链滴

# Kaleidoscope 系列第七章：扩展语言—可变变量

作者：Hanseltu
原文链接：https://ld246.com/article/1570072984687
来源网站：链滴

原文链接: Kaleidoscope系列第七章：扩展语言—可变变量

本文是使用LLVM开发新语言Kaleidoscope教程系列第七章，继续扩展Kaleidoscope语言特性，增可变变量处理。

# 第七章简介

欢迎来到“使用LLVM开发新语言Kaleidoscope教程”教程的第七章。在第一章至第六章中，我们构了一种尽管简单的但是非常像样的 函数式编程语言。在我们的开发过程中，我们学习了一些解析技术包括如何构建和表示AST，如何构建LLVM IR，如何优化最终代码以及使用JIT对其进行编译等。

尽管Kaleidoscope作为一种功能语言很有意思，但它具有功能性的事实使其“太容易”为其生成LLVM IR。特别是，一种功能语言使以 SSA形式 直接构建LLVM IR非常容易。由于LLVM要求输入代码为SS格式，所以这是一个非常好的属性，对于新手来说，如何为具有可变变量的命令式语言生成代码通常不明确。

本章的简短（很高兴）总结是，前端无需重新构建SSA格式，也就是说，虽然对于某些人来说它的工方式有些出乎意料，但是LLVM为此提供了高度调优和经过良好测试的支持。

# 为什么很难?

要了解可变变量为何导致SSA构造复杂，请考虑以下极其简单的C示例：

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
```

在这种情况下，我们有变量 X ，其值取决于程序中执行的路径。由于在返回指令之前X可能有两个不的值，因此将插入PHI节点以合并这两个值。对于此示例，我们想要的LLVM IR如下所示：

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
```

```
  ret i32 %X.2
}
```

在此示例中，G和H全局变量的负载在LLVM IR中是显式的，并且它们位于if语句（cond_true / cond_alse）的then / else分支中。为了合并输入值，cond_next块中的X.2 phi节点根据控制流的来源来选正确的值：如果控制流来自cond_false块，则X.2获得X的值0.1。或者，如果控制流来自cond_true，它将获得X.0的值。本章的目的不是解释SSA表单的详细信息。有关更多信息，请参见许多 在线参考料。

本文的问题是“在降低对可变变量的分配时，谁放置phi节点？”。这里的问题是LLVM 要求 其IR必为SSA形式：没有“non-ssa”模式。但是，SSA构造需要非平凡的算法和数据结构，因此，每个前都必须重现此逻辑既不方便又浪费。

## LLVM中的内存模型

这里的“窍门”是，尽管LLVM确实要求所有寄存器值都采用SSA形式，但它并不要求（或允许）存对象采用SSA形式。在上面的示例中，请注意，来自G和H的负载是对G和H的直接访问：它们没有重名或版本化。这与其他一些尝试对内存对象进行版本控制的编译器系统不同。在LLVM中，不是将对存的数据流分析编码为LLVM IR，而是使用按需计算的分析Passes来处理。

考虑到这一点，高级想法是我们要为函数中的每个可变对象创建一个堆栈变量（由于存在堆栈中而位内存中）。要利用此技巧，我们需要讨论LLVM如何表示堆栈变量。

在LLVM中，所有内存访问都通过加载/存储指令进行了明确显示，并且经过精心设计，使其不具有（不需要）address-of 运算符。注意，即使变量定义为“i32”，@ G / @ H全局变量的类型实际上还“i32 *”。这意味着@G为全局数据区域中的i32定义了_空间_，但其_名称_实际上是指该空间的地。堆栈变量的工作方式相同，除了堆栈变量不是使用全局变量定义声明外，它们使用LLVM alloca指令明：

```
define i32 @example() {
entry:
  %X = alloca i32           ; type of %X is i32*.
  ...
  %tmp = load i32* %X       ; load the stack value %X from the stack.
  %tmp2 = add i32 %tmp, 1   ; increment it
  store i32 %tmp2, i32* %X  ; store it back
  ...
```

此代码显示了如何在LLVM IR中声明和操作堆栈变量的示例。用alloca指令分配的堆栈内存是完全通的：您可以将堆栈插槽的地址传递给函数，可以将其存储在其他变量中，等等。在上面的示例中，我可以重写该示例以使用alloca技术来避免使用PHI节点：

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next
```

```
cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32* %X       ; Read X
  ret i32 %X.2
}
```

这样，我们发现了一种无需创建任何Phi节点即可处理任意可变变量的方法：

1. 每个可变变量成为堆栈分配。
2. 每次读取变量都会成为堆栈的负载。
3. 变量的每次更新都将存储到堆栈中。
4. 获取变量的地址仅直接使用堆栈地址。

尽管此解决方案解决了我们迫在眉睫的问题，但它引入了另一个问题：我们现在显然已经为非常简单常见的操作引入了很多堆栈流量，这是一个主要的性能问题。对我们来说幸运的是，LLVM优化器具一个名为“mem2reg”的高度优化的优化通道，可以处理这种情况，将这样的分配提升到SSA寄存中，并在适当时插入Phi节点。例如，如果通过传递运行此示例，则将获得：

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

mem2reg pass实现了用于构造SSA表单的标准“迭代支配边界”算法，并具有许多优化措施，可加（非常常见）简并案例的速度。mem2reg优化pass是处理可变变量的答案，我们强烈建议使用它。注意，mem2reg仅在某些情况下适用于变量：

1. mem2reg是由alloca驱动的：它查找alloca，并且如果可以处理它们，则将其升级。它不适用于全变量或堆分配。
2. mem2reg仅在函数的入口块中查找alloca指令。在入口块中保证了alloca仅执行一次，这使分析更简单。

3. mem2reg仅促进使用直接负载和存储的alloca。如果将堆栈对象的地址传递给函数，或者涉及任有趣的指针算法，则不会提升alloca。

4. mem2reg仅适用于allocas的 第一类值（如指针，标量和向量），且仅当所述分配的阵列大小为（或者在.ll文件丢失）。mem2reg无法将结构或数组提升为寄存器。请注意，sroa 传递更强大，在多情况下可以提升结构，“联合”和数组。

所有这些属性对于大多数命令式语言来说都是很容易满足的，我们将在下面用Kaleidoscope对其进行明。你可能要问的最后一个问题是：我是否应该在我的前端中忽略这些属性？如果我直接进行SSA构，避免使用mem2reg优化通道，那会更好吗？简而言之，我们强烈建议使用此技术来构建SSA格式除非有非常充分的理由不这样做。使用这种技术是：

- 经过验证且经过良好测试：clang使用此技术处理局部可变变量。因此，LLVM的最常见客户端正在用它来处理大量变量。你可以确定可以早日发现并修复错误。
- 极快：mem2reg具有许多特殊情况，可以使它在普通情况下以及在一般情况下都快速。例如，它有仅用于单个块中的变量的快速路径，仅具有一个分配点的变量，避免插入不需要的phi节点的良好发法等。
- 生成 调试所需信息: LLVM中的调试信息依赖于公开变量的地址，以便可以将调试信息附加到该变。这种技术与这种调试信息风格非常自然地吻合。

如果没有其他问题，这将使我们的前端启动和运行变得更加容易，并且实现起来非常简单。现在让我在Kaleidoscope中扩展可变变量！

## Kaleidoscope中的可变变量

现在我们知道了要解决的问题，让我们在Kaleidoscope这种小语言的背景下看一下这是什么样子。我将添加两个功能：

1. 使用 = 运算符对变量进行变异的能力。
2. 定义新变量的能力。

虽然第一项实际上就是它的意义，但是我们只有传入参数和归纳变量的变量，并且重新定义它们的范很广。同样，定义新变量的功能是有用的，无论你是否要对其进行突变。下面是一个有动机性的示例展示了我们如何使用它们：

```
# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

# Recursive fib, we could do this before.
def fib(x)
  if (x < 3) then
    1
  else
    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :
     a = b :
```

```
    b = c) :
  b;

# Call it.
fibi(10);
```

为了使变量变异，我们必须更改现有变量以使用“分配技巧”。一旦有了，我们将添加新的运算符，后扩展Kaleidoscope以支持新的变量定义。

## 为可变特性调整已存在的变量

Kaleidoscope中的符号表在代码生成时通过 NamedValues 映射进行管理。该映射当前跟踪LLVM Value* ，该值持有命名变量的双精度值。为了支持突变，我们需要稍作更改，以便 NamedValues 保留讨论变量的 存储位置 。请注意，此更改是重构：它更改了代码的结构，但是（本身）不更改编译器行为。所有这些更改都在Kaleidoscope代码生成器中隔离。

在Kaleidoscope的开发阶段，它仅支持变量干两件事情：函数的传入参数和 for循环的归纳变量。为保持一致性，除了其他用户定义的变量外，我们还允许对这些变量进行突变。这意味着它们都需要存位置。

要开始Kaleidoscope的转换，我们将更改NamedValues映射，以使其映射到AllocaInst * 而不是 Value *。完成此操作后，C ++编译器将告诉我们我们需要更新代码的哪些部分：

```
static std::map<std::string, AllocaInst*> NamedValues;
```

同样，由于我们将需要创建这些alloca，因此我们将使用一个辅助函数，以确保在该函数的入口块中建alloca：

```
/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(TheContext), 0,
                           VarName.c_str());
}
```

这个看起来很有趣的代码创建了一个IRBuilder对象，该对象指向入口块的第一条指令（.begin()）。后，它使用预期名称创建一个alloca并将其返回。因为Kaleidoscope中的所有值都是双精度的，所以需要传递类型来使用。

有了这个，我们要进行的第一个功能更改就是变量引用。在我们的新方案中，变量存在于堆栈中，因生成对其引用的代码实际上需要从堆栈插槽中产生负载：

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    return LogErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```

如上所示，这非常简单。现在，我们需要更新定义变量的内容以设置alloca。我们将从开始 ForExprAST::codegen()（请参阅完整的代码清单以获取未删节的代码）：

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create an alloca for the variable in the entry block.
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

// Emit the start code first, without 'variable' in scope.
Value *StartVal = Start->codegen();
if (!StartVal)
  return nullptr;

// Store the value into the alloca.
Builder.CreateStore(StartVal, Alloca);
...

// Compute the end condition.
Value *EndCond = End->codegen();
if (!EndCond)
  return nullptr;

// Reload, increment, and restore the alloca.  This handles the case where
// the body of the loop mutates the variable.
Value *CurVar = Builder.CreateLoad(Alloca);
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);
...
```

该代码实际上与允许可变变量之前的代码相同。最大的区别是我们不再需要构造PHI节点，并且根据要使用加载/存储来访问变量。

为了支持可变的参数变量，我们还需要为其做分配。此代码也非常简单：

```
Function *FunctionAST::codegen() {
  ...
  Builder.SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args()) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

    // Store the initial value into the alloca.
    Builder.CreateStore(&Arg, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Arg.getName()] = Alloca;
  }

  if (Value *RetVal = Body->codegen()) {
    ...
```

对于每个参数，我们创建一个alloca，将函数的输入值存储到alloca中，然后将alloca注册为该参数的

储位置。FunctionAST::codegen() 在为函数设置入口块之后，将立即调用此方法。

最后缺少的部分是添加mem2reg传递，这使我们能够再次获得良好的代码生成：

```
// Promote allocas to registers.
TheFPM->add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
TheFPM->add(createInstructionCombiningPass());
// Reassociate expressions.
TheFPM->add(createReassociatePass());
...
```

有趣的是，在运行mem2reg优化之前和之后，代码看起来是什么样的。例如，这是递归fib函数的befre / after代码。优化前：

```
define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double, double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  br label %ifcont

else:       ; preds = %entry
  %x3 = load double, double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double, double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

这里只有一个变量（x, 输入参数），但是你仍然可以看到我们正在使用的极为简单的代码生成策略。输入块中，创建一个alloca，并将初始输入值存储到其中。每个对该变量的引用都会从堆栈中重新加。另外，请注意，我们并未修改if / then / else表达式，因此它仍会插入PHI节点。虽然我们可以为其建alloca，但实际上为它创建PHI节点更容易，因此我们仍然仅制作PHI。

这是mem2reg传递运行后的代码：

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
```

```
  br i1 %ifcond, label %then, label %else

then:
  br label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

对于mem2reg而言，这是一个复杂的情况，因为没有对该变量进行重新定义。显示这一点的目的是免对插入的低效率。

在其余的优化器运行之后，我们得到：

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp ueq double %booltmp, 0.000000e+00
  br i1 %ifcond, label %else, label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  ret double %addtmp

ifcont:
  ret double 1.000000e+00
}
```

在这里，我们看到了simplecfg传递决定将返回指令克隆到 else 块的末尾。这样就可以消除一些分支PHI节点。

现在，所有符号表引用都已更新为使用堆栈变量，我们将添加赋值运算符。

## 新的赋值运算符

在我们当前的框架下，添加新的赋值运算符非常简单。我们将像解析任何其他二元运算符一样解析它但是在内部对其进行处理（而不是允许用户定义它）。第一步是设置优先级：

```
int main() {
  // Install standard binary operators.
```

```
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

既然解析器知道二元运算符的优先级，那么它将处理所有解析和AST生成。我们只需要为赋值运算符现codegen。看起来像：

```
Value *BinaryExprAST::codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    // Assignment requires the LHS to be an identifier.
    VariableExprAST *LHSE = dynamic_cast<VariableExprAST*>(LHS.get());
    if (!LHSE)
      return LogErrorV("destination of '=' must be a variable");
```

与其余的二元运算符不同，我们的赋值运算符不遵循“生成LHS，生成RHS，进行计算”模型。这样在处理其他二元运算符之前，将其作为特殊情况进行处理。另一个奇怪的是，它要求LHS是一个变量具有 `(x + 1) = expr` 是无效的: 仅允许使用类似于 `x = expr` 的表达式。

```
  // Codegen the RHS.
  Value *Val = RHS->codegen();
  if (!Val)
    return nullptr;

  // Look up the name.
  Value *Variable = NamedValues[LHSE->getName()];
  if (!Variable)
    return LogErrorV("Unknown variable name");

  Builder.CreateStore(Val, Variable);
  return Val;
}
...
```

有了变量后，对分配进行代码生成就很简单了：我们生成分配的RHS，创建一个存储，然后返回计算的值。返回一个值可进行链式分配，例如 `X = (Y = Z)` 。

既然有了赋值运算符，就可以对循环变量和参数进行突变。例如，我们现在可以运行如下代码：

```
# Function to print a double.
extern printd(x);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

def test(x)
  printd(x) :
  x = 4 :
  printd(x);

test(123);
```

运行时，此示例先打印 123 ，然后打印 4 ，表明我们确实对值进行了突变！好的，我们现在已经正实现了我们的目标：要使其正常工作，通常需要进行SSA构建。但是，要真正有用，我们希望能够定我们自己的局部变量，接下来我们将添加它！

## 用户定义的局部变量

添加var / in就像我们对万花筒所做的任何其他扩展一样：我们扩展了词法分析器，解析器，AST和代生成器。添加我们的新“ var / in”构造的第一步是扩展词法分析器。和以前一样，这很简单，代码下：

```
enum Token {
  ...
  // var definition
  tok_var = -13
...
}
...
static int gettok() {
...
    if (IdentifierStr == "in")
      return tok_in;
    if (IdentifierStr == "binary")
      return tok_binary;
    if (IdentifierStr == "unary")
      return tok_unary;
    if (IdentifierStr == "var")
      return tok_var;
    return tok_identifier;
...
```

下一步是定义我们将构建的AST节点。对于var / in，它看起来像这样：

```
/// VarExprAST - Expression class for var/in
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
  std::unique_ptr<ExprAST> Body;

public:
  VarExprAST(std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames,
             std::unique_ptr<ExprAST> Body)
    : VarNames(std::move(VarNames)), Body(std::move(Body)) {}

  Value *codegen() override;
};
```

var / in允许一次定义一个名称列表，并且每个名称都可以有一个初始化值。因此，我们在VarNames量中捕获了此信息。另外，var / in有一个主体，允许该主体访问var / in定义的变量。

有了这个，我们可以定义解析器片段。我们要做的第一件事是将其添加为主表达式：

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
```

```
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
  case '(':
    return ParseParenExpr();
  case tok_if:
    return ParseIfExpr();
  case tok_for:
    return ParseForExpr();
  case tok_var:
    return ParseVarExpr();
  }
}
```

接下来，我们定义ParseVarExpr：

```
/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static std::unique_ptr<ExprAST> ParseVarExpr() {
  getNextToken();  // eat the var.

  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)
    return LogError("expected identifier after var");
```

该代码的第一部分将标识符/表达式对的列表解析为局部 VarNames 向量。

```
while (1) {
  std::string Name = IdentifierStr;
  getNextToken();  // eat identifier.

  // Read the optional initializer.
  std::unique_ptr<ExprAST> Init;
  if (CurTok == '=') {
    getNextToken(); // eat the '='.

    Init = ParseExpression();
    if (!Init) return nullptr;
  }

  VarNames.push_back(std::make_pair(Name, std::move(Init)));

  // End of var list, exit loop.
  if (CurTok != ',') break;
```

```
  getNextToken(); // eat the ','.

  if (CurTok != tok_identifier)
    return LogError("expected identifier list after var");
}
```

解析完所有变量后，我们将解析正文并创建AST节点：

```
  // At this point, we have to have 'in'.
  if (CurTok != tok_in)
    return LogError("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  auto Body = ParseExpression();
  if (!Body)
    return nullptr;

  return std::make_unique<VarExprAST>(std::move(VarNames),
                                      std::move(Body));
}
```

现在我们可以解析并表示代码，我们需要为其支持LLVM IR的生成。该代码以以下内容开头：

```
Value *VarExprAST::codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second.get();
```

基本上，它遍历所有变量，一次安装一个。对于我们放入符号表中的每个变量，我们都记得在OldBindings中替换的先前值。

```
  // Emit the initializer before adding the variable to scope, this prevents
  // the initializer from referencing the variable itself, and permits stuff
  // like this:
  //  var a = 1 in
  //    var a = a in ...   # refers to outer 'a'.
  Value *InitVal;
  if (Init) {
    InitVal = Init->codegen();
    if (!InitVal)
      return nullptr;
  } else { // If not specified, use 0.0.
    InitVal = ConstantFP::get(TheContext, APFloat(0.0));
  }

  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
  Builder.CreateStore(InitVal, Alloca);

  // Remember the old variable binding so that we can restore the binding when
  // we unrecurse.
```

```
  OldBindings.push_back(NamedValues[VarName]);

  // Remember this binding.
  NamedValues[VarName] = Alloca;
}
```

这里的注释比代码更多。基本思想是，我们发出初始化程序，创建alloca，然后更新符号表以指向它将所有变量安装在符号表中之后，我们将运行var / in表达式的主体：

```
// Codegen the body, now that all vars are in scope.
Value *BodyVal = Body->codegen();
if (!BodyVal)
  return nullptr;
```

最后，在返回之前，我们恢复之前的变量绑定：

```
  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}
```

所有这些的最终结果是，我们获得了适当范围内的变量定义，并且甚至（普通地）允许对它们进行突。

这样，我们完成了我们要做的工作。我们介绍中不错的迭代fib示例可以编译并正常运行。mem2reg递将所有堆栈变量优化到SSA寄存器中，在需要时插入PHI节点，并且前端保持简单：在任何地方都不到“迭代支配边界”计算。

## 完整代码清单

这是我们正在运行的示例的完整代码清单，增加了可变变量和var / in支持。要运行并演示此示例，请用以下命令：

```
# Compile
clang++ -g chapter7-Extending-the-Language-Mutable-Variables.cpp `llvm-config --cxxflags -ldflags --system-libs --libs all mcjit native` -O3 -o toy
# Run
./toy
```

以下是完整代码清单：

chapter7-Extending-the-Language-Mutable-Variables.cpp

---

参考: Kaleidoscope: Extending the Language: Mutable Variables