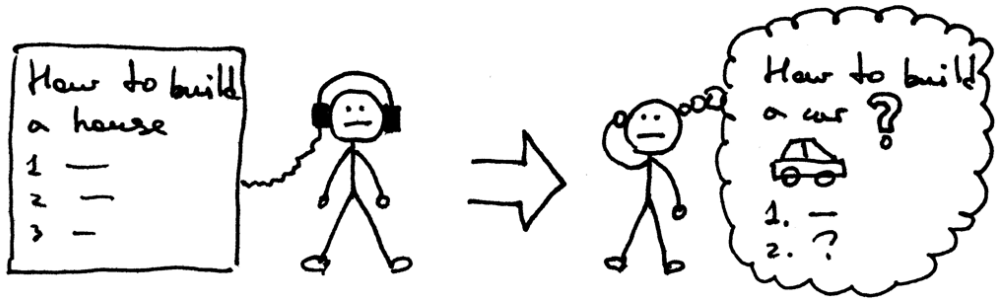
让我们构建一个简单的解释器。第 4 部分。(https://ruslanspivak.com/lsbasi-part4/)

日期 2015 年 9 月 11 日星期五

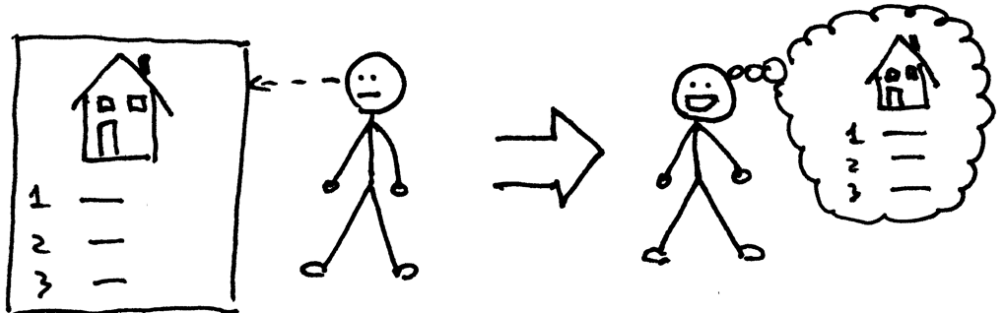
你是被动学习这些文章中的材料还是主动练习？我希望你一直在积极练习。我真的这样做:)

还记得孔子的话吗？

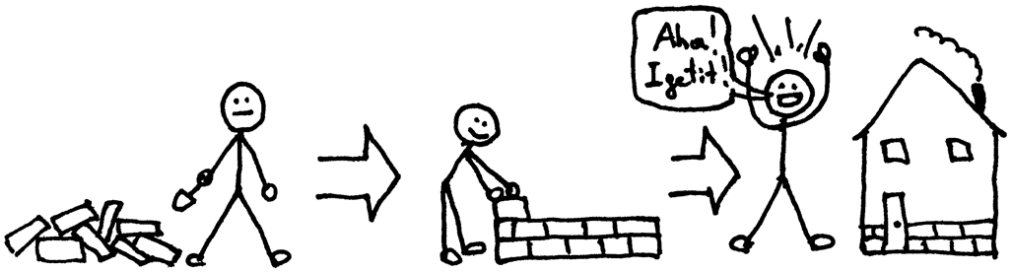
> “我听到了，我忘记了。”



> “我看到了，我记得。”



> “我知道，我明白。”



在上一篇文章中，您学习了如何解析（识别）和解释包含任意数量的加号或减号运算符的算术表达式，例如“7 - 3 + 2 - 1”。您还了解了语法图以及如何使用它们来指定编程语言的语法。

今天，您将学习如何解析和解释包含任意数量的乘法和除法运算符的算术表达式，例如“7 * 4 / 2 * 3”。本文中的除法将是一个整数除法，所以如果表达式是“9 / 4”，那么答案将是一个整数：2。

今天我还将讨论另一种广泛使用的用于指定编程语言语法的符号。它被称为**上下文无关文法**（简称为**文法**）或**BNF**（巴科斯-诺尔形式）。出于本文的目的，我不会使用纯BNF (https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)表示法，而是更像是一种修改后的EBNF (https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form) 表示法。

以下是使用语法的几个原因：

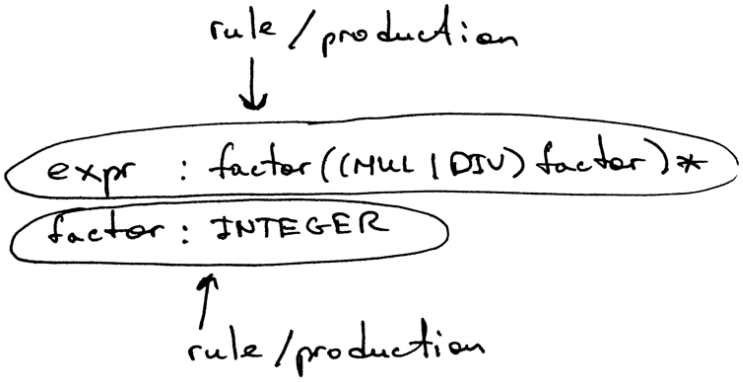
1. 语法以简洁的方式指定了编程语言的语法。与语法图不同，语法非常紧凑。在以后的文章中，您会看到我越来越多地使用语法。
2. 语法可以作为很好的文档。
3. 即使您从头开始手动编写解析器，语法也是一个很好的起点。通常，您只需遵循一组简单的规则即可将语法转换为代码。
4. 有一组工具，称为解析器生成器，它们接受语法作为输入并根据该语法自动为您生成解析器。我将在本系列的后面部分讨论这些工具。

现在，让我们谈谈语法的机械方面，好吗？

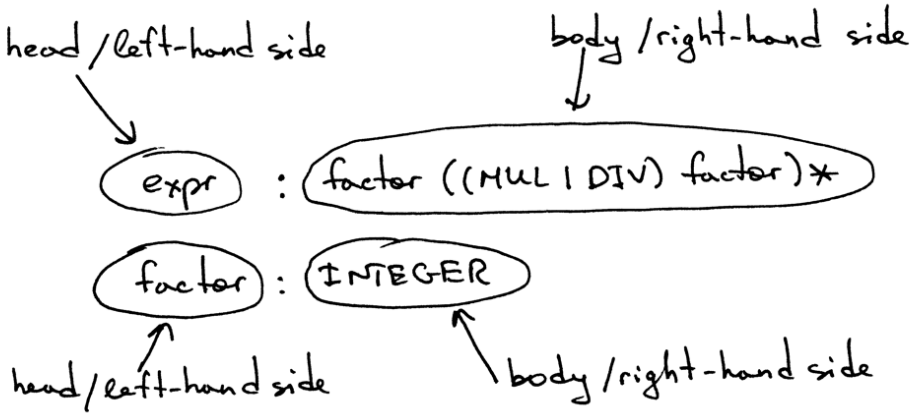
这是一个描述算术表达式的语法，如“7 * 4 / 2 * 3”（它只是该语法可以生成的众多表达式之一）：

```
expr   : factor ((MUL | DIV) factor)*
factor : INTEGER
```

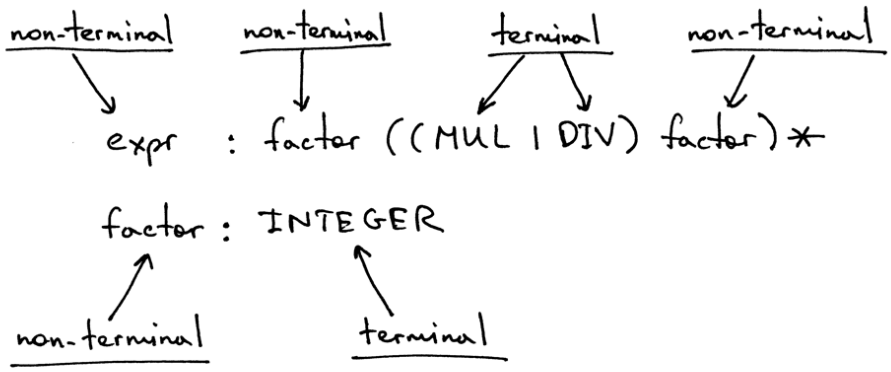
文法由一系列规则组成，也称为产生式。我们的语法中有两条规则：



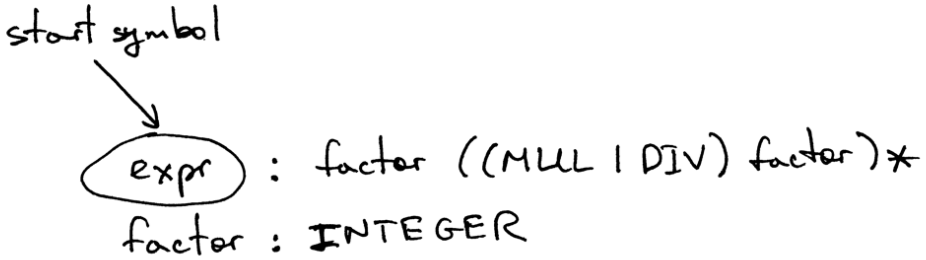
规则由一个非终结符（称为产生式的**头部**或**左侧**）、一个冒号和一系列终结符和/或非终结符（称为产生式的**主体**或**右侧**）**组成**：



在我上面展示的语法中，像MUL、DIV和INTEGER这样的标记被称为**终端，**而像expr和factor这样的变量被称为**非终端。**非终结符通常由一系列终结符和/或非终结符组成：



第一条规则左侧的非终结符称为**起始符号。**在我们的文法中，起始符号是expr：



您可以将规则expr解读为“一个expr可以是一个因子，可选地后跟乘法或除法运算符，然后是另一个因子，后者又可选地后跟乘法或除法运算符，然后是另一个因子，依此类推。”

什么是因子？就本文而言，因子只是一个整数。

让我们快速浏览一下语法中使用的符号及其含义。

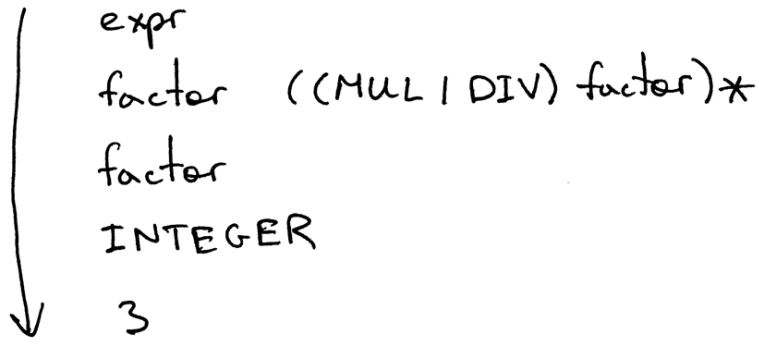
- **|** - 备择方案。条形表示“或”。所以（MUL | DIV）表示MUL或DIV。
- **(...)** - 左括号和右括号表示对终端和/或非终端进行分组，如（MUL | DIV）。
- **(...)** * - 匹配组内的内容零次或多次。

如果您过去使用过正则表达式，那么符号**|**，**()**和**(...)** * 对您来说应该很熟悉。

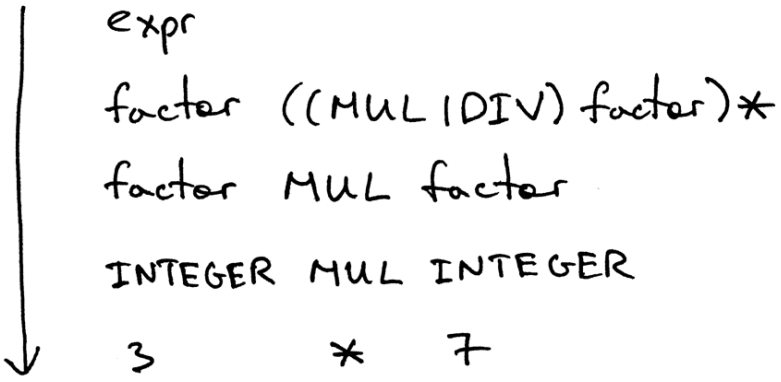
语法通过解释语言可以形成的句子来定义语言。这是使用语法推导出算术表达式的方法：首先以起始符号expr开始，然后用该非终结符的规则体重复替换非终结符，直到生成仅由终结符组成的句子. 这些句子形成语言的语法定义。

如果语法无法导出某个算术表达式，则它不支持该表达式，并且解析器在尝试识别该表达式时将产生语法错误。

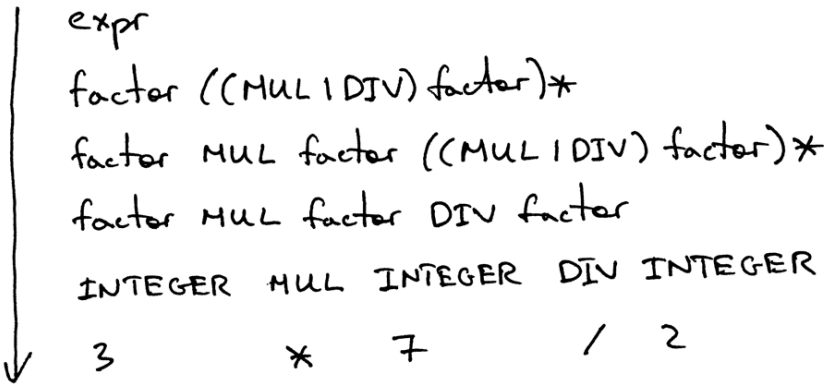
我认为有几个例子是有序的。这是语法推导表达式3 的方式：



这就是语法推导表达式3 * 7 的方式：



这就是语法如何推导出表达式3 * 7 / 2：



哇，那里有相当多的理论！

我想当我第一次阅读语法、相关术语和所有爵士乐时，我的感觉是这样的：

我可以向你保证，我绝对不是这样的：

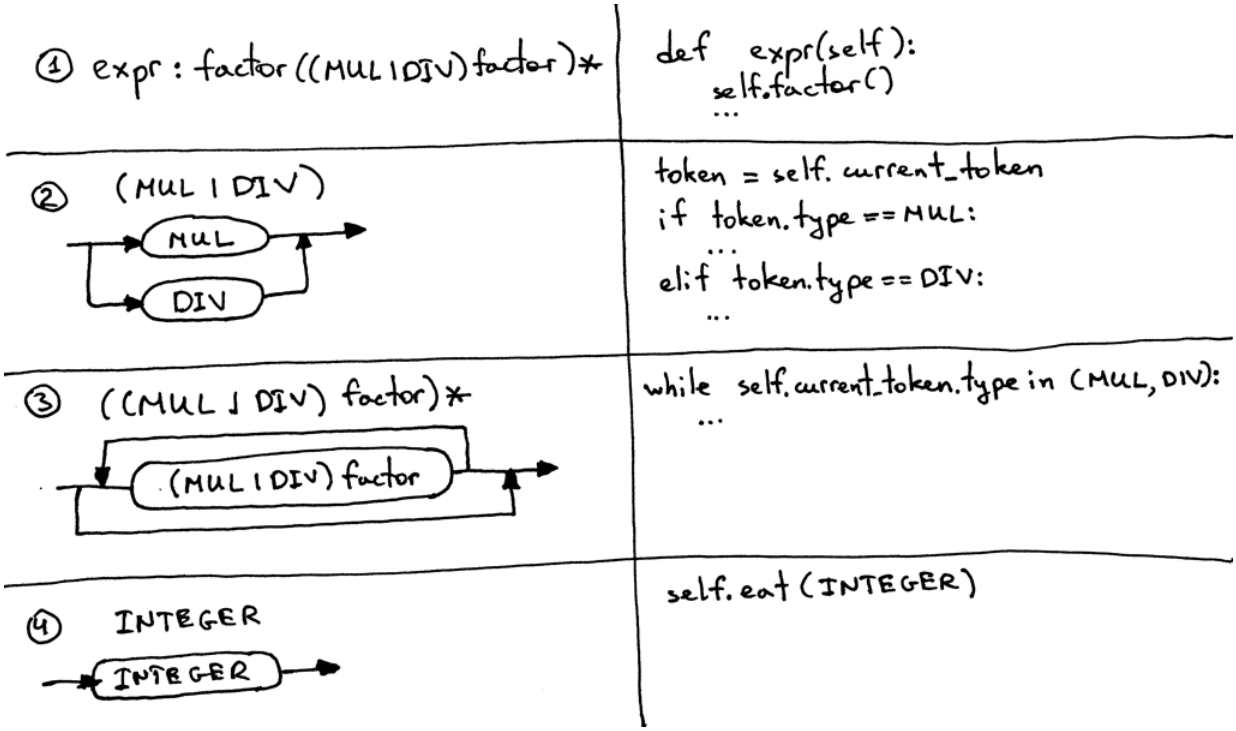
我花了一些时间来熟悉这个符号、它是如何工作的以及它与解析器和词法分析器的关系，但我必须告诉你，从长远来看，学习它是值得的，因为它在实践和编译器文献中被广泛使用你一定会在某个时候遇到它。那么，为什么不早不宜迟呢？:)

现在，让我们将该语法映射到代码，好吗？

以下是我们将用于将语法转换为源代码的指南。通过遵循它们，您可以从字面上将语法转换为工作解析器：

1. 语法中定义的每条规则**R**成为同名的方法，对该规则的引用成为方法调用：**R()**。该方法的主体遵循使用完全相同的准则的规则主体的流程。
2. 替代方案**(a1 | a2 | aN)**成为**if-elif-else** 语句
3. 一个可选的分组**(...)***变成一个可以循环零次或多次的**while**语句
4. 每个标记引用**T**成为对方法**eat**的调用：**eat(T)**。的方式吃的方法作品中，它消耗的令牌牛逼，如果当前的匹配超前记号，然后从词法分析器中获得一个新的令牌和受让该令牌到current_token内部变量。

从视觉上看，指南如下所示：



让我们开始行动，按照上述指南将我们的语法转换为代码。

我们的文法中有两条规则：一条expr规则和一条factor规则。让我们从因子规则（生产）开始。根据指南，您需要创建一个名为factor（指南 1）的方法，该方法只需调用Eat方法即可使用INTEGER令牌（指南 4）：

```
定义 因子（自我）：
    自我。吃（整数）
```

那很容易，不是吗？

向前！

规则expr变成了expr方法（再次根据准则 1）。规则的身体开始与一参考因子变为一个因子（）方法的调用。可选的分组(...)*成为一个while循环，而(MUL | DIV)替代品成为一个if-elif-else语句。通过将这些部分组合在一起，我们得到以下expr 方法：

```
def  expr ( self ):
    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 ）
            自我。因子()
        elif 令牌。类型 ==  DIV :
            自我。吃（DIV ）
            自我。因子()
```

请花一些时间研究我如何将语法映射到源代码。确保你理解那部分，因为它稍后会派上用场。

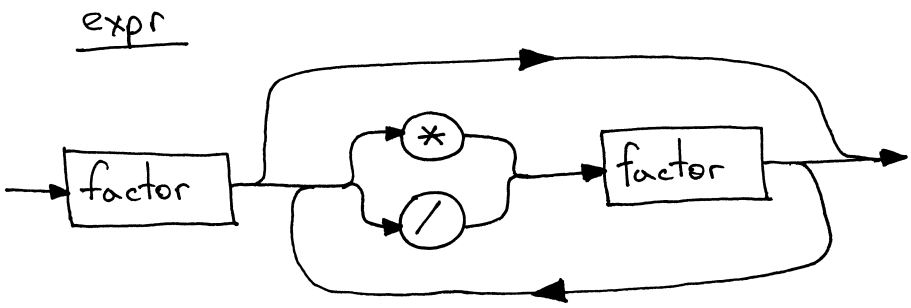
为方便起见，我将上述代码放入parser.py文件中，该文件包含一个词法分析器和一个没有解释器的解析器。您可以直接从GitHub (https://github.com/rspivak/lsbasi/blob/master/part4/parser.py)下载该文件并使用它。它有一个交互式提示，您可以在其中输入表达式并查看它们是否有效：也就是说，根据语法构建的解析器是否可以识别表达式。

这是我在计算机上运行的示例会话：

```
$ python parser.py
计算> 3
计算> 3 * 7
计算> 3 * 7 / 2
计算> 3 *
回溯（最近一次通话）：
  文件"parser.py"，第155行，在 <module> 中
    main ()
  文件"parser.py"，第151行，在 main
    parser.parse ()
  文件“parser.py”，第136行，在解析中
    self.expr ()
  文件“parser.py”，第130行，在 expr
    self.factor ()
  文件“parser.py”，第114行，因子
    self.eat ( INTEGER )
  文件“parser.py”，第107行，在eat
    self.error ()
  文件“parser.py”，第97行，出错
    引发异常（'无效语法' ）
例外：无效的语法
```

试试看！

我忍不住再次提到语法图。这是相同expr规则的语法图的外观：



是时候深入研究新算术表达式解释器的源代码了。下面是一个计算器的代码，它可以处理包含整数和任意数量的乘法和除法（整数除法）运算符的有效算术表达式。您还可以看到，我将词法分析器重构为一个单独的Lexer类，并更新了Interpreter类以将Lexer实例作为参数：

```
# 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 , MUL , DIV , EOF = 'INTEGER' , 'MUL' , 'DIV' , ' EOF'


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

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

        实例：
            令牌（INTEGER，3）
            令牌（MUL， '*'）
        “””
        返回 '令牌（{类型}，{值}）' 。格式（
            类型=自我。类型，
            值=再版（自我。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 不 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( 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 ):
        """返回一个整数标记值。

        因素：INTEGER
        """
        令牌 = 自我。current_token
        自我。吃（INTEGER ）
        返回 令牌。值

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

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

        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  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calc4.py文件中或直接从GitHub (https://github.com/rspivak/lsbasi/blob/master/part4/calc4.py)下载。像往常一样，尝试一下，看看它是否有效。

这是我在笔记本电脑上运行的示例会话：

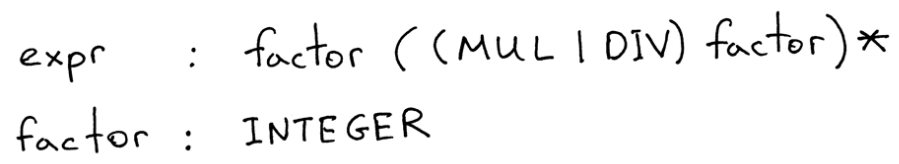
```
$ python calc4.py
计算> 7 * 4 / 2
14
计算> 7 * 4 / 2 * 3
42
计算> 10 * 4  * 2 * 3 / 8
30
```

我知道你等不及这部分了 :) 以下是今天的新练习：

- 编写一个语法来描述包含任意数量的 +、-、* 或 / 运算符的算术表达式。使用语法，您应该能够推导出诸如“2 + 7 * 4”、“7 - 8 / 4”、“14 + 2 * 3 - 6 / 2”等表达式。
- 使用语法编写一个解释器，该解释器可以计算包含任意数量的 +、-、* 或 / 运算符的算术表达式。您的解释器应该能够处理诸如“2 + 7 * 4”、“7 - 8 / 4”、“14 + 2 * 3 - 6 / 2”等表达式。
- 如果您完成了上述练习，请放松并享受:)

检查你的理解。

记住今天文章中的语法，回答以下问题，根据需要参考下图：



1. 什么是上下文无关文法（grammar）？
2. 语法有多少规则/产生式？
3. 什么是终端？（识别图中所有终端）
4. 什么是非终端？（识别图中所有非终端）
5. 什么是规则头？（识别图片中的所有头部/左侧）
6. 什么是规则体？（识别图片中的所有身体/右侧）
7. 语法的起始符号是什么？

嘿嘿，你一直读到最后！这篇文章包含了相当多的理论，所以我为你完成它而感到自豪。

下次我会带着新文章回来——请继续关注，不要忘记做练习，它们会对你有好处。

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 12.
5 years ago • 29 comments
"Be not afraid of going slowly; be afraid only of standing still." - Chinese …

Let's Build A Simple Interpreter. Part 6.
6 years ago • 20 comments
Today is the day :) "Why?" you might ask. The reason is that today we're …

Let's Build A Simple Interpreter. Part 19: …
2 years ago • 24 comments
What I cannot create, I do not understand. —- Richard Feynman

Let's Build A Simple Interpreter. Part 14: …
4 years ago • 52 comments
Only dead fish go with the flow. As I promised in the last article, today we're …

Let's Interp
6 years
As I pr today the ce

14 Comments **Ruslan's Blog** **Disqus' Privacy Policy** **Login**

Recommend 5 **Tweet** **Share** **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS**

Name

Daniel Coronado • 6 years ago
Good!!
37 • Reply • Share ›

mike • a year ago
you are a god-send. thank you!
1 ^ | v • Reply • Share ›

liang zhang • 2 years ago
Make me feeling good!! Great Article
1 ^ | v • Reply • Share ›

Krishnadas PC • 3 years ago
Great Article.
1 ^ | v • Reply • Share ›

Toby • 5 years ago
I've been programming since I was a kid and then professionally for about 20 years. Never done any CS course so I'm enjoying following this process as a start at going back and filling in gaps (and a bit more approachable than that book on Erlang I have).
1 ^ | v • Reply • Share ›

ZhouQuan • 5 years ago
Great Article !
1 ^ | v • Reply • Share ›

Jan Kreps • 6 years ago
Many thanks for this series. Keep up the good job please.
1 ^ | v • Reply • Share ›

Domenic Del Nano • 6 years ago
This series has been awesome cannot wait for part 5!!
1 ^ | v • Reply • Share ›

Andrea Pivetta • 6 years ago
Great!! Looking forward to the next part.
1 ^ | v • Reply • Share ›

GomathiNayagam • 6 years ago
awesome, essence of all the above mentioned book
1 ^ | v • Reply • Share ›

solstice333 • 2 years ago
Is number 2 in the exercises (
Using the grammar, write an interpreter that can evaluate arithmetic
expressions containing any number of +, -, *, or / operators. Your
interpreter should be able to handle expressions like “2 + 7 * 4”, “7 - 8
/ 4”, “14 + 2 * 3 - 6 / 2”, and so on), supposed to be with taking into account operator precedence? I'm assuming that's the case. Any thoughts on how to go about this given parts 1 through 4 of this series?
^ | v • Reply • Share ›

solstice333 ➚ solstice333 • 2 years ago
Oh, I did something lame like expanding the tokenized stream to a token list, then left-folding/reducing every MUL or DIV expression and replacing the expression token slice with that of the evaluated result. In other words, a multi-pass operation where the first pass squashes the MUL/DIV expressions, and the second pass does the ADD/SUB. Anyway, all that complexity wasn't even needed since another way to read `self.current_token` is `self.lookahead_token`, and then you can recurse into another method that takes higher parsing precedence
^ | v • Reply • Share ›

Վիգեն Բաղդասարյան ➚ solstice333 • 6 months ago
I looked to the wikipedia to be truth, here is basic grammar which doesn't require `self.current_token`

```
expr : addition-expression
addition-expression : multiplication-expression ((PLUS|MINUS) multiplication-expression)*
```

```
multiplication-expression : factor ((MUL|DIV) factor)*
factor : INTEGER
```

Reply • Share ›

Ahmed Hamdan • 4 years ago

At first I was not getting grammars
but when I heard REGEX I was like Yesssss!

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。

回到顶部