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. Rekurze 1

1 Příklady

Procentuální podíl

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

```
(define (percentage-1 part whole)
  (* (/ part whole) 100))
```

Použití:

```
> (percentage-1 20 300)
6 2/3
> (percentage-1 #i20 300)
#i6.666666666666667
```

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

```
(define the-whole 10625449)
```

Aby si uživatel nemusel hodnotu ani její jméno pamatovat, umožníme mu jako druhý argument použít `#true`. Význam aplikace procedury s touto hodnotou druhého argumentu bude, že procedura má použít číslo `the-whole`. Proceduru tedy přizpůsobíme:

```
(define (percentage-2 part whole)
  (let ((whole (if (eq? whole #true)
                   the-whole
                   whole)))
    (* (/ part whole) 100)))
```

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

```
> (eq? 1 1)
#true
> (eq? 1 #true)
#false
> (eq? #false #false)
#true
```

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

```
> (= 1 #true)
=: expects a number as 2nd argument, given #true
```

V proceduře jsme použili speciální operátor `let`, pomocí kterého jsme zastínili existující vazbu symbolu `whole` (viz látku z minulé přednášky).

Test procedury:

```
> (percentage-2 #i100378 10625449)
#i0.9446941959817415
> (percentage-2 #i100378 #true)
#i0.9446941959817415
```

Jiná možnost je použít proceduru `percentage-1`:

```
(define (percentage-3 part whole)
  (percentage-1 part
                (if (eq? whole #true)
                    the-whole
                    whole)))
```

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

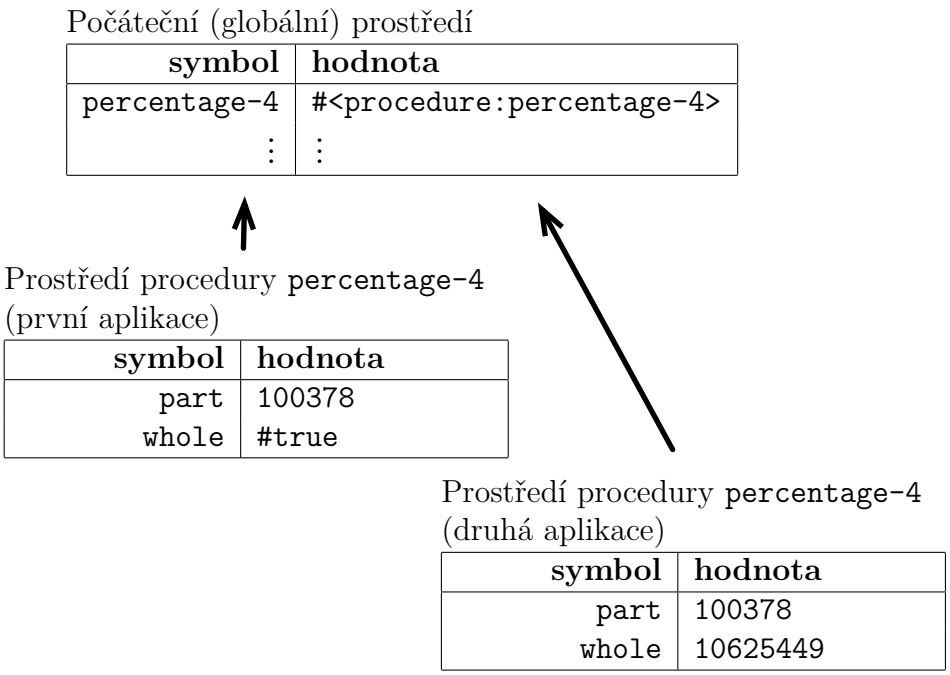
```
> (percentage-3 #i100378 10625449)
#i0.9446941959817415
> (percentage-3 #i100378 #true)
#i0.9446941959817415
```

A ještě jedna, nejdůležitější možnost:

```
(define (percentage-4 part whole)
  (if (eq? whole #true)              ;je-li whole rovno #true
      (percentage-4 part the-whole)  ;aplikujeme znovu percentage-4
      (* (/ part whole) 100)))       ;jinak výpočet
```

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

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



Prohledávání intervalu

Napíšeme proceduru (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. Ten poznáme podle toho, že jeho odmocnina je celé číslo. Použijeme proceduru `sqrt` na druhou odmocninu, kterou už známe, a nový predikát `integer?`, který zjišťuje, zda zadané číslo je celé:

```
> (integer? 10)
#true
> (integer? #i10)
#true
> (integer? 1/2)
#false
```

Teď můžeme napsat první verzi predikátu:

```
(define (square-1? n)
  (if (integer? (sqrt n))
      #true
      #false))
```

Doufám, že jste si všimli, že je zbytečně složitá (takhle programuje začátečník). Jednodušší a správnější je napsat prostě

```
(define (square-2? n)
  (integer? (sqrt n)))
```

Proceduru by bylo dobré ještě vylepšit, protože takto by vracela nesprávné výsledky, pokud by procedura `sqrt` počítala odmocninu nepřesně (což by mohla). Bez dalšího vysvětlování uvedu vylepšenou verzi:

```
(define (square-3? n)
  (= (sqr (round (sqrt n))) n))
```

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 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, procedura může skončit.
3. Levý koncový bod čtvercem není. Vypustíme ho z intervalu a pro nový interval provedeme tutéž činnost.

Výslednou proceduru vidíme tady:

```
(define (contains-square-1? a b)
  (if (> a b)                              ;když je interval prázdný
      #false                               ;čtverec neobsahuje
      (if (square-2? a)                    ;je-li dolní konec čtverec
          #true                            ;interval čtverec obsahuje
          (contains-square-1? (+ a 1) b))) ;jinak zkoumáme
                                           ;interval [a + 1, b]
```

Jiná varianta téže procedury:

```
(define (contains-square-2? a b)
  (cond ((> a b) #false)
        ((square? a) #true)
        (#true (contains-square-2? (+ 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) #false)
  podmínka větve  tělo větve
      ((square? a) #true)                          } další větve
      (#true (contains-square-2? (+ 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, dojde k chybě.

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

```
(define (contains-square-3? a b)
  (and (<= a b)
       (or (square? a)
           (contains-square-3? (+ a 1) b))))
```

Odbočka: speciální operátory `and` a `or`, procedura `not`

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

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

```
> (and (= 1 1) (= 1 0) (= (/ 1 0) 0))
#false
```

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

Vrací `#true`, pokud se některé *ei* vyhodnotí na `#true`, jinak `#false`. Používá *zkrácené vyhodnocování*:

```
> (or (= 2 2) (= (/ 1 0) 0))
#true
```

Procedura `not` počítá *logickou negaci*:

```
> (not (= 1 1))
#false
```

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 $|x_{n+1} - x_n| \leq \epsilon$ (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 (*precision*). U čísel vypočítá absolutní hodnotu jejich rozdílu (tedy jejich vzdálenost) a výsledek porovná s požadovanou přesností.

```
(define (approx-= a b prec)
  (<= (abs (- a b)) prec))
```

Testy:

```
> (approx-= 1 2 0.5)
#false
> (approx-= 22/7 pi 0.01)
#true
```

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

```
(define (cos-fixpoint-iter x prec)
  (let ((y (cos x)))
    (if (approx-= x y prec)
        y
        (cos-fixpoint-iter y prec))))

(define (cos-fixpoint prec)
  (cos-fixpoint-iter 0 prec))
```

Testy:

```
> (cos-fixpoint 0.1)
#i0.7013687736227565
> (cos-fixpoint 0.001)
#i0.7387603198742112
> (cos-fixpoint 0.00000001)
#i0.7390851366465718
> (cos-fixpoint 0.00000000000001)
#i0.7390851332151572
> (cos #i0.7390851332151572)
#i0.7390851332151629
```

2 Rekurzivní procedury a rekurzivní výpočetní proces

Definition 1 (rekurzivní procedura)**.** Procedura je *rekurzivní*, když ve svém těle obsahuje aplikaci sebe sama.

- je poznat ze zdrojového kódu procedury
- procedury `percentage-4`, `contains-square-1?`, `contains-square-2?`, `contains-square-3?`, `cos-fixpoint-iter` jsou rekurzivní

Definition 2 (rekurzivní výpočetní proces)**.** Výpočetní proces je *rekurzivní*, když **během** aplikace procedury dojde znovu k aplikaci téže procedury.

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

Speciální případ:

Definition 3 (iterativní výpočetní proces)**.** Výpočetní proces je *iterativní*, když **na konci** aplikace procedury dojde opět k aplikaci téže procedury.

- uvedené procedury generují iterativní výpočetní proces

3 Další příklady

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

Obecná mocnina

Na minulém cvičení jsme programovali procedury na umocňování:

```
(define (power2 a)
  (* a a))

(define (power3 a)
  (* a (power2 a)))

(define (power4 a)
  (* a (power3 a)))

(define (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 v proceduře:

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

Procedura `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 procedury:

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

Tato verze faktoriálu generuje iterativní výpočetní proces:

```
(define (fact-2-iter n ir)
  (if (= n 0)
      ir
      (fact-2-iter (- n 1) (* ir n))))

(define (fact-2 n)
  (fact-2-iter n 1))
```

Výpočet probíhá možná přirozenějším způsobem, než u procedury `fact-1`, 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 je třeba ještě vynásobit:

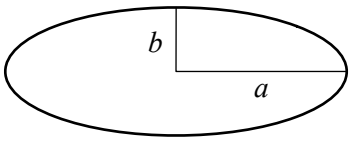
krok	**zbývá vypočítat**	**mezivýsledek**
1.	5!	1
2.	4!	$5 \cdot 1 = 5$
3.	3!	$4 \cdot 5 = 20$
4.	2!	$3 \cdot 20 = 60$
5.	1!	$2 \cdot 60 = 120$
5.	0!	$1 \cdot 120 = 120$

Kontrolní otázky

1. Podívejte se na definici procedury `percentage-2`. Ve kterém prostředí se při její aplikaci vyhodnocuje výraz `(eq? whole #true)` a ve kterém výraz `(/ part whole)`?
2. Jaký je rozdíl mezi rekurzivní procedurou a rekurzivním výpočetním procesem?
3. Existuje rekurzivní procedura, která nikdy negeneruje rekurzivní výpočetní proces?
4. Existuje nerekurzivní procedura, která generuje rekurzivní výpočetní proces?

Otázky a úkoly na cvičení

1. Obsah elipsy s poloosami a a b je πab.



   Proto jej můžeme vypočítat pomocí následující procedury:

```
(define (ellipse-area-1 a b)
  (* pi a b))
```

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

2. Napište proceduru `my-gcd` (*greatest common divisor*; procedura `gcd` už ve Scheme je), která Eukleidovým algoritmem vypočte největší společný dělitel zadaných dvou přirozených čísel:

```
> (my-gcd 5 3)
1
> (my-gcd 12 8)
4
> (my-gcd 9 24)
3
```

   Jak víme, Eukleidův algoritmus vychází z následujícího poznatku:

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

   Na zjištění zbytku po dělení použijte proceduru `remainder`.

3. 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 proceduru `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. (Tedy jinak, než jak jsme dělali na prvním cvičení.)

4. Ukažte na příkladě, že kdyby procedura `square-2?` používala na výpočet odmocniny proceduru `heron-sqrt`, tak by nemusela vracet správné výsledky.

5. Napište „hloupou“ proceduru na výpočet součtu prvků intervalu celých čísel.

6. Upravte ji tak, aby generovala iterativní výpočetní proces.

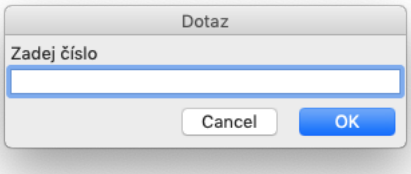
7. Upravte proceduru `power`, aby generovala iterativní výpočetní proces.

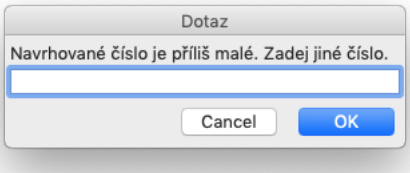
8. Číslo π lze s libovolnou přesností vypočítat pomocí *Leibnizovy formule*:

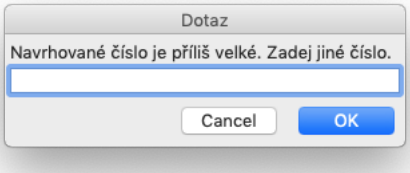
$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

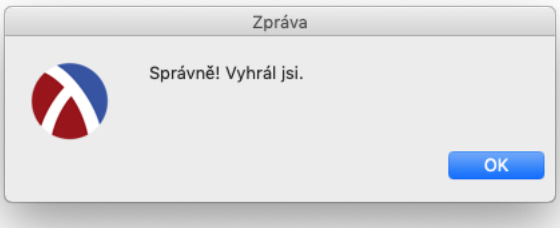
   Dosažená přesnost odhadu čísla $\frac{\pi}{4}$ je přitom dána posledním přičítaným (odečítaným) zlomkem. Napište proceduru `leibniz`, která vypočítá číslo `pi` se zadanou přesností. Napište jak obyčejnou, tak iterativní verzi této procedury.

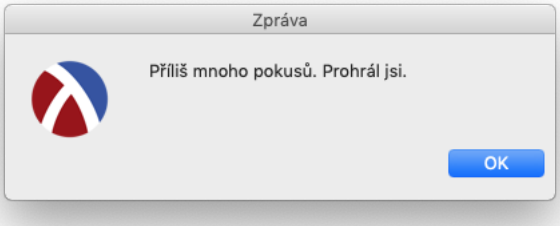
9. Napište jednoduchou hru na hádání čísel. Počítač náhodně vygeneruje číslo od 1 do 100 a úkolem hráče je číslo co nejrychleji uhodnout. Píše programu své tipy a dostává odpovědi čtyř typů: úspěch, navrhované číslo je příliš malé, navrhované číslo je příliš velké, příliš mnoho pokusů. Program se spustí vyhodnocením `(start-game)` a může s uživatelem komunikovat například takovými dialogy:











Abyste mohli s dialogy pracovat, musíte použít knihovnu. Tu načtete tím, že na začátek vašeho souboru napíšete `(require racket/gui/base)`. Dialog s dotazem vyvoláte procedurou `get-text-from-user`. Například toto:

```
> (get-text-from-user "Dotaz" "Zadej číslo")
```

otevře dialog a jako výsledek vrátí (ve formě textu) číslo napsané uživatelem. Na převod textu na číslo použijete proceduru `string->number`, takže k získání čísla od uživatele můžete použít

```
> (string->number (get-text-from-user "Dotaz" "Zadej číslo"))
```

K zobrazení dialogového okna na závěr použijte proceduru `message-box`:

```
> (message-box "Zpráva" "Příliš mnoho pokusů. Prohrál jsi.")
```

K vygenerování náhodného čísla od 1 do 100 použijte

```
> (+ (random 100) 1)
```

(samotný výraz `(random 100)` vrací náhodné číslo od 0 do 99).