KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

7. Funkce

verze z 8. listopadu 2024

1 Ekvivalentní úpravy výrazů

Uvažujme pro začátek jen aritmetické výrazy, jako jsou například výrazy `1 + x` nebo `x * (2 + x)`. Zavedeme 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. Pokud například nastavíme

```
x = 3
y = 4
```

pak bude hodnota výrazu `(x ** 2) + (y ** 2)` vždy číslo 25.

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
```

2 Funkce s jedním parametrem

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 výraz `(`*argumet*`)`. Například:

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

Při nahrazování musíme výraz *argumet* obecně sevřít do závorek kvůli jednoznačnosti. Většinou to ale potřeba není. Jako například v předchozím příkladě. U složitějších výrazů existuje více cest, jak dojít k jejich hodnotě. Například pro výraz `succ(succ(0))` je

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

jednou z cest a

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

druhou. Zajímavé je, že ať použijeme kteroukoliv cestu, dojdeme vždy ke stejné hodnotě.

Pomocí ekvivalence výrazů můžeme dokazovat vlastnosti programů. Například pokud `x` je libovolné číslo, pak dostáváme:

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

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

Tělo funkce může přirozeně 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
```

Tedy funkce předchůdce je inverzní k funkci následníka.

Na pořadí aplikací funkcí obecně záleží. Vezměme například funkci druhé mocniny:

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

Dále se podívejme na hodnotu výrazu:

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

a hodnotu výrazu:

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

Dostáváme, že výrazy `square(succ(2))` a `succ(square(2))` nejsou ekvivalentní.

Vezměme predikát rozhodující sudost:

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

Můžeme dokázat, že dvojnásobek čísla je sudé číslo:

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

Definujeme funkci vracející negaci obdržené logické hodnoty:

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

Například 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.

3 Funkce vyšších řádů

Funkce může očekávat funkci jako argument. Vezměme například funkci, která aplikuje zadanou funkci na číslo dva:

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

Dostáváme:

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

nebo:

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

Zatím jsme nové funkce vytvářeli pomocí příkazu `def`. Příkaz vytvoří novou funkci a vloží ji do globální proměnné. Zavedeme si výraz, který vytvoří funkci, aniž by ji vložil do proměnné. Výraz

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

se vyhodnotí na funkci s parametrem ***parameter***, která po zavolání vrací hodnotu výrazu ***body***. Právě definovaný výraz nazýváme **lambda výrazem**. Funkci vytvořené lambda výrazem se říká **anonymní funkce**.

Funkci následníka můžeme definovat i takto:

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

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

Jak bylo zmíněno lambda výraz umožňuje definovat funkce bez jména. Například:

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

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

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

Pro libovolné `x` dostáváme:

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

Funkce, která po zavolání vrací 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
```

Technice, kde funkci více parametrů vyjádříme pomocí funkcí jednoho parametru, se říká **currying** podle vědce Haskella Curryho, který ji proslavil.

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 pozičně odpovídající 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))
```

Říkáme, že funkce je **vyššího řádu**, pokud jako argument očekává funkci nebo vrací funkci jako návratovou hodnotu. Příklady funkcí vyššího řádu jsou funkce `apply_twice`, která očekává funkci, funkce `const`, která vrací funkci, a konečně funkce `comp`, která očekává i vrací funkci.

4 Rekurzivní funkce

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)
```

Ukázka použití:

```
    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
```

Otázky a úkoly na cvičení

1. Níže naleznete definice funkcí a příklady jejich volání. U výrazů spočítejte ekvivalentními úpravami jejich hodnoty.

   (a)
   ```
   def double(x):
       return x + x

   double(double(2))
   ```

   (b)
   ```
   def apply_twice(f, x):
       return f(f(x))

   apply_twice(double, 2)
   ```

   (c)
   ```
   def twice(f):
       return lambda x: f(f(x))

   twice(double)(2)
   ```

   (d)
   ```
   quadruple = twice(double)

   quadruple(2)
   ```

(e)
```
def sub(m):
    return lambda n: m - n

sub(5)(3)
```

(f)
```
def swap(f):
    return lambda x: lambda y: f(y)(x)

swap(sub)(5)(3)
```

(g)
```
comp(sub(2), sub(3))(1)
```

(h)
```
def twice2(f):
    return comp(f, f)

twice2(double)(2)
```

2. Naprogramujte funkci `fib`, která počítá prvky Fibonacciho posloupnosti. Musí platit:

```
    fib(0)
=== 0

    fib(1)
=== 1
```

a pro `n` větší než jedna:

```
    fib(n)
=== fib(n - 1) + fib(n - 2)
```

3. Ověřte v interpretu, že funkce `fib` z předchozího úkolu spočítá:

```
>>> fib(50)
12586269025
```