# MiniDecaf Stage 4 Report

梁业升 2019010547（计 03）

2022 年 12 月 25 日

## 1 实验内容

### 1.1 Step 9：函数

#### 1.1.1 词法语法分析

主要增加对于形参和实参列表的解析：

```
ParamList :  /* EMPTY */ { $$ = new ast::VarList(); }
          | CommaSepParamList { $$ = $1; }
          ;

CommaSepParamList :
Type IDENTIFIER
    { $$ = new ast::VarList(); $$->append(new ast::VarDecl($2,
    $1, POS(@1))); }
| CommaSepParamList COMMA Type IDENTIFIER
    { $1->append(new ast::VarDecl($4, $3, POS(@3))); $$ = $1; }
;

ExprList    : /* EMPTY */ { $$ = new ast::ExprList(); }
            | CommaSepExprList { $$ = $1; }
            ;
CommaSepExprList :
Expr
    { $$ = new ast::ExprList(); $$->append($1); }
| CommaSepExprList COMMA Expr
    { $$ = $1; $$->append($3); }
;
```

另外，增加 FuncCallExpr 节点：

```
class FuncCallExpr : public Expr {
```

```
  public:
    FuncCallExpr(std::string name, ExprList *args, Location *l
  );
    virtual void accept(Visitor *);
    virtual void dumpTo(std::ostream &);

  public:
    std::string name;
    ExprList *args;

    symb::Function *ATTR(sym);
};
```

在语法分析中：

```
Expr:
    ...
    | IDENTIFIER LPAREN ExprList RPAREN
        { $$ = new ast::FuncCallExpr($1, $3, POS(@1)); }
    ;
```

### 1.1.2 符号表构建

在第一个 Pass，对于 FuncDefn 节点，需要修改原框架中对于符号表已存在的处理。具体来说，当函数符号为前向声明时，不认为是冲突定义。在这里我们用 Symbol 的 mark 成员来保存此信息。

```
void SemPass1::visit(ast::FuncDefn *fdef) {
    ...
    Symbol *sym = scopes->lookup(fdef->name, fdef->getLocation
    (), false);
    if (sym == NULL) {
        f->mark = fdef->forward_decl;
        scopes->declare(f);
    } else {
        if (fdef->forward_decl)
            return;
        if (sym->mark == 1) // previously forward declaration
            sym->mark = 0;
        else // redefinition
            issue(fdef->getLocation(), new DeclConflictError(
    fdef->name, sym));
    }
```

```
    ...
}
```

### 1.1.3 类型检查

针对新增的表达式 FuncCallExpr 增加类型检查，主要处理符号未定义和参数个数不一致的情况（目前仅支持 int 类型，因此暂时不需要检查参数类型是否正确：

```
void SemPass2::visit(ast::FuncCallExpr *e) {
    // Find the symbol in the symbol table.
    Symbol *s = scopes->lookup(e->name, e->getLocation());
    Function *func;
    size_t numArgs = 0;

    if (s == nullptr) {
        issue(e->getLocation(), new SymbolNotFoundError(e->name
    ));
        goto issue_error_type;
    } else if (!s->isFunction()) {
        issue(e->getLocation(), new NotMethodError(s));
        goto issue_error_type;
    }

    func = dynamic_cast<Function *>(s);
    mind_assert(func != nullptr);

    for (auto arg = e->args->begin(); arg != e->args->end(); ++
    arg) {
        (*arg)->accept(this);
        ++numArgs;
    }

    // Check the number of arguments.
    if (func->getType()->numOfParameters() != numArgs) {
        issue(e->getLocation(), new BadArgCountError(func));
        goto issue_error_type;
    }

    e->ATTR(type) = func->getType()->getResultType();
    e->ATTR(sym) = func;
    return;
```

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

### 1.1.4 翻译为中间代码

增加三个 TAC 类型：

- `CALL label`：调用 `label` 函数
- `PARAM src, x`：从 `src` 传入第 `x` 个参数
- `GET_PARAM dest, x`：取出第 `x` 个参数并赋值给 `dest`

对应的翻译代码如下：

翻译 `FuncCallExpr`，注意应先生成函数参数对应的 TAC，再生成 `PARAM`：

```
void Translation::visit(ast::FuncCallExpr *e) {
    // We should visit all the arguments first, and then
    generate the PARAM tac.
    for (auto iter = e->args->begin(); iter != e->args->end();
    iter++) {
        (*iter)->accept(this);
    }
    // Push the args in the reversed order.
    int total = e->args->length() - 1;
    auto iter = e->args->end();
    if (iter != e->args->begin()) {
        while (true) {
            iter--;
            tr->genParam((*iter)->ATTR(val), total);
            total--;
            if (iter == e->args->begin()) {
                break;
            }
        }
    }

    Temp res = tr->genCall(e->ATTR(sym)->getEntryLabel());
    e->ATTR(val) = res;
}
```

对于 FuncDefn，当函数为前向声明时，直接返回：

```
if (f->forward_decl)
    return;
```

另外，在翻译函数体前，先生成 GET_PARAM 取出参数：

```
for (auto it = f->formals->begin(); it != f->formals->end(); ++
    it) {
    auto v = (*it)->ATTR(sym);
    tr->genGetParam(v->getTemp(), v->getOrder());
}
```

其余保持不变。

### 1.1.5 翻译为汇编代码

将 CALL、PARAM 和 GET_PARAM 翻译为对应的汇编代码。

翻译 PARAM 时，分寄存器传参和栈传参两种情形。在这里我们将传参用寄存器的 general 临时设为 false，以避免被覆盖（一个 trick，现有框架下似乎这是最简单的方法）。

```
void RiscvDesc::emitParamTac(Tac *t) {
    if (t->op1.ival < 8) {
        passParamReg(t, t->op1.ival);
        // A trick to protect the arg regs from being
    overwritten, which seems
        // not being done by the original framework.
        _reg[RiscvReg::A0 + t->op1.ival]->general = false;
    } else {
        int regIndex = getRegForRead(t->op0.var, 0, t->LiveOut)
    ;
        addInstr(RiscvInstr::SW, _reg[regIndex], _reg[RiscvReg
    ::SP], NULL,
                 (t->op1.ival - 8) * 4, EMPTY_STR, "pass
    param into stack");
    }
}
```

翻译 CALL 时，先将翻译 PARAM 时对传参用寄存器的修改恢复，然后保存寄存器、跳转。

```
void RiscvDesc::emitCallTac(Tac *t) {
    // Unprotect the arg regs.
```

```
    for (int i = 0; i < 8; i++)
        _reg[RiscvReg::A0 + i]->general = true;
    spillDirtyRegs(t->LiveOut);
    addInstr(RiscvInstr::JAL, NULL, NULL, NULL, 0, t->op1.label
    ->str_form, EMPTY_STR);
    int res = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    addInstr(RiscvInstr::MOVE, _reg[res], _reg[RiscvReg::A0],
        NULL, 0, EMPTY_STR, "move return value to result
    register");
}
```

在函数体内，翻译 GET_PARAM，将参数赋值到对应的符号上：

```
void RiscvDesc::emitGetParamTac(Tac *t) {
    if (t->op1.ival < 8) {
        getParamReg(t, t->op1.ival);
    } else {
        int regIndex = getRegForWrite(t->op0.var, 0, 0, t->
    LiveOut);
        int frameSize = _frame->getStackFrameSize();
        addInstr(RiscvInstr::LW, _reg[regIndex], _reg[RiscvReg
    ::SP], NULL, (t->op1.ival - 8) * 4 - frameSize, EMPTY_STR,
    EMPTY_STR);
    }
}
```

## 1.2 Step 10：全局变量

### 1.2.1 词法语法分析

在 FoDList 中增加 DeclStmt 即可：

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

### 1.2.2 符号表构建

原有框架已完成此工作。

### 1.2.3 类型检查

针对 VarDecl，如果有初始值时，检查其是否为常数字面值：

```
void SemPass2::visit(ast::VarDecl *decl) {
    if (decl->init) {
        decl->init->accept(this);
        if (decl->ATTR(sym)->isGlobalVar() &&
            decl->init->getKind() != ast::ASTNode::INT_CONST) {
            issue(decl->getLocation(), new NotConstInitError())
    ;
        }
    }
}
```

### 1.2.4 翻译为中间代码

增加四个 TAC 类型：

- DECL_GLOB_VAR label, size, defaults：声明大小为 size，初始值（可选）为 defaults、名称为 label 全局变量
- LOAD_SYM dest, label：将 label 的地址存入 dest
- LOAD dest, src, offset：将 src + offset 地址处的 4 字存加载到 dest 中
- STORE dest, offset, src：将 src + offset 地址处的 4 字节存到 dest 所指的内存地址中

对应的翻译代码如下：

翻译 VarDecl，对全局变量的情形进行处理，插入 DECL_GLOB_VAR 节点：

```
void Translation::visit(ast::VarDecl *decl) {
    ...
    if (decl->ATTR(sym)->isGlobalVar()) {
        // Create label for the global variable
        decl->ATTR(sym)->attachLabel(tr->getNewGlobVarLabel(
    decl->ATTR(sym)));

        int *defaultValues = NULL;
        if (decl->init != NULL) {
            ast::IntConst *intConst = dynamic_cast<ast::
    IntConst *>(decl->init);
            defaultValues = new int[1];
```

```
            defaultValues[0] = intConst->value;
        }

        tr->genDeclGlobVar(decl->ATTR(sym)->getLabel(), 1,
    defaultValues);
    }
    ...
}
```

对 AssignExpr，存入符号对应的内存中（LOAD_SYM 和 STORE）：

```
void Translation::visit(ast::AssignExpr *s) {
    ...
    if (ref->ATTR(sym)->isGlobalVar()) {
        Temp symAddr = tr->genLoadSym(ref->ATTR(sym)->getLabel
    ());
        tr->genStore(symAddr, 0, s->e->ATTR(val));
    }
    ...
}
```

对 LvalueExpr，从符号对应的内存中读出（LOAD_SYM 和 LOAD）：

```
void Translation::visit(ast::LvalueExpr *e) {
    ref->accept(this);
    if (ref->ATTR(sym)->isGlobalVar()) {
        Temp addr = tr->genLoadSym(ref->ATTR(sym)->getLabel());
        e->ATTR(val) = tr->genLoad(addr, 0);
    }
    ...
}
```

### 1.2.5 翻译为汇编代码

将 DECL_GLOB_VAR、LOAD_SYM、LOAD 和 STORE 翻译为对应的汇编代码。

DECL_GLOB_VAR 比较特殊，其不属于任何基本块，需要增加 Piece。我们在 Piece 中增加新的类型 VAR_DECL，与其关联的值为 DECL_GLOB_VAR 的 TAC varDecl。

```
void TransHelper::genDeclGlobVar(Label label, int size, int *
    defaultValue) {
    ptail = ptail->next = new Piece();
    ptail->kind = Piece::VAR_DECL;
```

```
    ptail->as.varDecl = Tac::DeclGlobVar(label, size,
    defaultValue);
}
```

其余 TAC 的翻译比较平凡：

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

Temp TransHelper::genLoad(Temp src, int offset) {
    Temp dest = getNewTempI4();
    chainUp(Tac::Load(dest, src, offset));
    return dest;
}

void TransHelper::genStore(Temp dest, int offset, Temp src) {
    chainUp(Tac::Store(dest, offset, src));
}
```

## 2 思考题

1. **Step 9**：

   (a) 结果可能为 4 或 3：

   ```
   int sum(int a, int b) {
       return a + b;
   }
   int b = 1;
   int c = 1;
   int a = sum(b = c, c = b + 1);
   ```

   (b) 完全由一方保存，保存一些不会由子过程修改的寄存器会带来额外的保存到栈上的开销；子过程返回时需要用到 `ra` 的值，不可能在返回之后再恢复父过程的返回地址。

2. **Step 10**：`li v0, <addr>` 或 `auipc v0, <offset>`。