PYTHON

Kalkulator i konwerter Odwrotnej Notacji Polskiej

Sebastian Zieja

Wprowadzenie

Zagadnieniem było stworzenie programu, który po otrzymaniu na wejściu wyrażenia w zapisie standardowej konwencji algebraicznej (infix), dokona konwersji wyrażenia na zapis Odwrotnej Notacji Polskiej (w skrócie ONP).

ONP jest to sposób zapisu wyrażenia arytmerycznego, w którym znak operacji jest umieszczany po operandach, a nie pomiędzy nimi jak w standardowym zapisie algebraicznym. Co pozwala jednoznacznie określić kolejność wykonywania działań.

Został stworzony przez australijskiego naukowaca Charlesa Hamblina jako odwrócenie notacji polskiej Jana Łukasiewicza. Sugestią Hamblina było nazwanie tej notacji Notacją Azciweisakuł – zapisane od tyłu „Łukasiewicza".

Program na początku musi rozpoznać, czy wprowadzone wyrażenie jest zapisane w ONP, czy w zwykłym zapisie. W tym celu wykorzystuje fakt, iż tylko w ONP pojawiają się dwie liczby obok siebie. (np. 2 3 +, w infix jest to 2 + 3).

Obliczenie wyrażenia zapisanego w ONP:

- Dopóki jest znak na wejściu:
    - Jeśli znak jest liczbą, umieść na stosie
    - Jeśli znak jest operatorem, pobierz ze stosu dwie liczby, wykonaj działanie i zwróć wynik na stos.
- Zwróć wynik jako jedyny pozostały element na stosie.

Algorytm konwersji wyrażenia na ONP:

- Póki zostały symbole do przeczytania wykonuj:
  Przeczytaj symbol.

    - Jeśli symbol jest liczbą dodaj go do kolejki wyjście.
    - Jeśli symbol jest funkcją włóż go na stos.
    - Jeśli symbol jest znakiem oddzielającym argumenty funkcji (nawiasy):

- Dopóki najwyższy element stosu nie jest lewym nawiasem, zdejmij element ze stosu i dodaj go do kolejki wyjście.

    - Jeśli symbol jest operatorem, o_1, wtedy:

1) dopóki na górze stosu znajduje się operator, o_2 taki, że:

o_1 jest lewostronnie łączny i jego kolejność wykonywania jest mniejsza lub równa kolejności wyk. o_2,
lub
o_1 jest prawostronnie łączny i jego kolejność wykonywania jest mniejsza od o_2,
zdejmij o_2 ze stosu i dołóż go do kolejki wyjściowej i wykonaj jeszcze raz 1)
2) włóż o_1 na stos operatorów.

    - Jeżeli symbol jest lewym nawiasem to włóż go na stos.
    - Jeżeli symbol jest prawym nawiasem to zdejmuj operatory ze stosu i dokładaj je do wyjścia, dopóki symbol na górze stosu nie jest lewym nawiasem, kiedy dojdziesz do tego miejsca zdejmij lewy nawias ze stosu bez dokładania go do wyjścia. Teraz, jeśli najwyższy element na stosie jest funkcją, także dołóż go do wyjścia.

- Jeśli nie ma więcej symboli do przeczytania, zdejmuj wszystkie symbole ze stosu (jeśli jakieś są) i dodawaj je do wyjścia.

Opis interfejsu

Program przyjmuje z terminala wyrażenie zapisane w postaci każdego znaku oddzielonego spacją. Przykładowe wejścia:

(1 + 2) * 3

1 2 + 3 *

Po wpisaniu wyrażenia, program zwraca w konsoli albo wynik konwersji na ONP, albo wartość wyrażenia podanego w ONP.

Implementacja

```
#-*- coding: utf-8 -*-

"""Opisuja łączność argumentów - niektóre są prawo, niektóre lewostronnie łączne"""
leftAssociative = 0
rightAssociative = 1
```

Słownik, który wg mnie jest najlepszą opcją przy implementacji tego problemu. Do każdego operatora przypisana jest tupla zawierająca jego priorytet, łączność i wyrażenie korzystające z lambda do wykorzystania przy obliczaniu wartości.

```
"""Słownik operatorów, zawierający ich priorytet, łączność i ich operacje."""
OPERATORS = {
	'+' : (0, leftAssociative, lambda x,y: y+x),
	'-' : (0, leftAssociative, lambda x,y: y-x),
	'*' : (1, leftAssociative, lambda x,y: y*x),
	'/' : (1, leftAssociative, lambda x,y: y/x),
	'^' : (2, rightAssociative, lambda x,y: y**x)
}

def isOperator(token):
	"""Zwraca true jeśli znak jest operatorem, false jeśli nie jest."""
	return token in OPERATORS.keys()


def cmpAssociative(token, associative):
	"""Zwraca true jeśli operator ma łączność podaną w wywołaniu funkcji."""
	if not isOperator(token):
		raise VauleError("Bledny operator: %s" % token)
	return OPERATORS[token][1] == associative


def cmpPrecedence(token1, token2):
	"""Zwraca liczbę >0 gdy operator 1 ma większy priorytet, 0 gdy są o takim samym priorytecie,
	<0 gdy 1 operator jest o mniejszym priorytecie."""
	if not isOperator(token1) or not isOperator(token2):
		raise ValueError("Bledne operatory: %s %s" % (token1, token2))
	return OPERATORS[token1][0] - OPERATORS[token2][0]

def infixToRPN(tokens):
	"""Funkcja zwracająca rownanie w ONP."""

	stack=[]
	output=[]

	for token in tokens:
		if isOperator(token):
			while len(stack)!=0 and isOperator(stack[-1]):
				if (cmpAssociative(token, leftAssociative) and cmpPrecedence(token,
stack[-1])<=0) or (cmpAssociative(token, rightAssociative) and cmpPrecedence(token, stack[-1])<0):
					output.append(stack.pop())
					continue
				break
			stack.append(token)
		elif token == "(":
			stack.append("(")
		elif token == ")":
			while len(stack)!=0 and stack[-1]!="(":
				output.append(stack.pop())
			stack.pop()
```

```
                else:
                        output.append(token)

        #po skonczonej petli przenieś wszystko ze stosu do wyjścia
        while len(stack)!=0:
                output.append(stack.pop())

        #zwraca str z wynikiem
        return " ".join(output)


def calcRPN(tokens):
        """Funkcja zwracajaca wynik wyrazenia ONP."""

        stack=[]

        for token in tokens:
                if token in OPERATORS:
                        Wprowadza na stos wartość wyrażenia obliczonego z danym operatorem który został zczytany. Wykorzystuje wyrażenie z lamdą ze słownika. Pobiera dwa najwyższe elementy ze stosu (będą to liczby), przekazuje do odpowiadającego operatorowi wyrażeniu lamda i wynik wprowadza na stos.
                        stack.append(OPERATORS[token][2](stack.pop(), stack.pop()))
                else:
                        stack.append(float(token))
        return stack.pop()

def makeCalculation(tokens):
        """Funkcja ktora rozpoznaje czy zostalo wpisane rownanie w
        zapisie ONP, czy w zwyklym, ktore ma zostac zamienione na ONP.
        """

        Wykorzystanie listy złożonej, tak iż tworzy identyczną listę jak tokens, jednak nowopowstała lista nie posiada żadnych nawiasów. Potrzebne aby stwierdzić czy wprowadzone wyrażenie jest w ONP czy też nie.
        #tworzy liste taka jak tokens, tylko nie zawierajaca nawiasow
        temp=[y for y in tokens if (y != "(" and y != ")")]

        #sprawdza czy tokens jest rownaniem w ONP czy nie
        for i in range(0, len(temp)-1):
                if (not isOperator(temp[i])) and (not isOperator(temp[i+1])):
                        return calcRPN(tokens)
                else:
                        return infixToRPN(tokens)


tokens = raw_input().split(" ") Z wprowadzonego stringa z konsoli tworzy listę uznając za separator spację.
print makeCalculation(tokens)
```

Podsumowanie, wyniki testów

W programie użyto testów za pomocą biblioteki unittest:

```
class TestONP(unittest.TestCase):
    """Klasa wykonujaca testy calego programu i poprawnosci algorytmu."""
    def setUp(self):
        self.r1 = "2 3 + 4 *".split(" ")
        self.r2 = "( 1 + 2 ) * ( 3 + 3 )".split(" ")
        self.r3 = "5 3 + 2 ^ 4 3 * -3 / +".split(" ")
        self.r4 = "( 5 + 3 ) ^ 2 + 4 * -3 / 3".split(" ")
        self.r5 = "1 4 3 * -3 / +".split(" ")
    def testCalcRpn(self):
        self.assertEqual(repr(makeCalculation(self.r1)), "20.0")
        self.assertEqual(repr(makeCalculation(self.r3)), "60.0")
        self.assertEqual(repr(makeCalculation(self.r5)), "-3.0")
    def testInfixToRPN(self):
        self.assertEqual(makeCalculation(self.r2), "1 2 + 3 3 + *")
        self.assertEqual(makeCalculation(self.r4), "5 3 + 2 ^ 4 -3 * 3 / +")
```

Sprawdza ona poprawność konwersji wyrażenia na ONP, jak i obliczania wyrażenia z ONP.

Sprawdziłem wszystkie możliwe operatory jak i występowanie nawiasów. Wszystkie testy zostały zaliczone w 100% co wskazuje na poprawną implementację algorytmu.

Literatura:

1) Opis algorytmu na angielskiej Wikipedii o ONP: https://en.wikipedia.org/wiki/Shunting-yard_algorithm , https://en.wikipedia.org/wiki/Reverse_Polish_notation