Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Principy funkcionálního programování a jejich aplikace v moderních OOP jazycích

Diplomová práce

2025

Bc. Jan Stehno

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jan Stehno**
Osobní číslo:	**I23283**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Principy funkcionálního programování a jejich aplikace v moderních OOP jazycích**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je provést rešerši a popis principů funkcionálního programování v různých moderních objektově orientovaných programovacích jazycích a také v jazycích primárně aplikujících paradigma funkcionálního programování. V praktické části bude připravena sada ukázek různých technik v jednotlivých jazycích.

Teoretická část práce se bude zabírat rešerší a základním popisem paradigmatu funkcionálního programování, základními termíny a vlastnostmi funkcionálního programování (např. pure functions, currying, immutability, recursion, higher-order functions, ...). Forma popisu by měla odpovídát seznámení s funkcionálním programováním pro osoby znalé OOP jazykům, popis dále může pokračovat vybranými praktickými tématy (např. funktory, monády, základní typy funktorů/monád v knihovnách funkcionálních jazyků aj.), popis by měl rovněž obsahovat informace o možné praktické aplikaci popisovaných funkcionálních technik ve vybraných OOP jazycích.

Praktická část práce bude vhodně navazovat na teoretická témata a bude obsahovat sadu demonstračních příkladů realizující vybrané techniky v minimálně jednom primárně funkcionálním jazyku a v jednom primárně OOP jazyku (např. Java, C#).

Rozsah pracovní zprávy: **50 – 60 stran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

WIDMAN, Jack. *Learning Functional Programming: Managing Code Complexity by Thinking Functionally*. O'Reilly, 2022. ISBN 9781098111700.
SWAINE, Michael. *Functional Programming: A PragPub Anthology: Exploring Clojure, Elixir, Haskell, Scala, and Swift*. P1.0. Pragmatic Bookshelf, 2017. ISBN 9781680503586.

Vedoucí diplomové práce: **Ing. Roman Diviš, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2024**
Termín odevzdání diplomové práce: **23. května 2025**

L.S.

prof. Ing. Petr Doležel, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 29. listopadu 2024

Prohlašuji:

Práci s názvem Principy funkcionálního programování a jejich aplikace v moderních OOP jazycích jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 22. 8. 2025

Jan Stehno v.r.

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce, Ing. Romanu Divišovi, Ph.D., za jeho cenné rady a podporu při vedení práce. Díky jeho odbornému přístupu a ochotě konzultovat jednotlivé kroky jsem mohl práci dokončit na kvalitní úrovni. Jeho přínos v průběhu vypracování práce byl neocenitelný a velmi si vážím veškeré jeho pomoci při její tvorbě.

ANOTACE

Diplomová práce analyzuje principy funkcionální programování a jejich aplikaci v moderním objektově orientovaném jazyce. Teoretická část systematicky popisuje klíčové koncepty funkcionálního programování, s důrazem na jejich výhody pro psaní bezpečného, testovatelného a paralelizovatelného kódu. Praktická část demonstruje použití těchto technik v čistě funkcionálním jazyce Haskell a v objektově orientovaném jazyce Java. Práce také zkoumá nástroje pro hybridní vývoj a ukazuje, jak funkcionální programování řeší problémy moderního softwarového inženýrství.

KLÍČOVÁ SLOVA

funkcionální programování, objektově orientované programování, čisté funkce, neměnnost, funktory, monády, rekurze, Java, Haskell

TITLE

Principles of Funcional Programming and their application in modern OOP languages

ANNOTATION

This thesis analyzes the principles of functional programming and their application in modern object-oriented language. The theoretical part systematically describes core concepts of functional programming, emphasizing their benefits for writing safe, testable, and parallelizable code. The practical part demonstrates the use of these techniques in a purely functional language Haskell and object-oriented language Java. The thesis also explores tools for hybrid development and shows how principles of functional programming address challanges in modern software engineering.

KEYWORDS

functional programming, object-oriented programming, pure functions, immutability, functors, monads, recursion, Java, Haskell

OBSAH

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK A ZNAČEK

FP	funkcionální programování
OOP	objektově orientované programování
IO	input/output, vstup/výstup
UX	user experience, uživatelská zkušenost
.NET	softwarový framework vyvíjený Microsoftem
LINQ	Language Integrated Query, technologie pro dotazování v jazyce C#
API	Application Programming Interface, rozhraní pro komunikaci mezi softwarovými komponentami
GHC	Glasgow Haskell Compiler, kompilátor jazyka Haskell
ADT	Algebraic Data Type, algebraický datový typ
GADT	Generilized Algebraic Data Type, generalizovaný algebraický datový typ
JVM	Java Virtual Machine, virtuální stroj pro spouštění Java bytecode
ART	Android Runtime, běhové prostředí pro aplikace Android
REST	Representational State Transfer, architektonický styl pro webové služby
CI/CD	Continuous Integration/Continuous Delivery, nástroje a procesy pro automatizovaný vývoj aplikací
PBT	Property-Based Testing, testování obecných vlastností místo konkrétních vstupů
TCO	Tail Call Optimization, optimalizační metoda rekurzivních volání
DSL	Domain-Specific Language, doménově specifické jazyky (pro specifické úkoly)
JIT	Just-In-Time, kompilace za běhu programu

TERMINOLOGIE

Lambda kalkulus	formální systém pro definici funkcí a výpočtů
Turingova úplnost	schopnost systému vyjádřit libovolný výpočet
Church-Roserova věta	věta o konfluenci (kombinace výrazů) v lambda kalkulu
Normální forma	tvar výrazu, kdy již není možná žádná další redukce
Paralelizace, konkurentnost	schopnost spouštět souvisle výpočty nezávislé na sobě
Formální verifikace	matematická kontrola správnosti výpočtu
Multiparadigmatické programování	kombinace více programovacích paradigmat
Deklarativní styl	specifikace, co má být provedeno
Imperativní styl	specifikace, jak to má být provedeno
Kompatibilita, interoperabilita	schopnost různých jazyků spolupracovat
Statická typová inference	typy jsou určeny kompilátorem bez explicitního uvedení
Dědičnost, abstrakce	schopnost rozšiřování funkcionality v OOP
Adaptivní optimalizace	dynamická optimalizace podle chování běhu programu
Bytecode verifikace	kontrola bezpečnosti bytecode před spuštěním
Kontejnerizace	izolace aplikací pomocí kontejnerů (např. Docker)
Dependency injection	návrhový vzor pro předávání závislostí
Sandbox	oddělené prostředí pro omezené spuštění kódu
Distribuované architektury	systémy běžící na více fyzicky oddělených uzlech
Ekosystém	dostupná sada nástrojů, frameworků a podpory
Kompozice	skládání
Asociativita	pořadí skládání nemění výstup
Determinismus	jasně dané chování
Modularita	struktura programu je složena ze samostatných částí
Memoizace	uložení výsledku pro další použití
Inlining	nahrazení volání funkce jejím tělem
Unboxing	převod objektu na primitivní hodnotu
Callback	zpětné volání
Sebedokumentace	schopnost vysvětlení kódu pomocí použitých názvů
Dekompozice	rozložení struktury na části
Garbage collection	automatická správa paměti

ÚVOD

V posledních desetiletích prošel svět programovacích jazyků výrazným vývojem. Vedle tradičních imperativních a objektově orientovaných přístupů se do popředí dostává paradigma funkcionální programování. To klade důraz na deklarativní styl, práci s neměnnými hodnotami a minimalizaci vedlejších efektů. Zatímco dříve bylo funkcionální programování doménou akademického prostředí a specifických jazyků, dnes se jeho principy uplatňují i v běžné praxi a stávají se důležitou součástí vývoje moderních softwarových systémů.

Cílem práce je analyzovat klíčové principy funkcionálního programování, jejich výhody a úskalí, a následně ukázat, jak lze tyto principy aplikovat nejen v čistě funkcionálních jazycích, jako je Haskell, ale také v objektově orientovaných jazycích, například Javě. Práce se rovněž zaměří na dostupné nástroje a knihovny, které podporují funkcionální styl v objektově orientovaném prostředí, a na konkrétních příkladech demonstruje možnosti jejich využití. V praktické části budou vybrané techniky porovnány z hlediska čitelnosti, expresivity a vhodnosti pro různé typy úloh.

Motivací k tomuto tématu je rostoucí důraz na psaní udržitelného, paralelizovatelného, a bezpečného kódu, což jsou oblasti, kde funkcionální přístup přirozeně vyniká. Práce si klade za cíl přispět k hlubšímu porozumění tomuto paradigmatu, a to jak z teoretického hlediska, tak z hlediska jeho praktického využití v každodenní vývojářské praxi.

1 FUNKCIONÁLNÍ PROGRAMOVÁNÍ

1.1 Původ a základy

Funkcionální programování představuje paradigma založené na matematickém konceptu funkcí, kde programy jsou konstruovány výhradně aplikací a kompozicí funkcí. [1] Na rozdíl od imperativních přístupů, které pracují se změnou stavů, funkcionální programování využívá neměnné datové struktury a čisté funkce, což přináší výhody v oblasti spolehlivosti kódu, paralelizace a testovatelnosti.

Funkcionální programování je přístup k programování, který vychází z matematického pojetí funkcí a je postaven na zcela odlišných základech než běžné imperativní jazyky. Zatímco imperativní programovací jazyky jsou založeny na von Neumannově architektuře, kde primárním kritériem je efektivita a modelem je univerzální Turingův stroj, funkcionální jazyky se opírají o matematické funkce a jejich modelem je lambda kalkulus, vyvinutý Alonzo Churchem ve 30. letech 20. století. [2]

Základním stavebním kamenem v imperativních jazycích je příkaz, který mění stavový prostor programu. Naproti tomu, ve funkcionálním programování je základ konstrukcí výraz, který definuje jak algoritmus, tak i vstupní data. [1] Program ve funkcionálním jazyce je v podstatě definicí funkce a výsledkem je funkční hodnota.

Klíčovou charakteristikou funkcionálního programování je práce s tzv. čistými funkcemi. Čistá funkce je taková funkce, která:

- Závisí pouze na parametrech, které dostane.
- Vždy vrátí stejný výsledek pro stejné parametry.
- Nevidí ani nemění vnější stav.
- Nemá žádné vedlejší efekty.

Díky těmto vlastnostem mají čisté funkce mnoho výhod. Protože závisí pouze na svých parametrech, je možné ukládat a později znovu používat výsledky funkcí. Taktéž je možné libovolně paralelizovat a prohazovat pořadí výpočtů, protože závislosti tvoří strom. [1]

1.1.1 Lambda kalkulus

Nejjednodušším abstraktním modelem, který již má sílu univerzálního Turingova stroje a na kterém je funkcionální programování postaveno, je lambda kalkulus. [3] Ten má tři základní konstrukce:

- *Proměnné* – označují libovolný objekt.
- *Lambdy* – funkce s jedním parametrem, definované hlavou a tělem.
- *Aplikace* – operace, která aplikuje funkci na argument.

Tato jednoduchá, ale mocná konstrukce demonstruje, že funkcionální přístup k programování je turingovsky úplný, což znamená, že může řešit stejné problémy jako tradiční imperativní programování. Funkcionální program sestává z definic funkcí (algoritmu) a aplikace funkcí na argumenty (vstupní data). [2] Aparát pro popis funkcí, lambda kalkulus, používá dvě základní operace:

- Aplikace funkce F na argumenty A, zapisováno jako FA.
- Abstrakce ve tvaru $\lambda(\mathrm{x})M[x]$, která definuje funkci (zobrazení) $x \rightarrow M[x]$.

Lambda výrazy popisují bezejmenné funkce, které mohou být aplikovány na parametry. Například $(\lambda(\mathrm{x})\ x * x * x)\ 5 = 5 * 5 * 5 = 125$. Zde zápis představuje popis funkce, kde 5 je argument a celý výraz aplikací funkce, jejímž výsledkem je hodnota 125. [3, 4]

1.2 Klíčové principy

Funkcionální programování je postaveno na několika principech, které jej odlišují od jiných programovacích paradigmat. Jedním ze nejdůležitějších aspektů jsou čisté výrazy a jejich vlastnosti. [4] Čisté výrazy mají několik důležitých vlastností:

- Hodnota výrazu nezávisí na pořadí vyhodnocování.
- Výraz lze vyhodnocovat paralelně.
- Nahrazení podvýrazu jeho hodnotou je nezávislé na výrazu, ve kterém je uskutečněno.
- Vyhodnocení nezpůsobuje vedlejší efekty.
- Operandy operace jsou zřejmé ze zápisu výrazu.
- Výsledky operace jsou zřejmé ze zápisu výrazu.

Church-Roserova věta, která je jedním ze základních teoretických výsledků v oblasti funkcionálního programování říká, že získání normální formy je nezávislé na pořadí vyhodnocování subvýrazů. To znamená, že bez ohledu na to, v jakém pořadí jsou jednotlivé části výrazu vyhodnocovány, vždy se dospěje ke stejnému výsledku, což je klíčové pro paralelizaci výpočtů. [1, 2]

V praktickém programování se funkcionální přístup využívá především tam, kde je důležitá bezpečnost kódu, paralelizace výpočtů nebo kde je potřeba pracovat s transformacemi dat. Díky vlastnostem jako je referenční transparentnost a absence vedlejších efektů je kód ve funkcionálních jazycích často lépe testovatelný a méně náchylný k chybám. [1]

1.3 Definice a rozdíly FP a OOP

Funkcionální programování (FP) lze definovat jako programovací paradigma, ve kterém je výpočet chápán jako vyhodnocení matematických funkcí a které se vyhýbá změně stavu. Je to deklarativní programovací styl, což znamená, že programování je prováděno vyjádřením logiky programu bez popisování jeho řídícího toku. [5]

Zatímco funkcionální programování je zaměřeno na transformace dat pomocí funkcí [6], objektově orientované programování (OOP) se soustředí na modelování světa pomocí objektů, které mají stav a chování. V OOP je základní jednotkou objekt, který zapouzdřuje data (atributy) a funkce (metody), které s těmito daty pracují. Objekty jsou instancemi tříd, které definují jejich strukturu a chování. Klíčovými koncepty OOP jsou zapouzdření, dědičnost a polymorfismus.

Hlavní rozdíly mezi FP a OOP:

- Přístup ke stavu: FP se vyhýbá stavům a pracuje s neměnnými daty, zatímco OOP je postaveno na objektech, které mají stav a chování.
- Manipulace s daty: V FP jsou data transformována čistými funkcemi. V OOP objekty samy manipulují se svými daty prostřednictvím metod.
- Organizace kódu: FP organizuje kód kolem funkcí a jejich kompozice. OOP organizuje kód kolem objektů a jejich vztahů.
- Řízení toku: FP používá rekurzi a funkce vyššího řádu k řízení toku programu. OOP používá metody a zprávy mezi objekty.

V moderním programování se stále častěji objevují i hybridní přístupy, které kombinují prvky funkcionálního a objektově orientovaného programování. Mnoho současných programovacích jazyků podporuje oba styly, což umožňuje vývojářům využít výhody obou paradigmat podle konkrétní situace a potřeby. [6]

1.4 Historický vývoj a kontext

Funkcionální programování má své kořeny v teoretických základech lambda kalkulu, který vyvinul Alonzo Church ve 30. letech 20. století jako formální systém pro studium funkcí a rekurze. [1] Tento matematický aparát se stal základem pro první funkcionální jazyky, přičemž klíčovým milníkem byl vznik Lispu v roce 1958, navrženého pro umělou inteligenci. Lisp zavedl koncepty jako rekurzi a práci se seznamy, čímž položil základy praktického využití funkcionálních principů. [7]

Vývoj funkcionálních jazyků byl těsně spjat s akademickým výzkumem, ale jejich praktické uplatnění rostlo s nástupem vícejádrových procesorů a potřebou paralelního zpracování. Zatímco rané jazyky jako Lisp a Scheme byly hybridní, moderní jazyky jako Haskell nebo Elm striktně vynucují neměnnost dat, což umožňuje kompilátorům agresivnější optimalizace. [8]

Teoretický pokrok v oblasti typových systémů a monadických struktur pro řízení vedlejších efektů umožnil funkcionálním jazykům expandovat do průmyslových aplikací. Současně se ukázalo, že koncepty jako vyšší kinded typy nebo dependentní typy poskytují nástroje pro formální verifikaci kódu, což je klíčové pro kritické systémy. [8] Tento vývoj odráží synergii mezi teoretickou informatikou a praktickými potřebami softwarového inženýrství.

1.5 Motivace pro využití FP v moderním vývoji

Rostoucí popularita funkcionálního programování ve 21. století souvisí s expanzí distribuovaných systémů a potřebou zvládat složitost moderních aplikací. Neměnnost dat a čisté funkce eliminují kolize při přístupu (závislost na pořadí výpočtu), což usnadňuje vývoj spolehlivých konkurentních systémů. [9]

Další motivací je explozivní růst datové analýzy a strojového učení, kde funkcionální přístup umožňuje deklarativní popis transformací. Lazy evaluation v Haskellu nebo Clojure umožňuje pracovat s nekonečnými datovými streamy, což je klíčové pro real-time zpracování. Přechod k mikroservisním architekturám a zpracování bez nutnosti serveru zvýšil poptávku po kompozovatelnosti a testovatelnosti, což jsou inherentní vlastnosti funkcionálního programování. Paradoxně, i když vzniklo funkcionální programování z teoretických pohnutek,

jeho současný vzestup je hnán pragmatickými požadavky na škálovatelnost a udržovatelnost systémů. [9]

1.6 Přínosy a výzvy FP

Hlavní přínos funkcionálního programování spočívá v redukci latentních chyb prostřednictvím referenční transparentnosti. To usnadňuje formální analýzu a automatizované testování. [1] Neměnnost dat eliminuje náhodné kolize při paralelním přístupu, což zásadně snižuje režii synchronizace ve vícevláknovém prostředí. [6]

Mezi praktické výhody patří i lepší modularita. Funkce vyššího řádu jako map nebo filter umožňují komponovat transformace jako stavebnice. Typové systémy jazyků jako Haskell nebo F# detekují nekompatibilní operace již při kompilaci, což urychluje vývojový cyklus. [10]

Výzvami zůstává strmá křivka učení a omezená dostupnost knihoven. Optimalizace paměťové náročnosti u rekurzivních algoritmů vyžaduje hlubší porozumění evaluačním strategiím. Integrace s imperativními systémy někdy nutí použít řešení jako IO monády, což narušuje čistotu kódu. [11]

1.7 Aplikační domény a využití FP

Funkcionální jazyky excelují v oblastech vyžadujících vysokou míru abstrakce a formalizace. V akademickém prostředí se Haskell používá pro specifikaci doménově založených jazyků a formální verifikaci hardwaru. V kryptografii umožňují dependentní typy (jazyka Idris) automatickou kontrolu šifrovacích protokolů. [4]

V oblasti distribuce dat jsou funkcionální jazyky výborné v budování systémů tolerantních k chybám díky actor modelu a „nechť selže proces“ filozofii. Funkcionální principy se prosadily i v UX designu. Redux standardizoval předvídatelné řízení stavu ve frontendových aplikacích. Biomedicínský výzkum využívá jazyky jako Scala nebo Spark pro analýzu genomických dat, kde paralelní funkce jako map a reduce přirozeně odpovídají funkcionálnímu stylu. [9]

1.8 Multiparadigmatické programování

Moderní programování jazyky stírají hranice mezi paradigmaty. Scala kombinuje OOP s FP, F# integruje .NET ekosystém, zatímco TypeScript přidává statické typování do JavaScriptu. [12] Tento trend umožňuje používat funkcionální přístup pro jádro business logiky a imperativní styl pro I/O operace nebo optimalizace.

V enterprise prostředí se prosazuje „graduální adopce“ funkcionálních principů. [9] Java Stream API zavádí funkcionální práci s kolekcemi, C# LINQ umožňuje deklarativní dotazy. [13] Tento hybridní přístup minimalizuje rizika přechodu na nové paradigma při zachování výhod čistých funkcí pro kritické části systému.

Budoucnost vidí výzkumníci v jazykových rozšířeních pro efektové systémy (pro sledování vedlejších efektů v typech) a automatickou paralelizaci. Současně některé nástroje pro spouštění kódu umožňují kompilovat funkcionální jazyky do univerzálního bytecode pro běh v jakémkoliv prostředí. [9] Multiparadigmatický vývoj se tak stává mostem mezi akademickou důsledností a průmyslovou účelností.

2 VYBRANÉ PROGRAMOVACÍ JAZYKY

Pro praktickou část diplomové práce byl vybrán jeden čistě funkcionální jazyk – Haskell a jeden objektově orientovaný jazyk – Java. Jazyky Haskell a Java tedy představují dva odlišné přístupy k softwarovému vývoji, ztělesňující obě paradigmata. Zatímco Haskell vznikl jako akademický projekt pro výzkum čistě funkcionálních konceptů [8], Java se prosadila jako průmyslový standard pro enterprise aplikace. [13]

2.1 Haskell

Vznik Haskellu souvisí s potřebou vytvořit standardizovaný funkcionální jazyk pro akademický výzkum. V roce 1987 na konferenci FPCA v Portlandu vznikla iniciativa, která vyústila v první verzi Haskell 1.0 v roce 1990. Klíčovým milníkem se stalo vydání Haskellu 98, které definovalo stabilní jádro jazyka a standardní knihovnu. Současný „standard“ tvoří Glasgow Haskell Compiler (GHC), který implementuje četná rozšíření včetně typového systému s pokročilými vlastnostmi jako jsou typové rodiny a generalizované algebraické datové typy (GADT). [8]

Funkcionální jazyky jako Haskell se vyznačují silným důrazem na čisté funkce, což v praxi znamená absenci vedlejších efektů a důsledné oddělení výpočtu od interakce s okolním světem. To vede ke zvýšené předvídatelnosti a snazší formální analýze programů. Typickými rysy jsou referenční transparentnost, líné vyhodnocování a staticky typovaný systém s typovou inferencí, které dohromady poskytují silný základ pro bezpečné a efektivní zpracování dat. [4] Díky těmto vlastnostem je možné v Haskellu pracovat s abstrakcemi, které by v imperativních jazycích vyžadovaly mnohem více kódu a explicitního řízení toku dat. [5]

Ve výsledku se Haskell často uplatňuje v oblastech, kde je klíčová matematická přesnost, paralelismus či robustní správa vedlejších efektů. Mezi typické aplikační domény patří kompilátory, nástroje pro zpracování jazyků a finanční systémy s požadavky na formální verifikaci nebo výzkumné simulace. [8] Dále se Haskell osvědčil v oblasti doménově specifických jazyků a při návrhu bezpečných rozhraní nad komplexními systémy, kde se výhody typového systému a čisté sémantiky projeví nejvýrazněji. [5]

2.2 Java

Java představuje robustní objektově orientovaný jazyk, jehož vývoj byl motivován snahou o přenositelnost a bezpečnost napříč platformami. Zavedení runtime prostředí Java Virtual Machine (JVM) umožnilo koncept „Write once, Run Anywhere“, který výrazně ovlivnil

průmyslové standardy. [14] Přísná objektová orientace v kombinaci se statickým typováním a automatickou správou paměti činí z Javy preferovanou volbu pro rozsáhlé aplikace v různých doménách. Generická syntaxe a type erasure zachovává zpětnou kompatibilitu, přestože omezuje některé pokročilé typové operace v runtime prostředí. [13]

Moderní JVM implementace využívají pokročilé optimalizační techniky včetně „just-in-time“ kompilace a adaptivní optimalizace, čímž vyvažují výhody interpretace s výkonem blízkým nativnímu kódu. [13] Java zachovává bytecode verifikaci jako základ bezpečnostního modelu Java Virtual Machine (JVM), avšak tradiční sandboxový přístup byl nahrazen moderními kontejnerovými izolačními mechanismy. [14] Bezpečnost enterprise aplikací se nyní opírá o integraci s cloudovými platformami a modulární architekturu, zatímco mobilní nasazení na systémech Android využívá specifický Android Runtime (ART) s vlastním permission systémem. [13]

V oblasti enterprise vývoje hraje Java klíčovou roli díky rozsáhlému ekosystému knihoven a frameworků. Dominantní postavení zde zaujímá Spring Framework, který zajišťuje infrastrukturu pro dependency injection, správu transakcí a integraci s databázemi či REST službami. Automatizace sestavování a nasazování se opírá o nástroje jako Maven, Gradle a moderní CI/CD postupy, přičemž cloudová řešení využívají kontejnerizaci, čímž Java zůstává relevantní i v současných distribuovaných architekturách. [15]

2.3 Srovnání

Haskell a Java reprezentují odlišná programovací paradigmata s rozdílným přístupem k návrhu softwaru. Zatímco Haskell staví na funkcích, neměnnosti a abstrakcích jako typové třídy či monády, Java využívá objekty, měnitelný stav a dědičnost přes rozhraní a abstraktní třídy. Výkonnostně Haskell vyniká v numerických výpočtech díky pokročilým optimalizacím překladače GHC, zatímco Java dosahuje vysoké efektivity v I/O scénářích, zejména v síťových aplikacích.

Z hlediska ekosystému má Java výrazně širší průmyslové pokrytí, stabilní zázemí a nástroje pro vývoj velkých aplikací. Haskell naproti tomu nachází uplatnění v akademickém výzkumu a úzce specializovaných odvětvích, kde jsou klíčové formální záruky a matematická přesnost. Nicméně, oba jazyky se vzájemně inspirují. Java integruje funkcionální prvky jako lambda výrazy a Stream API, zatímco Haskell nachází cestu do průmyslového využití například v oblasti blockchainu nebo finančních nástrojů. Tyto tendence ukazují na sbližování obou světů a rostoucí význam multiparadigmatického přístupu v moderním vývoji. [12]

3 ZÁKLADNÍ PRINCIPY FP

3.1 Datové typy

Na úvod je za potřebí představit možnosti datových typů a jejich použití. Haskell má k dispozici velmi podobné datové typy jako Java. Mezi ně patří například *Int*, *Integer*, *Float*, *Double*, *Bool* nebo *Char*. *String* je typový alias a je definován jako list *[Char]*. [10] Nad těmito datovými typy využívá Haskell typových tříd, které jsou „souborem“ datových typů. Příkladem může být typová třída *Num*, pod kterou spadají všechny číselné datové typy. *Integral* také spadá pod třídu *Num* (je „podtřídou“ *Num*). Tato typová třída zaobaluje už pouze *Int* a *Integer*. [6]

```haskell
someInt :: Int
someInt = 6

someWord :: String
someWord = "Hello"

someFloat :: Float = 2.6

someDouble = 3.7
```

Zdrojový kód 1 - Vytváření základních proměnných

3.2 Tuples

V určitém směru jsou tuples (n-tice) něco jako listy v Javě. Používají se pro uložení více hodnot jako jedna. Oproti listům mají ale několik podstatných rozdílů. N-tice mohou obsahovat hodnoty různých datových typů. [10] Hodí se zejména v případě, kde je znám počet hodnot, které jsou potřeba jako kombinace. Pro dvojice existují dvě funkce, kterými lze zjistit hodnotu prvního a hodnotu druhého prvku. Jedná se o funkce *fst* a *snd*. [16]

```haskell
tupleCharInt :: (Char, Int)
tupleCharInt = ('a', 1)

firstElement :: Char = fst tupleCharInt -- 'a'
secondElement :: Int = snd tupleCharInt -- 1

tuple3dVector :: (Int, Int, Int)
tuple3dVector = (5, 7, 2)
```

Zdrojový kód 2 - Příklady tuples (n-tic)

Jakákoliv kombinace typů hodnot a jejich počet v n-tici se interně chová jako specifický datový typ. Tím jsou n-tice heterogenní, což znamená, že je jejich obsah předem určen a nemůže být

zaměněn nebo porovnáván s n-ticemi, které obsahují jiný počet hodnot nebo jsou jejich hodnoty jiných datových typů. [10] Tato vlastnost dává n-ticím výhodu v době, kdy je potřeba uchovávat specifické kombinace hodnot.

3.3 List

Nejvíce používanou strukturou v jazyce Haskell je list. Ten může být použit mnoha způsoby k modelování a řešení různých problémů. Listy jsou v Haskellu homogenní datovou strukturou. Lze do nich uložit množství hodnot stejného datového typu. [6]

```haskell
someIntList :: [Int]
someIntList = [1, 2, 3, 4, 5]

someCharList :: String
someCharList = ['H', 'e', 'l', 'l', 'o'] -- String je implicitně [Char]
```

Zdrojový kód 3 - Základní inicializace listů

Obsahy konečných listů lze také generovat. Pokud je potřeba list s hodnotami od 1 do 20, list lze vypsat ručně nebo lze využít rozsahů. Rozsahy umožňují definovat posloupnost, kterou se bude generování hodnot v listu řídit. Funkce replicate umožňuje generovat listy s konečně se opakující hodnotou. Vstupními parametry funkce replicate jsou počet hodnot a samotná hodnota. [10]

```haskell
someRangeList :: [Int]
someRangeList = [1 .. 20] -- [1, 2, 3, ...]

sequenceRangeList :: [Int]
sequenceRangeList = [2, 4 .. 20] -- [2, 4, 6, ...]

reversedRangeList :: [Char]
reversedRangeList = ['z', 'y' .. 'a'] -- ['z', 'y', 'x', ...]

replicateList :: [Int]
replicateList = replicate 3 1 -- [1, 1, 1]
```

Zdrojový kód 4 - Inicializace listů pomocí rozsahů a funkce replicate

V Haskellu lze díky línému vyhodnocování generovat i listy nekonečné. Pro vygenerování nekonečných listů lze využít několik technik. První možností je nespecifikování horní hranice rozsahu. Dalšími možnostmi jsou funkce repeat a cycle. Repeat generuje list hodnot zadaných jako vstupní parametr. Vstupem funkce cycle je pole, jehož hodnoty jsou pak opakovány ve výsledném listu. [6]

Protože líné vyhodnocování nedovolí vygenerovat list ihned při inicializaci, je potřeba využít další funkce, *take*, která umožňuje „vzít“ si určitý počet hodnot od počátku listu. Vstupními parametry funkce *take* jsou počet a list hodnot. Tuto funkci lze použít i pro listy konečné. [10]

```
takeList :: [Int]
takeList = take 6 [2, 4 ..] -- [2, 4, 6, 8, 10, 12]

takeRepeatList :: [Int]
takeRepeatList = take 6 (repeat 1) -- [1, 1, 1, 1, 1, 1]

takeCycleList :: [Int]
takeCycleList = take 6 (cycle [1, 2, 3]) -- [1, 2, 3, 1, 2, 3]
```

Zdrojový kód 5 - Inicializace nekonečených listů a funkce take

Další možností vytváření listů je použití generátoru s předpisem, tzv. list comprehension. Jedná se o syntaktický prostředek, který umožňuje deklarativním způsobem konstruovat nové listy na základě existujících kolekcí, přičemž je možné hodnoty zároveň filtrovat a transformovat. Struktura výrazu vychází z matematické notace množin. Základní součástí je výraz výsledné hodnoty, generátor pro zdroj hodnot a volitelné podmínky. [10]

```
listEvenNumbers :: [Int]
listEvenNumbers = [x * 2 | x <- [1 .. 10], even x] -- [4, 8, 12, 16, 20]
```

Zdrojový kód 6 - List s využitím list comprehension

Předpis výše lze definovat jako konstrukci nového listu čísel *x*, ve kterém se budou nacházet taková celá čísla od 1 do 10 (zdroj), která jsou sudá (podmínka, filtrace). Každé takové číslo v seznamu bude vynásobeno dvěma (výsledná hodnota, transformace).

Interní reprezentace listů v jazyce Haskell není založena na indexovaných strukturách jako jsou pole v jazyce Java, ale na rekurzivním datovém typu, který využívá konstruktor (:) pro vkládání prvku na začátek listu. Každý prvek neprázdného listu je tedy konstruován jako hodnota, která má tzv. hlavu (*head*, první prvek) a ocas (*tail*, zbytek listu bez hlavy). [16] List *numberList* v následujícím příkladě bude použit i pro další části kapitoly o listech.

```haskell
numberList :: [Int]
numberList = [1, 2, 3]

numberListCompilator :: [Int]
numberListCompilator = 1 : (2 : (3 : []))
```

Zdrojový kód 7 - Konstruktor listu

Výše uvedená verze listu, který „vidí“ kompilátor, odhaluje jeho skutečnou strukturu. Jedná se o rekurzivní strukturu, kde každý prvek je navázán na zbytek listu. Tento přístup umožňuje efektivní práci s prvky na začátku seznamu a přirozeně podporuje definici funkcí rekurzivního charakteru. [6]

Pro přístup k jednotlivým částem listu lze využít funkce *head*, *tail*, *init* a *last*. Všechny tyto funkce mají jako vstupní parametr neprázdný list. [10] Tvoří základní operace nad listy pro složitější zpracování sekvenčních dat, které bude dále rozvedeno v návaznosti na rekurzi, funkce vyššího řádu a další techniky funkcionálního programování.

```haskell
listHead :: Int
listHead = head numberList -- 1

listTail :: [Int]
listTail = tail numberList -- [2, 3]

listInit :: [Int]
listInit = init numberList -- [1, 2]

listLast :: Int
listLast = last numberList -- 3
```

Zdrojový kód 8 - Funkce head, tail, init a last nad listy

Nad listy lze volat, kromě výše zmíněných, také další užitečné funkce. Podobně jako v Javě lze například zjistit délku listu pomocí funkce *length*, vyhodnotit, zda je list prázdný díky funkci *null*, nebo převrátit list funkcí *reverse*. [10]

```haskell
listLength :: Int = length numberList      -- 3
listNull :: Bool = null numberList         -- False
listContains :: Bool = elem 4 numberList   -- False

listReverse :: [Int] = reverse numberList  -- [3, 2, 1]
listTake :: [Int] = take 2 numberList      -- [1, 2]
listDrop :: [Int] = drop 2 numberList      -- [3]

listMinimum :: Int = minimum numberList    -- 1
listMaximum :: Int = maximum numberList    -- 3
listSum :: Int = sum numberList            -- 1 + 2 + 3 = 6
listProduct :: Int = product numberList    -- 1 * 2 * 3 = 6
```

Zdrojový kód 9 - Další základní funkce nad listy

3.4 Algebraické datové typy

Algebraic Data Type (ADT) slouží k definování složitějších struktur, které mohou obsahovat různé varianty datových typů. Tyto datové typy jsou často používány k reprezentaci objektů nebo entit, podobně jako třídy v objektově orientovaných jazycích. V jazyce Haskell jsou datové konstrukce definovány pomocí klíčového slova *data* a nevyžadují definování metod, pokud to není nutné. [6]

```haskell
data Person = Person {name :: String, age :: Int}
```

Zdrojový kód 10 - Sestrojení algebraického datového typu Person

Datový typ *Person* obsahuje dvě pole, *name* a *age*. Pokud je potřeba celý datový typ zobrazit, Haskell neprovádí převod na výstup automaticky. K tomu je nutné implementovat instanci třídy *Show*. Ta určuje, jak bude daný datový typ zobrazen. Funkce *show*, která je součástí této třídy, následně používá instanci datového typu k jeho převodu na řetězec.

```haskell
instance Show Person where
  show (Person name age) = "Person(" ++ name ++ ", " ++ show age ++ ")"
```

Zdrojový kód 11 - Použití třídy Show k zobrazení algebraického datového typu

Pokud by nebyla implementována tato instance, pokus o výpis hodnoty typu *Person* by vedl ke kompilační chybě, protože Haskell neví, jak tento datový typ převést na řetězec. Alternativně může být pro převod datového typu na řetězec použita samostatná funkce, která se nemusí připojovat ke třídě *Show*. Tím se zajistí explicitní převod z datového typu *Person* na *String*.

V jazyce Haskell se při práci s algebraickými datovými typy používá přístup k jednotlivým polím tohoto typu prostřednictvím funkcionálního zápisu. Pole definovaná v datovém typu jsou v podstatě funkce, které mají typ, jenž odpovídá typu daného pole. Tato pole se tedy chovají jako funkce, které berou instanci typu jako vstup a vrátí hodnotu příslušného pole.

```
name :: Person -> String
age :: Person -> Int
```

Zdrojový kód 12 - Explicitní vyjádření implicitního převodu polí datového typu na funkce

3.5 Neměnnost

Neměnnost (immutability) je základním konceptem funkcionálního programování. Významně ovlivňuje strukturu a provádění programů. Immutability princip říká, že jakmile je „proměnné“ přiřazena nějaká hodnota, nelze ji po celou dobu jejího trvání měnit. [6]

Neměnitelnost hodnoty znamená, že všechny vytvořené hodnoty jsou konstantní a jakákoliv změna vyžaduje nové instance. Nelze upravovat stávající hodnotu. K úpravě stávajících hodnot je potřeba proměnnou nebo datovou strukturu vytvořit znovu s kopií nezměněných hodnot a požadovanými změněnými hodnotami.

3.5.1 Vlastnosti

Neměnitelná data zajišťují, že jednou nastavená hodnota zůstane nezměněna, což zvyšuje předvídatelnost a snižuje pravděpodobnost neočekávaných změn v kódu. Protože se data nemohou neočekávaně měnit, je vyhledávání chyb jednodušší. To vede k rychlejšímu řešení, ladění a testování. Udržuje integritu dat a snižuje riziko nechtěných výsledků. [17]

Immutability také přispívá k vytváření čistých funkcí. Protože musí být funkce deterministické, tedy musí vždy produkovat stejný výstup pro stejný vstup, je zapotřebí, aby spoléhaly na neměnná data. Nezměnitelnost také zajišťuje, aby funkce nemodifikovaly žádný vnější stav nebo proměnné mimo svůj rozsah. Hraje také velkou roli v paralelním a distribuovaném programování. [17]

I když neměnnost přináší řadu výhod, jedním z hlavních problémů je spotřeba paměti. Vytváření nových instancí pro každou změnu může vést ve srovnání s proměnnými strukturami ke zvýšené spotřebě paměti. Některé jazyky optimalizují práci s immutable daty a strukturami promocí technik jako persistent data structures, které efektivně sdílí části dat mezi různými verzemi, aby snížily paměťovou zátěž. [17]

3.5.2 Použití

Díky neměnnosti dat nebo struktur odpadá potřeba synchronizace při přístupu z více vláken. V distribuovaných systémech usnadňuje replikaci a zajišťuje konzistenci dat mezi uzly. Reaktivní programování využívá immutable stav pro lepší zaznamenávání jeho změn. [6] V Haskellu je neměnnost dat výchozí vlastností jazyka. Všechny základní datové struktury jsou immutable.

V porovnání s jazykem Java a díky neměnnosti v jazyce Haskell nelze měnit již přiřazené hodnoty k proměnným. Místo toho je nutné založit novou proměnnou, do které bude nová hodnota uložena.

```
someInt :: Int
someInt = 6
someInt = someInt + 1 -- nelze

someIntPlusOne :: Int = someInt + 1
someIntPlusThree = someInt + 3
someNewDouble = 3.7
```

Zdrojový kód 13 - Neměnnost a změna hodnoty proměnné

Ačkoli Haskell klade důraz na neměnnost, poskytuje mechanismy pro správu mutací a dalších vedlejších efektů prostřednictvím monadického rozhraní. V Javě lze pracovat s mutable objekty bez nutnosti podpůrných mechanismů. Umožňuje ale i tvorbu immutable proměnných a struktur. [15]

3.6 Čistá funkce

Čistá funkce neboli pure function, vytváří vždy stejný výstup pro stejný vstup. Funkce také nemodifikují vnější stavy, proměnné ani data mimo svůj rozsah, čímž zapouzdřují svoji logiku. Pracují s neměnnými datovými strukturami. Místo toho, aby měnily vstupní data, vytvářejí a vracejí nové datové struktury. Nespoléhají se na proměnlivá data. [6]

3.6.1 Vlastnosti

Pro stejný vstup vrací vždy stejný výstup. Nemají žádný vedlejší efekt, nezapisují do souboru, nebo nevyvolávají výjimky. Jsou deterministické a tím zjednodušují uvažování o kódu. Také zajišťují neměnné chování, které je výhodné zejména při transformaci dat.

Čisté funkce jsou referenčně transparentní a modulární, což umožňuje jejich znovupoužitelnost. Lze je snadněji testovat a ladit. Každá funkce může být otestována relativně jednoduchými

testy. Dobře se také hodí k paralelnímu zpracování, protože nesdílejí proměnlivý stav. [17] To umožňuje, například při provádění rozsáhlých transformací dat, efektivní provádění napříč více jádry procesoru, a tak zvyšují celkový výkon. Mezi možnou optimalizaci patří například ukládání výsledků čistých funkcí do mezipaměti, které může zabránit nadbytečným výpočtům.

3.6.2 Použití

Čisté funkce lze použít k celé řadě deterministických operací, hlavně těch matematických. Sčítání, násobení, počítání průměrů, minimálních a maximálních hodnot, nebo zaokrouhlování jsou operace, které jsou vhodnými představiteli čistých funkcí. Díky konzistentním výsledkům jsou také vhodné při transformacích dat, filtrování, nebo mapování seznamů a kolekcí dat.

```
add5 a = a + 5
```

Zdrojový kód 14 - Jednoduchá čistá funkce

Výchozím typem funkcí v Haskellu jsou právě ty čisté. Jsou to funkce, které dodržují matematickou definici. Ke každé vstupní hodnotě přiřadí výstupní hodnotu. Koncept čistých funkcí lze uplatnit i v objektově orientovaných jazycích. Je ale za potřebí dodržet přinejmenším dvou vlastností. Funkce nesmí mít vedlejší účinky. Musí přijímat a vracet nějakou hodnotu.

3.7 Obecná funkce

Funkce jsou základním stavebním prvkem funkcionálního programování. V Haskellu jsou všechny funkce definovány v deklarativním stylu, což znamená, že funkce se přímo vztahují k výpočtům, a ne k procedurálním akcím. Každá funkce je definována jako mapování vstupu na výstup a její parametry jsou neoddělitelně spojeny s návratovou hodnotou.

```
addOne :: Int -> Int
addOne n = n + 1

result = addOne 4 -- 5
```

Zdrojový kód 15 - Jednoduchá funkce

V příkladu výše je *addOne* funkce, která má jeden vstupní argument typu *Int* a vrací *Int* jako vstupní hodnotu zvýšenou o jedna. Písmeno *n* zastupuje vstupní hodnotu. Funkce běžně uplatňují pattern matching, který jim umožňuje reagovat na různé formy vstupu. [6] V tomto případě má funkce vzor pouze jeden, a to *addOne n*, který přijímá jakoukoliv *Int* hodnotu. Za rovnítkem je návratová hodnota pro definovaný vzor.

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False

result = isZero 4 -- False
```

Zdrojový kód 16 - Jednoduchá funkce se vzory

Funkce *isZero*, stejně jako funkce *addOne*, má jeden vstupní parametr typu *Int* a vrací hodnotu typu *Bool*. Funkce vrací *True*, pokud je na vstupu nula. Při volání funkce *isZero 4* se kontroluje rovnost vzoru *isZero 0*. Protože 4 není 0, tímto vzorem volání funkce „propadne“ a zkontroluje se další vzor. Protože další vzor odpovídá vstupu, úspěšně se vrátí výstupní hodnota *False*.

3.7.1 Guards

Guards jsou konstrukce ve funkcionálním programování používané k řízení toku programu na základě podmínek. Umožňují definovat více alternativních větví výpočtu tak jako konstrukce if-else v objektově orientovaných programovacích jazycích. Guards se obvykle používají v definicích funkcí, vzorovém porovnávání (pattern matching) a dalších konstrukcích, kde je třeba rozhodovat na základě různých podmínek. Použití guards je obzvláště výhodné při definování funkcí s více možnostmi výpočtu. [10] Místo složitých podmínek v těle funkce umožňují guards přehledné vyjádření jednotlivých případů jako oddělených pravidel.

```
decisionWithGuards x
  | x < 0     = x -- v případě, že x bude menší než 0
  | otherwise = -x -- v ostatních případech
```

Zdrojový kód 17 - Guards

V kódu jsou guards označeny tzv. svislítkem, za kterým je definována podmínka a návratová hodnota. Jejich fungování lze přirovnat k fungování konstrukcí *if-else*. Guards se typicky vyhodnocují shora dolů. Program postupně kontroluje jednotlivé podmínky, dokud nenajde první splněnou. Každá podmínka představuje logický výraz, který musí být pravdivý, aby se vybraná větev vykonala. Pokud žádná z podmínek není splněna, může být definována výchozí větev otherwise, která zajistí zpracování všech ostatních případů. [10]

```haskell
isPositive :: Int -> Bool
isPositive n
  | n >= 0 = True
  | otherwise = False

result = isPositive (-2) -- False
```

Zdrojový kód 18 - Jednoduchá funkce využívající guards

V příkladu funkce *isPositive* přijímá hodnotu typu *Int*, vrací *Bool*, a kontroluje, zda je vstup kladné nebo záporné číslo. Technika pattern matching zajistí zpracování jakékoliv hodnoty typu *Int*. Následně guards definují podmínky, na jejichž základě bude vyhodnocena návratová hodnota funkce. Stejně jako u vzorů se porovnává vstup s podmínkou a postupně „propadává“ dokud nenarazí na podmínku, která je vyhodnocena jako *True*. [10]

Pokud funkce využívající guards obsahuje v každé větvi výpočet na základě vstupní hodnoty, který se následně porovnává, dochází k opakování stejného výpočtu. Tento přístup může vést k redundanci, která není žádoucí. V Haskellu je možné tento výpočet definovat jednou a použít ho několikrát díky konstrukci *where*.

```haskell
calculateOddEven :: Int -> String
calculateOddEven x
  | c == 0 = "Výsledek je 0"
  | even c = "Výsledek je sudý"
  | odd c = "Výsledek je lichý"
  where
    c = x * 2 - 3

result = calculateOddEven 5 -- "Výsledek je lichý"
```

Zdrojový kód 19 - Funkce využívající where

Funkce by bez použití konstrukce *where* musela mít definovaný stejný výpočet pro všechna porovnání. S využitím *where* lze definovat výpočty jednou a uložit je do lokálních proměnných. Tyto proměnné pak zjednodušují definici funkce a zlepšují celkovou čitelnost.

Alternativou ke konstrukci *where* je konstrukce *let-in*, která umožňuje definovat pomocné hodnoty nebo výpočty přímo v těle funkce. Na rozdíl od *where*, který umisťuje definice na konec funkce, *let-in* umožňuje definovat proměnné na začátku těla funkce nebo v konkrétním výrazu.

```haskell
calculateNumber :: Int
calculateNumber = 5 * (let a = 3 in a - 8) + 1 -- -24

calculateAddFive :: Int -> Int
calculateAddFive x =
  let y = 5
   in x + y

result = calculateAddFive 10 -- 15
```

Zdrojový kód 20 - Funkce využívající let-in

Guards také připomínají známou konstrukci z objektově orientovaného programování, a to *if-else*. V jazyce Haskell lze tuto konstrukci také využít. Hlavním rozdílem je nutnost použití části *else*. Přísné použití *if* a *else* v zásadě představuje ternární operátor, kde je nutné, stejně jako v tomto případě, použít obě větve podmínky. [10] Přestože je *if-else* velmi přímočaré, v některých případech, kdy je potřeba provést více než jednu podmínku, jsou guards výhodnější.

```haskell
isPositive :: Int -> Bool
isPositive x = if x >= 0 then True else False

result = isPositive (-2) -- False
```

Zdrojový kód 21 - Jednoduchá funkce využívající if-else

Stejně jako má jazyk Java k dispozici switch, nabízí Haskell konstrukci *case-of*, která umožňuje rozdělit výpočet na základě hodnoty výrazu. [10] Konstrukce *case* je flexibilnější, protože může pracovat s libovolným výrazem, nikoliv pouze s hodnotami jednoduchých typů.

```haskell
describeNumber :: Int -> String
describeNumber x = case x of
  0 -> "Zero"
  1 -> "One"
  _ -> "Other"

result = describeNumber 3 -- "Other"
```

Zdrojový kód 22 - Jednoduchá funkce využívající case-of

Funkce využívá konstrukci *case-of* pro pojmenování čísel. Pokud jeden z případů bude odpovídat vstupní hodnotě, vrátí funkce příslušné pojmenování. Ve funkci se využívá zástupného znaku (_), který v Haskellu představuje tzv. wildcard. [10] Používá se zejména

v pattern matching konstrukcích ve chvíli, kdy není potřeba kontrolovat přesnou hodnotu. V této *case-of* konstrukci bude do třetího případu spadat jakékoliv číslo, které nezachytily první dva případy.

3.8 Referenční transparentnost

Referenční transparentnost znamená, že výsledek funkce nebo výrazu závisí pouze na jejích vstupních parametrech. Globální proměnné, jakékoliv vnější faktory nebo aktuální čas nemají na výsledek žádný vliv. To umožňuje předpovědět výsledek funkce pouze na základě jejich vstupů, bez ohledu na kontext, ve kterém je funkce volána. [6] Tato vlastnost dovoluje nahradit výrazy jejich hodnotami bez změny chování programu nebo matematického výrazu.

Funkce s referenční transparentností jsou deterministické, protože pro stejný vstup vrací vždy stejný výstup. Nepoužívají náhodná čísla nebo jiné nepředvídatelné prvky. Tyto funkce také nejsou závislé na vnějším stavu a nepracují s vnějšími zdroji. To znamená, že nevyužívají datových zdrojů jako jsou soubory nebo síť. [6] Například funkce, která čte čas nebo používá globální proměnné, není referenčně transparentní. Příkladem je funkce, která vrací aktuální datum.

Ve funkcionálním programování jsou funkce často navrženy tak, aby byly referenčně transparentní. Například funkce, která vrací součet dvou čísel, je referenčně transparentní, protože její výsledek závisí pouze na dvou vstupních číslech.

```
x = 2 + 2
y = x + 1 -- x lze nahradit číslem 4 bez změny významu
```

Zdrojový kód 23 - Referenční transparentnost

3.9 First-class funkce

Princip funkcí první třídy hraje velkou roli ve funkcionálním programování. Umožňují psát modulární a znovupoužitelný kód. Díky tomuto principu lze s funkcemi zacházet jako s hodnotami. Lze je přiřazovat proměnným, předávat jako argumenty, vracet z jiných funkcí nebo ukládat do datových struktur. Díky tomu podporují tvorbu funkcí vyššího řádu a implementací technik jako *map*, *filter* a *reduce*. [6]

Důležitou vlastností je funkční kompozice, kdy lze menší funkce kombinovat do složitějších operací. Funkce první třídy také zpřístupňují použití currying a částečné aplikace, které usnadňují tvorbu specializovaných variant obecných funkcí a zvyšují modularitu kódu. [17]

3.10 Anonymní funkce

Také nazývané lambda funkce nebo lambda výrazy, jsou funkce, které nemají přiřazený název. Jsou často používány pro jednorázové operace nebo pro funkce, které jsou definovány přímo na místě, kde je potřeba je použít, pro krátkodobé použití, nebo pro málo opakované operace. Nevyžadují samostatnou definici pomocí názvu. [6]

Anonymní funkce jsou běžně používány jako argumenty funkcí vyššího řádu nebo pro konstrukci výsledku funkce vyššího řádu, které vrací funkci. Jsou také vhodné jako zpětná volání, obsluhu událostí nebo jednorázové výpočty. Lze díky nim definovat malé, inline funkce, které nezahlcují kód definovanými názvy.

Používání anonymních funkcí má i svá úskalí. Kód s mnoha anonymními funkcemi, zejména rozsáhlými, může být hůře čitelný a srozumitelný. Absence popisného názvu snižuje sebedokumentaci, což může ztížit pochopení účelu funkce. Anonymní funkce mohou způsobovat problémy při ladění, kdy nepojmenovaná funkce může být hůře identifikovatelná. Jejich omezená znovupoužitelnost vede k duplikacím, což zvyšuje nekonzistenci kódu i jeho údržbu. Ačkoli rozdíl výkonu mezi anonymními a pojmenovanými funkcemi může být v některých případech zanedbatelný, nadměrné vytváření objektů funkcí může mít vliv na výkon.

3.11 Částečná aplikace

V Haskellu je funkce chápána jako matematický předpis, který mapuje vstupní hodnoty na výstupní. Výše zmíněné předpisy funkcí popisovaly funkce, které přijímaly jeden argument a vracely jednu hodnotu. Tento zápis však zároveň implikuje hlubší princip.

```
numberSum :: Int -> Int -> Int
numberSum a b = a + b

numberSum :: Int -> (Int -> Int)
```

Zdrojový kód 24 - Funkce s více vstupními argumenty

Funkce v Haskellu jsou totiž koncipovány jako funkce jednoho argumentu. To znamená, že i funkce, které pracují se dvěma nebo více argumenty, jsou ve skutečnosti definovány jako posloupnost funkcí, z nichž každá přijímá právě jeden argument a vrací další funkci. Funkce *numberSum* tedy přijímá jeden vstupní parametr typu *Int* a vrací funkci $Int \rightarrow Int$. Tento způsob zápisu je důsledkem, v Haskellu implicitního, currying. Díky tomu lze funkci aplikovat

na první argument a výsledkem bude nová funkce, která očekává zbývající argumenty. Tato technika se nazývá partial application (částečná aplikace).

```
numberSum :: Int -> Int -> Int
numberSum a b = a + b

numberSumWithFive :: Int -> Int
numberSumWithFive = numberSum 5

result = numberSumWithFive 15 -- 20
```

Zdrojový kód 25 - Funkce s více argumenty a partial application

Partial application umožňuje velmi přirozeným způsobem vytvářet specializované verze obecnějších funkcí. V příkladu byla funkce *numberSum* použita pro částečnou aplikaci ve funkci *numberSumWithFive*. Protože je funkci *numberSum* předán pouze první vstupní parametr, vrátí funkci, která přijímá parametr druhý.

3.12 Currying

Currying je technika funkcionálního programování, která převádí funkce s více parametry na řetězec funkcí. Každá funkce v tomto řetězci přijímá pouze část všech argumentů. Takto rozdělené funkce se pak nazývají curried funkce.

Klíčová myšlenka curryingu spočívá v použití sekvence funkcí na místo klasického přístupu, kde funkce přijímá všechny své parametry najednou. Každá z jednotlivých funkcí má jeden nebo více argumentů. S těmito argumenty je provedena potřebná operace. Návratovou hodnotou je funkce, která očekává zbytek parametrů. To znamená, že lze „zavázat“ některé argumenty předem, aniž by bylo nutné definovat novou funkci od začátku. Tento přístup podporuje nejen efektivnější kód, ale také lepší přehlednost, protože odstraňuje opakování stejných argumentů. [17]

Tato technika je přirozeně propojena s dalšími technikami funkcionálního programování, jako jsou funkce vyššího řádu, immutability a lazy evaluation. Díky částečné aplikaci funkcí umožňuje tvorbu specializovaných funkcí a podporuje jejich znovupoužitelnost. Currying se stává mimořádně užitečným při práci v prostředích, kde je potřeba maximalizovat opakované použití a modulárnost kódu.

Ikdyž to není na první pohled zřejmé, Haskell implicitně „transformuje“ každou funkci na curried funkci. Haskell například dříve uvedenou funkci pro sčítání dvou čísel, tedy funkci

se dvěma parametry, interně interpretuje jako funkci, která přijímá první argument a vrací novou funkci čekající na druhý argument. [16]

Přestože je tato technika v Haskellu implicitně implementována, její principy lze aplikovat i v jiných paradigmatech, včetně objektově orientovaných jazyků. V jazyce Java není currying součástí jazyka, ale lze jej simulovat pomocí lambda funkcí nebo funkcionálních rozhraní. Syntax může být ale pro složitější funkce nepřehledná.

3.12.1 Vlastnosti

Currying má mnoho výhod, ale jeho použití může mít i určité nevýhody, zejména v prostředích, která nejsou optimalizována pro funkcionální paradigmata. Velkou výhodou je čitelnost a pochopitelnost dílčích operací. Umožňuje také kombinování operací, čímž podporuje techniku funkcí vyššího řádu. Další výhodou může být tvorba specializovaných funkcí bez nutnosti měnit původní logiku. Currying usnadňuje práci v prostředích, kde je potřeba přizpůsobit stávající funkce na nové požadavky.

Výhoda čitelnosti a pochopitelnosti může při výrazně vysokém rozdělení funkcí vést spíše k nečitelnosti nebo neintuitivnosti kódu, zejména pro vývojáře neobeznámené s funkcionálním přístupem. Další nevýhodou je skutečnost, že každé rozdělení funkce na menší části vede k vytvoření nové funkce. To může zvýšit paměťové nároky aplikace, zejména při práci s velkým množstvím curried funkcí. Ve výkonnostně kritických aplikacích může být currying méně efektivní než optimalizované imperativní přístupy.

3.12.2 Použití

Currying lze využít zejména v situacích, kdy je za potřebí opakovaně volat funkci s parametry, které se mění jen z části. Například v prostředí, kde je potřeba opakovaně aplikovat matematickou operaci na konkrétní hodnotu, umožňuje currying vytvořit funkci, která má tuto hodnotu „přednastavenou“. Tím se snižuje duplicita kódu a jeho čitelnost.

Currying si lze představit na jednoduché funkci, která sčítá dvě čísla. Tuto funkci lze díky curryingu převést na funkci, která bude druhé číslo vždy sčítat s číslem 10. Tímto způsobem se lze vyhnout opakovanému předávání čísla 10 jako parametru funkce.

```
add a b = a + b
add10 = add 10 -- díky currying lze add použít pouze s jedním parametrem
result = add10 5 -- 15
```

Zdrojový kód 26 - Currying a částečná aplikace

3.13 Pattern Matching

V příkladech s funkcemi bylo implicitně využíváno tzv. pattern matching. Do češtiny volně přeloženo jako porovnávání vzorů, poskytuje expresivní a deklarativní způsob práce s datovými strukturami pomocí porovnávání jejich tvarů a získávání informací. Porovnávání vzorů zahrnuje kontrolu hodnoty podle vzoru a provádění akcí na základě shody. [6] Na rozdíl od jednoduchých podmíněných kontrol umožňuje rozložit datové struktury za účelem získání hodnot, rozhodovat při výběru větví kódu na základě struktury dat a zachytávat neshodné případy.

```
describe [] = "empty" -- v případě prázdné kolekce na vstupu
describe (x:_) = "non-empty" -- v ostatních případech
```

Zdrojový kód 27 - Pattern matching

Pattern matching je často používán pro dekompozici komplexních datových typů jako jsou seznamy, neměnné listy nebo vytvořené objekty, kdy extrahuje potřebné informace pro další zpracování. Mezi další případy užití patří také rekurze, kdy pomáhají při definici základních případů pro její ukončení a samotné kroky rekurzivní funkce. Porovnávání může zjednodušit různé tvary při transformaci dat stručným zpracováním různých případů (jednotlivých hodnot nebo vnořených struktur). V neposlední řadě jej lze použít pro zpracování různých výstupních (chybových) stavů a poskytovat potřebné odpovědi.

```
sum [] = 0
sum (x:xs) = x + sum xs
 -- jakmile se výpočet dostane k prázdnému seznamu, tak se vrátí 0
```

Zdrojový kód 28 - Pattern matching v případě rekurze

Pattern matching nabízí několik výhod oproti tradičním podmínkovým výrazům jako *if-else* a *case-of*. Tyto benefity jsou nejvíce znatelné při práci se složitými datovými strukturami, kontrole typů nebo mnohonásobném podmíněném větvení programu. Ve srovnání s vnořenými příkazy *if-else* poskytuje pattern matching deklarativnější a přehlednější syntaxi. Dokáže eliminovat opakující se kód spojením kontroly podmínek a extrakci dat do jediné operace. Přidávání nových vzorů je přímočaré, díky čemuž je porovnávání snadněji rozšiřitelné. Komplexní zpracování případů zajišťuje, že jsou explicitně řešeny všechny možné scénáře, což snižuje pravděpodobnost výskytu chyb z přehlédnutých podmínek. Porovnávání vzorů ladí s dalšími paradigmaty funkcionálního programování, protože umožňuje neměnnost a deklarativní logiku.

3.13.1 Copattern matching

Pattern matching se používá hlavně v případě neměnných dat. U nekonečných datových typů však neexistuje ekvivalentní nástroj. Proto byl zaveden koncept copattern matching. Namísto destrukce existujících dat a porovnávání hodnot se vzory se používá pro definici nekonečných struktur. Porovnávají se evaluační kontexty s copatterny. Copattern lze chápat jako schéma definice pro nekonečné datové typy. Jsou užitečné pro práci s tzv. černými skříňkami reprezentujícími nekonečné objekty, jako jsou funkce a proudy (streams). Spojení copatternu s jeho výslednou hodnotou se nazývá covering observation. [18]

3.14 Funkce vyššího řádu

Funkce vyššího řádu neboli higher-order functions, jsou takové funkce, které mají jako vstupní parametr funkci nebo které vrací funkci jako návratovou hodnotu. Označují se i jako funkcionály. Ztělesňují princip funkcí prvního řádu. To znamená, že se chovají k funkcím jako k vysoko postaveným lidem. Funkce mohou být přiřazeny k proměnné, předány jako argumenty nebo mohou být návratovými hodnotami. [6]

3.14.1 Vlastnosti

Funkce vyššího řádu mohou přijímat funkci jako argument. To umožňuje delegaci chování programu předáváním jeho logiky. Funkce také mohou vracet funkce, dovolující částečnou aplikaci a kompozici funkcí. Toho se často využívá v currying. [5] Funkce vyššího řádu usnadňují proces kompozice funkcí. Umožňují řetězení funkcí, což vede k jasně popsaným a čitelným strukturám kódu.

Jedním z úskalí využívání funkcí vyššího řádu může být snížená čitelnost při velkém počtu zanořených funkcí nebo také jejich nadměrné užívání. Zřetězené volání funkcí může také představovat problém při jejich ladění a testování. Pro jednoduché iterace mohou být klasické cykly přehlednější a efektivnější než vzory jako *map* nebo *reduce*. [1]

3.14.2 Použití

Umožňují využití abstrakcí nad akcemi, namísto pouhých hodnot. Často se využívají k vytváření opakovaně použitelných abstrakcí, které mohou zjednodušit složitý kód a usnadnit jeho pochopení.

Používají se v různých vzorech funkcionálního programování jako jsou transformace a mapování. Při těchto akcích jsou na kolekci hodnot aplikovány různé úpravy. Tento přístup je klíčový při zpracování velkého objemu dat. Jsou rovněž vhodné při provádění redukcí

a agregací dat. Tyto operace dokážou redukovat kolekci na jedinou akumulovanou hodnotu, což je běžné například při sumarizaci dat nebo funkcionálních iteracích.

Dalším důležitým vzorem je currying a částečná aplikace funkcí, které transformují víceparametrové funkce do sekvence jednoparametrových funkcí. Tento přístup zvyšuje flexibilitu při opětovném použití a skládaní funkcí.

```
map (\x -> x * 2) [1, 2, 3] -- [2, 4, 6]
```

Zdrojový kód 29 - Příklad aplikace funkce vyššího řádu – map a lambda funkce

Funkce vyššího řádu umožňují přijímat funkce jako vstup nebo vracet funkce jako výstup. Díky nim lze vytvářet obecnější a flexibilnější abstrakce, které mohou výrazně zjednodušit kód.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

result = applyTwice (+ 1) 5 -- 7
```

Zdrojový kód 30 - Funkce vyššího řádu – předání funkce jako parametru

Funkce *applyTwice* přijímá funkci *f* (jejím vstupem i výstupem je hodnota typu *a*) a hodnotu typu *a*. Výstupem je pak hodnota, která vznikne dvojnásobným použitím vstupní funkce na vstupní hodnotu *x*. Je důležité zmínit, že v definici funkce se nepracuje s konkrétním datovým typem, jako je například *Int* nebo *Bool*. Místo toho je použit obecný typový parametr *a*, který slouží jako zástupný symbol pro libovolný konkrétní typ.

V příkladu aplikace funkce *applyTwice* je použita jako vstupní funkce $(+1)$ na hodnotu 5. Funkce $(+1)$ je zkrácený zápis pro funkci, která přičítá k hodnotě číslo jedna. Jedná se o funkci typu $Int \rightarrow Int$. Tento zápis lze také využít pro operace odčítání, násobení, dělení, umocňování nebo porovnání.

```
a = (+ 3) 6             -- 9
b = (-) 10 5            -- 5, nelze použít (- 5), zápis pro záporné číslo
c = (* 7) 4             -- 28
d = (/ 3) 24            -- 8
e = (== 1) 1            -- True
f = (<= 100) 101        -- False
g = (&& True) False     -- False
```

Zdrojový kód 31 - Zkrácené verze jednoduchých operací

Nejznámějšími příklady funkcí vyššího řádu jsou funkce pro práci s listy – *map*, *filter* a *fold*. Tyto funkce jsou klíčovými nástroji pro práci s listy a umožňují elegantní manipulaci s daty. Představují možnosti, jakými lze funkce využít pro transformaci nebo agregaci seznamů a dalších datových struktur.

```
map :: (a -> b) -> [a] -> [b]
result = map (+ 1) [1, 2, 3] -- [2, 3, 4]
```

Zdrojový kód 32 - Funkce map

Funkce *map* přijímá jako vstupní parametry funkci $(a \rightarrow b)$ a list s hodnotami typu a. Výstupem je transformovaný vstupní list s hodnotami typu b. Transformací se rozumí aplikování vstupní funkce na každou z hodnot ve vstupním listu.

```
filter :: (a -> Bool) -> [a] -> [a]
filter even [1, 2, 3, 4, 5, 6] -- [2, 4, 6]
```

Zdrojový kód 33 - Funkce filter

Funkce *filter* přijímá jako vstupní parametry funkci $(a \rightarrow Bool)$ a list s hodnotami typu a. Výstupem je vyfiltrovaný vstupní list s hodnotami typu a. Vyfiltrováním se rozumí odebrání hodnot ze vstupního listu na základě podmínky danou vstupní funkcí.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (-) 0 [1, 2, 3, 4] -- ((((0 - 1) - 2) - 3) - 4) = -10

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (-) 0 [1, 2, 3, 4] -- (1 - (2 - (3 - (4 - 0)))) = -2
```

Zdrojový kód 34 - Funkce foldl a foldr

Funkce *fold* přijímá jako vstupní parametry binární funkci, počáteční hodnotu (tzv. akumulátor) a list hodnot. Binární funkce má dva vstupní parametry. Je zavolána společně s akumulátorem a hodnotou listu. Tím vzniká nová hodnota akumulátoru. Proces se opakuje, dokud nezůstane pouze akumulátor, který je pak výstupní hodnotou. Funkce *foldl* „skládá“ list zleva, *foldr* zprava.

```haskell
sortNumbers :: [Int] -> [Int]
sortNumbers xs = sortBy (comparing id)

smallest :: [Int] -> Int
smallest xs = head (sortNumbers xs)

largest :: [Int] -> Int
largest xs = last (sortNumbers xs)
```

Zdrojový kód 35 - Funkce sortBy a comparing

K řazení kolekcí se v jazyce Haskell používá funkce vyššího řádu *sortBy*. V příkladu funkce *sortNumbers* řadí pole čísel vzestupně. To je zajištěno využitím funkce *sortBy* z modulu *Data.List* a funkce *comparing*. *Comparing* slouží k vytvoření srovnávací funkce ($a \rightarrow a \rightarrow Ordering$), která porovnává hodnoty kolekce. Funkce *id* je v tomto kontextu identická funkce, která jednoduše vrací svůj vstupní argument beze změny. Použití identické funkce znamená, že každé číslo pole je porovnáno s dalšími tak, jak je. Funkce *smallest* pak vrací nejmenší číslo. Použití *head* na seřazený seznam vrátí první, tedy nejmenší, číslo. Podobně funguje funkce *largest*, která používá funkci *last* ke zjištění největšího čísla seřazené kolekce.

3.15 Kompozice funkcí

Kompozice je princip ve funkcionálním programování, který umožňuje kombinovat funkce do nových funkcí. Tento princip dokáže vytvářet komplexní operace z jednoduchých funkcí. Jedná se o proces, při kterém se dvě nebo více funkcí kombinují do nové funkce. Tato nová funkce aplikuje vstupní hodnoty na první funkci a výstup z této funkce pak předává jako vstup do druhé funkce. [17] Tímto způsobem lez vytvářet komplexní funkce z jednoduchých komponent.

3.15.1 Vlastnosti

Kompozice funkcí zachovává vlastnosti jako asociativita a identita. Kompozice je asociativní, což znamená, že lze funkce skládat v libovolném pořadí bez změny výsledku. Identická funkce je funkce, která vrací svůj vstup jako výstup. Při skládání jakékoliv funkce s funkcí identickou se nezmění chování původní funkce. [6]

Skládání funkcí je deterministické díky principu čistých funkcí. Ty zajišťují, že pro stejný vstup vrátí vždy stejný výstup. Toto se stejným způsobem propisuje do složených funkcí. Mimo to těží i z principu funkcí prvního řádu.

Kompozice funkcí podporuje modulární programování. Místo používání dlouhých komplexních a nečitelných funkcí lze kombinovat menší, jednodušší funkce. To zvyšuje čitelnost a znovupoužitelnost kódu.

```
f x = x * 2
g x = x + 1
h = f . g -- f(g(x)) = 2 * (x + 1)
```

Zdrojový kód 36 - Kompozice funkcí

V jazyce Java lze funkční kompozici realizovat prostřednictvím rozhraní *Function*, které bylo zavedeno jako součást podpory funkcionálního stylu. Metoda *andThen* umožňuje sestavit novou funkci spojením dvou funkcí tak, že výstup první funkce slouží jako vstup funkce druhé. [13] Na rozdíl od jazyka Haskell, kde má kompozice pravou asociativitu, metoda *andThen* v Javě skládá funkce zleva doprava.

```
sumLengthOfXs :: [String] -> Int
sumLengthOfXs = foldr ((+) . length . filter (== 'x')) 0

result = sumLengthOfXs ["axe", "apex", "world", "suffix"] - 3
```

Zdrojový kód 37 - Kompozice více složitějších funkcí

Je možné skládat i více funkcí najednou. V uvedeném příkladu je pomocí kompozice definována funkce, která nejprve pomocí funkce *filter* extrahuje ze vstupního řetězce znaky *x*. Výsledek je předán funkci *length*, která spočítá jejich počet. Tento počet je následně předán funkci (+), která jej přičte k akumulované hodnotě v rámci funkce *foldr*.

V jazyce Haskell se často používá také operátor ($). Slouží jako pravostranná aplikace funkce. Tento operátor sice není kompoziční, ale velmi úzce s ní souvisí. Umožňuje přehlednější zápis výrazů, kde by jinak bylo nutné vnořovat závorky.

```
print (sum (map (+ 1) [1, 2, 3]))
print $ sum $ map (+ 1) [1, 2, 3]
```

Zdrojový kód 38 - Použití syntaktického operátoru ($)

Použití operátoru ($) má čistě syntaktický význam. Vyhodnocuje výraz vpravo a jeho výsledek předává funkci vlevo. Je pravostranně asociativní a má velmi nízkou prioritu, což umožňuje například přepsat výraz *print* výše na přehlednější tvar.

3.16 Rekurze

Rekurze je základním konceptem ve funkcionálním programování a informatiky. Rekurze je obecně volání sebe sama. V programování lze označit funkci jako rekurzivní v případě, že v jejím těle je volána ta samá funkce. Rekurzivní funkce většinou řeší menší instance problémů, dokud není dosaženo hledaného výsledku. Ukázka níže řeší výpočet faktoriálu čísla pomocí rekurze. [16]

```
fact 0 = 1
fact n = n * fact (n - 1)
 -- 5 * (fact 4 * (fact 3 * (fact 2 * (fact 1 * (fact 0)))))
```

Zdrojový kód 39 - Příklad rekurze – faktoriál

Funkcionální jazyky nepodporují konstrukce jako cykly. Proto rekurze ve funkcionálním programování nahrazuje tyto tradiční iterační konstrukce pro provádění opakujících se úloh. Rekurzivní řešení jsou často stručnější a srozumitelnější pro problémy zahrnující hierarchické nebo opakující se struktury. Jsou obzvláště účinné při procházení stromů a grafů. Mezi další možné oblasti, kde je vhodné použití rekurze, patří algoritmy rozděl a panuj, například quicksort nebo mergesort.

Nevýhoda použití rekurze vyplývá ze způsobu, jakým pracuje se zásobníkem. Každé nové rekurzivní volání funkce přidá nový rámec do zásobníku paměti. To může vést k vysoké paměťové náročnosti při hlubokých rekurzích. Bez vhodné podmínky pro ukončení rekruze nebo při velkých vstupech může rekurze způsobit přetečení zásobníku. Pokud nejsou řešení optimalizována technikami, jako je memoizace nebo tail-call, mohou být některá rekurzivní řešení pomalejší než ta iterativní.

V jazycích, které podporují optimalizaci pomocí tail-call (koncového volání), může být rekurze stejně efektivní jako iterace. Tzv. tail-recursive funkce opakovaně používají stejný rámec zásobníku, čímž odpadá režie vytváření nových rámců pro každé volání a tím se snižuje paměťová náročnost. [6]

Některé problémy jsou přirozeně rekurzivní a vyžadují udržování zásobníku. Jedná se především o algoritmy pro procházení stromů nebo řazení seznamů. Jejich iterativní implementace často vyžadují ruční správu zásobníku, což může být méně efektivní a náchylnější k chybám. Příkladem může být procházení binárního stromu, které při iterativním řešení vyžaduje pomocný zásobník.

```haskell
sumUpTo :: Int -> Int
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n - 1)

result = sumUpTo 3 -- 6
```

Zdrojový kód 40 - Součet čísel 1 až n, rekurzivně

Příklad výpočtu součtu čísel od 1 do n pomocí rekurze ukazuje, jak lze využít rekurzivní volání k postupnému snižování problému, dokud se nedosáhne základního případu.

```haskell
sumUpTo 3
= 3 + sumUpTo 2
= 3 + (2 + sumUpTo 1)
= 3 + (2 + (1 + sumUpTo 0)) -- dosaženo základního případu
= 3 + (2 + (1 + 0))
= 3 + (2 + 1)
= 3 + 3
= 6
```

Zdrojový kód 41 - Součet čísel 1 až n, rekurzivně (explicitně)

Rozepsaná verze funkce *sumUpTo* ukazuje, jak postupně rekurze funguje. Funkce *sumUpTo* přičítá aktuální číslo *n* k výsledku volání opět funkce *sumUpTo*, ale čísla $n - 1$. Tato sekvence se opakuje tak dlouho, dokud *n* není nula. Díky případu pro nulu se tím výpočet uzavře poslední operací, a to přičtením čísla nula.

Je důležité si uvědomit, že reálný výpočet probíhá zevnitř, tedy od základního případu směrem ven. Nejprve se vyhodnotí *sumUpTo 0* a teprve poté se jednotlivé hodnoty postupně vracejí a skládají zpět. Problém se rozloží na podproblémy a následně jsou postupně vyhodnoceny.

Java oproti Haskellu postrádá optimalizaci TCO (Tail Call Optimalization), takže hluboká rekurze může vést k přetečení zásobníku. [15] Proto se v praxi častěji používají iterace, které využívají klasické *for* nebo *while* smyčky. Jsou jednodušší na implementaci a efektivnější z hlediska paměťového využití.

```haskell
fibonacci :: Int -> [Int]
fibonacci n = fib n 0 1
  where
    fib 0 _ _ = []
    fib n a b = a : fib (n - 1) b (a + b)

result = fibonacci 5 -- [0,1,1,2,3]
```

Zdrojový kód 42 - Fibonacciho posloupnost, rekurzivně

Fibonacciho posloupnost je klasickým příkladem rekurzivního výpočtu, kde každé číslo vzniká jako součet dvou předchozích. V této implementaci je použita rekurze s pomocnými parametry (akumulátory), které si při každém volání předávají aktuální a předchozí hodnotu.

Funkce *fib* přijímá počet zbývajících členů a dvě čísla, která představují aktuální a následující členy posloupnosti. Při každém kroku se vloží aktuální číslo jako „ocas“ do výsledného seznamu a pokračuje se rekurzí s posunutými hodnotami.

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  quicksort [y | y <- xs, y <= x]
    ++ [x]
    ++ quicksort [y | y <- xs, y > x]

result = quicksort [3, 1, 4, 7, 1] -- [1,1,3,4,7]
```

Zdrojový kód 43 - Quicksort, rekurzivně

Při řešení složitějších problémů pomocí rekurze se často naráží na úkoly, které zahrnují více než jeden rekurzivní výpočet. Příkladem může být implementace algoritmu quicksort, který využívá principu „rozděl a panuj.“ Tento algoritmus rozděluje seznam na menší podseznamy podle pivotního prvku a rekurzivně je třídí, dokud není seznam seřazený.

V ukázkové implementaci funkce *quicksort* definuje případ pro prázdný seznam. Tento případ je také ukončovací podmínkou rekurze. Pokud seznam není prázdný, první prvek seznamu, tedy *x*, se stává pivotem. Zbytek seznamu, v tomto případě *xs*, je následně rozdělen na dvě části. Jedna obsahuje všechny prvky menší nebo rovné pivotu, ta druhá prvky větší než pivot. Každá z těchto dvou částí je poté opět rekurzivně rozdělena na dvě části. Výsledkem celé funkce je seřazený seznam od nejmenšího po největší číslo. Efektivita algoritmu záleží na výběru pivotního čísla pro každý podseznam.

3.17 Líné vyhodnocování

Líné vyhodnocování neboli lazy evaluation ve funkcionálním programování zajišťuje opoždění nebo pozastavení výpočtu výrazu až do doby, kdy je tento výpočet potřeba. Liší se tím od dvou dalších možností vyhodnocování, eager a strict, kdy jsou výrazy vypočteny ihned po přiřazení do proměnné. [6] Haskell jej má implicitně zabudovaný. V jiných jazycích lze explicitně vyhodnocovat výrazy se zpožděním pomocí speciálních funkcí.

Z informace, že můžou být výrazy vypočteny až v době, kdy se s nimi musí pracovat, vyplývá, že se díky lazy evaluation lze vyhnout zbytečným výpočtům. To může vést k možným vylepšením výkonu. [6] Jakmile je výraz vypočten, jeho výsledek je většinou uložen do paměti. Tím lze předejít k opakovanému vyhodnocování. Líné vyhodnocování také umožňuje definovat potenciálně nekonečné datové struktury, protože může pracovat pouze s jejími potřebnými částmi.

Lazy evaluation ale může přinést i několik problému, které mohou zkomplikovat vývoj a údržbu programu. Ve funkcionálním programování se často spoléhá na změny stavu programu. Pokud je výraz vyhodnocen se zpožděním a mezitím se změnil stav, na kterém tento výpočet závisí, může to vést k nesprávným výsledkům. Líné vyhodnocení může také přinést problémy v oblasti ladění, kdy opožděné vyhodnocení vyvolá výjimku v jiném kontextu. Přináší s sebou i režijní náklady spojené s evidencí, které výrazy již byly vyhodnoceny a které ještě čekají na zpracování. V situacích, kde jsou operace jednoduché a vykonávají se často, může tato režie způsobit snížení výkonu.

Problémy s výkonem lze řešit odebráním lazy evaluation pro výrazy, kde to není potřeba, nebo které by způsobily velkou režii. Haskell například dokáže generovat striktní vyhodnocování pro cykly, kde by líné vytvářelo problémy. Dalšími optimalizačními metodami jsou inlining, který při kompilaci nahrazuje volání funkce jejím tělem. A unboxing, který řeší optimalizaci při přístupu k datům. Rozbaluje hodnotu v objektu do její primitivní podoby, což umožňuje přímý přístup bez režie objektových referencí. [16]

Optimalizaci lze řešit již zmíněným ukládáním vypočtených hodnot. Na další volání funkce nebo operace je odpovězeno uloženou hodnotou. To redukuje opakované vyhodnocování a tím i režii. Další možností redukce režie je možnost sdílení operací a uložených hodnot.

4 POKROČILÉ KONCEPTY FP

4.1 Funktory

Funktor je jednou ze základních abstrakcí ve funkcionálním programování, která umožňuje pracovat s hodnotami zabalenými v určitém kontextu, aniž by bylo nutné tento kontext měnit. Představují strukturu, která podporuje operaci aplikace funkcí na hodnoty uvnitř této struktury. Funktory umožňují zachovat obecnost a modularitu v práci s různými datovými typy tím, že poskytují jednotný mechanismus pro transformaci jejich obsahu. [6]

Funktor je typový konstruktor spárovaný s mapovací funkcí (často nazývanou *fmap*), která transformuje hodnoty v rámci kontextu F, aniž by změnila jeho strukturu. Formálně lze funktor chápat jako objekt, který definuje operaci mapování hodnoty typu A na hodnotu typu B v rámci kontextu F. [16] Tato operace musí splňovat určité podmínky. Operace mapování musí být definována tak, aby umožňovala aplikaci libovolné funkce na hodnotu, která je zabalená v dané struktuře. Zároveň musí zachovávat strukturu této obálky. [10]

Každý funktor musí také splňovat dva základní axiomy. [11]

- První, $F(id_A) = id_{F(A)}$, říká, že pokud bude na hodnoty uvnitř funktoru aplikována identitní funkce, výsledná struktura se nezmění.
- Druhým je kompozice, $F(g \circ f) = F(g) \circ F(f)$, která stanovuje, že aplikace složené funkce na hodnoty uvnitř funktoru musí být ekvivalentní aplikaci jednotlivých funkcí v sekvenci.

Tyto zákony zajišťují konzistentní chování funktorů a umožňují jejich bezpečné použití v různých kontextech.

4.1.1 Použití

Funktory se často vyskytují v běžných programovacích strukturách, jako jsou seznamy, volitelné hodnoty nebo výsledky výpočtů, které mohou skončit chybou. Ve funkcionálním programování jsou široce využívány, protože poskytují elegantní způsob, jak zacházet s operacemi nad složitějšími datovými strukturami bez nutnosti explicitního rozbalování a manipulace s jejich vnitřním stavem. [16]

Díky tomu přispívají k vyšší expresivitě a bezpečnosti programového kódu, protože umožňují aplikovat funkce na hodnoty přímo, aniž by bylo nutné se starat o podrobnosti implementace

dané struktury. V oblasti návrhu softwaru jsou funktory významné zejména pro jejich roli v obecnějších abstrakcích, jako jsou aplikativní funktory a monády. [10]

Funktor je typová struktura, která umožňuje aplikaci funkce na hodnoty zabalené v kontextu, aniž by musel být tento kontext rozbalen nebo změněn. Pro práci s funktory v jazyce Haskell je klíčová operace *fmap*, která aplikuje danou funkci na hodnoty uvnitř této struktury. [16]

```
incrementAll :: [Int] -> [Int]
incrementAll = fmap (+ 1)

result = incrementAll [1, 2, 3] -- [2, 3, 4]
```

Zdrojový kód 44 - Funktory a aplikace základní funkce fmap

V příkladu je seznam *[Int]* konkrétním případem funktoru. V Haskellu je funktor reprezentován typovou třídou *Functor*, která definuje operaci *fmap*. Funkce *incrementAll* pak *fmap* používá pro aplikaci funkce (+1) na každý prvek kolekce.

```
data Student = Student {name :: String, grade :: Int}

studentNamesToUpper :: [Student] -> [Student]
studentNamesToUpper = fmap (\s -> s {name = map toUpper (name s)})
```

Zdrojový kód 45 - Funktory a aplikace fmap na kolekci ADT

Funkce *studentNamesToUpper* ilustruje použití funkce *fmap* na seznam studentů, kde je každý student reprezentován datovým typem *Student*. Cílem funkce je převést jména všech studentů na velká písmena. *Fmap* umožňuje aplikovat funkci na každý prvek seznamu, aniž by změnil samotnou strukturu. Vstupní funkcí transformace je lamda funkce *\s*, jejímž úkolem je vrátit transformovanou instanci studenta. Vnitřní funkce map aplikuje funkci *toUpper* na každý znak celého jména, čímž převede všechny znaky na velké.

```
data Box = Box {number :: Int}

instance Functor Box where
  fmap f (Box x) = Box (f x)

fmap (+ 1) (Box 10) -- Box 11
```

Zdrojový kód 46 - Vlastní definice instance Functor

Haskell umožňuje definovat i vlastní datové typy jako instance typové třídy *Functor*, pokud jejich struktura definuje operaci *fmap*. V příkladu je použit jednoduchý parametrizovaný datový

typ *Box*, který obaluje jednu hodnotu. Následně je pro tento typ implementována instance *Functor*, v níž je funkce *fmap*. *Fmap* pak aplikuje danou funkci na vnitřní hodnotu. Po aplikaci zůstane výsledek zabalen v konstrukci *Box*.

4.1.2 Aplikativní funktory

Aplikativní funktory představují rozšíření základních funktorů. Zatímco běžné funktory poskytují operaci pro aplikaci funkce na hodnotu zabalenou v nějaké struktuře, aplikativní funktor umožňuje aplikovat funkci, která je sama zabalená ve struktuře, na jinou hodnotu, která je rovněž ve stejném nebo jiném kontextu. [10] Poskytuje tedy schopnost provádět operace mezi více strukturami bez nutnosti explicitního zpracování. To umožňuje efektivní práci s výpočty v kontextech, kde je třeba kombinovat více nezávislých hodnot a zároveň zachovat původní strukturu výpočtu.

Formálně aplikativní funktor definuje dvě základní operace. [11]

- Operaci zabalení, $pure :: a \rightarrow F\ a$, která vloží hodnotu do kontextu F
- Operaci aplikace, $<*> :: F(a \rightarrow b) \rightarrow F\ a \rightarrow F\ b$, která aplikuje zabalenou funkci na zabalenou hodnotu

```
functions :: [Int -> Int]
functions = [(+ 1), (* 2)]

values :: [Int]
values = [10, 20]

result :: [Int]
result = functions <*> values -- [11, 21, 20, 40]
```

Zdrojový kód 47 - Příklad aplikativního funktoru

Jak již bylo zmíněno, list představuje jednu ze základních instancí funktoru. Pro demonstraci aplikativního funktoru byly funkce uložené do seznamu *functions* a aplikovány na hodnoty v seznamu *values*. Obě struktury jsou v kontextu listu. K aplikaci slouží operátor $<*>$, který aplikuje každou funkci na každou hodnotu. Výsledkem je opět list, který obsahuje všechny hodnoty po aplikaci funkcí. Operátor $<*>$ zde ilustruje schopnost aplikativního funktoru pracovat s více hodnotami v kontextu bez nutnosti explicitní iterace nebo rozbalování struktur.

```haskell
data Box a = Box a

instance Functor Box where
  fmap f (Box x) = Box (f x)

instance Applicative Box where
  pure x = Box x
  (Box f) <*> (Box x) = Box (f x)

result :: Box Int
result = pure (+5) <*> Box 3  -- Box 8
```

Zdrojový kód 48 - Vlastní definice instance aplikativního funktoru

Předchozí instance typu *Box*, která byla použita pro demonstraci funktoru, byla rozšířena o instanci typu *Applicative*. Implementace *fmap* zůstává zachována a zajišťuje aplikaci funkce na hodnotu uvnitř *Box*. V rámci aplikativního funktoru je dále definována funkce *pure*, která vkládá hodnotu do kontextu, a operátor <∗>, který umožňuje aplikaci funkce zabalené v *Box* na hodnotu v jiném *Box*. Hodnota *result* je pak výsledek aplikace funkce (+5) na hodnotu 3 v rámci definovaného kontextu.

Aplikativní funktory jsou základním stavebním kamenem pro další pokročilé koncepty, jako jsou monády. Zajišťují konzistentní způsob, jakým lze kombinovat různé výpočetní kontexty, a umožňují elegantní formulaci složitějších operací bez nutnosti ruční správy vnitřní struktury dat. Jsou široce využívány v knihovnách funkcionálních jazyků i v moderních objektově orientovaných jazycích, které implementují funkcionální prvky. [11]

4.2 Monády

Monády představují další pokročilou abstrakci ve funkcionálním programování, která navazuje na aplikativní funktory. Umožňuje pracovat s výpočty v různých kontextech a zároveň zachovat čistotu funkcí. Jsou zobecněním aplikativních funktorů a poskytují mechanismus pro sekvenční zřetězení operací, přičemž spravují vedlejší efekty, výpočetní kontext nebo jiné aspekty výpočtu. [6] Monáda je struktura, která rozšiřuje aplikativní funktor o operaci *bind* (≫=). Ta umožňuje aplikaci funkce vracející hodnotu v monadickém kontextu na hodnotu již uvnitř tohoto kontextu. [16]

Monády musí splňovat tři základní zákony. [10]

- Levá identita, $return(a) \gg= f \equiv f(a)$, říká, že aplikace operace return na hodnotu a její následné zpracování pomocí operace *bind* je ekvivalentní přímé aplikaci zpracovávající funkce na hodnotu.
- Pravá identita, $m \gg= return \equiv m$, říká, že pokud je na hodnotu v monadickém kontextu aplikována operace *bind* s funkcí return, výsledkem je hodnota v původním kontextu.
- Asociativita, $(m \gg= f) \gg= g \equiv m \gg= (x \mapsto f(x) \gg= g)$, říká, že sekvenční aplikace více operací *bind* musí být ekvivalentní jedné aplikaci *bind*, která využívá složené funkce.

Díky těmto vlastnostem lze monády považovat za nástroj pro řízení výpočetních kontextů. To znamená, že umožňují pracovat se strukturami, které zahrnují neúplné nebo chybové hodnoty, vedlejší efekty, proměnlivý stav nebo asynchronní výpočty, aniž by bylo nutné tyto aspekty explicitně řešit na každém kroku programu. [11] Existuje několik typů monád, z nichž každá má specifické vlastnosti a uplatnění.

Maybe

Maybe monáda řeší problém práce s hodnotami, které mohou nebo nemusí existovat. Používá se jako alternativa k *null*, *nil* či *None*, což pomáhá eliminovat časté chyby způsobené neexistujícími hodnotami. S touto monádou lze bezpečně skládat výpočty. Pokud se kdykoli v řetězci operací objeví neexistující hodnota, výpočet se zastaví a výsledkem je přímo tato neexistující hodnota. [16]

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing
safeDivide x y = Just (div x y)

result = safeDivide 10 2 -- Just 5
```

Zdrojový kód 49 - Bezpečné dělení s využitím monády Maybe

Either

Either monáda je podobná monádě *Maybe*, ale místo jedné speciální hodnoty umožňuje reprezentovat dvě možné cesty – úspěch a neúspěch. Typicky se používá k práci s chybami,

kde jedna větev obsahuje výsledek úspěšného výpočtu a druhá chybu. Díky tomu lze pracovat s výjimkami bez nutnosti využití imperativních struktur jako je například *try-catch*. [16]

```
data Error = DivisionByZero | UnknownError

safeDivide :: Int -> Int -> Either Error Int
safeDivide _ 0 = Left DivisionByZero
safeDivide x y = Right (div x y)

result = safeDivide 10 2 -- Right 5
```

Zdrojový kód 50 - Bezpečné dělení s využitím monády Either

List

List monáda umožňuje pracovat s výpočty, které mohou vracet více výsledků. Základní vlastností monády je aplikace funkce na každý prvek seznamu a spojit tak výsledky do jednoho nového seznamu. Je užitečná například při generování kombinací nebo při práci s výpočty, které mohou vracet více alternativních hodnot. V programovacích jazycích často souvisí s operacemi jako *map*, *flatMap* nebo *concatMap*. [11]

```
result = [1, 2] >>= \x -> [x, x * 10] -- [1, 10, 2, 20]
```

Zdrojový kód 51 - Spojení výsledků pomocí monády List

IO

Tato monáda řeší problém při práci s vedlejšími efekty, jako jsou čtení a zápis souborů, komunikace s databází nebo uživatelským vstupem. Ve funkcionálních jazycích poskytuje *IO* monáda způsob, jak tyto operace izolovat a bezpečně řídit. Operace v této monádě jsou vykonávány až v okamžiku, kdy jsou explicitně spuštěny. [6]

```
main :: IO ()
main = do
  putStrLn "Input:"
  input <- getLine
  putStrLn ("Output: " ++ input)
```

Zdrojový kód 52 - Funkce main s monádou IO

State

V prostředí funkcionálního programování umožňuje *State* monáda pracovat s proměnlivým stavem. Stav je explicitně předáván mezi výpočty, namísto uchovávání v globálních

proměnných nebo mutable objektech. Výhodou této monády je zachování čistoty funkcí a referenční transparentnosti. [11]

```
increment :: State Int Int
increment = do
  n <- get -- get vrací aktuální stav
  put (n + 1) -- put aktualizuje stav
  return n

result = runState increment 0 -- (0, 1) výsledek funkce, nový stav
```

Zdrojový kód 53 - Změna stavu pomocí monády State

Reader

Reader monáda poskytuje způsob, jak přistupovat k vnějšímu kontextu bez nutnosti jeho explicitního předávání do každé funkce. Používá se například při konfiguraci aplikací nebo při závislostech mezi různými částmi kódu. [11]

```
configuration :: Reader String String
configuration = do
  env <- ask -- ask získá hodnotu z prostředí
  return ("Asked: " ++ env)

result = runReader configuration "production" -- "Asked: production"
```

Zdrojový kód 54 - Načítání hodnot prostředí pomocí monády Reader

Writer

Writer monáda umožňuje kromě výpočtu akumulovat dodatečné informace, jako jsou logy, statistiky nebo ladící výstupy. Jsou vhodné například pro sledování průběhu výpočtů nebo shromažďování zpráv během vykonávání programu. [11]

```
logNumber :: Int -> Writer [String] Int
logNumber x = writer (x, ["Logged: " ++ show x])

result = runWriter $ do
  a <- logNumber 3
  b <- logNumber 5
  return (a + b) -- (8, ["Logged: 3", "Logged: 5"])
```

Zdrojový kód 55 - Logování průběhu výpočtu s využitím monády Writer

4.2.1 Monad transformers

Monad transformers zastupují mechanismus umožňující kombinovat efekty různých monád do jednoho výpočetního kontextu. Řeší problém monadického zanořování (tzv. monad stacking), kdy je nutné využít vlastnosti více monád současně. Jedná se o speciální druh monády, která obaluje jinou monádu a rozšiřuje tak její chování. Bez monad transformers je nutné hodnoty rozbalovat a znovu balit. [11] Příkladem může být použití monády *Maybe* pro práci s možnými chybějícími hodnotami a *IO* monády pro vstup/výstup.

Transformery mají typový parametr reprezentující základní monádu. Například *MaybeT IO* kombinuje monádu *Maybe* s monádou *IO*. Operace definované pro vnitřní monádu nejsou automaticky dostupné v té složené, protože transformery fungují pouze jako obalující konstrukce. Pro použití operací ve vnějším transformeru se využívá funkce *lift*, která „zvedne“ operace z vnitřní monády do vnější. [11]

```
getNonEmptyLine :: MaybeT IO String
getNonEmptyLine = do
  line <- liftIO getLine -- zvednutí IO funkce do MaybeT
  if null line
    then MaybeT $ return Nothing
    else return line

main :: IO ()
main = do
  putStrLn "Input:"
  result <- runMaybeT getNonEmptyLine
  case result of
    Nothing -> putStrLn "Nothing"
    Just input -> putStrLn $ "Output: " ++ input
```

Zdrojový kód 56 - Kontrola vstupních parametrů s využitím monad transformerů

MaybeT IO je transformer, který kombinuje chování monády *Maybe* s efekty monády *IO*. Pomocí funkce *liftIO* se standardní *IO* operace zvedají do kontextu *MaybeT IO*. Funkce *getNonEmptyLine* přečte vstup, ale pokud je řetězec prázdný, vrátí *Nothing*, čímž signalizuje neexistující hodnotu. V hlavní funkci *main* se výsledek rozbalí pomocí *runMaybeT* a podle hodnoty se vypíše odpovídající zpráva. Tímto způsobem se lze vyhnout složitému manuálnímu rozbalování a zabalování hodnot mezi různými kontexty.

5 APLIKACE FUNKCIONÁLNÍCH PRINCIPŮ

Praktická část navazuje na představené teoretické poznatky a zaměřuje se na konkrétní ukázku použití vybraných technik funkcionálního programování. Cílem je nejen ukázat samotné použití daných technik, ale i podrobně popsat jejich praktické důsledky a rozdíly v přístupu mezi funkcionálním a objektově orientovaným stylem. Pro všechny praktické příklady uvedené v této části byl vybrán Haskell jako čistě funkcionální jazyk a pro kapitolu následující byla vybrána Java jako objektově orientovaný programovací jazyk.

5.1 Zadání problému

Pro ukázky principů funkcionálního programování byl vybrán praktický příklad, který se zabývá datovou pipeline. Jejím úkolem je postupně transformovat vstupní data na požadovaná výstupní data. Zvolená úloha spočívá ve zpracování senzorových meteorologických dat uložených ve formátu CSV. Každý soubor reprezentuje výstup jednoho senzoru a obsahuje časovou řadu měření, která se skládá z časového razítka a naměřené hodnoty. Cílem aplikace je analyzovat tyto datové sady a vypočítat z nich vybrané statistiky. Výsledky těchto výpočtů jsou následně serializovány do formátu JSON. Program je navržen jako univerzální nástroj pro načtení konfiguračního souboru určující některé parametry programu a zpracování více souborů paralelně. Pro práci se soubory CSV byla použita knihovna *cassava* a pro práci se soubory JSON byla použita knihovna *aeson*.

5.2 Návrh datových struktur

Základní datové struktury aplikace byly navrženy tak, aby přesně odpovídaly povaze vstupních i výstupních data a zároveň podporovaly silnou typovou kontrolu a neměnnost. Definovány byly tři klíčové typy.

```
data Config = Config
  { outlierThreshold :: Double,
    aboveThreshold :: Double,
    outputFilePath :: FilePath
  }
  deriving (Show, GEN.Generic, AESON.FromJSON)

data SensorRecord = SensorRecord
  { timestamp :: TIME.UTCTime,
    value :: Double
  }
  deriving (Show)
```

```haskell
data SensorReport = SensorReport
  { name :: FilePath,
    average :: Maybe Double,
    averageMonthly :: [MonthlyAverage],
    outliers :: [SensorRecord],
    longestSequence :: Int
  }
  deriving (Show, GEN.Generic, AESON.ToJSON)

data MonthlyAverage = MonthlyAverage
  { month :: String,
    avg :: Maybe Double
  }
  deriving (Show, GEN.Generic, AESON.ToJSON)
```

Zdrojový kód 57 – Definice algebraických datových typů a použití deriving

Typ *Config* reprezentuje externě načítanou konfiguraci, která obsahuje např. práh pro detekci odlehlých hodnot. *SensorRecord* popisuje jednotlivé záznamy měření jako dvojici časové značky a hodnoty. Výsledky analýzy jsou uchovány ve struktuře *SensorReport*, která obsahuje vybrané výstupní statistiky. Jednou ze statistik je i průměr nebo sezónní index, které jsou díky možnosti nevypočitatelných statistik definovány jako *Maybe*, což umožňuje reprezentovat situace, kdy danou statistiku nelze z dostupných dat vypočítat.

Doplňkový typ *MonthlyAverage* byl vytvořen za účelem usnadnit práci se statistikou zabývající se měsíčními průměry. Pokud by tento typ nebyl nadefinován, ve výpočtech by se muselo využít například konstrukce tuple (n-tice) a v případě parsování *SensorReport* by bylo nutné ho explicitně parsovat také.

U všech datových typů je využito konstrukce *deriving*. Ta automaticky generuje instance běžných typových tříd, například Show. Tím se eliminuje nutnost explicitně tuto funkcionalitu implementovat. Datové typy jsou navrženy jako neměnné a plně popsané typovým systémem. Každá funkce pak pracuje s jasně definovanými vstupy a výstupy, což výrazně zvyšuje bezpečnost.

```haskell
instance CSV.FromNamedRecord SensorRecord where
  parseNamedRecord :: CSV.NamedRecord -> CSV.Parser SensorRecord
  parseNamedRecord r = do
    sT <- r CSV..: "timestamp"
    sV <- r CSV..: "value"
    pT <- parseTime sT
    return $ SensorRecord pT sV
```

```
instance AESON.ToJSON SensorRecord where
  toJSON (SensorRecord t v) =
    AESON.object
      [ "timestamp" AESON..= formatTime t,
        "value" AESON..= v ]
```

Zdrojový kód 58 – Definice instancí pro parsování typů

Pro správné načtení vstupních a výstupních datových formátů je nezbytné definovat příslušné instance typových tříd *FromNamedRecord* a *ToJSON*. Všechny instance specifikují, jak se mají konkrétní datové typy převádět z textové reprezentace a naopak. Využívá se knihoven, *aeson* a *cassava*, a ručně definovaných funkcí *parseTime* a *formatTime*. Definice těchto instanci zajišťují silnou typovou kontrolu.

```
timeFormat :: String
timeFormat = "%Y-%m-%dT%H:%M:%SZ"

parseTime :: String -> CSV.Parser TIME.UTCTime
parseTime = TIME.parseTimeM True TIME.defaultTimeLocale timeFormat

formatTime :: TIME.UTCTime -> String
formatTime = TIME.formatTime TIME.defaultTimeLocale timeFormat

formatTimeMonth :: TIME.UTCTime -> String
formatTimeMonth = TIME.formatTime TIME.defaultTimeLocale "%Y-%m"
```

Zdrojový kód 59 – Pomocné funkce pro parsování času

5.3 Načtení konfigurace

Pro načtení konfigurace ze souboru je v programu nadefinována funkce *loadConfig*. Funkce využívá monádu *IO* k bezpečnému zachycení vedlejších efektů. Operace čtení je díky tomu bezpečně a explicitně zakomponována do výpočtu, čímž zachovává čistotu zbytku programu.

```
loadConfig :: FilePath -> IO Config
loadConfig path =
  BSL.readFile path >>= \contents ->
    case AESON.decode contents of
      Just config -> return config
      Nothing -> error "Error reading configuration file"
```

Zdrojový kód 60 – Načtení konfigurace, monády IO a Maybe

Použitím operátoru *bind* se efektivně spojují jednotlivé kroky. Nejprve se načtou data pomocí funkce *readFile*, která se poté dekódují do typu *Config*. *Bind* umožňuje předat výsledek čtení jako vstup dekódovací funkce, což zajišťuje správné pořadí efektů.

Dekódování pomocí funkce *decode* z knihovny *aeson* pak ošetřuje akci díky monádě *Maybe*. Kontrola výstupu funkce je zajištěna konstrukcí *case-of*. V případě chyby při parsování konfiguračního souboru dojde k výjimce a ukončení programu.

5.4 Vyhledání zdrojových dat

Pro načtení všech vstupních souborů je zapotřebí projít zdrojovou složku a vyhledat správné soubory podle přípony. To zajišťuje funkce *loadFiles*, která, obdobně jako funkce *loadConfig*, pracuje s monádou *IO* a vrací seznam cest ke zdrojovým souborům.

```
loadFiles :: FilePath -> IO [FilePath]
loadFiles dir =
  DIR.listDirectory dir >>= \allFiles ->
    return [dir FS.</> f | f <- allFiles, FS.takeExtension f == ".csv"]
```

Zdrojový kód 61 – Vyhledání vstupních souborů, monáda IO

Funkce vrací seznam cest k souborům, který je vytvořen pomocí list comprehension. List bude vytvořen ze seznamu souborů v *allFiles*, které splňují podmínku přípony. Končené cesty v listu budou složeny z cesty ke složce a názvu souboru pomocí spojovací funkce </>.

5.5 Výpočet statistik

```
safeAvg :: [Double] -> Maybe Double
safeAvg [] = Nothing
safeAvg xs = Just (sum xs / fromIntegral (length xs))
```

Zdrojový kód 62 – Bezpečný výpočet průměru, monáda Maybe

Pro bezpečný výpočet průměrů byla definována funkce *safeAvg*, která pracuje s monádou *Maybe* a vrací *Nothing*, pokud je vstupní pole hodnot prázdné. V případě alespoň jedné hodnoty je průměr vypočítán klasickým způsobem, a to sečtením hodnot a vydělením jejich počtem.

```haskell
safeStandardDev :: [Double] -> Maybe Double
safeStandardDev [] = Nothing
safeStandardDev xs =
  case safeAvg xs of
    Nothing -> Nothing
    Just m ->
      let s = sum [(x - m) ^ (2 :: Int) | x <- xs]
       in Just (sqrt s / fromIntegral (length xs))
```

Zdrojový kód 63 – Bezpečný výpočet směrodatné odchylky, monáda Maybe

Funkce *safeStandardDev* bezpečně vypočítává směrodatnou odchylku. Pracuje s funkcí *safeAvg* a díky tomu musí definovat *case-of* na zpracování jejího výsledku. Při úspěšném výpočtu průměru spočítá součet čtverců odchylek každé hodnoty od průměru díky list comprehension a funkci *sum*. Tento list je pomocí *let-in* konstrukci uložen do proměnné *s* a použit ve výsledném výpočtu odchylky.

```haskell
computeAverage :: [SensorRecord] -> Maybe Double
computeAverage r = safeAvg (map value r)
```

Zdrojový kód 64 – Výpočet celkového průměru

Funkce demonstruje aplikaci funkce vyššího řádu, *map*, pro extrakci hodnot ze seznamů záznamů a jejich předání funkci *safeAvg*. Používá také interní vlastnost algebraických datových typů, které definují funkce pro extrakci hodnot instance. Využití *safeAvg* zabraňuje šíření výjimek či nulových referencí. Výstupem bude průměrná hodnota nebo *Nothing*, která je končeným výsledkem pro statistický průměr závěrečného reportu.

```haskell
computeAverageMonthly :: [SensorRecord] -> [MonthlyAverage]
computeAverageMonthly r =
  [ MonthlyAverage m (fmap (roundN 2) (safeAvg xs))
    | (m, xs) <- groupByMonth r ]
```

Zdrojový kód 65 – Výpočet měsíčních průměrů

Pro výpočet průměrů podle měsíce byla definována funkce *computeAverageMonthly*. Nejprve jsou vstupní záznamy seskupeny podle měsíce pomocí funkce *groupByMonth*, která vrací seznam dvojic *(měsíc, [hodnoty])*. Následně se pomocí list comprehension vytvoří výsledný seznam hodnot typu *MonthlyAverage*. Průměr je vypočítán funkcí *safeAvg*, která vrací hodnotu v kontextu *Maybe*. Zaokrouhlení na dvě desetinná místa je aplikováno pomocí *fmap* nad tímto obalem *Maybe*. Tím je zajištěno správné zpracování i v případě absence hodnot.

```
detectOutliers :: [SensorRecord] -> Double -> [SensorRecord]
detectOutliers recs thr
  | null xs = []
  | m == 0 = []
  | sd == 0 = []
  | otherwise = filter isOutlier recs
  where
    xs = map value recs
    m = MAYBE.fromMaybe 0 (safeAvg xs)
    sd = MAYBE.fromMaybe 0 (safeStandardDev xs)
    t = 100 * thr * sd

    isOutlier r = abs (value r - m) > t
```

Zdrojový kód 66 – Detekce odlehlých hodnot, využití konfigurace a filter

Funkce *detectOutliers* zajišťuje vytvoření listu odlehlých záznamů pomocí dříve definovaných funkcí pro průměr a směrodatnou odchylku. Funkce zachytává díky guards situace, kdy jsou vstupní list, průměr nebo směrodatná odchylka neplatné. Poslední větev využívá funkci vyššího řádu, *filter*, pro vytvoření seznamu, které splňují podmínku *isOutlier*. Všechny proměnné včetně podmínky jsou pak definované pomocí konstrukce *where*.

```
computeLongestAbove :: [SensorRecord] -> Double -> Int
computeLongestAbove recs thr = compute 0 0 recs
  where
    t = maximum (map value recs) * thr
    compute :: Int -> Int -> [SensorRecord] -> Int
    compute curr acc [] = fromIntegral (max curr acc)
    compute curr acc (r : rs)
      | value r > t = compute (curr + 1) acc rs
      | otherwise = compute 0 (max curr acc) rs
```

Zdrojový kód 67 – Rekurzivní výpočet statistické hodnoty

Funkce *computeLongestSequence* slouží k výpočtu nejdelší souvislé sekvence hodnot, které přesahují práh zadaný konfiguračním souborem. Práh je zadán relativně vůči maximální hodnotě v celém seznamu. Výpočet probíhá rekurzivně pomocí pomocné vnořené funkce *compute*, která nese dva akumulátory. Akumulátor *curr* pro aktuální délku právě probíhající sekvence a *acc* pro zatím nejdelší nalezenou sekvenci.

Seznam je zpracováván po prvcích, a pokud aktuální hodnota přesahuje vypočítaný práh *t*, aktuální délka sekvence se zvyšuje. Pokud hodnota podmínku nesplňuje, aktuální sekvence je

ukončena a akumulátor *acc* se aktualizuje, pokud byla právě ukončená sekvence delší než dosud nejdelší nalezená. Na konci je výsledkem maximální délka sekvence.

Rekurze zde zajišťuje efektivní jednorázové projití celého seznamu bez nutnosti dalších pomocných struktur nebo iterací. Použití pattern matchingu a guards přispívá k přehlednosti a čistotě zápisu. Takový přístup snižuje riziko vedlejších efektů, protože stav výpočtu je plně řízen pouze parametry rekurzivní funkce.

5.6 Zpracování dat

Pro zpracování dat byla implementována funkce *processDat*, která načte obsah, bezpečně rozparsuje jednotlivé řádky do struktury *SensorRecord*, vypíše chyby při parsování a následně vypočítá statistické údaje a detekuje odlehlé hodnoty. Výsledkem je sestavení reportu typu *SensorReport*, který shrnuje všechny relevantní výpočty a informace.

```
processData :: Config -> FilePath -> IO SensorReport
processData config file =
  BSL.readFile file >>= \records -> do
    let ls = BSLC.lines records
    case ls of
      [] -> error $ "Empty file: " ++ file
      (header : rows) -> do
        parsed <- mapM (parseLine header) rows

        let recs = EITHER.rights parsed
        let errs = EITHER.lefts parsed
        mapM_
          (\e -> putStrLn $ "Wrong line in " ++ file ++ ": " ++ e)
          errs

        let a = computeAverage recs
        let aM = computeAverageMonthly recs
        let oL = detectOutliers recs (outlierThreshold config)
        let lSA = computeLongestAbove recs (aboveThreshold config)
        return $ SensorReport file a aM oL lSA
```

Zdrojový kód 68 – Funkce pro zpracování dat, monády a funkce mapM

Pro implementaci je zvolena do-notace v monádě *IO*, která nahrazuje explicitní použití funkce *bind*. *Do*-notace zvyšuje čitelnost a přehlednost kódu tím, že umožňuje psát sekvenční monadické operace ve stylu imperativního jazyka, přičemž zachovává výhody funkcionálního přístupu, jako je bezpečná manipulace s vedlejšími efekty. Použití monadické funkce *mapM* umožňuje zpracování seznamů monadických akcí a odděluje čisté výpočty od vedlejších efektů.

```haskell
parseLine ::
  BSLC.ByteString ->
  BSLC.ByteString ->
  IO (Either String SensorRecord)
parseLine header line =
  case CSV.decodeByName (BSLC.unlines [header, line]) of
    Left err     -> return $ Left err
    Right (_, v) -> case VEC.toList v of
      [rec] -> return $ Right rec
      _     -> return $ Left "Unexpected number of records"
```

Zdrojový kód 69 – Funkce pro parsování řádků zdrojového souboru

Funkce *parseLine* přijímá jako vstup hlavičku CSV souboru a jeden řádek dat, přičemž umožňuje bezpečné parsování každého řádku do typu *SensorRecord*. Spojení hlavičky a řádku se pomocí funkce *decodeByName* z knihovny *cassava* spojí a data se dekódují. Konstrukce *case-of* rozhoduje o výstupu.

V případě selhání, *parseLine* vrací *String* s chybou. Pokud dekódování proběhne v pořádku, převede se vektor záznamů na seznam. Pokud je v seznamu právě jeden prvek (tedy jeden celý dekódovaný *SensorRecord*), tento prvek se vrátí jako výsledný záznam použitelný ke statistice.

5.7 Pomocné funkce

Pro výpočet měsíčních průměrů bylo nezbytné zpracovávat vstupní data podle časové složky, konkrétně seskupit záznamy měření podle měsíce jejich pořízení.

```haskell
groupByMonth :: [SensorRecord] -> [(String, [Double])]
groupByMonth rs =
  MAP.toList $
    MAP.fromListWith
      (++)
      [(formatTimeMonth (timestamp r), [value r]) | r <- rs]
```

Zdrojový kód 70 – Seskupení podle měsíce, využití typu Map

Funkce *groupByMonth* vytváří mapování mezi řetězcovou reprezentací měsíce a seznamem hodnot. Použitím funkce *fromListWith* se zajistí seskupení všech hodnot podle klíče, aniž by bylo nutné explicitně třídit či dělit seznamy ručně. Výstupem funkce je seznam dvojic, který je získán převodem výsledné mapy zpět na běžný seznam pomocí list comprehension.

5.8 Paralelní zpracování

Funkce *main* zajišťuje kompletní spuštění aplikace s využitím paralelního zpracování vstupních dat. Nejprve pomocí *getArgs* načte potřebné argumenty příkazové řádky. Vstupními argumenty jsou složka se zdrojovými daty a cesta ke konfiguračnímu souboru. Následně je načten obsah konfigurace a seznam všech vstupních souborů. Výsledkem je seznam reportů, který je serializován do formátu JSON a uložen do výstupního souboru definovaného v konfiguraci pomocí funkce *writeFile*.

```
main :: IO ()
main = do
  [folder, config] <- ENV.getArgs
  cfg <- loadConfig config
  files <- loadFiles folder
  reports <- ASYNC.mapConcurrently (processData cfg) files
  BSL.writeFile (outputFilePath cfg) (AESON.encode reports)
```

Zdrojový kód 71 – Funkce main, paralelní zpracování

Klíčovou části je využití funkce *mapConcurently*, která umožňuje současné zpracování více souborů v samostatných vláknech. Každý soubor je zpracován funkcí *processData*, čímž se výrazně zvyšuje efektivita při práci s větším množstvím dat. Použití paralelismu v této části zvyšuje výkon a škálovatelnost programu, což je důležité při analýze rozsáhlých datových sad. Celý běh je organizován v monádě *IO*, která zajišťuje provedení vedlejších efektů jako je čtení souborů, výpis chyb nebo zápis výsledků.

6 FP V JAZYCE JAVA

Funkcionální programování se v posledním desetiletí stalo nedílnou součástí vývoje i v tradičně objektově orientovaných jazycích. Jazyky jako Java nebo C# integrují funkcionální prvky do svých jazykových konstrukcí s cílem zlepšit expresivitu, bezpečnost a paralelizaci. [13] Java od verze 8 obsahuje lambda výrazy, funkcionální rozhraní a Stream API, které umožňují deklarativní manipulaci s daty a eliminaci vedlejších efektů. [14]

Funkcionální principy v OOP prostředí kladou důraz na neměnnost, čisté funkce, funkce vyššího řádu a jejich kompozici. V jazyce Java to znamená využití *final* proměnných, návrh metod bez vedlejších efektů a preferenci složení objektů před dědičností. [15] Výsledkem je kód, který je předvídatelnější, snadněji testovatelný a lépe škálovatelný v paralelním prostředí.

Pro hlubší pochopení možností a omezení funkcionálního programování v jazyce Java je vhodné analyzovat konkrétní příklad implementovaný třemi odlišnými způsoby. Každá z těchto implementací je následně srovnávána s analogickým řešením v jazyce Haskell, jak bylo popsáno v kapitole 5. Cílem této části je ukázat, jak rozdílné mohou být možnosti vyjádření funkcionálního stylu v jazyce, který není čistě funkcionální, a jaké kompromisy jsou s jednotlivými přístupy spojeny.

6.1 Objektově orientovaná implementace

Tato část představuje implementaci zvoleného problému v tradičním objektově orientovaném stylu, jaký je běžný pro jazyk Java. Kód je strukturován kolem tříd s měnitelným stavem, využívá explicitní řízení toku programu pomocí cyklů a větvení. K ošetření chybových stavů používá výjimky a práce s daty je převážně imperativní. Výpočetní logika je rozptýlena v jednotlivých metodách. V implementaci je patrný důraz na stav a sekvenční zpracování místo výrazové a kompoziční logiky.

```
class SensorRecord {
    Instant timestamp;
    double value;

    public SensorRecord(Instant timestamp, double value) {
        this.timestamp = timestamp;
        this.value = value;
    }
}
```

Zdrojový kód 72 – Třída v jazyce Java (objektově)

V objektové orientované verzi jsou data i jejich struktura primárně mutabilní. Pole *timestamp* a *value* nejsou deklarovány jako *final*, což umožňuje jejich změnu po vytvoření instance. To usnadňuje postupné doplňování či úpravu záznamů, ale zároveň zvyšuje riziko neúmyslných vedlejších efektů při souběžném přístupu z více vláken.

```
public Config loadConfig(String path) {
    try (Reader reader = new FileReader(path)) {
        return new Gson().fromJson(reader, Config.class);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Zdrojový kód 73 – Načítání konfigurace v jazyce Java (objektově)

Funkce pro načítání konfigurace využívá *try-with-resources* konstrukci pro správu souboru a výjimku *IOException*, které musí být ošetřeno pomocí catch konstrukce. V porovnání s Haskell verzí zde chybí výrazová čistota. I když je funkce *fromJson* z knihovny *gson* čistá, kód je nutně zabalen do *try* bloku kvůli správě zdrojů. Výsledek je zároveň méně flexibilní. Není snadné zachytit chybu parsování jako *Optional* nebo *Either* bez přidání dalších konstrukcí.

```
public Double safeAvg(List<Double> xs) {
    if (xs.isEmpty()) return null;
    double sum = 0;
    for (Double x : xs) sum += x;
    return sum / xs.size();
}

public Double computeAverage(List<SensorRecord> recs) {
    if (recs.isEmpty()) return null;
    double sum = 0;
    for (SensorRecord r : recs) sum += r.value;
    return sum / recs.size();
}
```

Zdrojový kód 74 – Statistické funkce v jazyce Java (objektově)

Java implementace hlavních výpočetních statistických funkcí pracuje imperativně. Výpočet průměru je rozdělen do dvou metod, ale obě používají cyklus a akumulační proměnnou. To vede k opakování kódu a nízké znovupoužitelnosti. Funkci *safeAvg* nelze jednoduše využít ve funkci *computeAverage*, protože Java bez funkcionálního Stream API neumožňuje jednoduchou transformaci přes *map* a *collect*. Výsledkem je zbytečná duplikace a nemožnost

snadno skládat funkce jako v jazyce Haskell. Návratová hodnota *null* je nebezpečná a v kontrastu s funkcionálními jazyky.

Generika v Javě má navíc určitá omezení, která mohou vést k méně bezpečnému kódu. Například *List<Double>* může obsahovat hodnoty *null*, což vyžaduje další ošetření, aby nedocházelo k chybám za běhu. Toto riziko v kombinaci s používáním *null* jako indikátoru absence hodnoty zvyšuje složitost a snižuje robustnost kódu.

```
public double computeMax(List<SensorRecord> recs) {
    double maxVal = 0.0;
    for (SensorRecord r : recs)
        if (r.value > maxVal) maxVal = r.value;
    return maxVal;
}

public int computeLongestAbove(List<SensorRecord> recs, double thr) {
    if (recs.isEmpty()) return 0;

    double t = computeMax(recs) * thr;
    int curr = 0;
    int best = 0;

    for (SensorRecord r : recs) {
        if (r.value > t) {
            curr++;
            if (curr > best)
                best = curr;
        } else curr = 0;
    }

    return best;
}
```

Zdrojový kód 75 – Složitější statistická funkce v jazyce Java (objektově)

Java verze funkce *computeLongestAbove* je plně imperativní. Výpočet probíhá ve dvou oddělených cyklech. Vše je řízeno mutabilními proměnnými, které se mění v průběhu cyklu. Tento styl je efektivní z hlediska výkonu a paměti, ale nevýhodou je těsné provázání výpočtu se stavem a kontrolou toku. Tento přístup vede k vícenásobnému průchodu daty a opakování stavové správy. Nevyužitím abstrakce pro zpracování kolekcí omezuje modularitu a optimalizaci. Kód je méně vhodný pro paralelizaci. Řešení je sice efektivní z hlediska výkonu, ale obtížněji rozšiřitelné v porovnání s funkcionálními přístupy.

```
public SensorReport processData(Config cfg, File file) {
    List<SensorRecord> records = new ArrayList<>();
    try (CSVReader reader = new CSVReader(new FileReader(file))) {
        String[] header = reader.readNext();
        String[] line;
        while ((line = reader.readNext()) != null) {
            try {
                SensorRecord rec = parseLine(header, line);
                records.add(rec);
            } catch (Exception e) {
                System.err.println(
                        "Wrong line: " + Arrays.toString(line));
            }
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

    return new SensorReport(
            file.getName(),
            computeAverage(records),
            computeAverageMonthly(records),
            detectOutliers(records, cfg.outlierThreshold),
            computeLongestAbove(records, cfg.aboveThreshold));
}
```

Zdrojový kód 76 – Zpracování dat v jazyce Java (objektově)

Metoda *processData* v Javě kombinuje vstupní operace, validaci a výpočty v rámci jednoho bloku, přičemž využívá mutabilní kolekce a explicitní řízení chyb přes výjimky a podmínky. Tento přístup usnadňuje přímou kontrolu nad procesem, ale zároveň vede k větší závislosti na vnitřním stavu a pořadí operací. Výsledný kód je méně modulární a hůře škálovatelný pro paralelní zpracování či rozšíření funkcionality bez úprav stávajícího toku. Také ztěžuje testování dílčích částí kvůli prolínání efektů a logiky.

6.2 Implementace s funkcionálními prvky

Následující implementace ukazuje, jak lze v jazyce Java aplikovat funkcionální principy bez nutnosti použití externích knihoven. Využívá jazykové konstrukce představené ve verzi Java 8 a novějších, zejména Stream API, *Optional*, lambda výrazy a efektivní práci s neměnnými hodnotami. Cílem tohoto přístupu je omezit výskyt vedlejších efektů, zvýšit deklarativnost a čitelnost kódu a umožnit snazší kompozici funkcí. Vzhledem k omezením samotného jazyka není možné dosáhnout stejné úrovně abstrakce jako v čistě funkcionálním jazyce, nicméně výsledný kód je výrazně bezpečnější, testovatelnější a lépe paralelizovatelný než v tradiční objektové verzi.

```
record SensorRecord(
    LocalDateTime timestamp,
    double value
) {}
```

Zdrojový kód 77 – Records v jazyce Java (funkcionálně)

V této verzi jsou datové struktury definovány jako neměnné záznamy pomocí *record*, čímž se eliminuje možnost změny stavu po vytvoření instance.

```
public Optional<Double> safeAvg(List<Double> values) {
    if (values == null) return Optional.empty();
    return values.stream().filter(Objects::nonNull)
                 .mapToDouble(Double::doubleValue)
                 .average().stream().boxed().findFirst();
}

public Optional<Double> computeAverage(List<SensorRecord> records) {
    return records.stream().mapToDouble(r -> r.value)
                  .average().stream().boxed().findFirst();
}
```

Zdrojový kód 78 Použití Optional (funkcionálně)

Metody demonstrují funkcionální přístup k výpočtu průměru s využitím *Optional* jako bezpečného kontejneru pro výsledek. Metody zajišťují ochranu proti *null* hodnotám a prázdným seznamům, přičemž používají stream operace pro filtrování a výpočet průměru. Použití *Optional* eliminuje riziko vyvolání výjimky a usnadňuje práci s chybovými stavy bez nutnosti explicitní kontroly.

```java
public int computeLongestAbove(List<SensorRecord> records, double thr) {
    if (records.isEmpty()) return 0;

    double maxVal = records.stream()
                        .mapToDouble(SensorRecord::value)
                        .max().getAsDouble();
    double t = maxVal * thr;

    return records.stream()
            .map(r -> r.value() > t ? 1 : 0)
            .reduce(new int[] { 0, 0 },
                    (acc, v) -> new int[] {
                            v == 1 ? acc[0] + 1 : 0,
                            Math.max(acc[1], v == 1 ? acc[0] + 1 : 0)
                    },
                    (a1, a2) -> new int[] {
                            0,
                            Math.max(a1[1], a2[1]) })[1];
}
```

Zdrojový kód 79 – Složitější statistická funkce v jazyce Java (funkcionálně)

Metoda *computeLongestAbove* je ukázkou čistě výrazového řešení úlohy bez použití imperativních konstrukcí. Využívá Stream API k transformaci záznamů a složenou redukci pro výpočet délky nejdelší posloupnosti. Výpočet probíhá bez stavových proměnných a bez cyklů, což zajišťuje referenční transparentnost a snadnou testovatelnost. Na druhou stranu, kvůli zapouzdření logiky do jediného výrazu *reduce* je výsledný kód obtížně čitelný a méně srozumitelný. Tento přístup je vhodný především pro prostředí, kde je kladen důraz na neměnnost a kompozici.

```
public Optional<SensorReport> processData(Config cfg, File file) {
    try (CSVReader reader = new CSVReader(new FileReader(file))) {
        List<String[]> lines = reader.readAll();
        if (lines.isEmpty()) {
            System.err.println("Empty file: " + file.getName());
            return Optional.empty();
        }

        String[] header = lines.getFirst();
        List<SensorRecord> records = lines.stream()
                .skip(1).map(line -> {
                    try {
                        return parseLine(header, line);
                    } catch (Exception e) {
                    System.err.println(
                        "Wrong line: " + Arrays.toString(line));
                        return Optional.<SensorRecord>empty();
                    }
                }).flatMap(Optional::stream).toList();

        return Optional.of(
                new SensorReport(
                        file.getName(),
                        computeAverage(records).orElse(null),
                        computeAverageMonthly(records),
                        detectOutliers(records, cfg.outlierThreshold()),
                        computeLongestAbove(records, cfg.aboveThreshold())));

    } catch (Exception e) {
        System.err.println("Failed to read file " + file.getName());
        return Optional.empty();
    }
}
```

Zdrojový kód 80 – Zpracování dat v jazyce Java (funkcionálně)

Funkce *processData* využívá *Optional* pro bezpečné vyjádření přítomnosti či absence výsledku při zpracování souboru. Místo výjimek vrací prázdný kontejner, čímž eliminuje nutnost ošetřovat chyby v nadřazeném kódu. Transformace vstupních řádků pomocí *map* a *flatMap* umožňuje filtrování neplatných záznamů bez přerušení toku zpracování. Lambda výraz s interní výjimkou odděluje parsing jednotlivých řádků a zachovává plynulost zpracování. Výsledkem je deklarativní styl, který omezuje vedlejší efekty a zvyšuje robustnost při práci s nekonzistentními vstupy.

6.3 Nástroje a knihovny pro FP

Vedle jazykové podpory existuje řada knihoven a nástrojů, které umožňují využívat funkcionální přístupy i v objektově orientovaných jazycích. V jazyce Java je prominentní knihovnou Vavr, která poskytuje immutable kolekce, monadické typy, pattern matching a datové struktury inspirované Haskellem.

Dalším příkladem je FunctionalJava, starší knihovna, která implementuje základní funkcionální konstrukce pro currying, rekurzivní datové typy nebo lazy evaluation. (Functional Java) Pro pokročilé uživatele existuje projekt F-Java, který se pokouší rozšířit expresivitu Javy o typové konstrukty a vzory z jazyka Haskell, včetně algebraických datových typů a vlastních typových tříd.

V oblasti testování funkcionálního kódu nabízejí knihovny jako jqwik nebo junit-quickcheck nástroje pro property-based testování, známé z Haskellu. V kombinaci s immutable strukturami je tento přístup efektivní pro formální ověřování chování programu. (jqwik) (junit-quickcheck)

6.4 Vybrané knihovny

6.4.1 Vavr (Javaslang)

Vavr (původně Javaslang) je moderní knihovna pro funkcionální programování v jazyce Java. Jejím cílem je doplnit jazyk o funkcionální datové struktury a idiomy, které nejsou v Javě dostupné nativně. Vavr poskytuje immutable kolekce (*List*, *Map*, *Set*), monadické konstrukce (*Option*, *Try*, *Either*), pattern matching a podporu funkcionálních rozhraní s více než jedním parametrem. [19] V praxi tak umožňuje psát kód, který se více podobá stylu jazyků jako Haskell, ale zůstává plně kompatibilní s „ekosystémem“ Javy.

```
Try<Integer> result = Try.of(() -> Integer.parseInt("42"));
Option<Integer> maybeResult = result.toOption();

maybeResult.map(x -> x * 2).peek(System.out::println); // 84
```

Zdrojový kód 81 - Využití Try a Option z knihovny Vavr

Řešení využívá bezpečné zpracování výjimek a práci s hodnotami, které nemusejí být vypočítány. Pomocí *Try.of* je obalen výraz, který může selhat. V tomto případě se jedná o převod textu na číslo, přičemž výsledek je *Success*, nebo *Failure*. Metoda toOption pak tento výsledek převádí na *Option*, obdoba interního *Optional*. Tím kód zajišťuje funkcionální způsob zpracování možné chybějící hodnoty. Funkce map následně hodnotu transformuje uvnitř

Option. Funkce *peek* pak slouží k provedení vedlejšího efektu (výpis hodnoty). Výhodou je eliminace ručního zachytávání výjimek a práce s *null*, což zvyšuje bezpečnost kódu. [19]

```
Either<String, Integer> parseInt(String input) {
   try {
       return Either.right(Integer.parseInt(input));
   } catch (NumberFormatException e) {
       return Either.left("Wrong input: " + input);
   }
}

Either<String, Integer> result = parseInt("xyz");

result
   .peek(value -> System.out.println("Value: " + value))
   .peekLeft(error -> System.out.println("Error: " + error));
```

Zdrojový kód 82 - Využití Either z knihovny Vavr

Either z knihovny Vavr se velmi podobá monádě *Either* z jazyka Haskell. Slouží k reprezentaci výpočtu, který může skončit chybou *(Left)* nebo úspěchem *(Right)*. Funkce, která může skončit jednou z uvedených variant je *parseInt*. Funkce *peek* a *peekLeft* pak umožňují dále výsledek operace převodu zpracovat (výpis hodnoty). Přístup nahrazuje ruční správu výjimek a podporuje čistý a bezpečný kód. [19]

6.4.2 Functional Java

Functional Java je jedna z prvních knihoven, které přinesly funkcionální paradigma do jazyka Java. Nabízí nástroje pro funkcionální transformace, *Option*, *Either*, líné vyhodnocování, immutable kolekce, kompozici funkcí a podporu pro rekurzi. [20] Díky své architektuře je však syntaxe výrazně těžkopádnější než u moderních alternativ jako Vavr. Přesto je vhodná pro výuku a pochopení principů jako je currying nebo funkcionální iterace.

```
F<Integer, F<Integer, Integer>> add = a -> b -> a + b;
F<Integer, Integer> add10 = add.f(10);
Integer result = add10.f(5); // 15
```

Zdrojový kód 83 - Využití Funktor z knihovny F-Java

Ukázka demonstruje currying, kde funkce *add* vrací jinou funkci. Místo tradičního dvouparametrového zápisu se funkce rozkládá na sekvenci funkcí s jedním parametrem. Funkce *add10* je výsledkem aplikace první části funkce, čímž vznikne funkce čekající na druhý parametr. Do proměnné *result* se pak uloží hodnota provedeného konečného výpočtu. Tento

styl usnadňuje částečnou aplikaci funkcí a přispívá k vyšší modularitě a znovupoužitelnosti kódu.

```
Lazy<Integer> lazyValue = Lazy.lazy(() -> {
   System.out.println("Evaluating...");
   return 42;
});

System.out.println("Before value() call");
System.out.println("Value: " + lazyValue.value());
```

Zdrojový kód 84 - Využití Lazy a líného vyhodnocení z knihovny Functional Java

V příkladu je použita třída *Lazy* pro odložení výpočtu až do momentu, kdy je skutečně potřeba. Výraz obalený v *Lazy.lazy* se nevyhodnocuje ihned, ale až při prvním zavolání metody *value*. Díky tomu se například náročné nebo vedlejší výpočty neprovádějí zbytečně. Výstup ukazuje, že k vyhodnocení dochází až ve chvíli, kde je hodnota poprvé vyžádána. To umožňuje efektivnější využití prostředků a podporuje princip referenční transparentnosti.

6.4.3 Junit-Quickcheck, Jqwik

Property-based testing (PBT) je technika známá z jazyka Haskell, kde jsou testovány obecné vlastnosti funkcí namísto konkrétních vstupů. V Javě tuto možnost přináší knihovny jako quickcheck a jqwik. Obě jsou integrovány s JUnit a umožňují generovat vstupy na základě anotací a náhodného generování. Vhodné jsou zejména pro testování čistých funkcí a algoritmů s jednoznačným chováním. [21, 22]

```
@Property
boolean additionIsCommutative(@ForAll int a, @ForAll int b) {
   return a + b == b + a;
}
```

Zdrojový kód 85 - Testování komutativnosti sčítání pomocí Jqwik

Ukázka ověřuje vlastnosti komutativnosti sčítání pomocí knihovny jqwik. Anotace *@Property* označuje metody jako testovací vlastnost a *@ForAll* zajistí, že vstupní hodnoty budou při každém spuštění testu náhodně generovány. Test vrací *true*, pokud pro všechny vstupy platí, že $a + b$ je rovno $b + a$. Tento přístup umožňuje ověřit obecné algebraické zákonitosti na mnoha různých případech bez nutnosti psát jednotlivé vstupy ručně. [21]

```java
@Property
boolean multiplicationByZero(@ForAll int x) {
   return x * 0 == 0;
}
```

Zdrojový kód 86 - Testování násobení nulou pomocí Jqwik

Test *multiplicationByZero* testuje, že násobení jakéhokoliv celého čísla nulou vrátí vždy nulu. Anotace *@ForAll* stejně jako v předchozí ukázce generuje různé vstupní hodnoty *x*. Tím je zajištěno ověření platnosti výrazu v širokém rozsahu vstupů. I přesto, že se jedná o primitivní test, podobné matematické testy jsou vhodné pro kontrolu základních axiomů a neporušitelnosti matematických pravidel ve složitějších implementacích. [21]

```java
@Property
boolean reverseTwiceIsIdentity(@ForAll List<Integer> xs) {
   List<Integer> reversedTwice = new ArrayList<>(xs);
   Collections.reverse(reversedTwice);
   Collections.reverse(reversedTwice);
   return xs.equals(reversedTwice);
}
```

Zdrojový kód 87 - Testování indentity po dvojnásobném otočení seznamu pomocí Jqwik

Ukázka testuje dvojnásobné otočení seznamu pomocí funkce reverse ze třídy *Collections*. Test kontroluje, že se po aplikaci funkcí vrátí původní seznam. Vstupní data *xs* jsou náhodně generované seznamy celých čísel. Vlastnost identity po dvou otočeních je příkladem, který se vyplatí ověřit v případech, kdy se implementují operace nad datovými strukturami. [21]

6.5 Implementace s využitím knihovny Vavr

Při využití knihovny Vavr lze zápis funkcí výrazně zjednodušit a úzce přiblížit idiomům čistě funkcionálního programování. Vavr poskytuje sadu datových typů, které umožňují sestavit bezpečnější, deklarativnější a spolehlivější kód.

```
public static Option<Double> safeAvg(List<Double> values) {
    return values.isEmpty()
            ? Option.none()
            : Option.some(values.sum().doubleValue() / values.size());
}

public static Option<Double> computeAverage(List<SensorRecord> records) {
    return safeAvg(records.map(SensorRecord::value)).orElse(Option.none());
}
```

Zdrojový kód 88 – Option z knihovny Vavr a jeho použití

Využít typ *Option* z knihovny Vavr lze například při práci s bezpečnými výpočty průměrů. Funkce *safeAvg* pracuje nad neměnným seznamem, přičemž výsledek vrací jako *Option*. Tím explicitně vyjadřuje možnost absence hodnoty bez použití *null*. Výpočet je výrazový a bez vedlejších efektů. Funkce *computeAverage* pak kombinuje mapování přes datový typ s kompozicí volání, kdy využívá *orElse* pro ošetření případného *None*. Oproti běžné verzi v Javě používající *Stream* a *Optional* je kód stručnější, deklarativnější a lépe odpovídá idiomům čistě funkcionálního programování.

```
public int computeLongestAbove(List<SensorRecord> records, double thr) {
    if (records.isEmpty()) return 0;

    double maxVal = records.map(SensorRecord::value).max().get();
    double t = maxVal * thr;

    return records.foldLeft(Tuple.of(0, 0), (acc, record) -> {
        int current = record.value() > t ? acc._1 + 1 : 0;
        int max = Math.max(acc._2, current);
        return Tuple.of(current, max);
    })._2;
}
```

Zdrojový kód 89 – Využití foldLeft a typu Tuple z knihovny Vavr

Metoda *computeLongestAbove* využívá plně neměnné kolekce a datové struktury. Místo imperativní logiky používá funkci *foldLeft* pro akumulaci stavu bez mutace. Stav je

reprezentován pomocí typu *Tuple*, který zachycuje délku aktuální a maximální sekvence. Výpočet prahové hodnoty je realizován přes map a max, bez potřeby explicitní iterace. Tímto způsobem je celý výpočet bez vedlejších efektů a přehledně odděluje transformaci dat od jejich akumulace.

```
public Option<SensorRecord> parseLine(Map<String, Integer> m, String[] line) {
    return Try.of(() -> {
        Instant timestamp = Instant.from(
                TIME_FORMAT.parse(line[m.get("timestamp").get()]));
        double value = Double.parseDouble(
                line[m.get("value").get()]);
        return new SensorRecord(timestamp, value);
    }).toOption();
}
```

Zdrojový kód 90 – Typ Try z knihovny Vavr a jeho použití

Funkce *parseLine* ukazuje typické použití konstrukce *Try* pro bezpečné zachycení výjimek při parsování dat. Místo explicitního ošetření výjimek pomocí *try-catch* konstrukce je potenciálně selhávající výpočet zapouzdřen do výrazu *Try*, který výsledek převede na *Option*. Tím se výjimky transofrmují na čistě funkcionální reprezentaci selhání bez narušení toku programu. Tento přístup zjednodušuje práci s chybami, podporuje složení výpočtů a zajišťuje, že výsledek bude vždy bezpečně zachycen v typovém systému.

```
public List<SensorRecord> detectOutliers(List<SensorRecord> recs, double thr){
    List<Double> values = recs.map(SensorRecord::value);
    return safeAvg(values).toList()
            .zipWith(safeStdDev(values), (mean, stdDev) -> {
                double t = 100 * thr * stdDev;
                return recs.filter(r -> Math.abs(r.value() - mean) > t);
            }).getOrElse(List.empty());
}
```

Zdrojový kód 91 – Využití funkce vyššího řádu zipWith z knihovny Vavr

Metoda *detectOutliers* kombinuje několik funkcionálních principů a idiomů. Používá expresivní řetězení volání a implicitní práci s hodnotami, které nemusí být přítomné opět díky typu *Option*. Důležitým prvkem je i funkce *zipWith*, která umožňuje sloučit výsledky dvou samostatných výpočtů do jedné funkce bez potřeby explicitních kontrol přítomnosti hodnot.

6.6 Omezení a úskalí

Přestože moderní objektově orientované jazyky, jako jsou Java nebo C#, postupně integrují funkcionální prvky do své syntaxe včetně standardních knihoven, jejich použití v praxi naráží na řadu omezení. Tyto limity jsou důsledkem historického vývoje jazyků, jejich návrhových cílů i kompatibility se stávajícím imperativním a objektovým ekosystémem.

Omezená expresivita typového systému

Jedním z hlavních rozdílů mezi funkcionálními a objektově orientovanými jazyky je úroveň expresivity jejich typových systémů. Zatímco funkcionální jazyky jako Haskell disponují pokročilými konstrukcemi, např. algebraické datové typy nebo typové třídy, jazyk jako Java tyto koncepty nepodporují, nebo je podporují pouze částečně.

Java neumožňuje definovat obecné operace nad typy vyšších řádů, což znemožňuje přímou implementaci obecné monadické abstrakce. Výsledkem je nutnost opakovaně definovat podobné konstrukce (*Optional*, *Try*, *Stream*) bez možnosti jejich sjednocení pod jednotné rozhraní.

Verbosita a omezené vyjadřovací možnosti

Funkcionální styl klade důraz na výrazovou stručnost, kompozici a minimalizaci kódu. Java nebo C# však trpí vysokou rozsáhlostí syntaxe, zejména v kontextu práce s funkcemi vyššího řádu, immutable strukturami a pattern matchingem.

Práce s kolekcemi ve stylu Stream API v Javě vyžaduje řetězení několika metod, často s anonymními funkcemi, bez jazykových konstrukcí usnadňujících kompozici. Navíc absence plnohodnotného pattern matchingu komplikuje práci s algebraickými strukturami a vyžaduje používání méně čitelných *if-else* nebo switch konstrukcí.

Chybějící optimalizace pro rekurzi

Rekurze je základní konstrukcí ve funkcionálním programování, zejména v kombinaci s neměnnými strukturami, kdy klasická iterace přes stavové proměnné není vhodná. Funkcionální jazyky proto běžně využívají optimalizace jako tail call optimization (TCO), která umožňuje efektivní provádění rekurzivních volání bez zahlcení zásobníku.

Objektově orientované jazyky však TCO často nativně nepodporují, což činí hlubokou rekurzi nebezpečnou a nevhodnou pro produkční prostředí. Vzniká nutnost rekurzi nahradit imperativními cykly, což oslabuje konzistenci funkcionálního stylu a komplikuje práci s funkcemi vyššího řádu nebo líným vyhodnocováním.

Neplná neměnnost

Ačkoliv jazyková podpora pro neměnnost v OOP jazycích existuje, ve skutečnosti zajišťuje pouze částečnou ochranu. Proměnná v Javě s označením *final* chrání referenci, ale ne vnitřní stav objektu. Neexistuje ani jednoduchý způsob, jak definovat komplexní neměnné objekty bez značného množství kódu.

Na rozdíl od jazyků jako Haskell, které podporují neměnnost výchozím způsobem a celý runtime je optimalizován pro práci s immutable daty, v Javě nebo C# je třeba neměnnost explicitně prosazovat a hlídat, čímž vzniká prostor pro chyby a nejednotné styly.

Interoperabilita s imperativním kódem

Funkcionální styl není vždy snadno slučitelný s tradičním imperativním přístupem, který dominuje většině existujících syntaxí. Při použití například *Optional*, *Either* nebo funkcionální kolekce z knihoven jako Vavr je často nutné převádět tyto struktury na klasické typy (např. *null*, *List*, *Exception*) pro účely kompatibility s rozhraními, API knihovnami nebo frameworky.

Tato interoperabilita s imperativním světem vyžaduje konverze, duplikaci kódu a zvyšuje komplexitu aplikace. V extrémních případech může vést ke vzniku tzv. „funkcionálně vypadajícího imperativního kódu“, který se tváří jako funkcionální, ale ve skutečnosti porušuje jeho základní principy.

Nároky na vývojáře

Funkcionální přístup vyžaduje jiný způsob uvažování, odlišný od objektově orientovaného či imperativního programování. Principy jako neměnnost, složení funkcí, lazy evaluace nebo vyšší abstrakce (funktory, monády) nejsou běžně zmiňovány v rámci standardních OOP jazyků a mohou představovat výrazné zvýšení náročnosti.

Kromě změny myšlení navíc vzniká nutnost osvojit si nové knihovny, idiomy a často i metodiky testování. Bez dostatečné míry znalostí se může funkcionální přístup stát zdrojem nečitelných nebo chybně pochopených konstrukcí, což má negativní dopad na údržbu a rozšiřitelnost systému.

ZÁVĚR

Cílem této práce bylo představit základní i pokročilé principy funkcionálního programování a analyzovat jejich využitelnost v současných objektově orientovaných jazycích. Byly popsány klíčové koncepty, jako je referenční transparentnost, práce s immutable daty, funkce vyššího řádu nebo monády, a jejich implementace a použití byly ilustrovány na příkladech v jazycích Haskell a Java. Práce zároveň ukázala, že i v objektově orientovaných jazycích je možné funkcionální techniky efektivně využívat, a to zejména díky podpoře multiparadigmatického programování a dostupnosti knihoven.

V praktické části byly jednotlivé principy aplikovány na konkrétní úlohy, což umožnilo porovnat přístupy obou paradigmat z hlediska srozumitelnost, modularity a potenciální udržovatelnosti kódu. Přestože jazyk Haskell nabízí vyšší míru vyjádřitelnosti čistě funkcionálních technik, ukázalo se, že i v jazycích jako Java lze dosáhnout podobných výsledků při využití vhodných idiomů a nástrojů.

Funkcionální programování přináší silný nástrojový rámec pro tvorbu spolehlivého, predikovatelného a modulárního kódu. Jeho širší adopce však závisí nejen na technických možnostech daného jazyka, ale i na ochotě vývojářů osvojit si nový způsob uvažování o problémech. Vzhledem k rostoucím nárokům na robustnost a paralelizovatelnost moderních aplikací lze očekávat, že význam funkcionálních principů bude nadále růst.

POUŽITÁ LITERATURA

[1] MAUELLER, John Paul. *Functional Programming For Dummies*. B.m.: Wiley, 2019. ISBN 978-1-119-52751-0.

[2] TURNER, David. Church's Thesis and Functional Programming. In: *Church's Thesis After 70 Years*. B.m.: De Gruyter, 2006.

[3] STEINERT-THRELKELD, Shane. *Lambda Calculi*. Online. nedatováno. Dostupné z: https://iep.utm.edu/lambda-calculi. [citováno 2025-05-10]

[4] JAN VAN LEEUWEN. *Handbook of theoretical computer science*. Vol. B, Formal models and semantics. Amsterdam: Elsevier, 1990. ISBN 0-444-88075-5.

[5] BOSSARD, Antoine. *A Gentle Introduction to Functional Programming in English*. 3. vyd. B.m.: Ohmsha, 2020. ISBN 978-4-274-70106-1.

[6] WIDMAN, Jack. *Learning functional programming: Managing code complexity by thinking functionally*. Beijing: O'Reilly Media, 2022. ISBN 978-1-0981-1175-5.

[7] TOURETZKY, David S. *Common LISP: A Gentle Introduction to Symbolic Computation*. B.m.: Dover Publications, 2013. ISBN 978-0-486-49820-1.

[8] HUDAK, Paul, John HUGHES, Simon Peyton JONES a Philip WADLER. A history of Haskell: being lazy with class. Online. 2007. ISSN 9781595937667. Dostupné z: doi:10.1145/1238844.1238856.

[9] SHARMA, Pardeep. What is the Future of Functional Programming? *Analytics Insight*. Online. 20. 10. 2024. Dostupné z: https://www.analyticsinsight.net/programming/what-is-the-future-of-functional-programming. [citováno 2025-05-12]

[10] HUTTON, Graham. *Programming in Haskell*. 2. vyd. B.m.: Cambridge University Press, 2016. ISBN 978-1-316-62622-1.

[11] SAJANIKAR, Yogesh. *Haskell Cookbook: Build functional applications using Monads, Applicatives, and Functors*. B.m.: Packt Publishing, 2017. ISBN 1-78646-265-6.

[12] CHUKWUBE, Michael. The Role of Functional Programming in Modern Software Development. *DZone*. Online. 24. 4. 2025. Dostupné z: https://dzone.com/articles/role-of-functional-programming. [citováno 2025-05-15]

[13] ALMOG, Shai. *Java 8 to 21: Explore and work with the cutting-edge features of Java 21 (English)*. B.m.: BPB Publications, 2023. ISBN 978-93-5551-392-2.

[14] GOSLING, James, Bill JOY, Guy STEELE, Gilad BRACHA a Alex BUCKLEY. *The Java® Language Specification*. B.m.: Oracle, 2015.

[15] BLOCH, Joshua. *Effective Java: Programming Language Guide*. 3. vyd. B.m.: Addison-Wesley Professional, 2017. ISBN 978-0-13-468599-1.

[16] KURT, Will. *Get programming with Haskell*. B.m.: Manning, 2018. ISBN 978-1-61729-376-4.

[17] SWAINE, Michael. *Functional Programming: A PragPub Anthology: Exploring Clojure, Elixir, Haskell, Scala, and Swift*. B.m.: Pragmatic Bookshelf, 2017. ISBN 1-68050-233-6.

[18] ABEL, Andreas, Brigitte PIENTKA, David THIBODEAU a Anton SETZER. Copatterns: programming infinite structures by observations. Online. 2013, 27–38. ISSN 0362-1340. Dostupné z: doi:10.1145/2480359.2429075.

[19] DIETRICH, Daniel, Robert WINKLER a Grzegorz PIWOWAREK. *Vavr User Guide*. 6. 12. 2024

[20] *Functional Java*. Online. Dostupné z: https://github.com/functionaljava/functionaljava. [citováno 2025-05-17]

[21] *jqwik User Guide, Property-based testing in Java*. Online. 3. 12. 2024. Dostupné z: https://jqwik.net/docs/current/user-guide.html. [citováno 2025-05-19]

[22] PAUL R. HOLSER, JR. *junit-quickcheck: Property-based testing, JUnit-style*. Online. 21. 11. 2020. Dostupné z: https://pholser.github.io/junit-quickcheck/site/1.0. [citováno 2025-05-18]