# Funkcionální programování, rekurze a její typy. Jazyk LISP.

**Co je to funkcionální programování**

**Funkcionální programovací** paradigma patří mezi tzv. deklarativní paradigmata. Slovo paradigma (z řeckého παραδειγμα – vzor) značí myšlenkový vzor, metodu, přístup, model uvažování.

**Klasický imperativní přístup**se zaměřuje na to, jak problém řešit. Popis řešení je rozepsán do kroků probíhajících v čase.

Naproti tomu **deklarativní přístup** klade důraz na to, co je řešením daného problému. Deklarativní zápis programu představuje již samotné řešení problému zapsané v jistém tvaru. Výpočet podle tohoto programu je pak transformací takového popisu do tvaru jednoduššího, použitelnějšího.

Rozdíl mezi **imperativním** a **funkcionálním** přístupem k řešení problému lze ilustrovat třeba na úloze „Najít index největšího prvku v posloupnosti a1, a2, … an navzájem různých kladných celých čísel“. Imperativní program zavádí pomocné proměnné – paměťová místa i, j, max – a skládá se z kroků.

Polož max = 0. Pro j od 1 do n opakuj: jestliže aj > max, pak polož i = j a max = aj. Výsledkem je hodnota proměnné i.

Deklarativní popis řešení téže úlohy může vypadat takto:

Takové i z množiny {1, … n}, pro které platí: ∀ j z {1, … n}; ai≥aj

Funkcionální paradigma je založeno na zápisu programu ve tvaru výrazu. Nejdůležitějšími složkami těchto výrazů jsou funkce a jejich aplikace na argumenty. Výpočet funkcionálního programu spočívá ve zjednodušování výrazu až do doby, kdy výraz dále zjednodušit nelze. Tento dále nezjednodušitelný tvar je výsledkem výpočtu.

Například problém „Najít součin čísel 6 a 7“ je zapsán výrazem

1. 6 \* 7

Tento výraz je aplikací funkce násobení (\*) na argumenty 6 a 7. Výpočtem – zjednodušením výrazu – dostaneme hodnotu 42. Ta je dále nezjednodušitelná, tj. je odpovědí (řešením) problému.

**Funkcionální jazyky, překladače, interprety**

Matematickým vzorem, z něhož vychází většina funkcionálních jazyků, je tzv. lambda kalkul, který vznikl už počátkem 30. let minulého století.

Prvním skutečným funkcionálním programovacím jazykem s implementovaným překladačem byl jazyk **Lisp**, který měl velmi jednoduchou syntaxi, nebyl typovaný a nebyl vlastně ani čistě funkcionální – měl i řadu imperativních rysů. Z **Lispu** se později vyvinul jeho dialekt **Scheme**.

Koncem 70. let vznikl v Edinburghu jazyk **ML** s velmi silnou typovou disciplínou. Na jeho principech bylo navrženo několik dalších jazyků (Hope, Clean, Miranda …).

V 90. letech vznikl jazyk **Haskell**.

## Rekurze a její typy

V oblasti matematiky pojem rekurze chápeme jako definování objektu pomocí volání sebe sama. Využívá se například pro definici [přirozených čísel](http://cs.wikipedia.org/wiki/P%C5%99irozen%C3%A9_%C4%8D%C3%ADslo), [stromových struktur](http://cs.wikipedia.org/wiki/Strom_(datov%C3%A1_struktura)) a některých [funkcí](http://cs.wikipedia.org/wiki/Funkce_(matematika)).

V [programování](http://cs.wikipedia.org/wiki/Programov%C3%A1n%C3%AD) rekurze představuje opakované vnořené volání stejné [funkce](http://cs.wikipedia.org/wiki/Funkce_(programov%C3%A1n%C3%AD)) (podprogramu), v takovém případě se hovoří o [rekurzivní funkci](http://cs.wikipedia.org/wiki/Rekurzivn%C3%AD_funkce_(programov%C3%A1n%C3%AD)). Nedílnou součástí rekurzivní funkce musí být ukončující podmínka určující, kdy se má vnořování zastavit. Jelikož bývá nejčastějším zdrojem chyb, je třeba ji navrhnout dostatečně robustním způsobem a prověřit veškeré možné stavy.

Pro uplatnění rekurzivních algoritmů je zapotřebí, aby programovací jazyk umožňoval volání podprogramu ještě před ukončením jeho předchozího volání.

Po každém kroku volání sebe sama musí dojít ke zjednodušení problému. Pokud nenastane koncová situace, provede se rekurzivní krok.

Každý [algoritmus](http://cs.wikipedia.org/wiki/Algoritmus) využívající rekurzi lze přepsat do nerekurzivního tvaru při použití [zásobníku](http://cs.wikipedia.org/wiki/Z%C3%A1sobn%C3%ADk_(informatika)) nebo jiné paměťové struktury.

**Rekurzivní formule obsahuje:**

1. Omezující podmínku

- řeší triviální případ

- ukončuje rekurzi

2. Rekurzivní formuli

- redukuje daný problém na jeden či více jednodušších problémů

**Typy rekurze**

přímá

- fce volá přímo sama sebe

nepřímá

- rekurze probíhá přes prostředníka

**a(n):**

**vra**ť **n \* b(n)**

**b(n):**

**pokud n = 0, potom**

**vra**ť **1**

**jinak**

**vra**ť **a(n - 1)**

**Vno**ř**ená rekurze**

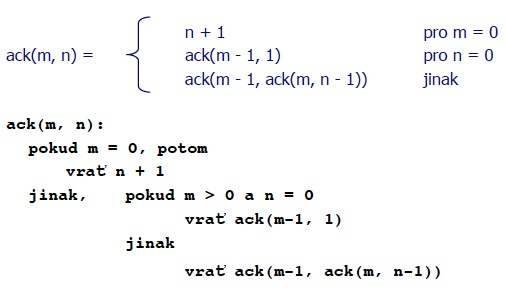
Tato rekurze používá jinou rekurzi jakoargument

• je typu f(x) = pokud(…)… jinak f(g(f(x)))

• tato rekurze má exponenciální časovou složitost a

je tedy nutno se jí vyhnout

• příklad: Ackermannova funkce



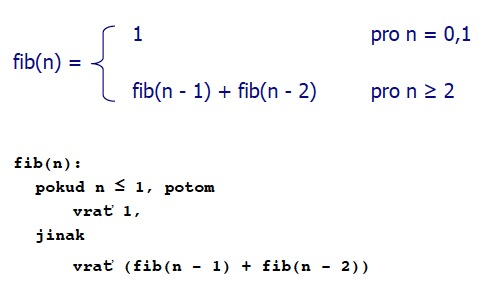
**Stromová rekurze**

V těle funkce je volána více jak jedna aktivace sebe sama

• pokud je funkce volána dvakrát při jednom průchodu, tak vytváří binární strom volání.

• pokud je funkce volána n-krát při jednom průchodu, tak vytváří n-nární strom volání

• příklad: Fibonacciho posloupnost

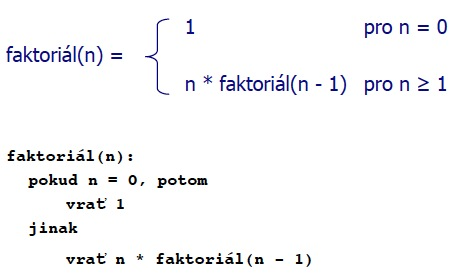


**Lineární rekurze**

V těle funkce je volána právě jedna aktivace sama sebe

• Pokud jsou operace v těle funkce složitosti O(1), bude mít takový algoritmus složitost O(2 n) = O(n), kde n je počet zanoření.

• Prostorová složitost je O(n) – je nutno uchovávat mezivýsledky



**Koncová rekurze**

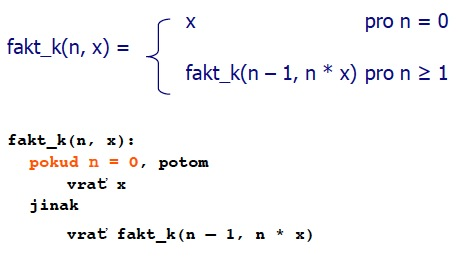
Koncově rekurzívní funkce mají jako rekurzívní formuli volání sama sebe

• výsledek již není při vynoření zpracováván

• výsledek vrácený nejhlubším zanořením je již konečný výsledek

• redukci je možno optimalizovat a vyhnout se vynořování

• Časová složitost bude O(n) a prostorová O(1)



**LISP** viz http://cs.wikipedia.org/wiki/Lisp