链滴

Python-CookBook：38、编写一个简单的递归下降解析器

作者：zhaolixiang

原文链接：https://ld246.com/article/1566977293963

来源网站：链滴

问题

我们需要根据一组语法规则来解析文本，以此执行相应的操作或构建一个抽象语法树来表示输入。语规则很简单，因此我们倾向于自己编写解析器而不是使用某种解析器框架。

解决方案

在这个问题中，我们把重点放在根据特定的语法来解析文本上。要做到这些，应该以BNF或EBNF的式定义出语法的正式规格。比如，对于简单的算术运算表达式，语法看起来是这样的：

```
expr ::= expr + term
    | expr - term
    | term
term ::= term * factor
    | term / factor
    | factor
factor ::= ( expr )
     | NUM
```

又或者以EBNF的形式定义为如下形式：

```
expr ::= term { (+|-) term }*
term ::= factor { (*|/) factor }*
factor ::= ( expr )
    | NUM
```

在EBNF中，部分包括在{ ... }*中的规则是可选的。*意味着零个或更多重复项（和在正则表达式中的义相同）。

现在，如果我们对BNF还不熟悉的话，可以把它看做是规则替换或取代的一种规范形式，左侧的符号以被右侧的符号所取代（反之亦然）。一般来说，在解析的过程中我们会尝试将输入的文本同语法做

配，通过BNF来完成各种替换和扩展。为了说明，假设正在解析一个类似于3 + 4 * 5这样的表达式。个表达式首先应该被分解为标记流，这可以使用2.18节中描述的技术来实现。得到的结果可能是下面样的标记序列：

```
NUM + NUM * NUM
```

从这里开始，解析过程就涉及通过替换的方式将语法匹配到输入标记上：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

完成所有的替换需要花上一段时间，这是由输入的规模和尝试去匹配的语法规则所决定的。第一个输标记是一个NUM，因此替换操作首先会把重点放在匹配这一部分上。一旦匹配上了，重点就转移到一个标记+上，如此往复。当发现无法匹配下一个标记时，右手侧的特定部分（{ (/) factor }）就会消。在一个成功的解析过程中，整个右手侧部分会完全根据匹配到的输入标记流来相应地扩展。

有了前面这些基础，下面就向各位展示如何构建一个递归下降的表达式计算器：

```
import re
import collections

# Token specification
NUM    = r'(?P<NUM>\d+)'
PLUS   = r'(?P<PLUS>\+)'
MINUS  = r'(?P<MINUS>-)'
TIMES  = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer
Token = collections.namedtuple('Token', ['type','value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser
class ExpressionEvaluator:
```

```
'''
Implementation of a recursive descent parser. Each method
implements a single grammar rule. Use the ._accept() method
to test and accept the current lookahead token. Use the ._expect()
method to exactly match and discard the next token on on the input
(or raise a SyntaxError if it doesn't match).
'''

def parse(self,text):
    self.tokens = generate_tokens(text)
    self.tok = None             # Last symbol consumed
    self.nexttok = None         # Next symbol tokenized
    self._advance()             # Load first lookahead token
    return self.expr()

def _advance(self):
    'Advance one token ahead'
    self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

def _accept(self,toktype):
    'Test and consume the next token if it matches toktype'
    if self.nexttok and self.nexttok.type == toktype:
        self._advance()
        return True
    else:
        return False

def _expect(self,toktype):
    'Consume next token if it matches toktype or raise SyntaxError'
     if not self._accept(toktype):
         raise SyntaxError('Expected ' + toktype)

# Grammar rules follow

def expr(self):
    "expression ::= term { ('+'|'-') term }*"

    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }*"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
```

```
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
         raise SyntaxError('Expected NUMBER or LPAREN')
```

下面是以交互式的方式使用ExpressionEvaluator类的示例：

```
>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5
>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exprparse.py", line 40, in parse
    return self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 93, in factor
    exprval = self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 97, in factor
    raise SyntaxError("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>
```

如果我们想做的不只是纯粹的计算，那就需要修改ExpressionEvaluator类来实现。比如，下面的实构建了一棵简单的解析树：

```
class ExpressionTreeBuilder(ExpressionEvaluator):
```

```
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                exprval = ('-', exprval, right)
        return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval = ('*', termval, right)
        elif op == 'DIVIDE':
            termval = ('/', termval, right)
        return termval

def factor(self):
    'factor ::= NUM | ( expr )'

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
        raise SyntaxError('Expected NUMBER or LPAREN')
```

下面的示例展示了它是如何工作的：

```
>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>
```

讨论

文本解析是一个庞大的主题，一般会占用学生们编译原理课程的前三周时间。如果你正在寻找有关语

、解析算法和其他相关信息的背景知识，那么应该去找一本编译器方面的图书来读。无需赘言，本书不会重复那些内容的。

然而，要编写一个递归下降的解析器，总体思路还是比较简单的。我们要将每一条语法规则转变为一函数或方法。因此，如果我们的语法看起来是这样的：

```
expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
       | NUM
```

就可以像下面这样将它们转换为对应的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
        ...
    def term(self):
        ...

    def factor(self):
        ...
```

每个方法的任务很简单——必须针对语法规则的每个部分从左到右扫描，在扫描过程中处理符号标记从某种意义上说，这些方法的目的就是顺利地将规则消化掉，如果卡住了就产生一个语法错误。要做这点，需要应用下面这些实现技术。

● 如果规则中的下一个符号标记是另一个语法规则的名称（例如，term或者factor），就需要调用同的方法。这就是算法中的“下降”部分——控制其下降到另一个语法规则中。有时候规则中会涉及调已经在执行的方法（例如，在规则factor ::= '(' expr ')'中对expr的调用）。这就是算法中的“递归”分。

● 如果规则中的下一个符号标记是一个特殊的符号（例如'('），需要检查下一个标记，看它们是否能全匹配。如果不能匹配，这就是语法错误。本节给出的_expect()方法就是用来处理这些步骤的。

● 如果规则中的下一个符号标记存在多种可能的选择（例如+或-），则必须针对每种可能性对下一个记做检查，只有在有匹配满足时才前进到下一步。这就是本节给出的_accept()方法的目的所在。这有像_except()的弱化版，在_accept()中如果有匹配满足，就前进到下一步，但如果没有匹配，它只是单的回退而不会引发一个错误（这样检查才可以继续进行下去）。

● 对于语法规则中出现的重复部分（例如 expr ::=term { ('+' | '-') term }*），这是通过while循环来实的。一般在循环体中收集或处理所有的重复项，直到无法找到更多的重复项为止。

● 一旦整个语法规则都已经处理完，每个方法就返回一些结果给调用者。这就是在解析过程中将值进传递的方法。比如，在计算器表达式中，表达式解析的部分结果会作为值来返回。最终它们会结合在起，在最顶层的语法规则方法中得到执行。

尽管本节给出的例子很简单，但递归下降解析器可以用来实现相当复杂的解析器。例如，Python代本身也是通过一个递归下降解析器来解释的。如果对此很感兴趣，可以通过检查Python源代码中的Grmmar/Grammar文件来一探究竟。即便如此，要自己手写一个解析器时仍然需要面对各种陷阱和局。

局限之一就是对于任何涉及左递归形式的语法规则，都没法用递归下降解析器来解决。例如，假设需解释如下的规则：

```
items ::= items ',' item
        | item
```

要完成这样的解析，我们可能会试着这样来定义items()方法：

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

唯一的问题就是这么做行不通。实际上这会产生一个无穷递归的错误。

我们也可能会陷入到语法规则自身的麻烦中。例如，我们可能想知道表达式是否能以这种加简单的语形式来描述：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
         | NUM
```

这个语法从技术上说是能实现的，但是它却并没有遵守标准算术中关于计算顺序的约定。比如说，表式“3 + 4 * 5”会被计算为35，而不是我们预期的23。因此这里需要单独的“expr”和“term”规来确保计算结果的正确性。

对于真正复杂的语法解析，最好还是使用像PyParsing或PLY这样的解析工具。如果使用PLY的话，解计算器表达式的代码看起来是这样的：

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]

# Ignored characters

t_ignore = ' \t\n'

# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
```

```
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
    | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    '''
    expr : term
    '''
    p[0] = p[1]

def p_term(p):
    '''
    term : term TIMES factor
         | term DIVIDE factor
    '''
    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''
    term : factor
    '''
    p[0] = p[1]

def p_factor(p):
    '''
    factor : NUM
    '''
    p[0] = p[1]

def p_factor_group(p):
    '''
    factor : LPAREN expr RPAREN
    '''
    p[0] = p[2]

def p_error(p):
    print('Syntax error')
```

```
parser = yacc()
```

在这份代码中会发现所有的东西都是以一种更高层的方式来定义的。我们只需编写匹配标记符号的正表达式，以及当匹配各种语法规则时所需要的高层处理函数就行了。而实际运行解析器、接收符号标等都完全由库来实现。

下面是如何使用解析器对象的示例：

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果想在编程中增加一点激动兴奋的感觉，编写解析器和编译器会是非常有趣的课题。再次说明，一编译器方面的教科书会涵盖许多理论之下的底层细节。但是，在网上同样也能找到许多优秀的在线资。Python自带的ast模块也同样值得去看看。