编译实践报告

2000013138 信息科学技术学院 黄哲涛

前言

此报告讲述了我的 2022 年秋季学期编译实践的完成情况以及实现过程。

我的实践主要参考了北大编译实践在线文档中的指导。

我的实践环境是在 Windows 10 家庭中文版的笔记本上配置实验环境的 Docker 镜像，在具体实践时使用了 Visual Studio Code 的 Docker 拓展来在 Docker 镜像环境中编写和管理代码。

一、编译器概述

1、基本功能

1. 将 SysY 语言编译到 Koopa IR
2. 将 Koopa IR 生成到 RISC-V 汇编

2、主要特点

我使用了C++作为编程语言进行编译器的实现，并且使用了 Flex 和 Bison 来分别生成词法分析器和语法分析器。

我在 IR 生成阶段采用了面向对象的编程方法生成一系列 AST 并在 AST 上进行处理，而在目标代码生成阶段则主要采用面向过程的编程方法。

在生成 RISC-V 目标代码后，我还做了一个简单的窥孔优化，删除冗余的 load 指令。

二、编译器设计

1、主要模块组成和代码文件结构

我的编译器主要包含四个彼此之间较为独立的模块：

1. 由 Flex 和 Bison 组成的词法分析器和语法分析器为第一个模块（sysy.l 和 sysy.y 文件）
2. 在词法/语法分析后生成的 AST 上生成 Koopa IR 为第二个模块（ast.h 文件）
3. 将 Koopa IR 生成到 RISC-V 汇编代码为第三个模块（koopa2riscv.h 文件）
4. 调用上述三个模块的内容，并对生成的汇编代码进行优化的主程序为第四个模块（main.cpp 文件）

除了以上文件，global.h 和 global.cpp 文件中则包含一些全局变量的定义和函数的声明。

2、主要数据结构

我在设计编译器时定义和使用一些数据结构来进行中间代码生成和目标代码生成。

a.中间代码生成

在中间代码生成时主要使用的数据结构有自定义的 symbol 类，symbol_table 符号表，loop_label_table 管理循环嵌套层次的基本块号，以及各 AST 类。

一个全局的 string 对象 koopa_ir 为最终产生的中间代码的内容。此外还有很多全局变量记录一些全局信息。

```
int koopa_tmp_id=0; // 变量号(%x)
int now_symbol_table_id=0; // 当前符号表
int if_tmp_id=0; // 用于分支语句的then号,else号和end号
int while_tmp_id=0; // 用于循环语句的while号
int loop_num=0; // 当前循环嵌套层数
int unreachable_tmp_id=0; // 用于不可到达的label号
int var_def_id=0; // 用于变量定义的变量号(name_x)
bool is_global_decl=0; // 是否为全局变量声明
bool func_arg_is_arr=0; // 函数参数是否为数组
```

symbol

symbol 类定义了“符号"，符号可能为一个变量或者函数。当其对应一个变量时，还记录了其变量名、变量值、是否为常量、是否已赋值、是否为指针或数组，以及数组或指针的维数；当其对应一个函数时，则记录了其函数名，是否为 void 型（无返回值）。

```
class symbol{
 public:
  std::string name; // koopa代码中的变量名(a->a_x)或函数名
  std::string val; // 此符号对应的值(一个数或者一个中间变量名)("x"或"%x")
  bool is_const; // 是否为常量
  bool is_assigned; // 是否已赋值
  bool is_func; // 此符号是否对应一个函数
  bool is_void; // 此函数是否为void型
  bool is_ptr; // 此符号是否对应一个指针
  bool is_arr; // 此符号是否对应一个数组
  int axis; // 数组或指针的维数

  symbol(bool i_f,bool i_c_or_i_v,std::string n="",std::string v="",bool
i_p=0,bool i_a=0,int ax=0){
    if (!i_f){
      is_func=0;
      is_const=i_c_or_i_v;
      name=n;
      val=v;
      if (val==""){
        val="0";
        is_assigned=0;
      }
      else {
        is_assigned=1;
      }
      is_ptr=i_p;
      is_arr=i_a;
      axis=ax;
    }
    else {
```

```
      is_func=1;
      is_void=i_c_or_i_v;
      name=n;
    }
  }
  symbol(){}
};
```

symbol_table

symbol_table为具有层次的符号表，每一层是一个作用域内的 SysY代码中的变量名（ ident ） 到 symbol 对象的字典，越低层（下标越小）则对应的作用域越靠外（比如 symbol_table[0] 对应了全局作用域）。

在代码实现中使用了 std::vector 和 std::map 这两个 c++ 标准库提供的数据结构来实现 symbol_table。

```
std::vector< std::map< std::string, symbol > > symbol_table; // 符号表
```

loop_label_table

loop_label_table 是记录嵌套循环中层次与基本块的编号。我在处理循环时使用的基本块号均为 %while_entry_x，%while_body_x，%while_end_x 这种形式，x为数字编号（由全局变量 while_tmp_id 确定），每一个循环使用一个唯一的编号（ while_tmp_id 不断增加，故不会造成基本块名称的重复 ）。

在代码实现中使用了 std::map 数据结构，key值为循环嵌套层数，value值为循环基本块的编号。

```
std::map< int, int > loop_label_table; // 当前循环嵌套层数对应的while号
```

各 AST 类

我对语法分析的文法中的所有非终结符号都定义了一个 AST 类，所有的 AST 都继承了一个名为 BaseAST 的抽象类。BaseAST 中定义了全部或一些 AST 类都包含的成员变量和方法，比如 op （采取第几个文法规则），ir_id （在中间代码中对应的变量名）等。

比如 op 成员变量代表了此 AST 对应的非终结符号采用了第几个文法（当一个非终结符号具有多个文法规则时）。

```
// 所有 AST 的基类
class BaseAST {
 public:
  int op;
  std::string ir_id;
  virtual ~BaseAST() = default;
  virtual std::string Dump(){return "";}
  virtual std::string get_opt(){return "";}
  virtual void get_rest_of_const_def_vec(std::vector< std::unique_ptr<BaseAST> >
&vec){return;};
  virtual void get_rest_of_var_def_vec(std::vector< std::unique_ptr<BaseAST> >
&vec){return;};
  virtual void get_rest_of_block_item_vec(std::vector< std::unique_ptr<BaseAST>
> &vec){return;};
  virtual void get_rest_of_global_item_vec(std::vector< std::unique_ptr<BaseAST>
> &vec){return;};
```

```
  virtual void get_rest_of_param_vec(std::vector< std::unique_ptr<BaseAST> >
&vec){return;};
  virtual void get_rest_of_exp_vec(std::vector< std::unique_ptr<BaseAST> > &vec)
{return;};
  virtual std::string get_ir_id(){return ir_id;}
  virtual std::string get_ident(){return "";}
  virtual std::string get_params(){return "";}
  virtual int get_val(){return 0;}
  virtual bool get_have_ret(){return 0;} // 是否为返回语句(如果为分支语句，则全部情况均
返回)
  virtual bool is_void(){return 0;}
  virtual void pre_alloc_store(){return;} // 函数提前将输入参数加入符号表
  virtual void get_rest_of_axis_info_vec(std::vector< std::unique_ptr<BaseAST> >
&vec){return;} // 获取数组各维信息
  virtual void get_rest_of_const_init_val_vec(std::vector<
std::unique_ptr<BaseAST> > &vec){return;}
  virtual void get_rest_of_init_val_vec(std::vector< std::unique_ptr<BaseAST> >
&vec){return;}
  virtual std::string get_rest_of_init_vec(std::vector<std::string>
&vec,std::vector<int> axis_length,int now_axis){return "";}
  virtual std::string load(std::string &loaded_id){return "";} // 若目标为指针，则
load并更新loaded_id，否则等同于get_ir_id()
  virtual std::string get_func_type(){return "";}
  virtual bool is_number(){return 0;}
};
```

除 BaseAST 外，各 AST 类彼此之间为包含关系。比如下面代码中定义的 FuncDefAST 为函数定义的 AST ，其包含的 func_type 为一个指向 FuncTypeAST 的指针，func_f_params 和 block 同理。在运行过程中一个 AST 对象内部的 AST 指针可能不会同时有意义，这取决于其对应的非终结符号采用的文法规则推导出了哪些非终结符号。

所有 AST 都有一个 Dump 方法，此方法在用于修改 koopa_ir 以生成中间代码，以及进行一些全局信息的修改，比如修改符号表和其他全局变量等。有些 AST 不会在 Dump 方法中直接修改 koopa_ir ，而是返回一个保存了生成指令的字符串，由上层的 AST 决定在什么地方插入这些指令，或者不插入（比如变量的定义，由于全局作用域内不应该出现指令，所以下层的 AST 不应直接修改 koopa_ir，而是由上层 AST 先检查是否为全局变量定义，再决定koopa_ir的修改）。

```
class FuncDefAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> func_type;
  std::string ident;
  std::unique_ptr<BaseAST> func_f_params;
  std::unique_ptr<BaseAST> block;

  std::string Dump(){
    bool is_void=func_type->is_void();
    symbol_table[now_symbol_table_id][ident]=symbol(1,is_void,ident);
    koopa_ir+="fun @";
    koopa_ir+=ident;
    if (op==1){
      koopa_ir+="()";
    }
    else if (op==2){
      koopa_ir+="(";
```

```
      func_f_params->Dump();
      koopa_ir+=")";
    }
    func_type->Dump();
    koopa_ir+=" {\n";
    koopa_ir+="%entry:\n";

    now_symbol_table_id++;
    symbol_table.push_back(std::map< std::string, symbol >());

    // 将函数参数提前alloc和store
    if (op==2){
      func_f_params->pre_alloc_store();
    }
    block->Dump();

    symbol_table.pop_back();
    now_symbol_table_id--;

    if (!block->get_have_ret()){
      if (func_type->get_func_type()=="int"){
        koopa_ir+="  ret 0\n";
      }
      else {
        koopa_ir+="  ret\n";
      }
    }
    koopa_ir+="}\n";
    return "";
  }
};
```

b.目标代码生成

在目标代码生成时主要使用的数据结构有 ins2var 为记录指令到其对应的变量名的字典，和 stack_dic 记录变量名在栈上的偏移量的字典。

一个全局的 string 对象 riscv_code 为最终产生的目标代码的内容。此外还有很多全局变量记录一些全局信息。

```
string reg_name[15]=
{"t0","t1","t2","t3","t4","t5","t6","a0","a1","a2","a3","a4","a5","a6","a7"};
int stack_offset=0; // 已使用的栈偏移量
int reg_cnt=0; // 已使用的寄存器数量
int ins_cnt=0; // 保存在栈上的指令返回值数量
int var_cnt=0; // 保存在栈上的已分配变量数量
int stack_size=0; // 开辟的栈空间
bool if_restore_ra=0; // 是否需要恢复ra寄存器
```

ins2var

ins2var 保存了从指令到其产生的变量名称的映射。

在代码实现中使用了 std::map 数据结构来实现 ins2var。

```
std::map< koopa_raw_value_t ,std::string> ins2var; // alloc指令分配的变量名对应为"@n"，指令返回值对应的临时变量名对应为"%n"
```

stack_dic

stack_dic 保存了从变量名称到其在栈上的偏移量的映射。

在代码实现中使用了 std::map 数据结构来实现 stack_dic。

```
std::map< std::string, int > stack_dic; // 栈上的内存
```

3、主要算法设计考虑

在生成 Koopa IR 阶段，主要算法设计是在面向对象的基础上，从抽象语法树的根节点开始自顶向下遍历，并最终回到根节点。父节点的 AST 可以询问获得子节点的 AST 的属性。一些不便于在树上传递的全局信息则保存为全局变量。在遍历一个节点时，处理该节点对应 AST 的信息，并直接将指令加入 koopa_ir 或将其字符串返回给父节点，由父节点选择时机加入 koopa_ir 。

而在生成 RISC-V 阶段，其实经过处理的内存形式的 raw program 也是类似的树状结构，故也可以遍历此树，在每个节点处生成对应的 RISC-V 指令加入r iscv_code 。

三、编译器实现

1、对所涉工具软件的介绍

我在词法/语法分析阶段使用了 Flex 和 Bison 工具软件，使用 Flex 通过正则表达式定义终结符（token），以及使用 Bison 根据定义的文法生成 parser 。此外，我还使用了提供的 libkoopa 中的接口将文本形式的 Koopa IR 转换为内存形式的 raw program 来表示 Koopa IR 的数据结构，以将 Koopa IR 生成到 RISC-V 汇编代码。

在这一部分，我先讲述一下我在词法/语法分析阶段使用 Flex 和 Bison 工具进行的工作，因为我认为这一模块的工作独立性较强，且没有必要分阶段展示（各阶段的词法/语法分析所做的编码工作大同小异，并且在后面的阶段对之前定义的文法进行了不少修改）。

词法分析（ Flex ）

词法分析部分将特定的字符串定义为 token 。

```
"int"           { return INT; }
"void"          { return VOID; }
"return"        { return RETURN; }
"const"         { return CONST; }
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
"break"         { return BREAK; }
"continue"      { return CONTINUE; }

"<="            { yylval.str_val = new string(yytext); return LEq; }
">="            { yylval.str_val = new string(yytext); return GEq; }
"=="            { yylval.str_val = new string(yytext); return Eq; }
"!="            { yylval.str_val = new string(yytext); return NEq; }
"&&"            { yylval.str_val = new string(yytext); return LAnd; }
"||"            { yylval.str_val = new string(yytext); return LOr; }
```

唯一需要注意的是使用正则表达式来定义块注释。

```
BlockComment  \/\*(\n|[^*]|\*+[^*/])*\*+\/
```

语法分析（Bison）

词法分析部分，首先声明 token 和非终结符的定义。

```
// lexer 返回的所有 token 种类的声明
%token INT VOID RETURN CONST IF ELSE  WHILE BREAK CONTINUE
%token <str_val> IDENT LEq GEq Eq NEq LAnd LOr
%token <int_val> INT_CONST

// 非终结符的类型定义
%type <ast_val> GlobalItem RestOfGlobalItem Block BlockItem RestOfBlockItem Stmt
OpenStmt ClosedStmt SimpleStmt
%type <ast_val> FuncDef FuncType FuncFParams FuncFParam RestOfFuncFParam
FuncRParams RestOfFuncRParam
%type <ast_val> Exp UnaryExp PrimaryExp UnaryOp AddExp MulExp LOrExp LAndExp
EqExp RelExp ConstExp
%type <ast_val> Decl ConstDecl VarDecl BType ConstDef VarDef RestOfConstDef
RestOfVarDef ConstInitVal RestOfConstInitVal InitVal RestOfInitVal LVal Axis
%type <int_val> Number
```

然后是定义文法规则并生成 AST 。除了在生成 AST 时（对 AST 中的内容进行初始化）可能会调用 AST 的成员函数以外，这部分的编码内容过于冗长且重复性强，这里只讨论一种特殊的文法规则的形式。

形如 `FuncFParams  ::= FuncFParam {"," FuncFParam};` 这样的文法，由于其中一部分内容可以出现多次，故需要递归定义。对于这样的非终结符，我的处理方法是定义一个额外的中间非终结符来处理递归定义。比如在这个例子中我定义的中间非终结符名为 RestOfFuncFParam 。

```
FuncFParams
  : FuncFParam {
    auto ast = new FuncFParamsAST();
    ast->param.push_back(unique_ptr<BaseAST>($1));
    $$ = ast;
  }
  | FuncFParam RestOfFuncFParam {
    auto ast = new FuncFParamsAST();
    ast->first_param = unique_ptr<BaseAST>($1);
    ast->rest_of_param = unique_ptr<BaseAST>($2);
    ast->get_param_vec();
    $$ = ast;
  }
  ;

RestOfFuncFParam
  : ',' FuncFParam {
    auto ast = new RestOfFuncFParamAST();
    ast->rest_of_param = NULL;
    ast->param = unique_ptr<BaseAST>($2);
    $$ = ast;
  }
  | ',' FuncFParam RestOfFuncFParam {
```

```
    auto ast = new RestOfFuncFParamAST();
    ast->param = unique_ptr<BaseAST>($2);
    ast->rest_of_param = unique_ptr<BaseAST>($3);
    $$ = ast;
  }
  ;
```

此外，我采用了https://en.wikipedia.org/wiki/Dangling_else的文法，定义额外的 OpenStmt，ClosedStmt，SimpleStmt 非终结符来处理空悬 else 问题。

```
Stmt
  : OpenStmt {
    auto ast = new StmtAST();
    ast->op = 1;
    ast->open_stmt = unique_ptr<BaseAST>($1);
    $$ = ast;
  }
  | ClosedStmt {
    auto ast = new StmtAST();
    ast->op = 2;
    ast->closed_stmt = unique_ptr<BaseAST>($1);
    $$ = ast;
  }
  ;

OpenStmt
  : IF '(' Exp ')' Stmt {
    auto ast = new OpenStmtAST();
    ast->op = 1;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->stmt = unique_ptr<BaseAST>($5);
    $$ = ast;
  }
  | IF '(' Exp ')' ClosedStmt ELSE OpenStmt {
    auto ast = new OpenStmtAST();
    ast->op = 2;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->if_stmt = unique_ptr<BaseAST>($5);
    ast->else_stmt = unique_ptr<BaseAST>($7);
    $$ = ast;
  }
  | WHILE '(' Exp ')' OpenStmt {
    auto ast = new OpenStmtAST();
    ast->op = 3;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->open_stmt = unique_ptr<BaseAST>($5);
    $$ = ast;
  }
  ;

ClosedStmt
  : SimpleStmt {
    auto ast = new ClosedStmtAST();
    ast->op = 1;
    ast->simple_stmt = unique_ptr<BaseAST>($1);
```

```
    $$ = ast;
  }
  | IF '(' Exp ')' ClosedStmt ELSE ClosedStmt {
    auto ast = new ClosedStmtAST();
    ast->op = 2;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->if_stmt = unique_ptr<BaseAST>($5);
    ast->else_stmt = unique_ptr<BaseAST>($7);
    $$ = ast;
  }
  | WHILE '(' Exp ')' ClosedStmt {
    auto ast = new ClosedStmtAST();
    ast->op = 3;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->closed_stmt = unique_ptr<BaseAST>($5);
    $$ = ast;
  }
  ;

SimpleStmt
  : LVal '=' Exp ';' {
    auto ast = new SimpleStmtAST();
    ast->op = 1;
    ast->lval = unique_ptr<BaseAST>($1);
    ast->exp = unique_ptr<BaseAST>($3);
    $$ = ast;
  }
  | ';' {
    auto ast=new SimpleStmtAST();
    ast->op = 2;
    $$ = ast;
  }
  | Exp ';' {
    auto ast = new SimpleStmtAST();
    ast->op = 3;
    ast->exp = unique_ptr<BaseAST>($1);
    $$ = ast;
  }
  | Block {
    auto ast = new SimpleStmtAST();
    ast->op = 4;
    ast->block = unique_ptr<BaseAST>($1);
    $$ = ast;
  }
  | RETURN ';' {
    auto ast = new SimpleStmtAST();
    ast->op = 5;
    $$ = ast;
  }
  | RETURN Exp ';' {
    auto ast = new SimpleStmtAST();
    ast->op = 6;
    ast->exp = unique_ptr<BaseAST>($2);
    $$ = ast;
  }
```

```
  | BREAK ';' {
    auto ast = new SimpleStmtAST();
    ast->op = 7;
    $$ = ast;
  }
  | CONTINUE ';'{
    auto ast = new SimpleStmtAST();
    ast->op = 8;
    $$ = ast;
  }
  ;
```

在我完成Lv8时，在语法分析阶段仍然出现了运行时错误的问题。我猜测是因为我定义的文法不符合 LALR 文法而出现了冲突，但我不知道是什么地方出现了问题，因此无奈之下，我在 sysy.y 中加入以下代码，使其生成一个GLR parser，而不是LALR parser。此后可以正常解析并生成 AST 。

```
%glr-parser
```

2、各个阶段的编码

下面将讲述我的编译器的c++部分代码的实现，主要说明一下各阶段的程序设计部分以及我认为的编码难点，必要时会结合具体代码来辅助说明。

由于撰写此报告时编译器已经完成，且在实现过程中后面的阶段对前面的阶段进行了规模不一的代码重构，故涉及到具体代码时，我将结合最终版本的代码。

Lv1 & Lv2：main函数

这一阶段内容不多，跟着在线文档一步一步做即可。

但当时在做这部分的时候其实不是很容易，主要是初见项目流程，需要熟悉使用的各种工具。

不过这一阶段的算法设计部分确实没有什么内容可讲，这里直接略过了。

Lv3：表达式

这一阶段需要处理各种各样的表达式。

中间代码生成

表达式从文法层面上通过分层来实现了不同运算的优先级，但在代码实现时不同层的实现逻辑是类似的。这里以 MulExp 为例：`MulExp ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;`

这里主要讲讲我的 MulExpAST 的 Dump 方法，其他方法主要是为后续阶段服务的。当 MulExp 采用第一个文法规则时，调用其对应的 UnaryExp 的 Dump 方法，获取其生成的指令的字符串，保存到 tmp_str，并更新自己的 ir_id；采用第二个文法规则时，先依次调用 MulExp 和 UnaryExp 的 Dump 方法，将它们生成的指令的字符串保存到 tmp_str，调用它们的 load 方法获取他们对应的变量名（load 方法可以看作兼容了 Lv9 的数组和指针的 get_ir_id 方法，当对象是数组或指针时会加入对应的 load 指令），然后再将此 MulExp 自己的指令加入 tmp_str。最后再将tmp_str 返回给上一层 AST 。

```
class MulExpAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> mul_exp;
  std::string mul_op;
  std::unique_ptr<BaseAST> unary_exp;
```

```
std::string Dump(){
  std::string tmp_str="";
  if (op==1){
    tmp_str=unary_exp->Dump();
    ir_id=unary_exp->get_ir_id();
  }
  else if (op==2){
    tmp_str+=mul_exp->Dump();
    tmp_str+=unary_exp->Dump();
    std::string unary_id="",mul_id="";
    tmp_str+=mul_exp->load(mul_id);
    tmp_str+=unary_exp->load(unary_id);

    ir_id="%"+std::to_string(koopa_tmp_id);
    koopa_tmp_id++;

    if (mul_op=="*"){
      tmp_str+="  "+ir_id+" = mul "+mul_id+", "+unary_id+"\n";
    }
    else if (mul_op=="/"){
      tmp_str+="  "+ir_id+" = div "+mul_id+", "+unary_id+"\n";
    }
    else if (mul_op=="%"){
      tmp_str+="  "+ir_id+" = mod "+mul_id+", "+unary_id+"\n";
    }
  }
  return tmp_str;
}

int get_val(){
  if (op==1){
    return unary_exp->get_val();
  }
  else if (op==2){
    if (mul_op=="*"){
      return (mul_exp->get_val())*(unary_exp->get_val());
    }
    else if (mul_op=="/"){
      return (mul_exp->get_val())/(unary_exp->get_val());
    }
    else if (mul_op=="%"){
      return (mul_exp->get_val())%(unary_exp->get_val());
    }
  }
  return 0;
}

std::string load(std::string &loaded_id){
  std::string tmp_str="";
  if (op==1){
    tmp_str=unary_exp->load(loaded_id);
  }
  else if (op==2){
    loaded_id=ir_id;
```

```
    }
    return tmp_str;
  }

  bool is_number(){
    if (op==1){
      return unary_exp->is_number();
    }
    else if (op==2){
      return mul_exp->is_number()&&unary_exp->is_number();
    }
    return 0;
  }
};
```

我的其他 AST 的 Dump 方法的代码逻辑都与此相似，父结点调用子结点的 Dump 方法生成子结点的指令，然后组织自己的指令。叶子结点则直接生成自己的指令。

目标代码生成

这一阶段的目标代码生成部分主要也是实现各种各样的表达式，即处理各种各样的 binary 结构生成的汇编代码。

我写了一个 Visit_binary 函数来完成这一功能，其代码逻辑也非常简单，根据 binary 结构的运算类型分别生成对应的汇编代码即可。

```
int Visit_binary(const koopa_raw_binary_t &binary){
  koopa_raw_binary_op_t op = binary.op;

  string l,r;
  int dst=reg_cnt;
  string now_reg=reg_name[reg_cnt];
  reg_cnt++;

  switch (op) {
    case KOOPA_RBO_ADD: // 加
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  add "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_SUB: // 减
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  sub "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_MUL: // 乘
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  mul "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_DIV: // 除
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  div "+now_reg+", "+l+", "+r+"\n";
      break;
```

```
    case KOOPA_RBO_MOD: // 模
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  rem "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_NOT_EQ: // 不等
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  xor "+now_reg+", "+l+", "+r+"\n";
      riscv_code+="  snez "+now_reg+", "+now_reg+"\n";
      break;
    case KOOPA_RBO_EQ: // 等于
      if (r=="x0"){
        riscv_code+="  li "+now_reg+", ";
        riscv_code+=l;
        riscv_code+="\n";
        riscv_code+="  xor "+now_reg+", "+now_reg+", x0\n";
        riscv_code+="  seqz "+now_reg+", "+now_reg+"\n";
      }
      if (l=="x0"){
        riscv_code+="  li "+now_reg+", ";
        riscv_code+=r;
        riscv_code+="\n";
        riscv_code+="  xor "+now_reg+", "+now_reg+", x0\n";
        riscv_code+="  seqz "+now_reg+", "+now_reg+"\n";
      }
      else {
        get_operand_load_reg(binary.lhs,l);
        get_operand_load_reg(binary.rhs,r);
        riscv_code+="  xor "+now_reg+", "+l+", "+r+"\n";
        riscv_code+="  seqz "+now_reg+", "+now_reg+"\n";
      }
      break;
    case KOOPA_RBO_GT: // 大于
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  slt "+now_reg+", "+r+", "+l+"\n";
      break;
    case KOOPA_RBO_LT: // 小于
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  slt "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_GE: // 大于等于
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  slt "+now_reg+", "+l+", "+r+"\n";
      riscv_code+="  xori "+now_reg+", "+now_reg+", 1\n";
      break;
    case KOOPA_RBO_LE: // 小于等于
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  slt "+now_reg+", "+r+", "+l+"\n";
      riscv_code+="  xori "+now_reg+", "+now_reg+", 1\n";
      break;
```

```
    case KOOPA_RBO_AND: // 按位与
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  and "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_OR: // 按位或
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  or "+now_reg+", "+l+", "+r+"\n";
      break;
    case KOOPA_RBO_XOR: // 按位异或
      get_operand_load_reg(binary.lhs,l);
      get_operand_load_reg(binary.rhs,r);
      riscv_code+="  xor "+now_reg+", "+l+", "+r+"\n";
      break;
    default:
      // 其他类型暂时遇不到
      assert(false);
  }
  return dst;
}
```

我认为 Lv3 的难点主要在于，这一阶段比起上一阶段增加了非常多的文法，并且不像上一阶段的引导那样具体到代码层面。因此这一阶段是真正意义上由自己来设计和组织编译器的代码的开始。

完成 Lv3 后，对增加编译器功能的流程更加熟悉，因此在后续的阶段能够更熟练地完成一些重复性的工作（比如词法/语法分析、AST 的常规方法定义等），从而能够将思考的重心转移到算法设计和数据结构使用上。

Lv4：常量和变量

这一阶段需要处理常量/变量定义和赋值语句。

中间代码生成

这一阶段引入了常量和变量，故需要一个符号表，符号表的定义已经在上面说过。此外文法中 Block 非中介符号可以包含不确定数量的 BlockItem 。故我的做法是在相应的 AST 中预先将其整理到一个 vector 中。

在我的代码实现中，BlockAST 的 void get_block_item_vec 方法将所有的 BlockItemAST 整理到一个名为 block_item 的 vector 中，以在其 Dump 方法中可以依次调用 block_item 中的各 BlockItemAST 的 Dump 方法。

```
class BlockAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> rest_of_block_item;
  std::vector< std::unique_ptr<BaseAST> > block_item;
  bool have_ret; // 标记block最终是否返回
  bool have_create_symbol_table; // 是否已经创建符号表

  std::string Dump(){
    int n=block_item.size();

    bool flag=0; // 标记是否被返回语句打断
    for (int i=0;i<n;i++){
      block_item[i]->Dump();
```

```
      if (block_item[i]->get_have_ret()){
        flag=1;
        break;
      }
    }

    if (flag){
      have_ret=1;
    }
    else {
      have_ret=0;
    }
    return "";
  }

  void get_block_item_vec(){
    rest_of_block_item->get_rest_of_block_item_vec(block_item);
  }

  bool get_have_ret(){
    return have_ret;
  }

};
```

在常量赋值时，将该常值计算出来后保存到符号表中，ConstDefAST 的 Dump 方法中相关的代码如下，注意到其中调用了子结点的 get_val 方法，这个方法返回子结点对应的 AST 的值。

```
std::string Dump(){
    if (op==1){
      std::string val="";
      if (is_global_decl){
        const_init_val->Dump();
        const_init_val->load(val);
      }
      else {
        koopa_ir+=const_init_val->Dump();
        koopa_ir+=const_init_val->load(val);
      }

      int digit_val=const_init_val->get_val();

      std::string name=ident+"_"+std::to_string(var_def_id);
      symbol_table[now_symbol_table_id]
[ident]=symbol(0,1,name,std::to_string(digit_val));
    }

    ...

    return "";
  }
```

而变量定义则会将变量名修改为 ident_x，x 为一个不会重复的值，以防止中间代码中出现重复的变量名，然后保存到符号表中。VarDefAST 的 Dump 方法中相关的代码如下。

```
std::string Dump(){
  if (op==1){
    std::string name=ident+"_"+std::to_string(var_def_id);
    symbol_table[now_symbol_table_id][ident]=symbol(0,0,name);
    if (is_global_decl){
      koopa_ir+="global";
    }
    koopa_ir+="  @"+name+" = alloc i32";
    if (is_global_decl){
      koopa_ir+=", zeroinit";
      symbol_table[now_symbol_table_id][ident].val="0";
      symbol_table[now_symbol_table_id][ident].is_assigned=1;
    }
    koopa_ir+="\n";
    var_def_id++;
  }
  else if (op==2){
    std::string val="";
    if (is_global_decl){
      init_val->Dump();
      init_val->load(val);
    }
    else {
      koopa_ir+=init_val->Dump();
      koopa_ir+=init_val->load(val);
    }
    std::string name=ident+"_"+std::to_string(var_def_id);
    symbol_table[now_symbol_table_id][ident]=symbol(0,0,name,val);

    if (is_global_decl){
      koopa_ir+="global";
    }
    koopa_ir+="  @"+name+" = alloc i32";
    if (is_global_decl){
      koopa_ir+=", "+val+"\n";
    }
    else {
      koopa_ir+="\n  store "+val+", @"+name+"\n";
    }
    var_def_id++;
  }

   ...

  return "";
}
```

其他 AST 解析常量和变量时，在符号表中查询需要的信息以进行处理即可。

目标代码生成

我按照文档的引导为函数设置了 prologue ，扫描函数的所有指令，求出需要在栈上保留多少空间给局部变量，相关的代码如下。

```
// 访问函数
```

```
void Visit(const koopa_raw_function_t &func) {
  if (!func->bbs.len){
    return;
  }
  // 执行一些其他的必要操作
  riscv_code+="  .text\n";
  riscv_code+="  .global ";
  std::string func_name=(func->name)+1;
  riscv_code+=func_name+"\n";
  riscv_code+=func_name+":\n";

  if_restore_ra=0;
  stack_size=set_prologue(func);

  if (if_restore_ra){
    stack_offset=stack_size-4;
  }
  else {
    stack_offset=stack_size;
  }

  // 访问所有基本块
  Visit(func->bbs);
}

// 函数prologue，返回分配的栈空间总量
int set_prologue(const koopa_raw_function_t &func){
  int s=0,r=0,a=0;
  koopa_raw_slice_t bbs=func->bbs;

  int cnt=0,bbs_cnt=0;

  for (size_t i = 0; i < bbs.len; ++i) {
    bbs_cnt++;
    auto ptr1 = bbs.buffer[i];
    // 根据 slice 的 kind 决定将 ptr 视作何种元素
    koopa_raw_basic_block_t bb=reinterpret_cast<koopa_raw_basic_block_t>(ptr1);
    koopa_raw_slice_t insts=bb->insts;

    for (size_t j = 0; j < insts.len; ++j) {
      cnt++;
      auto ptr2 = insts.buffer[j];
      const koopa_raw_value_t value=reinterpret_cast<koopa_raw_value_t>(ptr2);
      s+=get_value_size(value);

      if (value->kind.tag==KOOPA_RVT_CALL){
        r=4;
        int arg_sum=value->kind.data.call.args.len;
        if ((arg_sum-8)*4>a){
          a=(arg_sum-8)*4;
        }
      }
    }
  }
```

```
  s=s+r+a;
  if (s%16){ // 与16字节对齐
    s=(s/16+1)*16;
  }
  if (!s) return 0;
  if (s>=2048){
    riscv_code+="  li t0, "+std::to_string(-s)+"\n";
    riscv_code+="  add sp, sp, t0\n";
  }
  else {
    riscv_code+="  addi sp, sp, "+std::to_string(-s)+"\n";
  }

  if (r){
    if_restore_ra=1;
    if (s-4>=2048){
      riscv_code+="  li t0, "+std::to_string(s-4)+"\n";
      riscv_code+="  add t0, t0, sp\n";
      riscv_code+="  sw ra, 0(t0)\n";
    }
    else {
      riscv_code+="  sw ra, "+std::to_string(s-4)+"(sp)\n";
    }
  }
  return s;
}

// 计算指令所需为局部变量分配的栈空间
int get_value_size(const koopa_raw_value_t &value){
  int s=0;

  const auto &kind = value->kind;
  if (kind.tag==KOOPA_RVT_ALLOC) {
    int size=get_type_size(value->ty,1);
    s+=size;
  }
  else if (kind.tag!=KOOPA_RVT_GLOBAL_ALLOC){
    const auto ty=value->ty;
    if (ty->tag!=KOOPA_RTT_UNIT){
      s+=4;
    }
  }
  return s;
}

int get_type_size(const koopa_raw_type_t &ty,bool get_whole_arr){
  switch(ty->tag){
    case KOOPA_RTT_INT32:{
      return 4;
    }
    case KOOPA_RTT_ARRAY:{
      if (get_whole_arr){
        return get_type_size(ty->data.array.base,1)*(ty->data.array.len);
      }
      else {
```

```
        return get_type_size(ty->data.array.base,1);
      }
    }
    case KOOPA_RTT_POINTER:{
      return get_type_size(ty->data.pointer.base,get_whole_arr);
    }
    default:
      std::cout<<ty->tag<<"\n";
      assert(false);
      break;
  }
}
```

这一阶段除了 Lv3已有的KOOPA_RVT_BINARY 类型，还新增了很多指令。在访问指令时，我根据指令的不同类型，调用相应的访问函数。对于有返回值的指令，需要设置 dst 为返回值所在的寄存器的编号，最后将返回值保存到栈上，并释放执行此指令使用的临时寄存器。

```
// 访问指令
void Visit(const koopa_raw_value_t &value) {
  // 根据指令类型判断后续需要如何访问
  int origin_cnt=reg_cnt; // 记录当前已使用的寄存器
  int dst=-1; // 记录返回值所在的寄存器

  const auto &kind = value->kind;
  switch (kind.tag) {
    case KOOPA_RVT_RETURN:{
      Visit_ret(kind.data.ret);
      break;
    }
    case KOOPA_RVT_INTEGER:{
      Visit_integer(kind.data.integer);
      break;
    }
    case KOOPA_RVT_BINARY:{
      dst=Visit_binary(kind.data.binary);
      break;
    }
    case KOOPA_RVT_ALLOC:{
      Visit_alloc(value);
      break;
    }
    case KOOPA_RVT_LOAD:{
      dst=Visit_load(kind.data.load);
      break;
    }
    case KOOPA_RVT_STORE:{
      Visit_store(kind.data.store);
      break;
    }
    case KOOPA_RVT_BRANCH:{
      Visit_branch(kind.data.branch);
      break;
    }
    case KOOPA_RVT_JUMP:{
      Visit_jump(kind.data.jump);
```

```
      break;
    }
    case KOOPA_RVT_CALL:{
      Visit_call(kind.data.call);
      dst=7;
      break;
    }
    case KOOPA_RVT_GLOBAL_ALLOC:{
      Visit_global_alloc(value);
      break;
    }
    case KOOPA_RVT_GET_ELEM_PTR:{
      dst=Visit_getelemptr(kind.data.get_elem_ptr);
      break;
    }
    case KOOPA_RVT_GET_PTR:{
      dst=Visit_getptr(kind.data.get_ptr);
      break;
    }
    default:
      // 其他类型暂时遇不到
      std::cout<<kind.tag<<endl;
      assert(false);
      break;
  }

  // 保存返回值
  const auto ty=value->ty;
  if (ty->tag!=KOOPA_RTT_UNIT&&dst<0){
    if (kind.tag!=6&&kind.tag!=7){
      std::cout<<kind.tag<<"\n";
      assert(false);
    }
  }
  if (ty->tag!=KOOPA_RTT_UNIT&&dst>=0){
    stack_offset-=4;
    if (stack_offset>=2048||stack_offset<=-2048){
      std::string tmp_reg=reg_name[reg_cnt];
      riscv_code+="  li "+tmp_reg+", "+std::to_string(stack_offset)+"\n";
      riscv_code+="  add "+tmp_reg+", "+tmp_reg+", sp\n";
      riscv_code+="  sw "+reg_name[dst]+", "+"0("+tmp_reg+")\n";
    }
    else {
      riscv_code+="  sw "+reg_name[dst]+", "+std::to_string(stack_offset)+"
(sp)\n";
    }
    std::string var="%"+std::to_string(ins_cnt);
    ins2var[value]=var;
    ins_cnt++;
    stack_dic[var]=stack_offset;
  }

  reg_cnt=origin_cnt; // 释放执行此指令使用的临时寄存器
}
```

Lv4 中新增的指令有 KOOPA_RVT_ALLOC ， KOOPA_RVT_LOAD，KOOPA_RVT_STORE 三种类型，相应的我写了 Visit_alloc ， Visit_load，Visit_store 三个函数。

其中 Visit_alloc 函数不生成汇编指令，只分配一个变量到栈上，即更新 ins2var 和 stack_dic 两个字典的信息。

而 Visit_load，Visit_store 这两个函数则根据指令的 src 和 dst 的信息生成相应指令即可，这里不再赘述。

```
void Visit_alloc(const koopa_raw_value_t &value){ // 分配一个变量到栈上
  koopa_raw_type_t ty=value->ty;
  switch (ty->tag){
    case KOOPA_RTT_POINTER:
    {
      std::string var="@"+std::to_string(var_cnt);
      var_cnt++;
      ins2var[value]=var;
      int size=get_type_size(ty,1);
      stack_offset-=size;
      stack_dic[var]=stack_offset;
      break;
    }
    default:
      std::cout<<ty->tag<<endl;
      assert(false);
      break;
  }
}
```

这一阶段的难点我认为主要在于汇编代码生成部分，对函数的栈的操作（预分配空间、栈偏移变化等）需要保持清晰。

Lv5：语句块和作用域

这一阶段要求实现语句块和作用域，其关键在于实现符号表的层次结构。

中间代码生成

在上面我已经提到过我的符号表的层次结构由 vector 实现，每一层都是一个符号表；而一个全局变量 now_symbol_table_id 表示当前在第几层。在解析一个块之前，now_symbol_table_id 增加 1 ，退出后 now_symbol_table_id 减少 1 。

当定义变量或常量并更新符号表时，只能插入当前层的符号表；而在符号表中查询某个变量或常量时，从当前层开始查找，查找不到再向上一层查找。在我的代码中，查询变量的操作都是下面这样的形式：

```
      for (int i=now_symbol_table_id;i>=0;i--){
        if (symbol_table[i].count(ident)){
          // 做需要做的事情

            ...

          break;
        }
      }
```

目标代码生成

这一阶段不涉及目标代码生成。

我认为这一阶段没有什么难点。

Lv6：if 语句

关于在文法设计上解决空悬 else 问题的部分已经在上面提过。

中间代码生成

我本以为中间代码生成会非常简单，只需要生成对应的基本块号，并在正确的位置生成跳转语句即可，即类似这样（以 OpenStmtAST 的 Dump 方法的第一个文法规则的部分为例，即没有 else 的 if 语句）：

```
    if (op==1){
      koopa_ir+=exp->Dump();
      std::string tmp_id="";
      koopa_ir+=exp->load(tmp_id);
      std::string then_label="%then_"+std::to_string(if_tmp_id);
      std::string end_label="%end_"+std::to_string(if_tmp_id);
      if_tmp_id++;

      koopa_ir+="  br "+tmp_id+", "+then_label+", "+end_label+"\n";

      koopa_ir+=then_label+":\n";
      stmt->Dump();

      // 把下面部分注释掉，则出现问题
      // if (!stmt->get_have_ret()){
      //   koopa_ir+="  jump "+end_label+"\n";
      // }

      koopa_ir+=end_label+":\n";
    }
```

但由于 Koopa IR 的一个规定，在本地测试中会出现问题：

“基本块的结尾必须是 `br`, `jump` 或 `ret` 指令其中之一 (并且, 这些指令只能出现在基本块的结尾). 也就是说, 即使两个基本块是相邻的, 例如上述程序的 `%else` 基本块和 `%end` 基本块, 如果你想表达执行完前者之后执行后者的语义, 你也必须在前者基本块的结尾添加一条目标为后者的 `jump` 指令. 这点和汇编语言中 label 的概念有所不同.“

比如本地测试的第一个样例：

```
int main() {
  if (1) return 1;
  return 0;
}
```

上面的 Dump 函数会导致最终出现这样的中间代码：

```
fun @main(): i32 {
%entry:
  br 1, %then_0, %end_0
%then_0:
  ret 1     //  ret语句不在基本块的结尾，出现错误
  jump %end_0
%end_0:
  ret 0
}
```

因此必须额外处理以消除掉不必要的跳转语句，在上面那个没有 else 的 if 语句的例子中，额外处理就是检查 if 语句的 then 分支是否返回，如果是则不产生相应的跳转语句。因此我定义了各 AST 的 get_have_ret 方法，返回此 AST 是否最终返回的信息。

而在有 else 的 if 语句中，还要额外检查一个点：如果 then 和 else 两个分支均没有跳转到 end_label（即全都返回了），那么不产生此 end_label 对应的基本块号。例如下面是 ClosedStmtAST 的 if-else 文法规则的对应部分，我定义了一个 need_end 布尔型变量来检查是否需要产生 end_label 对应的基本块号：

```
    else if (op==2){
      bool need_end=0; // 是否需要跳转到end

      koopa_ir+=exp->Dump();
      std::string tmp_id="";
      koopa_ir+=exp->load(tmp_id);
      std::string then_label="%then_"+std::to_string(if_tmp_id);
      std::string else_label="%else_"+std::to_string(if_tmp_id);
      std::string end_label="%end_"+std::to_string(if_tmp_id);
      if_tmp_id++;
      koopa_ir+="  br "+tmp_id+", "+then_label+", "+else_label+"\n";

      koopa_ir+=then_label+":\n";
      if_stmt->Dump();
      if (!if_stmt->get_have_ret()){
        koopa_ir+="  jump "+end_label+"\n";
        need_end=1;
      }

      koopa_ir+=else_label+":\n";
      else_stmt->Dump();

      if (!else_stmt->get_have_ret()){
        koopa_ir+="  jump "+end_label+"\n";
        need_end=1;
      }

      if (need_end){
        koopa_ir+=end_label+":\n";
      }
    }
```

而短路求值部分只需要将跳转语句加入 LorExpAST 和 LandExpAST 的 Dump 方法中即可。下面是 LorExpAST 中涉及到短路求值部分的代码。

```
    else if (op==2){
      std::string res_var="result_"+std::to_string(if_tmp_id);
      std::string then_label="%then_"+std::to_string(if_tmp_id);
      std::string end_label="%end_"+std::to_string(if_tmp_id);
      if_tmp_id++;

      symbol_table[now_symbol_table_id][res_var]=symbol(0,0,"1","1");
      tmp_str+="  @"+res_var+" = alloc i32\n";
      tmp_str+="  store "+std::to_string(1)+", @"+res_var+"\n";

      tmp_str+=lor_exp->Dump();
      std::string lor_id="";
      tmp_str+=lor_exp->load(lor_id);
      tmp_str+="  br "+lor_id+", "+end_label+", "+then_label+"\n";
      tmp_str+=then_label+":\n";

      tmp_str+=land_exp->Dump();
      std::string land_id="";
      tmp_str+=land_exp->load(land_id);
      std::string tmp_id="%"+std::to_string(koopa_tmp_id);
      koopa_tmp_id++;
      tmp_str+="  "+tmp_id+" = ne "+land_id+", 0"+"\n";
      tmp_str+="  store "+tmp_id+", @"+res_var+"\n";
      tmp_str+="  jump "+end_label+"\n";

      tmp_str+=end_label+":\n";

      ir_id="%"+std::to_string(koopa_tmp_id);
      koopa_tmp_id++;
      tmp_str+="  "+ir_id+" = load @"+res_var+"\n";
    }
```

目标代码生成

汇编代码只需完成对 br 和 jump 指令的访问函数即可，没有值得特别提的地方，相应的代码如下：

```
void Visit_branch(const koopa_raw_branch_t &branch){
  koopa_raw_value_t cond=branch.cond;
  koopa_raw_basic_block_t true_bb=branch.true_bb;
  koopa_raw_basic_block_t false_bb=branch.false_bb;

  std::string now_reg=reg_name[reg_cnt];
  reg_cnt++;
  switch(cond->kind.tag){
    case KOOPA_RVT_INTEGER:{ // 分支条件是立即数
        int32_t int_val = cond->kind.data.integer.value;
        riscv_code+="  li "+now_reg+", "+std::to_string(int_val)+"\n";
      break;
    }
    case KOOPA_RVT_CALL:
    case KOOPA_RVT_LOAD:
    case KOOPA_RVT_BINARY:{
      std::string var=ins2var[cond];
      int offset=stack_dic[var];
      if (offset>=2048||offset<=-2048){
```

```
        std::string tmp_reg=reg_name[reg_cnt];
        riscv_code+="  li "+tmp_reg+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+tmp_reg+", "+tmp_reg+", sp\n";
        riscv_code+="  lw "+now_reg+", "+"0("+tmp_reg+")\n";
      }
      else {
        riscv_code+="  lw "+now_reg+", "+std::to_string(offset)+"(sp)\n";
      }
      break;
    }
    default:
      std::cout<<cond->kind.tag<<"\n";
      assert(false);
  }

  std::string true_label=(true_bb->name)+1;
  std::string false_label=(false_bb->name)+1;
  riscv_code+="  bnez "+now_reg+", "+true_label+"\n";
  riscv_code+="  j "+false_label+"\n";
}

void Visit_jump(const koopa_raw_jump_t &jump){
  koopa_raw_basic_block_t target=jump.target;
  std::string target_label=(target->name)+1;
  riscv_code+="  j "+target_label+"\n";
}
```

这一阶段的主要难点我认为就是我上面提到的在中间代码生成时需要对 if-else 语句做的额外处理。

Lv7：while 语句

中间代码生成

while 语句的实现思路与 if 语句基本相同，同样需要注意消除多余的跳转语句。

而由于 break 语句和 continue 语句需要跳转到当前循环对应的基本块号，因此需要一个全局变量记录当前循环的数字编号（我在上面"主要数据结构"一节中提到过：我在处理循环时使用的基本块号均为 %while_entry_x，%while_body_x，%while_end_x 这种形式，x为数字编号），在我的代码中通过当前循环层数 loop_num 可以在 loop_label_table 中查询到当前循环的数字编号（loop_num 和 loop_label_table 在进入和退出循环时需要相应地进行更新）。

下面是 SimpleStmtAST 中处理 break 语句和 continue 语句的部分。注意到在跳转语句后会生成一个 unreachable 基本块号，以保证此跳转语句位于一个基本块的末尾。

```
    else if (op==7){ // break
      std::string
end_label="%while_end_"+std::to_string(loop_label_table[loop_num]);
      koopa_ir+="  jump "+end_label+"\n";

      std::string
unreachable_label="%unreachable_"+std::to_string(unreachable_tmp_id);
      unreachable_tmp_id++;
      koopa_ir+=unreachable_label+":\n";
    }
    else if (op==8){ // continue
```

```
      std::string
entry_label="%while_entry_"+std::to_string(loop_label_table[loop_num]);
      koopa_ir+="  jump "+entry_label+"\n";

      std::string
unreachable_label="%unreachable_"+std::to_string(unreachable_tmp_id);
      unreachable_tmp_id++;
      koopa_ir+=unreachable_label+":\n";
    }
```

目标代码生成

这一阶段不涉及目标代码生成。

由于 while 语句的处理与 if 语句异曲同工，我认为写过 Lv6 后，这一阶段没有什么难点。

Lv8：函数和全局变量

这一阶段需要处理函数和全局变量。

中间代码生成

所有的函数定义都会记录在全局作用域中，即符号表的第一层。

我根据文档的引导，在函数定义中将函数的输入参数先加载到局部变量中，以便后续的处理。下面是 FuncDefAST 的 Dump 方法。

```
std::string Dump(){
    bool is_void=func_type->is_void();
    symbol_table[now_symbol_table_id][ident]=symbol(1,is_void,ident);
    koopa_ir+="fun @";
    koopa_ir+=ident;
    if (op==1){
      koopa_ir+="()";
    }
    else if (op==2){
      koopa_ir+="(";
      func_f_params->Dump();
      koopa_ir+=")";
    }
    func_type->Dump();
    koopa_ir+=" {\n";
    koopa_ir+="%entry:\n";

    now_symbol_table_id++;
    symbol_table.push_back(std::map< std::string, symbol >());

    // 将函数参数提前alloc和store
    if (op==2){
      func_f_params->pre_alloc_store();
    }
    block->Dump();

    symbol_table.pop_back();
    now_symbol_table_id--;

    if (!block->get_have_ret()){
```

```
      if (func_type->get_func_type()=="int"){
        koopa_ir+="  ret 0\n";
      }
      else {
        koopa_ir+="  ret\n";
      }
    }
    koopa_ir+="}\n";
    return "";
  }
```

以及最终生成函数参数预分配指令的 FuncFParamAST 的 pre_alloc_store 方法（下面 op=2和op=3的情况是 Lv9 时加入了数组和指针的情况）。

```
  void pre_alloc_store(){
    std::string name=ident+"_"+std::to_string(var_def_id);
    if (op==1){
      koopa_ir+="  @"+name+" = alloc i32\n";
      koopa_ir+="  store @"+ident+", @"+name+"\n";
      symbol_table[now_symbol_table_id][ident]=symbol(0,0,name,"0",1,0,0); // 认为输入参数已被赋值
    }
    else if (op==2){
      koopa_ir+="  @"+name+" = alloc *i32\n";
      koopa_ir+="  store @"+ident+", @"+name+"\n";
      symbol_table[now_symbol_table_id][ident]=symbol(0,0,name,"0",1,0,1); // 认为输入参数已被赋值
    }
    else if (op==3){
      koopa_ir+="  @"+name+" = alloc *"+axis_str+"\n";
      koopa_ir+="  store @"+ident+", @"+name+"\n";
      symbol_table[now_symbol_table_id][ident]=symbol(0,0,name,"0",1,0,axis_info.size()+1); // 认为输入参数已被赋值
    }
    var_def_id++;
  }
```

函数调用非常简单，只需要查询符号表中函数的返回值类型并生成 call 指令，如果函数有输入参数则生成对应参数的指令并将变量号加入 call 指令中。

SysY库函数统一在 Koopa IR 中提前声明并加入符号表即可。

全局变量定义要注意两个地方。

1. 由于表达式求值是在 Dump 函数中完成的，故全局变量定义需要调用子结点的 Dump 函数，但是不能将生成的语句加入 koopa_ir 中。我使用一个全局变量 is_global_decl 来判断是否是全局变量定义。关于 is_global_decl 的设置的相关代码如下，后续 ConstDefAST 和 VarDefAST 会根据此全局变量的值确定对 koopa_ir 的操作。

```
class GlobalItemAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> decl;
  std::unique_ptr<BaseAST> func_def;

  std::string Dump(){
```

```
    if (op==1){ // 全局变量声明
      is_global_decl=1;
      decl->Dump();
      is_global_decl=0;
    }
    else if (op==2){ // 函数定义
      func_def->Dump();
    }
    return "";
  }
};
```

2. 全局变量必须被赋值，没有显式赋值的全局变量以 zeroinit 赋值。

目标代码生成

这一阶段的在线文档再次提到了函数的 prologue ， 主要是 call 指令的栈空间分配和 ra 寄存器的设置，根据文档的引导即可。

需要注意函数调用时要先处理输入参数，尤其注意超过8个的输入参数要保存到栈上，并且保存的位置处于当前函数栈帧的最底部。

```
void Visit_call(const koopa_raw_call_t &call){
  koopa_raw_slice_t args=call.args;

  int reg_args=8; // 需要存入寄存器的输入参数数量
  if (args.len<reg_args){
    reg_args=args.len;
  }
  // 将输入参数存入寄存器中
  for (int i=0;i<reg_args;i++){
    auto ptr = args.buffer[i];
    const koopa_raw_value_t value=reinterpret_cast<koopa_raw_value_t>(ptr);

    switch (value->kind.tag){ // 立即数
      case KOOPA_RVT_INTEGER:{
        int32_t int_val = value->kind.data.integer.value;
        riscv_code+="  li a"+std::to_string(i)+", "+
(std::to_string(int_val))+"\n";
        break;
      }
      case KOOPA_RVT_GET_ELEM_PTR:
      case KOOPA_RVT_CALL:
      case KOOPA_RVT_LOAD:
      case KOOPA_RVT_BINARY:{ // 变量
        std::string var=ins2var[value];
        int offset=stack_dic[var];
        if (offset>=2048||offset<=-2048){
          std::string tmp_reg=reg_name[reg_cnt];
          riscv_code+="  li "+tmp_reg+", "+std::to_string(offset)+"\n";
          riscv_code+="  add "+tmp_reg+", "+tmp_reg+", sp\n";
          riscv_code+="  lw a"+std::to_string(i)+", "+"0("+tmp_reg+")\n";
        }
        else {
          riscv_code+="  lw a"+std::to_string(i)+", "+std::to_string(offset)+"
(sp)\n";
```

```
        }
        break;
      }
      default:
        std::cout<<value->kind.tag<<endl;
        assert(false);
    }
  }
  // 将其他输入参数存入栈中
  std::string now_reg=reg_name[reg_cnt];
  reg_cnt++;

  for (int i=reg_args;i<args.len;i++){
    auto ptr = args.buffer[i];
    const koopa_raw_value_t value=reinterpret_cast<koopa_raw_value_t>(ptr);

    switch (value->kind.tag){
      case KOOPA_RVT_INTEGER:{ // 立即数
        riscv_code+="  li "+now_reg+", "+std::to_string(value-
>kind.data.integer.value)+"\n";
        int offset=(i-8)*4;
        riscv_code+="  sw "+now_reg+", "+std::to_string(offset)+"(sp)\n";
        break;
      }
      case KOOPA_RVT_GET_ELEM_PTR:
      case KOOPA_RVT_CALL:
      case KOOPA_RVT_LOAD:
      case KOOPA_RVT_BINARY:{ // 变量
        std::string var=ins2var[value];
        int offset=stack_dic[var];
        if (offset>=2048||offset<=-2048){
          std::string tmp_reg=reg_name[reg_cnt];
          riscv_code+="  li "+tmp_reg+", "+std::to_string(offset)+"\n";
          riscv_code+="  add "+tmp_reg+", "+tmp_reg+", sp\n";
          riscv_code+="  lw "+now_reg+", "+"0("+tmp_reg+")\n";
        }
        else {
          riscv_code+="  lw "+now_reg+", "+std::to_string(offset)+"(sp)\n";
        }
        offset=(i-8)*4;
        riscv_code+="  sw "+now_reg+", "+std::to_string(offset)+"(sp)\n";
        break;
      }
      default:
        std::cout<<value->kind.tag<<endl;
        assert(false);
        break;
    }
  }

  std::string func_name=(call.callee->name)+1;
  riscv_code+="  call "+func_name+"\n";
}
```

SysY 库函数的声明直接根据在线文档中的引导跳过即可。

全局变量分配的访问函数如下：

```
void Visit_global_alloc(const koopa_raw_value_t &value){ // 全局变量
  std::string var_name=(value->name)+1;
  riscv_code+="  .data\n";
  riscv_code+="  .globl "+var_name+"\n";
  koopa_raw_value_t init=value->kind.data.global_alloc.init;
  riscv_code+=var_name+":\n";

  switch (init->kind.tag){
    case KOOPA_RVT_INTEGER:{ // 立即数
      riscv_code+="  .word "+std::to_string(init->kind.data.integer.value)+"\n";
      break;
    }
    case KOOPA_RVT_ZERO_INIT:{ // 零初始化
      koopa_raw_type_t ty=value->ty;
      int size=get_type_size(ty,1);
      riscv_code+="  .zero "+std::to_string(size)+"\n";
      break;
    }
    case KOOPA_RVT_AGGREGATE:{
      koopa_raw_aggregate_t aggregate=init->kind.data.aggregate;
      aggregate_init(aggregate);
      break;
    }
    default:
      std::cout<<init->kind.tag<<endl;
      assert(false);
  }
}
```

而涉及到全局变量的使用时，只需在相应的位置加入 la 和 lw 指令即可。

这一阶段难度较大，主要是增加或修改了很多文法规则，并且加入了函数这个在之前的阶段中不太涉及的概念，因此中间代码生成部分需要做很多全新的编码工作而不能复用并修改之前已完成的代码，甚至需要做一定程度的代码重构以支持新的文法规则；同时在目标代码生成阶段对函数参数的处理需要非常仔细，以免错误地分配栈空间。

Lv9：数组

这一阶段要求处理数组和数组参数。

中间代码生成

数组的定义与普通变量的定义区别很大，我定义了 AxisAST 来辅助处理数组的每个维度。

```
class AxisAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> const_exp;
  std::unique_ptr<BaseAST> axis;

  void get_rest_of_axis_info_vec(std::vector< std::unique_ptr<BaseAST> > &vec){
    vec.push_back(std::move(const_exp));
    if (axis!=NULL){
      axis->get_rest_of_axis_info_vec(vec);
    }
  }
};
```

每个 AxisAST 的 const_exp 都会被预先加入 ConstDefAST 或 VarDefAST 的 axis_info 列表中，以便于后续的处理。

首先来看数组的初始化，我使用 aggregate / zeroinit 初始化全局变量，而使用多个 getelemptr 和 store 指令初始化局部变量。不管是全局变量还是局部变量，都需要先得到数组的全部元素的初始值，这是由 ConstInitValAST 的 get_rest_of_init_vec 方法完成的，此方法根据 "Lv9.2. 多维数组" 这一节的引导，递归地获取数组的初始化列表。

```
  virtual std::string get_rest_of_init_vec(std::vector<std::string>
&vec,std::vector<int> axis_length,int now_axis){
    std::string tmp_str="";
    if (op==1){ // ConstExp
      if (const_exp->is_number()){
        vec.push_back(std::to_string(const_exp->get_val()));
      }
      else {
        std::string tmp_id="";
        tmp_str+=const_exp->load(tmp_id);
        vec.push_back(tmp_id);
      }
    }
    else if (op==2){ // 空的{}
      int tot=1;
      for (int i=now_axis;i<axis_length.size();i++){
        tot*=axis_length[i];
      }
      for (int i=0;i<tot;i++){
        vec.push_back("0");
      }
    }
    else if (op==3){  // 非空的{}
      for (int i=0;i<const_init_val_vec.size();i++){
        int now_tot=vec.size(),tmp_axis=now_axis+1;
        for (int j=axis_length.size()-1;j>now_axis;j--){
          if (now_tot%axis_length[j]==0){
            now_tot/=axis_length[j];
          }
          else {
            tmp_axis=j;
            break;
          }
```

```
        }
        tmp_str+=const_init_val_vec[i]-
>get_rest_of_init_vec(vec,axis_length,tmp_axis);
      }

      int tot=1;
      for (int i=now_axis;i<axis_length.size();i++){
        tot*=axis_length[i];
      }
      int rest=vec.size()%tot;
      if (rest){
        for (int i=rest;i<tot;i++){
          vec.push_back("0");
        }
      }
    }
    return tmp_str;
  }
```

成功得到数组的初始化列表后，数组的初始化就非常轻松了，生成数组初始化的中间代码就非常轻松了，这里直接略过了。

而关于数组的使用，需要 getelemptr 和 load / store 指令的配合，之前提到过为了兼容数组，指针与普通变量的使用，我定义了AST 的 load 方法，大多数的 load 方法最终都会追溯到 LvalAST 的 load 方法，其他则会根据其文法规则的特点直接将 loaded_id 设置为 ir_id 。

LvalAST 的 load 方法会在此变量是一个数组或者指针时生成 load 指令，并且设置 loaded_id 为此 load 指令对应的变量名。下面是 LvalAST 的 load 方法。

```
  std::string load(std::string &loaded_id){
    std::string tmp_str="";
    if (op==1){
      for (int i=now_symbol_table_id;i>=0;i--){
        if (symbol_table[i].count(ident)){
          if (symbol_table[i][ident].is_const){
            loaded_id=ir_id;
            break;
          }
          else {
            if (symbol_table[i][ident].is_arr){
              loaded_id=ir_id;
              func_arg_is_arr=1;
              break;
            }
            else {
              loaded_id="%"+std::to_string(koopa_tmp_id);
              koopa_tmp_id++;
              tmp_str+="  "+loaded_id+" = load "+ir_id+"\n";
              break;
            }
          }
        }
      }
    }
    else if (op==2){
```

```
      for (int i=now_symbol_table_id;i>=0;i--){
        if (symbol_table[i].count(ident)){
          if (axis_info.size()<symbol_table[i][ident].axis){
            loaded_id=ir_id;
            func_arg_is_arr=1;
            break;
          }
          loaded_id="%"+std::to_string(koopa_tmp_id);
          koopa_tmp_id++;
          tmp_str+="  "+loaded_id+" = load "+ir_id+"\n";
          break;
        }
      }
    }
    return tmp_str;
  }
```

最后来看指针变量的使用，这是唯一需要使用到 getptr 指令的地方。下面是 LvalAST 的 Dump 方法处理第二条文法规则 `IDENT Axis` 的相关代码。当 ident 在符号表中对应一个数组时，不断执行 getelemptr 直到消耗完 Axis 中的所有维度；当 ident 在符号表中对应一个指针时，需要先执行一次 load，再执行一次 getptr，再执行一次 load，此时如果还有维度，则按照数组处理。

```
    else if (op==2){
      for (int i=now_symbol_table_id;i>=0;i--){
        if (symbol_table[i].count(ident)){
          for (int j=0;j<axis_info.size();j++){
            tmp_str+=axis_info[j]->Dump();
          }

          std::string now_ptr_id="@"+symbol_table[i][ident].name;
          if (symbol_table[i][ident].is_ptr){
            is_ptr=1;

            int cnt=2;
            for (int j=0;j<axis_info.size();j++){
              std::string tmp_ptr_id="";
              if (cnt){
                tmp_ptr_id="%"+std::to_string(koopa_tmp_id);
                koopa_tmp_id++;
                tmp_str+="  "+tmp_ptr_id+" = load "+now_ptr_id+"\n";
                cnt--;
              }

              std::string loaded_id="";
              tmp_str+=axis_info[j]->load(loaded_id);
              std::string next_ptr_id="%"+std::to_string(koopa_tmp_id);
              koopa_tmp_id++;
              if (cnt){
                tmp_str+="  "+next_ptr_id+" = getptr "+tmp_ptr_id+",
"+loaded_id+"\n";
              }
              else {
                tmp_str+="  "+next_ptr_id+" = getelemptr "+now_ptr_id+",
"+loaded_id+"\n";
              }
```

```
              now_ptr_id=next_ptr_id;
            }
            ir_id=now_ptr_id;
          }
          else {
            is_ptr=0;
            for (int j=0;j<axis_info.size();j++){
              std::string next_ptr_id="%"+std::to_string(koopa_tmp_id);
              koopa_tmp_id++;
              std::string loaded_id="";
              tmp_str+=axis_info[j]->load(loaded_id);
              tmp_str+="  "+next_ptr_id+" = getelemptr "+now_ptr_id+",
"+loaded_id+"\n";
              now_ptr_id=next_ptr_id;
            }
            ir_id=now_ptr_id;
          }
          break;
        }
      }
    }
```

目标代码生成

由于引入了数组和指针，故需要计算类型的大小。并且需要将此改动应用于对函数栈空间分配和 alloc / global alloc 的访问函数中。

我写的 get_type_size 函数递归调用自身得到类型的大小。其中输入参数 get_whole_arr 表示其考虑的是数组的单个元素还是全部元素的大小。相关代码如下：

```
int get_type_size(const koopa_raw_type_t &ty,bool get_whole_arr){
  switch(ty->tag){
    case KOOPA_RTT_INT32:{
      return 4;
    }
    case KOOPA_RTT_ARRAY:{
      if (get_whole_arr){
        return get_type_size(ty->data.array.base,1)*(ty->data.array.len);
      }
      else {
        return get_type_size(ty->data.array.base,1);
      }
    }
    case KOOPA_RTT_POINTER:{
      return get_type_size(ty->data.pointer.base,get_whole_arr);
    }
    default:
      std::cout<<ty->tag<<"\n";
      assert(false);
      break;
  }
}
```

在 global alloc的访问函数中还要考虑 aggregate 的初始化，我写了一个 aggregate_init 函数，其可以递归调用自身来完成对 aggregate 的初始化。相关代码如下：

```
void Visit_global_alloc(const koopa_raw_value_t &value){ // 全局变量
  std::string var_name=(value->name)+1;
  riscv_code+="  .data\n";
  riscv_code+="  .globl "+var_name+"\n";
  koopa_raw_value_t init=value->kind.data.global_alloc.init;
  riscv_code+=var_name+":\n";

  switch (init->kind.tag){
    case KOOPA_RVT_INTEGER:{ // 立即数
      riscv_code+="  .word "+std::to_string(init->kind.data.integer.value)+"\n";
      break;
    }
    case KOOPA_RVT_ZERO_INIT:{ // 零初始化
      koopa_raw_type_t ty=value->ty;
      int size=get_type_size(ty,1);
      riscv_code+="  .zero "+std::to_string(size)+"\n";
      break;
    }
    case KOOPA_RVT_AGGREGATE:{
      koopa_raw_aggregate_t aggregate=init->kind.data.aggregate;
      aggregate_init(aggregate);
      break;
    }
    default:
      std::cout<<init->kind.tag<<endl;
      assert(false);
  }
}

void aggregate_init(const koopa_raw_aggregate_t &aggregate){
  koopa_raw_slice_t elems=aggregate.elems;
  for (size_t i = 0; i < elems.len; ++i) {
    auto ptr = elems.buffer[i];
    const koopa_raw_value_t value=reinterpret_cast<koopa_raw_value_t>(ptr);
    switch (value->kind.tag)
    {
      case KOOPA_RVT_INTEGER:{
        riscv_code+="  .word "+std::to_string(value-
>kind.data.integer.value)+"\n";
        break;
      }
      case KOOPA_RVT_AGGREGATE:{
        aggregate_init(value->kind.data.aggregate);
        break;
      }
      default:
        std::cout<<value->kind.tag<<"\n";
        assert(false);
        break;
    }
  }
}
```

还需要处理 getelemptr 指令和 getptr 指令。

getelemptr 指令的访问函数的代码如下。只需分别解析 src 和 index 的类型，插入相应的 li / lw / la 和 add / mul 指令即可。

```
int Visit_getelemptr(const koopa_raw_get_elem_ptr_t &get_elem_ptr){
  koopa_raw_value_t src=get_elem_ptr.src,index=get_elem_ptr.index;
  koopa_raw_type_t ty=src->ty;
  int ptr_size=get_type_size(ty,0);

  int dst=reg_cnt;
  std::string now_reg=reg_name[reg_cnt];
  reg_cnt++;

  // 计算目标在栈上的地址
  switch(src->kind.tag){
    case KOOPA_RVT_ALLOC:{
      std::string var=ins2var[src];
      int offset=stack_dic[var];
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+now_reg+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+now_reg+", sp, "+now_reg+"\n";
      }
      else {
        riscv_code+="  addi "+now_reg+", sp, "+std::to_string(offset)+"\n";
      }
      break;
    }
    case KOOPA_RVT_GET_PTR:
    case KOOPA_RVT_GET_ELEM_PTR:{
      std::string var=ins2var[src];
      int offset=stack_dic[var];
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+now_reg+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+now_reg+", sp, "+now_reg+"\n";
      }
      else {
        riscv_code+="  addi "+now_reg+", sp, "+std::to_string(offset)+"\n";
      }
      riscv_code+="  lw "+now_reg+", 0("+now_reg+")\n";
      break;
    }
    case KOOPA_RVT_GLOBAL_ALLOC:{
      std::string var_name=(src->name)+1;
      riscv_code+="  la "+now_reg+", "+var_name+"\n";
      break;
    }
    default:
      std::cout<<src->kind.tag<<"\n";
      assert(false);
      break;
  }

  switch(index->kind.tag){
    case KOOPA_RVT_INTEGER:{
      int idx=index->kind.data.integer.value;
      std::string tmp_reg1=reg_name[reg_cnt];
```

```
      reg_cnt++;
      std::string tmp_reg2=reg_name[reg_cnt];
      reg_cnt++;
      riscv_code+="  li "+tmp_reg1+", "+std::to_string(idx)+"\n";
      riscv_code+="  li "+tmp_reg2+", "+std::to_string(ptr_size)+"\n";
      riscv_code+="  mul "+tmp_reg1+", "+tmp_reg1+", "+tmp_reg2+"\n";
      riscv_code+="  add "+now_reg+", "+now_reg+", "+tmp_reg1+"\n";
      break;
    }
    case KOOPA_RVT_BINARY:
    case KOOPA_RVT_LOAD:{
      std::string var=ins2var[index];
      int offset=stack_dic[var];
      std::string tmp_reg1=reg_name[reg_cnt];
      reg_cnt++;
      std::string tmp_reg2=reg_name[reg_cnt];
      reg_cnt++;
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+tmp_reg1+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+tmp_reg1+", "+tmp_reg1+", sp\n";
        riscv_code+="  lw "+tmp_reg1+", "+"0("+tmp_reg1+")\n";
      }
      else {
        riscv_code+="  lw "+tmp_reg1+", "+std::to_string(offset)+"(sp)\n";
      }
      riscv_code+="  li "+tmp_reg2+", "+std::to_string(ptr_size)+"\n";
      riscv_code+="  mul "+tmp_reg1+", "+tmp_reg1+", "+tmp_reg2+"\n";
      riscv_code+="  add "+now_reg+", "+now_reg+", "+tmp_reg1+"\n";
      break;
    }
    default:
      std::cout<<index->kind.tag<<"\n";
      assert(false);
      break;
  }

  return dst;
}
```

getptr 指令的处理与 getelemptr 几乎完全相同，唯一的区别是解析 src 类型时，如果是 KOOPA_RVT_ALLOC 则需要最后再加入一条 lw 指令，否则解析得到的不是 src 对应指针在栈上的真实地址，而是保存此真实地址值的栈地址。而 getelemptr 指令的操作对象是数组，整个数组都会被保存在栈上。

```
int Visit_getptr(const koopa_raw_get_ptr_t &get_ptr){
  koopa_raw_value_t src=get_ptr.src,index=get_ptr.index;
  koopa_raw_type_t ty=src->ty;
  int ptr_size=get_type_size(ty,1);

  int dst=reg_cnt;
  std::string now_reg=reg_name[reg_cnt];
  reg_cnt++;

  // 计算目标在栈上的地址
  switch(src->kind.tag){
```

```
    case KOOPA_RVT_LOAD:
    case KOOPA_RVT_ALLOC:{
      std::string var=ins2var[src];
      int offset=stack_dic[var];
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+now_reg+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+now_reg+", sp, "+now_reg+"\n";
      }
      else {
        riscv_code+="  addi "+now_reg+", sp, "+std::to_string(offset)+"\n";
      }
      riscv_code+="  lw "+now_reg+", 0("+now_reg+")\n";  // 这里需要增加一条 lw 指令
      break;
    }
    case KOOPA_RVT_GET_ELEM_PTR:{
      std::string var=ins2var[src];
      int offset=stack_dic[var];
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+now_reg+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+now_reg+", sp, "+now_reg+"\n";
      }
      else {
        riscv_code+="  addi "+now_reg+", sp, "+std::to_string(offset)+"\n";
      }
      riscv_code+="  lw "+now_reg+", 0("+now_reg+")\n";
      break;
    }
    case KOOPA_RVT_GLOBAL_ALLOC:{
      std::string var_name=(src->name)+1;
      riscv_code+="  la "+now_reg+", "+var_name+"\n";
      break;
    }
    default:
      std::cout<<src->kind.tag<<"\n";
      assert(false);
      break;
  }

  switch(index->kind.tag){
    case KOOPA_RVT_INTEGER:{
      int idx=index->kind.data.integer.value;
      std::string tmp_reg1=reg_name[reg_cnt];
      reg_cnt++;
      std::string tmp_reg2=reg_name[reg_cnt];
      reg_cnt++;
      riscv_code+="  li "+tmp_reg1+", "+std::to_string(idx)+"\n";
      riscv_code+="  li "+tmp_reg2+", "+std::to_string(ptr_size)+"\n";
      riscv_code+="  mul "+tmp_reg1+", "+tmp_reg1+", "+tmp_reg2+"\n";
      riscv_code+="  add "+now_reg+", "+now_reg+", "+tmp_reg1+"\n";
      break;
    }
    case KOOPA_RVT_BINARY:
    case KOOPA_RVT_LOAD:{
      std::string var=ins2var[index];
      int offset=stack_dic[var];
```

```
      std::string tmp_reg1=reg_name[reg_cnt];
      reg_cnt++;
      std::string tmp_reg2=reg_name[reg_cnt];
      reg_cnt++;
      if (offset>=2048||offset<=-2048){
        riscv_code+="  li "+tmp_reg1+", "+std::to_string(offset)+"\n";
        riscv_code+="  add "+tmp_reg1+", "+tmp_reg1+", sp\n";
        riscv_code+="  lw "+tmp_reg1+", "+"0("+tmp_reg1+")\n";
      }
      else {
        riscv_code+="  lw "+tmp_reg1+", "+std::to_string(offset)+"(sp)\n";
      }
      riscv_code+="  li "+tmp_reg2+", "+std::to_string(ptr_size)+"\n";
      riscv_code+="  mul "+tmp_reg1+", "+tmp_reg1+", "+tmp_reg2+"\n";
      riscv_code+="  add "+now_reg+", "+now_reg+", "+tmp_reg1+"\n";
      break;
    }
    default:
      std::cout<<index->kind.tag<<"\n";
      assert(false);
      break;
  }

  return dst;
}
```

这一阶段的难度较大，主要是数组和指针的操作更加抽象和复杂，容易出现 bug ，但是有本地测试的样例可以参考和 debug 。

优化

经过以上阶段之后，我的编译器已经可以生成完整的汇编代码了。在此之上，我根据 “Lv9+.3. 优化” 一节中 “消除冗余 load ” 的引导，删除了紧随在 sw 指令之后且目的寄存器与该 sw 指令的源寄存器相同的 lw 指令。

我的 main.cpp 的代码中相关的部分如下：

```
  else if (mode[1]=='r'||mode[1]=='p'){
    koopa_process();

    // 删除冗余load
    vector<string> split_riscv_code;
    string tmp_str=riscv_code;
      int pos=tmp_str.find("\n");
      while (pos!=tmp_str.npos){
          string tmp=tmp_str.substr(0,pos);
          split_riscv_code.push_back(tmp);
          tmp_str=tmp_str.substr(pos+1,tmp_str.size());
          pos=tmp_str.find("\n");
      }

    int n=split_riscv_code.size();
    bool deleted[n];
    memset(deleted,0,sizeof(deleted));
    for (int i=0;i<n;i++){
```

```
      if (deleted[i]){
        continue;
      }
      else {
        if (split_riscv_code[i].size()>=4){
          string opt=split_riscv_code[i].substr(0,4);
          if (opt=="  sw"){
            for (int j=i+1;j<n;j++){
              if (split_riscv_code[j].size()==split_riscv_code[i].size()){
                string another_opt=split_riscv_code[j].substr(0,4);
                if (another_opt=="  lw"){
                  string
str1=split_riscv_code[i].substr(3,split_riscv_code[i].size());
                  string
str2=split_riscv_code[j].substr(3,split_riscv_code[j].size());
                  if (str1==str2){
                    deleted[j]=1;
                  }
                }
              }
              if (!deleted[j]){
                break;
              }
            }
          }
        }
      }
    }
    string final_riscv_code="";
    for (int i=0;i<n;i++){
      if (!deleted[i]){
        final_riscv_code+=split_riscv_code[i];
        final_riscv_code+="\n";
      }
    }

    out<<final_riscv_code;
  }
```

以上是我在各个阶段的主要的编码思路和算法设计的细节。

四、实习总结

1、收获与体会

经过编译实习，我按照文档的引导独立地实现了一个编译器，这也是我至今为止独立实现的最大的编程项目，不仅锻炼了我的代码能力，而且带给了我极大的成就感。

实现编译器的各种功能有许多困难，我认为主要集中在 Lv3，Lv4，Lv8 和 Lv9。其中 Lv3 和 Lv4 主要是对项目流程和使用工具还不够熟悉，同时这也是符号表等需要使用的数据结构从无到有的编码阶段。而 Lv8 和 Lv9 则是引入了函数和数组的概念，其操作更加复杂和抽象，不论是在中间代码生成时需要在正确的位置插入更多的指令，还是在目标代码生成时需要对栈的情况保持清晰，而这都需要更精细的设计。

实现一个编译器花费了我这一学期大量的时间和精力，但此过程也非常有趣，经过具体地编写程序之后，我相信我对编译器的理解也会更加具体深入。

2、对课程的建议

我对编译实习的内容非常满意，但我认为编译优化方面可能可以有更多的实践。

虽然编译优化放在 Lv9+. 这一阶段，并且由学生自由发挥。但就我个人的体验来说，实现前面的阶段已经耗费了学生太多的时间和精力，没有太大的意愿继续在自己的编译器框架上进行比较复杂的编译优化。

但编译优化是编译器领域非常重要的部分，让学生进行相关的实践是很好的。因此我认为在保留编译器实习 Lv9+. 阶段的基础上，可以在学期中布置一些课后的工作量较小的 lab ，以代码填空或者其他方式让学生实现一些编译优化的经典算法。

3、总结

以上是我的编译实践报告的全部内容。

最后想说的就是，我认为编译实习项目是编译原理课程的精华，实在是非常良心的课程大作业。感谢老师和助教让我能够拥有这么好的课程作业体验。