

# 编译原理 · stage-2 · 实验报告

计01 容逸朗 2020010869

## 实验内容

### 实验目标

本阶段需要支持 MiniDecaf 语言局部变量的定义和赋值、if 语句（框架已完成）以及条件表达式。

### 具体实现

#### 局部变量定义

首先增加非终结符 `VarDecl` 及对应的语法：

```
1 StmtList      : StmtList VarDecl
2                { $1→append($2);
3                $$ = $1; }
4                ;
5
6 VarDecl       : Type IDENTIFIER SEMICOLON
7                { $$ = new ast::VarDecl($2, $1, POS(@2)); } |
8                Type IDENTIFIER ASSIGN Expr SEMICOLON
9                { $$ = new ast::VarDecl($2, $1, $4, POS(@3)); }
10               ;
```

然后在 `build_sym` 阶段遍历 `VarDecl` 节点处理变量定义，具体来说是利用 `Scope` 将新增的变量加入符号表中，同时利用 `ScopeStack::lookup` 函数检查同一作用域内是否出现相同的变量名称，若无则可以利用 `ScopeStack::declare` 函数宣告变量，最后若有赋值语句则需要一併执行之。

```
1 void SemPass1::visit(ast::VarDecl *vdecl) {
2     Type *t = NULL;
3     vdecl→type→accept(this);
4     t = vdecl→type→ATTR(type);
5     Variable *v = new Variable(vdecl→name, t, vdecl→getLocation());
6
7     Symbol *sym = scopes→lookup(vdecl→name, vdecl→getLocation(), false);
8
9     if (NULL ≠ sym)
10         issue(vdecl→getLocation(), new DeclConflictError(vdecl→name, sym));
11     else
12         scopes→declare(v);
13
14     if (vdecl→init ≠ NULL)
15         vdecl→init→accept(this);
```

```

16
17     vdecl→ATTR(sym) = v;
18 }

```

类型检查只要递归成员变量检查即可。

翻译部分则比较简单，首先生成一个临时变量，再把此变量绑定到 **Variable**。最后若有赋值语句则需要递归执行，最后生成一条 **Assign** 语句即可。

```

1 void Translation::visit(ast::VarDecl *decl) {
2     Variable *var = decl→ATTR(sym);
3     Temp t = tr→getNewTempI4();
4     var→attachTemp(t);
5
6     if (decl→init ≠ NULL) {
7         decl→init→accept(this);
8         tr→genAssign(t, decl→init→ATTR(val));
9     }
10 }

```

## 局部变量赋值

首先增加非终结符 **VarRef**、**LvalueExpr** 及对应的语法：

```

1 Expr      : VarRef ASSIGN Expr
2           { $$ = new ast::AssignExpr($1, $3, POS(@2)); }
3           | LvalueExpr
4           { $$ = $1; }
5           ;
6
7 LvalueExpr : VarRef
8           { $$ = new ast::LvalueExpr($1, POS(@1)); }
9           ;
10
11 VarRef     : IDENTIFIER
12           { $$ = new ast::VarRef($1, POS(@1)); }
13           ;

```

类型检查只要递归成员变量检查即可。

翻译语句同样只需要递归执行左值语句和表达式语句即可，最后根据左值的结果绑定其类型和值，最后生成一条 **Assign** 语句即可。

```

1 void Translation::visit(ast::AssignExpr *s) {
2     s→left→accept(this);
3     s→e→accept(this);
4
5     switch (s→left→ATTR(lv_kind)) {
6         case ast::Lvalue::SIMPLE_VAR: {

```

```

7         const auto &sym = ((ast::VarRef*)s→left)→ATTR(sym);
8         s→ATTR(val) = sym→getTemp();
9         break;
10    }
11    default:
12        mind_assert(false); // impossible
13    }
14
15    tr→genAssign(s→ATTR(val), s→e→ATTR(val));
16 }

```

由于增加了新的 **Assign** 指令，因此需要在后端加入对应的发射语句，然而在本阶段可以利用寄存器表示变量，因此只需要 **MV** 指令即可完成任务。

## 条件表达式

首先增加终结符 **QUESTION** 及对应语法：

```

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

```

类型检查只要递归成员变量检查即可。

中间代码生成时需要根据逻辑顺序编写，同时需要注意 **Expr** 类型需要提供 **val** 供其他语句调用。

```

1 void Translation::visit(ast::IfExpr *s) {
2     Label L1 = tr→getNewLabel(); // entry of the false branch
3     Label L2 = tr→getNewLabel(); // exit
4
5     Temp t = tr→getNewTempI4();
6
7     s→condition→accept(this);
8     tr→genJumpOnZero(L1, s→condition→ATTR(val));
9
10    s→true_brch→accept(this);
11    tr→genAssign(t, s→true_brch→ATTR(val));
12    tr→genJump(L2);
13
14    tr→genMarkLabel(L1);
15    s→false_brch→accept(this);
16    tr→genAssign(t, s→false_brch→ATTR(val));
17
18    tr→genMarkLabel(L2);
19    s→ATTR(val) = t;
20 }

```

由于后端已经提供 **BEQZ** 和 **JUMP** 指令，因此不需要其他特别操作。

## 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。）

- 利用如下代码即可完成要求：

```
1 | 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）：

```
1 | fn main() {  
2 |     let a = 0;  
3 |     let a = f(a);  
4 |     let a = g(a);  
5 | }
```

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？）

- 只要更改语义分析阶段（`SemPass1`）对于同名变量的处理方式：
  - 即当 `ScopeStack::lookup` 的结果为空时才使用 `ScopeStack::declare` 的方法定义变量
  - 其余情况（如变量绑定）使用 `lookup` 的结果即可

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。

- 在语法分析阶段设置了两个不同的语法匹配 `if` 语句：

```
1 | IF LPAREN Expr RPAREN Stmt  
2 |   { $$ = new ast::IfStmt($3, $5, new ast::EmptyStmt(POS(@5)), POS(@1)); } |  
3 | IF LPAREN Expr RPAREN Stmt ELSE Stmt  
4 |   { $$ = new ast::IfStmt($3, $5, $7, POS(@1)); }
```

由于不带 `else` 的 `if` 语句优先级较高，因此当程序遇到悬吊 `else` 问题时会根据就近匹配原则处理。

4. 在实验要求的语义规范中，条件表达式存在短路现象。即：

```
1  int main() {  
2      int a = 0;  
3      int b = 1 ? 1 : (a = 2);  
4      return a;  
5  }
```

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。

- 在中间代码生成时需要增加临时变量储存两个表达式的值，然后通过条件的成功与否再绑定返回值：

```
1  void Translation::visit(ast::IfExpr *s) {  
2      Label L1 = tr→getNewLabel(); // entry of the false branch  
3      Label L2 = tr→getNewLabel(); // exit  
4  
5      Temp t = tr→getNewTempI4();  
6      Temp t1 = tr→getNewTempI4();  
7      Temp t2 = tr→getNewTempI4();  
8  
9      s→condition→accept(this);  
10  
11     s→true_brch→accept(this);  
12     tr→genAssign(t1, s→true_brch→ATTR(val));  
13  
14     s→false_brch→accept(this);  
15     tr→genAssign(t2, s→false_brch→ATTR(val));  
16  
17     tr→genJumpOnZero(L1, s→condition→ATTR(val));  
18  
19     tr→genAssign(t, t1);  
20     tr→genJump(L2);  
21  
22     tr→genMarkLabel(L1);  
23     tr→genAssign(t, t2);  
24  
25     tr→genMarkLabel(L2);  
26     s→ATTR(val) = t;  
27 }
```

经测试，题干中的例子利用上面的代码会输出 2 而不是 0，此时条件表达式不短路，满足题干要求。

## 参考

实现代码的过程中参考了[实验思路指导与问答墙](#)。