链滴

Kaleidoscope 系列第六章：扩展语言—用户自定义运算符

作者：Hanseltu

原文链接：https://ld246.com/article/1570028565306

来源网站：链滴

原文链接:Kaleidoscope系列第六章：扩展语言—用户自定义运算符

本文是使用LLVM开发新语言Kaleidoscope教程系列第六章，继续扩展Kaleidoscope语言特性，增用户自定义运算符处理。

第六章简介

欢迎来到“使用LLVM开发新语言Kaleidoscope教程”教程的第六章。至此，在本教程中，我们现在了一种功能齐全的语言，该语言相当少，但也很有用。但是，仍然存在一个大问题。我们的语言没有多有用的运算符（例如除法，逻辑求反，甚至除小于以外的任何比较运算符）。

本教程的这一章将重点放在将用户定义的运算符添加到简单有效的Kaleidoscope语言中。现在，这种离在某种程度上给了我们一种简单而丑陋的语言，但同时也给了我们一种强大的语言。创建自己的语的好处之一就是我们可以决定什么是好是坏。在本教程中，我们假设可以将其用作显示一些有趣的解技术的方法。

在本教程的最后，我们将介绍一个 呈现Mandelbrot集 的示例Kaleidoscope应用程序。这给出了可使用Kaleidoscope及其功能集构建的示例。

用户自定义运算符: 思路

我们将添加到万花筒中的“运算符重载”比诸如C ++之类的语言更通用。在C ++中，只允许重新定现有的运算符：不能以编程方式更改语法，引入新的运算符，更改优先级等。在本章中，我们将向Kaledoscope中添加此功能，从而使用户对支持的运算符集更加满意。

在这样的教程中进入用户定义的运算符的目的是展示使用手写解析器的功能和灵活性。到目前为止，们一直在执行的解析器对大部分语法使用递归下降，对表达式使用运算符优先级解析。有关详细信息请参见第二章。通过使用运算符优先级解析，很容易允许程序员将新的运算符引入语法中：语法在JIT行时可以动态扩展。

我们将添加的两个特定功能是可编程的一元运算符（现在，Kaleidoscope中根本没有一元运算符）以更多二元运算符。例如：

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;
```

```
# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

许多语言都希望能够用语言本身来实现其标准运行时库。在Kaleidoscope中，我们可以在库中实现语的重要部分！

我们将这些功能的实现分为两部分：实现对用户定义的二元运算符的支持以及添加一元运算符。

用户自定义二元运算符

在我们当前的框架中，添加对用户定义的二元运算符的支持非常简单。我们将首先添加对一元/二元键字的支持：

```
enum Token {
  ...
  // operators
  tok_binary = -11,
  tok_unary = -12
};
...
static int gettok() {
...
    if (IdentifierStr == "for")
      return tok_for;
    if (IdentifierStr == "in")
      return tok_in;
    if (IdentifierStr == "binary")
      return tok_binary;
    if (IdentifierStr == "unary")
      return tok_unary;
    return tok_identifier;
```

就像我们在前几章中所做的那样，这仅增加了对一元和二元关键字的lexer支持。关于当前AST的一件事是，我们通过使用其ASCII码作为操作码来表示二元运算符并进行全面概括。对于我们的扩展运算，我们将使用相同的表示形式，因此我们不需要任何新的AST或解析器支持。

另一方面，我们必须能够在函数定义的“ def binary | 5”中表示这些新运算符的定义。到目前为止在我们的语法中，函数定义的“name”被解析为“prototype”生成并进入 PrototypeAST AST节。为了将新的用户定义的运算符表示为原型，我们必须 PrototypeAST 像这样扩展AST节点：

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its argument names as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool IsOperator;
  unsigned Precedence;  // Precedence if a binary op.

public:
  PrototypeAST(const std::string &name, std::vector<std::string> Args,
               bool IsOperator = false, unsigned Prec = 0)
  : Name(name), Args(std::move(Args)), IsOperator(IsOperator),
    Precedence(Prec) {}
```

```
  Function *codegen();
  const std::string &getName() const { return Name; }

  bool isUnaryOp() const { return IsOperator && Args.size() == 1; }
  bool isBinaryOp() const { return IsOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size() - 1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }
};
```

基本上，除了知道原型的名称之外，我们现在还要跟踪它是否是一个运算符，如果是，则跟踪该运算的优先级。优先级仅用于二元运算符（如下所示，它不适用于一元运算符）。现在我们有了一种表示户定义的运算符原型的方法，我们需要解析它：

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return LogErrorP("Invalid precedence: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }
```

```
  if (CurTok != '(')
    return LogErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return LogErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return LogErrorP("Invalid number of operands for operator");

  return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames), Kind != 0,
                                        BinaryPrecedence);
}
```

这都是非常简单的解析代码，并且过去我们已经看到很多类似的代码。关于上面的代码，一个有趣的分是FnName为二进制运算符设置的几行代码。这将为新定义的“@”运算符生成“binary @”之的名称。然后利用以下事实：允许LLVM符号表中的符号名称中包含任何字符，包括嵌入的nul字符。

要添加的下一个有趣的事情是对这些二元运算符的代码生成支持。鉴于我们当前的结构，这是对现有元运算符节点的默认情况的简单添加：

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder.CreateFSub(L, R, "subtmp");
  case '*':
    return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext),
                                "booltmp");
  default:
    break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = getFunction(std::string("binary") + Op);
  assert(F && "binary operator not found!");
```

```
  Value *Ops[2] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}
```

如在上面看到的，新代码实际上非常简单。它只是在符号表中查找适当的运算符并生成对其的函数调。由于用户定义的运算符只是作为常规函数构建的（因为“原型”可以归结为名称正确的函数），因所有内容都可以使用。

我们缺少的最后一段代码是一些重要技巧：

```
Function *FunctionAST::codegen() {
  // Transfer ownership of the prototype to the FunctionProtos map, but keep a
  // reference to it for use below.
  auto &P = *Proto;
  FunctionProtos[Proto->getName()] = std::move(Proto);
  Function *TheFunction = getFunction(P.getName());
  if (!TheFunction)
    return nullptr;

  // If this is an operator, install it.
  if (P.isBinaryOp())
    BinopPrecedence[P.getOperatorName()] = P.getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
  ...
```

基本上，在对函数进行代码生成之前，如果它是用户定义的运算符，则将其注册在优先级表中。这使我们已经可以使用二元运算符解析逻辑来处理它。由于我们正在开发一种通用的运算符优先级解析器因此这是“扩展语法”所要做的全部。

现在，我们有了有用的用户定义的二元运算符。这是在我们为其他运算符构建的先前框架的基础上构的。添加一元运算符更具挑战性，因为我们还没有任何框架，让我们看看它需要什么。

用户自定义一元运算符

由于我们目前不支持Kaleidoscope语言的一元运算符，因此我们需要添加所有内容以支持它们。上面我们向词法分析器添加了对“一元”关键字的简单支持。除此之外，我们还需要一个AST节点：

```
/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
  char Opcode;
  std::unique_ptr<ExprAST> Operand;

public:
  UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
    : Opcode(Opcode), Operand(std::move(Operand)) {}

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

到目前为止，此AST节点非常简单明显。它直接改进二元运算符AST节点，即有一个孩子节点。这样我们需要添加解析逻辑。解析一元运算符非常简单：我们将添加一个新函数来做到这一点：

```
/// unary
///   ::= primary
///   ::= '!' unary
static std::unique_ptr<ExprAST> ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (auto Operand = ParseUnary())
    return std::make_unique<UnaryExprAST>(Opc, std::move(Operand));
  return nullptr;
}
```

我们添加的语法在这里非常简单。如果在解析主运算符时看到一元运算符，则将该运算符作为前缀使，而将其余部分作为另一个一元运算符进行解析。这使我们可以处理多个一元运算符（例如“ !! x”。注意，一元运算符不能像二元运算符那样具有模棱两可的解析，因此不需要优先级信息。

这个函数的问题是，我们需要从某个地方调用ParseUnary。为此，我们将以前的ParsePrimary调用更改为调用ParseUnary：

```
/// binoprhs
///   ::= ('+' unary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  ...
    // Parse the unary expression after the binary operator.
    auto RHS = ParseUnary();
    if (!RHS)
      return nullptr;
  ...
}
/// expression
///   ::= unary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParseUnary();
  if (!LHS)
    return nullptr;

  return ParseBinOpRHS(0, std::move(LHS));
}
```

通过这两个简单的修改，我们现在可以解析一元运算符并为其构建AST。接下来，我们需要为原型添解析器支持，以解析一元运算符原型。我们将上面的二进制运算符代码扩展为：

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName;
```

```
  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    ...
```

与二元运算符一样，我们用包含运算符的名称来命名一元运算符。这在代码生成时为我们提供了帮助
说到最后，我们需要添加的内容是对一元运算符的代码生成支持。看起来像这样：

```
Value *UnaryExprAST::codegen() {
  Value *OperandV = Operand->codegen();
  if (!OperandV)
    return nullptr;

  Function *F = getFunction(std::string("unary") + Opcode);
  if (!F)
    return LogErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}
```

该代码类似于但比二元运算符的代码简单。它之所以简单，主要是因为它不需要处理任何预定义的运
符。

测试

很难让人相信，但是通过上几章已经介绍的一些简单扩展，我们已经发展出一种真实的语言。这样，
们可以做很多有趣的事情，包括I / O，数学和许多其他事情。例如，我们现在可以添加一个不错的排
运算符（已定义printd以打印出指定的值和换行符）：

```
ready> extern printd(x);
Read extern:
declare double @printd(double)

ready> def binary : 1 (x y) 0;  # Low-precedence operator that ignores operands.
...
ready> printd(123) : printd(456) : printd(789);
```

```
123.000000
456.000000
789.000000
Evaluated to 0.000000
```

我们还可以定义许多其他“原始”操作，例如：

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Unary negate.
def unary-(v)
  0-v;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary logical or, which does not short circuit.
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Binary logical and, which does not short circuit.
def binary& 6 (LHS RHS)
  if !LHS then
    0
  else
    !!RHS;

# Define = with slightly lower precedence than relationals.
def binary = 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);

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

有了先前的if / then / else支持，我们还可以为I / O定义有趣的功能。例如，以下命令打印出一个字，其“密度”反映了传入的值：值越低，字符越密集：

```
ready> extern putchard(char);
...
ready> def printdensity(d)
  if d > 8 then
```

```
    putchard(32)  # ' '
  else if d > 4 then
    putchard(46)  # '.'
  else if d > 2 then
    putchard(43)  # '+'
  else
    putchard(42); # '*'
...
ready> printdensity(1): printdensity(2): printdensity(3):
       printdensity(4): printdensity(5): printdensity(9):
       putchard(10);
**++.
Evaluated to 0.000000
```

基于这些简单的原始操作，我们可以开始定义更多有趣的事情。例如，以下是一个小函数，它确定复平面中某个函数发散所需的迭代次数：

```
# Determine whether the specific location diverges.
# Solve for z = z^2 + c in the complex plane.
def mandelconverger(real imag iters creal cimag)
  if iters > 255 | (real*real + imag*imag > 4) then
    iters
  else
    mandelconverger(real*real - imag*imag + creal,
                    2*real*imag + cimag,
                    iters+1, creal, cimag);

# Return the number of iterations required for the iteration to escape
def mandelconverge(real imag)
  mandelconverger(real, imag, 0, real, imag);
```

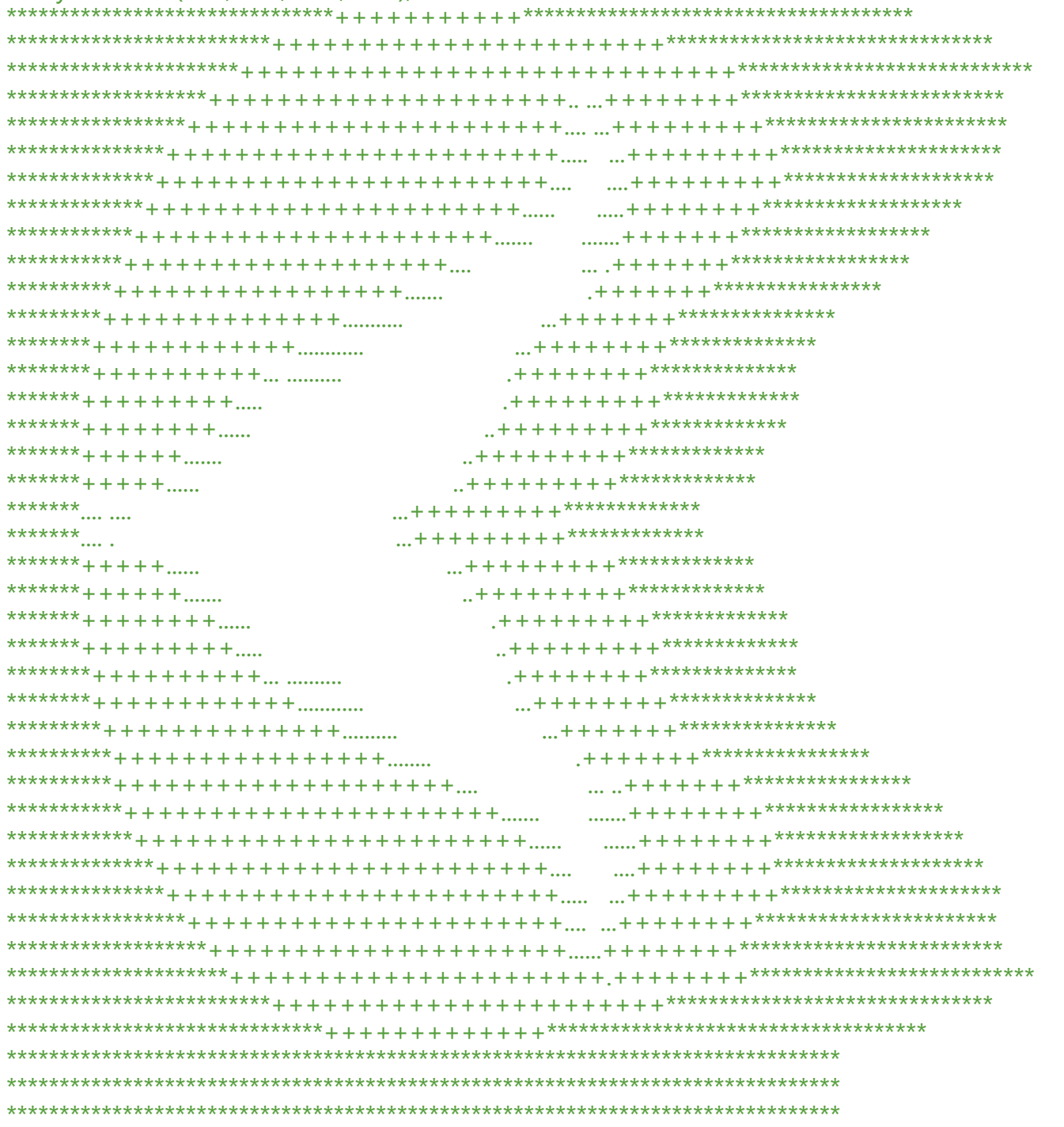
这个 z=z^2+c 函数是一个美丽的小动物，是计算曼德布罗特集的基础。我们的函数返回逃脱复杂轨所需的迭代次数，饱和后达到255。这本身并不是一个非常有用的函数，但是如果将其值绘制在二维面上，则可以看到Mandelbrot集。鉴于我们仅限于在此处使用putchard，我们令人惊叹的图形输出受到限制，但是我们可以使用上面的密度图将一些东西混合在一起：(http://en.wikipedia.org/wiki/andelbrot_set)mandelconverge

```
# Compute and plot the mandelbrot set with the specified 2 dimensional range
# info.
def mandelhelp(xmin xmax xstep   ymin ymax ystep)
  for y = ymin, y < ymax, ystep in (
    (for x = xmin, x < xmax, xstep in
       printdensity(mandelconverge(x,y)))
    : putchard(10)
  )

# mandel - This is a convenient helper function for plotting the mandelbrot set
# from the specified position with the specified Magnification.
def mandel(realstart imagstart realmag imagmag)
  mandelhelp(realstart, realstart+realmag*78, realmag,
             imagstart, imagstart+imagmag*40, imagmag);
```

鉴于此，我们可以尝试绘制mandelbrot集！让我们尝试一下：

```
ready> mandel(-2.3, -1.3, 0.05, 0.07);
```



```
Evaluated to 0.000000
ready> mandel(-2, -1, 0.02, 0.04);
```

```
***************+++++++++++++++++++++++++++++++++++++++++++++++++++++
++++........
**************+++++++++++++++++++++++++++++++++++++++++++++++++++++
++...........
************+++++++++++++++++++++++++++++++++++++++++++++++++++++++.
............
***********+++++++++++++++++++++++++++++++++++++++++++++++++++........
     .
**********+++++++++++++++++++++++++++++++++++++++++++++++.............
********++++++++++++++++++++++++++++++++++++++++++++..................
*******++++++++++++++++++++++++++++++++++++++++.......................
******++++++++++++++++++++++++++++++++++++............................
*****+++++++++++++++++++++++++++++++++..............................
*****+++++++++++++++++++++++++++++.................................
****+++++++++++++++++++++++++++......    ...........................
***+++++++++++++++++++++++++..........       .......      ............
***+++++++++++++++++++++++.............
**++++++++++++++++++++++...............
**++++++++++++++++++++.................
*++++++++++++++++++..................
*++++++++++++++++............. ...
*++++++++++++++...............
*+++....++++.................
*...........   ............
*
*...........   ............
*+++....++++.................
*++++++++++++++...............
*++++++++++++++++............. ...
*++++++++++++++++++..................
**++++++++++++++++++++.................
**++++++++++++++++++++++...............
***+++++++++++++++++++++++.............
***+++++++++++++++++++++++++..........       .......      ............
****+++++++++++++++++++++++++++......    ...........................
*****+++++++++++++++++++++++++++++.................................
*****+++++++++++++++++++++++++++++++++..............................
******++++++++++++++++++++++++++++++++++++............................
*******++++++++++++++++++++++++++++++++++++++++.......................
********++++++++++++++++++++++++++++++++++++++++++++..................
Evaluated to 0.000000
ready> mandel(-0.9, -1.4, 0.02, 0.03);
******************************************************************************
******************************************************************************
******************************************************************************
**********++++++++++++++++++++++***********************************************
*++++++++++++++++++++++++++++++++++++++++**********************************
****
++++++++++++++++++++++++++++++++++++++++++++++*************************
********
+++++++++++++++++++++++++++++++++++++++++++++++++++*****************
***********
+++++++++++++++++++++++++++++++++++++++++++++++++++++++**********
**************
```

```
Evaluated to 0.000000
ready> ^D
```

此时，你可能已经开始意识到Kaleidoscope是一种真实而强大的语言。它可能不是自相似的，但是可用来绘制实际的东西！

这样，我们结束了本教程的“添加用户定义运算符”一章。我们已经成功地扩充了语言，增加了在库扩展语言的能力，并且我们展示了如何在Kaleidoscope中使用它来构建简单但有趣的最终用户应用程。在这一点上，Kaleidoscope可以构建具有功能的各种应用程序，并且可以调用具有副作用的函数，是它本身无法真正定义和变异变量。

引人注目的是，变量突变是某些语言的重要功能，而且在不向前端添加“ SSA构造”阶段的情况下，何添加对可变变量的支持一点也不明显。在下一章中，我们将描述如何在不构建前端SSA的情况下添

变量突变。

完整代码清单

这是我们正在运行的示例的完整代码清单，增加对用户定义的运算符的支持。运行构建此示例，请使以下命令：

```
# Compile
clang++ -g chapter6-Extending-the-language-User-defined-Operators.cpp `llvm-config --cxxflags --ldflags --system-libs --libs all mcjit native` -O3 -o toy
# Run
./toy
```

注意: 在某些平台上，链接时需要指定-rdynamic或-Wl，–export-dynamic。这样可确保将主可执行件中定义的符号导出到动态链接器，以便在运行时可用于符号解析。如果将支持代码编译到共享库中则不需要这样做，尽管这样做会在Windows上引起问题。

以下是完整代码清单：

chapter6-Extending-the-language-User-defined-Operators.cpp

参考: Kaleidoscope: Extending the Language: User-defined Operators