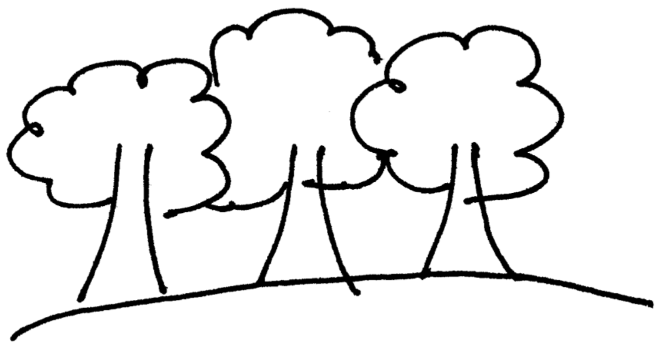
让我们构建一个简单的解释器。第 7 部分：抽象语法树 (https://ruslanspivak.com/lsbasi-part7/)

日期 2015 年 12 月 15 日，星期二

正如我上次向您保证的那样，今天我将讨论我们将在本系列的其余部分中使用的中心数据结构之一，所以系好安全带，让我们开始吧。

到目前为止，我们将解释器和解析器代码混合在一起，一旦解析器识别出某种语言结构，如加法、减法、乘法或除法，解释器就会对表达式求值。这种解释器被称为语法导向解释器。它们通常对输入进行一次传递，适用于基本语言应用程序。为了分析更复杂的 Pascal 编程语言结构，我们需要构建一个中间表示(IR)。我们的解析器将负责构建一个 IR，我们的解释器将使用它来解释表示为IR的输入。

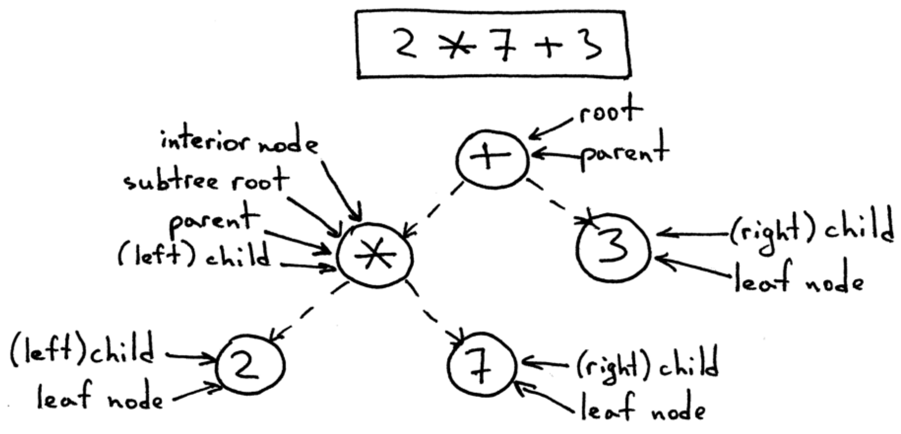
事实证明，树是非常适合IR 的数据结构。



让我们快速讨论一下树术语。

- 一个树是一种数据结构，由组织成一个层次中的一个或多个节点。
- 这棵树有一个根，它是顶部节点。
- 除了根节点之外的所有节点都有一个唯一的parent。
- 下图中标有*****的节点是父节点。标记为**2**和**7 的**节点是它的子节点；孩子们是从左到右排列的。
- 没有子 节点的节点称为叶节点。
- 具有一个或多个子节点且不是根的 节点称为内部节点。
- 子树也可以是完整的子树。在下图中，**+**节点的左子节点（标记为*****） 是一个完整的子树，带有自己的子节点。
- 在计算机科学中，我们从顶部的根节点和向下生长的分支开始倒置绘制树。

这是表达式 2 * 7 + 3 的一棵树，并附有说明：

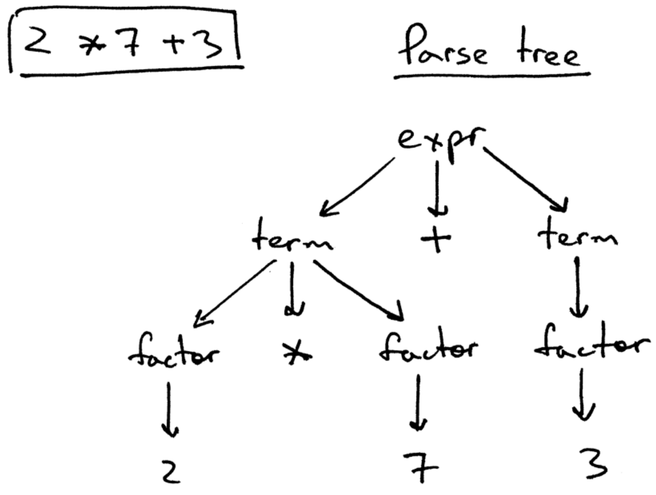


我们将在整个系列中使用的IR称为抽象语法树(AST)。但在我们深入研究 AST 之前，让我们先简单地谈谈解析树。尽管我们不会在解释器和编译器中使用解析树，但它们可以通过可视化解析器的执行跟踪来帮助您了解解析器如何解释输入。我们还将它们与 AST 进行比较，以了解为什么 AST 比解析树更适合用于中间表示。

那么，什么是解析树？一个解析树（有时称为具体语法树）是表示根据我们的语法定义语言结构的句法结构树。它基本上显示了您的解析器如何识别语言结构，或者换句话说，它显示了您的语法的起始符号如何派生出编程语言中的某个字符串。

解析器的调用堆栈隐式地表示一个解析树，它会在您的解析器尝试识别特定语言结构时自动构建在内存中。

让我们看一下表达式 2 * 7 + 3 的解析树：



在上图中，您可以看到：

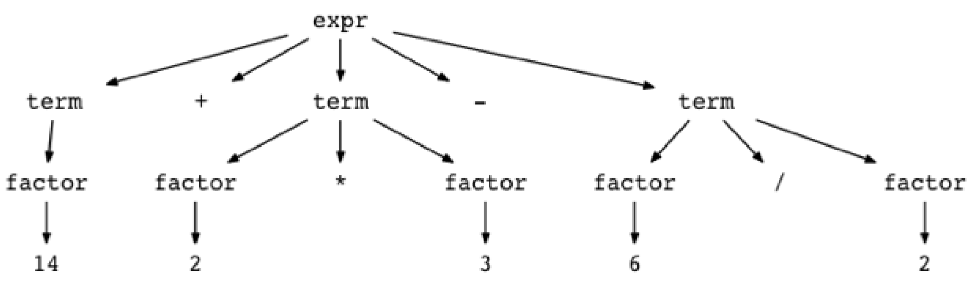
- 解析树记录了解析器用于识别输入的一系列规则。
- 解析树的根用语法开始符号标记。
- 每个内部节点代表一个非终结符，即它代表一个语法规则应用，如我们的例子中的expr、term或factor。
- 每个叶节点代表一个令牌。

正如我已经提到的，我们不会手动构建解析器树并将它们用于我们的解释器，但是解析树可以通过可视化解析器调用序列来帮助您理解解析器如何解释输入。

您可以通过试用一个名为genptdot.py (https://github.com/rspivak/lsbasi/blob/master/part7/python/genptdot.py)的小实用程序来了解不同算术表达式的解析树的外观，我很快编写了该实用程序来帮助您将它们可视化。要使用该实用程序，您首先需要安装 Graphviz (http://graphviz.org)包，在运行以下命令后，您可以打开生成的图像文件 parsetree.png 并查看作为命令行参数传递的表达式的解析树：

```
$蟒genptdot.py “14 + 2 * 3 - 6/2” > \
  parsetree.dot &&点-Tpng -o parsetree.png parsetree.dot
```

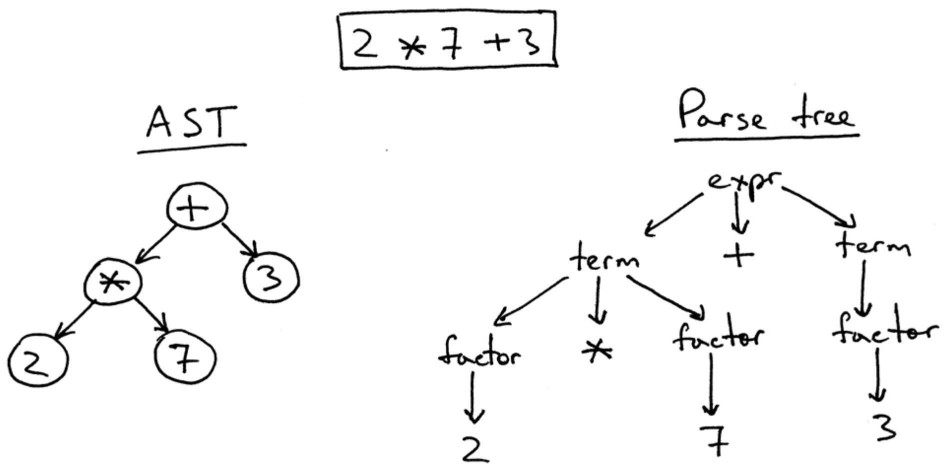
这是为表达式 14 + 2 * 3 - 6 / 2 生成的图像 parsetree.png：



通过向它传递不同的算术表达式来稍微玩一下该实用程序，并查看特定表达式的解析树是什么样的。

现在，让我们谈谈抽象语法树(AST)。这是我们将在本系列的其余部分大量使用的中间表示(IR)。它是我们解释器和未来编译器项目的核心数据结构之一。

让我们通过查看表达式 2 * 7 + 3的AST和解析树来开始我们的讨论：



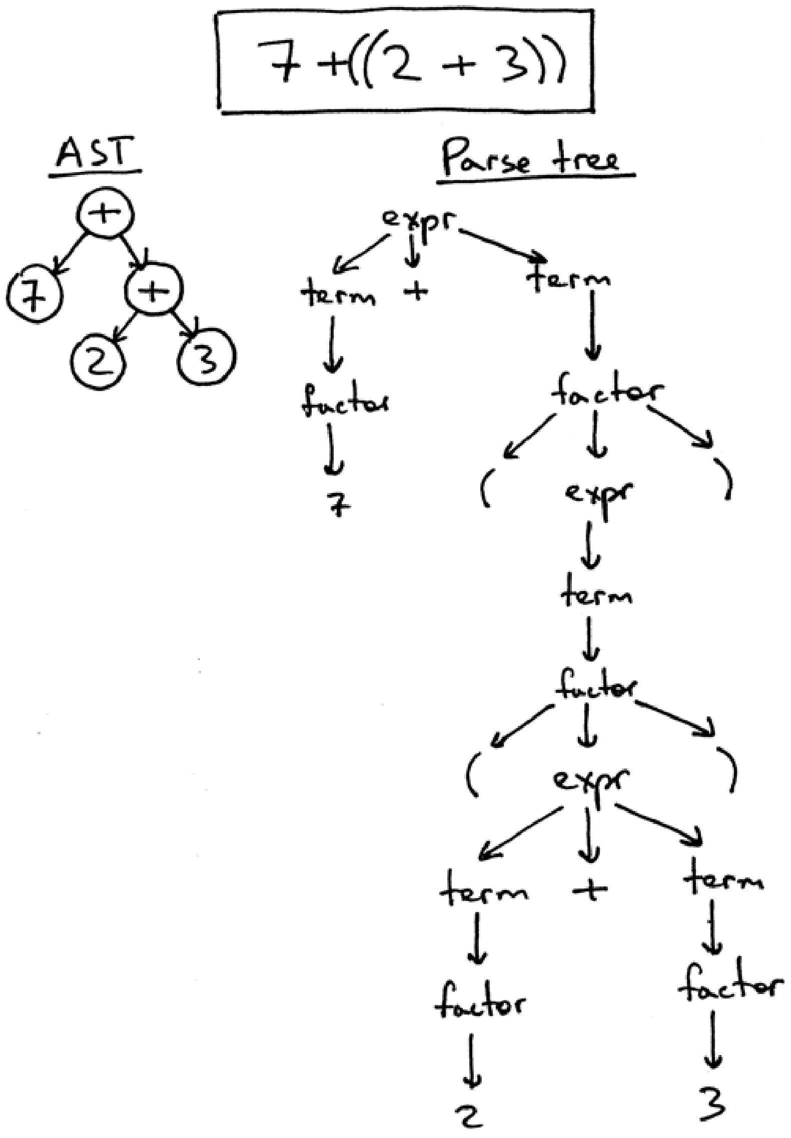
从上图可以看出，AST在变小的同时捕捉到了输入的本质。

以下是 AST 和解析树之间的主要区别：

- AST 使用操作符/操作作为根节点和内部节点，并使用操作数作为它们的子节点。
- 与解析树不同，AST 不使用内部节点来表示语法规则。
- AST 并不代表真实语法中的每一个细节（这就是为什么它们被称为abstract）——例如，没有规则节点和括号。
- 与相同语言构造的解析树相比，AST 是密集的。

那么，什么是抽象语法树呢？一个抽象语法树（AST）是表示一个语言结构，其中每个内部节点和根节点表示操作者的抽象句法结构的树，并且节点的子节点表示操作者的操作数。

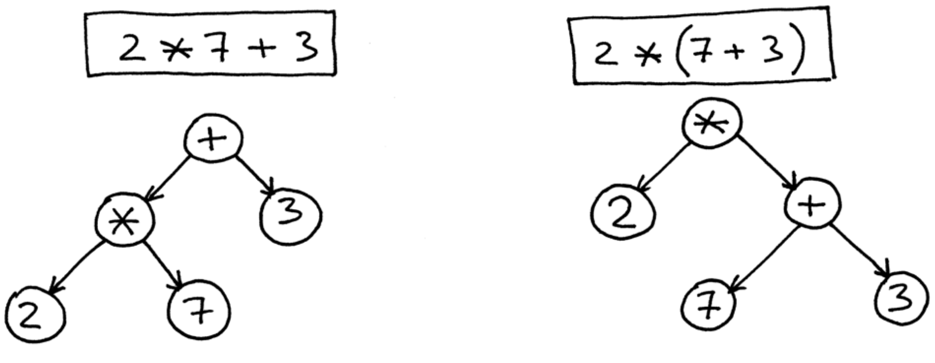
我已经提到 AST 比解析树更紧凑。让我们看一下表达式 7 + ((2 + 3))的AST和解析树。可以看到下面的AST比解析树小很多，但还是抓住了输入的本质：



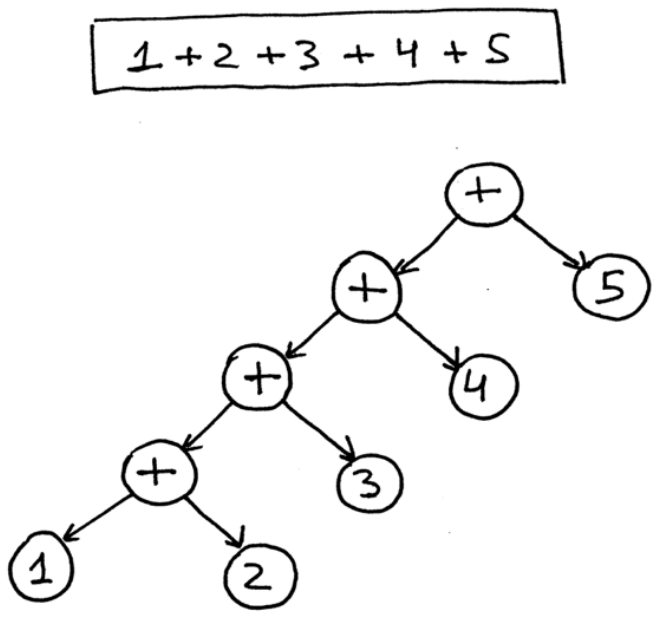
到目前为止一切顺利，但是您如何在AST 中编码运算符优先级？为了在AST 中对运算符优先级进行编码，即表示“X 发生在 Y 之前”，您只需将 X 在树中放在比 Y 低的位置。您已经在前面的图片中看到了这一点。

让我们再看一些例子。

在下图中的左侧，您可以看到表达式 2 * 7 + 3的AST。让我们通过将 7 + 3 放在括号内来更改优先级。您可以在右侧看到修改后的表达式 2 * (7 + 3)的AST是什么样的：



这是表达式 1 + 2 + 3 + 4 + 5的AST：



从上面的图片中，您可以看到优先级较高的运算符在树中的位置较低。

好的，让我们编写一些代码来实现不同的AST节点类型并修改我们的解析器以生成由这些节点组成的AST树。

首先，我们将创建一个名为AST的基节点类，其他类将从该类继承：

```
class AST(object):
    pass
```

Not much there, actually. Recall that ASTs represent the operator-operand model. So far, we have four operators and integer operands. The operators are addition, subtraction, multiplication, and division. We could have created a separate class to represent each operator like AddNode, SubNode, MulNode, and DivNode, but instead we’ re going to have only one BinOp class to represent all four binary operators (a binary operator is an operator that operates on two operands):

```
class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right
```

The parameters to the constructor are left, op, and right, where left and right point correspondingly to the node of the left operand and to the node of the right operand. Op holds a token for the operator itself: Token(PLUS, ‘+’) for the plus operator, Token(MINUS, ‘-‘) for the minus operator, and so on.

To represent integers in our AST, we’ ll define a class Num that will hold an INTEGER token and the token’ s value:

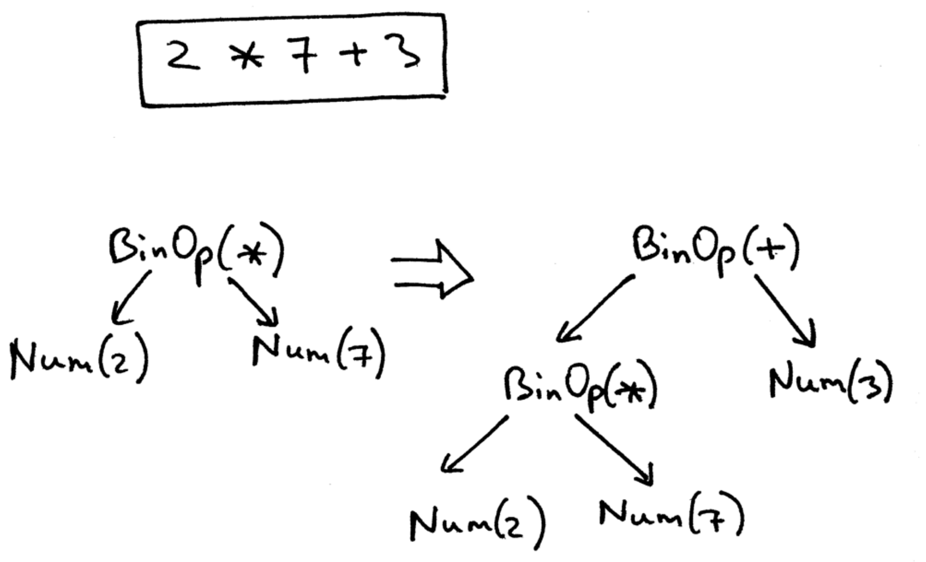
```
class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

As you’ ve noticed, all nodes store the token used to create the node. This is mostly for convenience and it will come in handy in the future.

Recall the AST for the expression 2 * 7 + 3. We’ re going to manually create it in code for that expression:

```
>>> from spi import Token, MUL, PLUS, INTEGER, Num, BinOp
>>>
>>> mul_token = Token(MUL, '*')
>>> plus_token = Token(PLUS, '+')
>>> mul_node = BinOp(
...     left=Num(Token(INTEGER, 2)),
...     op=mul_token,
...     right=Num(Token(INTEGER, 7))
... )
>>> add_node = BinOp(
...     left=mul_node,
...     op=plus_token,
...     right=Num(Token(INTEGER, 3))
... )
```

Here is how an AST will look with our new node classes defined. The picture below also follows the manual construction process above:

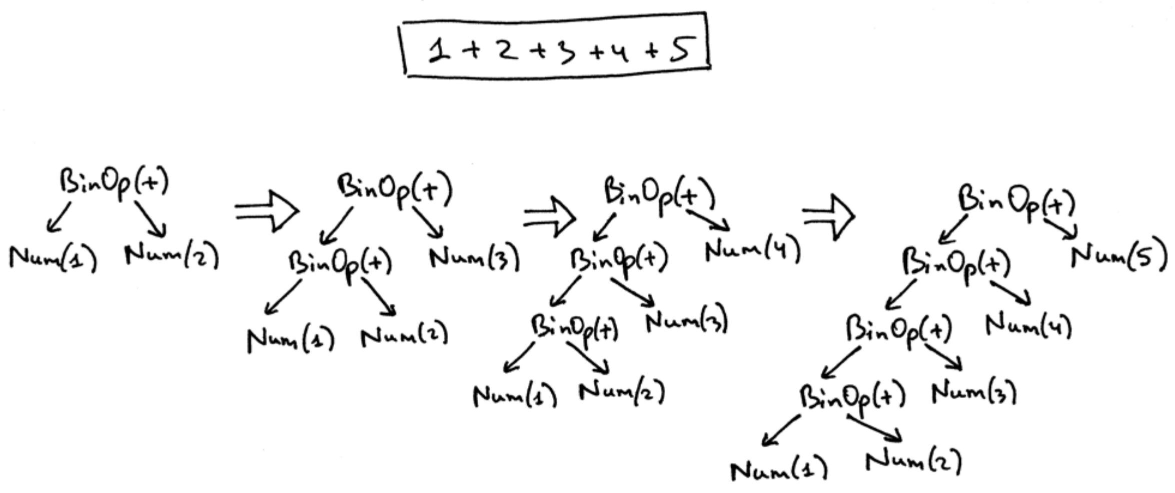


Here is our modified parser code that builds and returns an AST as a result of recognizing the input (an arithmetic expression):

```
class AST(object):
    pass


class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right


class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value


class Parser(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')

    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()

    def factor(self):
        """factor : INTEGER | LPAREN expr RPAREN"""
        token = self.current_token
        if token.type == INTEGER:
            self.eat(INTEGER)
            return Num(token)
        elif token.type == LPAREN:
            self.eat(LPAREN)
            node = self.expr()
            self.eat(RPAREN)
            return node

    def term(self):
        """term : factor ((MUL | DIV) factor)*"""
        node = self.factor()

        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
            elif token.type == DIV:
                self.eat(DIV)

            node = BinOp(left=node, op=token, right=self.factor())

        return node

    def expr(self):
        """
        expr   : term ((PLUS | MINUS) term)*
        term   : factor ((MUL | DIV) factor)*
        factor : INTEGER | LPAREN expr RPAREN
        """
        node = self.term()

        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
            elif token.type == MINUS:
                self.eat(MINUS)

            node = BinOp(left=node, op=token, right=self.term())

        return node

    def parse(self):
        return self.expr()
```

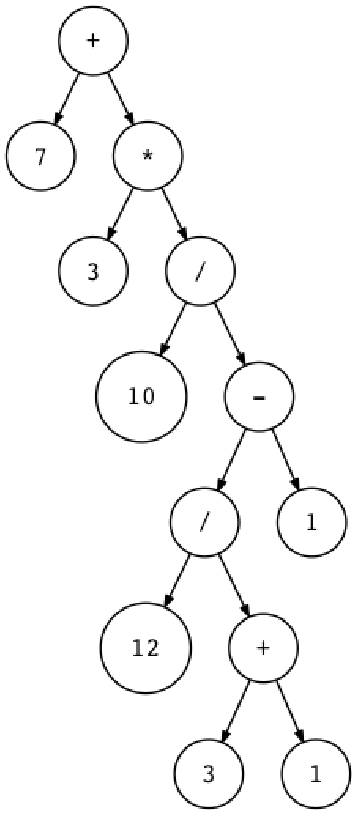
Let’ s go over the process of an AST construction for some arithmetic expressions.

If you look at the parser code above you can see that the way it builds nodes of an AST is that each BinOp node adopts the current value of the node variable as its left child and the result of a call to a term or factor as its right child, so it’ s effectively pushing down nodes to the left and the tree for the expression 1 +2 + 3 + 4 + 5 below is a good example of that. Here is a visual representation how the parser gradually builds an AST for the expression 1 + 2 + 3 + 4 + 5:



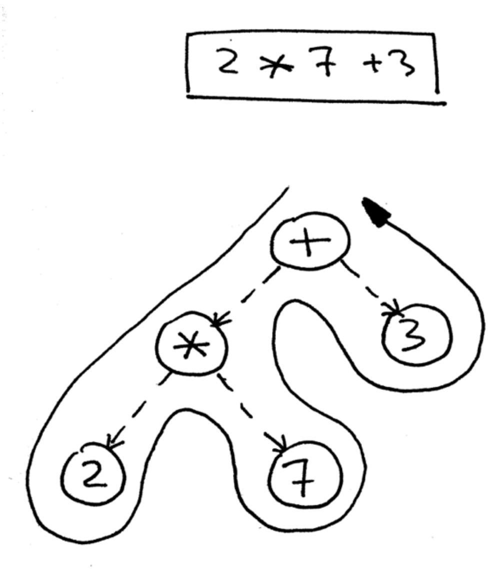
To help you visualize ASTs for different arithmetic expressions, I wrote a small utility that takes an arithmetic expression as its first argument and generates a DOT file that is then processed by the dot utility to actually draw an AST for you (dot is part of the Graphviz (http://graphviz.org) package that you need to install to run the dot command). Here is a command and a generated AST image for the expression 7 + 3 * (10 / (12 / (3 + 1) - 1)):

```
$ python genastdot.py "7 + 3 * (10 / (12 / (3 + 1) - 1))" > \
  ast.dot && dot -Tpng -o ast.png ast.dot
```



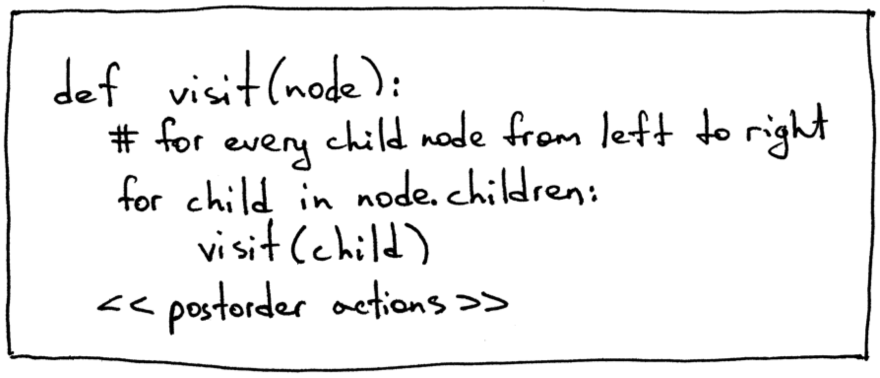
It’s worth your while to write some arithmetic expressions, manually draw ASTs for the expressions, and then verify them by generating AST images for the same expressions with the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part7/python/genastdot.py) tool. That will help you better understand how ASTs are constructed by the parser for different arithmetic expressions.

Okay, here is an AST for the expression 2 * 7 + 3:



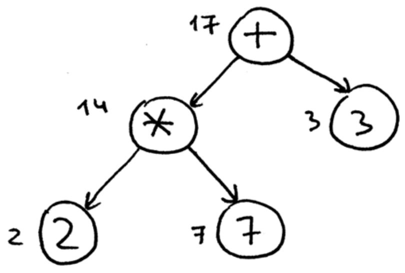
您如何导航树以正确评估该树表示的表达式？您可以通过使用后序遍历（深度优先遍历的一种特殊情况）来实现这一点，它从根节点开始并从左到右递归访问每个节点的子节点。后序遍历尽可能快地访问远离根的节点。

这是后序遍历的伪代码，其中<<postorder actions>>是BinOp节点的加法、减法、乘法或除法等操作的占位符，或者是返回Num 节点的整数值等更简单的操作：

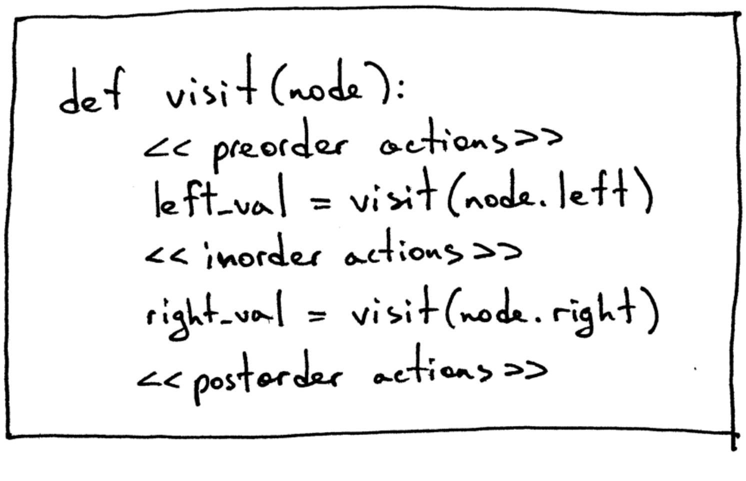


```
def visit(node):
    # for every child node from left to right
    for child in node.children:
        visit(child)
    << postorder actions >>
```

我们要为解释器使用后序遍历的原因是，首先，我们需要评估树中较低的内部节点，因为它们代表具有更高优先级的运算符；其次，我们需要在应用运算符之前评估运算符的操作数到那些操作数。在下图中，您可以看到，通过后序遍历，我们首先计算表达式 2 * 7，然后才计算 14 + 3，得到正确结果 17：



为了完整起见，我将提到深度优先遍历的三种类型：前序遍历、中序遍历和后序遍历。遍历方法的名字来自你在访问代码中放置动作的地方：



```
def visit(node):
    << preorder actions >>
    left_val = visit(node.left)
    << inorder actions >>
    right_val = visit(node.right)
    << postorder actions >>
```

Sometimes you might have to execute certain actions at all those points (preorder, inorder, and postorder). You'll see some examples of that in the source code repository for this article.

Okay, let's write some code to visit and interpret the abstract syntax trees built by our parser, shall we?

Here is the source code that implements the Visitor pattern (https://en.wikipedia.org/wiki/Visitor_pattern):

```
class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))
```

And here is the source code of our Interpreter class that inherits from the NodeVisitor class and implements different methods that have the form visit_NodeType, where NodeType is replaced with the node’s class name like BinOp, Num and so on:

```
class Interpreter(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinOp(self, node):
        if node.op.type == PLUS:
            return self.visit(node.left) + self.visit(node.right)
        elif node.op.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        elif node.op.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        elif node.op.type == DIV:
            return self.visit(node.left) / self.visit(node.right)

    def visit_Num(self, node):
        return node.value
```

There are two interesting things about the code that are worth mentioning here: First, the visitor code that manipulates AST nodes is decoupled from the AST nodes themselves. You can see that none of the AST node classes (BinOp and Num) provide any code to manipulate the data stored in those nodes. That logic is encapsulated in the Interpreter class that implements the NodeVisitor class.

Second, instead of a giant if statement in the NodeVisitor’s visit method like this:

```
def visit(node):
    node_type = type(node).__name__
    if node_type == 'BinOp':
        return self.visit_BinOp(node)
    elif node_type == 'Num':
        return self.visit_Num(node)
    elif ...
    # ...
```

or like this:

```
def visit(node):
    if isinstance(node, BinOp):
        return self.visit_BinOp(node)
    elif isinstance(node, Num):
        return self.visit_Num(node)
    elif ...
```

NodeVisitor 的访问方法非常通用，它根据传递给它的节点类型将调用分派到适当的方法。正如我之前提到的，为了使用它，我们的解释器从NodeVisitor类继承并实现了必要的方法。所以如果传递给visit方法的节点类型是BinOp，那么visit方法会将调用分派到visit_BinOp方法；如果节点类型是Num，那么visit方法会将调用分派到visit_Num方法， 等等。

花一些时间研究这种方法（标准 Python 模块ast (https://docs.python.org/2.7/library/ast.html#module-ast)使用相同的节点遍历机制），因为将来我们将使用许多新的visit_NodeType方法扩展我们的解释器。

该generic_visit方法是抛出一个异常，表明它遇到的实现类没有相应的节点的后备visit_NodeType方法。

现在，让我们为表达式 2 * 7 + 3手动构建一个AST，并将其传递给我们的解释器，以查看访问方法的作用，以评估表达式。以下是从 Python shell 执行此操作的方法：

```
>>>  from  spi  import  Token ,  MUL ,  PLUS ,  INTEGER ,  Num ,  BinOp
>>>
>>>  mul_token  =  Token ( MUL ,  '*' )
>>>  plus_token  =  Token ( PLUS ,  '+' )
>>>  mul_node  =  BinOp (
...      left = Num ( Token ( INTEGER ,  2 )),
...      op =mul_token ,
...      right = Num ( Token ( INTEGER ,  7 ))
...  )
>>>  add_node  =  BinOp (
...      left = mul_node ,
...      op = plus_token ,
...      right = Num ( Token ( INTEGER ,  3 ))
...  )
>>>  from  spi  import  Interpreter
>>>  inter  =  Interpreter（无）
>>> 间。访问( add_node )
17
```

如您所见，我将表达式树的根传递给了访问方法，并通过将调用分派到解释器类的正确方法（visit_BinOp和visit_Num）并生成结果来触发树的遍历。

好的，为了您的方便，这是我们新解释器的完整代码：

```
""" SPI - 简单的 Pascal 解释器 """

############################################### #############################
##
＃LEXER＃
＃＃
############## ############################################### ###############

# 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):
        # client string input, e.g. "4 + 2 * 3 - 6 / 2"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '/':
                self.advance()
                return Token(DIV, '/')

            if self.current_char == '(':
                self.advance()
                return Token(LPAREN, '(')

            if self.current_char == ')':
                self.advance()
                return Token(RPAREN, ')')

            self.error()

        return Token(EOF, None)


###############################################################################
#                                                                             #
#  PARSER                                                                     #
#                                                                             #
###############################################################################

class AST(object):
    pass


class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right
```

```
class  Num ( AST ):
    def  __init__ ( self ,  token ):
        self 。令牌 = 令牌
        自我。价值 = 令牌。价值


类 解析器（对象）：
    def  __init__ （自我， 词法分析器）：
        自我。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( INTEGER )
            返回 Num (令牌)
        elif 令牌。类型 ==  LPAREN :
            self 。吃（LPAREN ）
            节点 = 自我。expr ()
            自我。吃（RPAREN ）
            返回 节点

    def  term ( self ):
        """term : factor ((MUL | DIV) factor)*"""
        node  =  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 )
            elif 令牌。类型 ==  DIV ：
                自我。吃（DIV ）

            node  =  BinOp ( left = node ,  op = token ,  right = self . factor ())

        返回 节点

    def  expr ( self ):
        """
        expr : term ((PLUS | MINUS) term)*
        term : factor ((MUL | DIV) factor)*
        factor : INTEGER | LPAREN expr RPAREN
        """
        node  =  self 。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 )
            elif 令牌。类型 == 减号：
                自我。吃（减）
            node = BinOp(left=node, op=token, right=self.term())

        return node

    def parse(self):
        return self.expr()


###############################################################################
#                                                                             #
#  INTERPRETER                                                                #
#                                                                             #
###############################################################################

class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))


class Interpreter(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinOp(self, node):
        if node.op.type == PLUS:
            return self.visit(node.left) + self.visit(node.right)
        elif node.op.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        elif node.op.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        elif node.op.type == DIV:
            return self.visit(node.left) / self.visit(node.right)

    def visit_Num(self, node):
        return node.value

    def interpret(self):
        tree = self.parser.parse()
        return self.visit(tree)


def  main ():
    while  True :
        try :
            try :
                text  =  raw_input ( 'spi> ' )
            except  NameError :   # Python3
                text  =  input ( 'spi> ' )
        except  EOFError :
            break
        if  not  text :
            continue

        lexer  =  Lexer ( text )
        parser  =  Parser ( lexer )
        interpreter  =  Interpreter ( parser )
        result  =  interpreter 。解释（）
        打印（结果）
```

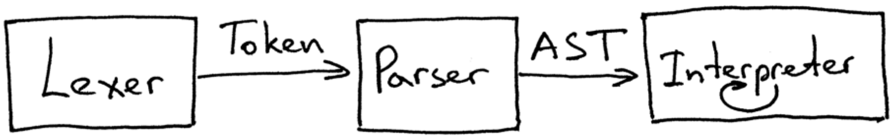
```
如果 __name__ == '__main__' :
    main ()
```

将上述代码保存到spi.py文件中或直接从GitHub (https://github.com/rspivak/lsbasi/blob/master/part7/python/spi.py)下载。尝试一下，亲眼看看您的新的基于树的解释器是否正确评估了算术表达式。

这是一个示例会话：

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

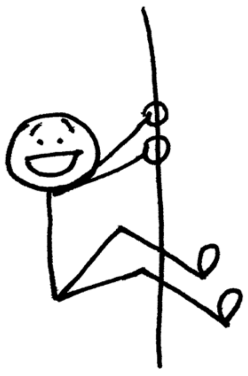
今天，您已经了解了解析树、AST、如何构造 AST 以及如何遍历它们以解释由这些 AST 表示的输入。您还修改了解析器和解释器并将它们分开。词法分析器、解析器和解释器之间的当前接口现在看起来像这样：



您可以将其理解为“解析器从词法分析器中获取标记，然后返回生成的AST以供解释器遍历和解释输入。”

今天就到此为止，但在结束之前，我想简要谈谈递归下降解析器，即给它们一个定义，因为我上次承诺会更详细地讨论它们。所以你开始了：递归下降解析器是一个自顶向下的解析器，它使用一组递归过程来处理输入。自顶向下反映了解析器从构造解析树的顶部节点开始，然后逐渐构造较低节点的事实。

现在是练习的时候了:)



- 编写一个翻译器（提示：节点访问者），将算术表达式作为输入并以后缀表示法打印出来，也称为反向波兰表示法（RPN）。例如，如果翻译器的输入是表达式 (5 + 3) * 12 / 3，那么输出应该是 5 3 + 12 * 3 /。请在此处 (https://github.com/rspivak/lsbasi/blob/master/part7/python/ex1.py)查看答案，(https://github.com/rspivak/lsbasi/blob/master/part7/python/ex1.py)但请先尝试自己解决。
- 编写一个翻译器（节点访问者），将算术表达式作为输入并以LISP风格的符号打印出来，即 2 + 3 将变成 (+ 2 3) 并且 (2 + 3 * 5) 将变成 (+ 2 (* 3 5))。您可以在此处 (https://github.com/rspivak/lsbasi/blob/master/part7/python/ex2.py)找到答案，

(https://github.com/rspivak/lsbasi/blob/master/part7/python/ex2.py)但在查看提供的解决方案之前再次尝试先解决它。

在下一篇文章中，我们将向不断增长的 Pascal 解释器添加赋值和一元运算符。在那之前，玩得开心，很快就会见到你。

PS我还提供了解释器的 Rust 实现，您可以在GitHub (https://github.com/rspivak/lsbasi/blob/master/part7/rust/spi/src/main.rs)上找到 (https://github.com/rspivak/lsbasi/blob/master/part7/rust/spi/src/main.rs)。这是我学习Rust 的 (https://www.rust-lang.org/)一种方式，所以请记住，代码可能还不是“惯用的”。关于如何改进代码的意见和建议总是受欢迎的。

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/)

注释

ALSO ON **RUSLAN'S BLOG**

Let's Build A Simple Interpreter. Part 18: …
2 years ago • 8 comments
Do the best you can until you know better. Then when you know better, do …

Let's Build A Web Server. Part 2.
6 years ago • 61 comments
Remember, in Part 1 I asked you a question: "How do you run a Django application, …

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 1.
6 years ago • 61 comments
"If you don't know how compilers work, then you don't know how …

Let's Interp
4 years
Anythi worth doing

23 Comments **Ruslan's Blog** **Disqus' Privacy Policy** 1 **Login**

Recommend 8 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Just Rinat • a year ago
This seems like a bad implementation of the visitor pattern. From what I read, the receiving objects are supposed to have an .accept() method which then direct the visitor to the appropriate function - you ended up handling the .accept() logic in the visitor itself. Correct me if I am wrong because I just went off a few explanations that I quickly found.
1 | • Reply • Share ›

Wang Xin Just Rinat • 6 months ago
Have the same question about this, I am curious if there is a better implementation
| • Reply • Share ›

bressonnemesis • 3 years ago
Very interesting and informative but you have to take it one step further - how to handle precedence in math without parentheses? Or do you discuss that somewhere else on the blog? Thank you for for taking the time to write this blog.
1 | • Reply • Share ›

Anqur Vanillapy • 6 years ago
I found that some syntax errors like `1 +` will raise the exception from `generic_visit()` method giving no-NoneType-method message, so maybe this kind of exception issue could be great for learning and practicing. I also found that some expression like `1 (1+2)` will return `1` which seemed to generate non-relative ASTs (not sure). But still, it takes time to consume the new things. The learning curve of this tutorial is pretty nice and I really really can't wait for the next post! Great thx!
1 ^ | ˅ • Reply • Share ›

rspivak Mod ➦ Anqur Vanillapy • 6 years ago
Thank you for reading and trying out the code.

I've updated the parser to throw an error if there are unconsumed tokens left after the parsing has been finished, to deal with cases like the one you brought up: https://github.com/rspivak/...

Proper error handling (error detection, displaying a useful error message, and recovery) is something I hope to add in the near future.
1 ^ | ˅ • Reply • Share ›

Pavel Karateev ➦ rspivak • 6 years ago
Can we add `self.error()` as a last line to `factor` method to capture syntax errors like `1 +` before Interpreter do it?

I also found it is sensible and more testable to use concrete python exceptions like `SyntaxError` instead of `Exception`

And thank you very much for this awesome tutorial! It's absolutely outstanding =)
^ | ˅ • Reply • Share ›

Quevidia • 5 months ago
For those that are writing this in C# and are stuck on the NodeVisitor part, here is the visit code that I decided to use (with .NET 5.0) which should work if a NodeVisitor class exits and the interpreter class inherits off of it:

```
internal object Visit(AST Node) =>
GetType().InvokeMember($"visit_{Node.GetType().Name}", BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.InvokeMethod, null, this, new object[] { Node });
```

If the visitor methods used are public, change BindingFlags.NonPublic to BindingFlags.Public. Essentially, what this does is that it gets the type of itself (in this case, the interpreter class), and then the InvokeMember() method is called. The first argument is obviously the method name. The second argument is all of the flags specified, which defines what InvokeMember() should look for and what it should do. BindingFlags.InvokeMethod is the key here, as this will call the specified method and return `object` depicting the value that the method returned - which should be an int, at least in my case. The binder (third argument) is not mandatory in this case, and it can just be nulled out. The fourth argument is the target class containing the method (again, the interpreter class). Finally, the fifth argument is an array of objects, which will be supplemented as the method's argumemts.

Hope this helps!
^ | ˅ • Reply • Share ›

selim öztürk • 2 years ago
thank you Ruslan, much appreciated
^ | ˅ • Reply • Share ›

Yuipas • 3 years ago
I know, a bit late, but I have a question. Why do you make the methods visit_class in the interpreter, instead in the class itself?
^ | ˅ • Reply • Share ›

Anon ➦ Yuipas • 7 months ago
It's mainly just to keep all the different functions (behaviors) that are more related to the interpreter than to the AST together. An AST shouldn't ideally have to care about how it's going to be evaluated, so we put all that logic together. It just helps us avoid a giant "if isinstance" chain.
^ | ˅ • Reply • Share ›

wandeber ➜ Yuipas • a year ago

In case someone else has the same doubt, what I think: Interpreter is a node visitor class since it extends from NodeVisitor. NodeVisitor is a generic class that allows you to define several and different interpreters with different outputs for the same input.

By the way, that's what excercises say you have to do.

^ | ⌄ • Reply • Share ›

Nyarmith • 4 years ago

Thanks for giving such cool exercises. I would never have thought to do something like translating from the AST, and the result is really cool and gratifying!

^ | ⌄ • Reply • Share ›

Shreeya Patel • 4 years ago

Very nicely explained :)

^ | ⌄ • Reply • Share ›

Zagros Samet • 5 years ago

Hi, Thank you very much for this very interesting and useful project. I have never seen such complex ideas explained so clearly. I was wondering how the AST will look like in c++ ?

^ | ⌄ • Reply • Share ›

Jake Nottellingyou • 5 years ago

How would we go about changing the format of the calculator. Like what if i want to start by entering my expressions in reverse polish or LISP notation. So the initial input can be "(+ 1 2)" or "1 2 +" ?

^ | ⌄ • Reply • Share ›

Nikos C. • 6 years ago

How many parts is this excellent series estimated to be?

^ | ⌄ • Reply • Share ›

Pekka Klärck • 6 years ago

Awesome series! Looking forward for future posts.

I noticed the lexer doesn't handle empty input too well. That could be fixed, and code simplified elsewhere, by making `current_char` a Python property. Would also consider using an empty string instread of `None` when all input is consumed to allow using, for example, `char.isspace()` instead of `char is not None and char.isspace()`. I can submit a PR if you want.

^ | ⌄ • Reply • Share ›

Dima • 6 years ago

Thanks for the series. There are a few problems though. While first posts were introducing info slowly, explaining every aspect and practicing problems, this post is so full of new info, that it can be easily split into several posts. Also, what was the point of introducing the Visitor pattern in this particular tutorial, which is being positioned as a layman compilers course?

^ | ⌄ • Reply • Share ›

Diego Marcia ➜ Dima • 5 years ago • edited

I believe it was just for sake of code tidiness... The way it's presented, it relies on reflection:
visitor = getattr(self, method_name, self.generic_visit)
Not every language has this feature, so I believe it's just a matter of aesthetics...

^ | ⌄ • Reply • Share ›

solstice333 ➜ Diego Marcia • 2 years ago • edited

The way visitor pattern is used here (with introspection/reflection and decorating/mangling of `method_name`) is needed b.c. python (unlike a statically typed language) doesn't support function overloading. In other words, python cannot disambiguate between methods with the same name but different signatures (number and type of arguments). Additionally, with visitor pattern, it moves the "interpretation" behavior to the front as a way of extending behavior of the AST nodes (AST, BinOp, Num). Without the visitor pattern, the "interpretation behavior" would exist as AST methods. Instead, with the visitor pattern, it decouples the logical domains so that AST nodes only contain state, while the interpreter class contains the "extended AST" behavior, which is supposed to make things easier to maintain or read later; but I suppose that's ultimately a matter of opinion.

1 ^ | ⌄ • Reply • Share ›

chaojie ➦ Diego Marcia • 2 years ago
thank you.
^ | ⌄ • Reply • Share ›

Matteo Smaila ➦ Diego Marcia • 3 years ago
That thing BLEW-MY-MIND-AWAY.
I don't know Python, I'm not a skilled programmer, not in any language. I just attended a Java beginner course at the university and I didn't even get to lambda expressions, nor even generics... and studied just bits of stuff here and there about C or C++.

Thanks for saying it's called reflection cause I was struggling so hard to find if Java implements a sort of... that, and how, as I'm following these articles coding everything in Java. And there's nothing worse than knowing what you're looking for without knowing how that thing is called.

More on, I always thought that would be such a powerful tool in programming, and I'm glad to come to know it is there for us to use it.
^ | ⌄ • Reply • Share ›

Dan Boxall ➦ Matteo Smaila • 3 years ago • edited
Any chance i can see your code? I've been at this for ages. and I really can't work it out now
^ | ⌄ • Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

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