目 录

[1. 万花筒：万花筒简介和词法分析器 1](#_Toc1173806754)

[1.1. 万花筒语言 1](#_Toc613622181)

[1.2.  词法分析器lexer 1](#_Toc2081593665)

[2. 万花筒：实现解析器和 AST  3](#_Toc895498557)

[2.1. 简介 3](#_Toc487035717)

[2.2.  抽象语法树（AST） 3](#_Toc589745397)

[2.3.  解析器基础 5](#_Toc2014020521)

[2.4.  基本表达式解析 6](#_Toc1013043054)

[2.5.   Binary Expression Parsing二进制表达式解析 8](#_Toc1034230089)

[2.6.  解析其余部分 10](#_Toc1842582833)

[2.7.  driver驱动 11](#_Toc1434050920)

[2.8.  结论 12](#_Toc388704428)

[2.9.  运行 12](#_Toc1238626648)

[3. 万花筒：代码生成到 LLVM IR 13](#_Toc1230994500)

[3.1. 简介 13](#_Toc1165322436)

[3.2. 代码生成设置 13](#_Toc1337513573)

[3.3. 表达式代码生成 14](#_Toc2051296021)

[3.4.  函数代码生成 16](#_Toc1307085070)

[3.5. Driver Changes and Closing Thoughts 18](#_Toc546324438)

[3.6. 代码运行 20](#_Toc565792900)

[4. 万花筒：添加 JIT 和优化器支持 20](#_Toc102480885)

[4.1. 简介 20](#_Toc2034232337)

[4.2. 平凡的常量折叠 20](#_Toc950750492)

[4.3. LLVM 优化pass 21](#_Toc454070224)

[4.4. 添加 JIT 编译器 22](#_Toc2090467276)

[4.5. 运行 29](#_Toc2072709607)

[5. 附录 29](#_Toc1823947796)

# 万花筒：万花筒简介和词法分析器

https://llvm.org/docs/tutorial/index.html

## 万花筒语言

本教程使用一种名为“Kaleidoscope ”（源自“美丽、形式和视图”的意思）的玩具语言进行说明。Kaleidoscope 是一种过程语言，允许您定义函数、使用条件、数学等。在本教程的过程中，我们将扩展 Kaleidoscope 以支持 if/then/else 结构、for 循环、用户定义的运算符、JIT使用简单的命令行界面、调试信息等进行编译。

我们希望让事情保持简单，因此 Kaleidoscope 中唯一的数据类型是 64 位浮点类型（在 C 语言中也称为“double”）。因此，所有值都是隐式双精度，并且该语言不需要类型声明。这为该语言提供了非常漂亮和简单的语法。例如，以下简单示例计算 斐波那契数：

|  |
| --- |
| # Compute the x'th fibonacci number.  def fib(x)  if x < 3 then  1  else  fib(x-1)+fib(x-2)  # This expression will compute the 40th number.  fib(40) |

我们还允许 Kaleidoscope 调用标准库函数——LLVM JIT 使这变得非常容易。这意味着您可以在使用函数之前使用“extern”关键字来定义它（这对于相互递归函数也很有用）。例如：

|  |
| --- |
| extern sin(arg);  extern cos(arg);  extern atan2(arg1 arg2);  atan2(sin(.4), cos(42)) |

一个更有趣的例子包含在第 6 章中，我们在其中编写了一个小的 Kaleidoscope 应用程序，它以不同的放大级别[显示 Mandelbrot 集](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl06.html" \l "kicking-the-tires)。

让我们深入研究这种语言的实现！

## 词法分析器lexer

在实现一种语言时，首先需要的是能够处理文本文件并识别它所说的内容。执行此操作的传统方法是使用“词法分析器([lexer](http://en.wikipedia.org/wiki/Lexical_analysis))”（又名“scanner”）将输入分解为“标记(token)”。词法分析器返回的每个标记都包含一个标记代码和一些潜在的元数据（例如，数字的数值）。首先，我们定义可能性：

|  |
| --- |
| // The lexer returns tokens [0-255] if it is an unknown character, otherwise one  // of these for known things.  enum Token {  tok\_eof = -1,  // commands  tok\_def = -2,  tok\_extern = -3,  // primary  tok\_identifier = -4,  tok\_number = -5,  };  static std::string IdentifierStr; // Filled in if tok\_identifier  static double NumVal; // Filled in if tok\_number |

我们的词法分析器返回的每个标记要么是 Token 枚举值之一，要么是一个“未知”字符，如“+”，作为其 ASCII 值返回。如果当前标记是标识符，则 全局变量IdentifierStr保存标识符的名称。如果当前标记是数字文字（如 1.0），则NumVal保留其值。为简单起见，我们使用全局变量，但这不是真正语言实现的最佳选择:)。

词法分析器的实际实现是一个名为 gettok的简单函数。调用该gettok函数以从标准输入返回下一个标记。它的定义开始于：

|  |
| --- |
| /// gettok - Return the next token from standard input.  static int gettok() {  static int LastChar = ' ';  // Skip any whitespace.  while (isspace(LastChar))  LastChar = getchar(); |

gettok通过调用 C的getchar()函数从标准输入一次读取一个字符来工作。它在识别它们时吃掉它们，并将最后读取但未处理的字符存储在 LastChar 中。它必须做的第一件事是忽略标记之间的空白。这是通过上面的循环完成的。

接下来gettok需要做的是识别标识符和特定关键字，如“def”。Kaleidoscope 使用这个简单的循环来做到这一点：

|  |
| --- |
| if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]\*  IdentifierStr = LastChar;  while (isalnum((LastChar = getchar())))  IdentifierStr += LastChar;  if (IdentifierStr == "def")  return tok\_def;  if (IdentifierStr == "extern")  return tok\_extern;  return tok\_identifier;  } |

请注意，此代码会在对标识符进行词法分析时将“IdentifierStr”设置为全局。此外，由于语言关键字由同一循环匹配，因此我们在这里以内联方式处理它们。数值相似：

|  |
| --- |
| if (isdigit(LastChar) || LastChar == '.') { // Number: [0-9.]+  std::string NumStr;  do {  NumStr += LastChar;  LastChar = getchar();  } while (isdigit(LastChar) || LastChar == '.');  NumVal = strtod(NumStr.c\_str(), 0);  return tok\_number;  } |

这些都是用于处理输入的非常简单的代码。从输入中读取数值时，我们使用 C语言的strtod函数将其转换为我们存储在中的数值NumVal。请注意，这并没有进行足够的错误检查：它会错误地读取“1.23.45.67”并像您输入“1.23”一样处理它。随意扩展它！接下来我们处理评论：

|  |
| --- |
| if (LastChar == '#') {  // Comment until end of line.  do  LastChar = getchar();  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');  if (LastChar != EOF)  return gettok();  } |

我们通过跳到行尾来处理评论，然后返回下一个标记。最后，如果输入与上述情况之一不匹配，则它要么是像“+”这样的运算符字符，要么是文件末尾。这些是用这段代码处理的：

|  |
| --- |
| // Check for end of file. Don't eat the EOF.  if (LastChar == EOF)  return tok\_eof;  // Otherwise, just return the character as its ascii value.  int ThisChar = LastChar;  LastChar = getchar();  return ThisChar;  } |

这样，我们就有了基本 Kaleidoscope 语言的完整词法分析器（词法分析器的完整代码清单可在本教程的下一章中找到）。接下来我们将构建一个简单的解析器，使用它来构建抽象语法树。当我们拥有它时，我们将包含一个驱动程序，以便您可以一起使用词法分析器和解析器。

Next: [实现解析器和 AST](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html)

# 万花筒：实现解析器和 AST

## 简介

欢迎阅读“使用 LLVM 实现语言”教程的第 2 章。本章向您展示如何使用第 1 章中构建的词法分析器为我们的 Kaleidoscope 语言构建一个完整的解析器(parser)。有了解析器后，我们将定义并构建一个抽象语法树(AST)。

我们将构建的解析器结合使用递归下降解析和 运算符优先解析来解析 Kaleidoscope 语言（后者用于二进制表达式，前者用于其他所有内容）。在我们开始解析之前，让我们先谈谈解析器的输出：抽象语法树。

## 抽象语法树（AST）

程序的 AST 以一种便于编译器的后期阶段（例如代码生成）解释的方式捕获其行为。我们基本上希望语言中的每个结构都有一个对象，而 AST 应该紧密地模拟语言。在 Kaleidoscope 中，我们有表达式、原型和函数对象。我们首先从表达式开始：

|  |
| --- |
| /// ExprAST - Base class for all expression nodes.  class ExprAST {  public:  virtual ~ExprAST() = default;  };  /// NumberExprAST - Expression class for numeric literals like "1.0".  class NumberExprAST : public ExprAST {  double Val;  public:  NumberExprAST(double Val) : Val(Val) {}  }; |

上面的代码显示了基本 ExprAST 类和我们用于数字文字的一个子类的定义。关于此代码需要注意的重要一点是 NumberExprAST 类将文字的数值捕获为实例变量。这允许编译器的后续阶段知道存储的数值是什么。

现在我们只创建 AST，所以它们没有有用的访问器方法。例如，添加一个虚拟方法来漂亮地打印代码将非常容易。以下是我们将在 Kaleidoscope 语言的基本形式中使用的其他表达式 AST 节点定义：

|  |
| --- |
| /// VariableExprAST - Expression class for referencing a variable, like "a".  class VariableExprAST : public ExprAST {  std::string Name;  public:  VariableExprAST(const std::string &Name) : Name(Name) {}  };  /// BinaryExprAST - Expression class for a binary operator.  class BinaryExprAST : public ExprAST {  char Op;  std::unique\_ptr<ExprAST> LHS, RHS;  public:  BinaryExprAST(char Op, std::unique\_ptr<ExprAST> LHS,  std::unique\_ptr<ExprAST> RHS)  : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}  };  /// CallExprAST - Expression class for function calls.  class CallExprAST : public ExprAST {  std::string Callee;  std::vector<std::unique\_ptr<ExprAST>> Args;  public:  CallExprAST(const std::string &Callee,  std::vector<std::unique\_ptr<ExprAST>> Args)  : Callee(Callee), Args(std::move(Args)) {}  }; |

这一切（有意）相当直接：变量捕获变量名称，二元运算符捕获它们的操作码（例如“+”），调用捕获函数名称以及任何参数表达式的列表。我们的 AST 的优点之一是它捕获了语言特性，而无需讨论语言的语法。请注意，这里没有讨论二元运算符的优先级、词法结构等。

对于我们的基本语言，这些是我们将定义的所有表达式节点。因为它没有条件控制流，所以它不是图灵完备的；我们将在以后的文章中解决这个问题。接下来我们需要做的两件事是一种谈论函数接口的方式，以及一种谈论函数本身的方式：

|  |
| --- |
| /// PrototypeAST - This class represents the "prototype" for a function,  /// which captures its name, and its argument names (thus implicitly the number  /// of arguments the function takes).  class PrototypeAST {  std::string Name;  std::vector<std::string> Args;  public:  PrototypeAST(const std::string &Name, std::vector<std::string> Args)  : Name(Name), Args(std::move(Args)) {}  const std::string &getName() const { return Name; }  };  /// FunctionAST - This class represents a function definition itself.  class FunctionAST {  std::unique\_ptr<PrototypeAST> Proto;  std::unique\_ptr<ExprAST> Body;  public:  FunctionAST(std::unique\_ptr<PrototypeAST> Proto,  std::unique\_ptr<ExprAST> Body)  : Proto(std::move(Proto)), Body(std::move(Body)) {}  }; |

在 Kaleidoscope 中，函数的类型只包含参数的个数。由于所有值都是双精度浮点数，因此不需要在任何地方存储每个参数的类型。在更激进和现实的语言中，“ExprAST”类可能有一个类型字段。

有了这个脚手架，我们现在可以讨论 Kaleidoscope 中的解析表达式和函数体。

## 解析器基础

现在我们有了要构建的 AST，我们需要定义解析器代码来构建它。这里的想法是，我们想要将诸如“x+y”之类的东西（由词法分析器作为三个标记返回）解析为一个 AST，可以通过这样的调用生成：

|  |
| --- |
| auto LHS = std::make\_unique<VariableExprAST>("x");  auto RHS = std::make\_unique<VariableExprAST>("y");  auto Result = std::make\_unique<BinaryExprAST>('+', std::move(LHS),  std::move(RHS)); |

为了做到这一点，我们将从定义一些基本的辅助例程开始：

|  |
| --- |
| /// CurTok/getNextToken - Provide a simple token buffer. CurTok is the current  /// token the parser is looking at. getNextToken reads another token from the  /// lexer and updates CurTok with its results.  static int CurTok;  static int getNextToken() {  return CurTok = gettok();  } |

这在词法分析器周围实现了一个简单的令牌缓冲区。这使我们能够提前查看词法分析器返回的内容。我们解析器中的每个函数都假定 CurTok 是当前需要解析的标记。

|  |
| --- |
| /// LogError\* - These are little helper functions for error handling.  std::unique\_ptr<ExprAST> LogError(const char \*Str) {  fprintf(stderr, "Error: %s\n", Str);  return nullptr;  }  std::unique\_ptr<PrototypeAST> LogErrorP(const char \*Str) {  LogError(Str);  return nullptr;  } |

这些LogError例程是简单的辅助例程，我们的解析器将使用它们来处理错误。我们的解析器中的错误恢复不是最好的，也不是特别用户友好，但对于我们的教程来说已经足够了。这些例程可以更轻松地处理具有各种返回类型的例程中的错误：它们始终返回 null。

有了这些基本的辅助函数，我们就可以实现语法的第一部分：数字文字。

## 基本表达式解析

我们从数字文字开始，因为它们最容易处理。对于语法中的每个产生式，我们将定义一个解析该产生式的函数。对于数字文字，我们有：

|  |
| --- |
| /// numberexpr ::= number  static std::unique\_ptr<ExprAST> ParseNumberExpr() {  auto Result = std::make\_unique<NumberExprAST>(NumVal);  getNextToken(); // consume the number  return std::move(Result);  } |

这个例程非常简单：它期望在当前令牌是一个tok\_number令牌时被调用。它获取当前数值，创建一个NumberExprAST节点，将词法分析器推进到下一个标记，最后返回。

这有一些有趣的方面。最重要的一点是，这个例程会吃掉与产生式对应的所有标记，并返回带有下一个标记（不属于语法产生式）的词法分析器缓冲区。这是递归下降解析器的一种相当标准的方法。举个更好的例子，括号运算符定义如下：

|  |
| --- |
| /// parenexpr ::= '(' expression ')'  static std::unique\_ptr<ExprAST> ParseParenExpr() {  getNextToken(); // eat (.  auto V = ParseExpression();  if (!V)  return nullptr;  if (CurTok != ')')  return LogError("expected ')'");  getNextToken(); // eat ).  return V;  } |

这个函数说明了关于解析器的一些有趣的事情：

1) 它显示了我们如何使用 LogError 例程。调用时，此函数期望当前标记为 '(' 标记，但在解析子表达式后，可能没有 ')' 等待。例如，如果用户输入“(4 x”而不是“(4)”，解析器应该发出一个错误。因为可能会发生错误，解析器需要一种方法来表明它们发生了：在我们的解析器中，我们返回错误时为 null。

2）这个函数还有一个有趣的地方是它通过调用来使用递归ParseExpression（我们很快就会看到 ParseExpression可以调用ParseParenExpr）。这很强大，因为它允许我们处理递归语法，并使每个产生式都非常简单。请注意，括号不会导致 AST 节点本身的构造。虽然我们可以这样做，但括号最重要的作用是引导解析器并提供分组。一旦解析器构造了 AST，就不需要括号了。

下一个简单的产生式用于处理变量引用和函数调用：

|  |
| --- |
| /// ::= identifier  /// ::= identifier '(' expression\* ')'  static std::unique\_ptr<ExprAST> ParseIdentifierExpr() {  std::string IdName = IdentifierStr;  getNextToken(); // eat identifier.  if (CurTok != '(') // Simple variable ref.  return std::make\_unique<VariableExprAST>(IdName);  // Call.  getNextToken(); // eat (  std::vector<std::unique\_ptr<ExprAST>> Args;  if (CurTok != ')') {  while (true) {  if (auto Arg = ParseExpression())  Args.push\_back(std::move(Arg));  else  return nullptr;  if (CurTok == ')')  break;  if (CurTok != ',')  return LogError("Expected ')' or ',' in argument list");  getNextToken();  }  }  // Eat the ')'.  getNextToken();  return std::make\_unique<CallExprAST>(IdName, std::move(Args));  } |

此例程遵循与其他例程相同的样式。（如果当前令牌是tok\_identifier令牌，它预计会被调用）。它还具有递归和错误处理。一个有趣的方面是它使用前瞻性来确定当前标识符是一个独立的变量引用还是一个函数调用表达式。它通过检查标识符后的标记是否为“(”标记来处理此问题，并根据需要构造一个VariableExprAST或 CallExprAST节点。

现在我们已经准备好所有简单的表达式解析逻辑，我们可以定义一个辅助函数来将它包装到一个入口点中。我们称这类表达式为“primary”表达式，原因在本教程后面会变得更加清楚。为了解析任意主表达式，我们需要确定它是哪种表达式：

|  |
| --- |
| /// primary  /// ::= identifierexpr  /// ::= numberexpr  /// ::= parenexpr  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();  }  } |

既然您看到了这个函数的定义，就更清楚了为什么我们可以在各种函数中假设 CurTok 的状态。这使用前瞻来确定正在检查哪种表达式，然后使用函数调用对其进行解析。

现在处理了基本表达式，我们需要处理二进制表达式。它们有点复杂。

## Binary Expression Parsing二进制表达式解析

二进制表达式更难解析，因为它们通常是模棱两可的。例如，当给定字符串“x+y\*z”时，解析器可以选择将其解析为“(x+y)\*z”或“x+(y\*z)”。根据数学上的常见定义，我们期望后面的解析，因为“\*”（乘法）比“+”（加法）具有更高的优先级。

有很多方法可以处理这个问题，但一种优雅而有效的方法是使用Operator-Precedence Parsing。这种解析技术使用二元运算符的优先级来指导递归。首先，我们需要一个优先级表：

|  |
| --- |
| /// BinopPrecedence - This holds the precedence for each binary operator that is  /// defined.  static std::map<char, int> BinopPrecedence;  /// GetTokPrecedence - Get the precedence of the pending binary operator token.  static int GetTokPrecedence() {  if (!isascii(CurTok))  return -1;  // Make sure it's a declared binop.  int TokPrec = BinopPrecedence[CurTok];  if (TokPrec <= 0) return -1;  return TokPrec;  }  int main() {  // Install standard binary operators.  // 1 is lowest precedence.  BinopPrecedence['<'] = 10;  BinopPrecedence['+'] = 20;  BinopPrecedence['-'] = 20;  BinopPrecedence['\*'] = 40; // highest.  ...  } |

对于 Kaleidoscope 的基本形式，我们将只支持 4 个二元运算符（这显然可以由您，我们勇敢无畏的读者进行扩展）。该GetTokPrecedence函数返回当前标记的优先级，如果标记不是二元运算符，则返回 -1。有了映射可以很容易地添加新的运算符，并清楚地表明该算法不依赖于所涉及的特定运算符，但是消除映射并在 GetTokPrecedence函数中进行比较就足够容易了。（或者只使用固定大小的数组）。

通过上面定义的帮助器，我们现在可以开始解析二进制表达式了。运算符优先级解析的基本思想是将具有可能不明确的二元运算符的表达式分解成多个部分。例如，考虑表达式“a+b+(c+d)\*e\*f+g”。运算符优先级解析将其视为由二元运算符分隔的主要表达式流。因此，它将首先解析前导主表达式“a”，然后它会看到对 [+, b] [+, (c+d)] [\*, e] [\*, f] 和 [+, g ]. 请注意，因为括号是基本表达式，二进制表达式解析器根本不需要担心像 (c+d) 这样的嵌套子表达式。

首先，表达式是一个主表达式，后面可能跟着一系列 [binop,primaryexpr] 对：

|  |
| --- |
| /// expression  /// ::= primary binoprhs  ///  static std::unique\_ptr<ExprAST> ParseExpression() {  auto LHS = ParsePrimary();  if (!LHS)  return nullptr;  return ParseBinOpRHS(0, std::move(LHS));  } |

ParseBinOpRHS是为我们解析对序列的函数。它需要一个优先级和一个指向到目前为止已解析部分的表达式的指针。请注意，“x”是一个完全有效的表达式：因此，“binoprhs”允许为空，在这种情况下，它返回传递给它的表达式。在我们上面的示例中，代码将“a”的表达式传递给ParseBinOpRHS当前标记为“+”。

传入的优先级值ParseBinOpRHS表示 允许函数吃掉的最小运算符优先级。例如，如果当前对流是 [+, x] 并ParseBinOpRHS以 40 的优先级传递，则它不会消耗任何标记（因为 '+' 的优先级只有 20）。考虑到这一点，ParseBinOpRHS 开始于：

|  |
| --- |
| /// binoprhs  /// ::= ('+' primary)\*  static std::unique\_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,  std::unique\_ptr<ExprAST> LHS) {  // If this is a binop, find its precedence.  while (true) {  int TokPrec = GetTokPrecedence();  // If this is a binop that binds at least as tightly as the current binop,  // consume it, otherwise we are done.  if (TokPrec < ExprPrec)  return LHS; |

此代码获取当前令牌的优先级并检查是否太低。因为我们将无效标记定义为具有 -1 的优先级，所以此检查隐含地知道当标记流用完二元运算符时 pair-stream 结束。如果检查成功，我们就知道令牌是一个二元运算符，并且它将包含在这个表达式中：

|  |
| --- |
| // Okay, we know this is a binop.  int BinOp = CurTok;  getNextToken(); // eat binop  // Parse the primary expression after the binary operator.  auto RHS = ParsePrimary();  if (!RHS)  return nullptr; |

因此，此代码吃掉（并记住）二元运算符，然后解析后面的主要表达式。这构建了整个对，其中第一个是运行示例的 [+, b] 。

现在我们已经解析了表达式的左侧和一对 RHS 序列，我们必须决定表达式的关联方式。特别是，我们可以有“(a+b) binop unparsed”或“a + (b binop unparsed)”。为了确定这一点，我们向前看“binop”以确定其优先级并将其与 BinOp 的优先级（在本例中为“+”）进行比较：

|  |
| --- |
| // If BinOp binds less tightly with RHS than the operator after RHS, let  // the pending operator take RHS as its LHS.  int NextPrec = GetTokPrecedence();  if (TokPrec < NextPrec) { |

如果“RHS”右侧的二元运算符的优先级低于或等于我们当前运算符的优先级，那么我们知道括号关联为“(a+b)二元运算符…”。在我们的示例中，当前运算符是“+”，下一个运算符是“+”，我们知道它们具有相同的优先级。在这种情况下，我们将为“a+b”创建 AST 节点，然后继续解析：

|  |
| --- |
| ... if body omitted ...  }  // Merge LHS/RHS.  LHS = std::make\_unique<BinaryExprAST>(BinOp, std::move(LHS),  std::move(RHS));  } // loop around to the top of the while loop.  } |

在我们上面的例子中，这会将“a+b+”变成“(a+b)”并执行循环的下一次迭代，“+”作为当前标记。上面的代码会将“(c+d)”作为主要表达式来获取、记忆和解析，这使得当前对等于 [+, (c+d)]。然后它将评估上面的“if”条件，“\*”作为主要右侧的 binop。在这种情况下，“\*”的优先级高于“+”的优先级，因此将进入 if 条件。

这里留下的关键问题是“if 条件如何完整解析右侧”？特别是，要为我们的示例正确构建 AST，它需要获取所有“(c+d)\*e\*f”作为 RHS 表达式变量。执行此操作的代码出奇地简单（根据上下文重复上述两个块中的代码）：

|  |
| --- |
| // If BinOp binds less tightly with RHS than the operator after RHS, let  // the pending operator take RHS as its LHS.  int NextPrec = GetTokPrecedence();  if (TokPrec < NextPrec) {  RHS = ParseBinOpRHS(TokPrec+1, std::move(RHS));  if (!RHS)  return nullptr;  }  // Merge LHS/RHS.  LHS = std::make\_unique<BinaryExprAST>(BinOp, std::move(LHS),  std::move(RHS));  } // loop around to the top of the while loop.  } |

在这一点上，我们知道我们主要的 RHS 的二元运算符比我们当前正在解析的 binop 具有更高的优先级。因此，我们知道任何运算符优先级都高于“+”的对序列应该一起解析并返回为“RHS”。为此，我们递归调用ParseBinOpRHS指定“TokPrec+1”作为其继续所需的最低优先级的函数。在我们上面的示例中，这将导致它返回“(c+d)\*e\*f”的 AST 节点作为 RHS，然后将其设置为 '+' 表达式的 RHS。

最后，在 while 循环的下一次迭代中，“+g”部分被解析并添加到 AST 中。通过这一小段代码（14 行非常重要的代码），我们以一种非常优雅的方式正确地处理了完全通用的二进制表达式解析。这是这段代码的旋风之旅，它有些微妙。我建议通过一些棘手的例子来了解它是如何工作的。

这结束了表达式的处理。此时，我们可以将解析器指向任意标记流并从中构建表达式，并在不属于表达式的第一个标记处停止。接下来我们需要处理函数定义等。

## 解析其余部分

接下来缺少的是函数原型的处理。在 Kaleidoscope 中，它们既用于“外部”函数声明，也用于函数体定义。执行此操作的代码很简单，也不是很有趣（一旦你从表达式中幸存下来）：

|  |
| --- |
| /// ::= id '(' id\* ')'  static std::unique\_ptr<PrototypeAST> ParsePrototype() {  if (CurTok != tok\_identifier)  return LogErrorP("Expected function name in prototype");  std::string FnName = IdentifierStr;  getNextToken();  if (CurTok != '(')  return LogErrorP("Expected '(' in prototype");  // Read the list of argument names.  std::vector<std::string> ArgNames;  while (getNextToken() == tok\_identifier)  ArgNames.push\_back(IdentifierStr);  if (CurTok != ')')  return LogErrorP("Expected ')' in prototype");  // success.  getNextToken(); // eat ')'.  return std::make\_unique<PrototypeAST>(FnName, std::move(ArgNames));  } |

鉴于此，函数定义非常简单，只是一个原型加上一个表达式来实现函数体：

|  |
| --- |
| /// definition ::= 'def' prototype expression  static std::unique\_ptr<FunctionAST> ParseDefinition() {  getNextToken(); // eat def.  auto Proto = ParsePrototype();  if (!Proto) return nullptr;  if (auto E = ParseExpression())  return std::make\_unique<FunctionAST>(std::move(Proto), std::move(E));  return nullptr;  } |

此外，我们支持'extern'来声明'sin'和'cos'等函数，并支持用户函数的前向声明。这些“外部”只是没有主体的原型：

|  |
| --- |
| /// external ::= 'extern' prototype  static std::unique\_ptr<PrototypeAST> ParseExtern() {  getNextToken(); // eat extern.  return ParsePrototype();  } |

最后，我们还将让用户键入任意顶级表达式并即时计算它们。我们将通过为它们定义匿名 nullary（零参数）函数来处理这个问题：

|  |
| --- |
| /// toplevelexpr ::= expression  static std::unique\_ptr<FunctionAST> ParseTopLevelExpr() {  if (auto E = ParseExpression()) {  // Make an anonymous proto.  auto Proto = std::make\_unique<PrototypeAST>("", std::vector<std::string>());  return std::make\_unique<FunctionAST>(std::move(Proto), std::move(E));  }  return nullptr;  } |

现在我们有了所有的部分，让我们构建一个小驱动程序，让我们实际执行我们构建的代码！

## driver驱动

这个驱动程序简单地调用所有带有顶层调度循环的解析部分。这里没有太多有趣的地方，所以我只包含顶级循环。请参阅下面“顶级解析”部分中的完整代码。

|  |
| --- |
| static void MainLoop() {  while (true) {  fprintf(stderr, "ready> ");  switch (CurTok) {  case tok\_eof:  return;  case ';': // ignore top-level semicolons.  getNextToken();  break;  case tok\_def:  HandleDefinition();  break;  case tok\_extern:  HandleExtern();  break;  default:  HandleTopLevelExpression();  break;  }  }  } |

其中最有趣的部分是我们忽略了顶级分号。你问这是为什么？基本原因是，如果您在命令行键入“4 + 5”，解析器不知道这是否是您将键入的内容的结尾。例如，在下一行中，您可以键入“def foo ...”，在这种情况下，4+5 是顶级表达式的结尾。或者，您可以键入“\* 6”，这将继续表达式。有了顶级分号，您就可以输入“4+5;”，解析器就会知道您已经完成了。

## 结论

通过不到 400 行带注释的代码（240 行非注释和非空白的代码），我们完全定义了我们的最小语言，包括词法分析器、解析器和 AST 构建器。完成后，可执行文件将验证 Kaleidoscope 代码并告诉我们它是否在语法上无效。例如，这是一个示例交互：

|  |
| --- |
| $ ./a.out  ready> def foo(x y) x+foo(y, 4.0);  Parsed a function definition.  ready> def foo(x y) x+y y;  Parsed a function definition.  Parsed a top-level expr  ready> def foo(x y) x+y );  Parsed a function definition.  Error: unknown token when expecting an expression  ready> extern sin(a);  ready> Parsed an extern  ready> ^D  $ |

这里有很大的扩展空间。您可以定义新的 AST 节点，以多种方式扩展语言等。在下一期中，我们将介绍如何从 AST 生成 LLVM 中间表示 (IR)。

## 运行

因为这使用了 LLVM 库，所以我们需要链接它们。为此，我们使用 llvm-config工具来通知我们的 makefile/命令行关于使用哪些选项：

|  |
| --- |
| # Compile  clang++ -g -O3 toy.cpp `llvm-config --cxxflags`  # Run  ./a.out |

完整代码在llvm-project/llvm/examples/Kaleidoscope/Chapter2/toy.cpp。

# 万花筒：代码生成到 LLVM IR

## 简介

欢迎阅读“使用 LLVM 实现语言”教程的第 3 章。本章向您展示如何将第 2 章中构建的抽象语法树转换为 LLVM IR。这将教您一些关于 LLVM 的工作原理，并展示它的易用性。构建词法分析器和解析器比生成 LLVM IR 代码做的工作要多的多。:)

请注意：本章及更高版本中的代码需要 LLVM 3.7 或更高版本。LLVM 3.6 及之前的版本将无法使用它。另请注意，您需要使用与您的 LLVM 版本相匹配的本教程版本：如果您使用的是官方 LLVM 版本，请使用版本中包含的文档版本或llvm.org 版本页面上的版本。

## 代码生成设置

为了生成 LLVM IR，我们需要一些简单的设置来开始。首先我们在每个 AST 类中定义虚拟代码生成（codegen）方法：

|  |
| --- |
| /// ExprAST - Base class for all expression nodes.  class ExprAST {  public:  virtual ~ExprAST() = default;  virtual Value \*codegen() = 0;  };  /// NumberExprAST - Expression class for numeric literals like "1.0".  class NumberExprAST : public ExprAST {  double Val;  public:  NumberExprAST(double Val) : Val(Val) {}  Value \*codegen() override;  };  ... |

codegen() 方法表示为该 AST 节点发射的 IR 以及它所依赖的所有事物，它们都返回一个 LLVM Value 对象。“Value ”是用于表示 LLVM 中的“静态单一赋值 (SSA) 寄存器”或“SSA 值”的类。SSA 值最明显的方面是它们的值是在相关指令执行时计算的，并且在（如果）指令重新执行之前它不会获得新值。换句话说，无法“更改”SSA 值。有关更多信息，请阅读静态单一赋值 (SSA) - 一旦理解这些概念，它们就会非常自然。

请注意，不是将虚拟方法添加到 ExprAST 类层次结构，而是使用访问者模式或其他一些方式对其进行建模也很有意义。同样，本教程不会详述良好的软件工程实践：就我们的目的而言，添加虚拟方法是最简单的。

我们想要的第二件事是像我们用于解析器的“LogError”方法，它将用于报告在代码生成期间发现的错误（例如，使用未声明的参数）：

|  |
| --- |
| static std::unique\_ptr<LLVMContext> TheContext;  static std::unique\_ptr<IRBuilder<>> Builder(TheContext);  static std::unique\_ptr<Module> TheModule;  static std::map<std::string, Value \*> NamedValues;  Value \*LogErrorV(const char \*Str) {  LogError(Str);  return nullptr;  } |

静态变量将在代码生成期间使用。

TheContext 是一个不透明的对象，它拥有许多核心 LLVM 数据结构，例如类型和常量值表。我们不需要详细了解它，我们只需要将单个实例传递到需要它的 API 中。

该Builder对象是一个帮助对象，可以轻松生成 LLVM 指令。IRBuilder类模板的实例 跟踪当前插入指令的位置，并具有创建新指令的方法。

TheModule是一个包含函数和全局变量的 LLVM 结构。在许多方面，它是 LLVM IR 用于包含代码的顶级结构。它将拥有我们生成的所有 IR 的内存，这就是 codegen() 方法返回原始 Value\* 而不是 unique\_ptr<Value> 的原因。

该NamedValues映射跟踪在当前范围内定义了哪些值以及它们的 LLVM 表示是什么。（换句话说，它是代码的符号表）。在这种形式的万花筒中，唯一可以引用的就是函数参数。因此，函数参数在为其函数体生成代码时将在此映射中。

有了这些基础知识，我们就可以开始讨论如何为每个表达式生成代码了。请注意，这假定Builder 已设置为将代码生成到某物中。现在，我们假设这已经完成，我们将只用它来发出代码。

## 表达式代码生成

为表达式节点生成 LLVM 代码非常简单：我们所有四个表达式节点的注释代码少于 45 行。首先，我们将执行数字文字：

|  |
| --- |
| Value \*NumberExprAST::codegen() {  return ConstantFP::get(TheContext, APFloat(Val));  } |

在 LLVM IR 中，数字常量用 ConstantFP类表示，它在内部保存数值APFloat （APFloat具有保存任意精度浮点常量的能力）。这段代码基本上只是创建并返回一个ConstantFP。请注意，在 LLVM IR 中，常量都是唯一的并且是共享的。出于这个原因，API 使用“foo::get(…)”而不是“new foo(..)”或“foo::Create(..)”。

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

使用 LLVM 对变量的引用也非常简单。在 Kaleidoscope 的简单版本中，我们假设变量已经在某处发出并且它的值可用。实际上，NamedValues映射中唯一可以存在的值是函数参数。此代码只是检查指定的名称是否在映射中（如果不在，则引用了一个未知变量）并返回它的值。在以后的章节中，我们将在符号表中添加对循环归纳变量和局部变量的支持。

|  |
| --- |
| 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:  return LogErrorV("invalid binary operator");  }  } |

二元运算符开始变得更有趣。这里的基本思想是我们递归地为表达式的左侧发出代码，然后是右侧，然后我们计算二进制表达式的结果。在这段代码中，我们对操作码进行了简单的切换，以创建正确的 LLVM 指令。

在上面的示例中，LLVM 构建器类开始显示其价值。IRBuilder 知道在哪里插入新创建的指令，您所要做的就是指定要创建的指令（例如 with CreateFAdd），要使用的操作数（L和R此处），并可选地为生成的指令提供名称。

LLVM 的一个好处是名称只是一个提示。例如，如果上面的代码发出多个“addtmp”变量，LLVM 将自动为每个变量提供一个递增的、唯一的数字后缀。指令的本地值名称完全是可选的，但它使读取 IR 转储变得更加容易。

LLVM 指令受严格规则约束：例如，add 指令的 Left 和 Right 操作数必须具有相同的类型，add 的结果类型必须与操作数类型匹配。因为 Kaleidoscope 中的所有值都是双精度值，所以这使得 add、sub 和 mul 的代码非常简单。

另一方面，LLVM 指定fcmp 指令总是返回一个“i1”值（一位整数）。问题是 Kaleidoscope 希望该值是 0.0 或 1.0 值。为了获得这些语义，我们将 fcmp 指令与uitofp 指令结合起来。该指令通过将输入视为无符号值来将其输入整数转换为浮点值。相反，如果我们使用sitofp 指令，Kaleidoscope '<' 运算符将返回 0.0 和 -1.0，具体取决于输入值。

|  |
| --- |
| Value \*CallExprAST::codegen() {  // Look up the name in the global module table.  Function \*CalleeF = TheModule->getFunction(Callee);  if (!CalleeF)  return LogErrorV("Unknown function referenced");  // If argument mismatch error.  if (CalleeF->arg\_size() != Args.size())  return LogErrorV("Incorrect # arguments passed");  std::vector<Value \*> ArgsV;  for (unsigned i = 0, e = Args.size(); i != e; ++i) {  ArgsV.push\_back(Args[i]->codegen());  if (!ArgsV.back())  return nullptr;  }  return Builder->CreateCall(CalleeF, ArgsV, "calltmp");  } |

使用 LLVM 函数调用的代码生成非常简单。上面的代码最初在 LLVM 模块的符号表中查找函数名称。回想一下，LLVM 模块是包含我们正在 JIT 的函数的容器。通过为每个函数赋予与用户指定的名称相同的名称，我们可以使用 LLVM 符号表为我们解析函数名称。

一旦我们有了要调用的函数，我们就会递归地编码生成要传入的每个参数，并创建一个 LLVM调用指令。请注意，LLVM 默认使用本机 C 调用约定，允许这些调用也调用标准库函数，如“sin”和“cos”，无需额外的努力。

到目前为止，我们对 Kaleidoscope 中的四个基本表达式的处理到此结束。随意进入并添加更多内容。例如，通过浏览LLVM 语言参考，您会发现其他一些有趣的指令，它们非常容易插入我们的基本框架。

## 函数代码生成

原型和函数的代码生成必须处理许多细节，这使得它们的代码不如表达式代码生成那么美观，但可以让我们说明一些要点。首先，让我们谈谈原型的代码生成：它们既用于函数体，也用于外部函数声明。代码以：

|  |
| --- |
| Function \*PrototypeAST::codegen() {  // Make the function type: double(double,double) etc.  std::vector<Type\*> Doubles(Args.size(),  Type::getDoubleTy(\*TheContext));  FunctionType \*FT =  FunctionType::get(Type::getDoubleTy(\*TheContext), Doubles, false);  Function \*F =  Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get()); |

这段代码在几行代码中包含了很多功能。首先请注意，此函数返回“Function\*”而不是“Value\*”。因为“prototype”实际上是在谈论函数的外部接口（而不是表达式计算的值），所以在代码生成时返回它对应的 LLVM 函数是有意义的。

调用FunctionType::get创建FunctionType应该用于给定原型的 。由于 Kaleidoscope 中的所有函数参数都是双精度类型，因此第一行创建了一个包含“N”个 LLVM 双精度类型的向量。然后它使用该Functiontype::get方法创建一个函数类型，该函数类型以“N”双精度作为参数，返回一个双精度作为结果，并且不是可变参数（false 参数表明这一点）。请注意，LLVM 中的类型就像常量一样是唯一的，因此您不是“new”一个类型，而是“get”它。

上面的最后一行实际上创建了与原型对应的 IR 函数。这表明要使用的类型、链接和名称，以及要插入的模块。“extern linkage”意味着该函数可以在当前模块外部定义和/或可以由模块外部的函数调用。传入的 Name 是用户指定的名称：因为指定了“ TheModule”，所以这个名称被注册到 “ TheModule”的符号表中。

|  |
| --- |
| // Set names for all arguments.  unsigned Idx = 0;  for (auto &Arg : F->args())  Arg.setName(Args[Idx++]);  return F; |

最后，我们根据原型中给出的名称设置每个函数参数的名称。此步骤并非绝对必要，但保持名称一致可使 IR 更具可读性，并允许后续代码直接引用其名称的参数，而不必在 Prototype AST 中查找它们。

在这一点上，我们有一个没有主体的函数原型。这就是 LLVM IR 表示函数声明的方式。对于 Kaleidoscope 中的 extern 语句，这就是我们需要做的。然而，对于函数定义，我们需要代码生成并附加一个函数体。

|  |
| --- |
| Function \*FunctionAST::codegen() {  // First, check for an existing function from a previous 'extern' declaration.  Function \*TheFunction = TheModule->getFunction(Proto->getName());  if (!TheFunction)  TheFunction = Proto->codegen();  if (!TheFunction)  return nullptr;  if (!TheFunction->empty())  return (Function\*)LogErrorV("Function cannot be redefined."); |

对于函数定义，我们首先在 TheModule 的符号表中搜索该函数的现有版本，以防已经使用“extern”语句创建了一个版本。如果 Module::getFunction 返回 null，那么之前的版本不存在，所以我们将从原型中生成一个。在任何一种情况下，我们都想在开始之前断言函数是空的（即还没有主体）。

|  |
| --- |
| // Create a new basic block to start insertion into.  BasicBlock \*BB = BasicBlock::Create(\*TheContext, "entry", TheFunction);  Builder->SetInsertPoint(BB);  // Record the function arguments in the NamedValues map.  NamedValues.clear();  for (auto &Arg : TheFunction->args())  NamedValues[std::string(Arg.getName())] = &Arg; |

现在我们到了设置Builder的点了。第一行创建一个新的基本块 （名为“entry”），它被插入到TheFunction. 然后第二行告诉构建器应该将新指令插入到新基本块的末尾。LLVM 中的基本块是定义控制流图的函数的重要组成部分。由于我们没有任何控制流，我们的函数此时将只包含一个块。我们将在第 5 章解决这个问题:)。

接下来，我们将函数参数添加到 NamedValues 映射（在首先清除它之后），以便VariableExprAST节点可以访问它们。

|  |
| --- |
| if (Value \*RetVal = Body->codegen()) {  // Finish off the function.  Builder->CreateRet(RetVal);  // Validate the generated code, checking for consistency.  verifyFunction(\*TheFunction);  return TheFunction;  } |

一旦设置了插入点并填充了 NamedValues 映射，我们就codegen()为函数的根表达式调用该方法。如果没有错误发生，它会发出代码来计算入口块中的表达式并返回计算出的值。假设没有错误，我们然后创建一个 LLVM ret 指令，它完成了功能。构建函数后，我们调用verifyFunction，它由 LLVM 提供。此函数对生成的代码进行各种一致性检查，以确定我们的编译器是否正确执行所有操作。使用它很重要：它可以捕获很多错误。函数完成并验证后，我们将其返回。

|  |
| --- |
| // Error reading body, remove function.  TheFunction->eraseFromParent();  return nullptr;  } |

这里唯一剩下的就是处理错误情况。为简单起见，我们仅通过删除使用该 eraseFromParent方法生成的函数来处理此问题。这允许用户重新定义他们之前错误输入的函数：如果我们不删除它，它会存在于符号表中，带有主体，防止将来重新定义。

不过，这段代码确实有一个错误：如果该FunctionAST::codegen()方法找到一个现有的 IR 函数，它不会根据定义自己的原型验证其签名。这意味着较早的“外部”声明将优先于函数定义的签名，这可能导致代码生成失败，例如，如果函数参数的命名不同。有很多方法可以修复这个错误，看看你能想到什么！这是一个测试用例：

|  |
| --- |
| extern foo(a); # ok, defines foo.  def foo(b) b; # Error: Unknown variable name. (decl using 'a' takes precedence). |

## Driver Changes and Closing Thoughts

目前，LLVM 的代码生成并没有真正让我们得到太多，除了我们可以查看漂亮的 IR 调用。示例代码将对 codegen 的调用插入到“ HandleDefinition”、“ HandleExtern”等函数中，然后转储出 LLVM IR。这提供了一种查看 LLVM IR 以获得简单功能的好方法。例如：

|  |
| --- |
| ready> 4+5;  Read top-level expression:  define double @0() {  entry:  ret double 9.000000e+00  } |

请注意解析器如何将顶级表达式转换为我们的匿名函数。当我们在下一章中添加JIT 支持时，这会很方便。另请注意，代码是按字面意思转录的，除了 IRBuilder 完成的简单常量折叠外，没有执行任何优化。我们将在下一章中明确添加优化。

|  |
| --- |
| ready> def foo(a b) a\*a + 2\*a\*b + b\*b;  Read function definition:  define double @foo(double %a, double %b) {  entry:  %multmp = fmul double %a, %a  %multmp1 = fmul double 2.000000e+00, %a  %multmp2 = fmul double %multmp1, %b  %addtmp = fadd double %multmp, %multmp2  %multmp3 = fmul double %b, %b  %addtmp4 = fadd double %addtmp, %multmp3  ret double %addtmp4  } |

这显示了一些简单的算术。请注意与我们用来创建指令的 LLVM 构建器调用惊人的相似。

|  |
| --- |
| ready> def bar(a) foo(a, 4.0) + bar(31337);  Read function definition:  define double @bar(double %a) {  entry:  %calltmp = call double @foo(double %a, double 4.000000e+00)  %calltmp1 = call double @bar(double 3.133700e+04)  %addtmp = fadd double %calltmp, %calltmp1  ret double %addtmp  } |

这显示了一些函数调用。请注意，如果您调用此函数，将需要很长时间才能执行。将来我们将添加条件控制流来真正使递归有用 :)。

|  |
| --- |
| ready> extern cos(x);  Read extern:  declare double @cos(double)  ready> cos(1.234);  Read top-level expression:  define double @1() {  entry:  %calltmp = call double @cos(double 1.234000e+00)  ret double %calltmp  } |

这显示了 libm “cos”函数的 extern，以及对它的调用。

|  |
| --- |
| ready> ^D  ; ModuleID = 'my cool jit'  define double @0() {  entry:  %addtmp = fadd double 4.000000e+00, 5.000000e+00  ret double %addtmp  }  define double @foo(double %a, double %b) {  entry:  %multmp = fmul double %a, %a  %multmp1 = fmul double 2.000000e+00, %a  %multmp2 = fmul double %multmp1, %b  %addtmp = fadd double %multmp, %multmp2  %multmp3 = fmul double %b, %b  %addtmp4 = fadd double %addtmp, %multmp3  ret double %addtmp4  }  define double @bar(double %a) {  entry:  %calltmp = call double @foo(double %a, double 4.000000e+00)  %calltmp1 = call double @bar(double 3.133700e+04)  %addtmp = fadd double %calltmp, %calltmp1  ret double %addtmp  }  declare double @cos(double)  define double @1() {  entry:  %calltmp = call double @cos(double 1.234000e+00)  ret double %calltmp  } |

当您退出当前演示时（通过在 Linux 上通过 CTRL+D 或在 Windows 上通过 CTRL+Z 和 ENTER 发送 EOF），它会转储生成的整个模块的 IR。在这里，您可以看到所有功能相互引用的大图。

万花筒教程的第三章到此结束。接下来，我们将介绍如何为此添加 JIT 代码生成和优化器支持，以便我们真正开始运行代码！

## 代码运行

这是我们运行示例的完整代码清单，使用 LLVM 代码生成器进行了增强。因为这使用了 LLVM 库，所以我们需要链接它们。为此，我们使用 llvm-config工具来通知我们的 makefile/命令行关于使用哪些选项：

# Compile

clang++ -g -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy

# Run

./toy

# 万花筒：添加 JIT 和优化器支持

https://llvm.org/docs/tutorial/index.html

## 简介

欢迎阅读“[使用 LLVM 实现语言](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html)”教程的第 4 章。第 1-3 章描述了一种简单语言的实现，并添加了对生成 LLVM IR 的支持。本章介绍了两种新技术：为您的语言添加优化器支持，以及添加 JIT 编译器支持。这些新增内容将演示如何为 Kaleidoscope 语言获得漂亮、高效的代码。

## 平凡的常量折叠

我们第 3 章的演示非常优雅且易于扩展。不幸的是，它不会产生美妙的代码。然而，IRBuilder 在编译简单代码时确实为我们提供了明显的优化：

|  |
| --- |
| ready> def test(x) 1+2+x;  Read function definition:  define double @test(double %x) {  entry:  %addtmp = fadd double 3.000000e+00, %x  ret double %addtmp  } |

此代码不是通过解析输入构建的 AST 的文字转录。那将是：

|  |
| --- |
| ready> def test(x) 1+2+x;  Read function definition:  define double @test(double %x) {  entry:  %addtmp = fadd double 2.000000e+00, 1.000000e+00  %addtmp1 = fadd double %addtmp, %x  ret double %addtmp1  } |

特别是，如上所示，常量折叠是一种非常常见且非常重要的优化：以至于许多语言实现者在其 AST 表示中实现常量折叠支持。

使用 LLVM，您不需要 AST 中的这种支持。由于构建 LLVM IR 的所有调用都通过 LLVM IR buider，因此buider本身会在您调用它时检查是否有持续的折叠机会。如果是这样，它只是折叠常量并返回常量而不是创建指令。

好吧，这很容易 :)。在实践中，我们建议在生成这样的代码时始终使用 IRBuilder。它的使用没有“语法开销”（你不会因为不断到处常量检测，丑化你的编译器）并且它可以显着减少在某些情况下生成的 LLVM IR 的数量（特别是对于具有宏预处理器或使用了很多常量）。

另一方面，IRBuilder的局限性在于它会在构建代码时进行所有内联分析。如果你举一个稍微复杂一点的例子：

|  |
| --- |
| ready> def test(x) (1+2+x)\*(x+(1+2));  ready> Read function definition:  define double @test(double %x) {  entry:  %addtmp = fadd double 3.000000e+00, %x  %addtmp1 = fadd double %x, 3.000000e+00  %multmp = fmul double %addtmp, %addtmp1  ret double %multmp  } |

在这种情况下，乘法的 LHS 和 RHS 是相同的值。我们真的很想看到它生成“tmp = x+3; result = tmp\*tmp ”而不是计算“x+3;”两次。

不幸的是，再多的本地分析也无法检测和纠正这一点。这需要两个转换：表达式的重新关联（使 add 的词法相同）和公共子表达式消除 (CSE) 以删除冗余的 add 指令。幸运的是，LLVM 以“pass”的形式提供了广泛的优化供您使用。

## LLVM 优化pass

|  |
| --- |
| 警告 |
| 由于过渡到新的 PassManager 基础结构，本教程基于llvm::legacy::FunctionPassManager可在 [LegacyPassManager.h](https://llvm.org/doxygen/classllvm_1_1legacy_1_1FunctionPassManager.html)中找到的基础结构。出于本教程的目的，在 pass manager 转换完成之前，应该使用上面的内容。 |

LLVM 提供了许多优化passes，它们可以做许多不同种类的事情并有不同的权衡。与其他系统不同，LLVM 不会坚持认为一组优化适用于所有语言和所有情况的错误观念。LLVM 允许编译器实现者就使用什么优化、以什么顺序以及在什么情况下使用做出完整的决定。

作为一个具体的例子，LLVM 支持两种“whole module”passes，它们会查看尽可能多的代码体（通常是整个文件，但如果在链接时运行，这可能是整个程序的重要部分） . 它还支持并包括“per-function”passes，一次只对一个函数进行操作，而不考虑其他函数。有关passes及其运行方式的更多信息，请参阅 [如何编写passes](https://llvm.org/docs/WritingAnLLVMPass.html)文档和 [LLVM passes列表](https://llvm.org/docs/Passes.html)。

对于 Kaleidoscope，我们目前正在运行中生成函数，一次一个，当用户输入它们时。我们不是为了在这个设置中获得最终的优化体验，但我们也想抓住可能的简单快捷的东西。因此，我们将选择在用户键入函数时运行一些针对每个函数的优化。如果我们想制作一个“静态 Kaleidoscope 编译器”，我们将完全使用我们现在拥有的代码，只是我们将推迟运行优化器，直到整个文件被解析。

为了进行每个函数的优化，我们需要设置一个 [FunctionPassManager](https://llvm.org/docs/WritingAnLLVMPass.html" \l "what-passmanager-doesr)来保存和组织我们想要运行的 LLVM 优化。一旦我们有了它，我们就可以添加一组优化来运行。我们需要为每个要优化的模块创建一个新的 FunctionPassManager，因此我们将编写一个函数来为我们创建和初始化module 和 pass manager ：

|  |
| --- |
| void InitializeModuleAndPassManager(void) {  // Open a new module.  TheModule = std::make\_unique<Module>("my cool jit", TheContext);  // Create a new pass manager attached to it.  TheFPM = std::make\_unique<legacy::FunctionPassManager>(TheModule.get());  // Do simple "peephole" optimizations and bit-twiddling optzns.  TheFPM->add(createInstructionCombiningPass());  // Reassociate expressions.  TheFPM->add(createReassociatePass());  // Eliminate Common SubExpressions.  TheFPM->add(createGVNPass());  // Simplify the control flow graph (deleting unreachable blocks, etc).  TheFPM->add(createCFGSimplificationPass());  TheFPM->doInitialization();  } |

此代码初始化全局模块TheModule，以及附加到TheModule的function pass manager TheFPM。设置pass manager后，我们使用一系列“add”调用来添加一堆 LLVM passes。

在这种情况下，我们选择添加四个优化过程。我们在这里选择的passes是一组非常标准的“cleanup”优化，可用于各种代码。我不会深入研究他们的工作，但相信我，他们是一个很好的起点 :)。

一旦设置了 PassManager，我们就需要使用它。我们通过在构造新创建的函数之后（在  FunctionAST::codegen()中）但在将其返回给客户端之前运行它来执行此操作：

|  |
| --- |
| if (Value \*RetVal = Body->codegen()) {  // Finish off the function.  Builder.CreateRet(RetVal);  // Validate the generated code, checking for consistency.  verifyFunction(\*TheFunction);  // Optimize the function.  TheFPM->run(\*TheFunction);  return TheFunction;  } |

如您所见，这非常简单。FunctionPassManager优化和更新 LLVM 函数\*，改进（希望）它的body部分。 有了这个，我们可以再次尝试上面的测试：

|  |
| --- |
| ready> def test(x) (1+2+x)\*(x+(1+2));  ready> Read function definition:  define double @test(double %x) {  entry:  %addtmp = fadd double %x, 3.000000e+00  %multmp = fmul double %addtmp, %addtmp  ret double %multmp  } |

正如预期的那样，我们现在得到了经过很好优化的代码，在每次执行该函数时都保存了一条浮点加法指令。

LLVM 提供了多种可在特定情况下使用的优化。可以使用一些关于[各种passes的文档](https://llvm.org/docs/Passes.html)，但不是很完整。另一个好的想法来源可以来自查看 Clang开始运行的passes。“ opt”工具允许您从命令行试验passes，因此您可以查看它们执行的任何操作。

现在我们的前端已经有了合理的代码，让我们来谈谈执行吧！

## 添加 JIT 编译器

LLVM IR 中可用的代码可以应用多种工具。例如，您可以对其进行优化（如我们上面所做的），可以将其转储为文本或二进制形式，可以将代码编译为某个目标的汇编文件 (.s)，或者可以对它进行 JIT 编译。 LLVM IR representation 的好处在于它是编译器许多不同部分之间的“通用货币”(“common currency”)。

在本节中，我们将为我们的解释器添加 JIT 编译器支持。我们希望 Kaleidoscope 的基本思想是让用户像现在一样输入函数体，但立即评估他们输入的顶级表达式。例如，如果他们输入“1 + 2;”，我们应该评估并打印出 3。如果他们定义了一个函数，他们应该能够从命令行调用它。

为此，我们首先准备环境为当前本机目标创建代码并声明和初始化 JIT。这是通过调用一些InitializeNativeTarget\\*函数并添加一个全局变量TheJIT，并在main中对其进行初始化来完成的 ：

|  |
| --- |
| static std::unique\_ptr<KaleidoscopeJIT> TheJIT;  ...  int main() {  InitializeNativeTarget();  InitializeNativeTargetAsmPrinter();  InitializeNativeTargetAsmParser();  // Install standard binary operators.  // 1 is lowest precedence.  BinopPrecedence['<'] = 10;  BinopPrecedence['+'] = 20;  BinopPrecedence['-'] = 20;  BinopPrecedence['\*'] = 40; // highest.  // Prime the first token.  fprintf(stderr, "ready> ");  getNextToken();  TheJIT = std::make\_unique<KaleidoscopeJIT>();  // Run the main "interpreter loop" now.  MainLoop();  return 0;  } |

我们还需要为 JIT 设置数据布局：

|  |
| --- |
| void InitializeModuleAndPassManager(void) {  // Open a new context and module.  TheContext = std::make\_unique<LLVMContext>();  TheModule = std::make\_unique<Module>("my cool jit", TheContext);  TheModule->setDataLayout(TheJIT->getDataLayout());  // Create a new builder for the module.  Builder = std::make\_unique<IRBuilder<>>(\*TheContext);  // Create a new pass manager attached to it.  TheFPM = std::make\_unique<legacy::FunctionPassManager>(TheModule.get());  ... |

KaleidoscopeJIT 类是专门为这些教程构建的简单 JIT，可在 [llvm-src/examples/Kaleidoscope/include/KaleidoscopeJIT.h](https://github.com/llvm/llvm-project/blob/main/llvm/examples/Kaleidoscope/include/KaleidoscopeJIT.h) 的 LLVM 源代码中找到。在后面的章节中，我们将了解它是如何工作的，并用新功能扩展它，但现在我们将把它当作既定的。它的 API 非常简单： addModule在 JIT 中添加一个 LLVM IR module，使其函数可用于执行(它的内存通过ResourceTracker来管理)。

我们可以使用这个简单的 API 并将解析顶级表达式的代码更改为如下所示：

|  |
| --- |
| static ExitOnError ExitOnErr;  ...  static void HandleTopLevelExpression() {  // Evaluate a top-level expression into an anonymous function.  if (auto FnAST = ParseTopLevelExpr()) {  if (FnAST->codegen()) {  // Create a ResourceTracker to track JIT'd memory allocated to our  // anonymous expression -- that way we can free it after executing.  auto RT = TheJIT->getMainJITDylib().createResourceTracker();  auto TSM = ThreadSafeModule(std::move(TheModule), std::move(TheContext));  ExitOnErr(TheJIT->addModule(std::move(TSM), RT));  InitializeModuleAndPassManager();  // Search the JIT for the \_\_anon\_expr symbol.  auto ExprSymbol = ExitOnErr(TheJIT->lookup("\_\_anon\_expr"));  assert(ExprSymbol && "Function not found");  // Get the symbol's address and cast it to the right type (takes no  // arguments, returns a double) so we can call it as a native function.  double (\*FP)() = (double (\*)())(intptr\_t)ExprSymbol.getAddress();  fprintf(stderr, "Evaluated to %f\n", FP());  // Delete the anonymous expression module from the JIT.  ExitOnErr(RT->remove());  } |

如果解析和代码生成成功，下一步就是将包含顶级表达式的模块添加到 JIT。我们通过调用 addModule 来完成此操作，它会触发模块中所有函数的代码生成，并接受ResourceTracker，稍后ResourceTracker可用于从 JIT 中删除该模块。一旦模块被添加到 JIT 中，它就不能再被修改，所以我们也打开一个新模块来保存后续代码，方法是调用InitializeModuleAndPassManager().

一旦将模块添加到 JIT 后，我们需要获得指向最终生成代码的指针。为此，我们调用 JIT 的 lookup 方法，并传递顶级表达式函数的名称：\_\_anon\_expr。因为我们刚刚添加了这个函数，所以我们断言 findSymbol 返回了一个结果。

接下来，我们通过调用 getAddress()符号来获取\_\_anon\_expr函数的内存地址。回想一下，我们将顶级表达式编译成一个独立的 LLVM 函数，该函数不接受任何参数并返回计算的双精度值。因为 LLVM JIT 编译器匹配本机平台 ABI，这意味着您可以将结果指针转换为该类型的函数指针并直接调用它。这意味着，JIT 编译的代码和静态链接到您的应用程序的本机机器代码之间没有区别。

最后，由于我们不支持对顶级表达式的重新求值，因此当我们完成释放相关内存时，我们会从 JIT 中删除模块。然而，请回想一下，我们在几行前（通过 InitializeModuleAndPassManager）创建的模块仍处于打开状态，等待添加新代码。

仅通过这两个更改，让我们看看 Kaleidoscope 现在是如何工作的！

|  |
| --- |
| ready> 4+5;  Read top-level expression:  define double @0() {  entry:  ret double 9.000000e+00  }  Evaluated to 9.000000 |

好吧，这看起来基本上可以正常工作。函数的转储显示了我们为输入的每个顶级表达式合成的“始终返回双精度的无参数函数”。这演示了非常基本的功能，但我们可以做更多吗？

|  |
| --- |
| ready> def testfunc(x y) x + y\*2;  Read function definition:  define double @testfunc(double %x, double %y) {  entry:  %multmp = fmul double %y, 2.000000e+00  %addtmp = fadd double %multmp, %x  ret double %addtmp  }  ready> testfunc(4, 10);  Read top-level expression:  define double @1() {  entry:  %calltmp = call double @testfunc(double 4.000000e+00, double 1.000000e+01)  ret double %calltmp  }  Evaluated to 24.000000  ready> testfunc(5, 10);  ready> LLVM ERROR: Program used external function 'testfunc' which could not be resolved! |

函数定义和调用也可以，但最后一行出了点问题。调用看起来有效，那么发生了什么？正如您可能已经从 API 中猜到的那样，模块是 JIT 的分配单元，而 testfunc 是包含匿名表达式的同一模块的一部分。当我们从 JIT 中删除该模块以释放匿名表达式的内存时，我们也删除了testfunc的定义。然后，当我们第二次尝试调用 testfunc 时，JIT 再也找不到它了。

解决此问题的最简单方法是将匿名表达式与其余函数定义放在一个单独的模块中。只要被调用的每个函数都有一个原型，并且在调用之前添加到 JIT 中，JIT 就会愉快地解决跨模块边界的函数调用。通过将匿名表达式放在不同的模块中，我们可以在不影响其余函数的情况下删除它。

事实上，我们将更进一步，将每个函数放入其自己的模块中。这样做允许我们利用 KaleidoscopeJIT 的一个有用属性，这将使我们的环境更像 REPL：函数可以多次添加到 JIT（不像每个函数必须有唯一定义的模块）。当您在 KaleidoscopeJIT 中查找符号时，它总是会返回最新的定义：

|  |
| --- |
| ready> def foo(x) x + 1;  Read function definition:  define double @foo(double %x) {  entry:  %addtmp = fadd double %x, 1.000000e+00  ret double %addtmp  }  ready> foo(2);  Evaluated to 3.000000  ready> def foo(x) x + 2;  define double @foo(double %x) {  entry:  %addtmp = fadd double %x, 2.000000e+00  ret double %addtmp  }  ready> foo(2);  Evaluated to 4.000000 |

为了让每个函数都存在于它自己的模块中，我们需要一种方法来将以前的函数声明重新生成到我们打开的每个新模块中：

|  |
| --- |
| static std::unique\_ptr<KaleidoscopeJIT> TheJIT;  ...  Function \*getFunction(std::string Name) {  // First, see if the function has already been added to the current module.  if (auto \*F = TheModule->getFunction(Name))  return F;  // If not, check whether we can codegen the declaration from some existing  // prototype.  auto FI = FunctionProtos.find(Name);  if (FI != FunctionProtos.end())  return FI->second->codegen();  // If no existing prototype exists, return null.  return nullptr;  }  ...  Value \*CallExprAST::codegen() {  // Look up the name in the global module table.  Function \*CalleeF = getFunction(Callee);  ...  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; |

为了实现这一点，我们将首先添加一个新的全局FunctionProtos变量 ，它保存每个函数的最新原型。我们还将添加一个便捷方法 getFunction()，以替换对TheModule->getFunction() 的调用。我们的便捷方法搜索TheModule现有的函数声明，如果找不到，则回退到从 FunctionProtos 生成新的声明。在CallExprAST::codegen()，我们只需要替换对TheModule->getFunction()。在FunctionAST::codegen()，我们需要先更新 FunctionProtos 映射，然后调用getFunction()。完成此操作后，我们始终可以在当前模块中为任何先前声明的函数获取函数声明。

我们还需要更新 HandleDefinition 和 HandleExtern：

|  |
| --- |
| static void HandleDefinition() {  if (auto FnAST = ParseDefinition()) {  if (auto \*FnIR = FnAST->codegen()) {  fprintf(stderr, "Read function definition:");  FnIR->print(errs());  fprintf(stderr, "\n");  ExitOnErr(TheJIT->addModule(  ThreadSafeModule(std::move(TheModule), std::move(TheContext))));  InitializeModuleAndPassManager();  }  } else {  // Skip token for error recovery.  getNextToken();  }  }  static void HandleExtern() {  if (auto ProtoAST = ParseExtern()) {  if (auto \*FnIR = ProtoAST->codegen()) {  fprintf(stderr, "Read extern: ");  FnIR->print(errs());  fprintf(stderr, "\n");  FunctionProtos[ProtoAST->getName()] = std::move(ProtoAST);  }  } else {  // Skip token for error recovery.  getNextToken();  }  } |

在 HandleDefinition 中，我们添加两行将新定义的函数传递给 JIT 并打开一个新模块。在 HandleExtern 中，我们只需要添加一行，将原型添加到 FunctionProtos 中。

|  |
| --- |
| 警告 |
| 自 LLVM-9 以来，不允许在单独的模块中重复符号。这意味着您不能在 Kaleidoscope 中重新定义功能，如下所示。跳过这部分。  原因是较新的 OrcV2 JIT API 试图保持非常接近静态和动态链接器规则，包括拒绝重复符号。要求符号名称是唯一的允许我们使用（唯一的）符号名称作为跟踪键来支持符号的并发编译。 |

做出这些更改后，让我们再次尝试我们的 REPL（这次我删除了匿名函数的转储，你现在应该明白了:)：

|  |
| --- |
| ready> def foo(x) x + 1;  ready> foo(2);  Evaluated to 3.000000  ready> def foo(x) x + 2;  ready> foo(2);  Evaluated to 4.000000 |

有用！

即使使用这个简单的代码，我们也可以获得一些令人惊讶的强大功能 - 检查一下：

|  |
| --- |
| ready> extern sin(x);  Read extern:  declare double @sin(double)  ready> extern cos(x);  Read extern:  declare double @cos(double)  ready> sin(1.0);  Read top-level expression:  define double @2() {  entry:  ret double 0x3FEAED548F090CEE  }  Evaluated to 0.841471  ready> def foo(x) sin(x)\*sin(x) + cos(x)\*cos(x);  Read function definition:  define double @foo(double %x) {  entry:  %calltmp = call double @sin(double %x)  %multmp = fmul double %calltmp, %calltmp  %calltmp2 = call double @cos(double %x)  %multmp4 = fmul double %calltmp2, %calltmp2  %addtmp = fadd double %multmp, %multmp4  ret double %addtmp  }  ready> foo(4.0);  Read top-level expression:  define double @3() {  entry:  %calltmp = call double @foo(double 4.000000e+00)  ret double %calltmp  }  Evaluated to 1.000000 |

哇哦，JIT 是怎么知道 sin 和 cos 的？答案出奇地简单：KaleidoscopeJIT 有一个简单的符号解析规则，用于查找任何给定模块中不可用的符号：首先，它搜索所有已添加到 JIT 的模块，从最近的到最旧的，找到最新的定义。如果在 JIT 中没有找到定义，它会回退到在 Kaleidoscope 进程本身上调用“dlsym("sin")”。由于“ sin”是在 JIT 的地址空间中定义的，它只是简单地修补模块中的调用以直接地调用sin的 libm 版本。但在某些情况下，这甚至更进一步：由于 sin 和 cos 是标准数学函数的名称，常量文件夹将直接评估函数调用，以便在使用上面的“sin(1.0)”中的常量调用时获得正确的结果。

将来我们将看到如何调整此符号解析规则以启用各种有用的功能，从安全性（限制可用于 JIT 代码的符号集）到基于符号名称的动态代码生成，以及甚至懒惰的编译。

符号解析规则的一个直接好处是我们现在可以通过编写任意 C++ 代码来扩展语言来实现操作。例如，如果我们添加：

|  |
| --- |
| #ifdef \_WIN32  #define DLLEXPORT \_\_declspec(dllexport)  #else  #define DLLEXPORT  #endif  /// putchard - putchar that takes a double and returns 0.  extern "C" DLLEXPORT double putchard(double X) {  fputc((char)X, stderr);  return 0;  } |

请注意，对于 Windows，我们需要实际导出函数，因为动态符号加载器将使用 GetProcAddress 来查找符号。

现在我们可以通过使用类似以下内容的方式向控制台生成简单的输出：“ extern putchard(x); putchard(120)”，它在控制台上打印一个小写的“x”（120 是“x”的 ASCII 代码）。类似的代码可用于在 Kaleidoscope 中实现文件 I/O、控制台输入和许多其他功能。

Kaleidoscope 教程的 JIT 和优化器章节到此结束。此时，我们可以编译一个非图灵完备的编程语言，以用户驱动的方式对其进行优化和JIT编译。接下来我们将研究使用控制流结构来扩展语言，同时解决一些有趣的 LLVM IR 问题。

## 运行

这是我们运行示例的完整代码清单，使用 LLVM JIT 和优化器进行了增强。要构建此示例，请使用：

# Compile

clang++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` -O3 -o toy# Run

./toy

如果您在 Linux 上编译它，请确保也添加“-rdynamic”选项。这可确保在运行时正确解析外部函数。

# 附录