Funkcionální programování

- **Typované** - Haskell
- **Netypované** - Lisp, Scheme

λ-kalkul

- Teoretický základ funkcionálniho programování
- Lambda kalkul analyzuje funkce nikoli z hlediska původního matematického smyslu zobrazení z množiny do množiny, ale jako metodu výpočtu. Dá se chápat jako nejjednodušší univerzální programovací jazyk. Je univerzální, neboť libovolnou rekurzivně spočetnou funkci lze vyjádřit a vyčíslit pomocí tohoto formalismu, lambda kalkul je tedy výpočetní silou ekvivalentní Turingovu stroji. [z cs.wiki]

Co to vlastně je...

Funkcionální programování je založeno na přístupu delklarativního programování. To se vyznačuje tím, že programátor říká, co se má udělat a ne jak se to má udělat (čímž se vyznačuje imperativní přístup, popisuje se přesná posloupnost příkazů - tedy většina známých jazyků - Java, C atd.).

Jazyky patřící do skupiny funkcionální přistupují k programu jako k (matematickému) výrazu. Provádění programu spočívá v deterministickém vyhodnocování výrazů. Program tedy popisuje, co chceme vypočíst, a to, jak se výpočet bude provádět, není podstatné. Podobný přístup můžeme vidět např. u SQL dotazů (select * from tabulka - chceme vybrat data z tabulky a je nam "jedno" jak) nebo ještě lépe u tabulkových kalkulátorů (spreadsheet) - buňka obsahuje výraz, který popisuje její vztah k ostatním buňkám. Uživatele nezajímá, v jakém pořadí kalkulátor vyhodnotí obsahy jednotlivých buněk, ale potřebuje, aby byly dodrženy jednotlivé závislosti. Alokace paměti je také ponechána na kalkulátoru - tabulka je tvořena (teoreticky) nekonečnou rovinou buněk, ale místo se alokuje, až když je daná buňka potřebná. Shrnutím tedy je, že pro funkcionální jazyky je podstatné, co se má dělat, a nikoli, jakým způsobem se to provede.

„Funkcionální program“ nemá implicitní stav a stavové informace se udržují explicitně. Opakované provádění se nevyjadřuje iteracemi (cykly), ale rekurzí.

Trochu jinak...

Struktura čistě funkcionálního programu je definována výrazem, který zachycuje požadovaný cíl. Každý term daného výrazu je zase vyjádřením určité charakteristiky řešeného problému. Vyhodnocením každého z těchto termů nakonec dospějeme k řešení. Term neboli terminální podvýraz je v rámci zápisu výrazu jeho dále nedělitelná součást. Můžeme si zde představit volání pojmenované funkce.

Výpočtem funkcionálního programu je posloupnost vzájemně ekvivalentních výrazů, které se postupně zjednodušují. Výsledkem výpočtu je výraz v normální formě, tedy dále nezjednodušitelný. Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce.

- Program je složen s funkcí bez vedlejších efektů
- Funkce jsou brány jako běžné hodnoty („first-class values“)
- Funkcionální jazyky mají lepší mechanismus abstrakce
  - Možnost abstrahovat chování pomocí funkcí vyššího řádu
  - Program je budován z funkcí jejich kompozicí
  - Většinou mnohem stručnější programy
- Funkcionální programovací jazyky neobsahují přiřazení, příkazy, cykly atd.
  - Opakování je řešeno pomocí rekurze
  - Přiřazení má „matematický“ význam
  - Proměnná má v daném kontextu vždy tutéž hodnotu – referenční transparence
- Umožňují nové algebraické přístupy

  - Lazy evaluation (x eager evaluation)
    * Možnost používat potencionálně nekonečné struktury
  - Možnost oddělení dat od řízení – nemusíme se starat o to, jak proběhne vyhodnocení
- Umožňuje nové přístupy k vývoji programů
  - Možnost dokazovat programy
  - Možnost transformovat program na základě algebraických vlastností
- Umožňuje lepší využití paralelního provádění programů
  - Jednoduchá dekompozice programů na části, které lze vyhodnocovat paralelně
    * Potencionálně příliš mnoho paralelismů
  - Možnost kompozice dvou paralelních úloh jednoduchou kompozicí funkcí

Použití

V praxi se můžeme setkat jak s čistě funkcionálními jazyky, které striktně vycházejí z teorie (např. FP, Haskell, Miranda, Hope), tak i s hybridními jazyky, které mohou obsahovat i prvky, které jsou se základními principy funkcionálního programování v rozporu. Například často citovaný „typicky“ funkcionální jazyk Lisp je ve skutečnosti jazykem hybridním, neboť umožňuje modifikovat hodnoty již definovaných proměnných. Mezi hybridní jazyky patří rovněž jazyk Standard ML a jazyk Scheme.

Funkcionální programovací jazyky, hlavně čistě funkcionální, se používají spíše v akademických kruzích, než v komerčním softwarovém vývoji. Nicméně funkční programovací jazyky používané v průmyslu a komerčních aplikacích zahrnují Erlang, R (statistický), Matematika (symbolická matematika), Haskell, ML, J a K (finanční analýza) a domain-specific programovací jazyky jako XSLT. Dále jsou funkcionální programovací jazyky důležité pro některá odvětví informatiky, například zabývající se umělou inteligencí, formální specifikací, modelováním nebo rychlým prototypováním.

Příklad

Proč se vlastně zabývat něčím, jako je funkcionální programování? Skoro klasickou odpovědí je ukázka implementace algoritmu QuickSort (jak jinak než v jazyce Haskell), takže proč ji tady nepoužít :-)

Příklad v Haskellu:

```
qsort []     = []
qsort (x:xs) = qsort mensi ++ [x] ++ vesti_rovno
               where
                       mensi  = [ y | y <- xs, y < x]
                       vetsi  = [ y | y <- xs, y >= x]
```

Toto je jen pro názornost, aby bylo vidět jak to vypadá…

Pokusím se uvedený příklad trochu osvětlit. Zjednodušeně řečeno, program v Haskellu sestává z definic funkcí. Definice funkce má dvě části - jméno funkce a případné parametry a tělo funkce (za rovnítkem). Jednu funkci může popisovat i více rovnic. U naší funkce qsort první řádek říká, že setříděním prázdného seznamu (o seznamech bude řeč za chvíli) získáme prázdný seznam. Druhá rovnice je trochu komplikovanější. Můžeme ji popsat asi takto: výsledkem třídění seznamu, jehož první prvek je x a zbytek xs, je seznam, který vznikne spojením (operace ++) setříděného seznamu mensi, prvku x a setříděného seznamu vetsi_rovno. Seznam mensi získáme ze seznamu xs výběrem takových prvků y, které jsou menší než x. Seznam vetsi_rovno získáme obdobně výběrem prvků větších nebo rovných x. Jak vidíme, je při definici funkce qsort použita rekurze, která je pro funkcionální programování typická.

Polymorfizmus, abstrakce

Vlastností známou i z imperativních programovacích jazyků je polymorfismus. Náš příklad s tříděním pomocí QuickSort bude fungovat na celá čísla, znaky, čísla s plovoucí řádovou čárkou i na seznamy seznamů. Obecně je možné setřídit libovolné seznamy, pro jejichž prvky jsou definovány operace porovnávání. Funkcionální jazyky obecně poskytují vysokou míru abstrakce. Jedním z mechanismů abstrakce jsou funkce vyššího řádu (High-order funkce). Haskell umožňuje pracovat s funkcemi jako s běžnými hodnotami - funkce může být parametrem jiné funkce, může být vrácena jako výsledek nebo uložena v datové struktuře (seznamy, ...)

Z hlediska programátora je velmi užitečný také systém modulů - možnost rozčlenit zdrojový kód na logické celky uložené v samostatných souborech. V neposlední řadě je podstatná i automatická správa paměti (garbage collector).

Velikost kódu

V porovnání například s kódem v jazyce C je ihned jasná první přednost - funkcionální programy bývají obvykle kratší (2 až 10 krát) než jejich imperativní ekvivalenty. Další, velmi užitečnou vlastností, kterou u imperativních jazyků nenalezneme, je tzv. **líné vyhodnocování** (**lazy evaluation**). Princip spočívá v tom, že při výpočtu se výrazy vyhodnocují, až když je jich zapotřebí. Může tedy dojít k situaci, kdy některé výrazy nemusí být vyhodnoceny nikdy.

Lazy evaluation - líné vyhodnocení

Další příjemnou věcí funkcionálních jazyku je tzv. líné vyhodnocování (lazy evaluation). Toto znamená, že výsledky jsou vypočítány až ve chvíli, kdy jsou doopravdy potřeba. tj. např. při kompozici funkcí f.g, je g volána tak, jak to požaduje funkce f, tj. může se stát, že funkce g se nezavolá ani jednou. Toto umožňuje jednu krásnou věc - nekonečné seznamy. Můžeme například vytvořit seznam všech prvočísel, nebo nekonečnou Fibonaccioho posloupnost.

Příklad v Haskellu:

```
primes = r [2..]
r (l:list) = [l] ++ r (filter (not.(== 0).(‘mod‘ l)) list
```

Takto jsou definována VŠECHNA prvočísla (podotýkám že prvočísel je neomezeně mnoho), v programu potom tedy můžete použít tuto funkci jako zdroj nekonečně mnoha prvočísel.

High-order funkce - funkce vyššího řádu

- Jako vstup (nebo parametr) je jedna nebo více funkcí
- Jako vystup je funkce

Funkce jsou higher-order ve chvíli, kdy můžou převzít druhou funkci jako argument a navrátit ji jako výsledek. Higher-order funkce aktivují **currying**, což je technika, v které je funkce používána na vlastní argument najednou, přičemž při každém použití navrátí další higher-order funkci, která přijme další argument.

Příklad high-order funkce v Javě:

```
interface HighOrder
{
    public int value(int i);
}
public class Anon
{
    int i;
    public Anon(int i)
    {
        this.i = i ;
    }
    public void modifyI(HighOrder highOrder)
    {
        i = highOrder.value(i);
    }
```

```
    public static void main(String[] args)
    {
        Anon anonymous = new Anon(5);
        anonymous.modifyI(new HighOrder(){
                public int value(int i)
                {
                   return i+1 ;
                }
        });
        System.out.println(anonymous.i);
    }
}
  result => 6
```

Currying

Jedná se o převod funkce s více parametry na funkce s jedním parametrem.

Funkce dostane nějaký parametr (další funkci). Tato funkce z parametru je aplikována na výsledek. Nebo pokud dostane jako parametr dvě funkce, výsledek se může skládat z použití jedné funkce z parametru na druhou fci (f(g(x))).

Příklad v javascriptu:

```
var compose = curry(function( f, g, x, y ){
    return g( x, f(y) );
});
var delegate = curry(function( f, v ){
    f(v);
});
```

Některé jazyky, které to podporují:

- JavaScript
- Python
- Ruby
- SmallTalk
- Scala
- Perl
- ...

Rychlost programu

Funkcionální programovací jazyky mají automatické spravování paměti s garbage collection, v kontrastu se staršími programovacími jazyky jako je C a Pascal, které používají explicitní spravování paměti. Funkcionální programovací jazyky jsou náročnější na systémové prostředky.

Přestože čistě funkcionální jazyky jsou obecně považovány za pomalejší, jakýkoliv imperativní algoritmus se dá vyjádřit v těchto jazycích, přinejhorším s logaritmickým zpomalením. Navíc neměnnost dat může v mnoha případech vést k větší efektivitě díky tomu, že kompilátor může provádět předpoklady, které by byly v imperativních jazycích nejisté.

O tom, že funkcionální programování nepatří pouze do oblasti teoretické, svědčí i to, že jej používají i velké firmy jako například Ericsson.

Funkcionální programování v Javě

Jedná se jen o prvky z funkcionálního programování...

Anonymní třídy

V Javě se s funkcionalním programováním můžeme setkat poměrně často. Prvním takovým příkladem jsou anonymní třídy.

```
Runnable worker = new Runnable()
{
  public void run()
  {
    parseData();
  }
 };
```

Metoda `parseData()` je uzavřena do instance objektu `Runnable worker`. Je to vlastně uzavřená část kodu, která může být kdykoli spuštěna. Tato uzavřená část kódu je spuštěna high-order funkcí (parametr volání je funkce).

Comparators - komparátory

Dalším příkladem jsou kolekce a s nimi spojené *komparátory*. Komparátor je vlastně část kódu, která obstarává porovnání dvou prvků. Komparátor je argumentem řadící funkce. Řadící funkce je high-order funkce.