Stage-3

2019011008

无92 刘雪枫

目录

Step7

代码修改

本步骤需实现作用域与块语句。由于作用域栈等相关操作已经编写完成，因此只需增加语法分析即可。

修改语法规则如下：

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

```
- FuncDefn : Type IDENTIFIER LPAREN FormalList RPAREN LBRACE StmtList RBRACE {
-               $$ = new ast::FuncDefn($2,$1,$4,$7,POS(@1));
+ FuncDefn : Type IDENTIFIER LPAREN FormalList RPAREN CompStmt {
+               $$ = new ast::FuncDefn($2,$1,$4,$6->stmts,POS(@1));
+               delete $6;
            } |
            Type IDENTIFIER LPAREN FormalList RPAREN SEMICOLON{
               $$ = new ast::FuncDefn($2,$1,$4,new
ast::EmptyStmt(POS(@6)),POS(@1));
            }
```

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;
}
}
```

答：将该代码分成基本块如下：

```
// 基本块 0（B0）
int a = 2;
if (a < 3) {
```

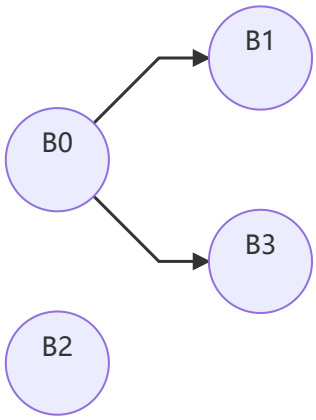
```
// 基本块 1（B1）
    {
        int a = 3;
        return a;
    }
```

```
// 基本块 2（B2）
    return a;
```

根据 C 语言规定，`main` 函数末尾隐含 `return 0;`，因此：

```
// 基本块 3（B3）
return 0; // 根据 C 语言规定，`main` 函数默认返回 0
```

则控制流图如下：



Step8

代码修改

`break` 检查

首先，增加对 `break` 是否在循环中的检查支持。先增加对 `current_break_label` 的初始化，将其初始化为 `nullptr`：

```
  class Translation : public ast::Visitor {
      /* ... */
    private:
      tac::TransHelper *tr;
-     tac::Label current_break_label;
+     tac::Label current_break_label = nullptr;
      // TODO: label for continue
  };
```

在翻译阶段检查其是否为 `nullptr`，若是则不在循环内，增加一个报错：

```
void Translation::visit(ast::BreakStmt *s) {
    if (current_break_label == nullptr) {
        mind::err::issue(s->getLocation(),
                         new mind::err::SyntaxError("Break not in loop!"));
    }
    tr->genJump(current_break_label);
}
```

`for` 循环语法支持

为了支持 `for` 循环语法，做出改动如下。

新增 `ForStmt` 结点类型

为了支持 `for` 语句，需要增加 `ForStmt` 结点。在 `ast/ast.hpp` 中编写声明。由于 `for` 循环的初始条件位置声明的变量需要在一个单独的作用域中，因此 `for` 循环需要带有一个作用域 `ATTR(scope)`，其余和 `WhileStmt` 类似：

```
```

```
/* Node representing a for statement.
 *
 * SERIALIZED FORM:
 *   (for INIT CONDITION UPDATE LOOP_BODY)
 */

class ForStmt : public Statement {
  public:
    ForStmt(Statement *init, Expr *cond, Expr *update, Statement *loop_body,
            Location *l);

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

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

新建 `ast/ast_for_stmt.cpp` 文件，编写各个函数的定义。由于内容和 `WhileStmt` 基本一致，此处略去。

为了将其加入编译，对 `Makefile` 做出如下更改：

```
  AST     = ast/ast.o ast/ast_add_expr.o ast/ast_and_expr.o  ast/ast_or_expr.o\
            ast/ast_not_expr.o \
            ast/ast_program.o ast/ast_func_defn.o \
            ast/ast_return_stmt.o ast/ast_sub_expr.o \
-           ast/ast_var_decl.o ast/ast_var_ref.o  ast/ast_while_stmt.o
ast/ast_comp_stmt.o
+           ast/ast_var_decl.o ast/ast_var_ref.o  ast/ast_while_stmt.o
ast/ast_for_stmt.o ast/ast_comp_stmt.o
```

```
+ ast/ast_for_stmt.o: config.hpp 3rdparty/boehmgc.hpp define.hpp
+ ast/ast_for_stmt.o: 3rdparty/list.hpp error.hpp ast/ast.hpp ast/visitor.hpp
```

并在 `ast/ast.hpp` 中增加枚举类型：

```
   class ASTNode {
          /* ... */
          WHILE_STMT,
          FOD,
+         FOR_STMT
      } NodeType;
```

编写 `for` 循环语句需要的文法和语法规则

首先在 `frontend/scanner.l` 中增加文法规则，为 `for` 生成相应的 `FOR` token：

```
"for"       { return yy::parser::make_FOR  (loc);       }
```

在 `frontend/parser.y` 中也增加：

```
FOR "for"
```

在 `frontend/parser.y` 中增加语法规则：

```
  Stmt        : ReturnStmt {$$ = $1;} |
                ExprStmt   {$$ = $1;} |
                IfStmt     {$$ = $1;} |
                WhileStmt  {$$ = $1;} |
+               ForStmt    {$$ = $1;} |
                CompStmt   {$$ = $1;} |
                BREAK SEMICOLON
                  {$$ = new ast::BreakStmt(POS(@1));} |
```

由于 `for` 循环的循环条件表达式和更新表达式可以为空，因此为了简化语法，增加非终结符 `NulableExpr` 作为"可空表达式"，然后再编写 `ForStmt` 的文法规则：

```
NulableExpr : /* empty */
                { $$ = nullptr; }
            | Expr
                { $$ = $1; }
            ;
ForStmt     : FOR LPAREN VarDecl NulableExpr SEMICOLON NulableExpr RPAREN Stmt
                { $$ = new ast::ForStmt($3, $4, $6, $8, POS(@1)); }
            | FOR LPAREN Expr SEMICOLON NulableExpr SEMICOLON NulableExpr RPAREN
Stmt
                { $$ = new ast::ForStmt(new ast::ExprStmt($3, POS(@3)), $5, $7,
$9, POS(@1)); }
            | FOR LPAREN SEMICOLON NulableExpr SEMICOLON NulableExpr RPAREN Stmt
                { $$ = new ast::ForStmt(new ast::EmptyStmt(POS(@3)), $4, $6, $8,
POS(@1)); }
            ;
```

并声明两个新增的非终结符类型：

```
%nterm<mind::ast::Statement*> ForStmt
%nterm<mind::ast::Expr*> NulableExpr
```

增加对 `ForStmt` 结点的访问代码

首先，在 `ast/visitor.hpp` 中对 `Visitor` 基类增加访问 `ForStmt` 的成员函数：

```
  class Visitor {
      /* ... */
      virtual void visit(WhileStmt *) {}
+     virtual void visit(ForStmt *) {}
      virtual void visit(EmptyStmt *) {}
      /* ... */
  }
```

建立符号表阶段

在 `translation/build_sym.cpp` 中增加对 `ForStmt` 的访问者代码，建立符号表。先加入声明：

```
  class SemPass1 : public ast::Visitor {
      /* ... */
      virtual void visit(ast::WhileStmt *);
+     virtual void visit(ast::ForStmt *);
      virtual void visit(ast::CompStmt *);
      /* ... */
  }
```

然后编写代码。先为 `for` 建立一个新的局部作用域，然后依次对初始化语句、判断条件、循环体和更新表达式建立符号表。根据之前编写的文法规则，`condition` 和 `update` 均可能为 `nullptr`，故需要判断是否为空指针在进行访问。而 `init` 在初始化语句为空时为 `EmptyStmt`，不需要判空。

```
void SemPass1::visit(ast::ForStmt *s) {
    Scope *scope = new LocalScope();
    s->ATTR(scope) = scope;
    scopes->open(scope);
    s->init->accept(this);
    s->condition ? s->condition->accept(this) : void();
    s->loop_body->accept(this);
    s->update ? s->update->accept(this) : void();
    scopes->close();
}
```

类型检查阶段

类型检查阶段在 `translation/type_check.cpp` 中，实现是类似的：

```
void SemPass2::visit(ast::ForStmt *s) {
    scopes->open(s->ATTR(scope));

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

    s->loop_body->accept(this);

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

    scopes->close();
}
```

生成三地址码阶段

在 `translation/translation.cpp` 中编写三地址码生成代码，如下：

```
void Translation::visit(ast::ForStmt *s) {
    s->init->accept(this);

    Label L1 = tr->getNewLabel();
    Label L2 = tr->getNewLabel();

    Label old_break = current_break_label;
    current_break_label = L2;

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

    s->loop_body->accept(this);
    s->update ? s->update->accept(this) : void();
    tr->genJump(L1);

    tr->genMarkLabel(L2);

    current_break_label = old_break;
}
```

`continue` 语法支持

增加 `ContStmt` 结点类型

代码框架给出了 `ContStmt` 的结点类型定义，但没给出完整实现，需要自己添加。首先，增加相应的枚举类型：

```
      class ASTNode {
          /* ... */
          BREAK_STMT,
+         CONT_STMT,
          CALL_EXPR,
          /* ... */
      }
```

其成员函数实现与 `BreakStmt` 几乎完全一致，在 `ast/ast_while_stmt.cpp` 中增加相应的代码，此处略过。

增加词法和语法分析

首先在 `frontend/scanner.l` 中增加 `continue` 的词法：

```
"continue"      { return yy::parser::make_CONTINUE (loc);      }
```

在 `frontend/parser.y` 中也增加：

```
CONTINUE "continue"
```

然后增加 `continue` 语句的文法规则：

```
  Stmt        : ReturnStmt {$$ = $1;} |
                CompStmt   {$$ = $1;} |
                BREAK SEMICOLON
                  {$$ = new ast::BreakStmt(POS(@1));} |
+               CONTINUE SEMICOLON
+                 {$$ = new ast::ContStmt(POS(@1));} |
                SEMICOLON
                  {$$ = new ast::EmptyStmt(POS(@1));}
              ;
```

增加 `continue` 语句的翻译代码

首先类似于 `break`，增加 `current_continue_label` 记录 `continue` 跳转的位置：

```
  class Translation : public ast::Visitor {
      /* ... */
      tac::Label current_break_label = nullptr;
+     tac::Label current_continue_label = nullptr;
 };
```

然后在翻译 `ContStmt` 时检查是否在循环内：

```
/* Translating an ast::ContStmt node.
 */
void Translation::visit(ast::ContStmt *s) {
    if (current_continue_label == nullptr) {
        mind::err::issue(s->getLocation(),
                         new mind::err::SyntaxError("Continue not in loop!"));
    }
    tr->genJump(current_continue_label);
}
```

增加 `while` 语句和 `for` 语句内 `continue` 跳转位置的记录

对于 `while` 语句，设置 `continue` 跳转标签：

```
  void Translation::visit(ast::WhileStmt *s) {

      Label old_break = current_break_label;
      current_break_label = L2;
+     Label old_continue = current_continue_label;
+     current_continue_label = L1;

      tr->genMarkLabel(L1);
      s->condition->accept(this);
      /* ... */

      tr->genMarkLabel(L2);

+     current_continue_label = old_continue;
      current_break_label = old_break;
  }
```

对于 `for` 语句，需要引入第三个标签 `L3`，作为 `continue` 跳转的位置（即更新表达式求值之前）

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

      Label L1 = tr->getNewLabel();
      Label L2 = tr->getNewLabel();
+     Label L3 = tr->getNewLabel();

      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->update ? s->update->accept(this) : void();
      tr->genJump(L1);

      tr->genMarkLabel(L2);

+     current_continue_label = old_continue;
      current_break_label = old_break;
  }
```

`do` ... `while` 语句语法支持

`do` ... `while` 语句语法支持，需要增加新的语法结点 `DoStmt`。在内容上与 `WhileStmt` 几乎完全一致。新建 `ast/ast_do_stmt.cpp` 用于编写其实现，此处略。

然后增加对 `do` 终结符的词法分析：

```
"do"       { return yy::parser::make_DO  (loc);       }
```

并增加文法规则：

```
  Stmt        : ReturnStmt {$$ = $1;} |
                /* ... */
                WhileStmt  {$$ = $1;} |
+               DoStmt  {$$ = $1;} |
                ForStmt    {$$ = $1;} |
                CompStmt   {$$ = $1;} |
                BREAK SEMICOLON
```

```
DoStmt   : DO Stmt WHILE LPAREN Expr RPAREN SEMICOLON
                { $$ = new ast::DoStmt($5, $2, POS(@1)); }
           ;
```

建立符号表的代码、类型检查的代码与 `WhileStmt` 几乎完全一致，只是循环体与判断条件的顺序不同：

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

```
void SemPass2::visit(ast::DoStmt *s) {
    s->loop_body->accept(this);

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

最后增加三地址码生成：

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

    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_continue_label = old_continue;
    current_break_label = old_break;
}
```

思考题

1. 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？

答：在执行的指令条数这个角度，第二种更好。因为在最后一次跳出循环时的判断（即判断条件为假）时，第一种还要跳回循环条件进行判断，判断条件为假后再跳转到后面执行，而第二种在执行完循环体后直接执行循环条件，判断为假则直接向下执行，会比第一种方案跳转次数少，即少了两条跳转指令的执行。