链滴

Kaleidoscope 系列第一章：新语言特性和 Lexer

作者：Hanseltu

原文链接：https://ld246.com/article/1569942534419

来源网站：链滴

原文链接 Kaleidoscope系列第一章：新语言特性和Lexer

本文是 [使用LLVM开发新语言Kaleidoscope教程] (https://www.tuhaoxin.cn/articles/2019/10/01/569927157476.html) 系列第一章，主要介绍Kaleidoscope语言特性和词法分析器的构建。

Kaleidoscope语言特性

本教程以一种名为“Kaleidoscope”（google翻译为万花筒，源自“美丽，形式和视野”）的玩具言进行开发。Kaleidoscope是一种过程语言，可让我们轻松定义函数，使用条件语句，数学表达式等在本教程中，我们将扩展Kaleidoscope以支持 if/then/else语句，for循环，用户定义运算符，支持用JIT进行简单的命令行界面编译、调试等。

再次说明，我们希望使设计语言保持简单，因此Kaleidoscope中唯一的数据类型是64位浮点类型（在语言中为“ double”）。这样，所有值都隐式地具有双精度，并且该语言不需要类型声明。这为该言提供了一种非常不错且简单的语法。例如，以下简单示例计算斐波纳契数：

```
# 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，因为LLVM JIT使得此操作非常容易。这意味着我们可以使用函数之前使用'extern'关键字定义一个函数（这对于相互递归的函数也很有用）。例如：

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(.4), cos(42))
```

第6章中提供了一个更有趣的示例，其中我们编写了一个迷你Kaleidoscope应用程序，该应用程序以同的放大倍数显示Mandelbrot集。

接下来我们开始深入探讨这种语言的实现。

词法分析器

在实现语言方面，首先需要的是处理文本文件并识别其内容。传统方法是使用“词法分析器”（又称扫描器”）将输入分解为“token”。词法分析器返回的每个token都包含token代码和潜在的一些元据（例如数字的数值等）。首先，我们定义以下token：

```
// 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token要么是Token枚举类型中的某值之一，要么是“未知”字符（如“+”），并以其ASCII值返回。如果当前token是标识符，则 IdentifierStr 全局变量将保存标识符的名。如果当前标记是数字（如1.0），则 NumVal 保留其值。为了简单起见，我们使用全局变量，但这是真正的语言实现的最佳选择。

词法分析器实际由 gettok 的函数实现。gettok 调用该函数以从标准输入返回下一个标记。其定义开于：

```
/// 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中 getchar() 函数从标准输入一次读取一个字符来工作。它在识别到它们后就删除们，并将最后读取但未处理的字符存储在LastChar中。它要做的第一件事是忽略token之间的空格。功能主要由while循环完成。

接下来 gettok 要做的是识别标识符和特定的关键字，例如 “def”。Kaleidoscope通过以下简单循完成此操作：

```
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 IdentifierStr 对标识符进行词法化时都会设置成全局值。另外，由于语言关键字是同一循环匹配的，因此我们在此对它们进行内联处理。对于处理数值也是类似的：

```
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;
```

```
}
```

这是用于处理输入的非常简单的代码。从输入读取数值时，我们使用C中 strtod 函数将其转换为存储中的数值 NumVal。请注意，这并没有进行足够的错误检查：它将错误地读取“1.23.45.67”，并像理“1.23”一样处理它。当然，我们可以随意更改它！

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();
}
```

我们通过跳到行尾来处理注释，然后返回下一个标记。最后，如果输入与以上情况之一不匹配，则该入可能是运算符，例如“+”，或者是文件结尾。这些使用以下代码处理：

```
  // 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，我们就拥有了用于基本Kaleidoscop语言的完整词法分析器（该词法分析的完整代码清单可在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。当我们了它时，我们将包括一个驱动程序，以便我们可以同时使用lexer和解析器。

参考：Kaleidoscope Introduction and the Lexer