编译器实现实践报告：Compile, complete and comprehend.

一、编译器概述

1.1 基本功能

本编译器基本具备如下功能：

1.识别sysY语言并生成KoopaIR中间代码；

2.读入生成的KoopaIR代码并生成riscv汇编代码；

3.做了对代码的一些简单优化。

1.2 主要特点

我开发的编译器的主要特点是“**麻雀虽小，五脏俱全**”、“**变量之间各得其所**”、 “**模拟栈结构**”

- **麻雀虽小**：这里指编译器的结构非常简单，没有冗余的文件结构和难懂的数据结构。主要的代码均由c++语言写成，仅仅用了最基本的库文件和koopa.h来完成任务。前后端代码之间耦合程度低，分工明确且互相独立，读入各自的文件并进行处理；同时同一代码的不同板块间依赖性也较小，方便根据已知问题找到对应模块进行增删和修改。同时，代码基本不涉及寄存器的分配，所有的变量基本都保存在栈上。
- **各得其所**：这里指处理嵌套的方式，我们给每一个代码块分配了一个自己的属性depth，与一般定义不同，这里的depth是一对一的关系，可以唯一确定一个代码块。在该代码块中定义的数据后我们统一加上_depth来唯一标识。如在depth=0中定义的变量，即全局变量g，在我们代码中存储的名字为@g_0，同时我们对函数参数和局部变量也进行了区分。这里在数据结构方面会具体讲解。
- **模拟栈结构**：这里指我们在每个函数创立时为其创建一个栈的局部缩影，存储其局部变量、返回地址和调用函数参数，实时动态分配和清空，这里和具体进行执行时不同，我们不考虑其调用者，只考虑对于其自身的栈增减。

二、编译器设计

2.1 主要模块组成

我的编译器结构简单，分工明确。源代码由sysy.l/sysy.y, AST.h/AST.cpp, visit.h/visit.cpp以及main.cpp构成。分别负责前端、中端和后端。

1.sysy.l/sysy.y负责前端读入并解析sysy源文件，前者为词法分析器，后者为语法分析器。通过这部分处理，可以将源文件处理为AST抽象语法树。

- sysy.l读取文件，并将文件的内容拆分成一个个 token 作为输出，传递给语法分析器。
- sysy.y作为语法分析器，读入前面的token序列并生成AST结构。这里我们采用的是LR分析法，该部分确定了规约规则，以及构建节点的规则，我们根据这些规则通过移入-规约分析构造树并分配节点间的关系和节点相关属性。

2.AST.h和AST.cpp负责中端的IR生成。这里我们对每个语法树的节点制定一些规则，通过对其子节点的调用以及其本身具有的属性增量式生成中间代码。我们以三地址代码作为目标生成三元式并实现静态分配。通过阅读AST.h可以清晰得知每个节点所具有的属性。基本所有节点都有一个Dump()函数，这个函数负责输出IR的内容，同时部分节点（如表示表达式的节点）有Calc()函数，用于计算【常量】表达式的值，若表达式中有变量则使用Dump()进行相关计算并返回结果的地址。

3.visit.h和visit.cpp负责后端，读入KoopaIR代码并生成riscv汇编代码。这个部分需要与koopa.h进行结合，解析KoopaIR代码并将信息存储为相关数据结构，我们的visit文件则是对parse后的结构进行分析，通过指令选择和指令调度，增量式生成riscv汇编代码。

2.2 主要数据结构

2.2.1 AST抽象语法树。

抽象语法树在AST.h中定义。所有抽象语法树节点都继承自基类BaseAST。

```
// 所有 AST 的基类
class BaseAST {
 public:
  virtual ~BaseAST() = default;
};
```

这里为了构造的简便性，我们选择在每个节点内部自己定义所需函数和数据结构。节点类的定义是根据sysy.y中提供的语法规则来定义的，其属性都是继承属性，在归约过程中予以定义和赋值。

给出一个简单的例子，对于产生式 LOrExp ::= LOrExp "||" LAndExp;

```
LOrExp
  : LOrExp OR LAndExp {
    auto l_or_exp = new LOrExpAST();
    l_or_exp->HasOp = true;
    l_or_exp->l_or_exp = unique_ptr<LOrExpAST>((LOrExpAST *)$1);
    l_or_exp->l_and_exp = unique_ptr<LAndExpAST>((LAndExpAST *)$3);
    $$ = l_or_exp;
  }
  ;
```

对应于

```
class LOrExpAST : public BaseAST {
public:
    bool HasOp;
    std::unique_ptr<LAndExpAST> l_and_exp;
    std::unique_ptr<LOrExpAST> l_or_exp;
    int depth;
```

```
    std::string Dump() const;
    int Calc();
};
```

这里我们通过该过程首先构造了一个LOrExpAST节点，然后为其分配属性。首先我们可以发现该节点有一个bool类型的属性HasOp，该属性表示被归约的表达式中有没有含有运算符（该式中特指“or”），由于对于LOrExp有两种产生式LOrExp ::= LAndExp | LOrExp "||" LAndExp， 该属性帮助我们加以区分，以便后续处理。

同时我们可以发现该节点中有两个子节点，分别对应归约的参数1和参数3，通过类似unique_ptr((LOrExpAST *)$1);的操作，可以把其转化为子节点的对应形式并进行指针赋值。如此节点之间便产生了父子关系，可以后续进行相关的调用和遍历操作。另外我们可以发现有些属性（如depth）并未在构造过程中进行赋值，这些属性是我们在生成IR时的辅助属性，目前值未知。

其他的节点结构大致相似，这里不再说明。

2.2.2 “符号表”

这里加引号的意思是我们秉持简单的原则，真的只使用了一些基础的数据结构来表示符号。这里用的最多的是unordered_map数据结构，一对一地赋予属性。在前端，我们对符号进行如下表示：

```
static std::unordered_map<std::string, int> const_val;
static std::unordered_map<std::string, short> isvar_val; // 0: const, 1: variable,
2: parameter, 3: array, 4: para_array
static std::unordered_map<std::string, bool> func_is_int;
```

前两个为符号表，存储变量的类型和值，第三个为函数表，存储一个函数的名称以及其类型。这些表都有着两个用途：记录名称、查询对应值。

具体来看：

- const_val是记录变量的值的，这里取名为const_val是该表中只有常量的值才有意义，但变量也会在此记录，这里记录为0
- isvar_val是记录变量的类型，这里类型为short，可以取五个值：0为常量，此时对应的const_val值有效；1为普通变量，不包含函数参数和数组类型；2为函数参数，按照文档中的说明，对于参数的存储和普通变量有着一些区别，我们在此特别标识；3为一般数组；4为参数数组。

需要注意的是，上面两个表的key值排序和内容完全相同，均为‘@+变量名+深度标识’我们通常先通过查isvar_val确定类型，然后在需要的时候用const_val获得常数值。

- func_is_int记录函数的类型，0标识为void、1表示为int，这关系到我们在函数定义时的说明以及返回值的说明。具体作用与isvar_val一致。

2.2.3 深度表

我们需要一个数据结构来表示嵌套调用，以清楚确定变量作用域。正如前面所讲，由于在我们的实现中depth并不是真的表示“深度”，而是作为一个一对一的标识符存在，所以我们不能直接确定该函数的嵌套关系。故我们专门设计此数据结构来存储嵌套关系：

```
static std::unordered_map<int, int> depfather;
```

采用一个类似树的方式来构造，每当发现一个新的语句块时，我们就在定义时将其条目加到该表中，第一个int表示新作用域，第二个int为当前作用域，即depfather[i]表示直接嵌套了i的语句块标号。通过建立这样类似access link的形式，我们可以递归地找到相关的数据并使用。具体的寻找方式会在实现细节中加以说明。

2.2.4 “栈帧”

在visit中，我们同样用

```
static unordered_map<koopa_raw_value_t, size_t> stack_addr;
```

来表示栈。存储每个value对应的栈中偏移位置。

同时，栈需要与相关数据配合使用，这里我们定义了四个属性A, S, R, D，它们具体的作用是：

- A: 这里处理的是相关函数调用参数较多的情况。如果超过8个则需要对其额外分配空间。
- S: 为这个函数的局部变量分配的栈空间。
- R: 在该函数有调用时取4，保存ra。
- D: 函数总共分配的空间大小。需要ceil到16的倍数。

具体来说，在每个函数定义时，我们调用alloc()函数来进行栈的分配，然后进行具体操作，然后再恢复栈帧并返回。我们要模拟实现如下的栈帧结构： [stack]{栈帧.png}

2.3 主要设计考虑及算法选择

2.3.1 符号表的设计考虑

在我们的编译器中，通过三个表（即(1)const_val、(2)isvar_val和(3)depfather）来唯一确定一个符号的相关信息。在实际书写和应用的过程中，一个符号可以表示变量、常量和数组，可能会出现在以下情景下：

1.等式中左部（定义）/右部（使用）

2.作为函数参数

3.作为表达式参数

同时，同一个符号可能出现在不同的域中，也可能调用不直接属于本域的符号。所以由此我们从多个维度来定义一个符号：

1.它是否为常量？它是否为数组？是否是函数参数？

2.它是否为全局变量？又或者说它所属的域是什么？

3.对于参数我们如何设计？

考虑以上的两点，我们用上面的数据结构来设计符号表。(1)(2)的作用已经明确，解决了问题1，而对于问题2，通过我们程序变量“各得其所”的特性，加上上面三个表就可以完成。我们实际上是将不同域中同名符号当作不

同的符号来存储的，这点在前面有说明，主要是在定义时给每个变量加上一些标识域的后缀。而在寻找时，我们根据depth和depfather递归查询直到找到最近的定义式。具体细节我们后面会补充。

对于问题3详见3.1.8

2.3.2 寄存器分配策略

我们的汇编代码中变量大多数都是在栈上存储。我们采用的方式是按照扫描顺序将返回地址、函数调用大于8的参数、局部变量（包括KoopaIR中的临时变量）存到该函数的虚拟局部栈上，并设置map索引进行查询。

除去将不大于8个的调用参数存在a0-a7中这么一个固定的分配规则，在其他实现中，我们使用的寄存器仅限于t0 t1 t2和t3，用于存储**中间值**。但我们使用这些寄存器的假设是其中存储的变量**仅在本条语句活跃，不影响任何其他语句的使用**。如在翻译getptr语句时，我们用t0存储基地址，t1存储索引值，t2存储变量宽度，然后将t1*t2作为偏移量保存至t1。该例子中，t1表示偏移量仅在该语句中活跃，故可以直接分配。

特别地，在变量个数较少的lv3相关实现中，代码中曾经设计过一种非常naive的寄存器分配算法：FirstUnusedReg()，即文档中所说的"分配, 但没完全分配"。该函数通过查询我们事先定义的一个寄存器表（即寄存器名-是否被占用）从头开始寻找一个未被占用的寄存器并返回，实现思路非常简单。

具体实现如：

```
string FirstUnusedReg()
{
  string firstTrueKey = "No";
  for (const auto& pair : reg_available) {
        if (pair.second) {
            firstTrueKey = pair.first;
            break;
        }
    }
  return firstTrueKey;
}
```

2.3.3 采用的优化策略

代码中采用了一个最基础的优化，即**消除冗余load**。由于我们是增量式生成的riscv代码，所以想要完成这一点必须记录上一条语句的情况。

由于我们处理的情况总是load紧接store后，所以具体实现是在load函数中的，如：

```
if(just_stored && store_address == address && store_i == i)
  {
    just_stored = false;
    if(store_src != dst)
      cout << "  mv  " << dst << ", " << store_src << endl;
    return;
  }
```

我们设计了一些全局变量来存储上一条语句是 sw 时的相关情况。just_stored表示上一句是否为 sw，由 store 设置，并由其他任何指令解除。当满足后一条 lw 用到了前一条 sw 的结果这个条件时，开始执行，若目的寄存器和 sw 的源寄存器一致，则跳过 lw 的生成，如果目的寄存器不一致，则用 mv 指令代替。

通过这个简单的优化，程序运行时间（perf测试）由584s提升到了487s，将近一个优化便提升了将近100s，可见还有很大的优化空间。我们的编译器还“不够好”。

2.3.4 其它补充设计考虑

三、编译器实现

3.1 各阶段编码细节

Lv1. main函数和Lv2. 初试目标代码生成

这部分是我们编写一个编译器的开端，是后续复杂代码生成的基础。在这一部分，我主要的收获有二：

一是知道了有那么多千奇百怪的代码，比如缺少return、以及后面可能int类型的函数return没有值等等，一直没有解决的一个点好像在写到lv5的时候莫名其妙解决了，于是感慨一个小小的return还和语句块有那么大的联系，同时也感受到我平时对编译器肯定没少折磨。

二是阅读了koopa.h这份代码，主要了解了相关程序是怎么处理KoopaIR代码的，将一个语句切分成相关的数据结构，并存储了相关信息。同时学会了一些相关结构的表示，比如KOOPA_RVT_INTEGER表示一个int类型的tag等等。

Lv3. 表达式

这部分是第一次感觉到上了难度，一下加了那么多的生成式，导致代码变长了很多。但这部分也是第一次仅在文档high level提示下做的，主要在于理解简单例子，并依葫芦画瓢。

关于运算的优先级，从这个部分可以看出是在前端语法分析部分就已经处理了的。比如+和*，相关的生成式是：

```
MulExp      ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
AddExp      ::= MulExp | AddExp ("+" | "-") MulExp;
```

我们可以看出，定义优先级的关键是定义一个无二义性的归约文法。这里要想归约成ADD，我们必须先归约Mul，于是实现了乘除余优先级大于加减的优先级。同时对于AND和OR我们也是这么处理的。

最后其实可以得出所有表达式的计算优先级顺序：

unary > mul > add > rel > eq > and > or

也是按照这个顺序，我们用相似的办法可以得到各个生成式。这也为后面解决二义性问题提供了思路。

然后是中端方面，我们最主要的进展是学会了用三元式来表示，而这需要我们建立、维护好临时变量的产生顺序。这里我们使用全局变量来实现这一点，我们用初值为0的全局变量now来计算当前下一条指令应该具有的标号，在每次需要存储中间变量时，我们创建一个temp来拷贝目前now的值，并用以下形式生成三地址代码：

```
int temp_c = now;
now = now + 1;
cout << "  %" << temp_c << " = " << "or" << " %" <<  temp_a << ", %" << temp_b <<
endl;
```

这里temp_a和temp_b应该是Dump()返回的，子节点传递上来的中间变量标号，temp_c是我们本次生成的，我们在前面加'%'来标识语句编号。通过这种形式我们能够正确地生成前后关联的三地址代码。

同时，这部分是第一次意识到一个好的AST树的重要性，因为优先级的限制，我们追溯一个表达式的值往往需要不断深入，直到叶节点，所以我们一定要设计好Dump()调用前对子节点的属性赋值（在第五章的depth上体现最为明显），同时也是为了防止segmentation fault。

最后，在后端方面，这部分是完全使用寄存器进行的变量分配（前文已经写出），通过具体的例子还是会发现目前寄存器的分配策略确实只能支撑很小一部分情况，安排更好的寄存器分配策略确实是需要思考的点。

Lv4. 常量和变量

这部分最主要的感悟是"栈"和"符号表"，也是整个lab做下来最重要的两个概念。多了变量，意味着我们不能直接在某地将一个符号的值存入或者取出，而是要生成一个计算表达式。通过构建const_val和isvar_val，实现了相关的识别。对于一条语句赋多个值，我们设置以下规则来表示：

VarDefList : VarDef ',' VarDefList

此时应该用一个vector<VarDefAST*>作为VarDefList的属性，来存储其已经识别的VarDef集合。

对于const，我们有专门的数据结构存储其值，而这就是它所有的信息了。对于var的处理比较麻烦，我们首先应该对其进行alloc，分配一片空间。

另一点区别于const的是，我们不能直接对val取值的直接后果是我们只能在叶节点出返回变量的名字，并逐渐往上传递。这里我们对变量存储的定义是在变量之前加一个'@'号作为标识，这是生成KoopaIR代码的需要，我们在const_val和is_val的存储最好保持一致。在Primary归约的时候，除了number和(E)，我们加了一个lval标号来表示变量符号，当运行到这个节点时，它查询符号表并判断当前变量是否为常数，是则返回 to_string(const_val[ident])， 否则类似3中做法，建立一个新的中间变量使用 load 将其加载下来。具体类似

```
int temp_d = now;
now = now + 1;
std::cout<< "  %" << temp_d << " = load " << ident <<std::endl;
```

这里ident是我们定义过的，与isvar_val的key值一致的变量名。

这里需要注意的另一点是，因为定义了const和var两种符号，所以我们的定义式中出现了

```
Decl          ::= ConstDecl | VarDecl;
```

前者是带有const标志的定义式，后者不带。对于全是const类型的元素参与求值，我们其实是直接查询符号表就可以得到的，也就是表达式的值是可以在编译时就确定的。为了减少运行的时间，我们在此引入了Calc()函

数，区别于Dump()返回子节点执行后的中间变量代码，Calc()的返回值恒为int类型，并且只会在ConstInitVal以及lv3中表达式节点中定义并调用，因为只有在这些情况下我们才能保证返回一个int值不出错。 举例来说明，考虑下面这个语句：

```
(1) int x = 7 + 4;
(2) const int y = 7 + 4;
```

需要用到AddExpAST，这个类有着两个函数：Dump()和Calc(), Dump()负责"生成计算代码"，即对(1)，应该调用Dump()，生成的应该类似

```
%0 = add 7, 4
store %0, @x
```

而对(2)，此时我们能够保证右边的表达式中的值我们都可以在const_val中查到，所以可以直接计算，这时调用Calc()，不会生成任何代码，但是会在符号表中建立一个新的符号，这个符号是常量，对应的值为11，等下一次调用y的时候即直接使用该值，减去了复杂的load操作，即 @y 这个名字不会出现在我们的IR中。

同时，这也为我们提出了一个更深远的问题，即设计相关表达式计算时选择Dump()还是Calc()的问题。这里我们在分析的时候很难确定右边的表达式中是否含有我们不能在符号表中查询到有意义的值的变量，所以我们这里只在有const标识的相关内容中调用Calc()，其余还是调用Dump()，（毕竟后者不会出错）。但实际上我们可以在更多的情况下调用Calc()，（如上面的x如果在后面使用且这之间没有重定义时），这也是一个可以考虑的优化思路。

同时，这里考虑到'='的问题，正如前文所讲，一个变量出现的时候可能是左值，也可能是右值，赋值和计算的处理方式肯定是有差别的，所以我们这里会定义当使用

```
Stmt          ::= LVal "=" Exp ";"
```

进行归约时，LVal为左值，其余时候为右值，并分别设计了一套Dump方案。但后续看下来，其实该区分没有必要，二者的Dump方案会趋于相同，所以这里不再深入。

对于后端，我们最主要的变化就是由使用寄存器的策略转成了使用栈的策略。这里我想着重强调栈分配的问题。由于我们是增量式生成的代码，所以我们必须事先扫描一个函数的所有涉及变量并计算需要分配的栈空间。这里我们就使用了alloc()函数，通过getTypeSize()函数获取其中每个变量的字节大小（其中int类型的恒为4）来进行累加栈分配，并计算出整个栈的大小（对齐到16字节）。具体分配的内容和A, S, R, D这四个变量有关。这里我们只涉及到S，即使用以下方式来记录变量的偏移，并累加计算局部变量栈帧大小。

```
if(value->kind.tag == KOOPA_RVT_ALLOC ){
    int width = getTypeSize(value->ty->data.pointer.base);
    stack_addr[value] = S;
    S += width;
    continue;
    }
```

这里需要注意的是，我们在alloc()中进行的操作相当于是预先分配，于是我们其实就不用再考虑当一条指令是ALLOC时我们应该怎么Visit了，毕竟目的已经达成了。

Lv5. 语句块和作用域

这部分我们完成了语句块和作用域的分配，这是sysy和c++相比唯一一点我们不熟悉的地方。在前端部分，比较重要的是Block的定义发生了变化，一个Block可以分成多个BlockItem。

我们将{}之间的内容看成一个语句块，语句块会创建作用域，语句块内声明的变量的生存期在该语句块内。作用域是可以嵌套的，因为语句块是可以嵌套的。这就会导致一个问题——同名变量。在这个部分主要处理的就是这个问题。

具体的处理方法其实上文已经提到，就是"一个变量一个坑"，可能在源程序中两个变量在不同的嵌套域中同名，可是在我们的符号表中坚决不允许出现这一点。所以我们进行了如下改进：

设计出depth值用来唯一标识块，同时类似前面的now和temp，我们也设计出nowdepth作为全局变量记录下一个要分配的depth值，给最外部的程序块分配0，后面每考察一个语句块都依次分配。例如在以下嵌套程序中：

```
int main()
{
    int r, s;
    int a = 1;
    int b = 2;
    {
        const int a = 3;
        r = a;
        s = b;
    }
    return r + s;
}
```

由于分别处于块1和块2，两个a在定义取名时就是不同的名字，外层的a为@a_1，里面的a为@a_2，外层的b也为@b_1，（具体的命名规则前面有提）。在使用时，我们会从当前作用域开始找，没找到就利用depfather找父作用域，看看有没有定义相关变量。具体实现上是：

```
while(isvar_val.find('@' + lval-
>ident+"_"+std::to_string(tempdep))==isvar_val.end() && tempdep >
0)tempdep=depfather[tempdep];
```

即对于a，我们在块2中就能找到，于是r=3，而对于s，需要经过一个往前递归的过程：@b_2是没有定义的->由depthfather往前找2的父域为1->找@b_1，发现有定义->s = @b_1 -> 最后返回值应该为5

另一方面，对于块嵌套，在测试时发现了另一个问题，即return问题。在子作用域中return，那么在父作用域中也应return（至少在lv5中应该这样）。为了解决这个问题，设计了一个全局变量has_return，初值为假，每次发现return语句就将其设置为真，后面的statement在分析前先检查这个变量，如果为真则不执行。

本节在后端的实现没什么不一样的。

Lv6 Lv7. if语句和while语句

本节我们第一次涉及到条件和跳转的概念。这两者有些相似，我们一起讨论。

首先，在前端，我们来回收二义性的伏笔。非常经典的悬空-else二义性，通过文档中给出的生成式可能无法正确生成AST树。如面对：

if E1 then if E2 then S1 else S2 <\center>

时。这里我们参照课上ppt的内容，将Stmt先划分成OpenStmt和MatchStmt，然后再进行归约。

具体来讲，OpenStmt涉及的规则有

```
OpenStmt : IF '(' Exp ')' Stmt
OpenStmt : IF '(' Exp ')' MatchedStmt ELSE OpenStmt
OpenStmt : WHILE '(' Exp ')' Stmt // 这个比较重要，不然会发生移入-归约错误
```

MatchedStmt涉及的规则有

```
MatchedStmt : IF '(' Exp ')' MatchedStmt ELSE MatchedStmt
// 以及其余任意由原Stmt可推出的语句
```

经过拆分，可以消除二义性，放心大胆地进行中端分析了。

在中端分析中，需要注意的是条件跳转。这里我们涉及到 br 和 jump 指令的生成。这里我们使用递归下降。对于一条if指令，我们先生成条件语句，然后进行br判断跳转，接下来分别对跳转分支生成对应编号，然后对其代码块Dump()。对于While指令，则是要在条件判断的代码生成前再加一个标号。

这里涉及到了标号的生成，所以秉持我们“一个萝卜一个坑”的原则，我们不能产生相同的标号，但在一个作用域内可能会直接生成多个if语句或while语句，这里我们的depth思想又可以派上用场了，给每个块指定一个对应的标号，然后对于相应的标号，在后面加上其专有标识符就好了。

例如：while_begin_3, while_end_4等。

承接Lv5，这个部分也涉及到语句块嵌套的问题，但与上面不同的是return方面，由于该部分是条件跳转，所以子语句块的return并不能代表着父语句块的return。所以我们给每个语句块节点加一个属性in_condition，用来表示其是否是条件跳转到的语句块，在这样的语句块里面return不会设置has_return。其余与Lv5相同。

在后端生成方面，我们需要处理br和jump指令，不是很难，用true/false/target来判断指定语句然后产生bnez/jump语句即可。

在这一块中，我们发现了前面的一个遗留问题，那就是立即数的范围是有限的[-2048, 2047]，当我们的栈分配过大时，用前面的addi sp, sp, imm 已经不能满足要求，这里我们采用的方式是若imm不满足范围则用t3寄存器作为缓冲，用

```
li t3, imm
add sp, sp, t3
```

来代替前面的addi指令。同理，对于 sw 和 lw 指令我们也要这么考虑。再考虑回来，br指令的跳转范围也有限制，我们在中端生成可能会有错误，所以我们使用jump来代替br。例子如：

```
br %0, %if_then0, %if_else0
%if_then0:
  %1 = xxxx
  jump %if_end0
%if_else0:
  %2 = xxxx
  jump %if_end0
```

另外，该部分涉及到条件表达式的短路求值，以对a||b计算为例，如果a已经是true，则b不该被计算，计算结果为true；只有a为false，b才应该被计算。下面的LOrExpAST::Dump正是要做这样的短路求值。先分配一个具名符号result = 1, 计算左边的值lhs，如果非零，则直接跳转到结尾；如果为零，计算rhs的值，并将rhs规范化存入result中。最终程序返回result中的值。类似该思路，我们在LAndAST和LOrAST中根据短路求值的规则，构造虚拟的if表达式，然后再按照这个进行跳转即可。（具体见这两个块中的Dump()函数）

这里需要注意的是我们需要自己生成一个变量，来记录当前分析到的位置的真值，这里为了避免与用户定义变量重复，我使用了@tempx变量，如@temp1，与用户的不同是没有了中间的'_', x和上面的depth一样，是一个重新专门为短路分配的全局变量short_count值。通过短路，生成的IR和汇编代码都变长了，但运行时间极大缩短了。

Lv8. 函数和全局变量

首先关于全局变量，其实是定义了一种新的IR生成模式，在每个变量的定义节点上添加is_global，来标识其是否为全局变量，如果是则用全局变量的分配方式。具体的内容是

```
global @var = alloc i32, 233
```

（对于const我们照样不生成）

这里我们可以确定的是，即使是Var变量，其作为全局变量的时候，值也是可以唯一确定的。所以现在一定要确定其要分配的值，而且其初始化定义时右侧为表达式时，我们必须计算出值，使用Calc()。但var值在const_val中无意义，怎么判定呢？这里我们先设下悬念。

同时我们增加了一些需要链接的函数，在KoopaIR生成过程中我们需要把它们加入到func_type这个表中，以便后续直接调用。而在riscv代码生成中我们需要忽略。

这部分在visit函数中我们加了一个void VisitGlobalVar(koopa_raw_value_t value);的函数，其主要内容是根据规则生成对应的.data块，并用zero/word进行初赋值。同时我们在访问全局变量的时候需要用 la 指令将其地址加载到t0中，然后再用0(t0)取得值。

然后是函数部分。这部分比较棘手的是参数的传递，为此我们甚至专门将isvar_val的bool位升级成了short位来专门标识其是否为参数。主要原因是我们需要在函数内为每个参数变量重新分配一块同名存储空间（如@x->%x），生成 store @x, %x 这样的语句。但我们在符号表中存放的符号仍然是@x，这就需要标识位了，当isvar_val[@+ident+_depth]值为2时，我们对相关变量应该更名为%+ident+_depth来进行传递。

在参数调用时，我们设置了vector型func_params来存储参数，其内部实际上存储了一个个表达式的返回值。然后我们生成call语句，然后逐个写入参数名和类型（这时肯定都是int）。这样就完成了call。

然后这节还有一个特点是我们生成了多个函数，它们之间可以互相调用，等级相同。所以我们的has_return相关逻辑也要改变。当我们进入一个函数生成%entry时，我们要顺带把has_return设成false标明这是我的地盘，然后再进行后续的分析。

接下来我们其实可以发现has_return现在变成了函数与函数之间的空格标志，当has_return为false是一定标明我们现在处于depth=0的全局状态。而对于本节一开始设下的悬念，正好可以用这个方式解决。当我们遇到类似

```
//global
int x = 8;
int y = x + 3;
```

这样的情况时，我们可以把全局变量的值先存到const_val中，且该值不会再因为函数中相关赋值而变化。当我们遇到x+3这样的表达式时，检查has_return，如果为true那就证明我们在缝隙中，var值存在时合法的，于是直接调用x+3计算出11并存到const_val[@y_0]中即可。

对于此部分的后端生成，栈的应用我们之前已经说的比较完整了。在这里我们主要利用栈增加了对call指令的处理。这里主要涉及一个参数存寄存器的问题。前面在栈分配的过程中讲到了A和R的作用，A所有存储调用过程中大于8个的参数的个数的最大值，R负责有调用的时候分配4的返回地址空间，在这时就有用了。对不大于8个的参数，我们用寄存器a0-a7来存储，并用 _save("t0", "sp", (i-8) * 4);将大于8个的参数存储。然后我们生成call callee_name即可。

Lv9. 数组

这个部分是前面所有部分的集大成者。特别是与函数调用的结合将Lv8的难点，如参数、全局变量、函数调用全都卷土重来了一遍，而且更加残忍。

这里对const数组和val数组的处理基本相同，我们下面将以val数组为例来说明处理要点。我们在考虑一个数组时，它可以是哪些角色呢?

*全局变量

*（调用别的函数传入的）参数

*调用时被传入的参数

*局部变量

首先考虑作为局部变量。这包含了定义一个数组最基本的操作。首先，数组可以是一维，也可以是多维。

- 考虑定义：我们在定义变量的时候为其添加一个vector类型的属性const_exps，其中包含了一些定义表达式，来表示各个维度的长度。例如a[3*4][6][10]在定义时ident为a，const_exps为{12，6，10}。同时，对数组的初始化是一大难点，尤其是多维数组。可以是空初始化，可以指定某些变量元素。我们要正确地对初始化数组补0，如以下这种情况：

```
int arr[2][3][4] = {1, 2, 3, 4, {5}, {6}, {7, 8}};
```

我们用getInitVal这个函数来进行补全。这种不好理解的定义要求当前已经填充完毕的元素的个数必须是 Len(n) 的整数倍。我们在这里的处理是先将多维数组铺平，计算出整个数组的长度，创建一个等长的init数组来表示初始值，然后用“0”填满该数组，再创建一个等长的record数组，来记录每一维对应的宽度，如上面的arr的record数组应为{24，12，4}，将record传入函数作为参数。接着我们根据初始化值对init进行修改。用next记录现在填到的位置，初始为0，然后观察初始数组中每个元素，若是int就直接加入，若是另一个初始块则进行递归调用。这里我们需要找到能够兼容该初始块的最小维度，并要求当前next可以整除record最小维，即最后一维，然后我们依次尝试用next除record的每个维度，当不能整除时证明找到，回退并将能整除那一部分的record作为新的参数传入，进行递归寻找填补，接着等到递归返回，next应当加上目前填补的维度宽度。（我已经尽力解释了）具体函数见下：

```
void ConstInitValAST::getInitVal(std::string*ptr, const std::vector<int> &record)
const {
    int n = record.size();

    int next = 0;
    for(auto &init_val : inits){
        init_val->depth = depth;
        if(init_val->is_exp){
            ptr[next] = to_string(init_val->Calc());
            next++;
        } else {
            int j = 0;
            int deltaw = record[n-2];
            if(next != 0){
              for(; j <= n-1; ++j){
                  if(next % record[j] != 0)
                      break;
              }
              j--;
              deltaw = record[j];
            }
            init_val->getInitVal(ptr + next, vector<int>(record.begin(),
record.end()-j));
            next += deltaw;
        }
        if(next >= record[n-1]) break;
    }
}
```

然后我们需要用这个init数组来进行全局初始化。这里要涉及到两个新的关于数组的KoopaIR语句：getptr和getelemptr。 getelemptr 指令的用法类似于 getelemptr 指针, 偏移量。 其中，指针必须是一个数组类型的指针*[T, len]，getelemptr ptr, index 计算新指针ptr + index * sizeof(T)，返回*T；而getptr返回的指针和getelemptr完全相同，但其类型仍为*[T, len]。我们将用这两种指令的组合处理多种复杂情况。首先应用到我们的初始化函数store_array中，该函数接受init并由此生成初始化arr的语句，具体如下：

```
void store_array(std::string name,  std::string* init, std::vector<int> len, int*
k, int i)
{
```

```
  int layer_size = len[i];
  for(int j = 0; j < layer_size; j++)
  {
    if(init[*k] == "0" && i == len.size()-1)
    {
      *k = *k+1;
      continue;
    }
    int temp = now;
    now = now + 1;
    std::cout<< "  %" << temp << " = getelemptr "  << name << ", " << j <<
std::endl;
    string subname = "%" + std::to_string(temp);
    if(i < len.size()-1)
      store_array(subname, init, len, k, i+1);
    else
    {
      cout << "  store " << init[*k] << ", " << subname << endl;
      *k = *k + 1;
    }
  }
}
```

简单来说，我们用递归调用的方法层层初始化，为了避免对于太长维的数组导致IR过长，我们选择对“0”初始化的维度跳过。对非零初始化，主要思想就是潜入到最底层（中间伴随着getelemptr指令的生成）然后找到要填写的位置，填入。 另外，还需要注意数组类型的生成，类似[[[T1, len1], len2], len3]这种形式，我们仍然是采用递归的方式生成这种嵌套类型的字符串。

- 考虑使用和赋值，我们应当用上面的getelemptr找到要赋值的元素的指针，然后用新值填充或者将其存入中间变量中以便使用。
- 接着，对于全局数组，我们初始化的方式比较简单，对于有初值的数组，生成global @arr = alloc [i32, 3], {1, 2, 3}这样的形式，这需要我们计算出arr的类型和init数组（上面说过），然后根据数组形状来输出初始化值（就是加一些‘，’和‘{}’）；而对于没有初始化的全局数组，应当用global @arr = alloc [i32, 3], zeroinit的形式。总体来说在上面的基础上还是很简单的。
- 接着，对于参数传递，这个比较费劲。因为我们在函数之间传递的并不是真正的数组，而是一个指针。说起来比较麻烦，我们通过一个表格来表示情况：（行表示定义方式，列表示使用方式）| | 正常定义 | 为函数参数 | |:-------- | :------------ | :---------------- | |正常使用（右值） | getelemptr()直到所需维度(这里肯定是最后一维） |首先load下来，然后对第一维取用用getptr()，后面用getelemptr()到所需维度（这里肯定是最后一维） | |正常使用（左值） |getelemptr()直到所需维度(这里肯定是最后一维） | 首先load下来，然后对第一维取用用getptr()，后面用getelemptr()到所需维度（这里肯定是最后一维） | |作为函数参数使用 |一直用getelemptr()直到最后一维，最后加一次偏移量为0的getelemptr() |首先load下来，然后一直用getelemptr()直到倒数第二维，对最后一维先用getptr()，再用偏移量为0的getelemptr()取第一个元素的指针|
- 按照以上方式考虑结合使用两个指令即可。具体函数参考LValAST::Dump()和StmtAST::Dump()-ASSIGN部分。

然后我们考虑后端：这部分我们首先对于global_array的分配专门设计了一个新函数InitGlobalArray，该函数通过递归调用的方法初始化有定义的全局数组。即遍历前面生成的带有{}和，的初始化init数组，对于每个元素i生成.word i。

对于zeroinit，我们需要递归调用getTypeSize计算该数组的大小（展平成一维），然后乘上4，得到zeroinit的空间r，用.zero r来初始化。

接下来我们要处理两个新的指令getelemptr和getptr。二者的计算方法完全一致，为base+offset * width。这里我们固定用t0存放基地址，t1存放索引值，t2存放宽度，接下来t1存放t1*t2，t0存放t0+t1，得到的目标地址存在t0中，并存入栈。

3.2 工具软件介绍

1. Make：自动化的编译工具。make 工具通过一个称为 Makefile 的文件来完成并自动维护编译工作。Makefile 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。编写好Makefile后，在命令行简单敲make指令即可完成编译，避免了手动输入大量指令的麻烦。
2. Flex：词法分析的工具。基于正则表达式的匹配，得到token流。
3. Bison：语法分析的工具。与Flex配合使用，指定上下文无关文法，通过LR分析，完成对源代码的语法分析，可以得到语法分析树。
4. koopa 和 libkoopa：助教团为Koopa IR 开发了对应的框架在使用 C/C++/Rust 编程时，可以直接调用框架的接口，实现 Koopa IR 的生成/解析/转换。

3.3 测试情况说明

- 首先说明在平台上测试的情况，截至目前riscv得分99.55，有一个样例在koopa编译时错误，但所有正确生成KoopaIR的情况都可以正确生成riscv汇编代码。性能测试排名13
- 接下来说些仍记得的样例构造。首先，关于return，构造了许多奇奇怪怪的return情况，在各种语句块之内return、条件语句return、return不是很正规的情况都有构造尝试。
- 然后关于数组构造，当数组的使用产生问题时，构造了很多数组的使用方法，包括上面提到的表格六种情况，分别查看生成的KoopaIR和riscv。
- 还有很多其他的用例，通过具体分析出的bug来构造，如构造特别长的局部数组导致栈长度大于2048等，目前时间较久有些记不清了。
- 值得注意的是，在调试过程中，我使用了rars来帮助我查看汇编代码的生成问题，我可以在其上运行riscv汇编代码，单步运行、设置断点、并实时查看寄存器和栈的值，这对我debug有着很大的帮助，尤其是在数组分配的阶段。