编译实验补充材料

刘彬彬

2024 年 4 月 2 日

此文档是编译实验的配套材料，是整个实验体系的重要一环，旨在让读者更好理解相关的实验内容。目前文档的内容包含各种难度的文法、Flex 和 Bison 的相关知识和使用案例，中端 LLVM 编译系统相关知识，后端 LoongArch 体系结构和指令生成相关内容。

1 前言

现代编译器都把编译器分成了三个部分，分别是前端（front-end）、中端（middle-end）、后端（back-end），每一部分都承担不同的功能：

- 前端：将源语言翻译成中间表示
- 中端：在中间表示上做优化
- 后端：得到目标平台语言

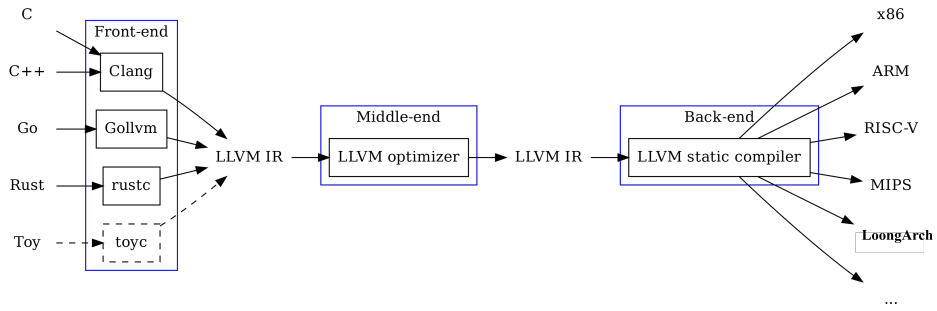

图 1. LLVM 编译器架构

本实验将这三个模块划分为 5 个实验，其中前三个实验包含前端的内容，后两个实验包含中端和后端的内容。实验一和实验二使用 Flex 和 Bison 两个工具对前端进行构建，通过实验一，我们可以熟悉使用 Flex 完成词法分析；通过实验二，我们可以熟悉使用 Bison 完成部分语法分析，生成一个抽象语法树，而实验三介绍了中间语言 IR，并要求同学们使用实验提供的框架将抽象语法树转变为中间语言。实验四要求同学们在中间语言上进行优化，实验主要介绍了 Mem2Reg 优化理论，要求同学们根据实验提供的理论自己对数据进行建模，实现优化代码。实验五将经过优化的代码转化为 LoongArch 后端代码，主要包括寄存器分配和指令翻译两个部分。

2 各等级文法

文法采用扩展的 Backus 范式 (EBNF, Extended Backus-NaurForm) 表示，其中：

- 符号 [...] 表示方括号内包含的为可选项
- 符号 {...} 表示花括号内包含的为可重复 0 次或多次的项
- 终结符或者是单引号括起的串 (比如 ' int ' 表示字符串 int)，或者是 Ident, InstConst 这样的记号。
- 课本 P75 有详细的例子和对此表示方法的解释，此处不作赘述。

2.1 a - -

这个文法仅仅包含最简单的函数定义和一个 return 语句
a - -的完整文法如下：
CompUnit → [CompUnit] FuncDef
FuncDef → FuncType **Ident** '(' ')' Block
FuncType → 'void' | 'int'
Block → ' { ' { BlockItem } ' } '
BlockItem → Stmt
Stmt → ' return ' [Exp] ' ; '
Exp → AddExp
AddExp → MulExp
MulExp → IntConst

例 2.1.1. 例如下面的程序，

```
int main(){
    return 3;
}
```

推导过程如下：
CompUnit ⇒ FuncDef ⇒ FuncType **Ident** '(' ')' Block ⇒ 'int' main '(' ')' Block
Block ⇒ ' { ' { BlockItem } ' } '
BlockItem ⇒ Stmt ⇒ 'return' Exp ';'
Exp ⇒ AddExp ⇒ MulExp ⇒ IntConst ⇒ 3

2.2 a-

除了函数的定义之外，此文法增加了变量定义和基本的加减运算的部分
a-的完整文法如下：
CompUnit → [CompUnit] FuncDef
FuncDef → FuncType **Ident** '(' ')' Block
FuncType → 'void' | 'int'

Block → ' { ' { BlockItem } ' } '
BlockItem → Decl | Stmt
Decl → VarDecl
VarDecl → ' int ' VarDef { ' , ' VarDef }' ; '
VarDef → **Ident** | **Ident** ' = ' InitVal
InitVal → Exp
Stmt → Block | ' return ' [Exp] ' ; '
Exp → AddExp
AddExp → PrimaryExp | AddExp(' + ' | ' - ') PrimaryExp
PrimaryExp → ' (' Exp ') ' | **Ident** | **IntConst**

例 2.2.1. 针对下面一段程序，

```
int main(){
    int a = 0, b = 1;
    return 1+3;
}
```

推导过程如下：
CompUnit ⇒ FuncDef ⇒ FuncType **Ident** '(' ')' Block ⇒ 'int' main '(' ')' Block
Block ⇒ ' { ' BlockItem1 BlockItem2 ' } ' ⇒ ' { ' Decl Stmt ' } '
Decl ⇒ VarDecl ⇒ 'int' VarDef1 ',' VarDef2 ';'
VarDef1 ⇒ Ident1 '=' InitVal1
VarDef2 ⇒ Ident2 '=' InitVal2
Ident1 ⇒ a
Ident2 ⇒ b
InitVal1 ⇒ Exp1 ⇒ AddExp1 ⇒ PrimaryExp1 ⇒ IntConst1 ⇒ 0
InitVal2 ⇒ Exp2 ⇒ AddExp2 ⇒ PrimaryExp2 ⇒ IntConst2 ⇒ 1
Stmt ⇒ 'return'Exp3';'
Exp3 ⇒ AddExp3 ⇒ AddExp3 '+' PrimaryExp3 ⇒ PrimaryExp4 '+' PrimaryExp3
PrimaryExp4 ⇒ IntConst2 ⇒ 1
PrimaryExp3 ⇒ IntConst3 ⇒ 3

2.3 a

包含 a-中所有内容赋值语句的内容，以及乘法除法的内容，即将上述 AddExp 的内容变更为
AddExp → MulExp | AddExp(' + ' | ' - ') MulExp
MulExp → PrimaryExp | MulExp (' * ' | ' / ') PrimaryExp
赋值语句的内容需要将 Stmt 的内容变更为
Stmt → Lval ' = ' Exp | Block | ' return ' [Exp] ' ; '
Lval → **Ident**
a 的完整文法如下：
CompUnit → [CompUnit] FuncDef

FuncDef → FuncType **Ident** '(' ')' Block
FuncType → 'void' | 'int'
Block → ' { ' { BlockItem } ' } '
BlockItem → Decl | Stmt
Decl → VarDecl
VarDecl → ' int ' VarDef { ' , ' VarDef }' ; '
VarDef → **Ident** | **Ident** ' = ' InitVal
InitVal → Exp
Stmt → Lval ' = ' Exp | Block | ' return ' [Exp] ' ; '
Exp → AddExp
Lval → **Ident**
AddExp → MulExp | AddExp(' + ' | ' - ') MulExp
MulExp → PrimaryExp | MulExp (' * ' | ' / ') PrimaryExp
PrimaryExp → ' (' Exp ') ' | **Ident** | **IntConst**

例 2.3.1. 针对下面一段程序，

```
1 int main(){
2     int a = 2, b = 100;
3     int c = a*b;
4     return c;
5 }
```

推导过程如下：
CompUnit ⇒ FuncDef ⇒ FuncType **Ident** '(' ')' Block ⇒ 'int' main '(' ')' Block
Block ⇒ ' { ' BlockItem1 BlockItem2 BlockItem3 ' } ' ⇒ ' { ' Decl1 Decl2 Stmt ' } '
Decl1 ⇒ VarDecl ⇒ 'int' VarDef1 ',' VarDef2 ';'
VarDef1 ⇒ Ident1 '=' InitVal1
VarDef2 ⇒ Ident2 '=' InitVal2
Ident1 ⇒ a
Ident2 ⇒ b
InitVal1 ⇒ Exp1 ⇒ AddExp1 ⇒ PrimaryExp1 ⇒ IntConst1 ⇒ 2
InitVal2 ⇒ Exp2 ⇒ AddExp2 ⇒ PrimaryExp2 ⇒ IntConst2 ⇒ 100
Decl2 ⇒ 'int' VarDef3 ';'
VarDef3 ⇒ Ident3 '=' InitVal3
Ident3 ⇒ c
InitVal3 ⇒ Exp3 ⇒ AddExp3 ⇒ MulExp1
MulExp1 ⇒ MulExp2 '*' PrimaryExp3 ⇒ PrimaryExp4 '*' PrimaryExp3
PrimaryExp4 ⇒ Ident1 ⇒ a
PrimaryExp3 ⇒ Ident2 ⇒ b
Stmt ⇒ 'return'Exp4';'
Exp4 ⇒ AddExp4 ⇒ MulExp3 ⇒ PrimaryExp5 ⇒ Ident3 ⇒ c

2.4 a+

除了包含 a 里面的内容，还需要增加 if-else 内容，即将 Stmt 增加如下内容

Stmt → ' **if** ' ' (' Cond ') ' Stmt [' **else** ' Stmt]

除此之外，还将增加下面的内容

Cond → LOrExp

LOrExp → LAndExp | LOrExp ' || ' LAndExp

LAndExp → EqExp | LAndExp ' && ' EqExp

EqExp → RelExp | EqExp (' == ' | ' != ') RelExp

RelExp → AddExp | RelExp (' < ' | ' > ' | ' <= ' | ' >= ') AddExp

a+ 的完整文法如下：

CompUnit → [CompUnit] FuncDef

FuncDef → FuncType **Ident** '(' ')' Block

FuncType → 'void' | 'int'

Block → ' { ' { BlockItem } ' } '

BlockItem → Decl | Stmt

Decl → VarDecl

VarDecl → ' int ' VarDef { ' , ' VarDef }' ; '

VarDef → **Ident** | **Ident** ' = ' InitVal

InitVal → Exp

Stmt → ' **if** ' ' (' Cond ') ' Stmt [' **else** ' Stmt]

| Lval ' = ' Exp | Block | ' return ' [Exp] ' ; '

Exp → AddExp

Lval → **Ident**

AddExp → MulExp | AddExp(' + ' | ' - ') MulExp

MulExp → PrimaryExp | MulExp (' * ' | ' / ') PrimaryExp

PrimaryExp → ' (' Exp ') ' | **Ident** | **IntConst**

Cond → LOrExp

LOrExp → LAndExp | LOrExp ' || ' LAndExp

LAndExp → EqExp | LAndExp ' && ' EqExp

EqExp → RelExp | EqExp (' == ' | ' != ') RelExp

RelExp → AddExp | RelExp (' < ' | ' > ' | ' <= ' | '>= ') AddExp

例 2.4.1. 针对下面一段程序，

```
1 int main(){
2     int a = 1, b = 100;
3     if(a==1 && b==100){
4         a = 0;
5     }
6     else{
7         a = b;
```

```
    }
    return a;
}
```

推导过程如下：

CompUnit ⇒ FuncDef ⇒ FuncType **Ident** '(' ')' Block ⇒ 'int' main '(' ')' Block

Block ⇒ ' { ' BlockItem1 BlockItem2 BlockItem3 ' } ' ⇒ ' { ' Decl Stmt1 Stmt2 ' } '

Decl ⇒ VarDecl ⇒ 'int' VarDef1 ',' VarDef2 ';'

VarDef1 ⇒ Ident1 '=' InitVal1

VarDef2 ⇒ Ident2 '=' InitVal2

Ident1 ⇒ a

Ident2 ⇒ b

InitVal1 ⇒ Exp1 ⇒ AddExp1 ⇒ MulExp1 ⇒ PrimaryExp1 ⇒ IntConst1 ⇒ 1

InitVal2 ⇒ Exp2 ⇒ AddExp2 ⇒ MulExp2 ⇒ PrimaryExp2 ⇒ IntConst2 ⇒ 100

Stmt1 ⇒ 'if''('Cond')'Stmt3 'else'Stmt4

Cond ⇒ LOrExp ⇒ LAndExp

LAndExp ⇒ LAndExp1'&&'EqExp1 ⇒ EqExp2'&&'EqExp1

EqExp1 ⇒ EqExp3'=='RelExp1 ⇒ RelExp2'=='RelExp1

RelExp1 ⇒ AddExp2 ⇒ MulExp2 ⇒ PrimaryExp2 ⇒ IntConst2 ⇒ 100

RelExp2 ⇒ AddExp4 ⇒ MulExp4 ⇒ PrimaryExp4 ⇒ Ident2 ⇒ b

EqExp2 ⇒ EqExp4'=='RelExp4 ⇒ RelExp3'=='RelExp4

RelExp4 ⇒ AddExp1 ⇒ MulExp1 ⇒ PrimaryExp1 ⇒ IntConst1 ⇒ 1

RelExp3 ⇒ AddExp3 ⇒ MulExp3 ⇒ PrimaryExp3 ⇒ Ident1 ⇒ a

Stmt3 ⇒ Lval '='Exp3

Exp3 ⇒ AddExp4 ⇒ MulExp4 ⇒ PrimaryExp4 ⇒ IntConst3 ⇒ 0

Stmt4 ⇒ Lval '='Exp4

Exp4 ⇒ AddExp4 ⇒ MulExp4 ⇒ PrimaryExp4 ⇒ Ident2 ⇒ b

Lval ⇒ Ident1 ⇒ a

Stmt2 ⇒ 'return'Exp1';'

Exp1 ⇒ AddExp3 ⇒ MulExp3 ⇒ PrimaryExp3 ⇒ Ident1 ⇒ a

2.5 a++

包含 a+ 内的所有内容，并且增加一元运算符内容，一元运算符的内容需要将原来的 MulExp 的内容变更为

MulExp → UnaryExp | MulExp (' * ' | ' / ') UnaryExp

UnaryExp → PrimaryExp | UnaryOp UnaryExp

UnaryOp → ' + ' | ' - ' | ' ! '

a++ 的完整文法如下：

CompUnit → [CompUnit] FuncDef

FuncDef → FuncType **Ident** '(' ')' Block

FuncType → 'void' | 'int'

Block → ' { ' { BlockItem } ' } '
BlockItem → Decl | Stmt
Decl → VarDecl
VarDecl → ' int ' VarDef { ' , ' VarDef }' ; '
VarDef → **Ident** | **Ident** ' = ' InitVal
InitVal → Exp
Stmt → ' **if** ' ' (' Cond ') ' Stmt [' **else** ' Stmt]
| Lval ' = ' Exp | Block | ' return ' [Exp] ' ; '
Exp → AddExp
Lval → **Ident**
AddExp → MulExp | AddExp(' + ' | ' - ') MulExp
MulExp → UnaryExp | MulExp (' * ' | ' / ') PrimaryExp
UnaryExp → PrimaryExp | UnaryOp UnaryExp
UnaryOp → '+' | '-' | '!'
PrimaryExp → ' (' Exp ') ' | **Ident** | **IntConst**
Cond → LOrExp
LOrExp → LAndExp | LOrExp ' || ' LAndExp
LAndExp → EqExp | LAndExp ' && ' EqExp
EqExp → RelExp | EqExp (' == ' | ' != ') RelExp
RelExp → AddExp | RelExp (' < ' | ' > ' | ' <= ' | ' >= ') AddExp

例 2.5.1. 针对下面一段简单的程序，

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

推导过程如下：
CompUnit ⇒ FuncDef ⇒ FuncType **Ident** '(' ')' Block ⇒ 'int' main '(' ')' Block
Block ⇒ ' { ' BlockItem1 BlockItem2 BlockItem3 ' } ' ⇒ ' { ' Decl Stmt1 Stmt2 ' } '
Decl ⇒ VarDec ⇒ 'int' VarDef ';'
VarDef ⇒ Ident '=' InitVal
Ident ⇒ a
InitVal ⇒ Exp ⇒ AddExp ⇒ MulExp ⇒ UnaryExp
UnaryExp ⇒ UnaryOp UnaryExp ⇒ UnaryOp PrimaryExp

UnaryOp ⇒ '-'

PrimaryExp ⇒ IntConst ⇒ 1

Stmt1 ⇒ 'if''('Cond')'Stmt3 'else'Stmt4

Cond ⇒ LOrExp ⇒ LAndExp ⇒ EqExp ⇒ RelExp ⇒ RelExp '>' AddExp2

RelExp ⇒ AddExp1 ⇒ MulExp1 ⇒ UnaryExp1 ⇒ PrimaryExp1 ⇒ Ident ⇒ a

AddExp2 ⇒ MulExp2 ⇒ UnaryExp2 ⇒ PrimaryExp2 ⇒ IntConst1 ⇒ 0

Stmt3 ⇒ 'return'Exp';'

Stmt4 ⇒ Lval '='Exp1

Exp1 ⇒ AddExp3 ⇒ MulExp3 ⇒ UnaryExp1 ⇒ UnaryOp UnaryExp

UnaryOp ⇒ '-'

UnaryExp ⇒ PrimaryExp1 ⇒ Ident ⇒ a

Stmt4 ⇒ 'return'Exp';'

Exp ⇒ AddExp2 ⇒ MulExp1 ⇒ UnaryExp1 ⇒ PrimaryExp1 ⇒ Ident ⇒ a

3 Flex 脚本格式及案例

3.1 Flex 简介

编写一个后缀为 ".l"的文件，然后通过 flex 生成一个.c 文件，该文件实际上是一个词法分析器，通过这一个文件，我们可以自动地对输入进行分析，从而产生 token stream。

需要注意的是本节（即第 3 节）和之后的第 4 节介绍只是介绍 Flex 和 Bison 的基本使用方法，并不涉及实验的具体内容。在第 5 节（Flex 和 Bison 协同使用）中才涉及到具体的实验内容，所以请读者将这三节一并读完，再去完成实验。

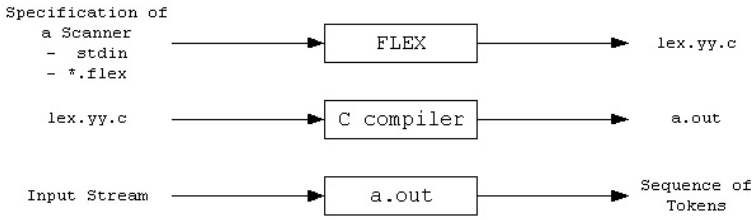

图 2. Work Flow of Flex

Flex 分为三个部分

- 开头部分包括在一对**%{%}** 中，包含一些 c/cpp 代码，该部分代码会被原封不动的复制到生成的.c 文件中
- 中间部分包括在一对%% %% 中，使用**模式动作**方式，对出现在输入流进行词法分析，其中，模式为正则表达式，动作为匹配到该正则表达式时所执行的 c/c++ 代码
- 第三个部分为 c++ 代码，该部分代码会被原封不动的复制到生成的.c 文件中

生成的.c 文件中会提供一个接口 **void yylex()**。yylex() 默认输入读取 stdin

3.2 案例

下面通过一个简单的案例（单词个数统计）来学习一下**.l** 文件是如何编写的

```
1  %option noyywrap
2  %{
3  //在%{和%}中的代码会被原样照抄到生成的lex.yy.c文件的开头，
4  //也就是在%{和}%中，你应该按C语言写代码，在这里可以完成变量
5  //声明与定义、相关库的导入和函数定义
6  #include <string.h>
7  int chars = 0;
8  int words = 0;
9  %}
10
11 %%
```

```
12 /* 注意这里的%%开头*/
13 /* %%开头和%%结尾之间的内容就是使用flex进行解析的部分 */
14 /* 你可以按照这种方式在这个部分写注释，注意注释最开头的空格，这是必须的 */
15 /* 你可以在这里使用你熟悉的正则表达式来编写模式，你可以用C代码来指定模式匹配时对
       应的动作 */
16 /* 在%%和%%之间，你应该按照如下的方式写模式和动作 */
17 /* 模式 动作 */
18 /* 其中模式就是正则表达式，动作为模式匹配执行成功后执行相应的动作，这里的动作就
       是相应的代码 */
19 /* 你可以仔细研究下后面的例子 */
20 /* [a-zA-Z]+ 为正则表达式，用于匹配大小写字母 */
21 /* {chars += strlen(yytext);words++;} 则为匹配到大小写字母后，执行的动作（代
       码），这里是完成一个字符累加操作 */
22 /* 这里yytext的类型为 char*， 是一个指向匹配到字符串的指针 */
23 /* yytext是flex自动生成的，在%%和%%之中无需额外定义或者声明 */
24
25 /* 一条 模式 + 动作 */
26 [a-zA-Z]+ { chars += strlen(yytext);words++;}
27
28 /* 另一条 模式 + 动作； . 匹配任意字符，这里匹配非大小写字母的其他字符。这里思考
       一个问题，A既可以被[a- zA-Z]+匹配，也可以被.匹配，在这个程序中为什么A优先被[
       a-zA-Z]+匹配？如果你感兴趣可以去看另一个文档 */
29 . {}
30 /* 对其他所有字符，不做处理，继续执行 */
31 /* 注意这里的%%结尾 */
32 %%
33
34 // flex部分结束，这里可以正常写c代码了
35 int main(int argc, char **argv){
36     // yylex()是flex提供的词法分析例程，调用yylex()即开始执行Flex的词法分析，同
           样的yylex()也是flex自 行生成的，无需额外定义和生成，默认输入读取stdin
37     // 如果不清楚什么是stdin，可以自己百度查一下
38     yylex();
39     // 输出 words和chars，这些变量在匹配过程中，被执行相应的动作
40     printf("look, I find %d words of %d chars\n", words, chars);
41     return 0;
42 }
```

利用 Flex 生成.c 文件

```
1 $ flex -o wc.c wc.l
2 $ gcc wc.l
3 $ ./a.out
```

```
4 hello world
5 ^D
6 look, I find 2 words of 10 chars
```

注 3.2.1. 注: 在以 stdin 为输入时，需要按下 ctrl+D 以退出

4 Bison 脚本格式及案例

4.1 Bison 简介

Bison 是一款解析器生成器（parser generator），它可以将 LALR 文法转换成可编译的 C 代码，从而大大减轻程序员手动设计解析器的负担。Bison 是 GNU 对早期 Unix 的 Yacc 工具的一个重新实现，所以文件扩展名为 ‘.y‘。（Yacc 的意思是 Yet Another Compiler Compiler。）

.y 文件分为四个部分

第一个部分包含在一对%{%} 中，里面可以写 c/cpp 代码，代码会原封不动的复制到生成的.c/cpp 文件中

第二个部分是定义部分，包含 token、type、union、start 的定义

- token：可以提供给 Flex 使用，告诉 bison 在语法分析程序中记号的名称。(实际上 token 定义指定了终结符在程序中使用的变量)
- type：和 token 类似，只不过 type 指定的非终结符 (实际上指定了非终结符在程序中使用的变量)
- union：指定了所有 token 和 type 可能使用的变量
- start：指定了起始符

第三部分是文法部分，包含在一对%% %% 中，除了文法之外还要指定语义动作，如果没有规定语义动作。

第四部分可以用来编写 c/cpp 代码，此部分代码和第一部分一样，将会原封不动的复制到生成的.c 文件中

4.2 案例

```
%{
#include <stdio.h>
/* 这里是序曲 */
/* 这部分代码会被原样拷贝到生成的 .c 文件的开头 */
int yylex(void);
void yyerror(const char *s);
%}

/* 这些地方可以输入一些 bison 指令 */
/* 比如用 %start 指令指定起始符号，用 %token 定义一个 token */
%start reimu
%token REIMU

%%
/* 从这里开始，下面是解析规则 */
reimu : marisa { /* 这里写与该规则对应的处理代码 */ puts("rule1"); }
```

```
17        | REIMU { /* 这里写与该规则对应的处理代码 */ puts("rule2"); }
18        ; /* 规则最后不要忘了用分号结束哦~ */
19
20  /* 这种写法表示 —— 空输入 */
21  marisa : { puts("Hello!"); }
22
23  %%
24  /* 这里是尾声 */
25  /* 这部分代码会被原样拷贝到生成的 .c 文件的末尾 */
26
27  int yylex(void)
28  {
29      int c = getchar(); // 从 stdin 获取下一个字符
30      switch (c) {
31          case EOF: return YYEOF;
32          case 'R': return REIMU;
33          default: return YYUNDEF;  // 报告 token 未定义，迫使 bison 报错。
34          // 由于 bison 不同版本有不同的定义。如果这里 YYUNDEF 未定义，请尝试
                YYUNDEFTOK 或使用一个随意的整数。
35      }
36  }
37
38  void yyerror(const char *s)
39  {
40      fprintf(stderr, "%s\n", s);
41  }
42
43  int main(void)
44  {
45      yyparse(); // 启动解析
46      return 0;
47  }
```

一些值得注意之处：

1. Bison 传统上将 token 用大写单词表示，将 symbol 用小写字母表示。
2. Bison 能且只能生成解析器源代码（一个 **.c** 文件），并且入口是 **yyparse**，所以为了让程序能跑起来，你需要手动提供 **main** 函数。（第四部分可以选择写一个 main 函数进去，这样就可以直接运行这个.c 文件）
3. Bison 不能检测你的 action code 是否正确——它只能检测文法的部分错误，其他代码都是原样粘贴到.c 文件中。

4. Bison 需要你提供一个 **yylex** 来获取下一个 token。(可以通过 Flex 进行提供)

5. Bison 需要你提供一个 **yyerror** 来提供合适的报错机制。

此外，上面这个 **.y** 是可以工作的——尽管它只能接受两个字符串。把上面这段代码保存为 **reimu.y**，执行如下命令来构建这个程序：

```
$ bison reimu.y
$ gcc reimu.tab.c
$ ./a.out
R<-- 不要回车在这里按 Ctrl-D
rule2
$ ./a.out
<-- 不要回车在这里按 Ctrl-D
Hello!
rule1
$ ./a.out
blablabla <-- 回车或者 Ctrl-D
Hello!
rule1    <-- 匹配到了 rule1
syntax error <-- 发现了错误
```

5 Flex 和 Bison 协同使用

下面通过一个具体的案例，a- -版文法来讲解 Flex 和 Bison 是如何协同使用的，首先我们需要知道几个存在于.y 文件和.l 文件生成的.cpp/.c 文件中的常用全局变量

- yylval：.y 文件中定义的 union 的一个实例化对象
- yytext：当前识别的文本
- yyleng：当前识别文本的长度

首先让我们来看 lexer.l 文件的内容

```
%option noyywrap
%{
    /*
        这部分会被原样拷贝到生成的cpp文件的开头
    */
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>

#include "SyntaxTree.h"
/*SyntaxTree.h用来定义一些可能用到的数据结构*/
#include "parser.hpp"
/*parser.hpp是parser.y生成的hpp文件,里面包含了token, 供Flex使用*/

int line_number;//行号
int column_start_number;//token开始的列号
int column_end_number;//token结束的列号

int current_token;//保存当前识别到的token

bool is_head_print;//用来标记是否为第一次打印，如果是第一次打印则需要先输出一个
    表头
void print_msg(std::ostream &out) { //此函数用来打印词法分析的结果
    if(!is_head_print){
        out << std::setw(10) << "Token"
            << std::setw(10) << "Text"
            << std::setw(10) << "line"
            << std::setw(10) << "(s,e)"
            << std::endl;
            is_head_print = true;
```

```
31      }
32      out << std::setw(10) << current_token
33          << std::setw(10) << yytext
34          << std::setw(10) << line_number
35          << std::setw(10) << "(" << column_start_number << "," <<
                column_end_number << ")"
36          << std::endl;
37  }
38  int handle_token(int token) {
39      current_token = token;
40      column_start_number = column_end_number;
41      yylval.symbol_size = strlen(yytext);
42      yylval.current_symbol = new char[yylval.symbol_size];
43      strcpy(yylval.current_symbol,yytext);
44      column_end_number += strlen(yytext);//yytext是正则表达式匹配的部分
45      print_msg(std::cout);
46      return token;
47  }
48  %}
49  %%
50  \/\*([^\*]|\*[^\/])*\*+\/ {
51      column_start_number = column_end_number;
52      for (unsigned i = 0; i < strlen(yytext); i++) {
53          if (yytext[i] == '\n') {
54              line_number++;
55              column_end_number = 0;
56          } else
57              column_end_number++;
58      }
59  }
60  \/\/.* {
61      column_start_number = column_end_number;
62      column_end_number += strlen(yytext);
63  }
64  /* 可以发现有很多模式的动作是十分相似的*/
65  /*int, void , ( , ) 等是在parser.y里面定义的*/
66
67  int { return handle_token( INT);}
68  void { return handle_token( VOID);}
69  return { return handle_token(RETURN);}
70  [a-zA-Z_][a-zA-Z_0-9]* { return handle_token( Ident);}
```

```
71 [0-9]+ { return handle_token( IntConst);}
72 "(" { return handle_token( LPAREN);}
73 ")" { return handle_token( RPAREN);}
74 "{" { return handle_token( LBRACE); }
75 "}" { return handle_token( RBRACE);}
76 ";" { return handle_token( SEMICOLON); }
77 \n {
78     line_number++;
79     column_start_number = column_end_number;
80     column_end_number = 1;
81 }
82 " "|\r {
83     column_start_number = column_end_number;
84     column_end_number += strlen(yytext);
85 }
86 \t {
87     column_start_number = column_end_number;
88     column_end_number += 4;
89 }
90 . { return handle_token(ERROR); }
91
92 %%
```

下面再来查看下 parser.y 文件是如何配置的

```
1
2  %{
3  #include "SyntaxTree.h"
4  #include "SyntaxAnalyse.hpp"
5  #include <iostream>
6
7  int yylex();
8  int yyparse();
9  int yyrestart();
10
11 extern FILE* yyin;
12 extern char* yytext;
13 extern int line_number;
14 extern int column_end_number;
15 extern int column_start_number;
16
17 void yyerror(const char *s) {
18     std::cerr << s << std::endl;
```

```
19      std::cerr << "Error at line " << line_number << ": " << column_end_number
            << std::endl;
20      std::cerr << "Error: " << yytext << std::endl;
21      std::abort();
22  }
23
24  using namespace ast;
25  %}
26
27  %union {
28      char* current_symbol; //we can't use string or any other object with
            construct function in union.
29      int symbol_size;
30      struct ast::compunit_syntax *compunit ;
31      struct ast::func_def_syntax *func_def;
32      struct ast::expr_syntax *expr;
33      struct ast::literal_syntax *literal;
34      struct ast::stmt_syntax *stmt;
35      struct ast::block_syntax *block;
36      struct ast::return_stmt_syntax *return_stmt;
37      enum vartype var_type;
38  }
39  /*下面是存储终结点所使用的变量的定义*/
40  %token <current_symbol> INT VOID RETURN Ident
41  %token <current_symbol> LPAREN RPAREN LBRACE RBRACE
42  %token <current_symbol> IntConst
43  %token <current_symbol> SEMICOLON
44  %token <current_symbol> ERROR
45
46  /*下面是归约到type之后，存储type结点所使用的变量*/
47  %type <compunit> CompUnit
48  %type <func_def> FuncDef
49  %type <var_type> FuncType
50  %type <block> Block
51  %type <block> BlockItems
52  %type <stmt> Stmt
53  %type <expr> Exp
54  %type <expr> AddExp
55  %type <expr> PrimaryExp
56  /*定义一个起始符*/
57  %start CompUnit
```

```
58
59
60  %%
61  /*下面的部分用来写文法和语义动作*/
62  CompUnit
63  :CompUnit FuncDef { SyntaxAnalyseCompUnit($$,$1,$2);}
64  |FuncDef { SyntaxAnalyseCompUnit($$,nullptr,$1); }
65  /*
66      我们会发现$$代指存储CompUnit的结点，即compunit；$2代指存储FuncDef的结点，即
          funcdef
67  */
68
69  FuncDef
70  :FuncType Ident LPAREN RPAREN Block { SyntaxAnalyseFuncDef($$,$1,$2,$5);}
71
72  FuncType
73  :VOID { SynataxAnalyseFuncType($$,$1);}
74  |INT { SynataxAnalyseFuncType($$,$1);}
75
76  Block
77  : LBRACE BlockItems RBRACE { SynataxAnalyseBlock($$,$2);}
78
79  BlockItems
80  : BlockItems Stmt { SynataxAnalyseBlockItems($$,$1,$2);}
81  | { SynataxAnalyseBlockItems($$,nullptr,nullptr);}
82
83  Stmt
84  : RETURN Exp SEMICOLON { SynataxAnalyseStmtReturn($$,$2); }
85
86  PrimaryExp
87  : IntConst { SynataxAnalysePrimaryExpIntConst($$,$1); }
88
89  AddExp
90  : PrimaryExp{ $$=$1;}
91
92  Exp
93  : AddExp{$$=$1;}
94
95  %%
```

编写完成之后，我们可以运行 parser 文件夹下面的 compile.sh 脚本文件来结合生成对应的.hpp 和.cpp 文件。该脚本的内容如下所示：

```
flex --header-file=lexer.hpp -o lexer.cpp lexer.l
bison -d -o parser.cpp parser.y
```

他们的具体含义是分别使用 flex 和 bison 工具将编写的 lexer.l 文件和 parser.y 文件生成的文件名重定向为 lexer.cpp 和 parser.cpp，头文件重定向为 lexer.hpp 和 parser.hpp。

之后我们要编写 main 文件来驱动语法分析器，在 main 文件里面，我们调用了一个接口，此接口可以读取一个文本。从而进行语法分析。

至此，对 Flex 和 Bison 的介绍已经完毕，同学们可以参照 word 文档中的实验步骤来完成实验。

6 中端语言 LLVM-IR 简介

本节首先通过一个 C 语言示例引入 LLVM-IR，让读者能够初窥 IR 的形式，然后具体介绍 LLVM-IR 的语法和相关表示形式，最后通过代码框架介绍引入实验及其相关内容。

6.1 LLVM-IR 示例

现代编译器工程开发中，一般将源代码编译成与平台无关的中间表示（IR，如课本中所讲的三元式、四元式），这种形式更方便构建与源语言和目标平台无关的优化设计（如死代码删除），减少不必要的工作量，提高了代码复用的可能。LLVM-IR 是编译器 LLVM 使用的一种中间表示，下面通过例子来说明一个 C 语言程序会转化成什么形式的 LLVM-IR。

例 6.1.1.

```c
int main()
{
    int a ;
    int b ;
    int c ;
    a = 1; b = 2; c = 3;
    if(a > b && b > c){
        a = (a*3+5 + b) * c;
        return a + b;
    }else{
        return c + a;
    }
    return 1;
}
```

通过运行我们提供的脚本程序，可以把这个 C 语言变为 llvm-ir，如下所示：

```
; 下面的内容是一些描述目标平台信息的代码，对了解llvm-ir没有帮助，可以跳过
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80
    :128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
; 下面的代码用来定义一个函数
; 函数定义以 `define` 开头，i32 标明了函数的返回类型，其中 `main`是函数的名
    字，`@` 是其前缀
define dso_local i32 @main() #0 {
    ;每次声明一个新的变量都必须在内存中为这个变量开辟空间
```

```
12      ;如 %1 = alloca i32, align 4
13      ;表示 为一个32位整数开辟空间，并且4字节对齐，这段空间的首地址被放在一个虚
            拟寄存器中（虚拟寄存器编号可以无限递增），后端会负责将虚拟寄存器转换为
            物理寄存器（寄存器分配）
14      %1 = alloca i32, align 4
15      %2 = alloca i32, align 4
16      %3 = alloca i32, align 4
17      %4 = alloca i32, align 4
18      ;对于赋值语句，llvm-ir选择使用store语句将一个常量存储进内存
19      store i32 0, i32* %1, align 4
20      store i32 1, i32* %2, align 4
21      store i32 2, i32* %3, align 4
22      store i32 3, i32* %4, align 4
23      ;当我们企图使用这两个变量进行大小比较时，先要把这两个变量从内存中加载出来
24      %5 = load i32, i32* %2, align 4
25      %6 = load i32, i32* %3, align 4
26      ;该语句是大小比较语句 sgt: signed greater than ，比如下一条语句，就把比
            较的结果放到%7寄存器中
27      %7 = icmp sgt i32 %5, %6
28      :该语句根据%7的值选择跳到8号block还是16号block
29      ;这里可以看出 br实际上起到了划分block块的作用，称这种可以划分块的语句为终
            结语句
30      br i1 %7, label %8, label %16
31      ;下面是标号为8的寄存器的内容
32      ;实际上描述的是 代码中 b > c 的内容
33  8:                                                ; preds = %0
34      %9 = load i32, i32* %3, align 4
35      %10 = load i32, i32* %4, align 4
36      %11 = icmp sgt i32 %9, %10
37      br i1 %11, label %12, label %16
38  ; 下面的block描述的是if-true里面的内容，注意看，这里实际上只是return a+b，并
        没有前面的对a进行赋值的语句，因此可以判断，这段代码是死代码，最终将被优
        化，也即消除
39  12:                                               ; preds = %8
40      %13 = load i32, i32* %2, align 4
41      %14 = load i32, i32* %3, align 4
42      %15 = add nsw i32 %13, %14
43      store i32 %15, i32* %1, align 4
44      br label %20
45  ; 下面这段代码是else block里面的内容
```

```
16:                                               ; preds = %8, %0
   %17 = load i32, i32* %4, align 4
   %18 = load i32, i32* %2, align 4
   %19 = add nsw i32 %17, %18
   store i32 %19, i32* %1, align 4
   br label %20
; 加载寄存器里面的内容并返回
20:                                               ; preds = %16, %12
   %21 = load i32, i32* %1, align 4
   ret i32 %21
}
; 后面的代码删除也不妨碍我们理解llvm-ir代码
attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
    "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+
    cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

6.2 LLVM-IR 语法简介

- 在 LLVM-IR 中，一个.c 文件被描述为一个 module，一个 module 中包含若干个 function 和全局变量，每一个 function 包含若干个 basic block，一个 basic block 包含若干条 llvm-ir 指令，每个 block 实际上通过 block 中的最后一条语句连接成一个图结构。
- 每一个 function 里面的编号是独立的，也即每一个函数里面的寄存器都是从%1 开始。
- LLVM-IR 指令满足 SSA 格式，后面将对其进行详细叙述。
- LLVM-IR 所有的编号必须连续（如果使用数字作为编号的话）。

6.2.1 LLVM-IR 指令

下面说明常见的 llvm-ir 的指令。

llvm ir	uasage	intro
add	<result> = add <ty> <op1>, <op2>	加法
sub	<result> = sub <ty> <op1>, <op2>	减法
mul	<result> = mul <ty> <op1>, <op2>	乘法
div	<result> = div <ty> <op1>, <op2>	除法
icmp	<result> = icmp<cond> <ty> <op1>, <op2>	比较指令
and	<result> = and <ty> <op1>, <op2>	与
or	<result> = or <ty> <op1>, <op2>	或
alloca	<result> = alloca <ty>	申请内存空间
load	<result> = load <ty> <ty*>, <pointer>	从内存中读取一个值
store	<result> = store<ty> <value>,<ty*> <pointer>	将一个值写入内存
phi	<result> = phi [fast-math-falg] <ty> [<val0>,<label0>], ...	详见 SSA
br	br i1 <cond>, label <iftrue>, label <iffalse> br label <dest>	终结指令，跳转到某个位置
ret	ret <type> <value> ,ret void	return 指令

表 1. 常见 LLVM 指令

6.2.2 SSA

静态单赋值（Static Single Assignment, SSA）是编译器 LLVM 中间表示的一个重要概念，它是一种变量的命名约定。当程序中的每个变量都有且只有一个赋值语句时，称一个程序是 SSA 形式的。

在 LLVM-IR 中，每个变量在使用前必须先定义，且每个变量只能被赋值一次，所以我们称 LLVM-IR 是静态单赋值的。

例 6.2.1. 举一个例子，如果想要写一个变量定义语句，并对其赋值，我们可以如下操作

```
int main(){
    int a = 1;
    a = a + 2;
}
```

变为 llvm-ir，我们可能会认为

```
%1 = alloc i32
store i32 1, i32* %1
%1 = add i32 %1, 2
```

实际上这样写是错误的，正确的写法应该是

```
%1 = alloc i32
store i32 1, i32* %1
%2 = add i32 %1, 2
```

静态单赋值表示形式并不是没有缺点的，其也会引起一些麻烦。但可以通过使用 phi 语句来解决这个问题，比如下面的例子。

例 6.2.2.

```
int main(){
    int a = 0;
    if(a) {
        a = 2;
    }else{
        a = 4;
    }
    return a;
}
```

我们会发现，SSA 无法将一个变量的两个可能值使用同一个寄存器来保存，这个时候，我们就可以使用 phi 语句来完成，详情请参考**第7.1节**内容。具体的操作是将 if-ture 和 if-false 构建两个 block，然后在每一个 block 里面都对 a 进行赋值，并在它们最终都到达的第一个 block 里面构建一个 phi 语句。这个 phi 语句记录了若干个 pair，每一个 pair 都是一个寄存器编号（或者常量值）和 block 编号的组合，这样就可以根据是从哪一个 block 进入该 blcok 来确定要使用哪一个寄存器来保存 a 的值。

6.3 LLVM-IR 中端框架简介

6.3.1 符号表

符号表主要用于保存程序中使用的标识符的相关属性信息，用于编译器的各个工作流程。本实验给出的代码框架只涉及变量和函数的符号表。变量的符号表可以用于检验是否有未定义的使用、获取变量对应的地址、作用域的实现等功能，函数的符号表可以用于函数调用的处理。下面用一个简单示例来解释符号表的使用，需要注意的是，函数的参数和函数的内部变量属于同一个作用域。

例 6.3.1.

```
int main(){//调用enter创建一个新的作用域并进入其中
    int a;//将a变量放到当前的作用域中，并记录其地址
    a=2;//查询符号表，然后取出其地址进行对应的操作
    {//调用enter创建一个新的作用域并进入其中
        int a;//将a变量放到当前的作用域中，并记录其地址
        a=1;//查询符号表,按照从内到外的顺序查找
    }//调用exit，退出当前作用域
    return a;
}
```

6.3.2 Basic Block

Basic Bloc(BB) 简称基本块，是编译优化分析中的一个基本的概念。基本块中包含基本的指令序列，并且是以跳转语句，分支语句，返回语句结尾。通过基本块的最后一条语句，可以构建出一个图结构，这个图结构能够反映代码可能的执行流程。比如对于一段典型的 if 语句代码段，我们可以得到对应的基本块和他们之间的连接关系。

例 6.3.2.

```
1 int main()
2 {
3     int a=1;
4     if(a==1)
5         return 1
6     else
7         return 0;
8     return 0;
9 }
```

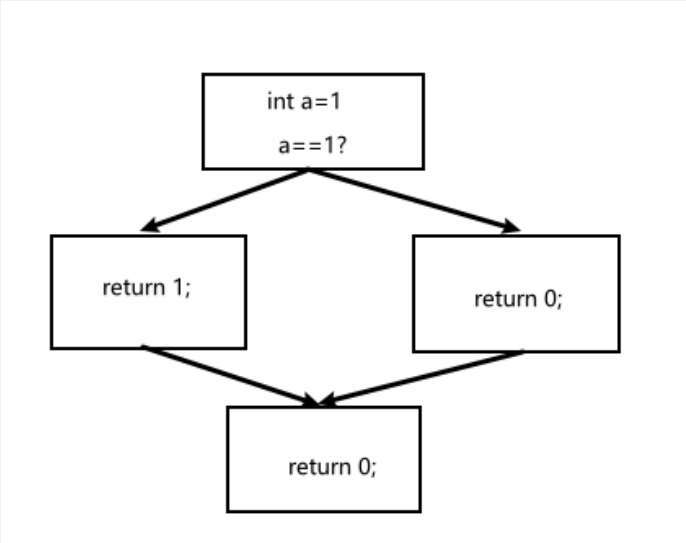

图 3. 代码对应的 BB

通过构建基本块之间的连接关系可以得到代码执行时的一些跳转情况，这个图叫做控制流图（CFG）。控制流图可以告诉我们很多信息，比如根据特别的结构来识别出 if，while 这些模块，通过基本块之间的连接关系计算出哪些基本块是必经基本块（从程序开始到目标基本块过程中必须经过的基本块）。这些信息的收集能够用于后续对 IR 的优化，比如通过 CFG 得到基本块之间的必经关系，从而决定哪些基本块中需要插入 phi 指令；以及根据基本块是否可达来删除一些不可达的指令。

6.3.3 对中间语言其他元素的建模

在翻译时，我们将每一个 *.c 文件翻译后的信息保存在一个数据结构 ir_moudle 里面。考虑一个中间语言 *.ll 文件，我们发现里面的内容其实只有两类（根据文法可以很容易看出），即全局变量和函数。首先，我们需要一个表来保存某个作用域内的变量，这个表的信息放在类 ir_scope 里面。然后，所有的函数将被分为两个类，分别是库函数和自定义函数，通过新建两个类（ir_userfunc, ir_libfunc）来区分它们。函数里面包括局部变量，因此需要将 ir_scope 作为 ir_userfunc 的成员。对于每一个变量，使用一个新的类 ir_memobj 来保存它们的信息，包括变量的标识符，变量所占用内存的多少等信息。

观察前面章节列出的中间语言代码可以得知，每一个函数里面包括若干 BasicBlock，每一个 BasicBlock 里面包含若干条中间语言指令。因此，我们分别使用 ir_basicblock 和 ir_instr 两个类来对其进行建模。对于不同类型的中间语言指令，使用 ir_instr 派生出不同的类来实现。

对于不同类型的中间语言，可能存在若干不同的参数，我们将这些参数分成两类，第一类是常量，第二类是寄存器，分别使用 ir_constant 和 ir_reg 两个类进行实现。

6.3.4 抽象语法树遍历

本框架使用访问者模式（Visitor Pattern）对抽象语法树进行遍历，访问者模式常用于对某个数据结构进行遍历，如果在遍历数据结构的同时，对某个节点进行一些额外的操作，就可以使用这种设计模式。访问者模式的存在，可以使代码结构清晰以及方便维护，同时也将不同的功能区分开来。比如对于抽象语法树的每一个节点，为了将对应的信息转变为中间语言，可以使用一个访问者类的方法进行处理。此外，为了对树中的信息进行打印，可以新建一个访问者类，重写访问每一个节点的代码。综上所述，本实验实际上只需要同学们改写访问者类 irbuilder 中的接口即可。

6.4 短路求值

对于逻辑处理语句（包含 && 和 ||），比如如下的示例。我们要求这个句子在不满足第一个条件之后，才能进行第二个条件的判断。然而第二个条件会执行 b++，如果我们使用短路求值的策略，那么当第一个条件满足时，就不会执行第二个条件，从而导致 b 的值不会改变；而当我们不使用短路求值策略的时候，不管第一个语句是否为真都会执行第二个语句，也即改变 b 的值。

例 6.4.1.

```
1  if(c == d || a == b++ || a == c)
```

那么，该如何处理短路求值呢？在给出的代码框架处理这个操作变得很简单，因为不同的逻辑处理语句分别在不同的 block 中，它们被不同的 block 隔开，我们只需要控制好 block 之间的关系即可。那么，可以自然联想到下面的 block 构建方式。

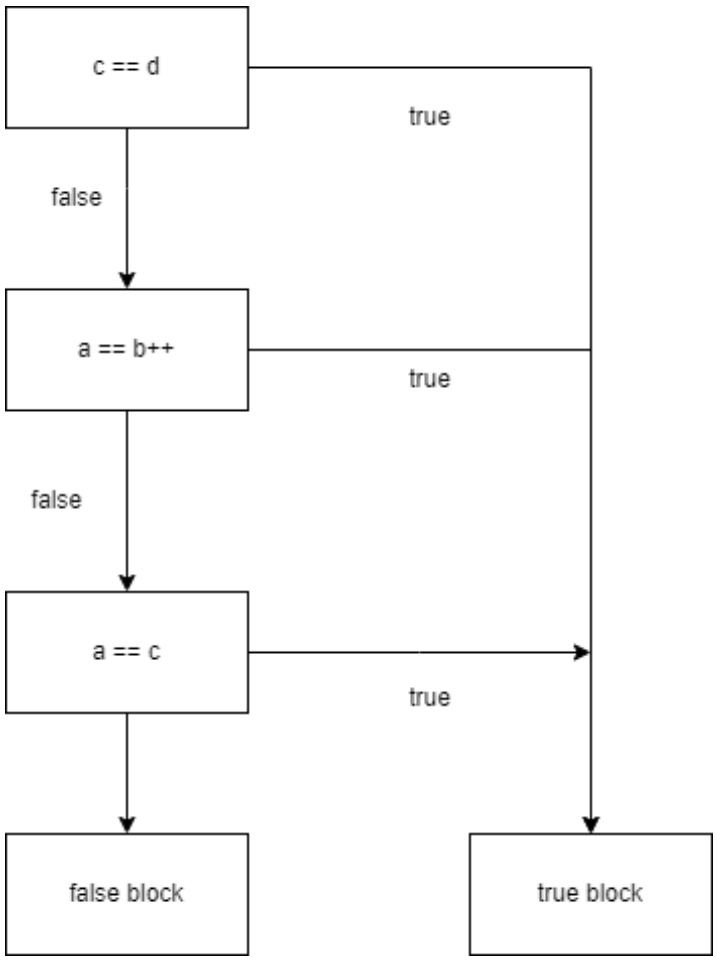

图 4. block 之间连接关系

综上，我们就可以根据第一个条件的执行结果，判断第二个条件的中间代码是否生成，也即解决了短路求值的问题。

7 中间代码优化

中间代码优化是编译器的核心和重点，是生成高效率代码的关键。本节以 mem2reg(提升内存变量为寄存器变量) 优化为例，讲解如何实现代码优化。

在 LLVM-IR 中含有若干需要访问内存的变量，如 alloc，store，load。频繁访问内存会导致代码性能下降，但如果将这些操作放入寄存器中进行存取，将大大提高相关操作的效率。所以，我们期望某些操作能够在寄存器内完成，也即将某些内存操作转移到寄存器中进行完成。mem2reg 的一般过程为

- 插入 phi 函数
- 变量重命名
- 删除冗余指令

7.1 插入 phi 指令

当一个内存变量跨越多个基本块时，可能会出现内存变量的值不确定的情况，如对于以下的 C 语言程序而言

```
1  int main {
2      int cond;
3      int x;
4      cond = 1;
5      if(cond > 0)
6          x = 1;
7      else
8          x = -1;
9      return x;
10 }
```

其对应没有添加 Phi 指令的 LLVM-IR 如下所示：

```
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  5:
9      store i32 1, i32* %2
10     br label %6
11 6:
12     %7 = load i32, i32* %2          ;%7中的内容有可能是1，也可能是 %9 的内容，此
           时有多个定义，基于栈的到达定义分析无法处理
```

```
    ret i32 %7
8:
    %9 = sub i32 0, 1
    store i32 %9, i32* %2
    br label %6
}
```

通过引入一个 phi 函数来处理这个问题，下面是一个关于插入位置的例子。

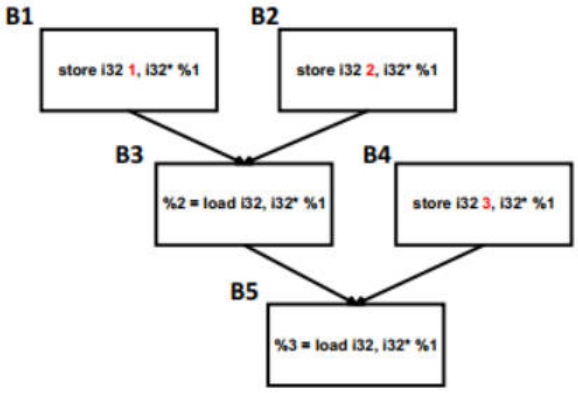

图 5. Introduce phi

我们通过在被定义变量的支配边界来插入这条 phi 指令（也就是两个对同一个变量定义的交汇点），如上图中 B1，B2 的支配边界都是 B3，B4 的支配边界是 B5。如上所述，可以发现 B3 中对于地址为%1 处的内容有可能来自于 B1，也有可能来自于 B2。此时需要插入一条 phi 指令来确定是从哪一个基本块引出的。

插入的 phi 指令如下所示：

```
%4 = phi [?,?] [?,?]
```

由于在这个阶段并不清楚 phi 指令的来源，合理的想法是先置为‘?’，之后再进行回填。类似的，我们也需要在 B5 中插入一条 phi 指令（B5 涉及到一条 load，该指令使用了一个不在 B5 中定义的变量）。下一步是变量重命名的工作。

7.2 变量重命名

变量重命名主要分为以下几步：

- 使用深度优先的方式遍历程序流图 (CFG)，对于 alloca 指令建立相关变量的栈；对于 store 指令，将需要 store 的值入栈，对于 load 指令，使用栈顶元素替换目标寄存器的所有使用；对于 phi 函数，将 phi 函数的目的寄存器入栈；
- 回填 phi 函数参数；

对于以下已经填充完 phi 函数的代码（在上面例子中的 bb6 中添加了一条 phi 指令，以确定%7 中的数据来源），具体的代码如下所示：

```
define dso_local i32 @main(){
    %1 = alloca i32
    %2 = alloca i32
    store i32 1, i32* %1
    %3 = load i32, i32* %1
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, lable %8
5:
    store i32 1, i32* %2
    br label %6
6:
    %10 = phi [??,??],[??,??]
    %7 = load i32, i32* %2
    ret i32 %7
8:
    %9 = sub i32 0,1
    store i32 %9, i32* %2
    br label %6
}
```

对程序流程图按深度优先进行遍历，遍历顺序为 entry→5→6→8，处理步骤为：

1. 对基本块 entry 进行操作：每扫描到一个 alloc，就对这个这个变量使用一个栈来保存可能的值，比如，对于前两条指令，分别创建两个栈。当遇到第一个 store 指令时，我们把立即数 1 加入第一个变量的栈顶，当遇见第一个 load 指令时，我们把所有对于%1 的使用都替换为栈顶元素 1。后继块 5 和 8 中没有 phi 指令，无需回填。

2. 对基本块 5 进行操作：遇到一个 store 指令，将 1 加入%2 的栈顶。后继块 6 中有%2 的 phi 函数，需要进行函数回填。

3. 对基本块 6 进行操作：发现 phi 指令，将 phi 指令目的寄存器%10 加入%2 的栈顶。扫描到%7 = load i32,i32* %2，使用%10 替换后续所有%7 的所有使用。

4. 对基本块 8 进行扫描，仅仅扫描到 store i32 %9, i32 * %2，将%9 加入%2 的栈顶，后继基本块 6 有%2 的 phi 函数，进行回填。

通过删除所有的 store，load，alloc 指令，得到的结果如下所示：

```
define dso_local i32 @main(){
    %4 = icmp sgt i32 1, 0
    br i1 %4, label %5, lable %8
%bb5:
    br label %6
%bb6:
    %10 = phi [1,%bb5],[%9,%bb8]
    ret i32 %10
```

```
%bb8:
    %9 = sub i32 0,1
    br label %6
}
```

至此，以上示例展示 mem2reg 优化的理论，请读者根据实验阅读材料中的示例和理论对该优化进行编码实现。

8 基于龙芯的编译器后端

本节介绍如何根据具体的体系结构，生成相应的汇编代码。首先，简单介绍龙芯体系结构及其指令集。之后，展示针对龙芯体系结构的代码生成流程及说明。最后，给出一个简单的 main 函数翻译示例。

8.1 龙芯 LA64 指令集简介

2020 年，龙芯中科公司基于二十年的 CPU 研制和生态积累推出了龙架构（LoongArch™），包括基础架构部分和向量指令、虚拟化、二进制翻译等扩展部分，近 2000 条指令。龙架构具有较好的自主性、先进性与兼容性。其中，LA64 是该指令集的 64 位版本。

首先，我们介绍龙芯架构应用程序二进制接口 API，主要包括：

- 数据的表示和对齐
- 栈布局
- 简单代码示例

其中，基本数据类型如**表 2**所示，通用寄存器约定如**图 6**所示。

表 2. 基本数据类型以及对齐

类型	大小	对齐
int	4	4
float	4	4
double	8	8

寄存器号	别名	Saver
r0	zero	-
r1	ra	caller
r2	tp	-
r3	sp	callee
r4-r11	a0-a7 v0/v1=a0/a1	caller
r12-r20	t0-t8	caller
r21	reserved	-
r22	fp	callee
r23-r31	s0-s8	callee

可用于寄存器分配参数：a0~a7、t0~t8

图 6. Enter Caption[1]

[1]caller: 调用者（指调用函数的那个函数）负责寄存器的保存和恢复工作。callee: 被调用者（指被父函数调用的子函数）负责寄存器的保存和恢复工作。

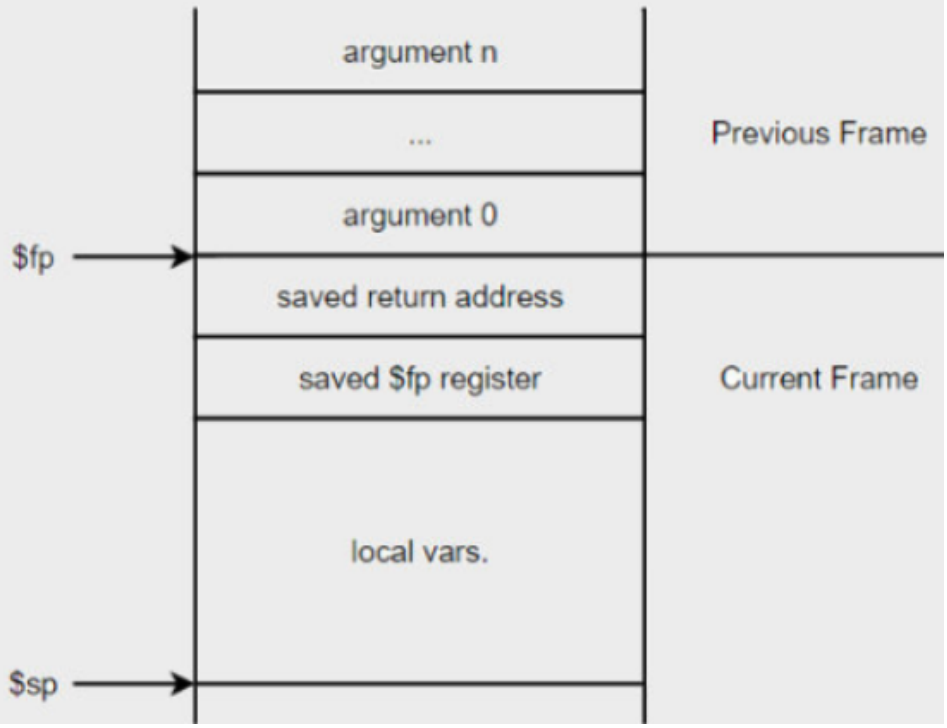

图 7. 函数调用栈帧布局

关于龙芯的具体指令，可以查看课程给出的其他资料。下面介绍函数调用时的栈帧布局：

- 整个函数中，堆栈从高地址增长到低地址。
- ABI 使用栈帧指针 (fp)，和栈顶指针 (sp) 访问堆栈。
- 函数的栈帧大小需要 16 字节对齐。

下面给出一个简单的函数调用代码示例帮助理解（汇编代码参考 loongarch64 gcc 13.2.0，方便理解进行了一定程度的修改和简化，若想了解编译器实际生成的代码，可以参考在线编译器网站 https://godbolt.org/）

```
1  int add(int a,int b) {
2      return a + b;
3  }
4
5  int main(){
6      int a, b, c;
7      a = 1, b = 2;
8      c = add(a,b);
9      return c;
10 }
```

对应的汇编代码如下所示：

```
1  add(int, int):
```

```
2     addi.d $sp,$sp,-32          # 先分配空间
3     st.d   $fp,$sp,24           # 先保存 fp 的值
4     addi.d $fp,$sp,32           # 确定新的 fp 的值
5     or     $t0,$a0,$zero        # 取参数的值
6     or     $t1,$a1,$zero        # 取参数的值
7     st.w   $t0,$fp,-20          # 把 a 的值放到内存的某个位置
8     or     $t0,$t1,$zero        # 把 b 的值放到 t0 这个寄存器
9     st.w   $t0,$fp,-24          # 把 b 放到内存的某个位置
10    ld.w   $t1,$fp,-20          # 加载 a 的值到寄存器
11    ld.w   $t0,$fp,-24          # 加载 b 的值到寄存器
12    add.w  $t0,$t1,$t0          # 将两个值相加，然后存到 t0 寄存器
13                                # 通过上面的操作，我们可看出没有经过优化的编译器
                                      是非常“傻”的
14    or     $a0,$t0,$zero        # 将值放到返回值寄存器
15    ld.d   $fp,$sp,24           # 经典操作，恢复 fp 的值
16    addi.d $sp,$sp,32           # 经典操作，恢复 sp 的值
17    jr     $r1                  # 返回到调用指令的下一条
18 main:
19    addi.d $sp,$sp,-32          # 进入函数第一步，分配空间，确定新的 $sp 位置
20    st.d   $zero,$sp,24         # 将父函数的地址存到指定的位置
21    st.d   $fp,$sp,16           # 将旧的 fp 的值保存起来
22    addi.d $fp,$sp,32           # 设置新的 fp 的值
23    addi.w $t0,$zero,1          # 设置 a = 1
24    st.w   $t0,$fp,-20          # 将 a 存入内存
25    addi.w $t0,$zero,2          # 设置 b = 2
26    st.w   $t0,$fp,-24          # 将 b 存入内存
27
28    ldptr.w $t1,$fp,-24         # 理解为可扩展符号位的 ld.w 指令，将内存中的东
          西取出来放到寄存器里面，这里取出 b
29    ldptr.w $t0,$fp,-20         # 这里取出 a
30    or     $a1,$t1,$zero        # 这里编译器用了 or 来进行寄存器之间值的移动，
          移动 b 的值到参数寄存器
31    or     $a0,$t0,$zero        # 移动 a 的值到参数寄存器
32    bl     add(int, int)        # 调用函数 add
33    or     $t0,$a0,$zero        # 移动返回值到临时寄存器 t0
34    st.w   $t0,$fp,-28          # 存一个字到内存指定的位置，也就是变量 c
35    or     $t0,$zero,$zero      # 重置 临时寄存器 t0 的值
36    or     $a0,$t0,$zero        # 重置 临时寄存器 a0 的值
37    ld.d   $ra,$sp,24           # 将保存的父函数的地址取出来
38    ld.d   $fp,$sp,16           # 将旧的 fp 的值取出来
39    addi.d $sp,$sp,32           # 将旧的 sp 的值取出来
```

```
40      jr      $ra                     # 用来进行函数返回
```

对应的栈帧布局如下所示：

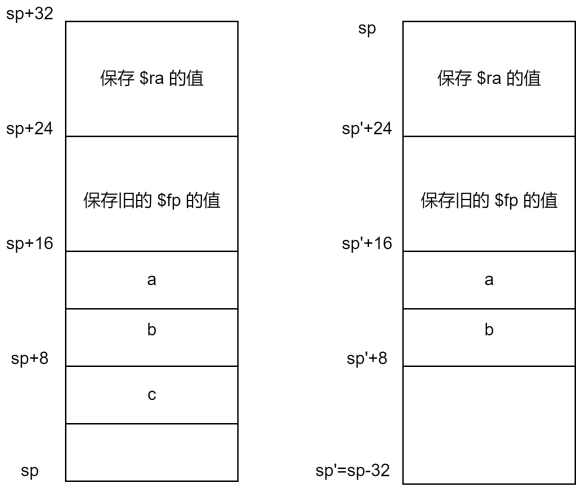

图 8. 函数调用栈帧布局

由上面的内容，我们可以得知，在生成汇编代码的部分，需要保存每一个变量和这个变量在内存中相应的偏移量对应关系。

8.2 代码生成

我们将代码生成分为三个方面的问题，分别是：指令选择，指令调度，寄存器分配。

- 指令选择：决定生成的汇编代码能否在物理机上正确运行。
- 指令调度：决定生成的汇编代码能否根据流水线高效率运行。
- 寄存器分配：高效运用寄存器，提高程序的执行效率。

在本节中我们将重点讲解指令选择，这是一个编译器后端的基本。不能正确执行指令，所谓的编译器就无从谈起。指令选择只需要把中间语言一句一句的翻译成对应的一条或者几条汇编语言即可。在框架中，我们依然使用访问者模式对生成的中间语言进行遍历。

8.2.1 指令翻译模板

- 访存保存指令翻译模板

```
1       store i32 1, i32* %op0
```

这条指令将一个常量保存在某个变量上

```
1       addi.d   $t0, $zero, 1
2       st.w     $t0, $sp, offset_op0
```

在汇编层面，我们可以先把立即数 1 保存到一个寄存器里面，然后再把这个寄存器的内容放到内存的某个位置。

- 访存读取指令翻译模板

```
%op1 = load i32, i32 %op0
```

这条指令将内存某个位置中的数据读取出来，对应的汇编代码如下所示：

```
ld.w    $t0, $sp, offset_op0
st.w    $t0, $sp, offset_op1
```

- 跳转指令模板

```
br i1 %op3, label %label0, label %label1
```

跳转指令对应的汇编代码如下所示：

```
ld.w    $t0, $fp, offset_op3
beqz    $t0, .main_label1
b       .main_label0
```

更多中间语言的翻译方法，同学们可以参考 https://godbolt.org/ 网站，这里由于篇幅限制就不多做介绍了。

8.2.2 寄存器分配

通过上面的介绍，我们在翻译程序的时候实际上较少使用寄存器，如上面一节提到的指令

```
ld.w    $t0, $fp, offset_op3
beqz    $t0, .main_label1
b       .main_label0
```

实际上只使用了一个寄存器，而除了几个关键的寄存器（比如保存返回值，保存参数）的寄存器之外，其他的操作可以使用有限个寄存器和内存的交互来完成。

实际上，对于形似下面伪码的语句

```
a = f(b1,b2, ... )
```

其中 f 可以是函数调用，也可以是从内存中读取一个值，ld 表示从内存中读取，st 表示将寄存器的值存入内存中，我们可以得到如下形式语句。

```
r1 = ld a_b1
r2 = ld a_b2
...
r1 = f(r1,r2, ... )
st r1, a_y
```

发现在这个过程只使用和参数个数相同的有限寄存器。对于形似下面伪码的语句（描述了一个条件跳转过程）

```
if( f(x,y) ) l1 else l2
```

我们可以使用如下的伪码进行寄存器和内存之间的交互

```
r1 = ld a_x
r2 = ld a_y
r1 = f(r1,r2)
if(r1) l1 else l2
```

这个方法简单粗暴，由于增加很多次的访存操作，而且很多寄存器没有充分的利用，生成程序的速度会大幅慢于专用的寄存器分配算法（图着色算法，线性扫描）。由于课程实验时间等限制，同学们可以使用这个较为简单的方法完成任务。时间充裕，感兴趣的同学可以自行查找相关资料，完善框架中提供的图着色算法。

8.3 程序翻译示例

图 9. 简单程序 LA 汇编举例

更多程序，同学们可以前往https://godbolt.org/ ，选择 loongarch 指令集进行查看。

9 致谢

本实验体系由龙芯中科公司的星火计划提供经费支持，刘彬彬负责项目执行，中国科技大学编译原理课程组提供技术支持。在项目的执行过程中，获得了合肥工业大学编译原理课程组 (李宏芒，唐益明，凤维杰，蒋哲远) 和中国科技大学编译原理课程组 (张昱，郑启龙，李诚，徐伟) 的大力支持，得到了多位同学 (肖同欢，曹晨曦，牟长青，邓杰湧，贺嘉，陈清源) 的帮忙。最后，再次感谢合肥工业大学、龙芯中科公司、中国科技大学以及各位老师和同学。