

让我们构建一个简单的解释器。第 5 部分。 (<https://ruslanspivak.com/lbasi-part5/>)

日期 2015 年 10 月 14 日, 星期三

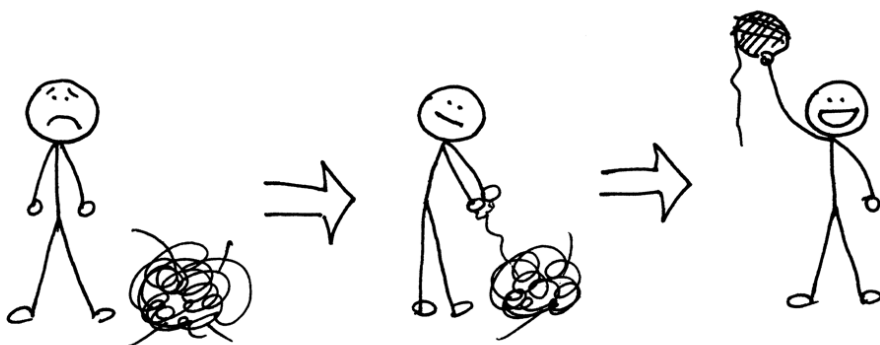
你如何处理像理解如何创建解释器或编译器这样复杂的事情？一开始，这一切看起来都像是一团乱七八糟的纱线，您需要将其解开才能获得完美的球。

到达那里的方法是将它解开一根线，一次解开一个结。但是，有时您可能会觉得自己没有立即理解某些内容，但您必须继续前进。如果你足够坚持，它最终会“咔哒”一声，我向你保证（哎呀，如果我每次不明白某事时我就留出 25 美分，我很久以前就会变得富有了：））。

在理解如何创建解释器和编译器的过程中，我能给你的最好建议之一就是阅读文章中的解释，阅读代码，然后自己编写代码，甚至将相同的代码编写几个一段时间后，使材料和代码对您感觉自然，然后才能继续学习新主题。不要着急，放慢脚步，花时间深入了解基本思想。这种方法虽然看似缓慢，但会在未来取得成效。相信我。

你最终会得到完美的毛线球。而且，你知道吗？即使它不是那么完美，它仍然比替代方案更好，即什么都不做，不学习主题或快速浏览并在几天内忘记它。

记住 - 继续解开：一个线程，一次一个结，通过编写代码来练习你学到的东西，很多：



今天，您将使用从本系列前几篇文章中获得的所有知识，并学习如何解析和解释具有任意数量的加法、减法、乘法和除法运算符的算术表达式。您将编写一个解释器，该解释器将能够计算诸如“ $14 + 2 * 3 - 6 / 2$ ”之类的表达式。

在深入研究和编写一些代码之前，让我们先谈谈运算符的**结合性**和**优先级**。

按照惯例， $7 + 3 + 1$ 与 $(7 + 3) + 1$ 相同，并且 $7 - 3 - 1$ 相当于 $(7 - 3) - 1$ 。这里没有意外。我们都在某个时候了解到这一点，并且从那时起就认为这是理所当然的。如果我们将 $7 - 3 - 1$ 视为 $7 - (3 - 1)$ ，结果将是意外的 5 而不是预期的 3。

在普通算术和大多数编程语言中，加法、减法、乘法和除法都是左结合的：

$7 + 3 + 1$ 等价于 $(7 + 3) + 1$
 $7 - 3 - 1$ 等价于 $(7 - 3) - 1$
 $8 * 4 * 2$ 相当于 $(8 * 4) * 2$
 $8 / 4 / 2$ 等价于 $(8 / 4) / 2$

运算符的左关联意味着什么？

当表达式 $7 + 3 + 1$ 中像 3 这样的操作数两边都有加号时，我们需要一个约定来决定哪个运算符适用于 3。它是操作数 3 的左侧还是右侧？运算符 + 与左侧相关联，因为两边都有加号的操作数属于其左侧的运算符，因此我们说运算符 + 是左结合的。这就是为什么 $7 + 3 + 1$ 根据结合性约定等价于 $(7 + 3) + 1$ 。

好的，对于像 $7 + 5 * 2$ 这样的表达式，我们在操作数 5 的两边都有不同类型的运算符呢？表达式等价于 $7 + (5 * 2)$ 还是 $(7 + 5) * 2$ ？我们如何解决这种歧义？

在这种情况下，结合性约定对我们没有帮助，因为它仅适用于一种运算符，即加法 (+, -) 或乘法 (*, /)。当我们在同一表达式中有不同类型的运算符时，我们需要另一种约定来解决歧义。我们需要一个约定来定义运算符的相对优先级。

它是这样的：我们说如果运算符 * 在 + 之前接受其操作数，那么它具有更高的优先级。在我们所知道和使用的算术中，乘法和除法的优先级高于加法和减法。因此，表达式 $7 + 5 * 2$ 等价于 $7 + (5 * 2)$ ，而表达式 $7 - 8 / 4$ 等价于 $7 - (8 / 4)$ 。

在我们有一个具有相同优先级的运算符的表达式的情况下，我们只使用结合性约定并从左到右执行运算符：

$7 + 3 - 1$ 等价于 $(7 + 3) - 1$
 $8 / 4 * 2$ 相当于 $(8 / 4) * 2$

我希望你不会认为我想通过谈论运算符的结合性和优先级来让你厌烦。这些约定的好处是我们可以从一个表中构造算术表达式的语法，该表显示算术运算符的结合性和优先级。然后，我们可以按照我在第 4 部分中 (<http://ruslanspivak.com/lbasi-part4/>) 概述的准则将语法翻译成代码，我们的解释器将能够处理除结合性之外的运算符的优先级。

好的，这是我们的优先级表：

higher precedence ↓	precedence level	associativity	operators
	2	left	+, -
	1	left	*, /

从表中，您可以看出运算符 + 和 - 具有相同的优先级，并且它们都是左关联的。您还可以看到运算符 * 和 / 也是左结合的，它们之间具有相同的优先级，但比加法和减法运算符具有更高的优先级。

以下是如何从优先级表构造语法规则：

1. 为每个优先级定义一个非终结符。非终结符的产生式主体应包含该级别的算术运算符和下一个更高优先级的非终结符。
2. 为基本的表达单位创建一个额外的非终结因子，在我们的例子中是整数。一般规则是，如果您有 N 个优先级，则总共需要 N + 1 个非终结符：每个级别一个非终结符加上一个基本表达式单位的非终结符。

向前！

让我们遵循规则并构建我们的语法。

根据规则1，我们将定义两个非端子：非末端称为EXPR 2级和非末端称为术语为1级，并按照规则2我们将定义一个因子非终端的运算单元的基本表达式，整数。

在开始符号我们的新语法将EXPR和expr的生产将包含代表从2级使用运营商的身上，这对我们来说是运营商 + 和 -，并且将包含项非端子的一级优先级，级别 1：

$expr : term (PLUS | MINUS) term *$

该术语生产将有一个代表从1级，这是操作员使用操作员的身体 * 和 /，并将包含非末端因子表达，整数的基本单位：

$\text{term} : \text{factor} ((\text{MUL} | \text{DIV}) \text{factor})^*$

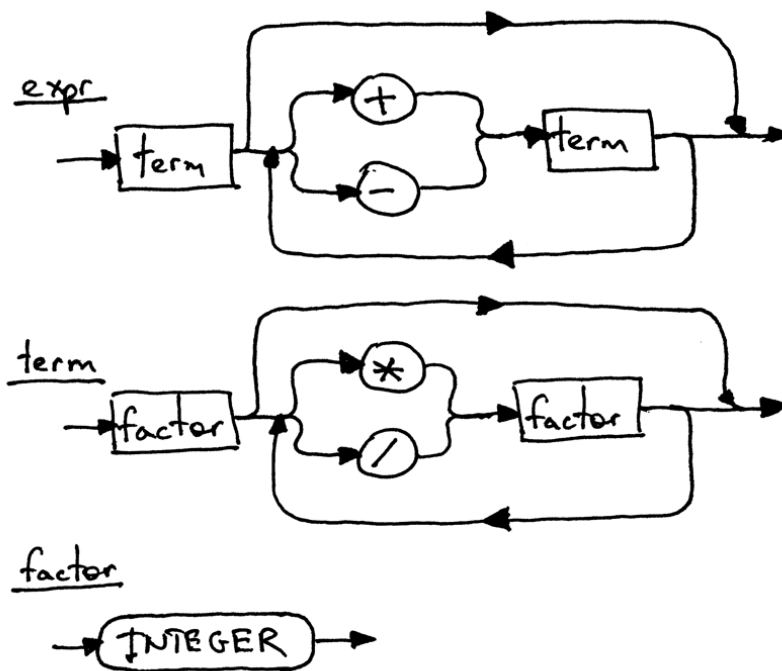
非终端因子的产生将是：

$\text{factor} : \text{INTEGER}$

您已经在前面的文章中将上述产生式视为语法和语法图的一部分，但在这里我们将它们组合成一个语法来处理结合性和运算符的优先级：

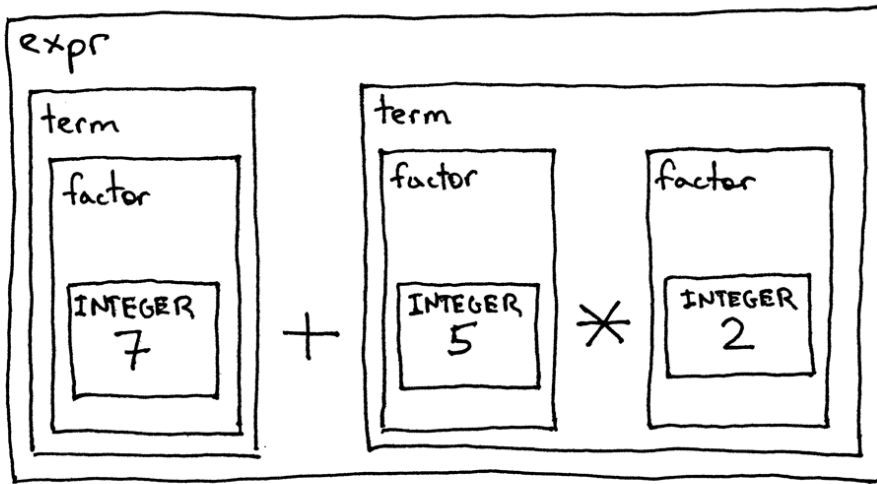
$\text{expr} : \text{term} ((\text{PLUS} | \text{MINUS}) \text{term})^*$
 $\text{term} : \text{factor} ((\text{MUL} | \text{DIV}) \text{factor})^*$
 $\text{factor} : \text{INTEGER}$

这是对应于上述语法的语法图：



图中的每个矩形框都是对另一个图的“方法调用”。如果您采用表达式 $7 + 5 * 2$ 并从最上面的图表 expr 开始，一直走到最底部的图表 factor ，您应该能够看到下图中的高优先级运算符 $*$ 和 $/$ 在运算符 $+$ 之前执行和 $-$ 在更高的图表中。

为了推动运算符的优先级，让我们看一下根据上面的语法和语法图完成的相同算术表达式 $7 + 5 * 2$ 的分解。这只是表明优先级较高的运算符在优先级较低的运算符之前执行的另一种方式：



好的，让我们按照第 4 部分的 (<http://ruslanspivak.com/lbasi-part4/>)指南将语法转换为代码，看看我们的新解释器是如何工作的，好吗？

这里又是语法：

```

expr   : term (PLUS | MINUS) term) *
term   : factor ((MUL | DIV) factor) *
factor : INTEGER
  
```

这是一个计算器的完整代码，它可以处理包含整数和任意数量的加法、减法、乘法和除法运算符的有效算术表达式。

以下是与第 4 部分中 (<http://ruslanspivak.com/lbasi-part4/>)的代码相比的主要变化：

- 该词法分析器类现在可以记号化 +, -, *, 和 / (这里没有什么新，我们只是结合的编码从之前的文章中为一类，它支持所有的令牌)
- 回想一下，在语法中定义的每个规则（产生式）**R**成为具有相同名称的方法，并且对该规则的引用成为方法调用：**R()**。因此，Interpreter类现在具有三个对应于语法中非终结符的方法：expr、term和factor。

源代码：

```

# 标记类型
#
# EOF (end-of-file) 标记用于表示
# 没有更多的输入可供词法分析使用
INTEGER , PLUS , MINUS , MUL , DIV , EOF = (
    'INTEGER' , 'PLUS' , '减' , 'MUL' , 'DIV' , 'EOF'
)

class Token ( object ):
    def __init__ ( self , type , value ):
        # token type: INTEGER, PLUS, MINUS, MUL, DIV, or EOF
        self . type = type
        # 标记值: 非负整数值、'+'、'-'、'*'、'/' 或 None
        self . value = value

    def __str__ ( self ):
        """类实例的字符串表示。

        示例:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})' . format (
            type = self . type ,
            value =代表 (自我。价值)
        )

    def __repr__ ( self ):
        返回 self . __str__ ()

class Lexer ( object ):
    def __init__ ( self , text ):
        # 客户端字符串输入, 例如"3 * 5"、"12 / 3 * 4"等
        self . text = text
        # self.pos 是 self.text
        self的索引。pos = 0
        自我。current_char = self . 文本[自我。位置]

    def 错误 (自我):
        引发 异常 ( '无效字符' )

    DEF 提前 (自):
        """ , "提前`pos`指针, 并设置`current_char`变量""
        自我。pos += 1
        如果 self . pos > len ( self . text ) - 1 :
            self . current_char = None # 表示输入结束
        else :
            self . current_char = self . 文本[自我。位置]

    def skip_whitespace ( self ):
        while self . current_char 是 不 无 和 自我。current_char . isspace ():
            self . 提前()

    def integer ( self ):
        """返回从输入中消耗的 (多位) 整数。"""
        result = ''
        while self . current_char 是 不 无 和 自我。current_char . isdigit ():
            结果 += self . current_char
            自我。提前 ()
        返回 整数 (结果)

    def get_next_token ( self ):

```

"""词法分析器（也称为扫描器或分词器）

此方法负责将句子
分解为标记。一次一个令牌。

"""

而自我。current_char 是无:

```

    如果 自。current_char。isspace():
        self.skip_whitespace()
        继续

    如果 自。current_char。ISDIGIT():
        返回 令牌(INTEGER, 自我。整型())

    如果 自。current_char == '+':
        self.提前()
        返回 令牌(PLUS, '+')

    如果 自。current_char == '-':
        self.提前()
        返回 令牌(MINUS, '-')

    如果 自。current_char == '*':
        self.提前()
        返回 令牌(MUL, '*')

    如果 自。current_char == '/':
        self.提前()
        返回 令牌(DIV, '/')

    自我。错误()

    返回 令牌(EOF, 无)

```

```

class Interpreter ( object ):
    def __init__ ( self , lexer ):
        self.lexer = lexer
        # 将当前标记设置为从输入
        self 中获取的第一个标记。current_token = self。词法分析器。get_next_token ()

    def 错误 (自我):
        引发 异常 ('无效语法' )

    def eat ( self , token_type ):
        # 比较当前标记类型与传递的标记
        # 类型, 如果它们匹配, 则“吃”当前标记
        # 并将下一个标记分配给 self.current_token,
        # 否则引发异常。
        如果 自。current_token。type == token_type:
            self。current_token = self。词法分析器。get_next_token ()
        其他:
            自我。错误()

    def factor ( self ):
        """factor : INTEGER"""
        token = self。current_token
        自我。吃 (整数)
        返回 令牌。价值

    def term ( self ):
        """term : factor ((MUL | DIV) factor)"""
        result = self。因子()

        而 自我。current_token。键入 在 (MUL, DIV):
            令牌 = 自我。current_token

```

```

如果是 token 。类型 == MUL :
    self 。吃 (MUL )
    结果 = 结果 * 自我。因子()
elif 令牌。类型 == DIV :
    自我。吃 (DIV )
    结果 = 结果 / 自我。因子()

```

返回 结果

```

def expr ( self ):
    """算术表达式解析器/解释器。

```

```

计算> 14 + 2 * 3 - 6 / 2
17

```

```

expr : term ((PLUS | MINUS) term)*
term : factor ((MUL | DIV) factor)*
factor : INTEGER
"""
result = self . term ()

```

而 自我。current_token 。键入 在 (PLUS , MINUS) :

```

令牌 = 自我。current_token
如果是 token 。类型 == PLUS :
    自我。吃 (PLUS )
    结果 = 导致 + 自我。term ()
elif 令牌。类型 == 减号:
    自我。吃 (减)
    结果 = 结果 - 自我。术语()

```

返回 结果

```

def main ():
    while True :
        try :
            # 在 Python3 下运行替换 'raw_input' call
            # with 'input'
            text = raw_input ( 'calc> ' )
        except EOFError :
            break
        if not text :
            continue
        lexer = Lexer ( text )
        interpreter = 解释器 (词法分析器)
        结果 = 解释器。expr ()
        打印 (结果)

```

```

如果 __name__ == '__main__' :
    main ()

```

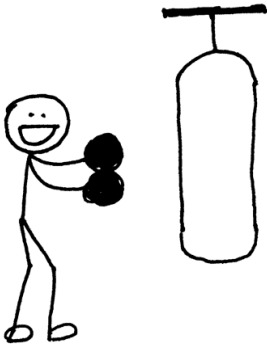
将上述代码保存到calc5.py文件中或直接从GitHub

(<https://github.com/rspivak/lbasi/blob/master/part5/calc5.py>)下载。像往常一样，亲自尝试一下，看看解释器是否正确评估了具有不同优先级的运算符的算术表达式。

这是我的笔记本电脑上的示例会话：

```
$ python calc5.py
计算> 3
3
计算> 2 + 7 * 4
30
计算> 7 - 8 / 4
5
计算> 14 + 2 * 3 - 6 / 2
17
```

以下是今天的新练习：



- 按照本文中的描述编写一个解释器，无需查看文章中的代码。为您的解释器编写一些测试，并确保它们通过。
- 扩展解释器以处理包含括号的算术表达式，以便您的解释器可以评估深度嵌套的算术表达式，例如： $7 + 3 * (10 / (12 / (3 + 1) - 1))$

检查你的理解。

1. 运算符的左关联意味着什么？
2. 运算符 + 和 - 是左关联还是右关联？* 和 / 呢？
3. 运算符 + 是否比运算符 * 具有更高的优先级？

嘿嘿，你一直读到最后！那真是太棒了。下次我会带着新文章回来 - 敬请期待，变得精彩，并且像往常一样，不要忘记做练习。

以下是我推荐的书籍清单，它们将帮助您学习解释器和编译器：

1. 语言实现模式：创建您自己的特定领域和通用编程语言（实用程序员）
(http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)
2. 编写编译器和解释器：一种软件工程方法
(http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)
3. Java 中的现代编译器实现 (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)
4. 现代编译器设计 (http://www.amazon.com/gp/product/1461446988/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1461446988&linkCode=as2&tag=russblo0b-20&linkId=PAXWJP5WCPZ7RKRD)

5. 编译器：原理、技术和工具（第 2 版）

(http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=GOEGDQG4HIHU56FQ)

如果您想在收件箱中获取我的最新文章，请在下方输入您的电子邮件地址，然后单击“获取更新”！

输入您的名字 *

输入您最好的电子邮件 *

获取更新!

本系列所有文章：

- [让我们构建一个简单的解释器。第1部分。 \(/lsbasi-part1/\)](#)
- [让我们构建一个简单的解释器。第2部分。 \(/lsbasi-part2/\)](#)
- [让我们构建一个简单的解释器。第 3 部分。 \(/lsbasi-part3/\)](#)
- [让我们构建一个简单的解释器。第 4 部分。 \(/lsbasi-part4/\)](#)
- [让我们构建一个简单的解释器。第 5 部分。 \(/lsbasi-part5/\)](#)
- [让我们构建一个简单的解释器。第 6 部分。 \(/lsbasi-part6/\)](#)
- [让我们构建一个简单的解释器。第 7 部分：抽象语法树 \(/lsbasi-part7/\)](#)
- [让我们构建一个简单的解释器。第 8 部分。 \(/lsbasi-part8/\)](#)
- [让我们构建一个简单的解释器。第 9 部分。 \(/lsbasi-part9/\)](#)
- [让我们构建一个简单的解释器。第 10 部分。 \(/lsbasi-part10/\)](#)
- [让我们构建一个简单的解释器。第 11 部分。 \(/lsbasi-part11/\)](#)
- [让我们构建一个简单的解释器。第 12 部分。 \(/lsbasi-part12/\)](#)
- [让我们构建一个简单的解释器。第 13 部分：语义分析 \(/lsbasi-part13/\)](#)
- [让我们构建一个简单的解释器。第 14 部分：嵌套作用域和源到源编译器 \(/lsbasi-part14/\)](#)
- [让我们构建一个简单的解释器。第 15 部分。 \(/lsbasi-part15/\)](#)
- [让我们构建一个简单的解释器。第 16 部分：识别过程调用 \(/lsbasi-part16/\)](#)
- [让我们构建一个简单的解释器。第 17 部分：调用堆栈和激活记录 \(/lsbasi-part17/\)](#)
- [让我们构建一个简单的解释器。第 18 部分：执行过程调用 \(/lsbasi-part18/\)](#)
- [让我们构建一个简单的解释器。第 19 部分：嵌套过程调用 \(/lsbasi-part19/\)](#)

注释

ALSO ON RUSLAN'S BLOG

Let's Build A Simple Interpreter. Part 19: ...

2 years ago • 24 comments

What I cannot create, I do not understand. — Richard Feynman

EOF is not a character

2 years ago • 16 comments

I was reading Computer Systems: A Programmer's Perspective the other day ...

Let's Build A Simple Interpreter. Part 3.


6 years ago • 15 comments

I woke up this morning and I thought to myself: "Why do we find it so difficult to ...

Let's B Server.

6 years a

"We learn have to i Part 2 yc




Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name



Joe MB • 6 years ago

Best series of tutorials on the whole internet. The draws helped a lot on making it easier to read, also it's great the way you don't leave any concept without a proper explanation. Thanks a lot, I'll stay tuned for future posts.

2 ^ | v • Reply • Share ›

社会的

- github (<https://github.com/rspivak/>)
- 推特 (<https://twitter.com/rspivak>)
- 链接 (<https://linkedin.com/in/ruslanspivak/>)

热门帖子

- 让我们构建一个 Web 服务器。第1部分。 (<https://ruslanspivak.com/lsbaws-part1/>)
- 让我们构建一个简单的解释器。第1部分。 (<https://ruslanspivak.com/lsbasi-part1/>)
- 让我们构建一个 Web 服务器。第2部分。 (<https://ruslanspivak.com/lsbaws-part2/>)
- 让我们构建一个 Web 服务器。第 3 部分。 (<https://ruslanspivak.com/lsbaws-part3/>)
- 让我们构建一个简单的解释器。第2部分。 (<https://ruslanspivak.com/lsbasi-part2/>)

免责声明

这个网站上的一些链接有我的亚马逊推荐 ID，它为我提供了每次销售的小额佣金。感谢您的支持。