链滴

《Let'"'"'s Build a Simple Interpreter》系列文章翻译： Part 2

作者：StephenZhang

原文链接：https://ld246.com/article/1597884968538

来源网站：链滴

在他们的神奇的书《有效思考的五个因素》中，作者Burger[1]和Starbird分享了一个关于如何观察Tony Plog这位国际小号演奏大师的故事，他为出色的小号演奏家举办了一个大师班。在这里，学生们首先奏复杂的、那些他们已经可以演奏的很好的乐章。然后当他们被要求演奏非常基本、简单的乐章时，之前复杂的乐章相比，那些音符听起来却显得非常幼稚。学生们演奏完后，大师也演奏了这些基本的章，但是却并不显得幼稚。这其中的差异是惊人的。Tony解释说，掌握简单音符的演奏可以让一个人演奏复杂乐曲时有更大的控制力。这个教训很清楚——要想拥有真正的艺术技能，你必须专注于掌握单的、基本的思想。

这个故事中的道理不仅仅适用于音乐，而且也适用于软件开发。这个故事很好的提醒了我们所有人，要忽视基本思想在深入工作的重要性，即便是有时感觉像后退了一步似的。虽然精通你使用的工具和架很重要，但是了解其背后的原理也很重要。正如Ralph Waldo Emerson说的那样：

> “如果你只学习方法，那么你将会被方法困扰；不过你若学习原理，那么你可以拥有自己的方法。”

关于这一点，让我们再次更深入的去学习解释器和编译器。

今天我会展示给你基于Part 1的计算器的新版本，它即将可以做到：

1. 处理输入字符串中任意的空白字符
2. 处理输入中的多位整形数字
3. 支持两个整形的减法（目前只支持加法）

这是你的新版计算器的源代码，它已经实现了上述所有功能：

```
# Token types
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
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
        # token value: non-negative integer value, '+', '-', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

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


class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3 + 5", "12 - 5", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        # current token instance
        self.current_token = None
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Error parsing input')

    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.
    """
    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, '-')

        self.error()

    return Token(EOF, None)

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.get_next_token()
    else:
        self.error()

def expr(self):
    """Parser / Interpreter

    expr -> INTEGER PLUS INTEGER
    expr -> INTEGER MINUS INTEGER
    """
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    # we expect the current token to be an integer
    left = self.current_token
    self.eat(INTEGER)

    # we expect the current token to be either a '+' or '-'
    op = self.current_token
    if op.type == PLUS:
```

```
            self.eat(PLUS)
        else:
            self.eat(MINUS)

        # we expect the current token to be an integer
        right = self.current_token
        self.eat(INTEGER)
        # after the above call the self.current_token is set to
        # EOF token

        # at this point either the INTEGER PLUS INTEGER or
        # the INTEGER MINUS INTEGER sequence of tokens
        # has been successfully found and the method can just
        # return the result of adding or subtracting two integers,
        # thus effectively interpreting client input
        if op.type == PLUS:
            result = left.value + right.value
        else:
            result = left.value - right.value
        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)


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

把这段代码保存为calc2.py或者从我的GitHub下载。尝试运行它，并且自己观察它是否按照预期工作它是否可以处理输入中任意位置、任意数量的空格？它是否可以接受两位及以上的整形？它是否可以求两数之和那样求两数之差？

这是在我的笔记本上运行的结果：

```
$ python calc2.py
calc> 27 + 3
30
calc> 27 - 7
20
calc>
```

与Part 1相比，代码的主要变动在于：

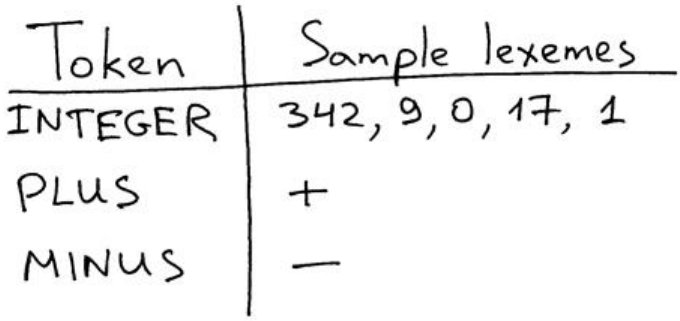
1. 方法 get_next_token()进行了轻微的重构。逻辑指针pos增加的逻辑被独立成为一个新的方法advance()；

2. 添加了两个新方法，方法 skip_whitespace()用于忽略空白字符，方法integer()用于处理输入中位及以上的整形数字；

3. 方法 expr()在原来识别INTEGER -> PLUS -> INTEGER序列的基础上添加了对INTEGER -> MINUS -> INTEGER序列的识别。现在该方法可以在成功识别到正确的序列后执行加法或者减法运算。

在Part 1中你已经了解到两个重要的概念，即Token和词法分析器。今天我会讨论一些关于词素（Lexeme）、解析和解析器相关的知识。

你已经知道了Token的概念，但是为了能完美结束对Token的讨论，词素的概念是必不可少的。词素什么？词素是从Token中提取到的一串字符序列。下图中你将可以看到一些Token和词素的示例，而它将会使二者的关系更加清晰：



现在，还记得我们的老朋友，expr()方法吗？之前我说过这是算术表达式被实际解释执行的地方。但在你解释执行表达式之前，你首先需要知道你到底识别到了什么样的表达式，例如，它是加法还是减？这就是expr()实际上在做的：它从方法get_next_token()给出的Token流中搜索语法结构，然后对别到的语法结构进行解释执行，并生成算是表达式的结果。

在Token流中寻找语法结构的过程，或者换句话说，识别Token流中“短语”的过程，叫做解析；解器或编译器中执行这一部分工作的部分叫做解析器。

这样，你知道方法expr()在你的解释器中既充当解析器，也充当执行器——该方法首先尝试在Token中识别（解析）INTEGER -> PLUS -> INTEGER或者INTEGER -> MINUS -> INTEGER，在成功识（解析）到其中一个“短语”后，该方法就对它解释执行，并把两个整数进行加减运算的结果返回给用者。

现在又到了做练习的时间了。

1. 扩展这个计算器，使之支持两个整数的乘法运算；
2. 扩展这个计算器，使之支持两个整数的除法运算；
3. 修改代码使之可以执行任意数目的数字进行加减运算，例如$9 - 5 + 3 + 11$

检查你的理解程度：

1. 词素是什么？
2. 在Token流中寻找语法结构的过程，或者换句话说，识别Token流中“短语”的过程 被称作什么？
3. 进行上一个问题中工作的解释器（或编译器）组件的名字是什么？

我希望你喜欢今天的学习材料，在本系列的下一篇文章中，你将学习到如何扩展你的计算器以至于可处理更多的算术表达式。请继续关注。

这里是我推荐的一份书单，它们将会对你的学习有所帮助：

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)
2. Writing Compilers and Interpreters: A Software Engineering Approach
3. Modern Compiler Implementation in Java
4. Modern Compiler Design
5. Compilers: Principles, Techniques, and Tools (2nd Edition)

1. 人名不做翻译，下同