链滴

Kaleidoscope 系列第二章：实现解析器和 AST

作者：Hanseltu

原文链接：https://ld246.com/article/1569985784270

来源网站：链滴

原文链接：Kaleidoscope系列第二章：实现解析器和AST

本文是使用LLVM开发新语言Kaleidoscope教程系列第二章，主要实现Kaleidoscope语言的语法解并生成AST的功能。

第二章简介

欢迎来到“使用LLVM开发新语言Kaleidoscope教程”教程的第二章。本章向我们展示如何使用第一章构建的词法分析器为我们的Kaleidoscope语言构建完整的解析器。有了解析器后，我们将定义并构建个抽象语法树（AST）。

我们将构建的解析器使用递归下降解析和运算符优先解析的组合来解析Kaleidoscope语言（后者用于进制表达式，前者用于其他所有内容）。在进行解析之前，让我们先讨论一下解析器的输出：抽象语树。

抽象语法树 (AST)

一段程序的抽象语法树很容易在接下来的阶段编译器（比如：代码生成阶段）翻译成机器码。我们通喜欢用一种对象来构建语言，毫无疑问，抽象语法树是最贴近我们要求的模型。我们将从表达式开始：

```
/// 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ExprAST基类的定义和一个用于数字文字的子类的定义。关于此代码，需要注意的要一点是NumberExprAST类将文字的数字值捕获为实例变量。这使编译器的后续阶段可以知道所存的数值是什么。

现在，我们仅创建AST，因此在它们上没有有用的方法。例如，添加一个虚拟方法来方便地打印代码以下是Kaleidoscope中其他的AST节点的定义：

```
/// 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)) {}
};
```

以上代码要做的事情非常简单：变量节点捕获变量名，二元运算符节点捕获其操作码（例如'+'），函调用节点捕获函数名以及任何参数表达式的列表。我们的AST优势是它捕获了语言功能，而没有关注言的具体语法细节。请注意，这里没有关于二元运算符，词法结构等的优先级的讨论。

对于我们的基本语言，这些都是我们将定义的所有表达节点。因为它没有条件控制流，所以它不是图完备的。我们将在以后的文章中解决。我们现在需要做两件事情，一件是实现在Kaleidoscope中调用数，另一件是记录函数体的本身。

```
/// 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，函数仅以其参数数量来输入。由于所有值都是双精度浮点数，因此每个参数的类都不需要存储在任何地方。在现代计算机语言语言中，ExprAST类应当有一个记录类型的变量。

有了以上类型后，我们就可以讨论在Kaleidoscope中如何解析表达式和函数体了。

解析器基础

现在我们有了AST，我们需要定义解析器代码来解析它。这里的想法是我们想要将类似 x + y（由词分析器作为三个标记返回）的内容解析为可以通过如下调用生成AST：

```
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));
```

为此，我们将从定义一些基本的辅助程序开始：

```
/// 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token缓冲区。这使我们可以预测词法分析器将返回什么。解器中的每个函数将假定CurTok是需要解析的当前token。

```
/// 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;
}
```

这些 LogError 部分是我们的解析器用来处理错误的简单程序。我们的解析器中的错误恢复并不是最的方法，并且不是对用户友好的方法，但是对于我们的教程而言，这已经足够了。这些错误处理例子我们处理具有各种返回类型的例程中的错误更加容易：它们始终返回null。

使用这些基本的辅助函数，我们便可以实现语法的第一部分：数字。

解析基本表达式

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

```
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = llvm::make_unique<NumberExprAST>(NumVal);
  getNextToken(); // consume the number
  return std::move(Result);
}
```

此例程非常简单：它期望在当前token是 tok_number token时被调用。它采用当前数字值，创建一个umberExprAST 节点，并将词法分析器移至下一个标记，最后返回。

这有一些有趣的方面。最重要的是，该例程将删去与该生成物相对应的所有token，并返回准备好下个token（不属于语法生成的一部分）的词法分析器缓冲区。这是用于递归下降解析器的相当标准的法。举一个更好的例子，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几个有意思的地方：

1）它显示了我们如何使用LogError例程。调用此函数时，该函数期望当前标记是一个 (标记，但是解析该子表达式之后，可能没有) 等待。例如，如果用户键入 (4x 而不是 (4)，则解析器将发出错误因为可能发生错误，所以解析器需要一种指示错误发生的方法：在我们的解析器中，我们返回错误时null。

2）该函数的另一个有意思的方面是它通过调用使用递归 ParseExpression（我们很快就会看到 Parse xpression 可以调用 ParseParenExpr ）。它之所以强大是因为它使我们能够处理递归语法，并使每产生式都非常简单。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，但是括号最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。

下一个简单的过程是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 llvm::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 llvm::make_unique<CallExprAST>(IdName, std::move(Args));
}
```

该例程遵循与其他例程相同的样式。（如果当前token是 tok_identifier 令牌，则期望被调用）。它具有递归和错误处理功能。一个有趣的方面是，它使用 超前 方法判断来确定当前标识符是独立变量用还是函数调用表达式。它通过检查标识符后的token是否为 (token，并根据情况构造一个 VariableExprAST 或 CallExprAST 节点来处理此问题。

现在，我们已经有了所有简单的表达式解析逻辑，我们可以定义一个辅助函数，将其封装以便调用。们将此类表达式称为“基本”表达式，其原因在本教程的后面部分将变得更加清楚。为了解析任意表式，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();
  }
}
```

通过基本表达式解析器，我们可以明白为什么我们要使用 CurTok 了，这里用了前置判断来选择并调解析器。

现在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。

解析二元表达式

二进制表达式通常很难确定其实际意义，因此很难解析。例如，当给定字符串 x + y * z 时，解析器以选择将其解析为 (x + y) * z 或 x + (y * z) 。使用数学上的通用定义，我们希望是按照后面这种进解析，因为 * （乘法）的 优先级 高于 + （加法）的 优先级。

有很多方法可以解决此问题，但是一种优雅而有效的方法是使用 Operator-Precedence Parsing。解析技术使用二元运算符的优先级来选择运算顺序。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，我们将仅支持4个元运算符（当然我们可以随意扩展）。以上 GetTokPrecedence 函数返回当前token的优先级；如果token不是元运算符，则返回-1。有了映射可以轻松加新的运算符，并且可以清楚地表明该算法不依赖于所涉及的特定运算符，消除映射并在 GetTokPrecedence 函数中进行比较将很容易。（或者只使用固定大小的数组）。

有了上面定义的帮助程序，我们现在就可以开始分析二元表达式了。运算符优先级解析的基本思想是具有潜在歧义的二元运算符的表达式分解为多个部分。考虑例如表达式 a + b + (c + d) * e * f + g。算符优先级解析将其视为由二元运算符分隔的主表达式流。这样，它将首先解析前导主表达式 a，然将看到对 [+, b] [+, (c + d)] [, e] [, f] 和 [+, g]。请注意，因为括号是基础表达式，所以二元表达式析器根本不需要担心像（c + d）这样的嵌套子表达式。

首先，一个表达式是一个主表达式，其后可能是[binop，primaryexpr]对的序列：

```
/// expression
///   ::= primary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  if (!LHS)
    return nullptr;

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

ParseBinOpRHS 是为我们解析配对序列的函数。它具有一个优先级和一个指向到目前为止已解析的分的表达式的指针。请注意，x 是一个完全有效的表达式：因此，binoprhs 允许为空，在这种情况下它将返回传递给它的表达式。在上面的示例中，代码将 a 的表达式传递到其中 ParseBinOpRHS，当标记为 +。

传递的优先级值ParseBinOpRHS指示允许该函数使用的_最小运算符优先级_。例如，如果当前对流为+，x]并 ParseBinOpRHS 以40的优先级传递，则它将不删去任何token（因为“ +”的优先级仅为2）。考虑到这一点，ParseBinOpRHS从以下内容开始：

```
/// 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token的优先级，并与传入的优先级进行比较。因为我们将无效token定义为优先级为1，它比任何一个运算符的优先级都小，我们可以借助它来获知二元表达式已经结束。若当前的token运算符，我们继续：

```
// 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;
```

这样，此代码将删除（先记住）二元运算符，然后解析随后的主表达式。这样就构成了整个对，对于在运行的示例，第一对是[+，b]。

现在，我们解析了表达式的左侧和一对RHS序列，现在我们必须确定表达式关联的方式。特别是，我可以设置为 (a + b) binop unparsed 或 a + (b binop unparsed) 。为了确定这一点，我们先看 binop 以确定其优先级，并将其与之前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，当前运算符为 +，下一个运算符为 +，我们知道它们具有相同的优先级。在种情况下，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.
}
```

在上面的示例中，这会将 a + b + 变成 (a + b) 并执行循环的下一个迭代，并以 + 作为当前标记。面的代码记录下来。接下来解析 (c + d) 作为基础表达式，这使得当前对等于[+, (c + d)]。然后，它使用 * 作为主要对象右侧的binop评估上面的 if 条件。在这种情况下，* 的优先级高于 + 的优先级，此将输入if条件。

这里剩下的关键问题是“如果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，需将所有 (c + d) * e * f 作为RHS表达变量。做到这一点的代码非常简单（上面两个块中的代码重复了下文）：

```
    // 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，我们知道运符优先级均高于 + 的任何对序列都应一起解析，并作为 RHS 返回。为此，我们递归调用 ParseBinORHS 指定 TokPrec + 1 的函数作为继续运行所需的最低优先级。在上面的示例中，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 段被解析并添加到AST中。有了这段代码（14行平凡的代），我们就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。这是这段代码的精妙处。我们建议再通过一些复杂的例子来贯穿它，以了解其工作原理。

以上结束了对二元表达式的处理。此时，我们可以将解析器指向任意token流，并根据该token流构一个表达式，并在不属于该表达式的第一个token处停止。接下来，我们需要处理函数定义等。

解析其他部分

接下来缺少的是函数原型的处理。在Kaleidoscope中，这些用于“外部”函数声明以及函数主体定义执行此操作的代码简单明了，而且不是很有趣（一旦我们保留了表达式，就可以直接处理了）：

```
/// 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，只需一个原型以及一个用于实现主体的表达式即可：

```
/// 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;
}
```

另外，我们支持 'extern' 来声明诸如 'sin' 和 'cos' 之类的函数，并支持用户函数的正向声明。这些 extern 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，我们将让用户输入任意的外层表达式（top-level expressions），在运行的同时会计算出表达结果。为此，我们需要处理无参数函数：

```
/// 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工作，接下来我们构建一个小驱动程序，它将使我们能够实际执行我们目为止已构建的代码！

驱动代码

该驱动程序仅通过顶级调度循环调用所有解析块。这里没有什么注意的，所以我只包括顶级循环。请阅以下第二章完整代码清单的“top-level expressions”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;
    }
  }
}
```

最有趣的部分是我们忽略了顶级分号。你问为什么呢？根本原因是，如果您在命令行中键入 4 + 5 ，解析器将不知道这是否是您要键入的内容的结尾。例如，在下一行中，您可以键入 def foo... ，在这情况下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。使用顶级分号可以使键入 4 + 5; ，解析器将知道已完成输入。

小结

通过不到400行的注释代码（240行非注释，非空白代码），我们完全定义了最小语言，包括词法分器，解析器和AST构建器。完成此操作后，可执行文件将验证Kaleidoscope代码，并告诉我们其语法否无效。例如，这是一个交互示例：

```
$ ./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
$
```

这里有很多扩展空间。我们可以定义新的AST节点，以多种方式扩展语言，等等。在下一部分: 第三生成LLVM中间代码IR中，我们将描述如何从AST生成LLVM中间表示（IR）。

完整代码清单

特别注意: 官方给的例子有些问题，运行出错了，请参考以下运行参数和程序进行编译和运行并演示。

这是我们运行示例的完整代码清单。因为这使用了LLVM库，所以我们需要将它们链接起来。为此，们使用llvm-config工具通知makefile /命令行有关要使用哪些选项的信息：

```
# Compile
clang++ -g -O3 chapter2-Implementing-a-Parser-and-AST.cpp `llvm-config --cxxflags --syst
m-libs --libs`
# Run
./a.out
```

以下是本章节代码：

chapter2-Implementing-a-Parser-and-AST.cpp

参考: Kaleidoscope: Implementing a Parser and AST