Recursive Descent Parser for arithmetic expressions with real numbers ¶

https://d.cxcore.net/Python/Python_Cookbook_3rd_Edition.pdf (https://d.cxcore.net/Python/Python_Cookbook_3rd_Edition.pdf)

The Parser

In [1]:
```
import re
import collections
```

In [2]:
```
# Token specification
# ?P is used to name a pattern for later use.
# r'-?\d+\.?\d*e?-?\d*?'
NUM = r'(?P<NUM>-?\d+\.?\d*e?-?\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+)'
```

In [3]:
```
# Use the same pattern to make multiple matches.
# First precompile the pattern string as a pattern object.
master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer
Token = collections.namedtuple('Token', ['type', 'value'])
```

In [4]:
```
def generate_tokens(text):
    # This method creates a scanner object on which the match() method is constantly ...
    # ... called scans the target text step by step, one match at a time.
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok
```

In [5]:
```
# 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 float(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')
```

In [6]:
```
def descent_parser():
    e = ExpressionEvaluator()
    print(e.parse('2'))
    print(e.parse('2.4 + 3'))
    print(e.parse('2 + 3 * 4'))
    print(e.parse('2 + (3.6 + 4) * 5'))
    print(e.parse('2 + (3 + 4) / 5'))

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

```
2.0
5.4
14.0
40.0
3.4
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: