



Úvod do programovacích stylů ◊ poznámky k přednášce

7. Funkce

verze z 1. listopadu 2023

Uvažujme pro začátek jen aritmetické výrazy, jako jsou například výrazy $1 + x$ nebo $x * (2 + x)$. A zavedme si omezení, že nebudeme měnit jednou nastavené hodnoty proměnných. Například nemůžeme zvýšit hodnotu proměnné o jedna:

```
x = 1
x = x + 1
```

Zajímavým důsledkem je, že aritmetické výrazy budou mít vždy stejnou hodnotu.

Dva výrazy nazveme *ekvivalentní*, jestliže mají stejnou hodnotu. Ekvivalenci výrazů *expr1* a *expr2* zapíšeme:

```
expr1
=== expr2
```

Například:

```
1 + 2
=== 3
```

ale i:

```
1 + 2
=== 2 + 1
```

Hodnotou každého z výrazů $1 + 2$, 3 a $2 + 1$ je číslo tři.

Proměnná je ekvivalentní své hodnotě. Například:

```
>>> x = 2

x
=== 2
```

Výraz v programu můžeme nahradit za jemu ekvivalentní výraz. Například:

```
>>> x = 2
```

```
      x * (2 + x)
=== 2 * (2 + 2)
=== 2 * 4
=== 8
```

Uvažujme dále funkce, které mají jeden parametr a tělo tvoří pouze příkaz návratu:

```
def function(parameter):
    return body
```

Například:

```
def succ(x):
    return x + 1
```

Výraz aplikace *function*(*argumet*) funkce *function* na výraz *argumet* je ekvivalentní výrazu *body*, kde se všechny výskyty proměnné *parameter* nahradí za *argumet*.

Například:

```
      succ(2)
=== 2 + 1
=== 3
```

Nebo také:

```
      succ(succ(0))
=== succ(0 + 1)
=== succ(1)
=== 1 + 1
=== 2
```

Hodnotu výrazu můžeme vypočítat i jinou cestou:

```
      succ(succ(0))
=== succ(0) + 1
=== (0 + 1) + 1
=== 1 + 1
=== 2
```

Pokud *x* je libovolné číslo, dostáváme:

```
    succ(succ(x))
=== succ(x + 1)
=== (x + 1) + 1
=== x + (1 + 1)
=== x + 2
```

Právě jsme ukázali, že výraz `succ(succ(x))` je ekvivalentní s $x + 2$.

Tělo funkce může obsahovat aplikaci funkce:

```
def add_two(x):
    return succ(succ(x))
```

Dostáváme například:

```
    add_two(3)
=== succ(succ(3))
=== succ(4)
=== 5
```

Vezměme si funkci počítající předchůdce čísla:

```
def pred(x):
    return x - 1
```

Dostáváme:

```
    pred(5)
=== 5 - 1
=== 4
```

Pro libovolné číslo x platí:

```
    pred(succ(x))
=== pred(x + 1)
=== (x + 1) - 1
=== x + (1 - 1)
=== x + 0
=== x
```

Funkce druhé mocniny:

```
def square(x):
    return x * x
```

Na pořadí aplikací funkcí obecně záleží. Například:

```
square(succ(2))
=== square(3)
=== 3 * 3
=== 9
```

ale:

```
succ(square(2))
=== succ(4)
=== 5
```

Tedy výrazy `square(succ(2))` a `succ(square(2))` nejsou ekvivalentní.

Predikát rozhodující sudost:

```
def is_even(x):
    return x % 2 == 0
```

Důkaz, že dvojnásobek čísla je sudé číslo:

```
is_even(x * 2)
=== (x * 2) % 2 == 0
=== 0 == 0
=== True
```

Negace:

```
def negation(x):
    return not x
```

Máme:

```
negation(is_even(succ(2)))
=== not is_even(succ(2))
=== not (succ(2) % 2 == 0)
=== not ((2 + 1) % 2 == 0)
=== not (3 % 2 == 0)
=== not (1 == 0)
=== not False
=== True
```

Tedy následník čísla dva není sudé číslo.

Funkce může očekávat funkci jako argument:

```
def apply_to_two(f):
    return f(2)
```

Dostáváme:

```
    apply_to_two(succ)
=== succ(2)
=== 3
```

Nebo:

```
    apply_to_two(square)
=== square(2)
=== 4
```

Lambda výraz

```
(lambda parameter: body)
```

se vyhodnotí na funkci s parametrem *parameter*, která po zavolání vrátí hodnotu výrazu *body*.

Následníka můžeme definovat i takto:

```
>>> succ2 = lambda x: x + 1

    succ2(3)
=== (lambda x: x + 1)(3)
=== 3 + 1
=== 4
```

Lambda výraz umožňuje definovat funkce bez jména. Například:

```
    (lambda x: x * x)(2)
=== 2 * 2
=== 4
```

Funkci nazýváme *konstantní*, pokud vždy vrátí stejnou hodnotu. Příklad konstantní funkce:

```
def const_2(x):
    return 2
```

Pro libovolné *x* dostáváme:

```
    const_2(x)
=== 2
```

Funkce, která po zavolání vrátí konstantní funkci:

```
def const(x):
    return lambda y: x
```

Například máme:

```
>>> const_3 = const(3)
```

```
const_3
=== const(3)
=== (lambda y: 3)

const_3(2)
=== (lambda y: 3)(2)
=== 3
```

Nebo také:

```
const_3(2)
=== (const(3))(2)
=== const(3)(2)
=== (lambda y: 3)(2)
=== 3
```

Při nahrazování během aplikace funkce si musíme dát pozor, aby parametr lambda výrazu v těle funkce nebyl obsažen v argumentu. Následující je tedy chybné, protože parametr y je obsažen v argumentu y.

```
const(y)(2)
=== (lambda y: y)(2)
=== 2
```

Problém vyřešíme vhodným přejmenováním parametru lambda výrazu a jeho výskytů v těle:

```
const(y)(2)
=== (lambda z: y)(2)
=== y
```

Funkci `const` můžeme definovat i takto:

```
>>> const2 = lambda x: lambda y: x
```

Například pro libovolné y máme:

```
const2(y)(2)
=== (lambda x: lambda y: x)(y)(2)
=== (lambda x: lambda z: x)(y)(2)
=== (lambda z: y)(2)
=== y
```

Funkci s více parametry můžeme vyjádřit funkcemi s jedním parametrem. Například:

```
def add(x):  
    return lambda y: x + y
```

Funkce `add` po zavolání na číslo x vrací funkci přičítající k argumentu x . Například:

```
>>> add_3 = add(3)  
  
    add_3(4)  
=== (lambda y: 3 + y)(4)  
=== 3 + 4  
=== 7
```

Součet čísel x a y zapíšeme `add(x)(y)`. Například:

```
    add(3)(4)  
=== (lambda y: 3 + y)(4)  
=== 3 + 4  
=== 7
```

Funkci `add` můžeme zapsat jen pomocí `lambda` výrazů:

```
>>> add2 = lambda x: lambda y: x + y
```

Skutečně:

```
    add2(3)(4)  
=== (lambda x: lambda y: x + y)(3)(4)  
=== (lambda y: 3 + y)(4)  
=== 3 + 4  
=== 7
```

Zde při aplikaci musíme přejmenovat parametr:

```
    add2(y + 1)(y)  
=== (lambda x: lambda y: x + y)(y + 1)(y)  
=== (lambda x: lambda z: x + z)(y + 1)(y)  
=== (lambda z: (y + 1) + z)(y)  
=== (y + 1) + y
```

Definujeme skládání funkcí:

```
def comp(f, g):  
    return lambda x: f(g(x))
```

Pokud má volaná funkce více parametrů, provede se nahrazení každého parametru v těle za příslušný argument. Například:

```
comp(succ, square)
=== lambda x: succ(square(x))
```

Vytvoříme funkci počítající následníka čtverce:

```
>>> succ_square = comp(succ, square)
```

Vyzkoušíme:

```
succ_square(2)
=== comp(succ, square)(2)
=== (lambda x: succ(square(x)))(2)
=== succ(square(2))
=== 5
```

U kompozice funkcí přirozeně záleží na pořadí funkcí, které komponujeme. Tedy funkce počítající čtverec následníka:

```
>>> square_succ = comp(square, succ)
```

se od předchozí liší:

```
square_succ(2)
=== comp(square, succ)(2)
=== (lambda x: square(succ(x)))(2)
=== square(succ(2))
=== 9
```

Kompozice funkcí definovaná lambda výrazy:

```
>>> comp2 = lambda f, g: lambda x: f(g(x))
```

Hodnotou výrazu

```
(then_expr if condition else else_expr)
```

je *then_expr*, pokud je podmínka *condition* splněna, jinak je hodnotou *else_expr*.

Dostáváme, že

```
(then_expr if True else else_expr)
=== then_expr
```

a


```
(then_expr if False else else_expr)  
=== else_expr
```

Například:

```
1 if 1 + 1 == 2 else 2  
=== 1 if True else 2  
=== 1
```

```
1 if 1 + 1 == 3 else 2  
=== 1 if False else 2  
=== 2
```

Protože se hodnoty proměnných nemůžou měnit, nelze k opakování použít cykly. Místo cyklů použijeme *rekurzivní funkce*. Funkce je rekurzivní, pokud ve svém těle volá samu sebe.

Příklad rekurzivní funkce počítající faktoriál.

```
def fact(n):  
    return 1 if n == 0 else n * fact(n - 1)
```

Například máme:

```
fact(3)  
=== 1 if 3 == 0 else 3 * fact(3 - 1)  
=== 1 if False else 3 * fact(3 - 1)  
=== 3 * fact(3 - 1)  
=== 3 * fact(2)  
=== 3 * (2 * fact(1))  
=== 3 * (2 * (1 * fact(0)))  
=== 3 * (2 * (1 * 1))  
=== 6
```