cnt++;
154     }
155 }
156
157 auto label = new MachineOperand(func -> toStr().c_str());
158 cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL,
    label);
159 cur_block -> InsertInst(cur_inst);
160 if ((gpreg_cnt >= 5 || fpreg_cnt >= 5) && stk_cnt != 0)
161 {
162     auto off = genMachineImm(stk_cnt * 4);
163     auto sp = new MachineOperand(MachineOperand::REG, 13);
164     cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
165         sp, sp, off);
166     cur_block -> InsertInst(cur_inst);
167 }
168 if (dst)
169 {
170     if (dst -> getType() -> isFloat())
171     {
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 }
178 else
179 {
180     operand = genMachineOperand(dst);
181     auto r0 = new MachineOperand(MachineOperand::REG, 0);
182     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
183         operand, r0);
184     cur_block->InsertInst(cur_inst);
185 }
186 }
187 }

```

在其机器码部分，首先是栈操作，先根据操作输出相应类型，然后判断使用列表的大小，如果小于 16 则代表没有浮点操作，直接根据大小输出相应寄存器，否则需要将相应的浮点类型寄存器也输出。

函数部分的构造，根据参数个数来赋予相应的整型和浮点型的个数，如果大于四个则需要将其压入栈中，并且根据压入栈的个数判断是否需要对齐；输出部分，首先生成 push 指令保存 FP 寄存器以及一些 called saved 寄存器，并生成 mov 指令使得 FP 寄存器指向新栈底，接下来须生成 sub 指令为局部变量分配栈内空间，如果参数大于四个需要调用内存空间，然后对每个基本块进行输出。

对于基本块的输出，先输出其对应的块编号，然后对每条指令进行判断，如果是普通的指令，则直接将其输出即可，对于类似存储或者二元指令有加载常量或者是全局变量须将其加载到寄存器，再输出，对于 bx 指令则需要弹出相应保存的值。

进阶要求 对于进阶要求部分，由于许多操作已经在生成目标代码部分进行了展示，所以在这里就不再列出代码，只对其介绍。

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。

在数组指令生成机器码时，对于初始化，赋予其栈空间并返回，也需要对偏移进行大小的判断；如果为立即数须将其加载到寄存器中，如果是最开始的维度，则赋值大小，否则将其加载到寄存器，根据是否是全局变量来将其以不同的方式加载，并根据指针类型计算大小，根据此值重

新赋给寄存器；如果不是最终维度，则需要递归调用数组指令生成汇编代码；如果是最后一个维度，则根据其是否是全局变量来将其以不同的方式存储，在生成机器码部分的代码如下：

数组的目标代码生成

```

1 void GepInstruction::genMachineCode(AsmBuilder *builder)
2 {
3     auto cur_block = builder->getBlock();
4     MachineInstruction *cur_inst;
5     auto dst = genMachineOperand(operands[0]);
6     auto idx = genMachineOperand(operands[2]);
7     if (init)
8     {
9         if (last)
10        {
11            auto base = genMachineOperand(init);
12            MachineOperand *imm = genMachineImm(off + 4);
13            int off = this->off + 4;
14            if (off > 255)
15            {
16                MachineOperand *temp = genMachineVReg();
17                cur_block->InsertInst(new LoadMInstruction(
18                    cur_block, LoadMInstruction::LDR, temp, imm));
19                imm = temp;
20            }
21            cur_inst = new BinaryMInstruction(
22                cur_block, BinaryMInstruction::ADD, dst, base, imm);
23            cur_block->InsertInst(cur_inst);
24        }
25        else
26        {
27            noAsm = true;
28        }
29        return;
30    }
31    MachineOperand *base = nullptr;
32    int size;
33    auto idx1 = genMachineVReg();
34    if (idx->isImm())
35    {
36        if (idx->getVal() < 255)
37        {
38            cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
39                idx1, idx);
40        }
41        else
42        {
43            cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
44                idx1, idx);

```

```

43     }
44     idx = new MachineOperand(*idx1);
45     cur_block->InsertInst(cur_inst);
46 }
47 if (paramFirst)
48 {
49     size = ((PointerType *) (operands[1]->getType()))->getType()->getSize
        () / 8;
50 }
51 else
52 {
53     if (first)
54     {
55         base = genMachineVReg();
56         if (operands[1]->getEntry()->isVariable() && ((
            IdentifierSymbolEntry *) (operands[1]->getEntry()))->isGlobal
                ())
57         {
58             auto src = genMachineOperand(operands[1]);
59             cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::
                LDR, base, src);
60         }
61         else
62         {
63             int offset = ((TemporarySymbolEntry *) (operands[1]->getEntry
                ()))>getOffset();
64             if (offset > -255 && offset < 255)
65             {
66                 cur_inst = new MovMInstruction(cur_block, MovMInstruction
                    ::MOV, base, genMachineImm(offset));
67             }
68             else
69             {
70                 cur_inst = new LoadMInstruction(cur_block,
                    LoadMInstruction::LDR, base, genMachineImm(offset));
71             }
72         }
73         cur_block->InsertInst(cur_inst);
74     }
75
76     ArrayType *type = (ArrayType *) (((PointerType *) (operands[1]->getType
        ()))>getType());
77     Type *elementType = type->getElementType();
78     size = elementType->getSize() / 8;
79 }
80 auto size1 = genMachineVReg();
81 if (size > -255 && size < 255)
82 {

```

```

83     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, size1
84         , genMachineImm(size));
85 }
86 else
87 {
88     cur_inst = new LoadMInstruction(cur_block, LoadMInstruction::LDR,
89         size1, genMachineImm(size));
90 }
91 cur_block->InsertInst(cur_inst);
92 size1 = new MachineOperand(*size1);
93 auto off = genMachineVReg();
94 cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL, off
95     , idx, size1);
96 off = new MachineOperand(*off);
97 cur_block->InsertInst(cur_inst);
98 if (paramFirst || !first)
99 {
100     auto arr = genMachineOperand(operands[1]);
101     auto in = operands[1]->getDef();
102     if (in && in->isGep())
103     {
104         auto gep = (GepInstruction *)in;
105         if (gep->hasNoAsm())
106         {
107             gep->setInit(nullptr, 0);
108             gep->genMachineCode(builder);
109         }
110     }
111     cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
112         dst, arr, off);
113     cur_block->InsertInst(cur_inst);
114 }
115 else
116 {
117     auto addr = genMachineVReg();
118     auto base1 = new MachineOperand(*base);
119     if (operands[1]->getEntry()->isVariable() && ((IdentifierSymbolEntry
120         *) (operands[1]->getEntry()))->isGlobal())
121     {
122         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::
123             ADD, addr, base1, off);
124         cur_block->InsertInst(cur_inst);
125         addr = new MachineOperand(*addr);
126         cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
127             dst, addr);
128     }
129 }
130 else
131 {

```

```

124         auto fp = genMachineReg(11);
125         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::
            ADD, addr, fp, base1);
126         cur_block->InsertInst(cur_inst);
127         addr = new MachineOperand(*addr);
128         cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::
            ADD, dst, addr, off);
129     }
130     cur_block->InsertInst(cur_inst);
131 }
132 }

```

浮点类型 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在扩展寄存器中。

对于浮点类型的指令比如有 vadd,vsub,vmsr,vcvt,vldr 等，根据相应的情况进行判断。

break、continue 语句 对于 break 和 continue 语句，由于其并不会出现在循环的外部，所以在语法分析阶段已经进行检查，如果没有在 while 循环的结构体中，直接报错；在语法树中，不论是在 break 还是 continue 都会生成无条件跳转指令，通过 while 父节点判断跳转的位置，并进行跳转，在 while 中设置相应的回填和基本块。

非叶函数 对于非叶函数，其和叶函数一个最重要的不同是需要保存更多的寄存器，所以其序言和尾声的实现方式会不同，非叶函数的序言需要将更多的寄存器保存堆栈中，比如需要保存 lr 寄存器，在尾声方面，遇到叶函数就分支到存储在 LR 寄存器中的地址，而遇到非叶函数是直接 pop 到 PC 寄存器。

3. 结果展示

由于本次的样例代码很多，所以选其中一个进行展示，样例代码如下：

目标代码生成样例代码

```

1  int func(int n) {
2      if (n <= 50) {
3          putint(n);
4          return 1;
5      }
6      else {
7          putint(n);
8          return 0;
9      }
10 }
11
12 int main() {
13     int i;

```

```
14
15  if (func(0) == 1 || func(50) == 1 && func(100) == 0)
16      i = 0;
17  else
18      i = 1;
19
20  if (func(50) == 1 && func(40) == 1 || func(1) == 1 )
21      i = 0;
22  else
23      i = 1;
24
25  return 0;
26 }
```

目标代码测试结果

```
1      .cpu cortex-a72
2      .arch armv8-a
3      .fpu vfpv3-d16
4      .arch_extension crc
5      .text
6      .global func
7      .type func , %function
8  func:
9      push {r3, r4, fp, lr}
10     mov fp, sp
11     sub sp, sp, #8
12  .L27:
13     str r0, [fp, #-4]
14     ldr r4, [fp, #-4]
15     cmp r4, #50
16     movle r4, #1
17     movgt r4, #0
18     ble .L29
19     b .L34
20  .L29:
21     ldr r4, [fp, #-4]
22     mov r0, r4
23     bl putint
24     mov r0, #1
25     add sp, sp, #8
26     pop {r3, r4, fp, lr}
27     bx lr
28  .L34:
29     b .L30
30  .L30:
31     ldr r4, [fp, #-4]
32     mov r0, r4
33     bl putint
```

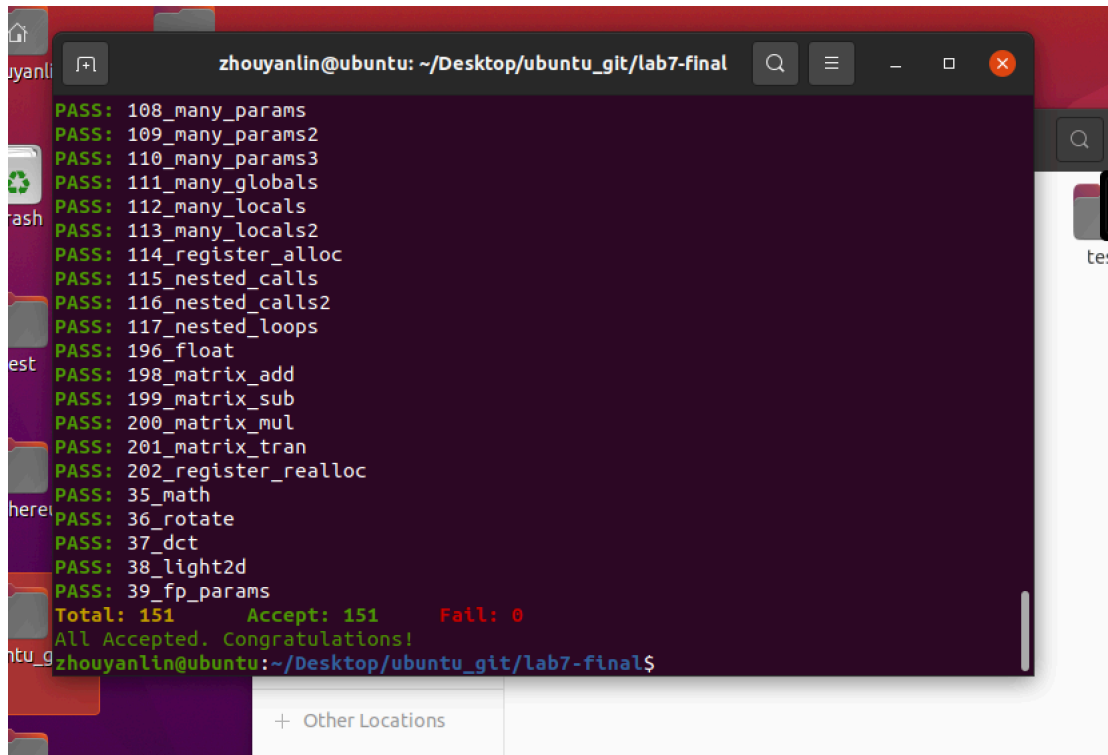


```
34      mov r0 , #0
35      add sp , sp , #8
36      pop {r3, r4, fp, lr}
37      bx lr
38
39      .global main
40      .type main , %function
41 main:
42      push {r3, r4, fp, lr}
43      mov fp , sp
44      sub sp , sp , #8
45 .L35:
46      mov r0 , #0
47      bl func
48      mov r4 , r0
49      cmp r4 , #1
50      moveq r4 , #1
51      movne r4 , #0
52      beq .L37
53      b .L43
54 .L37:
55      ldr r4 , =0
56      str r4 , [fp , #-4]
57      b .L39
58 .L39:
59      mov r0 , #50
60      bl func
61      mov r4 , r0
62      cmp r4 , #1
63      moveq r4 , #1
64      movne r4 , #0
65      beq .L55
66      b .L58
67 .L55:
68      mov r0 , #40
69      bl func
70      mov r4 , r0
71      cmp r4 , #1
72      moveq r4 , #1
73      movne r4 , #0
74      beq .L51
75      b .L61
76 .L51:
77      ldr r4 , =0
78      str r4 , [fp , #-4]
79      b .L53
80 .L53:
81      mov r0 , #0
```

```
82      add sp, sp, #8
83      pop {r3, r4, fp, lr}
84      bx lr
85  .L61:
86      b .L54
87  .L54:
88      mov r0, #1
89      bl func
90      mov r4, r0
91      cmp r4, #1
92      moveq r4, #1
93      movne r4, #0
94      beq .L51
95      b .L64
96  .L64:
97      b .L52
98  .L52:
99      ldr r4, =1
100     str r4, [fp, #-4]
101     b .L53
102  .L58:
103     b .L54
104  .L43:
105     b .L40
106  .L40:
107     mov r0, #50
108     bl func
109     mov r4, r0
110     cmp r4, #1
111     moveq r4, #1
112     movne r4, #0
113     beq .L44
114     b .L47
115  .L44:
116     mov r0, #100
117     bl func
118     mov r4, r0
119     cmp r4, #0
120     moveq r4, #1
121     movne r4, #0
122     beq .L37
123     b .L50
124  .L50:
125     b .L38
126  .L38:
127     ldr r4, =1
128     str r4, [fp, #-4]
129     b .L39
```

```
130 .L47:  
131     b .L38  
132  
133     .ident "hlast"
```

最终本学期的编译器测试代码的结果如图1所示。可以看出，已经通过了所有的 151 个样例，也是对本学期编译原理课程一个最好的交代。



```
zhouyanlin@ubuntu: ~/Desktop/ubuntu_git/lab7-final  
PASS: 108_many_params  
PASS: 109_many_params2  
PASS: 110_many_params3  
PASS: 111_many_globals  
PASS: 112_many_locals  
PASS: 113_many_locals2  
PASS: 114_register_alloc  
PASS: 115_nested_calls  
PASS: 116_nested_calls2  
PASS: 117_nested_loops  
PASS: 196_float  
PASS: 198_matrix_add  
PASS: 199_matrix_sub  
PASS: 200_matrix_mul  
PASS: 201_matrix_tran  
PASS: 202_register_realloc  
PASS: 35_math  
PASS: 36_rotate  
PASS: 37_dct  
PASS: 38_light2d  
PASS: 39_fp_params  
Total: 151      Accept: 151      Fail: 0  
All Accepted. Congratulations!  
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/lab7-final$
```

图 1: 编译器运行结果

六、 代码优化

(一) 思想简介

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 ϕ 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 ϕ 函数。

例如下面这段代码：

样例函数

```
1 int main() {  
2     int x, cond = 1;  
3     if (cond > 0)  
4         x = 1;  
5     else  
6         x = -1;  
7     return x;  
8 }
```

前端生成的代码：

中间代码

```
1 define dso_local i32 @main() {  
2     %1 = alloca i32  
3     %2 = alloca i32  
4     store i32 1, i32* %1  
5     %3 = load i32, i32* %1  
6     %4 = icmp sgt i32 %3, 0  
7     br i1 %4, label %5, label %8  
8  
9     5:  
10    store i32 1, i32* %2  
11    br label %6  
12  
13    6:  
14    %7 = load i32, i32* %2  
15    ret i32 %7  
16  
17    8:  
18    %9 = sub i32 0, 1  
19    store i32 %9, i32* %2  
20    br label %6  
21 }
```

Mem2Reg 转换后的代码：

转换

```
1 define dso_local i32 @main() {  
2     %1 = icmp sgt i32 1, 0  
3     br i1 %1, label %2, label %5  
4  
5 2:  
6     br label %3  
7  
8 3:  
9     %4 = phi i32 [ 1, %2 ], [ %6, %5 ]  
10    ret i32 %4  
11  
12 5:  
13    %6 = sub i32 0, 1  
14    br label %3  
15 }
```

2. 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。

图着色寄存器分配也是对线性寄存器分配的一个优化。

（二） 个人贡献

由于代码优化部分主要是由我的队友负责，我在这部分是与队友讨论思想和算法，然后确定好一个比较高效的实现方式后去完成。

七、 总结与展望

(一) 学期总结

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。

(二) 未来展望

最后这次实验做完也已经准备开启大三下，其实大学生活也剩的不多了。大学生活是对一个人整体的考验，正是如此，才造就了今天的我。现在也在人生的岔路口中，不论将来是出国还是保研，我都很感谢编译原理这门课程对我的磨练，希望自己在今后的人生中可以得到更好的发展，万事胜意、心想事成、未来可期。

致谢

行文至此，落笔为终。人生已二十载，岁月匆匆。目之所及，尽是回忆；心之所想，皆是过往。值此之际，亦道不尽心中谢意。

独学而无友，则孤陋而寡闻。我要感谢王刚老师在学习上给予的帮助，在此表示诚挚的谢意！桂花同载酒，不负少年游。更要感谢一路陪伴的朋友们，让我可以分享快乐与失意，感受温暖和力量，愿我们都前程似锦。

父母之爱子，则为之计深远。借此机会，特别感谢我的父母和家人。在我二十载的求学路上，是你们的默默付出与支持，激励着我砥砺前行。灿灿萱草花，罗生北堂下。南风吹其心，摇摇为谁吐？春晖寸草，难以回报。愿你们平安康乐，岁月无恙。

欲谢之人众多，然纸短情长，未能提及者，吾深表歉意。诸君之恩，吾没齿难忘。梧高凤必至，花香蝶自来。人终向前走，花自向阳开。

MIN

参考文献

- [1] MIT Laboratory for Computer Science and IBM Thomas J. Watson Research Center. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

NIKU