

让我们构建一个简单的解释器。第 6 部分。 (<https://ruslanspivak.com/lbasi-part6/>)

日期 2015 年 11 月 2 日, 星期一

今天是这一天:) “为什么?” 你可能会问。原因是今天我们结束了对算术表达式的讨论 (好吧, 几乎) 通过在我们的语法中添加括号表达式并实现一个解释器, 该解释器将能够评估具有任意深度嵌套的括号表达式, 例如表达式 $7 + 3 * (10 / (12 / (3 + 1) - 1))$ 。

让我们开始吧, 好吗?

首先, 让我们修改语法以支持括号内的表达式。正如您在第 5 部分 中 (<http://ruslanspivak.com/lbasi-part5/>)记得的那样, 因子规则用于表达式中的基本单位。在那篇文章中, 我们唯一的基本单位是整数。今天我们要添加另一个基本单位——括号表达式。我们开始做吧。

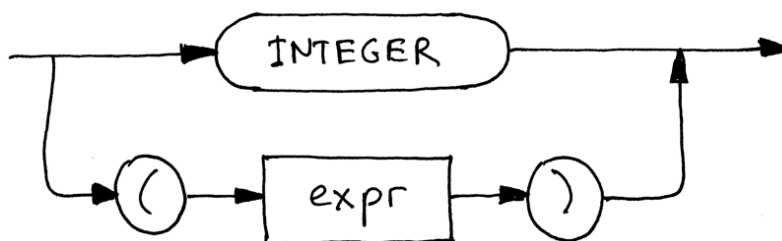
这是我们更新的语法:

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

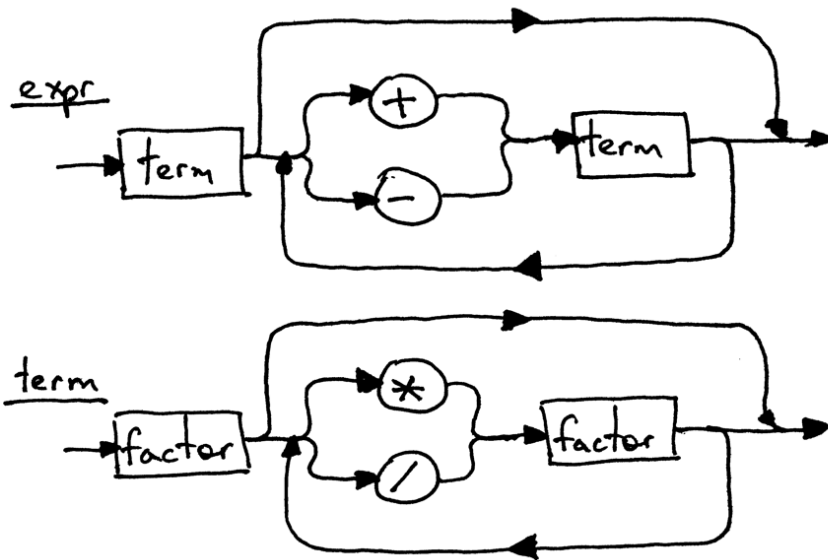
的EXPR和术语制作完全相同如在第5部分 (<http://ruslanspivak.com/lbasi-part5/>)和唯一的变化是在因子生产其中终端LPAREN表示左括号 “(”, 终端RPAREN表示右括号 “)”, 和非括号之间的终止expr指的是expr 规则。

这是factor的更新语法图, 现在包括替代方案:

factor

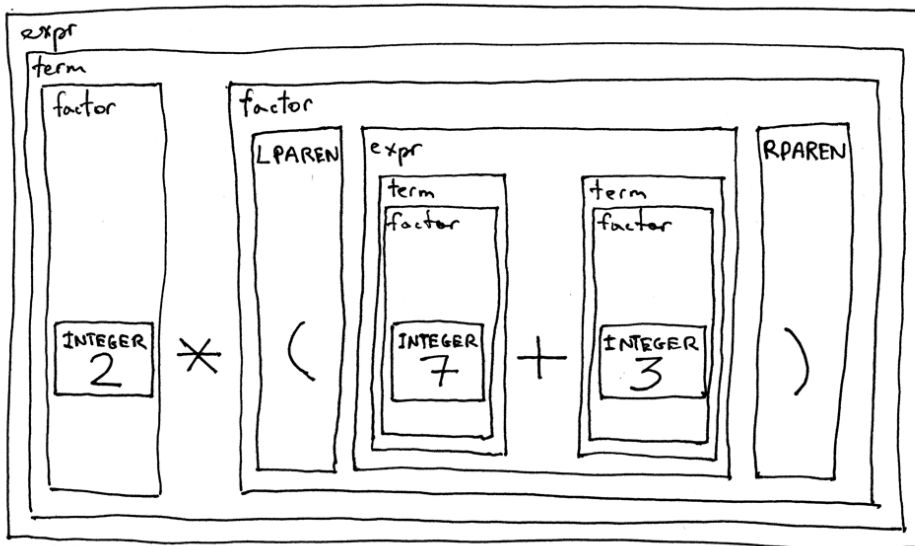


因为expr和术语的语法规则没有改变, 它们的语法图看起来与第 5 部分中 (<http://ruslanspivak.com/lbasi-part5/>)的相同:



这是我们新语法的一个有趣特性——它是递归的。如果您尝试推导表达式 $2 * (7 + 3)$ ，您将从 `expr` 开始符号开始，最终您将再次递归使用 `expr` 规则推导 $(7 + 3)$ 部分原始算术表达式。

让我们根据语法对表达式 $2 * (7 + 3)$ 进行分解，看看它的样子：



顺便说一句：如果您需要复习递归，请查看 Daniel P. Friedman 和 Matthias Felleisen 的 *The Little Schemer* (http://www.amazon.com/gp/product/0262560992/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0262560992&linkCode=as2&tag=russblo0b-20&linkId=IM7CT7RLWNGJ7J54) 一书 - 这本书非常好。

好的，让我们开始吧，将我们新更新的语法翻译成代码。

以下是对上一篇文章中代码的主要更改：

1. 该词法已被修改为返回两个标记：LPAREN的左括号和RPAREN一个右括号。

2. 该解释的因素法已略有更新解析除了整数括号表达式。

这是可以计算包含整数的算术表达式的计算器的完整代码；任意数量的加法、减法、乘法和除法运算符；和带有任意深度嵌套的括号表达式：

```

# 标记类型
#
# EOF (end-of-file) 标记用于表示
# 没有更多的输入可以用于词法分析
INTEGER , PLUS , MINUS , MUL , DIV , LPAREN , RPAREN , EOF = (
    'INTEGER' , '加' , '减' , 'MUL' , 'DIV' , '(' , ')' , 'EOF'
)

class Token ( object ):
    def __init__ ( self , type , value ):
        self . 类型 = 类型
        self . 价值 = 价值

    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 "4 + 2 * 3 - 6 / 2"
        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( 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, '/' )

    如果 自。current_char == '(':
        自我。提前()
        返回 令牌( LPAREN, '(' )

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

    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 | LPAREN expr RPAREN"""
        token = self.current_token
        如果是 token。类型 == 整数:
            自我。吃 ( 整数 )
            返回 令牌。价值

```

```

elif 令牌。类型 == LPAREN :
    self。吃( LPAREN )
    结果 = 自我.expr ()
    自我。吃( RPAREN )
    返回 结果

```

```

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> 7 + 3 * (10 / (12 / (3 + 1) - 1))
    22

```

```

expr : term ((PLUS | MINUS) term)*
term : factor ((MUL | DIV) factor)*
factor : INTEGER | LPAREN EXPR RPAREN
"""

```

结果 = 自我。术语()

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 ()

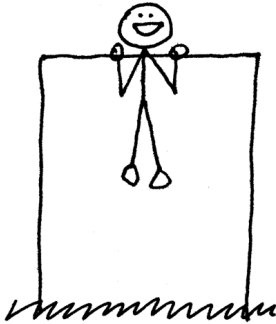
```

将上述代码保存到 `calc6.py` (<https://github.com/rspivak/lbasi/blob/master/part6/calc6.py>) 文件中，自己尝试一下，看看您的新解释器是否正确评估了具有不同运算符和括号的算术表达式。

这是一个示例会话：

```
$ python calc6.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
计算> 7 + 3 * ( 10 / ( 12 / ( 3 + 1 ) - 1 ) )
22
计算> 7 + 3 * ( 10 / ( 12 / ( 3 + 1 ) - 1 ) ) / ( 2 + 3 ) - 5 - 3 + ( 8 )
10
计算> 7 + ((( 3 + 2 )))
12
```

这是今天为您准备的新练习：



- 如本文所述，编写您自己的算术表达式解释器版本。请记住：重复是所有学习之母。

嘿嘿，你一直读到最后！恭喜，您刚刚学会了如何创建（如果您已经完成了练习 - 您已经实际编写了）一个可以计算非常复杂的算术表达式的基本递归下降解析器/解释器。

在下一篇文章中，我将更详细地讨论递归下降解析器。我还将在解释器和编译器构造中介绍一种重要且广泛使用的数据结构，我们将在整个系列中使用它。

请继续关注，很快就会见到你。在此之前，继续为您的口译员工作，最重要的是：玩得开心，享受这个过程！

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 13: ...

4 years ago • 11 comments

Anything worth doing is worth overdoing. Before doing a deep dive into ...

Let's Build A Simple Interpreter. Part 17: ...

2 years ago • 8 comments

You may have to fight a battle more than once to win it. - Margaret Thatcher

EOF is

2 years ago

I was re: Systems Perspec

20 Comments

Ruslan's Blog

Disqus' Privacy Policy

Login

Recommend 1

Tweet

Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



Cuong • 5 years ago

Hi,

Thanks a lot for the guide.

I've pushed myself through this part, and implemented the interpreter without peeping onto the sample code :-).

But I noticed that the code does not handle error effectively, for example:

```
calc> 10 + 2 * 3 2 * 10 / 4 5 8
16
```

I actually expected that an error was raised. Digging into the code, it seems that this type of error is silently ignored. Anyway to improve the grammar to handle this issue?

Cheers,

Cuong.

2 ^ | v • Reply • Share ›



Vanguard ➔ Cuong • 2 years ago • edited

I know that this might be late but this error was brought up in an earlier part of the series, a fix is adding right before the return on the expression function outside the while loop.

```
if CurrentToken.TokenType != EOF :
    self.error()
```

^ | v • Reply • Share ›



Taian ➔ Vanguard • a year ago • edited

Hey! I know that error, and this solution can solve it at earlier part of this series, but not this version.

When the `expr()` done, it will check if the interpreter has got the end, but expressions has parenthesis like `3 * (3 + 3) - 3` will check the token type before `'`', however, `current_type.type` will be `RPAREN`, and the error will be raised.

My solution is add a new function `subexpr()`, and move all the thing in `expr()` to it. The function `expr()` will get the result from `subexpr()` and check token type after call `subexpr()`. Of course, the factor will call the `subexpr()` instead of `expr()`.

Thank you so much, I learned (very)^N much from you!!!!

And my English is poor, so....I don't know I said above is right in syntax or not.

Thank you again!!!

^ | v • Reply • Share ›



yang tang • 6 months ago

Hi, I'm very curious...What is your drawing tool?

^ | v • Reply • Share ›



super_toto • 2 years ago

thank you very much

^ | v • Reply • Share ›



Bryse Meijer • 2 years ago • edited

Thank you for the awesome guide!!!

I am writing it in PHP.

<https://github.com/brysem/i...>

^ | v • Reply • Share ›



rspivak Mod ➔ **Bryse Meijer** • 2 years ago

You're welcome. Good luck with your project!

2 ^ | v • Reply • Share ›



Dan Boxall • 3 years ago

At the end of the last chapter I was kinda confused. In this one, it finally clicked. Thanks again for these!

^ | v • Reply • Share ›



Ahmed Hamdan • 4 years ago

I already made my own version in java and it's working

^ | v • Reply • Share ›



generic commenter • 4 years ago

I did the second exercise in part 5 so this was mostly redundant. I however had a parens rule and still had a separate rule for factor, which was just an integer.

^ | v • Reply • Share ›



Diego Marcia • 5 years ago

Hi, great guide!

But note: your code accepts an expr to end with an unmatched RPAREN, like 2+3)

```
calc> 1 + (1 + 1) + 1)
```

```
4
```

^ | v • Reply • Share ›



rspivak Mod ➔ **Diego Marcia** • 5 years ago

Hi Diego,

Thanks. The issue was fixed in Part 7 by adding slightly better error detection to the parser.

^ | v • Reply • Share ›



Diego Marcia ➔ **rspivak** • 5 years ago

Oh... Ok! I just skimmed through Part 7 and saw ASTs are introduced there...

Anyway, if someone reads this and for some reason really gets disturbed by the RPAREN issue, try invoking this method from main instead of directly calling expr (off the top of my head):

```
def parse(self):
    >result = self.expr()
```

```
>if self.current_token.type == EOF:
>>return result
>else:
>>self.error()
```

Again, Ruslan: great guide!

I found it last week, and I see you published Part 9 less than two weeks ago...
Guess I have to slow down, since I don't know if I can resist until part 10!

I also believe that Python was the right choice, since it's very easy to understand... I was able to understand everything, and guess what: the Python code I put in this comment is the first one I ever wrote!

KUDOS!!

^ | v • Reply • Share ›



Holger Joukl • 6 years ago

Nice series.

But unless I'm missing something: Does

```
def eat(self, token_type):
# compare the current token type with the passed token
# type and if they match then "eat" the current token
# and assign the next token to the self.current_token,
# otherwise raise an exception.
if self.current_token.type == token_type:
self.current_token = self.lexer.get_next_token()
else:
self.error()
```

ever reach the else code path?

Because wherever it's called there's a previous check like

```
token = self.current_token
if token.type == PLUS:
self.eat(PLUS)
```

which ensures that `self.current_token.type == <eat arg="" token_type="">`, anyway.

^ | v • Reply • Share ›



GOKOP ➔ Holger Joukl • 5 months ago • edited

There's a big chance you've found the answer already in 5 years, but: In cases you've specified, yes, the error check is redundant, but notice that `eat()` is also called in `factor()`, where it corresponds to what we expect and not what we've found. The else is reached when syntax is incorrect

^ | v • Reply • Share ›



magine • 6 years ago

dying for part7...

^ | v • Reply • Share ›



远航 • 6 years ago

I cannot wait for part7...
such great articles~

^ | v • Reply • Share ›



magine • 6 years ago

part 7! part 7! part 7! :P

^ | v • Reply • Share ›



John Ab • 6 years ago




Can't wait for part 7! I really want to know how to add keywords such as "print" or "class"

^ | 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/lbaws-part1/>)

让我们构建一个简单的解释器。第1部分。 (<https://ruslanspivak.com/lbasi-part1/>)

让我们构建一个 Web 服务器。第2部分。 (<https://ruslanspivak.com/lbaws-part2/>)

让我们构建一个 Web 服务器。第 3 部分。 (<https://ruslanspivak.com/lbaws-part3/>)

让我们构建一个简单的解释器。第2部分。 (<https://ruslanspivak.com/lbasi-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。