KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Paradigmata programování 1

3 Rekurzivní funkce

Michal Krupka

Obsah

Příklad: Vzdálenost bodů v rovině

Příklad: Vzdálenost bodů v rovině

```
(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))))
```

Příklad: Vzdálenost bodů v rovině

```
(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))))
```

```
(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: procentuální podíl

Příklad: procentuální podíl

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

Příklad: procentuální podíl

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

Použití:

Příklad: procentuální podíl

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

Použití:

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

Příklad: procentuální podíl

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

Použití:

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

S defaultní hodnotou:

Příklad: procentuální podíl

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

Použití:

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

S defaultní hodnotou:

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

Prostředí

Prostředí

Prostředí během aplikace `(percentage-4 100378 t)`:

Prostředí

Prostředí během aplikace `(percentage-4 100378 t)`:

Globální prostředí

symbol	hodnota
pi	3.141592653589793D0
⋮	⋮

Prostředí funkce `percentage-2`

symbol	hodnota
part	102293
whole	t

Prostředí operátoru `let`

symbol	hodnota
whole	10900555

Příklad: procentuální podíl

Příklad: procentuální podíl

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

Příklad: procentuální podíl

```
(defun percentage-4 (part whole)
```

Příklad: procentuální podíl

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

Příklad: procentuální podíl

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

Příklad: procentuální podíl

```
(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
```

Příklad: procentuální podíl

```
(defun percentage-4 (part whole)
  (if (eql whole t)
      (percentage-4 part 10900555)
    (* (/ part whole) 100.0)))
```

rekurzivní aplikace

Bude to fungovat?

Bude to fungovat?

Prostředí během aplikace `(percentage-4 102293 t)`:

Bude to fungovat?

Prostředí během aplikace `(percentage-4 102293 t)`:

Globální prostředí

Bude to fungovat?

Prostředí během aplikace `(percentage-4 102293 t)`:

Globální prostředí

Prostředí funkce `percentage-4`
(první aplikace)

symbol	hodnota
`part`	`102293`
`whole`	`t`

Bude to fungovat?

Prostředí během aplikace `(percentage-4 102293 t)`:

Prostředí funkce `percentage-4` (první aplikace)

symbol	hodnota
part	102293
whole	t

Prostředí funkce `percentage-4` (druhá aplikace)

symbol	hodnota
part	102293
whole	10900555

Příklad: prohledávání intervalu

Příklad: prohledávání intervalu

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

Příklad: prohledávání intervalu

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

```
(defun contains-square-p (a b)
```

Příklad: prohledávání intervalu

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

```
(defun contains-square-p (a b)
  (if (> a b)                                ;když je interval prázdný,
```

Příklad: prohledávání intervalu

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

```
(defun contains-square-p (a b)
  (if (> a b)                          ;když je interval prázdný,
      nil                              ;čtverec neobsahuje
```

Příklad: prohledávání intervalu

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

```
(defun contains-square-p (a b)
  (if (> a b)                          ;když je interval prázdný,
      nil                              ;čtverec neobsahuje
    (if (squarep a)                    ;je-li dolní konec čtverec,
```

Příklad: prohledávání intervalu

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

```
(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
```

Příklad: prohledávání intervalu

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

```
(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]
```

Příklad: prohledávání intervalu

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

```
(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]
```

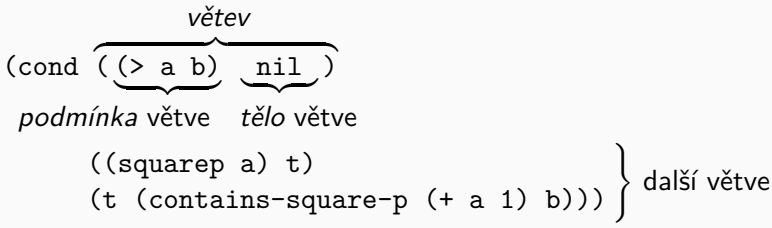
rekurzivní aplikace

Příklad: prohledávání intervalu (druhá verze)

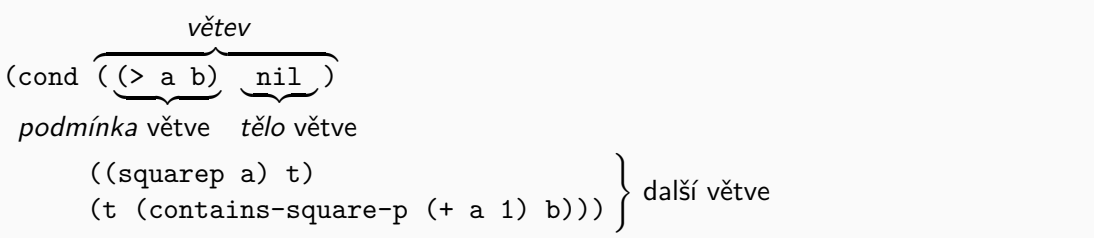
Příklad: prohledávání intervalu (druhá verze)

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

Odbočka: speciální operátor `cond`



Odbočka: speciální operátor `cond`



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`.

Prohledávání intervalu: verze s logickými spojkami

Prohledávání intervalu: verze s logickými spojkami

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

Prohledávání intervalu: verze s logickými spojkami

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

Je napsána deklarativně: neřešili jsme, *jak* má být výsledek vypočítán, ale *co* má výsledkem být:

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

Zobecněné logické hodnoty, speciální operátory `and` a `or`

Zobecněné logické hodnoty, speciální operátory `and` a `or`

Operátor `and`

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

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

Zobecněné logické hodnoty, speciální operátory `and` a `or`

Operátor `and`

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

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

Operátor `or`

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

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

Zobecněné logické hodnoty, speciální operátory `and` a `or`

Operátor `and`

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

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

Operátor `or`

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

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

Funkce `not`

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

Faktoriál

Faktoriál

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

Faktoriál

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

Napsáno do funkce:

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

Obecný způsob řešení problému rekurzivním výpočtem

Obecný způsob řešení problému rekurzivním výpočtem

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

Obsah

Rekurzivní funkce

Rekurzivní funkce

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

Rekurzivní funkce

Rekurzivní funkce

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

- je poznat ze zdrojového kódu funkce

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

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:

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.

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

Obsah

Obecná mocnina

Obecná mocnina

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

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

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

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

Obecná mocnina

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

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

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

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

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

Opět: bude to fungovat?

Prostředí při aplikaci `(power 10 4)`

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

Opět: bude to fungovat?

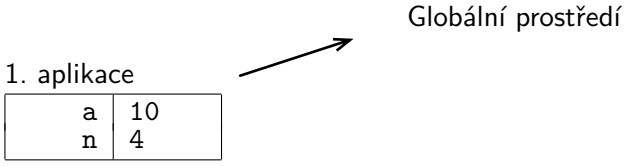
Prostředí při aplikaci `(power 10 4)`

Globální prostředí

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

Opět: bude to fungovat?

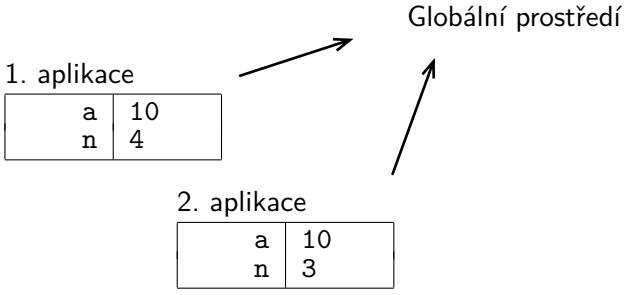
Prostředí při aplikaci `(power 10 4)`




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

Opět: bude to fungovat?

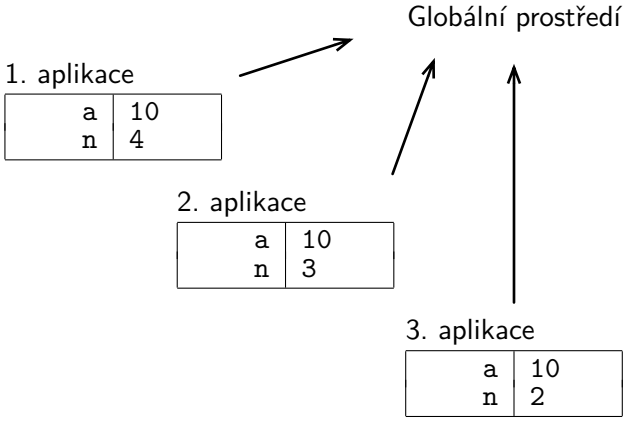
Prostředí při aplikaci `(power 10 4)`




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

Opět: bude to fungovat?

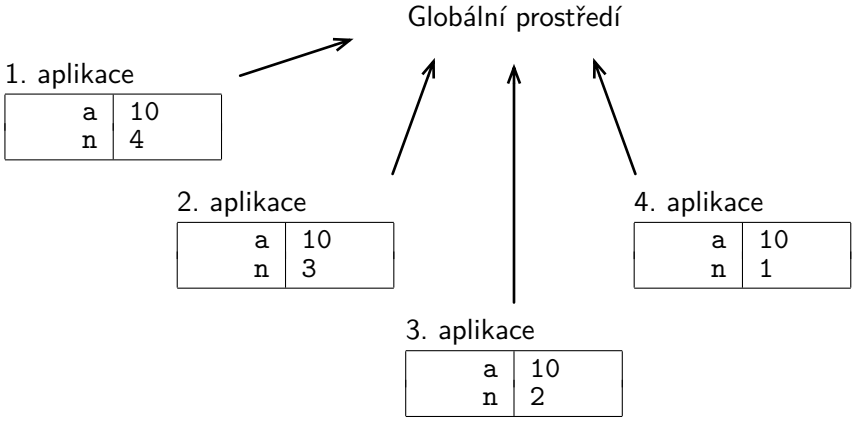
Prostředí při aplikaci `(power 10 4)`




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

Opět: bude to fungovat?

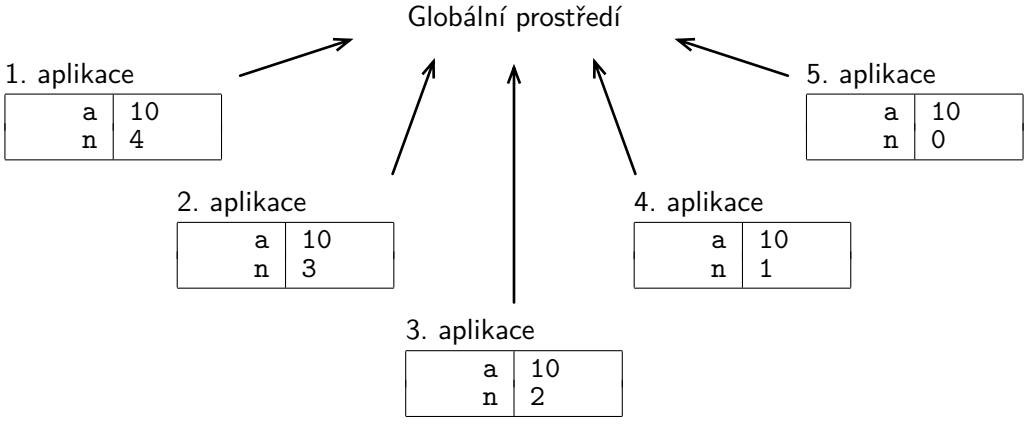
Prostředí při aplikaci `(power 10 4)`




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

Opět: bude to fungovat?

Prostředí při aplikaci `(power 10 4)`




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

Pevný bod funkce cos

Pevný bod funkce cos

Hledáme přibližnou hodnotu čísla x takového, že $\cos x = x$.

Pevný bod funkce cos

Hledáme přibližnou hodnotu čísla x takového, že $\cos x = x$.

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

Pevný bod funkce cos

Hledáme přibližnou hodnotu čísla x takového, že $\cos x = x$.

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

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

Pevný bod funkce cos

Hledáme přibližnou hodnotu čísla x takového, že $\cos x = x$.

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

(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))
```

Faktoriál — iterativní verze

Faktoriál — iterativní verze

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

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

Fibonacciho posloupnost

Fibonacciho posloupnost

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

Fibonacciho posloupnost

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

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

Fibonacciho posloupnost

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

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

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