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

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ⋄ poznámky k přednášce

3. Rekurzivní funkce

verze z 15. října 2024

1 Příklady

Vzdálenost bodů v rovině

Jako úlohu k minulé přednášce jste psali funkci na výpočet délky přepony pravoúhlého trojúhelníka. Tady je možné řešení:

```
(defun hypotenuse (a b)
  "Computes the hypotenuse of a right triangle with legs a, b by
means of the Pythagorean theorem."
  (sqrt (+ (power2 a) (power2 b))))
```

Pomocí funkce `hypotenuse` jste pak měli napsat funkci na výpočet vzdálenosti dvou bodů v rovině zadaných kartézskými souřadnicemi. Možné řešení:

```
(defun point-distance (A-x A-y B-x B-y)
  "Computes the distance of point A with coordinates A-x and A-y
and point B with coordinates B-x and B-y."
  (hypotenuse (- A-x B-x) (- A-y B-y)))
```

Příklad demonstruje jednu samozřejmou, ale důležitou věc: *Uživatelské funkce mohou používat jiné uživatelské funkce.* To nám umožňuje dělit program na co nejmenší ucelené části.

Procentuální podíl

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

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

Použití:

```
> (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 tímto druhým argumentem bude, že funkce má použít počet obyvatel České republiky (k 31. 12. 2023). Funkci tedy přizpůsobíme:

```
(defun percentage-2 (part whole)
  (let ((whole (if (eql whole t)
                   10900555
                 whole)))
    (* (/ part whole) 100.0)))
```

Test funkce:

```
> (percentage-2 102293 10900555)
0.9384201

> (percentage-2 102293 t)
0.9384201
```

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

Globální prostředí

symbol	hodnota
pi	3.141592653589793D0
⋮	⋮

Prostředí funkce `percentage-2`

symbol	hodnota
part	102293
whole	t

Prostředí operátoru `let`

symbol	hodnota
whole	10900555

Jiná možnost je použít funkci `percentage-1`:

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

Funkci můžeme opět otestovat:

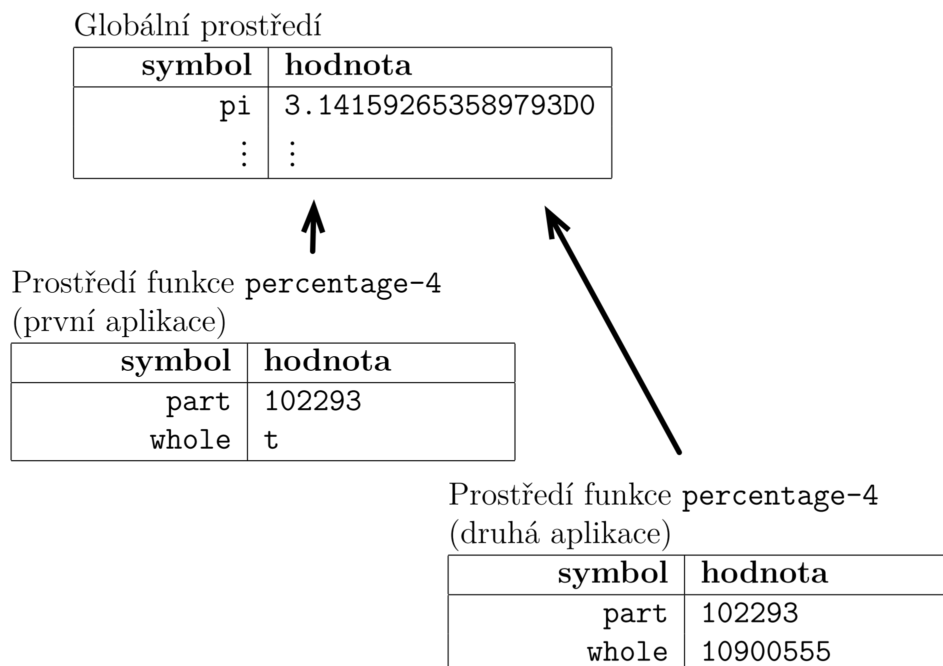
```
> (percentage-3 102293 10900555)
0.9384201

> (percentage-3 102293 t)
0.9384201
```

A ještě jedna, nejzajímavější možnost:

```
(defun percentage-4 (part whole)
  (if (eql whole t)                     ;je-li whole rovno t
      (percentage-4 part 10900555)      ;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 102293 t)`:



Prohledávání intervalu

Napíšeme 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:

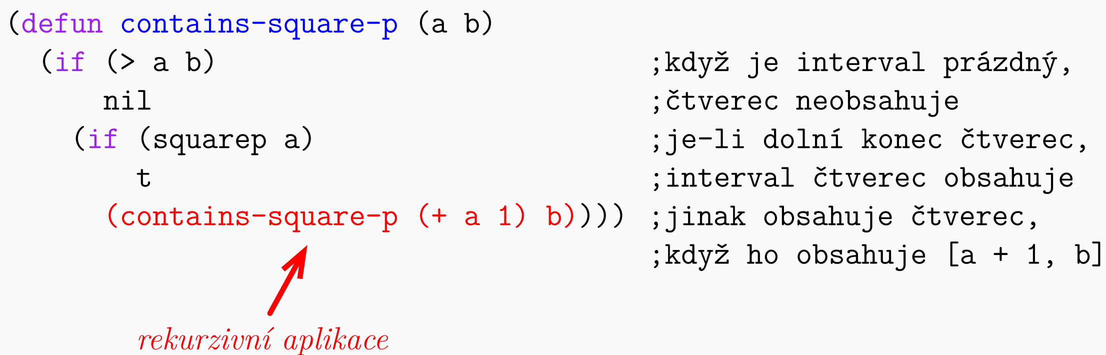
```
(defun squarep (n)
  (= (power2 (round (sqrt n))) n))
```

(Funkci `power2` jsme programovali na minulém cvičení.)

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 čísly 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 následující funkcí:

```
(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 obsahuje čtverec,
                                               ;když ho obsahuje [a + 1, b]
```



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 speciální operátor `cond`, který používáme při větvení na více než dvě větve.

Odbočka: speciální operátor `cond`

```
              větev
(cond ((> a b)  nil )
 podmínka větve  tělo větve
      ((squarep a) t)                  } další větve
      (t (contains-square-p (+ a 1) b)))
```

1. Postupně se vyhodnocují podmínky větví.
2. Jakmile je nějaká splněna, vyhodnotí se tělo příslušné větve.
3. Další podmínky se nevyhodnocují.
4. Vrátí 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))))
```

Všimněte si, že funkci `contains-square-p` jsme napsali **deklarativně**: neřešili jsme, *jak* má být výsledek vypočítán, ale *co* má výsledkem být. To je nejlépe vidět z poslední verze:

Interval $[a, b]$ *obsahuje čtverec, když je neprázdný a buď je číslo a čtverec, nebo interval* $[a + 1, b]$ *obsahuje čtverec.*

Odbočka: zobecněné logické hodnoty, speciální operátory `and` a `or`

Zopakujme informaci o podmíněných výrazech z první přednášky: **Podmíněný výraz** je složený výraz s operátorem `if`. Takový výraz musí mít za operátorem tři podvýrazy. (Lisp umožňuje i verzi se dvěma, 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 podvýrazu za operátorem `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.

Operátory `and` a `or` jsem bez komentáře uvedl už v textu k první přednášce.

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 podvýrazů, aby mohl rozhodnout o výsledku:

```
> (and (= 1 1) (= 1 0) (= (/ 1 0) 0))
NIL
```

Operátor `or`

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

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

```
> (or (= 2 2) (= (/ 1 0) 0))
T
```

Funkce `not`

Funkce `not` byla definována na minulé přednášce. Počítá *logickou negaci* :

```
> (not (> 2 1))
NIL
```

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)))))
```

Všimněte si, jak jsme k definici funkce `fact` došli. Nejprve jsme si řekli *co je faktoriál čísla* n (jeho vyjádřením pomocí faktoriálu čísla $n-1$) a podle toho ji pak definovali.

K výpočtu faktoriálu čísla $n > 0$ potřebujeme znát faktoriál $n-1$. Aby výpočet vedl k cíli, je nutné, aby problém vypočítat $(n-1)!$ byl jednodušší, než problém původní. Ale on jednodušší je, protože je blíže *základnímu případu* $n = 0$. K výpočtu faktoriálu čísla 0 už nepotřebujeme počítat faktoriál jiného čísla: odpověď rovnou známe.

Obecný princip řešení daného problému rekurzivním výpočtem je

1. stanovit základní případ nebo případy, pro které není nutné používat rekurzi,
2. ostatní případy vyjádřit rekurzivně jednodušším problémem, který je blíže základním případům.

Základním případem u problému vypočítat $n!$ je případ $n = 0$. Faktoriál nuly umíme vypočítat bez použití rekurze. Ostatní případy, tedy případy, kdy $n > 0$, přiblížíme základnímu případu tím, že místo $n!$ zadáme úkol vypočítat $(n-1)!$. Pomocí hodnoty $(n-1)!$ pak původně hledanou hodnotu $n!$ zjistíme vynásobením číslem n.

Při psaní funkce, která používá sama sebe, musíme pracovat, **jako by už byla napsána a fungovala**. Například u tohoto výrazu v těle funkce `fact`

```
(* n (fact (- n 1)))
```

se nesmíme snažit rozebírat, jak pracuje použitá funkce `fact`; nesmí nás mást, že ji teprve programujeme.

2 Rekurzivní funkce

Rekurzivní funkce

Funkce je *rekurzivní*, když ve svém těle obsahuje aplikaci sebe samé (*rekurzivní aplikaci*).

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

Speciální případ:

Koncově rekurzivní funkce

Funkce je *koncově rekurzivní*, když její rekurzivní aplikace v těle je poslední aplikací.

- funkce `percentage-4` a `contains-square-p` (všechny verze) jsou koncově rekurzivní

3 Další příklady

Nyní si ukážeme rekurzivní funkce, které nejsou koncově rekurzivní.

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, ale není koncově rekurzivní, protože její rekurzivní aplikace není poslední aplikací — po ní ještě následuje aplikace funkce `*`.

Na prezentaci k přednášce můžete vidět animaci vytváření prostředí při jednotlivých rekurzivních aplikacích funkce `power`.

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:

$$\begin{aligned} x_1 &= \cos x_0 \\ x_2 &= \cos x_1 \\ x_3 &= \cos x_2 \\ x_4 &= \cos x_3 \\ &\vdots \end{aligned}$$

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:

```
> (approx-= 1 2 0.5)
NIL

> (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:

```
> (cos-fixpoint 0.1)
0.7013688

> (cos-fixpoint 0.01)
0.73560477

> (cos-fixpoint 0.001)
0.7387603

> (cos-fixpoint 0.000001)
0.73908484

> (cos *)
0.7390853
```

Faktoriál podruhé

Faktoriál napsaný pomocí koncové rekurze:

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

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

Funkce může obsahovat více rekurzivních aplikací, než jen jednu. Ukážeme si to na příkladě Fibonacciho posloupnosti.

Fibonacciho posloupnost

Jde o tuto posloupnost: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$ Její prvky jsou dány

následujícím předpisem:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-2} + a_{n-1}$$

(Každý prvek kromě prvních dvou je součtem předcházejících dvou prvků.) Následující funkce vrací daný prvek Fibonacciho posloupnosti:

```
(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 2)) (fib (- n 1))))))
```

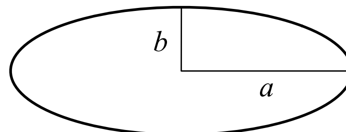
Nové symboly

speciální operátor: `cond`

Otázky a úkoly na cvičení

1. Podívejte se na definici funkce `percentage-2`. Ve kterém prostředí se při její aplikaci vyhodnocuje výraz `(eql whole t)` a ve kterém výraz `(/ part whole)`?

2. 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.

3. Lze funkci `squarep` napsat jiným způsobem, než jak je uvedeno v textu?

4. Předpokládejme, že funkce `fact` je definována jedním z uvedených způsobů. K čemu by vedlo vyhodnocení výrazu `(fact 10)`, kdyby operátor `if` byl funkce?

5. Napište funkci `gcd` (*greatest common divisor*), která Eukleidovým algoritmem vypočte největší společný dělitel zadaných dvou přirozených čísel:

```
> (my-gcd 10 15)
5

> (my-gcd 5 3)
1

> (my-gcd 5 10)
5

> (my-gcd 9 24)
3
```

   Jak víme, Eukleidův algoritmus vychází z následujícího poznatku:

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

   Na zjištění zbytku po dělení použijte funkci `rem`. *(do jazyka)*

6. 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.

7. Napište „hloupou“ funkci na výpočet součtu prvků intervalu celých čísel.

8. Napište ji tak, aby používala koncovou rekurzi.

9. Upravte funkci `power`, aby používala koncovou rekurzi.

10. Známé kritérium říká, že celé kladné číslo je dělitelné devíti, právě když je jeho ciferný součet dělitelný devíti. Napište predikát, který tímto kritériem zjistí, zda kladné celé číslo, na které je aplikován, je dělitelné devíti. Použijte funkci `digit-count` ze zdrojového kódu a funkci `digit` z cvičení k minulé přednášce. Funkce `digit-count` vrací počet cifer zadaného čísla:

    ```
    > (digit-count 123)
    3
    ```

    Pochopitelně nesmíte použít žádnou funkci na výpočet celočíselného podílu a zbytku po dělení dvou čísel.

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

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

    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 ji jak obyčejným způsobem, tak pomocí koncové rekurze.