Stage-2

2019011008

无92 刘雪枫

目录

Step5

代码修改

本步骤需实现局部变量和赋值

由于本步骤未增加任何新的终结符，因此不需要修改词法分析文件

对变量定义，以及赋值表达式的类型检查已经由框架写出，因此不需增加

修改语法分析文件

增加赋值与变量定义的语法：

```
   Stmt        : ReturnStmt {$$ = $1;} |
                 ExprStmt   {$$ = $1;} |
                 VarDecl    {$$ = $1;} |
+                IfStmt     {$$ = $1;} |
                 WhileStmt  {$$ = $1;} |
                 CompStmt   {$$ = $1;} |
```

```
+ VarDecl     : Type IDENTIFIER SEMICOLON
+                 { $$ = new ast::VarDecl($2, $1, POS(@1)); }
+              | Type IDENTIFIER ASSIGN Expr SEMICOLON
+                 { $$ = new ast::VarDecl($2, $1, $4, POS(@1)); }
+              ;
```

```
 Expr        : Expr QUESTION Expr COLON Expr
                 { $$ = new ast::IfExpr($1, $3, $5, POS(@2)); }
+            | AssignExpr
+            ;
+AssignExpr  : IDENTIFIER ASSIGN AssignExpr
+               { $$ = new ast::AssignExpr(new ast::VarRef($1, POS(@1)), $3,
POS(@2)); }
             | LogicalOrExpr
```

```
  PrimaryExpr :
      /* ... */
+             | IDENTIFIER
+                 { $$ = new ast::LvalueExpr(new ast::VarRef($1, POS(@1)),
POS(@1)); }
```

构建符号表

当变量定义时，将其加入符号表。在 `build_sym.cpp` 中添加代码：

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

    // 3. Declare the symbol in `scopes`
    // 4. Special processing for global variables
    // 5. Tag the symbol to `vdecl->ATTR(sym)`

    Variable *v = new Variable(vdecl->name, t, vdecl->getLocation());

    Symbol *sym = scopes->lookup(vdecl->name, vdecl->getLocation(), false);
    if (NULL != sym)
        issue(vdecl->getLocation(), new DeclConflictError(vdecl->name, sym));
    else
        scopes->declare(v);

    // TODO: Special processing for global variables

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

由于本步骤尚未实现全局作用域，因此暂时不考虑全局变量

增加三地址码生成

为 `LvalueExpr` 增加翻译代码，并为赋值表达式和变量定义增加三地址码生成代码：

```
void Translation::visit(ast::LvalueExpr *e) {
    // TODO
    e->lvalue->accept(this);
    mind_assert(e->lvalue->ATTR(lv_kind) == ast::Lvalue::SIMPLE_VAR);
    e->ATTR(val) = static_cast<ast::VarRef *>(e->lvalue)->ATTR(sym)->getTemp();
}
```

```
void Translation::visit(ast::VarDecl *decl) {
    // TODO
    decl->ATTR(sym)->attachTemp(tr->getNewTempI4());
    if (decl->init) {
        decl->init->accept(this);
        tr->genAssign(decl->ATTR(sym)->getTemp(), decl->init->ATTR(val));
    }
}
```

```
void Translation::visit(ast::AssignExpr *s) {
    // TODO
    s->left->accept(this);
    s->e->accept(this);
    tr->genAssign(static_cast<ast::VarRef *>(s->left)->ATTR(sym)->getTemp(),
                  s->e->ATTR(val));
    s->ATTR(val) = static_cast<ast::VarRef *>(s->left)->ATTR(sym)->getTemp();
}
```

将三地址码生成汇编码

使用 unary TAC，生成 `mv` 指令：

```
    case Tac::ASSIGN:
        emitUnaryTac(RiscvInstr::ASSIGN, t);
        break;
```

```
    case RiscvInstr::ASSIGN:
        oss << "mv" << i->r0->name << ", " << i->r1->name;
        break;
```

思考题

1. 我们假定当前栈帧的栈顶地址存储在 `sp` 寄存器中，请写出一段 **risc-v 汇编代码**，将栈帧空间扩大 16 字节。（提示1：栈帧由高地址向低地址延伸；提示2：risc-v 汇编中 addi reg0, reg1, <立即数> 表示将 reg1 的值加上立即数存储到 reg0 中。）

   答：`addi sp, sp, -16`

2. 有些语言允许在同一个作用域中多次定义同名的变量，例如这是一段合法的 Rust 代码（你不需要精确了解它的含义，大致理解即可）：

   ```
   fn main() {
     let a = 0;
     let a = f(a);
     let a = g(a);
   }
   ```

   其中 `f(a)` 中的 `a` 是上一行的 `let a = 0;` 定义的，`g(a)` 中的 `a` 是上一行的 `let a = f(a);`。

   如果 MiniDecaf 也允许多次定义同名变量，并规定新的定义会覆盖之前的同名定义，请问在你的实现中，需要对定义变量和查找变量的逻辑做怎样的修改？（提示：如何区分一个作用域中**不同位置**的变量定义？）

> 答：当同名变量定义时，使用新定义的变量创建一个新的符号，替换掉原来本作用域内符号表中的变量。这样在之后再进行符号表查找时，在本作用域中查找到的便是最近一次放入符号表内的变量了

Step6

代码修改

由于 `if` 语句的实现已经给出，因此只需增加条件表达式的实现，以及增加 `BlockItem` 用于后续使用

首先修改词法分析文件，增加对 `?` 和 `:` 的词法分析：

```
+ "?"          { return yy::parser::make_QUESTION  (loc);       }
+ ":"          { return yy::parser::make_COLON  (loc);       }
```

然后修改语法分析文件

首先，增加 `BlockItem`：

```
  StmtList    : /* empty */
                  { $$ = new ast::StmtList(); }
-             | StmtList Stmt
+             | StmtList BlockItem
                  { $1->append($2);
                    $$ = $1; }
              ;
+ BlockItem   : Stmt  {$$ = $1;}      |
+              VarDecl {$$ = $1;}
+             ;
```

```
  Stmt        : ReturnStmt {$$ = $1;} |
                ExprStmt   {$$ = $1;} |
-               VarDecl    {$$ = $1;} |
                IfStmt     {$$ = $1;} |
                WhileStmt  {$$ = $1;} |
                CompStmt   {$$ = $1;} |
```

然后增加对条件表达式的支持：

```
- Expr        : Expr QUESTION Expr COLON Expr
-                { $$ = new ast::IfExpr($1, $3, $5, POS(@2)); }
-             | AssignExpr
+ Expr        : AssignExpr
              ;
```

```
  AssignExpr  : IDENTIFIER ASSIGN AssignExpr
                  { $$ = new ast::AssignExpr(new ast::VarRef($1, POS(@1)), $3,
POS(@2)); }
-             | LogicalOrExpr
+             | ConditionalExpr
              ;
+ ConditionalExpr     : LogicalOrExpr
+                     | Expr QUESTION Expr COLON Expr
+                         { $$ = new ast::IfExpr($1, $3, $5, POS(@2)); }
+                     ;
```

然后增加对条件表达式的类型检查：

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

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

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

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

以及条件表达式翻译为三地址码：

```
void Translation::visit(ast::IfExpr *s) {
    Label L1 = tr->getNewLabel();
    Label L2 = tr->getNewLabel();
    s->ATTR(val) = tr->getNewTempI4();
    s->condition->accept(this);
    tr->genJumpOnZero(L1, s->condition->ATTR(val));

    s->true_brch->accept(this);
    tr->genAssign(s->ATTR(val), s->true_brch->ATTR(val));
    tr->genJump(L2);

    tr->genMarkLabel(L1);
    s->false_brch->accept(this);
    tr->genAssign(s->ATTR(val), s->false_brch->ATTR(val));

    tr->genMarkLabel(L2);
}
```

思考题

1. 你使用语言的框架里是如何处理悬吊 else 问题的？请简要描述。

   答：C++ 语言框架使用 bison 进行语法分析，其默认在 shift-reduce conflict 的时候选择 shift，从而对悬吊 else 进行就近匹配

2. 在实验要求的语义规范中，条件表达式存在短路现象。即：

   ```
   int main() {
       int a = 0;
       int b = 1 ? 1 : (a = 2);
       return a;
   }
   ```

   会返回 0 而不是 2。如果要求条件表达式不短路，在你的实现中该做何种修改？简述你的思路。

   答：具有短路特性时，部分代码实现如下：

   ```
   tr->genJumpOnZero(L1, s->condition->ATTR(val));

   s->true_brch->accept(this);
   tr->genAssign(s->ATTR(val), s->true_brch->ATTR(val));
   tr->genJump(L2);

   tr->genMarkLabel(L1);
   s->false_brch->accept(this);
   tr->genAssign(s->ATTR(val), s->false_brch->ATTR(val));

   tr->genMarkLabel(L2);
   ```

   这种实现当中，通过跳转，避免了 `true` 和 `false` 的其中一个分支代码的执行。若要不短路，只需让两个分支的代码均移到跳转之前即可，即改成如下形式：

   ```
   s->true_brch->accept(this);
   s->false_brch->accept(this);

   tr->genJumpOnZero(L1, s->condition->ATTR(val));

   tr->genAssign(s->ATTR(val), s->true_brch->ATTR(val));
   tr->genJump(L2);

   tr->genMarkLabel(L1);
   tr->genAssign(s->ATTR(val), s->false_brch->ATTR(val));

   tr->genMarkLabel(L2);
   ```