目 录

[1. 概述 1](#_Toc1838941285)

[2. 万花筒语言和Lexer 2](#_Toc1143507182)

[2.1 万花筒语言 2](#_Toc1944617928)

[2.2 Lexer 2](#_Toc1818102086)

[3. Kaleidoscope: Implementing a Parser and AST 4](#_Toc1374547745)

[3.1 简介 4](#_Toc93319659)

[3.2 抽象语法树（AST） 4](#_Toc965142438)

[3.3 分析器基础 6](#_Toc620371218)

[3.4 基本表达式分析 7](#_Toc1552574730)

[3.5 二进制表达式分析 9](#_Toc791185976)

[3.6 分析其余部分 11](#_Toc1862524902)

[3.7 驱动 12](#_Toc749879564)

[3.8 结论 13](#_Toc410530618)

[3.9 完整代码列表 13](#_Toc1248328365)

[4. 参考文档 21](#_Toc1639537446)

# 概述

My First Language Frontend with LLVM Tutorial

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

要求：本教程假设您了解C++，但不需要以前的编译器经验。

欢迎来到“我的第一语言前端与LLVM”教程。在这里，我们运行了一个简单语言的实现，展示了它是多么有趣和容易。本教程将帮助您快速入门并运行，并展示一个使用LLVM生成代码的具体示例。

本教程介绍了简单的“万花筒”语言，在几个章节的过程中反复构建它，展示了它是如何随着时间的推移而构建的。这使我们能够涵盖一系列语言设计和LLVM特定的思想，一路展示和解释它的代码，并减少了大量的预先细节。我们强烈建议您使用此代码-复制并修改它并进行实验。

警告：为了特别关注编译器技术和LLVM的教学，本教程没有展示软件工程原理中的最佳实践。例如，代码普遍使用全局变量，不使用访问者等……而是使事情简单化，并关注手头的主题。

本教程分为多个章节，涵盖各个主题，允许您根据需要跳过：

第1章：万花筒语言和Lexer——这展示了我们的目标和我们想要构建的基本功能。词法分析器也是为语言构建解析器的第一部分，我们使用一个简单的易于理解的C++词法分析器。

第二章：实现Parser和AST——在lexer就位后，我们可以讨论解析技术和基本的AST构造。本教程介绍递归下降解析和运算符优先解析。

第3章：LLVMIR的代码生成-在AST就绪的情况下，我们将展示生成LLVMIR是多么容易，并展示一种将LLVM合并到项目中的简单方法。

第4章：添加JIT和Optimizer支持——LLVM的一个优点是它支持JIT编译，所以我们将深入研究它，并向您展示添加JIT支持所需的3行代码。后面的章节将介绍如何生成.o文件。

第五章：扩展语言：控制流——随着基本语言的启动和运行，我们将展示如何使用控制流操作（“if”语句和“for”循环）来扩展它。这给了我们一个机会来谈谈SSA的建设和控制流程。

第6章：扩展语言：用户定义运算符-本章扩展了语言，让用户定义任意的一元和二元运算符-具有可分配的优先级！这允许我们将“语言”的重要部分构建为库例程。

第7章：扩展语言：可变变量-本章讨论添加用户定义的局部变量和赋值运算符。这表明在LLVM中构造SSA表单是多么容易：LLVM不需要前端构造SSA表格就可以使用它！

第8章：编译成对象文件——本章解释了如何像静态编译器那样，获取LLVMIR并将其编译成对象文档。

第9章：调试信息-一种真正的语言需要支持调试器，所以我们添加了调试信息，允许在Kaleidoscope函数中设置断点、打印参数变量和调用函数！

第10章：结论和其他花絮-本章通过讨论扩展语言的方法来结束本系列，并包括指向“特殊主题”信息的指针，如添加垃圾收集支持、异常、调试、支持“意大利面堆栈”等。

到本教程结束时，我们将编写不到1000行（非注释、非空白）的代码。有了这些少量的代码，我们将为一种非平凡的语言构建一个漂亮的小编译器，包括手写的lexer、解析器、AST，以及代码生成支持——静态和JIT！这一广度很好地证明了LLVM的优势，并说明了为什么它是语言设计者和其他需要高性能代码生成的人的热门目标。

# 万花筒语言和Lexer

## 万花筒语言

本教程使用一种名为“万花筒”的玩具语言进行演示（源于“美丽、形式和视图的含义”）。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Kaleidocpe以支持if/then/else构造、for循环、用户定义的运算符、具有简单命令行界面的JIT编译、调试信息等。

我们希望保持简单，所以万花筒中唯一的数据类型是64位浮点类型（C语言中的“double”）。因此，所有值都是隐式双精度的，并且该语言不需要类型声明。这为语言提供了一个非常好且简单的语法。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使这变得非常简单。这意味着您可以在使用函数之前使用“extern”关键字来定义函数（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)) |

第六章中有一个更有趣的例子，我们编写了一个小的万花筒应用程序，它以不同的放大倍数显示Mandelbrot集。

让我们深入了解这门语言的实现！

## Lexer

在实现一种语言时，首先需要的是处理文本文件并识别其内容的能力。传统的方法是使用“lexer”（又名“scanner”）将输入分解为“tokens”。lexer返回的每个标记都包含一个标记代码和一些元数据（例如数字的数值）。首先，我们定义了可能性：

|  |
| --- |
| // 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 |

lexer返回的每个标记要么是token枚举值之一，要么是像“+”这样的“未知”字符，作为其ASCII值返回。如果当前标记是标识符，则IdentifierStr全局变量保存标识符的名称。如果当前标记是数字字面值（如1.0），则NumVal保留其值。为了简单起见，我们使用了全局变量，但这并不是真正的语言实现的最佳选择：）。

lexer的实际实现是一个名为gettok的函数。调用gettok函数以从标准输入返回下一个标记。其定义如下：

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

gettok的工作原理是调用C getchar（）函数，从标准输入中一次读取一个字符。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。

gettok需要做的下一件事是识别标识符和特定关键字，如“def”。万花筒通过这个简单的循环实现了这一点：

|  |
| --- |
| 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，此代码在读取标识符时设置“IdentifierStr”全局变量。此外，由于语言关键字由同一个循环匹配，因此我们在这里内联处理它们。数值类似：

|  |
| --- |
| 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;  } |

这是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 strtod函数将其转换为存储在NumVal中的数值。请注意，这没有进行足够的错误检查：它将错误地读取“1.23.45.67”，并将其处理为您键入的“1.23”。请随意扩展！接下来我们处理注释：

|  |
| --- |
| 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，然后返回下一个标记来处理注释。最后，如果输入与上述情况之一不匹配，那么它要么是像“+”这样的运算符字符，要么是文件的结尾。这些是用以下代码处理的：

|  |
| --- |
| // 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lexer（lexer的完整代码清单将在本教程的下一章中提供）。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。当我们有了它，我们将包括一个驱动程序，以便您可以一起使用lexer和解析器。

# Kaleidoscope: Implementing a Parser and AST

## 简介

欢迎阅读“使用LLVM实现语言”教程的第2章。本章向您展示了如何使用第2章中构建的lexer为我们的万花筒语言构建完整的解析器。一旦我们有了解析器，我们将定义并构建一个抽象语法树（Abstract Syntax Tree，AST）。

我们将构建的解析器使用递归下降分析和运算符优先分析的组合来解析万花筒语言（后者用于二进制表达式，前者用于其他所有内容）。在开始解析之前，让我们先讨论一下解析器的输出：抽象语法树。

## 抽象语法树（AST）

程序的AST捕获其行为的方式使编译器的后期阶段（例如代码生成）易于解释。我们基本上希望语言中的每个构造都有一个对象，AST应该对语言进行紧密建模。在万花筒中，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() {}  };  /// 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类和一个子类的定义，我们将其用于数字字面量。关于这段代码需要注意的重要一点是，NumberExprAST类将文本的数值捕获为实例变量。这允许编译器的后续阶段了解存储的数值是什么。

现在我们只创建AST，因此它们上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是很容易的。下面是我们将在万花筒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）都很简单：变量捕获变量名，二进制运算符捕获其操作码（例如“+”），调用捕获函数名以及任何参数表达式的列表。我们的AST有一个优点，那就是它可以捕获语言特性，而不必讨论语言的语法。注意，这里没有讨论二进制运算符的优先级、词汇结构等。

对于我们的基本语言，这些是我们要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；我们将在稍后的部分中解决这个问题。接下来我们需要的两件事是讨论函数接口的方法和讨论函数本身的方法：

|  |
| --- |
| /// 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)) {}  }; |

在万花筒中，函数的类型只包含其参数的计数。因为所有的值都是双精度浮点，所以每个参数的类型不需要存储在任何地方。在一种更激进、更真实的语言中，“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。

## 分析器基础

现在我们有一个AST要构建，我们需要定义解析器代码来构建它。这里的想法是，我们希望将类似“x+y”的内容（由lexer作为三个标记返回）解析为一个AST，该AST可以通过如下调用生成：

|  |
| --- |
| 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，我们将首先定义一些基本的助手例程：

|  |
| --- |
| /// 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lexer周围实现了一个简单的令牌缓冲区。这允许我们提前查看lexer返回的一个标记。解析器中的每个函数都将假定CurTok是需要解析的当前标记。

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

LogError例程是我们的解析器用来处理错误的简单助手例程。我们的解析器中的错误恢复并不是最好的，也不是特别友好，但对于我们的教程来说，这就足够了。这些例程使处理具有不同返回类型的例程中的错误变得更容易：它们总是返回null。

有了这些基本的帮助函数，我们可以实现语法的第一部分：数字字面量。

## 基本表达式分析

我们从数字字面量开始，因为它们是最容易处理的。对于语法中的每个产品，我们将定义一个解析该产品的函数。对于数字字面量，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：它希望在当前标记为tok\_number标记时调用。它接受当前的数字值，创建NumberExprAST节点，将lexer前进到下一个标记，最后返回。

这有一些有趣的方面。最重要的一点是，这个例程会吃掉与生成相对应的所有标记，并返回lexer缓冲区，其中下一个标记（不是语法生成的一部分）准备就绪。这是递归下降解析器的一种相当标准的方法。更好的例子是，括号运算符的定义如下：

|  |
| --- |
| /// 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;  } |

此函数说明了有关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。调用此函数时，该函数预期当前标记为“（”标记，但在分析子表达式后，可能没有“）”等待。例如，如果用户键入“（4x”而不是“（4）”，解析器应该发出错误。因为错误可能发生，所以解析器需要一种方法来指示错误发生：在我们的解析器中，错误返回null。

2） 这个函数的另一个有趣的方面是，它通过调用ParseExpression使用递归（我们很快就会看到ParseExpress可以调用ParseParentExpr）。这很强大，因为它允许我们处理递归语法，并使每个生成非常简单。请注意，括号本身并不会导致AST节点的构造。虽然我们可以这样做，但圆括号最重要的作用是指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。

下一个简单的产品用于处理变量引用和函数调用：

|  |
| --- |
| /// identifierexpr  /// ::= 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 (1) {  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。（如果当前标记是tok\_identifier标记，则需要调用它）。它还具有递归和错误处理功能。其中一个有趣的方面是，它使用前瞻来确定当前标识符是独立变量引用还是函数调用表达式。它通过检查标识符后面的标记是否是“（”标记来处理此问题，并根据需要构造VariableExprAST或CallExprAST节点。

既然我们已经有了所有的简单表达式解析逻辑，我们可以定义一个helper函数来将其包装到一个入口点。我们将这类表达式称为“primary”表达式，其原因将在本教程的后面部分更加明确。为了解析任意主表达式，我们需要确定它是什么类型的表达式：

|  |
| --- |
| /// 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();  }  } |

既然您已经看到了这个函数的定义，那么我们为什么可以在各个函数中假设CurTok的状态就更加明显了。这使用前瞻性来确定要检查哪种类型的表达式，然后使用函数调用对其进行解析。

既然已经处理了基本表达式，我们需要处理二进制表达式。它们有点复杂。

## 二进制表达式分析

二进制表达式很难解析，因为它们常常是模糊的。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）”。对于数学中常见的定义，我们期望后面的解析，因为“\*”（乘法）的优先级高于“+”（加法）。

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运算符优先解析。这种解析技术使用二进制运算符的优先级来指导递归。首先，我们需要一个优先级映射表：

|  |
| --- |
| /// 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个二进制运算符（这显然可以由您扩展，我们勇敢无畏的读者）。GetTokPrecedence函数返回当前标记的优先级，如果标记不是二进制运算符，则返回-1。拥有映射可以很容易地添加新操作符，并清楚地表明算法不依赖于所涉及的特定操作符，但消除映射并在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）。

定义了上面的助手后，我们现在可以开始解析二进制表达式了。运算符优先解析的基本思想是将具有潜在二元运算符的表达式分解为多个部分。例如，考虑表达式“a+b+（c+d）\*e\*f+g”。运算符优先解析将其视为由二进制运算符分隔的主表达式流。因此，它将首先解析前导主表达式“a”，然后将看到对[+，b][+，（c+d）][\*，e][\*、f]和[+，g]。注意，因为括号是主表达式，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，后面可能跟有一系列[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。它具有优先权和指向迄今为止已解析的部件的表达式的指针。注意，“x”是一个完全有效的表达式：因此，“binoprhs”可以为空，在这种情况下，它返回传递给它的表达式。在上面的示例中，代码将“a”的表达式传递到ParseBinOpRHS中，当前标记为“+”。

传递给ParseBinOpRHS的优先级值表示函数允许使用的最小运算符优先级。例如，如果当前流对为[+，x]，并且ParseBinOpRHS以40的优先级传递，则不会使用任何令牌（因为“+”的优先级仅为20）。考虑到这一点，ParseBinOpRHS从以下方面开始：

|  |
| --- |
| /// binoprhs  /// ::= ('+' primary)\*  static std::unique\_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,  std::unique\_ptr<ExprAST> LHS) {  // If this is a binop, find its precedence.  while (1) {  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; |

此代码获取当前标记的优先级，并检查是否过低。因为我们定义了优先级为-1的无效令牌，所以此检查隐式地知道当令牌流用完二进制运算符时，配对流结束。如果此检查成功，我们知道令牌是一个二进制运算符，它将包含在此表达式中：

|  |
| --- |
| // 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，这段代码吃掉（并记住）二进制运算符，然后解析接下来的主表达式。这就构成了整对，第一个是运行示例中的[+，b]。

既然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，我们可以使用“（a+b）binop unparsed”或“a+（b binop unpassed）”。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binop的优先级低于或等于当前运算符的优先级，则我们知道括号关联为“（a+b）binop…”。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，这将把“a+b+”变成“（a+b）”，并执行循环的下一次迭代，以“+”作为当前标记。上面的代码将吃、记住并解析“（c+d）”作为主表达式，这将使当前对等于[+，（c+d）]。然后，它将评估上面的“if”条件，并将“\*”作为主项右侧的binop。在这种情况下，“\*”的优先级高于“+”的优先级，因此将输入if条件。

这里留下的关键问题是“if条件如何完全解析右侧”？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。这样做的代码非常简单（上面两个块中的代码为了上下文而重复）：

|  |
| --- |
| // 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.  } |

至此，我们知道主节点RHS的二进制运算符的优先级高于我们当前正在解析的binop。因此，我们知道，任何运算符的优先级都高于“+”的对序列都应该一起解析，并作为“RHS”返回。为此，我们递归调用ParseBinOpRHS函数，将“TokPrec+1”指定为继续执行所需的最低优先级。在上面的示例中，这将导致它将“（c+d）\*e\*f”的AST节点作为RHS返回，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，解析“+g”片段并将其添加到AST中。通过这一小段代码（14行非平凡的代码），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。这是对这段代码的一次旋风式浏览，有些微妙。我建议用几个棘手的例子来解释它，看看它是如何工作的。

这就完成了表达式的处理。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，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。

## 分析其余部分

接下来缺少的是函数原型的处理。在Kaleidoscope中，它们既用于“extern”函数声明，也用于函数体定义。这样做的代码是直接的，不太有趣（一旦你从表达式中幸存下来）：

|  |
| --- |
| /// prototype  /// ::= 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，只需一个原型加上一个表达式即可实现主体：

|  |
| --- |
| /// 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”声明“sin”和“cos”等函数，并支持向前声明用户函数。这些“外部”只是没有实体的原型：

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

最后，我们还将允许用户输入任意顶级表达式，并动态地对其求值。我们将为此定义匿名nullary（零参数）函数：

|  |
| --- |
| /// 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;  } |

现在我们已经完成了所有的部分，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！

## 驱动

此驱动程序只需使用顶级调度循环调用所有解析片段。这里没有什么有趣的，所以我只包含顶级循环。请参阅下面的“顶级解析”部分中的完整代码。

|  |
| --- |
| /// top ::= definition | external | expression | ';'  static void MainLoop() {  while (1) {  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;  }  }  } |

最有趣的是，我们忽略了顶级分号。你问，这是为什么？基本原因是，如果您在命令行中键入“4+5”，解析器不知道这是否是您要键入的内容的结尾。例如，在下一行可以键入“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”，以继续表达式。使用顶级分号可以输入“4+5；”，解析器会知道您已经完成了。

## 结论

使用不到400行注释代码（240行非注释、非空白代码），我们完全定义了我们的最小语言，包括lexer、解析器和AST构建器。完成后，可执行文件将验证万花筒代码，并告诉我们它是否语法无效。例如，下面是一个交互示例：

|  |
| --- |
| $ ./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，我们将描述如何从AST生成LLVM中间表示（IR）。

## 完整代码列表

下面是我们运行示例的完整代码列表。因为这使用了LLVM库，所以我们需要将它们链接进去。为此，我们使用llvm-config工具通知makefile/命令行要使用的选项：

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

下面是代码

|  |
| --- |
| #include <cctype>  #include <cstdio>  #include <cstdlib>  #include <map>  #include <memory>  #include <string>  #include <utility>  #include <vector>  //===----------------------------------------------------------------------===//  // Lexer  //===----------------------------------------------------------------------===//  // 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  /// gettok - Return the next token from standard input.  static int gettok() {  static int LastChar = ' ';  // Skip any whitespace.  while (isspace(LastChar))  LastChar = getchar();  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;  }  if (isdigit(LastChar) || LastChar == '.') { // Number: [0-9.]+  std::string NumStr;  do {  NumStr += LastChar;  LastChar = getchar();  } while (isdigit(LastChar) || LastChar == '.');  NumVal = strtod(NumStr.c\_str(), nullptr);  return tok\_number;  }  if (LastChar == '#') {  // Comment until end of line.  do  LastChar = getchar();  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');  if (LastChar != EOF)  return gettok();  }  // 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;  }  //===----------------------------------------------------------------------===//  // Abstract Syntax Tree (aka Parse Tree)  //===----------------------------------------------------------------------===//  namespace {  /// 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) {}  };  /// 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)) {}  };  /// 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)) {}  };  } // end anonymous namespace  //===----------------------------------------------------------------------===//  // Parser  //===----------------------------------------------------------------------===//  /// 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(); }  /// 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;  }  /// 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;  }  static std::unique\_ptr<ExprAST> ParseExpression();  /// numberexpr ::= number  static std::unique\_ptr<ExprAST> ParseNumberExpr() {  auto Result = std::make\_unique<NumberExprAST>(NumVal);  getNextToken(); // consume the number  return std::move(Result);  }  /// 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;  }  /// identifierexpr  /// ::= 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));  }  /// 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();  }  }  /// 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;  // 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;  // 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));  }  }  /// expression  /// ::= primary binoprhs  ///  static std::unique\_ptr<ExprAST> ParseExpression() {  auto LHS = ParsePrimary();  if (!LHS)  return nullptr;  return ParseBinOpRHS(0, std::move(LHS));  }  /// prototype  /// ::= 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");  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));  }  /// 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;  }  /// toplevelexpr ::= expression  static std::unique\_ptr<FunctionAST> ParseTopLevelExpr() {  if (auto E = ParseExpression()) {  // Make an anonymous proto.  auto Proto = std::make\_unique<PrototypeAST>("\_\_anon\_expr",  std::vector<std::string>());  return std::make\_unique<FunctionAST>(std::move(Proto), std::move(E));  }  return nullptr;  }  /// external ::= 'extern' prototype  static std::unique\_ptr<PrototypeAST> ParseExtern() {  getNextToken(); // eat extern.  return ParsePrototype();  }  //===----------------------------------------------------------------------===//  // Top-Level parsing  //===----------------------------------------------------------------------===//  static void HandleDefinition() {  if (ParseDefinition()) {  fprintf(stderr, "Parsed a function definition.\n");  } else {  // Skip token for error recovery.  getNextToken();  }  }  static void HandleExtern() {  if (ParseExtern()) {  fprintf(stderr, "Parsed an extern\n");  } else {  // Skip token for error recovery.  getNextToken();  }  }  static void HandleTopLevelExpression() {  // Evaluate a top-level expression into an anonymous function.  if (ParseTopLevelExpr()) {  fprintf(stderr, "Parsed a top-level expr\n");  } else {  // Skip token for error recovery.  getNextToken();  }  }  /// top ::= definition | external | expression | ';'  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;  }  }  }  //===----------------------------------------------------------------------===//  // Main driver code.  //===----------------------------------------------------------------------===//  int main() {  // 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();  // Run the main "interpreter loop" now.  MainLoop();  return 0;  } |

# 参考文档