

# Programování 2

---

## 11. cvičení, 27-04-2023

---

tags: Programování 2, Čtvrtek 1 Čtvrtek 2

### Farní oznamy

---

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.
2. **Domácí úkoly:**
  - Jedna úloha na zpracování řetězců - lehká
  - Výpočet prefixového výrazu
  - Přepis z postfixové do infixové notace: závorky. Promluvíme podrobněji.
3. **Zápočtový program:** Pokud jste si ještě nezvolili téma, udělejte tak co nejdřív. Klidně mi také napište, pokud si neumíte vybrat nebo máte jiné nejasnosti.

---

#### Dnešní program:

- Kvíz
- Pythonské okénko
- Domácí úkoly
- Min-max: Piškvorky

---

### Na zahřátí

---

In order to understand recursion, one must first understand recursion.

---

### Co dělá tento kód

```

# What is in list_1 at the end?

>>> list_1 = [1, 2, 3, 4]
>>> for idx, item in enumerate(list_1):
...     del item
...
>>> print(list_1)
???
```

Rozlišujeme, co je hodnota a co je pointer.

## Domácí úkoly

### Evaluace prefixového výrazu

Vstup:

1 | / + 1 2 2

Výstup:

1 | 1

Jak na to:

#### Nerekurzivní řešení

```

1  from operator import add, sub, mul    # pro Pythonský způsob volby funkce
2
3  ops = {"+":add, "-":sub, "*":mul}    # znaménko: operace
4
5
6  def read_prefix() -> list[str]:
7      return input().split()
8
9
10 def process_prefix(terms: list[str]) -> int:
11     stack = []
12     for t in reversed(terms):    # možno i t = terms.pop()
13         if t in ops:
14             stack.append(ops[t](stack.pop(), stack.pop()))
15         else:
```

```

16         stack.append(int(t))
17     return stack[0]
18
19
20 def main() -> None:
21     print(process_prefix(read_prefix()))
22

```

## Rekurzivní řešení (Petro Velychko)

```

1  from collections import deque
2
3
4  def parse(tokens):
5      token = tokens.popleft()
6      if token == '+':
7          return parse(tokens) + parse(tokens)
8      elif token == '-':
9          return parse(tokens) - parse(tokens)
10     elif token == '*':
11         return parse(tokens) * parse(tokens)
12     else:
13         return int(token)
14
15
16 if __name__ == '__main__':
17     print(parse(deque(input().split())))
18

```

## Převod z postfixové do infixové notace

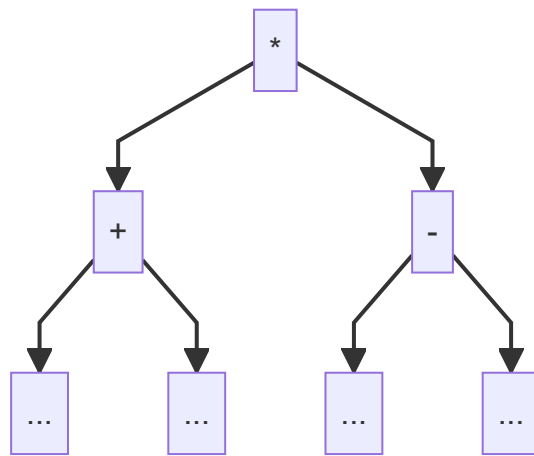
Vstup:

```
1 | 1 2 * 3 4 * 5 6 * 7 8 * - - -
```

Výstup:

```
1 | 1*2-(3*4-(5*6-7*8))
```

Největší problém je, kam dát závorky: pro rozhodnutí potřebujeme mít k dispozici správný kontext - operátor a nadřazený operátor.



Tedy pro rozhodnutí, zda mají být kolem podvýrazu závorky, potřebuji nadřazený uzel. Jednodušší je nejdřív z výrazu vystavět strom:

```
1  import sys
2
3  ops = ["+", "-", "*", "/"]
4
5
6  class Node:
7      ...
8
9
10 class Constant(Node):
11     def __init__(self, value: int):
12         self.value = value
13
14
15     def __str__(self):
16         return str(self.value)
17
18
19 class Operation(Node):
20     def __init__(self, op:str, left:Node=None, right:Node=None):
21         self.op = op
22         self.left = left
23         self.right = right
24
25     def needs_brackets_left(self) -> bool:
26         left_op = "."
27         if isinstance(self.left, Operation):
28             left_op = self.left.op
29         if self.op in ["*", "/"]:
30             if left_op in ["+", "-"]:
31                 return True
32             else:
33                 return False
34         return False
```

```

35
36 def needs_brackets_right(self) -> bool:
37     right_op = "."
38     if isinstance(self.right, Operation):
39         right_op = self.right.op
40     if self.op == "*":
41         if right_op in ["+", "-"]:
42             return True
43         else:
44             return False
45     if self.op == "/":
46         if right_op in ops:
47             return True
48         else:
49             return False
50     if self.op == "-":
51         if right_op in ["+", "-"]:
52             return True
53         else:
54             return False
55     return False
56
57 def __str__(self):
58     left_string = f"{self.left}"
59     if self.needs_brackets_left():
60         left_string = "(" + left_string + ")"
61     right_string = f"{self.right}"
62     if self.needs_brackets_right():
63         right_string = "(" + right_string + ")"
64
65     return left_string + f"{self.op}" + right_string
66
67
68 def postfix_to_tree() -> Node:
69     stack = []
70     for term in input().split():
71         if term in ops:
72             right = stack.pop()
73             left = stack.pop()
74             stack.append(Operation(term, left, right))
75         else:
76             stack.append(Constant(int(term)))
77     return stack.pop()
78
79
80 def main() -> None:
81     tree = postfix_to_tree()
82     print(tree)
83
84
85 if __name__ == "__main__":
86     sys.setrecursionlimit(10000)

```

```
87     main()
88
```

Pro delší výrazy si toto vyžaduje velikou rekurzní hloubku.

### Nerekurzivní řešení

Opět vybudujeme strom, ale nebudeme rekurzivně volat metodu `__str__()`. Namísto toho projdeme stromem a od každého operátoru, který potřebuje závorky, tyto posuneme k příslušnému levému a pravému listu. Pak in-order průchodem pomocí zásobníku vypíšeme výraz:

```
1  import sys
2
3  ops = ["+", "-", "*", "/"]
4
5
6  class Node:
7      ...
8
9
10 class Constant(Node):
11     def __init__(self, value: int):
12         self.value = value
13         self.left = None
14         self.right = None
15         self.left_brackets = 0
16         self.right_brackets = 0
17
18     def __str__(self):
19         return "(" * self.left_brackets + str(self.value) + ")" *
self.right_brackets
20
21
22 class Operation(Node):
23     def __init__(self, op:str, left: Node = None, right: Node = None):
24         self.op = op
25         self.left = left
26         self.right = right
27         self.left_brackets = 0
28         self.right_brackets = 0
29
30     def needs_brackets_left(self) -> bool:
31         left_op = "."
32         if isinstance(self.left, Operation):
33             left_op = self.left.op
34         if self.op in ["*", "/"]:
35             if left_op in ["+", "-"]:
36                 return True
37             else:
38                 return False
39         return False
40
```

```

41     def needs_brackets_right(self) -> bool:
42         right_op = "."
43         if isinstance(self.right, Operation):
44             right_op = self.right.op
45         if self.op == "*":
46             if right_op in ["+", "-"]:
47                 return True
48             else:
49                 return False
50         if self.op == "/":
51             if right_op in ops:
52                 return True
53             else:
54                 return False
55         if self.op == "-":
56             if right_op in ["+", "-"]:
57                 return True
58             else:
59                 return False
60         return False
61
62     def __str__(self):
63         return self.op
64
65
66 def postfix_to_tree() -> Node:
67     stack = []
68     for term in input().split():
69         if term in ops:
70             right = stack.pop()
71             left = stack.pop()
72             stack.append(Operation(term, left, right))
73         else:
74             stack.append(Constant(int(term)))
75     return stack.pop()
76
77
78 def set_brackets(tree) -> Node:
79     stack = [tree]
80     while stack:
81         node = stack.pop()
82         if node.left:
83             if node.needs_brackets_left():
84                 node.left.left_brackets = node.left_brackets + 1
85                 node.left.right_brackets = 1
86             else:
87                 node.left.left_brackets = node.left_brackets
88             stack.append(node.left)
89         if node.right:
90             if node.needs_brackets_right():
91                 node.right.left_brackets = 1
92                 node.right.right_brackets = node.right_brackets + 1

```

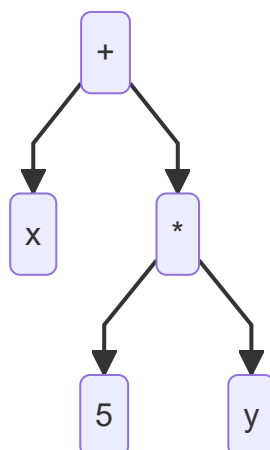
```

93         else:
94             node.right.right_brackets = node.right_brackets
95             stack.append(node.right)
96     return tree
97
98
99 def tree_to_infix(tree: Node):
100     stack = []
101     current = tree
102     while True:
103         if current is not None:
104             stack.append(current)
105             current = current.left
106         elif stack:
107             current = stack.pop()
108             yield str(current)
109             current = current.right
110         else:
111             break
112
113
114 def main() -> None:
115     tree = postfix_to_tree()
116     tree = set_brackets(tree)
117     print("".join(tree_to_infix(tree)))
118
119
120 if __name__ == "__main__":
121     main()
122

```

## Opakování

### Operace s výrazy ve tvaru stromů





Takovýto strom definuje polynom. Naučili jsme se počítat její *derivaci* přeměnou na jiný strom. Toto nám ale dává obecně větší strom, ve kterém bude spousta hlušiny:

```
1 | (x + (5 * y))          <-- funkce
2 | (1 + ((0 * y) + (5 * 0))) <-- derivace podle x
3 | (0 + ((0 * y) + (5 * 1))) <-- derivace podle y
```

- přičítání nuly a násobení nulou
- násobení jedničkou

Můžeme si vytvořit čistící proceduru, která stromy rekurzivně vyčistí, a opět postupujeme tak, že určité uzly či struktury ve stromu rekurzivně nahrazujeme jinými uzly či strukturami.

```
1 | class Expression:
2 |     ...
3 |
4 |
5 | class Constant(Expression):
6 |     def __init__(self, value):
7 |         self.value = value
8 |
9 |     def __str__(self):
10 |         return str(self.value)
11 |
12 |     def eval(self, env):
13 |         return self.value
14 |
15 |     def derivative(self, by):
16 |         return Constant(0)
17 |
18 |     def prune(self):
19 |         return self
20 |
21 | # Testování konstanty, zdali je či není 0 nebo 1 !!
22 |
23 | def is_zero_constant(x):
24 |     return isinstance(x, Constant) and x.value == 0
25 |
26 |
27 | def is_unit_constant(x):
28 |     return isinstance(x, Constant) and x.value == 1
29 |
30 |
31 | class Variable(Expression):
32 |     def __init__(self, name):
33 |         self.name = name
34 |
35 |     def __str__(self):
36 |         return self.name
37 |
38 |     def eval(self, env):
```

```

39         return env[self.name]
40
41     def derivative(self, by):
42         if by == self.name:
43             return Constant(1)
44         else:
45             return Constant(0)
46
47     def prune(self):
48         return self
49
50
51 class Plus(Expression):
52     def __init__(self, left, right):
53         self.left = left
54         self.right = right
55
56     def __str__(self):
57         return "(" + str(self.left) + " + " + str(self.right) + ")"
58
59     def eval(self, env):
60         return self.left.eval(env) + self.right.eval(env)
61
62     def derivative(self, by):
63         return Plus(
64             self.left.derivative(by),
65             self.right.derivative(by)
66         )
67
68     def prune(self):
69         self.left = self.left.prune()
70         self.right = self.right.prune()
71         if is_zero_constant(self.left):
72             if is_zero_constant(self.right):
73                 return Constant(0)
74             else:
75                 return self.right
76         if is_zero_constant(self.right):
77             return self.left
78         return self
79
80
81 class Times(Expression):
82     def __init__(self, left, right):
83         self.left = left
84         self.right = right
85
86     def __str__(self):
87         return "(" + str(self.left) + " * " + str(self.right) + ")"
88
89     def eval(self, env):
90         return self.left.eval(env) * self.right.eval(env)

```

```

91
92     def derivative(self, by):
93         return Plus(
94             Times(
95                 self.left.derivative(by),
96                 self.right
97             ),
98             Times(
99                 self.left,
100                 self.right.derivative(by)
101             )
102         )
103
104     def prune(self):
105         self.left = self.left.prune()
106         self.right = self.right.prune()
107         if is_zero_constant(self.left) | is_zero_constant(self.right):
108             return Constant(0)
109         if is_unit_constant(self.left):
110             if is_unit_constant(self.right):
111                 return Constant(1)
112             else:
113                 return self.right
114         if is_unit_constant(self.right):
115             return self.left
116         return self
117
118
119 def main():
120     vyraz = Plus(
121         Variable("x"),
122         Times(
123             Constant(5),
124             Variable("y")
125         )
126     )
127     print(vyraz)
128     print(vyraz.derivative(by="x"))
129     print(vyraz.derivative(by="x").prune())
130     print(vyraz.derivative(by="y"))
131     print(vyraz.derivative(by="y").prune())
132
133
134 if __name__ == '__main__':
135     main()
136     -----
137     (x + (5 * y))
138     (1 + ((0 * y) + (5 * 0)))
139     1
140     (0 + ((0 * y) + (5 * 1)))
141     5

```

- Všimněte si post-order procházení stromu při prořezávání.
- Metodu `prune` definujeme také pro konstanty a proměnné, i když s nimi nedělá nic. Ulehčuje to rekurzivní volání metody.
- Musíme být pozorní při testování, zda je daný uzel/výraz nulová nebo jedničková konstanta. Nestačí operátor rovnosti, musíme nejdřív zjistit, zda se jedná o konstantu a pak otestovat její hodnotu. V principu bychom mohli dvě testovací funkce proměnit v metody třídy `Expression`.

## Min-max: Piškvorky

Toto je jednoduchá hra, a chceme najít optimální strategii. Kam dát následující kroužek?



Data: seznam znaků x, o, . o délce 9 (nechceme 2D pole)

Hodnocení: Pokud se mřížce nachází trojice xxx, plus nekonečno. Pokud se v mřížce nachází trojice ooo, plus nekonečno. Jinak 0.

Detekce: Pro každý znak najdeme všechna místa, kde se nachází, a porovnáme se seznamem 8 možných trojic:

```

1 INFINITY = 1
2 MINUS_INFINITY = - INFINITY
3
4 empty_grid = ["."] * 9
5 triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8},
6           {0, 4, 8}, {2, 4, 6}]
7
8 def find_triple(grid, sign):
9     positions = {i for i in range(9) if grid[i] == sign}
10    result = [t for t in triples if t.issubset(positions)]
11    return result
12
13
14 def grade(grid) -> int:
15     if find_triple(grid, "x"):
16         return INFINITY
17     elif find_triple(grid, "o"):

```

```

18         return MINUS_INFINITY
19     else:
20         return 0
21
22
23 def get_sign(player: bool) -> str:
24     return "o" if player else "x"
25

```

Tisk mřížky:

```

1 def print_grid(grid) -> None:
2     print()
3     for i in range(3):
4         for j in range(3):
5             print(grid[3*i + j], end = " ")
6         print()
7     print(grade(grid))
8     print()

```

Strom:

```

1 class Node:
2     def __init__(self, grid):
3         self.grid = grid
4         self.df = self.grid.count(".")
5         self.player = (9 - self.df) % 2
6         self.score = grade(self.grid)
7         self.children = []

```

Stavíme strom:

Musíme dát pozor na kombinatoriku. Mnohé pozice můžeme dosáhnout několika způsoby, takže pro pozici, kterou jsme již viděli, použijeme existující uzel stromu:

```

1 def build_tree(start_grid: list[int] = empty_grid) -> Node:
2     node_dict = {}
3     root = Node(start_grid)
4     queue = deque([root])
5     node_dict[tuple(start_grid)] = root
6     n_nodes = 1
7     while queue:
8         node = queue.popleft()
9         if node.score != 0:
10             continue
11         sign = get_sign(node.player)
12         for pos in range(9):
13             if node.grid[pos] == ".":
14                 new_grid = node.grid.copy()
15                 new_grid[pos] = sign

```

```

16         if tuple(new_grid) in node_dict:
17             new_node = node_dict[tuple(new_grid)]
18         else:
19             new_node = Node(new_grid)
20             node_dict[tuple(new_grid)] = new_node
21             queue.append(new_node)
22             n_nodes += 1
23             node.children.append(new_node)
24     print(n_nodes)
25     return root
26

```

A konečně min-max:

```

1  class Choice:
2      def __init__(self, choice, value):
3          self.choice = choice
4          self.value = value
5
6
7  def minmax(node):
8      if not node.children:
9          return Choice("end", node.score)
10
11     choices = [minmax(c) for c in node.children]
12     if node.player == 0:
13         max_result = max(c.value for c in choices)
14         max_choices = [i for i in range(len(node.children)) if choices[i].value
15 == max_result]
16         return Choice(max_choices, max_result)
17     else:
18         min_result = min(c.value for c in choices)
19         min_choices = [i for i in range(len(node.children)) if choices[i].value
20 == min_result]
21         return Choice(min_choices, min_result)
22
23 def play(start_grid = empty_grid):
24     tree = build_tree(start_grid)
25     current_node = tree
26     while True:
27         print_grid(current_node.grid)
28         choice = minmax(current_node)
29         if choice.choice == "end":
30             print("Game finished")
31             break
32         select = random.choice(choice.choice)
33         current_node = current_node.children[select]

```

Výsledný program:

```

1  from collections import deque
2  import random
3
4  INFINITY = 1
5  MINUS_INFINITY = - INFINITY
6
7  empty_grid = ["."] * 9
8  triples = [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8},
9             {0, 4, 8}, {2, 4, 6}]
10
11 def find_triple(grid, sign):
12     positions = {i for i in range(9) if grid[i] == sign}
13     result = [t for t in triples if t.issubset(positions)]
14     return result
15
16
17 def grade(grid) -> int:
18     if find_triple(grid, "x"):
19         return INFINITY
20     elif find_triple(grid, "o"):
21         return MINUS_INFINITY
22     else:
23         return 0
24
25
26 def get_sign(player: bool) -> str:
27     return "o" if player else "x"
28
29
30 def print_grid(grid) -> None:
31     print()
32     for i in range(3):
33         for j in range(3):
34             print(grid[3*i + j], end = " ")
35         print()
36     print(grade(grid))
37     print()
38
39
40 class Node:
41     def __init__(self, grid):
42         self.grid = grid
43         self.df = self.grid.count(".")
44         self.player = (9 - self.df) % 2
45         self.score = grade(self.grid)
46         self.children = []
47
48
49 def build_tree(start_grid: list[int] = empty_grid) -> Node:
50     node_dict = {}
51     root = Node(start_grid)

```

```

52     queue = deque([root])
53     node_dict[tuple(start_grid)] = root
54     n_nodes = 1
55     while queue:
56         node = queue.popleft()
57         if node.score != 0:
58             continue
59         sign = get_sign(node.player)
60         for pos in range(9):
61             if node.grid[pos] == ".":
62                 new_grid = node.grid.copy()
63                 new_grid[pos] = sign
64                 if tuple(new_grid) in node_dict:
65                     new_node = node_dict[tuple(new_grid)]
66                 else:
67                     new_node = Node(new_grid)
68                     node_dict[tuple(new_grid)] = new_node
69                     queue.append(new_node)
70                     n_nodes += 1
71                 node.children.append(new_node)
72     print(n_nodes)
73     return root
74
75
76 class Choice:
77     def __init__(self, choice, value):
78         self.choice = choice
79         self.value = value
80
81     def __str__(self):
82         return f"Choosing {self.choice} to reach {self.value}"
83
84
85 def minmax(node):
86     if not node.children:
87         return Choice("end", node.score)
88
89     choices = [minmax(c) for c in node.children]
90     if node.player == 0:
91         max_result = max(c.value for c in choices)
92         max_choices = [i for i in range(len(node.children)) if
93 choices[i].value == max_result]
94         return Choice(max_choices, max_result)
95     else:
96         min_result = min(c.value for c in choices)
97         min_choices = [i for i in range(len(node.children)) if
98 choices[i].value == min_result]
99         return Choice(min_choices, min_result)
100
101 def play(start_grid = empty_grid):
102     tree = build_tree(start_grid)

```



```
102     current_node = tree
103     while True:
104         print_grid(current_node.grid)
105         choice = minmax(current_node)
106         if choice.choice == "end":
107             print("Game finished")
108             break
109         select = random.choice(choice.choice)
110         current_node = current_node.children[select]
111
112
113 def main() -> None:
114     start_grid = input().split()
115     play(start_grid)
116
117
118 if __name__ == "__main__":
119     main()
120
```