Lab 9

COMP9021, Session 2, 2018

1 Using a stack to evaluate fully parenthesised expressions

Modify the program `postfix.py` from the 9th lecture to a program `fully_parenthesised.py` so that a stack is used to evaluate an arithmetic expression written in infix, fully parenthesised, and built from natural numbers using the binary +, -, * and / operators. *Fully parenthesised* means that all expressions of the form e + e', e - e', e * e' and e / e' are surrounded by a pair of parentheses, brackets or braces. Of course a simple solution would be to replace all brackets and braces by parentheses and call `eval()`, but here we want to use a stack.

Hint: think of popping when and only when a closing parenthesis, bracket or brace is being processed.

Next is a possible interaction.

```
$ python3
...
>>> from fully_parenthesised import *
>>> evaluate('100')
100
>>> evaluate('[(1 - 20) + 300]')
281
>>> evaluate('[1 - {20 + 300}]')
-319
>>> evaluate('( { 20*4 }/5 )')
16.0
>>> evaluate('(20*[4/5])')
16.0
>>> evaluate('({1 + (20 * 30)} - [400 / 500])')
600.2
>>> evaluate('{1 + [((20*30)-400) / 500]}')
1.4
>>> evaluate('[1 + {(2 * (3+{4*5})) / ([6*7]-[8/9]) }]')
2.1189189189189186
>>> evaluate('100 + 3')
>>> evaluate('(100 + 3')
>>> evaluate('(100 + -3)')
>>> evaluate('(100 @ 50)')
>>> evaluate('(100 / 0)')
```

2 Word ladders

Write a program word_ladder.py that computes all transformations of a word word_1 into a word word_2, consisting of sequences of words of minimal length, starting with word_1, ending in word_2, and such that two consecutive words in the sequence differ by at most one letter. All words have to occur in a dictionary with name dictionary.txt, stored in the working directory.

It is convenient and effective to first create a dictionary whose keys are all words in the dictionary with one letter replaced by a "slot", the value for a given key being the list of words that match the key with the "slot" being replaced by an appropriate letter. From this dictionary, one can then build a dictionary with words as keys, and as value for a given key the list of words that differ in only one letter from the key.

The program implements a function word_ladder(word_1, word_2) that returns the list of all solutions, a solution being as previously described.

Next is a possible interaction.

```
$ python3
...
>>> from word_ladder import *
>>> for ladder in word_ladder('cold', 'warm'): print(ladder)
...
['COLD', 'CORD', 'WORD', 'WORM', 'WARM']
['COLD', 'CORD', 'WORD', 'WARD', 'WARM']
['COLD', 'CORD', 'CARD', 'WARD', 'WARM']
>>> for ladder in word_ladder('three', 'seven'): print(ladder)
...
['THREE', 'THREW', 'SHREW', 'SHRED', 'SIRED', 'SITED', 'SATED', 'SAVED', 'SAVER', 'SEVER', 'SEVEN']
```

```
>>> for ladder in word_ladder('train', 'bikes'): print(ladder)
...
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'BROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'BRAIN', 'BRAWN', 'BROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DRAWS', 'DRAGS', 'BRAGS', 'BRATS', 'BEATS', 'BESTS', 'BUSTS', 'BUSES', 'BASES', 'BAKES', 'BIKES']
['TRAIN', 'DRAIN', 'DRAWN', 'DRAWS', 'DRAGS', 'BRAGS', 'BRATS', 'BEATS', 'BELTS', 'BELLS', 'BALLS', 'BALES', 'BAKES', 'BIKES']
```

3 Using linked lists to represent polynomials (optional)

Extend the program `polynomial.py` that implements a class `Polynomial` from the previous lab to implement the functions `__add__()`, `__sub__()`, `__mul__()` and `__truediv__()`.

Next is a possible interaction.

```
$ python3
...
>>> from polynomial import *
>>> poly_6 = Polynomial('-2x + 7x^3 +x   - 0  + 2 -x^3 + x^23 - 12x^8 + 45 x ^ 6 -x^47')
>>> print(poly_6)
-x^47 + x^23 - 12x^8 + 45x^6 + 6x^3 - x + 2
>>> poly_7 = Polynomial('2x^5 - 71x^3 + 8x^2 - 93x^4 -6x + 192')
>>> poly_8 = Polynomial('192 -71x^3 + 8x^2 + 2x^5 -6x - 93x^4')
>>> poly_9 = poly_7 + poly_8
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_8)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_9)
4x^5 - 186x^4 - 142x^3 + 16x^2 - 12x + 384
>>> print(poly_7 * poly_7)
4x^10 - 372x^9 + 8365x^8 + 13238x^7 + 3529x^6 + 748x^5 - 34796x^4 - 27360x^3 + 3108x^2
                                                                        - 2304x + 36864
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_7 - poly_7)
0
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_9 / poly_7)
2
>>> print(poly_9)
4x^5 - 186x^4 - 142x^3 + 16x^2 - 12x + 384
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> poly_10 = Polynomial('-11x^4 + 3x^2 + 7x + 9')
>>> poly_11 = Polynomial('5x^2 -8x - 6')
>>> poly_12 = poly_10 * poly_11
>>> print(poly_12)
-55x^6 + 88x^5 + 81x^4 + 11x^3 - 29x^2 - 114x - 54
>>> print(poly_12 / poly_10)
5x^2 - 8x - 6
>>> print(poly_12 / poly_11)
-11x^4 + 3x^2 + 7x + 9
>>> poly_13 = poly_6 * poly_7
```

```
>>> print(poly_13 / poly_6)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_13 / poly_7)
-x^47 + x^23 - 12x^8 + 45x^6 + 6x^3 - x + 2
```

4 Context free grammars (advanced, optional)

A *context free* grammar is a set of *production rules* of the form

`symbol_0 ---> symbol_1 ... symbol_n`

where `symbol_0`, ..., `symbol_n` are either *terminal* or *nonterminal symbols*, with `symbol_0` being necessarily nonterminal. A symbol is a nonterminal symbol iff it is denoted by a word built from underscores or uppercase letters. A special nonterminal symbol is called the *start symbol*. The language *generated* by the grammar is the set of sequences of terminal symbols obtained by replacing a nonterminal symbol by the sequence on the right hand side of a rule having that nonterminal symbol on the left hand side, starting with the start symbol. For instance, the following, where `EXPRESSION` is the start symbol, is a context free grammar for a set of arithmetic expressions.

```
EXPRESSION --> EXPRESSION TERM_OPERATOR TERM
EXPRESSION --> TERM
TERM --> TERM FACTOR_OPERATOR FACTOR
TERM --> FACTOR
FACTOR --> NUMBER
FACTOR --> (EXPRESSION)
NUMBER --> DIGIT NUMBER
NUMBER --> DIGIT
DIGIT --> 0
...
DIGIT --> 9
TERM_OPERATOR --> +
TERM_OPERATOR --> -
FACTOR_OPERATOR --> *
FACTOR_OPERATOR --> /
```

Moreover, blank characters (spaces or tabs) can be inserted anywhere except inside a number. For instance, `(2 + 3) * (10 - 2) - 12 * (1000 + 15)` is an arithmetic expression generated by the grammar.

Note that operators associate to the left. The grammar is *unambiguous*, in the sense that every expression generated by the grammar has a unique evaluation.

Write down a program `context_free_grammar.py` that implements a function `evaluate()` which takes a string representing an expression as an argument, checks whether the expression can be generated by the grammar, and in case the answer is yes, returns the value of the expression, provided that no division by 0 is attempted; otherwise, the function returns `None`.

Next is a possible interaction.

```
$ python3
...
>>> from context_free_grammar import *
>>> evaluate('100')
100
>>> evaluate('(100)')
100
>>> evaluate('1 - 20 + 300')
281
>>> evaluate('(((((1))-((20))+((300)))))')
281
>>> evaluate('20 * 4 / 5')
16.0
>>> evaluate('(((((20))*((4))/((5)))))')
16.0
>>> evaluate('1 + 20 * 30 - 400 / 500')
600.2
>>> evaluate('1 + (20*30-400) / 500')
1.4
>>> evaluate('1+(20 / 30 * 400)- 500')
-232.33333333333337
>>> evaluate('1 + 2 * (3+4*5) / (6*7-8/9)')
2.1189189189189186
>>> evaluate('100)')
>>> evaluate('100 + ')
>>> evaluate('100 + -3')
>>> evaluate('100 # 50')
>>> evaluate('100 / 0')
```

Before you tackle the exercise, find out about *recursive descent parsers*. To easily tokenise the string, check out the `findall()` function from the `re` module.