6 Aula 06: 20/AGO/2019

6.1 Aula passada

- Problema de motivação: sequências de parênteses, colchetes e chaves bem-formadas
- Tipo pilha usando classe `Stack`.
- classe `Stack` usa `list` do Python, com `append` e `pop`.

6.2 Hoje

Notação polonesa e mais pilhas.

Sobre notação polonesa podemos primeiro calcular o valor de uma expressão aritmética em notação polonesa reversa (= **notação posfixa**) e em seguida fazer o programa que transforma de infixa para posfixa.

Outro problema bacana é o de *transformar a representação de um valor da notação decimal para a binária*. O uso da pilha é meio sem graça, já que só empilhamos todos os dígitos e depois desempilhamos todos esses dígitos. O problema é bacana para depois falarmos de recursão. Podemos fazer uma conversão de uma representação para outra através de uma recursão de calda (=*tail recursion*).

6.3 Precedência

Qual é o valor de

```
2 + 3 * 4 = 14
2 * 3 + 4 = 10
2 - 3 + 4 = 3
2 + 3 - 4 = 1
```

Cada operador tem a sua precedência. A única coisa que muda a precedência é a presença de parênteses

```
(2 + 3) * 4 = 20
2 * (3 + 4) = 14
2 - (3 + 4) = -5
2 + (3 - 4) = 1
```

A expressão `A + B * C + D` pode ser reescrita como `((A + (B * C)) + D)`.

6.4 Notação infixa

Usualmente os operadores são escritos *entre* os operandos

```
(A + B) * D + E / (F + A * D) + C
   1    2   6   5    4   3    7
(1 + 2) * 3 + 4 / (5 + 6 * 7) + 8 = 17.085106382978722
```

Essa é a chamada **notação infixa**.

6.5 Notação polonesa

Na **notação polonesa** (reversa) ou **posfixa** os operadores são escritos depois dos operandos

```
A B + D * E F A D * + / + C +
    1   1         2 3   2   4
1 2 + 3 * 4 5 6 7 * + / + 8 +
```

infixa	posfixa
`2 + 3 * 4`	`2 3 4 * +`
`( 2 + 3 ) * 4`	`2 3 + 4 *`
`2 * ( 3 + 4 ) / 5 - 6`	`2 3 4 + * 5 / 6 -`
`2 - 1`	`2  1 -`
`10 + 3 * 5 / (16 - 4)`	`10 3 5 * 16 4 - / +`

Uma expressão é composta de **itens léxicos** (=*tokens*).

Itens lexicais são palavras simples ou grupos de palavras no léxico de uma língua

6.6 Problema: valor de expressão posfixa

Versão vaga: Escreva um programa (= função `main()`) que lê uma expressão numérica posfixa que contém apenas os operadores binários `+`, `-`, `*` e `/` e os operandos são valores da classe `int` ou `float` e calcula o valor da expressão.

Exemplos:

```
Calculadora de expressões numéricas posfixas
expr >>>  7 8 + 3 2 + /
3
expr >>> 4 5 6 * +
34
expr >>>  7 8 + 3 2 + /
3
expr >>> 4 5 6 * +
34
expr >>> 1 2 + 3 * 4 5 6 7 * + / + 8 +
17.0851
expr >>> 10 3 5 * 16 4 - / +
11.25
expr >>> 1 2 3 + -
-4
expr >>> 1 2 3 - +
0
expr >>> 4 7 9 - +
2
expr >>> 1 2 3 + 4 5 * 2 / +
15
expr >>> 22 33 44 * *
31944
```

Depois é melhor supor que há um espaço separando os itens léxicos.

6.7 Solução

Arquivo `calculadora_polonesa.py`. Na lousa eu, coelho, escrevo uma versão simplificada.

```
from stack import Stack

PROMPT = "expr >>> "
QUIT = ""
ADD  = "+"
SUB  = "-"
MUL  = "*"
DIV  = "/"

def main():
    '''
    Calcula o valor de uma expressão numérica
    posfixa que contém apenas os operadores
    binários +, -, * e /.

    Pré-condição: o programa supões que os operadores
      e operandos estão separados por pelo menos um
      espaço.
    '''
    print("Calculadora de expressões numéricas posfixas")
    print("Deve haver um espaço entre os operadores e operandos")
    print("(Tecle ENTER para encerrar o programa.)")

    expressao = input(PROMPT).strip()
    while expressao != QUIT:
        posfixa = expressao.split()
        try:
            valor = valor_expressao(posfixa)
            if valor != None: print("%s" %valor)
        except:
            print("Erro na expressão")
        expressao = input(PROMPT).strip()


#---------------------------
```

Escreva uma função `valor_expressao()` que respeita a seguinte especificação:

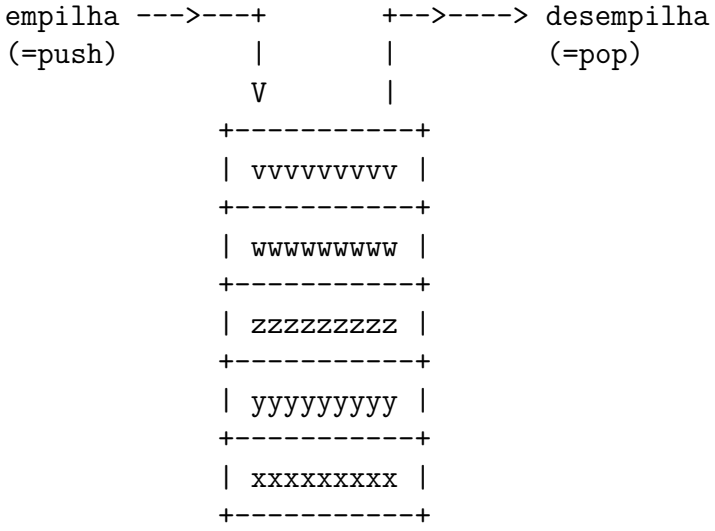
```
#-----------------------------------------------------
def valor_expressao(posfixa):
    ''' (list) -> int|float|None

    Recebe um lista com os itens representando uma expressão
    numérica em notação posfixa e calcula e retorna o valor
    da expressao.

    Pré-condiçao: a funçao supõe que a expressao está
         correta e os operadores e operandos estao separados
         por pelo menos um espaço.
    '''
    pilha = Stack()
    for item in posfixa:
        if item in [ADD,SUB,MUL,DIV]:
            if len(pilha) < 2:
                print("Erro: faltam operandos")
                return None
            valor_2 = pilha.pop()
            valor_1 = pilha.pop()
            if item == ADD:
                valor = valor_1 + valor_2
            elif item == SUB:
                valor = valor_1 - valor_2
            elif item == MUL:
                valor = valor_1 * valor_2
            elif item == DIV:
                valor = valor_1 / valor_2
        else:
            # é um numero
            if "." in item: valor = float(item)
            else: valor = int(item)

        pilha.push(valor)

    resultado = pilha.pop()
    if not pilha.isEmpty():
        print("Erro: faltam operadores")
        return None
    return resultado
```

6.8 Pilhas

Uma **pilha** (=*stack*) é uma lista dinâmica em que todas as operações:

- inserções;
- remoções; e
- consultas

são feitas em uma mesma extremidade chamada de **topo**.




```
class Stack:
    #--------------------------------------------------
    def __init__(self):
        '''(Stack) -> None

        Usado pelo construtor da classe.

        Monta um objeto da classe Pilha.
        '''
        self.itens = []

    #--------------------------------------------------
    def __str__(self):
        '''(Stack) -> str

        Recebe uma Pilha referenciada por `self` e constroi e
        retorna o string exibido por print() para imprimir uma
        pilha. Esse também é o string retornado por str().
        '''
        return str(self.itens)

    #--------------------------------------------------
    def __len__(self):
        '''(Stack) -> int

        Recebe uma Pilha referenciada por self e retorna
```

```
    o número de itens na pilha.

    Usado pelo Python quando escrevemos len(Stack).
    '''
    return len(self.itens)

#-----------------------------------------------------
def isEmpty(self):
    '''(Stack) -> bool

    Recebe uma Pilha referenciada por self e retorna
    True se ela está vazia e False em caso contrário.
    '''
    return self.itens == []

#-----------------------------------------------------
def push(self, item):
    '''(Stack, objeto) -> None

    Recebe uma Pilha referenciada por self e um objeto
    item e coloca item no topo da pilha.
    '''
    self.itens.append(item)

#-----------------------------------------------------
def pop(self):
    '''(Stack) -> objeto

    Recebe uma Pilha referenciada por self e desempilha
    e retorna o objeto no topo da pilha.
    '''
    return self.itens.pop()

#-----------------------------------------------------
def peek(self):
    '''(Stack) -> objeto

    Recebe uma Pilha referenciada por self e retorna
    o objeto no topo da pilha. O objeto não é removido
    da pilha.
    '''
    return self.itens[-1]
```