2.19 实现一个简单的递归下降分析器¶

问题¶

你想根据一组语法规则解析文本并执行命令，或者构造一个代表输入的抽象语法树。 如果语法非常简单，你可以不去使用一些框架，而是自己写这个解析器。

解决方案¶

在这个问题中，我们集中讨论根据特殊语法去解析文本的问题。 为了这样做，你首先要以BNF或者EBNF形式指定一个标准语法。 比如，一个简单数学表达式语法可能像下面这样：

```
expr ::= expr + term
   |  expr - term
   |  term

term ::= term * factor
   |  term / factor
   |  factor

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

或者，以EBNF形式：

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

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

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

在EBNF中，被包含在 `{...}*` 中的规则是可选的。*代表0次或多次重复(跟正则表达式中意义是一样的)。

现在，如果你对BNF的工作机制还不是很明白的话，就把它当做是一组左右符号可相互替换的规则。 一般来讲，解析的原理就是你利用BNF完成多个替换和扩展以匹配输入文本和语法规则。 为了演示，假设你正在解析形如 `3 + 4 * 5` 的表达式。 这个表达式先要通过使用2.18节中介绍的技术分解为一组令牌流。 结果可能是像下列这样的令牌序列：

```
NUM + NUM * NUM
```

在此基础上， 解析动作会试着去通过替换操作匹配语法到输入令牌：

```
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
```

下面所有的解析步骤可能需要花点时间弄明白，但是它们原理都是查找输入并试着去匹配语法规则。 第一个输入令牌是NUM，因此替换首先会匹配那个部分。 一旦匹配成功，就会进入下一个令牌+，以此类推。 当已经确定不能匹配下一个令牌的时候，右边的部分(比如 `{ (*/) factor }*`)就会被清理掉。 在一个成功的解析中，整个右边部分会完全展开来匹配输入令牌流。

有了前面的知识背景，下面我们举一个简单示例来展示如何构建一个递归下降表达式求值程序：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 下降解析器
Desc :
"""
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()
```

```
        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')


def descent_parser():
    e = ExpressionEvaluator()
    print(e.parse('2'))
```

```
    print(e.parse('2 + 3'))
    print(e.parse('2 + 3 * 4'))
    print(e.parse('2 + (3 + 4) * 5'))
    # print(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


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

讨论¶

文本解析是一个很大的主题， 一般会占用学生学习编译课程时刚开始的三周时间。 如果你在找寻关于语法，解析算法等相关的背景知识的话，你应该去看一下编译器书籍。 很显然，关于这方面的内容太多，不可能在这里全部展开。

尽管如此，编写一个递归下降解析器的整体思路是比较简单的。 开始的时候，你先获得所有的语法规则，然后将其转换为一个函数或者方法。 因此如果你的语法类似这样：

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

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

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

你应该首先将它们转换成一组像下面这样的方法：

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

每个方法要完成的任务很简单 - 它必须从左至右遍历语法规则的每一部分，处理每个令牌。 从某种意义上讲，方法的目的就是要么处理完语法规则，要么产生一个语法错误。 为了这样做，需采用下面的这些实现方法：

如果规则中的下个符号是另外一个语法规则的名字(比如term或factor)，就简单的调用同名的方法即可。这就是该算法

如果规则中的下个符号是另外一个语法规则的名字(比如term或factor)，就简单的调用同名的方法即可。 这就是该算法中"下降"的由来 - 控制下降到另一个语法规则中去。 有时候规则会调用已经执行的方法(比如，在 `factor ::= '('expr ')'` 中对expr的调用)。 这就是算法中"递归"的由来。
如果规则中下一个符号是个特殊符号(比如()，你得查找下一个令牌并确认是一个精确匹配)。 如果不匹配，就产生一个语法错误。这一节中的 `_expect()` 方法就是用来做这一步的。
如果规则中下一个符号为一些可能的选择项(比如 + 或 -)， 你必须对每一种可能情况检查下一个令牌，只有当它匹配一个的时候才能继续。 这也是本节示例中 `_accept()` 方法的目的。 它相当于_expect()方法的弱化版本，因为如果一个匹配找到了它会继续， 但是如果没找到，它不会产生错误而是回滚(允许后续的检查继续进行)。
对于有重复部分的规则(比如在规则表达式 `::= term { ('+'|'-') term }*` 中)， 重复动作通过一个while循环来实现。 循环主体会收集或处理所有的重复元素直到没有其他元素可以找到。
一旦整个语法规则处理完成，每个方法会返回某种结果给调用者。 这就是在解析过程中值是怎样累加的原理。 比如，在表达式求值程序中，返回值代表表达式解析后的部分结果。 最后所有值会在最顶层的语法规则方法中合并起来。
尽管向你演示的是一个简单的例子，递归下降解析器可以用来实现非常复杂的解析。 比如，Python语言本身就是通过一个递归下降解析器去解释的。 如果你对此感兴趣，你可以通过查看Python源码文件Grammar/Grammar来研究下底层语法机制。 看完你会发现，通过手动方式去实现一个解析器其实会有很多的局限和不足之处。

其中一个局限就是它们不能被用于包含任何左递归的语法规则中。比如，假如你需要翻译下面这样一个规则：

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

为了这样做，你可能会像下面这样使用 `items()` 方法：

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

唯一的问题是这个方法根本不能工作，事实上，它会产生一个无限递归错误。

关于语法规则本身你可能也会碰到一些棘手的问题。 比如，你可能想知道下面这个简单扼语法是否表述得当：

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

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

这个语法看上去没啥问题，但是它却不能察觉到标准四则运算中的运算符优先级。 比如，表达式 `"3 + 4 * 5"` 会得到35而不是期望的23. 分开使用"expr"和"term"规则可以让它正确的工作。

对于复杂的语法，你最好是选择某个解析工具比如PyParsing或者是PLY。 下面是使用PLY来重写表达式求值程序的代码：

```
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_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()
```

这个程序中，所有代码都位于一个比较高的层次。你只需要为令牌写正则表达式和规则匹配时的高阶处理函数即可。 而实际的运行解析器，接受令牌等等底层动作已经被库函数实现了。

下面是一个怎样使用得到的解析对象的例子：

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

如果你想在你的编程过程中来点挑战和刺激，编写解析器和编译器是个不错的选择。 再次，一本编译器的书籍会包含很多底层的理论知识。不过很多好的资源也可以在网上找到。 Python自己的ast模块也值得去看一下。