Funkcionální programování

SSPŠ
DVOP4 PIT
2025 / 2026
Ing. Matěj Cajthaml

Funkcionální programování

- Funkce jsou základním stavebním kamenem
- Neexistují proměnné, třídy, objekty
- Funkce nemají vedlejší efekty - nemění stav mimofunkčních věcí
- Funkce se mohou předávat jako parametry
- Funkce mohou vracet funkce

Důvod existence

- Vede k čistšímu kódu
- Přehlednost
- Jednoduchost:
- Máme jen funkce
- A ty jde volat

- Jednoduchá možnost snadné paralelizace

LISP

- LISt processing
- 1958 (C/CPP až okolo 1980)
- John McCarthy
- První funkcionální jazyk
- Přišel se spousty věcmi, které používá skoro každý jazyk:
- Makra
- Garbage collection (automatická správa paměti)
- Podmínky typu if-else
- Revoluční

Racket

- LISP má nespočet dialektů
- Jeden nejznámější je Racket, který budeme používat
- Další např. Clojure, Scheme, ...
- Lehce mění jazyk, přináší standard a další

OTÁZKA

Jaký SW používá funkcionální programování?

Moderní funkcionální jazyky

- Haskell, Erlang, F#
- Pořád jsou středem funkce
- Stejně jako jinde, existují paradigmata
- Jeden jazyk jich podporuje více
- Viz JS, C#

Použití Racketu

- IDE DrRacket
- Určení dialektu: #lang racket
- racket spustí překlad a intepretaci

Funkce

- Klíčové slovo define
- Následuje:
- jméno funkce
- seznam parametrů
- tělo funkce

Definice funkce

```
1 #lang racket
2
3 ; (define (jmenoFunkce parametry...)
4 ;   teloFunkce)
5
6 (define (add x y)
7   (+ x y))
```

Vyhodnocení

- Při volání funkce se vyhodnotí všechny parametry
- Výsledky se přiřadí do proměnných
- Vyhodnotí se tělo funkce
- Výsledek se vrátí do volající funkce

- Volání funkce je **prefixové**

- Prefixové: + 1 2
- Sufixové: 1 2 +
- Infixové: 1 + 2

- Obsahuje základní operátory, + - / * floor ceiling

Práce / výpočet

- Vyhodnotťe:

1. (/ 1 2)
2. (floor (/ 1 2))
3. (+ 1 (* 2 3))
4. (+ 1 (* 2 (- 3 4)))
5. (+ (ceiling (/ 1 2)) (* 2 (3 (- 4 5))))

Booleanovy hodnoty

- #t
- #f

- (not x)
- (and x y)
- (or x y)

- (= x y)
- (< x y)
- (> x y)
- (<= x y)
- (>= x y)

Další funkce

- (print x)
- (newline)
- A další později

Práce / vyhodnocení pravdy

- Vyhodnotťe:

1. (not #t)
2. (and #t #f)
3. (or (not #t) #f)
4. (= 1 (+ 1 (* 2 3)))
5. (< 1 (+ 1 (* 2 3)))
6. (> 1 (* (+ 2 (- 3 2)) (* 2 3)))

Zavolání funkce

```
1 #lang racket
2
3 (define (add x y)
4   (+ x y))
5
6 (add 1 2)
7
8 (add (add 1 2) 3)
```

Podmínky

- (if test then else)
- test se musí vyhodnotí na boolean hodnotu

- Pokud je #t, vyhodnotí se then
- Pokud je #f, vyhodnotí se else

Podmínka

```
1 #lang racket
2
3 (define (abs x)
4   (if (< x 0)
5       (- x)
6       x))
```

Rekurze

- Neexistuje iterace
- Vše se řeší pomocí rekurzivních volání funkcí
- Existuje více typů rekurze, viz později
- Uvnitř sebe voláme stejnou funkci
- Ukončující podmínka

Podmínka

```
1 #lang racket
2
3 (define (factorial n)
4   (if (= n 0)
5       1
6       (* n (factorial (- n 1)))))
```

Práce / funkce

- Vytvořte následující funkce, otestujte je:

1. (inc x) — zvýší číslo o 1
2. (dec x) — sníží číslo o 1
3. (zero? x) — zjistí, zda je číslo 0
4. (add x y) — sečte dvě čísla pomocí funkcí výše
5. (mul x y) — vynásobí dvě čísla pomocí funkce add
6. (exp x n) — mocnina čísla x s exponentem n
7. (fib n) — vypočítá n-tý prvek Fibonacciho posloupnosti

Listy

- Vše je funkce? Ne. Vše je list.

- List je uzavřen v kulatých závorkách
- (1 2 3 4 5)
- List může obsahovat libovolný počet prvků
- Prvky jsou odděleny mezerou
- Reprezentováno buňkami

- Buňka má hodnotu a ukazuje na další buňku
- (a . b)
- První je hlava, druhý je "zbytek"
- Zbytek může být empty
- (1 2 3) = (1 . (2 . (3 . empty)))

- Takto to ale v programu nemůžeme zapsat

Escape/quoting

- Kulaté závorky se používají i na volání funkcí
- ' = nevyhodnotí se jako volání funkce
- ` = nevyhodnotí se, ale nějaký se může vyhodnotit pomocí ,
- '(1 2 3)
- `(1 2 ,x)
- V buňce může být seznam
- Můžeme vložit "rozložením" pomocí ,@

Zápis listů

```
1 #lang racket
2
3 (sum '(1 2 3 4 5 6 7 8 9 10))
```

Zápis listů pokročile

```
1 #lang racket
2
3 (define numbers '(5 6))
4
5 `(1 2 ,numbers)
6 ;; (1 2 (5 6))
7
8 `(1 2 ,@numbers)
9 ;; (1 2 5 6)
```

Práce s listy

- (cons a b) — vytvoří buňku (a . b)
- (car x) — vrátí hlavu buňky
- (cdr x) — vrátí zbytek buňky
- (list a b c) — vytvoří list (a b c)
- (null? x) — zjistí, zda je list prázdný

OTÁZKA

Co dělá funkce funkce cddddr, cdaddr a caddar?

Porovnání

- (equal? x y)
- Vrací #t pokud jsou x a y stejné
- Ale i strukturálně (pro listy)
- Jinak #f

- Pozor na (eq? x y)

Práce s listy

```
1  #lang racket
2
3  (define (sum list)
4    (if (null? list)
5        0
6        (+ (car list) (sum (cdr list)))))
7
8  (sum '(1 2 3 4 5 6 7 8 9 10))
9
10 (define (second list)
11   (car (cdr list)))
12
13 (second '(1 2 3 4 5 6 7 8 9 10))
14 (second '(1))
15
```

Práce

- Napište si vlastní funkci:
- (length list)
- Vrátí délku listu

Práce

- Napište si vlastní funkci:
- (nth n list)

- Vrátí prvek na n-té pozici (od 1)
- Pokud neexistuje, tak empty

Práce

- Napište si vlastní funkci:
- (min list)
- (max list)

- Vrátí nejmenší nebo největší prvek listu
- Pokud je prázdný?

Práce

- Napište si vlastní funkci:
- (contains? list x)

- Vrátí #t, pokud list obsahuje hodnotu x
- Pokud je prázdný?

Práce

- Napište si vlastní funkci:
- (range a b)

- Vrátí nový list, který má hodnoty od a do b
- Co když bude b < a?

Práce

- Napište si vlastní funkci:
- (reverse list)

- Prohodí prvky listu
- list = (1 2 3 4 5)
- (reverse list) = (5 4 3 2 1)

Práce

- Napište si vlastní funkci:
- (remove list x)

- Odstraní všechny výskyty prvku x z listu
- list = (1 2 5 4 5)
- (remove list 5) = (1 2 4)

Práce

- Napište si vlastní funkci:
- (prime? x)
- Zjistí (vrátí #t), zda je x je prvočíslo

Optimalizace výpočtu

- Někdy porovnáváme nějakou hodnotu vícekrát
- Pokud hodnotu získáváme voláním funkce
- Volá se vícekrát

- Máme možnost si předem vypočítat hodnoty
- Ty se uloží do určité konstanty jména a opakovaně používáme

- (let (variables) body)
- Vyhodnotí a vrátí body, do které se vloží variables
- Variables je list plný dvojic (name value)

Let

```
1 #lang racket
2
3 (define (f x)
4   (let ([y (+ x 1)])
5     (+ (* x y) y)))
```

Podmíněné vyhodnocení

- Vylepšený if
- Co když je hodnota a, b, x, y nebo z?
- Moc vnoření, nepřehledné
- (cond (test1 body1) (test2 body2) ...)
- testy jsou vyhodnoceny postupně, dokud není nalezeno první pravdivé
- pokud není nalezeno žádné pravdivé, je vyhodnocen test s hodnotou else

Cond

```
1 #lang racket
2
3 (define (fib n)
4     (cond
5         [(= n 0) 0]
6         [(= n 1) 1]
7         [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

Volání funkcí v jazycích

- Funkce - vstupní parametry, vnitřní stav, "kód"
- Při zavolání funkce uvnitř funkce se musí tento stav uložit
- Protože zavolaná funkce někdy skončí
- Tzv. stack - zásobník, který tyto data ukládá
- Omezené místo, někdy může dojít
- Stackoverflow, např. rekurzí bez ukončovací podmínky
- Ale i tím, že je problém moc velký
- Např. cyklus ale stacky nutně netvoří
- Rekurze svým opakovaným voláním může přehltit stack
- Existuje typ rekurze, který tento problém nemá
- End-tail recursion, koncová rekurze

Koncová rekurze

- Interně se netvoří při zavolání funkce nový rámec, ale ten starý se přepíše tím novým
- Záleží na jazyku, např. Python "neumí"
- Stack tedy zůstává stejně velký
- Při navrácení se vrátí hodnota z posledního volání rekurze

- Často využíváme akumulátory
- Máme parametr, do které pořád vkládáme hodnoty
- A tento parametr se poté vrátí

Rekurze

```
1  #lang racket
2
3  ; recursion normal
4  (define (factorial n)
5    (if (= n 0)
6        1
7        (* n (factorial (- n 1)))))
8
9  (factorial 5)
10
11 ; recursion tail recursive
12 (define (factorialTail n)
13   (factorialTailInner n 1))
14
15 (define (factorialTailInner n acc)
```

Práce

- Napište si vlastní funkci:
- (last list)

- Vrátí poslední prvek
- Použijte koncovou rekurzi

Práce

- Napište si vlastní funkci:
- (odd-list list)

- Vrátí pouze lichá čísla ze seznamu
- Použijte koncovou rekurzi

Práce

- Napište si vlastní funkci:
- (palindrom? list)

- Vrátí #t, pokud se čte obsah stejně ze začátku i z konce
- (palindrom '(1 2 3)) => #f
- (palindrom '(1 2 3 2 1)) => #t
- (palindrom '(1 2 3 3 2 1)) => #t
- (palindrom '(1)) => #t

Slovník

- Vytvořte si vlastní strukturu - slovník
- Slovník obsahuje páru klíče a hodnoty
- Pro slovník vytvořte vlastní operace
- (newDictionary)
- (addKey dict key val)
- (find dict key)
- (del dict key)
- Bonusově: zamezení duplicitních klíčů