Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra Informačních Technologií

Funkcionální Programování

Bakalářská práce

Autor: Martin Kaněra
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. Josef Hynek, MBA, Ph.D.

Hradec Králové duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 21.4.2024

Martin Kaněra

Poděkování:

Děkuji vedoucímu bakalářské práce prof. RNDr. Josefu Hynkovi, MBA, Ph.D. za metodické vedení práce a inspiraci, která mě motivovala ke studiu funkcionálního programování a zejména Haskellu.

Abstrakt

Tato bakalářská práce se zabývá tématem funkcionálního programování s konkrétním zaměřením na programovací jazyk Haskell. Hlavním cílem této práce je poskytnout studentům základní přehled o programování v jazyku Haskell, a tak doplnit studijní materiály. Práce systematicky popisuje základní koncepty a principy funkcionálního programování, jako jsou čisté funkce, neměnnost stavu a funkce jako prvotřídní občané, a dále se věnuje specifikům jazyka Haskell, včetně jeho historie, syntaxe, typového systému a jeho programovacím konceptům.

V práci je detailně probrána teorie programování v jazyce Haskell, která je doplněna o části kódu, které demonstrují způsob zápisu a případy užití. Součástí práce je také řada vypracovaných úloh, které obsahují zadání, řešení a vysvětlení, což dodává studentům motivaci pokračovat ve studiu Haskellu a zkoumání funkcionálního paradigmatu jako celku.

Abstract

Title: Functional programming

This bachelor thesis addresses the topic of functional programming, with a primary focus on Haskell programming language. The main goal of this thesis is to provide students with a basic overview of Haskell programming to complement their study materials. The thesis systematically describes the fundamental concepts and principles of functional programming, such as pure functions, immutability, and functions as first-class citizens, and further explores the specifics of the Haskell language, including its history, syntax, type system, and programming concepts that are used in it.

The work thoroughly examines the theory of programming in Haskell, supplemented by code sections that demonstrate the notation and use cases. Additionally, the thesis includes a series of solved tasks, which contain assignments,

solutions, and explanations, thereby motivating students to continue their study of Haskell and explore the functional paradigm in general.

Key words: Functional programming, Haskell, ghci, Lazy evaluation, Recursion, Pure functions

Obsah

1 Úvod

Vstup do světa funkcionálního programování, zejména prostřednictvím Haskellu, pro mě představoval výzvu i příležitost. S bagáží znalostí z jednosemestrálního kurzu logického programování v Prologu jsem se obával, že se budu muset znovu potýkat s obtížemi spojenými s přístupem k programování založeným na predikátech a pravidlech, které jsem jako student považoval za obtížně srozumitelné. Místo toho jsem byl příjemně překvapen; Haskell zcela předčil má očekávání, co se týče jeho syntaxe a typového systému.

Funkcionální programování nabízí odlišný pohled na řešení problémů a tvorbu algoritmů. S jeho principy, jako je neměnnost stavu, čisté funkce a funkce vyššího řádu, jsem objevil nový způsob programování a abstrakce. Tuto cestu jsem zejména podnikl proto, jelikož existují studenti jako já, kteří mají problém pochopit problematiku pouze z přednášek a cvičení.

S tímto poznáním a zkušenostmi jsem se rozhodl věnovat svou bakalářskou práci tvorbě podpůrných materiálů pro studenty, kteří se snaží proniknout do základů funkcionálního programování a Haskellu. Cílem je nabídnout studentům materiály, které jim pomohou pochopit základy funkcionálního programování, a především základy programovacího jazyku Haskell.

2 Cíl a metodika práce

Hlavním cílem této bakalářské práce je poskytnout studentům základní přehled o funkcionálním programování s hlubším zaměřením na jazyk Haskell v rozsahu jednosemestrálního kurzu. Práce si klade za úkol nejen demonstrovat, jak lze s pomocí funkcionálního programování efektivně řešit běžné programovací úkoly, ale také usnadnit pochopení těchto principů skrze praktické příklady. Cílem je provést čtenáře základními principy, které se uplatňují v jazyce Haskell, a nabídnout srozumitelné příklady, na kterých si mohou studenti ověřit získané teoretické a praktické vědomosti. Vzhledem k tomu, že většina materiálů o Haskellu je dostupná převážně v anglickém jazyce, což může pro některé studenty představovat bariéru, se práce zaměřuje na zpřístupnění těchto materiálů prostřednictvím důkladně zpracovaných příkladů a vysvětlení v českém jazyce.

K dosažení těchto cílů byla zvolena metodika kombinující teoretický výklad s praktickým programováním. Důraz je kladen na interaktivní přístup k učení, kde studenti mají možnost aplikovat teorii v praxi prostřednictvím řešení úloh. Práce zahrnuje široké spektrum témat. V rámci metodiky byl využit přístup založený na studiu a analýze dostupných zdrojů a na programování konkrétních příkladů. Specifická pozornost byla věnována i potřebě podpořit studenty v samostatném řešení problémů a motivovat je k dalšímu prohlubování svých znalostí v oblasti funkcionálního programování.

3 Programovací paradigmata

Programovací paradigmata jsou způsoby, kterými jsou programy koncipovány a organizovány. Každé paradigma se skládá z určitých struktur, vlastností a různých názorů, jak by se v programování měly řešit běžné problémy (Cocca, 2022).

Je třeba si uvědomit, že programovací paradigmata jako taková nejsou programovacím jazykem, nýbrž souhrnem ideálů, které se v jednotlivých programovacích jazycích mohou uplatňovat.

Ač je pravda, že některé programovací jazyky byly vyvíjeny s ohledem na určité paradigma – např. Haskell, tak je důležité mít na paměti, že ne vždy tomu tak je. Programovací jazyky jako takové nejsou nutně spojeny s jedním paradigmatem, ale existují i multiparadigmatické programovací jazyky, které programátorům umožňují adaptovat kód různým přístupům, např. *JavaScript* či *Python* (Cocca, 2022).

3.1 Imperativní programování

Imperativní paradigma je nejstarší programovací paradigma. Jeho hlavní charakteristikou je definování posloupností instrukcí, které představují změny stavů počítačového systému. Programy řídící se imperativním paradigmatem tedy můžeme chápat jako „návody“, které popisují, jak splnit nějaký úkol (Fulber-Garcia, 2021).

Procedurální programování je založeno na paradigmatu imperativním. Zaměřuje se na rozdělení programu z jednoduché posloupnosti instrukcí na vícero procedur (funkcí) s konkrétními instrukcemi a proměnnými. Tyto procedury jsou přizpůsobeny k provedení jedné přesně definované úlohy. Rozdělení na jednotlivé úlohy formou spustitelných procedur vede k čistší organizaci kódu a modularizaci (Fulber-Garcia, 2021).

Nejrozšířenějším programovacím paradigmatem, které zároveň vychází z imperativního programování, je objektově orientované programování neboli OOP. Klíčovým konceptem OOP je rozdělení problémů do určitých entit – objektů. Jednotlivé entity seskupují danou množinu informací (vlastnosti) a akcí (metody), které může entita provádět. OOP je založeno na psaní tříd, které slouží jako předpis při zakládání nových objektů – instancí (Cocca, 2022).

3.2 Deklarativní programování

Hlavním rysem deklarativního paradigmatu je definice úkolů, které musí počítačové systémy splnit. Na rozdíl od imperativního paradigmatu není v deklarativním paradigmatu jasné jak úlohu zpracovat. Naopak, programy řídící se deklarativním paradigmatem fungují jako „průvodci" s instrukcemi definujícími, jaké úkoly má počítačový systém splnit, bez ohledu na to, jakým způsobem jsou tyto úkoly realizovány (Fulber-Garcia, 2021).

Z deklarativního paradigmatu se odvíjí logické paradigma, které využívá formální logiku k řešení problémů. Toto paradigma se opírá o databázi, jenž se skládá z faktů a pravidel pro zodpovídání dotazů.

Podstatným zástupcem deklarativního paradigmatu je funkcionální programování, kde se uvažuje takovým způsobem, že programy jako takové jsou složeny z více funkcí (Fulber-Garcia, 2021). Funkcionální programování je postaveno na třech základních principech:

3.2.1 Čisté funkce (pure functions)

Na rozdíl od ostatních paradigmat, ve funkcionálním jsou funkce opravdu funkcemi, tak jak je to například v matematice. Aby nedocházelo k záměně s funkcemi z imperativních jazyků, jsou tyto funkce obvykle označovány jako „čisté" (Milewski, 2014). Čisté funkce nám v zásadě poskytují dvě hlavní vlastnosti.

První vlastností je, že za předpokladu zavolání funkce se stejnými argumenty bude výsledek této funkce pokaždé zcela identický s výsledky předchozími.

Druhá vlastnost je spojena s předchozí – absence vedlejších efektů (side effects). V knize „Real World Haskell“ (O'Sullivan et al., 2008) je uvedeno, že vedlejší efekt ve své podstatě představuje závislost mezi globálním stavem systému a chováním funkce samotné. V kontextu alternativní situace, která se může objevit u imperativních programovacích jazyků. Je možné identifikovat funkci, která přečte globální proměnnou a vrátí její výsledek. K této globální proměnné mají přístup i jiné části systému a můžou ji dle libosti upravovat. Z toho vyplývá, že výsledek volání funkce je závislý na aktuální hodnotě globální proměnné, a tudíž vzniká nechtěný vedlejší efekt, i přestože proměnnou neupravuje funkce samotná.

Příklad čisté funkce v jazyce Haskell, který ilustruje obě zmíněné vlastnosti.

```
1.  faktorial :: Integer -> Integer
2.  faktorial 0 = 1
3.  faktorial n = n * faktorial (n - 1)
```

Pokud tuto funkci zavoláme s konkrétním číslem, např. *faktorial 5*, výsledek bude vždy stejný – tedy 120 a to platí bez ohledu na to, kolikrát nebo odkud bude funkce zavolána. Zároveň tato funkce nezávisí na žádném vnějším stavu a ani ho nemění. Její výsledek je zcela závislý na vstupní hodnotě.

3.2.2 Neproměnný stav (immutability)

Jedna z věcí, kterou různé imperativní jazyky sdílejí, je možnost přepisování hodnot v proměnných. Toto může vést k hodinám řešení problémů, které by nikdy nevznikly, kdyby po přiřazení hodnoty do proměnné nebylo možné tuto hodnotu změnit.

Některé jazyky jako například *Rust* se tomuto nepředvídatelnému chování snaží předejít pomocí přístupů, kde jsou veškeré proměnné v základu neměnné, ale toto chování je velmi jednoduše změnitelné (Klabnik & Nichols, b.r.). Nicméně na toto chování je v Haskellu kladen větší důraz.

V Haskellu je vše výrazem a všechny výrazy jsou neměnné. To znamená, že po deklaraci proměnné není možné hodnotu změnit. Na první pohled se může zdát, že některé funkce modifikují původní hodnotu, ale je tomu přesně naopak, nedochází ke změně hodnoty, ale nýbrž k vytvoření hodnoty nové a k jejímu následovnému navrácení (Bowen, 2018). Při práci s funkcí, která přijímá seznam v jazyce Haskell, lze předpokládat, že ať budou s daným seznamem provedeny jakékoliv operace, nikdy nenastane situace, kde by došlo k modifikaci prvního seznamu. Na druhé straně, v kontextu jazyka jako je *Java*, nemůže být zaručeno, že funkce přijímající seznam jako argument tento seznam nějakým způsobem nemodifikuje.

Tuto vlastnost je možné demonstrovat pomocí ghci a vestavěné funkce *reverse*, která přijímá seznam a vrací seznam nový, jehož prvky byly otočeny.

```
ghci> let seznam = [1, 2, 3]
ghci> reverse seznam
[3,2,1]
ghci> seznam
[1,2,3]
```

V příkladu je možné pozorovat definování proměnné *seznam*, které je přiřazen seznam se třemi prvky *[1, 2, 3]*. Následovně je zavolána funkce *reverse* a jako vstupní argument je poskytnuta proměnná *seznam* – výsledkem volání této funkce je seznam s otočenými prvky, tedy *[3, 2, 1]*. Nicméně na dalším řádku probíhá dotazování na proměnnou *seznam*, jejíž hodnota zůstává nezměněná.

3.2.3 Funkce jakožto prvotřídní občané (first-class citizens)

V Haskellu, a ve funkcionálním programování obecně, se s funkcemi nakládá jakožto s prvotřídními hodnotami. Tento fakt umožňuje používat funkce jako jakoukoliv jinou hodnotou – je možné ji uložit do proměnné, ke které je později přistoupeno s úmyslem ji zavolat, poskytnout ji ve formě argumentu jiné funkci, a dokonce je možné funkci vrátit formou výsledku volání funkce (Allen & Moronuki, 2016). Toto chování umožňuje velmi efektivní práci s funkcemi a navádí k psaní funkcí, které budou plnit jednu určitou úlohu a tím pádem budou i znovupoužitelné.

```
1.   duplikuj :: Int -> Int
2.   duplikuj x = x * 2
3.
4.   pouzijFunkci :: (Int -> Int) -> Int -> Int
5.   pouzijFunkci f x = f x
```

V ukázce jsou definovány dvě funkce – *duplikuj*, která na svém vstupu očekává celočíselnou hodnotu a vrací její dvojnásobek. A také *pouzijFunkci*, jenž jako první parametr přijímá funkci a jako druhý parametr celočíselnou hodnotu, výsledkem volání této funkce je zavolání poskytnuté funkce s předaným celočíselným parametrem.

```
ghci> pouzijFunkci duplikuj 2
4
```

V ukázce konzolového okna je možné pozorovat, že pokud je zavolána funkce *pouzijFunkci* s funkcí *duplikuj* a číslem 2 jako argumenty, je navrácen výsledek 4. Toto ukazuje sílu funkcionálního programování v Haskellu, kde můžeme snadno předávat funkce jako argumenty a vytvářet složitější kompozice funkcí.

4 Funkcionální programování v praxi

V současné době není příliš vysoká pracovní poptávka po programátorech, kteří jsou zběhlí v čistě funkcionálních programovacích jazycích jako je např. Haskell (erkmos, 2017/2024). Ale je důležité si uvědomit, že funkcionální programování nutí programátory přemýšlet z jiného úhlu – je to spíše myšlenka, díky které je možné programovat kompletně jiným způsobem, než je normálně zvykem.

V praxi jako takové se funkcionální programování často kombinuje s ostatními paradigmaty, jakožto například s objektově orientovaným. Pokud tedy vypilujete své znalosti čistě funkcionálních jazyků, bude je poté možné aplikovat i v jiných případech, což bude mít za výsledek čistší kód a celkovou strukturu programu.

Konkrétním příkladem, kde je možné uplatnit znalosti ze světa funkcionálního programování, je například vývoj webových aplikací pomocí *JavaScript* knihovny *React*.

React je open source knihovna od společnosti Meta, určená k vývoji uživatelského a nativního rozhraní, která byla zveřejněna v roce 2013 (*React*, 2013/2023) a patří mezi nejpopulárnější frameworky pro vývoj front-endu (*State of JavaScript 2022*, b.r.).

Aplikace vyvíjené pomocí této knihovny se skládají z komponentů, které jsou do sebe dle libosti zanořovány. Komponent tvoří část uživatelského rozhraní, kde jednotlivé komponenty mohou mít svojí vlastní logiku a stylování. Komponent může být malý a plnit například funkci tlačítka nebo může být velký jako celá webová stránka (*Quick Start – React*, b.r.). Jedno ale mají tyto komponenty společné, v určité chvíli vrací HTML, jenž je následovně vykreslováno uživateli.

Knihovna jako taková si za roky používání prošla mnoha změnami, ale jedna z nich značně vyčnívá nad ostatními. V dřívějších verzích *Reactu* se komponenty deklarovaly pomocí tříd, ale tento způsob byl od verze 16.8 nahrazen deklarováním komponent pomocí funkcí (McGinnis, 2019).

Tato změna je fundamentálně založena na pilířích funkcionálního programování, a to následovně – definice komponenty je realizována pomocí funkce, která přijímá parametry (props) na základě, kterých je navráceno výsledné HTML. Funkce sama o sobě nemá v základu žádný stav, tudíž výsledek funkce záleží čistě na poskytnutých argumentech.

Funkcionální přístup se v posledních letech stává více a více populárním a jeho uplatnění je možné pozorovat i mimo webové frameworky, v jazycích jako je například *Java*, *C#* či *Ruby*.

5 Programovací jazyk Haskell

5.1 Historie

V roce 1978 přednesl John Backus svou přednášku o Turingově ceně s názvem „Může být programování osvobozeno od von Neumannova stylu?“ (Backus, 1978), v níž vyhlásil pomyslnou válku proti celému programátorskému podniku od hardwarové architektury nahoru. Backusova myšlenka inspirovala vývojáře k tvorbě jazyků Fortran a Backus Naur Form (BNF). Toto mělo za důsledek, že se na funkcionální programování začalo nahlížet jako na praktický programovací nástroj namísto pouhé matematické kuriozity (Hudak et al., 2007).

I v této době měly funkcionální jazyky bohatou historii, počínaje jazykem *Lisp*, který byl představen Johnem McCarthym na konci 50. let 19. století (McCarthy, 1960). Další klíčový okamžik nastal v roce 1964, kdy Peter Landin a Christopher Strachey poukázali na podstatnou důležitost lambda kalkulu pro modelovací, programovací jazyky a položili základy pro operační sémantiku (Landin, 1964). Na toto navázal David Turner s programovacím jazykem *SASL*, který vyvinul. Jednalo se o čistě funkcionální jazyk vyššího řádu s proměnnými v lexikální oblasti – obohacený o lambda kalkulus odvozený z Landinova *ISWIM* (Landin, 1966).

Na konci 70. a začátku 80. let se uskutečnilo něco doposud nevídaného, série seminárních publikací započala zájem o myšlenku „líných“ (call-by-need) funkcionálních jazyků jakožto hlavního zprostředkovatele pro psaní programů. Líné vyhodnocování jako takové bylo vyvíjeno nezávazně na sobě ve třech různých případech (Hudak et al., 2007). Dan Friedman a David Wise ve své publikaci z roku 1976 s názvem „Cons should not evaluate its arguments” nahlížejí na líné vyhodnocování z perspektivy jazyku *Lisp*. V další řadě se jedná o Peter Hendersona a James H. Morris Jr. s publikací z téhož roku s názvem „A lazy evaluator”, kde také používají variantu jazyku *Lisp*, v níž prokázali spolehlivost svého evaluátoru. A v poslední řadě David Turner, který použil *KRC* a jazyk *SASL*, jenž byl původně jazykem striktním, ale v roce 1976 se stal jazykem líným. Turner poukázal na krásu

programování s líným vyhodnocováním a to zejména při použití seznamů pro napodobení různých situací (Hudak et al., 2007).

V polovině 80. let tato veškerá aktivita vedla k nadšení v řadách výzkumníků, včetně autorů samotného jazyka *Haskell*, kteří projevili zájem jak o návrh, tak implementaci čistého a líného programovacího jazyku. Jedním z výsledků tohoto trendu byl jazyk *Miranda*, který byl navržen a vyvinut Davidem Turnerem. Jedná se o nástupce jazyků *SASL* a *KRC*. Jelikož oba jazyky postrádaly otypování, představil Turner v jazyce *Miranda* silné polymorfní otypování a odvozování typů. Což byly vlastnosti, které se prokázaly velmi užitečnými v *ML* (Hudak et al., 2007).

Ke klíčovému zlomu došlo na podzim v roce 1987, kdy se Peyton Jones zastavil na univerzitě Yale na rychlou návštěvu Paula Hudaka při cestě na FPCA (Functional Programming and Computer Architecture Conference). Po diskusi o aktuální situaci ve světě funkcionálního programování bylo dohodnuto, že uspořádají setkání během samotného FPCA, aby vzbudili zájem o vývoj nového funkcionálního jazyka. Toto setkání je později označováno jako začátek vývoje Haskellu (Hudak et al., 2007).

Z důvodu urychlení vývoje se dospělo k závěru, že bude nejjednodušší začít s již existujícím jazykem a upravovat ho dle potřeb. Jako hlavní adept se nabízel jazyk *Miranda* od Davida Turnera, který byl ve své době nejpokročilejším – čistý, dobře navrhnutý, s robustní implementací. Nicméně po snaze získat od Turnera povolení k adaptaci Mirandy skončil tento pokus neúspěchem. Haskell měl umožňovat komukoliv rozšířit nebo modifikovat jazyk samotný a v neposlední řadě mělo být možné Haskell dále distribuovat. Ale Turner byl úplně opačného názoru, důrazně se zasazoval o zachování jednotné jazykové syntaxe. Nepřál si, aby byly v oběhu různé dialekty Mirandy a požádal tvůrce Haskellu, aby si vytvořili vlastní jazyk, který bude dostatečně odlišný od Mirandy tak, aby nebylo možné tyto dva jazyky jakkoliv zaměnit. Toto bylo velmi podstatné z hlediska vývoje Haskellu. Ačkoliv to znamenalo vývoj jazyka od nuly, umožnilo to vývojářům svobodně uvažovat o více

radikálních přístupech k jednotlivým aspektům návrhu (Hudak et al., 2007) – díky odmítnutí ze strany Turnera je dnes Haskell tím čím je.

Prvním z řady významných rozhodnutí bylo vybrání jména pro vyvíjený programovací jazyk – mezi kandidáty se objevila jména jako: Semla, Vivaldi, Mozart, CFL atd. Po dlouhém uvážení se rozhodlo pokračovat se jménem *Curry* na počest matematika a logika Haskella B. Curryho, jehož práce inspirovala všechny autory Haskellu. Vzápětí si ale autoři uvědomili, že tímto jménem by sebe i jazyk vystavili řadě vtipných narážek, a rozhodli se tedy použít jméno *Haskell*. V následujících letech byl Haskell vyvíjen až do 1. dubna 1990, kdy byla publikována verze 1.0 (Hudak et al., 2007).

5.2 Inovace

Inovace v Haskellu přinášejí revoluční přístupy k řešení problémů v oblasti funkcionálního programování zdůrazňující efektivitu a čistotu kódu.

5.2.1 Syntaxe

Syntaxe jazyka Haskell je výsledkem pečlivé úvahy, jejímž cílem je spojit teoretickou přísnost s praktickou použitelností. Výbor Haskellu věnoval úsilí tomu, aby syntaxe byla nejen efektivní pro kompilátory, ale také čitelná a výrazná pro programátory. Jedním z charakteristických rysů syntaxe jazyka Haskell je její matematicky orientovaná struktura (Hudak et al., 2007). Tato návrhová volba odráží kořeny jazyka Haskell ve funkcionálním programování a jeho důraz na abstrakci a čistotu.

Další podstatnou součástí jazyka Haskell je jeho zacházení s funkcemi a aplikacemi. Jazyk Haskell umožňuje různé způsoby definování funkcí, přičemž každý z nich souvisle zapadá do zastřešujícího funkcionálního paradigmatu. Tato flexibilita při definování funkcí nejen podtrhuje závazek Haskellu k principům funkcionálního programování, ale také zvyšuje vyjadřovací schopnosti a sílu jazyka samotného.

5.2.2 Funkce a aplikování funkcí

Aplikace funkcí v jazyce Haskell je stručná a efektivní. Jazyk používá *„currying“*, kdy funkce přebírají argumenty po jednom. To programátorům umožňuje jejich částečné použití, což má za výsledek stručnější a flexibilnější kód. Například funkci přijímající dva argumenty lze částečně aplikovat s jedním argumentem a vytvořit tak funkci novou (*Currying - HaskellWiki*, b.r.).

```
1.   secti x y = x + y
2.   pridej5 = secti 5
```

V ukázce je definována funkce *secti*, která přijímá dva parametry *x* a *y*, jejich součet je výsledkem volání funkce. Na dalším řádku je ovšem definována funkce *pridej5*, která vychází již z předem definované funkce *secti*, kde je jako první parametr *x* poskytnuta hodnotu 5. Toto má za výsledek částečné vyhodnocení funkce, kde by taková definice odpovídala zápisu *pridej5 y = 5 + y*. Tento přístup je základní schopností Haskellu a přispívá k jeho eleganci a užitečnosti při řešení složitých programovacích problémů pomocí rozložení funkcionality do několika jednodušších funkcí.

V Haskellu je možné volat funkce dvěma různými způsoby. Prvním z nich je *prefixní* formou, která je uplatňována ve většině imperativních jazyků. Jedná se o případ, kdy jméno funkce následují její argumenty, které jsou funkci předány za účelem získání nějakého výsledku např. *secti 1 2*.

Druhým způsobem je použití takzvaných *infixních* operátorů, ty v Haskellu existují především z toho důvodu, že jeden z cílů Haskellu bylo, aby výrazy připomínaly co možná nejvíce matematiku (Hudak et al., 2007). A i z toho důvodu se primárně využívají při aritmetických, porovnávacích nebo logických operacích.

```
ghci> 1 + 1
ghci> 1 < 2
ghci> True && True
```

V této ukázce se jedná o znaky „+“, „<“ a „&&“. Haskell nám ale dává možnost konvertovat originální *infixní* operátor a použít ho *prefixní* formou pouhým obalením závorkami.

```
ghci> (+) 1 1
ghci> (<) 1 2
ghci> (&&) True True
```

Obě tyto použití vedou ke stejnému výsledku, z toho tedy vyplývá, že *infixní* operátory jakožto takové existují za účelem zlepšení čitelnosti kódu.

5.2.3 Dopad a vliv

Od samotného vydání má Haskell velký dopad na jedno z nejdůležitějších odvětví na světě, a to sice školství, což jak popisují ve svém článku přední představitelé Haskell komise, byl jedním z jeho explicitních cílů. Tomu ovšem v té době stál v cestě nedostatek textů, či robustních implementací, ze kterých by studenti mohli čerpat. Nicméně jak plynul čas, začalo Haskell pro své odborné učebnice využívat více a více autorů, díky čemuž se stal Haskell velmi cenným nástrojem pro vštěpení základů funkcionálního programování do mladých myslí. Avšak funkcionální programování není jediným případem použití Haskellu ve školství. Stal se také trendem pro vyučování diskrétní matematiky či logiky, k čemuž ve výsledku přispívá matematický vzhled a dojem, kterým se Haskell pyšní (Hudak et al., 2007).

Ačkoliv je složité říct naprosto přesně, že existuje spojitost mezi konkrétními programovacími jazyky a Haskellem nebo že by se jím daný jazyk inspiroval, poukazují představitelé Haskell komise na určité podobnosti mezi několika velmi populárními jazyky. Mezi příklady je zmíněn například *Python*, dynamicky typovaný jazyk určený pro skriptování, který si adaptoval notace pro generování seznamů. Dalším ze zmíněných je *Java*, jejíž používání typů s omezením (bounded types) je úzce spojeno s typovými třídami, které existují v Haskellu. V neposlední řadě je jazyk *C#* a jeho funkce LINQ (Language INtegrated Query), jež je založena na monadické komprehenzi, která se vyskytuje v jazyku Haskell (Hudak et al., 2007).

5.3 Líný programovací jazyk

Haskell je líný programovací jazyk a to znamená, že pokud mu není specificky poručeno jinak, Haskell nebude vykonávat funkce a počítat výrazy, dokud ho k tomu něco nepřinutí. Díky tomuto chování můžeme nad programy přemýšlet jako nad sérií transformací na datech. Toto nám také umožňuje používat zajímavé vlastnosti jako např. nekonečné datové struktury. Uvažme situaci, kdy existuje neměnný seznam čísel *xs = [1,2,3,4,5,6,7,8]* a funkce *dvojnasobek*, která vynásobí každý prvek v poli dvěma a poté vrátí nový seznam. Pokud je tedy požadováno vynásobit seznam číslem 8, vypadal by zápis následovně: *dvojnasobek(dvojnasobek (dvojnasobek (xs)))*. V imperativním jazyce by se program choval tak, že by funkce udělala kopii vstupního seznamu, vynásobila ho a vrátila výsledek a tímto způsobem ještě dvakrát. Nicméně v líném jazyce jako je Haskell zavolání funkce *dvojnasobek* se seznamem, aniž bychom okamžitě vyžadovali výsledek, vede k odloženému výpočtu. Ve své podstatě program naplánuje vykonání operace v pozdější době, kdy bude výsledek bezpodmínečně vyžadován. Toto chování je důvodem pro označení „líný jazyk“.

Předem uvažovaná situace by v Haskellu vypadala následovným způsobem: první funkce *dvojnasobek* informuje druhou funkci *dvojnasobek* o tom, že je potřeba provést výpočet, stejně informuje druhá funkce třetí, která vynásobí číslem 2 první prvek seznamu – číslo 1, což navrátí číslo 2. Tato hodnota je poté vynásobena druhou funkcí, načež vrátí 4 a poslední funkce obdobně udělá to samé a dostaneme odpověď, kde první prvek je roven 8. Tímto způsobem tedy můžeme v líném programovacím jazyce předat funkcím nějaká výchozí data a transformovat je do podoby, do které se chceme ve výsledku dostat (Lipovaca, 2011).

5.4 Typový systém

Na nejnižší úrovni počítač pracuje s bajty a nemá téměř žádnou jinou strukturu, ale typy nám dávají určitou úroveň abstrakce. Typy zapouzdřují tyto bajty a tím jim pro programátory dávají význam – například dovolují specifikovat, že určitá skupina bajtů je textovým řetězcem. Tato abstrakce umožňuje programátorům ignorovat tyto detaily nízké úrovně. Pokud je specifikováno, že určitá hodnota v programu je

textovým řetězcem, není nutné dále řešit podrobné detaily toho, jak jsou textové řetězce implementovány, je možné očekávat, že se bude chovat stejně jako všechny ostatní textové řetězce (O'Sullivan et al., 2008).

Haskell se pyšní tím, že je staticky typovaný, což značí, že již při kompilování jsou veškeré typy známé. Za předpokladu, že by v programu byla logická hodnota typu boolean dělena číslem, obdržíme chybovou hlášku a program se ani nezkompiluje. To je z hlediska programování dobře, jelikož takové chyby je z pravidla dobré odchytit co možná nejdříve, aby se minimalizovaly chyby, které by mohly nastat za běhu programu (Lipovaca, 2011).

5.4.1 Základní typy

V Haskellu jsou určité základní typy, se kterými se provádí většina operací. Mezi takové typy spadá například: *Char*, který reprezentuje jeden Unicode znak. *Bool* díky jemuž je možné reprezentovat logické hodnoty jako je *True* a *False*. Mezi základní typy reprezentující čísla Haskell poskytuje *Int*, který se používá pro celočíselné hodnoty se znaménkem a pevnou délkou, standard Haskellu zajišťuje, že *Int* má minimální velikost 28 bitů a maximální velikost je závislá na architektuře počítače – 32 nebo 64 bitů. Dalším číselným typem, které je vhodné zmínit je *Integer*, ten reprezentuje celé číslo se znaménkem, neomezené velikosti, ačkoliv je v porovnání s typem *Int* používaný výrazně méně, jelikož je dražší jak z hlediska výkonu, tak z hlediska spotřeby místa, při výpočtech s typem *Integer* nenastane situace, kdy by tiše přetekl. V neposlední řadě nám Haskell poskytuje typ *Double*, díky kterému můžeme reprezentovat desetinná čísla až do velikosti 64 bitů. Menší typ, který nám umožňuje reprezentovat desetinná čísla je *float*, nicméně jeho používání se nedoporučuje – autoři kompilátoru jazyka Haskell se soustředili na to, aby byl *Double* efektivnější, což ponechává *Float* výrazně pomalejší (O'Sullivan et al., 2008).

5.4.2 Odvozování typů

Kompilátor v Haskellu umí automaticky odvodit typy většiny výrazů v programu. Haskell umožňuje explicitně deklarovat typ jakékoliv hodnoty, nicméně díky odvozování typů je toto skoro ve všech případech dobrovolné a ne něco, co je po nás striktně vyžadováno (O'Sullivan et al., 2008). Stejně tomu je tak i u funkcí, kterým můžeme explicitně definovat jak typy parametrů, tak typ návratový. V případě funkcí se definování typů považuje za dobrou praxi (Lipovaca, 2011), která vede hlavně k čistšímu kódu, u kterého je na první pohled zřejmé, co se od dané funkce očekává.

5.4.3 Typové třídy

Jednou z nejsilnějších stránek Haskellu jsou typové třídy (typeclasses). Umožňují nám definovat rozhraní pro celou řadu různých typů. Tato funkčnost je jádrem některých základních funkcionalit jako například testování rovnosti nebo numerických operátorů. Jinými slovy pokud se bavíme o typových třídách v kontextu Haskellu, máme na mysli set funkcí, které mohou mít různou implementaci v závislosti na poskytnutém typu (O'Sullivan et al., 2008).

5.4.4 Polymorfismus

Další podstatnou funkcionalitou Haskellu je polymorfismus. Pokud již máte nějaké zkušenosti s objektově orientovanými jazyky, pravděpodobně se sami sebe ptáte, jak možné v čistě funkcionálním jazyce uplatnit polymorfismus, aniž bychom pracovali s třídami. V jazycích jako jsou například *Java* nebo *C++*, které spadají do kategorie objektově orientovaných jazyků, je myšlen podtypový polymorfismus. Ten umožňuje v rodičovské třídě definovat chování určitých funkcí, které následovně dědí potomkové této třídy, jež mohou toto chování upravit či rozšířit (O'Sullivan et al., 2008). Na druhou stranu v Haskellu existuje parametrický polymorfismus, který je z OOP možné znát pod pojmem generické typy a typové parametry.

Parametrický polymorfismus je pozorovatelný na následujícím příkladu, konkrétně se jedná o typový předpis funkce *last*.

```
ghci> :t last
last :: GHC.Stack.Types.HasCallStack => [a] -> a
```

Zde můžeme pozorovat použití příkazu „:t“, který vrací typ poskytnutého parametru. V našem případě se dotaz týká typového předpisu funkce *last* a jako odpověď je navráceno *[a] -> a*. Můžete pozorovat, že se zde nenachází žádný konkrétní typ jakožto *Int*, *Char* nebo *Bool*, namísto toho tato funkce spoléhá na parametrický polymorfismus s parametrem *a*. Pro kompilátor není podstatné, jaké typy tvoří strukturu poskytnutého seznamu nýbrž pouze to, že funkce *last* obdrží seznam skládající se z jakéhokoliv typu a vrátí jeden výsledek, který bude odpovídat typu prvku, ze kterého se poskytnutý seznam skládal.

6 Programování v jazyce Haskell

Tato kapitola je věnována podrobnému výkladu programování v jazyce Haskell, od základních konceptů, přes funkcionalitu a syntaxi, až po pokročilé techniky a principy, které tento jazyk činí unikátním.

6.1 Základní aritmetika

Pomocí základních infixních funkcí jako „+“, „-“, „/“ , „*“ nebo „^“ je možné v Haskellu provádět matematické operace. Při výpočtech platí základní pravidla pro matematické operace, kde násobení a dělení má přednost před sčítáním a odečítáním, v případě potřeby je možné na výrazy uplatnit závorky, za účelem zvýšení jejich priority.

```
ghci> 10 + 5
15
ghci> 10 - 5
5
ghci> 10 / 5
2.0
ghci> 10 * 5
50
ghci> 10 + 5 * 2
20
ghci> (10 + 5) * 2
30
```

Tyto infixní funkce je možné zapsat prefixní formou, což sice není doporučováno, jelikož to vede ke snížení čitelnosti výrazů, ale je dobré o této možnosti vědět a ukázat si ji v praxi.

```
ghci> (+) 10 5
15
ghci> (-) 10 5
5
ghci> (/) 10 5
2.0
ghci> (*) 10 5
50
ghci> (+)((*) 5 2) 10
20
ghci> (*)((+) 10 5) 2
30
```

Při provádění aritmetických výpočtů je potřeba být ostražitý v případě záporných čísel, což je demonstrováno následující ukázkou.

```
ghci> 10 * -5
```

Očekávaným výsledkem tohoto výrazu je samozřejmě hodnota *-50*, nicméně ghci zaplaví konzolové okno touto chybovou hláškou:

```
<interactive>:1:1: error:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix `-' [infixl 6] in
the same infix expression
```

K této chybě dochází, jelikož Haskell interpretuje znaménko „-“ jako operátor pro odečítání, a ne jako znak pro záporná čísla, z toho důvodu je výraz *10 * -5* označen jako nekompletní operace odečítání. Proto je dobré si pamatovat, že v Haskellu je potřeba záporná čísla obalit do závorek, tak aby nejdříve proběhla negace daného čísla.

```
ghci> 10 * (-5)
-50
```

6.2 Logické hodnoty

Podstatnou součástí každého programovacího jazyku je schopnost ovládat tok programu, a tak je tomu i v Haskellu. Stejně jako v ostatních programovacích jazycích, je k dispozici typ Bool, který může nabývat dvou hodnot True a False – na rozdíl od většiny jazyků, v Haskellu tyto hodnoty začínají velkým písmenem.

Obdobně k ostatním programovacím jazykům Haskell poskytuje infixní logické operátory „||“ a „&&“, díky kterým můžeme tvořit složené logické výrazy. Dalším účinným nástrojem je prefixní funkce *not*, kterou využíváme k negování logického výrazu.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not (True && False)
True
```

Do kategorie logických hodnot spadá i testování rovnosti pomocí infixních funkcí „==“ a „/=“, nebo porovnávání pořadí pomocí dalších infixních funkcí „<“, „>“, „<=“ a „>=“.

```
ghci> 10 == 10
True
ghci> 10 /= 10
False
ghci> [10] == [10]
True
```

Je důležité dbát na to, že porovnávání v Haskellu je založeno na typové třídě, a tudíž je povoleno porovnávat jen výrazy stejného typu.

```
ghci> 10 == "text"
<interactive>:3:1: error:
    • No instance for (Num String) arising from the literal ‘10’
    • In the first argument of ‘(==)’, namely ‘10’
      In the expression: 10 == "text"
      In an equation for ‘it’: it = 10 == "text"
```

6.3 Datové struktury

I přestože je možné si do jisté míry vystačit s obyčejnými typy, poskytuje Haskell příležitost pracovat s o něco komplexnějšími datovými strukturami, které programátorům ulehčují práci při manipulaci s daty.

6.3.1 Seznamy

Seznamy jsou nejpoužívanější datovou strukturou, kterou Haskell poskytuje. Seznamy v Haskellu jsou homogenní, což znamená, že jednotlivé prvky daného seznamu mohou nabývat jediného typu – například seznam čísel nebo znaků – kombinování typů není povoleno (Lipovaca, 2011).

Definování seznamu je velmi podobné ostatním jazykům. Jednotlivé prvky jsou odděleny čárkami a všechny tyto prvky jsou následovně obehnány hranatými závorkami.

```
ghci> [1,2,3,4,5]
[1,2,3,4,5]
ghci> ['a','h','o','j',' ','s','v','e','t','e']
"ahoj svete"
```

6.3.1.1 Užitečné funkce pro práci se seznamem

Jelikož jsou seznamy základním stavebním kamenem pro manipulaci s daty, existuje celá řada funkcí, které práci s nimi ulehčují. Textové řetězce jako takové jsou pouze zkráceným zápisem pro seznamy znaků. To znamená, že veškeré funkce vytvořené pro práci se seznamy je možné aplikovat i na textové řetězce.

Jednou ze základních operací je schopnost spojit dva seznamy dohromady, toto je v Haskellu realizováno pomocí infixní funkce „++“.

```
ghci> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Nicméně je potřeba si uvědomit, že při spojování dvou seznamů pomocí operátoru „++“ musí Haskell projít celý seznam na levé straně od operátoru a poté k němu připojit seznam na straně pravé. Tak je tomu i kdybychom připojovali seznam s jediným prvkem *[1,2,3] ++ [4]*. Toto není problém v případě, kdy seznamy nenabývají velké délky, ale v případě, kdybychom měli seznam o délce několika miliónů na levé straně od operátoru, tak tato operace může trvat velmi dlouho. Pro situace, kdy potřebujeme k seznamu přidat jediný prvek existuje operátor „:”. Prvek je vždy přidán na samotný začátek seznamu a na rozdíl od operátoru „++“ je tato operace okamžitá (Lipovaca, 2011).

```
ghci> 1:[2,3,4]
[1,2,3,4]
ghci> 'A':['h','o','j']
"Ahoj"
```

Všimněte si toho, že operátor „:“ přebírá pouze jediný prvek. V případě operátoru „++“ je ale pokaždé potřeba obehnat připojovaný prvek hranatými závorkami.

V Haskellu jako v ostatních programovacích jazycích je možné přistupovat k jednotlivým prvkům pomocí indexu, obdobně se prvky v seznamu indexují od 0. Přístup k prvkům je realizován pomocí operátoru „!!“.

```
ghci> [1,2,3,4,5] !! 0
1
```

Nicméně je důležité si uvědomovat délku seznamu ze kterého se čerpá, jelikož při poskytnutí indexu, který je větší než poslední index, vrátí ghci chybovou hlášku.

```
ghci> [1,2] !! 2
*** Exception: Prelude.!!: index too large
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1368:14 in base:GHC.List
  tooLarge, called at libraries/base/GHC/List.hs:1378:50 in base:GHC.List
  !!, called at <interactive>:18:8 in interactive:Ghci14
```

Zajímavou vlastností Haskellu je, že seznamy je možné porovnávat pomocí funkcí rovnosti či pořadí. Respektive záleží na tom, jestli je možné porovnávat prvky, které jsou v seznamu obsaženy.

```
ghci> [1,2,3] == [1,2,3]
True
ghci> [1,2,3] /= [1,2]
True
ghci> [3,2,1] > [3,2]
True
ghci> [4,2,1] > [3,2,1]
True
```

Pokud seznamy používají infixní funkce pro porovnávání pořádí, probíhá lexografické porovnávání jednotlivých prvků s odpovídajícími indexy mezi jednotlivými seznamy – nejdříve se zkontroluje první prvek a pokud je námi stanovená podmínka vyhodnocena jako *True* posune se porovnávání na druhý prvek, a tak pokračuje až do konce samotného seznamu (Lipovaca, 2011).

Tato část je věnována funkcím, které umožňují bezpečnější přístup k prvkům seznamu. První z těchto funkcí je *head*, jak název napovídá vrací hlavu seznamu, což je název pro první prvek. K ní má Haskell obdobnou funkci, která dělá přesný opak, a to sice *tail*, která vrací seznam bez hlavy (Lipovaca, 2011).

```
ghci> head [1,2,3,4]
1
ghci> head "Ahoj"
'A'
ghci> tail [1,2,3,4]
[2,3,4]
ghci> tail "Ahoj"
"hoj"
```

V Haskellu existuje i funkce opačného chování s názvem *last*, díky ní můžeme pohodlně přistoupit k poslednímu prvku v seznamu. Jak tomu bylo s funkcí *head* a *tail*, i pro funkci *last* Haskell poskytuje sesterskou funkci s názvem *init*, jejíž voláním získáme seznam bez posledního prvku (Lipovaca, 2011).

```
ghci> last [1,2,3,4]
4
ghci> last "Ahoj"
'j'
ghci> init [1,2,3,4]
[1,2,3]
ghci> init "Ahoj"
"Aho"
```

I u této čtveřice přístupových funkcí je třeba dbát na to, aby poskytnutý seznam obsahoval alespoň jeden prvek, jinak ghci vypíše obdobnou chybovou hlášku.

```
ghci> head []
*** Exception: Prelude.head: empty list
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1646:3 in base:GHC.List
  errorEmptyList, called at libraries/base/GHC/List.hs:85:11 in
base:GHC.List
  badHead, called at libraries/base/GHC/List.hs:81:28 in base:GHC.List
  head, called at <interactive>:39:1 in interactive:Ghci30
```

Mezi funkce, které nám svým způsobem umožňují přístup k prvkům v seznamu jsou *take* a *drop*. Obě funkce očekávají na svém vstupu dva argumenty – číslo a seznam. V případě funkce *take* stanovuje první číselný parametr, kolik prvních prvků má být vráceno. Opačně je tomu u funkce *drop*, kde číselný parametr určuje, kolik prvků má být ze seznamu vyloučeno (Lipovaca, 2011).

```
ghci> take 2 [1,2,3,4]
[1,2]
ghci> take 10 [1,2,3,4]
[1,2,3,4]
ghci> take 2 []
[]
ghci> drop 2 [1,2,3,4]
[3,4]
ghci> drop 10 [1,2,3,4]
[]
ghci> drop 2 []
[]
```

Haskell v neposlední řadě poskytuje funkce pro případné kontroly vlastností seznamu. Jmenovitě se jedná o funkci *length*, která vrací číselnou délku poskytnutého seznamu nebo funkce *null*, pomocí které je možné kontrolovat, zda je poskytnutý seznam prázdný. V takovém případě obdržíme logickou hodnotu *True* či *False* (Lipovaca, 2011).

```
ghci> length [1,2,3,4]
4
ghci> null [1,2,3,4]
False
ghci> null []
True
```

Do této kategorie by se dala zařadit i funkce *elem*, jejímž aplikováním je získána logická hodnota. Ta reprezentuje, zda se poskytnutý prvek nachází v daném seznamu. Přestože je tato funkce prefixní, používá se povětšinou infixním způsobem, jelikož se tak zvyšuje čitelnost kódu (Lipovaca, 2011).

```
ghci> 1 `elem` [1,2,3,4]
True
ghci> 10 `elem` [1,2,3,4]
False
```

Pomocí funkcí jako je *minimum* či *maximum* můžeme v seznamu, v němž se nacházejí prvky, které můžeme nějakým způsobem porovnat, obdržet hodnotu odpovídající jménu funkce (Lipovaca, 2011).

```
ghci> maximum [1,2,3,4]
4
ghci> minimum [1,2,3,4]
1
ghci> maximum "Hello"
'o'
ghci> minimum "Hello"
'H'
```

Při porovnávání číselných hodnot v seznamu je výsledek z ukázky předvídatelný, nicméně při aplikování funkce *minimum* a *maximum* na textové řetězce nemusí být výsledek na první pohled zřetelný. Každý znak má definovanou Unicode hodnotu, což je číselná hodnota, pomocí které je možné bezpečně identifikovat jakýkoliv znak. A jelikož se jedná o číselné hodnoty, může je Haskell při zavolání funkce *minimum* nebo *maximum* porovnat a na jejich základě vrátit odpovídající výsledek.

V poslední řadě Haskell zpřístupňuje funkce pro číselné operace na seznamu. Jmenovitě se jedná o funkce *sum* a *product*. Kde *sum* vrací výslednou sumu všech prvků v seznamu a *product* vrací odpovídají součin všech prvků (Lipovaca, 2011).

```
ghci> sum [1,2,3,4]
10
ghci> product [1,2,3,4]
24
ghci> product [0,1,2,3,4]
0
```

6.3.1.2 Rozsahy v seznamech (ranges)

Rozsahy se v Haskellu využívají především k vygenerování lineární sekvence dat. Existují různé způsoby zápisu, kde každá plní jiný úkol. Tato vlastnost seznamů je zprostředkována operátorem „..“, který ve své podstatě prochází námi definované hodnoty. To tedy znamená, že je možné použít rozsahy jen v případě, kdy je možné určit v námi generované sekvenci hodnotu následujícího prvku – jedná se zejména o sekvence čísel či znaků (O'Sullivan et al., 2008).

Prvním ze zápisů odpovídá předpisu *[<první>..<poslední>]*, kde výsledkem této syntaxe je uzavřený interval, který obsahuje oba koncové body.

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Druhý předpis, který lze použít, obsahuje navíc druhý parametr v námi požadované sekvenci *[<první>, <druhý>..<poslední>]*. Pomocí rozdílu mezi druhým a prvním parametrem dochází k výpočtu pomyslného kroku, který určuje danou posloupnost sekvence.

```
ghci> [1,3..10]
[1,3,5,7,9]
ghci> ['a','c'..'z']
"acegikmoqsuwy"
```

K oběma zmíněným předpisům existují jejich verze, které se zbavují *<posledního>* parametru v sekvenci. Tímto způsobem jsme schopni velmi lehce generovat nekonečné seznamy, které lze různými způsoby používat. V následující ukázce je zobrazen příklad, který používá funkci *take* a nekonečný seznam pro zjištění prvních deseti sudých čísel.

```
ghci> take 10 [0,2..]
[0,2,4,6,8,10,12,14,16,18]
```

6.3.1.3 Generátory seznamů (list comprehensions)

Za předpokladu, že jste někdy absolvovali matematický kurz, je vysoce pravděpodobné, že jste se již setkali s pojmem intenzionální zápis množin. V praxi se využívají pro budování specifičtějších sad z obecných setů. Zde je ukázka, jak by vypadal zápis pro prvních deset sudých, přirozených čísel – $S = \{2 \cdot x | x \in N, x \leq 10\}$. V tomto zápisu se část před znakem „|“ nazývá výstupní funkce, *x* je pro nás proměnná, *N* je vstupní množina a *x <= 10* je predikát. To znamená, že výsledná množina obsahuje dvojnásobek všech přirozených čísel, které uspokojují námi definovaný predikát (Lipovaca, 2011).

V minulém příkladu bylo demonstrováno, že stejného výsledku je možné dosáhnout kombinací funkce *take* a nekonečného seznamu. Ale tento přístup lze použít jen za předpokladu, že generovaný seznam je možné definovat pomocí lineární posloupnosti. Pokud by bylo potřeba vygenerovat komplikovanější množinu prvků, umožňuje Haskell využít generátor seznamů. Pro ilustraci syntaxe se podíváme na příklad získání prvních deseti sudých čísel, tentokrát za využití generátoru seznamu.

```
ghci> [2*x | x <- [0..9]]
[0,2,4,6,8,10,12,14,16,18]
```

Zde je možné pozorovat, že vstupní množinou je rozsah mezi čísly 0 a 9, kde se postupným procházením ukládá hodnota do proměnné *x*, na kterou je následovně aplikována výstupní funkce a dochází k součinu proměnné *x* s číslem 2. V následující ukázce zkusíme tento příklad upravit, tak abychom získali pouze sudá čísla, která jsou větší nebo rovna číslu 10.

```
ghci> [2*x | x <- [0..9], 2*x >= 10]
[10,12,14,16,18]
```

Toho lze snadno dosáhnout pomocí doplnění predikátu, který je testován před aplikováním výstupní funkce. Tato základní syntaxe lze podle potřeby rozšířit o další vstupní množiny či predikáty, se kterými se pracuje.

```
ghci> [x*y | x <- [1,2,3], y <- [10,20,30]]
[10,20,30,20,40,60,30,60,90]
```

V této ukázce dochází ke všem možným součinům dvou seznamů. Nejprve je do proměnné *y* uložena hodnota 10 a do proměnné *x* hodnota 1, po aplikování výstupní funkce dochází k přesunu v proměnné *x* na hodnotu 2 a v proměnné *y* pro tuto chvíli zůstává hodnota 10. Tímto způsobem se proces opakuje ještě jednou, kdy se generátor dostane na konec prvního seznamu a dochází k nastavení hodnoty proměnné *y* na další prvek množiny – 20, poté dojde k uložení hodnoty 1 do proměnné *x*. Tímto způsobem se proces opakuje dle délky vstupních množin, dokud nenastane vyčerpání všech prvků všech množin.

Generátory seznamů mohou také obsahovat libovolný počet predikátu, které je potřeba uspokojit před aplikováním výstupní funkce. Pro aplikování výstupní funkce musí pro daný prvek platit všechny definované predikáty (Lipovaca, 2011).

```
ghci> [x | x <- [0..10], x /= 3, x /= 7]
[0,1,2,4,5,6,8,9,10]
```

6.3.2 Uspořádané n-tice (tuple)

Uspořádané n-tice jsou kolekce s pevně danou velikostí, kde každý prvek může nabývat jiného typu – to je podstatný rozdíl v porovnání se seznamem, kde musí být všechny prvky stejného typu. Jednotlivé typy n-tice se obyčejně označují pomocí číselného prefixu, který určuje z kolika prvků se daná n-tice skládá – 2-tice (známá také jako pár) s dvěma prvky, 3-tice (triple) s třemi prvky nebo například 5-tice s pěti prvky, i přestože není omezeno jakého rozměru může n-tice nabývat, v praxi se využívá především pár a triple. V Haskellu existuje výjimka, a to taková, že neexistuje n-tice, která by měla pouze jeden prvek (O'Sullivan et al., 2008).

N-tice se především využívají, pokud je potřeba dohromady uchovat více informací různých typů. Například jméno a rok narození známých osobností.

```
ghci> ("Ryan","Gosling",1980)
("Ryan","Gosling",1980)
```

Při práci s n-ticemi je důležité uvědomit si, že typ n-tice vyjadřuje počet, pozice a typy jejích prvků. To znamená, že n-tice obsahující různé počty nebo typy prvků mají odlišné typy, stejně jako n-tice, jejichž typy se vyskytují v různém pořadí (O'Sullivan et al., 2008).

```
ghci> :t (True, "Ahoj")
(True, "Ahoj") :: (Bool, String)
ghci> :t ("Ahoj", True)
("Ahoj", True) :: (String, Bool)
```

6.3.2.1 Užitečné funkce pro práci s n-ticemi

Hlavní dvě funkce, které je potřebné znát při práci s n-ticemi jsou funkce *fst* a *snd*, díky nímž je možné přistupovat k prvnímu a druhému prvku.

```
ghci> fst (True, False)
True
ghci> snd (True, "Ahoj")
"Ahoj"
```

6.4 Funkce

Tato kapitola je věnována hlavnímu stavebnímu bloku funkcionálního programování – funkcím. V této kapitole jsou řádky kódu v ukázkách očíslovány podle běžné praxe ve většině vývojových editorů.

6.4.1 Základní syntaxe

Napíšeme si velmi jednoduchou funkci *dvojnasobek*, která bude na svém vstupu očekávat jeden celočíselný parametr a vracet jeho dvojnásobek.

```
1.   dvojnasobek x = x + x
```

Na ukázce je vidět, že definování funkce je velmi podobné jejímu volání. Definice začíná názvem dané funkce, který následují parametry oddělené mezerami. Avšak na rozdíl od volání funkce se zde nachází rovnítko, za kterým je tělo funkce, kde je definována logika. Na první pohled je vidět, že funkce nemá explicitně definovaný typ, ale kvůli čitelnosti kódu je dobré ho doplnit.

```
1.   dvojnasobek :: Int -> Int
2.   dvojnasobek x = x + x
```

Typový předpis funkce se obvykle píše před implementaci samotné funkce a skládá se z názvu funkce, který odpovídá funkci, jejíž předpis definujeme. Názvu následuje dvojice znaků „::“ a za nimi typ vstupního a výstupního parametru, který je oddělený šipkou „->“.

Pokud by bylo požadováno, aby funkce přijímala jakékoliv číslo, musel by být typový předpis upraven následujícím způsobem.

```
1.   dvojnasobek :: Num a => a -> a
2.   dvojnasobek x = x + x
```

Tato definice se liší od té minulé, a to je z toho důvodu, že typ *a* je členem typové třídy *Num*. Zbytek je podobný a značí, že stejného typu *a* nabývá jak vstupní parametr, tak výsledek volání funkce.

V Haskellu je běžnou praxí definovat jednoduché funkce, které jsou poté skládány do komplexnějších celků. Tento běžný vzor demonstruje následující funkce *dvojnasobky*.

```
1.   dvojnasobek :: Num a => a -> a
2.   dvojnasobek x = x + x
3.
4.   dvojnasobky :: Num a => a -> a -> a
5.   dvojnasobky x y = dvojnasobek x + dvojnasobek y
```

V této velmi jednoduché ukázce je volána dvakrát funkce *dvojnasobek* s parametry *x* a *y.* Následovně je součet jejich výsledků navrácen.

Při volání *dvojnasobky 1 2* je nejprve zavolána funkce *dvojnasobek* s argumentem 1, která vrátí 2, protože *1 + 1 = 2*. Poté je zavolána funkce *dvojnasobek* s argumentem 2, což vrátí 4, jelikož *2 + 2 = 4*. Výsledné hodnoty jsou pak sečteny, takže *2 + 4 = 6*. Celý výraz *dvojnasobky 1 2* tedy vrátí hodnotu 6.

6.4.2 Principy rozvětvení kódu s if-then-else

Mezi základní principy větvení kódu je ve většině programovacích jazyků dostupný příkaz *if*. I v Haskellu je tento koncept k dispozici pro základní rozhodovací operace. Nicméně na rozdíl od ostatních programovacích jazyků je v Haskellu nutností, aby výraz *if* byl doplněn i o *else* blok. Na ukázku je demonstrována funkce *zdvojnasobPokudMale*, která očekává číselný parametr a v závislosti na hodnotě vrací hodnotu dvojnásobnou či stejnou.

```
1.   zdvojnasobPokudMale x = if x > 100
2.                         then x
3.                         else x*2
```

V ukázce probíhá testování hodnoty oproti číslu 100, pokud je tato podmínka splněna, vrací se původní hodnota parametru *x*. V opačném případě je vrácena hodnota, která je vynásobena číslem 2.

Při volání *zdvojnasobPokudMale 99*, dojde k vyhodnocení podmínky *x > 100* jako False, jelikož *99 > 100* neplatí. Z toho důvodu uplatní funkce else blok, a tím tedy navrátí hodnotu *99 * 2* neboli 198.

V Haskellu je možné klauzule *if-then-else* i vnořovat, díky čemuž je možné dosáhnout komplexnější logiky.

```
1.   popisCislo x =
2.       if x > 0
3.       then
4.           if even x
5.           then "Kladne a sude"
6.           else "Kladne a liche"
7.       else
8.           if x == 0
9.           then "Nula"
10.          else "Zaporne"
```

6.4.3 Porovnávání vzorů (pattern matching)

Jedním z prvních syntaktických konstruktů k představení je porovnávání vzorů. Jak už název napovídá jedná se o definování vzorů, kterým by měla nějaká data odpovídat. Následovně probíhá kontrola, zda tomu tak opravdu je. V případě, že data splňují definovaný vzor, probíhá vykonání námi určené logiky (Lipovaca, 2011). Tato funkcionalita bude demonstrována na jednoduchém příkladu.

```
1.   popisCislo2 1 = "Jedna"
2.   popisCislo2 2 = "Dva"
3.   popisCislo2 3 = "Tri"
4.   popisCislo2 _ = "Cislo neni v rozsahu 1-3"
```

V ukázkovém příkladu je několikrát definována funkce *popisCislo2*, namísto toho, aby byl definován poskytnutý parametr formou proměnné jako tomu bylo v předchozích ukázkách, vypíšeme určité vzory, na které je potřeba rozdílně reagovat. V případě, kdy dojde k zavolání funkce *popisCislo2* s parametrem 1, bude navrácen výsledek „Jedna", jelikož dojde k přiřazení vzoru, který je explicitně definován. Obdobně se program chová i pro parametry 2 a 3. Nicméně v případě,

kdy poskytnutý argument neodpovídá žádnému explicitně definovanému vzoru, dojde k zachycení vzoru v případě znaku „_“, který se označuje jako "wildcard pattern" neboli vzor zástupného symbolu. Tento vzor slouží jako univerzální odpovídač, který zachytí veškeré hodnoty, které neodpovídají žádnému z předchozích specifických vzorů. Je klíčové mít na paměti, že porovnávání vzorů v Haskellu probíhá od shora dolů. Z toho důvodu je nezbytné umístit symbol „_“, který zachytí všechny zbývající hodnoty, až za všechny specificky definované vzory. Zde je ukázka kódu, kde nastává chyba při porovnávání vzorů.

```
1.   popisCislo2 _ = "Cislo neni v rozsahu 1-3"
2.   popisCislo2 1 = "Jedna"
3.   popisCislo2 2 = "Dva"
4.   popisCislo2 3 = "Tri"
```

I v případě volání funkce *popisCislo2* s parametry 1-3 dojde k univerzálnímu porovnání vzorů na prvním řádku a jako výsledek je navrácen textový řetězec „Cislo neni v rozsahu 1-3“.

Porovnávání vzorů lze aplikovat i při práci se seznamy nebo dokonce n-ticemi.

```
1.   popisSeznam [] = "Seznam je prazdny"
2.   popisSeznam [prvni] =
3.       "Seznam ma jeden prvek: " ++ show prvni
4.   popisSeznam [prvni, druhy] =
5.       "Seznam ma dva prvky: " ++ show prvni ++ " a " ++ show druhy
6.   popisSeznam _ = "Seznam ma vice nez dva prvky"
```

V ukázce jsou definovány vzory pro funkci *popisSeznam*, kde je specifikovaný případ, kdy je seznam prázdný pomocí prázdných hranatých závorek *[ ]*. Poté následují dva vzory, které kontrolují velikost seznamu pomocí dekonstrukce. V případě, že seznam obsahuje pouze jeden prvek dojde k extrahování prvku, který se nachází na prvním indexu a jeho následovnému uložení do proměnné *prvni*, ke které je možné následovně přistupovat v těle funkce. Obdobně je tomu v dalším vzoru, kde jediný rozdíl je ten, že poskytnutý seznam má velikost přesně dvou prvků. Nicméně opět dochází k dekonstrukci, tentokrát do dvou proměnných *prvni* a *druhy*.

Posledním vzorem je opět vzor univerzální, který je určen k zachycení ostatních případů.

Pokud bylo potřeba striktně kontrolovat pouze velikost seznamu, aniž by bylo nutné pracovat s danými hodnotami, je možné v Haskellu opět použít univerzální odpovídač, ale tentokrát místo konkrétních proměnných v samotném vzoru seznamu.

```
1.  popisVelikostSeznamu [] = "Seznam je prazdny"
2.  popisVelikostSeznamu [_] = "Seznam ma jeden prvek"
3.  popisVelikostSeznamu [_, _] = "Seznam ma dva prvky"
4.  popisVelikostSeznamu _ = "Seznam ma vice nez dva prvky"
```

Obdobným způsobem lze uplatnit porovnávání vzorů v případě n-tic. Haskell poskytuje při práci s n-ticemi funkce *fst* a *snd*, díky kterým je možné přistupovat k prvnímu a druhému prvku n-tice. Pomocí vzorů je velmi jednoduché nadefinovat funkci *treti*, která bude pro n-tice velikosti 3 vracet hodnotu třetího prvku.

```
1.  treti (_, _, x) = x
```

6.4.4 Stráže (guards)

Vzory zajišťují, že některá data splňují danou formu a umožňují její dekonstrukci, ale na druhé straně se nachází stráže, kteří nám umožňují otestovat, jestli určité hodnoty splňují námi definovanou podmínku. Tento syntaktický konstrukt je velmi podobný *if-then-else* klauzulím. Nicméně stráže jsou mnohem čitelnější v případech, kdy máme mnoho podmínek, které potřebují být ošetřeny (Lipovaca, 2011).

Pomocí ukázky si vysvětlíme jejich syntaktickou strukturu. Vytvoříme funkci, která na svém vstupu přijímá číslo reprezentující výšku, ze které předmět padá volným pádem a naším úkolem bude vyhodnotit, zda čas do nárazu je menší než 1 sekunda – padá rychle, v intervalu 1 až 5 sekund – padá střední rychlostí, a cokoliv, co padá déle než 5 vteřin – padá pomalu.

```
1.   casPadu draha
2.       | sqrt(2 * draha / 9.81) < 1 = "Objekt pada rychle"
3.       | sqrt(2 * draha / 9.81) < 5 = "Objekt pada stredne rychle"
4.       | otherwise  = "Objekt pada pomalu"
```

K výpočtu používáme vzorec pro volný pád: $t = \sqrt{\frac{2d}{g}}$, jehož aplikací získáme dobu trvání pádu v sekundách. Pokud se podíváme na syntaxi stráží, zjistíme, že prvním rozdílem v definici samotné funkce je chybějící rovnítko za vstupními parametry, které je nahrazeno bezprostředním znakem „|“, ten následuje podmínka, která nabývá logické hodnoty True nebo False. Podmínka je následovně ukončena rovnítkem, které následuje tělo funkce, jež odpovídá dané podmínce.

Pro lepší pochopení funkčnosti zavolejme tuto funkci se vstupním parametrem 10 (metrů). Do proměnné *draha* je vložena hodnota 10 a začíná testování pro jednotlivé stráže. První výraz je vyhodnocen jako False, jelikož po aplikování vzorce vychází hodnota přibližně 1.43s, a to nesplňuje námi specifikovanou podmínku, kdy čas je menší než 1s. V následujícím kroku se postupuje k druhé podmínce, kde je výpočet opětován a výsledek 1.43s zůstává stejný. Tentokrát je podmínka splněna, jelikož *1.43 < 5* je vyhodnoceno jako True. Splnění této podmínky vede k aplikaci těla funkce za splněnou podmínkou, v tomto případě k vrácení textového řetězce „Objekt pada stredne rychle“.

V případě, kdy nebyla splněna ani jedna ze dvou podmínek, je riskována chyba. Z toho důvodu je jako poslední strážný nastaven na klíčové slovo *otherwise*, kde *otherwise* je vždy definováno jako *True*, což má za důsledek odchycení všech podmínek, které nebyly doposud splněny.

6.4.5 Where

V minulém příkladu byla pomocí stráží definována funkce *casPadu*, jejímž aplikováním je možné určit, jak dlouho předmět padá volným pádem. Nicméně způsob, kterým je prováděna kontrola jednotlivých podmínek, se zbytečně opakuje. Pro programátory opakování většinou znamená, že existuje lepší způsob zápisu, který následuje princip DRY – don't repeat yourself neboli neopakuj se. Z toho důvodu Haskell poskytuje syntaktický konstrukt *where*. Pomocí něho je možné definovat lokální proměnné, které jsou následovně použitelné jak uvnitř samotných podmínek stráží, tak i v tělech funkcí.

```
1.   casPadu draha
2.     | cas < pomalu  = "Objekt pada rychle:" ++ show cas ++ "s"
3.     | cas < stredne = "Objekt pada stredne rychle:" ++ show cas ++ "s"
4.     | otherwise     = "Objekt pada pomalu:" ++ show cas ++ "s"
5.       where
6.           pomalu = 1
7.           stredne = 5
8.           g = 9.81
9.           cas = sqrt(2 * draha / g)
```

Kód je nyní mnohem přehlednější a hlavně kratší, což vede ke snížené chybnosti kódu. Přidaným bonusem je i snadné zobrazení vypočítaného času na výstupu.

6.4.6 Let

Pomocí výrazu *let*, můžeme kdekoliv zavést lokální proměnné. Zápis vypadá následovně – *let <proměnné> in <výraz>*. Klíčové slovo *let* označuje začátek bloku s deklarací proměnných a klíčové slovo *in* začíná blok pro výraz. Proměnné definované v deklaračním bloku jsou přístupné jak v deklaračním bloku samotném, tak v bloku s výrazem (O'Sullivan et al., 2008).

Syntaxi je demonstrována na funkci *vypocitejPreprodejniCenu*, jež vypočítá cenu předmětu, který je potřeba přeprodat. Funkce bude na svém vstupu očekávat původní cenu reprezentovanou jako číslo a stav věci, kterou se chystáme prodat. Ten bude reprezentován jakožto textový řetězec. Jelikož předmět je přeprodávaný, je nutné novou cenu snížit na 90% ceny původní a následovně je potřeba přiřadit jednotlivým stavům (nový, dobrý, ujde a sešlý) adekvátní číselné násobky v rozmezí 0-1, které budou cenu dodatečně ovlivňovat.

```
1.   vypocitejPreprodejniCenu puvodniCena stav =
2.       let preprodejniCena = puvodniCena * 0.9
3.           faktorStavu = if stav == "novy"
4.                             then 1.0
5.                             else if stav == "dobry"
6.                                 then 0.8
7.                                 else if stav == "ujde"
8.                                     then 0.5
9.                                     else if stav == "sesly"
10.                                        then 0.2
11.                                        else 0.1
12.      in preprodejniCena * faktorStavu
```

V ukázce je možné pozorovat deklarování dvou lokálních proměnných. První je deklarována pouhým součinem a druhá pomocí výrazu *if-then-else*. Na deklarační blok navazuje blok *in* s výrazem součinu mezi těmito dvěma proměnnými. Ten je následovně funkcí vracen jako výsledek.

Dále je možné si povšimnout, že příklad by bylo jednoduché přepsat, tak aby používal syntaktický konstrukt *where*. Takže si aktuálně možná kladete otázku, jaký je rozdíl mezi *let* a *where*. *Where* je použitelný pouze v kontextu funkce, kdežto *let* je výraz, kde z nátury Haskellu vyplývá, že výraz je použitelný v každé situaci (Lipovaca, 2011).

```
1.   vypocitejSlevy :: [(Double, Double)] -> [Double]
2.   vypocitejSlevy polozky =
3.     [ konecnaCena
4.       | (cena, sleva) <- polozky,
5.       let konecnaCena = cena * (1 - sleva / 100),
6.       konecnaCena > 50
7.     ]
```

Zde je příklad funkce, která používá výraz *let* při generování seznamu. Jako vstupní parametr je očekáván seznam dvojic s typy *Double*. Při generování výstupního seznamu je uvnitř generátoru procházen poskytnutý seznam. Následovně je dvojice dekonstruována do proměnných *cena* a *sleva*. Ty se dále používají při deklarování lokální proměnné *konecnaCena*, která je v posledním kroku použita pro filtrování a případné vracení pomocí výstupní funkce.

Při volání *vypocitejSlevy [(100, 25), (200, 80)]*, Haskell okamžitě vchází do generátoru seznamu, kde začíná iterovat v poskytnutému seznam *polo*zky. Zde dochází k dekonstrukci prvku elementu do dvou proměnných – *cena* a *sleva*, kde tyto proměnné odpovídají prvnímu a druhému prvku n-tice. U prvního prvku seznamu je tedy *cena = 100* a *sleva = 25*. Následovně dochází k deklaraci proměnné *konecnaCena*, jenž je při tomto průchodu rovna 75.0. Pomocí predikátu je tato proměnná porovnána s hodnotou 50. Jelikož platí *75.0 > 50*, je aplikována výstupní funkce – v tomto případě pouze přidání hodnoty 75.0 do výsledného seznamu. Pro druhý prvek seznamu se opakuje dekonstrukce n-tice, tedy *cena = 200* a *sleva = 80*. Opětovně dochází k deklaraci *konecnaCena*, nicméně tentokrát s hodnotou 40.0. Obdobně je na tuto proměnnou aplikován predikát, ale pro tuto hodnotu neplatí, jelikož *40.0 > 50* je vyhodnoceno jako False. Z tohoto důvodu není hodnota přidána do výsledného seznamu a funkce navrací jako výsledek *[75.0]*.

6.4.7 Výraz case

Většina imperativních jazyků dává programátorům přístup ke klauzulím jako je například switch, které umožňují zachycovat určité případy, které mohou nastat. Haskell tento koncept přebírá a vylepšuje, jelikož na rozdíl od jiných jazyků nám umožňuje provádět na výrazech i porovnávání vzorů (Lipovaca, 2011).

V minulé podkapitole byl pomocí funkce *vypocitejPreprodejniCenu* představen výraz *let*. Nicméně ve funkci se nachází spousta vnořených výrazů *if-then-else*, které značně snižují čitelnost. Jako skvělý adept se nabízí výraz *case*, u kterého bude demonstrována syntaxe pomocí příkladu.

```
1.   vypocitejPreprodejniCenu puvodniCena stav =
2.       let preprodejniCena = puvodniCena * 0.9
3.           faktorStavu = case stav of
4.               "novy"  -> 1.0
5.               "dobry" -> 0.8
6.               "ujde" -> 0.5
7.               "sesly" -> 0.2
8.               _      -> 0.1
9.       in  preprodejniCena * faktorStavu
```

Logika funkce zůstává zachována, ale čitelnost kódu se rapidně zvýšila. Namísto zanořování výrazů *if-then-else* je použita syntaxe *case <výraz> of*, kterou následují všechny možné případy, jichž může *stav* nabývat. Vyhodnocování jednotlivých případů je stejné jako u porovnávání vzorů, tedy shora dolů, přičemž na konci nechybí univerzální zachytávač ve formě „_“.

Výrazy *case* fungují stejným způsobem jako porovnávání vzorů na úrovni funkcí. Zde je upravený příklad funkce *popisSeznam*, který pracuje stejně, ale logika je přesunuta pouze do jediné funkce za použití výrazů *case*.

```
1.   popisSeznam seznam = case seznam of
2.     []             -> "Seznam je prazdny"
3.     [prvni]        -> "Jeden prvek: " ++ show prvni
4.     [prvni, druhy] -> "Dva prvky: " ++ show prvni ++ " and " ++ show
     druhy
5.     _              -> "Seznam ma vice nez dva prvky"
```

6.5 Rekurze

Rekurze je v programování způsob definice funkce takovým způsobem, kdy dochází k aplikování funkce samotné uvnitř své vlastní definice. Rekurze je podstatnou součástí Haskellu, jelikož na rozdíl od imperativních jazyků, v Haskellu provádíme výpočty pomocí deklarování závislostí mezi daty, namísto toho, abychom deklarovali, jakým způsobem se k výsledku dostaneme. Z toho důvodu v jazyku Haskell nejsou žádné smyčky a místo nich je povětšinou potřeba použít rekurzi (Lipovaca, 2011).

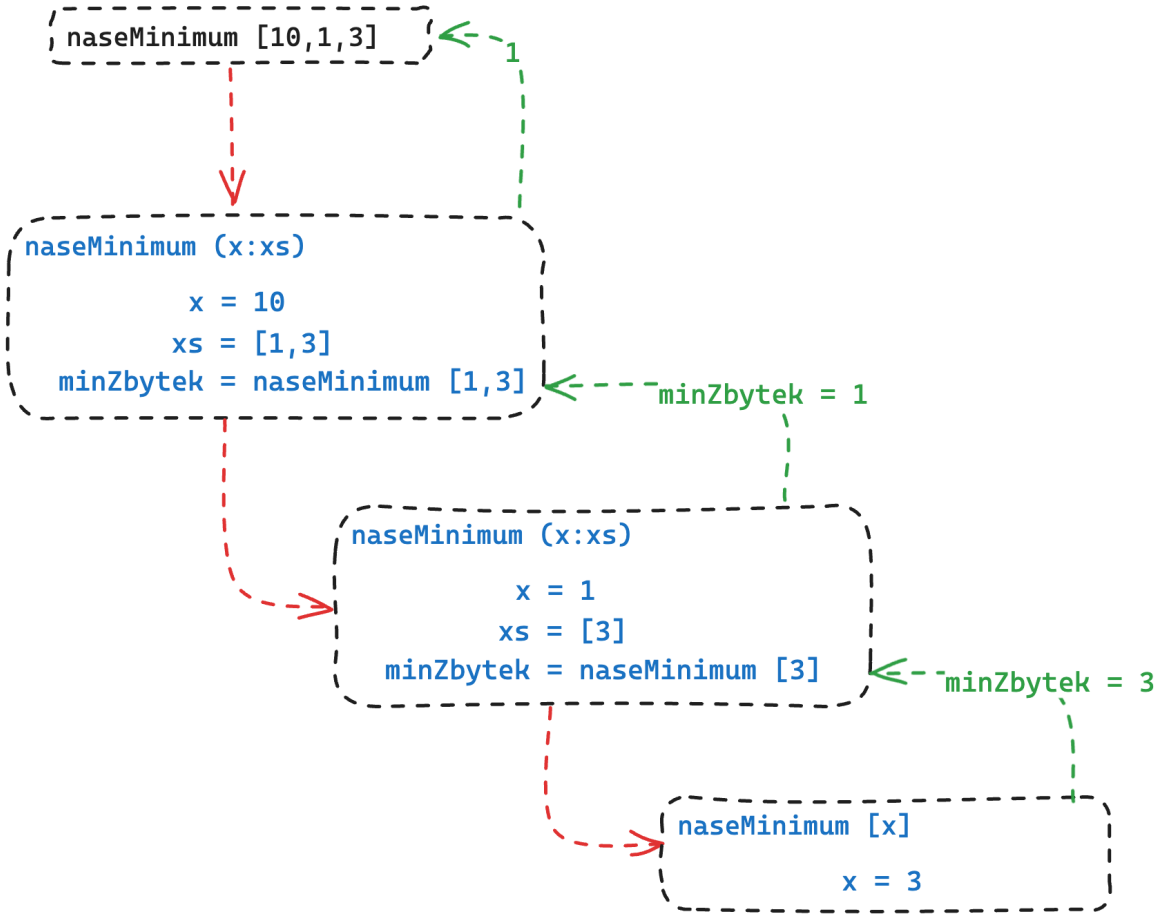
Na ukázku rekurze v Haskellu si představíme vlastní implementaci funkce *minimum*. Tato funkce přebírá seznam, jehož prvky lze seřadit a vrací nejmenší hodnotu uvnitř.

```
1.   naseMinimum :: Ord a => [a] -> a
2.   naseMinimum [] = error "Prazdny seznam!"
3.   naseMinimum [x] = x
4.   naseMinimum (x:xs)
5.       | x < minZbytek = x
6.       | otherwise = minZbytek
7.       where minZbytek = naseMinimum xs
```

Implementace takové metody by mohla vypadat například tímto způsobem. Všimněte si, že při rekurzi je použito porovnávání vzorů a dekonstrukce seznamu. Je vhodné se nejdříve soustředit na třetí definici funkce *naseMinimum*, kde nastává samotná rekurze. Zde dochází k porovnávání vzorů a dekonstrukci na hlavu seznamu a tělo. Následuje aplikování stráží, kde porovnáváme hlavu s proměnnou *minZbytek*, ve které je uložena nejmenší hodnota zbytku seznamu. Ta je získána rekurzivním voláním funkce *naseMinimum* v bloku *where*. Každá rekurze musí obsahovat ukončovací podmínku, v našem případě dává smysl ukončit rekurzi v bodě, kdy seznam obsahuje pouze jeden prvek – v takovém případě ho vrátíme. Poslední věcí, kterou zbývá ošetřit je případ, kdy obdržíme prázdný seznam – v prázdném seznamu, není co porovnávat, takže dojde k vyvolání chyby „Prazdny seznam!“.

Abychom si ukázali, jak taková rekurze probíhá, zkusíme si tento příklad projít se vstupním seznamem *[10,1,3]*.

- Haskell začne procházet definice našich funkcí, pokusí se pomocí porovnávání vzorů otestovat, zdali je poskytnutý seznam prázdný. Jelikož není, přesune se na další vzor, kde zjistí, že seznam obsahuje více než jeden prvek a přesune se dál.
- Třetímu vzoru již odpovídá, takže rozdělí pole na hlavu do proměnné *x (10)* a tělo do proměnné *xs ([1,3])*.
- Následovně se pokusí aplikovat stráže, ale aby to bylo možné, potřebuje znát hodnotu proměnné *minZbytek* a tu zjistí tak, že opětovně zavolá funkci *naseMinimum*, ale tentokrát se seznamem *[1,3]*.
- Stejným způsobem se aplikuje i druhý průchod, kdy Haskell přejde stejným způsobem do třetího vzoru – rozdělí seznam na hlavu a tělo. Znovu zavolá funkci *naseMinimum*, tentokrát se seznamem *[3]*.
- Jelikož se jedná o seznam o velikosti 1, dojde k přiřazení druhého vzoru, kde je jako výsledek vrácen jediný prvek, který je v seznamu obsažen.
- Nyní se Haskell vrací o úroveň výše, kdy už je známo, že hodnota *minZbytek = 3*.
- Následovně dochází k vyhodnocení stráží, kdy se porovnává hlava *x (1)* se získanou hodnotou *minZbytek (3)*, jelikož je podmínka splněna vrátí toto volání proměnnou *x (1)*.
- V zásobníku se nacházíme na první úrovni volání rekurze a aktuální stav proměnných je *x = 10* a *minZbytek = 1*.
- I na této úrovni dojde k porovnání pomocí stráží, ale tentokrát není podmínka splněna a případ zachytí výraz *otherwise*, ten vrací finální výsledek s hodnotou proměnné *minZbytek* = 1.



Obrázek 1: Diagram rekurze (zdroj: vlastní práce)

6.6 Typové třídy

Typové třídy mají v Haskellu podstatnou roli. Tato část demonstruje, za pomoci jednoduchého příkladu, jakým způsobem se typová třída implementuje a používá.

Uvažme situaci, kdy v Haskellu není možnost provádět testování rovnosti či nerovnosti a naším úkolem je to napravit. Existuje náš vlastní typ *Barva*, který může nabývat 3 hodnot – *Red*, *Green* a *Blue*. Rozhodneme se, že si napíšeme funkci *jeBarvaRovna*, kde aplikujeme porovnávání vzorů, abychom vyhodnotili, jestli se poskytnuté barvy shodují.

```
1.   data Barva = Red | Green | Blue
2.
3.   jeBarvaRovna :: Barva -> Barva -> Bool
4.   jeBarvaRovna Red Red = True
5.   jeBarvaRovna Green Green = True
6.   jeBarvaRovna Blue Blue = True
7.   jeBarvaRovna _ _ = False
```

Toto řešení by samozřejmě fungovalo, ale může se vyskytnout situace, kdy je potřeba umět otestovat rovnost na novém konstruktu *Point2D*, který je typu *Float Float*, jež reprezentují x a y souřadnici ve 2D prostoru. Zde nastává problém, kdy by bylo potřeba implementovat funkci *jePoint2DRoven*. Z toho důvodu existují typové třídy, které programátorům dávají moc definovat předpisy funkcí pro různé typy. Jednoduchá typová třída pro porovnávání by vypadala následovně.

```
1.   class JednoduchaRovnost a where
2.       jeRovno :: a -> a -> Bool
3.       neniRovno :: a -> a -> Bool
```

V typové třídě *JednoduchaRovnost* máme předpisy pro funkce *jeRovno* a *neniRovno*. Povšimněte si typového parametru *a*, který značí, že tyto funkce lze implementovat pro libovolný typ. To naznačuje, že funkce *jeRovno* a *neniRovno* očekávají na vstupu dva parametry stejného typu a výsledkem je typ *Bool*. Jediné, co v tuto chvíli zbývá je založit instanci pro konkrétní typ a implementovat tyto dvě metody.

```
1.   instance JednoduchaRovnost Barva where
2.       jeRovno Red Red = True
3.       jeRovno Green Green = True
4.       jeRovno Blue Blue = True
5.       jeRovno _ _ = False
6.
7.       neniRovno Red Red = False
8.       neniRovno no Green Green = False
9.       neniRovno Blue Blue = False
10.      neniRovno _ _ = True
```

Nicméně stále je místo pro zlepšení, *jeRovno* a *neniRovno* vrací pokaždé opačné hodnoty. Místo toho, aby byl uživatel této typové třídy nucen definovat obě funkce, je možné pro něho předpřipravit základní implementace obou funkcí.

```
1.   class JednoduchaRovnost a where
2.       jeRovno :: a -> a -> Bool
3.       jeRovno x y = not (neniRovno x y)
4.
5.       neniRovno :: a -> a -> Bool
6.       neniRovno x y = not (jeRovno x y)
```

Zde lze pozorovat, jak byla typová třída *JednoduchaRovnost* rozšířena o konkrétnější implementace. Jelikož jsou na sobě obě funkce závislé, stačí při deklarování instance implementovat pouze jednu z funkcí a pro druhou funkci bude použita implementace základní, jež je odvozena z typové třídy. V případě, že by předdefinovaná implementace pro potřebný typ nefungovala, může si ji její autor dle libosti přepsat.

```
1.   data Point2D = Point2D Float Float
2.
3.   instance JednoduchaRovnost Point2D where
4.       jeRovno (Point2D x1 y1) (Point2D x2 y2) = dx < epsilon && dy <
     epsilon
5.           where epsilon = 0.001
6.                 dx = abs (x1 - x2)
7.                 dy = abs (y1 - y2)
```

7 Řešené úlohy

Tato kapitola je věnována řadě úloh, které obsahují zadání, příklady použití s očekávanými výsledky, řešení, ke kterému je možné dospět, vysvětlení daného řešení a případné poznámky o chybách, které mohou v kódu vzniknout.

7.1 Je textový řetězec palindrom?

Napiš funkci *jePalindrom*, která na svém vstupu očekává textový řetězec a vrací *True* či *False*, pokud je poskytnutý text palindrom. Předpokládá se, že poskytnutý textový řetězec neobsahuje mezery a skládá se pouze z malých písmen.

```
ghci> jePalindrom "nezařazen"
True
ghci> jePalindrom "ahoj"
False
```

Tuto funkci je velmi jednoduché implementovat za využití vestavěné funkce *reverse*, nicméně při řešení nejprve uplatníme rekurzi a vlastnost textových řetězců, kde text je ve své podstatě pouze seznamem znaků.

Prvním krokem při psaní většiny rekurzivních funkcí je uvědomit si, jaká je ukončovací podmínka. V tomto konkrétním případě existují dvě podmínky, kdy je možné určit, že poskytnutý text je opravdu palindromem – pokud text (seznam) neobsahuje žádný znak nebo pokud text (seznam) obsahuje právě jeden prvek. Tím jsou vyřešeny podmínky a stačí vymyslet, jakým způsobem se tato funkce bude volat rekurzivně. Rychlým způsobem, jak zjistit, jestli je text palindromem, je porovnávání prvního a posledního prvku daného textu a v případě, že se shodují provézt rekurzivnímu volání.

Jedno z možných řešení pomocí rekurze:

```
1.   jePalindrom :: String -> Bool
2.   jePalindrom [] = True
3.   jePalindrom [x] = True
4.   jePalindrom (x:xs) = (x == last xs) && jePalindrom (init xs)
```

Rekurzivní volání funkce probíhá na 4. řádku, kde je poskytnutý seznam rozdělen na hlavu a zbytek seznamu. Následovně je porovnána hlava *x* (první znak textu) s posledním znakem textu za využití funkce *last*. Pokud se shodují, následuje samotné rekurzivní volání, kde je jako vstupní parametr poskytnut výsledek volání funkce *init*, který je aplikován na seznam *xs* – toto zaručuje to, že dojde k odstranění posledního prvku, takže efektivně můžeme v další rekurzi uplatnit stejnou logiku.

V této funkci dbejte na to, že při rekurzivním volání je *init xs* obehnáno závorkami tak, aby došlo nejdříve k navrácení nového seznamu bez posledního prvku a až poté k jeho použití jako vstupní argument pro funkci *jePalindrom*.

Jedno z možných řešení pomocí funkce *reverse*:

```
1.   jePalindrom :: String -> Bool
2.   jePalindrom text = text == reverse text
```

V tomto velmi jednoduchém řešení je uplatněna funkce reverse, která otáčí prvky pole a vrací nové pole. Abychom tedy zjistili, zda je poskytnutý text palindrom, stačí původní text porovnat s otočeným textem a vrátit výslednou logickou hodnotu.

7.2 Chybějící znak

Cílem této úlohy je napsat funkci *najdiChybejiciZnak*, která na svém vstupu očekává seznam znaků. Tyto znaky tvoří sekvenci a jsou seřazeny vzestupně. Vstupní seznam vždy nabývá délky alespoň dvou prvků a vždy mezi prvním a posledním prvkem chybí právě jeden znak.

```
ghci> najdiChybejiciZnak ['a','b','c','e']
'd'
ghci> najdiChybejiciZnak ['x', 'z']
'y'
```

Řešení této úlohy je založeno na rekurzi. Při řešení této úlohy je dobré si uvědomit, že znak (Char) v Haskellu je ve skutečnosti Enum, což má za důsledek to, že je skrze něj možné iterovat pomocí funkcí jako *succ* a *pred*, které nám umožňují získat následující nebo předchozí Unicode hodnotu.

Jedno z možných řešení:

```
1.  najdiChybejiciZnak :: [Char] -> Char
2.  najdiChybejiciZnak (z1:z2:xs)
3.      | ocekavanyZnak == z2 = najdiChybejiciZnak (z2:xs)
4.      | otherwise = ocekavanyZnak
5.      where
6.          ocekavanyZnak = succ z1
```

Ukázkové řešení závisí na použití funkce *succ*, která je použita ke zjištění očekávaného znaku v sekvenci. Pomocí stráži je zjištěno, jestli se očekávaný znak shoduje se znakem na druhé pozici (*z2*). Pokud ano, je rekurzivně zavolána funkce s nově vytvořeným seznamem, který vzniká připojením druhého znaku ke zbytku seznamu. V případě, že podmínka neplatí, se našlo místo v seznamu, kde chybí očekávaný znak a je možné ho bezpečně navrátit pomocí klíčového slova *otherwise*.

7.3 Obsahuje textový řetězec pouze unikátní znaky

Vytvořte funkci s názvem *maUnikatniZnaky*, jejímž vstupem je textový řetězec. Tento řetězec může obsahovat jakékoli znaky a je citlivý na malá a velká písmena, což znamená, že např. znaky 'A' a 'a' jsou považovány za odlišné.

```
ghci> maUnikatniZnaky "Ahoj"
True
ghci> maUnikatniZnaky "Test"
True
ghci> maUnikatniZnaky "test"
False
```

Jedno z možných řešení:

```
1.   maUnikatniZnaky :: String -> Bool
2.   maUnikatniZnaky [] = True
3.   maUnikatniZnaky (x:xs) = x `notElem` xs && maUnikatniZnaky xs
```

Ukončovací podmínka rekurze v této funkci nastane, pokud je seznam znaků (text) prázdný. Pokud do té doby není nalezen žádný duplicitní znak, funkce vrátí hodnotu True, což značí, že všechny znaky jsou unikátní. Rekurze se provádí na třetím řádku, kde je vstupní řetězec rozdělen na první znak (hlavu *x*) a zbytek řetězce (*xs*). Poté se použije funkce *notElem* k ověření, zda se hodnota *x* nevyskytuje ve zbytku seznamu *xs*. Pokud *x* není součástí *xs*, funkce *maUnikatniZnaky* pokračuje rekurzivně s argumentem *xs*.

Tato funkce využívá vlastnosti textových řetězců, kde textový řetězec je seznamem znaků. Jediné, co je potřeba upravit, aby tato funkce kontrolovala unikátnost prvků v jakémkoli seznamu, je její typová signatura. Ta naznačuje, že funkce na vstupu očekává seznam prvků typu *a*, kde typ *a* implementuje typovou třídu *Eq*, která se stará o porovnávání.

```
1.   maUnikatniPrvky :: Eq a => [a] -> Bool
2.   maUnikatniPrvky [] = True
3.   maUnikatniPrvky (x:xs) = x `notElem` xs && maUnikatniPrvky xs
```

7.4 Rozdělení camelCase

Napiš funkci *rozdelCamelCase*, která na svém vstupu očekává textový řetězec a vrací textový řetězec, kde „camelCase“ je rozdělen pomocí mezer. Pokud poskytnutý textový řetězec nevyužívá metodu zápisu „camelCase“, je text vrácen bez úprav.

```
ghci> rozdelCamelCase "krasnyDen"
"krasny Den"
ghci> rozdelCamelCase "ahoj"
"ahoj"
```

K vyřešení úlohy je možné uplatnit rekurzi. Aby bylo možné detekovat, jestli text obsahuje „camelCase“, je potřeba zjistit, jestli se v textu vyskytuje velké písmeno, na což bude potřeba vedlejší pomocná funkce, nebo je možnost využít vestavěnou funkci *isUpper*, která je použitelná, pokud je naimportován balíček *Data.Char.*

Jedno z možných řešení s vlastní implementací funkce pro zjištění velkého písmena:

```
1.  jeVelkePismeno :: Char -> Bool
2.  jeVelkePismeno z = z `elem` ['A'..'Z']
3.
4.  rozdelCamelCase :: String -> String
5.  rozdelCamelCase [] = []
6.  rozdelCamelCase (x:xs)
7.      | jeVelkePismeno x = ' ' : x : rozdelCamelCase xs
8.      | otherwise = x : rozdelCamelCase xs
```

Ukončovací podmínka v tomto řešení nastává, když rekurze dosáhne samotného konce seznamu (textu). Součástí řešení je i implementace funkce *jeVelkePismeno*, která využívá funkci *elem* v kombinaci s rozsahem, který obsahuje veškerá velká písmena od A do Z. Díky ní jsme následovně schopni při kontrole ve strážích rozhodnout, jestli se jedná o nové slovo, které je odděleno mezerou, či ne.

Jedno z možných řešení s využitím funkce *isUpper*:

```
1.   import Data.Char
2.
3.   rozdelCamelCase :: String -> String
4.   rozdelCamelCase [] = []
5.   rozdelCamelCase (x:xs)
6.       | isUpper x = ' ' : x : rozdelCamelCase xs
7.       | otherwise = x : rozdelCamelCase xs
```

V tomto řešení je obdobně využita rekurze, nicméně pro kontrolu velkého písmena, je zde uplatněna funkce *isUpper*, která plní stejný úkol jako námi implementovaná funkce *jeVelkePismeno*. Tuto funkci je možné využít až po naimportování balíčku *Data.Char*, které probíhá na prvním řádku ukázky.

7.5 Rozdíl dvou seznamů

Vytvořte funkci *rozdilSeznamu*, která obdrží dva seznamy celých čísel. Úkolem této funkce je odstranit z prvního seznamu všechna čísla, která se objevují také ve druhém seznamu. Pokud se některé číslo z druhého seznamu vyskytuje v prvním seznamu vícekrát, funkce odstraní všechny jeho výskyty z prvního seznamu.

```
ghci> rozdilSeznamu [1, 2, 3, 4, 5, 6, 7] [1, 7, 3, 2]
[4,5,6]
ghci> rozdilSeznamu [1, 2, 2, 2, 3] [2]
[1,3]
```

Úlohu lze řešit několika způsoby. Jednou z možností je využití rekurze ve spojení s funkcí *filter*. Tato funkce vyžaduje jako první parametr jinou funkci, která vrací logickou hodnotu. Na základě této hodnoty poté funkce *filter* rozhoduje, zda se konkrétní prvek seznamu zachová, nebo bude odstraněn. A jako druhý parametr se funkci *filter* předává seznam, ve kterém chceme provádět filtraci. Dalším, o něco elegantnějším přístupem, je využití generátoru seznamů, který umožňuje přímou konstrukci nového seznamu podle definovaných kritérií.

Jedno z možných řešení pomocí rekurze:

```
1.  rozdilSeznamu :: [Int] -> [Int] -> [Int]
2.  rozdilSeznamu [] _ = []
3.  rozdilSeznamu seznam [] = seznam
4.  rozdilSeznamu seznam (y:ys) =
5.                rozdilSeznamu (filter (\x -> x /= y) seznam) ys
```

Při využití rekurze je klíčové definovat ukončovací podmínku, která v tomto případě závisí na obsahu druhého seznamu. Tento seznam určuje, které prvky mají být z prvního seznamu odstraněny. Jakmile jsou prozkoumány všechny prvky druhého seznamu, vše, co v prvním seznamu zbude, je konečným výsledkem, který následovně funkce vrátí. Dále může nastat situace, kdy funkce obdrží prázdný první seznam, v takovém případě není co filtrovat a vracíme jako výsledek prázdný seznam.

V poslední části vypracovaného řešení dochází k aplikaci funkce *filter*, do které jako první argument předáváme anonymní funkci – což je ve své podstatě normální funkce, s rozdílem toho, že není pojmenována. Tato funkce je spuštěna pro každý prvek prvního seznamu. Ve svém těle tato funkce ověřuje, zda daný prvek neodpovídá prvnímu prvku ze seznamu určeného pro filtrování (označeného jako *y*). Výsledkem použití funkce *filter* je nový seznam, který je poté použit jako vstupní argument pro další rekurzivní volání spolu se zbytkem filtračního seznamu (*ys*).

V tomto řešení je potřeba správně oddělit funkci *filter* závorkami tak, aby došlo ke správné posloupnosti aplikování funkcí.

Jedno z možných řešení pomocí generátoru seznamů:

```
1.  rozdilSeznamu :: [Int] -> [Int] -> [Int]
2.  rozdilSeznamu seznam1 seznam2 = [x | x <- seznam1, x `notElem`
    seznam2]
```

Je zřejmé, že použití generátoru seznamů vede ke kratšímu zápisu, než je tomu u rekurzivního způsobu. V generátoru seznamů jsou postupně procházeny prvky prvního seznamu (*seznam1*), přičemž každý prvek uložíme do proměnné *x*. Na tuto proměnnou je poté aplikován predikát, který ověřuje, zda prvek *x* není součástí druhého seznamu získaného na vstupu. Pokud prvek tento predikát splňuje, je přidán do výsledného seznamu pomocí výstupní funkce.

7.6 Každý n-tý prvek

Napište funkci *kazdy*, která očekává dva parametry – celé číslo *n* a seznam hodnot. Tato funkce vygeneruje a vrátí nový seznam obsahující každý n-tý prvek původního seznamu. Pokud je parametr *n* záporný, funkce bude prvky vybírat zpětně, tedy od konce seznamu.

```
ghci> kazdy 0 [1,2,3,4,5,6]
[]
ghci> kazdy 1 [1,2,3,4,5,6]
[1,2,3,4,5,6]
ghci> kazdy 2 [1,2,3,4,5,6]
[2,4,6]
ghci> kazdy 4 [1,2,3,4,5,6]
[4]
ghci> kazdy (-1) [1,2,3,4,5,6]
[6,5,4,3,2,1]
ghci> kazdy (-2) [1,2,3,4,5,6]
[5,3,1]
```

Pro řešení tohoto příkladu je vhodné použít funkce *reverse* a *drop*, pomocí kterých je možné manipulovat s daným seznamem.

Jedno z možných řešení:

```
1.  kazdy :: Int -> [a] -> [a]
2.  kazdy 0 _ = []
3.  kazdy n seznam
4.    | n < 0 = kazdy (-n) (reverse seznam)
5.    | otherwise = case drop (n - 1) seznam of
6.      []          -> []
7.      (x : zbytek) -> x : kazdy n zbytek
```

Jeden z případu, který je vhodné ošetřit pomocí samostatného vzoru, je situace kdy *n = 0*, v takovém případě není potřeba složitých výpočtů a může být okamžitě navrácen prázdný seznam.

V rekurzivní části funkce je na čtvrtém řádku ošetřen případ, kdy hodnota *n* je záporná. V takovém případě se *n* neguje, aby bylo získáno kladné číslo a zároveň se provádí obrácení seznamu aplikováním funkce *reverse*, následovně je funkce rekurzivně zavolána s upravenými hodnotami. Na dalším řádku se pomocí klíčového slova *otherwise* zachytávají všechny ostatní situace. Používá se zde funkce *drop*, která vrací nový seznam s vynechanými prvky na začátku podle zadaného počtu pozic, zde konkrétně *n* - 1. To znamená, že při volání *kazdy 4 [1,2,3,4,5,6]* se při prvním průchodu odstraní první tři prvky (1, 2, 3), a tak bude navrácený seznam začínat čtvrtým prvkem, což je přesně ten, jenž chceme získat pro *n = 4*.

7.7 Je číslo prvočíslo

Prvočíslo je přirozené číslo, které je beze zbytku dělitelné pouze jedničkou a sebou samým, přičemž číslo jedna prvočíslem není (Kratochvíl, 2020). Napiš funkci *jePrvoCislo*, která na vstupu přijímá celé číslo a vrací logickou hodnotu v závislosti na tom, zda je poskytnuté číslo opravdu prvočíslem.

```
ghci> jePrvoCislo 2
True
ghci> jePrvoCislo 3
True
ghci> jePrvoCislo 10
False
```

Jedno z možných řešení s využitím funkce *all*:

```
1.   jePrvoCislo :: Int -> Bool
2.   jePrvoCislo n = n > 1 && all (\x -> n `mod` x /= 0) [2..n-1]
```

Toto řešení využívá pro zjištění výsledku vestavěnou funkci *all*, ta na svém vstupu očekává funkci a seznam. Funkce *all* se používá, pokud je potřeba zjistit, zda námi poskytnutá funkce platí pro všechny jednotlivé prvky poskytnutého seznamu a v závislosti na této skutečnosti je navrácena hodnota *True* či *False*.

Pomocí rozsahu je vytvořen seznam začínající číslem 2, což je nejmenší možný dělitel čísla n, a končí číslem *n - 1*, jelikož jakékoliv větší číslo je pro naše účely irelevantní. Funkce *all* poté prochází tento seznam a pro každý prvek, označovaný jako *x*, spouští definovanou kontrolní funkci. Tato funkce ověřuje, zda je uživatelem zadané číslo *n* dělitelné číslem *x* beze zbytku.

Jedno z možných řešení s využitím rekurze:

```
1.   jePrvoCislo :: Int -> Bool
2.   jePrvoCislo n = n > 1 && jePrvoCisloPomocna n 2
3.     where
4.       jePrvoCisloPomocna :: Int -> Int -> Bool
5.       jePrvoCisloPomocna n d
6.         | d * d > n = True
7.         | n `mod` d == 0 = False
8.         | otherwise = jePrvoCisloPomocna n (d + 1)
```

V tomto řešení je využita rekurze. Na rozdíl od předchozího řešení je zde uvnitř bloku where definována pomocná funkce *jePrvoCisloPomocna*, která očekává dva parametry: celé číslo *n*, které kontrolujeme, a celé číslo *d*, které reprezentuje dělitele pro kontrolu. V těle této pomocné funkce dochází nejprve k uplatnění stráží. První strážný kontroluje, zda druhá mocnina dělitele je větší než kontrolované číslo *n*. Tento princip je založen na faktu, že pokud nenajdeme žádné dělitele až do odmocniny z *n*, pak žádné vyšší dělitele neexistují, a můžeme bezpečně vrátit True. Druhý strážný kontroluje, zda je zbytek po dělení *n* dělitelem *d* roven nule. Pokud ano, konstatujeme, že číslo není prvočíslem, a vrátíme False. V ostatních případech dojde k rekurzivnímu volání funkce s inkrementovaným dělitelem o jedničku. Tato rekurze začíná voláním funkce *jePrvoCisloPomocna n 2*, jelikož *d = 2* je nejmenší možný dělitel kontrolovaného čísla *n*.

7.8 Postav věž

Napiš funkci *postavVez*, která přijímá celočíselnou hodnotu *n*. Úkolem je pomocí hvězdiček a mezer postavit věž ve tvaru pyramidy. Parametr *n* určuje, kolik pater má výsledná věž obsahovat. Každé patro je o jeden blok širší na každou stranu než patro předešlé. Stavební bloky jsou reprezentovány pomocí symbolu hvězdy „*".

```
ghci> postavVez 3
[
 "  *  ",
 " *** ",
 "*****"
]
ghci> postavVez 5
[
 "    *    ",
 "   ***   ",
 "  *****  ",
 " ******* ",
 "*********"
]
```

Při řešení úlohy je možné využít vestavěnou funkci *replicate*. Této funkci lze předat číselný parametr, který specifikuje, kolikrát se má poskytnutá hodnota opakovat. Výsledkem je seznam obsahující tyto opakované hodnoty. Pomocnou funkci podobného charakteru lze snadno implementovat, například pomocí generátorů seznamů.

Jedno z možných řešení s využitím funkce *replicate*:

```
1.   postavVez :: Int -> [String]
2.   postavVez n = [postavPatro patro | patro <- [1..n]]
3.     where
4.       postavPatro aktualniPatro = mezery ++ bloky ++ mezery
5.         where
6.           mezery = replicate (n - aktualniPatro) ' '
7.           bloky = replicate (aktualniPatro * 2 - 1) '*'
```

V těle funkce v bloku *where* je definována pomocná *funkce postavPatro*, která na svém vstupu očekává číselnou hodnotu, pomocí níž je reprezentováno aktuální generované patro. V těle této pomocné funkce se následovně kombinují vygenerované seznamy mezer a samotných bloků. Tato pomocná funkce je následovně využívaná v rámci generátoru seznamu jakožto část výstupní funkce.

Jedno z možných řešení s vlastní implementací funkce *replicate*:

```
1.   postavVez n = [postavPatro patro | patro <- [1..n]]
2.     where
3.       opakujZnak pocet znak = [znak | _ <- [1..pocet]]
4.       postavPatro aktualniPatro = mezery ++ bloky ++ mezery
5.         where
6.           mezery = opakujZnak (n - aktualniPatro) ' '
7.           bloky  = opakujZnak (aktualniPatro * 2 - 1) '*'
```

Vestavěnou funkci *replicate* je v této úloze velmi jednoduché nahradit vlastní implementací, což je zde učiněno pomocí funkce *opakujZnak*. Ta, stejně jako *replicate*, očekává na svém vstupu počet, který určuje, kolikrát se má daný znak opakovat, a samotný znak. Opakování znaku probíhá s využitím generátoru seznamu, kde je definován rozsah v rozmezí *[1..pocet]*, který je následně použit jako vstupní množina pro generátor. Výstupní funkce pak používá daný znak, který se má opakovat. Zbytek logiky zůstává zachován oproti předchozímu řešení.

7.9 Tabulka násobení

Tabulka násobků slouží jako pomůcka pro žáky prvního stupně základní školy. Místo kde se protíná vybraný sloupec a řádek označuje výsledek součinu příslušných čísel. Napiš funkci *tabulkaNasobeni*, která vygeneruje tabulku násobků. Funkce na svém vstupu očekává 2 celočíselné parametry, které určují šířku a výšku generované tabulky.

```
ghci> tabulkaNasobeni 5 4
[
  [1, 2,  3,  4,  5],
  [2, 4,  6,  8, 10],
  [3, 6,  9, 12, 15],
  [4, 8, 12, 16, 20]
]
```

Jedno z možných řešení:

```
1.   tabulkaNasobeni :: Int -> Int -> [[Int]]
2.   tabulkaNasobeni w h = [[x * y | x <- [1..w]] | y <- [1..h]]
```

Úloha vytvoření tabulky násobení je efektivně řešitelná pomocí vnořeného generátoru seznamů. Generátor iteruje přes dva intervaly, které reprezentují hodnoty pro osy x a y. Vnějším generátoru seznamu *y <- [1..h]* začíná prvním řádkem tabulky a postupuje směrem dolů k poslednímu řádku *h*. Uvnitř tohoto generátoru je pak vnořený generátor seznamu *[x * y | x <- [1..w]]*, který pro každý řádek generuje sloupce tabulky od prvního sloupce až po poslední sloupec *w*, přičemž *x* a *y* jsou současně násobeny, aby vytvořily hodnoty pro danou pozici v tabulce.

7.10 Nahraď písmena za jejich pozici v abecedě

Implementuj funkci *nahradAbecedniPozici*, jenž přijímá jako vstupní parametr textový řetězec. V něm nahradí každý znak abecedy jeho pozicí v anglické abecedě – to znamená, že znak „a“ a „A“ budou oba na pozici 1. Znak není konvertován, jestliže není znakem anglické abecedy. Očekávaným výstupem je textový řetězec pozic znaků oddělených mezerou.

```
ghci> nahradAbecedniPozici "Skakal pes pres oves"
"19 11 1 11 1 12 16 5 19 16 18 5 19 15 22 5 19"
ghci> nahradAbecedniPozici "Kral Karel s Buskem z Vilhartic"
"11 18 1 12 11 1 18 5 12 19 2 21 19 11 5 13 26 22 9 12 8 1 18 20 9 3"
```

Jedno z možných řešení:

```
1.   import Data.Char
2.
3.   nahradAbecedniPozici :: String -> String
4.   nahradAbecedniPozici text =
5.     unwords [
6.               show (ord (toLower x) - ord 'a' + 1) |
7.               x <- text,
8.               isAlpha x
9.             ]
```

Ukázkové řešení používá k získání výsledku generátor seznamu. V těle generátoru jsou do proměnné *x* postupně ukládány jednotlivé znaky textového řetězce. Hodnota proměnné *x* je následovně kontrolována pomocí jednoho predikátu. Ten využívá funkci *isAlpha*, která ověřuje, že *x* je opravdu znakem anglické abecedy. V dalším kroku ve výstupní funkci dochází k aplikování funkce *toLower*, pomocí které získáme znak reprezentovaný malým písmenem, na proměnnou *x*. Na tento znak je následovně použita funkce *ord*, pomocí které získáme Unicode hodnotu, od té je poté odečtena hodnota znaku „a“ (97), čímž získáme pozici znaku v abecedě. Výsledná pozice, počítaná od jedničky, je pak určena přičtením čísla 1 k rozdílu. Na úplném konci je číselná hodnota převedena do textové podoby za použití funkce *show*. Vygenerovaný seznam je v posledním kroku spojen do jednoho textového řetězce

pomocí funkce *unwords* – ta na vstupu očekává seznamy textových řetězců, jehož prvky spojí do jednoho seznamu (textového řetězce), kde každý prvek je oddělen mezerou.

K využití funkcí *isAlpha* a *ord* je nutné naimportovat balíček *Data.Char*, tak jak je naznačeno v ukázkovém řešení na prvním řádku.

7.11 Ciferace

Ciferace je pojem z oblasti aritmetiky označující zobrazení, které každému přirozenému číslu přiřazuje jednociferné číslo vzniklé opakováním ciferného sčítání původního čísla. Je-li výsledek jednociferný, jedná se o ciferaci původního čísla. V opačném případě se opakuje sčítání cifer výsledku, dokud nedostaneme jednociferné číslo – ciferaci původního čísla („Ciferace", 2023). Napiš funkci *ciferace*, která jako vstupní parametr přijímá celé číslo a následovně provádí jeho ciferaci, která je výsledkem volání této funkce.

```
ghci> ciferace 1
1
ghci> ciferace 11
2 (1 + 1 => 2)
ghci> ciferace 3872
2 (3 + 8 + 7 + 2 = 20 => 2 + 0 => 2)
ghci> ciferace 99128
2 (9 + 9 + 1 + 2 + 8 => 29 => 2 + 9 => 11 => 1 + 1 => 2)
```

Jedno z možných řešení:

```
1.   import Data.Char
2.
3.   ciferace :: Int -> Int
4.   ciferace n
5.     | n < 10 = n
6.     | otherwise = ciferace (sum (textNaIntSeznam (show n)))
7.     where
8.       textNaIntSeznam :: String -> [Int]
9.       textNaIntSeznam [] = []
10.      textNaIntSeznam (x:xs) = digitToInt x : textNaIntSeznam xs
```

Základem ukázkového řešení je aplikování rekurze. Zde je aplikována na dvou místech – při volání samotné funkce *ciferace* a funkce *textNaIntSeznam*, která je definovaná uvnitř bloku *where*. Také jsou zde uplatněny stráže, pomocí kterých kontrolujeme, zda je vstupní parametr menší než číslo 10, pokud ano je výsledkem volání samotné číslo.

V opačném případě je potřeba provést výpočet ciferace. Ten probíhá na 6. řádku – nejdříve je vstupní celočíselný parametr převeden na textový řetězec pomocí funkce *show*, na ten je následovně aplikována funkce *textNaIntSeznam*. Tato funkce vrací seznam celých čísel, který je následovně sečten pomocí funkce *sum*. Její výsledek je použit jako vstupní parametr pro rekurzivní volání funkce *ciferace*.

Implementace funkce *textNaIntSeznam* je také založena na rekurzi. Ukončovací podmínkou je případ, kdy dojdeme na konec textu – tehdy vracíme prázdný seznam. V opačném případě dochází k rozdělení seznamu na první prvek *x* a zbytek seznamu *xs*. Na *x* je následovně aplikována funkce *digitToInt*, která je součástí balíčku *Data.Char*, jenž je nutné naimportovat, tak jak je ukázáno v řešení. Funkce *digitToInt* na svém vstupu očekává znak (Char) a provádí jeho převod na celočíselnou hodnotu (*'9' => 9*). K tomuto číslu je v dalším kroku připojen seznam, který vzniká rekurzivním voláním funkce, kde jako vstupní parametr poskytujeme zbytek textu *xs*.

7.12 Rozděl text do párů

Napiš funkci *rozdelText*, která na svém vstupu očekává textový řetězec. Výsledkem volání vzniká seznam textových řetězců, kde každý prvek obsahuje dva znaky ze vstupního textového řetězce. Pokud je délka textu lichá, tak poslední dvojice obsahuje místo jednoho znaku znak podtržítka „_“.

```
ghci> rozdelText "Ahoj"
["Ah","oj"]
ghci> rozdelText "AhojSvete"
["Ah","oj","Sv","et","e_"]
```

Jedno z možných řešení:

```
1.   rozdelText :: String -> [String]
2.   rozdelText [] = []
3.   rozdelText [znak] = [[znak, '_']]
4.   rozdelText (a:b:xs) = [a, b] : rozdelText xs
```

V ukázkovém řešení je pro získání výsledku uplatněna rekurze. Pro tuto úlohu se nabízí dvě ukončovací podmínky. První – rekurze došla na konec textu a žádný znak nezbyl, v takovém případě může funkce navrátit prázdný seznam. Druhá podmínka – seznam obsahuje právě jeden prvek, to znamená, že poskytnutý textový řetězec byl liché délky – jako výsledek navrátíme znak obsažený v textovém řetězci ve spojení se znakem podtržítka „_“.

Rekurzivní volání se odehrává na 4. řádku, kde je poskytnutý textový řetězec rozdělen na první *a*, druhý *b* znak a zbytek textu *xs*. Následovně dochází k vytvoření seznamu o dvou prvcích *[a, b]*, ke kterému je zprava připojen výsledek rekurzivního volání funkce *rozdelText*, již je poskytnut jako vstupní argument zbytek textu *xs*.

8 Shrnutí a diskuse výsledků

V průběhu této bakalářské práce jsem poznal základy a pokročilé aspekty funkcionálního programování a jazyka Haskell. Při práci na praktických úlohách jsem nejen aplikoval teoretické principy funkcionálního programování, ale také jsem uvažoval nad nevědomým začleněním některých z těchto principů do mých předchozích programovacích návyků. Zejména využívání funkcí jako prvotřídních občanů nebo odvozování vzorů. Tato skutečnost přinesla zvýšené pochopení pro hodnotu funkcionálního přístupu, který nabízí elegantní a efektivní řešení problémů.

Tato práce také ukázala, že přestože zdroje pro učení funkcionálního programování jsou bohaté, je většina z nich v anglickém jazyce. Tato jazyková bariéra může pro některé studenty představovat problém, a proto je zapotřebí vytvořit materiály v jazyce českém, které budou poskytovat snadnější přístup ke studiu Haskellu. Takový přístup by mohl podstatně zlepšit pochopení a zájem studentů o tento programovací jazyk či funkcionální paradigma obecně.

9 Závěry a doporučení

Tato práce se zaměřila na prozkoumání a aplikaci principů funkcionálního programování s důrazem na jazyk Haskell. Paradigmata imperativního a deklarativního programování představují fundamentálně odlišné přístupy k získání výsledku. Zatímco imperativní programování formuluje postupnou sérii instrukcí, deklarativní paradigma, jehož je funkcionální programování součástí, preferuje vyjádření vztahů a závislostí mezi daty, nikoliv explicitní postup dosažení výsledku.

Práce rovněž plní úlohu dodatečných materiálů, které by podporovaly porozumění studentů, a to nejenom těch, kteří se funkcionálním programováním zabývají poprvé. S přihlédnutím k těmto požadavkům byla vytvořena řada řešených úloh a příkladů, které studentům poskytují praktické zkušenosti s aplikací teorie do praxe. Tato práce tak slouží jako most mezi teoretickými koncepty prezentovanými v akademickém prostředí a reálnými programovacími situacemi, s nimiž se mohou studenti setkat.

Vzhledem k rostoucí popularitě funkcionálního programování v průmyslu a akademické sféře, může tato práce sloužit jako cenný zdroj pro ty, kteří hledají místo, kde začít s funkcionálním programováním a Haskellem.

10 Seznam použité literatury

Allen, C., & Moronuki, J. (2016). *Haskell Programming from First Principles*. Lorepub LLC. https://books.google.cz/books?id=5FaXDAEACAAJ

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, *21*(8), 613–641. https://doi.org/10.1145/359576.359579

Bowen, J. (2018, leden 8). Immutability: The Less Things Change, The More You Know. *Medium*. https://medium.com/@james_32022/immutability-the-less-things-change-the-more-you-know-210a94956ddb

Ciferace. (2023). In *Wikipedie*. https://cs.wikipedia.org/w/index.php?title=Ciferace&oldid=22546184

Cocca, G. (2022, květen 2). *Programming Paradigms – Paradigm Examples for Beginners*. freeCodeCamp.Org. https://www.freecodecamp.org/news/an-introduction-to-programming-paradigms/

Currying—HaskellWiki. (b.r.). Získáno 8. leden 2024, z https://wiki.haskell.org/Currying

erkmos. (2024). *Erkmos/haskell-companies* [Software]. https://github.com/erkmos/haskell-companies (Original work published 2017)

Fulber-Garcia, V. (2021, říjen 31). *Imperative and Declarative Programming Paradigms | Baeldung on Computer Science*. https://www.baeldung.com/cs/imperative-vs-declarative-programming

Hudak, P., Hughes, J., Peyton Jones, S., & Wadler, P. (2007, červen 9). A history of Haskell: Being lazy with class. *Proceedings of the Third ACM SIGPLAN*

Conference on History of Programming Languages. HOPL-III '07: ACM SIGPLAN History of Programming Languages Conference III, San Diego California. https://doi.org/10.1145/1238844.1238856

Klabnik, S., & Nichols, C. (b.r.). *Variables and Mutability—The Rust Programming Language*. Získáno 9. prosinec 2023, z https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html

Kratochvíl, D. (2020). *Zajímavá prvočísla a jejich vlastnosti* [Bakalářská práce, Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta]. https://theses.cz/id/x4l7oj/

Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, *6*(4), 308–320. https://doi.org/10.1093/comjnl/6.4.308

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, *9*(3), 157–166. https://doi.org/10.1145/365230.365257

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide* (1st edition). No Starch Press.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, *3*(4), 184–195. https://doi.org/10.1145/367177.367199

McGinnis, T. (2019). *Why React Hooks?* ui.dev. https://ui.dev/why-react-hooks

Milewski, B. (2014, prosinec 25). *3. Pure Functions, Laziness, I/O, and Monads—School of Haskell | School of Haskell*. https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

O'Sullivan, B., Goerzen, J., & Stewart, D. (2008). *Real World Haskell* (1st edition). O'Reilly Media.

Quick Start – React. (b.r.). Získáno 9. prosinec 2023, z https://react.dev/learn

React. (2023). [JavaScript]. Meta. https://github.com/facebook/react (Original work published 2013)

State of JavaScript 2022: Front-end Frameworks. (b.r.). Získáno 9. prosinec 2023, z https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/

11 Seznam obrázků

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Martin Kaněra**
Studium:	I2100219
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Funkcionální programování**
Název bakalářské práce AJ:	Functional Programming

Cíl, metody, literatura, předpoklady:

Cílem práce je vytvoření pomocného studijního materiálu pro studenty FIM UHK, který jim umožní snazší pochopení základů funkcionálního programování. Tato bakalářská práce bude dodatečně studenty informovat o tom, co je funkcionální programování a jeho uplatnění. Hlavní část práce budou tvořit úlohy pro programovací jazyk Haskell, které budou studenta postupně provádět jejich řešením a tím pomohou k pochopení probírané látky. K těmto úlohám bude vhodně doplněn kód a zároveň i popisky, které budou specifikovat v jaké části je snadné udělat chybu.

Osnova:

1. Úvod
2. Funkcionální programování
3. Programovací jazyk Haskell
4. Základy programování v jazyku Haskell
5. Řešené úlohy
6. Závěr

Dybvig, R.K.: The Scheme Programming Language, Fourth Edition. The MIT Press. 2009

LIPOVAČA, M.: *Learn You a Haskell for Great Good!*, No Starch Press, 2011.

WHITINGTON, J.: *Haskell from the Very Beginning*, Coherent Press, 2019

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	prof. RNDr. Josef Hynek, MBA, Ph.D.
Datum zadání závěrečné práce:	26.1.2021