Funkcionální programování

Programovací paradigma založené na matematických funkcích

Autoři: Bc.Karim Alakbarov, Bc.Yerkhan Bazylbekov

Předmět: 18OOP

Fakulta:Fakulta jaderná a fyzikálně inženýrská

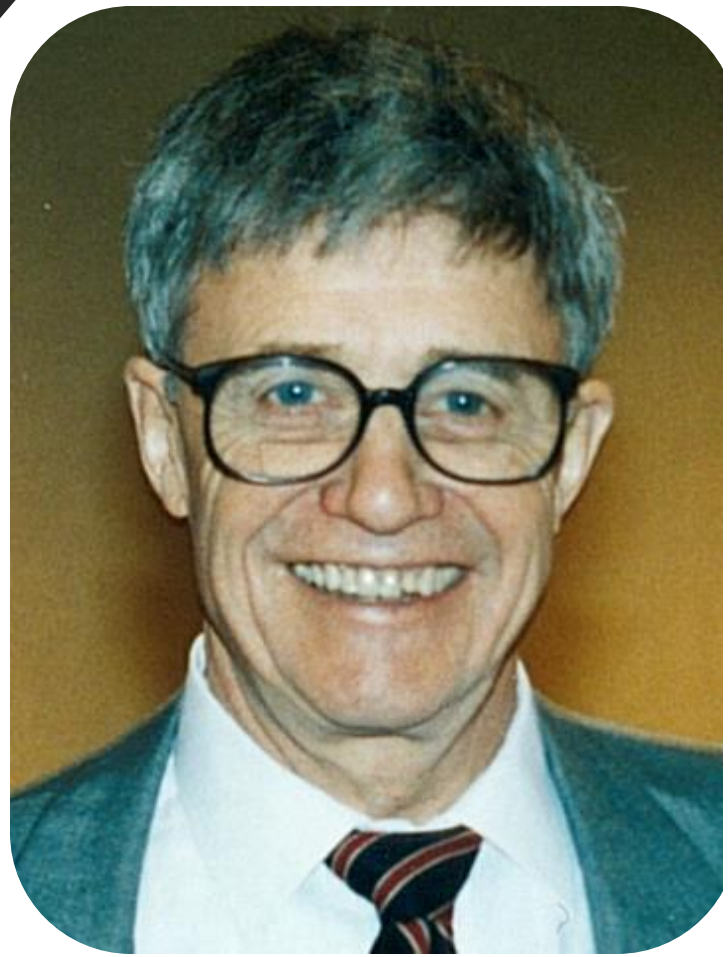
Co je to funkcionální programování?

- **Funkcionální programování** je deklarativní programovací paradigma, které chápe výpočet jako vyhodnocení matematických funkcí. Funkcionální programování má své kořeny v lambda-kalkulu, formálním systému vyvinutém v 30. letech k vyšetřování definicí funkcí, jejich aplikace a rekurze. Mnoho funkcionálních programovacích jazyků může být považováno za rozšíření lambda kalkulu.
- Výpočtem funkcionálního programu je tedy posloupnost vzájemně ekvivalentních výrazů, které se postupně zjednodušují. Výsledkem výpočtu, pokud se k němu podaří dospět, je výraz v dále nezjednodušitelné *normální formě*. Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce.

- Jeden z prvních jazyků, který v sobě zahrnoval funkcionální část, byl LISP, vytvořený Johnem McCarthym pro IBM série 700/7000 vědeckých počítačů na konci 50. let. LISP představil mnoho vlastností, které můžeme najít v nynějších funkcionálních jazycích, ačkoliv LISP je technicky multi-paradigmatický jazyk. Scheme a Dylan byly pozdější pokusy zjednodušit a vylepšit LISP.

- Informační procesní jazyk IPL je někdy uváděn jako první počítačový funkcionální jazyk. Je to jazyk pro manipulaci se seznamem znaků. Má svůj generátor funkcí, který se stará o to, aby funkce mohla přijmout funkci jako argument a vzhledem k tomu, že je to assembly-level jazyk, kód může být použit jako data, takže IPL může být považován za higher-order funkční. Každopádně hodně závisí na měnící se struktuře seznamu a podobných přímých vlastnostech.

- Kenneth E. Iverson vyvinul programovací jazyk APL na začátku 60. let a popsal ho roku 1962 ve své knize „A programming Language“. APL měl hlavní vliv na programovací jazyk FP Johna Backuse. Na začátku 90. let, vytvořili Iverson a Roger Hui nástupce APL, J programming. Uprostřed let 90. Artur Whitney, který pracoval s Iversonem, vytvořil programovací jazyk K, který se používá v komerčním a finančním průmyslu.



Použití

- Funkcionální programovací jazyky, především čistě funkcionální, se používají spíše v akademickém než komerčním prostředí. Přesto široké spektrum organizací využívá některé funkcionální programovací jazyky jako např. Erlang, R (statistika), Mathematica (symbolická matematika), Haskell, ML, J a K (finanční analýza) a doménově specifické programovací jazyky jako XQuery/XSLT (XML). Dále jsou funkcionální programovací jazyky důležité pro některá odvětví informatiky, například zabývající se umělou inteligencí, formální specifikací, modelováním nebo rychlým prototypováním.

Srovnání s imperativním programováním

Imperativní programování	Funkcionální programování
Popisuje, **jak** něco dělat	Popisuje, **co** dělat
Používá měnitelné stavy	Používá neměnitelné údaje
Převážně cykly	Převážně rekurze
Příklad: C, Java	Příklad: Haskell, Lisp

Základní principy FP

Čisté funkce

- Funkce vrací vždy stejnou hodnotu se stejnými vstupními daty.
- Neexistují žádné vedlejší efekty (změna externích dat).

Imutabilita

- Data se po vytvoření nemění, což usnadňuje přehlednost programu.

Funkce vyššího řádu

- Funkcím lze předávat argumenty, vracet je z jiných funkcí.

Líné vyhodnocování

- Výpočty se provádějí pouze v případě potřeby.

Čistě funkce

Čisté funkce (nebo výrazy) nemají žádné vedlejší účinky (paměťové nebo vstupně-výstupní). To znamená, že čisté funkce mají několik užitečných vlastností, z nichž mnohé lze využít k optimalizaci kódu:

ČF mají tyto vlastnosti:

- pokud se výsledek ČF nepoužívá, lze její volání smazat, aniž by to poškodilo ostatní výrazy;
- výsledek volání ČF může být memoizován (uložen v tabulce hodnot spolu s argumenty volání) a při dalším volání převzat bez výpočtů;
- pokud mezi dvěma ČF není datová závislost, lze změnit pořadí jejich výpočtu nebo je paralelizovat.
- Pokud celý jazyk nepovoluje vedlejší efekty, pak lze použít libovolnou vyhodnocovací strategii; to dává překladači volnost měnit pořadí nebo kombinovat vyhodnocování výrazů v programu (například pomocí odlesňování).

Funkce vyššího řádu

Funkce jsou higher-order-function, česky *funkce vyššího řádu*, v případě, kdy mohou převzít nějakou funkci jako argument nebo navrátit funkci jako výsledek. (Derivace a neurčitý integrál jsou toho příkladem v matematice).

V následující ukázce je naznačena funkce vyššího řádu, protože jí je jako parametr předávána funkce zpětného volání.

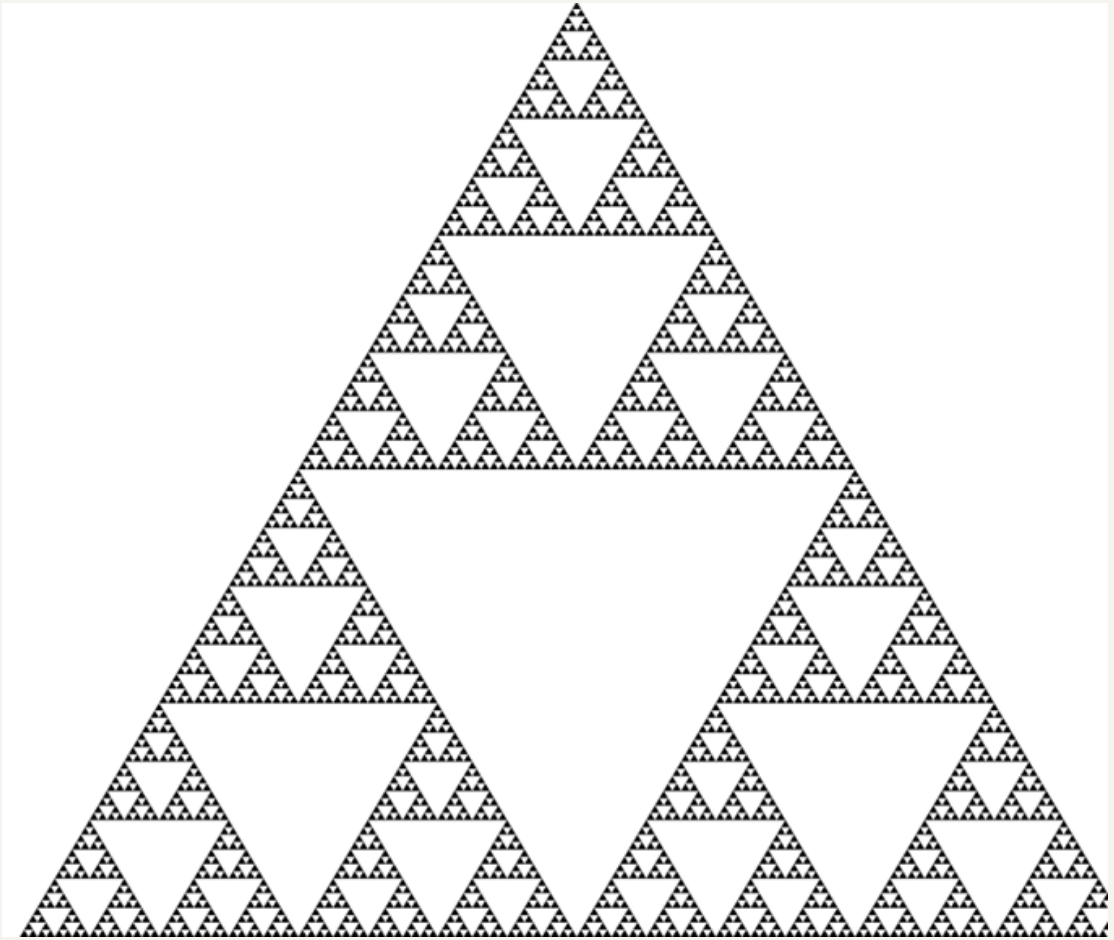
```
// Callback metoda, která se předává jako parametr do volané metody na řádku 12
function callbackFunction(){
    console.log('Já jsem zavolaná callBack metoda');
}

// High order function
function higherOrderFunction(calledFunction){
    console.log('Já jsem high order metoda')
    calledFunction()
}

higherOrderFunction(callbackFunction);
```

Rekurze

- Rekurze - definice, popis objektu nebo procesu v rámci tohoto objektu nebo procesu samotného, tj. situace, kdy je objekt součástí sebe sama.
- Funkční jazyky používají rekurzi místo cyklu;
- rekurzivní funkce volají samy sebe, což umožňuje provádět operaci znovu a znovu;
- rekurzivní funkce lze zobecnit na funkce vyššího řádu pomocí katamorfismu a anamorfismu.



Sierpińského trojúhelník je fraktální útvar vytvořený rekurzivním vykreslováním rovnostranných trojúhelníků. Jmenuje se tak podle Wacława Sierpińského, polského matematika, který ho v roce 1915 poprvé popsal.

Výhody a nevýhody rekurze

Výhody :

Zjednodušuje kód při práci s úlohami, které lze rozdělit na podúkoly (např. výpočet faktoriálu, průchod stromem, vyhledávací algoritmy).

Nevýhody:

Může spotřebovávat více paměti kvůli hlubokým voláním funkcí (zásobník volání).

Není vždy zřejmé, jak optimalizovat výkon.

Striktní, nestriktní a líné vyhodnocení

Striktní vyhodnocení:

- Argumenty funkcí se **vyhodnocují okamžitě** při volání funkce.
- Pokud je argument potřebný nebo není, vždy se vyhodnotí.
- Vede k jednodušší kontrole chyb, protože výrazy se vždy vyhodnocují hned.

Příklad:

```
def add(x, y):
    return x + y

result = add(3 + 2, 5 * 2)  # Než se funkce zavolá, oba výrazy se vyhodnotí.
```

Nestriktní vyhodnocení

- Argumenty funkcí se **nevyhodnocují okamžitě**, pokud nejsou potřeba.
- **Nepodmíněné větve** mohou být vynechány, pokud nejsou potřeba.
- Často používá rekurzivní struktury a abstrakci dat.

Příklad nestriktního vyhodnocení v Haskell:

```
-- Funkce, která vrací první prvek dvojice a ignoruje druhý
first :: (a, b) -> a
first (x, _) = x


-- Příklad volání funkce s odloženým vyhodnocením
result = first (5, undefined)
```

Vysvětlení:

- Funkce first vrací první prvek dvojice a ignoruje druhý. V tomto příkladu je druhý prvek undefined, což by obvykle způsobilo chybu, pokud by byl vyhodnocen.
- Nicméně v Haskellu, jelikož druhý prvek není použit, nikdy nedojde k jeho vyhodnocení a program se úspěšně ukončí a vrátí hodnotu 5.

Líné vyhodnocení (Laziness):

- Variantou nestriktního vyhodnocení.
- **Výpočty se provádějí pouze tehdy**, když jsou výsledky opravdu potřeba.
- Výraz je vyhodnocen **co nejpozději** (v tzv. „call by need“ stylu).
- Pomáhá optimalizovat výkonnost programu, zvláště pokud existují nevyužité výrazy.

Příklad:

take 3 [1..] -- Vrátí prvních 3 prvky nekonečného seznamu [1, 2, 3].

Výhody líného vyhodnocení:

- **Úspora paměti a výpočetního času**, protože se počítají jen ty výrazy, které jsou skutečně potřeba.
- Umožňuje práci s **nekonečnými datovými strukturami**.

Nevýhody:

- Může být složitější na debugování a sledování průběhu výpočtů.
- Možné nebezpečí paměťových úniků („space leaks“).

Haskell

Čistě funkcionální jazyk

Haskell je čistě funkcionální jazyk navržený tak, aby eliminoval vedlejší efekty a zajistil, že programy jsou předvídatelné a snadno testovatelné. Je známý svou přísnou statickou typizací a líným vyhodnocováním (lazy evaluation).

- **Klíčové funkce**:

  Čisté funkce: Každá funkce je čistá, což zajišťuje, že je její chování konzistentní a předvídatelné.

  Líné vyhodnocování (Lazy evaluation): Výpočty se provádějí pouze tehdy, když je výsledek potřebný, což zlepšuje efektivitu a paměťové nároky.

  Typový systém: Haskell využívá velmi silný a flexibilní typový systém, který umožňuje bezpečné a výkonné programování.

Priklad: Funkce, která vezme seznam a vrátí nový seznam s každým prvkem zvýšeným o 1

```
add :: Int -> Int -> Int
add x y = x + y
incrementList :: [Int] -> [Int]
incrementList = map (+1)
main = print (incrementList [1, 2, 3, 4])
```

Výsledek: `[2,3,4,5]`

Scala

Hybridní objektově-orientovaný a funkcionální styl

Scala kombinuje objektově-orientovaný přístup s funkcionálním programováním, což jí dává výhodu flexibility. Scala je často používána pro budování velkých, škálovatelných systémů díky své jednoduchosti a robustním abstrakcím.

- **Klíčové funkce**:

  Funkce jako objekty: Funkce jsou v jazyce Scala první třídy, což znamená, že mohou být přiřazeny k proměnným, předávány jako argumenty a vráceny jako výstupy.

  Neměnitelné kolekce: Scala upřednostňuje neměnitelné datové struktury, což zjednodušuje správu stavu a zamezuje chybám.

  Srovnávání vzorů (Pattern matching): Scala nabízí výkonné nástroje pro dekompozici datových struktur.

Priklad: Funkce, která vezme seznam a vrátí nový seznam s každým prvkem zvýšeným o 1

```
val add = (x: Int, y: Int) => x + y
val numbers = List(1, 2, 3, 4)
val incremented = numbers.map(_ + 1)
println(incremented)
```

Výsledek: `[2,3,4,5]`

Clojure

Moderní dialekt Lispu

Clojure je moderní funkcionální jazyk založený na Lispu, který běží na JVM (Java Virtual Machine). Nabízí výkonné abstrakce pro práci s neměnitelnými datovými strukturami a podporuje paralelní zpracování.

Klíčové funkce:

- **Neměnitelné datové struktury**: Všechny datové struktury jsou neměnitelné, což zjednodušuje paralelní a souběžné programování.
- **Interoperabilita s Javou**: Clojure umožňuje používat knihovny z jazyka Java, což z něj činí flexibilní jazyk pro vývoj robustních aplikací.

Rekurze a Tail-Call Optimalizace: *Rekurze je hlavním nástrojem pro iteraci, a jazyk nabízí optimalizaci pro efektivní rekurzivní volání.*

Priklad: Funkce, která vezme seznam a vrátí nový seznam s každým prvkem zvýšeným o 1

```
(defn add [x y] (+ x y))
(def numbers [1, 2, 3, 4])
(def incremented (map inc numbers))
(println incremented)
```

Výsledek: `[2,3,4,5]`

JavaScript a Python: Podpora funkcionálního stylu

JavaScript:

•JavaScript je primárně objektově orientovaný jazyk, ale podporuje i funkcionální programování prostřednictvím funkcí jako `map()`, `filter()`, a `reduce()`.

Klíčové funkce: Funkce první třídy, vyššího řádu, a anonymní (lambda) funkce.

```
const numbers = [1, 2, 3, 4];
const incremented = numbers.map(x => x + 1);
console.log(incremented);
```

Výsledek: `[2,3,4,5]`

Python:

- Python podporuje funkcionální programování přes vestavěné funkce jako `map()`, `filter()`, `reduce()`, a lambda funkce.

```
from functools import reduce
numbers = [1, 2, 3, 4]
incremented = list(map(lambda x: x + 1, numbers))
print(incremented)
```

Výsledek: `[2,3,4,5]`

Funkce a nástroje pro funkcionální programování

Map, Filter, Reduce:

- **Map**: Aplikuje funkci na každý prvek seznamu *a vrátí nový seznam*.
- ***Filter***: *Filtruje prvky seznamu na základě podmínky (predikátu).*
- ***Reduce***: *Snižuje seznam na jednu hodnotu pomocí funkce, která kombinuje dva prvky.*

Příklad v Pythonu:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

// Map: zvýšení každého čísla o 1

squares = list(map(lambda x: x**2, numbers))

// Filter: výběr sudých čísel

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

// Reduce: suma všech čísel

sum_numbers = reduce(lambda x, y: x + y, numbers)

print(squares, even_numbers, sum_numbers)
```

Výsledek: `[1, 4, 9, 16, 25] [2, 4] 15`

Lambda funkce:

Lambda funkce jsou anonymní funkce používané pro krátké operace, které nepotřebují plnou deklaraci.

Monády a funktory

Funktor:

Funktor je struktura, ke které lze aplikovat funkci. Typickým příkladem je seznam, na který lze použít `map`.

Příklad v Haskellu:

```
fmap (+1) [1, 2, 3]  -- výsledek [2, 3, 4]
```

Monáda:

Monáda umožňuje zřetězení výpočtů a správu vedlejších efektů. Příkladem je monáda `Maybe` v Haskellu, která umožňuje elegantní práci s výpočty, které mohou selhat.

Příklad v Haskellu (monáda Maybe)

```
Just 3 >>= (\x -> Just (x + 1)) -- výsledek Just 4
```

Příklady použití monád:

- **I/O operace**: Správa vedlejších efektů, jako je vstup a výstup.
- **Řešení chyb**: Monády jako `Maybe` nebo `Either` se používají pro práci s chybami.

Zaver

- **Výhody funkcionálního programování:**
- Zvýšená předvídatelnost kódu a snadnost testování
- Méně chyb díky čistým funkcím a neměnnosti dat
- Efektivní využití paměti a procesoru díky línému vyhodnocování
- **Aplikace:**
- Ideální pro paralelní výpočty, práci s velkými daty, analýzu a finanční výpočty
- Použití v oblastech jako umělá inteligence, zpracování dat a rychlý prototypový vývoj

děkujeme za pozornost