让我们构建一个简单的解释器。第 3 部分。 (https://ruslanspivak.com/lsbasi-part3/)

日期 2015 年 8 月 12 日，星期三

我今天早上醒来，心里想：“为什么我们觉得学习一项新技能这么难？”

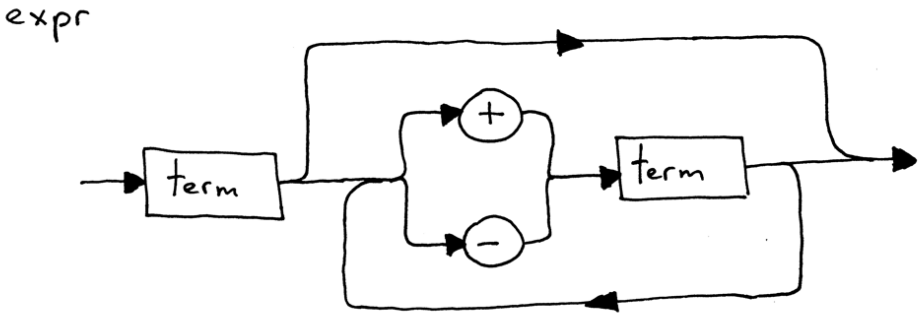
我不认为这只是因为努力工作。我认为其中一个原因可能是我们花了很多时间和努力通过阅读和观看来获取知识，而没有足够的时间通过实践将这些知识转化为技能。以游泳为例。您可以花大量时间阅读数百本有关游泳的书籍，与经验丰富的游泳运动员和教练交谈数小时，观看所有可用的训练视频，但在您第一次跳入游泳池时，您仍然会像石头一样下沉。

底线是：无论您认为自己了解该主题的程度如何 - 您必须将这些知识付诸实践才能将其转化为一项技能。为了帮助您完成练习部分，我将练习放入该系列的第 1 (http://ruslanspivak.com/lsbasi-part1/)部分 (http://ruslanspivak.com/lsbasi-part2/)和第 2 部分 (http://ruslanspivak.com/lsbasi-part2/)。是的，你会在今天的文章和以后的文章中看到更多的练习，我保证:)

好的，让我们开始今天的材料，好吗?

到目前为止，您已经学习了如何解释对两个整数进行加减运算的算术表达式，例如“7 + 3”或“12 - 9”。今天我将讨论如何解析（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”。

从图形上看，本文中的算术表达式可以用以下语法图表示：



什么是语法图？一个**语法图**是一种编程语言的语法规则的图示。基本上，语法图直观地向您展示了哪些语句在您的编程语言中是允许的，哪些是不允许的。

语法图很容易阅读：只需按照箭头指示的路径进行操作即可。一些路径表示选择。并且一些路径表示循环。

您可以阅读上面的语法图如下：一个术语可选地后跟一个加号或减号，然后是另一个术语，而另一个术语又可选地后跟一个加号或减号，然后是另一个术语，依此类推。你得到了图片，字面意思。您可能想知道什么是“术语”。就本文而言，“术语”只是一个整数。

语法图有两个主要目的：

- 它们以图形方式表示编程语言的规范（语法）。
- 它们可用于帮助您编写解析器——您可以按照简单的规则将图表映射到代码。

您已经了解到在标记流中识别短语的过程称为**解析**。执行该工作的解释器或编译器的部分称为**解析器**。解析也称为**语法分析**，解析器也被恰当地称为，你猜对了，**语法分析器**。

根据上面的语法图，以下所有算术表达式都是有效的：

- 3
- 3 + 4
- 7 - 3 + 2 - 1

由于不同编程语言中算术表达式的语法规则非常相似，我们可以使用 Python shell 来“测试”我们的语法图。启动你的 Python shell 并亲眼看看：

```
>>> 3
3
>>> 3 + 4
7
>>> 7 - 3 + 2 - 1
5
```

这里没有惊喜。

表达式“3 + ”不是有效的算术表达式，因为根据语法图，加号后面必须跟一个术语（整数），否则就是语法错误。再次，用 Python shell 尝试一下，亲眼看看：

```
>>> 3 +
  文件"<stdin>"，第1
    行3 +
      ^
语法错误：无效语法
```

能够使用 Python shell 进行一些测试是很棒的，但是让我们将上面的语法图映射到代码并使用我们自己的解释器进行测试，好吗？

您从之前的文章（第 1 部分 (http://ruslanspivak.com/lsbasi-part1/)和第 2 部分 (http://ruslanspivak.com/lsbasi-part2/)）中知道expr方法是我们的解析器和解释器所在的地方。同样，解析器只是识别结构以确保它符合某些规范，并且一旦解析器成功识别（解析）它，解释器就会实际评估表达式。

以下代码片段显示了与图表对应的解析器代码。语法图（term）中的矩形框成为解析整数的term方法，而expr方法只是遵循语法图流程：

```
定义 项（自我）：
    自我。吃（整数）

def  expr ( self ):
    # 将当前标记设置为从输入
    self 中获取的第一个标记。current_token  =  self 。get_next_token ()

    自我。term ()
    而 self 。current_token 。键入 在 （PLUS ， MINUS ）：
        令牌 = 自我。current_token
        如果是 token 。类型 ==  PLUS ：
            自我。吃（PLUS ）
            的自我。term ()
        elif 令牌。类型 == 减号：
            自我。吃（减）
            自我。术语()
```

可以看到expr首先调用了term方法。然后expr方法有一个可以执行零次或多次的while循环。在循环内部，解析器根据标记（无论是加号还是减号）做出选择。花一些时间向自己证明上面的代码确实遵循算术表达式的语法图流程。

解析器本身并不解释任何东西：如果它识别出一个表达式，它就会保持沉默，如果没有，它会抛出一个语法错误。让我们修改expr方法并添加解释器代码：

```
def  term ( self ):
    """返回一个整数标记值"""
    token  =  self 。current_token
    自我。吃（整数）
    返回 令牌。价值

def  expr ( self ):
    """Parser / Interpreter """
    # 将当前标记设置为从输入
    self 中获取的第一个标记。current_token  =  self 。get_next_token ()

    结果 = 自我。term ()
    而 self 。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
```

由于解释器需要对表达式求值，因此修改term方法以返回整数值，修改expr方法以在适当的位置执行加减运算并返回解释结果。尽管代码非常简单，但我还是建议花一些时间研究它。

现在开始动起来，看看解释器的完整代码，好吗？

这是新版本计算器的源代码，它可以处理包含整数和任意数量的加减运算符的有效算术表达式：

```
# 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
INTEGER , PLUS , MINUS , EOF = 'INTEGER' , 'PLUS' , 'MINUS' , ' EOF'


class Token ( object ):
    def __init__ ( self , type , value ):
        # token type: INTEGER, PLUS, MINUS, 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（PLUS， '+'）
        “””
        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 Interpreter ( object ):
    def __init__ ( self , text ):
        # 客户端字符串输入，例如 "3 + 5"、"12 - 5 + 3" 等
        self 。text = text
        # self.pos 是 self.text
        self的索引。pos = 0
        # 当前令牌实例
        self . current_token = None
        self 。current_char = self 。文本[自我。位置]

    ############################################## #########
    词法分析器代码
    ################################### ###################
    def error ( self ):
        raise Exception ( 'Invalid syntax' )

    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( PLUS ,  '+' )

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

            自我。错误()

        返回 令牌（EOF ， 无）

    ########################################### #########
    解析器/解释器代码
    ################################## ####################
    def  eat ( self ,  token_type ):
        # 将当前标记类型与传递的标记进行比较
        # 类型，如果匹配则“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 。get_next_token()
        其他：
            自我。错误()

    def  term ( self ):
        """返回一个整数标记值。"""
        token  =  self 。current_token
        自我。吃（整数）
        返回 令牌。价值

    def  expr ( self ):
        """算术表达式解析器/解释器。"""
        # 将当前标记设置为从输入
        self 中获取的第一个标记。current_token  =  self 。get_next_token ()

        结果 = 自我。term ()
        而 self 。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
        interpreter = Interpreter ( text )
        result = 口译员。expr ()
        打印（结果）


如果 __name__ == '__main__' :
    main ()
```

将上述代码保存到calc3.py文件中或直接从GitHub (https://github.com/rspivak/lsbasi/blob/master/part3/calc3.py)下载。试试看。亲眼看看它可以处理算术表达式，这些表达式可以从我之前向您展示的语法图中派生出来。

这是我在笔记本电脑上运行的示例会话：

```
$ python calc3.py
计算> 3
3
计算> 7 - 4
3
计算> 10 + 5
15
计算> 7 - 3 + 2 - 1
5
计算> 10 + 1 + 2 - 3 + 4 + 6 - 15
5
计算> 3 +
回溯（最近一次通话）：
  文件"calc3.py"，第147行，在 <module> 中
    main ()
  文件“calc3.py”，第142行，在 main
     result = interpreter.expr ()
  文件“calc3.py”，第123行，在 expr
     result = result + self.term ()
  文件“calc3.py”中，线110，在术语
    self.eat ( INTEGER )
  文件“calc3.py”，第105行，在eat
    self.error ()
  文件"calc3.py"，第45行，出错
    引发异常（'无效语法' ）
例外：无效的语法
```

记住我在文章开头提到的那些练习：它们在这里，正如承诺的那样:)

- 为仅包含乘法和除法的算术表达式绘制语法图，例如“7 * 4 / 2 * 3”。说真的，只要拿起一支钢笔或一支铅笔，试着画一个。
- 修改计算器的源代码以解释仅包含乘法和除法的算术表达式，例如“7 * 4 / 2 * 3”。
- 编写一个解释器，从头开始处理像“7 - 3 + 2 - 1”这样的算术表达式。使用您熟悉的任何编程语言，并在不查看示例的情况下将其写下来。当你这样做，想想涉及的组件：一个词法分析器，其采用输入，并将其转换成标记流，解析器是关闭的所提供的令牌流饲料词法分析器，并试图在该流识别的结构，在解析器之后生成结果的解释器已成功解析（识别）一个有效的算术表达式。将这些碎片串在一起。花一些时间将您获得的知识翻译成算术表达式的工作解释器。

检查你的理解。

1. 什么是语法图?
2. 什么是语法分析?
3. 什么是语法分析器?

你看！你一直读到最后。感谢您今天在这里闲逛，不要忘记做练习。:) 下次我会回来写一篇新文章 - 请继续关注。

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 9.

5 years ago • 42 comments

I remember when I was in university (a long time ago) and learning systems …

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 B Interpr

6 years a

Today w operator plus (+)

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

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

lpkn • 4 years ago

Hello,

Thanks for these posts, well explained!

A question: it looks like the expression '3 3+3' is accepted, while not valid, is it expected?

Thanks

3 • Reply • Share ›

Svetoslav Chingov ipkn • 3 years ago • edited

This is a bug which is fixed by putting the following exactly before the last line of the expr method:

```
if self.current_token.type != EOF:
          self.error()
```

Reply • Share ›

Illia Kuksenko • 3 years ago

I think, it would be not a bad idea to mention CS262 course in Udacity.com.(no ads,I just started this free course and found it really helpful as much as this tutorial)

1 Reply • Share ›

Luca Polito • 6 years ago

I love your tutorials, I'm writing an interpreter in PHP and I can't wait for part 4!
P.S.: Keep up the good work!

1 Reply • Share ›

rspivak Mod ipkn Luca Polito • 6 years ago

Thank you and good luck with your interpreter!

2 Reply • Share ›

Aleksandr Zhuravlev Luca Polito • 6 years ago

Could you share the source code of your interpreter in PHP?

Reply • Share ›

Luca Polito Aleksandr Zhuravlev • 6 years ago • edited

Here: https://github.com/lucapoli...
I tried to make the code as easy to understand as possible.
If you want some help, feel free to ask me anything (you can find my email in my GitHub's profile).
Also, note that my code is quite different from Ruslan's one.

1 Reply • Share ›

Aleksandr Zhuravlev Luca Polito • 6 years ago

Thank you. I'll play with it this weekend. I have a friend who wrote his own interpreter in PHP as well. It's interesting to compare both approach.

Reply • Share ›

JStitch • 5 years ago • edited

Another Python calculator supporting +, -, *, / and ^ (exponentiation), with integers and float numbers support

It uses a generator for the lexer

https://gist.github.com/jst...

Reply • Share ›

James Sully • 5 years ago • edited

Spoiler for exercise 1:

Reply • Share ›

Annonn Yimous James Sully • 2 years ago

You seem to be right because as you said, multiplication and division have the same precedence.

Reply • Share ›

anonymus • 6 years ago

Thanks for this tutorial

Thanks for this tutorial.

As before, I've implemented a version in Java. For others to play with available here (part 3):

https://codeboard.io/projec...

Keep up the good work.

Reply • Share ›

Nikita • 6 years ago

The tutorials were very helpful. Could you please tell how to handle brackets.
Thank you for the tutorials.

Reply • Share ›

Emil Guseynov • 6 years ago

Very good and easy to read articles, Ruslan! But I think you should have told us how to handle brackets in expressions because what you explain here is trivial and anybody more or less familiar with programming can implement it. Anyway, thanks for your tutorials.

Reply • Share ›

anonymus • 6 years ago