# MiniDecaf Stage 2: 变量和语句

> My-laniaKeA

## 实验内容

### Step 5: 局部变量和赋值

#### 词法分析和语法分析

在 `src/frontend/scanner.l` 中添加 `=` 的词法规则：

```
"="         { return yy::parser::make_ASSIGN (loc);       }
```

在 `src/frontend/parser.y` 中添加 `AssignExpr`，`DeclrStmt` 和 `IDENTIFIER` 的语法规则：

```
DeclrStmt   : Type IDENTIFIER ASSIGN Expr
                {$$ = new ast::VarDecl($2, $1, $4, POS(@2));}
            | Type IDENTIFIER
                {$$ = new ast::VarDecl($2, $1, POS(@2));}
            ;
Expr        : AssignExpr    { $$ = $1; }
AssignExpr  : Logical_Or    { $$ = $1; }
            | IDENTIFIER ASSIGN Expr
                { $$ = new ast::AssignExpr($1, $3, POS(@1)); }
            ;
Logical_Or  : Logical_And   { $$ = $1; }
            | Logical_Or OR Logical_And
                { $$ = new ast::IntConst($1, POS(@1)); }
            | LPAREN Expr RPAREN
                { $$ = $2; }
            | IDENTIFIER
                { $$ = new ast::LvalueExpr($1, POS(@1)); }
            ;
```

在本 step 中，更改了部分 AST 节点的定义与接口。

- `AssignExpr` 的成员由

  ```
  Lvalue *left; Expr *e; Location *l;
  ```

  变为

  ```
  LvalueExpr *left; Expr *e; Location *l;
  ```

- 增加如下构造函数（`lv_name` 是左值表达式中变量的名字）：

```
AssignExpr::AssignExpr(std::string lv_name, Expr *e0, Location *l);
LvalueExpr::LvalueExpr(std::string lv_name, Location *l);
```

## 语义分析

### 符号表构建

在 `src/translation/build_sym.cpp` 中，新建一个符号 `v`。在作用域栈中检查当前作用域有无同名符号：有则报错；无则声明符号，并将符号存在 AST 节点 `vdecl` 上。如果变量定义设置了初值，还需要继续访问初值表达式 `vdecl->init`。

```
void SemPass1::visit(ast::VarDecl *vdecl) {
    Type *t = NULL;
    vdecl->type->accept(this);
    t = vdecl->type->ATTR(type);

    // Add a new symbol to a scope
    // 1. Create a new `Variable` symbol
    Variable *v = new Variable(vdecl->name, t, vdecl->getLocation());
    // 2. Check for conflict in `scopes`, which is a global variable refering to
    // a scope stack
    Symbol *sym = scopes->lookup(vdecl->name, vdecl->getLocation(), true);
    if (NULL != sym && sym->getScope() == scopes->top())
        issue(vdecl->getLocation(), new DeclConflictError(vdecl->name, sym));
    // 3. Declare the symbol in `scopes`
    else
        scopes->declare(v);
    // 4. Tag the symbol to `vdecl->ATTR(sym)`
    vdecl->ATTR(sym) = v;
    // 5. Visit initialization expression if existed
    if (NULL != vdecl->init)
        vdecl->init->accept(this);
}
```

## 中间代码生成

在 `src/translation/translation.cpp` 中，得到 `LvalueExpr`，`AssignExpr` 和 `VarDecl` 的 val 属性并生成中间代码。

```
void Translation::visit(ast::AssignExpr *s) {
    s->left->accept(this);
    s->e->accept(this);

    tr->genAssign(s->left->ATTR(val), s->e->ATTR(val));
    s->ATTR(val) = s->left->ATTR(val);
}

void Translation::visit(ast::LvalueExpr *e) {
    ((ast::VarRef*)e->lvalue)->accept(this);
    const auto &sym = ((ast::VarRef*)e->lvalue)->ATTR(sym);
    e->ATTR(val) = sym->getTemp();
}
```

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

### 目标平台汇编代码生成

使用 mv 指令来翻译中间表示里的 ASSIGN 指令。

在 `src/asm/riscv_md.cpp` 的 `RiscvDesc::emitTac` 函数中，增加 TAC 为 `ASSIGN` 的情况，并实现 `emitAssignTac` 函数。

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

```
void RiscvDesc::emitAssignTac(RiscvInstr::OpCode op, Tac *t) {
    // eliminates useless assignments
    if (!t->LiveOut->contains(t->op0.var))
        return;

    int r1 = getRegForRead(t->op1.var, 0, t->LiveOut);
    int r0 = getRegForWrite(t->op0.var, r1, 0, t->LiveOut);

    addInstr(op, _reg[r0], _reg[r1], NULL, 0, EMPTY_STR, NULL);
}
```

## Step 6: if 语句和条件表达式

### 词法分析和语法分析

在 `src/frontend/scanner.l` 中添加 `?` 和 `:` 的词法规则：

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

在 `src/frontend/parser.y` 中添加 `BlockItemList`，`BlockItem` 和 `Conditional` 的语法规则，修改 `DeclrStmt` 的语法规则：

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

```
            ;

Stmt        : ReturnStmt {$$ = $1;}|
              ExprStmt   {$$ = $1;}|
              IfStmt     {$$ = $1;}|
              SEMICOLON
                {$$ = new ast::EmptyStmt(POS(@1));}
            ;
DeclrStmt   : Type IDENTIFIER ASSIGN Expr SEMICOLON
                {$$ = new ast::VarDecl($2, $1, $4, POS(@2));}
            | Type IDENTIFIER SEMICOLON
                {$$ = new ast::VarDecl($2, $1, POS(@2));}
            ;
IfStmt      : IF LPAREN Expr RPAREN Stmt
                { $$ = new ast::IfStmt($3, $5, new ast::EmptyStmt(POS(@5)),
POS(@1)); }
            | IF LPAREN Expr RPAREN Stmt ELSE Stmt
                { $$ = new ast::IfStmt($3, $5, $7, POS(@1)); }
            ;
AssignExpr  : Conditional
                { $$ = $1; }
            | IDENTIFIER ASSIGN Expr
                { $$ = new ast::AssignExpr($1, $3, POS(@1)); }
            ;
Conditional : Logical_Or
                { $$ = $1; }
            | Logical_Or QUESTION Expr COLON Conditional
                { $$ = new ast::IfExpr($1, $3, $5, POS(@2)); }
            ;
```

## 语义分析

### 类型检查

在 `src/translation/type_check.cpp` 中，对 `IfExpr` 进行类型检查。

- 递归访问三个子节点。
- `condition` 的类型应为 `int`。
- `true_brch` 与 `false_brch` 的类型应一致。
- `e` 的类型应为 `true_brch` 的类型。

```
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);
    e->false_brch->accept(this);

    if (!e->true_brch->ATTR(type)->equal(e->false_brch->ATTR(type))) {
        issue(e->true_brch->getLocation(), new BadTestExprError());
        ;
    }
```

```
    e->ATTR(type) = e->true_brch->ATTR(type);
}
```

### 中间代码生成

在 `src/translation/translation.cpp` 中，得到 `IfExpr` 的 val 属性并生成中间代码。由于短路求值特性，在 jump 之后再访问对应的分支子节点。

```
void Translation::visit(ast::IfExpr *e) {
    e->ATTR(val) = tr->getNewTempI4();  // temp variable for expression result

    Label L1 = tr->getNewLabel(); // entry of the false branch
    Label L2 = tr->getNewLabel(); // exit
    e->condition->accept(this);
    tr->genJumpOnZero(L1, e->condition->ATTR(val));

    e->true_brch->accept(this);
    tr->genAssign(e->ATTR(val), e->true_brch->ATTR(val));
    tr->genJump(L2); // done

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

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

## 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。）

   ```
   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？）

定义变量：在 `SemPass1::visit(ast::VarDecl *vdecl)` 函数中，创建 `Variable` symbol 时，将 symbol 的名称后附上定义位置。例如，在 `(3,2)` 定义了变量 `a`，则在创建 symbol 时，实际采用的是 `a_3_2`。

查找变量：根据变量名称与引用位置进行查询。查询结果为行数小于引用所在行的声明中，行数最大的定义。例如，变量 `a` 在 `(1, 2)` 和 `(5, 2)` 均有定义，除此之外没有定义，`(4, 3)` 处和 `(5, 6)` 处的引用都应当查找得到 `a_1_2`。

3. 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。

   Bison 默认在 shift-reduce conflict 的时候选择shift，从而对悬挂else进行就近匹配。参见 Bison 文档 Shift/Reduce (Bison 3.8.1)

   > Bison is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations.

4. 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。

   在中间代码生成时进行修改。

   现有实现为：在生成跳转中间代码后，再去访问子节点 `ture_brch` 和 `false_brch` 并为条件表达式赋值。

   修改实现为：先访问子节点 `ture_brch` 和 `false_brch`，再生成跳转中间代码并为条件表达式赋值。具体代码如下。

```
void Translation::visit(ast::IfExpr *e) {
    // visit children
    e->condition->accept(this);
    e->true_brch->accept(this);
    e->false_brch->accept(this);

    // generate jumps
    Label L1 = tr->getNewLabel(); // entry of the false branch
    Label L2 = tr->getNewLabel(); // exit
    tr->genJumpOnZero(L1, e->condition->ATTR(val));
    tr->genAssign(e->ATTR(val), e->true_brch->ATTR(val));
    tr->genJump(L2); // done
    tr->genMarkLabel(L1);
    tr->genAssign(e->ATTR(val), e->false_brch->ATTR(val));
    tr->genMarkLabel(L2);
}
```

   针对思考题中示例代码，不再支持短路求值，结果如下：

```
FAIL testcases/step6/ternary_assign.c
==== Fail information (above: expected, below: actual) ========================
1c1
< 0
---
> 2
================================================================================
```

## 代码借鉴

无。