

Paradigmata programování 1 ♦ poznámky k přednášce

3. Rekurze 1

verze z 23. října 2019

1 Příklady

Procentuální podíl

Funkce `percentage-1` počítá procentuální podíl hodnoty parametru `part` v celku `whole`:

```
(defun percentage-1 (part whole)
  (* (/ part whole) 100.0))
```

Použití:

```
CL-USER 1 > (percentage-1 20 300)
6.666667
```

Řekněme, že chceme, aby celek (hodnota parametru `whole`) měl nějakou výchozí hodnotu (v našem případě to bude počet obyvatel ČR).

Aby si uživatel nemusel číslo pamatovat, umožníme mu jako druhý argument použít `t`. Význam aplikace funkce s touto hodnotou druhého argumentu bude, že funkce má použít počet obyvatel České republiky. Funkci tedy přizpůsobíme:

```
(defun percentage-2 (part whole)
  (let ((whole (if (eql whole t)
                    10668641
                    whole))))
  (* (/ part whole) 100.0)))
```

Použili jsme zatím neznámý predikát `eql`, který zjišťuje, zda jsou dvě zadané hodnoty totožné:

```
CL-USER 3 > (eql 1 1)
T

CL-USER 4 > (eql 1 t)
```

```
NIL
```

```
CL-USER 5 > (eql nil nil)  
T
```

Od už známého predikátu `=` se liší tím, že ten požaduje jako argumenty čísla:

```
CL-USER 6 > (= 1 t)
```

```
Error: In = of (1 T) arguments should be of type NUMBER.
```

Pokud víme, že budeme porovnávat čísla, je lepší použít funkci `=`, protože nám pomůže odhalit, když omylem zkusíme porovnat něco jiného.

Ve funkci jsme použili speciální operátor `let`, pomocí kterého jsme zastínili existující vazbu symbolu `whole` (vzpomeňte si na minulou přednášku).

Test funkce:

```
CL-USER 8 > (percentage-2 100381 10668641)  
0.9408977
```

```
CL-USER 9 > (percentage-2 100381 t)  
0.9408977
```

Jiná možnost je použít funkci `percentage-1`:

```
(defun percentage-3 (part whole)  
  (percentage-1 part  
    (if (eql whole t)  
        10668641  
        whole)))
```

Stejný test jako u funkce `percentage-2` ukáže, že funkce `percentage-3` by mohla fungovat:

```
CL-USER 10 > (percentage-3 100381 10668641)  
0.9408977
```

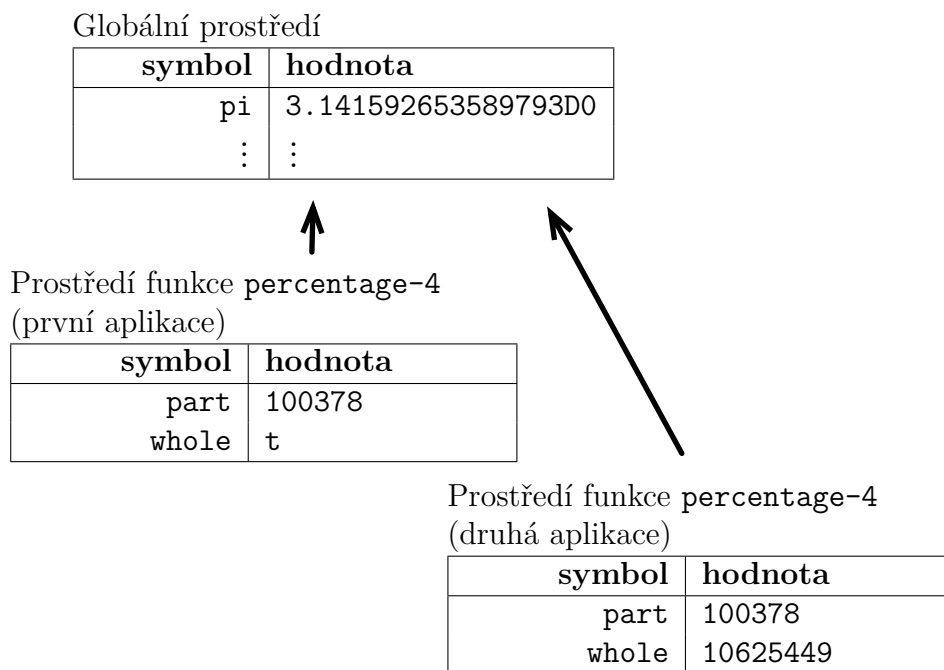
```
CL-USER 11 > (percentage-3 100381 t)  
0.9408977
```

A ještě jedna, nejdůležitější možnost:

```
(defun percentage-4 (part whole)
  (if (eq1 whole t) ;je-li whole rovno t
      (percentage-4 part 10668641) ;aplikujeme znovu percentage-4
      (* (/ part whole) 100.0))) ;jinak výpočet
```

(**Středníkem** začíná komentář, který se nevyhodnocuje.)

V těle funkce vidíme *rekurzivní aplikaci* téže funkce. Informace o vyhodnocovacím procesu z minulých přednášek nám umožní pochopit, jak funkce pracuje. K tomu nám pomůže obrázek prostředí, která se používají během aplikace (percentage-4 100378 t):



Prohledávání intervalu

Napišeme funkci (predikát), která zjistí, zda se mezi danými celými čísly nachází druhá mocnina celého čísla (tzv. *čtverec*).

Budeme potřebovat predikát rozhodující, zda dané celé číslo je čtverec:

```
(defun squarep (n)
  (= (power2 (round (sqrt n))) n))
```

(Funkci `power2` jsme programovali na minulém cvičení. Funkce `round` zaokrouhluje na nejbližší celé číslo.)

A teď k hlavnímu úkolu v tomto příkladu. Množina čísel mezi zadanými dvěma čísly se nazývá *interval*. V našem příkladě jde ovšem jen o celá čísla. Například mezi čísla 2 a 5 (včetně) najdeme čísla 2, 3, 4, 5. Jde o interval s *koncovými body* 2 a 5. Tento interval obsahuje čtverec, a to číslo 4. Interval s koncovými body 5 a


5 obsahuje jen číslo 5 (a žádný čtverec), pro koncové body 5 a 4 neobsahuje nic, protože číslo 5 není menší nebo rovno číslu 4.

Interval prohledáme tak, že

1. Podíváme se, zda není prázdný. Pokud je, určitě neobsahuje čtverec.
2. Když není prázdný, podíváme se na jeho levý koncový bod. Pokud jde o čtverec, prohledávání bylo úspěšné, funkce může skončit.
3. Levý koncový bod čtvercem není. Vypustíme ho z intervalu a pro nový interval provedeme tutéž činnost.

Výslednou funkci vidíme tady:

```
(defun contains-square-p (a b)
  (if (> a b)                                ;když je interval prázdný
      nil                                    ;čtverec neobsahuje
      (if (squarep a)                        ;je-li dolní konec čtverec
          t                                ;interval čtverec obsahuje
          (contains-square-p (+ a 1) b)))    ;jinak zkoumáme
                                             ;interval [a + 1, b]
```


rekurzivní aplikace

Jiná varianta téže funkce:

```
(defun contains-square-p (a b)
  (cond ((> a b) nil)
        ((squarep a) t)
        (t (contains-square-p (+ a 1) b))))
```

Zde jsme použili makro `cond`, které používáme při větvení na více než dvě větve.

Odbočka: makro `cond`

```
(cond (  $\overbrace{((> a b) \quad nil)}^{\text{větev}}$  )
      (  $\underbrace{((squarep a) t)}_{\text{podmínka větve}} \quad \underbrace{(t (contains-square-p? (+ a 1) b)))}_{\text{tělo větve}}$  )
      ... )
```

} další větve

1. Postupně se vyhodnocují podmínky větví.
2. Jakmile je nějaká splněna, vyhodnotí se tělo příslušné větve.
3. Další podmínky se nevyhodnocují.
4. Vráť se výsledek vyhodnoceného těla, pokud žádná podmínka nebyla splněna, vrátí se `nil`.

A ještě jedna varianta stejné funkce, tentokrát s použitím *logických spojek*.

```
(defun contains-square-p (a b)
  (and (<= a b)
       (or (squarep a)
            (contains-square-p (+ a 1) b))))
```

Odbočka: zobecněné logické hodnoty, makra `and` a `or`, funkce `not`

Zopakujme informaci o podmíněných výrazech z první přednášky: **Podmíněný výraz** je složený výraz s operátorem `if`. Takový výraz musí mít tři argumenty. (Lisp umožňuje i verzi se dvěma argumenty, ale tu nebudeme používat.) Vyhodnocuje se takto:

Vyhodnocení výrazu (`if a b c`)

1. Vyhodnotí se *a* na hodnotu *u*.
2. Pokud je *u* rovno `NIL`, vyhodnotí se *c* a vrátí jeho hodnota.
3. Pokud není, vyhodnotí se *b* a vrátí jeho hodnota.

Z bodu 3 plyne, že hodnotou prvního argumentu operátoru `if` může být i jiná hodnota než `t` nebo `nil`. Pro každou jinou hodnotu než `nil` se vyhodnotí podvýraz *b*. Každá hodnota různá od `nil` tedy může být chápána jako logická hodnota *Pravda*. Takto chápaným hodnotám se říká *zobecněné logické hodnoty*; v Lispu (a většině dalších jazyků) se používají.

Operátor `and`

```
(and e1 e2 ... en)
```

Vrací *Pravdu*, pokud se všechny *ei* vyhodnotí na *Pravdu*, jinak vrací `nil`. Používá tzv. *zkrácené vyhodnocování*: vyhodnocuje (zleva doprava) pouze tolik svých argumentů, aby mohl rozhodnout o výsledku:

```
CL-USER 20 > (and (= 1 1) (= 1 0) (= (/ 1 0) 0))
NIL
```

Operátor or

```
(or e1 e2 ... en)
```

Vrací *Pravdu*, pokud se některé e_i vyhodnotí na *Pravdu*, jinak vrací `nil`. Používá zkrácené vyhodnocování:

```
CL-USER 21 > (or (= 2 2) (= (/ 1 0) 0))
T
```

Funkce not

Funkce `not` počítá *logickou negaci*:

```
CL-USER 22 > (not (> 2 1))
NIL
```

Pevný bod funkce cos

Hledáme přibližnou hodnotu čísla x takového, že $\cos x = x$. To jde s libovolnou přesností udělat *metodou postupných aproximací*:

Začneme libovolným číslem x_0 a budeme na ně stále aplikovat funkci `cos`:

$$x_1 = \cos x_0$$

$$x_2 = \cos x_1$$

$$x_3 = \cos x_2$$

$$x_4 = \cos x_3$$

$$\vdots$$

Budeme získávat čísla, která se budou stále více přibližovat hledané hodnotě. (To je zvláštnost funkce `cos`; pro jiné funkce to samozřejmě nejde, zkuste si třeba funkci $f(x) = x^2$.)

Matematickými úvahami můžeme zjistit, že pokud je $|x_{n+1} - x_n| \leq \varepsilon$ (ε je zadaná největší přípustná chyba), pak se x_{n+1} liší od hledaného čísla nejvýše o ε a je to tedy dostatečné přiblížení k hledanému číslu.

Následující řešení používá na přibližné porovnávání predikát `approx=`, který má tři parametry: dvě čísla, která porovnáváme, a požadovanou přesnost. U čísel vypočítá absolutní hodnotu jejich rozdílu (tedy jejich vzdálenost) a výsledek porovná s požadovanou přesností.

```
(defun approx= (a b epsilon)
  (<= (abs (- a b)) epsilon))
```

Testy:

```
CL-USER 23 > (approx= 1 2 0.5)
NIL

CL-USER 24 > (approx= 22/7 pi 0.01)
T
```

Funkce `cos-fixpoint-iter` vypočítá hledané číslo na základě počáteční hodnoty a požadované přesnosti. Funkce `cos-fixpoint` je řešením příkladu, používá předchozí funkci s počátečním bodem 0:

```
(defun cos-fixpoint-iter (x epsilon)
  (let ((y (cos x)))
    (if (approx= x y epsilon)
        y
        (cos-fixpoint-iter y epsilon))))

(defun cos-fixpoint (epsilon)
  (cos-fixpoint-iter 0 epsilon))
```

Testy:

```
CL-USER 46 > (cos-fixpoint 0.1)
0.7013688

CL-USER 47 > (cos-fixpoint 0.01)
0.73560477

CL-USER 48 > (cos-fixpoint 0.001)
0.7387603

CL-USER 49 > (cos-fixpoint 0.000001)
0.73908484

CL-USER 50 > (cos *)
0.7390853
```

2 Rekurzivní funkce a rekurzivní výpočetní proces

Rekurzivní funkce

Funkce je *rekurzivní*, když ve svém těle obsahuje aplikaci sebe sama.

- je poznat ze zdrojového kódu funkce
- funkce `percentage-4`, `contains-square-p` (všechny verze), `cos-fixpoint-iter` jsou rekurzivní

Rekurzivní výpočetní proces

Výpočetní proces je *rekurzivní*, když **během** aplikace funkce dojde znovu k aplikaci téže funkce.

- je poznat, když program běží
- některá aplikace funkce by k aplikaci téže funkce vést neměla (*ukončovací podmínka*)

Speciální případ:

Iterativní výpočetní proces

Výpočetní proces je *iterativní*, když **na konci** aplikace funkce dojde opět k aplikaci téže funkce.

- uvedené funkce generují iterativní výpočetní proces

3 Další příklady

Nyní si ukážeme rekurzivní funkce, které generují rekurzivní výpočetní proces, ale nikoliv iterativní výpočetní proces.

Obecná mocnina

Na minulém cvičení jsme programovali funkce na umocňování:

```
(defun power2 (a)
  (* a a))

(defun power3 (a)
  (* a (power2 a)))
```



```
(defun power4 (a)
  (* a (power3 a)))

(defun power5 (a)
  (* a (power4 a)))
```

Obecnou (n -tou) mocninu čísla a můžeme vypočítat takto:

1. Je-li $n = 0$, je výsledkem číslo 1. (To neplatí pro $a = 0$, ale tuto možnost pomineme.)
2. Je-li $n > 0$, je výsledkem číslo $a \cdot a^{n-1}$.

Napsáno ve funkci:

```
(defun power (a n)
  (if (= n 0)
      1
      (* a (power a (- n 1)))))
```

Funkce `power` je rekurzivní a **negeneruje** iterativní výpočetní proces, protože aplikaci sebe sama neprovádí nakonec (po aplikaci musí výsledek ještě vynásobit číslem a).

Faktoriál

Jak víme, faktoriál nezáporného celého čísla n je dán tímto předpisem:

$$n! = \begin{cases} 1 & \text{když } n = 0 \\ n \cdot (n-1)! & \text{když } n > 0 \end{cases}$$

Napsáno do funkce:

```
(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Tato verze faktoriálu generuje iterativní výpočetní proces:

```

(defun fact-iter (n ir)
  (if (= n 0)
      ir
      (fact-iter (- n 1) (* ir n))))

(defun fact (n)
  (fact-iter n 1))

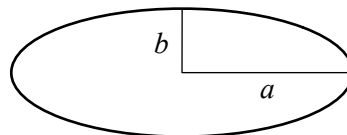
```

Výpočet probíhá možná přirozenějším způsobem, než u předchozí verze, protože kopíruje způsob, jak bychom počítali faktoriál ručně. Například $5!$ bychom počítali tak, že bychom postupně násobili dvě čísla a pamatovali si mezivýsledky (*ir*, *intermediate result*) a čísla, kterými je třeba ještě vynásobit:

krok	zbývá vypočítat	mezivýsledek
1.	5!	1
2.	4!	$5 \cdot 1 = 5$
3.	3!	$4 \cdot 5 = 20$
4.	2!	$3 \cdot 20 = 60$
5.	1!	$2 \cdot 60 = 120$
6.	0!	$1 \cdot 120 = 120$

Otázky a úkoly na cvičení

- Podívejte se na definici funkce `percentage-2`. Ve kterém prostředí se při její aplikaci vyhodnocuje výraz `(eq1 whole t)` a ve kterém výraz `(/ part whole)`?
- Jaký je rozdíl mezi rekurzivní funkcí a rekurzivním výpočetním procesem?
- Existuje rekurzivní funkce, která nikdy negeneruje rekurzivní výpočetní proces?
- Existuje nerekurzivní funkce, která generuje rekurzivní výpočetní proces?
- Obsah elipsy s poloosami a a b je πab .



Proto jej můžeme vypočítat pomocí následující funkce:

```

(defun ellipse-area (a b)
  (* pi a b))

```

Když $a = b$, je elipsa kružnicí. Upravte funkci tak, aby v takovém případě stačilo místo druhého argumentu zadat t . Udělejte to co nejvíce způsoby, jeden z nich by měl být rekurzivní.

6. Lze funkci `squarep` napsat jiným způsobem, než jak je uvedeno v textu?
7. Napište funkci `my-gcd` (*greatest common divisor*; funkce `gcd` už v Lispu je), která Eukleidovým algoritmem vypočte největší společný dělitel zadaných dvou přirozených čísel:

```
CL-USER 1 > (my-gcd 10 15)
5

CL-USER 2 > (my-gcd 5 3)
1

CL-USER 3 > (my-gcd 5 10)
5

CL-USER 4 > (my-gcd 9 24)
3
```

Jak víme, Eukleidův algoritmus vychází z následujícího poznatku:

$$\gcd(a, b) = \begin{cases} a & \text{jestliže } b = 0, \\ \gcd(b, c) & \text{jinak (} c \text{ je zbytek po dělení } a : b \text{).} \end{cases}$$

Na zjištění zbytku po dělení použijte funkci `rem`.

8. Zvolme kladné číslo a . Podobně jako dříve pro funkci `cos` můžeme metodou postupných aproximací najít pevný bod funkce f dané předpisem

$$f(x) = \frac{x + \frac{a}{x}}{2}.$$

Jak víme, pevným bodem bude číslo x , pro které platí $f(x) = x$.

Zajímavé je, že takovým pevným bodem je číslo \sqrt{a} . (K ověření stačí dosadit \sqrt{a} do vzorečku; vyjde $f(\sqrt{a}) = \sqrt{a}$.) Metodou postupných aproximací tedy v případě této funkce f najdeme odmocninu z čísla a .

Napište funkci `heron-sqrt`, která metodou postupných aproximací vypočítá odmocninu ze zadaného čísla se zadanou přesností. Přesnost testujte tak, že přibližné číslo umocníte na druhou a porovnáte s a .

9. Napište „hloupou“ funkci na výpočet součtu prvků intervalu celých čísel.
10. Upravte ji tak, aby generovala iterativní výpočetní proces.
11. Upravte funkci `power`, aby generovala iterativní výpočetní proces.

12. Číslo π lze s libovolnou přesností vypočítat pomocí *Leibnizovy formule*:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Dosažená přesnost odhadu čísla $\frac{\pi}{4}$ je přitom dána posledním přičítaným (odečítaným) zlomkem. Napište funkci `leibniz`, která vypočítá číslo π se zadanou přesností. Napište jak obyčejnou, tak iterativní verzi této funkce.