# Stage-3 实验报告

范逍宇 **2019013273**

## 一、工作概述。

### Step 7

**1.**前端。

- 加入块语句对应的语法规则，但实际上框架中已经实现如下：

```
CompStmt    : LBRACE StmtList RBRACE
                {$$ = new ast::CompStmt($2,POS(@1));}
            ;
```

经检验，框架的实现是准确无误的。

**2.**中端。

我们需要在符号表构建和类型检查阶段构建作用域栈，实现局部作用域。事实上，框架中已经实现如下：

- 符号表构建：

```
void SemPass1::visit(ast::CompStmt *c) {
    // opens function scope
    Scope *scope = new LocalScope();
    c->ATTR(scope) = scope;
    scopes->open(scope);

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

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

- 类型检查：

```
void SemPass2::visit(ast::CompStmt *c) {
    scopes->open(c->ATTR(scope));
    for (auto it = c->stmts->begin(); it != c->stmts->end(); ++it)
        (*it)->accept(this);
    scopes->close();
}
```

在中间代码生成阶段，我们只需要递归生成块作用域的每一条指令即可：

```
void Translation::visit(ast::CompStmt *c) {
    // translates statement by statement
    for (auto it = c->stmts->begin(); it != c->stmts->end(); ++it)
        (*it)->accept(this);
}
```

**3.**后端。

没有新增指令，所以后端没有变化。

## Step 8

**1.**前端。

- 在 `scanner.l` 文件中添加 for, do, continue, break 对应的词法规则：

```
"while"       { return yy::parser::make_WHILE  (loc);       }
"do"          { return yy::parser::make_DO  (loc);       }
"for"         { return yy::parser::make_FOR (loc);       }
"break"       { return yy::parser::make_BREAK (loc);       }
"continue"    { return yy::parser::make_CONTINUE (loc);       }
```

- 经检查，在 `parser.y` 文件中添加 for, do while, continue, break 对应的语法规则：

```
Stmt        : ReturnStmt {$$ = $1;}|
              ExprStmt   {$$ = $1;}|
              IfStmt     {$$ = $1;}|
              WhileStmt  {$$ = $1;}|
              ForStmt    {$$ = $1;}|
              DoWhileStmt{$$ = $1;}|
              CompStmt   {$$ = $1;}|
              BREAK SEMICOLON
                {$$ = new ast::BreakStmt(POS(@1));} |
              CONTINUE SEMICOLON
                {$$ = new ast::ContStmt(POS(@1));} |
              SEMICOLON
                {$$ = new ast::EmptyStmt(POS(@1));}
            ;
DoWhileStmt : DO Stmt WHILE LPAREN Expr RPAREN SEMICOLON
                { $$ = new ast::DoWhileStmt($5, $2, POS(@1)); }
            ;
ForStmt     : FOR LPAREN Expr SEMICOLON Expr SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt($3, $5, $7, $9, POS(@1)); }|
              FOR LPAREN Expr SEMICOLON Expr SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt($3, $5, nullptr, $8, POS(@1)); }|
              FOR LPAREN Expr SEMICOLON SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt($3, nullptr, $6, $8, POS(@1)); }|
              FOR LPAREN Expr SEMICOLON SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt($3, nullptr, nullptr, $7, POS(@1)); }|
              FOR LPAREN SEMICOLON Expr SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt(nullptr, $4, $6, $8, POS(@1)); }|
              FOR LPAREN SEMICOLON Expr SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt(nullptr, $4, nullptr, $7, POS(@1)); }|
              FOR LPAREN SEMICOLON SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt(nullptr, nullptr, $5, $7, POS(@1)); }|
              FOR LPAREN SEMICOLON SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt(nullptr, nullptr, nullptr, $6, POS(@1)); }|

              FOR LPAREN VarDecl Expr SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt($3, $4, $6, $8, POS(@1), 0); }|
              FOR LPAREN VarDecl Expr SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt($3, $4, nullptr, $7, POS(@1), 0); }|
              FOR LPAREN VarDecl SEMICOLON Expr RPAREN Stmt
                { $$ = new ast::ForStmt($3, nullptr, $5, $7, POS(@1), 0); }|
              FOR LPAREN VarDecl SEMICOLON RPAREN Stmt
                { $$ = new ast::ForStmt($3, nullptr, nullptr, $6, POS(@1), 0); }
            ;
```

**2.**中端。

- 在符号表构建时，对于 while, do while, 递归地构建其循环体中的语句即可，对于 for 语句，还需要维护作用域栈：

```
void SemPass1::visit(ast::WhileStmt *s) {
    s->condition->accept(this);
    s->loop_body->accept(this);
}

void SemPass1::visit(ast::DoWhileStmt *s) {
    s->condition->accept(this);
    s->loop_body->accept(this);
}

void SemPass1::visit(ast::ForStmt *s) {
    // opens function scope
    Scope *scope = new LocalScope();
    s->ATTR(scope) = scope;
    scopes->open(scope);

    if(s->exprInit!=nullptr) {
        s->exprInit->accept(this);
        // std::cout<<"exprInit!"<<std::endl;
    }
    else if(s->varDeclInit!=nullptr){
        s->varDeclInit->accept(this);
        // std::cout<<"varDeclInit!"<<std::endl;
    }
    if(s->condition!=nullptr) s->condition->accept(this);
    if(s->update!=nullptr) s->update->accept(this);
    s->loop_body->accept(this);

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

- 在类型检查时，对于 while, do while, 递归地构建其循环体中的语句即可，对于 for 语句，还需要维护作用域栈：

```
void SemPass2::visit(ast::WhileStmt *s) {
    s->condition->accept(this);
    if (!s->condition->ATTR(type)->equal(BaseType::Int)) {
        issue(s->condition->getLocation(), new BadTestExprError());
    }

    s->loop_body->accept(this);
}

void SemPass2::visit(ast::DoWhileStmt *s) {
    s->condition->accept(this);
    if (!s->condition->ATTR(type)->equal(BaseType::Int)) {
        issue(s->condition->getLocation(), new BadTestExprError());
    }

    s->loop_body->accept(this);
}

void SemPass2::visit(ast::ForStmt *s) {
    // opens function scope
```

```
    scopes->open(s->ast_attr_scope_);

    if(s->exprInit!=nullptr) {
        s->exprInit->accept(this);
        if (!s->exprInit->ATTR(type)->equal(BaseType::Int)) {
            issue(s->exprInit->getLocation(), new BadTestExprError());
        }
    }
    if(s->condition!=nullptr) {
        s->condition->accept(this);
        if (!s->condition->ATTR(type)->equal(BaseType::Int)) {
            issue(s->condition->getLocation(), new BadTestExprError());
        }
    }
    if(s->update!=nullptr) {
        s->update->accept(this);
        if (!s->update->ATTR(type)->equal(BaseType::Int)) {
            issue(s->update->getLocation(), new BadTestExprError());
        }
    }
    s->loop_body->accept(this);

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

- 在生成三地址码时，首先分析 while, do while, for 语句对应的汇编指令的结构，根据这种结构顺序生成跳转标签和递归生成其他指令。而 break, continue 语句只需要生成跳转到对应标签的三地址码即可，注意这里用全局变量的形式记下当前的 break, continue 需要跳到哪去，这样就可以实现嵌套 break, continue 。

```
void Translation::visit(ast::WhileStmt *s) {
    Label L1 = tr->getNewLabel();
    Label L2 = tr->getNewLabel();
    Label L3 = tr->getNewLabel(); // continue label

    Label old_break = current_break_label;
    current_break_label = L2;
    Label old_continue = current_continue_label;
    current_continue_label = L3;

    tr->genMarkLabel(L1);
    s->condition->accept(this);
    tr->genJumpOnZero(L2, s->condition->ATTR(val));

    s->loop_body->accept(this);
    tr->genMarkLabel(L3);
    tr->genJump(L1);

    tr->genMarkLabel(L2);

    current_break_label = old_break;
    current_continue_label = old_continue;
}

void Translation::visit(ast::DoWhileStmt *s) {
    Label L1 = tr->getNewLabel();
    Label L2 = tr->getNewLabel(); // break label
    Label L3 = tr->getNewLabel(); // continue label
```

```
    Label old_break = current_break_label;
    current_break_label = L2;
    Label old_continue = current_continue_label;
    current_continue_label = L3;

    tr->genMarkLabel(L1);
    s->loop_body->accept(this);
    tr->genMarkLabel(L3);
    s->condition->accept(this);
    tr->genJumpOnZero(L2, s->condition->ATTR(val));
    tr->genJump(L1);
    tr->genMarkLabel(L2);

    current_break_label = old_break;
    current_continue_label = old_continue;
}

void Translation::visit(ast::ForStmt *s) {

    Label L1 = tr->getNewLabel();
    Label L2 = tr->getNewLabel();
    Label L3 = tr->getNewLabel(); // continue label

    Label old_break = current_break_label;
    current_break_label = L2;
    Label old_continue = current_continue_label;
    current_continue_label = L3;

    if(s->exprInit!=nullptr) {
        s->exprInit->accept(this);

    }
    else if(s->varDeclInit!=nullptr) {
        s->varDeclInit->accept(this);
    }
    tr->genMarkLabel(L1);

    if(s->condition!=nullptr) {
        s->condition->accept(this);
        tr->genJumpOnZero(L2, s->condition->ATTR(val));
    }
    s->loop_body->accept(this);
    tr->genMarkLabel(L3);
    if(s->update!=nullptr) {
        s->update->accept(this);
    }
    tr->genJump(L1);
    tr->genMarkLabel(L2);

    current_break_label = old_break;
    current_continue_label = old_continue;
}

void Translation::visit(ast::BreakStmt *s) { tr->genJump(current_break_label); }

void Translation::visit(ast::ContStmt *s) { tr->genJump(current_continue_label); }
```

**3.**后端。

没有新增指令，所以后端没有变化。

## 二、思考题。

### Step 7

---

请画出下面 MiniDecaf 代码的控制流图。

```
int main(){
 int a = 2;
 if (a < 3) {
     {
         int a = 3;
         return a;
     }
     return a;
 }
}
```

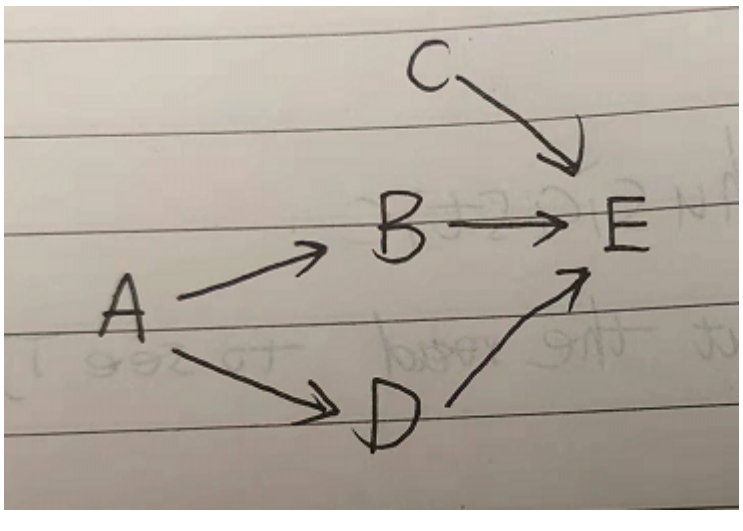
**My answer：**

这段代码对应的 RISCV 汇编代码如下：（使用老师提供的 MiniDecaf Playground 生成）

```
# 以下为基本块 A
    .text
    .globl main
main:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw fp, 8(sp)
    addi fp, sp, 16
    li t0, 2
    sw t0, -16(fp)
    lw t0, -16(fp)
    addi sp, sp, -4
    sw t0, 0(sp)
    li t0, 3
    lw t1, 0(sp)
    addi sp, sp, 4
    slt t0, t1, t0
    beqz t0, .L1
# 以下为基本块 B
    li t0, 3
    sw t0, -12(fp)
    lw t0, -12(fp)
    mv a0, t0
    j main_exit
# 以下为基本块 C
    lw t0, -16(fp)
    mv a0, t0
    j main_exit
# 以下为基本块 D
.L1:
    li t0, 0
    mv a0, t0
    j main_exit
# 以下为基本块 E
main_exit:
    lw ra, 12(sp)
```

```
    lw fp, 8(sp)
    addi sp, sp, 16
    ret
```

控制流图为：




## Step 8

将循环语句翻译成 IR 有许多可行的翻译方法，例如 while 循环可以有以下两种翻译方式：

第一种（即实验指导中的翻译方式）：

1. `label BEGINLOOP_LABEL`：开始下一轮迭代
2. `cond 的 IR`
3. `beqz BREAK_LABEL`：条件不满足就终止循环
4. `body 的 IR`
5. `label CONTINUE_LABEL`：continue 跳到这
6. `br BEGINLOOP_LABEL`：本轮迭代完成
7. `label BREAK_LABEL`：条件不满足，或者 break 语句都会跳到这儿

第二种：

1. `cond 的 IR`
2. `beqz BREAK_LABEL`：条件不满足就终止循环
3. `label BEGINLOOP_LABEL`：开始下一轮迭代
4. `body 的 IR`
5. `label CONTINUE_LABEL`：continue 跳到这
6. `cond 的 IR`
7. `bnez BEGINLOOP_LABEL`：本轮迭代完成，条件满足时进行下一次迭代
8. `label BREAK_LABEL`：条件不满足，或者 break 语句都会跳到这儿

从执行的指令的条数这个角度（`label` 指令不计算在内，假设循环体至少执行了一次），请评价这两种翻译方式哪一种更好？

**My answer：**

第一种更好，假设循环体执行一次即跳出，按第一种方法，一共执行了 6 条指令，而按第二种方法，则一共执行了 8 条指令。