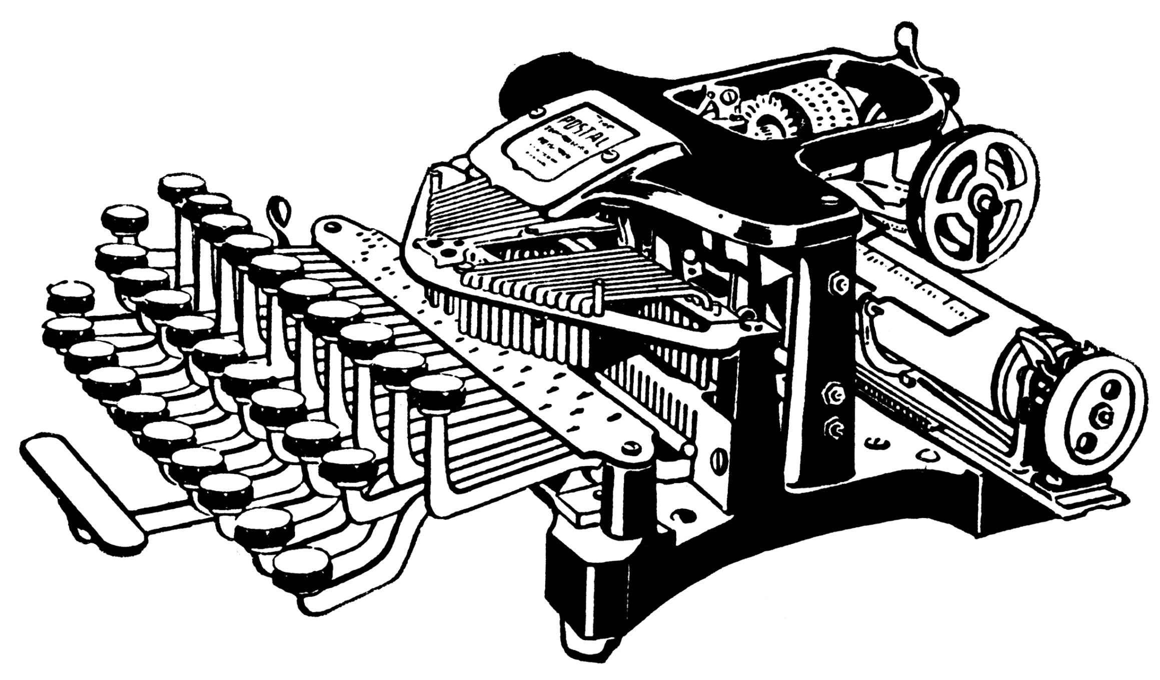


Funkcko

Sbírka příkladů ke cvičením

Fakulta informatiky
Masarykovy univerzity

Změněno: 4. 12. 2023 09.23

Obsah

Cvičení 0: Technické okénko 3
- 0.1 Základní potřebné programy a nastavení . . . 7
- 0.2 Používání GHCi . . . 8
- 0.3 Vzdálené připojení na fakultní počítače . . . 9
- 0.4 Práce s dokumentací . . . 10

Cvičení 1: Základní konstrukce 12
- 1.1 Jednoduché funkce, `if`, `where`/`let` ... `in` 14
- 1.2 Priority operátorů a volání funkcí . . . 15
- 1.3 Definice s více definičními rovnostmi . . . 16
- 1.4 Základní typy . . . 17

Cvičení 2: Rekurze a seznamy 19
- 2.1 Rekurze na číslech . . . 20
- 2.2 Rekurze na seznamech . . . 22

Cvičení 3: Funkce vyšších řádů, λ-funkce, částečná aplikace, skládání 26
- 3.1 Užitečné seznamové funkce & lambda funkce . . . 28
- 3.2 Částečná aplikace a operátorové sekce . . . 31
- 3.3 Skládání funkcí, η-redukce, odstraňování argumentů . . . 32

Interludium: Psaní hezkého kódu 37

Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe` 44
- 4.1 Vlastní datové typy . . . 45
- 4.2 Konstruktor `Maybe` . . . 48
- 4.3 Rekurzivní datové typy . . . 50
- 4.4 Další příklady . . . 52

Cvičení 5: Intensionální seznamy, lenost, foldy 54
- 5.1 Intensionální seznamy . . . 56
- 5.2 Lenost, nekonečné datové struktury . . . 57
- 5.3 Akumulační funkce na seznamech . . . 60
- 5.4 Akumulační funkce na vlastních datových typech . . . 64

Cvičení 6: Manipulace s funkcemi, typy, opakování 69
- 6.1 Typy a typové třídy . . . 72
- 6.2 Opakování základních funkcí . . . 73
- 6.3 Vlastní datové typy . . . 74
- 6.4 Typové třídy podrobněji . . . 75

Cvičení 7: Vstup a výstup (bonus) 77
- 7.1 Skládání akcí operátorem `>>=` . . . 78
- 7.2 IO pomocí `do`-notace, převody mezi notacemi . . . 78
- 7.3 Vstupně-výstupní programy . . . 80

Řešení **82**
Cvičení 0: Technické okénko 82
Cvičení 1: Základní konstrukce 84
Cvičení 2: Rekurze a seznamy 91
Cvičení 3: Funkce vyšších řádů, λ-funkce, částečná aplikace, skládání 100
Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe` 109
Cvičení 5: Intensionální seznamy, lenost, foldy 120
Cvičení 6: Manipulace s funkcemi, typy, opakování 134
Cvičení 7: Vstup a výstup (bonus) 143

Přiložený kód **149**
Soubor `04_data.hs` 149
Soubor `05_data.hs` 151
Soubor `05_treeFold.hs` 153
Soubor `06_data.hs` 154
Soubor `07_guess.hs` 155

Tato sbírka vzniká přispěním mnoha autorů již několik let a stále se vyvíjí. Narazíte-li na nějakou chybu či nejasnost, nahlaste nám ji, prosíme, do diskusního fóra předmětu.

Podoba úvodní strany je s úctou obšlehnuta od O'Reillyho nakladatelství. Ilustraci psacího stroje poskytlo Florida Center for Instructional Technology: `https://etc.usf.edu/clipart`.

Cvičení 0: Technické okénko

Fialové rámečky na začátku každého cvičení obsahují věci, které je bezpodmínečně nutné umět ještě před začátkem cvičení. Neslouží *jako opakování přednášky, avšak jsou vám ku pomoci při přípravě na cvičení.*

Před nultým, technickým cvičením je zapotřebí:

- znát svoje učo (universitní číslo osoby, správnost si ověříte přihlášením do ISu);
- znát svoje fakultní přihlašovací jméno (tzv. fakultní login či *xlogin*) a fakultní heslo (to je jiné než do ISu); tyto přihlašovací údaje si můžete ověřit na `https://fadmin.fi.muni.cz/auth`; svůj *xlogin* zjistíte a heslo můžete změnit na `https://is.muni.cz/auth/system/heslo_fi`;
- projít si tuto kapitolu až po etudy (včetně) – dozvíte se jak funguje sbírka a co si připravit po technické stránce.

Oranžové rámečky obsahují upozornění a varování.

Jindřiška varuje: Na cvičení je normální se připravovat. Nebudete-li s pojmy ve fialových rámečcích srozuměni a neprojdete-li si etudy, pravděpodobně si ze cvičení kromě pocitu neúspěchu příliš neodnesete a hrozí vám vyloučení ze cvičení.

Věnujte prosím pozornost jak fungování fakultních systémů (počítačů v učebnách, studentského serveru Aisa), tak nastavení svých strojů. Mít vše rozjeté na svých strojích se vám bude hodit při řešení domácích úkolů, pokud byste například skončili doma nemocní. Být schopný programovat ve vlaku či jinde bez internetu je též velmi užitečné.

Nulté cvičení je velmi nestandardní. Probíhá pro všechny studenty v prvním týdnu (nejde tedy o 14denní blok) a slouží především k nastavení programů nutných pro fungování v tomto předmětu. Cvičení trvá necelou hodinu a nezabývá se náplní předmětu, ale má za úkol vás seznámit s počítačovou učebnou B130 a jejím programovým vybavením, aby se na prvním cvičení nemusely řešit problémy technického rázu. Čas na něm můžete také využít ke konzultaci problémů s instalací na svých strojích – později na to již na cvičeních čas nebude, takže se budete muset spokojit s konzultacemi na diskusním fóru.

Vysvětlivky I nultá kapitola sbírky je trochu zvláštní – slouží totiž zároveň jako seznámení se sbírkou samotnou. Jednou z věcí zasluhujících vysvětlení je význam piktogramů objevujících se u některých příkladů:

»= Symbolem »= jsou označeny ty příklady, které by se měly stihnout na cvičení. Příklady bez tohoto symbolu tak slouží spíše pro domácí studium a přípravu. Mimochodem, přesný význam tohoto symbolu se dozvíte případně na navazujícím předmětu IB016 Seminář z funkcionálního programování. Pokud mu chcete nějak říkat, můžete používat termín *bind*. Ještě více mimochodem, podobnost s logem Haskellu není náhodná.

Tužkou ( ) jsou označeny příklady, které byste měli být schopni vyřešit správně s použitím pouze tužky a papíru, stejně jako se to po vás bude chtít u zkoušky.[1] Interpret doporučujeme použít až ke kontrole.

[1] Nepodlehněte však iluzi, že před zkouškou stačí projít tužkou označené příklady a zelenou známku máte v kapse. Piktogram totiž nijak nesouvisí s typem příkladů objevujících se u zkoušky.

Démantem (💎) označujeme příklady, které sice nejsou v plánu cvičení, ale přesto je velmi doporučujeme vyřešit, neboť je považujeme za zajímavé či zvláště přínosné. Očekáváme, že tyto příklady budete řešit pravidelně v průběhu semestru a mohou vám pomoci při řešení odpovídajících domácích úkolů.

Mírně obtížnější, ale o to poutavější příklady pak nesou hvězdičku (☆) či několik hvězdiček podle obtížnosti či pracnosti. Trojhvězdičkové příklady mohou sahat i za rámec tohoto předmětu.

K vybraným příkladům existují videa demonstrující postup řešení příkladu. Tato videa jsou alternativou k živému vysvětlení od cvičícího a také slouží k jeho připomenutí.

Etudy

Každá kapitola na začátku obsahuje několik základních příkladů, tzv. etud. Očekává se, že tyto příklady si vyřešíte ještě *před* samotným cvičením. Slouží především k ozkoušení syntaxe jazyka Haskell, případně připomenutí jeho standardních funkcí – tedy přibližně to, co po vás chce fialový rámeček na začátku.

V případě problémů či dotazů se zeptejte na diskusním fóru.

> *Zelené rámečky obsahují rady, doporučení a tipy na další zdroje ke studiu.*
>
> **Pan Fešák doporučuje:** K většině příkladů se na konci sbírky (případně kapitoly, čtete-li tyto samostatně) nachází řešení. Podporuje-li váš prohlížeč dokumentů odkazy, můžete klepnutím na číslo příkladu na jeho řešení skočit.

Etuda 0.η.0 Zkuste si to!

Etuda 0.η.1 Seznamte se s nultou kapitolou sbírky a s organizačními pokyny k předmětu v interaktivní osnově.

Etuda 0.η.2 **Nainstalujte si GHC na svůj počítač.** Standardem pro tento semestr je překladač Glasgow Haskell Compiler (GHC) verze 8.4 nebo vyšší. Doporučujeme instalovat nejnovější dostupný pro váš systém, ideálně tedy verzi 9.2.

Linux Pokud je GHC verze nejméně 8.4 k dispozici v repozitářích vaší distribuce, instalujte přímo z repozitáře. Níže uvádíme návody pro některé běžné distribuce, ve všech případech ukazujeme jak nainstalovat GHC z příkazové řádky pod uživatelem `root`.

- **Ubuntu/Debian** Pokud máte dostatečně novou verzi operačního systému (Debian alespoň 10 = buster = oldstable, Ubuntu alespoň 20.04), stačí vám nainstalovat balík `ghc`:

  ```
  apt install ghc
  ```

  Pro starší verse, či pokud chcete novější versi GHC najdete návod v řešení tohoto příkladu.

- **Fedora**: Ve všech podporovaných verzích Fedory je již dostatečně nové GHC, stačí nainstalovat balík `ghc`:

  ```
  dnf install ghc
  ```

Windows Na Windows je možné instalovat GHC ze stránek GHC, nebo alternativně do Windows Subsystem for Linux (WSL; konkrétně ideálně verzi Ubuntu 20.04 LTS).

Zde popíšeme pouze první možnost, pro tu druhou je třeba aktivovat WSL a následně postupovat podle návodu pro Ubuntu výše.

- Stáhněte GHC ze stránek GHC („Windows 64-bit (x86_64) (GMP)“).
  - rozbalte stažený archiv (například pomocí 7-Zip)
  - přesuňte složku `ghc-X.Y.Z` z rozbaleného archivu do `C:\` (`X.Y.Z` zde odpovídá verzi staženého GHC, např. `ghc-9.2.3` – chcete-li si práci usnadnit, můžete si složku přejmenovat na `ghc` a příponu `-X.Y.Z` v rámci návodu směle ignorovat)
- Dále potřebujeme nastavit proměnné prostředí, abychom mohli GHC/GHCi spouštět kdekoliv v rámci systému.
  - Jděte do „Upravit proměnné prostředí systému“ (najdete přes vyhledávání nebo „Počítač“ → „vlastnosti“ → „upřesnit nastavení systému“ → „upravit proměnné prostředí“);
  - klikněte na „proměnné prostředí“
  - v části „systémové proměnné“ vyberte proměnnou **path** a zvolte upravit
  - klikněte na „nový“ a přidejte cestu `C:\ghc-X.Y.Z\bin` (nebo jinou cestu kam jste GHC přesunuli)
  - uložte

Pro ověření spusťte příkazový řádek (stiskněte win + R a zadejte příkaz `cmd`) a do příkazového řádku napište `ghci`. Mělo by se spustit GHCi ve verzi odpovídající staženému archivu.

macOS Na macOS jsou dvě možnosti instalace – přes Homebrew (doporučené, vyžaduje nainstalovat Homebrew, které se vám však bude hodit pro instalaci mnoha dalších nástrojů při studiu na FI) a nebo pomocí nástroje `ghcup`. V obou případech instalace probíhá z příkazové řádky.

Homebrew instalace

- otevřete Terminal pod `/Applications/Utilities/`
- nainstalujte si Homebrew z příkazové řádky příkazem z `https://brew.sh/`
- nainstalujte GHC příkazem `brew install ghc` (opět z příkazové řádky)
- spusťte `ghci` – macOS jej prvním spuštění pravděpodobně zablokuje, povolte jej tedy v nastavení (viz též apple support)
- dále by již mělo být možné GHCi spouštět normálně

Instalace pomocí ghcup GHC se v tomto případě instaluje pouze pro toho uživatele, který provede níže zmíněný postup.

- jděte na `https://www.haskell.org/ghcup/` a zkopírujte příkaz uvedený na této stránce
- otevřete Terminal pod `/Applications/Utilities/` a vložte do něj příkaz z webu `ghcup`
- postupujte podle instrukcí instalátoru, nezapomeňte si nechat automaticky upravit `PATH` (to zajistí, že GHC budete moci spustit bez zadání celé cesty k binárce v `/.ghcup`)
- restartujte terminál, pak by již mělo být možné GHCi spustit

Etuda 0.η.3 **Nainstalujte si vhodný editor čistého textu na svůj počítač.** Pro psaní kódu v jazyce Haskell, který v tomto předmětu používáme, budete potřebovat editor schopný pracovat se zdrojovým kódem. Jde o editor, který ukládá čistý, neformátovaný text a případně

má funkce, která pomáhají programátorům v pochopení kódu – především zvýrazňování syntaxe (tedy rozlišení různých konstruktů v kódu barvami).

> **Jindřiška varuje:** Word ani LibreOffice Writer **nejsou** editory čistého textu.

Na různých systémech jsou obvyklé různé editory zdrojového kódu, zde uvedeme 2 příklady, které mohou fungovat na všech platformách, můžete však používat i jiné. Ve všech editorech doporučujeme pro Haskell nastavit odsazování pomocí mezer (nikoli tabulátorů), což je v Haskellu standardní konvence. Navíc kombinování odsazování mezerami a tabulátory vede v Haskellu ke špatně pochopitelným chybám. Pokud již máte svůj oblíbený editor zdrojových kódů (který umí pracovat s Haskellem), můžete zbytek této sekce ignorovat.

Visual Studio Code Visual Studio Code (také VS Code, nebo jen code) je zdarma dostupný multiplatformní editor zdrojového kódu od firmy Microsoft. Informace k jeho instalaci na vaši platformu naleznete na webu výrobce, na některých Linuxových distribucích je VS Code k dispozici rovněž v repozitářích (např. jako `code` na Arch Linuxu).

Základní nastavení

- Pod „View“ → „Extensions“ si vyhledejte Haskell a nainstalujte „Haskell Syntax Highlighting“
- Následně již bude fungovat zvýrazňování syntaxe pro soubory s příponou .hs (v pravém dolním rohu byste měli vidět „Haskell“)
- V pravé dolní části si zkontrolujte, že máte nastaveno odsazování mezerami: „Spaces: N“ (nikoli „Tab Size: N“), kde N označuje délku odsazení (pro Haskell je vhodné 4).
- VS Code má i integrovaný terminál, který vám může zjednodušit práci tím, že budete mít editor i terminál v jednom okně.
  - Terminál aktivujete ve „View“ → „Terminal“
  - Nezapomeňte se před spuštěním ghci přepnout do adresáře, v němž máte uložen editovaný soubor (pomocí cd CESTA).
  - Nezapomínejte, že pro to, aby se změny v kódu projevili, je třeba soubor uložit a v ghci provést reload (:r).
- **Pozor, neinstalujte si žádná „pokročilejší“ Haskell rozšíření (například to jménem „Haskell“)** – tato rozšíření vyžadují komplexnější nastavení a často předpokládají, že budete tvořit tzv. balíčky. Pro naše použití tak prakticky způsobují jen komplikace a chyby editoru.

Tip: Pokud si ve VS Code otevřete pracovní složku, kde chcete programovat v Haskellu pomocí „File“ → „Open Folder“, integrovaný terminál se vám pak také otevře v této složce.

Gedit Gedit je výchozí textový editor v několika variantách Linuxové distribuce Ubuntu a na všech Linuxových distribucích by měl být dostupný (obvykle v balíku `gedit`) a je rovněž dostupný pro Windows a macOS. Gedit doporučujeme používat především pokud chcete co možná nejjednodušší editor.

Základní nastavení

- Zvýrazňování syntaxe funguje v Geditu automaticky pro soubor s příponou .hs, není tedy třeba nic nastavovat.
- Doporučujeme zapnout si zobrazování čísel řádků (přes druhou položku zprava na spodní liště).
- Doporučujeme zapnout odsazování mezerami (v Haskellu se nedoporučuje používat tabulátory a už vůbec ne je kombinovat s mezerami). Na spodní liště vpravo klikněte na „Tab Width: N“ a zaškrtněte „Use Spaces“. Může být rovněž dobrý nápad snížit výchozí odsazení na 4 mezery.

Etuda 0.η.4 intro Zhlédněte záznam úvodního democvičení. Můžete k tomu použít odkaz pod číslem tohoto příkladu; vede do studijních materiálů předmětu v ISu.

Po etudách následují příklady určené k řešení na cvičení nebo samostatně po cvičení, například jako příprava na domácí úkol. Nic vám ale samozřejmě nebrání se s příklady seznámit ještě před cvičením.

0.1 Základní potřebné programy a nastavení

> Je v podstatě jedno, zda budete na cvičeních používat svůj počítač nebo školní. Je však silně doporučené abyste měli vše potřebné nainstalované u sebe (kvůli přípravám a domácím úkolům), ale zároveň je i vhodné, abyste uměli používat školní počítače.

Př. 0.1.1 »= **Základní využívání školních počítačů.** Přihlaste se na školní počítač (pomocí svého xloginu a fakultního hesla). Po přihlášení spusťte terminál (buď jej vyhledejte, nebo pomocí klávesové zkratky Ctrl+Alt+T). Po spuštění by měl terminál pracovat ve vašem domovském adresáři (`/home/xLOGIN`). Příkazem `pwd` si vypište aktuální adresář.

> **Základní příkazy příkazové řádky**
> Tyto příkazy by měly fungovat v podstatě na všech platformách, včetně Windows (jejich přesný výstup v případě, že něco vypisují, se však může lišit). Na Windows však může být nutné použít PowerShell místo příkazové řádky (neboli `cmd`).
>
> **`ls`** vypíše obsah aktuálního adresáře. Případně `ls -A` vypíše adresář včetně souborů a složek, jejichž jméno začíná tečkou.
>
> **`mkdir`** slouží k vytvoření adresáře. Argumentem je cesta k adresáři, který chcete vytvořit: `mkdir ib015`. Pokud je potřeba vyrobit mezilehlé adresáře, lze použít `mkdir -p longer/path/to/ib015`.
>
> **`cd`** přepíná adresáře. Jediný argument je cesta k adresáři, kam přepnout. Např. `cd ib015` pro přepnutí do adresáře `ib015`, `cd ..` pro přechod nahoru v adresářové struktuře nebo `cd ~` (vlnovka) pro skok do domovského adresáře (mimo Windows i jen `cd`).

ssh slouží pro vzdálený terminálový přístup na počítač (typicky se systémem Linux/Unix). Argumentem je adresa počítače a případně login: `ssh xLOGIN@aisa.fi.muni.cz` (s vaším loginem). Při prvním přihlášení je třeba potvrdit klíč stroje. Heslo se při zadávání nezobrazuje. Pokud je váš lokální login stejný, jako ten, do kterého se chcete přihlásit, nemusíte jej uvádět: `ssh aisa.fi.muni.cz` (také po vyřešení 0.3.3 níže).

scp slouží ke kopírování souborů mezi počítači. První argument je vždy zdroj, druhý cíl. Jedna z cest je lokální, jedna může být ve tvaru `login@stroj:cesta` a pak představuje cestu na daném stroji. Cílem může být i složka; cesta by pak měla končit lomítkem a složka musí na daném stroji existovat. Např. `scp Foo.hs xLOGIN@aisa.fi.muni.cz:` pro kopírování souboru `Foo.hs` do domovského adresáře na Aise (relativní cesty jsou vzhledem k domovskému adresáři). Aktuální adresář je označen „`.`“ (tečka), má-li být cílem ten.

exit ukončení příkazové řádky nebo sezení SSH. Můžete použít i Ctrl+D.

cp jako `scp`, ale jen lokálně.

mv přesun/přejmenování souboru, argumenty jako u `cp`.

rm/rmdir maže soubory, respektive prázdné složky zadané jako argumenty.

nano jednoduchý editor textu v terminálu (z Aisy nemůžete spustit grafický editor). Příkazy pro ovládání se zobrazují dole,ˆ znamená Ctrl. Na vašem počítači možná nebude.

Př. 0.1.2 Vytvořte si ve svém domovském adresáři adresář `ib015`. Můžete využít terminál nebo grafický správce souborů.

Př. 0.1.3 Většinu úkolů budete vypracovávat v textovém editoru podle svého výběru. Proto si jej nastavte podle instrukcí v etudě 0.η.3, jen s tím rozdílem, že oba editory jsou již nainstalované a stačí je nastavit.

Př. 0.1.4 V adresáři `ib015` vytvořete nový soubor `Sem0.hs` s následujícím obsahem:

```
hello = "Hello, world"
```

0.2 Používání GHCi

Pokud si kapitolu procházíte samostatně, tak v této a následující sekci již přijde vhod democvičení (`intro`), je proto dobrý nápad jej nyní zhlédnout. Může být přínosné si nahrávku zastavovat a postupovat podle ní, případně se k nejasným částem vracet.

Př. 0.2.1 V terminálu v adresáři `ib015` si příkazem `ghci` spusťte interpret Haskellu.

Komu zrovna zadáváte text, poznáte podle tzv. výzvy, neboli promptu. Prompt GHCi má obvykle tvar `Prelude>`, `Main*>` nebo `ghci>` – sem zadáváte výrazy jazyka Haskell a příkazy interpretu (ty, co začínají dvojtečkou). Prompt terminálu nabývá rozličných podob, ale obvykle končí znakem dolaru (např. `login@počítač:~/ib015 $`). Sem zadáváte např. příkazy pro práci se soubory a pohyb po adresářích.

Př. 0.2.2 »= V interpretu si zkuste vyhodnotit jednoduché aritmetické výrazy, tedy využít ho jako kalkulačku.

Př. 0.2.3 »= Do GHCi načtěte soubor `Sem0.hs` a nechte si vypsat konstantu `hello`. Následně si od interpretu vyžádejte její typ.

Př. 0.2.4 »= Do souboru `Sem0.hs` přidejte konstantu `myNumber` typu `Integer` a nastavte ji na svou oblíbenou hodnotu. Po uložení souboru jej znovu načtěte do GHCi a konstantu vypište. Interpret následně ukončete.

> **Pan Fešák doporučuje:** Pokud se někdy program nechová tak, jak očekáváte, ačkoli jste si jisti, že tentokrát už jste ho určitě opravili, zkontrolujte, že jste soubor uložili a znovu načetli do interpretu. Je to jedna z nejčastějších chyb.

0.3 Vzdálené připojení na fakultní počítače

> Používání vzdáleného připojení přes SSH není v tomto předmětu nezbytně nutné, ale může se vám hodit a určitě jej budete potřebovat jinde. Pan Fešák proto doporučuje se ho naučit již nyní. Vzdálené připojení vám umožní používat GHCi a další nástroje na počítačích na FI a dostat se k vašim kódům ze cvičení odkudkoli.

Na školní stroje se systémem Linux se lze připojit pomocí SSH. Na Linuxu či macOS se obvykle jako klient SSH používá příkaz `ssh`. Pokud klient nemáte, mělo by být možné jej nainstalovat z balíčků (bude se nejspíš jmenovat `openssh-client` nebo `openssh`). Na Windows 10 je příkaz `ssh` k dispozici v PowerShellu. Záložní možností je pak klient Putty.

K přihlašování z vnějšku fakulty slouží server `aisa.fi.muni.cz` (ze sítě FI dostupný i jako `aisa`). Přes tento počítač se pak případně dá dostat i k jiným, ale to nebývá potřeba, protože váš domovský adresář je sdílen přes všechny fakultní linuxové stroje.

> **Co se kde děje?** Při práci se vzdáleným počítačem může být trochu nepořádek v tom, co se děje u vás a co na vzdáleném stroji. Měli byste to být schopni poznat podle promptu, tedy textu, který příkazová řádka vypisuje před místem, kam píšete příkaz. Na Aise byste měli (pokud jste si to nezměnili) vidět něco jako `aisa:/home/xLOGIN>$`. V promptu je tedy vidět i cesta k adresáři, v němž se aktuálně nacházíte.
>
> Aby v tom nebyl nepořádek v našich příkladech, používáme konvenci, že příkazy spouštěné na Aise začínají `(aisa)$`, zatímco ty spouštěné u vás, začínají `(local)$` (pokud je to jedno, bude tam jen `$`). Ani jeden z těchto řetězců však nepatří na příkazovou řádku – slouží jen k označení příkazů, které můžete spustit.

> Nápovědu k příkazům a jejich volbám si vyžádejte příkazem `man` nebo volbou `--help`:
>
> ```
> $ man rm
> $ cd --help
> ```

Př. 0.3.1 »= Přihlaste se ze svého nebo školního počítače k serveru `aisa.fi.muni.cz`.

Př. 0.3.2 Pomocí příkazu `scp` si na svůj počítač zkopírujte z Aisy zdrojový kód ze cvičení. U sebe jej upravte a pošlete zpět na Aisu. Pokud jste pracovali na svém počítači, udělejte to obráceně – kód zkopírujte na Aisu, tam ho upravte (třeba pomocí `nano`) a zkopírujte zpět.

Alternativně můžete vyzkoušet některý z klikacích nástrojů: na Windows WinSCP, v Linuxových grafických správcích souborů hledejte volbu „Připojit k serveru“ nebo podobnou.

Př. 0.3.3 Nastavte si klient SSH tak, aby pro připojení k Aise stačilo zadat `ssh aisa`. Stejnou zkratku je pak možné využívat při kopírování souborů mezi počítači: `scp Sem0.hs aisa:ib015/`.

> **Pan Fešák doporučuje:** Než navštívíte svůj oblíbený vyhledávač, vzpomeňte si na příkaz `man`. Zde se bude hodit `man ssh_config`.

Př. 0.3.4

> Tento krok je určen pokročilejším uživatelům a sahá daleko nad rámec předmětu.

Pokud nedisponujete dvojicí klíčů SSH pro asymetrické šifrování, vytvořte si ji. Přidejte na Aise svůj veřejný klíč mezi autorizované, abyste při přihlašování ze svého počítače nemuseli psát heslo.

0.4 Práce s dokumentací

Tak jako u ostatních programovacích jazyků je většina funkcí a typů zdokumentovaná. Primárním zdrojem dokumentace pro jazyk Haskell v rozsahu našeho kurzu je webová dokumentace základního modulu `Prelude`. Dále můžete využít lokální mirror vyhledávače Hoogle, kde můžete vyhledávat funkce podle názvu nebo typu.[3]

> **Jindřiška varuje:** Naučit se pracovat s dokumentací je nevyhnutelné pro libovolný programovací jazyk, nejenom pro Haskell. Čím dříve se naučíte číst dokumentaci, tím budete mít lehčí život nejen tu, ale i v dalších předmětech.

Př. 0.4.1 Pomocí vyhledávače funkcí v jazyce Haskell najděte všechny funkce, které mají typ `Bool -> Bool -> Bool` a jsou v balíčku `base`.

* * *

> **Jindřiška varuje:** Ať už budete používat vlastní či školní počítač, je naprosto nezbytné, abyste před prvním cvičením zvládli jeho obsluhu alespoň v rozsahu fialového rámečku na začátku následující kapitoly.

> *Fialové rámečky na konci cvičení shrnují probrané koncepty. Můžete si s jejich využitím ověřit, jestli všemu ze cvičení rozumíte, nebo je potřeba se k některému tématu vrátit.*
>
> **Na konci cvičení byste měli zvládnout:**

[2] Na Aise je nainstalované příliš staré GHC, které se od toho námi používaného liší typy některých seznamových funkcí.

[3] Obě tyto služby pro vás hostujeme lokálně, protože ty oficiálně se ukázaly být pro naše účely neoptimální – originální dokumentaci najdete na Hackage a vyhledávání i v nestandardních balíčcích na hoogle.haskell.org.

- vytvořit textový soubor s Haskellovým kódem;
- spustit v terminálu GHCi a načíst do něj vytvořený soubor;
- pracovat v terminálu vzdáleně na školním počítači Aisa;
- pracovat s webovou dokumentací.

Cvičení 1: Základní konstrukce

Před cvičením je nezbytné umět:

- otevřít terminál (příkazovou řádku) a vytvořit textový soubor;
- spustit interpret GHCi a načíst do něj soubor;
- používat GHCi jako kalkulačku;
- znát na intuitivní úrovni pojmy *funkce* a *typ*;
- vědět, jakým způsobem se volají funkce v Haskellu;
- vědět, jaké základní typy v Haskellu jsou;
- vědět, jak fungují základní konstrukce `if ... then ... else ...`, `let ... in ...` a `where`;
- umět napsat jednoduchou funkci včetně využití více definičních rovností a vzorů.

Příkazy interpretu GHCi:

`:t[ype]` *výraz* – typ výrazu

`:i[nfo]` *jméno* – informace o operátorech, funkcích a typech

`:doc` *jméno* – dokumentace operátorů, funkcí a typů (až od GHC 8.6)

`:l[oad]` *soubor.hs* – načtení souboru s Haskellovým kódem

`:r[eload]` – znovunačtení posledního souboru

`:q[uit]` – ukončení práce s interpretem

`:m[odule]` *Modul* – načtení modulu (bude používáno později)

`:h[elp]` – nápověda

Pan Fešák připomíná: Všechny funkce v Haskellu jsou *čisté funkce*, tj. nemají žádný vnitřní stav a pro stejné argumenty vždy vrací stejnou hodnotu (na rozdíl od funkcí v Pythonu).

Etudy

Etuda 1.η.1 S pomocí dokumentace zjistěte, jaký je rozdíl mezi typy `Int` a `Integer`.

Etuda 1.η.2 Definujte funkci `isSucc :: Integer -> Integer -> Bool`, která pro dvě celá čísla `x`, `y` rozhodne, jestli je `y` bezprostředním následníkem `x`.

```
isSucc 1 4 ⇝* False
isSucc 4 5 ⇝* True
```

Etuda 1.η.3 S pomocí konstrukce `if ... then ... else ...` definujte funkci `firstNonZero :: Integer -> Integer -> Integer`, která vezme dvě celá čísla; pokud je první z nich nenulové, tak je vrátí, v opačném případě vrátí to druhé.

```
firstNonZero 1 4 ⇝* 1          firstNonZero 0 0 ⇝* 0
firstNonZero 0 4 ⇝* 4
```

*
* *

Pan Fešák doporučuje: Priorita operátorů je jednou z věcí, které jsou nutné pro správné pochopení vyhodnocování výrazů. Je vhodné naučit se používat dotaz `:i`, který mimo jiné obsahuje i prioritu zadaného operátoru.

Příklad dotazu v interpretu (řádek začínající `>` je zadán uživatelem):

```
> :i *
class Num a where
  ...
  (*) :: a -> a -> a
  ...
        -- Defined in ‘GHC.Num’
infixl 7 *
```

Pro nás je nyní podstatný poslední řádek, který říká, že se jedná o infixový operátor, a popisuje jeho prioritu a asociativitu. V tomto případě `infixl 7 *` znamená, že se jedná o operátor priority 7, který se závorkuje (asociuje) zleva (`infixl`; zprava by bylo `infixr`), tedy například `2 * 3 * 7` se vyhodnocuje jako by bylo uzávorkované `(2 * 3) * 7`. Například pro `==` dostaneme `infix 4 ==`, což znamená, že `==` nelze řetězit s dalšími operátory na stejné úrovni priority a jeho priorita je 4.

Pokud nemá operátor (funkce) explicitně definovanou prioritu, jeho priorita je 9 a je asociativní zleva. I některé binární funkce zapsané písmeny mají určenou prioritu a asociativitu, kterou využijí, pokud jsou zapsány infixově. Příkladem takové funkce je `div`.

Konečně prefixová aplikace má vždy přednost před aplikací infixovou, například ve výrazu `div 3 2 ^ 4` se provede nejprve dělení a až pak umocňování, jako by byl výraz uzávorkovaný `(div 3 2) ^ 4`.

Etuda 1.η.4 S použitím interpretu jazyka Haskell porovnejte vyhodnocení následujících dvojic výrazů a rozdíl vysvětlete.

a) `5 + 9 * 3` versus `(5 + 9) * 3`
b) `2 ^ 2 ^ 2 == (2 ^ 2) ^ 2` versus `3 ^ 3 ^ 3 == (3 ^ 3) ^ 3`
c) `3 + 3 + 3` versus `3 == 3 == 3`
d) `("Haskell" == "je") == "super"` versus `('a' == 'a') == ('a' == 'a')`

Pan Fešák doporučuje: Operátory a funkce se můžou vyskytovat v prefixové i infixové verzi. Proto si dohledejte, jaký je rozdíl mezi `*` vs. `(*)` a `mod` vs. `` `mod` ``.

Etuda 1.η.5 Definujte s využitím více definičních rovností funkci `isWeekendDay :: String -> Bool`, která rozhodne, jestli je daný řetězec jméno víkendového dne.

```
isWeekendDay "Saturday" ⇝* True
isWeekendDay "Monday" ⇝* False
isWeekendDay "apple" ⇝* False
```

Etuda 1.η.6 Určete typy následujících výrazů a najděte další výrazy stejného typu. Své řešení si ověřte s pomocí interpretu.

a) `'a'`
b) `"Don't Panic."`
c) `not`
d) `(&&)`
e) `True`
f) `('a', True)`

1.1 Jednoduché funkce, `if`, `where`/`let ... in ...`

> **Pan Fešák doporučuje:** Pokud jste si ještě nevytvořili soubor pro toto cvičení, teď je dobrý čas jej vytvořit. Funkce z dalších příkladů pište do souboru a pak je testujte v GHCi.

Př. 1.1.1 S využitím interního příkazu `:info` interpretu GHCi zjistěte prioritu a asociativitu následujících operací:

```
^, *, /, +, -, ==, /=, >, <, >=, <=, &&, ||
```

Př. 1.1.2 Vytvořte funkci `circleArea :: Double -> Double`, která pro zadaný poloměr spočítá obsah kruhu o tomto poloměru. Přibližná hodnota konstanty π se dá v Haskellu získat pomocí konstanty `pi`.

Př. 1.1.3 Napište funkci `snowmanVolume :: Double -> Double -> Double -> Double`, která spočítá celkový objem sněhuláka s koulemi o zadaných poloměrech. Vzpomeňte si, že objem koule s poloměrem r je $\frac{4}{3} \cdot \pi \cdot r^3$. Přibližná hodnota konstanty π se dá v Haskellu získat pomocí konstanty `pi`. V řešení zkuste vhodně využít lokální definici (`where` nebo `let ... in ...`).

```
snowmanVolume 1 1 1     ⇝* 12.566370614359172
snowmanVolume 1 2 3     ⇝* 150.79644737231007
snowmanVolume 1 0 0     ⇝* 4.1887902047863905
```

> **Jindřiška varuje:** V Haskellu odsazujeme vždy mezerami, nikoli tabulátory. Používání tabulátorů rychle vede ke kombinaci obou variant a velmi podivným chybovým hláškám.

> **Jindřiška varuje:** V porovnání s jinými (převážně imperativními) jazyky má Haskell striktnější pravidla pro konstrukci `if ... then ... else ...` a je nutné je všechna znát.
>
> V imperativních jazycích totiž Haskellovému výrazu `if p then t else f` neodpovídá řídicí příkaz `if p then t else f`, ale ternární operátor: `p ? t : f` (nejen) v jazyce C nebo `t if p else f` v Pythonu.

Př. 1.1.4 1.1.4 Pro zadané tři kladné délky stran trojúhelníka rozhodněte, zda se jedná o pravoúhlý trojúhelník. Pravoúhlý trojúhelník je možné poznat tak, že pro délky jeho stran platí Pythagorova věta (tedy součet druhých mocnin dvou kratších stran je roven druhé mocnině nejdelší strany). V řešení zkuste vhodně využít lokální definici (`where` nebo `let ... in ...`).

```
isRightTriangle 3 4 5     ⇝* True      isRightTriangle 70 42 56 ⇝* True
isRightTriangle 42 42 42 ⇝* False     isRightTriangle 25 24 7  ⇝* True
```

Př. 1.1.5 Definujte funkci `max3 :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to největší z nich. Naprogramujte dvě verze, jednu pomocí funkce `max` a jednu pomocí `if ... then ... else ...`.

Př. 1.1.6 Naprogramujte funkci `mid :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to *prostřední* z nich (tj. to druhé v jejich uspořádané trojici podle $\leq$).

```
mid 1 2 3 ⇝* 2                mid 15 113 111 ⇝* 111
mid 42 16 69 ⇝* 42            mid 42 42 42 ⇝* 42
```

Př. 1.1.7 Pomocí `if` a funkce `mod` definujte funkci `tell :: Integer -> String`, která bere jako argument jedno kladné celé číslo `n` a vrací:

- `"one"` pro `n = 1`,
- `"two"` pro `n = 2`,
- `"(even)"` pro sudé `n > 2` a
- `"(odd)"` pro liché `n > 2`

Př. 1.1.8 U následujících výrazů rozhodněte, zda jsou správně, a pokud jsou špatně, zdůvodněte proč a vhodným způsobem je upravte.

a) `if 5 - 4 then False else True`
b) `if 0 < 3 && odd 6 then 0 else "FAIL"`
c) `(if even 8 then (&&)) (0 > 7) True`
d) `if 42 < 42 then (&&) else (||)`

> **Pan Fešák doporučuje:** Pokud v konstrukci `if ... then ... else ...` mají větve `then` a `else` typ `Bool`, pak je vhodné se zamyslet nad tím, zda celá konstrukce není zbytečná a problém nelze vyřešit použitím vhodných logických operátorů.
>
> Příkladem je výraz `if` *`podmínka`* `then True else False`, který se dá zjednodušit na výraz *`podmínka`*.

1.2 Priority operátorů a volání funkcí

Př. 1.2.1 Doplňte všechny implicitní závorky do následujících výrazů:

a) `recip 2 * 5`
b) `sin pi + 2 / 5`
c) ``f g 3 * g 5 `mod` 7``
d) `42 < 69 || 5 == 6`
e) `2 + div m 18 == m ^ 2 ^ n && m * n < 20`

> **Pan Fešák doporučuje:** Pokud nevíte, co daná funkce nebo operátor dělá, je vhodné si ji najít v dokumentaci. Příkladem může být operátor `(||)`.

Př. 1.2.2 Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:

a) ``4 ^ (7 `mod` 5)``
b) `max 3 ((+) 2 3)`

Př. 1.2.3 Doplňte všechny implicitní závorky do následujících výrazů:

1.2.3

a) `f . g x`
b) `2 ^ mod 9 5`
c) `f . (.) g h . id`
d) `2 + div m 18 * m `mod` 7 == m ^ 2 ^ n - m + 11 && m * n < 20`
e) `f 1 2 g + (+) 3 `const` g f 10`
f) `replicate 8 x ++ filter even (enumFromTo 1 (3 + 9 `mod` x))`
g) `id id . flip const const`

Př. 1.2.4 Zjistěte (bez použití interpretu), na co se vyhodnotí následující výraz. Poté všech sedm funkcí přepište do prefixového tvaru a pomocí interpretu ověřte, že se hodnota výrazu nezměnila.

```
5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1
```

Př. 1.2.5 Do následujícího výrazu doplňte implicitní závorky a pak převeďte všechny operátory v něm do prefixového tvaru.

```
2 + 2 * 3 == 2 * 4 && 8 `div` 2 * 2 == 2 || 0 > 7
```

1.3 Definice s více definičními rovnostmi

Také nazývána **definice podle vzoru** podle toho, že smysluplná definice s více definičními rovnostmi musí využívat různé vzory v různých definičních rovnostech.

Př. 1.3.1 Bez použití podmíněného výrazu `if ... then ... else ...` definujte funkce

```
logicalNot :: Bool -> Bool
logicalAnd :: Bool -> Bool -> Bool
logicalOr  :: Bool -> Bool -> Bool
```

které se chovají stejně jako funkce logické negace, konjunkce a disjunkce.

Nesmíte využít žádné logické funkce definované v Haskellu.

Př. 1.3.2 Definujte funkci `isSmallVowel :: Char -> Bool`, která rozhodne, jestli je dané písmeno malou samohláskou anglické abecedy. Znakové literály se v Haskellu píší do apostrofů, například literál znaku a se v Haskellu zapíše jako `'a'`.

> **Jindřiška varuje:** V Haskellu, na rozdíl od například Pythonu (a mnoha dalších skriptovacích jazyků), je rozdíl mezi věcmi uzavřenými mezi apostrofy `'...'` a uvozovkami `"..."`. Mezi apostrofy je uzavřen vždy jeden znak (typu `Char`), zatímco mezi uvozovkami se jedná o řetězec (typ `String`; řetězec se skládá ze znaků).

Př. 1.3.3 Naprogramujte funkci `solveQuad :: Double -> Double -> Double -> (Double, Double)`, která vezme na vstupu koeficienty `a`, `b`, `c` a vyřeší kvadratickou rovnici $ax^2 + bx + c = 0$ s v reálných číslech. Připomeňme, jak se řeší kvadratická rovnice:

- Pokud $a \neq 0$, je třeba spočítat diskriminant $d = b^2 - 4ac$. Řešení pak jsou $\frac{-b-\sqrt{d}}{2a}$ a $\frac{-b+\sqrt{d}}{2a}$. Tato řešení vraťte ve dvojici (může se stát, že řešení budou stejná, vždy však musíme vrátit dvojici).
- Pokud $a = 0$, pak jde o lineární rovnici a výsledek je $x = -\frac{c}{b}$. Vraťte dvě identické hodnoty ve dvojici.
- Situaci, kdy diskriminant vychází záporný či $a = b = 0$, neřešte (vůbec tento případ

nezohledňujte).

V řešení vhodně využijte více definičních rovností a lokální definice.

```
solveQuad 1 0 (-1) ⇝* (-1.0, 1.0)
solveQuad 0 2 2 ⇝* (-1.0, -1.0)
solveQuad 1 (-6) 5 ⇝* (1.0, 5.0)
```

Př. 1.3.4 Naprogramujte funkci `parallelToAxis`, která o úsečce zadané souřadnicemi bodů v rovině rozhodne, jestli je rovnoběžná s jednou ze souřadnicových os roviny. Můžete předpokládat, že vstup skutečně představuje úsečku, tedy že vstupní body jsou různé.

Poznámka: Vyhněte se funkcím `fst` a `snd` a použijte výhradně vzory.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (0, 0) (1, 1) ⇝* False
parallelToAxis (1, 1) (1, 4) ⇝* True
parallelToAxis (16, 42) (12, 19) ⇝* False
parallelToAxis (4, 2) (9, 2) ⇝* True
```

Př. 1.3.5 Vzpomeňte si na příklad 1.1.7 a zkuste jej napsat pomocí více definičních rovností.

1.4 Základní typy

Pan Fešák doporučuje: Příkaz GHCi `:t` je dobrý pro kontrolu, ale je vždy lepší být si schopen určit typy sám. A stejně tak je dobré u funkcí typy vždy psát. Tím si procvičujete přemýšlení v typech a budete moci plně využít toho, že vám je Haskell umí kontrolovat, a může tak poznat rozdíl mezi vaší představou o tom, co má funkce dělat (reflektovanou v typu, který jste napsali), a co skutečně dělá (což je reflektované v typu, který si GHC odvodí).

Př. 1.4.1 Nalezněte příklady hodnot následujících typů:

a) `Bool`
b) `Integer`
c) `Double`
d) `False`
e) `(Int, Integer)`
f) `(Integer, Double, Bool)`
g) `()`
h) `((), (), ())`

Př. 1.4.2 Určete typy následujících výrazů.

a) `True`
b) `"True"`
c) `not True`
d) `True || False`
e) `True && ""`
f) `fun 1`, kde funkce `fun` je nějaká funkce typu

```
fun :: Integer -> Integer
```

g) `fun 3.14`, kde `fun` je stejného typu jako v části f)
h) `solve 3 8`, kde `solve` je nějaká funkce typu

```
solve :: Int -> Int -> Int
```

Př. 1.4.3 1.4.3 Určete typ následujících funkcí, které jsou definovány předpisem:

a)
```
implication a b = not a || b
```

b)
```
foo _   "42" = True
foo 'a' _    = True
foo _   _    = False
```

c)
```
ft True x y = False
ft x    y z = y
```

Př. 1.4.4 Naprogramujte funkci `isDivisibleBy :: Integral a => a -> a -> Bool`, která pro dvě celá čísla `x` a `y` rozhodne, zda je `x` dělitelné `y`.

> `Integral a =>` je omezení na typy, které lze dosadit za typovou proměnnou `a`. V tomto případě za `a` lze dosadit celočíselné typy, například `Int`, `Integer`. Obdobně například omezení `Num a =>` znamená, že za typovou proměnnou `a` lze dosadit číselný typ (třeba `Integer` nebo `Double`).

> **Jindřiška varuje:** U operací *dělení* (a *zbytek po dělení*) je nutné rozlišovat operace pracující s celými čísly (`div` a `mod`) a dělení desetinných čísel (`/`).

Př. 1.4.5 Vysvětlete, jak je možné, že výrazy `(3 / 2)` a `(15 / 3)` vyprodukují desetinné číslo i přes to, že jejich argumenty jsou celá čísla, zatímco výraz `(3 + 2)` vyprodukuje celé číslo.
Poznámka: Může se vám hodit podívat se na typy různých aritmetických operací.

* * *

Na konci cvičení byste měli zvládnout:

- ovládat základní příkazy interpretu GHCi a prakticky je využívat;
- otypovat jednoduché výrazy a funkce;
- vytvářet jednoduché funkce pracující s čísly a pravdivostními hodnotami;
- vytvářet funkce s více definičními rovnostmi;
- prakticky umět využít podmíněný výraz `if` a lokální definice pomocí `let ... in ...` nebo `where`.

Cvičení 2: Rekurze a seznamy

Před druhým cvičením je nezbytné umět:

- porozumět zápisu typu funkce, včetně polymorfních funkcí a základních typových tříd;
- vytvářet funkce s více definičními rovnostmi;
- zapisovat seznamy pomocí výčtu prvků, tj. například `[1, 2, 10]`;
- používat základní funkce pro práci se seznamy, tj. například

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
```

- používat základní vzory pro seznamy, tj. například `[]`, `(x:xs)`, `[x]`, `[x, y]`.

Etudy

Etuda 2.η.1 Určete typy následujících seznamů. Nejprve intuitivně, následně je ověřte v interpretu.

a) `["a", "b", "c"]`
b) `['a', 'b', 'c']`
c) `"abc"`
d) `'a' : 'b' : 'c' : []`
e) `[(True, ()), (False, ())]`
f) `[(++) "a" "b", "X" ++ "Y"]`
g) `[(&&), (||)]`
h) `[]`
i) `[[]]`
j) `[[], [True]]`

Etuda 2.η.2 Bez použití jakýchkoli knihovních funkcí definujte funkci `isEmpty` typu `[a] -> Bool`, která pro prázdný vstupní seznam vrátí `True` a pro neprázdný vrátí `False`.

Etuda 2.η.3 Bez použití knihovních funkcí napište funkci `myHead` typu `[a] -> a`, která vrátí první prvek zadaného neprázdného seznamu, a funkci `myTail` typu `[a] -> [a]`, která pro zadaný neprázdný seznam vrátí tentýž seznam bez prvního prvku.

Etuda 2.η.4 Napište funkci `neck :: a -> [a] -> a`, která vrátí druhý prvek zadaného seznamu, případně „náhradní“ prvek, je-li zadaný seznam prázdný nebo jednoprvkový. Nevyužívejte knihovní funkce, pouze vlastní funkce s vhodným použitím vzorů.

Etuda 2.η.5 Pro následující vzory a seznamy určete, které vzory mohou reprezentovat které seznamy. Stanovte, jak se navážou proměnné ze vzoru.

- Vzory: `[]`, `x`, `[x]`, `[x, y]`, `(x : s)`, `(x : y : s)`, `[x : s]`, `((x : y) : s)`
- Seznamy: `[1]`, `[1, 2]`, `[1, 2, 3]`, `[[]]`, `[[1]]`, `[[1], [2, 3]]`

Etuda 2.η.6 Bez použití funkcí `mod`, `div`, `even` a `odd` naprogramujte rekurzivní funkci `isEven`, která pro dané nezáporné celé číslo rozhodne, jestli je sudé. Určete i její typ. Záporná čísla neřešte. Nejprve se zamyslete: jak se dá určit sudost čísla na základě sudosti menšího čísla?

Protože zadání omezuje vstupy na nezáporná čísla, nemusíte se chováním funkce na záporných vstupech trápit. Pro zajímavost si ale zkuste funkci se záporným argumentem vyhodnotit a chování vysvětlit. Pro násilné zastavení nekončícího výpočtu použijte klávesovou zkratku Ctrl + C (mnemotechnická pomůcka: *cancel*). Vyzkoušejte si i rozdíl mezi `isEven -1` a `isEven (-1)` a vysvětlete jej.

2.1 Rekurze na číslech

Př. 2.1.1 Bez použití knihovní funkce `mod` naprogramujte rekurzivně funkci `mod3`, která pro dané nezáporné celé číslo vypočítá jeho zbytek po dělení 3. Nezapomeňte uvést i její typ. Například:

```
mod3 0 ⇝* 0          mod3 7 ⇝* 1
mod3 5 ⇝* 2          mod3 9 ⇝* 0
```

⁂

Typové třídy. Minule jsme viděli, že v typových signaturách funkcí pracujících s čísly se nemusíme omezovat jen konkrétní typy (jako `Integer`), ale můžeme využívat *typové třídy*, které nám dovolí do funkcí předávat hodnoty různých, ale v něčem podobných typů. Vzpomeňte si na typovou třídu `Num` pro libovolná čísla a `Integral` pro celočíselné typy. Například předchozí funkce by mohla mít obecnější typ

```
mod3 :: Integral i => i -> i
```

nebo dokonce

```
mod3 :: (Integral i1, Integral i2) => i1 -> i2
```

V následujících příkladech již na vhodných místech místo konkrétních (tzv. *monomorfních*) číselných typů používejte *polymorfní* typy omezené typovými třídami. Jeden z obvyklých případů, kde se přesto používá monomorfní `Int`, jsou různé indexy, počty a délky.

Pan Fešák doporučuje: Informace o typových třídách a v nich definovaných funkcích můžeme zjistit pomocí příkazu `:i` v GHCi. Např.

```
> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
        -- Defined in ‘GHC.Real’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

To nám říká, že typová třída `Fractional` obsahuje funkce `(/)`, `recip` a `fromRational`, že do ní patří typy `Float` a `Double` a že je-li něco ve třídě `Fractional`, pak je to nutně také ve třídě `Num`. Krom toho nám tento výpis také dává informace o tom, které funkce bychom minimálně museli implementovat, pokud bychom chtěli do této třídy nějaký nový typ přidat.

Chceme-li zjistit, zda jde nějaký kontext, například (Num a, Floating a) zjednodušit, potřebujeme zjistit, zda některá z těchto typových tříd vyžaduje některou další. V tomto případě bude potřeba navíc jít přes mezikrok:

```
class Fractional a => Floating a where {- ... -}
class Num a => Fractional a where {- ... -}
```

Tedy pokud je nějaký typ ve Floating, musí být nutně ve Fractional a tedy i v Num. Kontext tedy zjednodušíme na Floating a.

Př. 2.1.2 2.1.div

Bez použití knihovní funkce div naprogramujte rekurzivně funkci div3, která dané nezáporné celé číslo celočíselně vydělí třemi. Nezapomeňte uvést typ s vhodnou typovou třídou. Například:

```
div3 0 ⇝* 0        div3 7 ⇝* 2
div3 5 ⇝* 1        div3 9 ⇝* 3
```

Př. 2.1.3 Implementujte rekurzivní funkci pro výpočet faktoriálu a určete její typ.

Př. 2.1.4 Definujte funkci power takovou, že power x n se pro nezáporné celé číslo n vyhodnotí na x na n-tou. Všimněte si, že x nemusí být celé číslo. Začněte napsáním typové signatury s vhodnými typovými třídami. Příklad vyhodnocení:

```
power 3 1 ⇝* 3       power 3 0 ⇝* 1
power 3 2 ⇝* 9       power 0 3 ⇝* 0
power 3 3 ⇝* 27      power 0.5 3 ⇝* 0.125
```

* * *

Predikát je funkce vracející Bool podle vlastností svých argumentů.

Př. 2.1.5 Napište predikát isPower2, který o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky. Nezapomeňte funkci otypovat s využitím vhodné typové třídy. Mohou se vám hodit funkce even a odd. Například:

```
isPower2 0 ⇝* False      isPower2 6 ⇝* False
isPower2 1 ⇝* True       isPower2 8 ⇝* True
isPower2 2 ⇝* True       isPower2 9 ⇝* False
```

Nápověda: Mocnina dvojky je dvojnásobkem jiné mocniny dvojky.

Př. 2.1.6 Definujte funkci digitsSum, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet. Nezapomeňte na typ. Můžete použít funkce div a mod. Například:

```
digitsSum 123 ⇝* 6
digitsSum 103 ⇝* 4
```

Př. 2.1.7 Napište funkci mygcd, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Nepoužívejte funkci gcd. Pokuste se o co nejefektivnější implementaci.

```
mygcd 18 10 ⇝* 2       mygcd  4 16 ⇝* 4
mygcd  9 27 ⇝* 9       mygcd 15  8 ⇝* 1
```

Př. 2.1.8 Naprogramujte funkci primeDivisors, která rozhodne, součinem kolika prvočísel je zadané kladné číslo. Můžete použít funkce div a mod. Například:

```
primeDivisors 3  ⇝* 1
primeDivisors 4  ⇝* 2
primeDivisors 5  ⇝* 1
primeDivisors 6  ⇝* 2
primeDivisors 8  ⇝* 3
primeDivisors 12 ⇝* 3
primeDivisors 15 ⇝* 2
primeDivisors 1  ⇝* 0
```

Př. 2.1.9 Definujte funkce `plus` a `times`, které budou ekvivalentní operátorům `(+)` a `(*)` na přirozených číslech. Nepoužívejte vestavěné `(+)` ani `(*)`. Můžete však používat libovolné jiné funkce, doporučujeme podívat se zejména na funkce `pred` a `succ` (jejich typ je ve skutečnosti o něco obecnější, ale můžete uvažovat, že to je `Integral i => i -> i`).

Bonus: implementujte funkce `plus'` a `times'`, které budou fungovat na všech celých číslech.

Př. 2.1.10 Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

```
fun 0 = 0
fun n = fun (n - 1) + 2 * n - 1
```

2.2 Rekurze na seznamech

Př. 2.2.1 Napište nerekurzivní funkci, která na začátek zadaného seznamu čísel vloží hodnotu 42. Jaký má vaše funkce typ? Nezapomeňte na existenci typových tříd!

Př. 2.2.2 Bez použití knihovní funkce `last` definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu.

Př. 2.2.3 Bez použití funkce `init` definujte funkci `stripLast` typu `[a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku.

Př. 2.2.4 Bez použití knihovní funkce `length` definujte funkci `len :: [a] -> Integer`, která spočítá délku zadaného seznamu.

Př. 2.2.5 2.2.nth Napište funkci `nth :: Int -> [a] -> a`, která ze zadaného seznamu vrátí prvek na pozici určené prvním argumentem funkce. Počítá se od nuly. Můžete předpokládat, že vstupní seznam má dostatečnou délku. Nepoužívejte knihovní funkce. Například:

```
nth 0 [4, 3, 5] ⇝* 4
nth 1 [4, 3, 5] ⇝* 3
nth 2 [4, 3, 5] ⇝* 5
nth 2 "Get Schwifty" ⇝* 't'
```

* * *

> Typová třída `Eq` sdružuje typy, jejichž hodnoty můžeme porovnávat operátory `(==)` a `(/=)`. Tuto typovou třídu potřebujeme i pro využití vzorů s konkrétními hodnotami čísel (typová třída `Num` negarantuje `Eq`, `Integral` však již ano).

Př. 2.2.6 Napište funkci `contains :: Eq a => [a] -> a -> Bool`, která vrací `True`, pokud seznam v prvním argumentu obsahuje prvek zadaný druhým argumentem, jinak vrací `False`. Nepoužívejte funkci `elem` a jí podobné. Například:

```
contains [1, 2, 3, 4] 42 ⇝* False
[1, 2, 3] `contains`  2 ⇝* True
[]        `contains` () ⇝* False
```

```
"TEAM"    `contains` 'I' ⇝* False
```

Př. 2.2.7 Napište funkci `containsNtimes` typu `Eq a => Integer -> a -> [a] -> Bool` takovou, že `containsNtimes n x xs` bude `True` právě tehdy, když seznam `xs` obsahuje alespoň `n` výskytů hodnoty `x`. Například:

```
containsNtimes 2 42 [1, 2, 42] ⇝* False
containsNtimes 2 42 [1, 42, 2, 42] ⇝* True
containsNtimes 3 42 [1, 42, 2, 42] ⇝* False
containsNtimes 0 42 [1, 2] ⇝* True
containsNtimes 2 'l' "Haskell" ⇝* True
```

Př. 2.2.8 Mějme neprázdný seznam typu `[(String, Integer)]`, který reprezentuje seznam jmen studentů s jejich počty bodů z předmětu IB015. Naprogramujte

a) funkci `getPoints :: String -> [(String, Integer)] -> Integer`, která vrátí počet bodů studenta se jménem zadaným v prvním argumentu (nebo 0, pokud takový student v seznamu není),

b) funkci `getBest :: [(String, Integer)] -> String`, která vrátí jméno studenta s nejvíce body.

Například tedy:

```
getPoints "Stan" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 20
getPoints "Tomas" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 0
getBest [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* "Eric"
```

Př. 2.2.9 2.2.app Napište funkci `append :: [a] -> [a] -> [a]`, jejíž výsledek pro dva seznamy bude seznam, který vznikne zřetězením těchto dvou seznamů. Nepoužívejte knihovní operátor `(++)`. Například:

```
append [1, 2] [3, 4, 8] ⇝* [1, 2, 3, 4, 8]
append [] [3, 4] ⇝* [3, 4]
append [3, 4] [] ⇝* [3, 4]
append "Legen" "dary" ⇝* "Legendary"
```

Nápověda: `[1,2] ++ [3,4] = [1,2,3,4] = 1:2:3:4:[] = 1:2:[3,4]`

Př. 2.2.10 Napište funkci `pairs :: [a] -> [(a, a)]`, která bere prvky ze vstupního seznamu po dvou prvcích a vytváří seznam dvojic těchto prvků. Pokud má seznam lichý počet prvků, poslední prvek se zahodí. Například:

```
pairs [4, 8, 15, 16, 23] ⇝* [(4, 8), (15, 16)]
pairs [4, 8, 15, 16, 23, 42] ⇝* [(4, 8), (15, 16), (23, 42)]
pairs "Humphrey" ⇝* [('H', 'u'), ('m', 'p'), ('h', 'r'), ('e', 'y')]
```

Př. 2.2.11 Napište následující funkce pracující se seznamy čísel pomocí rekurze a vzorů:

a) `listSum :: Num n => [n] -> n`, která dostane seznam čísel a vrátí součet všech jeho prvků.

b) `oddLength :: [a] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (bez použití funkce `length`).

c) `add1 :: Num n => [n] -> [n]`, která každé číslo ve vstupním seznamu zvýší o 1,

d) `multiplyN :: Num n => n -> [n] -> [n]`, která každé číslo ve vstupním seznamu vynásobí prvním argumentem funkce,

e) `deleteEven :: Integral i => [i] -> [i]`, která ze seznamu čísel odstraní všechna sudá čísla,

f) `deleteElem :: Eq a => a -> [a] -> [a]`, která ze seznamu odstraní všechny výskyty hodnoty zadaného prvním argumentem,
g) `largestNumber :: [Integer] -> Integer`, vrátí největší číslo ze zadaného neprázdného seznamu čísel,
h) `listsEqual :: Eq a => [a] -> [a] -> Bool`, která dostane na vstup dva seznamy a vrátí `True` právě tehdy, když se rovnají (bez použití funkce `(==)` na seznamy),
i) `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2 (lichá čísla vynechá),
j) `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní (může se vám hodit funkce `sqrt`).

Př. 2.2.12 2.2.every

Napište funkci `everyNth :: Integer -> [a] -> [a]` takovou, že seznam `everyNth n xs` bude obsahovat každý n-tý prvek ze seznamu `xs`. Například:

```
everyNth 2 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 1, 2, 7]
everyNth 3 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 3, 7]
everyNth 4 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 2]
everyNth 1 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 8, 1, 3, 2, 5, 7]
everyNth 2 "BoJack Horseman" ⇝* "BJc osmn"
```

Př. 2.2.13

Napište funkci `brackets :: String -> Bool`, která dostane řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování. Například:

```
brackets "(()())" ⇝* True
brackets "(()()" ⇝* False
brackets "())()" ⇝* False
brackets "())(" ⇝* False
brackets "" ⇝* True
```

Př. 2.2.14

Zadefinujte predikát `palindrome`, který o řetězci na vstupu rozhodne, jestli je palindrom. Napište také funkci `palindromize`, která ze zadaného řetězce udělá palindrom tak, že na jeho konec doplní co nejméně znaků. Tedy například:

```
palindrome "lol" ⇝* True
palindrome "ABBA" ⇝* True
palindrome "brienne" ⇝* False
palindromize "brienne" ⇝* "brienneirb"
```

Př. 2.2.15

Napište funkci `getMiddle :: [a] -> a`, která pro zadaný neprázdný seznam vrátí jeho prostřední prvek bez zjišťování jeho délky. Pokud má seznam sudý počet prvků, vraťte levý z prostředních dvou. Například:

```
getMiddle [1] ⇝* 1
getMiddle [2, 1] ⇝* 2
getMiddle [2, 1, 5] ⇝* 1
getMiddle [2, 1, 5, 6] ⇝* 1
getMiddle [2, 1, 5, 6, 3] ⇝* 5
getMiddle "Don't blink!" ⇝* ' '
```

Nápověda: Pokud zajíc běží dvakrát rychleji než želva, pak v okamžiku, kdy zajíc vyhrál závod, je želva v polovině trati.

* * *

Na konci druhého cvičení byste měli umět:

- definovat vlastní rekurzivní funkce pracující s celými čísly;

- pracovat se seznamy a definovat na nich funkce s více definičními rovnostmi a použitím vzorů;
- definovat rekurzivní funkce na seznamech;
- použít v typech svých funkcí základní typové třídy jako `Num` a `Eq`.

Cvičení 3: Funkce vyšších řádů, λ-funkce, částečná aplikace, skládání

Před třetím cvičením je zapotřebí znát:

- lokální definice pomocných funkcí pomocí klauzule `where`

```
power :: Int -> Int
power x = result
  where result = x * x
```

- chování funkcí

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- zápis anonymních funkcí pomocí λ-abstrakce, tj. například `\x y -> x + y`
- co je částečná aplikace;
- použití operátoru `(.) :: (b -> c) -> (a -> b) -> (a -> c)` pro skládání unárních funkcí;

Etudy

Etuda 3.η.1 Slovně vysvětlete, co dělají knihovní funkce `map` a `filter`. Pro jaký účel byste použili kterou z nich? Jaký je rozdíl mezi výsledkem vyhodnocení `map even [1, 3, 2, 5, 4, 8, 11]` a `filter even [1, 3, 2, 5, 4, 8, 11]`?

Pan Fešák připomíná: Lambda je řecké písmeno (λ; velká lambda je Λ). V teorii programování se používá pro uvození anonymní funkce a proto se těmto funkcím říká lambda funkce. Protože písmeno λ není na běžných klávesnicích, používá se místo něj v Haskellu `\`.

Etuda 3.η.2 Naprogramujte funkci `oddify :: Integral a => [a] -> [a]`, která projde vstupní seznam čísel a čísla, která nejsou lichá, přemění na lichá. Použijte funkci `map`, nebo `filter`. Může se vám také hodit lambda funkce.

Například:

```
oddify [1, 2, 3, 4] ⇝* [1, 3, 3, 5]
```

Etuda 3.η.3 S využitím funkce `oddify` z předchozího příkladu 3.η.2 naprogramujte funkci `inputWithOddified :: Integral a => [a] -> [(a, a)]`, která pro vstupní seznam čísel vrátí seznam dvojic, kde první složka odpovídá prvku ze vstupního seznamu a druhá složka výstupu funkce `oddify`. Například:

```
inputWithOddified [1, 2, 3, 4] ⇝* [(1, 1), (2, 3), (3, 3), (4, 5)]
```

Etuda 3.η.4 Bez použití interpretu určete hodnoty následujících výrazů:

a) `(head . head) [[1,2,3], [4, 5, 6]]`
b) `((\(x:xs) -> xs) . head) "ahoj"`
c) `let g x = x : [x] in g 10`
d) `(last . last) [[1, 2], [3, 4], []]`
e) `let f g x = (g . g) x in f (+ 21) 0`

Pokud výraz nelze vyhodnotit, určete přesně, která část výpočtu selhává a proč.

Etuda 3.η.5 Vysvětlete rozdíl mezi následujícími dvojicemi výrazů:

a) `(f . g) x` versus `g (f x)`
b) `(f . g) x` versus `f (g x)`
c) `(* 2) . (+ 2)` versus `\x -> (x + 2) * 2`
d) `head . head [[1], [2], [3]]` versus `(head . head) [[1], [2], [3]]`

Doposud jsme se se bavili o n-árních funkcích, tedy o funkcích, které braly na vstupu n argumentů a vracely nějakou hodnotu. Ve skutečnosti ale v Haskellu nemáme funkce, které by braly více než jeden argument – jinak řečeno, funkce v Haskellu jsou nejvýše unární. Jak je to možné?

Kouzlo tkví v jiném pohledu na funkce jako takové. Místo toho, abychom se na funkci dívali jako na něco, co bere n argumentů a vrátí hodnotu, se na ni můžeme dívat jako na unární funkci, která vrátí jinou $(n-1)$-ární funkci. Aritu potom můžeme brát jako počet unárních funkcí, které dostaneme postupnou aplikací na argumenty.

Vezměme si například funkci

```
add :: Int -> Int -> Int
add x y = x + y
```

Doposud bychom řekli, že funkce `add` bere dva argumenty a ty sečte.

Pokud se na ni ale podíváme druhým způsobem, zjistíme, že typ funkce `add` lze ekvivalentně zapsat jako `add :: Int -> (Int -> Int)`, a tedy že bere jeden argument – `x` – a vrací funkci, která ke svému vstupu přičte dané `x`.

A přesně tuto myšlenku využíváme u částečné aplikace. Díky tomu, že funkci dáme jeden argument, zafixujeme jeho hodnotu a dostaneme funkci, která *nasytí* zbytek. Potom pro nás není problém zavést si třeba funkci `addTo42` jako

```
addTo42 :: Int -> Int
addTo42 = add 42
```

Tato funkce svůj vstupní argument přičte k hodnotě `42`. Na tento zápis se můžeme dívat i tak, že do `addTo42` uložíme funkci, kterou nám vrátí výraz `add 42`.

Protože druhý pohled na funkce je Haskellu vlastní a operátory nejsou nic jiného než funkce, mohli jsme například úlohu 2.2.1 vyřešit jako `prepend42 = (:) 42`.

Etuda 3.η.6 Vysvětlete, co dělají následující funkce, a najděte argumenty, na něž je lze aplikovat. Následně si chování ověřte v GHCi.

a) `take 4`
b) `(++) "Hello, "`
c) `zip3 [1, 2, 3] ["a", "b"]`
d) `(^ 2)`

Etuda 3.η.7 Do následujících výrazů doplňte všechny implicitní závorky vycházející z částečné aplikace funkcí. Jinak řečeno, explicitně závorkami ukažte pořadí postupného vyhodnocování čás-

tečné aplikace.

a) `(==) 42 16`
b) `map ((==) 42) [1, 2, 3]`
c) `(g 4, f g 5)`
d) `zipWith3 f (g 4 a) xs`
e) typ: `Bool -> Bool -> Bool`
f) typ: `(a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]`
g) typ: `(b -> c) -> (a -> b) -> a -> c`

3.1 Užitečné seznamové funkce & lambda funkce

> **Pan Fešák připomíná** Pokud si nejste chováním některé funkce jistí, můžete ji najít v **dokumentaci**. Teď se například může hodit najít si seznamové funkce jako `map` a `filter`.

Př. 3.1.1 »= Naprogramujte funkci `filterOutShorter :: [String] -> Int -> [String]`, která pro vstupní seznam řetězců a zadané číslo `n`, vrátí seznam řetězců obsahující pouze ty, které mají délku alespoň `n`. Využijte knihovní funkci `filter` a vhodnou lambda funkci.

Př. 3.1.2 »= Mějme seznam typu `[(String, Integer)]`, který obsahuje jména studentů a jejich počty bodů z předmětu IB015. Pomocí vhodných seznamových funkcí naprogramujte

a) funkci `getNames :: [(String, Integer)] -> [String]`, která vrátí seznam jmen studentů,
b) funkci `successfulRecords :: [(String, Integer)] -> [(String, Integer)]`, která ze zadaného seznamu vybere záznamy těch studentů, kteří mají alespoň 8 bodů,
c) funkci `successfulNames :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere jména studentů, kteří mají alespoň 8 bodů,
d) funkci `successfulStrings :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere studenty, kteří mají alespoň 8 bodů, a vrátí seznam řetězců ve tvaru `"jmeno: xxx b"` (*nápověda*: pro převod čísla na řetězec můžete použít funkci `show`).

Tedy například pro databázi

```
st :: [(String, Integer)]
st = [("Finn", 5), ("Jake", 9), ("Bubblegum", 12),
      ("Ice King", 2), ("BMO", 15), ("Marceline", 9)]
```

budou požadované funkce vracet následující hodnoty:

```
getNames st ⇝* ["Finn", "Jake", "Bubblegum", "Ice King", "BMO",
                "Marceline"]
successfulRecords st ⇝* [("Jake", 9), ("Bubblegum", 12), ("BMO", 15),
                         ("Marceline", 9)]
successfulNames st ⇝* ["Jake", "Bubblegum", "BMO", "Marceline"]
successfulStrings st ⇝* ["Jake: 9 b", "Bubblegum: 12 b", "BMO: 15 b",
                         "Marceline: 9 b"]
```

Pan Fešák doporučuje: Pro práci s dvojicemi se můžou hodit funkce `fst` a `snd`, které vrací první, respektive druhou položku dvojice.

Pokud chceme s dvojicemi pracovat v lambda funkcích (nebo i pojmenovaných funkcích), tak dává často smysl používat vzory pro dvojice.

Př. 3.1.3 3.1.mf

Které z funkcí z příkladu 2.2.11 lze elegantně naprogramovat pomocí funkce `map`? Které lze elegantně naprogramovat pomocí funkce `filter`? Všechny tyto funkce pomocí `map` a `filter` naprogramujte.

Př. 3.1.4 S využitím funkce `map` a knihovní funkce `toUpper :: Char -> Char` z modulu `Data.Char` (tj. je třeba použít `import Data.Char`, na začátku souboru, nebo `:m +Data.Char` v interpretu) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen. Například:

```
toUpperStr "i am the one who knocks!" ⇝* "I AM THE ONE WHO KNOCKS!"
```

Př. 3.1.5 Napište funkci `vowels`, která dostane seznam řetězců a vrátí seznam řetězců takových, že v každém řetězci ponechá jenom samohlásky (ale zachová jejich pořadí). Například:

```
vowels ["Michael", "Dwight", "Jim", "Pam"] ⇝* ["iae", "i", "i", "a"]
vowels ["MICHAEL", "DWIGHT", "JIM", "PAM"] ⇝* ["IAE", "I", "I", "A"]
```

Př. 3.1.6 Slovně vysvětlete, co dělají funkce `zip` a `zipWith`. Pro jaký účel byste použili kterou z nich? Pomocí interpretu zjistěte, jak se tyto funkce chovají, pokud mají vstupní seznamy různou délku.

Př. 3.1.7 Mějme výsledky běžeckého závodu reprezentované pomocí seznamu typu `[String]`, který obsahuje jména běžců seřazených od nejlepšího po nejhoršího, a seznam peněžních výher typu `[Integer]` (rovněž seřazených). Naprogramujte

a) funkci `assignPrizes` typu `[String] -> [Integer] -> [(String, Integer)]`, která každému běžci, který něco vyhrál, přiřadí jeho výhru, a
b) funkci `prizeTexts` typu `[String] -> [Integer] -> [String]`, která vrátí seznam řetězců ve tvaru `"jmeno: xxx Kc"` pro každého běžce, který něco vyhrál.

Například:

```
assignPrizes ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    [("Mike", 100), ("Dustin", 50)]
prizeTexts ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    ["Mike: 100 Kc", "Dustin: 50 Kc"]
```

Př. 3.1.8 Nahraďte v následujících výrazech `_lf1`/`_lf2` vhodnými lambda funkcemi tak, aby výsledek obou výrazů odpovídal uvedenému vyhodnocení. Jaká je arita vašich funkcí a jaké argumenty berou?

Poznámka: `zip3` a `zipWith3` jsou obdoby funkcí `zip` a `zipWith` které ale pracují se třemi seznamy místo dvou.

```
map _lf1 (zip3 [True, False, False, True, False]
               [1, 2, 3, 4] [16, 42, 7, 1, 666])
    ⇝* [1, 42, 7, 4]
zipWith3 _lf2 [7, 4, 11, 2] [5, 7, 1] [16, 5, 0, 1]
    ⇝* [16, 7, 11]
```

Př. 3.1.9 Pomocí funkce `zip` napište funkci `neighbors :: [a] -> [(a, a)]`, která pro zadaný seznam vrátí seznam dvojic sousedních prvků. Například:

```
neighbors [3, 8, 2, 5] ⇝* [(3, 8), (8, 2), (2, 5)]
neighbors [3, 8] ⇝* [(3, 8)]
neighbors [3] ⇝* []
neighbors "Kree!" ⇝* [('K', 'r'), ('r', 'e'), ('e', 'e'), ('e', '!')]
```

Př. 3.1.10 Napište funkci, která zjistí, jestli jsou v seznamu čísel některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Př. 3.1.11 Implementujte funkce `myMap`, `myFilter` a `myZipWith`, které se budou chovat jako knihovní funkce `map`, `filter` a `zipWith`.

Př. 3.1.12 Uvažte funkci `anyEven :: [Integer] -> Bool`, která rozhodne, jestli je v seznamu čísel nějaké sudé číslo, a funkci `allEven :: [Integer] -> Bool`, která rozhodne, jestli jsou všechna čísla v seznamu sudá. Najděte ve standardní knihovně funkci nebo funkce, pomocí kterých lze funkce `anyEven` a `allEven` implementovat jednoduše bez explicitního použití rekurze.

Př. 3.1.13 Zjistěte, co dělají funkce `takeWhile` a `dropWhile`.

Př. 3.1.14 Slovně popište, co dělají následující funkce, určete jejich arity a typy:

a) `\x -> 4 * x + 2`
b) `\x y -> x + 2 * y`
c) `\(x, y) -> x + y`
d) `\x y -> x`
e) `\(x, y) -> x`
f) `\(x:xs) -> x`

Př. 3.1.15 Mějme seznam barev ve formátu RGB reprezentovaném trojicí `(Int, Int, Int)`, kde první složka odpovídá červené, druhá zelené a třetí modré. Dále předpokládejme, že hodnoty v trojicích náleží do intervalu $[0, 255]$. Naprogramujte s využitím λ-funkcí

a) funkci `blueless :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam barev obsahující pouze ty barvy, které neobsahují žádnou modrou složku,
b) funkci `greyscale :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam těch barev, které jsou odstínem šedi (tj. všechny složky mají stejnou hodnotu),
c) funkci `polychromatic :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam barev, které obsahují více než jednu nenulovou složku,
d) funkci `colorsToString :: [(Int, Int, Int)] -> [String]`, která převede barvy na řetězce ve formátu `"r: <Int> g: <Int> b: <Int>"`.

Př. 3.1.16 Pomocí rekurze a funkce `filter` napište funkci `quickSort :: [Integer] -> [Integer]`, která seřadí vstupní seznam vzestupně pomocí algoritmu *quick sort*. Například:

```
quickSort [5, 3, 8, 12, 1] ⇝* [1, 3, 5, 8, 12]
quickSort [5, 4, 3, 2] ⇝* [2, 3, 4, 5]
quickSort [2, 2, 2] ⇝* [2, 2, 2]
quickSort [2] ⇝* [2]
quickSort [] ⇝* []
```

Pokud algoritmus quick sort neznáte, zkuste ho nastudovat například na Wikipedii.

3.2 Částečná aplikace a operátorové sekce

Př. 3.2.1 Vysvětlete, co dělají následující funkce, a najděte argumenty, na něž je lze aplikovat. Následně si chování ověřte v GHCi.

a) `(^) 3`
b) `(^ 3)`
c) `(3 ^)`
d) `(- 2)`
e) `(2 -)`
f) `zipWith (+) [1, 2, 3]`
g) `map (++ "!")`
h) `/ 2`

Př. 3.2.2 3.3.part
Přepište v následujících definicích seznamových funkcí lambda funkce na částečnou aplikaci tak, aby funkčnost zůstala stejná.

a)
```
sumLists :: Num a => [a] -> [a] -> [a]
sumLists xs ys = zipWith (\x y -> x + y) xs ys
```
b)
```
import Data.Char
upper :: String -> String
upper xs = map (\x -> toUpper x) xs
```
c)
```
embrace :: [String] -> [String]
embrace xs = map (\x -> '[' : x) (map (\x -> x ++ "]") xs)
```
d)
```
sql :: (Ord a, Num a) => [a] -> a -> [a]
sql xs lt = map (\x -> x ^ 2) (filter (\x -> x < lt) xs)
```

Př. 3.2.3
Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` $\stackrel{?}{\equiv}$ `f 1 (g 2)`
b) `(f 1 g) 2` $\stackrel{?}{\equiv}$ `(f 1) g 2`
c) `(* 2) 3` $\stackrel{?}{\equiv}$ `2 * 3`
d) `(*) 2 3` $\stackrel{?}{\equiv}$ `(* 3) 2`

Př. 3.2.4
Rozhodněte, které z následujících výrazů jsou vzájemně ekvivalentní. Určete minimální arity všech identifikátorů v každém výrazu (tedy určete, jakou minimálně aritu daná entita musí mít, aby mohl být výraz korektní).

a) `a f g b`
b) `(a f) (g b)`
c) `((a f) g) b`
d) `a (f (g b))`
e) `(a f) g b`
f) `a f (g b)`
g) `a (f g b)`

Pan Fešák doporučuje: Všimněte si, že na to, abychom určili ekvivalenci výrazů, nemusíme vůbec vědět, co funkce dělají ani jaké jsou jejich typy. Podstatné je jen to, jak funguje volání funkcí v Haskellu. Z výrazu samotného můžeme vždy určit co jsou funkce a co jejich argumenty, přičemž argument samozřejmě může být funkčního typu, ale to nijak neovlivňuje volání funkce, do které ho předáváme (ovlivňuje to ale její typ).

Př. 3.2.5 Určete, které z následujících typů jsou vzájemně ekvivalentní. Určete aritu funkcí, které nesou tento typ a určete, zda jde o funkce vyšších řádů (tj. funkce, které berou jako argumenty funkce). Předpokládejme, že typy `A` až `E` jsou libovolné konkrétní typy.

a) `A -> B -> C -> D -> E`
b) `(A -> B) -> C -> D -> E`
c) `A -> (B -> C -> D -> E)`
d) `A -> B -> C -> (D -> E)`
e) `A -> ((B -> C) -> D -> E))`
f) `A -> (B -> (C -> (D -> E)))`
g) `(((A -> B) -> C) -> D) -> E`
h) `A -> (B -> C) -> D -> E`

3.3 Skládání funkcí, η-redukce, odstraňování argumentů

Pan Fešák vysvětluje: Řecké písmeno η je éta (a jeho ocásek se píše pod linku, stejně jako například u našeho j). Pojem η-redukce pochází z lambda kalkulu a představuje odstraňování formálních argumentů. Někdy se můžete setkat také s pojmem η-konverze, který představuje jak odebírání argumentů, tak i jejich přidávání (tam, kde to typ dovoluje). Písmeno η bylo vybráno kvůli souvislosti s *extensionalitou*, tedy s tvrzením $f = g \iff \forall x.(f(x) = g(x))$.

Pan Fešák vysvětluje: Odstraněním argumentů z funkce ji převedeme na *pointfree* tvar, naopak pokud funkce má v definici všechny argumenty, o nichž hovoří její typ, je v *pointwise* tvaru. Onen *point* v těchto názvech představuje argument funkce (bod), nikoli tečku (skládání funkcí), více naleznete na Haskell Wiki. Mezi těmito tvary lze vždy převádět, někdy to však není vhodné či snadné, jednak kvůli čitelnosti, jednak je obtížné odstranit argument, který je v definici funkce použit vícekrát.

Jindřiška varuje: Nic se nesmí přehánět, funkce jako `(.(,)) . (.) . (,)` nebo `(.) . (.)` nejsou ani hezké, ani čitelné.

Př. 3.3.1 Otypujte následující výrazy:

a) `map even`
b) `map head . snd`
c) `filter ((4 >) . last)`
d) `const const`

Př. 3.3.2 Implementujte následující funkce s použitím `map`/`filter` a bez použití lambda funkcí a vlastních pomocných funkcí – tedy použijte vhodně částečnou aplikaci a skládání funkcí.

Pracovat budeme opět se záznamy o studentech z příkladu 3.1.2, tedy s typem `[(String, Integer)]`.

a) `countStudentsByPoints :: Integer -> [(String, Integer)] -> Int`, která spočte, kolik studentů dostalo právě počet bodů daný druhým argumentem.

b) `studentNamesByPoints :: Integer -> [(String, Integer)] -> [String]`, která vrátí seznam jmen studentů, kteří dostali daný počet bodů.

c) `studentsStartingWith :: Char -> [(String, Integer)] -> [(String,`

`Integer)]`, která vrátí seznam záznamů studentů, jejichž jméno začíná písmenem daným prvním argumentem.

```
countStudentsByPoints 5 st ⇝* 1
countStudentsByPoints 9 st ⇝* 2
countStudentsByPoints 0 st ⇝* 0

studentNamesByPoints 5 st ⇝* ["Finn"]
studentNamesByPoints 9 st ⇝* ["Jake", "Marceline"]

studentsStartingWith 'J' st ⇝* [("Jake", 9)]
studentsStartingWith 'B' st ⇝* [("Bubblegum", 12), ("BMO", 15)]
studentsStartingWith 'X' st ⇝* []
```

Př. 3.3.3 3.4.comp

Přepište v následujících definicích seznamových funkcí lambda funkce pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí, pokud to je smysluplné.

a)
```
failing :: [(Int, Char)] -> [Int]
failing sts = map fst (filter (\t -> snd t == 'F') sts)
```

b)
```
embraceWith :: Char -> Char -> [String] -> [String]
embraceWith l r xs = map (\x -> l : x ++ [r]) xs
```

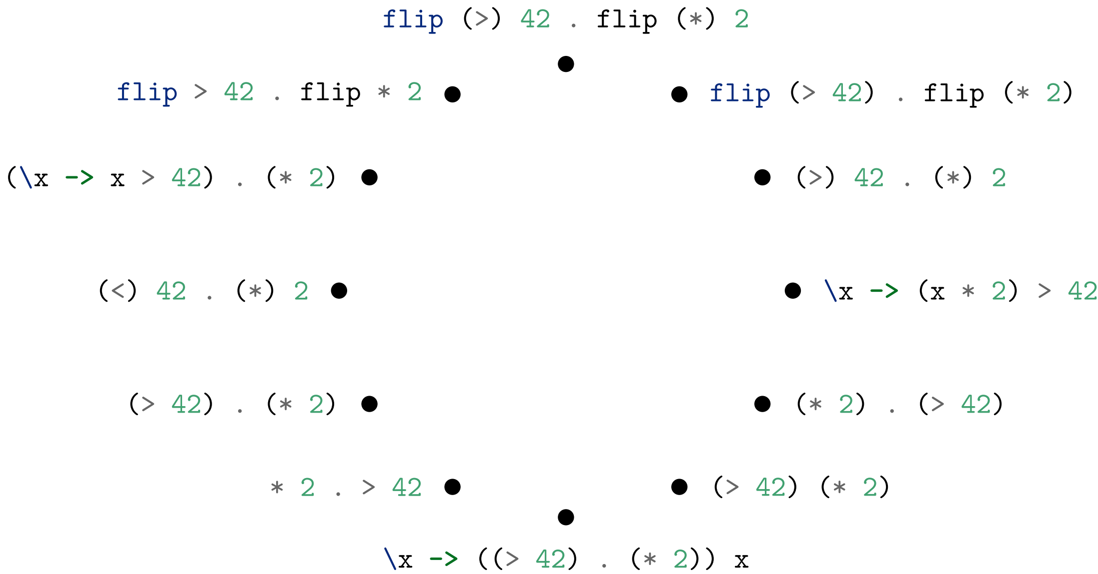
(`l` a `r` neodstraňujte)

c)
```
divisibleBy7 :: [Integer] -> [Integer]
divisibleBy7 xs = filter (\x -> x `mod` 7 == 0) xs
```

d)
```
import Data.Char
letterCaesar :: String -> String
letterCaesar xs = map (\x -> chr (3 + ord x)) (filter isLetter xs)
```

e)
```
zp :: (Integral a, Num b) => [a] -> [b] -> [b]
zp xs ys = zipWith (\x y -> y ^ x) xs ys
```

Př. 3.3.4 Mezi následujícími výrazy najděte všechny korektní a mezi nimi rozhodněte, které jsou vzájemně ekvivalentní (vzhledem k chování na libovolných vstupech povolených typem výrazu). Zdůvodněte neekvivalenci.



Př. 3.3.5 Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních predikátů (funkcí typu `a -> Bool`). Tj. funkce `negp pred` vrátí opačnou logickou hodnotu, než by vrátil predikát `pred` na zadané hodnotě. Tedy například `negp even` by mělo být ekvivalentní s `odd`.

a) Definujte funkci `negp` (můžete využít třeba funkci `not`).
b) Definujte funkci `negp` jako unární funkci (s použitím pouze jednoho formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Př. 3.3.6 Pokud to je možné, přepište lambda funkce v následujících definicích pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí.

```
import Data.Char

-- | Convert lowercase letters to numbers 0..25
l2c :: Char -> Int
l2c c = ord c - ord 'a'

-- | Convert 0..25 character codes to uppercase letters
c2l :: Int -> Char
c2l c = chr (c + ord 'A')

-- | Keep only lowercase English letters
lowAlphaOnly :: String -> String
lowAlphaOnly xs = filter (\x -> isLower x && isAscii x) xs

-- | Encrypt messages using Vigenere (one-time-pad) cipher
letterVigenere :: String -> String -> String
letterVigenere xs ks = zipWith
    (\x y -> c2l ((l2c x + l2c y) `mod` 26))
    (lowAlphaOnly xs)
    (lowAlphaOnly ks)
```

Nápověda: formální argument nelze odstranit (s tím, co jsme se učili), pokud je v definici použit vícekrát.

K odstranění formálního argumentu v použitého vícekrát v těle funkce lze použít funkci `(<*>)`. Její typ je sice velice obecný (a komplikovaný), ale pro tyto účely ji můžeme otypovat jako `(<*>) :: (a -> b -> c) -> (a -> b) -> a -> c`. Rovněž ji pro tyto účely můžeme nahradit následující funkcí:

```
dist :: (a -> b -> c) -> (a -> b) -> a -> c
dist f g x = f x (g x)
```

Př. 3.3.7 Převeďte následující funkce do pointfree tvaru, neboli odstraňte formální argumenty lambda abstrakcí:

a) `\x -> (f . g) x`
b) `\x -> f . g x`
c) `\x -> f x . g`

Př. 3.3.8 Převeďte následující výrazy do pointwise tvaru, neboli přidejte všechny argumenty, které plynou z typu výrazu:

a) `(^ 2) . mod 4 . (+ 1)`
b) `(+) . sum . take 10`
c) `map f . flip zip [1, 2, 3]` (funkce `f` je definována externě)
d) `(.)`

Př. 3.3.9 Určete typ následujících funkcí. Přepište tyto definice funkcí tak, abyste v jejich definici nepoužili λ-abstrakci a formální parametry (tj. chce se pointfree definice).

> **Pan Fešák vysvětluje:** Pokud potřebuji odstranit formální parametr, jenž se nevyskytuje v těle výrazu, pomůžu si funkcí `const`: pro libovolný výraz `w`, který nepřidává žádná nová typová omezení, je výraz `v` ekvivalentní s výrazem `const v w`. Tento výraz už obsahuje v těle navíc parametr `w`, který se dá použít pro η-redukci.

a) `f x y = y`
b) `f x y = 3 + x`

Př. 3.3.10 Převeďte následující funkce do pointfree tvaru:

a) `_ -> x`
b) `\x -> f x 1`
c) `\x -> f 1 x True`
d) `const x`
e) `\x -> 0`
f) `\x -> if x == 1 then 2 else 0`
g) `\f -> flip f x`

Př. 3.3.11 Převeďte všechny níže uvedené funkce do pointfree tvaru. Při převodu třetí si pomozte převodem druhé.

a) `f1 x y z = x`
b) `f2 x y z = y`
c) `f3 x y z = z`

Př. 3.3.12 Zapište v pointfree tvaru funkci `g x = f x c1 c2 c3 ... cn` (`f` je nějaká pevně daná funkce a `c1`, `c2`, ..., `cn` jsou konstanty).

* * *

Na konci třetího cvičení byste měli umět:

- poznat, kdy je na práci se seznamy vhodné použít knihovní funkce `map`, `filter`, `zip` a `zipWith`, a umět tyto funkce použít;
- poznat, kdy je vhodné použít lambda funkce, a umět je použít;
- použít a otypovat částečně aplikovanou funkci;
- použít a otypovat operátorové sekce;
- skládat unární funkce pomocí operátoru `(.)`.

Interludium: Psaní hezkého kódu

Zanedlouho budou pocvikové odpovědníky vystřídány velkými úlohami. Ty budou kromě automatického testování hodnoceny ještě cvičícími na kvalitu kódu. Co činí kód kvalitním, není snadné vyjádřit, a ani to není ambicí této vsuvky. Ta vám ale může pomoci vyhnout se nedostatkům, s nimiž se při opravování úloh setkáváme nejčastěji.

Takřka ke každému z pravidel níže by se daly najít výjimky. Jako základní přehled tzv. *antipatternů* by vám tento seznam měl ale posloužit dobře a určitě je vhodné, abyste si před odevzdáním svoje řešení ještě jednou prošli a ujistili se, že se žádných ze zde uvedených poklesků nedopouštíte. Naopak, pokud některý z těchto vzorů ve svém kódu máte, je celkem vysoká šance, že vám jej opravující vyčte a nezískáte tolik bodů za kvalitu.

Hodnotu typu `Bool` a výrazy `if`

Podmíněný výraz je jistě užitečný, v případě že je však jeho výsledkem pravdivostní hodnota je často nadužíván a je vhodnější použít logické spojky. To platí obvzlášť tehdy, je-li alespoň jednou z větví `if`u konstanta `True` nebo `False`.

Nechť `p` a `r` jsou výrazy typu `Bool`. Následující konstrukce jsou považovány za nečitelné a je žádoucí je nahradit, neboť jen komplikovaným způsobem popisují samotné `p` (nebo `not p`):

- Výraz tvaru `p == True` je možné nahradit výrazem `p`.
- Podobně `p == False` → `not p`.
- `if p then True else False` → `p`.
- `if p then False else True` → `not p`.

Také v případech následujícího tvaru bude použití logické spojky čitelnější:

- `if p then r else False` → `p && r`.
- `if p then True else r` → `p || r`.
- `if p then False else r` → `not p && r`.
- `if p then r else True` → `not p || r`.

Jiným zneužitím `if`u je jeho použití jako v imperativních jazycích, kde je to podmíněný *příkaz*. V Haskellu se ale jedná o *výraz* (a odpovídá třeba operátoru `?:` z jazyka C nebo výrazu `t if p else e` v jazyce Python), který nemusí být nejvnějšnějším výrazem funkce. Umožňuje nám to použít ho třeba jako parametr jiné funkce. Pokud jsou obě větve `if`u hodně podobné, zkuste společný kód „vytknout“ mimo něj:

- `if p then f x z else f y z` → `f (if p then x else y) z`

Je-li takto přepsaný kód dlouhý a nepřehledný (což dost možná bude), můžete čitelnosti výrazně pomoci přesunutím podmíněného výrazu do lokální definice:

```
let w = if p then x else y
in f w z        -- samozřejmě s lepším jménem než ilustračním ‚w‘
```

Takovéto případy se naštěstí v kódu snadno detekují a snadno přepisují.

Používání if a pomocných funkcí místo vzorů

V rekurzi je vhodné oddělit bázový případ do samostatných definičních řádků (pokud to jde). Je to výrazně přehlednější než používat `if`. Uvažme následující funkci:

```
digitsSum :: Integral i => i -> i
digitsSum x = if x == 0 then 0 else x `mod` 10 + digitsSum (x `div` 10)
```

Zde je jediný bázový případ, bylo by tedy lepší funkci přepsat takto:

```
digitsSum' :: Integral i => i -> i
digitsSum' 0 = 0
digitsSum' x = x `mod` 10 + digitsSum' (x `div` 10)
```

Výhodou tohoto zápisu je, že je na první pohled vidět, co je báze a co je rekurzívní část funkce. Obecně je většinou přehlednější používat vzory než podmínky.

Obdobně **při práci se seznamy je vhodné je dekomponovat pomocí vzorů**. Tedy například místo

```
listSum :: Num n => [n] -> n
listSum [] = 0
listSum x  = head x + listSum (tail x)
```

je lepší použít vzor pro neprázdný seznam:

```
listSum' :: Num n => [n] -> n
listSum' []     = 0
listSum' (x:xs) = x + listSum xs
```

O to více je tento přístup důležitý, když například **zkombinujeme seznamy a ntice**. Uvažme:

```
larger :: Ord a => [(a, a)] -> [a]
larger [] = []
larger xs = max (fst (head xs)) (snd (head xs)) : larger (tail xs)
```

Tato funkce je již poměrně nepřehledná, lze ji přitom napsat i velmi krátce:

```
larger' :: Ord a => [(a, a)] -> [a]
larger' [] = []
larger' ((a, b) : xs) = max a b : larger' xs
```

To je lepší a (po troše tréningu na vzory) přehlednější. *Poznámka*: tuto konkrétní funkci by bylo ještě lepší vyřešit pomocí `map`:

```
larger'' :: Ord a => [(a, a)] -> [a]
larger'' xs = map (\(x, y) -> max x y) xs
```

Vymýšlení kola

Nedostatky z této kategorie jsou obtížnější na detekci, neboť již vyžadují nějaký přehled o tom, jaké funkce balík `base` nabízí. Rozhodně se vám tak vyplatí se alespoň přibližně seznámit s poskytovanými funkcemi v dokumentaci. Často se může hodit zeptat se: „není tohle natolik základní funkcionalita, že by mohla být v `Prelude` nebo `Data.List`?“. Pokud jste se s ním doposud nesetkali, `Data.List` je modul, který, jak název napovídá, obsahuje užitečné funkce pro práci se seznamy. Máte-li již zkušenost s jinými programovacími jazyky, může být dobrým vodítkem i to, co nabízí v základu ony.

- **Používání explicitní rekurze místo použití funkcí `map`, `filter`, `zipWith`, `all`, `and` apod.** – výrazně zhoršuje čitelnost, protože na první pohled je jasné jen to, že funkce pracuje se seznamem. Naproti tomu použití těchto standardních funkcí spolu s dobře pojmenovanou pomocnou funkcí nebo krátkou a čitelnou lambdou umožňuje zápis přečíst téměř přirozeně.
  - Pokud potřebujete nějaký seznam zároveň protřídit i transformovat, můžete použít skládání `map` a `filter`, ale ještě čitelnější může být třeba použití intensionálního zápisu seznamu, pokud už jste se s nimi seznámili.
- **Definování vlastních verzí funkcí jako `fst`, `snd`, `last`, `replicate`, `elem`, `take` apod.** – zhoršuje čitelnost ostatními (tj. v tomto případě cvičícími), přidává vám práci a zvyšuje riziko zanesení chyby.
- **Konkrétní výrazy, kterých se dá elegantně zbavit:**
  - `s == []` nebo `length s == 0` → `null s` – tyto tři způsoby dokonce ani nejsou ekvivalentní: první zavádí omezení, že rovnítko musí dávat smysl i pro prvky seznamu (tedy musí být v typové třídě `Eq`), druhý bude na dlouhých seznamech pomalý (a na nekonečných bude cyklit). Jedině `null` nebo použití vzoru je ta správná volba.
  - ``(n `mod` 2) == 0`` → `even n`, obdobně pro `odd n`
  - `all (== True)` → `and`, obdobně pro `any (== True)` → `or`.

Duplikace kódu

Jeden z nejzásadnějších prohřešků – jakmile kopírujete kód, nebo píšete velmi podobný kód znovu, je to chyba. Místo kopírování či opakovaného programování podobných funkcionalit byste se měli vždy zamyslet nad tím, jak vše potřebné vypočítat pomocí jedné funkce, které můžete přidat vhodné parametry navíc (občas může být výhodné, aby tím parametrem byla funkce). Tuto funkci je také třeba vhodně pojmenovat. Pozor, jen slepit dve funkce do jedné, kde půlka vzorů řeší jednu funkčnost a druhá, velmi podobná, řeší druhou není odstranění duplicity. Obdobně kód může být duplicitní, i když nevypadá stejně – pokud dvě funkce dělají velmi blízké věci (či jednu lze nahradit druhou), jde o duplicitu i pokud pracují vnitřně různě. Hlavním problémem duplicitního kódu je, že případné opravy v duplikované části musíte dělat na více místech a že čtenář musí dekódovat více kódu.

Tento bod je součástí širšího a složitějšímu problému, kterým je vhodná dekompozice problému na podproblémy – to je velmi rozsáhlá záležitost, kterou se naučíte postupně především praxí a zpětnou vazbou na váš kód.

Pojmenování

Vhodné pojmenování funkcí a proměnných dramaticky přispívá k čitelnosti kódu. Pojmenování je závislé na kontextu, ve kterém se jméno používá – pro globální entity (funkce, proměnné, …) musíme volit velmi jasná projmenování, zatímco u lokálních funkcí si můžeme dovolit využít toho, že už jsou v kontextu vnější funkce a volit kratší název.

U argumentů je vhodné zamyslet se nad tím, zda mají nějaký inherentní význam („student“, „číslo účtu“…) – pak je třeba volit vypovídající názvy, či zda se jedná o nespecifické entity (typicky u argumentů polymorfních funkcí jako je `map`) – pak stačí jednopísmenná zkratky, např. `x` pro hodnotu, `xs` pro seznam či `f` pro funkci.

Nadbytečná výřečnost

Programy v Haskellu je často možné napsat velice úsporně a zároveň čitelně. Je pochopitelné, že než se s funkcionálním programováním spřátelíte, napíšete sem tam něco delším způsobem, než by bylo možné. Najít rovnováhu mezi přílišnou stručností a extrémní explicitností není vůbec jednoduchý úkol, ale některých konkrétních chyb si můžete poměrně snadno všimnout:

- **Závorky navíc** – obzvláště snaha o volání funkcí jako v Pythonu (`(last(ys)) == (head(xs))`) vede k nečitelnému kódu. U funkcí s více parametry to navíc volání funkcí v Pythonu přestane připomínat: `max(a)(0)`. Přípustné varianty jsou `last ys == head xs`, `max a 0` nebo ``a `max` 0``.
  - Neznamená to však, že se musíte zbavovat úplně všech syntakticky nadbytečných závorek. Ve výrazech s velkým množstvím infixových operátorů naopak psaní i některých závorek neměnících pořadí vyhodnocení přináší zlepšení čitelnosti – v začátcích je naprosto v pořádku si nepamatovat, zda má vyšší prioritu `&&`, nebo `==`.
  - Závorek může být příliš mnoho, i když žádné nejsou, syntakticky řečeno, navíc. Příliš velká úroveň zanoření se zkrátka špatně čte, takže pokud máte vnořeny třeba čtvery závorky, možná by bylo vhodnější nějakou část vytáhnout do lokální definice nebo pomocné funkce.
- **Redundantní vzory** – není potřeba navazovat seznam na `(x:xs)`, pokud ho vzápětí celý předhodíme funkci `map`. Navíc je pak potřeba vzor pro prázdný seznam.
- **Redundantní ošetřování krajních případů** – typicky zbytečné „zdvojení“ dna rekurze (pro mocnění stačí případy `0` a `n`, je zbytečné ošetřovat `1` zvlášť) nebo vzor navíc pro prázdný seznam, s nímž by si `filter` poradil stejně dobře.
- **Nepoužití vzorů** – pokud ve funkci bereme `x` a následně všude píšeme `fst x` a `snd x`, nejspíš by bylo vhodnější použít rovnou vzor `(x, y)`.
  - Pokud ale zároveň potřebujeme pracovat s celou dvojicí zároveň, není to tak jasné. Můžeme si vypomoci zvláštní syntaxí pro tyto účely: `t@(x, y)` naváže na `x` a `y` prvky dvojice a na `t` celou dvojici. Jen pozor, že na místě `t` může být jen proměnná, ne komplikovanější vzor.
- **Lambda-funkce místo (jednoduché) částečné aplikace/složení** – používání lambda-funkcí je hezké, ale v opravdu jednoduchých případech to bez nich může vypadat lépe. Například `(\x -> x ^ 2 - 3)` je pro většinu lidí čitelnější než `(subtract 3) . (^2)`, ale použít `(\x y -> x + y)` místo `(+)` smysl nemá.
- **Opakování podvýrazů** – pro zkrácení a zpřehlednění slouží lokální definice (`where`, `let … in`) nebo, pokud se výraz opakuje ve více funkcích, pomocné funkce.
- **Zbytečné komentáře** – krátké a jasné funkce, jako třeba `getPid (i, _, _) = i`, nepotřebují vůbec žádný komentář. Rozhodně není potřeba doslova popisovat funkci („Vrací první prvek trojice `ProcessInfo`“). U zadaných funkcí taky není žádoucí parafrázovat zadání. Naopak se může hodit popsat argumenty a výsledek (není-li to zřejmé třeba z jejich názvu) nebo hlavní myšlenku složitější funkce.

A pak jsou tu drobnosti, které nutně nevadí, ale vždycky potěší, když v kódu nejsou:

- **Pojmenování argumentu, ačkoli není použit** – obvykle se na místě nepoužívaných argumentů píše podtržítko.
- **Nepoužití η-redukce na *vhodných* místech**. Například pokud funkce jen třídí vstupní seznam funkcí `filter`. Naopak převádět všechno násilně do point-free tvaru, jen aby bylo možné η-redukovat, čitelnosti velmi škodí.

> K odhalení některých chyb se dají použít varování překladače/interpretu. Buď při spouštění můžete použít přepínač `-Wall`, nebo v interpretu příkaz `:set -Wall`. Místo `all` se dá použít konkrétní varování. Doporučujeme třeba:
>
> `unused-matches` (nepoužité nepodtržítkové formální argumenty)
>
> `unused-binds` (nepoužité funkce a lokální definice)
>
> `incomplete-patterns` (nedostatečné pokrytí vzory)
>
> `overlapping-patterns` (řádek definice se nikdy nepoužije; zapnutý implicitně)

Bílá místa a formátování

Čím byste naopak šetřit neměli, jsou mezery a zalomení řádků. Považuje se za samozřejmost psát mezery kolem operátorů, protože se to mnohem snáze čte.

Haskell vám dovoluje mezerami okrášlit kde co. Třeba v kostrách úloh si můžete povšimnout, že testovací data jsou úhledně zapsána na několik řádků. Vašemu oku může (a nemusí) lahodit třeba i zarovnávání vzorů pod sebe:

```
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

(Můžete si všimnout, že ve vzorech kolem dvojtečky mezery nepíšeme, ale v definici ano. Přísně vzato je to nekonzistentní, ale je to obecně přijímaná konvence, pokud jsou proměnné ve vzorech krátké.)

Dlouhé řádky rozhodně rozdělujte na více kratších; pokud možno tak, aby to dávalo smysl a například oddělovalo argumenty funkce. Jakýkoli řádek delší než 120 znaků si téměř jistě zaslouží zalomit, ale například podmíněné výrazy se leckomu lépe čtou zalomené bez ohledu na délku větví. Nezapomínejte taky na rozumné odsazování. Například

```
sgn x = if x < 0
        then -1
        else if x > 0
             then 1
             else 0
```

Potřebujete-li popsat argumenty funkce, využijte toho, že i typová signatura může zabírat víc řádků (primárně je však vhodné volit hezké názvy argumentů):

```
formatDouble :: Bool      -- always print sign
             -> Bool      -- scientific mode
             -> Int       -- max. decimal digits
             -> Double
             -> String
```

Zkrátka a dobře – nebojte se využívat mezer a řádkových zlomů k tomu, aby byl váš kód pěkný nejen myšlenkou, ale i čistě vizuálně.

> Odsazujte vždy jedině mezerami. Interpret vás na použití tabulátorů i sám upozorní. Je tomu tak proto, že místy je Haskell velmi vybíravý, co se velikosti odsazení týče. Například všechny lokální definice v jednom bloku `let` nebo `where` musí začínat ve stejném sloupci, a „sloupec“ je s tabulátory ošemetný pojem.

Neefektivní kód

Při psaní kódu je záhodno se zamýšlet i nad efektivitou jeho vyhodnocování. Při práci se seznamy nezapomeňte, že se nejedná o pole jako v jazycích C nebo Python, ale o jednosměrně zřetězený seznam, takže téměř všechny operace mají lineární složitost (vzhledem k délce vstupu) – vždy (nebo v nejhorším případě) se musí projít celý seznam. Výjimkami jsou v podstatě pouze operace přidávání a odebírání ze začátku seznamu, které mají konstantní složitost. Pokud například v rekurzívní funkci zjišťujeme v každém kroku délku vstupního seznamu, dostaneme funkci s nejméně kvadratickou složitostí.[4]

Jako příklad si představme funkci `piz`, která se chová jako `zip`, ovšem zarovnává seznam od konce: z delšího seznamu se zahodí prvky ze začátku, ne z konce. Tedy

```
piz [1,2] [3,4,5] ⇝* [(1,4), (2,5)]
```

Mohlo by nás napadnout něco takového (bázové případy vynechány):

```
piz xs ys = piz (init xs) (init ys) ++ [(last xs, last ys)]
```

Takové řešení je vcelku pěkně čitelné a je z něj jasné, co funkce dělá, nicméně má kvadratickou složitost[5]. To má hned několik příčin: `init` má lineární složitost a vlastně kopíruje celý seznam. `last` má lineární složitost. `(++)` má složitost lineární vzhledem k délce prvního seznamu a opět jej kopíruje. Dá se přitom dosáhnout lineární složitosti. Možným řešením je převést pomocí funkce `reverse` problém tak, abychom mohli použít funkci `zip`:

```
piz xs ys = reverse $ zip (reverse xs) (reverse ys)
```

Funkce `reverse` má také lineární složitost, ale stačí ji použít třikrát. To je u delších seznamů řádově lepší, nežli používat lineární funkci pro každý jeden prvek seznamu.

Ještě mnohem horší situace nastává, pokud **v rekurzi opakujeme rekurzívní aplikaci vícekrát**. Pokud voláme funkci rekurzivně dvakrát na stejný podproblém, pak i obě rekurzívní volání volají na své podproblémy tutéž funkci dvakrát a tak dále, z čehož vznikne složitost exponenciální.

Krásně je to vidět na následující funkci na deduplikaci prvků v seznamu:

```
unique :: Eq a => [a] -> [a]
unique [] = []
unique (x:xs) = if x `elem` unique xs then unique xs else x : unique xs
```

Vidíme, že podvýraz `unique xs` se opakuje. Překladač to ovšem nevidí a vyhodnocení rekurzívní aplikace proběhne dvakrát. Nápravu zjednáme lokální definicí:

```
unique (x:xs) = if x `elem` uxs then uxs else x : uxs
    where uxs = unique xs
```

[4] Seznam délky n se projde n-krát, provede se tedy n^2 operací.

[5] Možná si říkáte, že seznamy se přece zmenšují; a skutečně to není přesně n^2, ale něco kolem $\frac{n^2}{2}$. To je však stále kvadratická funkce délky vstupních seznamů.

To vyřeší i problém s opakovaným výpočtem – díky líné strategii se `uxs` (a tedy ani rekurzívní aplikace) zaručeně nebude vyhodnocovat vícekrát.

Podobně se zbytečné výpočty mohou opakovat i bez rekurze:

```
piz xs ys = zip (fst (sameSize xs ys)) (snd (sameSize xs ys))
```

Zde se opět stačí pomocí lokální definice zbavit opakovaného podvýrazu:

```
piz xs ys = zip xs' ys'
    where (xs', ys') = sameSize xs ys
```

To je nejen efektivnější, ale také se to lépe čte. Na rozdíl od příkladu s deduplikací se však složitost dramaticky nezměnila – obě implementace `piz` jsou lineární[6]. Přesto má smysl se opakovaným výpočtům vyhnout.

[6] za předpokladu lineárního `sameSize`

Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe`

Před čtvrtým cvičením je zapotřebí:

- znát koncept datových typů:
  - *hodnotový* a *typový* konstruktor;
  - klíčová slova `data` a `type` a rozdíl mezi nimi;
  - definice funkcí pomocí vzorů pro vlastní datové typy;
- znát datový typ `Maybe`;
- mít základní znalosti o *stromech* – pojmy *kořen*, *cesta*, *hloubka vrcholu*.

Etudy

Etuda 4.η.1 Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den. Datový typ `Day` je definován takto:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
           deriving (Show, Eq, Ord)
```

Etuda 4.η.2 Mějme následující definici:

```
data Object = Cube Double Double Double   -- a, b, c
            | Cylinder Double Double      -- r, v
```

a) Uveďte příklady hodnot, které mají typ `Object`.
b) Kolik je v definici použito hodnotových konstruktorů a které to jsou?
c) Kolik je v definici použito typových konstruktorů a které to jsou?
d) Definujte funkce `volume` a `surface`, které pro hodnoty uvedeného typu počítají objem, respektive povrch.
   Příklady vyhodnocení korektně definovaných funkcí jsou:
   `volume (Cube 1 2 3)` $\rightsquigarrow^*$ `6.0`
   `surface (Cylinder 1 3)` $\rightsquigarrow^*$ `25.132741228718345`

> V řešení můžete použít konstantu `pi`.

Etuda 4.η.3 Uvažte následující kód:

```
data Contained a = NoValue | Single a | Pair a a
data Compare a = SameContainer
               | SameValue (Contained a)
               | DifferentContainer

cmpContained :: Eq a => (Contained a)
                     -> (Maybe (Contained a))
                     -> (Compare a)
```

```
cmpContained (NoValue) (Just (NoValue)) =
    (SameValue (NoValue))
cmpContained (Single x1) (Just (Single x2)) =
    if (x1 == x2)
        then (SameValue (Single x1))
        else (SameContainer)
cmpContained (Pair x1 y1) (Just (Pair x2 y2)) =
    if (x1 == x2) && (y1 == y2)
        then (SameValue (Pair x1 y1))
        else (SameContainer)
cmpContained _ _ = DifferentContainer
```

Vaším úkolem je odstranit čtrnáct párů závorek tak, aby kód stále šel zkompilovat a zároveň si zachoval funkcionalitu.

U každého odstraňovaného páru si zkuste zdůvodnit, proč na daném místě nemusí být explicitně uvedený, a u každého páru, který necháváte, zdůvodnit, proč na daném místě musí zůstat.

Etuda 4.η.4 Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. Korektnost uvažujte pouze v kontextu standardně definovaných datových typů a funkcí (tj. těch z module `Prelude`), bez dalších hypotetických definic.

a) `Maybe Char`
b) `Maybe 42`
c) `Just "Dance!"`
d) `Just Integer`
e) `Just Nothing`
f) `[Nothing, Just 4]`
g) `\b matters -> if b then Nothing else matters`

Etuda 4.η.5 Definujte funkci `safeDiv :: Integral a => a -> a -> Maybe a`, která celočíselně podělí dvě čísla a ošetří případy dělení nulou, tedy například:

```
safeDiv 6 3 ⇝* Just 2
safeDiv 12 0 ⇝* Nothing
safeDiv (-4) (-4) ⇝* Just 1
```

Etuda 4.η.6 Naprogramujte funkci `valOrDef :: Maybe a -> a -> a`, která vezme `Maybe`-hodnotu a výchozí hodnotu a vrátí buď hodnotu obsaženou v `Maybe` pokud taková existuje, nebo výchozí hodnotu pokud je v `Maybe` hodnota `Nothing`.

Etuda 4.η.7 `4.errors` Podívejte se na video shrnující některé na první pohled nejasné chybové hlášky interpretru jazyka Haskell. Můžete k tomu použít odkaz pod číslem tohoto příkladu; vede do studijních materiálů předmětu v ISu.

> **Pan Fešák doporučuje:** Vlastní datové typy definované v příkladech této kapitoly najdete připravené k použití v souboru `04_data.hs` v příloze sbírky nebo ve studijních materiálech v ISu.

4.1 Vlastní datové typy

Př. 4.1.1 Definujeme následující datové typy:

```
type WeaponsPoints = Integer
type HealthPoints = Integer

data Armor = Leather | Steel -- kožené nebo ocelové brnění
             deriving (Eq, Show)

data Warrior = Warrior HealthPoints Armor WeaponsPoints
               deriving (Eq, Show)
```

a) Uveďte příklady hodnot, které mají typ `Armor` a `Warrior`.

b) Kolik je v definici použito hodnotových konstruktorů a které to jsou?

c) Kolik je v definici použito typových konstruktorů a které to jsou?

d) Definujte funkci `attack :: Warrior -> Warrior -> Warrior`. Prvním parametrem této funkce je útočník, druhým je obránce. Funkce vrací stav obránce po útoku.

Každý bod útoku útočníka způsobí snížení zdraví obránce o jedna (ne ale níž než na nulu). Pokud používá obránce ocelové brnění, dělí se útočná síla zbraně dvěma. Předpokládejte, že hodnoty `HealthPoints` a `WeaponsPoints` jsou nezáporné.

Příklady vyhodnocení korektně definované funkce jsou:

```
warrior1 = Warrior 30 Leather 30
warrior2 = Warrior 20 Steel 25

attack warrior1 warrior2 ⇝* Warrior 5 Steel 25
attack warrior2 warrior1 ⇝* Warrior 5 Leather 30
```

Př. 4.1.2 4.1.jar

Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

- je prázdná (`EmptyJar`);
- je v ní ovocná marmeláda (`Jam`), pamatujeme si typ ovoce, ze kterého byla vyrobena (`String`);
- jsou v ní okurky (`Cucumbers`), o nich si nemusíme nic pamatovat, stejně se hned snědí;
- je v ní kompot (`Compote`), pamatujeme si rok výroby (`Int`).

Vaší úlohou je pak nadefinovat funkci `stale :: Jar -> Bool`, která určí, jestli je obsah dané sklenice již zkažený. Prázdné sklenice, okurky ani marmelády se nekazí (možná je to tím, že se příliš rychle snědí), kompoty se pokazí za 10 let od zavaření (zadefinujte si celočíselnou konstantu `today`, ve které budete mít aktuální rok).

> **Pan Fešák doporučuje:** Pro úplné pochopení principů vlastních datových typů a rozdílů mezi hodnotovými a typovými konstruktory je doporučené projít a rozumět následujícím příkladům.

Př. 4.1.3 Identifikujte nově vytvořené typové a hodnotové konstruktory a určete jejich aritu.

a) `data X = Value Int`

b)
```
data M = A | B | N M
data N = C | D | M N
```

c) `data Ha = Hah Int Float [Hah]`

d) `data FMN = T (Int, Int) (Int -> Int) [Int]`
e) `type Fat = Float -> Float -> Float`
f) `data E = E (E, E)`

Př. 4.1.4 Které deklarace datových typů jsou správné?

a) `data N x = NVal (x -> x)`
b) `type Makro = a -> a`
c) `data M = N (x, x) | N Bool | O M`
d) `type Fun a = a -> (a, Bool) -> c`
e) `type Fun (a, c) (a, b) = (b, c)`
f) `data F = X Int | Y Float | Z X`
g) `data F = intfun Int`
h) `data F = Makro Int -> Int`
i) `type Val = Int | Bool`
j) `data X = X X X`

Př. 4.1.5 Uvažte datový typ `type Frac = (Int, Int)`, kde hodnota `(a, b)` představuje zlomek $\frac{a}{b}$ (můžete předpokládat, že $b \neq 0$). Napište funkci nad datovým typem `Frac`, která

a) zjistí, jestli zadané dva zlomky představují stejné racionální číslo;
b) vrátí `True`, jestli zlomek představuje nezáporné číslo;
c) vypočítá součet dvou zlomků;
d) vypočítá rozdíl dvou zlomků;
e) vypočítá součin dvou zlomků;
f) vypočítá podíl dvou zlomků (ověřte, že druhý zlomek je nenulový);
g) převede zlomek do základního tvaru; *Doporučujeme:* zkuste si najít v dokumentaci něco o funkci `gcd`.

Když budete mít všechno implementované, tak upravte funkce tak, aby byl výsledek v základním tvaru.

Př. 4.1.6 V řetězci kaváren StarBugs prodávají jediný druh šálků kávy. Obyčejní zákazníci platí 13 λ za šálek kávy a každý desátý šálek mají zdarma. Pokud si kávu kupuje zaměstnanec, má navíc 15% slevu ze základní ceny. V případě, že si kávu kupuje student ve zkouškovém období, platí za každý šálek 1 λ. Napište funkci, která spočítá výslednou cenu v závislosti na typu zákazníka. Ale pozor! Slevový systém se často mění, aby zaujal lidi. Proto je potřeba navrhnout funkci dostatečně obecně, aby se nemusela vždy celá přepisovat.

- Napište funkci `commonPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro běžného zákazníka.
- Napište funkci `employeeDiscount :: Float -> Float`, která aplikuje zaměstnaneckou slevu na cenu pro obyčejné zákazníky.
- Napište funkci `studentPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro studenta.
- Definujte datový typ `PricingType`, který bude značit, zdali je nakupující obyčejný zákazník (`Common`), zaměstnanec řetězce (`Employee`) nebo student (`Student`).
- Implementujte funkci

```
computePrice :: PricingType -> Int -> (Int -> Float)
             -> (Float -> Float) -> (Int -> Float) -> Float
```

Ta podle typu zákazníka, počtu šálků a tří funkcí `cp`, `ed` a `sp` (`common pricing`, `employee discount` a `student pricing`) spočítá výslednou cenu za nakoupené šálky.

Při řešení se vám může hodit funkce `fromIntegral`. Více se o ní můžete dočíst v dokumentaci.

Příklady vstupů a odpovídajících výsledků:

```
computePrice Common   28 commonPricing employeeDiscount studentPricing
    ⇝* 338
computePrice Employee 28 commonPricing employeeDiscount studentPricing
    ⇝* 287.30002
computePrice Student  28 commonPricing employeeDiscount studentPricing
    ⇝* 28
```

> Následující příklad je rozšířením předchozí úlohy. Doporučujeme vrátit se k němu, pokud máte na konci cvičení čas.

Př. 4.1.7 Řetězec StarBugs z úlohy 4.1.6 se rozhodl rozšířit svůj systém slev. Ještě neví jak přesně, ale každá cena bude buď závislá na počtu koupených šálků, nebo na běžné ceně pro obyčejné zákazníky za daný počet šálků.

Upravte datový typ `PricingType` tak, aby nabízel možnosti `Common` (běžný zákazník), `Special (Int -> Float)` (speciální druh nacenění závislý na počtu káv) a `Discount (Float -> Float)` (slevový druh nacenění závislý na běžné ceně). Každá instance tohoto typu (krom `Common`) tak v sobě bude nést funkci pro výpočet správné ceny.

Dále je nezbytné změnit i funkci `computePrice`, a to tak, že její typ bude `PricingType -> Int -> (Int -> Float) -> Float`. Akceptuje druh zákazníka, počet šálků a funkci pro výpočet běžné ceny a vrací správnou cenu za příslušný počet šálků.

Jako poslední definujte konstanty `common`, `employee`, `student :: PricingType`, které reprezentují typy zákazníků ze cvičení 4.1.6.

Příklady volání a správných vyhodnocení:

```
computePrice common   28 commonPricing ⇝* 338
computePrice employee 28 commonPricing ⇝* 287.30002
computePrice student  28 commonPricing ⇝* 28
```

4.2 Konstruktor Maybe

Př. 4.2.1 Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. Korektnost uvažujte pouze v kontextu standardně definovaných datových typů a funkcí (tj. těch z module `Prelude`), bez dalších hypotetických definic.

a) `Maybe (Just 2)`
b) `Just Just 2`
c) `Just (Just 2)`
d) `Maybe Nothing`
e) `Nothing 3`
f) `Just [Just 3]`

g) `Just`

Př. 4.2.2

Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. S výrazem `a` pracujte jako s externě definovaným typem.

a) `Maybe a`
b) `Just a`
c) `Just (\x -> x ^ 2)`
d) `Just Just`
e) `Just Just Just`

Př. 4.2.3 Definujte funkci `divlist :: Integral a => [a] -> [a] -> [Maybe a]`, s využitím typového konstruktoru `Maybe`, která celočíselně podělí dva celočíselné seznamy „po složkách“ a ošetří případy dělení nulou. Tedy například

```
divlist [12, 5, 7] [3, 0, 2] ⇝* [Just 4, Nothing, Just 3]
divlist [12, 5, 7] [3, 1, 2, 5] ⇝* [Just 4, Just 5, Just 3]
divlist [42, 42] [0] ⇝* [Nothing]
```

Př. 4.2.4 Napište funkci `addVars :: String -> String -> [(String, Integer)] -> Maybe Integer`, která dostane dva názvy proměnných a seznam přiřazující hodnoty proměnným a sečte hodnoty daných proměnných. Pokud se některá z proměnných v seznamu nenachází, pak vraťte `Nothing` (v opačném případě vraťte vhodně zabalený součet).

Pro vyhledání hodnoty proměnné použijte knihovní funkci `lookup :: Eq a => a -> [(a, b)] -> Maybe b`, která vyhledává podle svého prvního argumentu.

Nápověda: Zkuste nejprve vyhledat obě proměnné v seznamu proměnných a na výsledky těchto vyhledávání aplikovat vhodnou pomocnou funkci.

```
addVars "x" "y" [] ⇝* Nothing
addVars "x" "y" [("x", 42)] ⇝* Nothing
addVars "x" "y" [("a", 0), ("y", 42)] ⇝* Nothing
addVars "a" "a" [("a", 0), ("y", 42)] ⇝* Just 0
addVars "x" "y" [("a", 0), ("x", 12), ("y", 30)] ⇝* Just 42
```

> **Pan Fešák doporučuje:** Pokud si nejste jistí, co funkce `zip` přesně dělá, tak se podívejte do dokumentace.

Př. 4.2.5 Napište funkci `mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]`, která je analogií funkce `zip`. Rozdílem je, že výsledný seznam má délku rovnou delšímu ze vstupních seznamů. Chybějící hodnoty jsou nahrazeny hodnotami `Nothing`.

Př. 4.2.6 4.2.summar

Uvažme, že máme následující definice typů:

```
data Mark = A | B | C | D | E | F | X | S deriving (Eq, Show)
type StudentName = String
type CourseName = String
data StudentResult = StudentResult StudentName CourseName (Maybe Mark)
                     deriving (Eq, Show)
```

Definujte funkci `summarize :: StudentResult -> String`, která pro studentský výsledek vrátí řetězec následujícího tvaru:

- pro záznam se známkou bude tvaru `"NAME has MARK from COURSE"`;
- v opačném případě to bude `"NAME has no result from COURSE"`.

```
map summarize [StudentResult "Adam" "IA014" (Just A),
               StudentResult "Jan" "IV115" Nothing,
               StudentResult "Martin" "IA038" (Just X)]
    ⇝* ["Adam has A from IA014",
         "Jan has no result from IV115",
         "Martin has X from IA038"]
```

Poznámka: Připomínáme existenci funkce `show :: Show a => a -> String`.

4.3 Rekurzivní datové typy

Př. 4.3.1 Uvažme následující rekurzivní datový typ:

```
data Nat = Zero | Succ Nat deriving Show
```

a) Jaké hodnoty má typ `Nat`?
b) Jaký význam má dovětek `deriving Show`?
c) Nadefinujte funkci `natToInt :: Nat -> Int`, která převede výraz typu `Nat` na číslo, které vyjadřuje počet použití hodnotového konstruktoru `Succ` v daném výrazu.
d) Jak byste pomocí datového typu `Nat` zapsali nekonečno?

Př. 4.3.2 4.3.expr Uvažme následující definici typu `Expr`:

```
data Expr = Con Double
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
```

a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.
b) Definujte funkci `eval :: Expr -> Double`, která vrátí hodnotu daného výrazu.
c) Ošetřete korektně dělení nulou pomocí funkce `evalMay :: Expr -> Maybe Double`.

Př. 4.3.3 Rozšiřte definici z předchozího příkladu o nulární hodnotový konstruktor `Var`, který bude zastupovat proměnnou. Funkci `eval` upravte tak, aby jako první argument vzala hodnotu proměnné a vyhodnotila výraz z druhého argumentu pro dané ohodnocení proměnné.

* * *

> **Paní Bílá vysvětluje:** Binární strom (`BinTree a`) je struktura, která v každém svém uzlu `Node` udržuje hodnotu typu `a` a ukazatele na své dva potomky. Hodnotový konstruktor `Empty` reprezentuje prázdný strom. Používáme ho například na označení, že v daném směru uzel nemá potomka – `Node 1 Empty (Node 2 Empty Empty)` – uzel s hodnotou 1 nemá levého potomka, uzel s hodnotou 2 nemá ani jednoho potomka.

V následujících příkladech se využívá datová struktura

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
                 deriving Show
```

> Následují příklady několika stromů: jejich zápis v Haskellu a jejich struktura.
>
> ```
> tree00 = Node 42 (Node 28 Empty Empty) Empty
> ```

```
          42
      28        Empty
  Empty  Empty
```

```
tree01 = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
                (Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty))
```

```
                      4
          2                       6
    1           3           5           7
Empty Empty Empty Empty Empty Empty Empty Empty
```

```
tree02 = Node 9 Empty (Node 11 (Node 10 Empty Empty)
                               (Node 12 Empty Empty))
```

```
         9
Empty          11
          10         12
     Empty Empty Empty Empty
```

Prázdný strom (strom, který neuchovává žádnou hodnotu; podobně jako prázdný seznam) vypadá následovně:

```
emptyTree = Empty
```

```
Empty
```

Př. 4.3.4

a) Nakreslete všechny tříuzlové stromy typu `BinTree ()` a zapište je pomocí hodnotových konstruktorů `Node` a `Empty`.
b) Kolik existuje stromů typu `BinTree ()` s 0, 1, 2, 3, 4 nebo 5 uzly?
c) Kolik existuje stromů typu `BinTree Bool` s 0, 1, 2, 3, 4 nebo 5 uzly?

Pan Fešák doporučuje: V přiloženém souboru `04_data.hs` najdete taky předdefinované stromy, které můžete použít pro testování funkcí pracujících se strukturou `BinTree`.

Př. 4.3.5 4.3.tree

Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu. Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Int`, která spočítá počet uzlů ve stromě.

b) `listTree :: BinTree a -> [a]`, která vrátí seznam hodnot, které jsou uložené v uzlech vstupního stromu (na pořadí nezáleží).

c) `height :: BinTree a -> Int`, která určí výšku stromu.

d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě začínající v kořeni a vrátí ohodnocení na ní.

Pár příkladů vyhodnocení funkcí v této úloze:

```
treeSize Empty ⇝* 0
treeSize tree01 ⇝* 7
listTree tree01 ⇝* [1, 2, 3, 4, 5, 6, 7] -- Jedno z možných řešení.
listTree tree02 ⇝* [9, 10, 11, 12]
height tree02 ⇝* 3
longestPath tree05 ⇝* [100, 101, 102, 103, 104]
```

Př. 4.3.6 Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) Definujte funkci `fullTree :: Int -> a -> BinTree a`, která pro volání `fullTree n v` vytvoří binární strom výšky `n`, ve kterém jsou všechny větve stejně dlouhé a všechny uzly jsou ohodnocené hodnotou `v`.

b) Definujte funkci `treeZip :: BinTree a -> BinTree b -> BinTree (a, b)` jako analogii seznamové funkce `zip`. Výsledný strom tedy obsahuje pouze ty uzly, které jsou v obou vstupních stromech.

Př. 4.3.7 Napište `treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)` jako analogii seznamové funkce `mayZip` z příkladu 4.2.5. Vrchol v novém stromu bude existovat právě tehdy, pokud existuje aspoň v jednom ze vstupních stromů.

Př. 4.3.8 Uvažme datový typ `BinTree a`.

a) Definujte funkci `isTreeBST :: (Ord a) => BinTree a -> Bool`, která se vyhodnotí na `True`, jestli bude její první argument validní binární vyhledávací strom (BST).

b) Definujte funkci `searchBST :: (Ord a) => a -> BinTree a -> Bool`, která projde BST z druhého argumentu v smyslu binárního vyhledávání a vyhodnotí se na `True` v případě, že její první argument najde v uzlech při vyhledávání.

Můžete předpokládat, že vstupní datový typ `a` je uspořádaný lineárně.

4.4 Další příklady

Př. 4.4.1 Uvažte typ stromů s vrcholy libovolné arity definovaný následovně:

```
data RoseTree a = RoseNode a [RoseTree a]
                  deriving (Show, Read)
```

Definujte následující:

a) funkci `roseTreeSize :: RoseTree a -> Int`, která spočítá počet uzlů ve stromě,
b) funkci `roseTreeSum :: Num a => RoseTree a -> a`, která sečte ohodnocení všech uzlů stromu,
c) funkci `roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b`, která bere

funkci a strom a aplikuje danou funkci na hodnotu v každém uzlu:

```
roseTreeMap (+1) (RoseNode 0 [RoseNode 1 [], RoseNode 41 []])
    ⇝* RoseNode 1 [RoseNode 2 [], RoseNode 42 []]
```

Př. 4.4.2 Uvažme následující definici typu `LogicExpr`:

```
data LogicExpr = Pos | Neg
               | And LogicExpr LogicExpr
               | Or LogicExpr LogicExpr
               | Implies LogicExpr LogicExpr
               | Equiv LogicExpr LogicExpr
```

Definujte funkci `evalExpr :: LogicExpr -> Bool`, která vrátí hodnotu, na kterou se daný výraz vyhodnotí.

Př. 4.4.3 Uvažujme rekurzivní datový typ `IntSet` definovaný takto:

```
data IntSet = SetNode Bool IntSet IntSet -- Node isEnd zero one
            | SetLeaf
              deriving Show
```

Ve stromě typu `IntSet` každá cesta z vrcholu jednoznačně určuje binární kód složený z čísel přechodů mezi otcem a synem (podle označení syna `one` respektive `zero`). Toho můžeme využít pro ukládání přirozených čísel do takového stromu. Strom typu `IntSet` obsahuje číslo n právě tehdy, pokud obsahuje cestu odpovídající binárnímu zápisu čísla n, a navíc poslední vrchol této cesty má nastavenou hodnotu `isEnd` na `True`.

Implementujte tyto funkce pro práci se strukturou `IntSet`:

a) `insert :: IntSet -> Int -> IntSet` – obdrží strom typu `IntSet` a přirozené číslo n a navrátí strom obsahující číslo n.
b) `find :: IntSet -> Int -> Bool` – obdrží strom typu `IntSet` a přirozené číslo n a vrátí `True` právě tehdy, pokud strom obsahuje n.
c) `listSet :: IntSet -> [Int]` – obdrží strom typu `IntSet` a navrátí seznam čísel uložených v tomto stromě.

Př. 4.4.4 Podobná stromová struktura jako v příkladu 4.4.3 by mohla být použita i pro udržování množiny řetězců nad libovolnou abecedou (například slova složená z písmen anglické abecedy nebo konečné posloupnosti celých čísel). Definujte datový typ `SeqSet a` sloužící pro uchovávání posloupností prvků typu `a`. Dále definujte obdoby funkcí ze cvičení 4.4.3:

a) `insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a`
b) `findSeq :: Eq a => SeqSet a -> [a] -> Bool`

* * *

Na konci cvičení byste měli zvládnout:

- tvorbu vlastních datových typů;
- využívat datový typ `Maybe`;
- implementovat funkce na rekurzivních datových typech, a to především na strukturách typu strom.

Cvičení 5: Intensionální seznamy, lenost, foldy

Před pátým cvičením je zapotřebí znát:

- zápisy hromadným výčtem jako `[1..5]`, `[1,3..100]`, `[0,-2..10]`;
- co jsou intensionální seznamy a jak se v Haskellu zapisují (kvalifikátory, generátory, predikáty, lokální definice), tj. například umět přečíst zápis `[ 2 * y | x <- [1,2,3,4,5], even x, let y = 2 + x ]`;
- co je to vyhodnocovací strategie;
- jak probíhá striktní a líné vyhodnocování;
- jak fungují akumulační funkce na seznamech, tj. funkce:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl1 :: (a -> a -> a) -> [a] -> a
```

Etudy

Etuda 5.η.1 Pomocí intensionálního seznamu definujte funkci `divisors :: Integer -> [Integer]`, která k zadanému kladnému celému číslu vrátí seznam jeho kladných dělitelů (bez duplikátů).

```
divisors 1  ⇝* [1]
divisors 4  ⇝* [1, 2, 4]
divisors 12 ⇝* [1, 2, 3, 4, 6, 12]
```

Etuda 5.η.2 Uvažme bilance kreditních karet osob reprezentované seznamem dvojic (*jméno vlastníka*, *množství peněz*). Pomocí intensionálního seznamu definujte funkci `negativeCredit :: [(String, Integer)] -> [String]`, která pro takový seznam karet vrátí právě ta jména, na jejichž kartě je záporná částka.

```
negativeCredit [("James", 0), ("Mikael", -28), ("Eric", 18),
                ("Heather", 10), ("Irene", 7), ("Magnus", -4),
                ("Devon", -14), ("Devin", -30)]
    ⇝* ["Mikael", "Magnus", "Devon", "Devin"]
```

Poznámka: V řešení skutečně smysluplně použijte intensionální seznam, nepoužívejte explicitní rekurzi ani funkce jako `map` a `filter`.

Etuda 5.η.3 Pomocí intensionálního seznamu definujte funkci `allDivisibleSums`, která ze zadaných dvou seznamů celých čísel `xs` a `ys` a kladného čísla `d` vytvoří seznam všech možných trojic (*číslo z `xs`*, *číslo z `ys`*, *součet prvních dvou složek*) splňujících, že součet je dělitelný číslem `d`. Pro každou dvojici čísel počítejte součet právě jednou.

```
allDivisibleSums [0, 1, 2] [3, 4, 5] 3 ⇝*
    [(0, 3, 3), (1, 5, 6), (2, 4, 6)]
allDivisibleSums [0, 1] [2, 3] 1 ⇝*
    [(0, 2, 2), (0, 3, 3), (1, 2, 3), (1, 3, 4)]
allDivisibleSums [0, 1] [2, 3] 7 ⇝* []
```

Etuda 5.η.4 Definujme si typový alias `UCO` ekvivalentní typu `Int`:

```
type UCO = Int
```

Reprezentujme seznam studentů a jimi absolvovaných předmětů pomocí typu `[(UCO, [String])]`, kde první složka dvojice reprezentuje učo studenta a druhá složka kódy předmětů, které daný student absolvoval.

Pomocí intensionálních seznamů (bez explicitní rekurze a funkcí jako `map` a `filter`) napište funkce:

a) `countPassed :: [(UCO, [String])] -> [(UCO, Int)]`, která vrátí učo každého studenta spolu s počtem předmětů, které daný student absolvoval,
b) `atLeastTwo :: [(UCO, [String])] -> [UCO]`, která vrátí uča studentů, kteří absolvovali alespoň dva předměty,
c) `passedIB015 :: [(UCO, [String])] -> [UCO]`, která vrátí uča studentů, kteří absolvovali předmět `"IB015"` (může se vám hodit funkce `elem`),
d) `passedBySomeone :: [(UCO, [String])] -> [String]`, která vrátí kódy předmětů, které absolvoval alespoň jeden student (kódy se v seznamu můžou opakovat).

> **Pan Fešák doporučuje:** Nezapomeňte, že v intensionálních seznamech lze využívat vzory.

```
students :: [(UCO, [String])]
students = [(415409, []), (448093, ["IB111", "IB015", "IB000"]),
            (405541, ["IB111", "IB000", "IB005", "MB151", "MB152"])]

countPassed students     ⇝* [(415409, 0), (448093, 3), (405541, 5)]
atLeastTwo students      ⇝* [448093, 405541]
passedIB015 students     ⇝* [448093]
passedBySomeone students ⇝* ["IB111", "IB015", "IB000", "IB111", "IB000",
                              "IB005", "MB151", "MB152"]
```

Etuda 5.η.5 Naprogramujte unární funkci `naturalsFrom :: Integer -> [Integer]`, která pro vstup `n` vrátí nekonečný seznam `[n, n + 1, n + 2,...]`.

Pomocí funkce `naturalsFrom` definujte nekonečný seznam `naturals :: [Integer]` všech přirozených čísel včetně nuly.

> **Pan Fešák doporučuje:** Vyhodnocovat nekonečné seznamy celé není úplně praktické. Může se vám hodit klávesová zkratka Ctrl + C, která zabije aktuální výpočet a hlavně funkce `take :: Int -> [a] -> [a]`, například si můžete vyhodnotit `take 3 (naturalsFrom 100) ⇝* [100, 101, 102]`.

Etuda 5.η.6 S pomocí rekurze naprogramujte funkci `maxmin :: Ord a => [a] -> (a, a)`, která pro zadaný seznam *v jednom průchodu* zjistí jeho maximum a minimum. Předpokládejte, že vstupní seznam je neprázdný.

Nápověda: může se vám hodit pomocná funkce s tzv. akumulátorem – dodatečným argumentem sloužícím pro postupné vytváření výsledku (v tomto případě maxima a minima již zpracovaných hodnot).

```
maxmin [1, 2, 4, 3] ⇝* (4, 1)
maxmin [42] ⇝* (42, 42)
```

Pan Fešák doporučuje: Kostru úloh k této kapitole najdete připravené k použití v souboru `05_data.hs` v příloze sbírky nebo ve studijních materiálech v ISu.

5.1 Intensionální seznamy

Př. 5.1.1 »= Reprezentujme seznam účastníků plesu pomocí typu `[(String, Sex)]`, kde první složka dvojice reprezentuje jméno účastníka a druhá složka pohlaví definované následujícím typem:

```
data Sex = Female | Male
```

Pomocí intensionálních seznamů napište funkci

```
allPairs :: [(String, Sex)] -> [(String, String)]
```

která pro daný seznam účastníků vrátí seznam všech možných dvojic účastníků ve tvaru (*muž*, *žena*). Vyzkoušejte si, jak funkci definovat bez jakýchkoli instancí pro datový typ `Sex`. Potom si vyzkoušejte, jaké další možnosti byste měli, pokud bychom přidali instanci `Eq Sex`.

Jak ovlivňuje pořadí generátorů a kvalifikátorů ve vaší definici výsledný seznam a počet kroků potřebných k jeho vygenerování?

```
allPairs [("Jeff", Male), ("Britta", Female),
          ("Annie", Female), ("Troy", Male)] ⇝*
    [("Jeff", "Britta"), ("Jeff", "Annie"),
     ("Troy", "Britta"), ("Troy", "Annie")]
```

Př. 5.1.2 Intensionálním způsobem zapište následující seznamy nebo funkce:

a) `[1, 4, 9, ..., k ^ 2]` (pro pevně dané externě definované `k`)
b) funkci `f`, která ze seznamu seznamů vybere jenom ty delší než 3 prvky
c) `"*****"`
d) `["", "*", "**", "***", ...]`
e) seznam seznamů `[[1], [1, 2], [1, 2, 3], ...]`

Př. 5.1.3 5-seznamy Intensionálním způsobem zapište výrazy, které se chovají stejně jako následující (předpokládejte externě definované funkce/hodnoty `f`, `p`, `s`, `x`):

a) `map f s`
b) `filter p s`
c) `map f (filter p s)`
d) `repeat x`
e) `replicate n x`
f) `filter p (map f s)`

Př. 5.1.4 Napište funkci, která ze seznamu prvků vygeneruje všechny

a) permutace,
b) variace s opakováním,
c) kombinace.

Prvky ve výsledném seznamu můžou být v libovolném pořadí. Můžete předpokládat, že prvky vstupního seznamu jsou různé. Také se můžete v případě potřeby omezit na seznamy s porovnatelnými prvky (tj. typu `Eq a => a`).

Př. 5.1.5 Která z níže uvedených funkcí je časově efektivnější? Proč? Jak se uvedené funkce chovají pro nekonečné seznamy?

- ```
  f1 :: [a] -> [a]
  f1 s = [ s !! n | n <- [0, 2 .. length s] ]
  ```
- ```
  f2 :: [a] -> [a]
  f2 (x : _ : s) = x : f2 s
  f2 _ = []
  ```

5.2 Lenost, nekonečné datové struktury

Př. 5.2.1 Uvažte nekonečný seznam přirozených čísel `naturals`, který jste definovali v úvodním příkladu 5.η.5. Mějme dále standardní funkci `(!!) :: [a] -> Int -> a` definovanou následovně:

```
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n - 1)
```

Jakou hodnotu má výraz `naturals !! 2`? Ukažte celý výpočet, který k této hodnotě vede. Kde se v tomto výpočtu projeví líná vyhodnocovací strategie?

Vysvětlete, jak by výpočet výrazu `naturals !! 2` probíhal, kdyby Haskell používal *striktní* vyhodnocovací strategii.

Př. 5.2.2 Uvažte opět seznam `naturals` z příkladu 5.η.5. Mějme dále funkci `filter' :: (a -> Bool) -> [a] -> [a]` definovanou následovně:

```
filter' _ [] = []
filter' p (x : xs) = if p x then x : filter' p xs else filter' p xs
```

Jak se bude chovat interpret jazyka Haskell pro vstup `filter' (< 3) naturals`? Ověřte svou hypotézu v interpretu a jeho chování vysvětlete.

Dále uvažte funkci `takeWhile' :: (a -> Bool) -> [a] -> [a]` definovanou následovně:

```
takeWhile' _ [] = []
takeWhile' p (x : xs) = if p x then x : takeWhile' p xs else []
```

Jak se bude chovat interpret jazyka Haskell pro vstup `takeWhile' (< 3) naturals`? Ověřte svou hypotézu v interpretu a jeho chování vysvětlete.

Poznámka: výše uvedené funkce se chovají stejně jako standardní `filter` a `takeWhile`.

Př. 5.2.3 Jaký je význam líného vyhodnocování v následujících výrazech:

a) `let f = f in fst (2, f)`
b) `let f [] = 3 in const True (f [1])`
c) `0 * div 2 0`
d) `snd ("a" * 10, id)`

Př. 5.2.4 Zjistěte, jak se chovají funkce `zip` a `zipWith`, pokud jeden z jejich argumentů je nekonečný seznam.

Uvažte seznam studentů Fakulty informatiky reprezentovaný pomocí seznamu jmen studentů `[String]` tak, že studenti jsou v něm seřazeni podle počtu bodů, které získali z předmětu IB015. Napište funkci `addNumbers :: [String] -> [String]`, která ke každému studentovi přidá jeho pořadí ve vstupním seznamu. Například:

```
addNumbers ["Pablo", "Steve", "Javier", "Gustavo"] ⇝*
```

```
["1. Pablo", "2. Steve", "3. Javier", "4. Gustavo"]
```

Zkuste funkci `addNumbers` naprogramovat tak, aby vstupní seznam prošla právě jednou (tedy zejména nepoužívejte funkci `length`).

> **Pan Fešák doporučuje:** Vzpomeňte si na funkci `show :: Show a => a -> String`.

Př. 5.2.5 S pomocí intensionálního seznamu definujte nekonečný seznam `integers :: [Integer]`, který obsahuje právě všechna celá čísla. Seznam `integers` musí splňovat, že každé celé číslo v něm jde vygenerovat po konečném počtu kroků. Jinými slovy, pro každé celé číslo `z` musí existovat `i` takové, že `(integers !! i) == z`.

Př. 5.2.6 Definujte nekonečný seznam `threeSum :: [(Integer, Integer, Integer)]`, který obsahuje právě ty trojice kladných čísel `(x, y, z)`, pro které platí `x + y == z`. Seznam `threeSum` musí splňovat, že každá taková trojice v něm jde vygenerovat po konečném počtu kroků. Jinými slovy, pro každou trojici `(x, y, z)`, kde `x + y == z`, musí existovat `i` takové, že `(threeSum !! i)` $\rightsquigarrow^*$ `(x, y, z)`.

Př. 5.2.7 Uvažte libovolný výraz a počet kroků vyhodnocení tohoto výrazu při použití líné vyhodnocovací strategie a počet kroků při použití striktní vyhodnocovací strategie. Jaký je obecně vztah ($=$, $<$, $\leq$, $>$, $\geq$, ani jedno) mezi těmito počty kroků? Jaký je obecně vztah mezi počty kroků normální vyhodnocovací strategie a striktní vyhodnocovací strategie?

Př. 5.2.8 Jaké jsou výhody líné vyhodnocovací strategie? Jaké jsou výhody striktní vyhodnocovací strategie?

Př. 5.2.9 Pomocí některé z funkcí `iterate`, `repeat`, `replicate`, `cycle` vyjádřete nekonečné seznamy:

a) Seznam nekonečně mnoha hodnot `True`.
b) Rostoucí seznam všech mocnin čísla 2.
c) Rostoucí seznam všech mocnin čísla 3 na sudý exponent.
d) Rostoucí seznam všech mocnin čísla 3 na lichý exponent.
e) Alternující seznam `-1` a `1`: `[1, -1, 1, -1, ...]`.
f) Seznam řetězců `["", "*", "**", "***", "****", ...]`.
g) Seznam zbytků po dělení `4` pro seznam `[1 ..]`: `[1, 2, 3, 0, 1, 2, 3, 0, ...]`.

Př. 5.2.10 Naprogramujte funkci `differences :: [Integer] -> [Integer]`, která pro nekonečný seznam `[x1, x2, x3,...]` vypočítá seznam rozdílů po sobě jdoucích dvojic prvků, tedy seznam `[(x2 - x1), (x3 - x2), (x4 - x3), ...]`.

Zkuste funkci `differences` naprogramovat bez explicitního použití rekurze, pomocí funkcí `zipWith` a `tail`.

Př. 5.2.11 Naprogramujte funkci `values :: (Integer -> a) -> [a]`, která pro zadanou funkci `f :: Integer -> a` vypočítá nekonečný seznam jejích hodnot `[f 0, f 1, f 2, ...]`. Uvažte funkci `differences` z předchozí úlohy.

a) Čemu odpovídá seznam `differences (values f)`?
b) Čemu odpovídá seznam `differences (differences (values f))`?

Pomocí funkcí `values`, `differences`, `zip3` a dalších vhodných funkcí na seznamech napište funkci `localMinima :: (Integer -> Integer) -> [Integer]`, která pro zadanou funkci `f :: Integer -> Integer` vypočítá seznam hodnot, ve kterých funkce `f` nabývá na kladných vstupech lokálního minima (tedy hodnot `f n` takových, že `n > 0`, `f (n - 1) > f n`

a zároveň `f (n + 1) > f n`).

Př. 5.2.12 Definujte Fibonacciho posloupnost, tj. seznam kladných celých čísel `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...]`. Můžete ji definovat jako seznam hodnot (typ `[Integer]`) nebo jako funkci, která vrátí konkrétní Fibonacciho číslo (`Integer -> Integer`). Jaká je ve Vaší implementaci složitost výpočtu n-tého čísla Fibonacciho posloupnosti?

Př. 5.2.13 Pomocí rekurzivní definice a funkce `zipWith` vyjádřete Fibonacciho posloupnost tak, že výpočet každého dalšího prvku proběhl v konstantním čase (tj. počet výpočetních kroků nutný k získání dalšího čísla posloupnosti není nijak závislý na tom, kolikáté číslo to je).

Př. 5.2.14 Protože možnost definovat nekonečné seznamy není žádná magie, ale vyplývá z vlastností vyhodnocovací strategie jazyka Haskell, není překvapivé, že lze definovat nekonečné hodnoty i pro jiné vlastní datové typy. Vzpomeňte si na definici datového typu binárních stromů:

```
data BinTree a = Node a (BinTree a) (BinTree a) | Empty
                 deriving (Show, Eq)
```

Na rozdíl od nekonečných seznamů bohužel není zaručené, že při výpisu nekonečného stromu v interpretu uvidíte dříve nebo později každý jeho prvek. Výpis totiž probíhá *do hloubky*, a tudíž se nejprve vypisuje celá nejlevější větev stromu. Ta však nemusí být konečná, takže se výpis nemusí dostat k ostatním větvím stromu.

Pro další práci s nekonečnými binárními stromy se tedy bude hodit nejprve definovat funkci, která pro zadaný potenciálně nekonečný strom vrátí jeho konečnou část, kterou lze vypsat celou. Definujte tedy funkci `treeTrim :: BinTree a -> Integer -> BinTree a` takovou, že pro zadaný strom `t` a hloubku `h` bude výsledkem `treeTrim t h` konečný strom, který obsahuje ty uzly stromu `t`, které jsou nejvýše v hloubce `h`. Vzpomeňte si, že kořen stromu má hloubku 0. Poté definujte:

a) funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má v každém uzlu zadanou hodnotu. Tedy například

```
treeTrim (treeRepeat 42) 1 ⇝*
    Node 42 (Node 42 Empty Empty) (Node 42 Empty Empty)
```

b) funkci `treeIterate :: (a -> a) -> (a -> a) -> a -> BinTree a` jako analogii seznamové funkce `iterate`. Levý potomek každého uzlu bude mít hodnotu vzniklou aplikací první zadané funkce a pravý aplikací druhé zadané funkce. Tedy například

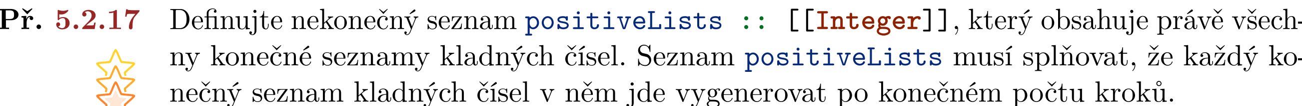
```
treeTrim (treeIterate (+1) (*4) 2) 1 ⇝*
    Node 2 (Node 3 Empty Empty) (Node 8 Empty Empty)
```

c) pomocí funkce `treeIterate` vyjádřete nekonečný binární strom `depthTree` typu `BinTree Integer`, jehož každý uzel v sobě obsahuje svou hloubku. Tedy například

```
treeTrim depthTree 1 ⇝*
    Node 0 (Node 1 Empty Empty) (Node 1 Empty Empty)
```

Př. 5.2.15 Definujte nějaký binární strom, který obsahuje alespoň jednu nekonečnou větev a alespoň jednu konečnou větev.

Př. 5.2.16 Definujte nekonečný seznam `nonNegativePairs :: [(Integer, Integer)]`, který obsahuje právě všechny dvojice kladných čísel `(x, y)`. Seznam `nonNegativePairs` musí splňovat, že každá dvojice kladných čísel v něm jde vygenerovat po konečném počtu kroků.



Př. 5.2.17 Definujte nekonečný seznam `positiveLists :: [[Integer]]`, který obsahuje právě všechny konečné seznamy kladných čísel. Seznam `positiveLists` musí splňovat, že každý konečný seznam kladných čísel v něm jde vygenerovat po konečném počtu kroků.

5.3 Akumulační funkce na seznamech

Jindřiška upozorňuje: S akumulačními funkcemi na seznamech se můžeme setkat i ve většině imperativních programovacích jazyků. Například v Pythonu je to funkce `functools.reduce`, v C++ `std::accumulate` a v C# metoda `Aggregate`. V drtivé většině případů tyto funkce zpracovávají data ve stejném pořadí jako funkce `foldl`.

Pan Fešák doplňuje: Pokud si necháme v interpretru otypovat například `foldr`, dozvíme se obecnější typ, než nám relativně čitelný `foldr :: (a -> b -> b) -> b -> [a] -> b`. Je to dáno tím, že foldy jde v Haskellu používat i na jiné datové struktury, než je seznam. Většina těchto datových struktur (nebo jejich fungování s foldy) je však mimo rozsah tohoto kurzu, nám tedy bude stačit si mentálně v typech nahradit `Foldable` za seznamy, například u `foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b` vidíme, že `t` je `Foldable` a tedy `t a` můžeme pro naše účely zjednodušit na `[a]`.

Připomeňme si, jak můžeme akumulační funkce definovat (definice v knihovně se na seznamech chovají stejně, a to včetně chování k lenosti, mají ale obecnější typ a mohou být trochu optimalizované):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ base []     = base
foldr f base (x:xs) = f x (foldr f base xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ []     = error "foldr1: empty list"
foldr1 _ [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 _ []     = error "foldl1: empty list"
foldl1 f (x:xs) = foldl f x xs
```

Př. 5.3.1 Definujte následující funkce rekurzivně:

a) `product'` – součin prvků seznamu
b) `length'` – počet prvků seznamu

c) `map'` – funkci `map`

Co mají tyto definice společného? Jak by vypadalo jejich zobecnění (tj. funkce, pomocí které se použitím vhodných argumentů dají všechny tyto tři funkce implementovat)?

Př. 5.3.2 5-fold »= Pomocí vhodné akumulační funkce `foldr`, `foldl`, `foldr1` nebo `foldl1` implementujte následující funkce. Jinými slovy, definice musí být jediný řádek tvaru `funkce argumenty = fold* ...`, tedy speciálně nesmíte samostatně ošetřovat prázdný seznam.

> **Jindřiška varuje:** Varianty `foldr1` a `foldl1` jsou použitelné jen pokud nepotřebujete, aby funkce fungovala na prázdném seznamu. Tento případ však chceme pokrýt kdykoli to rozumně lze.

a) Funkci `sumFold`, která vrátí součet čísel v zadaném seznamu.

```
sumFold :: Num a => [a] -> a
sumFold [1, 2, 4, 5, 7, 6, 2] ⇝* 27
```

b) Funkci `productFold`, která vrátí součin čísel v zadaném seznamu.

```
productFold :: Num a => [a] -> a
productFold [1, 2, 4, 0, 7, 6, 2] ⇝* 0
```

c) Funkci `orFold`, která vrátí `True`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `True`, jinak vrátí `False`.

```
orFold :: [Bool] -> Bool
orFold [False, True, False] ⇝* True
```

d) Funkci `lengthFold`, která vrátí délku zadaného seznamu.

```
lengthFold :: [a] -> Int
lengthFold ["Holographic Rick", "Shrimp Rick", "Wasp Rick"] ⇝* 3
```

e) Funkci `maximumFold`, která vrátí maximální prvek ze zadaného neprázdného seznamu.

```
maximumFold :: Ord a => [a] -> a
maximumFold "patrick star" ⇝* 't'
```

Př. 5.3.3 Všechny funkce z předchozího příkladu již jsou ve standardní knihovně jazyka Haskell implementované. Pomocí dokumentace zjistěte, jak se ve standardní knihovně jmenují.

Př. 5.3.4 »= Bez použití interpretru určete, jak se vyhodnotí akumulační funkce s následujícími hodnotami a zda vyhodnocování skončí. Následně si chování určete v interpretru, dávejte ale pozor, abyste případné nekončící výpočty včas zabili (ctrl + c) aby vám nedošla paměť.

a) `foldr (-) 0 [1, 2, 3]`
b) `foldl (-) 0 [1, 2, 3]`
c) `foldr (&&) True (True : False : repeat True)`
d) `foldl (&&) True (True : False : repeat True)`

> **Pan Fešák připomíná:** Funkce `foldr` je takzvaný *katamorfismus* na seznamech.

Př. 5.3.5 Definujte funkci `subtractlist`, která odečte druhý a všechny další prvky *neprázdného* seznamu od jeho prvního prvku, tj. `subtractlist [x1, x2, ..., xn]` se vyhodnotí na `x1 - x2 - ... - xn`.

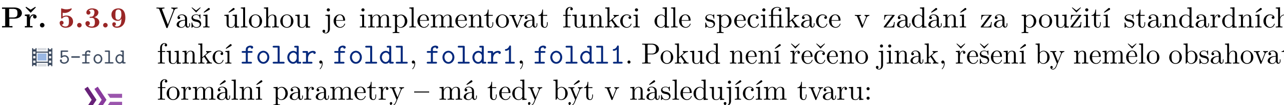
Př. 5.3.6 Uvažme funkci: `foldr (.) id`

a) Jaký je význam této funkce?
b) Jaký je její typ?
c) Uveďte příklad částečné aplikace této funkce na jeden argument.
d) Uveďte příklad úplné aplikace této funkce na kompletní seznam argumentů.

Př. 5.3.7 S použitím vhodné akumulační funkce definujte funkci `append' :: [a] -> [a] -> [a]`, která vypočítá zřetězení vstupních seznamů. Tedy funkce `append'` se bude chovat stejně jako knihovní funkce `(++)`.

Zkuste všechny výrazy použité při definici funkce `append'` co nejvíce η-redukovat.

Př. 5.3.8 S použitím vhodné akumulační funkce definujte funkci `reverse' :: [a] -> [a]`, která s *lineární* časovou složitostí otočí vstupní seznam (dejte si pozor na to, že operátor ++ má lineární časovou složitost vzhledem k délce prvního argumentu).



Př. 5.3.9 5-fold

Vaší úlohou je implementovat funkci dle specifikace v zadání za použití standardních funkcí `foldr`, `foldl`, `foldr1`, `foldl1`. Pokud není řečeno jinak, řešení by nemělo obsahovat formální parametry – má tedy být v následujícím tvaru:

```
functionName = foldr (function) (term)
```

Jestliže je možné příklad řešit více než jednou z nabízených akumulačních funkcí, vyberte tu, která je nejefektivnější. K většině zadání je dostupný i ukázkový výsledek na jednom seznamu sloužící jako ilustrace.

> **Jindřiška varuje:** Každé řešení zapisujte i s typem funkce, v některých příkladech by bez něj nemuselo jít zkompilovat (důvod je poněkud složitější).

a) Funkce `concatFold` vrátí zřetězení prvků zadaného seznamu seznamů.

```
concatFold :: [[a]] -> [a]
concatFold ["pineapple", "apple", "pen"] ⇝* "pineappleapplepen"
```

b) Funkce `listifyFold` nahradí každý prvek jednoprvkovým seznamem s původním prvkem.

```
listifyFold :: [a] -> [[a]]
listifyFold [1, 3, 4, 5] ⇝* [[1], [3], [4], [5]]
```

c) Funkce `nullFold` vrátí `True`, pokud je zadaný seznam prázdný, jinak vrátí `False`.

```
nullFold :: [a] -> Bool
nullFold ["Aang", "Appa", "Momo", "Zuko"] ⇝* False
```

d) Funkce `composeFold` vezme seznam funkcí a hodnotu, a vrátí hodnotu, která vznikne postupným aplikováním funkcí v seznamu na danou hodnotu (poslední funkce se aplikuje jako první, první jako poslední).

```
composeFold :: [a -> a] -> a -> a
composeFold [(* 4),(+ 2)] 3 ⇝* 20
```

e) Funkce `idFold` vrátí zadaný seznam beze změny.

```
idFold :: [a] -> [a]
idFold ["Lorelai", "Rory", "Luke"] ⇝* ["Lorelai", "Rory", "Luke"]
```

f) Funkce `mapFold` vezme funkci a seznam, a vrátí seznam, který vznikne aplikací zadané funkce na každý prvek zadaného seznamu. Pro funkci použijte formální argument.

```
mapFold :: (a -> b) -> [a] -> [b]
mapFold (+ 5) [1, 2, 3] ⇝* [6, 7, 8]
```

g) Funkce `headFold` vrátí první prvek zadaného neprázdného seznamu.

```
headFold :: [a] -> a
headFold ["Light", "L", "Ryuk", "Misa"] ⇝* "Light"
```

h) Funkce `lastFold` vrátí poslední prvek zadaného neprázdného seznamu.

```
lastFold :: [a] -> a
lastFold ["Edward", "Alphonse", "Winry", "Mustang"] ⇝* "Mustang"
```

i) Funkce `maxminFold` vrátí maximální a minimální prvek ze zadaného neprázdného seznamu ve formě uspořádané dvojice. V definici použijte i formální argument funkce `maxminFold`, bude se Vám hodit. Funkce by měla projít zadaný seznam pouze jednou! Vzpomeňte též na příklad 5.η.6, může vám dát nápovědu, jak problém vyřešit.

```
maxminFold :: Ord a => [a] -> (a, a)
maxminFold ["Lana", "Sterling", "Cyril"] ⇝* ("Sterling", "Cyril")
```

j) Funkce `suffixFold` vrátí seznam všech přípon zadaného seznamu (jako první bude samotný seznam, poslední bude prázdný seznam).

```
suffixFold :: [a] -> [[a]]
suffixFold "abcd" ⇝* ["abcd", "bcd", "cd", "d", ""]
```

k) Funkce `filterFold` vezme predikát a seznam a vrátí seznam, který vznikne ze zadaného seznamu vyloučením všech prvků, na kterých predikát vrátí `False`. Pro predikát použijte formální argument.

```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold odd [1, 2, 4, 8, 6, 2, 5, 1, 3] ⇝* [1, 5, 1, 3]
```

l) Funkce `oddEvenFold` vrátí v uspořádané dvojici seznamy prvků z lichých a sudých pozic původního seznamu.

```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold [1, 2, 7, 5, 4] ⇝* ([1, 7, 4], [2, 5])
```

m) Funkce `takeWhileFold` vezme predikát a seznam, a vrátí nejdelší prefix seznamu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold even [2, 4, 1, 2, 4, 5, 8, 6, 8] ⇝* [2, 4]
```

n) Funkce `dropWhileFold` vezme predikát a seznam, a vrátí zadaný seznam bez nejdelšího prefixu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold odd [1, 2, 5, 9, 1, 7, 4, 6] ⇝* [2, 5, 9, 1, 7, 4, 6]
```

Př. 5.3.10 Definujte funkci `foldl` pomocí funkce `foldr`.

Př. 5.3.11 Naprogramujte funkci `insert :: Ord a => a -> [a] -> [a]` takovou, že výsledkem vyhodnocení `insert x xs` pro uspořádaný seznam `xs` bude uspořádaný seznam, který vznikne vložením prvku `x` na vhodné místo v seznamu `xs`. Například tedy:

```
insert 5 [1, 3, 18, 19, 30] ⇝* [1, 3, 5, 18, 19, 30]
```

Funkci `insert` nemusíte implementovat pomocí akumulačních funkcí, avšak chcete-li si je procvičit, můžete.

Pomocí funkce `insert` a vhodné akumulační funkce naprogramujte funkci `insertSort :: Ord a => [a] -> [a]`, která seřadí vstupní seznam pomocí algoritmu *řazení vkládáním (insert sort)*.

Př. 5.3.12 Proč je implementace funkce `or` (logická disjunkce všech hodnot v seznamu) pomocí funkce `foldr` lepší než pomocí `foldl`?

Př. 5.3.13 Pomocí dokumentace a internetu zjistěte, co dělají funkce `foldl'` a `foldl1'` z modulu `Data.List`. Jak se liší od funkcí `foldl` a `foldl1`? Kdy byste použili funkci `foldl'` místo funkce `foldl`? Zamyslete se, proč knihovna neobsahuje funkci `foldr'`.

Př. 5.3.14 Je možné definovat funkci `f` tak, aby se `foldl f [] s` vyhodnotilo na seznam obsahující jenom prvky ze sudých míst v seznamu `s`?

Je možné definovat takovou funkci ve tvaru `foldEveryOther s = snd (foldl f v s)` pro vhodné hodnoty `f` a `v`?

Př. 5.3.15 Mějme funkci `foldr2` definovanou následovně:

```
foldr2 :: (a -> a -> b -> b) -> (a -> b) -> b -> [a] -> b
foldr2 f2 f1 f0 [] = f0
foldr2 f2 f1 f0 [x] = f1 x
foldr2 f2 f1 f0 (x : y : s) = f2 x y (foldr2 f2 f1 f0 s)
```

Zkuste definovat funkci `foldr` pomocí `foldr2` a funkci `foldr2` pomocí `foldr`, nebo zdůvodněte, proč to není možné.

5.4 Akumulační funkce na vlastních datových typech

> **Pan Fešák doplňuje:** V tomto případě myslíme takzvané *katamorfismy*, tedy funkce, které nahrazují výskyty příslušných hodnotových konstruktorů za funkce dané arity. Katamorfismy jsou něco jiného než instance typové třídy `Foldable` pro daný datový typ (těmi se tu v podstatě zabývat nebudeme, najít je můžete jedině v příkladu 5.4.7, který je výrazně nad rámec předmětu).

Př. 5.4.1 Mějme klasický datový typ `BinTree a` reprezentující binární stromy, které mají v uzlech hodnoty typu `a`:

```
data BinTree a = Node a (BinTree a) (BinTree a) | Empty
                 deriving (Show, Eq)
```

Definujte funkci `treeFold`, která bude analogií seznamové funkce `foldr`, tedy tzv. *katamorfismem* na typu `BinTree`. Tedy volání `treeFold n e t` nahradí ve stromě `t` všechny hodnotové konstruktory `Node` funkcí `n` a všechny hodnotové konstruktory `Empty` hodnotou `e`. Například chceme, aby

- funkce `treeFold (\v resultL resultR -> v + resultL + resultR) 0` sečetla všechna čísla v zadaném stromě,

- funkce `treeFold (\v resultL resultR -> v * resultL * resultR) 1` vynásobila všechna čísla v zadaném stromě a
- funkce `treeFold (\v resultL resultR -> v || resultL || resultR) False` rozhodla, jestli v zadaném stromě je alespoň jedna hodnota `True`.

Zkuste si před vlastní implementací rozmyslet, jaký má funkce `treeFold` mít typ.

Př. 5.4.2 5-treeFold

Vaší úlohou je implementovat pomocí funkce `treeFold` z předchozí úlohy několik funkcí, které pracují se stromy typu `BinTree a`. Jestli úloha neříká jinak, řešení by mělo být bez formálních parametrů, tedy v následujícím tvaru:

```
functionName = treeFold (function) (term)
```

Definici datového typu `BinTree a`, několika testovacích stromů a funkce `treeFold` naleznete v souboru `05_treeFold.hs` (dá se stáhnout i z ISu).

Ke většině úloh je dostupný i ukázkový výsledek na předem zvoleném stromě (slouží jako ilustrace, co zadání vlastně požaduje). Kvůli přehlednosti jsou ukázkové stromy pojmenované a jejich definice najdete až za poslední podúlohou.

a) Funkce `treeSize` vrátí počet uzlů v zadaném stromě.

```
treeSize :: BinTree a -> Int
treeSize tree01 ⇝* 6
treeSize tree06 ⇝* 5
```

b) Funkce `treeHeight` vrátí výšku zadaného stromu (poznámka: prázdný strom má výšku 0, jednouzlový strom má výšku 1).

```
treeHeight :: BinTree a -> Int
treeHeight tree03 ⇝* 2
treeHeight tree01 ⇝* 3
```

c) Funkce `treeList` vrátí seznam hodnot ze všech uzlů. Nejdříve uveďte hodnoty z levého podstromu, pak hodnotu v uzlu a následně hodnoty z pravého podstromu (tzv. *inorder* procházení stromu).

```
treeList :: BinTree a -> [a]
treeList tree01 ⇝* [5, 3, 2, 1, 4, 1]
treeList tree02 ⇝* ["A", "B", "C", "D", "E"]
```

d) Funkce `treeConcat` vrátí zřetězení hodnot ze všech uzlů.

```
treeConcat :: BinTree [a] -> [a]
treeConcat tree02 ⇝* "ABCDE"
```

e) Funkce `treeMax` vrátí maximální hodnotu ze všech hodnot v uzlech. Hodnoty musí být z typové třídy `Ord` a `Bounded` (poznámka: zkuste hodnoty `minBound` a `maxBound`). Upozornění: Stromy, které budete používat na vyhodnocování mějte explicitně otypovány, jinak můžete narazit na problém při kompilaci (důvod je poněkud složitější).

```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax tree01 ⇝* 5
treeMax tree03 ⇝* (3, 3)
```

f) Funkce `treeFlip` vrátí zadaný strom, avšak každá jeho pravá větev bude vyměněna s příslušnou levou větví.

```
treeFlip :: BinTree a -> BinTree a
treeFlip tree01 ⇝*
    Node 2 (Node 4 (Node 1 Empty Empty)
```

```
    (Node 1 Empty Empty)) (Node 3 Empty (Node 5 Empty Empty))
treeConcat (treeFlip tree02) ⇝* "EDCBA"
```

g) Funkce `treeId` vrátí zadaný strom v nezměněné podobě (Pozor! Stále vyžadujeme použití funkce `treeFold`!).

```
treeId :: BinTree a -> BinTree a
treeId tree05 ⇝* tree05
```

h) Funkce `rightMostBranch` vrátí seznam hodnot nejpravější větve zadaného stromu (v nejpravější větvi nikdy „nezatáčíme doleva“).

```
rightMostBranch :: BinTree a -> [a]
rightMostBranch tree01 ⇝* [2, 4, 1]
rightMostBranch tree02 ⇝* ["C", "E"]
```

i) Funkce `treeRoot` vrátí kořenový prvek zadaného stromu. Jestli je strom prázdný, program havaruje (poznámka: můžete použít hodnotu `undefined`).

```
treeRoot :: BinTree a -> a
treeRoot tree01 ⇝* 2
```

j) Funkce `treeNull` zjistí, jestli je zadaný strom prázdný (podobá se funkci `null` pro seznamy).

```
treeNull :: BinTree a -> Bool
treeNull tree01 ⇝* False
treeNull tree04 ⇝* True
```

k) Funkce `leavesCount` vrátí počet listů v zadaném stromě (list je každý uzel, který nemá potomky).

```
leavesCount :: BinTree a -> Int
leavesCount tree01 ⇝* 3
leavesCount tree04 ⇝* 0
```

l) Funkce `leavesList` vrátí seznam hodnot z listů zadaného stromu. Preferované pořadí listů v seznamu je zleva doprava.

```
leavesList :: BinTree a -> [a]
leavesList tree01 ⇝* [5, 1, 1]
leavesList tree02 ⇝* ["B", "D"]
```

m) Funkce `treeMap` aplikuje zadanou funkci na hodnotu v každém uzlu zadaného stromu (poznámka: funkce pracuje podobně jako `map` na seznamech). Výsledná funkce může mít jeden formální parametr.

```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMax (treeMap negate tree01) ⇝* -1
```

n) Funkce `treeAny` zjistí, jestli alespoň jedna hodnota v zadaném stromě splňuje zadaný predikát (tedy se na něm vyhodnotí na `True`). Výsledná funkce může mít jeden formální parametr.

```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny (==10) tree01 ⇝* False
treeAny even tree01 ⇝* True
treeAny null tree02 ⇝* False
```

o) Funkce `treePair` zjistí, jestli je v každém uzlu stromu první složka uspořádané dvojice rovná druhé složce této dvojice.

```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair tree03 ⇝* False
```

p) Funkce `subtreeSums` vloží do každého uzlu zadaného stromu součet všech uzlů podstromu určeného tímto uzlem.

```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums tree01 ⇝* Node 16 (Node 8 (Node 5 Empty Empty) Empty)
    (Node 6 (Node 1 Empty Empty) (Node 1 Empty Empty))
```

Př. 5.4.3 Mějme klasický datový typ `RoseTree a` reprezentující stromy libovolné arity, které mají v uzlech hodnoty typu `a`:

```
data RoseTree a = RoseNode a [RoseTree a]
                  deriving Show
```

Definujte funkci `roseTreeFold`, která bude analogií seznamové funkce `foldr`, tedy tzv. *katamorfismem* na typu `RoseTree`. Tedy volání `roseTreeFold n e t` nahradí ve stromě `t` všechny hodnotové konstruktory `RoseNode` funkcí `n`. Například chceme, aby

- funkce `roseTreeFold (\v sums -> v + sum sums)` sečetla všechna čísla v zadaném stromě,
- funkce `roseTreeFold (\v products -> v * product products)` vynásobila veškerá čísla v zadaném stromě a
- funkce `roseTreeFold (\v ors -> v || or ors)` rozhodla, jestli v zadaném stromě je alespoň jedna hodnota `True`.

Zkuste si před vlastní implementací rozmyslet, jaký má funkce `roseTreeFold` mít typ.

Př. 5.4.4 Uvažte datový typ `RoseTree a` a akumulační funkci `roseTreeFold` z předchozí úlohy.

Pomocí funkce `roseTreeFold` implementujte analogie všech funkcí z úlohy 5.4.2, které ale budou tentokrát pracovat se stromy libovolné arity.

Př. 5.4.5 Mějme datový typ `Nat` reprezentující přirozená čísla:

```
data Nat = Succ Nat | Zero deriving (Eq, Show)
```

Definujte funkci `natFold` typu `(a -> a) -> a -> Nat -> a`, která je tzv. *katamorfismem* na typu `Nat`. Jinými slovy funkce `natFold` nahrazuje všechny hodnotové konstruktory datového typu, podobně jako funkce `foldr`, `treeFold` a `roseTreeFold`.

Příklady zamýšleného použití funkce `nfold`:

a) Funkce `natFold (Succ . Succ) Zero :: Nat -> Nat` zdvojnásobuje hodnotu přirozeného čísla typu `Nat`.
b) Funkce `natFold (1 +) 0 :: Nat -> Int` převádí hodnotu typu `Nat` do celých čísel typu `Int`.

Př. 5.4.6 Pomocí funkce `natFold` z minulého příkladu naprogramujte

a) funkci, která sečte dvě přirozená čísla typu `Nat`,
b) funkci, která rozhodne, jestli je zadané přirozené číslo typu `Nat` sudé a
c) funkci, která vynásobí dvě přirozená čísla typu `Nat` (tady se Vám možná bude hodit některá z již naprogramovaných funkcí).

Všechny předchozí funkce zkuste naprogramovat bez převodu přirozeného čísla typu `Nat` na celé číslo typu `Int` nebo `Integer`.

Př. 5.4.7 Nastudujte si, co je minimální implementace `Foldable` a implementujte instance `Foldable BinTree` pro náš typ binárních stromů (jedná se o instanci pro typový konstruktor, nikoli pro konkrétní či polymorfní typ, za `BinTree` tedy v tomto případě nemá být parametr). Následně si ověřte, že standardní funkce `sum`, `product`, `any`, `all`, apod. fungují nyní pro `BinTree a` (s vhodným `a`).

```
tree123 :: Num a => BinTree a
tree123 = Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
```

```
sum tree123 ⇝* 6
any odd tree123 ⇝* True
all odd tree123 ⇝* False
foldr (:) [] tree123 ⇝* [1, 2, 3]
```

```
data BinTree a = Node a (BinTree a) (BinTree a)
               | Empty
               deriving (Show, Eq)
```

* * *

Na konci pátého cvičení byste měli umět:

- umět použít intensionální seznamy pro generování nových seznamů ze zadaných seznamů;
- poznat, kdy se dá zadaný problém vyřešit pomocí intensionálních seznamů;
- umět pomocí intensionálních seznamů definovat nekonečné seznamy;
- použít líné vyhodnocování k práci s nekonečnými seznamy;
- umět použít nekonečné seznamy v praxi;
- použít akumulační funkce na jednoduché operace na seznamech jako součet všech prvků, maximum seznamu a podobně;
- poznat, kdy je možné problém jednoduše vyřešit pomocí akumulačních funkcí a také vybrat akumulační funkci, která pro tento účel bude vhodná.

Cvičení 6: Manipulace s funkcemi, typy, opakování

Před šestým cvičením je zapotřebí znát:

- typy základních entit v Haskellu (čísel, řetězců, seznamu, n-tic);
- základní typové třídy (pro čísla, desetinná čísla, porovnatelné a seřaditelné typy, zobrazitelné typy);
- použití operátoru `(.) :: (b -> c) -> (a -> b) -> (a -> c)` pro skládání (unárních) funkcí;
- co je částečná aplikace;
- základní funkce pro manipulace s čísly a seznamy, včetně základních funkcí vyšších řádů nad seznamy;
- definice a používání vlastních datových typů, včetně rekurzivních;
- znalost pojmů hodnotový a typový konstruktor.

Na cvičení si prosím přineste papír a tužku, budou se hodit.

Pan Fešák doporučuje: Na tomto cvičení se vám bude hodit tužka a papír ještě více než obvykle. Cílem cvičení na papír je procvičit si přemýšlení nad jednotlivými koncepty, bez spoléhání se na interpret.

Pro tohle cvičení si potřebujeme zavést následující dva pojmy.

Definice 6.1: totální funkce. Funkce v Haskellu je **totální** právě tehdy, když pro libovolný platný vstup vyprodukuje v konečném čase výstup typu požadovaného její typovou signaturou. Platným vstupem se rozumí takový, který lze samostatně vyhodnotit bez pádu či zacyklení.

Například:

- funkce `(||) :: Bool -> Bool -> Bool` je totální, protože pro libovolné vstupy vrátí platnou hodnotu typu `Bool`;
- funkce `head :: [a] -> a` není totální, protože existuje hodnota `[]`, pro niž tato funkce vyhodí chybu, a tedy nevrátí hodnotu;
- níže uvedená funkce `isEven` není totální, protože pro záporná čísla neskončí.

```
isEven :: Integral a => a -> Bool
isEven 0 = True
isEven 1 = False
isEven x = isEven (x - 2)
```

Funkce v Haskellu obecně totální nejsou. To je jeden z rozdílů mezi funkcemi v Haskellu a funkcemi v matematice (a v předmětu Matematické základy informatiky), kde je funkce bez přídomku totální a pokud potřebujeme, můžeme odlišit *parciální funkce*. Z hlediska matematických základů je tedy například výše uvedená `isEven` parciální (pro definiční obor celých čísel, který odpovídá typu `Integer`).

Poznámka: Všimněte si, že pro funkce pracující s nekonečnými strukturami se nám situace komplikuje, protože často nechceme vyžadovat konečný běh (například protože vytváříme nekonečnou strukturu). Při úvahách o totálnosti v tomto předmětu si však vystačíme bez uvažování nekonečných struktur.

Definice 6.2: chováním různé funkce. Funkce `f` a `g` označíme za **chováním různé**, pokud existuje alespoň jeden vstup na němž se jejich výstup liší. Pokud nejsou funkce chováním různé, pak jsou **chováním stejné** (také *sémanticky ekvivalentní*).

Například:

- funkce `and` a `foldr (&&) True` jsou chováním stejné;
- funkce `(||)` a `(&&)` jsou chováním různé, protože existuje vstup (například `True False`), pro který se jejich výstupy liší.

Etudy

Etuda 6.η.1 Určete nejobecnější typy výrazů. Neuvádějte typové třídy, které jsou již implikovány jinými typovými třídami. Můžete použít interpret k zjištění typů knihovních funkcí a závislostí mezi typovými třídami.

a) `not True`
b) `[]`
c) `["", []]`
d) `[True, []]`
e) `[1, 2, 3.14]`
f) `[1, 2, True]`
g) `filter (\x -> x > 5) [1, 2, 4, 8]`
h) `\x -> show (x ^ x)`
i) `\x -> read x + 2`
j) `\x -> (fromIntegral x :: Double)`

Etuda 6.η.2 Určete typy následujících funkcí:

a) `swap (x, y) = (y, x)`

b)
```
swapIf True (x, y) = (y, x)
swapIf _    (x, y) = (x, y)
```

c)
```
sayLength [] = "empty"
sayLength x  = "nonempty"
```

d)
```
aOrX 'a' _ = True
aOrX _   x = x
```

Pan Fešák připomíná: V případě, že chci otypovat funkci definovanou s více definičními rovnicemi, otypuji každou definiční rovnost zvlášť a pak *unifikuji* typy argumentů na odpovídajících pozicích a typy návratových hodnot. Tedy hledám dosazení za proměnné tak, aby oba typy byly stejné, bez toho, abych zbytečně konkretizoval typové proměnné, u kterých to není nutné.

Unifikace typů v Haskellu funguje v podstatě stejně jako unifikace termů v Prologu. Rozdíl je v tom, že Haskell dělá kontrolu sebevýskytu, protože bez ní by mohly vznikat nekonečné typy.

Etuda 6.η.3 Napište všechny *chováním různé* (definice 6.2) totální funkce (definice 6.1) typu `Bool -> Bool`.

Etuda 6.η.4 S použitím standardních funkcí z modulů `Prelude`, `Data.List` a `Data.Char` naprogramujte následující funkce bez použití vlastních pomocných funkcí, explicitní rekurze a případných zakázaných funkcí nebo konstrukcí.

a) `nth :: [a] -> Int -> a`, která vrátí n-tý prvek seznamu (pokud existuje, jinak může libovolně selhat). Bez použití funkce `(!!) :: [a] -> Int -> a`.

```
[1, 4, 16] `nth` 2 ⇝* 16
[1..] `nth` 10 ⇝* 11
```

b) `firstUppercase :: String -> Char`, která vrátí první velké písmeno ze svého vstupu, nebo `'-'`, pokud tam žádné není.

```
firstUppercase "ahoj Světe!" ⇝* 'S'
firstUppercase "ahoj" ⇝* '-'
```

c) `everyNth :: [a] -> Int -> [a]`, která pro daný (potenciálně nekonečný) seznam vrátí seznam obsahující jeho prvky na pozicích dělitelných druhým argumentem.

```
everyNth [1, 2, 3, 4, 5] 2 ⇝* [1, 3, 5]
everyNth "ahoj" 10 ⇝* "a"
everyNth [0..] 10 ≈ [0, 10..]
```

Poznámka: poslední příklad není vyhodnocení, ale ekvivalence ve smyslu, že oba výrazy se vyhodnotí na stejný seznam (v tomto případě nekonečný).

d) Pokud jste to tak už neudělali, naprogramujte `everyNth` i bez použití lambda funkcí. Místo nich použijte skládání funkcí (formální argumenty samotné funkce `everyNth` neodstraňujte, bylo by to velmi náročné).

Etuda 6.η.5 Definujte funkci `binmap :: (a -> a -> b) -> [a] -> [b]`, která je obdobou `map`, ale aplikuje danou funkci vždy na dva po sobě následující prvky tak, že prvek na pozici i ve výsledném seznamu vznikne z prvků i a $i + 1$ vstupního seznamu. Výstupní seznam je tedy o jeden prvek kratší (nebo prázdný, pokud byl vstupní seznam prázdný).

```
binmap (,) [1, 2, 3, 4] ⇝* [(1, 2), (2, 3), (3, 4)]
binmap (-) [14, 12, 8, 3, 1] ⇝* [2, 4, 5, 2]
binmap (+) [1] ⇝* []
binmap (+) [] ⇝* []
```

Pan Fešák doporučuje: Kostru úloh k této kapitole najdete připravenou k použití v souboru `06_data.hs` v příloze sbírky nebo ve studijních materiálech v ISu.

6.1 Typy a typové třídy

Př. 6.1.1 Určete, jakou aritu mají funkce podle jejich typu. Aritou rozumíme počet argumentů, které musíme funkci dát, aby typ výsledku nebyl funkční typ.

a) `Eq a => a -> a -> a -> Bool`
b) `(a -> Bool) -> ([a] -> Int)`
c) `[Int -> Int -> Int]`
d) `Int -> Integer -> String -> String`
e) `(Int -> Integer) -> String -> String`
f) `Int -> Integer -> (String -> String)`
g) `Int -> Integer -> [String -> String]`
h) `(a -> b -> c) -> b -> a -> c`

Př. 6.1.2 Určete typy následujících funkcí a popište slovně, co funkce dělají.

a)
```
cm _ []       = []
cm x (y : z) = x y ++ cm x z
```

b)
```
mm []     = (maxBound, minBound)
mm (x:xs) = let (a, b) = mm xs in (min a x, max b x)
```

c)
```
c x y = \z -> x (y z)
```

Př. 6.1.3 6-typy Bez použití interpretu určete nejobecnější typy následujících výrazů. Neuvádějte typové třídy, které jsou implikovány jinými typovými třídami. Pokud potřebujete, můžete typy základních funkcí zjistit pomocí interpretu či dokumentace.

a) `\x y -> map y x`
b) `\x -> flip replicate`
c) `\x y -> take y [fst x, snd x]`
d) `\x y -> map x . filter y`
e) ``\x y -> fromIntegral x `div` y``

Př. 6.1.4 Bez použití interpretu určete aritu následujících funkcí. Aritou rozumíme počet argumentů, které musíme funkci dát, aby typ výsledku nebyl funkční typ.

a) `flip map`
b) `(not . null)`
c) `(\f p -> filter p . map f)`
d) `(\x -> x 1 || x 2)`
e) `flip (not .)`
f) `zipWith id`

Př. 6.1.5 Napište všechny *chováním různé* totální funkce typu `(a -> b) -> a -> b` (viz definice 6.2).

Př. 6.1.6 Napište všechny *chováním různé* totální funkce typu `[a] -> a` (viz definice 6.2).

Př. 6.1.7 Napište všechny *chováním různé* totální funkce typu `(a -> Maybe b) -> Maybe a -> Maybe b` (viz definice 6.2).

Př. 6.1.8 Určete typ funkce `f`. Jak se funkce chová na různých vstupech?

```
f x y       True  = if x > 42 then y else []
f _ (_ : s) False = s
f _ _       _     = "IB015"
```

Př. 6.1.9 Vysvětlete význam a najděte příklady použití následujících funkcí:

a) `show :: Show a => a -> String`
b) `read :: Read a => String -> a`
c) `fromIntegral :: (Integral a, Num b) => a -> b`
d) `round :: (RealFrac a, Integral b) => a -> b`

6.2 Opakování základních funkcí

Př. 6.2.1 Bez použití interpretu určete, jak se vyhodnotí následující výrazy, a určete rovněž jejich nejobecnější typ (bez typových tříd, které jsou implikovány jinými typovými třídami).

a) `last [42, 3.14, 16]`
b) `zipWith mod [3, 5, 7, 4] [4, 2, 3]`
c) `(concat . map (replicate 4)) ['a', 'b', 'c']`
d) `head (filter ((> 3) . length) ["hi!", "ahoj", "hello"])`
e) `cycle [3, 2, 4] !! 10`
f) `(head . drop 3 . iterate (^ 2)) 2`

Př. 6.2.2 Uvažte následující datový typ definující zarovnání textu:

```
data PadMode = PadLeft
             | PadRight
             | PadCenter
             deriving Show
```

Definujte funkci `pad :: Int -> Char -> PadMode -> String -> String`, která na základě zadané specifikace zarovná řetězec zleva, zprava, nebo na střed (v případě, že text nelze vycentrovat přesně, udělejte levé doplnění delší). První argument značí počet znaků, na které se má zarovnat, druhý pak znak, kterým se doplňují kratší řetězce, a třetí směr zarovnávání. Pokud je řetězec delší, než zadaná délka zarovnání, vrátí jej funkce celý. V maximální míře používejte standardní funkce. Existuje řešení bez použití explicitní rekurze.

```
pad 7 '-' PadLeft   "ahoj" ⇝* "---ahoj"
pad 7 '-' PadRight  "ahoj" ⇝* "ahoj---"
pad 7 '-' PadCenter "ahoj" ⇝* "--ahoj-"
pad 3 '-' PadLeft   "ahoj" ⇝* "ahoj"
```

Př. 6.2.3 Redefinujte standardní funkci `take` za pomoci rekurze a bez použití jiných funkcí. Funkci si nejprve celou napište bez použití interpretu, následně si ji v interpretu otestuje a porovnejte chování vaší funkce s tou standardní. *Poznámka*: aby nedocházelo ke konfliktu

jmen, pojmenujte si svou implementaci například `take'`.

Př. 6.2.4 6-breaks Naprogramujte funkci `breaks :: (a -> Bool) -> [a] -> [[a]]`, která rozdělí seznam podle zadaného predikátu a prvky odpovídající tomuto predikátu z výsledku vynechá. Při řešení využijte knihovní funkci `break` (dokumentace).

```
breaks (== ' ') "ahoj svete" ⇝* ["ahoj", "svete"]
breaks (== ',') "a,b,,,cde,fg" ⇝* ["a", "b", "", "", "cde", "fg"]
breaks even [1..10] ⇝* [[1], [3], [5], [7], [9]]
breaks even [1, 3, 5, 2, 7] ⇝* [[1, 3, 5], [7]]
breaks even [1, 2, 2, 5] ⇝* [[1], [], [5]]
```

6.3 Vlastní datové typy

Př. 6.3.1 Uvažte následující (parametrizovaný) datový typ stromů s ohodnocenými listy:

```
data BinLeafTree a = LLeaf a
                   | LNode (BinLeafTree a) (BinLeafTree a)
                   deriving (Show, Eq)
```

a) Uveďte alespoň dva různé příklady hodnot tohoto typu a jejich konkrétní typy.

b) Napište všechny typové a všechny hodnotové konstruktory definované touto definicí a určete jejich arity.

c) Naprogramujte funkci `ltSumEven`, která sečte všechna sudá čísla ze všech listů celočíselných stromů. Určete i její typ tak, aby byl nejobecnější možný.

Př. 6.3.2

```
data RoseLeafTree a = RLNode [RoseLeafTree a]
                    | RLLeaf a
                    deriving (Show, Eq)
```

a) Napište všechny typové a všechny hodnotové konstruktory definované touto definicí a určete jejich arity.

b) Naprogramujte funkci `countValueLeaves`, která spočítá počet listů obsahujících hodnotu (`RLLeaf`). Určete i její nejobecnější možný typ.

c) Naprogramujte funkci `rlFilter`, která je obdobou funkce `filter` pro tyto stromy s tím, že pokud nějaký vrchol nemá po filtraci pod sebou žádné hodnoty, bude rovněž vynechán (pokud se nejedná o kořen). Určete i její nejobecnější možný typ.

```
countValueLeaves (RLLeaf 3) ⇝* 1
countValueLeaves (RLNode [RLLeaf 4, RLLeaf 3, RLLeaf 2]) ⇝* 3
countValueLeaves (RLNode [RLNode [RLNode []], RLLeaf 42]) ⇝* 1
```

```
rlFilter even (RLLeaf 3) ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 4, RLLeaf 3, RLLeaf 2])
    ⇝* RLNode [RLLeaf 4, RLLeaf 2]
rlFilter even (RLNode [RLLeaf 3, RLLeaf 5, RLLeaf 7])
    ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 1])
    ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 2])
```

```
    ⇝* RLNode [RLLeaf 2]
rlFilter (< 4) (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 2])
    ⇝* RLNode [RLLeaf 3, RLLeaf 2]
rlFilter (const True) (RLNode [RLNode [RLNode []], RLLeaf 42])
    ⇝* RLNode [RLLeaf 42]
```

6.4 Typové třídy podrobněji

Př. 6.4.1 S pomocí příkazu `:i` interpretu GHCi nebo dokumentace určete vztahy mezi typovými třídami `Num`, `Integral`, `Eq`, `Ord`, `Show`.

Př. 6.4.2 Uvažte datový typ představující slovníkovou položku zadefinovaný níže.

```
data Entry = Word String
```

Umožněte zobrazování hodnot tohoto typu a jejich porovnávání na rovnost. Dvě hodnoty jsou si rovny, pokud jsou jejich řetězce identické *bez ohledu na velikost písmen*. Formát výpisu zvolte sami.

Jinými slovy, napište instanci `Entry` pro typové třídy `Show` a `Eq`.

> Nebojte se využít funkci `toLower` (nebo `toUpper`) z modulu `Data.Char`.

> **Pan Fešák doporučuje:** Abyste zjistili, které funkce je potřeba implementovat, aby se typ stal instancí typové třídy, použijte `:i` pro danou typovou třídu.
>
> Například na základě dotazu `:i Eq` výpis začíná:
>
> ```
> class Eq a where
>     (==) :: a -> a -> Bool
>     (/=) :: a -> a -> Bool
>     {-# MINIMAL (==) | (/=) #-}
> ```
>
> Instanciací typu do typové třídy `Eq` tedy získáte funkce `(==)` a `(/=)`. Zároveň pro instanci je potřeba definovat buď funkci `(==)`, nebo `(/=)`.

Př. 6.4.3 Mějme datový typ `Shape` definovaný následovně:

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving Show
```

Naprogramujte následující funkce:

- `isEqual :: Shape -> Shape -> Bool`, která vrátí `True`, právě tehdy, když jsou si oba argumenty rovny.
- `isGreater :: Shape -> Shape -> Bool`, která vrátí `True`, pokud je první argument větší než druhý (`Shape` je větší než druhý, když má větší obsah);

Př. 6.4.4 Uvažte datový typ představující semafor zadefinovaný níže.

```
data TrafficLight = Red | Orange | Green
```

Umožněte zobrazování hodnot tohoto typu, jejich vzájemné porovnávání a řazení (zelená $<$ oranžová $<$ červená). Řečeno jinak, napište instanci `TrafficLight` pro typové třídy

`Show`, `Eq` a `Ord`.

> **Jindřiška varuje:** Nezapomeňte, že funkce, které máte implementovat při psaní instance typu pro danou typovou třídu, si můžete zobrazit pomocí `:i` *`typová třída`*.

Př. 6.4.5 Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární hodnotový konstruktor `PairD` (viz definice níže).

```
data PairT a b = PairD a b
```

Vytvořte instanci `PairT` pro typové třídy `Show`, `Eq` a `Ord`. Ať jsou si dvě dvojice rovny právě tehdy, pokud jsou si rovny po složkách. Uspořádání použijte lexikografické. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1, 2)` vypište třeba `"pair of 1 and 2"`).

Př. 6.4.6 Deklarujte typ `data BinTree a = Empty | Node a (BinTree a) (BinTree a)` jako instanci typové třídy `Eq`. Instanci si napište sami (tj. nepoužívejte klauzuli `deriving`).

Na konci cvičení byste měli zvládnout:

- otypovat výrazy a funkce, a to včetně polymorfních funkcí využívajících typové třídy a včetně funkcí s více definičními rovnostmi a s vzory;
- z typu poznat, kolik argumentů funkce má;
- z typu poznat, co může funkce dělat;
- umět psát rekurzivní funkce na seznamech a vlastních datových typech, včetně jednodušších funkcí vyšších řádů;
- umět ověřit vztahy mezi typovými třídami;
- a vše ostatní, co bylo v předchozích cvičeních.

Cvičení 7: Vstup a výstup (bonus)

Pan Sazeč dává na vědomí: Na vstup a výstup historicky bylo cvičení, ale rozhodli jsme se zaměřit se více na věci, které ukazují typické funkcionální principy. Proto je tohle cvičení označené jako bonusové – nebude k němu žádné reálné cvičení ani konzultace, jen tyto materiály. Pokud máte k těmto příkladům dotazy, zeptejte se například na diskusním fóru. Stejně tak na cvičení nenavazuje žádný domácí úkol.

Cvičení bylo víceméně ponecháno v původním stavu, včetně označení které příklady byly určeny k práci na cvičení.

Před bonusovým vstupně-výstupním cvičením je zapotřebí:

- znát význam pojmů *vstupně-výstupní akce* a *vnitřní výsledek*;
- rozumět, co znamenají typy jako `IO Integer` a `IO ()`, a chápat jejich odlišnost od `Integer` a `()`;
- vědět, k čemu slouží funkce:

```
pure      :: a -> IO a
getLine   :: IO String
putStrLn  :: String -> IO ()
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

- být seznámeni s operátory pro skládání vstupně-výstupních akcí:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>)  :: IO a -> IO b -> IO b
```

- být seznámeni se syntaxí a významem konstrukce `do`:

```
main = do input <- getLine
          let hello = "Hello there, "
          putStrLn (hello ++ input)
```

- umět přeložit zdrojový kód na spustitelný program pomocí `ghc` *`zdroják`*.

Vstup a výstup v Haskellu je na první pohled zvláštní z prostého důvodu – vyhodnocení výrazu nemůže mít vedlejší efekty, jako je výpis na obrazovku. Je ale možné výraz vyhodnotit na „scénář“ obsahující popis vstupně-výstupních operací: kdy se má načítat vstup, co se má vkládat do kterých souborů apod. Při samotném vyhodnocení se sice nic nestane, ale podle výsledného scénáře pak může vstup a výstup provádět třeba interpret.

Těmto scénářům říkáme *vstupně-výstupní akce*. Hodnota typu `IO a` je pak scénář pro získání `a`. Zde `a` je typ *vnitřního výsledku* akce, který je naplněn po vykonání akce. Vnitřní výsledek (například načtený řetězec) nemůže vstupovat do vyhodnocování výrazů, ale jiné akce jej používat mohou (například ho vypsat na obrazovku).

K takovému řetězení scénářů do větších akcí slouží operátor `>>=`. Povšimněte si jeho typu: levý argument je „scénář pro získání `a`“, pravý zjednodušeně dává „scénář, podle kterého se z `a` vyrobí `b`[7]“.

[7] Ve skutečnosti spíše „jak z `a` vyrobit scénář pro `b`“. Pro zkrácení ale můžeme např. funkci typu

Akce jako hodnoty jsou pro nás zcela neprůhledné – nemáme možnost zjistit, jaké efekty bude akce mít, aniž ji spustíme (a tedy všechny efekty provedeme). *Spouštění* je kromě spojování přes `>>=` v podstatě jediná zajímavá věc, kterou s akcemi můžeme dělat. Spouští se každá akce, kterou si necháme vyhodnotit v interpretu, a v samostatných spustitelných souborech akce `main :: IO ()`.

> **Jindřiška varuje:** Vstup a výstup je matoucí, pokud důsledně nerozlišujeme mezi pojmy *hodnota*, *(vstupně-výstupní) akce* a *vnitřní výsledek akce*. Plést si nesmíme ani *vyhodnocení* a *spuštění*.

7.1 Skládání akcí operátorem >>=

Př. 7.1.1 »= Pomocí známých funkcí a operátoru `>>=` definujte vstupně-výstupní akci, která po spuštění přečte řádek ze standardního vstupu a:

a) vypíše jej beze změny na standardní výstup;
b) vypíše jej pozpátku;
c) vypíše jej, není-li prázdný, jinak vypíše „`<empty>`“;
d) vypíše jej a také jej uloží jako vnitřní výsledek akce (bude se hodit i operátor `>>`).

Př. 7.1.2 »= Bez použití `do`-notace definujte akci `getInteger :: IO Integer`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read :: (Read a) => String -> a`.

Př. 7.1.3 »= Bez použití `do`-notace definujte vstupně-výstupní akci `loopecho :: IO ()`, která při spuštění načítá a vypisuje řádky do té doby, než načte prázdný řádek.

Př. 7.1.4 Napište akci `getSanitized :: IO String`, která načte jeden řádek textu od uživatele a z něj odstraní všechny znaky, které nejsou znaky abecedy. V úkolu použijte funkci `isAlpha` z modulu `Data.Char`.

Př. 7.1.5 Bez použití `do`-notace definujte akci, která ze standardního vstupu přečte cestu k souboru a následně uživateli oznámí, zda zadaný soubor existuje. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`.

Př. 7.1.6 Definujte funkci `(>>)` pomocí funkce `(>>=)`.

Př. 7.1.7 ☆ Napište funkce `runLeft :: [IO a] -> IO ()` a `runRight :: [IO a] -> IO ()`, které spustí všechny akce v zadaném seznamu postupně zleva (respektive zprava).

7.2 IO pomocí do-notace, převody mezi notacemi

`String -> IO Integer` označovat jako „akci, která bere řetězec a vrací číslo“, byť formálně správně se jedná o „funkci, která bere řetězec a vrací vstupně-výstupní akci s vnitřním výsledkem typu číslo“.

do-notace	*bind*-notace
`do f` `   g`	`f >> g`
`do x <- f` `   g`	`f >>= \x -> g`
`do let x = y` `   f`	`let x = y in f`

Př. 7.2.1 Vraťte se k některému ze svých řešení příkladů z předchozí sekce a přepište je na `do`-notaci. Pro ilustraci zkuste přepsat jedno snadné a jedno mírně složitější řešení a srovnejte je s řešením pomocí `>>=`.

Př. 7.2.2 Naprogramujte funkci `leftPadTwo :: IO ()`, která od uživatele načte dva řetězce a pak je vypíše na obrazovku zarovnány doprava tak, že před kratší z nich vypíše ještě vhodný počet mezer. Použijte notaci `do`.

Př. 7.2.3 Převeďte následující program v `do`-notaci na notaci s použitím `>>=`.

```
main = do
    f <- getLine
    s <- getLine
    appendFile f (s ++ "\n")
```

Př. 7.2.4 Následující funkci přepište do tvaru, ve kterém nepoužijete konstrukci `do`. Určete také typ funkce.

```
query question = do putStrLn question
                    answer <- getLine
                    pure (answer == "ano")
```

Př. 7.2.5 Funkci `query` z předchozího příkladu dále vylepšete tak, aby:

a) Rozlišovala kladné i záporné odpovědi a při nekorektní nebo nerozpoznané odpovědi otázku opakovala.

b) Akceptovala odpovědi s malými i velkými písmeny, interpunkcí, případně ve více jazycích.

Př. 7.2.6 U každého z následujících výrazů určete typ a význam, případně vysvětlete, proč není korektní:

a) `getLine`

b) `x = getLine`

c) `let x = getLine in x`

d) `let x <- getLine in x`

e) `getLine >>= \x -> pure x`

f) `getLine >>= \x -> x`

g)
```
do let x = getLine
   pure x
```

h)
```
do let x = getLine
   x
```

i)
```
do x <- getLine
   pure x
```

j)
```
do x <- getLine
   x
```

7.3 Vstupně-výstupní programy

Př. 7.3.1 Vyrobte a spusťte program, který se při chová jako akce `leftPadTwo` z příkladu 7.2.2. Připomínáme, že zdrojový kód se na spustitelný soubor překládá programem `ghc` a musí mít zadefinovanou akci `main :: IO ()`. Výsledný program se jmenuje jako zdrojový soubor bez přípony `.hs` a je nutné ho spouštět pomocí `./program`.

Př. 7.3.2 Upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určil, zda mohou být délkami hran trojúhelníku. Akce `getInteger` pochází z příkladu 7.1.2. Nezdráhejte se kód refaktorovat a vytvořit si další pomocné funkce či akce.

```
main :: IO ()
main = do putStrLn "Enter one number:"
          x <- getInteger
          putStrLn (show (1 + x))
```

Př. 7.3.3 Napište program, který vyzve uživatele, aby zadal jméno souboru, a poté ověří, že zadaný soubor existuje. Pokud existuje, vypíše jeho obsah na obrazovku, pokud ne, informuje o tom uživatele. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`.

Př. 7.3.4 Vysvětlete význam a rizika rekurzivního použití akce `main` v následujícím programu.

```
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "You shall not pass!"
          else do putStrLn (reverse s)
          main
```

Př. 7.3.5 V dokumentaci nalezněte vhodnou funkci typu `Read a => String -> Maybe a` a s jejím využitím napište funkci `requestInteger :: String -> IO (Maybe Integer)` použitelnou na chytřejší načítání čísel. Akce `requestInteger` *`delim`* od uživatele čte řádky tak dlouho, než dostane řetězec *`delim`* (potom vrátí `Nothing`), nebo celé číslo (které vrátí zabalené v `Just`). Po každém neúspěšném pokusu by měl uživatel dostat výzvu k opětovnému zadání čísla.

Př. 7.3.6 S využitím akce z předchozího příkladu napište program, který která ze standardního vstupu čte řádky s čísly, dokud nenarazí na prázdný řádek, potom vypíše jejich aritmetický průměr. Vyřešte úlohu:

a) s ukládáním čísel do seznamu;
b) bez použití seznamu.

Př. 7.3.7 Napište program `leftPad`, který je obecnější variantou akce z příkladu 7.2.2 a zarovnává libovolný počet řádků na vstupu, dokud nenalezne řádek obsahující jen tečku.[8] Řádky jsou opět zarovnány doprava na délku nejdelšího.

Následně proveďte drobnou optimalizaci: prázdné řádky nechte prázdnými, tj. bez zbytečných výplňových mezer.

Př. 7.3.8 Upravte program 07_guess.hs tak, aby parametry funkce `guess` četl z příkazové řádky. Vhod může přijít modul `System.Environment`.

[8] Proč zrovna tečku, ptáte se? Bylo nebylo… https://en.wikipedia.org/wiki/ed_(text_editor), https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol#SMTP_transport_example

Př. 7.3.9 Vymyslete a naprogramujte několik triviálních prográmků manipulujících s textovými soubory:

- počítání řádků
- výpis konkrétního řádku podle zadaného indexu
- vypsání obsahu pozpátku
- seřazení řádků, ...

Definice alternativně přepište s a bez pomoci syntaktické konstrukce `do`.

Př. 7.3.10 Představme si bohy zatracený svět, v němž neexistuje konstruktor `IO` ani vstupně-výstupní akce, které je nutné spouštět. Místo toho k vedlejším efektům dochází rovnou při vyhodnocování výrazů. Např. `getLine' :: String` se *vyhodnotí* na řádek vstupu. Vyhodnocovací strategie ale funguje stejně, jak ji v Haskellu známe. Co bude výsledkem úplného vyhodnocení výrazu `(getLine', getLine')`?

* * *

Na konci cvičení byste měli zvládnout:

- napsat v Haskellu jednoduchý program pracující se vstupem od uživatele a vypisující informace na výstup;
- za pomoci dokumentace napsat jednoduchý program pracující se soubory;
- převádět mezi konstrukcí `do` a operátory `>>=` a `>>`.

Zdá se, že vstup a výstup vyžaduje velmi odlišný přístup, speciální operátory a syntaxi a celé to může působit velmi nehaskellovsky. Ve skutečnosti jsou ale podobné konstrukce Haskellu vlastní a běžně se používají i mimo `IO`! Napovídat tomu může typ operátoru `>>=`, který zmiňuje jakousi `Monad`. *Monáda* je velmi abstraktní, avšak fascinující koncept, o němž se můžete více dozvědět v jarním semestru v navazujících předmětech IB016 *Seminář z funkcionálního programování* a IA014 *Advanced Functional Programming.*

Řešení

Cvičení 0: Technické okénko

Řeš. 0.η.0 Gratulki.

Nenechávejte si ale řešení prozradit příliš brzy. Má-li mít práce se sbírkou smysl, je potřeba se nad každým příkladem nejprve zamyslet a následně ho samostatně vypracovat. Dokonce ani když se vám implementace nedaří, nenakukujte hned do řešení – nejvíce se naučíte, pokud se nad příkladem sami trochu potrápíte. Samozřejmě klidně konzultujte přednášku nebo již hotové příklady.

Teprve až příklad zdárně vyřešíte, můžete nahlédnout do vzorového řešení. Občas se můžete dozvědět jiný (a v něčem třeba elegantnější) postup nebo nějaké tipy a triky.

Klepnutím na číslo řešení přeskočíte zpátky na příklad.

Řeš. 0.η.2 **Návod pro starší Debian a Ubuntu**

1. podle distribuce:

   (a) Pokud máte Debian, postupujte podle návodu na `https://downloads.haskell.org/debian/` („Debian Instructions“).

   (b) Pro Ubuntu si přidejte PPA repozitář `https://launchpad.net/~hvr/+archive/ubuntu/ghc` – čtěte sekci „Adding this PPA to your system“).

2. Nainstalujte si `ghc-9.0.1` (nebo novější):

   ```
   apt install ghc-9.0.1
   ```

3. Výsledné GHCi/GHC pak naleznete v cestě `/opt/ghc/bin/ghci`, resp. `/opt/ghc/bin/ghc`.

4. Tuto cestu je vhodné přidat do proměnné prostředí `PATH` abyste mohli interpret spouštět jen jako `ghci`: otevřte v libovolném textovém editoru (třeba VS code, `gedit`, či `nano`) soubor `.bashrc` ve svém domovském adresáři (nenechte se zmást tím, že není vidět ve výpisu `ls`, soubory začínající tečkou `ls` samo o sobě nevypisuje) a na jeho konec následující řádek:

   ```
   export PATH="$PATH:/opt/ghc/bin"
   ```

   Nyní již můžete interpret spouštět přímo, bez zadání celé cesty.

Řeš. 0.1.2 Vytvořit jej můžete buď v terminálu (příkazem `mkdir` *jméno* a následně do něj přepnout pomocí `cd` *jméno*), nebo v grafickém správci souborů, který by se měl otevřít v domovském adresáři a nový adresář by mělo být lze vytvořit pravým myšítkem nebo přes panel nabídek.

Řeš. 0.1.4 Spusťte textový editor (například Gedit nebo VS Code – `code` na fakultních PC) a napište či překopírujte do něj text. Nezapomeňte soubor uložit do správného adresáře.

Alternativně můžete soubor vytvořit i v terminálu, například editorem `nano`, který vám v dolní části zobrazuje klávesové zkratky, kterými se ovládá (^ znamená Ctrl).

Jiným oblíbeným terminálovým editorem je `vim`. Jeho obsluha vypadá zpočátku krkolomně, ale dá se rychle naučit: zkuste příkaz `vimtutor`, nebo `vimtutor cs`, případně `vimtutor sk`. (Ukončit je lze zadáním ':q' a odentrováním.)

Řeš. 0.2.2 Zadejte výraz a stiskněte Enter. Např. `40 + 2` se vyhodnotí na `42`. Vyhodnocení výrazů zapisujeme ve studijních materiálech pomocí lomené šipky s hvězdičkou: `40 + 2` ⇝* `42`. Další příklady:

- `4 * 10 + 2` ⇝* `42`
- `2 ^ 8` ⇝* `256`
- `(3 + 4) * 6` ⇝* `42`

Řeš. 0.2.3 Načíst soubor lze buď pomocí `:load Sem0.hs` nebo jen `:l Sem0.hs` (pokud se nachází ve stejném adresáři, kde jsme spustili GHCi), nebo tak, že spustíme GHCi přímo s tímto názvem souboru (`ghci Sem0.hs`). Konstantu vypíšeme prostě tak, že ji napíšeme do GHCi a stiskneme enter: `hello` ⇝* `"Hello, world"`. Její typ pak zjistíme příkazem interpretu `:t`, který jako parametr bere libovolný výraz: `:t hello` ⇝* `hello :: [Char]`.

Řeš. 0.2.4

```
myNumber :: Integer
myNumber = 42
```

Znovunačtení lze provést příkazem `:r`, ukončení `:q`.

V případě, že vytvoříte proměnnou jenom v interpretu, pak ta přestane po ukončení interpretu existovat.

Řeš. 0.3.1 Na Linuxu, macOS a Windows s dostatečně novým PowerShellem by mělo stačit zadat do příkazové řádky (či PowerShellu) `ssh xLOGIN@aisa.fi.muni.cz` (samozřejmě s vaším loginem) a následně zadat své fakultní heslo. Nenechte se překvapit tím, že při zadávání hesla nevidíte žádné hvězdičky; v terminálu se hesla běžně zadávají „naslepo“. Ocitnete na Aise a můžete tam používat příkazovou řádku jako na cvičení. Na Windows s Putty musíte nastavit sezení (*session*), kde jako hostitele uvedete `xLOGIN@aisa.fi.muni.cz`.

Pro trvalé přidání modulu můžete příkaz vložit do svého souboru `.bashrc`:

```
(aisa)$ echo "module add ghc" >> ~/.bashrc   # Nebo textovým editorem
```

Verze GHCi se vypíše po spuštění interpretu, nebo po zadání příkazu `ghci --version`.

Řeš. 0.3.2 Aby první následující příkaz fungoval, je potřeba být lokálně v adresáři, kde je soubor `Sem0.hs` a na Aise musí existovat adresář `ib015`.

```
(local)$ scp Sem0.hs xLOGIN@aisa.fi.muni.cz:ib015/
(aisa)$ nano Sem0.hs         # Nebo jiný způsob editace
(local)$ scp xLOGIN@aisa.fi.muni.cz:ib015/Sem0.hs Sem0_edited.hs
```

Řeš. 0.3.3 Do souboru `~/.ssh/config` na svém počítači vložte (a nastavte svůj login):

```
Host aisa
    HostName %h.fi.muni.cz
    User xLOGIN
```

Řetězec `%h` v nastavení nechte, ten představuje speciální sekvenci, za kterou se doplní jméno napsané za `Host` – takových jmen může být více (oddělených mezerou). Pokud daný soubor neexistuje, vytvořte jej (i na Windows může být jednodušší vytvořit minimálně složku `~/.ssh` pomocí příkazové řádky, ta by však existovat měla). Soubor `config` nesmí mít žádnou příponu. Část `~` v cestě značí domovský adresář. Složka `.ssh` nebude ve výpisu přes `ls` vidět, pokud nepoužijete `ls -A`, což způsobí zobrazení skrytých souborů – těch co začínají `.` (tečkou).

Řeš. 0.4.1 Jděte na https://hoogle.haskell.org/. V rozbalovací nabídce u vyhledávacího políčka zvolte `package:base` a do vyhledávacího políčka zadejte hledaný typ. Po chvíli by vám

vyhledávač měl najít funkce `(&&)`, `(||)`, `(==)` a `(/=)`. Poslední dvě mají obecnější typ, ale fungují i pro argumenty typu `Bool` – Hoogle vyhledává i obecnější funkce, do kterých lze za typové proměnné dosadit požadované typy.[9] U každé funkce také vidíte, ve kterém balíčku (`base`) a modulu (`Prelude`[10]) se nachází, a také dokumentační text, pokud je uveden. Kliknutím na funkci se dostanete do její dokumentace v prvním modulu, kde byla nalezena.

Cvičení 1: Základní konstrukce

Řeš. 1.η.2 Potřebujeme zjistit, jestli se `y` rovná číslu o jedna většímu než `x`.

```
isSucc :: Integer -> Integer -> Bool
isSucc x y = y == x + 1
```

Řeš. 1.η.4 a) V důsledku priorit operátorů je implicitní závorkování kolem násobení, tj. `5 + (9 * 3)`.

b) Operace umocňování asociuje zprava, tedy v případě více výskytů `(^)` za sebou se implicitně závorkuje zprava. Obecně tedy

`n ^ n ^ n`	=	`n ^ (n ^ n)`	≠	`(n ^ n)^ n`	=	`n ^ (n ^ 2)`
n^{n^n}	=	$n^{(n^n)}$	≠	$(n^n)^n$	=	$n^{(n^2)}$

c) Tady se setkáváme s případem, kdy je operátor neasociativní, tedy není definováno, jak se výraz zpracovává v případě výskytu více operátorů stejné priority vedle sebe, a výraz je tedy nekorektní. Důvod neasociativity je jednoduchý: u `(==)` totiž nemá smysl definovat asociativitu, protože jeho výsledek je typu `Bool`, ale argumenty mohou být jiného typu – to nakonec vidíme i na našem příkladě `3 == 3 == 3`, ať jej uzávorkujeme libovolně, nebudou nám sedět typy (budeme porovnávat číslo a `Bool`).

d) V případě, že explicitně uvedeme závorkování pro relační operátory, dostaneme se do obdobné situace jako v předchozím podpříkladu, tedy porovnání logické hodnoty a řetězce, což v Haskellu nelze. Naproti tomu výsledkem porovnání `'a' == 'a'` dostaneme v obou případech logickou hodnotu, a ty mezi sebou porovnávat můžeme, protože jsou stejného typu. Všimněte si, že v tomto výrazu se vyskytuje `==` jednou ve verzi pro znaky (`'a' == 'a'`) a jednou pro `Bool` (prostřední výskyt).

Řeš. 1.η.5 Je potřeba se zamyslet nad tím, které případy má smysl vytáhnout pomocí vzorů jako ty „speciální“. V našem případě to budou právě víkendové dny a pro vše ostatní vrátíme `False`.

```
isWeekendDay "Saturday" = True
isWeekendDay "Sunday"   = True
isWeekendDay _          = False
```

Řeš. 1.η.6 Typ snadno ověříte pomocí příkazu `:t` k otypování výrazu v GHCi (typ `[Char]` je ekvivalentní typu `String`).

a) `'a' :: Char`; libovolný Unicode znak je stejného typu: `'Ľ'`, `'ř'` nebo speciální znak nového řádku `'\n'`.

[9] V tomto případě je navíc typ omezený typovým kontextem `Eq =>`, což znamená, že za `a` lze dosadit pouze porovnatelné typy, což `Bool` splňuje.

[10] `Prelude` je základní Haskellový modul, který je vždy k dispozici.

b) `"Don't Panic." :: String` (nebo ekvivalentně `[Char]`); `""`, `"a"` nebo `"nejvnějšnější"`.

c) `not :: Bool -> Bool`; např. funkce

```
boolId :: Bool -> Bool
boolId x = x
```

d) `(&&) :: Bool -> Bool -> Bool`; např. `(||)`

e) `True :: Bool`; např. `42 == 6 * 7`

f) `(Char, Bool)`; např. `('b', False)`

Řeš. 1.1.1 Celkově zjistíme, že operátory jsou uvedeny v zadání v pořadí od nejvyšší priority (9) až k nejnižší (1).

Poznámka: Ve skutečnosti existují i operátory s prioritou 0, například $, ke kterému se časem dostaneme.

Řeš. 1.1.2 Obsah kruhu o poloměru r se vypočítá vzorečkem πr^2. Do Haskellu to snadno přepíšeme jako:

```
circleArea :: Double -> Double
circleArea r = pi * r ^ 2
```

Řeš. 1.1.3 Použijeme lokální definici `sphereVolume :: Double -> Double` pro výpočet objemu jedné koule.

```
snowmanVolume :: Double -> Double -> Double -> Double
snowmanVolume a b c = sphereVolume a + sphereVolume b + sphereVolume c
    where
        sphereVolume r = 4 * pi * (r ^ 3) / 3
```

Řeš. 1.1.4 Nejprve zjistíme, která strana je nejdelší, a pak strany pošleme ve správném pořadí do rovnice pro Pythagorovu větu, kterou si zadefinujeme pomocí lokální definice.

```
isRightTriangle :: Integer -> Integer -> Integer -> Bool
isRightTriangle x y z = if x >= y && x >= z
    then pyt y z x       -- x is the maximum
    else if y >= z       -- x is not the maximum, one of y or z must be
         then pyt x z y  -- y is the maximum
         else pyt x y z  -- z is the maximum
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Alternativou je vyzkoušet všechny tři možné volby pro nejdelší stranu (víme jistě, že pokud vybereme některou z kratších stran místo té nejdelší, rovnost v Pythagorově větě nevyjde). I zde můžeme s výhodou využít lokální definice.

```
isRightTriangle' :: Integer -> Integer -> Integer -> Bool
isRightTriangle' x y z = pyt x y z || pyt y z x || pyt x z y
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Řeš. 1.1.5

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z = max x (max y z)
```

```
max3' :: Integer -> Integer -> Integer -> Integer
max3' x y z = if x > y
    then if z > x then z else x
    else if z > y then z else y
```

Alternativní řešení pomocí konstrukce `if ... then ... else ...`:

```
max3'' :: Integer -> Integer -> Integer -> Integer
max3'' x y z = if x > y && x > z
    then x
    else if z > y then z else y
```

Řeš. 1.1.6 Přímočaré je řešení pomocí `if` a `min`/`max`, stačí si uvědomit, že prostřední číslo je menší nebo rovno maximu a větší nebo rovno minimu:

```
mid :: Integer -> Integer -> Integer -> Integer
mid x y z = if min y z <= x && x <= max y z
    then x
    else if min x z <= y && y <= max x z
        then y
        else z
```

Alternativou je vypočítat prostřední číslo za pomoci součtu, minima a maxima:

```
mid' :: Integer -> Integer -> Integer -> Integer
mid' x y z = x + y + z - max z (max x y) - min z (min x y)
```

Další alternativou je využít sílu funkce `max3`, kterou jsme již definovali (předpokládejme tedy, že je definována). Ta nám umožňuje vybírat to největší z libovolných tří celých čísel. Pokud tedy vybereme čísla z původní trojice tak, že máme zaručeno, že vybereme všechna kromě toho největšího (některé se může zopakovat), maximem z těchto čísel bude právě prostřední prvek v uspořádání podle $\leq$:

```
mid'' :: Integer -> Integer -> Integer -> Integer
mid'' x y z = max3 (min x y) (min y z) (min x z)
```

Řeš. 1.1.7 Řešení je zdlouhavé, ale celkem přímočaré. Pokud víme, že číslo n je sudé, právě když $n \bmod 2 = 0$, stačí jenom zbylé podmínky vypsat do zanořených `if`ů a dát pozor na to, že `if` v Haskellu má vždy i neúspěšnou větev po `else`. Zároveň obě větve musí být stejného *typu* (co intuitivně znamená, že obě musí vracet číslo nebo znak nebo řetězec…):

```
tell :: Integer -> String
tell n = if n > 2
        then if mod n 2 == 0 then "(even)" else "(odd)"
        else if n == 1 then "one" else "two"
```

Vnořené `if`y lze i uzávorkovat, ale je to zbytečné.

Toto řešení je poněkud špatně čitelné a použití `if` v Haskellu není vždy žádoucí. Časem si ukážeme něco lepšího.

Řeš. 1.1.8

a) Podmínka musí být logický výraz typu `Bool`, což výraz `5 - 4` není – jde o výraz celočíselného typu. Haskell nikdy sám nekonvertuje výrazy jednoho typu na druhý. Vhodná úprava celého výrazu je pak třeba `5 - 4 == 0`.

b) Výrazy v `then` a `else` větvi musí být stejného typu, protože celý podmínkový výraz musí mít vždy stejný typ bez ohledu na hodnotu podmínky. Výraz lze opravit na

```
if 0 < 3 && odd 6 then "OK" else "FAIL"
```

což už je typově správně.

c) Na první pohled podivně vypadající konstrukce, kde výsledkem podmínkového výrazu je operátor `(&&)`, je správná. V Haskellu jsou funkce/operátory rovnocenné s číselnými či jinými konstantami. Problémem je chybějící větev `else`. Podmíněný výraz má syntaktické omezení, že vždy musí obsahovat jak `then`, tak `else` větev, i když by podmínka zaručovala použití jen jedné z nich. Kdyby podmínka mohla být vyhodnocena na nepravdu a chybělo by `else`, pak by výraz neměl žádnou hodnotu, kterou by vrátil. Ale výraz v Haskellu vždy musí mít nějakou hodnotu a je tedy třeba přidat `else` větev:

```
if even 8 then (&&) else (||)
```

d) Tento výraz je v pořádku. A to i přes to, že v interpretu dostaneme podivnou hlášku:

```
> if 42 < 42 then (&&) else (||)

<interactive>:1:1: error:
    • No instance for (Show (Bool -> Bool -> Bool))
        arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```

Tato hláška však jen říká, že výslednou hodnotu výrazu nelze vypsat (kritická je tu ta část „`In a stmt of an interactive GHCi command: print it`“, která říká, že se jedná o chybu při vypisování výsledku výrazu). Konkrétně zde se interpret snaží vypsat hodnotu typu `Bool -> Bool -> Bool`, tedy binární funkci, která bere dvě pravdivostní hodnoty a jednu vrací. Funkce ale nelze v GHCi za normálních okolností vypisovat. Pokud nebudete chtít výsledek výrazu vypsat, ale jen ho otypujete pomocí příkazu `:t if 42 < 42 then (&&) else (||)`, k žádné chybě nedojde.

Řeš. 1.2.1 Při doplňování implicitních závorek je potřeba se řídit prioritou/asociativitou infixově zapsaných operátorů a závorkováním aplikace funkcí na argumenty. Postupujeme následovně:

1. Obsahuje-li výraz infixově zapsané operátory, najdeme ty s nejnižší prioritou, které nejsou v závorkách, a jejich operandy uzávorkujeme. Pokud je těchto operátorů více než jeden, jednotlivé operandy závorkujeme dle asociativity daných operátorů (pokud se vedle sebe vyskytují dva operátory se stejnou prioritou, ale různou asociativitou nebo bez asociativity, výraz je nesprávně utvořený).

2. Pokud již ve výrazu nejsou infixové operátory, tj. výraz je jednoduchý (konstanta, proměnná nebo název funkce), prefixová aplikace funkce nebo aplikace infixově zapsaného binárního operátoru na dva jednoduché argumenty, skončili jsme.

3. V opačném případě aplikujeme stejný postup na všechny podvýrazy vzniklé buď doplněnými závorkami, nebo dosud nezpracovanými závorkami v původním výrazu.

Řešení jednotlivých příkladů budou následující:

a) `(recip 2) * 5`
b) `(sin pi) + (2 / 5)`
c) ``((f g 3) * (g 5)) `mod` 7`` (funkce `f` je aplikována na 2 argumenty; asociativita)
d) `(42 < 69) || (5 == 6)`
e) `((2 + (div m 18)) == (m ^ (2 ^ n))) && ((m * n) < 20)`

Řeš. 1.2.2 Je důležité zachovávat pořadí operandů! I když jde o komutativní operátor, nelze obecně při změně mezi prefixovým a infixovým zápisem měnit jejich pořadí, protože vzniklý výraz nebude totožný. Navíc Haskell nijak negarantuje, že např. + je komutativní.

a) `(^) 4 (mod 7 5)`
b) ``3 `max` (2 + 3)``

Řeš. 1.2.3

a) `f . (g x)`
b) `2 ^ (mod 9 5)`
c) `f . (((.) g h) . id)`
d) ``((2 + (((div m 18) * m) `mod` 7)) == (((m ^ (2 ^ n)) - m) + 11)) && ((m * n) < 20)``
e) ``(f 1 2 g) + (((+) 3) `const` (g f 10))``
f) ``(replicate 8 x) ++ ((filter even) (enumFromTo 1 (3 + (9 `mod` x))))``
g) `(id id) . (flip const const)`

Řeš. 1.2.4 Nejdříve podle priority operátorů do výrazu zapíšeme implicitní závorky (kvůli různým prioritám operátorů):

```
(5 + (((7 * 5) `mod` 3) `div` 2)) == ((3 * 2) - 1)
```

Pak už lehce zjistíme, že výraz se vyhodnotí na `False`.

Při vyhodnocování výrazu v zadání se jako poslední vyhodnotí funkce s nejnižší prioritou, v našem případě `(==)`. Přepíšeme tedy do prefixu nejdříve tuto funkci:

```
(==) (5 + 7 * 5 `mod` 3 `div` 2) (3 * 2 - 1)
```

Následně v každém z argumentů opět najdeme funkci s nejnižší prioritou – v prvním je to funkce `(+)`, ve druhém pak `(-)`. Přepisem těchto funkcí do prefixu dostaneme:

```
(==) ((+) 5 (7 * 5 `mod` 3 `div` 2)) ((-) (3 * 2) 1)
```

Stejným způsobem pokračujeme i nadále. Jestliže narazíme na skupinu operátorů se stejnou prioritou (například `(*)`, `mod`, `div`), ověříme si jejich směr sdružování (závorkování). V našem případě se sdružuje (závorkuje) zleva. To v praxi znamená, že jako poslední se vyhodnotí funkce `div`. Výraz tedy přepíšeme následovně:

```
div (7 * 5 `mod` 3) 2
```

Stejným způsobem pokračujeme, dokud nám nezůstanou žádné infixově zapsané operátory:

```
(==) ((+) 5 (div (mod ((*) 7 5) 3) 2)) ((-) ((*) 3 2) 1)
```

Řeš. 1.2.5 Se závorkami:

```
(((2 + (2 * 3)) == (2 * 4)) && (((8 `div` 2) * 2) == 2)) || (0 > 7)
```

A po přepisu do prefixu:

```
(||) ((&&) ((==) ((+) 2 ((*) 2 3)) ((*) 2 4))
           ((==) ((*) (div 8 2) 2) 2))
     ((>) 0 7)
```

Řeš. 1.3.1

a)
```
logicalNot :: Bool -> Bool
logicalNot True  = False
logicalNot False = True
```

b)
```
logicalAnd :: Bool -> Bool -> Bool
logicalAnd True True = True
logicalAnd _    _    = False
```

c)
```
logicalOr :: Bool -> Bool -> Bool
logicalOr False False = False
logicalOr _     _     = True
```

Řeš. 1.3.2 Opět se zamyslíme nad tím, které případy vytáhnout do vzorů. V tomto případě to budou samohlásky, protože samohlásek je v anglické abecedě oproti souhláskám značně méně. Pro všechno ostatní jenom jednoduše vrátíme `False`:

```
isSmallVowel 'a' = True
isSmallVowel 'e' = True
isSmallVowel 'i' = True
isSmallVowel 'o' = True
isSmallVowel 'u' = True
isSmallVowel _   = False
```

Řeš. 1.3.3 Případ $a = 0$ je vhodné oddělit do samostatné definiční rovnosti. Pro diskriminant a pro řešení lineárního případu je vhodné použít lokální definici.

```
solveQuad :: Double -> Double -> Double -> (Double, Double)
solveQuad 0 b c = let x = -c / b in (x, x)
solveQuad a b c = let sd = sqrt (b * b - 4 * a * c)
                  in ((- b - sd) / (2 * a), (- b + sd) / (2 * a))
```

Všimněte si, že v 2. řádku definice ukládáme do lokální proměnné odmocninu z diskriminantu a nikoli diskriminant jako takový. To proto, abychom nemuseli odmocninu počítat dvakrát. Rovněž si můžete všimnout, že složky dvojice v témže řádku jsou velmi podobné. To můžeme napravit například tak, že si definujeme pomocnou funkci:

```
solveQuad' :: Double -> Double -> Double -> (Double, Double)
solveQuad' 0 b c = let x = -c / b in (x, x)
solveQuad' a b c = let sd = sqrt (b * b - 4 * a * c)
                       res y = (- b + y) / (2 * a)
                   in (res (- sd), res sd)
```

Řeš. 1.3.4 Spojnice bodů bude rovnoběžná s osou, jestliže buď $x$ová souřadnice obou bodů bude stejná (pak bude spojnice rovnoběžná s osou x), nebo $y$ová souřadnice bude stejná.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (x1, y1) (x2, y2) = x1 == x2 || y1 == y2
```

Řeš. 1.3.5 Řešení pomocí více definičních rovností bývá obvykle více čitelné, zde se zbavíme zanořených podmínek.

```
tell' :: Integer -> String
tell' 1 = "one"
tell' 2 = "two"
tell' n = if mod n 2 == 0 then "(even)" else "(odd)"
```

Řeš. 1.4.1

a) `True`, `False`, `not False`, `3 > 3`, `"A" == "c"`, …
Obecně libovolný správně utvořený výraz z logických hodnot a logických spojek a mnohé další.

b) `-1`, `0`, `42`, …
Libovolné celé číslo.

c) `3.14`, `2.0e-21`, `2 ** (-4)`, ale také `1`, `42`, …
Libovolné desetinné číslo, libovolný výraz vracející desetinné číslo, ale také zápis celého čísla může být interpretován jako typu `Double`, pokud to odpovídá kontextu, v němž je vyhodnocen. V interpretu si můžete ověřit, že je výraz otypovatelný na typ `Double` pomocí `:t` *`výraz`* `:: Double`.

d) `False` **není typ**! Jedná se o hodnotu typu `Bool`.

e) `(1, 1)`, `(42, 16)`, `(10 - 5, 10 ^ 10000)`, …
Libovolná dvojice celých čísel. Pokud jako `Int` zvolíte dostatečně velké číslo pak vám může „přetéct“. Toto rozmezí můžete otestovat zadáním

```
> (minBound, maxBound) :: (Int, Int)
```

do svého interpretu. `Integer` je omezen pouze pamětí počítače.

f) `(0, 3.14, True)`, …
Trojice, složky musí odpovídat typům.

g) `()`
Takzvaná nultice je typem s jedinou hodnotou. Typ `()` někdy také označujeme jako jednotkový typ nebo v angličtině *unit*. Ačkoli význam takového typu nemusí zatím dávat v Haskellu smysl, časem se s ním setkáme. Nultice je jediným základním typem v Haskellu, kde je typ i hodnota zapisována stejným řetězcem znaků v kódu.

h) `((), (), ())`
Jediná možná hodnota je trojice, jejímž každým prvkem je nultice.

Řeš. 1.4.2

a) `Bool`, výraz je hodnotovým konstruktorem tohoto typu.

b) `String` (ekvivalentně `[Char]`), libovolný výraz v dvojitých uvozovkách je v Haskellu typu `String`.

c) `Bool`, při typování musíme nejprve znát typ funkce `not :: Bool -> Bool` a hodnoty `True :: Bool`. Aplikací funkce se signaturou `Bool -> Bool` na jeden parametr typu `Bool` dostaneme výraz typu `Bool`. Typ prvního parametru v signatuře funkce musí souhlasit s typem reálného prvního parametru při aplikaci, což zde platí.

d) `Bool`, jednotlivé podvýrazy: `(||) :: Bool -> Bool -> Bool`, `True :: Bool`, `False :: Bool`. Typy reálných parametrů odpovídají parametrům v signatuře operátoru `(||)`.

e) Nesprávně utvořený výraz. Jednotlivé podvýrazy: `True :: Bool`, `"" :: String`, `(&&) :: Bool -> Bool -> Bool`. Typ druhého reálného parametru `String` neodpovídá typu druhého parametru signatury, `Bool`. Haskell neprovádí žádné implicitní typové konverze, proto výraz nelze otypovat.

f) `Integer`, výraz `1` může být typu `Integer`, a tedy je možné jej dosadit jako parametr funkce `fun`.

g) Nesprávně utvořený výraz. Výraz `3.14` nemůže být typu `Integer`, protože se nejedná o celé číslo, tedy jej nelze dosadit do funkce `fun`.

h) `Int`, protože funkce bere dva parametry typu `Int` a `Int` vrací. Výrazy `3` i `8` mohou být `Int`, a tedy je lze dosadit jako parametry.

Řeš. 1.4.3 a) Výraz `not a || b` se uzávorkuje `(not a) || b`, přičemž operátor `(||)` má typ `Bool -> Bool -> Bool`, tedy výraz `not a` má být typu `Bool` a i výraz `b` je typu `Bool`. Nyní stačí jenom určit typ výrazu `a`, který známe z toho, že výraz `not` je typu `Bool -> Bool`, tedy výraz `a` je typu `Bool`. Platí tedy `implication :: Bool -> Bool -> Bool`.

b) Funkce `foo` bere dva argumenty a jejím výsledkem je výraz typu `Bool`. Z prvního řádku předpisu víme, že druhý argument je typu `String`, a ze druhého řádku předpisu víme, že první argument je typu `Char`. Funkce `foo` má tedy typ `Char -> String -> Bool`.

c) Z prvního řádku definice vidíme, že první argument musí být typu `Bool` a funkce taktéž vrací `Bool`. Z druhého řádku pak vidíme, že typ druhého argumentu musí být stejný jako typ návratové hodnoty. O typu třetího argumentu nevíme vůbec nic – může to být cokoli. Celkově tedy dostáváme `ft :: Bool -> Bool -> a -> Bool`.

Řeš. 1.4.5 Důležitý je zde typ operátoru `(/) :: Fractional a => a -> a -> a`. Pro srovnání uveďme další příklady aritmetických operací: `(+) :: Num a => a -> a -> a`, `div :: Integral a => a -> a -> a`. Všimněte si, že zatímco u sčítání máme typové omezení, že argumenty jsou čísla, u dělení pomocí `(/)` jsou to desetinná čísla[11]. U celočíselného dělení pak můžeme používat jen celá čísla. Právě tato omezení způsobí, že ve výrazech jako `(3 / 2)` se odvodí, že argumenty dělení musí být typy schopné reprezentovat desetinná čísla.

Ve skutečnosti celočíselný *literál* (tedy zápis čísla v kódu) může být libovolného číselného typu podle potřeby kódu, v němž je použit. Desetinný literál (např. `3.14`) pak může být libovolného desetinného typu. Odvozování typů z operací funguje samozřejmě i při zanořování operací – u výrazu `((7 + 8) / 5)` bude výsledkem desetinné číslo a i sčítání se bude provádět nad desetinnými čísly.

Poznámka: Na to, aby se operace mohla provést musí nakonec interpret odvodit konkrétní typ, v němž se výraz vyhodnotí, nestačí mu tedy jen vědět, že se jedná například o nějaký desetinný typ. Zde platí jednoduché pravidlo, celá čísla se bez dalších omezení vyhodnocují v typu `Integer` a desetinná v typu `Double`. *Tento postup však nijak neovlivňuje typ výrazu a přichází na řadu až při vyhodnocování v interpretu.*

Cvičení 2: Rekurze a seznamy

Řeš. 2.η.1 Použijte příkaz `:t` k otypování výrazu v `ghci` (typ `[Char]` je ekvivalentní typu `String`).

a) `[[Char]]` (což je stejné jako `[String]`)
b) `[Char]` (což je stejné jako `String`)
c) `[Char]` (což je stejné jako `String`)
d) `[Char]` (což je stejné jako `String`)
e) `[(Bool, ())]`
f) `[String]`, při otypování takovýchto výrazů je třeba si dát pozor. Výraz sice obsahuje funkci `++`, která má v tomto kontextu typ `String -> String -> String`, avšak `String -> String -> String` není výsledný typ, protože funkci už byly dodány argumenty, a tedy typ prvků v seznamu je `String`.
g) `[Bool -> Bool -> Bool]`
h) `[a]`, z výrazu nevyplývá žádné omezení na typ prvků, který může obsahovat, proto je typ prvků úplně obecný, tedy `a`.

[11] Přesněji jde o typy, pomocí niž lze reprezentovat zlomky. Tyto typy jsou definovány právě tím, že umožňují dělení.

i) `[[a]]`, podobně jako v předešlém případě, žádné omezení na typ prvků vnitřního seznamu.
j) `[[Bool]]` typové omezení vzniká kvůli konkrétní hodnotě ve druhém prvku.

Řeš. 2.η.2 Stačí použít dvě definiční rovnosti a vzory pro seznamy. Prázdný seznam je prázdný, žádný jiný seznam není prázdný (duh).

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Řeš. 2.η.3 Opět stačí použít vzory pro seznamy. Případy, kdy vstupem funkce je prázdný seznam, buď není potřeba vůbec definovat, nebo lze použít funkci `error` nebo hodnotu `undefined`.

```
myHead :: [a] -> a
myHead (x : _) = x
myHead []      = error "myHead: Empty list."

myTail :: [a] -> [a]
myTail (_ : xs) = xs
myTail []       = error "myTail: Empty list."
```

Řeš. 2.η.5

- `[]`
  Tento vzor představuje prázdný seznam. Nemůže reprezentovat žádný z uvedených seznamů.
- `x`
  Na tento vzor se může navázat libovolná hodnota, a tedy zejména libovolný ze zadaných seznamů.
- `[x]`
  Představuje libovolný jednoprvkový seznam. Z uvedených může reprezentovat seznamy `[1]`, `[[]]`, `[[1]]`.
- `[x, y]`
  Představuje libovolný dvouprvkový seznam. Z uvedených může reprezentovat seznamy `[1, 2]`, `[[1], [2, 3]]`.
- `(x : s)`
  Libovolný neprázdný seznam. Proměnná `x` reprezentuje první prvek, proměnná `s` seznam ostatních prvků. Tento vzor může reprezentovat všechny uvedené seznamy (ano, i `[[]]`).
- `(x : y : s)`
  Představuje libovolný seznam, který má alespoň 2 prvky. Proměnná `x` reprezentuje první prvek, `y` druhý prvek a `s` seznam ostatních prvků. Z uvedených může reprezentovat seznamy `[1, 2]`, `[1, 2, 3]`, `[[1], [2, 3]]`.
- `[x : s]`
  Jednoprvkový seznam, jehož jediným prvkem je neprázdný seznam. Proměnná `x` reprezentuje první prvek vnitřního seznamu, proměnná `s` seznam ostatních prvků vnitřního seznamu. Z uvedených může reprezentovat pouze seznam `[[1]]`.
- `((x : y) : s)`
  Představuje neprázdný seznam, jehož prvním prvkem je neprázdný seznam. Proměnné `x` a `y` reprezentují první prvek prvního prvku a seznam ostatních prvků prvního prvku, proměnná `s` reprezentuje ostatní prvky vnějšího seznamu. Z uvedených může reprezentovat seznamy `[[1]]`, `[[1], [2, 3]]`.

Řeš. 2.η.6 Myšlenka řešení může být následující: 0 je sudá, 1 není sudá, a každé jiné kladné číslo je sudé právě tehdy, když číslo o 2 menší je sudé.

```
isEven :: Integer -> Bool
isEven 0 = True
isEven 1 = False
isEven x = isEven (x - 2)
```

Vymyslet se dá i řešení snižující parametr o jedna a používající funkci `not`.

Výraz `isEven -1` skončí s typovou chybou, neboť minus se přednostně chápe jako binární operátor a jedná se tak o odečtení jedničky od funkce. V případě `isEven (-1)` se již jedná skutečně o zápornou jedničku a při definici výše dojde k zacyklení, neboť se funkce bude vždy rekurzivně volat s menším argumentem (podle posledního řádku) a nikdy se nezastaví na některém z prvních dvou vzorů.

Řeš. 2.1.1 Myšlenka řešení může být následující: zbytek 0 po dělení třemi je 0, zbytek 1 po dělení třemi je 1, zbytek 2 po dělení třemi je 2 a zbytek dělení třemi pro každé jiné kladné číslo je stejný, jako zbytek po dělení třemi pro číslo o 3 menší.

```
mod3 :: Integer -> Integer
mod3 0 = 0
mod3 1 = 1
mod3 2 = 2
mod3 x = mod3 (x - 3)
```

Řeš. 2.1.2 Myšlenka je podobná jako v předešlém příkladu. Jen se zde počítá, kolikrát je potřeba odečíst 3, aby se argument dostal do intervalu $\langle 0, 3 \rangle$.

```
div3 :: Integral i => i -> i
div3 0 = 0
div3 1 = 0
div3 2 = 0
div3 x = 1 + div3 (x - 3)
```

Řeš. 2.1.3

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

Funkce je opět definována po částech. Předpokládáme, že dostane jako argument jenom nezáporné celé číslo. Pokud je argument 0, výsledek je zřejmě 1. Pokud je naopak argument kladné číslo, víme, že $n! = n \times (n-1)!$, kde druhý výsledek získáme pomocí rekurzivního volání. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace prefixově zapsané funkce má vyšší prioritu než infixové operátory.

Řeš. 2.1.4 Bázový případ $z^0 = 1$. Pro každé kladné n naopak platí $z^n = z \cdot z^{n-1}$, přičemž hodnota z^{n-1} jde vypočítat rekurzivně.

```
power :: (Num n, Integral i) => n -> i -> n
power _ 0 = 1
power x n = x * power x (n - 1)
```

Všimněte si, že definovaná funkce vždy použije n rekurzivních volání. Tento počet se ale dá zmenšit, když si uvědomíme, že při každém rekurzivním volání není potřeba n snižovat jen o 1, ale je možné ho zmenšit přibližně na polovinu. Platí totiž

- $z^0 = 1$

- $z^{2n} = (z^n)^2$
- $z^{2n+1} = (z^n)^2 \cdot z$

Následující implementace, která tuto myšlenku využívá, tedy potřebuje jen přibližně $\log(n)$ rekurzivních volání.

```
power' _ 0 = 1
power' z n = if even n then half * half else half * half * z
    where half = power' z (n `div` 2)
```

Řeš. 2.1.5 Stačí si uvědomit, že číslo je mocninou 2 právě tehdy, když je 1 nebo je sudé a jeho polovina je mocninou 2.

```
isPower2 :: Integral i => i -> Bool
isPower2 0 = False
isPower2 1 = True
isPower2 x = even x && isPower2 (div x 2)
```

Řeš. 2.1.6

```
digitsSum :: Integral i => i -> i
digitsSum 0 = 0
digitsSum x = x `mod` 10 + digitsSum (x `div` 10)
```

Řeš. 2.1.7 K řešení můžeme použít známý Euklidův algoritmus. Konkrétně použijeme jeho rekurzivní verzi, která využívá zbytky po dělení.

```
mygcd :: Integral i => i -> i -> i
mygcd x 0 = x
mygcd x y = mygcd (min x y) ((max x y) `mod` (min x y))
```

Řeš. 2.1.8 Příklad lze vyřešit i bez generování všech prvočísel. Stačí zadané číslo dělit 2 tolikrát, kolikrát je to možné, a zároveň přičítat za každé vydělení k výsledku jedničku. Poté číslo budeme dělit 3, 4, 5, 6, atd., dokud po dělení nezbyde výsledek 1. Ačkoliv například čísla 4 a 6 nejsou prvočísla, ale přesto jimi dělíme, není to problém, protože pokud jsme číslo už vydělili 2 a 3, kolikrát to bylo možné, nemůže už být dělitelné ani 4 ani 6.

```
primeDivisors :: Integral i => i -> Integer
primeDivisors n = divisorsFrom n 2
    where
        divisorsFrom 1 _ = 0
        divisorsFrom n d = if mod n d == 0
                           then 1 + divisorsFrom (n `div` d) d
                           else divisorsFrom n (d + 1)
```

Řeš. 2.1.9

```
plus :: Integral i => i -> i -> i
plus 0 y = y
plus x y = plus (pred x) (succ y)

times :: Integral i => i -> i -> i
times 0 y = 0
times x 0 = 0
times 1 y = y
times x y = plus y (times (pred x) y)
```

```
plus' :: Integral i => i -> i -> i
plus' x y = if x >= 0
    then plus x y
    else negate (plus (negate x) (negate y))

times' :: Integral i => i -> i -> i
times' x y = if (x < 0 && y < 0) || (x >= 0 && y >= 0)
    then times (abs x) (abs y)
    else negate (times (abs x) (abs y))
```

Řeš. 2.1.10 Pro zadané číslo n funkce přičte $2n - 1$ k výsledku rekurzivního volání pro vstup o jedna menší. Ten vrátí $2(n-1) - 1$ a součet rekurzivního volání o 1 menší. To se bude dít tak dlouho, než vstup bude 0, pro nějž funkce vrátí 0. Tedy výsledkem bude součet

$$(2n - 1) + (2(n-1) - 1) + (2(n-2) - 1) + \ldots + (2 - 1) + 0.$$

To lze zapsat přesněji též jako

$$\sum_{i=1}^{n} (2i - 1).$$

Předchozí výraz je perfektně správné řešení úlohy. Nicméně s použitím trochy matematiky lze řešení zapsat i elegantněji, aby bylo opravdu vidět, co zadaná funkce počítá.

Odečtení n jedniček lze vytknout za celý součet, a tedy výsledek lze zapsat i jako

$$\left(\sum_{i=1}^{n} 2i\right) - n.$$

Násobení dvojkou lze též vytknout před součet, a tedy výsledek lze zapsat i jako

$$2\left(\sum_{i=1}^{n} i\right) - n.$$

Ze střední školy možná víte, že součet aritmetické řady $\sum_{i=1}^{n} i$ je $\frac{n \cdot (n+1)}{2}$. Takže výsledek lze zapsat též jako

$$2\frac{n \cdot (n+1)}{2} - n = n \cdot (n+1) - n = n \cdot n = n^2.$$

Pokud si chcete procvičit látku z Matematických základů informatiky, můžete si zkusit právě odvozenou rovnost

$$\sum_{i=1}^{n} (2i - 1) = n^2$$

dokázat matematickou indukcí.

Řeš. 2.2.1 Stačí použít funkci `(:)`. Při otypování funkce musíme uvážit, že všechny prvky seznamu musí mít stejný typ, a to i nově vložená dvaačtyřicítka. Chtělo by se říci, že `42` je typu `Integer`, ale z předchozích příkladů již víme, že neotypovaný číselný literál (tj. číslo objevující se ve zdrojovém kódu) se může chovat jako libovolné číslo z typové třídy `Num`. Proto:

```
add42 :: Num n => [n] -> [n]
add42 xs = 42 : xs
```

Řeš. 2.2.2 Případ prázdného seznamu nemusíme řešit. Pro jednoprvkový seznam vrátíme rovnou jeho poslední prvek:

```
getLast [x] = x
```

Všechny zbývající případy seznamů mají alespoň dva prvky. Jednoduchá úvaha vede k tomu, že poslední prvek seznamu, který má alespoň dva prvky, je stejný jako poslední prvek téhož seznamu, ale bez prvního prvku. Tedy ze vstupního seznamu odstraníme první prvek a na zbytek aplikujeme rekurzivně funkci `getLast`:

```
getLast (x : xs) = getLast xs
```

Řeš. 2.2.3 Funkci definujeme obdobně jako funkci `getLast`. Začneme jednoprvkovým seznamem, kdy výsledkem je prázdný seznam:

```
stripLast [x] = []
```

Všechny zbývající případy seznamů mají dva nebo více prvků. V takovém případě bude první prvek zadaného seznamu určitě ve výsledném seznamu a zbytek lze vypočítat rekurzivně:

```
stripLast (x : xs) = x : stripLast xs
```

Srovnejte s definicí funkce `getLast`.

Řeš. 2.2.4 Délka prázdného seznamu je 0. Délka alespoň jednoprvkového seznamu je o 1 větší, než délka vstupního seznamu bez prvního prvku.

```
len :: [a] -> Integer
len []       = 0
len (_ : xs) = 1 + len xs
```

Výpočet této funkce probíhá například takto:

```
len (1 : (2 : [])) ⇝ 1 + len (2 : []) ⇝ 1 + (1 + len [])
    ⇝ 1 + (1 + 0) ⇝* 2
```

Řeš. 2.2.5

```
nth :: Int -> [a] -> a
nth 0 (x : _)  = x
nth n (_ : xs) = nth (n - 1) xs
```

Řeš. 2.2.6 Prázdný seznam neobsahuje nic. Neprázdný seznam obsahuje zadaný prvek právě tehdy, když je onen prvek prvním prvkem zadaného seznamu, nebo ho obsahuje zbytek zadaného seznamu.

```
contains :: Eq a => [a] -> a -> Bool
contains []       _ = False
contains (x : xs) e = e == x || xs `contains` e
```

Řeš. 2.2.7 Myšlenka je podobná jako u funkce `contains`, jen je potřeba si počítat, kolikrát zadaný prvek ještě chceme vidět. Pokud se dostaneme na 0, číslo už jsme viděli dostatečněkrát, a vrátíme tedy `True`.

```
containsNtimes :: Eq a => Integer -> a -> [a] -> Bool
containsNtimes 0 _ _        = True
containsNtimes _ _ []       = False
containsNtimes n x (y : ys) = if x == y
    then containsNtimes (n - 1) x ys
    else containsNtimes n x ys
```

Řeš. 2.2.8 Funkce `getPoints` je podobná funkci `contains`, ale místo logické hodnoty budeme vracet příslušnou hodnotu.

```
getPoints :: String -> [(String, Integer)] -> Integer
getPoints _ [] = 0
getPoints wanted ((name, points) : xs) = if wanted == name
    then points
    else getPoints wanted xs
```

U funkce `getBest` se hodí definovat si pomocnou funkci, která jako další argument dostane i jméno a počet bodů aktuálně nejlepšího studenta. Tohoto aktuálně nejlepšího studenta bude v průběhu vypočtu měnit a na konci seznamu ho vrátí.

```
getBest :: [(String, Integer)] -> String
getBest (x : xs) = fst (getBestWithDefault x xs)
    where
        getBestWithDefault current [] = current
        getBestWithDefault (curN, curP) ((newN, newP) : xs) =
            if newP > curP
            then getBestWithDefault (newN, newP) xs
            else getBestWithDefault (curN, curP) xs
```

Řeš. 2.2.9 Nejjednodušší je funkci definovat podle vzoru na prvním argumentu. Pokud je první seznam prázdný, výsledkem je přímo druhý seznam. Pokud je první seznam neprázdný, výsledný seznam obsahuje první prvek prvního seznamu a pak zřetězení zbytku prvního seznamu s druhým seznamem.

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x : xs) ys = x : append xs ys
```

Řeš. 2.2.10 Zde se hodí vzor pro seznamy délky alespoň dva, protože potřebujeme pojmenovat první dva prvky vstupního seznamu. Poté stačí udělat z nich dvojici a zbytek seznamu vyřešit rekurzivně.

```
pairs :: [a] -> [(a, a)]
pairs (x : y : s) = (x, y) : pairs s
pairs  _          = []
```

Řeš. 2.2.11 a) Součet prázdného seznamu je 0. Součet neprázdného seznamu je součet prvního prvku a součtu zbytku seznamu.

```
listSum :: Num n => [n] -> n
listSum []       = 0
listSum (x : xs) = x + listSum xs
```

b) Existují nejméně dva přístupy k řešení, pokud nechceme explicitně pracovat s délkou seznamu. Jeden z nich je, že využijeme vzoru `(x : y : zs)`, který bere ze seznamu *po dvou* prvcích, tím pádem víme, že pokud tak skončíme na jednom prvku, seznam musel obsahovat lichý počet prvků:

```
oddLength :: [a] -> Bool
oddLength [] = False
oddLength [_] = True
oddLength (_ : _ : zs) = oddLength zs
```

Druhý přístup k řešení je, že odpověď postupně *vyskládáme* z prázdného seznamu,

protože víme, že ten obsahuje sudý počet prvků. Každým dalším prvkem odpověď změníme na opačnou, s posledním prvkem získáme odpověď pro celý seznam:

```
oddLength' :: [a] -> Bool
oddLength' [] = False
oddLength' (_ : xs) = not (oddLength xs)
```

c) Opět přímočará rekurzivní definice funkce podle vzorů.

```
add1 :: Num n => [n] -> [n]
add1 [] = []
add1 (x : xs) = (x + 1) : add1 xs
```

d) Opět přímočará rekurzivní definice funkce podle vzorů.

```
multiplyN :: Num n => n -> [n] -> [n]
multiplyN _ [] = []
multiplyN n (x : xs) = (n * x) : multiplyN n xs
```

e) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteEven :: Integral i => [i] -> [i]
deleteEven [] = []
deleteEven (x : xs) = if even x
    then deleteEven xs
    else x : deleteEven xs
```

f) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteElem :: Eq a => a -> [a] -> [a]
deleteElem _ [] = []
deleteElem n (x : xs) = if x == n
    then deleteElem n xs
    else x : deleteElem n xs
```

g) Opět přímočará rekurzivní definice funkce podle vzorů.

```
largestNumber :: [Integer] -> Integer
largestNumber [x] = x
largestNumber (x : xs) = x `max` largestNumber xs
```

h) Stačí si uvědomit, jaké všechny případy mohou nastat. Jediný zajímavý případ je, když oba vstupní seznamy jsou neprázdné. V takovém případě je potřeba porovnat první prvky obou seznamů a také rekurzivně porovnat zbytky obou seznamů.

```
listsEqual :: Eq a => [a] -> [a] -> Bool
listsEqual [] [] = True
listsEqual [] _ = False
listsEqual _ [] = False
listsEqual (x : xs) (y : ys) = x == y && listsEqual xs ys
```

i) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
multiplyEven :: [Integer] -> [Integer]
multiplyEven [] = []
multiplyEven (x : xs) = if even x
    then (2 * x) : multiplyEven xs
    else multiplyEven xs
```

j) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
sqroots :: [Double] -> [Double]
sqroots [] = []
```

```
    sqroots (x : xs) = if x > 0
        then sqrt x : sqroots xs
        else sqroots xs
```

Řeš. 2.2.12 Řešení se podobá funkci `nth`, při dosažení nuly se ale pokračuje dál s restartovaným čítačem. Povšimněte si, že v následující implementaci používá lokálně definovaná funkce `everyNthOffset` hodnotu `n`, která je argumentem vnější funkce.

```
everyNth :: Integer -> [a] -> [a]
everyNth n xs = everyNthOffset xs 0
    where
        everyNthOffset [] _ = []
        everyNthOffset (x : xs) 0 = x : everyNthOffset xs (n - 1)
        everyNthOffset (x : xs) m = everyNthOffset xs (m - 1)
```

Řeš. 2.2.13 Myšlenka řešení je procházet řetězec a držet si čítač právě otevřených a dosud neuzavřených závorek. Ten na konci musí být nulový a zároveň nikdy během průchodu nesmí klesnout pod nulu.

```
brackets :: String -> Bool
brackets s = bracketsWithDiff s 0
    where
        bracketsWithDiff [] k = k == 0
        bracketsWithDiff ('(' : xs) k = bracketsWithDiff xs (k + 1)
        bracketsWithDiff (')' : xs) k = k > 0 &&
                                        bracketsWithDiff xs (k - 1)
```

Řeš. 2.2.14 Funkci, která rozhodne, jestli je řetězec palindromem, zadefinujeme jednoduše pomocí funkce `reverse` a porovnání.

```
palindrome :: String -> Bool
palindrome str = str == reverse str
```

Po krátkém zamyšlení zjistíme, že na doplnění slova na palindrom nám stačí najít nejdelší příponu slova, která tvoří palindrom. Vynechané znaky ze začátku pak doplníme i na konec řetězce v obráceném pořadí.

```
palindromize :: String -> String
palindromize s = if palindrome s
    then s
    else [head s] ++ palindromize (tail s) ++ [head s]
```

Poznámka: Vzhledem k častému využívání sekvenčního spojování seznamů `(++)` nemá tato funkce optimální časovou složitost. Zkuste se zamyslet, jak by se dala napsat efektivnější funkce.

Řeš. 2.2.15

```
getMiddle :: [a] -> a
getMiddle xs = tortoiseRabbit xs xs
    where
        tortoiseRabbit (t : _) [_] = t
        tortoiseRabbit (t : _) [_, _] = t
        tortoiseRabbit (_ : ts) (_ : _ : rs) = tortoiseRabbit ts rs
```

Cvičení 3: Funkce vyšších řádů, λ-funkce, částečná aplikace, skládání

Řeš. 3.η.2

```
oddify :: Integral a => [a] -> [a]
oddify xs = map (\x -> if odd x then x else x + 1) xs
```

Řeš. 3.η.3

```
inputWithOddified :: Integral a => [a] -> [(a, a)]
inputWithOddified xs = zip xs (oddify xs)
```

Řeš. 3.η.5

a) V prvním případě se aplikují funkce v opačném pořadí než v druhém.
b) Zápisy jsou ekvivalentní. V prvním případě bychom mohli vynechat formální parametr `x` i závorky.
c) Zápisy jsou ekvivalentní. *Poznámka*: Závorky kolem `x + 2` v druhém výrazu jsou nutné.
d) V prvním případě dostaneme chybu. Při vyhodnocování výrazu se nejdříve uzávorkují operandy `(.)`, a tudíž dostaneme výraz `head . (head [[1], [2], [3]])`. Protože `(head [[1], [2], [3]])` není unární funkcí, kterou po nás požaduje typ operátoru `(.)`, dostaneme typovou chybu. Druhý výraz je validní.

Řeš. 3.η.6

a) `(take 4) [1, 2, 3, 4, 5, 6, 7, 8]` ⇝* `[1, 2, 3, 4]`
Funkce, která očekává seznam a zkrátí ho na nejvýše 4 prvky.

b) `((++) "Hello, ") "World"` ⇝* `"Hello, World"`
Funkce, která očekává řetězec a přidá před něj řetězec `"Hello, "`.

c) `(zip3 [1, 2, 3] ["a", "b"]) [True]` ⇝* `[(1, "a", True)]`
Funkce, která očekává jeden seznam a sezipuje ho se seznamy `[1, 2, 3]` a `["a", "b"]` na seznam trojic, přičemž prvky vstupního seznamu se dostanou do třetích složek trojic.

d) `(^ 2) 5` ⇝* `25`
Funkce, která očekává číslo a spočítá jeho druhou mocninu.

Řeš. 3.η.7

a) `((==) 42) 16` – každá Haskellová funkce arity alespoň dva se nejprve aplikuje na svůj první argument, čímž vznikne funkce o jedna nižší arity, která se dále aplikuje na další argumenty.

b) `(map ((==) 42)) [1, 2, 3]` – stejný případ jako minulý bod: funkce `map` se nejprve částečně aplikuje na svůj první argument a výsledná funkce se aplikuje na druhý argument. Pokud bychom chtěli jít do detailu, měli bychom i rozepsat seznam pomocí `(:)` a `[]`: `(map ((==) 42)) (1:(2:(3:[])))`.

c) `(g 4, (f g) 5)` – je důležité si především všimnout, že `f` je (alespoň binární) funkce aplikovaná na dva argumenty `g` a `5`.

d) `( (zipWith3 f) ((g 4) a) ) xs` – všimněte si, že funkce stále není plně aplikovaná.

e) `Bool -> (Bool -> Bool)` – závorkování typu zprava přímo odpovídá závorkování výrazu zleva (porovnejte s `(||) False True`).

f) `(a -> b -> c -> d) -> ([a] -> ([b] -> ([c] -> [d])))` – všimněte si, že závorka končící před šipkou (`->`) má výrazně jiný význam, než ta začínající za šipkou.

g) `(b -> c) -> ((a -> b) -> (a -> c))` – všimněte si, že závorka kolem `a -> c` odpovídá obvyklému pohledu na funkci `(.)`, která má tento typ – tedy jako na binární funkci skládání funkcí (která produkuje funkci).

Řeš. 3.1.1

```
filterOutShorter :: [String] -> Int -> [String]
filterOutShorter ls n = filter (\str -> length str >= n) ls
```

Řeš. 3.1.2

```
getNames :: [(String, Integer)] -> [String]
getNames s = map fst s

successfulRecords :: [(String, Integer)] -> [(String, Integer)]
successfulRecords s = filter (\(_, p) -> p >= 8) s

successfulNames :: [(String, Integer)] -> [String]
successfulNames s = getNames (successfulRecords s)

-- zde by lambda byla moc dlouhá
successfulStrings :: [(String, Integer)] -> [String]
successfulStrings s = map formatStudent (successfulRecords s)
    where
        formatStudent (n, p) = n ++ ": " ++ show p ++ " b"
```

Řeš. 3.1.3 Funkci `map` lze využít v případě, že potřebujeme jistým způsobem modifikovat každý prvek zadaného seznamu.

```
add1' :: [Integer] -> [Integer]
add1' xs = map (\x -> x + 1) xs

multiplyN' :: Integer -> [Integer] -> [Integer]
multiplyN' n xs = map (\x -> x * n) xs
```

Funkci `filter` naopak použijeme, chceme-li ze vstupního seznamu vybrat pouze některé prvky.

```
deleteEven' :: [Integer] -> [Integer]
-- chci odstranit sudá čísla, ponechám tedy ty prvky,
-- pro které platí odd (číslo je liché)
deleteEven' xs = filter odd xs

deleteElem' :: Integer -> [Integer] -> [Integer]
deleteElem' n xs = filter (\x -> x /= n) xs
```

Funkce `map` a `filter` lze vhodně kombinovat, pokud chci prvky modifikovat a zároveň filtrovat.

```
multiplyEven' :: [Integer] -> [Integer]
multiplyEven' xs = map (\x -> x * 2) (filter even xs)

sqroots' :: [Double] -> [Double]
sqroots' xs = map sqrt (filter (\x -> x > 0) xs)
```

Zbývající funkce nelze vhodně implementovat pomocí `map` a `filter`: `listSum`, `oddLength` a `listsEqual` se vyhodnocují na jeden prvek (typu `Integer` nebo `Bool`), ale `map` a `filter` vrací seznamy. Funkce `listsEqual` musí najednou procházet dva seznamy, ale `map` a `filter` rekurzivně prochází vždy pouze jeden seznam (lze elegantně řešit použitím funkce `zipWith`, je však potřeba ohlídat, zda mají seznamy stejnou délku).

Řeš. 3.1.4

```
import Data.Char
toUpperStr :: String -> String
```

```
toUpperStr = map toUpper
```

Řeš. 3.1.5 Nejdřív si zadefinujeme pomocný predikát `isvowel`, který o znaku určí, jestli je samohláskou. Následně jednotlivé řetězce projdeme funkcí `filter`.

```
vowels :: [String] -> [String]
vowels s = map (filter isvowel) s
  where
    isvowel :: Char -> Bool
    isvowel c = elem (toUpper c) "AEIOUY"
```

Řeš. 3.1.7

```
assignPrizes :: [String] -> [Integer] -> [(String, Integer)]
assignPrizes = zip

formatPrizeText :: String -> Integer -> String
formatPrizeText n p = n ++ ": " ++ show p ++ " Kc"

prizeTexts :: [String] -> [Integer] -> [String]
prizeTexts ns ps = zipWith formatPrizeText ns ps
```

Řeš. 3.1.8

```
map (\(x, y, z) -> if x then y else z)
    (zip3 [True, False, False, True, False]
          [1, 2, 3, 4] [16, 42, 7, 1, 666])
    ⇝* [1, 42, 7, 4]
zipWith3 (\x y z -> max x (max y z))
          [7, 4, 11, 2] [5, 7, 1] [16, 5, 0, 1]
    ⇝* [16, 7, 11]
```

V prvním případě se jedná o funkci arity 1, která bere trojici (kde první složka je `Bool` a další jsou čísla).

V druhém případě má lambda funkce aritu 3, bere tři čísla.

Řeš. 3.1.9

```
neighbors :: [a] -> [(a, a)]
neighbors xs = zip xs (tail xs)
```

Řeš. 3.1.10

```
f1 :: [Integer] -> Bool
f1 (x : y : s) = x == y || f1 (y : s)
f1 _           = False
```

Nebo kratší řešení používající funkci `zipWith` a funkci `or`, která spočítá logický součet všech hodnot v zadaném seznamu:

```
f2 :: [Integer] -> Bool
f2 s = or (zipWith (==) s (tail s))
```

Řeš. 3.1.11

```
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x : xs) = f x : myMap f xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x : xs) = if p x
    then x : myFilter p xs
```

```
    else myFilter p xs

myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith f (x : xs) (y : ys) = f x y : myZipWith f xs ys
myZipWith _ _ _ = []
```

Řeš. 3.1.12 Využít můžeme funkce `any` a `all`.
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:any
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:all

Řeš. 3.1.13 http://haskell.fi.muni.cz/doc/base/Prelude.html#v:takeWhile
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:dropWhile

Řeš. 3.1.15

```
blueless :: [(Int, Int, Int)] -> [(Int, Int, Int)]
blueless colors = filter (\(r, g, b) -> b == 0) colors

greyscale :: [(Int, Int, Int)] -> [(Int, Int, Int)]
greyscale colors = filter (\(r, g, b) -> r == g && g == b) colors

polychromatic :: [(Int, Int, Int)] -> [(Int, Int, Int)]
polychromatic cs = filter (\(r, g, b) -> (r > 0 && g > 0)
                                      || (g > 0 && b > 0)
                                      || (r > 0 && b > 0)) cs

colorsToString :: [(Int, Int, Int)] -> [String]
colorsToString cs = map (\(r, g, b) -> "r: " ++ show r ++ " g: "
                                    ++ show g ++ " b: " ++ show b) cs
```

Řeš. 3.1.16

```
quickSort :: [Integer] -> [Integer]
quickSort [] = []
quickSort [x] = [x]
quickSort (x : xs) =
    quickSort (filter (\y -> y < x) xs) ++
    [x] ++
    quickSort (filter (\y -> y >= x) xs)
```

Řeš. 3.2.1

a) `(^) 3 2` ⇝* `9`
Funkce, která vypočítá danou mocninu trojky.

b) `(^ 3) 2` ⇝* `8`
Funkce, která dané číslo umocní na třetí.

c) `(3 ^) 2` ⇝* `9`
Funkce, která umocní trojku daným číslem.

d) `(- 2)` ⇝* `-2`
Číslo -2. Jelikož `-` je binární i unární operátor, nelze jej použít v pravé operátorové sekci. Místo toho však existuje funkce `subtract`: `subtract 2 44` ⇝* `42`.

e) `(2 -) 1` ⇝* `1`
Funkce, která dané číslo odečte od dvojky.

f) `(zipWith (+) [1, 2, 3]) [41, 14]` ⇝* `[42, 16]`
Funkce, která očekává seznam a po prvcích jej sečte se seznamem `[1, 2, 3]`, produkuje tedy seznam délky nejvýše 3.

g) `(map (++ "!")) ["ahoj", "hi"]` $\rightsquigarrow^*$ `["ahoj!", "hi!"]`
Funkce, která očekává seznam řecezů a produkuje nový seznam řetězců ve kterém je ke každému řetězci na konec přidán „!“.

h) Syntakticky špatně utvořený výraz. Operátorové sekce musí být vždy v závorkách.

Řeš. 3.2.2

a)
```
sumLists' :: Num a => [a] -> [a] -> [a]
sumLists' xs ys = zipWith (+) xs ys
```

b)
```
upper' :: String -> String
upper' xs = map toUpper xs
```

c)
```
embrace' :: [String] -> [String]
embrace' xs = map ('[' :) (map (++ "]") xs)
```

Nebo lze pro `(:)` použít prefixový tvar:

```
embrace'' :: [String] -> [String]
embrace'' xs = map ((:) '[') (map (++ "]") xs)
```

d)
```
sql' :: (Ord a, Num a) => [a] -> a -> [a]
sql' xs lt = map (^ 2) (filter (< lt) xs)
```

Řeš. 3.2.3

a) Ne. První výraz je díky implicitním závorkám částečné aplikace ekvivalentní `((f 1) g) 2` a odpovídá funkci `f` beroucí tři parametry a druhý je ekvivalentní `(f 1) (g 2)`.

b) Ano, `(f 1 g) 2` ≡ `f 1 g 2` ≡ `(f 1) g 2` (tedy funkce `f` tu bere dva argumenty).

c) Ne, `(* 2) 3` ≡ `(*) 3 2` ≡ `3 * 2`. Neexistuje pravidlo, které by zaručovalo, že `3 * 2` se bude rovnat `2 * 3` (standard jazyka Haskell komutativitu operátoru `(*)` nevynucuje). Nezapomínejme, že všechny operátory definované typovými třídami můžeme předefinovat. *Poznámka*: (pokročilejší) Toto by bylo možné pouze v případě, že by komutativity vyžadovaly axiomy typové třídy, ve které je daný operátor/funkce definována. Ani to by však nezaručovalo skutečnou korektnost – interpret/kompilátor platnost axiomů nekontroluje (ani to není v jeho silách). Zůstává pouze důvěra v programátora, že jeho implementace je korektní.

d) Ano, `(* 3)` je pravá sekce.

Řeš. 3.2.4 Následující výrazy jsou ekvivalentní:

- `a f g b` ≡ `((a f) g) b` ≡ `(a f) g b` (a ≡ c ≡ e)

  V těchto výrazech jsou arity následující:

  - `a` – arita alespoň 3 (bere minimálně 3 argumenty – `f`, `g`, `b`)
  - `f`, `g`, `b` – arita alespoň 0 (konstanta)

- `(a f) (g b)` ≡ `a f (g b)` (b ≡ f)

  Arity:

  - `a` – arita alespoň 2 (argumenty `f` a `(g b)`)
  - `g` – arita alespoň 1
  - `f`, `b` – arita alespoň 0

- `a (f (g b))` nemá ekvivalentní výraz (d)

  Arity:

  - `a`, `f`, `g` – arita alespoň 1
  - `b` – arita alespoň 0

- `a (f g b)` nemá ekvivalentní výraz (g)

  Arity:

  - `f` – arita alespoň 2
  - `a` – arita alespoň 1
  - `g`, `b` – arita alespoň 0

Řeš. 3.2.5 Následující typy jsou ekvivalentní:

- `A -> B -> C -> D -> E` ≡ `A -> (B -> C -> D -> E)` ≡ `A -> B -> C -> (D -> E)` ≡ `A -> (B -> (C -> (D -> E)))` (a ≡ c ≡ d ≡ f)

  Arita funkce je 4, nejedná se o funkci vyššího řádu.

- `(A -> B) -> C -> D -> E` nemá ekvivalentní typ (b)

  Arita je 3, jde o funkci vyššího řádu (1. argument je funkce `A -> B`).

- `A -> ((B -> C) -> D -> E)` ≡ `A -> (B -> C) -> D -> E` (e ≡ h)

  Arita 3, jde o funkci vyššího řádu (2. argument je funkce `B -> C`).

- `(((A -> B) -> C) -> D) -> E` nemá ekvivalentní typ (g)

  Arita 1, jde o funkci vyššího řádu (která bere funkci vyššího řádu).

Řeš. 3.3.1

a) `map even :: Integral a => [a] -> [Bool]`
b) `map head . snd :: (a, [[b]]) -> [b]`
c) `filter ((4 >) . last) :: (Ord a, Num a) => [[a]] -> [[a]]`
d) `const const :: a -> b -> c -> b`

Řeš. 3.3.2

```
countStudentsByPoints :: Integer -> [(String, Integer)] -> Int
countStudentsByPoints pt s = length (filter (== pt) (map snd s))

countStudentsByPoints' :: Integer -> [(String, Integer)] -> Int
countStudentsByPoints' pt = length . filter ((== pt) . snd)

studentNamesByPoints :: Integer -> [(String, Integer)] -> [String]
studentNamesByPoints pt s = getNames (filter ((== pt) . snd) s)

studentsStartingWith :: Char -> [(String, Integer)]
                     -> [(String, Integer)]
studentsStartingWith c = filter ((== c) . head . fst)
```

Řeš. 3.3.3

a)
```
failing' :: [(Int, Char)] -> [Int]
failing' sts = map fst (filter ((== 'F') . snd) sts)

failing'' :: [(Int, Char)] -> [Int]
failing'' = map fst . (filter ((== 'F') . snd))
```

b)
```
embraceWith' :: Char -> Char -> [String] -> [String]
embraceWith' l r = map ((l :) . (++ [r]))
```

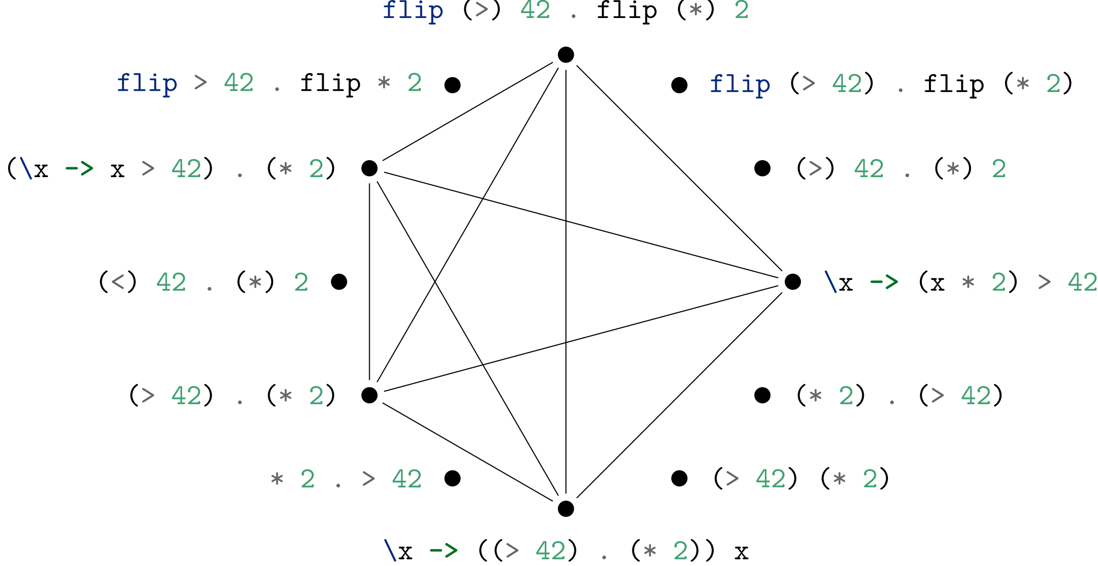
Argumenty `l` a `r` nelze rozumně odstranit.

c)
```
divisibleBy7' :: [Integer] -> [Integer]
divisibleBy7' = filter ((== 0) . (`mod` 7))
```

d)
```
letterCaesar' :: String -> String
letterCaesar' = map (chr . (3 +) . ord) . filter isLetter
```

e)
```
zp' :: (Integral a, Num b) => [a] -> [b] -> [b]
zp' = zipWith (flip (^))
```

Řeš. 3.3.4 Vzájemně ekvivalentní funkce jsou spojeny:



Všechny vzájemně ekvivalentní výrazy můžeme získat různými ekvivalentními úpravami z `\x -> (x * 2) > 42`.

Zbývající výrazy:

- `(* 2) . (> 42)` by nejprve porovnával vstup s hodnotou `42` a poté teprve přičítal `2` k výsledku typu `Bool`, je tedy typově nesprávný.
- `(> 42) (* 2)` aplikuje sekci `(> 42)` na `(* 2)`, `(> 42)` však vyžaduje na vstupu číslo, ale `(* 2)` je typu `Num a => a -> a`. Výraz je tedy typově nesprávný.
- `* 2 . > 42` je syntakticky nesprávný, operátorové sekce je vždy potřeba uzávorkovat.
- `(<) 42 . (*) 2` není nutně ekvivalentní pro všechny vstupy, protože nic negarantuje, že `(*)` je komutativní a při prohození argumentů lze zaměnit `(>)` za `(<)`. Jelikož jsou tyto funkce definovány zvlášť pro každý datový typ v dané typové třídě, je možné, že nějaká implementace toto splňovat nebude.
- `flip > 42 . flip * 2` se uzávorkuje jako `flip > ((42 . flip) * 2)`, a pokouší se tedy skládat číslo 42 a funkci `flip` a dokonce tento výsledek složení násobit dvěma. Tento výraz je tedy typově nesprávný.
- `flip (> 42) . flip (* 2)` – `flip` očekává binární funkci, ale `(> 42)` a `(* 2)` jsou nutně unární.
- `(>) 42 . (*) 2` je ekvivalentní s `\x -> 42 > (2 * x)`, argumenty jsou tedy otočeny.

Řeš. 3.3.5

a) Naším cílem je ze zadané funkce vytvořit negovanou funkci. Z typu funkce `negp` vidíme, že můžeme uvést dva argumenty – predikát a hodnotu. Pak jen na výsledek volání `f` zavoláme funkci `not`, která realizuje logickou negaci.

```
negp :: (a -> Bool) -> a -> Bool
negp f x = not (f x)
```

b) Funkci z předchozího příkladu můžeme přepsat do tvaru složení funkcí:

```
negp f x = (not . f) x
```

Odtud můžeme následně odstranit formální argument:

```
negp f = not . f
```

K tomuto výsledku můžeme dojít i přímo, uvědomíme-li si, že negace predikátu je složením predikátu s funkcí negace.

c) Dále lze tělo funkce přepsat do prefixového tvaru:

```
negp f = (.) not f
```

A následně lze odstranit poslední formální argument `f`, čímž dostaneme definici plně bez formálních argumentů:

```
negp = (.) not
```

Alternativně lze tělo funkce upravit pomocí operátorové sekce:

```
negp f = (not .) f
negp = (not .)
```

Poznámka: Z hlediska elegance a čistoty kódu by byla většinou programátorů v Haskellu pravděpodobně preferována varianta `negp f = not . f`.

Řeš. 3.3.7

a)
```
\x -> (f . g) x
f . g
```

b)
```
\x -> f . g x
\x -> (.) f (g x)
\x -> ((.) f . g) x
(.) f . g
```

c)
```
\x -> f x . g
\x -> (.) (f x) g
\x -> flip (.) g (f x)
\x -> (flip (.) g . f) x
flip (.) g . f
```

Řeš. 3.3.8

a)
```
(^ 2) . mod 4 . (+ 1)
\x -> ((^ 2) . mod 4 . (+ 1)) x
\x -> (^ 2) (mod 4 ((+ 1) x))
\x -> (mod 4 (x + 1)) ^ 2
```

b)
```
(+) . sum . take 10
\x -> ((+) . sum . take 10) x
\x -> (+) (sum (take 10 x))
\x y -> (+) (sum (take 10 x)) y
\x y -> sum (take 10 x) + y
```

c)
```
map f . flip zip [1, 2, 3]
\x -> (map f . flip zip [1, 2, 3]) x
```

```
\x -> map f (flip zip [1, 2, 3] x)
\x -> map f (zip x [1, 2, 3])
```

d)
```
(.)
\f g -> (.) f g
\f g -> f . g
\f g x -> (f . g) x
\f g x -> f (g x)
```

Řeš. 3.3.9

a)
```
f :: a -> b -> b
f x y = y
f x y = const y x
f x y = flip const x y
f = flip const
```

b)
```
f :: Num a => a -> b -> a
f x y = const (3 + x) y
f x = const (3 + x)
f x = const ((3 +) x)
f x = (const . (3 +)) x
f = const . (3 +)
```

Řeš. 3.3.10

a)
```
\_ -> x
\t -> x
\t -> const x t
const x
```

b)
```
\x -> f x 1
\x -> flip f 1 x
flip f 1
```

c)
```
\x -> f 1 x True
\x -> (f 1) x True
\x -> flip (f 1) True x
flip (f 1) True
```

d)
```
const x
```

e)
```
\x -> 0
\x -> const 0 x
const 0
```

f) Není možno převést, poněvadž `if ... then ... else ...` není klasická funkce, ale syntaktická konstrukce, podobně jako `let ... in ...`.

g)
```
\f -> flip f x
\f -> flip flip x f
flip flip x
```

Řeš. 3.3.11

a) Postupně převádíme:

```
f1 x y z = x
f1 x y z = const x z -- přidáme z tak, abychom ho mohli odstranit
f1 x y = const x
```

```
f1 x y = const (const x) y -- přidáme y
f1 x = const (const x)
f1 x = (const . const) x
f1 = const . const
```

b)
```
f2 x y z = y
f2 x y z = const y z
f2 x = const -- eta-redukujeme obojí
f2 x = const const x -- přidáme x
f2 = const const
```

c)
```
f3 x y z = z
f3 x y z = id z -- přidáme uměle identitu
f3 x y = id
f3 x y = const id y -- přidáme y
f3 x = const id
f3 x = const (const id) x -- přidáme x
f3 = const (const id)
```

Řeš. 3.3.12 Několikrát po sobě použijeme funkci `flip`.
```
g = flip (flip ... (flip (flip f c1) c2) ... cn)
```

Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe`

Řeš. 4.η.1
```
weekend :: Day -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False
```

Pokud je typ `Day` zaveden v typové třídě `Eq`, můžeme použít i následující alternativní definici funkce `weekend`:

```
weekend' :: Day -> Bool
weekend' d = d == Sat || d == Sun
```

Případně můžeme využít i instance `Ord`:

```
weekend'' :: Day -> Bool
weekend'' d = d >= Sat
```

Řeš. 4.η.2

a) Příklady hodnot jsou:

```
Cube 1 2 3
Cylinder (-3) (1/2)
```

Některé z těchto hodnot sice nemusí odpovídat skutečným tělesům, ale uvedený datový typ je umožňuje zapsat.

b) Hodnotové konstruktory jsou umístěny jako první identifikátor ve výrazech oddělených svislítky. Tedy v tomto případě to jsou `Cube`, `Cylinder`. Také hodnotový konstruktor začíná velkým písmenem.

c) Typové konstruktory můžeme rozlišit na nově definované a na ty, které jsou jenom použité. Typový konstruktor je vždy umístěn jako první identifikátor za klíčovým slovem `data`, tedy v tomto případě `Object`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Double`.

d) Funkci budeme definovat po částech. Pro každý možný tvar hodnoty typu `Object`, tj. pro každý typ tělesa definujeme samostatný řádek funkce. Poznamenejme, že je nutné použít závorky kolem argumentů funkcí, aby byl tento výraz považován jako jeden argument, ne za několik argumentů. K definici funkcí můžeme využít i konstantu `pi`, která je v Haskellu standardně dostupná.

```
volume :: Object -> Double
volume (Cube x y z)   = x * y * z
volume (Cylinder r v) = pi * r * r * v

surface :: Object -> Double
surface (Cube x y z)   = 2 * (x * y + x * z + y * z)
surface (Cylinder r v) = 2 * pi * r * (v + r)
```

Řeš. 4.η.3

```
data Contained a = NoValue | Single a | Pair a a
data Compare a = SameContainer
               | SameValue (Contained a)
               | DifferentContainer

cmpContained :: Eq a => Contained a
                     -> Maybe (Contained a)
                     -> Compare a
cmpContained NoValue (Just NoValue) =
    SameValue NoValue
cmpContained (Single x1) (Just (Single x2)) =
    if x1 == x2
        then SameValue (Single x1)
        else SameContainer
cmpContained (Pair x1 y1) (Just (Pair x2 y2)) =
    if x1 == x2 && y1 == y2
        then SameValue (Pair x1 y1)
        else SameContainer
cmpContained _ _ = DifferentContainer
```

Nezapomeňte, že hodnotový konstruktor lze vnímat jako funkci jako každou jinou. Proto je potřeba závorky psát jenom tam, kde je potřeba určit pořadí, v jakém se mají funkce aplikovat.

V případě parametrů je jen potřeba odlišit, co všechno tvoří jeden parametr.

Řeš. 4.η.4

a) Korektní typ s hodnotou například `Nothing` nebo `Just 'A'`.
b) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu.
c) Korektní hodnota typu `Maybe String`.
d) Nekorektní výraz – hodnotový konstruktor `Just` očekává hodnotu, `Integer` je ovšem typ.
e) Korektní hodnota typu `Maybe (Maybe a)`.
f) Korektní hodnota typu `Num a => [Maybe a]`.
g) Korektní hodnota typu `Bool -> Maybe a -> Maybe a`.

Řeš. 4.η.5

```
safeDiv :: Integral a => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

Řeš. 4.η.6 `Maybe`-hodnotu je třeba rozebrat pomocí vzorů a v závislosti na tom, zda jde o `Just` obsahující hodnotu, nebo o `Nothing`, vrátit buď obsaženou hodnotu, nebo výchozí hodnotu.

```
valOrDef :: Maybe a -> a -> a
valOrDef (Just x) _ = x
valOrDef Nothing  d = d
```

Téměř tato funkce existuje i ve standardní knihovně a to v modulu `Data.Maybe` a jmenuje se `fromMaybe` – tyto dvě funkce se liší jen pořadím argumentů.

Řeš. 4.1.1

a) Příklady hodnot jsou:

```
Steel

Warrior 100 Leather 5
Warrior (-42) Leather (-4)
```

Definice datových typů nevynucuje, aby celočíselné parametry byly nezáporné.

b) Hodnotové konstruktory jsou v tomto případě `Steel`, `Leather` a `Warrior`.

c) Typové konstruktory jsou v tomto případě `Armor` a `Warrior`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Integer`. `WeaponsPoints` a `HealthPoints` nejsou typové konstruktory, pouze typové aliasy.

d) Řešení může být následující:

```
attack :: Warrior -> Warrior -> Warrior
attack (Warrior _ _ hit) (Warrior hp Leather w) =
    Warrior (max 0 (hp - hit)) Leather w
attack (Warrior _ _ hit) (Warrior hp Steel w) =
    Warrior (max 0 (hp - hit `div` 2)) Steel w
```

Řeš. 4.1.2

```
data Jar = EmptyJar
         | Cucumbers
         | Jam String
         | Compote Int
           deriving (Show, Eq)
today :: Int
today = 2024

stale :: Jar -> Bool
stale EmptyJar    = False
stale Cucumbers   = False
stale (Jam _)     = False
stale (Compote x) = today - x >= 10
```

Alternativní řešení s menším počtem vzorů a hlavně přehlednějším zápisem (víme, že výsledek pro první tři případy je stejný):

```
stale' :: Jar -> Bool
stale' (Compote x) = today - x >= 10
stale' _           = False
```

Řeš. 4.1.3

a) Nulární typový konstruktor `X`, unární hodnotový konstruktor `Value`.

b) Nulární typové konstruktory `M` a `N`, nulární hodnotové konstruktory `A`, `B`, `C`, `D`, unární hodnotové konstruktory `N` a `M`.

c) Chybná deklarace: `Hah` je v seznamu použito jako typový konstruktor, jedná se však o hodnotový konstruktor.
d) Nulární typový konstruktor `FNM`, ternární hodnotový konstruktor `T`.
e) Vytváří se pouze typové synonymum (nulární).
f) Nulární typový konstruktor `E`, unární hodnotový konstruktor `E`.

Řeš. 4.1.4
a) Ok.
b) Nok, typová proměnná `a` musí být argumentem konstruktoru `Makro`.
c) Nok, hodnotový konstruktor není možné použít vícekrát.
d) Nok, typová proměnná `c` musí být argumentem konstruktoru `Fun`.
e) Nok, argumenty konstruktoru `Fun` mohou být pouze typové proměnné, ne složitější typové výrazy (tj. není možné použít definici podle vzoru).
f) Nok, `Z X` není korektní výraz, protože `X` je hodnotový konstruktor.
g) Nok, každý hodnotový konstruktor musí začínat velkým písmenem.
h) Nok, výraz je interpretován jako hodnotový konstruktor `Makro` se třemi argumenty: `Int`, `->` a `Int` – je nutné přidat závorky kolem `(Int -> Int)`.
i) Nok, syntax výrazu je chybná: `type` musí mít na pravé straně pouze jednu možnost (jedná se o typové synonymum, ne o nový datový typ).
j) Ok, typový i hodnotové konstruktory mají stejný název. Na pravé straně definice je `X` nejdřív binárním hodnotovým a pak dvakrát nulárním typovým konstruktorem. Jediná plně definovaná hodnota tohoto typu je `x = X x x`.

Řeš. 4.1.5
a) Matematická definice rovnosti zlomků nám říká, že $\frac{a}{b} = \frac{c}{d} \Leftrightarrow a \cdot d = b \cdot c$. Funkci pak už snadno postavíme na této rovnosti.

```
fraceq :: Frac -> Frac -> Bool
fraceq (a, b) (c, d) = a * d == b * c
```

b) Zde opět použijeme vestavěnou funkci `signum`.

```
nonneg :: Frac -> Bool
nonneg (a, b) = signum a == signum b
```

c) K řešení nám opět pomůže si nejdříve matematicky zapsat požadovaný výraz: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$.

```
fracplus :: Frac -> Frac -> Frac
fracplus (a, b) (c, d) = simplify (a * d + b * c, b * d)
```

d)

```
fracminus :: Frac -> Frac -> Frac
fracminus (a, b) (c, d) = fracplus (a, b) (-c, d)
```

e)

```
fractimes :: Frac -> Frac -> Frac
fractimes (a, b) (c, d) = simplify (a * c, b * d)
```

f)

```
fracdiv :: Frac -> Frac -> Frac
fracdiv (_, _) (0, _) = error "division by zero"
fracdiv (a, b) (c, d) = fractimes (a, b) (d, c)
```

g) Pro úpravu do základního tvaru postačí vydělit čitatele i jmenovatele jejich největším společným jmenovatelem (můžeme použít vestavěnou funkci `gcd`, která pracuje i se zápornými čísly). Musíme si však dát pozor na znaménko – základním tvarem zlomku $\frac{-2}{-4}$ je $\frac{1}{2}$ a nikoliv $\frac{-1}{-2}$. To můžeme zajistit následovně: číslo ve jmenovateli základního tvaru budeme mít vždy kladné a znaménko přeneseme do čitatele (například pomocí vestavěné funkce `signum`).

```
simplify :: Frac -> Frac
simplify (a, b) = ((signum b) * (a `div` d), abs (b `div` d))
```

```
      where d = gcd a b
```

Poznámka: Haskell má vestavěný typ pro zlomky, `Rational`. Ten je reprezentován v podstatě stejným způsobem, tedy jako dvě hodnoty typu `Integer`. Nicméně nejedná se přímo o dvojice, ale typ `Rational` definuje hodnotový konstruktor `%`, tedy například zlomek $\frac{1}{4}$ zapíšeme jako `1 % 4`. Datový typ `Rational` je instancí mnoha typových tříd, mimo jiné `Num` a `Fractional`, proto s ním lze pracovat jako s jinými číselnými typy v Haskellu a používat operátory jako `(+)`, `(-)`, `(*)`, `(/)`.

Řeš. 4.1.6

```
commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
    where
        pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups

data PricingType = Common | Employee | Student

computePrice :: PricingType -> Int -> (Int -> Float)
             -> (Float -> Float) -> (Int -> Float) -> Float
computePrice Common   cups cp _  _  = cp cups
computePrice Employee cups cp ed _  = (ed . cp) cups
computePrice Student  cups _  _  sp = sp cups
```

Řeš. 4.1.7

```
commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
    where
        pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups

data PricingType' = Common'
                  | Special (Int -> Float)
                  | Discount (Float -> Float)

computePrice :: PricingType' -> Int -> (Int -> Float) -> Float
computePrice Common'       cups cp = cp cups
computePrice (Special sp)  cups _  = sp cups
computePrice (Discount dp) cups cp = dp (cp cups)

common :: PricingType'
common = Common'

employee :: PricingType'
```

```
employee = Discount employeeDiscount

student :: PricingType'
student = Special studentPricing
```

Řeš. 4.2.1

a) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu.
b) Nekorektní výraz – hodnotový konstruktor `Just` je aplikovaný na příliš mnoho argumentů.
c) Korektní hodnota typu `Num a => Maybe (Maybe a)`.
d) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu `Nothing`.
e) Nekorektní výraz – nulární hodnotový konstruktor `Nothing` nebere žádné argumenty.
f) Korektní hodnota typu `(Num a) => Maybe [Maybe a]`.
g) Korektní hodnota (funkce) typu `a -> Maybe a`.

Řeš. 4.2.2

a) Korektní typ s hodnotou například `Nothing`.
b) Nekorektní výraz – hodnotový konstruktor `Just` očekává hodnotu, `a` je ovšem typ.
c) Korektní hodnota typu `(Num a) => Maybe (a -> a)`.
d) Korektní hodnota (ne funkce) typu `Maybe (a -> Maybe a)`.
e) Nekorektní výraz – implicitní závorky jsou `(Just Just) Just` a podle předchozího příkladu víme, že `Just Just :: Maybe (a -> Maybe a)`. Avšak tento výraz není funkcí (je to `Maybe` výraz – podstatný je vnější typový konstruktor), a proto ho nemůžeme aplikovat na hodnotu, jako by to byla funkce.

Řeš. 4.2.3

```
divlist :: Integral a => [a] -> [a] -> [Maybe a]
divlist = zipWith safeDiv
```

Funkce `safeDiv` je definovaná stejně jako v příkladu 4.η.5.

Řeš. 4.2.5

```
mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]
mayZip (a : xa) (b : xb) = (Just a, Just b) : (mayZip xa xb)
mayZip []       (b : xb) = (Nothing, Just b) : (mayZip [] xb)
mayZip (a : xa) []       = (Just a, Nothing) : (mayZip xa [])
mayZip []       []       = []
```

Řeš. 4.2.6

```
summarize :: StudentResult -> String
summarize (StudentResult name course res) =
        unwords [name, "has", showRes res, "from", course]
  where
    showRes (Just res) = show res
    showRes _          = "no result"
```

Řeš. 4.3.1

a) `Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero)), ...`

b) Zajistí, že kompilátor deklaruje `Nat` jako instanci typové třídy `Show` (tj. typové třídy poskytující funkci `show`, která umožní převést hodnotu typu na jeho řetězcovou interpretaci) a na základě definice datového typu `Nat` automaticky definuje intuitivním způsobem funkci `show`, tj. např. `show (Succ (Succ Zero))` $\leadsto^*$ `"Succ (Succ Zero)"`.

c)
```
natToInt :: Nat -> Int
natToInt Zero     = 0
natToInt (Succ x) = 1 + natToInt x
```

d) Takováto hodnota má tvar `Succ (Succ (Succ (Succ ...)))`. Lze se tedy inspirovat například funkcí `repeat`:

```
natInfinity :: Nat
natInfinity = Succ natInfinity
```

Řeš. 4.3.2

a) `Con 3.14`

b)
```
eval :: Expr -> Double
eval (Con x)   = x
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y
eval (Mul x y) = eval x * eval y
eval (Div x y) = eval x / eval y
```

c) Pro získání hodnoty z výrazu tvaru `Just x` použijeme funkci `fromJust` a pro zjištění, zda hodnota existuje funkci `isJust`. Obě naleznete v modulu `Data.Maybe`, nebo si je můžete definovat sami:

```
fromJust :: Maybe a -> a
fromJust (Just x) = x

isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust _        = False
```

Pozor, mezi použitím `isJust x` a `x /= Nothing` je podstatný rozdíl v tom, že to druhé vyžaduje, aby typ zabalený uvnitř `Maybe` hodnoty byl porovnatelný.
Nyní již samotné řešení.

```
import Data.Maybe

evalMay :: Expr -> Maybe Double
evalMay (Con x) = Just x
evalMay (Add x y) = if isJust evx && isJust evy
    then Just (fromJust evx + fromJust evy)
    else Nothing
    where
        evx = evalMay x
        evy = evalMay y

-- vyhodnocovani pro konstruktory Sub a Mul je uplne analogicke
-- vyhodnocovani Add, proto ho neuvadime

evalMay (Div x y) = if isJust evx && isJust evy
                        && fromJust evy /= 0
    then Just (fromJust evx / fromJust evy)
    else Nothing
    where
        evx = evalMay x
        evy = evalMay y
```

Tato úloha se dá o mnoho jednodušeji řešit s použitím tzv. monád (či alespoň Applicative). O nich se něco můžete dozvědět například na navazujících předmětech IB016 a IA014.

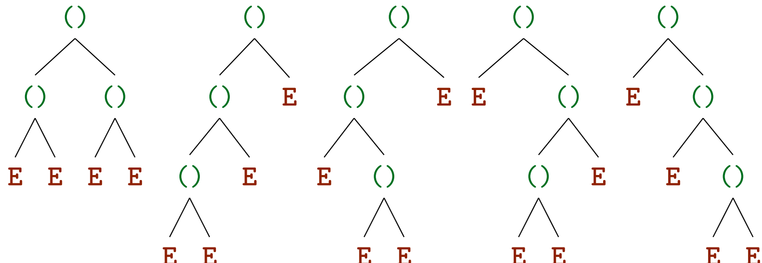
Řeš. 4.3.3

```
data Expr = Con Double     | Var
          | Add Expr Expr | Sub Expr Expr
```

```
          | Mul Expr Expr | Div Expr Expr

eval' :: Double -> Expr -> Double
eval' _ (Con x)   = x
eval' v Var       = v
eval' v (Add x y) = eval' v x + eval' v y
eval' v (Sub x y) = eval' v x - eval' v y
eval' v (Mul x y) = eval' v x * eval' v y
eval' v (Div x y) = eval' v x / eval' v y
```

Řeš. 4.3.4 a) Jsou to tyto stromy:



```
tree1 = Node () (Node () Empty Empty) (Node () Empty Empty)
tree2 = Node () (Node () (Node () Empty Empty) Empty) Empty
tree3 = Node () (Node () Empty (Node () Empty Empty)) Empty
tree4 = Node () Empty (Node () (Node () Empty Empty) Empty)
tree5 = Node () Empty (Node () Empty (Node () Empty Empty))
```

b) Nechť $\#_{()}(n)$ je počet stromů typu `BinTree ()`. Pak lze nahlédnout, že

$$\#_{()}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} \#_{()}(i)\#_{()}(n-i-1) & \text{if } n > 0 \end{cases}$$

Pokud bychom chtěli znát konkrétní hodnoty, můžeme formuli přepsat

```
count 0 = 1
count n = sum $ map (\i -> count i * count (n - i - 1))
                    [0 .. n - 1]
```

čímž lehce zjistíme, že `map count [0..5]` $\rightsquigarrow$ `[1,1,2,5,14,42]`.

c) Pro `BinTree Bool` platí

$$\#_{\texttt{Bool}}(n) = 2^n \#_{()}(n)$$

Obecně pro `BinTree` t máme:

$$\#_t(n) = |t|^n \#_{()}(n),$$

kde $|t|$ je počet různých hodnot typu t. Hledané hodnoty $\#_{\texttt{Bool}}(n)$ jsou:

```
countT t n = t ^ n * count n
map (countT 2) [0, 1, 2, 3, 4, 5] ⇝* [1, 2, 8, 40, 224, 1344]
```

Řeš. 4.3.5 a)

```
treeSize :: BinTree a -> Int
treeSize Empty          = 0
treeSize (Node _ t1 t2) = 1 + treeSize t1 + treeSize t2
```

b)

```
listTree :: BinTree a -> [a]
listTree Empty          = []
listTree (Node v t1 t2) = listTree t1 ++ [v] ++ listTree t2
```

c)
```haskell
height :: BinTree a -> Int
height Empty          = 0
height (Node _ l r)  = 1 + max (height l) (height r)
```

d)
```haskell
longestPath :: BinTree a -> [a]
longestPath Empty           = []
longestPath (Node v t1 t2) = if length p1 > length p2
    then v : p1
    else v : p2
    where
        p1 = longestPath t1
        p2 = longestPath t2
```

Řeš. 4.3.6 a)
```haskell
fullTree :: Int -> a -> BinTree a
fullTree 0 _ = Empty
fullTree n v = Node v (fullTree (n - 1) v) (fullTree (n - 1) v)
```

b)
```haskell
treeZip :: BinTree a -> BinTree b -> BinTree (a, b)
treeZip (Node x1 l1 r1) (Node x2 l2 r2) =
    Node (x1, x2) (treeZip l1 l2) (treeZip r1 r2)
treeZip _ _ = Empty
```

Řeš. 4.3.7
```haskell
treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)
treeMayZip (Node a l1 r1) (Node b l2 r2) =
    Node (Just a, Just b) (treeMayZip l1 l2) (treeMayZip r1 r2)
treeMayZip (Node a l1 r1) Empty =
    Node (Just a, Nothing) (treeMayZip l1 Empty) (treeMayZip r1 Empty)
treeMayZip Empty (Node b l2 r2) =
    Node (Nothing, Just b) (treeMayZip Empty l2) (treeMayZip Empty r2)
treeMayZip Empty Empty = Empty
```

Řeš. 4.3.8 a)
```haskell
-- Rozšíření množiny hodnot typu `a` o nekonečno
data MayInf a = Inf | NegInf | Fin a
                deriving (Eq)

instance (Ord a) => Ord (MayInf a) where
    _        <= Inf      = True
    Inf      <= _        = False
    NegInf   <= _        = True
    _        <= NegInf   = False
    (Fin a) <= (Fin b) = a <= b

isTreeBST :: (Ord a) => BinTree a -> Bool
isTreeBST = isTreeBST' NegInf Inf

isTreeBST' :: (Ord a) => MayInf a -> MayInf a -> BinTree a -> Bool
isTreeBST' _   _   Empty         = True
isTreeBST' low high (Node v l r) = let v' = Fin v in
    low <= Fin v && Fin v <= high &&
    isTreeBST' v' high r &&
    isTreeBST' low v' l
```

```haskell
-- alternativní definice
isTreeBST2 :: (Ord a) => BinTree a -> Bool
isTreeBST2 = isAscendingList . inorder

inorder :: BinTree a -> [a]
inorder Empty        = []
inorder (Node v l r) = inorder l ++ [v] ++ inorder r

isAscendingList :: (Ord a) => [a] -> Bool
isAscendingList [] = True
isAscendingList l  = and $ zipWith (<=) l (tail l)
```

b)
```haskell
searchBST :: (Ord a) => a -> BinTree a -> Bool
searchBST _ Empty = False
searchBST k (Node v l r) = case compare k v of
    EQ -> True
    LT -> searchBST k l
    GT -> searchBST k r
```

Řeš. 4.4.1
```haskell
roseTreeSize :: RoseTree a -> Int
roseTreeSize (RoseNode _ subtrees) =
    1 + sum (map roseTreeSize subtrees)

roseTreeSum :: Num a => RoseTree a -> a
roseTreeSum (RoseNode v subtrees) =
    v + sum (map roseTreeSum subtrees)

roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap f (RoseNode v subtrees) =
    RoseNode (f v) (map (roseTreeMap f) subtrees)
```

Řeš. 4.4.2
```haskell
evalExpr :: LogicExpr -> Bool
evalExpr Pos             = True
evalExpr Neg             = False
evalExpr (And     x y) =       evalExpr x  && evalExpr y
evalExpr (Or      x y) =       evalExpr x  || evalExpr y
evalExpr (Implies x y) = not (evalExpr x) || evalExpr y
evalExpr (Equiv   x y) =       evalExpr x  == evalExpr y
```

Řeš. 4.4.3
```haskell
data Bit = O | I
           deriving Show

toBitString :: Int -> [Bit]
toBitString 0 = [O]
toBitString a = (if a `mod` 2 == 1 then I else O)
    : toBitString (a `div` 2)

fromBitString :: [Bit] -> Int
fromBitString (O : xs) = 2 * fromBitString xs
fromBitString (I : xs) = 1 + 2 * fromBitString xs
fromBitString []       = 0
```

```haskell
insert :: IntSet -> Int -> IntSet
insert set num = insert' set (toBitString num)

insert' :: IntSet -> [Bit] -> IntSet
insert' SetLeaf []          = SetNode True SetLeaf SetLeaf
insert' (SetNode _ l r) [] = SetNode True l r
insert' SetLeaf (O : bits) = SetNode False (insert' SetLeaf bits) SetLeaf
insert' SetLeaf (I : bits) = SetNode False SetLeaf (insert' SetLeaf bits)
insert' (SetNode end l r) (O : bits) = SetNode end (insert' l bits) r
insert' (SetNode end l r) (I : bits) = SetNode end l (insert' r bits)

find :: IntSet -> Int -> Bool
find set num = find' set (toBitString num)

find' :: IntSet -> [Bit] -> Bool
find' (SetNode True _ _) []     = True
find' _ []                      = False
find' SetLeaf _                 = False
find' (SetNode _ l _) (O : xs) = find' l xs
find' (SetNode _ _ r) (I : xs) = find' r xs

listSet :: IntSet -> [Int]
listSet set = listSet' set []

listSet' :: IntSet -> [Bit] -> [Int]
listSet' SetLeaf _                   = []
listSet' (SetNode False l r) bits = listSet' l (O : bits)
    ++ listSet' r (I : bits)
listSet' (SetNode True l r) bits  = fromBitString (reverse bits)
    : listSet' l (O : bits) ++ listSet' r (I : bits)
```

Řeš. 4.4.4

```haskell
data SeqSet a = SeqNode Bool [(a, SeqSet a)]
                deriving Show

son :: (Eq a) => a -> [(a, SeqSet a)] -> SeqSet a
son a anc = getSon $ lookup a anc
    where
        getSon Nothing     = SeqNode False []
        getSon (Just node) = node

insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a
insertSeq (SeqNode _   anc) []        = SeqNode True anc
insertSeq (SeqNode end anc) (a : xa) = let s = son a anc in
    SeqNode end ((a, insertSeq s xa) : filter ((a/=) . fst) anc)

findSeq :: Eq a => SeqSet a -> [a] -> Bool
findSeq (SeqNode end _  ) []        = end
findSeq (SeqNode _   anc) (a : xa) = findSeq (son a anc) xa
```

Cvičení 5: Intensionální seznamy, lenost, foldy

Řeš. 5.η.1

```
divisors :: Integer -> [Integer]
divisors n = [ x | x <- [1 .. n], mod n x == 0 ]
```

Řeš. 5.η.4

a)
```
countPassed :: [(UCO, [String])] -> [(UCO, Int)]
countPassed m = [ (uco, length passed) | (uco, passed) <- m ]
```

b)
```
atLeastTwo :: [(UCO, [String])] -> [UCO]
atLeastTwo m = [ uco | (uco, passed) <- m, length passed >= 2 ]
```

Nebo alternativně s využitím vzorů:

```
atLeastTwo' :: [(UCO, [String])] -> [UCO]
atLeastTwo' m = [ uco | (uco, x:y:_) <- m ]
```

c)
```
passedIB015 :: [(UCO, [String])] -> [UCO]
passedIB015 m = [ uco | (uco, passed) <- m, elem "IB015" passed ]
```

d)
```
passedBySomeone :: [(UCO, [String])] -> [String]
passedBySomeone m = [ code | (_, passed) <- m, code <- passed ]
```

Řeš. 5.η.5 Funkci `naturalsFrom` jde definovat pomocí jednoduché rekurzivní definice, která nebude mít žádný bázový případ.

```
naturalsFrom :: Integer -> [Integer]
naturalsFrom n = n : naturalsFrom (n + 1)

naturals :: [Integer]
naturals = naturalsFrom 0
```

> **Pan Fešák upřesňuje:** Protože definici funkce `naturalsFrom` chybí bázový případ a její volání nikdy neskončí, tato definice není ve skutečnosti korektní rekurzivní definice v matematickém slova smyslu. Přesněji řečeno je tato definice *korekurzivní* (viz `https://en.wikipedia.org/wiki/Corecursion`).

Alternativně lze řešit pomocí enumeračního zápisu:

```
naturalsFrom' :: Integer -> [Integer]
naturalsFrom' n = [n..]
```

Řeš. 5.η.6

```
maxmin :: Ord a => [a] -> (a, a)
maxmin (x:xs) = maxmin' (x, x) xs
  where
    maxmin' res [] = res
    maxmin' (ma, mi) (a:as) = maxmin' (ma `max` a, mi `min` a) as
```

Řeš. 5.1.1 Elegantní možností je využít vzory s konkrétními hodnotami pro pohlaví:

```
allPairs ppl = [ (m, f) | (m, Male) <- ppl, (f, Female) <- ppl ]
```

Toto řešení využívá toho, že položky generátory (výrazu `... <- [...]`), které nesplňují vzor se automaticky vynechají. Do proměnné `m` se tak dosazují pouze ty hodnoty, které jsou ve dvojici s `Male`, zatímco do `f` se dosazují pouze ty, které jsou ve dvojici s `Female`.

Pokud bychom navíc měli instanci `Eq Sex`, mohli bychom využít i následujících řešení:

```
allPairs' people = [ (m, f) | (m, isM) <- people, isM == Male,
```

```haskell
                                    (f, isF) <- people, isF == Female ]
```

Jiným funkčním řešením by bylo například:

```haskell
allPairs'' people = [ (m, f) | (m, isM) <- people, (f, isF) <- people,
                                isM == Male, isF == Female]
```

To je ale zbytečně neefektivní: ke každému `m` se postupně zkusí všechna možná `f` i tehdy, když `m` není muž a proto stejně takové přiřazení vzápětí zahodíme. Kvalifikátory tedy chceme mít co nejvíce vlevo to jde, aby se nevyhovující přiřazení vyloučila co nejdříve. Vzájemné pořadí generátorů pak ovlivňuje, zda jsou ve výsledném seznamu shlukovány páry podle mužů, nebo žen.

Řeš. 5.1.2

a) `[ x^2 | x <- [1 .. k] ]`

b)
```haskell
f :: [[a]] -> [[a]]
f s = [ t | t <- s, length t > 3 ]
```

c) `[ '*' | _ <- [1 .. 5] ]`

d) `[ ['*' | _ <- [1 .. n]] | n <- [0 ..] ]`

e) `[ [1 .. n] | n <- [1 ..] ]`

Řeš. 5.1.3

a) `[ f x | x <- s ]`

b) `[ x | x <- s, p x ]`

c) `[ f x | x <- s, p x ]`

d) `[ x | _ <- [1 ..] ]`

e) `[ x | _ <- [1 .. n] ]`

f) `[ x | t <- s, let x = f t, p x ]`
případně `[ f x | x <- s, p (f x) ]`, ale toto řešení je méně efektivní

Řeš. 5.1.4

a)
```haskell
perm :: Eq a => [a] -> [[a]]
perm [] = [[]]
perm s  = [m : n | m <- s, n <- perm (filter (m /=) s)]
```

b)
```haskell
varrep :: Int -> [a] -> [[a]]
varrep 0 s = [[]]
varrep k s = [m : n | m <- s, n <- varrep (k - 1) s]
```

c)
```haskell
comb :: Int -> [a] -> [[a]]
comb 0 _   = [[]]
comb k xs0 =
    [m : t | (m, n) <- zip xs0 . tails . tail $ xs0, t <- comb (k - 1)
    ↪ n]
       where
           tails []       = [[]]
           tails (x : xs) = (x : xs) : tails xs
```

Tady lze případně použít funkci `tails` z modulu `Data.List`, viz `https://haskell.fi.muni.cz/doc/base/Data-List.html#v:tails`.

Řeš. 5.1.5 Nechť $n =$ `length s`. Lepší časovou složitost má funkce `f2`, protože projde seznamem jenom jednou, tedy má lineární časovou složitost. Na druhé straně `f1` vykoná nejvíce $n/2 + 1$ volání funkce `(!!)`. Tato volání se v tomto případě vykonají každé v lineárním čase vzhledem k druhému argumentu funkce `(!!)`. Dohromady tedy vyhodnocení funkce `f1` vyžaduje kvadratický čas.

Řeš. 5.2.1

```
naturals !! 2
    ⇝ naturalsFrom 0 !! 2
    ⇝ (0 : naturalsFrom (0 + 1)) !! 2
    ⇝ naturalsFrom (0 + 1) !! (2 - 1)
    ⇝ ((0 + 1) : naturalsFrom ((0 + 1) + 1)) !! (2 - 1)
    ⇝ ((0 + 1) : naturalsFrom ((0 + 1) + 1)) !! 1
    ⇝ naturalsFrom ((0 + 1) + 1) !! (1 - 1)
    ⇝ (((0 + 1) + 1) : naturalsFrom (((0 + 1) + 1) + 1)) !! (1 - 1)
    ⇝ (((0 + 1) + 1) : naturalsFrom (((0 + 1) + 1) + 1)) !! 0
    ⇝ (0 + 1) + 1
    ⇝ 1 + 1
    ⇝ 2
```

Nejdůležitější projev lenosti je vidět v předposledním řádku: zahození (a tedy nevyhodnocení) podvýrazu `naturalsFrom (((0 + 1) + 1) + 1)`, protože ho funkce `(!!)` nepotřebuje[12]. Při striktní strategii by jednak vypadalo jinak pořadí vyhodnocování podvýrazů, ale také by se vyhodnocovaly všechny. To by právě u zmíněného podvýrazu vedlo k nekonečnému výpočtu.

Řeš. 5.2.3

a) Funkce `f` při pokusu o vyhodnocení cyklí: `f` $\leadsto^*$ `f` $\leadsto^*$ `f` $\leadsto^*$ ... Avšak opět, funkce `fst` vybere z uvedené dvojice jenom první prvek. Tedy k vyhodnocení `f` nedojde a celý výraz bude vyhodnocen v konečném čase.

b) Funkce `f` je definována jen pro prázdný seznam, ale ve výrazu je volána na neprázdném seznamu. Normálně bychom tedy dostali chybovou zprávu `Non-exhaustive patterns in function f`. Ale protože funkce `const` nepoužívá svůj druhý argument, k vyhodnocení `f [1]` díky lenosti nikdy nedojde, a vyhodnocení tedy skončí bez chyby.

c) Výraz `div 2 0` sám o sobě vrátí chybu `divide by zero`. Může se zdát, že tady zafunguje líné vyhodnocování a `0 * div 2 0` se vyhodnotí na `0`, protože první argument je `0`. Obecně v tomto případě to však není pravda, protože u aritmetických operátorů vždy dochází k vyhodnocení obou operandů. Kvůli efektivitě totiž není vyhodnocování aritmetických operací definováno přímo v Haskellu, ale pomocí primitivních operací procesoru.

d) Při pokusu o vyhodnocení tohoto výrazu dostaneme typovou chybu. Je potřeba mít na paměti, že syntaktická a typová analýza výrazu předchází jeho vyhodnocování, a tedy líné vyhodnocování situaci nezachrání, protože k němu vůbec nedojde.

Řeš. 5.2.4

```
addNumbers :: [String] -> [String]
addNumbers = zipWith (\i n -> show i ++ ". " ++ n) [1 ..]
```

Řeš. 5.2.5

```
integers = 0 : [ sgn * n | n <- [1 ..], sgn <- [1, -1] ]
```

Při tvorbě nekonečných seznamů si musíme dát pozor na to, aby byl každý prvek dosažitelný na konečné pozici. Například řešení `[0 ..] ++ [-1, -2 ..]` toto nesplňuje – všechna záporná čísla se nachází až za nekonečným počtem kladných. V intensionálních seznamech tento problém nastává, pokud se nekonečný generátor objeví na jiné než první pozici. Uvažte třeba nesprávné řešení: `[ sgn * n | sgn <- [1, -1], n <- [1 ..] ]`. Při vyhodnocování se nejdříve zafixuje hodnota `sgn = 1` a následně probíhá nekonečně mnoho dosazení do `n`, takže na volbu `sgn = -1` nikdy nedojde.

[12] Přesněji řečeno, formální parametr `xs`, na který se výraz naváže, se nevyskytuje na pravé straně použité definice funkce `(!!)`.

Řeš. 5.2.6

```
threeSum = [ (x, y, z) | z <- [2 ..], x <- [1 .. z - 1],
                         let y = z - x ]
```

Jak bylo nastíněno v řešení 5.2.5, nekonečný generátor se nesmí objevit na jiné než první pozici, natož aby jich nekonečných bylo více. Nemůžeme tedy napsat třeba `[ (x, y, z) | x <- [1 ..], y <- [1 ..], let z = x + y ]`, protože bychom dostali jen nekonečně mnoho trojic tvaru `(1, y, 1 + y)`. Je proto zapotřebí jít do nekonečna po nějaké vhodné vlastnosti, kterou má vždy jen konečně mnoho prvků. Zde se přímo nabízí součet prvních dvou prvků – pro jeden konkrétní součet snadno vygenerujeme všechny vyhovující trojice, kterých je konečný počet.

Řeš. 5.2.7 Označme počty kroků při líném, normálním a striktním vyhodnocování popořadě L, N a S.

a) $L \leq S$. Nejdříve vyšetříme nekonečné výpočty:
   - $L = \infty$, tedy líná strategie vede k zacyklení. Potom dle věty o perpetualitě také striktní strategie vede k zacyklení a $L = S = \infty$.
   - $L \neq \infty$ a $S = \infty$, zřejmě $L \leq S$.
   - $L \neq \infty$ a $S \neq \infty$. Striktní strategie vynutí vyhodnocení každého podvýrazu právě jednou, kdežto líná nejvýše jednou, obě přitom bez ohledu na počet použití výsledku výrazu (Např. u `f x = x + x` bude výraz dosazený za `x` vyhodnocen pouze jednou). Opět tedy $L \leq S$.

b) Mezi N a S obecně vztah není. První dva případy z předchozí argumentace projdou stejně; ukážeme však, že může nastat $N > S$. Použijeme opět funkci `f x = x + x`. Odkrokujme si vyhodnocení výrazu `f (1 + 2)` oběma strategiemi:
   - Striktní: `f (1 + 2)` ⇝ `f 3` ⇝ `3 + 3` ⇝ `6`
   - Norm.: `f (1 + 2)` ⇝ `(1 + 2) + (1 + 2)` ⇝ `3 + (1 + 2)` ⇝ `3 + 3` ⇝ `6`

Řeš. 5.2.9

a)
```
repeat True
cycle [True]
iterate id True
```

b) `iterate (2 *) 1`

c) `iterate (9 *) 1`

d) `iterate (9 *) 3`

e)
```
iterate ((-1) *) 1
iterate negate 1
cycle [1, -1]
```

f)
```
iterate ('*' :) ""
iterate ("*" ++) ""
```

g)
```
iterate (\x -> (mod (x + 1) 4)) 1
cycle [1, 2, 3, 0]
```

Řeš. 5.2.10 Chceme-li něco provést pro každé dva sousední prvky seznamu, použijeme „zipování s vlastním ocasem“:

```
differences :: [Integer] -> [Integer]
differences xs = zipWith (-) (tail xs) xs
```

Řeš. 5.2.11

```
values :: (Integer -> a) -> [a]
values f = map f naturals
```

a) První derivaci (diskrétní) funkce `f`.

b) Druhé derivaci (diskrétní) funkce `f`.
   Necháme se zmást druhou derivací. Tu můžeme použít k vyšetření extrémů jen

u spojitých funkcí. První derivace nám ale pomůže; hledáme body, do nichž funkce klesá (a tedy první derivace je záporná) a z nichž roste (a tedy první derivace následujícího bodu je kladná).

```
localMinima :: (Integer -> Integer) -> [Integer]
localMinima f = map fst3 . filter lmin $ zip3 (tail fs) dfs (tail dfs)
    where
        fs  = values f
        dfs = differences fs
        lmin (_, din, dout) = din < 0 && dout > 0
        fst3 (x, _, _) = x
```

Řeš. 5.2.12 Existuje více řešení. Označíme je postupně `fib`N.

```
-- standardni, ale neefektivni definice
fib1 :: Integer -> Integer
fib1 0 = 0
fib1 1 = 1
fib1 n = fib1 (n - 1) + fib1 (n - 2)

-- kompaktnejsi zapis fib1
fib2 :: Integer -> Integer
fib2 n = if n == 0 || n == 1 then n else fib2 (n - 1) + fib2 (n - 2)

-- efektivni seznamova definice
fib3 :: [Integer]
fib3 = fib' (0, 1)
    where
        fib' (x, y) = x : fib' (y, x + y)

-- efektivni definice funkce s akumulacnim parametrem, odvozena z fib3
fib4 :: Integer -> Integer
fib4 n0 = fib' n0 (0, 1)
    where
        fib' 0 (x, y) = x
        fib' n (x, y) = fib' (n - 1) (y, x + y)
```

Různá další řešení lze nalézt na stránce http://www.haskell.org/haskellwiki/The_Fibonacci_sequence.

Řeš. 5.2.13

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Řeš. 5.2.14

```
treeTrim :: BinTree a -> Integer -> BinTree a
treeTrim Empty _ = Empty
treeTrim (Node v l r) 0 = Node v Empty Empty
treeTrim (Node v l r) n =
    Node v (treeTrim l (n-1)) (treeTrim r (n-1))
```

a)
```
treeRepeat :: a -> BinTree a
treeRepeat x = Node x (treeRepeat x) (treeRepeat x)
```

b)
```
treeIterate :: (a -> a) -> (a -> a) -> a -> BinTree a
treeIterate f g x =
```

```
    Node x (treeIterate f g (f x)) (treeIterate f g (g x))
```

c)
```
depthTree :: BinTree Integer
depthTree = treeIterate (+1) (+1) 0
```

Řeš. 5.2.15
```
infFinTree = Node "strom" infFinTree (Node "konec" Empty Empty)
```

Řeš. 5.2.16
```
nonNegativePairs = [ (x, y) | z <- [2 ..], x <- [1 .. z - 1],
                              let y = z - x ]
```

Řešení je prakticky totožné jako 5.2.6, jen se přes součet iteruje „skrytě“.

Řeš. 5.2.17 Je potřeba zvolit vhodnou vlastnost, přes niž se dá iterovat do nekonečna a má ji vždy konečný počet prvků. Vhodnou touto je díky kladnosti prvků součet seznamu. Všechny seznamy s daným součtem vyrobíme pomocí rekurze a intensionálního seznamu.

```
positiveLists = [ l | s <- [0 ..], l <- listsOfSum s ]
    where
        listsOfSum :: Integer -> [[Integer]]
        listsOfSum 0 = [[]]
        listsOfSum n = [x : l | x <- [1 .. n], l <- listsOfSum (n - x)]
```

Řeš. 5.3.1

a)
```
product' :: Num a => [a] -> a
product' []       = 1
product' (x : s) = x * product' s
```

b)
```
length' :: [a] -> Int
length' []       = 0
length' (_ : s) = 1 + length' s
```

c)
```
map' :: (a -> b) -> [a] -> [b]
map' _ []       = []
map' f (x : s) = f x : map' f s
```

Vždy jde o definici, která vrací určitou hodnotu na prázdném seznamu. V případě neprázdného seznamu se výsledek nějakým způsobem získá z prvního prvku seznamu a výsledku rekurzivního volání definované funkce na zbytku seznamu.

Funkcionalitu těchto tří funkcí lze tedy abstrahovat na funkci, která dostane jako jeden argument hodnotu vracenou na prázdném seznamu a jako druhý argument funkci, která se aplikuje na první prvek neprázdného seznamu a na výsledek volání požadované funkce na zbytku seznamu. Tedy:

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' _ z []       = z
foldr' f z (x : s) = f x (foldr' f z s)
```

Řeš. 5.3.2 Pro některé podúlohy je uvedeno více ekvivalentních řešení – tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

a)
```
sumFold :: Num a => [a] -> a
sumFold = foldr (+) 0

sumFold' :: Num a => [a] -> a
sumFold' = foldr (\e t -> e + t) 0
```

b)
```
productFold :: Num a => [a] -> a
productFold = foldr (*) 1
```

c)
```
orFold :: [Bool] -> Bool
orFold = foldr (||) False
```

d)
```
lengthFold :: [a] -> Int
lengthFold = foldr (\_ t -> t + 1) 0
```

e)
```
maximumFold :: Ord a => [a] -> a
maximumFold = foldr1 max

maximumFold' :: Ord a => [a] -> a
maximumFold' list = foldr max (head list) list
```

Řeš. 5.3.3 Jedná se o funkce `sum`, `product`, `or`, `and`, `length`, `minimum` a `maximum`.

Řeš. 5.3.4

a)
```
foldr (-) 0 [1, 2, 3] ⇝* 1 - (2 - (3 - 0))
      ⇝* 2
```

b)
```
foldl (-) 0 [1, 2, 3] ⇝* ((0 - 1) - 2) - 3
      ⇝* -6
```

c)
```
foldr (&&) True (True : False : repeat True)
    ⇝ True && foldr (&&) True (False : repeat True)   {- z def. foldr -}
    ⇝ foldr (&&) True (False : repeat True)           {- z def. (&&) -}
    ⇝ False && foldr (&&) True (repeat True)          {- z def. foldr -}
    ⇝ False                                           {- z def. (&&) -}
```

d)
```
foldl (&&) True (True : False : repeat True)
    ⇝ foldl (&&) (True && True)
                 (False : repeat True)              {- z def. foldl -}
    ⇝ foldl (&&) ((True && True) && False)
                 (repeat True)                      {- z def. foldl -}
    ⇝ foldl (&&) ((True && True) && False)
                 (True : repeat True)               {- z def. repeat -}
    ⇝ foldl (&&) (((True && True) && False) && True)
                 (repeat True)                      {- z def. foldl -}
    ⇝ foldl (&&) (((True && True) && False) && True)
                 (True : repeat True)               {- z def. repeat -}
    ⇝ {- … neskončí -}
```

Všiměte si, že u `foldl` můžeme hodnotu vrátit jen v okamžiku, kdy dojdeme na prázdný seznam. Naopak chování `foldr` na nekonečném seznamu závisí na lenosti dodané funkce.

Řeš. 5.3.5 Jasným kandidátem na řešení je použití některé z akumulačních funkcí. Celkem máme na výběr ze čtyř: `foldr`, `foldl`, `foldr1`, `foldl1`. Připomeňme si, jak která z nich funguje:

```
foldr  (@) w [1,2,3,4] = 1 @ (2 @ (3 @ (4 @ w)))
foldr1 (@)   [1,2,3,4] = 1 @ (2 @ (3 @ 4))
foldl  (@) w [1,2,3,4] = (((w @ 1) @ 2) @ 3) @ 4
foldl1 (@)   [1,2,3,4] = ((1 @ 2) @ 3) @ 4
```

Na základě těchto příkladů vidíme, že jediným vhodným kandidátem na přirozenou definici funkce `subtractlist` je `foldl1`. Tedy ve výsledku dostaneme:

```
subtractlist :: [Integer] -> Integer
subtractlist = foldl1 (-)
```

Řeš. 5.3.6 a) Funkce `foldr` pracuje na seznamech a nahrazuje `(:)` za funkci, v tomto případě za `(.)`, a `[]` za `id`. Intuitivně musí jít o seznam funkcí, které budeme postupně skládat. Ve výsledku tedy vytvoříme složení funkcí v pořadí, v jakém jsou uvedeny v seznamu.

b) Zkusme nejprve otypovat funkci intuitivně. Pro zkrácení řekněme `f = foldr (.) id`.

- Z použití `foldr` plyne, že funkce bude očekávat jako 1. argument seznam.
- Pokud tento seznam bude prázdný, pak `foldr` vrátí `id`, dostáváme tedy zatím typ `[a] -> b -> b` (tento typ je zatím nejspíš příliš obecný, protože jsme nevzali v úvahu všechno chování `foldr`).
- Funkce, která je argumentem `foldr` dostává vždy jako první argument prvek ze seznamu a jako druhý argument výsledek rekurzivního zpracování volání `foldr` na celém zbytku seznamu (což je pro poslední prvek seznamu koncová hodnota, zde `id`). Tím pádem v seznamu musí být funkce, které půjde složit s funkcí `id`.
- Zároveň, tím, že funkce `f` všechny tyto funkce složí za sebe, tak je jich výsledek musí být stejného typu, jako jejich vstup – tak, aby na sebe mohli navázat.
- Dostáváme tedy, že vstupní seznam bude typu `[a -> a]`.
- Složení takových funkcí je pak také typu `a -> a`, což rovněž odpovídá typu `id`.
- Celkem tedy dostáváme `f :: [a -> a] -> a -> a`.

Alternativně můžeme funkci otypovat algoritmicky.

- Máme daný výraz `foldr (.) id`
- Zjistíme si typy všech funkcí:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (a -> b) -> (c -> a) -> c -> b
id    :: a -> a
```

- Přejmenujeme typové proměnné, aby měl každý výskyt každé funkce vlastní typové proměnné:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (d -> e) -> (f -> d) -> f -> e
id    :: c -> c
```

- Určíme typové rovnosti na základě aplikací:

```
(d -> e) -> (f -> d) -> f -> e = a -> b -> b
c -> c = b
```

- Rozepíšeme typové rovnosti do jednodušších:

```
d -> e = a
f -> d = b
f -> e = b
c -> c = b
```

- Vyjádříme si všechny proměnné pomocí co nejmenšího počtu proměnných:

```
d = e = f = c
b = c -> c
a = c -> c
```

- Zjistíme, jaký typ vlastně hledáme. Je to typový výraz odpovídající typu `foldr` s odstraněnými dvěma typovými argumenty, tedy ve výsledku `[a] -> b`. Dosadíme do něj vyjádření proměnných získaných v předchozím kroku a tím dostaneme výsledný typ:

```
foldr (.) id :: [a] -> b = [c -> c] -> c -> c
```

c) `foldr (.) id [(+ 4), (* 10), (42 ^)]`
d) `foldr (.) id [(+ 4), (* 10), (42 ^)] 1`

Řeš. 5.3.7

```
append' :: [a] -> [a] -> [a]
append' xs ys = foldr (:) ys xs
append'' :: [a] -> [a] -> [a]
append'' = flip (foldr (:))
```

Řeš. 5.3.8

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Řeš. 5.3.9 Pro některé podúlohy je uvedeno více ekvivalentních řešení – tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

a)
```
concatFold :: [[a]] -> [a]
concatFold = foldr (++) []
```

b)
```
listifyFold :: [a] -> [[a]]
listifyFold = foldr (\x s -> [x] : s) []
listifyFold':: [a] -> [[a]]
listifyFold' = foldr ((:) . (: [])) []
```

c)
```
nullFold :: [a] -> Bool
nullFold = foldr (\_ _ -> False) True
```

d)
```
composeFold :: [(a -> a)] -> a -> a
composeFold = foldr (.) id
composeFold' :: [(a -> a)] -> a -> a
composeFold' = flip (foldr id)
```

e)
```
idFold :: [a] -> [a]
idFold = foldr (:) []
idFold' :: [a] -> [a]
idFold' = foldr (\e t -> e : t) []
```

f)
```
mapFold :: (a -> b) -> [a] -> [b]
mapFold f = foldr (\e t -> f e : t) []
```

g)
```
headFold :: [a] -> a
headFold = foldr1 const
headFold':: [a] -> a
headFold' = foldr1 (\e t -> e)
```

h)
```
lastFold :: [a] -> a
lastFold = foldr1 (flip const)
lastFold':: [a] -> a
lastFold' = foldr1 (\e t -> t)
```

i)
```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold (x:xs) = foldl (\(ma, mi) e -> (e `max` ma, e `min` mi))
                          (x, x) xs
```

Poznámka: pokud bychom rozbalili definici `foldl`, je toto řešení v podstatě ekvivalentní tomu rekurzivnímu v 5.η.6.

j)
```
suffixFold :: [a] -> [[a]]
suffixFold = foldr (\e (x : xs) -> (e : x) : x : xs) [[]]
```

k)
```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr (\e t -> if p e then e : t else t) []
```

l)
```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold = foldr (\x (l, r) -> (x : r, l)) ([], [])
```

m)
```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold p = foldr (\e t -> if p e then e : t else []) []
```

n)
```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold p = foldl (\t e ->
    if null t && p e
        then []
        else t ++ [e]) []
dropWhileFold':: (a -> Bool) -> [a] -> [a]
dropWhileFold' p list = foldl (\t e ->
    if null (t []) && p e
        then id
        else t . (e :) ) id list []
```

Druhé uvedené řešení má lepší složitost.

Řeš. 5.3.10
```
foldl' f z s = foldr (flip f) z (reverse s)
```

Řeš. 5.3.11
```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x < y
    then x : y : ys
    else y : insert x ys

insert' :: Ord a => a -> [a] -> [a]
insert' x = foldr (\y (z : zs) ->
    if y > z
        then z : y : zs
        else y : z : zs) [x]
```

Poznamenejme, že první varianta je díky lenosti tím rychlejší, čím blíže začátku se má prvek vložit, kdežto druhá varianta vždy prochází a kopíruje celý seznam.

```
insertSort :: Ord a => [a] -> [a]
insertSort = foldr insert []
```

Řeš. 5.3.12 Funkce `foldr` přestane vyhodnocovat výraz po prvním nalezeném `True` (díky línému vyhodnocování funkce `(||)`), avšak `foldl` ho vždy projde celý. Tedy v případě nekonečného seznamu `foldr` skončí po prvním nalezeném `True`, ale `foldl` neskončí nikdy.

Řeš. 5.3.13 Odpověď jste měli hledat na internetu, ne tady.

Řeš. 5.3.14 Ne, není to možné. Funkce `f` by musela dokázat rozlišit, kdy je volána na prvku ze sudého místa a kdy z lichého. K dispozici má však pouze n-tý prvek a seznam vzniklý zpracováním

prvních $n - 1$ prvků.

Důkaz. Předpokládejme pro spor, že taková funkce `f` existuje.

Ukážeme nejdřív, že funkce `f` si nemůže „dělat poznámky“ ve svém výsledku na to, aby poznala, zda je na sudé či liché pozici, ale musí ve svém n-tém volání vracet přímo výsledek na prvních n prvcích. Toto tvrzení je přímým důsledkem toho, že potřebujeme po poslední aplikaci funkce `f` dostat rovnou hledaný výsledek (a že v okamžiku kdy aplikaci provádíme, nemáme jak poznat, zda je poslední).

Uvažme dále seznamy `as = [1, 2, 3]` a `bs = [1, 3]`. Pak v nějakém momentě dojde pro oba seznamy k volání `f [1] 3`. V případě seznamu `as` však má dojít k přidání `3` do výsledku, zatímco v případě `bs` ne. To však není možné – funkce `f` je čistá funkce, nemá tedy jak tyto dva případy rozlišit. □

Ve druhém případě to možné je:

```
f = \(b, s) x -> (not b, if b then s ++ [x] else s)
v = (True, [])
```

Teď již postupujeme zleva (`foldl`) a v hodnotě typu `Bool` si ukládáme, jestli je aktuální pozice sudá.

Řeš. 5.3.15

```
foldr f z s = foldr2 (\x y s -> f x (f y s)) (\x -> f x z) z s
```

Opačná definice (`foldr2` pomocí `foldr`) není možná. Pomocí `foldr2` je možné vybrat každý druhý prvek seznamu (`foldr2 (\x y s -> x : s) (: []) []`), což však pomocí `foldr` není možné (viz úloha 5.3.14).

Řeš. 5.4.1

```
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold _ e Empty        = e
treeFold n e (Node v l r) = n v (treeFold n e l) (treeFold n e r)
```

Řeš. 5.4.2

a)
```
treeSize :: BinTree a -> Int
treeSize = treeFold (\_ l r -> 1 + l + r) 0
```

b)
```
treeHeight :: BinTree a -> Int
treeHeight = treeFold (\_ l r -> 1 + max l r) 0
```

c)
```
treeList :: BinTree a -> [a]
treeList = treeFold (\v l r -> l ++ [v] ++ r) []
```

d)
```
treeConcat :: BinTree [a] -> [a]
treeConcat = treeFold (\v l r -> l ++ v ++ r) []
```

e)
```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax = treeFold (\v l r -> maximum [v, l, r]) minBound
```

f)
```
treeFlip :: BinTree a -> BinTree a
treeFlip = treeFold (\v l r -> Node v r l) Empty
```

g)
```
treeId :: BinTree a -> BinTree a
treeId = treeFold (\v l r -> Node v l r) Empty
treeId' = treeFold Node Empty
```

h)
```
rightMostBranch :: BinTree a -> [a]
rightMostBranch = treeFold (\v l r -> v:r) []
```

i)
```
treeRoot :: BinTree a -> a
treeRoot = treeFold (\v l r -> v) undefined
treeRoot' = treeFold (const . const) undefined
```

j)
```
treeNull :: BinTree a -> Bool
treeNull = treeFold (\v l r -> False) True
```

k)
```
leavesCount :: BinTree a -> Int
leavesCount = treeFold (\v l r -> if l + r == 0 then 1 else l + r) 0
```

l)
```
leavesList :: BinTree a -> [a]
leavesList = treeFold (\v l r ->
    if null l && null r
        then [v]
        else l ++ r) []
```

m)
```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap f = treeFold (\v l r -> Node (f v) l r) Empty
treeMap' f = treeFold (\v -> Node (f v)) Empty
treeMap'' f = treeFold (Node . f) Empty
```

n)
```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny p = treeFold (\v l r -> p v || l || r) False
treeAny' p = treeFold (\v l r -> or [p v, l, r]) False
```

o)
```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair = treeFold (\(x,y) l r -> x == y && l && r) True
```

p)
```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums = treeFold (\v l r -> Node (v + root l + root r) l r)
                       Empty
    where
        root (Node v l r) = v
        root Empty = 0
```

Řeš. 5.4.3
```
roseTreeFold :: (a -> [b] -> b) -> RoseTree a -> b
roseTreeFold n (RoseNode v ts) = n v (map (roseTreeFold n) ts)
```

Řeš. 5.4.4

a)
```
roseTreeSize :: RoseTree a -> Int
roseTreeSize = roseTreeFold (\_ xs -> 1 + sum xs)
```

b)
```
roseTreeHeight :: RoseTree a -> Int
roseTreeHeight = roseTreeFold (\_ xs -> 1 + maximum (0:xs))
```

Všimněte si přidání 0 proto aby maximum fungovalo i pro listy. To v tomto případě můžeme bezpečně udělat, pokud seznam prázdný nebude, nic to nerozbije. Víme totiž, že výška nemůže být záporná a že seznam je seznamem čísel.

c) Inorder průchod se u těchto stromů nedá dobře definovat, můžeme ale udělat třeba preorder – v seznamu je nejprve hodnota uzlu a následně hodnoty všech podstromů.

```
roseTreeList :: RoseTree a -> [a]
roseTreeList = roseTreeFold (\v xs -> v : concat xs)
```

d)
```
roseTreeConcat :: RoseTree [a] -> [a]
roseTreeConcat = roseTreeFold (\v xs -> v ++ concat xs)
```

e)
```
roseTreeMax :: (Ord a, Bounded a) => RoseTree a -> a
roseTreeMax = roseTreeFold (\v xs -> maximum (v : xs))
```

f)
```
roseTreeFlip :: RoseTree a -> RoseTree a
roseTreeFlip = roseTreeFold (\v xs -> RoseNode v (reverse xs))
```

g)
```
roseTreeId :: RoseTree a -> RoseTree a
roseTreeId = roseTreeFold (\v xs -> RoseNode v xs)
```

```
roseTreeId' = roseTreeFold RoseNode
```

h)
```
roseRightMostBranch :: RoseTree a -> [a]
roseRightMostBranch = roseTreeFold (\v xs -> v :
    if null xs then [] else last xs)
```

i)
```
roseTreeRoot :: RoseTree a -> a
roseTreeRoot = roseTreeFold (\v _ -> v)
roseTreeRoot' = roseTreeFold const
```

j) Datový typ neumožňuje reprezentovat prázdné stromy.

```
roseTreeNull :: RoseTree a -> Bool
roseTreeNull = roseTreeFold (\_ _ -> False)
roseTreeNull' = roseTreeFold (const . const False)
```

k)
```
roseLeavesCount :: RoseTree a -> Int
roseLeavesCount = roseTreeFold (\_ xs ->
    if null xs then 1 else sum xs)
```

l)
```
roseLeavesList :: RoseTree a -> [a]
roseLeavesList = roseTreeFold (\v xs ->
    if null xs then [v] else concat xs)
```

m)
```
roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap f = roseTreeFold (\v xs -> RoseNode (f v) xs)
roseTreeMap' f = roseTreeFold (\v -> RoseNode (f v))
roseTreeMap'' f = roseTreeFold (RoseNode . f)
```

n)
```
roseTreeAny :: (a -> Bool) -> RoseTree a -> Bool
roseTreeAny p = roseTreeFold (\v xs -> p v || or xs)
roseTreeAny' p = roseTreeFold (\v xs -> or (p v : xs))
```

o)
```
roseTreePair :: Eq a => RoseTree (a, a) -> Bool
roseTreePair = roseTreeFold (\(x, y) xs -> x == y && and xs)
roseTreePair' :: Eq a => RoseTree (a, a) -> Bool
roseTreePair' = roseTreeFold (\(x, y) xs -> and ((x == y) : xs))
```

p)
```
roseSubtreeSums :: Num a => RoseTree a -> RoseTree a
roseSubtreeSums = roseTreeFold (\v xs -> RoseNode
                                           (sum (v : map root xs))
                                           xs)
    where root (RoseNode v _) = v
```

Řeš. 5.4.5 Nejprve je třeba promyslet si, jak by taková funkce intuitivně měla fungovat. Katamorfismus je obecně funkce na struktuře, která nahrazuje konstruktory této struktury zadanými funkcemi nebo hodnotami, a ve výsledku umožní rekurzivně projít celou strukturu. Datový typ `Nat` má konstruktory `Zero :: Nat` a `Succ :: Nat -> Nat`. Naším cílem je převod hodnoty tohoto typu na nějakou hodnotu, obecně typu `a`. Katamorfismus na hodnotách daného typu je definován funkcemi, které nahrazují jeho hodnotové konstruktory funkcemi stejné arity, jejichž výsledná hodnota je typu `a`, a v místě, kde má konstruktor argument původního typu (v tomto případě tedy `Nat`), uvedeme `a`.

S těmito znalostmi se tedy podívejme na typ `Nat`. Hodnotový konstruktor `Zero` nahradíme nulární funkcí, tedy hodnotou typu `a`. Hodnotový konstruktor `Succ` nahradíme unární funkcí s typem `a -> a`. Když definujeme tuto transformaci jako funkci, musíme ji definovat po částech pro jednotlivé hodnotové konstruktory. V těle pak použijeme dodané funkce a rekurzivně voláme `natFold`:

```
natFold :: (a -> a) -> a -> Nat -> a
natFold s z Zero     = z
natFold s z (Succ x) = s (natFold s z x)
```

Pokud bychom fixovali parametry `s` a `z`, lze lépe vidět, jak katamorfismus na `Nat` pracuje:

```
natFoldsz :: Nat -> a
natFoldsz Zero     = z
natFoldsz (Succ x) = s (natFoldsz x)
```

Řeš. 5.4.6 a) Číslo n typu `Nat` vlastně znamená přičtení n jedniček k nule. Když nezačneme nulou, ale jiným číslem m, dostaneme přesně součet $n + m$. Budeme tedy foldovat jedno z čísel, konstruktory `Succ` necháme být a `Zero` nahradíme druhým číslem.

```
natAdd :: Nat -> Nat -> Nat
natAdd m n = natFold Succ m n
natAdd' = natFold Succ
```

b) Nula je sudá, konstruktor `Zero` se tedy má vyhodnotit na `True`. Přičtení jedničky vždy paritu změní, konstruktory `Succ` proto nahradíme funkcí invertující logickou hodnotu.

```
natEven :: Nat -> Bool
natEven = natFold not True
```

c) Součin m a n je jen n-násobné přičtení m k nule. Přičítat už umíme funkcí `natAdd` a chceme-li funkci provést n-krát, nahradíme jí všechny konstruktory `Succ` v n.

```
natMul :: Nat -> Nat -> Nat
natMul m n = natFold (natAdd m) Zero n
```

Řeš. 5.4.7 Implementace `Foldable` je možná pomocí funkcí `foldr` nebo `foldMap`. Funkce `foldr` v tomto případě vede na následující, poměrně složitou, definici.

```
instance Foldable BinTree where
    -- type cannot be specified in instance
    -- foldr :: (a -> b -> b) -> b -> BinTree a -> b
    foldr _ e Empty = e
    foldr f e (Node v l r) = f v (foldr f (foldr f e r) l)
```

Zatímco bázový případ je poměrně přímočarý, ten rekurzivní není – v tomto případě máme problém s tím, že nemáme k dispozici nic, čím bychom dokázali zkombinovat více hodnot typu `b`, vyjma funkce `f`, která ale ještě bere argument typu `a`. Musíme tedy porušit symetrii přístupu k levému a pravému podstromu a výsledek zpracování jednoho z nich dát jako bázový případ pro druhý.

Jednodušší alternativou je postup s využitím *monoidů*. *Monoid* je daný nosnou množinou, binární operací a jejím neutrálním prvkem. Je třeba aby množina byla uzavřená vzhledem k dané binární operaci a operace byla asociativní. V kontextu typové třídy `Monoid` je operace reprezentována operátorem `(<>)` a neutrální prvek hodnotou `mempty`.

Instanci `Foldable` můžeme implementovat pomocí funkce `foldMap`, která provede projekci všech prvků stromu do daného monoidu a výsledky pospojuje monoidovou operací `(<>)`.

```
instance Foldable BinTree where
    -- type cannot be specified in instance
    -- foldMap :: Monoid m => (a -> m) -> BinTree a -> m
    foldMap proj tree = go tree
      where
```

```
go Empty = mempty
go (Node val left right) = proj val <> go left <> go right
```

Cvičení 6: Manipulace s funkcemi, typy, opakování

Řeš. 6.η.1

a) `Bool`

b) `[a]`

c) `[String]` (případně ekvivalentně `[[Char]]`)

Z podvýrazu `"" :: String` víme, že v seznamu musí být řetězce.

d) Nelze otypovat, seznamy musí být homogenní (všechny hodnoty musí mít stejný typ, ale první je typu `Bool` a druhý `[a]`, což jsou neunifikovatelné typy).

e) `Fractional a => [a]`; tedy `a` může být libovolný typ schopný reprezentovat zlomky (popřípadě také `(Num a, Fractional a) => [a]`, ale `Fractional` implikuje `Num`).

Pozor, stále se jedná o seznam, jehož všechny prvky mají stejný typ. Jen dosud není řečeno, jaký to bude, jen že to musí být nějaký typ z třídy `Fractional`. Celá čísla lze samozřejmě reprezentovat i pomocí typů z třídy `Fractional`, např. `Double`.

Jednotlivé prvky seznamu mají typ `1 :: Num a => a`, `2 :: Num a => a` a `3.14 :: Fractional a => a`. Jelikož se mají společně nacházet v jednom seznamu, musíme je unifikovat. Při tom dostaneme kontext `(Num a, Fractional a) =>`, ten však můžeme zjednodušit na `Fractional a =>`, protože `Fractional` je podtřídou `Num`.

f) Nelze otypovat. Chybová hláška `No instance for (Num Bool) arising from the literal ‘1’` říká, že `Bool` není číslo (instance `Num`). Přesněji vzato, nelze otypovat s definicemi, které máme v `Prelude` (a je celkem dobrý důvod na to, aby tato instance neexistovala).

Typy jednotlivých prvků seznamu jsou `1 :: Num a => a`, `2 :: Num a => a` a `True :: Bool`. Jelikož však `Bool` není číslo, nejde tyto typy unifikovat, a tedy nemohou být společně v seznamu.

g) `(Ord a, Num a) => [a]` (obě části kontextu jsou nutné, mezi `Ord` a `Num` není žádný vztah implikace)

Podvýrazy mají typy `filter :: (a -> Bool) -> [a] -> [a]`, `(\x -> x > 5) :: (Ord n, Num n) => n -> Bool` a `[1, 2, 4, 8] :: Num m => [m]`. Dosadíme-li typy argumentů za odpovídající typy `filter` a provedeme unifikaci, odstraněním již použitých argumentů dostáváme typ `(Ord a, Num a) => [a]`.

h) `(Show a, Integral a) => a -> String`; `Show` je typová třída typů, které lze převést do textové reprezentace.

Popřípadě také `(Show a, Num a, Integral a) => a -> String`, ale `Integral` implikuje `Num`.

i) `(Num a, Read a) => String -> a`; `Read` je typová třída typů, jejichž hodnoty lze parsovat z textové reprezentace (typ výsledku funkce `read` se odvodí z kontextu použití).

j) `Integral a => a -> Double`; explicitní otypování je zde použito k vynucení konverze na konkrétní typ. `fromIntegral :: (Integral a, Num b) => a -> b` totiž umožňuje konverzi celého čísla na libovolný číselný typ.

Řeš. 6.η.2

a) Ze vzoru vidíme, že vstupem je dvojice, a z výrazu na pravé straně vidíme, že i návratový typ je dvojice. Zároveň typ první složky v argumentu musí být stejný jako typ druhé složky v návratovém typu a naopak. Celkově tedy `swap :: (a, b) -> (b, a)`.

b) První argument musí být typu `Bool` podle prvního řádku definice. Podle prvního řádku tedy dostaneme typ `Bool -> (a, b) -> (b, a)`, zatímco podle druhého `Bool -> (c, d) -> (c, d)`. Jelikož typy na odpovídajících pozicích musíme unifikovat (tedy `(a, b)` $\sim$ `(c, d)` a `(b, a)` $\sim$ `(c, d)`), jediná možnost je, že oba typy ve dvojici budou stejné. `swapIf :: Bool -> (a, a) -> (a, a)`.

c) Z toho, že v obou vzorech je právě jeden argument a funkce vrací `String`, vidíme, že nejobecnější možný typ funkce je `a -> String`. Typ argumentů funkce však není závislý jen na jejich použití na pravé straně definice, ale i na vzorech. Jelikož `[]` je vzor prázdného seznamu, musí být argument funkce seznamového typu. Další omezení již nejsou, dostáváme tedy `sayLength :: [a] -> String`.

d) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty a že první je typu `Char`. Z návratové hodnoty prvního řádku můžeme odvodit typ `Bool`. Zbývá už jen určení typu druhého argumentu. Ve druhém vzoru si můžeme všimnout, že vracíme hodnotu, kterou bereme ve druhém argumentu. Takže obě mají stejný typ, a protože návratová hodnota má typ `Bool`, i druhý argument bude mít typ `Bool`. Dostáváme tedy `aOrX :: Char -> Bool -> Bool`.

Řeš. 6.η.3 Máme dvě možnosti funkcí, které svůj argument ignorují, a dvě varianty pro funkce, které s argumentem nějak pracují – identitu a negaci.

```
bfalse :: Bool -> Bool
bfalse _ = False

btrue :: Bool -> Bool
btrue _ = True

bid :: Bool -> Bool
bid x = x

bnot :: Bool -> Bool
bnot True  = False
bnot False = True
```

Řeš. 6.η.5

```
binmap :: (a -> a -> b) -> [a] -> [b]
binmap f xs = zipWith f xs (tail xs)

binmap' :: (a -> a -> b) -> [a] -> [b]
binmap' _ []       = []
binmap' _ [_]      = []
binmap' f (x:y:xs) = f x y : binmap f (y:xs)
```

Řeš. 6.1.1

a) Arita 3. Po aplikaci na tři argumenty libovolného, ale stejného typu, který navíc musí být instancí typové třídy `Eq`, je výsledek typu `Bool`. Příkladem funkce, která může mít tento typ, je `\x y z -> x == y && y == z`.

b) Arita 2. Závorka vpravo (kolem `[a] -> Int`) nemá žádný efekt – tato závorka by v typu implicitně byla, i kdybychom ji vynechali a odpovídá částečné aplikaci. Prvním argumentem je funkce `a -> Bool`, druhým je pak seznam, jehož prvky mají stejný

typ jako argumenty funkce v prvním argumentu. Výsledek je pak `Int`. Například `(\p -> length . filter p) :: (a -> Bool) -> ([a] -> Int)`.

c) Arita 0. Nejedná se o funkci (je to seznam funkcí). Například `[(+), (-)] :: [Int -> Int -> Int]`.

d) Arita 3. Po aplikaci na 3 argumenty typů `Int`, `Integer` a `String` dostaneme výsledek typu `String`. Například `(\x y z -> show x ++ " " ++ show y ++ " " ++ z) :: Int -> Integer -> String -> String`.

e) Arita 2. První argument je funkce typu `Int -> Integer`, druhý je `String`, výsledek je `String`. Například `(\f x -> show (f (read x))) :: (Int -> Integer) -> String -> String`.

f) Arita 3. Tento typ je ekvivalentní typu v d) (uzávorkování funkčního typu zprava odpovídá částečné aplikaci funkce).

g) Arita 2. Bere dva argumenty (`Int` a `Integer`) a produkuje seznam funkcí (typu `[String -> String]`). Například `(\x y -> [ \s -> show x ++ show i ++ s | i <- [1..y] ]) :: Int -> Integer -> [String -> String]`.

h) Aritu nelze z typu určit, bude ale nejméně 3. Důvodem je, že na místě výsledku je v typu typová proměnná `c`, za kterou ale můžeme dosadit i funkční typ. Víme však, že funkce bude brát minimálně tři argumenty – funkci typu `a -> b -> c` a následně dva argumenty typů `b` a `a` (které opět mohou být i funkce).

Uvažme příklady konkrétních typů, které můžeme za jednotlivé argumenty dosadit.

- Za první argument dosadíme typ funkce `(||) :: Bool -> Bool -> Bool` – druhý i třetí argument pak musí být `Bool` a stejně tak i výsledek. Pro tuto konkretizaci dostáváme tedy aritu 3.
- Za první argument dosadíme typ funkce `zipWith :: (d -> e -> f) -> [d] -> [e] -> [f]`. Ten můžeme uzávorkovat jako `zipWith :: (d -> e -> f) -> [d] -> ([e] -> [f])` a položit `a` ≡ `d -> e -> f`, `b` ≡ `[d]` a `c` ≡ `[e] -> [f]`. Tím dostaneme konkretizaci našeho typu na `((d -> e -> f) -> [d] -> [e] -> [f]) -> [d] -> ((d -> e -> f) -> [d] -> [e] -> [f]) -> [e] -> [f]`, a tedy aritu 4.

Konkrétním (a jediným totálním) příkladem funkce s tímto typem je `flip`. Můžete si v interpretru ověřit, že počet argumentů této funkce nutných k tomu, aby výsledek nebyla funkce, je vskutku závislý na prvním argumentu. Uvážit můžeme třeba `flip (||) True False :: Bool` a `flip zipWith [1, 2, 3] (+) [3, 2, 1] :: Num a => [a]`.

Řeš. 6.1.2 a) `cm :: (a -> [b]) -> [a] -> [b]`
Při typování rekurzivních funkcí může být výhodné dívat se nejprve na bázový příklad. V tomto případě z něj však moc nezjistíme: vidíme, že má typ `a -> [b] -> [c]`, tedy víme jen, že druhý argument je seznam a návratová hodnota je taktéž seznam. Z rekurzivní části definice pak plyne, že typ návratové hodnoty celé funkce musí být stejný jako typ návratové hodnoty funkce `f` (plyne z typu `++`), a že funkce `f` musí brát jako argumenty hodnoty ze seznamu v druhém argumentu `cm`.
Funkce je podobná funkci `map`, ale z každého prvku původního seznamu vytvoří seznam prvků a tyto seznamy spojí. Najdeme ji i mezi základními funkcemi v Haskellu pod názvem `concatMap`.

b) `mm :: (Bounded a, Ord a) => [a] -> (a, a)`
Z bázového případu (prvního řádku) odvodíme, že výsledek bude dvojice. Dále z typů `minBound :: Bounded a => a` a `maxBound :: Bounded a => a` odvodíme, že obě složky výsledné dvojice budou instancemi typové třídy `Bounded` (ale zatím nic neříká, že to musí být stejné typy). Ze vzoru odvodíme, že argument bude seznam. Dohromady tedy z bázového případu dostáváme typ `(Bounded b, Bounded c) => [a] -> (b, c)`.
Z rekurzivního případu pak opět vidíme, že argument je seznam a výsledek dvojice – to je konzistentní s již zjištěným typem. Dále ale vidíme, že obě složky výsledné dvojice musí být stejného typu jako prvky vstupu a navíc tento typ musí být instancí `Ord` – obojí plyne z typu `max :: Ord a => a -> a -> a`, resp. `min` stejného typu. Z druhého řádku definice tedy plyne typ `Ord a => [a] -> (a, a)`.
Typy získané z jednotlivých řádků definice unifikujeme a při tom i sloučíme kontexty. Tím dostaneme výsledný typ.
Funkce `mm` počítá minimum a maximum z hodnot v seznamu a dělá to v jednom průchodu. Navíc funguje i na prázdném seznamu, kde jako minimum vrací nejvyšší hodnotu daného typu a jako maximum nejnižší – tyto hodnoty můžeme chápat jako neutrální prvky pro minimum (resp. maximum) pokud je typ omezený. Tato funkce tedy nefunguje s neomezenými typy jako je `Integer`.

c) `c :: (b -> c) -> (a -> b) -> a -> c`
Funkci můžeme ekvivalentně zapsat jako

```
c' f g x = f (g x)
```

Jedná se tedy o funkci, která bere dvě funkce a hodnotu libovolného typu a aplikuje funkce postupně na tuto hodnotu. Tedy o ekvivalent funkce `(.)`.
Můžeme postupovat i více algoritmicky:

1. můžeme začít s tím, že parametrům (včetně toho v lambda funkci) dáme libovolný typ.

```
x :: t
y :: u
z :: a
```

2. dále si všimneme, že `y` je aplikováno na `z`, a tedy se musí jednat o funkci, která bere argumenty typu `a`; výsledek je zatím neznámého typu

```
x :: t
y :: a -> b
z :: a
```

3. z toho tedy plyne, že `(y z) :: b`
4. dále pak `x` je aplikováno na `(y z)`, tedy to opět musí být funkce, tentokrát beroucí argument typu `b`; výsledek je opět zatím neznámého typu

```
x :: b -> c
y :: a -> b
z :: a
```

5. tedy `x (y z) :: c`;
6. dále můžeme otypovat lambda funkci na pravé straně definice funkce ze zadání: `(\z -> x (y z)) :: a -> c`, výsledek je samozřejmě typ výsledku na pravé straně lambda funkce, zatímco před `->` je typ argumentu;
7. nyní již zbývá přidat jen typy argumentů `x` a `y`: `c :: (b -> c) -> (a -> b) -> a -> c`.

Řeš. 6.1.4

a) Arita 2. Funkce `map` bere jako první argument funkci, druhý pak seznam (jehož prvky mají stejný typ jako vstup funkce). Funkce `flip` pak pouze prohodí pořadí argumentů, takže seznam bude první a funkce druhá. Produkuje seznam.

b) Arita 1. Vstupem je seznam, výstup je `Bool`.

c) Arita 3. Kromě dvou argumentů explicitně zapsaných lambda funkcí (typů `a -> b`, a `b -> Bool`) bere tato funkce ještě seznam (`[a]`).

d) Arita 1. Nejobecnější typ výrazu je `(\x -> x 1 || x 2) :: Num a => (a -> Bool) -> Bool`. Jediným vstupem je tedy funkce (predikát na číslech).

e) Arita 2. Argument `flip` je operátorovou sekcí od `(.)`, druhým argumentem bude funkce, prvním pak hodnota, která může vstoupit do této funkce. `flip (not .) :: a -> (a -> Bool) -> Bool`.

f) Arita 2. Oba argumenty jsou seznamy (to plyne již z typu `zipWith`), výsledkem je rovněž seznam. Jediná potíž by mohla být, pokud by tento výraz nešel otypovat. Zde je však třeba si uvědomit, že `id` je polymorfní funkce typu `a -> a` a za `a` lze tedy dosadit i funkční typ (třeba `b -> c`, a tak dostat specializaci na `(b -> c) -> b -> c`). Dostáváme tedy `zipWith id :: [b -> c] -> [b] -> [c]`, tedy funkci, která vezme seznam funkcí a seznam argumentů, a aplikuje funkce na argumenty.

Řeš. 6.1.5 Naše funkce bere dva argumenty – funkci (pojmenujeme `f`) a hodnotu (pojmenujeme `x`). Z typu funkce víme, `x` lze dát jako argument `f`. Musíme si rozmyslet co s nimi můžeme dělat pro to, abychom vrátili něco typu `b`.

Dále se můžeme zamyslet nad tím, co můžeme dělat s argumentem typu `a`. Jelikož však tento argument může být *libovolného* typu, nevíme nic o jeho vlastnostech a struktuře a nemáme ho tedy jak modifikovat.

Zároveň potřebujeme nějak vyrobit hodnotu typu `b`. Opět o tomto typu nic nevíme, a tedy nemáme jinou možnost jak ho vyrobit, než pomocí `f`.

Existuje tedy jen jedna totální funkce tohoto typu:

```
app :: (a -> b) -> a -> b
app f a = f a
```

Tato funkce je ekvivalentní standardnímu operátoru `($)`.

Řeš. 6.1.6 Žádná totální funkce tohoto typu neexistuje. Je třeba si uvědomit, že aby taková totální funkce existovala, musela by být schopná vytvořit hodnotu typu `a` z prázdného seznamu. To však není možné, máme-li neznámý typ `a` nemáme jak vytvořit jeho hodnotu.

Řeš. 6.1.7 Uvažme funkci `foo :: (a -> Maybe b) -> Maybe a -> Maybe b`. Tu pak můžeme volat jako `foo f x`, kde `f` je funkce typu `a -> Maybe b` a `x` je hodnota typu `Maybe a`. Uvažme jaká chování může funkce `foo` mít.

a) Nezávisle na argumentech může vrátit `Nothing`.

```
constNothing :: (a -> Maybe b) -> Maybe a -> Maybe b
constNothing _ _ = Nothing
```

b) Pokud chceme vyprodukovat nějakou jinou hodnotu než `Nothing`, máme k dispozici pouze dané argumenty `f` a `x`. Speciálně nemáme jak vytvořit hodnotu typu `b` jinak, než pomocí funkce `f :: a -> Maybe b` – důvodem je to, že `b` může být libovolný typ a nic o něm nevíme.

Pro použití funkce `f` potřebujeme hodnotu typu `a`. Tu můžeme získat jedině z hodnoty `x :: Maybe a`. Ačkoli o `a` nevíme nic, o hodnotě typu `Maybe a` již víme, že může být `Nothing` nebo `Just z` pro nějaké `z`.

Můžeme tedy použít vzory pro rozložení hodnoty `x` a pokud není `Nothing`, použít uvnitř uloženou hodnotu typu `a` jako vstup do funkce `f`.

```
bind :: (a -> Maybe b) -> Maybe a -> Maybe b
bind _ Nothing  = Nothing
bind f (Just x) = f x
```

c) Žádná další možnost není.

Řeš. 6.1.8 Prvně otypujeme řádky jednotlivě:

1. První argument musí jistě být číslo a porovnatelný, typ druhého argumentu musí být seznamový, protože se objevuje v `then` větvi `if`u, kde se v `else` větvi objevuje seznam, typ třetího argumentu je `Bool`. Návratová hodnota je seznam stejného typu jako druhý argument, protože můžeme vracet přímo druhý argument.

   Dostáváme tedy `(Num a, Ord a) => a -> [b] -> Bool -> [b]`

2. Z druhého řádku o prvním argumentu nevíme nic, o druhém víme, že je to seznam a že je stejného typu jako návratová hodnota. O třetím argumentu opět víme, že je to `Bool`

   `c -> [d] -> Bool -> [d]`

3. O argumentech nevíme nic, ale víme, že návratová hodnota je řetězec.

   `e -> f -> g -> String`

V těchto případech je vhodné nechat zatím typové proměnné v jednotlivých typech různé, abychom zabránili náhodnému propojení typů, které spolu nesouvisí.

Nyní zbývá unifikovat typy na pozicích, které si v definici funkce odpovídají.

- `a` $\sim$ `c` $\sim$ `e`, zde nesmíme zapomenout, že s `a` se pojí typový kontext. Unifikace je naopak jednoduchá, protože unifikuje jen samotné typové proměnné. Nadále budeme místo nich všech používat `a` (substituce `c` $\mapsto$ `a` a `e` $\mapsto$ `a`).
- `[b]` $\sim$ `[d]` $\sim$ `f` vyřešíme substitucí `d` $\mapsto$ `b` a `f` $\mapsto$ `[b]`.
- `Bool` $\sim$ `Bool` $\sim$ `g`, zde je substituce jednoduchá: `g` $\mapsto$ `Bool`.
- `[b]` $\sim$ `[d]` $\sim$ `String`, což už ale máme substituováno za `[b]` $\sim$ `[b]` $\sim$ `String` (protože `d` se nahradilo za `b`).

  Toto na první pohled nevypadá moc dobře, protože se zdánlivě snažíme unifikovat seznam s něčím, co není seznam. Avšak `String` je jen alias pro `[Char]`, a tedy unifikovat se seznamovými typy jej lze. Dostáváme `b` $\mapsto$ `Char`.

Nyní je třeba provést substituce v typech z jednotlivých řádků definice. Nemělo by záležet na tom, který řádek vezmeme, za předpokladu, že substituujeme, dokud můžeme. Celkově dostaneme `f :: (Num a, Ord a) => a -> [Char] -> Bool -> [Char]`, neboli `f :: (Num a, Ord a) => a -> String -> Bool -> String`. Je důležité nezapomenout na kontext svázaný s typovou proměnnou `a`.

Řeš. 6.1.9 a) Jedná se o funkci, která dokáže (pro typy, které to umožňují) převést hodnoty daného typu na jejich textovou reprezentaci. Např. `show 42` ⇝* `"42"`, `show [16, 42]` ⇝* `"[16,42]"`. Funguje pro většinu typů s výjimkou funkčních typů. Textová forma vyprodukované `show` by typicky měla být zápis validního Haskellového výrazu.

b) Jedná se o funkci, která převádí textovou reprezentaci na hodnotu požadovaného typu. Typ výsledné hodnoty se odvodí z použití funkce `read`, v případě potřeby je možné jej vynutit explicitním otypováním:

```
(read "42" :: Int) ⇝* 42
(read "42" :: Float) ⇝* 42.0
(read "[1, 2, 3, 4]" :: [Int]) ⇝* [1, 2, 3, 4]
read "40" + read "2" ⇝* 42
```

c) Funkce pro převod celého čísla na libovolnou reprezentaci čísla, např. funkci pro výpočet e-té odmocniny čísla n, kde e je celé číslo a n je číslo s plovoucí desetinnou čárkou (např. `Double`), lze zapsat jako:

```
root :: (Floating a, Integral b) => a -> b -> a
root n e = n ** (1 / fromIntegral e)
```

(Operátor `(**)` slouží k umocňování čísel s plovoucí desetinnou čárkou.)

d) Funkce pro zaokrouhlování desetinných čísel na celá čísla (existuje i `floor` a `ceiling`). Např. `round 1.6` ⇝* `2`, `floor 1.6` ⇝* `1`. (Typová třída `RealFrac` obsahuje právě čísla, která lze zaokrouhlovat, z běžných typů do ní patří `Float` a `Double`.)

Řeš. 6.2.1 a)
```
last [42, 3.14, 16] ⇝* 16.0 :: Fractional a => a
```

b)
```
zipWith mod [3, 5, 7, 4] [4, 2, 3]
    ⇝* [3, 1, 1] :: Integral a => [a]
```

c)
```
(concat . map (replicate 4)) ['a', 'b', 'c']
    ⇝* concat (map (replicate 4) ['a', 'b', 'c'])
    ⇝* "aaaabbbbcccc" :: String
```

d)
```
head (filter ((> 3) . length) ["hi!", "ahoj", "hello"])
    ⇝* head ["ahoj", "hello"]
    ⇝* "ahoj" :: String
```

e)
```
cycle [3, 2, 4] !! 10  ⇝* 2 :: Num a => a
```

f)
```
(head . drop 3 . iterate (^ 2)) 2
    ⇝* ((2 ^ 2) ^ 2) ^ 2
    ⇝* 256 :: Num a => a
```

Řeš. 6.2.2
```
pad :: Int -> Char -> PadMode -> String -> String
pad padTo padChar padMode str = leftPad padMode
                                ++ str
                                ++ rightPad padMode
  where
    diff = max 0 (padTo - length str)
    rdiff = diff `div` 2
    ldiff = diff - rdiff
```

```
mkPad n = replicate n padChar

leftPad PadLeft   = mkPad diff
leftPad PadCenter = mkPad ldiff
leftPad PadRight  = ""

rightPad PadLeft   = ""
rightPad PadCenter = mkPad rdiff
rightPad PadRight  = mkPad diff
```

Řeš. 6.2.3

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' 0 _  = []
take' amount (x:xs) = x : (take' (amount - 1) xs)
```

Řeš. 6.3.1

a) Např.

- `LLeaf 42 :: BinLeafTree Int`
- `LNode (LLeaf 3) (LLeaf 0.2) :: BinLeafTree Double`
- `LNode (LLeaf "hi") (LLeaf []) :: BinLeafTree String`

b) Typový konstruktor `BinLeafTree` arity 1. Hodnotové konstruktory `LLeaf` (arity 1) a `LNode` (arity 2).

c)
```
ltSumEven :: Integral a => BinLeafTree a -> a
ltSumEven (LLeaf x)    = if even x then x else 0
ltSumEven (LNode l r) = ltSumEven l + ltSumEven r
```

Řeš. 6.3.2

a) Typové: `RoseLeafTree` arity 1. Hodnotové: `RLNode` a `RLLeaf`, oba arity 1.

b)
```
countValueLeaves :: Integral i => RoseLeafTree a -> i
countValueLeaves (RLLeaf _)   = 1
countValueLeaves (RLNode ls) = sum (map countValueLeaves ls)
```

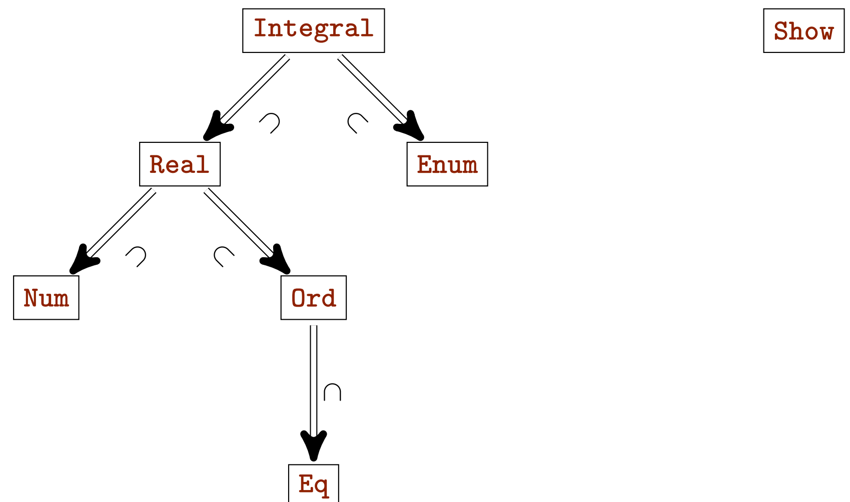
Čistě z hlediska Haskellu by nejobecnější možný typ této funkce měl kontext `Num a =>` místo `Integral a =>`, ale vzhledem k tomu, že výsledek má být počet, je vhodné, aby byl jeho typ celočíselný.

c) Je dobré si všimnout, že podmínka na prořezání stromu není závislá na tom, zda byl list odstraněn díky filtraci, nebo vůbec nebyl ve vstupu. Zároveň podle prvního příkladu se samotný list nahrazuje za uzel bez následníků. Těchto dvou vlastností můžeme využít a nejprve převést listy, které nechceme, na uzly bez následníků a následně tyto uzly prořezat. Ke zpracování seznamů v uzlech použijeme `map` a `filter`.

```
rlFilter :: (a -> Bool) -> RoseLeafTree a -> RoseLeafTree a
rlFilter p (RLLeaf x)   = if p x then RLLeaf x else RLNode []
rlFilter p (RLNode ls) =
    RLNode (filter nontrivial (map (rlFilter p) ls))
  where
    nontrivial (RLNode []) = False
    nontrivial _           = True
```

Řeš. 6.4.1 Vztahy lze vyčíst z hlavičky třídy zobrazené příkazem `:i`, např. `class (Real a, Enum a) => Integral a where` říká, že třídy `Read` a `Enum` jsou nadtřídami `Integral`. Jinak řečeno, každý typ, který je v `Integral`, musí být i v `Real` a `Enum`. Na třídy se také můžeme dívat jako na množiny typů – `Integral` je podmnožinou `Real`.

Kompletní vztahy mezi těmito třídami lze znázornit následujícím diagramem:



Je dobré si povšimnout následujícího:

- žádná z číselných tříd automaticky nevyžaduje zobrazitelnost pomocí `show`;
- obecně čísla (jiná než reálná) nemusí jít porovnávat/řadit – například komplexní čísla nemohou být instancí `Ord`; u některých reprezentací čísel (např. spočetná čísla[a] nebo čísla zadána pomocí výrazů, rovnic či funkcí) nemusí dávat dobrý smysl ani rovnost;
- všechny celočíselné typy jsou i instancemi `Enum`, což mimo jiné umožňuje používat zápisy jako `[1..42]`; dalším příkladem typu, který je instancí `Enum`, je `Bool`.

[a]Anglicky computable numbers, jsou čísla pro něž můžeme algoritmicky spočítat desetinný rozvoj do libovolné přesnosti.

Řeš. 6.4.2

```
import Data.Char

instance Show Entry where
    show (Word str) = show str

instance Eq Entry where
    (Word str1) == (Word str2) = lowStr str1 == lowStr str2
      where
        lowStr xs = map toLower xs
```

Řeš. 6.4.3

```
isEqual :: Shape -> Shape -> Bool
isEqual (Circle r1)       (Circle r2)       = r1 == r2
isEqual (Rectangle a1 b1) (Rectangle a2 b2) = a1 == a2 && b1 == b2
isEqual Point             Point             = True
isEqual _                 _                 = False

isGreater :: Shape -> Shape -> Bool
isGreater shape1 shape2 = area shape1 > area shape2
    where
        area (Circle r)      = pi * r * r
```

```haskell
    area (Rectangle a b) = a * b
    area Point           = 0
```

Řeš. 6.4.4

```haskell
instance Eq TrafficLight where
    Red == Red       = True
    Orange == Orange = True
    Green == Green   = True
    _ == _           = False

instance Ord TrafficLight where
    Green <= _       = True
    _ <= Red         = True
    Orange <= Orange = True
    _ <= _           = False

instance Show TrafficLight where
    show Red    = "červená"
    show Orange = "oranžová"
    show Green  = "zelená"
```

Řeš. 6.4.5

```haskell
instance (Eq a, Eq b) => Eq (PairT a b) where
    PairD a1 b1 == PairD a2 b2 = a1 == a2 && b1 == b2

instance (Ord a, Ord b) => Ord (PairT a b) where
    PairD a1 b1 <= PairD a2 b2 = a1 < a2 || (a1 == a2 && b1 <= b2)

instance (Show a, Show b) => Show (PairT a b) where
    show (PairD a b) = "pair of " ++ show a ++ " and " ++ show b
```

Řeš. 6.4.6

```haskell
instance Eq a => Eq (BinTree a) where
    Empty          == Empty          = True
    Node x1 l1 r1 == Node x2 l2 r2 =
        x1 == x2 && l1 == l2 && r1 == r2
    _              == _              = False
```

Poslední řádek nelze vynechat – pokrývá porovnávání prázdného a neprázdného stromu.

Cvičení 7: Vstup a výstup (bonus)

Řeš. 7.1.1

a)
```haskell
f  = getLine >>= \s -> putStrLn s
f' = getLine >>= putStrLn
```

b)
```haskell
f = getLine >>= putStrLn . reverse
```

c)
```haskell
f = getLine >>= \s -> putStrLn $ if null s then "<empty>" else s
```

d)
```haskell
f = getLine >>= \s -> putStrLn s >> pure s
```

Řeš. 7.1.2

```haskell
getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)
```

Řeš. 7.1.3

```haskell
loopecho = getLine >>= \s ->
           if null s then pure ()
                     else putStrLn s >>
```

loopecho

Řeš. 7.1.4

```
import Data.Char
getSanitized :: IO String
getSanitized = getLine >>= pure . filter isAlpha
```

Řeš. 7.1.5

```
import System.Directory

checkFile :: IO ()
checkFile = putStr "File: " >> getLine >>=
            doesFileExist >>= \b ->
            putStrLn $ if b then "File exists." else "No such file."
```

Řeš. 7.1.6

```
x >> f = x >>= \_ -> f
```

Řeš. 7.1.7

```
runLeft = foldl (\a b -> a >> b) (pure ())
runLeft' = foldr (\a b -> a >> b) (pure ())
runRight = foldl (\a b -> b >> a) (pure ())
runRight' = foldr (\a b -> b >> a) (pure ())
```

Všimněte si, že nezáleží na výběru akumulační funkce. Aplikací funkcí `foldl` a `foldr` sice vzniknou jiné stromy aplikací operátorů `>>`, ale pořadí listů zůstává stejné a operace `>>` je asociativní. Pro lepší názornost si nakreslete obrázek příslušných katamorfizmů.

Řeš. 7.2.2

```
leftPadTwo :: IO ()
leftPadTwo = do
    str1 <- getLine
    let l1 = length str1
    str2 <- getLine
    let l2 = length str2
    let m = max l1 l2
    putStrLn (replicate (m-l1) ' ' ++ str1)
    putStrLn (replicate (m-l2) ' ' ++ str2)
```

Řeš. 7.2.3

```
main = getLine >>= \f ->
       getLine >>= \s ->
       appendFile f (s ++ "\n")
```

Řeš. 7.2.4

```
query' :: String -> IO Bool
query' question =
    putStrLn question >> getLine >>= \answer -> pure (answer == "ano")
```

Nebo lze upravit vzniklou λ-abstrakci na pointfree tvar:

```
query'' :: String -> IO Bool
query'' question =
    putStrLn question >> getLine >>= pure . (== "ano")
```

Řeš. 7.2.5

a)

```
query2 :: String -> IO Bool
query2 question = do
    putStrLn question
    answer <- getLine
    if answer == "ano"
```

```
            then pure True
            else if answer == "ne"
                then pure False
                else query2 question
```

b)
```
import Data.Char

query3 :: String -> IO Bool
query3 question = do
    putStrLn question
    answer <- getLine
    let lcAnswer = map toLower answer
    if lcAnswer `elem` ["ano", "áno", "yes"] then
        pure True
    else
        if lcAnswer `elem` ["ne", "nie", "no"] then
            pure False
        else
            query3 question
```

Řeš. 7.2.6

a) Typ `IO String`; akce, která při spuštění načte řádek vstupu, jenž se stane vnitřním výsledkem.

b) Není výrazem. Při zadání do interpretu nebo do souboru se takto zadefinuje nová akce `x :: IO String`, která se chová stejně jako `getLine`.

c) Ekvivalentní `getLine`; `x` je lokální akce zadefinovaná jako výše.

d) Syntakticky nesprávné; konstrukci `<-` je lze použít pouze v `do`-bloku nebo intensionálním zápisu seznamu.

e) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek.

f) Typově nesprávné; na pravé straně `>>=` má v tomto případě stát funkce typu `String -> IO a`.

g) Může to být překvapivé, ale výraz je správně utvořený. Má ale poněkud děsivý typ `Monad m => m (IO String)`, což pro účely tohoto cvičení můžeme číst jako `IO (IO String)`. Je to akce, jejímž vnitřním výsledkem je po spuštění akce `getLine`.

h) Ekvivalentní `getLine`; `x` je opět lokálně zadefinovaná akce.

i) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek. Analogické k e).

j) Typově nesprávné; poslední výraz `do`-bloku musí být typu „akce“. Analogické k f).

Řeš. 7.3.1 Do souboru s akcí `leftPadTwo`, nechť se jmenuje třeba `Cv07.hs`, přidáme:

```
main :: IO ()
main = putStrLn "Always two lines there are:" >> leftPadTwo
```

Soubor přeložíme a spustíme:

```
$ ghc Cv07.hs
[1 of 1] Compiling Main             ( Cv07.hs, Cv07.o )
Linking Cv07 ...
$ ./Cv07
Always two lines there are:
They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
```

```
                       They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
$
```

Řeš. 7.3.2

```haskell
biggest :: Integer -> Integer -> Integer -> Integer
biggest x y z = max (max x y) z

lowerTwo :: Integer -> Integer -> Integer -> Integer
lowerTwo x y z = x + y + z - biggest x y z

getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)

main :: IO ()
main = do
  putStrLn "Enter first number:"
  x <- getInteger
  putStrLn "Enter second number:"
  y <- getInteger
  putStrLn "Enter third number:"
  z <- getInteger
  if biggest x y z < lowerTwo x y z
      then putStrLn "ANO"
      else putStrLn "NE"
```

Řeš. 7.3.3

```haskell
import System.Directory

main :: IO ()
main = do putStr "Enter filename: "
          fileName <- getLine
          fileExists <- doesFileExist fileName
          if fileExists then do
              fileContents <- readFile fileName
              putStr fileContents
          else putStrLn "Impossible. Perhaps the archives are incomplete."
```

Řeš. 7.3.4 Obecně, rekurze v kontextu `IO` umožňuje opakované vykonávání akcí. V tomto případě vidíme, že od první akce, která je součástí `main'`, až po její další výskyt se opakuje načtení řetězce a výpis jeho převrácené podoby. Dobře použitá rekurze musí být vhodným způsobem ukončitelná. Význam kódu měl nejspíš být takový, že zadáme-li prázdný řetězec, dojde k ukončení rekurze a akce se nebude znovu opakovat. Autor zde však rekurzi neuhlídal a `main'` se volá pokaždé; akci tedy není jak ukončit. Náprava je jednoduchá: stačí poslední řádek odsadit tak, aby byl součástí vnořeného `do`-bloku.

V takto jednoduchém případě se místo vnořeného `do`-bloku dá bez ztráty čitelnosti použít `>>`:

```haskell
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "You shall not pass!"
          else putStrLn (reverse s) >> main
```

Řeš. 7.3.5

```
import Text.Read

requestInteger :: String -> IO (Maybe Integer)
requestInteger delim = do
    str <- getLine
    if str == delim then pure Nothing else f (readMaybe str)
    where
        f Nothing = requestInteger delim
        f mayInt  = pure mayInt
```

Řeš. 7.3.7

```
leftPad :: IO ()
leftPad = leftPad' []

leftPad' ::  [String] -> IO ()
leftPad' lines = do
    line <- getLine
    if line == "."
        then printLines (maximum (map length lines)) (reverse lines)
        else leftPad' (line:lines)

printLines :: Int -> [String] -> IO ()
printLines _ [] = pure ()
printLines maxLen (line:xs) = do
    let prefix = replicate (maxLen - length line) ' '
    putStrLn (prefix ++ line)
    printLines maxLen xs
```

Řeš. 7.3.10 Dvojice obsahující v obou složkách týž řádek vstupu. Začne se vyhodnocovat první složka dvojice; vyhodnocení `getLine'` způsobí přečtení řádku vstupu, který se stane výsledkem výrazu. Kvůli línému vyhodnocování se už ve druhé složce nemá co vyhodnocovat (a tedy se nenačte další řádek), protože výraz `getLine'` už vyhodnocený je.

Příklad ilustruje jeden z důvodů, proč je v Haskellu potřeba řešit vstup a výstup poněkud neintuitivním způsobem. Dalším důvodem je pořadí spuštění – u vstupně-výstupních akcí ho většinou chceme přesně stanovené. To je u normální redukční strategie obtížnější dosáhnout. Řekněme, že výrazy s vedlejšími efekty se vždy vyhodnocují znovu (aby vůbec došlo k těm efektům). Například vyhodnocení `getLine'` vždy způsobí, že se načte řádek. Vezměme si hypotetickou funkci `writeFile' :: FilePath -> String -> ()`. Budeme chtít napsat program, který načte název souboru a obsah, který do něj uloží. Které řešení je vezme v jakém pořadí?

```
main =      writeFile' getLine' getLine'
main = flip writeFile' getLine' getLine'
```

Odpověď zní: nevíme, ale v obou bude stejné. Záleží na tom, co `writeFile'` bude chtít načíst první (dá se očekávat, že to bude název souboru). Zároveň s tím `flip` nic neudělá, protože oba argumenty jsou nevyhodnocený výraz (s vedlejšími efekty), a byť už máme zajištěno, že se oba výskyty vyhodnotí zvlášť, redukční strategie stále nevynucuje, aby se to událo před voláním `writeFile'`.

Chceme-li vynutit pořadí vykonávání akcí, můžeme mezi nimi uměle zavést závislost. Například první funkce by vracela kromě načteného řádku i nějaké nahodilé číslo, které by se jako štafetový kolík předalo druhé funkci. Ta by tak mohla byt vyhodnocena až poté, co by došlo k vyhodnocení první. Typ by se přitom změnil na

`getLine'' :: Integer -> (String, Integer)`. Implementace řetězení funkcí je jistě zřejmá. A protože k takovému řetězení by docházelo často, vymysleli bychom si na něj nějaký hezký operátor a nazvali jej třeba `>>`.

Článek `https://wiki.haskell.org/IO_inside` názorně krok po kroku ukazuje, že myšlenka štafetového kolíku je už poměrně blízko skutečné implementaci `IO`. Všem nebojácným zájemcům o Haskell ho Pan Fešák doporučuje si přečíst.

Přiložený kód

Pan Sazeč upozorňuje: Kopírování kódu ze souboru PDF nezachovává odsazení! Soubory jsou ale vloženy i jako přílohy tohoto dokumentu; s rozumným prohlížečem je můžete stáhnout kliknutím na název souboru, ať už zde, či v příslušných příkladech. Přílohy jsou k nalezení také ve studijních materiálech v ISu.

04_data.hs

```
-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
type WeaponsPoints = Integer
type HealthPoints = Integer

data Armor = Leather | Steel -- kožené nebo ocelové brnění
             deriving (Eq, Show)

data Warrior = Warrior HealthPoints Armor WeaponsPoints
               deriving (Eq, Show)

attack :: Warrior -> Warrior -> Warrior
attack = undefined

warrior1 = Warrior 30 Leather 30
warrior2 = Warrior 20 Steel 25

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --

addVars :: String -> String -> [(String, Integer)] -> Maybe Integer
addVars = undefined

data Mark = A | B | C | D | E | F | X | S deriving (Eq, Show)
type StudentName = String
type CourseName = String
data StudentResult = StudentResult StudentName CourseName (Maybe Mark)
                     deriving (Eq, Show)

summarize :: StudentResult -> String
summarize = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data Nat = Zero | Succ Nat deriving Show

natToInt :: Nat -> Int
natToInt = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data Expr = Con Double
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
          deriving Show

eval :: Expr -> Double
eval = undefined
```

```haskell
evalMay :: Expr -> Maybe Double
evalMay = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
                 deriving (Eq, Show)

tree00 :: BinTree Int
tree00 = Node 42 (Node 28 Empty Empty) Empty

tree01 :: BinTree Int
tree01 = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
                (Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty))

tree02 :: BinTree Int
tree02 = Node 9 Empty (Node 11 (Node 10 Empty Empty)
                               (Node 12 Empty Empty))

tree03 :: BinTree Int
tree03 = Node 8 tree01 tree02

tree04 :: BinTree Int
tree04 = Node 4 (Node 2 Empty (Node 3 Empty Empty))
                (Node 6 (Node 5 Empty Empty) Empty)

tree05 :: BinTree Int
tree05 = Node 100 (Node 101 Empty
                            (Node 102 (Node 103 Empty
                                                (Node 104 Empty Empty))
                                      Empty))
                  (Node 99 (Node 98 Empty Empty)
                           (Node 98 Empty Empty))

emptyTree :: BinTree Int
emptyTree = Empty

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
treeSize :: BinTree a -> Int
treeSize = undefined

listTree :: BinTree a -> [a]
listTree = undefined

height :: BinTree a -> Int
height = undefined

longestPath :: BinTree a -> [a]
longestPath = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
fullTree :: Int -> a -> BinTree a
fullTree = undefined

treeZip :: BinTree a -> BinTree b -> BinTree (a, b)
```

```haskell
treeZip = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)
treeMayZip = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
isTreeBST :: (Ord a) => BinTree a -> Bool
isTreeBST = undefined

searchBST :: (Ord a) => a -> BinTree a -> Bool
searchBST = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data RoseTree a = RoseNode a [RoseTree a]
                  deriving (Show, Read)

roseTreeSize :: RoseTree a -> Int
roseTreeSize = undefined

roseTreeSum :: Num a => RoseTree a -> a
roseTreeSum = undefined

roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data LogicExpr = Pos | Neg
               | And LogicExpr LogicExpr
               | Or LogicExpr LogicExpr
               | Implies LogicExpr LogicExpr
               | Equiv LogicExpr LogicExpr

evalExpr :: LogicExpr -> Bool
evalExpr = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
data IntSet = SetNode Bool IntSet IntSet -- Node isEnd zero one
            | SetLeaf
              deriving Show

insert :: IntSet -> Int -> IntSet
insert = undefined

find :: IntSet -> Int -> Bool
find = undefined

listSet :: IntSet -> [Int]
listSet = undefined

-- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * --
```

05_data.hs

```haskell
data Sex = Female | Male

allPairs :: [(String, Sex)] -> [(String, String)]
allPairs = undefined

people :: [(String, Sex)]
people =  [("Jeff", Male), ("Britta", Female), ("Annie", Female), ("Troy",
 ↪  Male)]


naturalsFrom :: Integer -> [Integer]
naturalsFrom n = n : naturalsFrom (n + 1)

naturals :: [Integer]
naturals = naturalsFrom 0

-- naturals !! 2


filter' _ [] = []
filter' p (x : xs) = if p x then x : filter' p xs else filter' p xs

-- Jak se bude chovat interpret jazyka Haskell pro vstup filter' (< 3)
 ↪  naturals?

takeWhile' _ [] = []
takeWhile' p (x : xs) = if p x then x : takeWhile' p xs else []

-- Jak se bude chovat interpret jazyka Haskell pro vstup takeWhile' (< 3)
 ↪  naturals?

addNumbers :: [String] -> [String]
addNumbers = undefined

integers :: [Integer]
integers = undefined

threeSum :: [(Integer, Integer, Integer)]
threeSum = undefined


sumFold :: Num a => [a] -> a
sumFold = undefined

productFold :: Num a => [a] -> a
productFold = undefined

orFold :: [Bool] -> Bool
orFold = undefined

lengthFold :: [a] -> Int
lengthFold = undefined

maximumFold :: Ord a => [a] -> a
```

```haskell
maximumFold = undefined


-- foldr1 (\x s -> x + 10 * s)
-- foldl1 (\s x -> 10 * s + x)


concatFold :: [[a]] -> [a]
concatFold = undefined

listifyFold :: [a] -> [[a]]
listifyFold = undefined

nullFold :: [a] -> Bool
nullFold = undefined

composeFold :: [a -> a] -> a -> a
composeFold = undefined

idFold :: [a] -> [a]
idFold = undefined

mapFold :: (a -> b) -> [a] -> [b]
mapFold = undefined

headFold :: [a] -> a
headFold = undefined

lastFold :: [a] -> a
lastFold = undefined

maxminFold :: Ord a => [a] -> (a, a)
maxminFold = undefined

suffixFold :: [a] -> [[a]]
suffixFold = undefined

filterFold :: (a -> Bool) -> [a] -> [a]
filterFold = undefined

oddEvenFold :: [a] -> ([a], [a])
oddEvenFold = undefined

takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold = undefined

dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold = undefined
```

05_treeFold.hs

```haskell
data BinTree a = Node a (BinTree a) (BinTree a)
               | Empty
               deriving (Eq, Show)
```

```haskell
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold n e (Node v l r) = n v (treeFold n e l)
                                (treeFold n e r)
treeFold n e Empty        = e

tree01 :: BinTree Int
tree01 = Node 2 (Node 3 (Node 5 Empty Empty) Empty)
                (Node 4 (Node 1 Empty Empty) (Node 1 Empty Empty))

tree02 :: BinTree String
tree02 = Node "C" (Node "A" Empty (Node "B" Empty Empty))
                  (Node "E" (Node "D" Empty Empty) Empty)

tree03 :: BinTree (Int,Int)
tree03 = Node (3,3) (Node (2,1) Empty Empty) (Node (1,1) Empty Empty)

tree04 :: BinTree a
tree04 = Empty

tree05 :: BinTree Bool
tree05 = Node False (Node False Empty (Node True Empty Empty))
                    (Node False Empty Empty)

tree06 :: BinTree (Int, Int -> Bool)
tree06 = Node (0,even)
              (Node (1,odd) (Node (2,(== 1)) Empty Empty) Empty)
              (Node (3,(< 5)) Empty
                      (Node (4,((== 0) . mod 12)) Empty Empty))
```

06_data.hs

```haskell
-- typy a typové třídy: na papír, viz sbírka

-- 6.1.7: tento typ si zkopírujte na všechny funkce podle příkladu
fMaybe :: (a -> Maybe b) -> Maybe a -> Maybe b
fMaybe = undefined

data BinLeafTree a = LLeaf a
                   | LNode (BinLeafTree a) (BinLeafTree a)
                   deriving (Show, Eq)

-- je třeba doplnit i typ:
-- bitLeafTreeVal1
-- bitLeafTreeVal2

-- typové konstruktory:
-- hodnotové konstruktory:

-- je třeba doplnit i typ:
-- tSumEven

data RoseLeafTree a = RLNode [RoseLeafTree a]
                    | RLLeaf a
```

```
                    deriving (Show, Eq)

-- typové konstruktory:
-- hodnotové konstruktory:

-- je třeba doplnit i typ:
-- countValueLeaves

-- je třeba doplnit i typ:
-- rlFilter
```

07_guess.hs

```haskell
import Data.Char

main = guess 1 10

query ot = do putStr ot
              ans <- getLine
              pure (ans == "ano")

guess :: Int -> Int -> IO ()
guess m n = do putStrLn ("Mysli si cele cislo od " ++ show m
                         ++ " do " ++ show n ++ ".")
               kv m n

kv m n = do
     if m == n
     then putStrLn ("Je to " ++ show m ++ ".")
     else do o <- query $ "Je tve cislo vetsi nez " ++ show k ++ "? "
             if o then kv (k + 1) n else kv m k
  where k = (m + n) `div` 2
```