MiniDecaf Stage 3: 作用域和循环

> My-laniaKeA

实验内容

Step 7: 作用域和块语句

语义分析

符号表构建

在 `src/translation/build_sym.cpp` 的 `SemPass1::visit(ast::VarDecl *vdecl)` 函数中，对于变量声明 `vdecl`，检查当前作用域有无同名符号：有则报错；无则在当前作用域声明符号。这一步在 step 5 已经完成，此处仅列出有关代码：

```
void SemPass1::visit(ast::VarDecl *vdecl) {
    // ...
    // 2. Check for conflict in `scopes`, which is a global variable refering to
    // a scope stack
    Symbol *sym = scopes->lookup(vdecl->name, vdecl->getLocation(), false);
    if (NULL != sym)
        issue(vdecl->getLocation(), new DeclConflictError(vdecl->name, sym));
   // ...
}
```

Step 8: 循环语句

词法分析和语法分析

在 `src/frontend/scanner.l` 中添加 `do`，`for` 和 `continue` 的词法规则：

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

在 `src/frontend/parser.y` 中添加 `ForStmt`，`DoWhileStmt` 和 `ContStmt` 的语法规则，修改 `Stmt` 的语法规则：

```
Stmt        : ReturnStmt {$$ = $1;}|
              ExprStmt   {$$ = $1;}|
              IfStmt     {$$ = $1;}|
              ForStmt    {$$ = $1;}|
              DoWhileStmt {$$ = $1;}|
              WhileStmt  {$$ = $1;}|
              CompStmt   {$$ = $1;}|
              BREAK SEMICOLON
```

```
                {$$ = new ast::BreakStmt(POS(@1));} |
              CONTINUE SEMICOLON
                { $$ = new ast::ContStmt(POS(@1));} |
              SEMICOLON
                {$$ = new ast::EmptyStmt(POS(@1));}
            ;
ForStmt     : FOR LPAREN ForExpr SEMICOLON ForExpr SEMICOLON ForExpr RPAREN Stmt
                { $$ = new ast::ForStmt($3, $5, $7, $9, POS(@1)); }
            | FOR LPAREN DeclrStmt ForExpr SEMICOLON ForExpr RPAREN Stmt
                { $$ = new ast::ForStmt($3, $4, $6, $8, POS(@1)); }
            ;
ForExpr     : /* empty */
                { $$ = NULL; }
            | Expr
                { $$ = $1; }
            ;
DoWhileStmt : DO Stmt WHILE LPAREN Expr RPAREN SEMICOLON
                { $$ = new ast::DoWhileStmt($2, $5, POS(@1)); }
            ;
```

增加 `ForStmt`，`DoWhildStmt` 和 `ContStmt` 对应的语法树节点。注意到 for 循环的初始化语句可能是 `Expression` 或 `Statement`，于是将成员变量设为 `ASTNode *` 类型。

在 `src/ast/ast.hpp` 中声明：

```
/* Node representing a for statement.
 *
 * SERIALIZED FORM:
 *   (for INIT CONDITION UPDATE LOOP_BODY)
 */
class ForStmt : public Statement {
  public:
    ForStmt(Expr *init, Expr *cond, Expr *update, Statement *loop_body,
 Location *l);
    ForStmt(Statement *init, Expr *cond, Expr *update, Statement *loop_body,
 Location *l);
    virtual void accept(Visitor *);
    virtual void dumpTo(std::ostream &);

  public:
    ASTNode *init;
    Expr *condition;
    Expr *update;
    Statement *loop_body;
    scope::Scope *ATTR(scope);
};

/* Node representing a do-while statement.
 *
 * SERIALIZED FORM:
 *   (do LOOP_BODY while CONDITION)
 */
class DoWhileStmt : public Statement {
  public:
    DoWhileStmt(Statement *loop_body, Expr *cond, Location *l);
```

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

  public:
    Expr *condition;
    Statement *loop_body;
};
```

在 `src/ast/ast_for_stmt.cpp` 中定义：

```
/* Creates a new ForStmt node.
 *
 * PARAMETERS:
 *   _init   - the initialization expression
 *   _cond   - the test expression
 *   _upd    - the update expression
 *   _body   - the loop body
 *   _l      - position in the source text
 */
ForStmt::ForStmt(Expr *_init, Expr *_cond, Expr *_upd, Statement *_body,
 Location *_l) {
    setBasicInfo(FOR_STMT, _l);
    init = (ASTNode*) _init;
    condition = _cond;
    update = _upd;
    loop_body = _body;
}

/* Creates a new ForStmt node.
 *
 * PARAMETERS:
 *   _init   - the initialization statement
 *   _cond   - the test expression
 *   _upd    - the update expression
 *   _body   - the loop body
 *   _l      - position in the source text
 */
ForStmt::ForStmt(Statement *_init, Expr *_cond, Expr *_upd, Statement *_body,
 Location *_l) {
    setBasicInfo(FOR_STMT, _l);
    init = (ASTNode*) _init;
    condition = _cond;
    update = _upd;
    loop_body = _body;
}

void ForStmt::accept(Visitor *v) { v->visit(this); }
void ForStmt::dumpTo(std::ostream &os) {
    // ...
}
```

在 `src/ast/ast_while_stmt.cpp` 中定义：

```
/* Creates a new DoWhileStmt node.
```

```
 *
 * PARAMETERS:
 *   body    - the loop body
 *   cond    - the test expression
 *   l       - position in the source text
 */
DoWhileStmt::DoWhileStmt(Statement *body, Expr *cond, Location *l) {
    setBasicInfo(DO_WHILE_STMT, l);
    condition = cond;
    loop_body = body;
}

void DoWhileStmt::accept(Visitor *v) { v->visit(this); }
void DoWhileStmt::dumpTo(std::ostream &os) {
    // ...
}
```

```
/* Creates a new ContStmt node.
 *
 * PARAMETERS:
 *   l       - position in the source text
 */
ContStmt::ContStmt(Location *l) { setBasicInfo(CONT_STMT, l); }

void ContStmt::accept(Visitor *v) { v->visit(this); }
void ContStmt::dumpTo(std::ostream &os) {
   // ...
}
```

语义分析

符号表构建

在 `src/translation/build_sym.cpp` 中，增加 `SemPass1::visit(ast::DoWhileStmt *s)` 和 `SemPass1::visit(ast::ForStmt *s)` 函数。处理 `ForStmt` 时需要注意，第一，初始化、结束条件、更新语句均可能为 `NULL`，需要判断以避免 Segmentation fault；第二，for 循环的局部变量自带一个作用域，需要在 `SemPass1` 和 `SemPass2` 中开放关闭。

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

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

    if (NULL != s->init)        s->init->accept(this);
    if (NULL != s->condition)   s->condition->accept(this);
    if (NULL != s->update)      s->update->accept(this);
    s->loop_body->accept(this);
```

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

类型检查

在 `src/translation/type_check.cpp` 中，增加 `SemPass2::visit(ast::DoWhileStmt *s)` 和 `SemPass2::visit(ast::ForStmt *s)` 函数，保证类型检查可以递归到子节点。均为 `Statement`，不需要设置自身的 `ATTR(type)`。`ForStmt` 同样需要处理成员为 `NULL` 的情况。

```
void SemPass2::visit(ast::ForStmt *s) {
    scopes->open(s->ATTR(scope));
    if (NULL != s->init)        s->init->accept(this);
    if (NULL != s->condition) {
        s->condition->accept(this);
        if (!s->condition->ATTR(type)->equal(BaseType::Int))
            issue(s->condition->getLocation(), new BadTestExprError());
    }
    if (NULL != s->update)      s->update->accept(this);
    s->loop_body->accept(this);
    scopes->close();
}
```

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

中间代码生成

在 `src/translation/translation.cpp` 中，处理 `ForStmt`，`DoWhileStmt` 和 `ContStmt`。

本 step 的核心点是跳转标签的生成，并在递归处理循环嵌套时做好备份。

对于 for 循环，代码

```
for (Init; Test; Update )
    Body
```

可以转化为：

```
Init
_Begin_Loop:
    if (!Test)  goto _Exit
    Body
    goto _Update
_Update:
    Update
    goto _Begin_Loop
_Exit:
```

据此分析，实现 `void Translation::visit(ast::ForStmt *s)` 函数。

```
void Translation::visit(ast::ForStmt *s) {
    Label L_Begin_Loop = tr->getNewLabel();
    Label L_Update = tr->getNewLabel();
    Label L_Exit = tr->getNewLabel();

    Label old_break = current_break_label;
    current_break_label = L_Exit;

    Label old_cont = current_continue_label;
    current_continue_label = L_Update;

    if (NULL != s->init)        s->init->accept(this);

    tr->genMarkLabel(L_Begin_Loop);
    if (NULL != s->condition)   {
        s->condition->accept(this);
        tr->genJumpOnZero(L_Exit, s->condition->ATTR(val));
    }

    s->loop_body->accept(this);
    tr->genJump(L_Update);

    tr->genMarkLabel(L_Update);
    if (NULL != s->update)      s->update->accept(this);
    tr->genJump(L_Begin_Loop);

    tr->genMarkLabel(L_Exit);

    current_break_label = old_break;
    current_continue_label = old_cont;
}
```

对于 do-while 循环，代码

```
do{
    Body
} while(Test)
```

可以转化为：

```
_Begin_Loop:
    Body
    if (!Test)  goto _Exit
    goto _Begin_Loop
_Exit:
```

据此分析，实现 `void Translation::visit(ast::DoWhileStmt *s)` 函数。

```
void Translation::visit(ast::DoWhileStmt *s) {
    Label L_Begin_Loop = tr->getNewLabel();
    Label L_Exit = tr->getNewLabel();

    Label old_break = current_break_label;
    current_break_label = L_Exit;

    tr->genMarkLabel(L_Begin_Loop);
    s->loop_body->accept(this);
    s->condition->accept(this);
    tr->genJumpOnZero(L_Exit, s->condition->ATTR(val));

    tr->genJump(L_Begin_Loop);

    tr->genMarkLabel(L_Exit);

    current_break_label = old_break;
}
```

思考题

1. 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;
 }
}
```

中间代码（默认返回0）

```
_main:                          ####### B0 BEGIN #######
    T1 <- 2
    T0 <- T1
    T2 <- 3
    T3 <- (T0 < T2)
    if (T3 == 0) jump __L1      ######## B0 END ########
    T5 <- 3                     ####### B1 BEGIN #######
```

```
    T4 <- T5
    return T4                   ######## B1 END ########
    return T0                   ########## B2 ##########
    jump   __L2                 ########## B3 ##########
__L1:                           ########## B4 ##########
__L2:                           ####### B5 BEGIN #######
    T6 <- 0
    return T6                   ######## B5 END ########
```

控制流图

2. 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？

   以如下代码为例：

```
void test (int n){
    int a = 0;
    while (a < n)
        a = a + 1;
}
```

	第一种	第二种
cond	n+1	n+1
beqz	n+1	1
body	n	n
br	n	
bnez		n+1
合计	4n+2	3n+3

n = 1 时，两种方法相同；n > 2 时，第二种方法更好。

代码借鉴

无。