Stage-4

2019011008

无92 刘雪枫

目录

Step9

代码修改

支持函数声明和定义

修改词法与语法规则

对于词法分析，由于参数列表需要使用逗号 `,` 分隔，因此增加新的 token：`COMMA` 表示逗号 `,`：

```
","          { return yy::parser::make_COMMA  (loc);       }
```

对于语法分析，增加函数声明和定义的语法规则，实现函数声明和参数列表：

```
  FuncDefn             : Type IDENTIFIER LPAREN FormalList RPAREN CompStmt {
                       Type IDENTIFIER LPAREN FormalList RPAREN SEMICOLON{
                           $$ = new ast::FuncDefn($2,$1,$4,new
ast::EmptyStmt(POS(@6)),POS(@1));
                       }
- FormalList           :  /* EMPTY */
-                      {$$ = new ast::VarList();}
+ FormalList           : ParamList
+                          {$$ = $1;}
+                      | /* EMPTY */
+                          {$$ = new ast::VarList();}
+                      ;
+ ParamList            : Type IDENTIFIER
+                          {$$ = new ast::VarList();
+                           $$->append(new ast::VarDecl($2,$1,POS(@1)));
+                          }
+                      | ParamList COMMA Type IDENTIFIER
+                          {$1->append(new ast::VarDecl($4,$3,POS(@3)));
+                           $$ = $1;
+                          }
                       ;
```

其中，新增的非终结符 `ParamList` 声明为 `ast::VarList` 类型：

```
%nterm<mind::ast::VarList*> FormalList ParamList
```

建立符号表

由于本 step 中支持了函数声明，因此符号表需要支持函数声明，使其与函数定义不冲突。

首先，在函数符号 `FunctionObject` 中增加 `is_decl` 成员表明该函数符号是否是一个声明：

```
  class Function : public Symbol {
      /* ... */
+     // Is decl
+     bool is_decl;
      /* ... */
  };
```

然后，在建立符号表时初始化该成员变量：

```
  void SemPass1::visit(ast::FuncDefn *fdef) {
      /* ... */
      Function *f = new Function(fdef->name, t, fdef->getLocation());
      fdef->ATTR(sym) = f;
+     f->is_decl = fdef->forward_decl;
      /* ... */
  }
```

然后判断是否在之前是否已经声明过该函数。如果没有声明过则加入符号表。如果之前声明过，则判断两者是否都是函数定义。如果两者都是函数定义，则判定为符号冲突，报错；如果两者都不是定义，或前者是定义本次只是声明不是定义，则什么也不做；如果前者是声明本次是定义，则舍弃本次构建的函数符号，而使用前一次声明所加入的函数符号。这是因为在之后的翻译过程中，只给函数定义生成 `label` 而不给前置声明生成 `label`，并将该 `label` 与函数符号 `Function` 关联起来。而如果在前置声明和函数定义之间有对该函数的调用，且前置声明和函数定义不使用同一个函数符号 `Function`，则该调用处看不到后续定义的符号而导致找不到该 `label`。因此让本次的函数定义去使用前置声明的 `label`。而由于两次声明的参数名字可能不同，因此用本次函数定义所声明的参数名替换掉前置声明的参数名。

随后，如果本次是函数定义，则对函数体的语句进行遍历：

```
void SemPass1::visit(ast::FuncDefn *fdef) {
    /* ... */

    scopes->open(f->getAssociatedScope());

    for (ast::VarList::iterator it = fdef->formals->begin();
         it != fdef->formals->end(); ++it) {
        (*it)->accept(this);
        f->appendParameter((*it)->ATTR(sym));
    }

    scopes->close();

    Symbol *sym = scopes->lookup(fdef->name, fdef->getLocation(), false);
    if (NULL != sym) {
        if (!sym->isFunction()) {
            issue(fdef->getLocation(),
                  new IncompatibleError(f->getType(), sym->getType()));
        } else {
            Function *org_func = static_cast<Function *>(sym);
            f->getType()->compatible(org_func->getType());
```

```
            if (!f->getType()->compatible(org_func->getType())) {
                issue(fdef->getLocation(),
                      new DeclConflictError(fdef->name, org_func));
            } else if (!f->is_decl && !org_func->is_decl) {
                issue(fdef->getLocation(),
                      new DeclConflictError(fdef->name, sym));
            } else if (!f->is_decl) {
                auto argList = org_func->getType()->getArgList();
                *argList = std::move(*f->getType()->getArgList());
                fdef->ATTR(sym) = org_func;
                delete f;
            }
        }
    } else {
        scopes->declare(f);
    }

    if (!f->is_decl) {
        scopes->open(f->getAssociatedScope());

        // adds the local variables
        for (auto it = fdef->stmts->begin(); it != fdef->stmts->end(); ++it)
            (*it)->accept(this);

        // closes function scope
        scopes->close();
    }
}
```

注意到，在 `compatible` 函数中检查两个函数类型是否兼容时，函数内未对无参函数进行判断，导致无参函数在检查时出现对空指针解引用的 BUG，因此加入判断：

```
  bool FuncType::compatible(Type *t) {
          /* ... */
+         if (arglist->length() == 0)
+             return true; // no need to compare further
          /* ... */
  }
```

随后，在进行类型检查时，只对函数定义的函数体内部语句检查其类型：

```
void SemPass2::visit(ast::FuncDefn *f) {
    if (!f->forward_decl) {
        ast::StmtList::iterator it;
        retType = f->ret_type->ATTR(type);

        scopes->open(f->ATTR(sym)->getAssociatedScope());
        for (it = f->stmts->begin(); it != f->stmts->end(); ++it)
            (*it)->accept(this);
        scopes->close();
    }
}
```

生成三地址码（不含传参）

然后生成函数的三地址码。此步骤暂时不进行传参操作，传参的三地址码生成将在之后统一编写。

首先在翻译整个 `Program` 时加入全局作用域：

```
  void Translation::visit(ast::Program *p) {
+     scopes->open(p->ATTR(gscope));
      for (auto it = p->func_and_globals->begin();
           it != p->func_and_globals->end(); ++it)
          (*it)->accept(this);
+     scopes->close();
  }
```

在翻译函数声明结点 `FuncDefn` 时，由于有多次函数声明，在生成函数入口标签 `entryLabel` 时加入是否已经生成过的判断，并且加入对是否是前置声明的判断，如果不是前置声明才进行函数体代码的生成：

```
  void Translation::visit(ast::FuncDefn *f) {
      Function *fun = f->ATTR(sym);

      // attaching function entry label
-     fun->attachEntryLabel(tr->getNewEntryLabel(fun));
+     if (fun->getEntryLabel() == NULL) {
+         fun->attachEntryLabel(tr->getNewEntryLabel(fun));
+     }
+
+     if (f->forward_decl) {
+         return;
+     }
+
+     scopes->open(fun->getAssociatedScope());

      /* ... */

+     scopes->close();
  }
```

同时在之前的步骤中实现的对 `for` 语句和复合语句的 `scope` 也在此三地址码翻译中补全（此前由于翻译过程中并没有用到符号表查找而不需要对 `scope` 操作，现在需要补足）：

```
  void Translation::visit(ast::ForStmt *s) {
  // 与 void Translation::visit(ast::CompStmt *c) {
      /* ... */
+     scopes->open(c->ATTR(scope));
      /* ... */
+     scopes->close();
  }
```

支持函数调用表达式

支持函数调用表达式语法规则

为了支持函数调用表达式，首先增加函数调用表达式结点 `CallExpr`：

```
class CallExpr : public Expr {
  public:
    CallExpr(std::string func_name, ExprList *args, Location *l);

    virtual void accept(Visitor *);
    virtual void dumpTo(std::ostream &);

  public:
    std::string func_name;
    ExprList *args;
};
```

```
CallExpr::CallExpr(std::string func_name, ExprList *args, Location *l) {
    setBasicInfo(CALL_EXPR, l);
    this->func_name = func_name;
    this->args = args;
}

void CallExpr::accept(Visitor *v) { v->visit(this); }

void CallExpr::dumpTo(std::ostream &os) {
    ASTNode::dumpTo(os);
    newLine(os);
    os << "call " << this->func_name;
    newLine(os);
    os << this->args << ")";
    decIndent(os);
}
```

并修改相关 `Makefile` 使其加入编译和链接（修改略）。

修改语法规则，使其支持函数调用表达式 `PostfixExpr`：

```
NonNullExprList     : Expr
                        { $$ = new ast::ExprList();
                          $$->append($1); }
                    | NonNullExprList COMMA Expr
                        { $1->append($3);
                          $$ = $1; }
                    ;
ExprList            : /* empty */
                        { $$ = new ast::ExprList(); }
                    | NonNullExprList
                        { $$ = $1; }
                    ;
UnaryExpr           : PostfixExpr
                    | MINUS UnaryExpr  %prec NEG
                        { $$ = new ast::NegExpr($2, POS(@1)); }
                    | LNOT UnaryExpr
                    ;
PostfixExpr         : PrimaryExpr
                    | IDENTIFIER LPAREN ExprList RPAREN
```

```
                        { $$ = new ast::CallExpr($1, $3, POS(@1)); }
                    ;
```

新增的非终结符类型：

```
%nterm<mind::ast::ExprList*> ExprList NonNullExprList
%nterm<mind::ast::Expr*> PostfixExpr
```

增加类型检查

接下来为函数调用表达式增加类型检查。首先为访问者基类增加对 `CallExpr` 的访问函数：

```
  class Visitor {
+     virtual void visit(CallExpr *) {}
  };
```

```
  class SemPass2 : public ast::Visitor {
+     virtual void visit(CallExpr *) {}
  };
```

首先，对于函数调用表达式 `func(arg0, arg1, ...)` 来说，`func` 必须是函数类型，而且实参的个数必须与形参一致，并且每个参数必须进行类型检查，即均为 `int` 类型。因此，类型检查代码如下：

```
void SemPass2::visit(ast::CallExpr *e) {
    Symbol *sym = scopes->lookup(e->func_name, e->getLocation());
    if (NULL == sym) {
        issue(e->getLocation(), new SymbolNotFoundError(e->func_name));
        goto issue_error_type;
    }
    if (!sym->isFunction()) {
        issue(e->getLocation(), new NotMethodError(sym));
        goto issue_error_type;
    }
    {
        Function *func = static_cast<Function *>(sym);
        if (func->getType()->numOfParameters() != e->args->length()) {
            issue(e->getLocation(), new BadArgCountError(func));
            goto issue_error_type;
        }

        for (auto itr = e->args->begin(); itr != e->args->end(); ++itr) {
            (*itr)->accept(this);
        }
        e->ATTR(type) = func->getResultType();
        return;
    }

issue_error_type:
    e->ATTR(type) = BaseType::Error;
}
```

增加三地址码生成函数

在 `Translation` 类中增加函数：

```
  class Translation : public ast::Visitor {
      /* ... */
+     virtual void visit(ast::CallExpr *);
      /* ... */
  };
```

具体实现将在后文叙述。

增加三地址码

设计三地址码格式

为了实现函数调用，拟增加三种三地址码：`CALL`、`ARG`、`PARAM`。三者的格式为：

```
CALL func
ARG T0
PARAM T0
```

具体说明如下：

- `CALL`：调用过程 `func`
- `ARG`：用在 `CALL` 之前，用于将寄存器 `T0` 的值传进参数，参数从左向右传递
- `PARAM`：用于函数内部，将参数取出到 `T0` 寄存器中，参数从左向右取出

需要注意的是，此处设计的三地址码的参数传递顺序是从左向右的，而要实现之后 RISC-V 的从右向左传栈的调用约定则由从三地址码转换成汇编的步骤来实现，三地址码本身不必考虑这件事情。

例如，下面的 C 程序：

```
int func(int a, int b) {
    return a + b;
}

int main() {
    return func(1, 2);
}
```

一种可能的三地址码是：

```
func:
    PARAM T0
    PARAM T1
    T2 <- ADD T0, T1
    return T2

main:
    T3 <- 1
    T4 <- 2
    ARG T3
    ARG T4
    T5 <- CALL func
    return T5
```

增加三地址码代码

首先增加枚举类型：

```
  struct Tac {
      /* ... */
+     ARG,
+     PARAM,
+     CALL,
      /* ... */
  };
```

然后增加生成该三地址码的函数：

```
  struct Tac {
      /* ... */
+     static Tac *Arg(Temp arg);
+     static Tac *Param(Temp param);
+     static Tac *Call(Temp dest, Label entry);
      /* ... */
  };
```

```
Tac *Tac::Arg(Temp arg) {
    REQUIRE_I4(arg);
    Tac *t = allocateNewTac(Tac::ARG);
    t->op0.var = arg;
    return t;
}

Tac *Tac::Param(Temp param) {
    REQUIRE_I4(param);
    Tac *t = allocateNewTac(Tac::PARAM);
    t->op0.var = param;
    return t;
}

Tac *Tac::Call(Temp dest, Label entry) {
    REQUIRE_I4(dest);
    Tac *t = allocateNewTac(Tac::CALL);
    t->op0.var = dest;
    t->op1.label = entry;
    return t;
}
```

然后编写其序列化格式：

```
  void Tac::dump(std::ostream &os) {
      /* ... */

+     case ARG:
+         os << "    arg  " << op0.var;
+         break;
+
+     case PARAM:
+         os << "    param " << op0.var;
```

```
+         break;
+
+     case CALL:
+         os << "    " << op0.var << " <- call " << op1.label;
+         break;

      /* ... */
  }
```

在 `TransHelper` 中增加生成三地址码的接口：

```
  class TransHelper {
      /* ... */
      // Bitwise
      Temp genBNot(Temp);
+     // Argument
+     void genArg(Temp);
+     // Param
+     void genParam(Temp);
+     // Call
+     Temp genCall(Label);
      // Memory Access
      Temp genPop(void);
      /* ... */
  };
```

```
void TransHelper::genArg(Temp arg) { chainUp(Tac::Arg(arg)); }
void TransHelper::genParam(Temp param) { chainUp(Tac::Param(param)); }
Temp TransHelper::genCall(Label entry) {
    Temp c = getNewTempI4();
    chainUp(Tac::Call(c, entry));
    return c;
}
```

维护三地址码的数据流信息

在 `dataflow` 中定义新定义的三个三地址码的数据流信息。注意到，`CALL` 和 `PARAM` 都只设计一个新寄存器的写入，而 `ARG` 只涉及到读一个新寄存器，因此规定如下：

```
void BasicBlock::computeDefAndLiveUse(void) {
        /* ... */
        case Tac::CALL:
        case Tac::PARAM:
            updateDEF(t->op0.var);
            break;

        case Tac::ARG:
            updateLU(t->op0.var);
            break;
        /* ... */
}
```

```
void BasicBlock::analyzeLiveness(void) {
        /* ... */
        case Tac::CALL:
        case Tac::PARAM:
            if (NULL != t_next->op0.var)
                t->LiveOut->remove(t_next->op0.var);
            break;

        case Tac::ARG:
            t->LiveOut->add(t_next->op0.var);
            break;
        /* ... */
}
```

生成三地址码

为函数调用表达式生成三地址码

然后为函数调用表达式生成三地址码。在遍历参数，对参数求值的代码生成过后，即生成 `ARG` 三地址码传参，然后生成 `CALL` 三地址码调用函数：

```
void Translation::visit(ast::CallExpr *e) {
    for (auto itr = e->args->begin(); itr != e->args->end(); ++itr) {
        (*itr)->accept(this);
    }
    for (auto itr = e->args->begin(); itr != e->args->end(); ++itr) {
        tr->genArg((*itr)->ATTR(val));
    }

    Function *func =
        static_cast<Function *>(scopes->lookup(e->func_name, e->getLocation()));
    e->ATTR(val) = tr->genCall(func->getEntryLabel());
}
```

为函数定义生成加载形参三地址码

在之前的函数定义中，生成三地址码用于加载形参：

```
  void Translation::visit(ast::FuncDefn *f) {
     /* ... */

     scopes->open(fun->getAssociatedScope());

     /* ... */

     tr->startFunc(fun);

     // You may process params here, i.e use reg or stack to pass parameters
+    for (auto it = f->formals->begin(); it != f->formals->end(); ++it) {
+        tr->genParam((*it)->ATTR(sym)->getTemp());
+    }

     /* ... */

     scopes->close();
 }
```

增加汇编指令

然后，增加与函数调用与立即数加 RISC-V 指令。

首先，增加枚举成员 `ADDI` 和 `CALL`：

```
      struct RiscvInstr : public Instr {
          /* ... */
          GEQ,
-         ASSIGN
+         ASSIGN,
+         ADDI,
+         CALL,
          // You could add other instructions/pseudo instructions here
      } op_code; // operation code
```

然后生成 RISC-V 汇编：

```
void RiscvDesc::emitFuncty(Functy f) {
    /* ... */

    case RiscvInstr::ADDI:
        oss << "addi" << i->r0->name << ", " << i->r1->name << ", " << i->i;
        break;

    case RiscvInstr::CALL:
        oss << "call _" << i->l;
        break;

    /* ... */
}
```

此处需要注意在本框架内，函数的入口标签是加下划线 `_` 的，因此在生成 `call` 指令时也应当加 `_`。

根据三地址码生成汇编指令

前置工作

首先，在汇编相关头文件 `riscv_md.hpp` 中定义指针大小便于编程：

```
+ #define POINTER_SIZE 4
```

然后，为了避免覆盖参数寄存器以及 `callee_saved` 寄存器，为简化代码编写，本步禁用 `callee_saved` 寄存器和参数寄存器，参数寄存器仅用于存参数。因此，在 `riscv_md.cpp` 中是否为通用寄存器的位置将它们的 `true` 改为 `false`，代码略。

之后为了将三地址码翻译成汇编指令，增加对三地址码的分类处理：

```
class RiscvDesc : public MachineDesc {
    /* ... */
    // translates a Arg TAC into assembly instructions
    void emitArgTac(tac::Tac *);
    // translates a Param TAC into assembly instructions
    void emitParamTac(tac::Tac *);
    // translates a Call TAC into assembly instructions
    void emitCallTac(tac::Tac *);
    /* ... */
};
```

```
void RiscvDesc::emitTac(Tac *t) {
    /* ... */

    case Tac::ARG:
        emitArgTac(t);
        break;

    case Tac::PARAM:
        emitParamTac(t);
        break;

    case Tac::CALL:
        emitCallTac(t);
        break;

    /* ... */
}
```

翻译 `ARG` 三地址码

`ARG` 三地址码用于传递实参。前八个参数在 RISC-V 中是由寄存器传递的，因此可以直接根据参数的序数翻译成相应的 RISC-V 汇编代码。而后面的参数则需要从右向左传栈，但三地址码是从左向右规定的，因此采用栈结构，以达到将其反序的效果。即从第九个参数开始，每遇到一个 `ARG` 参数就将其放入栈中，等到 `CALL` 指令位置再逐个生成其代码。此外还需要一个变量用于计数，记录当前是第几个参数。

计数变量和栈定义如下：

```
class RiscvDesc : public MachineDesc {
      /* ... */
    int _lastUsedParamReg = 0;   // which parameter register was used last?
    mind::util::Stack<mind::tac::Temp>
        _extraArgs;              // extra arguments for a function call
      /* ... */
};
```

则翻译代码如下：

```
void RiscvDesc::emitArgTac(Tac *t) {
    if (this->_lastUsedParamReg < 8) {
        passParamReg(t, this->_lastUsedParamReg);
    } else {
        this->_extraArgs.push(t->op0.var);
    }
    ++this->_lastUsedParamReg;
}
```

翻译 `CALL` 三地址码

保存 `caller_saved` 寄存器

在调用函数之前，首先要保存 `caller_saved` 寄存器。需要保存的有各个参数寄存器 `a0` ~ `a7` 和各个临时寄存器 `t0` ~ `t6`。具体做法是先分配足够的栈空间，然后将寄存器的值写入。注意只需要保存位于 `LiveOut` 集合中的寄存器即可，并且记录其保存过以便于之后的恢复。代码如下：

```
// Save caller saved registers
const int callerSavedRegs[] = {
    RiscvReg::A0, RiscvReg::RA, RiscvReg::T0, RiscvReg::T1,
    RiscvReg::T2, RiscvReg::A1, RiscvReg::A2, RiscvReg::A3,
    RiscvReg::A4, RiscvReg::A5, RiscvReg::A6, RiscvReg::A7,
    RiscvReg::T3, RiscvReg::T4, RiscvReg::T5, RiscvReg::T6,
};
constexpr int numCallerSavedRegs =
    sizeof(callerSavedRegs) / sizeof(callerSavedRegs[0]);
bool saved[numCallerSavedRegs] = {0};
constexpr int allocSavedMemSize =
    (numCallerSavedRegs * POINTER_SIZE + 0x0F) & ~0x0F; // 16 bytes
addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
         -allocSavedMemSize, EMPTY_STR,
         "Alloc for Saving caller saved registers");

for (int i = 0; i < numCallerSavedRegs; ++i) {
    if (t->LiveOut->contains(_reg[callerSavedRegs[i]]->var)) {
        addInstr(RiscvInstr::SW, _reg[callerSavedRegs[i]],
                 _reg[RiscvReg::SP], NULL, i * POINTER_SIZE, EMPTY_STR,
                 NULL);
        saved[i] = true;
    }
}
```

传参

由于大于八个的参数尚未生成汇编代码，只是放到了栈里，因此还需要传递大于八个的参数：

```
// pass extra parameters
const int allocMemSize = ((this->_extraArgs.size() * POINTER_SIZE) + 0x0F) &
    ~0x0F; // align to 16 bytes
if (allocMemSize != 0) {
    addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
             -allocMemSize, EMPTY_STR, "Alloc extra space for parameters");
    while (!this->_extraArgs.empty()) {
        Temp arg = this->_extraArgs.top();
        this->_extraArgs.pop();
        auto r = getRegForRead(arg, 0, t->LiveOut);
```

```
        addInstr(RiscvInstr::SW, _reg[r], _reg[RiscvReg::SP], NULL,
                 this->_extraArgs.size() * POINTER_SIZE, EMPTY_STR,
                 "Pass extra parameter");
    }
}
```

调用函数

然后生成函数调用指令：

```
// call function
addInstr(RiscvInstr::CALL, NULL, NULL, NULL, 0, t->op1.label->str_form,
         NULL);
```

释放因传递参数使用的栈空间

首先释放传递八个以上参数使用的栈空间，并将参数计数置为零：

```
// restore stack pointer for param
if (allocMemSize != 0) {
    addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
             allocMemSize, EMPTY_STR, "Restore stack pointer");
}

// reset
this->_lastUsedParamReg = 0;
```

接收返回值及恢复 `caller_saved` 寄存器

然后恢复 `caller_saved` 寄存器。注意到由于 `a0` 是返回值寄存器，因此暂不恢复 `a0`。

再接收返回值。最后如果返回值寄存器不是 `a0`，且 `a0` 被保存过，则恢复 `a0`。

代码如下：

```
// restore caller saved registers except a0
for (int i = 0; i < numCallerSavedRegs; ++i) {
    if (saved[i] && callerSavedRegs[i] != RiscvReg::A0) {
        addInstr(RiscvInstr::LW, _reg[callerSavedRegs[i]],
                 _reg[RiscvReg::SP], NULL, i * POINTER_SIZE, EMPTY_STR,
                 NULL);
    }
}

// save return value
auto r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
if (r0 != RiscvReg::ZERO) {
    addInstr(RiscvInstr::ASSIGN, _reg[r0], _reg[RiscvReg::A0], NULL, 0,
             EMPTY_STR, NULL);
}

// restore a0
if (r0 != RiscvReg::A0 && saved[0]) {
    addInstr(RiscvInstr::LW, _reg[RiscvReg::A0], _reg[RiscvReg::SP], NULL,
             0, EMPTY_STR, NULL);
}
```

综上，全部代码如下：

```
void RiscvDesc::emitCallTac(Tac *t) {
    // Save caller saved registers
    const int callerSavedRegs[] = {
        RiscvReg::A0, RiscvReg::RA, RiscvReg::T0, RiscvReg::T1,
        RiscvReg::T2, RiscvReg::A1, RiscvReg::A2, RiscvReg::A3,
        RiscvReg::A4, RiscvReg::A5, RiscvReg::A6, RiscvReg::A7,
        RiscvReg::T3, RiscvReg::T4, RiscvReg::T5, RiscvReg::T6,
    };
    constexpr int numCallerSavedRegs =
        sizeof(callerSavedRegs) / sizeof(callerSavedRegs[0]);
    bool saved[numCallerSavedRegs] = {0};
    constexpr int allocSavedMemSize =
        (numCallerSavedRegs * POINTER_SIZE + 0x0F) & ~0x0F; // 16 bytes
    addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
             -allocSavedMemSize, EMPTY_STR,
             "Alloc for Saving caller saved registers");

    for (int i = 0; i < numCallerSavedRegs; ++i) {
        if (t->LiveOut->contains(_reg[callerSavedRegs[i]]->var)) {
            addInstr(RiscvInstr::SW, _reg[callerSavedRegs[i]],
                     _reg[RiscvReg::SP], NULL, i * POINTER_SIZE, EMPTY_STR,
                     NULL);
            saved[i] = true;
        }
    }

    // pass extra parameters
    const int allocMemSize = ((this->_extraArgs.size() * POINTER_SIZE) + 0x0F) &
                             ~0x0F; // align to 16 bytes
    if (allocMemSize != 0) {
        addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
                 -allocMemSize, EMPTY_STR, "Alloc extra space for parameters");
        while (!this->_extraArgs.empty()) {
            Temp arg = this->_extraArgs.top();
            this->_extraArgs.pop();
            auto r = getRegForRead(arg, 0, t->LiveOut);
            addInstr(RiscvInstr::SW, _reg[r], _reg[RiscvReg::SP], NULL,
                     this->_extraArgs.size() * POINTER_SIZE, EMPTY_STR,
                     "Pass extra parameter");
        }
    }

    // call function
    addInstr(RiscvInstr::CALL, NULL, NULL, NULL, 0, t->op1.label->str_form,
             NULL);

    // restore stack pointer for param
    if (allocMemSize != 0) {
        addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL,
                 allocMemSize, EMPTY_STR, "Restore stack pointer");
    }

    // reset
    this->_lastUsedParamReg = 0;
```

```
    // restore caller saved registers except a0
    for (int i = 0; i < numCallerSavedRegs; ++i) {
        if (saved[i] && callerSavedRegs[i] != RiscvReg::A0) {
            addInstr(RiscvInstr::LW, _reg[callerSavedRegs[i]],
                     _reg[RiscvReg::SP], NULL, i * POINTER_SIZE, EMPTY_STR,
                     NULL);
        }
    }

    // save return value
    auto r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    if (r0 != RiscvReg::ZERO) {
        addInstr(RiscvInstr::ASSIGN, _reg[r0], _reg[RiscvReg::A0], NULL, 0,
                 EMPTY_STR, NULL);
    }

    // restore a0
    if (r0 != RiscvReg::A0 && saved[0]) {
        addInstr(RiscvInstr::LW, _reg[RiscvReg::A0], _reg[RiscvReg::SP], NULL,
                 0, EMPTY_STR, NULL);
    }
}
```

翻译 `PARAM` 三地址码

`PARAM` 三地址码用于取出形参。首先需要记录当前的 `PARAM` 指令是第几个参数，因此定义变量用于计数：

```
class RiscvDesc : public MachineDesc {
    /* ... */
    int _currentLoadedParam = 0; // which parameter is being loaded?
};
```

当当前参数序数小于 8 时，从寄存器中读取，否则基于寄存器 `fp` 从栈中读取：

```
void RiscvDesc::emitParamTac(Tac *t) {
    auto r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    if (this->_currentLoadedParam < 8) {
        addInstr(RiscvInstr::MOVE, _reg[r0],
                 _reg[RiscvReg::A0 + this->_currentLoadedParam], NULL, 0,
                 EMPTY_STR, NULL);
    } else {
        addInstr(RiscvInstr::LW, _reg[r0], _reg[RiscvReg::FP], NULL,
                 8 + (this->_currentLoadedParam - 8) * POINTER_SIZE, EMPTY_STR,
                 NULL);
    }
    ++this->_currentLoadedParam;
}
```

在每次翻译函数结束后，都将参数计数归零：

```
  void RiscvDesc::emitPieces(scope::GlobalScope *gscope, Piece *ps,
                             std::ostream &os) {
      /* ... */
      switch (ps->kind) {
      case Piece::FUNCTY:
          emitFuncty(ps->as.functy);
+         this->_currentLoadedParam = 0; // reset the counter of params
          break;
      /* ... */
  }
```

以上便是代码的全部更改。

思考题

1

> 1. MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

答:

```
int f(int x, int y) {
    return x + y;
}

int main() {
    int i = 114514;
    return f(++i, ++i);
}
```

2

> 2. 为何 RISC-V 标准调用约定中要引入 callee-saved 和 caller-saved 两类寄存器，而不是要求所有寄存器完全由 caller/callee 中的一方保存？为何保存返回地址的 ra 寄存器是 caller-saved 寄存器？

答:

1. 关于区分 callee-saved 和 caller-saved 寄存器：首先，内存的访问是相对来说比较耗时的，调用者和被调用者都可以通过自行选择寄存器，使得需要保存的寄存器达到最少，以达到访存时间最低。例如，`t0` ~ `t6` 寄存器作为调用者保存的临时寄存器，调用者可以用于存储临时的计算结果，其储存的值不跨越函数调用，因此可以无需保存；而 `s` 开头的寄存器作为 `callee_saved` 寄存器，可以用于一些函数内部的其生命周期跨越函数调用的局部变量，而被调用函数可以只保存自己用到的寄存器，那些被调用函数没有用到的寄存器便不需要保存，而不是所有调用者用到的寄存器都需要保存。这样便于减少寄存器保存到内存中的次数，以便于提高程序的执行效率。
2. 因为在执行 `call` 指令时，`ra` 寄存器会被设置为该条 `call` 指令的下一条指令的地址，即执行 `call` 指令时，`ra` 寄存器的值已经改变了，故被调用函数根本看不到 `ra` 本来的值，因此 `ra` 只能由调用者保存。

Step10

代码修改

修改语法树结点

首先修改 `ast::VarDecl` 语法树结点，增加记录全局变量是否初始化，以及初始值的成员变量：

```
  class VarDecl : public Statement {
      /* ... */
      Expr *init;
+     int global_init;
+     bool is_global_init;

      symb::Variable *ATTR(sym); // for semantic analysis
  };
```

然后修改相应的构造函数：

```
  VarDecl::VarDecl(std::string n, Type *t, Location *l) {
      name = n;
      type = t;
      init = NULL;
+     is_global_init = false;
  }

  VarDecl::VarDecl(std::string n, Type *t, Expr *i, Location *l) {
      name = n;
      type = t;
      init = i;
+     is_global_init = false;
  }

  VarDecl::VarDecl(std::string n, Type *t, int d, Location *l) {
      name = n;
      type = t;
      init = NULL;
+     is_global_init = true;
+     global_init = d;
  }
```

修改语法规则

在 `parser.y` 中修改语法规则，使其支持全局变量的定义：

```
-%nterm<mind::ast::VarDecl*> VarDecl
+%nterm<mind::ast::VarDecl*> VarDecl GlobalVarDecl

/* ... */

- FoDList             :
+ FoDList             :
+                     GlobalVarDecl
+                         {$$ = new ast::Program($1,POS(@1)); } |
                      FuncDefn
                          {$$ = new ast::Program($1,POS(@1)); } |
                      FoDList FuncDefn{
                           {$1->func_and_globals->append($2);
```

```
                            $$ = $1; }
+                          } |
+                     FoDList GlobalVarDecl{
+                           {$1->func_and_globals->append($2);
+                            $$ = $1; }
                           }
                      ;
+ GlobalVarDecl       : Type IDENTIFIER SEMICOLON{
+                          $$ = new ast::VarDecl($2,$1,POS(@1));
+                     }
+                     | Type IDENTIFIER ASSIGN ICONST SEMICOLON{
+                          $$ = new ast::VarDecl($2,$1,$4,POS(@1));
+                     }
                      ;
```

构建符号表

首先，为了记录有哪些全局变量，在 `ScopeStack` 中增加 `_global_vars`，用于记录所有的全局变量：

```
  class ScopeStack {
      util::Stack<Scope *> _stack;
      // a track of the global scope
      Scope *_global;
+     // global var loaded map
+     std::unordered_set<symb::Variable *> _global_vars;
    public:
      /* ... */
      // Looks up the topmost scope of a specific kind
      Scope *lookForScope(Scope::kind_t kind);
+     // Add global var
+     void AddGlobalVar(symb::Variable *name);
+     // Get global vars
+     std::unordered_set<symb::Variable *> &GetGlobalVars();
  };
```

```
void ScopeStack::AddGlobalVar(symb::Variable *v) { _global_vars.insert(v); }

std::unordered_set<symb::Variable *> &ScopeStack::GetGlobalVars() {
    return _global_vars;
}
```

在构建符号表时，设置符号的 `is_global_init` 用于记录变量是否初始化，并且将所有全局变量的符号加入到全局变量集合内：

```
  void SemPass1::visit(ast::VarDecl *vdecl) {
      /* ... */

      // TODO: Special processing for global variables
+     if (vdecl->is_global_init) {
+         v->setGlobalInit(vdecl->global_init);
+     }

      vdecl->ATTR(sym) = v;
+
```

```
+     if (v->isGlobalVar()) {
+         scopes->AddGlobalVar(v);
+     }
  }
```

静态存储期对象实体

与函数的 `FunctyObject` 类似，增加类型 `StaticObject` 用于存储具有静态存储期的对象实体，在这里用于记录全局变量：

```
typedef struct StaticObject {
    Label label;    // name of the global object
    int size;       // size of the global object
    int init_val;   // initial value of the global object
    bool tentitive; // whether it's declared with a tentitive definition
} * Static;
```

同时在 `define.hpp` 内：

```
+ struct StaticObject;
+ typedef struct StaticObject *Static;
```

然后在变量符号中加入：

```
  class Variable : public Symbol {
      /* ... */
+     // the associated static object
+     tac::Static static_obj;

    public:
      /* ... */
+
+   public:
+     // Attaches a static object to this symbol
+     void attachStatic(tac::Static);
+     // Gets the attached static object
+     tac::Static getStatic(void);
  };
```

```
Static Variable::getStatic(void) { return static_obj; }
void Variable::attachStatic(Static s) { static_obj = s; }
```

加入到 `Piece` 列表

由于全局变量是在程序顶级，因此需要加入到所有的 `Piece` 列表中。增加函数 `regGlobal`：

```
  struct Piece {
      /* ... */
      // kind of this Piece node
      enum {
          FUNCTY,
+         GLOBL,
      } kind;
```

```
      // data of this Piece node
      union {
          Functy functy;
+         Static globl;
      } as;
      /* ... */
  };
```

```
  class TransHelper {
      /* ... */
+     // allocates a new static object label
+     Label getNewVarLabel(std::string);
      /* ... */
  };
```

```
void TransHelper::regGlobal(symb::Variable *g, Static static_obj) {
    mind_assert(NULL != g);

    ptail = ptail->next = new Piece();
    ptail->kind = Piece::GLOBL;
    ptail->as.globl = static_obj;
}
```

生成变量标签

增加变量标签生成函数：

```
  class TransHelper {
      /* ... */
+     // allocates a new static object label
+     Label getNewVarLabel(std::string);
      /* ... */
  };
```

```
Label TransHelper::getNewVarLabel(std::string name) {
    Label l = new LabelObject();
    l->id = label_count++;
    l->str_form = name;
    l->target = false;

    return l;
}
```

增加三地址码

设计三地址码格式

拟增加 `LOAD_SYM` 和 `LOAD_MEM` 两个三地址码。前者终于将符号对应的地址加载到寄存器中，后者用于将某地址所存储的内容加载到寄存器中：

```
T0 <- label  # LOAD_SYM
T0 <- *T1    # LOAD_MEM
```

本 step 暂没有设计写回内存的三地址码，原因将在后面给出。

增加三地址码代码

首先增加枚举类型：

```
  struct Tac {
          POP,
          RETURN,
          LOAD_IMM4,
-         MEMO
+         MEMO,
+         LOAD_SYM,
+         LOAD_MEM,
      } kind;
```

然后增加序列化信息：

```
void Tac::dump(std::ostream &os) {
    /* ... */

    case LOAD_SYM:
        os << "    " << op0.var << " <- " << op1.label;
        break;

    case LOAD_MEM:
        os << "    " << op0.var << " <- *" << op1.var;
        break;

    default:
        mind_assert(false); // unreachable
        break;
}
```

然后增加生成该三地址码的代码：

```
  struct Tac {
      /* ... */
+     static Tac *LoadSym(Temp dest, Label sym);
+     static Tac *LoadMem(Temp dest, Temp addr);
      /* ... */
  };
```

```
Tac *Tac::LoadSym(Temp dest, Label sym) {
    REQUIRE_I4(dest);

    Tac *t = allocateNewTac(Tac::LOAD_SYM);
    t->op0.var = dest;
    t->op1.label = sym;

    return t;
}

Tac *Tac::LoadMem(Temp dest, Temp addr) {
    REQUIRE_I4(dest);
    REQUIRE_I4(addr);
```

```
    Tac *t = allocateNewTac(Tac::LOAD_MEM);
    t->op0.var = dest;
    t->op1.var = addr;

    return t;
}
```

维护三地址码数据流信息

由于 `LoadSym` 只需要改变一个寄存器的值，而 `LoadMem` 需要读取一个寄存器的值且需要改变一个寄存器的值，因此数据流信息如下：

```
void BasicBlock::computeDefAndLiveUse(void) {
        /* ... */
        case Tac::LOAD_MEM:
            updateLU(t->op1.var);
            updateDEF(t->op0.var);
            break;
        /* ... */
        case Tac::LOAD_SYM:
            updateDEF(t->op0.var);
            break;
        /* ... */
}
```

```
void BasicBlock::analyzeLiveness(void) {
        /* ... */
        case Tac::LOAD_MEM:
            if (NULL != t_next->op0.var)
                t->LiveOut->remove(t_next->op0.var);
            t->LiveOut->add(t_next->op1.var);
        /* ... */
        case Tac::LOAD_SYM:
            if (NULL != t_next->op0.var)
                t->LiveOut->remove(t_next->op0.var);
            break;
        /* ... */
}
```

生成三地址码

为新增的三地址码增加生成接口

代码如下：

```
  class TransHelper {
      /* ... */
+     // Memory Access
+     Temp genLoadSym(Label);
+     Temp genLoadMem(Temp);

      /* ... */
  };
```

```
Temp TransHelper::genLoadSym(Label sym) {
    Temp c = getNewTempI4();
    chainUp(Tac::LoadSym(c, sym));
    return c;
}

Temp TransHelper::genLoadMem(Temp addr) {
    Temp c = getNewTempI4();
    chainUp(Tac::LoadMem(c, addr));
    return c;
}
```

为变量定义生成三地址码

对于全局变量，在变量定义处，不需要生成三地址码，但是需要申请其为全局变量，将其加入 `Piece` 列表中，在顶级处理：

```
void Translation::visit(ast::VarDecl *decl) {
    /* ... */
    if (!decl->ATTR(sym)->isGlobalVar()) {
        if (decl->init) {
            decl->init->accept(this);
            tr->genAssign(decl->ATTR(sym)->getTemp(), decl->init->ATTR(val));
        }
    } else {
        Static static_obj = new StaticObject();
        static_obj->label = tr->getNewVarLabel(decl->ATTR(sym)->getName());
        static_obj->size = 4;
        if (decl->is_global_init) {
            static_obj->tentitive = false;
            static_obj->init_val = decl->ATTR(sym)->getGlobalInit();
        } else {
            static_obj->tentitive = true;
        }
        decl->ATTR(sym)->attachStatic(static_obj);
        tr->regGlobal(decl->ATTR(sym), static_obj);
    }
}
```

在函数中生成加载全局变量值的三地址码

由于在**构建符号表**阶段，已经记录了存在的所有全局变量，因此在所有函数进入函数后，将所有可能用到的全局变量从内存加载到寄存器中。

注意到，由于本次 step 没有未要求所有函数共享全局变量修改值的要求，即测例中不包含修改对其他函数产生影响的全局变量值的操作，因此为了简化代码，不设计将全局变量写回内存的操作。

但如果要设计，**需要在每一个函数调用前，以及每一个返回语句前，都增加对可能修改过的全局变量写回内存的操作；并在每一个函数调用之后，增加将所有可能用到的全局变量的值加载到寄存器**，以处理本函数与调用与被调用函数之间对全局变量的修改的可见性。但本 step 的要求中暂不需要此操作。

因此，在函数体语句执行前增加加载全局变量的三地址码，代码如下：

```
  void Translation::visit(ast::FuncDefn *f) {
      /* ... */

+     // Load global variables
```

```
+     for (auto globals : scopes->GetGlobalVars()) {
+         Temp addr = tr->genLoadSym(globals->getStatic()->label);
+         Temp val = tr->genLoadMem(addr);
+         tr->genAssign(globals->getTemp(), val);
+     }
+
      // translates statement by statement
      for (auto it = f->stmts->begin(); it != f->stmts->end(); ++it)
          (*it)->accept(this);
      /* .... */
  }
```

生成汇编

最后生成汇编代码。

处理顶级全局变量定义

首先处理位于顶级的全局变量的定义。对于未初始化的全局变量，将其放置在 `.bss` 段中；对于初始化的全局变量，放置在 `.data` 段中并赋初值。代码如下：

```
  class RiscvDesc : public MachineDesc {
      /* ... */
+     // outputs a global variable
+     void emitGlobl(tac::Static);
      /* ... */
  };
```

```
void RiscvDesc::emitPieces(scope::GlobalScope *gscope, Piece *ps,
                           std::ostream &os) {
    /* ... */
    case Piece::GLOBL:
        emitGlobl(ps->as.globl);
        break;
    /* ... */
}
```

```
void RiscvDesc::emitGlobl(Static globl) {
    if (globl->tentitive) {
        emit(EMPTY_STR, ".bss", NULL);
        emit(EMPTY_STR,
             (std::string(".globl ") + globl->label->str_form).c_str(), NULL);
        emit(EMPTY_STR, ".align 4", NULL);
        emit(globl->label->str_form.c_str(), NULL, NULL);
        emit(EMPTY_STR, ".space 4", NULL);
    } else {
        emit(EMPTY_STR, ".data", NULL);
        emit(EMPTY_STR,
             (std::string(".globl ") + globl->label->str_form).c_str(), NULL);
        emit(EMPTY_STR, ".align 4", NULL);
        emit(globl->label->str_form.c_str(), NULL, NULL);
        emit(EMPTY_STR,
             (std::string(".word ") + std::to_string(globl->init_val)).c_str(),
             NULL);
    }
}
```

翻译三地址码

然后将新增的三地址码翻译成汇编指令。

首先增加 `la` 汇编指令，将 `LOAD_SYM` 翻译为 `la` 指令：

```
  struct RiscvInstr : public Instr {
          /* ... */
          CALL,
+         LA,
          // You could add other instructions/pseudo instructions here
      } op_code; // operation code
```

```
  void RiscvDesc::emitInstr(RiscvInstr *i) {
      /* ... */

+     case RiscvInstr::LA:
+         oss << "la" << i->r0->name << ", " << i->l;
+         break;
+
      default:
          mind_assert(false); // other instructions not supported
  }
```

然后增加将 `LOAD_MEM` 和 `LOAD_SYM` 翻译成汇编指令的函数：

```
  class RiscvDesc : public MachineDesc {
+     // translate a LoadSym TAC into assembly instructions
+     void emitLoadSymTac(tac::Tac *);
+     // translate a LoadMem TAC into assembly instructions
+     void emitLoadMemTac(tac::Tac *);
  };
```

```
void RiscvDesc::emitLoadSymTac(Tac *t) {
    auto r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    addInstr(RiscvInstr::LA, _reg[r0], NULL, NULL, 0, t->op1.label->str_form,
             NULL);
}

void RiscvDesc::emitLoadMemTac(Tac *t) {
    int r1 = getRegForRead(t->op1.var, 0, t->LiveOut);
    int r0 = getRegForWrite(t->op0.var, r1, 0, t->LiveOut);
    addInstr(RiscvInstr::LW, _reg[r0], _reg[r1], NULL, 0, EMPTY_STR, NULL);
}
```

```
  void RiscvDesc::emitTac(Tac *t) {
      /* ... */

+     case Tac::LOAD_SYM:
+         emitLoadSymTac(t);
+         break;
+
+     case Tac::LOAD_MEM:
+         emitLoadMemTac(t);
```

```
+            break;
+
        default:
            mind_assert(false); // should not appear inside a basic block
    }
```

以上便是代码的全部更改。

思考题

1

> 1. 写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

答：

1. 可能被替换成 `auipc` 指令与 `addi` 指令的组合
2. 可能直接使用寄存器 `gp` 加上偏移量，即使用 `addi` 指令实现