Naučte se Haskell!

6. Funkce vyššího řádu

7. Moduly

8. Vytváříme si své typy a typové třídy

Připravuje se

Obsah Začínáme →

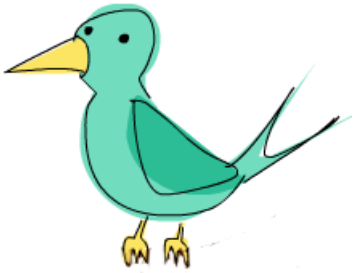
Úvod

English version

O tomto tutoriálu

Vítejte v příručce **Naučte se Haskell**! Jestliže čtete tento úvod, je naděje, že se chcete učit Haskell. Pokud ano, jste na správném místě, ale pojďme se trochu pobavit o tomhle textu.

Rozhodl jsem se ho napsat, protože jsem chtěl utužit mé znalosti Haskellu a protože jsem si myslel, že bych mohl pomoct nováčkům naučit se Haskell z mé perspektivy. Po internetu koluje celkem málo tutoriálů na Haskell. Když jsem s Haskellem začínal, neučil jsem se pouze z jednoho zdroje. Naučil jsem se ho pročítáním několika návodů a článků, protože každý z nich vysvětloval věci jiným způsobem než ostatní. Pročtením těchto zdrojů jsem byl schopný si propojit a utřídit znalosti. Tohle je tedy pokus o přidání dalšího použitelného zdroje na naučení Haskellu a vy máte větší šanci takový zdroj najít.



Návod je zaměřen na programátory, kteří mají zkušenosti s imperativními jazyky (C, C++, Java, Python…), ale nikdy neprogramovali ve funkcionálních jazycích (Haskell, ML, OCaml…). Ačkoliv bych se i vsadil, že pokud nemáte významné programátorské zkušenosti, chytrý maník jako vy bude moct sledovat příručku a naučit se Haskell.

IRC kanál #haskell (nebo #haskell.cz v češtině) na síti freenode je skvělé místo, kde se můžete ptát, pokud nebudete vědět jak dál. Lidé jsou tam nesmírně milí, trpěliví a mají pochopení pro začátečníky.

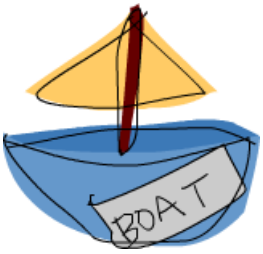
Haskell se mi nepodařilo naučit asi dvakrát, než jsem tomu konečně porozuměl, protože se mi to celé zdálo příliš divné a nechápal jsem to. Ale jednou mi to „docvaklo“ a jakmile jsem se dostal přes úvodní překážky, byla to celkem hračka. Snažím se říct, že: Haskell je skvělý a pokud se zajímáte o programování, měli byste se ho opravdu naučit, i když na začátku vypadá divně. Učit se Haskell je podobné jako se poprvé učit programovat — je to zábava! Donutí vás to myšlet odlišně, což nás přivádí k další sekci…

Takže co je to Haskell?

Haskell je **čistě funkcionální programovací jazyk**. V imperativních jazycích se provádí věci zadáváním sekvence úloh počítači, které se pak provádí. Při provádění mohou měnit stavy. Například pokud přiřadíte proměnné a číslo 5, děláte něco dalšího, a pak změníte hodnotu proměnné na jinou. Máte k dispozici struktury pro kontrolu toku na několikanásobné provádění určitých akcí. V čistě funkcionálním programování neříkáte počítači, co má dělat, spíše mu říkáte, jaká ta věc *je*. Faktoriál čísla je součin všech čísel od jedničky po zadané číslo, součet seznamu čísel je první číslo plus součet všech zbylých čísel a tak dále. Tohle se vyjadřuje ve formě funkcí. Není možné také přiřadit proměnné nějakou hodnotu a poté ji změnit na něco jiného později. Pokud řeknete, že a je 5, nemůžete později říct, že je něco jiného, protože jste právě prohlásili, že je to pětka. Co jste zač, nějaký lhář? Takže v čistě funkcionálních jazycích nemá funkce vedlejší efekty. Jediná věc, co funkce může dělat, je vypočítat něco a vrátit to jako výsledek. Na první pohled to vypadá jako dost omezující, ale ve skutečnosti to má dost pěkné důsledky: pokud je funkce zavolána dvakrát se stejnými parametry, je zaručeno, že vrátí stejný výsledek. Tomu se říká referenční transparentnost a kromě zjišťování překladače o zvyklostech programátora to také umožňuje jednodušeji vyvozovat (a dokonce dokázat) správnost funkce a poté budovat složitější funkce slepováním jednoduchých funkcí k sobě.

Haskell je **líný**. To znamená, že pokud mu nenařídíte opak, Haskell nevyhodnotí funkci a nepočítá věci, než po něm nezačnete chtít výsledek. To funguje dobře s referenční

transparentností a umožňuje to považovat programy jako řadu **trasformací na datech**. Také to dovoluje práci s bezva věcmi jako jsou nekonečné datové struktury. Řekněme, že máme neměnitelný seznam čísel `xs = [1,2,3,4,5,6,7,8]` a funkci `doubleMe`, která vynásobí každý prvek seznamu dvojkou a poté vrátí nový seznam. Pokud chceme vynásobit náš seznam osmičkou v imperativním jazyce a udělat `doubleMe(doubleMe(doubleMe(xs)))`, pravděpodobně by se předal seznam jedenkrát a vytvořila se kopie, která by se vrátila. Poté by se předal seznam funkci ještě dvakrát a vrátil by se výsledek. V líném jazyce se zavoláním funkce `doubleMe` na seznamu bez požadování výsledku skončí program zhruba tím, že si řekne: „Jo, jo, udělám to později!“ Ale pokud chcete vidět výsledek, první `doubleMe` řekne tomu druhému, že chce vidět výsledek, hned! Druhé řekne třetímu a třetí neochotně vrátí dvojnásobek 1, což je 2. Druhý obdrží hodnotu a vrací 4 prvnímu. První to uvidí, a zahlásí%že první prvek je 8. Takže udělá pouze jeden průchod seznamem a jenom tehdy když je to opravdu potřeba. Jedna z možností, jak chtít něco od líného jazyka, je vzít nějaká počáteční data a efektivně je transformovat a vylepšovat tak, aby se podobala našemu chtěnému výsledku.



Haskell je **staticky typovaný**. Jakmile překládáte svůj program, překladač ví, jaký kousek kódu je číslo, jaký je řetězec a tak dále. To znamená, že hodně potenciálních chyb je odchytáno v čase překladu. Pokud se budete snažit sečíst dohromady číslo a řetězec, překladač si vám bude stěžovat. Haskell má velmi dobrý typový systém, který používá **typové odvozování**. To znamená, že nemusíte explicitně otypovávat každý kus kódu, protože typový systém může inteligentně přijít na hodně věcí. Pokud prohlásíte `a = 5 + 4`, nemusíte Haskellu říkat, že a je číslo, může na to přijít sám. Odvozování typů také dovoluje mít kód více obecný. Pokud funkce požaduje dva parametry a sečte je dohromady, není potřeba explicitně uvádět jejich typ, funkce s nimi bude pracovat jako se dvěma parametry, které se chovají jako čísla.

Haskell je **elegantní a výstižný**. Protože používá hodně vysokoúrovňových konceptů, programy napsané v Haskellu jsou obvykle kratší než jejich imperativní ekvivalenty. A kratší programy se jednodušeji spravují než dlouhé a obsahují méně chyb.

Haskell vytvořilo několik **opravdu chytrých chlápků** (s doktorskými tituly). Práce na Haskellu začala v roce 1987, kdy se vytvořil výbor výzkumníků, aby navrhli převratný jazyk. V roce 2003 byl publikován Haskell Report, který definuje ustálenou verzi jazyka.

Co bude potřeba

Textový editor a překladač Haskellu. Pravděpodobně už máte svůj oblíbený textový editor nainstalován, takže tím nebudeme ztrácet čas. Dva hlavní překladače Haskellu jsou v současnosti GHC (Glasgow Haskell Compiler) a Hugs. Pro účely tohoto tutoriálu budeme používat GHC. Nebudu se zabývat detaily instalace. Na Windows je to otázka stáhnutí instalátoru, několika kliknutí na tlačítko „Další“ a poté restartu počítače. Na linuxových distribucích postavených na Debianu stačí pouze udělat `apt-get install ghc6 libghc6-mtl-dev` a jste vysmátí. Nevlastním Mac, ale zaslechl jsem, že pokud máte MacPorty, můžete získat GHC provedením příkazu `sudo port install ghc`. Také si myslím, že se dá v Haskellu dělat vývoj pomocí té potrhlé jednotlačítkové myši, ačkoliv si nejsem jistý.

GHC umí vzít skript napsaný v Haskellu (běžně mívají příponu .hs) a přeložit jej, ale má také interaktivní mód, který umožňuje interaktivně interagovat se skripty. Interaktivně. Můžete zavolat funkci z načteného skriptu a výsledky jsou zobrazeny ihned. Pro učení je to mnohem jednodušší a rychlejší než překládat program pokaždé, když se v něm provede změna, a spouštět ho z příkazového řádku. Interaktivní mód je vyvolán napsáním `ghci` do příkazového řádku. Pokud máte definovány nějaké funkce v souboru, nazvaného řekněme `mojefunkce.hs`, načtete ho napsáním `:l mojefunkce` a poté si s nimi můžete hrát, pokud je soubor `mojefunkce.hs` ve stejném adresáři, jako bylo `ghci` spuštěno. Pokud ve skriptu něco změníte, stačí znovu napsat `:l mojefunkce` nebo příkaz `:r`, který je stejný, protože znovu načte současný .hs skript. Mé obvyklé pracovní prostředí, když si pohrávám s programy, je .hs soubor, ve kterém mám definovány některé funkce, jež načtu, vrtám se v něm, načtu ho znovu a tak dále. To je také, čím se zde budeme zabývat.

Obsah Začínáme

← Úvod Obsah Typy a typové třídy →

Začínáme

English version

Připravit, pozor, teď!

Fajn, tak začneme! Pokud patříte do skupiny těch hrozných osob, které nečtou úvody do čehokoliv, a přeskočili ho, možná byste si stejně měli přečíst poslední část úvodu, protože se tam vysvětluje, co je potřeba na práci s tímto tutoriálem a jak budeme načítat soubory s funkcemi. První věc, kterou se budeme zabývat, je spuštění interaktivního módu GHC a zavolání nějakých funkcí, abyste se spřátelili s Haskellem. Spusťte si terminál a zadejte do něj `ghci`. Vypíše se zhruba takovéto uvítání:

```
                6.10 1
```

Gratuluji, jste v GHCi! Prompt je tady `Prelude>`, ale protože se může prodloužit po načtení nějakých dalších věcí, budeme dále používat `ghci>`. Pokud chcete mít stejný prompt, stačí napsat příkaz `:set prompt "ghci> "`.

Tady je příklad nějaké jednoduché aritmetiky.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

Tohle je celkem samozřejmé. Můžeme také použít více operátorů na jednom řádku a ukázat si obvyklou prioritu operátorů. Můžeme použít závorky z důvodů explicitnosti nebo pro změnu priority.

```
ghci>  50 * 100  - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 *  100 - 4999
 244950
```

Docela pěkné, co? Jo, vím, že není, ale vydržte to se mnou. Malá záludnost, na kterou je třeba si dát pozor, jsou záporná čísla. Pokud chceme pracovat se záporným číslem, je vždycky lepší jej obklopit závorkami. Při pokusu o výraz `5 * -3` bude na vás GHCi řvát, ale výraz `5 * (-3)` bude fungovat dobře.

Booleova algebra je také celkem jasná. Jak pravděpodobně víte, `&&` znamená booleovské *a*, `||` znamená booleovské *nebo*. Pomocí `not` se neguje `True` (*pravda*) nebo `False` (*nepravda*).

```
ghci>
```

```
ghci>
ghci>
ghci>
ghci>
```

Testování na rovnost se dělá nějak takhle.

```
ghci> 5 == 5
ghci> 1 == 0
ghci> 5 /= 5
ghci> 5 /= 4
ghci> "ahoj" == "ahoj"
```

A co zkusit zadat výraz `5 + "lama"` nebo `5 == True`? No, pokud zkusíme první kus kódu, dostaneme velkou strašidelnou chybovou zprávu!

```
    instance
                    of                      1 0 9
                      instance
                    5 + "lama"
                   of          = 5 + "lama"
```

Jejda! GHCi se nám snaží sdělit, že `"lama"` není číslo a tak by se neměla přičítat k číslu 5. I kdyby to nebyla `"lama"`, ale `"čtyřka"` nebo `"4"`, Haskell by to pořád nepovažoval za číslo. Operátor + očekává, že na pravé a levé straně budou čísla. Pokud se pokusíme zadat `True == 5`, GHCi nám sdělí, že nám nesouhlasí typy. Zatímco + funguje jenom na číslech, `==` funguje na jakýchkoliv dvou věcech, které se dají porovnávat. Háček je v tom, že oba musí mít odpovídající typ. Nemůžete porovnávat jablka s hruškami. Na typy se podíváme blíže později. Poznámka: můžete provést `5 + 4.0`, protože pětka je záludná a může se vydávat za celé nebo reálné číslo. Kdežto `4.0` se nemůže vydávat za celé číslo, takže se `5` musí přizpůsobit.

Možná to nevíte, ale teď jsme tu celou dobu používali funkce. Například * je funkce, která bere dvě čísla a násobí je. Jak jste viděli, zavoláme ji vecpáním mezi ně. Tomu se říká *infixová* funkce. Většina funkcí, které nepracují s čísly, jsou *prefixové* funkce. Podívejme se na ně.

Funkce jsou většinou prefixové, takže i když od teď explicitně neuvedeme, že je to funkce v prefixové formě, budeme ji za ni považovat. Ve většině imperativních jazyků jsou funkce zavolány napsáním názvu funkce a poté parametrů v závorkách, často oddělených čárkami. V Haskellu jsou funkce zavolány napsáním názvu funkce, mezery, a poté parametrů, oddělených mezerami. Pro začátek zkusíme zavolat jednu z nejnudnějších funkcí v Haskellu.

```
ghci>      8
9
```

Funkce `succ` požaduje jako parametr cokoliv, co má definováno následníka a následně ho vrátí. Jak můžete vidět, oddělili jsme název funkce od parametru mezerou. Zavoláním funkce s více parametry je také jednoduché. Funkce `min` a `max` vezmou dvě věci, které se dají porovnat (jako třeba čísla!) a vrátí menší nebo větší z nich.

```
ghci>     9 10
9
ghci>     3.4 3.2
3.2
ghci>     100 101
101
```

Aplikace funkcí (zavolání funkce vložením mezery za ní a přidáním parametrů) má největší přednost ze všeho. Což pro nás znamená, že tyto dva výrazy jsou stejné.

```
ghci>      9 +     5 4 + 1
16
ghci>       9  +      5 4  + 1
16
```

Nicméně pokud chceme získat následníka součinu čísel 9 a 10, neměli bychom psát `succ 9 * 10`, protože to bychom získali následníka devítky, který by byl násoben desítkou. Tedy číslo 100. Museli bychom napsat `succ (9 * 10)`, abychom obdrželi číslo 91.

Pokud funkce požaduje dva parametry, můžeme ji zavolat také infixově, když ji obklopíme zpětnými apostrofy. Například funkce `div` vezme dvě celá čísla a provede s nimi celočíselné dělení. Provedením `div 92 10` dostaneme výsledek 9. Pokud to zapíšeme tímto způsobem, může to vést k nejasnostem, které číslo je dělenec a které dělitel. Můžeme tedy funkci zavolat infixově jako `92 `div` 10`, což je hned jasnější.

Hodně lidí, kteří přešli k Haskellu z imperativních jazyků, se snaží držet zápisu, ve kterém závorky znázorňují aplikaci funkce. Kupříkladu v jazyce C se používají závorky na zavolání funkcí jako `foo()`, `bar(1)` nebo `baz(3, "haha")`. Jak jsme již uvedli, pro aplikaci funkce je používána v Haskellu mezera. Tyto funkce by v Haskellu byly zapsány jako `foo`, `bar 1` a `baz 3 "haha"`. Takže pokud uvidíte něco jako `bar (bar 3)`, tak to neznamená, že `bar` je zavoláno s parametry `bar` a `3`. Znamená to, že nejprve zavoláme funkci `bar` s parametrem `3`, abychom obdrželi nějaké číslo, a pak na něj znovu zavolali `bar`. V C by to vypadalo zhruba jako `bar(bar(3))`.

Miminko má svou první funkci

V předchozí sekci jsme si vyzkoušeli volání funkcí. A teď si zkusíme vytvořit vlastní! Spusťte si svůj oblíbený editor a naťukejte tam následující funkci, která vezme číslo a vynásobí ho dvěma.

```
          =    +
```

Funkce jsou definovány podobným způsobem, jako jsou volány. Název funkce je následován parametry, oddělenými mezerami. Ale při definování funkce následuje `=`, za kterým určíme, co funkce dělá. Uložte si kód jako `miminko.hs` nebo tak nějak. Nyní přejděte do adresáře, ve kterém je uložen, a spusťte v něm `ghci`. Jakmile budete v GHCi, napište `:l miminko`. Jakmile se náš skript načte, můžeme si hrát s funkcí, jež jsme definovali.

```
ghci>
 1 of 1

ghci>          9
18
ghci>          8.3
16.6
```

Protože + funguje s celými i s desetinnými čísly (s čímkoliv, co se dá považovat za číslo, vážně), naše funkce budou také

Jednoduché. Také bychom to mohli definovat jako `doubleUs x y = x + x + y + y`. Vyzkoušení přinese očekávaný výsledek (nezapomeňte přidat tuto funkci do souboru `miminko.hs`, uložit jej a poté napsat `:l miminko` v GHCi).

```
ghci>          4 9
26
ghci>          2.3 34.2
73.0
ghci>          28 88 +          123
478
```

Jak se dalo čekat, je možné volat funkci z jiných funkcí, které jste si vytvořili. Můžeme toho využít a předefinovat funkci `doubleUs` následovně:

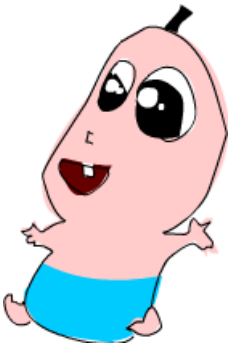
```
              =             +
```

Tohle je velmi jednoduchý příklad běžného schéma, jaké uvidíte všude v Haskellu. Vytvoření základní funkce, která je očividně správná, a pak je poskládána do více složitých funkcí. Takto se také vyhneme opakování. Co když nějaký matematik přijde na to, že dvojka je ve skutečnosti trojka a budete muset upravit svůj program? Stačí předefinovat `doubleMe` na `x + x + x` a protože `doubleUs` volá funkci `doubleMe`, mělo by to automaticky fungovat i v tom divném světě, kde je dvojka trojkou.

Funkce v Haskellu nemusí mít konkrétní pořadí, takže nezáleží, když definujete nejprve `doubleMe` a teprve poté `doubleUs`, nebo pokud to uděláte obráceně.

A teď vytvoříme funkci, která násobí číslo dvojkou, pokud to číslo je menší nebo rovno 100, protože čísla větší než 100 jsou pro nás dost velké!

```
= if   > 100
    then
    else   2
```



Tady jsme ukázali haskellový výraz if. Pravděpodobně znáte if z jiných jazyků. Rozdíl mezi tím v Haskellu a tím v imperativních jazycích je v tom, že část s else je v Haskellu povinná. V imperativních jazycích můžete přeskočit několik kroků, pokud není podmínka splněna, ale v Haskellu musí každý výraz a funkce něco vracet. Mohli bychom mít napsaný podmíněný výraz na jednom řádku, ale já pokládám první způsob za více přehledný. Další věc ohledně if v Haskellu: jedná se o *výraz*. Výraz je v podstatě kus kódu, který vrací hodnotu. Například `5` je výraz, protože vrací 5, `4 + 8` je výraz, `x + y` je také výraz, protože vrací součet `x` a `y`. Jelikož je else povinné, výraz if vždycky něco vrátí a proto je také výraz. Pokud chceme přidat jedničku ke každému číslu, které je vráceno naší předchozí funkcí, mohli bychom ji napsat zhruba takto.

```
=  if   > 100 then   else   2  + 1
```

Pokud bychom zanedbali závorky, přidali bychom jedničku pouze pokud by `x` nebylo větší než 100. Všimněte si čáry `'` na konci názvu funkce. Apostrof nemá v Haskellu žádný speciální syntaktický význam. Je to znak, který se dá použít v názvu funkce.

Obvykle používáme ' k označení striktní verzi funkce (která není líná), nebo lehce změněnou verzi funkce nebo proměnné. Protože je ' povolený znak v názvu funkce, můžeme vytvořit takovou funkci.

```
conan0'Brien = "To jsem já, Conan O'Brien!"
```

Jsou tu dvě pozoruhodné věci. První je, že v názvu funkce jsme Conanovo jméno nenapsali velkým písmenem. Je to proto, že funkce by jím neměly začínat. Proč tomu tak je, zjistíme později. Druhá věc je, že tahle funkce nepožaduje žádné parametry. Funkci bez parametrů se obvykle říká *definice* (nebo *pojmenování*). Protože nemůžeme měnit význam pojmenování (a funkce) po jejich definování, `conan0'Brien` a řetězec `"To jsem já, Conan O'Brien!"` se mohou při použití zaměňovat.

Úvod do seznamů

Stejně jako nákupní seznamy v reálném světě, seznamy v Haskellu jsou velmi užitečné. Je to nejvíce používaná datová struktura a může být použita na mnoho různých způsobů pro modelování a řešení spoustu problémů. Seznamy jsou TAK skvělé. V téhto sekci se podíváme na základy práce se seznamy, řetězce (které jsou také seznamy) a na generátor seznamu.

V Haskellu jsou seznamy **homogenní** datová struktura. Ukládá několik prvků stejného typu. Což znamená, že můžeme mít seznam čísel nebo seznam znaků, ale nemůžeme mít seznam, který obsahuje několik čísel a poté několik znaků. A nyní, seznam!

Poznámka: můžeme použít klíčové slovo `let`, abychom definovali pojmenování správně v GHCi. Napsat `let a = 1` v GHCi je totéž jako napsat `a = 1` do souboru a poté ho načíst.

```
ghci> let             =  4 8 15 16 23 42
ghci>
 4 8 15 16 23 42
```

Jak můžete vidět, seznamy se zadávají pomocí hranatých závorek a hodnoty se z nich oddělují čárkami. Pokud vyzkoušíte vytvořit seznam jako `[1,2,'a',3,'b','c',4]`, Haskell si bude stěžovat, že znaky (které se mimochodem zapisují pomocí znaku mezi jednoduchými uvozovkami) nejsou čísla. Když už mluvíme o znacích, tak textové řetezce jsou jenom seznamy znaků. Zápis `"ahoj"` je pouze syntaktický cukr (zkrácený zápis) pro řetězec `['a','h','o','j']`. Protože řetězce jsou seznamy, můžeme na ně používat funkci pro práci se seznamy, což je velmi šikovné.

Běžná úloha je spojení dvou seznamů dohromady. To se dělá pomocí operátoru ++.

```
ghci>  1 2 3 4  ++  9 10 11 12
 1 2 3 4 9 10 11 12
ghci> "ahoj" ++ " " ++ "světe"
"ahoj světe"
ghci>  'k' 'v'  ++  'á' 'k'
"kvák"
```

Poznámka překladatele: je velmi pravděpodobné, že GHCi vypíše místo diakritiky v řetězci hromadu divných čísel. Kupříkladu řetězec `"Příliš žluťoučký kůň úpěl ďábelské ódy."` se zobrazí jako `"P\345\237li\353 \382lu\357ou \269k\253 k\367\328 \250p\283l \271\225belsk\233 \243dy."`. Je to z důvodů převodu UTF-8 řetězců na ASCII. Řešením je použít knihovnu utf8-string, která se postará o správnou interpretaci, ale pro účely tohoto tutoriálu bude asi lepší to nechat být a ignorovat to, popřípadě psát příklady bez diakritiky.

Pozor na opakované používání operátoru ++ na dlouhé seznamy. Pokud spojujete dva seznamy (i když připojujete jednoprvkový seznam k delšímu seznamu, tedy například: `[1,2,3] ++ [4]`), Haskell musí interně projít přes celý seznam na

levé straně od ++. To není problém, pokud pracujeme s krátkými seznamy. Ale přidávat něco na konec seznamu, který má pět miliónů položek, bude chvíli trvat. Každopádně vložení prvku na začátek seznamu pomocí operátoru `:` je okamžité.

```
ghci> '1' " KOČIČKA"
"1 KOČIČKA"
ghci> 5  1 2 3 4 5
 5 1 2 3 4 5
```

Všimněte si, že `:` vezme jako argument číslo a seznam čísel nebo znak a seznam znaků, kdežto ++ dva stejné seznamy. I kdybyste chtěli přidat jeden prvek na konec seznamu pomocí ++, musí být obklopený hranatými závorkami, aby byl seznam.

Výraz `[1,2,3]` je vlastně syntaktický cukr pro `1:2:3:[]`. Dvě hranaté závorky `[]` jsou prázdný seznam. Když k nim připojíme `3` stane se z toho `[3]`. Jestliže připojíme k tomu `2` stane se z toho `[2,3]` a tak dále.

Poznámka: `[]`, `[[]]` a `[[],[],[]]` jsou tři odlišné věci. To první je prázdný seznam, to druhé je seznam obsahující jeden prázdný seznam a to třetí je seznam, který obsahuje tři prázdné seznamy.

Pokud chcete získat ze seznamu prvek na nějaké pozici, použijte `!!`. Číslování indexu začíná od nuly.

```
ghci> "Steve Buscemi"    6
'B'
ghci>  9.4 33.2 96.2 11.2 23.25     1
33.2
```

Pokud se ale budete snažit získat šestý prvek ze seznamu, který má pouze čtyři prvky, dostanete chybovou hlášku, takže opatrně!

Seznamy mohou také obsahovat seznamy. Taktéž mohou obsahovat seznamy obsahující seznamy obsahující seznamy…

```
ghci> let   =   1 2 3 4   5 3 3 3   1 2 2 3 4   1 2 3
ghci>
  1 2 3 4   5 3 3 3   1 2 2 3 4   1 2 3
ghci>   ++   1 1 1 1
  1 2 3 4   5 3 3 3   1 2 2 3 4   1 2 3   1 1 1 1
ghci>  6 6 6
  6 6 6   1 2 3 4   5 3 3 3   1 2 2 3 4   1 2 3
ghci>      2
 1 2 2 3 4
```

Seznamy uvnitř seznamu mohou být rozdílné délky, ale nesmí být jiného typu. Stejně jako nemůže být seznam obsahující několik znaků a několik čísel, nemůže být seznam, který obsahuje několik seznamů znaků a několik seznamů čísel.

Seznamy mohou být porovnávány, pokud je porovnatelný jejich obsah. Při používání `<`, `<=`, `>` a `>=` jsou seznamy porovnávány v lexikografickém pořadí. Nejprve jsou porovnány první prvky seznamů. Jestliže jsou stejné, pak jsou porovnány druhé prvky atd.

```
ghci>  3 2 1  >  2 1 0

ghci>  3 2 1  >  2 10 100

ghci>  3 4 2  >  3 4

ghci>  3 4 2  >  2 4

ghci>  3 4 2  ==  3 4 2
```

Co dalšího můžeme dělat se seznamy? Zde jsou některé základní funkce na práci se seznamy.

vezme seznam a vrátí jeho první prvek.

```
ghci>       5 4 3 2 1
5
```

vezme seznam a vrátí jeho zbytek, což je vlastně všechno kromě prvního prvku.

```
ghci>       5 4 3 2 1
 4 3 2 1
```

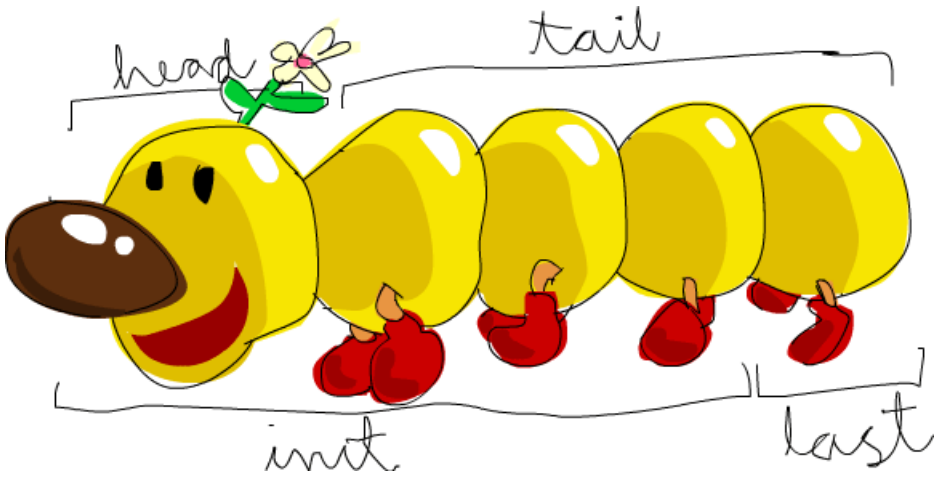
vezme seznam a vrátí jeho poslední prvek.

```
ghci>       5 4 3 2 1
1
```

vezme seznam a vrátí všechno kromě jeho posledního prvku.

```
ghci>       5 4 3 2 1
 5 4 3 2
```

Pokud budeme seznam považovat za příšeru, bude to asi takovéhle.



Ale co se stane, když budeme chtít první prvek z prázdného seznamu?

```
ghci>
```

Pane jo! Úplně se nám to vymklo kontrole! Pokud není příšera, nemůže mít ani začátek. Při použití funkcí `head`, `tail`, `last` a `init` dávejte pozor, aby nebyly použity na prázdný seznam. Tato chyba nemůže být odchycena v čase překladu, takže je potřeba dávat pozor, aby se náhodou nepřikázalo Haskellu vybrat prvky z prázdného seznamu.

vezme seznam a vrátí jeho délku.

```
ghci>        5 4 3 2 1
5
```

zjistí, jestli je seznam prázdný. Pokud ano, vrací `True`, v opačném případě `False`. Používejte tuto funkci místo `xs == []` (pokud máte seznam pojmenovaný `xs`).

```
ghci>       1 2 3

ghci>
```

obrátí seznam.

```
ghci>          5 4 3 2 1
 1 2 3 4 5
```

požaduje číslo a seznam. Vezme ze začátku seznamu tolik prvků, kolik je zadáno. Sledujte.

```
ghci>      3  5 4 3 2 1
 5 4 3
ghci>      1  3 9 3
 3
ghci>      5  1 2
 1 2
ghci>      0  6 6 6
```

Všimněte si, že pokud zkusíme vzít ze seznamu více prvků, než v něm je, prostě vrátí celý seznam. Jestliže zkusíme vzít 0 prvků, získáme tím prázdný seznam.

funguje podobně, akorát zahodí určitý počet prvků ze začátku seznamu.

```
ghci>      3  8 4 2 1 5 6
 1 5 6
ghci>      0  1 2 3 4
 1 2 3 4
ghci>      100  1 2 3 4
```

vezme seznam věcí, které se dají porovnat, a vrátí největší prvek.

vrátí nejmenší prvek.

```
ghci>          8 4 2 1 5 6
1
ghci>          1 9 2 3 4
9
```

vezme seznam čísel a vrátí jejich součet.

vezme seznam čísel a vrátí jejich součin.

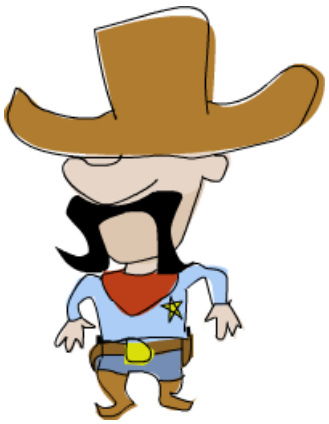
```
ghci>      5 2 1 6 3 2 5 7
31
ghci>          6 2 1 2
24
ghci>          1 2 5 6 7 9 2 0
0
```

vezme věc a seznam věcí a sdělí nám, jestli je ta věc prvkem seznamu. Je většinou volána jako infixová funkce, protože je jednodušší ji tak číst.

```
ghci> 4 `elem` 3 4 5 6
ghci> 10 `elem` 3 4 5 6
```

Tohle bylo pár základních funkcí na práci se seznamy. Na více funkcí se podíváme v následujících sekcích.

Šerifovy rozsahy



Co když budeme chtít seznam všech čísel mezi jedničkou a dvacítkou? Určitě bychom je mohli všechny prostě napsat, ale to není zřejmě řešení pro džentlmeny, kteří požadují od svých programovacích jazyků dokonalost. Místo toho použijeme rozsahy. Rozsahy jsou způsob vytváření seznamů, které jsou aritmetické posloupnosti prvků, které se dají vyjmenovat. Čísla mohou být vyjmenována. Jedna, dva, tři, čtyři atd. Znaky mohou být také vyjmenovány. Abeceda je posloupnost znaků od A do Z (česká abeceda je od A do Ž). Jména nemůžou být vyjmenována. Co následuje po jménu „Jan“? Nevím.

Pro vytvoření seznamu všech přirozených čísel od jedničky do dvacítky stačí napsat `[1..20]`. To je stejné, jako bychom napsali `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]` a není rozdíl mezi tímto zápisem a předchozím, kromě toho, že vypisování dlouhých posloupností ručně je hloupé.

```
ghci> 1 20
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
ghci> 'a' 'z'
"abcdefghijklmnopqrstuvwxyz"
ghci> 'K' 'Z'
"KLMNOPQRSTUVWXYZ"
```

Rozsahy jsou skvělé, protože se v nich také dá uvést přírůstek. Co když chceme všechna sudá čísla mezi jedničkou a dvacítkou? Nebo každé třetí číslo mezi jedničkou a dvacítkou?

```
ghci> 2 4 20
 2 4 6 8 10 12 14 16 18 20
ghci> 3 6 20
 3 6 9 12 15 18
```

Je to jednoduše otázka oddělení prvních dvou prvků čárkou a potě stanovení horní meze. I když to je celkem chytré, rozsahy nedovedou dělat věci, které od nich někteří lidé očekávají. Nemůžete napsat `[1,2,4,8,16..100]` a očekávat, že tím získáte všechny mocniny dvojky. Za prvé protože můžete pouze uvést pouze jeden přírůstek. A za druhé protože některé nearitmetické posloupnosti jsou víceznačné, pokud zadáme pouze několik jejich prvních členů.

Pro vytvoření seznamu všech čísel od dvacítky do jedničky nestačí napsat `[20..1]`, musíte uvést `[20,19..1]`.

Pozor na vytváření rozsahů desetinných čísel! Protože nejsou (už z definice) naprosto přesné, jejich používání v rozsazích může vést k celkem divokým výsledkům.

```
ghci> 0.1 0.3 1
 0.1 0.3 0.5 0.7 0.8999999999999999 1.0999999999999999
```

Moje rada je nepoužívat je v rozsazích.

Můžete také pomocí rozsahů vytvářet nekonečné seznamy jednoduše tím, že nestanovíte horní mez. Později se budeme zabývat detailněji nekonečnými seznamy. Teď pojďme prozkoumat, jak dostat prvních 24 násobků třináctky. Jasně, mohli bychom napsat `[13,26..24*13]`. Ale existuje lepší způsob: `take 24 [13,26..]`. Protože je Haskell líný, nebude se snažit vyhodnotit nekonečný seznam okamžitě, protože by s vyhodnocováním nikdy neskončil. Raději počká, co se všechno z toho nekonečného seznamu bude chtít. A uvidí, že chcete pouze prvních 24 prvků, které s radostí vrátí.

Užitečné funkce, které vytváří nekonečné seznamy:

vezme seznam a opakuje (cyklí) ho nekonečně dlouho. Pokud si chcete výsledný seznam zobrazit, bude pořád pokračovat, takže si ho budete muset někde ukrojit.

```
ghci>      10         1 2 3
 1 2 3 1 2 3 1 2 3 1
ghci>      12         "LOL "
"LOL LOL LOL "
```

vezme prvek a vytvoří z něj nekonečný seznam, obsahující pouze ten prvek.

```
ghci>      10         5
 5 5 5 5 5 5 5 5 5 5
```

Ačkoliv je jednodušší použít funkci , pokud chceme určitý počet opakování jednoho prvku v seznamu. Například `replicate 3 10` vrátí `[10,10,10]`.

Jsem generátor seznamu

Pokud jste někdy absolvovali matematický kurz, možná jste už slyšeli o *intenzionálním zápisu množin*. Ten se běžně používá pro generování určitých množin. Jednoduchý zápis množiny, jež obsahuje prvních deset sudých přirozených čísel, je $S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$. Část před svislítkem se nazývá výstupní funkce, `x` je proměnná, `N` je vstupní množina a `x <= 10` je predikát. To znamená, že množina obsahuje dvojnásobek všech přirozených čísel, které vyhovují predikátu.

Pokud to budeme chtít vyjádřit v Haskellu, můžeme zkusit něco jako `take 10 [2,4..]`. Ale co když nebudeme chtít dvojnásobky prvních deseti přirozených čísel, ale něco mnohem složitějšího? Mohli bychom ten seznam definovat intenzionálně, tedy ho vygenerovat. Generátor seznamu je velmi podobný intenzionálnímu zápisu množin. Budeme se zatím držet výpisu prvních deseti sudých čísel. Intenzionálně to můžeme zapsat jako `[x*2 | x <- [1..10]]`. Hodnota `x` je brána z rozsahu `[1..10]` a pro každý prvek z `[1..10]` (jež je vázaný na `x`) dostaneme naši hodnotu, akorát vynásobenou dvojkou. Tady je generátor seznamu v akci.

```
ghci>    2 |   <-  1  10
 2 4 6 8 10 12 14 16 18 20
```

Jak můžete vidět, dostaneme požadovaný výsledek. Nyní si přidáme podmínku (nebo také predikát) do naší definice generátoru. Predikáty se zapisují až za část s navázáním proměnné a odděluje se čárkou. Řekněme, že bychom chtěli pouze prvky, jejichž dvojnásobek je větší nebo rovný dvanácti.

```
ghci>    2 |   <-  1  10     2 >= 12
 12 14 16 18 20
```

Skvělé, funguje to. A co když budeme chtít všechna čísla od 50 do 100, jejichž zbytek po dělení číslem 7 je 3? Jednoduché.

```
ghci>      |    <-   50   100      `mod` 7 == 3
 52 59 66 73 80 87 94
```

Úspěch! Zapamatujte si, že se třídění seznamů pomocí predikátů také nazývá **filtrování**. Vezmeme seznam čísel a vyfiltrujeme je pomocí predikátů. A teď další příklad. Řekněme, že chceme vygenerovat přepsání každého lichého čísla většího než 10 řetězcem `"BANG!"` a každého lichého čísla, které je menší než 10, řetězcem `"BOOM!"`. Pokud číslo není liché, zahodíme ho. Z důvodů pohodlnosti vložíme náš generátor do funkce, abychom ho mohli jednoduše použít vícekrát.

```
              =    if    < 10 then "BOOM!" else "BANG!" |    <-
```

Poslední část definice generátoru je predikát. Funkce `odd` vrací `True`, pokud je číslo liché, a `False`, pokud je sudé. Prvek je přidán do seznamu pouze pokud jsou všechny predikáty vyhodnoceny jako `True`.

```
ghci>             7   13
 "BOOM!" "BOOM!" "BANG!" "BANG!"
```

Můžeme zapsat několik predikátů. Jestliže chceme všechna čísla od 10 do 20, která nejsou 13, 15 nebo 19, napíšeme:

```
ghci>      |    <-   10   20      /= 13     /= 15     /= 19
 10 11 12 14 16 17 18 20
```

Nejen že můžeme mít v definicích generátoru více predikátů (každý prvek musí splňovat veškeré predikáty, aby byl obsažen ve výsledném seznamu), můžeme také vybírat prvky z několika seznamů. Když vybíráme prvky z více seznamů, vygenerují se všechny kombinace ze zadaných seznamů a poté je můžeme zkombinovat ve výstupní funkci. Generátor seznamu, který bere prvky ze dvou seznamů délky 4, vrátí seznam délky 16, za předpokladu, že je nebude filtrovat. Pokud máme dva seznamy, `[2,5,10]` a `[8,10,11]` a budeme chtít součin všech možných kombinací čísel z těchto seznamů, uděláme následující.

```
ghci>        |    <-   2 5 10      <-   8 10 11
 16 20 22 40 50 55 80 100 110
```

Jak jsme čekali, délka nového seznamu je 9. Co když budeme chtít všechny součiny, které jsou větší než 50?

```
ghci>        |    <-   2 5 10      <-   8 10 11        > 50
 55 80 100 110
```

A co třeba generátor seznamu, který zkombinuje seznamy přídavných a podstatných jmen do bujaré epopeje?

```
ghci> let        =   "tulák" "žabák" "papež"
ghci> let             =   "líný" "nabručený" "pletichářský"
ghci>            ++ " " ++       |           <-                  <-
 "líný tulák" "líný žabák" "líný papež" "nabručený tulák" "nabručený žabák"
"nabručený papež" "pletichářský tulák" "pletichářský žabák" "pletichářský papež"
```

Už vím! Napišme si vlastní verzi funkce `length`! Nazveme ji `length'`.

```
            =      1 |    <-
```

Znak _ značí, že je nám jedno, co budeme dělat s prvkem ze seznamu, takže místo psaní názvu proměnné, kterou nikdy nepoužijeme, jednoduše napíšeme _. Tato funkce nahradí každý prvek v seznamu číslem `1` a poté je všechny sečte. Což znamená, že výsledný součet bude délka našeho seznamu.

Jenom přátelská připomínka: protože jsou řetězce seznamy, můžeme použít generátor seznamu na zpracování a vytváření řetězců. Zde je funkce, která vezme řetězce a odstraní z nich všechno kromě velkých písmen.

```
                    =     |   <-       `elem`   'A'   'Z'
```

Otestujeme ji:

```
ghci>                    "Hahaha! Ahahaha!"
"HA"
ghci>                    "neMAMRADZABY"
"MAMRADZABY"
```

Veškerou práci zde zastává predikát. Vyjadřuje, že znak bude obsažen v novém seznamu pouze pokud je prvkem seznamu `['A'..'Z']`. Zanořování generátorů je také možné, pokud operujete nad seznamy, jež obsahují další seznamy. Třeba seznam obsahující seznamy čísel. Pojďme odstranit všechna lichá čísla bez přeskupování seznamu.

```
ghci> let     =   1 3 5 2 3 1 2 4 5   1 2 3 4 5 6 7 8 9   1 2 4 2 1 6 3 1 3 2 3 6
ghci>       |   <-              |    <-
  2 2 4   2 4 6 8   2 4 2 6 2 6
```

Generátor seznamu je možné zapsat přes několik řádků. Takže pokud zrovna nepracujete v GHCi, je lepší rozdělit dlouhé seznamy přes více řádků, zvláště když jsou zanořené.

N-tice

V některých ohledech jsou n-tice (uspořádané heterogenní seznamy o n prvcích) podobné seznamům — slouží pro ukládání několika hodnot do jedné. Jenomže mají pár zásadních odlišností. Seznam čísel je seznam čísel. To je jeho typ a nezáleží na tom, jestli obsahuje jedno číslo nebo nekonečně mnoho. N-tice se ovšem používají, pokud přesně víte, kolik hodnot chcete zkombinovat a jejich typ závisí na počtu a typu jednotlivých složek. Jsou uvozeny kulatými závorkami a jejich složky odděleny čárkami.

Další důležitou odlišností je, že nemusí být homogenní. Narozdíl od seznamu může n-tice obsahovat kombinaci různých typů.

Zamysleme se nad tím, jak bychom v Haskellu vyjádřili dvourozměrnou souřadnici. Je možnost použít seznam. To by mohlo fungovat. Co když budeme chtít vložit pár vektorů do seznamu, abychom tak vyjádřili body nějakého útvaru na dvourozměrné ploše? Mohli bychom mít něco jako `[[1,2],[8,11],[4,5]]`. Problém s tímto způsobem spočívá v tom, že bychom z toho mohli udělat něco jako `[[1,2],[8,11,5],[4,5]]`, s čímž Haskell nemá problém, jelikož to je stále seznam seznamů čísel, ale nedává to smysl. Ale n-tice o velikosti dva (nazývána jako dvojice) má svůj vlastní typ, což znamená, že seznam nemůže obsahovat několik dvojic a zároveň nějaké trojice (n-tice velikosti tři), takže ji použijeme místo toho. Namísto obklopování souřadnic hranatými závorkami použijeme kulaté: `[(1,2),(8,11),(4,5)]`. Co když se budeme snažit vytvořit útvar jako `[(1,2),(8,11,5),(4,5)]`? No, dostaneme tuhle chybu:

```
          type
type
   8  11  5
     1  2    8  11  5    4  5
 of          =   1  2    8  11  5    4  5
```

Říká nám to, že jsme se pokusili použít dvojici a trojici ve stejném seznamu, k čemuž by nemělo dojít. Stejně jako nemůžete vytvořit seznam jako `[(1,2),("One",2)]`, protože první prvek v seznamu je dvojice čísel a druhý je dvojice sestávající se z řetězce a čísla. N-tice mohou být použity k vyjádření rozmanitých druhů dat. Kupříkladu pokud chceme v Haskellu vyjádřit

někoho jméno a jeho věk, můžeme použít trojici: `("Christopher", "Walken", 55)`. Na tomto příkladu můžete vidět, že n-tice mohou obsahovat i seznamy.

Použijte n-tice, pokud předem víte, kolik složek budete na data potřebovat. N-tice jsou mnohem méně tvárné, protože se od počtu složek odvíjí jejich typ, takže nelze napsat obecnou funkci na přidání prvku do n-tice — musí se napsat funkce zvlášť pro dvojici, funkce pro trojici, pro čtveřici atd.

I když existuje jednoprvkový seznam, neexistuje věc jako n-tice s jednou složkou. Nedává to moc velký smysl, když se nad tím zamyslíte. Jednosložková n-tice by byla pouze hodnota, kterou by obsahovala, což by pro nás nemělo žádný přínos.

Stejně jako seznamy, n-tice se dají porovnávat, pokud jsou její složky porovnatelné. Nedají se ovšem porovnávat dvě n-tice rozdílné velikosti, zatím co je možné porovnávat dva rozdílně dlouhé seznamy. Dvě užitečné funkce, které pracují se dvojicemi:

vezme dvojici a vrátí její první složku.

```
ghci>      8 11
8
ghci>      "Wow"
"Wow"
```

vezme dvojici a vrátí její druhou složku. Jaké překvapení!

```
ghci>      8 11
11
ghci>      "Wow"
```

Poznámka: tyto funkce pracují pouze se dvojicemi. Nebudou fungovat na trojicích, čtveřicích, pěticích atd. Dostaneme se k jiným způsobům získávání dat z n-tic později.

Skvělá funkce, která vytváří seznam dvojic: . Vezme dva seznamy a poté je sepne dohromady do seznamu spojením odpovídajích prvků do dvojic. Je to opravdu jednoduchá funkce, ale má hromadu použití. Je zvláště užitečná pro kombinaci nebo propojení dvou seznamů. Následuje názorná ukázka.

```
ghci>      1 2 3 4 5   5 5 5 5 5
  1 5   2 5   3 5   4 5   5 5
ghci>      1    5   "jedna"  "dva"  "tři"  "čtyři"  "pět"
  1 "jedna"   2 "dva"   3 "tři"   4 "čtyři"   5 "pět"
```

Funkce spáruje prvky a vytvoří z nich nový seznam. První prvek s první, druhý s druhým atd. Všimněte si, že jelikož dvojice může obsahovat různorodé typy složek, `zip` taktéž může vzít dva typově různé seznamy a sepnout je dohromady. Co se stane, když délka seznamů nesouhlasí?

```
ghci>      5 3 2 6 2 7 2 5 4 6 6   "já" "jsem" "želva"
  5 "já"   3 "jsem"   2 "želva"
```

Delší seznam se jednoduše ořízl, aby měl stejnou délku jako kratší. Protože je Haskell líný, můžeme párovat konečné seznamy s nekonečnými:

```
ghci>      1     "jablko"  "pomeranč"  "třešeň"  "mango"
  1 "jablko"   2 "pomeranč"   3 "třešeň"   4 "mango"
```

Zde je úloha, která kombinuje n-tice a generátor seznamu: jaký pravoúhlý trojúhelník s celočíselnými stranami má všechny strany rovné nebo menší než 10 a jeho obvod je 24? Nejprve zkusíme vypsat všechny trojúhelníky se stranami rovnými nebo menšími než 10:

```
ghci> let          =          |   <-  1  10     <-  1  10     <-  1  10
```

Vygenerovali jsem si čísla ze tří seznamů a zkombinovali jsme je v naši výstupní funkci do trojic. Pokud v GHCi zadáte příkaz `triangles`, dostanete seznam všech možných trojúhelníků se stranami menšími nebo rovnými 10. Dále přidáme podmínku, že to musí být pravoúhlý trojúhelník. Taktéž upravíme tuto funkci přihlédnutím k faktu, že strana *b* není větší než přepona a že strana *a* není větší než strana *b*.

```
ghci> let              =          |   <-  1  10     <-  1       <-  1       2 +   2 ==   2
```

Už jsme skoro hotovi. Teď jen změníme funkci prohlášením, že chceme jenom trojúhelníky s obvodem 24.

```
ghci> let              =          |   <-  1  10     <-  1       <-  1       2 +   2 ==   2
    == 24
ghci>
  6 8 10
```

A tady máme naši odpověď! Tohle je častý postup ve funkcionálním programování. Vezmete si počáteční množinu možných řešení a poté ji přetvářítе a aplikujete filtry, dokud nezískáte správné řešení.

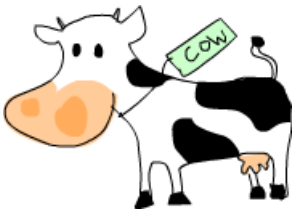
Úvod Obsah Typy a typové třídy

← Začínáme Obsah Syntaxe ve funkcích →

Typy a typové třídy

English version

Věřte typům



Již jsme zmínili, že Haskell má statický typový systém. Typ každého výrazu je znám už v době překladu, což vede k bezpečnějšímu kódu. Pokud napíšete program, kde se pokusíte dělit booleovský typ číslem, ani se ho nepodaří přeložit. To je dobré, protože je lepší odchytávat chyby tohoto druhu v čase překladu než aby program havaroval. Všechno v Haskellu má svůj typ, takže toho překladač ví o vašem programu celkem hodně, než ho vůbec začne překládat.

Na rozdíl od Javy nebo Pascalu má Haskell odvozování typů. Pokud napíšete číslo, nemusíte Haskellu říkat, že to je číslo. Může si ho *odvodit* sám, takže nemusíte explicitně vypisovat typy svých funkcí a výrazů pro jejich funkčnost. Zabývali jsme se základy Haskellu a na typy jsme se podívali jenom zběžně. Avšak porozumění typovému systému je velmi důležitá součást učení se Haskellu.

Typ je něco jako štítek, který je na každém výrazu. Říká nám, do jaké kategorie věcí výraz patří. Výraz `True` je booleovský, `"ahoj"` je řetězec apod.

A teď použijeme GHCi na zjištění typu nějakých výrazů. Učiníme tak pomocí příkazu `:t`, za který stačí napsat platný výraz pro zjištění jeho typu. Tak to teda prubneme.

```
ghci>    'a'
'a' ::
ghci>
     ::
ghci>    "NAZDAR!"
"NAZDAR!" ::
ghci>           'a'
       'a'  ::
ghci>    4 == 5
4 == 5 ::
```

Zde vidíme, že napsáním `:t` a výrazu vypíše zadaný výraz, následovaný `::` a jeho typem. Čtyři tečky `::` se čtou jako „má typ“. Explicitní typy jsou vždy označovány tak, že mají počáteční písmeno velké. Výraz `'a'`, jak můžeme vidět, má typ `Char`. Není těžké usoudit, že se jedná o znak (*character*). Výraz `True` je typu `Bool`. To dává smysl. Ale co je tohle? Prozkoumání typu výrazu `"NAZDAR!"` vypsalo `[Char]`. Hranaté závorky symbolizují seznam. Takže to čteme jako *seznam znaků*. Na rozdíl od seznamů, každá n-tice určité délky (arity) má svůj vlastní typ. Takže výraz `(True, 'a')` má typ `(Bool, Char)`, kdežto výraz jako `('a','b','c')` by měl mít typ `(Char, Char, Char)`. Výraz `4 == 5` bude vždycky vracet `False`, tedy je typu `Bool`.

Funkce také mají typy. Když si píšeme vlastní funkci, můžeme se rozhodnout jí explicitně deklarovat typ. To je obecně považováno za dobrý zvyk, pokud se ale nejedná o velmi krátkou funkci. Zkusíme deklarovat typ funkcím, jež jsme zatím vytvořili. Pamatujete si na ten generátor seznamu, který filtroval řetězec, aby z něj zůstala pouze velká písmena? Tady je ukázáno, jak vypadá s typovou deklarací.

```
::        ->
  =     |   <-       `elem`   'A'   'Z'
```

Funkce `removeNonUppercase` má typ `[Char] -> [Char]`, což znamená, že se řetězec zobrazí na řetězec. Je tomu tak, protože vezme řetězec jako parametr a vrátí jiný jako výsledek. Typ `[Char]` má synonymum `String`, takže je srozumitelnější napsat `removeNonUppercase :: String -> String`. Nemusíme psát k této funkci její typ, protože si překladač může sám odvodit, že se jedná o funkci z řetězce do řetězce, ale přesto jsme to udělali. Ale jak zapíšeme typ funkce, která požaduje několik parametrů? Zde je jednoduchá funkce, jež vezme tři celá čísla a sečte je:

```
::      ->      ->      ->
=    +    +
```

Parametry jsou odděleny šipkou `->` a tím pádem nejsou parametry a návratový typ explicitně odlišeny. Návratový typ je poslední položka v deklaraci a parametry jsou tady ty první tři položky. Později uvidíme, proč jsou všechny pouze odděleny `->`, místo aby bylo více zřejmé oddělení parametrů od návratových typů, jako třeba `Int, Int, Int -> Int` nebo podobně.

Pokud chcete napsat ke své funkci typovou deklaraci, ale nejste si jistí, jaká by měla být, můžete vždycky funkci napsat bez ní a ověřit si to pomocí `:t`. Funkce jsou taktéž výrazy, takže `:t` bude s funkcemi fungovat bez problémů.

Tady je přehled několika běžných typů.

zastupuje celá čísla. Třeba `7` může být `Int`, ale `7.2` ne. Typ `Int` je ohraničený, což znamená, že má minimální a maximální hodnotu. Na 32bitových počítačích je obvykle maximum hodnoty typu `Int` 2147483647 a minimum -2147483648.

zastupuje, ehm… také celá čísla. Hlavní rozdíl je v tom, že není ohraničený, takže může být použit pro vyjádření fakt fakt velkých čísel. Tím myslím fakt velkých. Nicméně typ `Int` je efektivnější.

```
::          ->
=           1
```

```
ghci>           50
304140932017133780436126081660647688443776415689605120000000000000
```

je reálné číslo s plovoucí desetinnou čárkou.

```
::        ->
= 2 *     *
```

```
ghci>               4.0
25.132742
```

je reálné číslo s plovoucí desetinnou čárkou a větší přesností!

```
::        ->
= 2 *     *
```

```
ghci>               4.0
25.132741228718345
```

je booleovský (logický) typ. %000 nabývat pouze dvou hodnot: `True` and `False`.

zastupuje znak. Znak se zapisuje mezi dvě jednoduché uvozovky. Seznam znaků je řetězec.

N-tice mají také svůj typ, ale ten závisí na jejich velikosti a typu jednotlivých složek, takže může být teoreticky nekonečně typů n-tic, což je víc než můžeme popsat v tomhle tutoriálu. Všimněte si, že prázdná n-tice je také typ, který může nabývat pouze jedné hodnoty: `()`.

Typové proměnné

Jaký si myslíte že je typ funkce `head`? Funkce `head` vezme seznam věcí libovolného typu a vrátí první prvek, takže jaký by to mohl být typ? Podívejme se na to!

```
ghci>
     ::      ->
```

Hmmm! Co je to `a`? Je to typ? Vzpomeňte si, že jsme předtím tvrdili, že typy se zapisují velkým počátečním písmenem, takže to není zrovna typ. Protože to není napsáno velkým písmenem, je to ve skutečnosti **typová proměnná**. Což znamená, že `a` může být jakéhokoliv typu. Je to podobné jako generika v jiných jazycích, jenomže haskellová typová proměnná je mnohem užitečnější, protože nám umožňuje jednoduše psát obecné funkce, pokud není potřeba určitých typových specifik. Funkce, které obsahují typové proměnné, se nazývají **polymorfní funkce**. Typová deklarace funkce `head` uvádí, že vezme seznam libovolného typu a vrací jeden prvek stejného typu.

I když typové proměnné mohou mít názvy delší než jeden znak, obvykle je pojmenováváme a, b, c, d…

Pamatujete si funkci `fst`? Vrátí první složku dvojice. Prozkoumejme její typ

```
ghci>
     ::         ->
```

Vidíme, že `fst` vezme n-tici, jež obsahuje dva typy a vrátí prvek stejného typu, jaký má první složka. To je důvod, proč můžeme použít `fst` na dvojici, která obsahuje jakékoliv dva typy. Všimněte si, že ačkoliv jsou `a` a `b` různé typové proměnné, nemusí mít rozdílný typ. Pouze to uvádí, že je typ první složky a návratové hodnoty stejný.

Základy typových tříd

Typová třída je druh rozhraní, které definuje nějaké chování. Pokud je typ součástí nějaké typové třídy, znamená to, že podporuje a implementuje chování, jež ta typová třída definuje. Hodně lidí, co někdy programovalo v objektově orientovaných jazycích, je zmatených, protože si myslí, že jsou stejné jako objektové třídy. No, nejsou. Můžete je považovat za taková lepší javová rozhraní.

Jaký typ má funkce ==?

```
ghci>
     ::         =>    ->    ->
```

Poznámka: operátor rovnosti == je funkce. Stejně jako +, *, -, / a skoro všechny operátory. Když se funkce skládá pouze ze zvláštních znaků, je obvykle považována za infixovou. Pokud se chceme podívat na její typ, předat ji jiné funkci nebo ji zavolat prefixově, musíme ji obklopit kulatými závorkami.

Zajímavé. Vidíme tu novou věc, symbol =>. Údaje před symbolem => se nazývají **typová omezení**. Můžeme přečíst předchozí deklaraci typu jako: funkce rovnosti vezme dvě libovolné hodnoty, které jsou stejného typu, a vrátí `Bool`. Typ těchto dvou hodnot musí být instancí třídy `Eq` (to bylo typové omezení).

Typová třída `Eq` poskytuje rozhraní pro testování rovnosti. Každý typ, u něhož dává smysl testovat dvě jeho hodnoty na rovnost, by měl být instancí třídy `Eq`. Všechny standardní haskellové typy s výjimkou IO (typ, který obstarává vstup a výstup) a funkcí jsou součástí typové třídy `Eq`.

Funkce `elem` je typu `(Eq a) => a -> [a] -> Bool`, protože využívá funkci == v seznamu, aby ověřila, jestli seznam obsahuje požadovanou hodnotu.

Některé základní typové třídy:

je použita pro typy podporující testování rovnosti. Funkce, implementované v této třídě, jsou == a /=. Takže pokud je u nějaké typové proměnné omezení třídou `Eq`, funkce používá ve své definici operátor == nebo /=. Všechny typy, jež jsme zmínili předtím, kromě funkcí, jsou součástí `Eq`, takže mohou být testovány na rovnost.

```
ghci> 5 == 5
ghci> 5 /= 5
ghci> 'a' == 'a'
ghci> "Ho, ho" == "Ho, ho"
ghci> 3.432 == 3.432
```

je typová třída podporující porovnávání. Je určena pro typy, na nichž je definováno uspořádání.

```
ghci>
    ::          =>    ->    ->
```

Všechny zatím probrané typy, kromě typů funkcí, jsou součástí třídy `Ord`. Typová třída `Ord` pokrývá standardní porovnávací funkce jako jsou >, <, >= a <=. Funkce `compare` vezme dvě instance třídy `Ord` stejného typu a vrátí jejich uspořádání. Pro uspořádání je určeny typ , který může nabývat hodnot `GT`, `LT` nebo `EQ`, které znamenají (v tomto pořadí) *větší než*, *menší než* a *rovný*.

Aby mohl být typ instancí `Ord`, musí nejprve patřit do prestižní a exkluzivní třídy `Eq`.

```
ghci> "Abrakadabra" < "Zebra"
ghci> "Abrakadabra" `compare` "Zebra"
ghci> 5 >= 2
ghci> 5 `compare` 3
```

Instance třídy může být převedena do řetězce. Všechny zatím probrané typy, kromě typů funkcí, jsou součástí třídy `Show`. Nejpoužívanější funkce, jež je zahrnutá v typové třídě `Show`, je `show`. Vezme hodnotu, která je instancí typu `Show` a převede ji na řetězec.

```
ghci>      3
"3"
ghci>      5.334
"5.334"
ghci>
"True"
```

je něco jako opačná typová třída k `Show`. Funkce `read` vezme řetězec a vrátí typ, který je instancí třídy `Read`.

```
ghci>       "True"

ghci>       "8.2" + 3.8
12.0
ghci>       "5" - 2
3
ghci>       "[1,2,3,4]" ++  3
 1 2 3 4 3
```

Zatím v pohodě. Opět jsou všechny typy zahrnuty v této typové třídě. Ale co se stane, jestliže zkusíme napsat `read "4"`?

```
ghci>       "4"
              1 0
              type              in
                                  of                          1 0 7
                        type                              type
```

GHCi se nám snaží sdělit, že neví, jakou hodnotu chceme vrátit. Všimněte si, že jsme v předchozích příkladech s `read` později něco dělali s výsledkem. Pomocí toho GHCi mohlo odvodit, jaký druh výsledku chceme dostat z funkce `read`. Pokud by to byl booleovský typ, GHCi by to vědělo a vrátilo by to jako `Bool`. Ale tady pouze ví, že chceme nějaký typ, který je součástí třídy `Read`, jenže neví, jaký. Podívejme se blíže na typ funkce `read`.

```
ghci>
     ::           =>         ->
```

Vidíte? Vrátí typ, jenž je součástí `Read`, jenomže pokud ho nepoužijeme později, nebude mít možnost zjistit, jaký typ to je. To je důvod, proč bychom měli použít explicitní **typovou anotaci**. Typová anotace je způsob konkrétního určení typu nějakého výrazu. Uděláme to přidáním čtyř teček `::` za výraz a poté uvedením typu. Sledujte:

```
ghci>       "5" ::
5
ghci>       "5" ::
5.0
ghci>        "5" ::         * 4
20.0
ghci>       "[1,2,3,4]" ::
 1 2 3 4
ghci>       "(3, 'a')" ::
 3  'a'
```

U většiny výrazů může překladač typ odvozovat sám. Ale někdy překladač neví, zda-li má vrátit kupříkladu typ `Int` nebo `Float` pro výraz jako `read "5"`. Protože je Haskell staticky typovaný jazyk, musí vědět všechny typy před tím, než je kód zkompilován (nebo v případě GHCi interpretován). Takže musíme říct Haskellu: „Hej, tenhle výraz má takovýhle typ, pro případ, že bys to nevěděl!“

Instance třídy jsou sekvenčně seřazené typy — mohou být vyjmenovány. Hlavní výhoda spočívá v tom, že třída `Enum` může být použita v rozsazích. Má definovány následníky a předchůdce, které můžeme dostat pomocí funkcí `succ` a `pred`. Typy, jenž jsou zahrnuty do této třídy: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` a `Double`.

```
ghci>  'a'  'e'
"abcde"
ghci>

ghci>  3    5
 3 4 5
ghci>       'B'
'C'
```

Instance třídy mají horní a spodní ohraničení.

```
ghci>          ::
 2147483648
ghci>          ::
'\1114111'
ghci>          ::

ghci>          ::
```

Funkce `minBound` a `maxBound` jsou zajímavé, protože mají typ `(Bounded a) => a`. Jsou v jistém smyslu polymorfní konstanty.

Všechny n-tice jsou součástí třídy `Bounded`, pokud do ní patři i jednotlivé složky.

```
ghci>          ::
      2147483647 '\1114111'
```

je numerická typová třída. Její instance mají tu vlastnost, že se chovají jako čísla. Podívejme se na typ čísla.

```
ghci>    20
20 ::         =>
```

Vypadá to, že celá čísla jsou také polymorfní konstanty. Mohou se chovat jako typ, jenž je instancí typové třídy `Num`.

```
ghci> 20 ::
20
ghci> 20 ::
20
ghci> 20 ::
20.0
ghci> 20 ::
20.0
```

Toto jsou typy z typové třídy `Num`. Jestliže se podíváme na typ operátoru `*`, uvidíme, že akceptuje veškerá čísla.

```
ghci>
    ::         =>   ->   ->
```

Vezme dvě čísla stejného typu a vrátí výsledek, jenž má shodný typ. To je důvod, proč vyhodnocení výrazu `(5 :: Int) * (6 :: Integer)` skončí typovou chybou, kdežto `5 * (6 :: Integer)` bude fungovat a výsledek bude typu `Integer`.

Aby se typ mohl přidat do `Num`, musí být už instancí tříd `Show` a `Eq`.

je také numerická typová třída. Narozdíl od `Num`, která zahrnuje všechna čísla včetně reálných a celých, `Integral` obsahuje pouze celá čísla. V této typové třídě jsou typy `Int` a `Integer`.

Třída obsahuje pouze čísla s plovoucí desetinnou čárkou, tedy `Float` a `Double`.

Pro zacházení s čísly je velmi užitečná funkce . Má deklarovaný typ `fromIntegral :: (Num b, Integral a) => a -> b`. Podle jejího typového omezení můžeme vidět, že vezme celé číslo a udělá z něj obecnější číslo. To je užitečné když chceme, aby spolupracovala celá a desetinná čísla. Například funkce `length` je typu `length :: [a] -> Int`, místo aby byla obecnějšího typu `(Num b) => length :: [a] -> b`. Řekl bych, že je to z historických důvodů nebo tak něco a zdá se mi to celkem hloupé. Každopádně, jestliže zkusíme zjistit délku seznamu a poté ji přičíst třeba k `3.2`, dostaneme chybu, protože se

snažíme spojit dohromady číslo typu `Int` a desetinné číslo. Je potřeba to obejít napsáním `fromIntegral (length [1,2,3,4] ) + 3.2`, což to vyřeší.

Všimněte si, že funkce `fromIntegral` má několik typových omezení v definici typu. To je úplně v pořádku a jak můžete vidět, typová omezení se v kulatých závorkách oddělují čárkami.

Začínáme Obsah Syntaxe ve funkcích

← Typy a typové třídy Obsah Rekurze →

Syntaxe ve funkcích

English version

Vzory

Tato kapitola se bude týkat některých užitečných syntaktických konstruktů a nejprve se pustíme do vzorů (pattern matching). Vzory se sestávají z určitých schémat, kterým mohou data odpovídat a poté se ověřuje, jestli ano, a podle těchto schémat se data dekonstruují.

Při definování funkce je možnost napsat samostatně tělo funkce pro jiný vzor. Což vede k velice čistému kódu, který je jednoduchý a čitelný. Vzory se dají použít u jakéhokoliv datového typy — čísla, znaky, seznamy, n-tice atd. Vytvořme si opravdu triviální funkci, která ověřuje, jestli je zadané číslo sedmička nebo ne.

```
::              =>   ->
7 = "ŠŤASTNÉ ČÍSLO SEDM!"
  = "Je mi líto, máš pech, kámo!"
```

Když zavoláte funkci `lucky`, schéma se bude ověřovat shora dolů a pokud bude sedět, použije se odpovídající tělo funkce. Jediné číslo, jež může odpovídat prvnímu vzoru, je číslo 7. Pokud není zadáno, přejde se na druhý vzor, který zachytí cokoliv a spojí to s `x`. Tahle funkce může být implementována použitím výrazu if. Ale co když chceme funkci, která vypíše číslo od jedničky po pětku a napíše `Není mezi 1 a 5.` pro ostatní čísla? Bez vzorů bychom museli vytvořit celkem spletitý strom z if, then a else. Avšak se vzory:

```
::              =>   ->
1 = "Jedna!"
2 = "Dva!"
3 = "Tři!"
4 = "Čtyři!"
5 = "Pět!"
  = "Není mezi 1 a 5."
```

Všimněte si, že kdybychom přesunuli poslední vzor (obecný, který zachytí všechno) úplně nahoru, tak by funkce vždycky vypsala `Není mezi 1 a 5.`, protože by zachytil všechna čísla a nebyla by šance, že by se pokračovalo dál a přešlo na další vzory.

Pamatujete si na funkci `factorial`, jež jsme implementovali předtím? Definovali jsme faktoriál čísla `n` jako `product [1..n]`. Můžeme také definovat faktoriál *rekurzivně*, způsobem, jakým se obvykle definuje v matematice. Začneme tvrzením, že faktoriál nuly je jednička. Pak uvedeme, že faktoriál každého přirozeného čísla je to číslo vynásobené faktoriálem jeho předchůdce. Takhle to vypadá přeložené do jazyka Haskellu.

```
::              =>   ->
0 = 1
  =   *             - 1
```

Poprvé jsme tu definovali funkci rekurzivně. Rekurze je v Haskellu důležitá a my se na ni podíváme později podrobněji. Ale zatím si v rychlosti ukážeme, co se děje při výpočtu faktoriálu řekněme trojky. Pokusí se vyhodnotit `3 * factorial 2`. Faktoriál dvojky je `2 * factorial 1`, takže zatím máme `3 * (2 * factorial 1)`. Rozepsaný `factorial 1` je `1 * factorial 0`, takže %0000ente `3 * (2 * (1 * factorial 0))`. A teď přichází trik — definovali jsme faktoriál nuly, že je jedna, a protože se

narazí na vzor před tím obecným, jednoduše vrátí jedničku. Takže se finální forma podobá `3 * (2 * (1 * 1))`. Kdybychom napsali druhý vzor před první, tak by zachytil všechna čísla včetně nuly a náš výpočet by nikdy neskončil. To je důvod, proč je pořadí vzorů důležité a je vždycky lepší uvádět dříve vzory pro konkrétní hodnoty a obecné až na konec.

Ověřování vzorů může také selhat. Jestliže si definujeme takovouto funkci:

```
::      ->
'a' = "Albert"
'b' = "Bedřich"
'c' = "Cecil"
```

a poté bude zavolána se vstupem, který jsme neočekávali, stane se tohle:

```
ghci>          'a'
"Albert"
ghci>          'b'
"Bedřich"
ghci>          'h'
                    53 0   55 21                          in
```

GHCi si oprávněně stěžuje, že nemáme definovány vzory kompletně. Při vytváření vzorů bychom nikdy neměli zapomenout přidat obecný vzor, aby náš program nepadal po zadání neočekávaném vstupu.

Vzory mohou být použity také s n-ticemi. Co když chceme vytvořit funkci, jež vezme dva vektory ve dvoudimenzionálním prostoru (tedy ve formě dvojic) a sečte je dohromady. Při sčítání vektorů sečteme odděleně jejich xové složky a pak jejich ypsilonové složky. Zde je ukázáno, jak bychom toho mohli dosáhnout, kdybychom nevěděli o vzorech:

```
::         =>         ->         ->
    =         +              +
```

Prima, funguje to, ale je lepší způsob, jak to udělat. Upravíme tu funkci, aby používala vzory.

```
::         =>         ->         ->
                  =     +         +
```

A je to! Mnohem lepší. Všimněte si, že tohle je už obecný vzor. Typ funkce `addVectors` (v obou příkladech) je `addVectors :: (Num a) => (a, a) -> (a, a) - > (a, a)`, takže můžeme zaručit, že dostaneme dvě dvojice jako parametr.

Funkce `fst` a `snd` získají složky dvojic. Ale co trojice? No, není na to standardní funkce, ale můžeme si napsat vlastní.

```
::          ->
        =

 ::          ->
      =

::          ->
        =
```

Znak _ představuje to stejné co představoval u generátorů seznamu. Což znamená, že se vůbec nestaráme o hodnotu v té části, takže prostě napíšeme _.

Což mi připomíná, že se vzory dají použít i v generátorech seznamu. Sledujte:

```
ghci> let    =   1 3    4 3    2 4    5 3    5 6    3 1
ghci>      |       <-
```

```
4 7 6 8 11 4
```

Při nesplnění vzoru se prostě přejde na další prvek.

Samotné seznamy mohou být také použity jako vzory. Můžete ověřit prázdný seznam `[]` nebo jakýkoliv vzor, který obsahuje dvojtečku `:` a prázdný seznam. Protože je `[1,2,3]` jenom syntaktický cukr pro `1:2:3:[]`, dá se použít prvně uvedený vzor. Vzor jako `x:xs` spojí první prvek ze seznamu s `x` a zbytek s `xs`, i když by byl seznam jednoprvkový, to by se z `xs` stal prázdný seznam.

Poznámka: vzor `x:xs` se používá často, hlavně v rekurzivních funkcích. Ale vzory, obsahující `:`, se dají použít jenom v seznamech délky 1 nebo více.

Pokud chcete spojit s proměnnými, řekněme, první tři prvky a zbytek seznamu s jinou proměnnou, můžete použít něco jako `x:y:z:zs`. Tohle bude fungovat jenom se seznamem, jež má tři prvky a více.

A když teď víme, jak používat vzory se seznamy, napišme si vlastní implementaci funkce `head`.

```
::     ->
  =       "Nemůžeš zjistit první prvek prázdného seznamu, hňupe!"
    =
```

Vyzkoušíme, jestli to funguje:

```
ghci>       4 5 6
4
ghci>      "Nazdar"
'N'
```

Nádhera! Všimněte si, že pokud chceme spojit několik proměnných (i když je nějaká z nich jenom _ a ve skutečnosti se vůbec nespojí), musíme vzor obklopit kulatými závorkami. Také si všimněte funkce `error`, jenž jsme použili. Vezme řetězec a vygeneruje běhovou chybu a použije ten řetězec jako informaci o tom, jaká chyba nastala. To způsobí pád programu, takže není moc dobrý nápad tuto funkci používat příliš často. Jenže zavolání funkce `head` na prázdný seznam nedává smysl.

Vytvořme si triviální funkci, která nám vypíše první prvky ze seznamu v (ne)vhodné formě v češtině.

```
::          =>     ->
  = "Seznam je prázdný."
    = "Seznam obsahuje jeden prvek: " ++
      = "Seznam obsahuje dva prvky: " ++       ++ " a " ++
      = "Seznam je dlouhý. První dva prvky jsou: " ++       ++ " a " ++
```

Tato funkce je bezpečná, protože se vypořádá s prázdným seznamem, jedno- a dvouprvkovým seznamem a se seznamem s více než dvěma prvky. Všimněte si, že vzory `(x:[])` a `(x:y:[])` mohou být zapsány jako `[x]` a `[x,y]` (syntaktický cukr, u něhož nepotřebujeme kulaté závorky). Nemůžeme přepsat vzor `(x:y:_)` na tvar s hranatými závorkami, protože potřebujeme, aby to ověřovalo seznamy délky dva a více.

Již jsme implementovali svou vlastní funkci `length` pomocí generátoru seznamu. Teď na to použijeme vzory a trochu rekurze:

```
::         =>     ->
  = 0
    = 1 +
```

Podobá se to funkci na počítání faktoriálu, jež jsme napsali dříve. Nejprve jsme si definovali výsledek známého vstupu — prázdného seznamu. To je též známo jako *okrajová podmínka*. Poté jsme ve druhém vzoru vzali kus seznamu pomocí rozdělení na první prvek a zbytek. Napsali jsme, že délka seznamu se rovná jedna plus délka zbytku. Je zde použito podtržítko _ na ověření prvního prvku, protože se nezajímáme o konkrétní hodnotu. Také si všimněte, že jsme se vypořádali s každým možným typem seznamu. První vzor ověřuje prázdný seznam a druhý cokoliv, co není prázdný seznam.

Podívejme se, co se stane, jestliže zavoláme funkci `length'` na řetězec `"pes"`. Nejprve se funkce přesvědčí, zda není vstupem prázdný seznam. Protože není, přejde se na druhý vzor. Druhý vzor odpovídá a výraz se přepíše na `1 + length' "es"`, protože se rozdělí na první prvek a zbytek, který se dále zpracuje. Dobrá. Délka `length'` řetězce `"es"` je, podobně, výraz `1 + length' "s"`. Takže teď máme výraz `1 + (1 + length' "s")`. Hodnota výrazu `length' "s"` je `1 + length' ""` (což může být zapsáno jako `1 + length' []`). A my jsme si definovali, že `length' []` bude `0`. Takže nám nakonec zbude výraz `1 + (1 + (1 + 0))`.

Vytvoříme si funkci `sum`. Víme, že součet prázdného seznamu je 0. Napíšeme to jako vzor. A víme také, že součet seznamu je první prvek plus součet zbytku seznamu. Takže když tohle celé zapíšeme, dostaneme:

```
::          =>      ->
   = 0
       =    +
```

Také existuje věc, nazvaná *zástupný vzor*. Je to užitečný způsob, jak rozdělit něco podle vzoru a navázat to na názvy, zatímce stále uchováváme referenci na tu celou věc. Provede se to vložením názvu a zavináče `@` před vzor. Kupříkladu vzor `xs@(x:y:ys)`. Tento vzor bude ověřovat přesně stejnou věc jako `x:y:ys`, jenom můžete přistupovat jednoduše k celému seznamu přes `xs`, aniž byste se museli opakovat a psát znovu `x:y:ys` do těla funkce. Tady je rychlý a hrubý příklad:

```
        ::        ->
        "" = "Prázdný řetězec, jejda!"
                 = "První písmeno řetězce " ++     ++ " je " ++


ghci>         "Drákula"
"První písmeno řetězce Drákula je D"
```

Normálně používáme zástupné vzory, abychom se vyhnuli opakování při ověřováňí složitějších vzorů, když musíme využít celý výraz znovu v těle funkce.

Ještě jedna věc — nemůžete ve vzorech použít operátor ++. Pokud se budete snažit ověřit výraz pomocí vzoru `(xs ++ ys)`, jaký bude první a jaký druhý seznam? Nedává to příliš smysl. Mohlo by dávat smysl, kdybychom chtěli udělat něco jako `(xs ++ [x,y,z])` nebo jenom `(xs ++ [x])`, ale není to možné, už ze samotné podstaty seznamů.

Stráže, stráže!

Zatímco vzory jsou určeny k ujištění, že hodnota vyhovuje určité formě, a k její dekonstrukci, stráže jsou pro testování, jestli je nějaká vlastnost hodnoty (či více hodnot) pravdivá nebo nepravdivá. To zní skoro jako výraz if a opravdu je to velice podobné. Věc se má tak, že stráže jsou mnohem čitelnější, když je těch podmínek více, a chovají se spíše jako vzory.

Místo vysvětlování jejich syntaxe se do toho rovnou pustíme a vytvoříme si funkci s využitím stráží. Napíšeme si jednoduchou funkce, která na vás bude nadávat v závislosti na vašem indexu tělesné hmotnosti (BMI). Hmotnostní index se rovná váze člověka vydělené druhou mocninou jeho výšky. Pokud je index menší než 18,5, je považován za podvyživeného. Jestliže je mezi 18,5 a 25, je považován za normálního. Hodnota mezi 25 a 30 je nadváha a lidé s BMI nad 30 jsou obézní. Takže tady je ta funkce (nebudeme ji teď počítat, tahle funkce pouze vezme BMI a vynadá vám).

```
          ::                 =>    ->

      |     <= 18.5 = "Jsi podvyživený, ty emo, ty!"
      |     <= 25.0 = "Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
      |     <= 30.0 = "Jsi tlustý! Zhubni, špekoune!"
      |            = "Jsi velryba, gratuluji!"
```

Stráž se zadává svislítkem, který následuje za názvem funkce a jejími parametry. Obvykle jsou svislítka o kousek odsazena doprava a zarovnána. Stráže jsou vlastně booleovské výrazy. Jestliže se vyhodnotí jako `True`, je použito odpovídající tělo funkce. Jestliže se vyhodnotí jako `False`, ověřování pokračuje na další stráž a tak dále. Pokud zavoláme tuto funkci s parametrem `24.3`, nejprve se ověří, jestli je menší nebo rovný `18.5`. Protože není, propadne se k další stráži. Ověření se provede u druhé stráže a protože je 24.3 menší než 25.0, je vrácen druhý řetězec.

Tohle celé očividně připomíná velký strom z if a else v imperativních jazycích, jenomže stráže jsou lepší a čitelnější. Zatímco z velkých if a else stromů se nebudete tvářit nadšeně, zvláště když je problém zadaný diskrétním způsobem, ze kterého se těžko přepracovává. Stráže jsou velice příjemnou alternativou.

Poslední stráž bývá častokrát `otherwise`. Výraz `otherwise` je definován jednoduše jako `otherwise = True` a odchytí všechno. Tohle je velice podobné vzorům, s tím rozdílem, že se u nich ověřuje, jestli vstup odpovídá nějakému schématu a u stráží se kontroluje, zda-li vstup vyhovuje booleovským podmínkám. Jestliže se všechny stráže ve funkci vyhodnotí jako `False` (a neposkytli jsme stráž `otherwise`, která odchytí všechno), vyhodnocení přejde na následující **vzor**. Takhle spolu vzory a stráže nádherně spolupracují. Pokud není nalezena vyhovující stráž nebo vzor, vyhodnocování skončí chybou.

Samozřejmě můžeme použít stráže ve funkci, která bere tolik parametrů, kolik chceme. Místo abychom nutili uživatele počítat svůj hmotnostní index před zavoláním funkce, upravíme tuto funkci, tak, že bude požadovat váhu a výšku a vypočítá to pro nás.

```
          ::                 =>    ->    ->

      |         /         ^ 2 <= 18.5 = "Jsi podvyživený, ty emo, ty!"
      |         /         ^ 2 <= 25.0 = "Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
      |         /         ^ 2 <= 30.0 = "Jsi tlustý! Zhubni, špekoune!"
      |                               = "Jsi velryba, gratuluji!"
```

Podívejme se, jestli jsem tlustý…

```
ghci>          85 1.90
"Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
```

Jéje! Nejsem tlustý! Ale Haskell mě nazval šeredným. Co už!

Všimněte si, že se nepoužívá rovnítko `=` za názvem funkce a jejími parametry, před první stráží. Hodně začátečníkům to vypíše chybu v syntaxi, protože ho tam občas vloží.

Další velmi jednoduchý příklad: napišme si vlastní funkci `max`. Určitě si pamatujete, že vezme dvě porovnatelné věci a vrátí větší z nich.

```
        ::          =>    ->    ->

      |   >        =
      |            =
```

Stráže se dají také zapsat jednořádkově, i když to nedoporučuji, protože je to méně čitelné, i u krátkých funkcí. Ale pro demonstraci můžeme napsat `max'` následovně:

```
        ::          =>    ->    ->
```

```
        |   >   =   |          =
```

Fuj! To není moc čitelné! Budeme pokračovat: napíšeme si svou vlastní funkci `compare` pomocí stráží.

```
        ::          =>   ->   ->
`myCompare`
  |   >       =
  |   ==      =
  |           =


ghci> 3 `myCompare` 2
```

Poznámka: funkci můžeme kromě infixového zavolání pomocí zpětných apostrofů také infixově definovat. Někdy je tenhle způsob čitelnější.

Lokální definice pomocí where

V předchozí sekci jsme si definovali funkci na počítání hmotnosti a nadávání. Bylo to nějak takhle:

```
    ::               =>   ->   ->

|       /        ^ 2 <= 18.5 = "Jsi podvyživený, ty emo, ty!"
|       /        ^ 2 <= 25.0 = "Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
|       /        ^ 2 <= 30.0 = "Jsi tlustý! Zhubni, špekoune!"
|                            = "Jsi velryba, gratuluji!"
```

Všimněte si, že tu třikrát opakujeme kód. Třikrát opakujeme kód. Opakování kódu (třikrát) při programování je asi tak žádoucí jako kopačka do hlavy. Jelikož jsme opakovali stejný výraz třikrát, bylo by ideální, kdybychom ho mohli spočítat, navázat na proměnnou a poté ji používat namísto výrazu. Můžeme tedy upravit naši funkci následovně:

```
    ::               =>   ->   ->

|     <= 18.5 = "Jsi podvyživený, ty emo, ty!"
|     <= 25.0 = "Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
|     <= 30.0 = "Jsi tlustý! Zhubni, špekoune!"
|             = "Jsi velryba, gratuluji!"
where     =       /        ^ 2
```

Vložili jsme klíčové slovo `where` za stráže (obvykle je nejlepší ho odsadit stejně jako svislítka) a poté definovat několik názvů nebo funkcí. Tyto definice jsou viditelné všem strážím a mají tu výhodu, že nemusíme opakovat kód. Jestliže jsme se rozhodli, že budeme počítat BMI jinak, stačí nám změnit kód na jednom místě. Také to zlepšuje čitelnost, když pojmenováváme věci a můžeme tím naše programy zrychlit, protože věci jako je tady proměnná `bmi` stačí vypočítat pouze jednou. Mohli bychom jít o kousek dál a přepsat naši funkci takhle:

```
    ::               =>   ->   ->

|     <=        = "Jsi podvyživený, ty emo, ty!"
|     <=        = "Jsi údajně normální. Pche, vsadím se, že jsi šereda!"
|     <=        = "Jsi tlustý! Zhubni, špekoune!"
|               = "Jsi velryba, gratuluji!"
where     =       /        ^ 2
            = 18.5
            = 25.0
            = 30.0
```

Názvy, jež definujeme u funkce v části s where, jsou dostupné pouze v té funkci, takže se nemusíme obávat o zaneřádění jmenných prostorů jiných funkcí. Všimněte si, ze jsou všechny názvy zarovnány do jednoho sloupce. Pokud je pořádně nezarovnáme, Haskell bude zmatený, protože nebude vědět, co je součástí stejného bloku.

Konstrukce where není sdílená v tělu funkce mezi různými vzory. Pokud chcete přistupovat u více vzorů v jedné funkci k nějakým definicím, musíte je definovat globálně.

Je také možné ve where definicích použít **vzory**! Můžeme přepsat sekci s where v naší předchozí funkci na:

```
  where      =          /          ^ 2
                        =   18.5   25.0   30.0
```

Vytvořme si další poctivě triviální funkci, ve které dostaneme někoho jméno a příjmení a vypíšeme jeho iniciály.

```
          ::          ->          ->
                      =       ++ ". " ++       ++ "."
      where         =
                    =
```

Mohli bychom to ověřovat přímo v parametrech funkce (bylo by to ve skutečnosti kratší a zřejmější), ale tohle mělo jenom ukázat, že je možné to udělat taky pomocí where definic.

Stejně jako jsem si definovali konstanty ve where blocích, můžete také definovat funkce. Abychom zůstali u našeho programovacího zdravotního tématu, vytvoříme se funkci, která vezme seznam dvojic vah a výšek a vrátí jejich index hmotnosti.

```
          ::                  =>              ->
            =             |           <-
      where                     =         /         ^ 2
```

A to je všecho, co je potřeba! Důvod, proč jsme v tomto příkladu zavedli `bmi` jako funkci, je protože nemůžeme vypočítat jedno BMI z parametrů funkce. Musíme projít celý seznam předaný funkci a každá dvojice ze seznamu má rozdílné BMI.

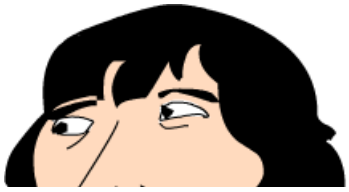
Konstrukce where se mohou také větvit. Je to běžný postup, vytvořit funkci a definovat k ní nějaké pomocné funkce se svými where klauzulemi, a pak těm funkcím vytvořit další pomocné funkce, každou s vlastními where klauzulemi.

… a pomocí let

Velmi podobné konstrukci where je konstrukce let. Where je syntaktický konstrukt, který umožní navázání proměnných na konec funkce a celá funkce k nim může přistupovat, včetně všech stráží. Let vám umožní navázat proměnné kamkoliv a je sama o sobě výrazem, ale je lokálnější, takže se nedostane přes stráže. Stejně jako každý konstrukt v Haskellu, jež se používá k navázání hodnoty na název, konstrukce let mohou být použity pro ověřování vzorů. Podívejme se na ně v činnosti! Takhle bychom mohli definovat funkci, která nám vrátí vypočítaný povrch válce na základě jeho výšky a poloměru:

```
          ::                  =>    ->    ->
              =
      let           = 2 *     *     *
                    =     *     2
      in            + 2 *
```

Podoba zápisu je `let <definice> in <výraz>`. Názvy, které definujete v části s let jsou přístupné výrazu v části za in. Jak můžete vidět, dalo by se to také zapsat pomocí konstrukce where. Všimněte si, že názvy jsou také zarovnány do jednoho sloupce. Takže



jaký je rozdíl mezi těmito dvěma zápisy? Zatím to vypadá, že se u let píše definice jako první a používaný výraz až později, zatímco u where to je naopak.

Rozdíl je v tom, že konstrukce let je sama o sobě výraz. Kdežto where je pouhý syntaktický konstrukt. Pamatujete si, když jsme se zabývali výrazem if a vysvětlovali jsme si, že if a else můžete nacpat téměř kamkoliv?

```
ghci>  if 5 > 3 then "Bla" else "Ble"  if 'a' > 'b' then "Něco" else "Nic"
 "Bla"  "Nic"
ghci> 4 *  if 10 > 5 then 10 else 0  + 2
42
```

Stejnou věc můžete udělat s konstrukcí let.

```
ghci> 4 *  let   = 9 in   + 1  + 2
42
```

Mohou být také použity na zavedení funkcí s lokální působností:

```
ghci>  let         =   *   in        5        3        2
  25 9 4
```

Jestliže chceme definovat několik proměnných na jednom řádku, evidentně je nemůžeme zarovnat do sloupců. Proto je můžeme oddělit pomocí středníků.

```
ghci>  let   = 100    = 200    = 300 in        let     "Hej "      = "ty!" in     ++
 6000000 "Hej ty!"
```

Nemusíte vkládat středník za poslední definici, ale můžete, pokud chcete. Jak jsme již řekli, v konstrukcích let je možnost použít vzory. Je to velice užitečné pro rychlé rozebrání n-tic na složky a navázání na názvy a tak.

```
ghci>  let         =  1 2 3  in        * 100
600
```

Můžete také vložit konstrukci let dovnitř generátoru seznamu. Přepišme si náš předchozí příklad na počítání seznamů dvojic vah a výšek a použijme v něm let v generátoru seznamu místo abychom definovali pomocnou funkci přes where.

```
          ::              =>          ->
            =      |        <-     let     =   /   ^ 2
```

Zařadili jsme let do generátoru seznamu, jako by byl predikát, jenom nefiltruje seznam, ale definuje názvy. Názvy, definované pomocí let v generátoru seznamu jsou viditelné výstupní funkci (část před svislítkem `|`) a všem predikátům a případným částem, které následují po definicích. Takže bychom mohli funkci předělat, aby vracela pouze BMI tlustých lidí:

```
          ::              =>          ->
            =      |        <-     let     =   /   ^ 2      >= 25.0
```

Nemůžeme použít název `bmi` v části s `(w, h) <- xs`, protože je definovaná před konstrukcí let.

Vynechali jsme část s in konstrukce let, když jsme pracovali s generátorem seznamu, protože tam je viditelnost názvů předdefinována. Nicméně jsme mohli použít konstrukci let-in v predikátu a názvy mohli definovat tak, aby byly viditelné pouze

predikátu. Část s in může být také vynechána, když definujeme funkce a konstanty přímo v GHCi. Pokud to uděláme, názvy pak budou viditelné po celý čas interaktivní relace.

```
ghci> let            =   *   +
ghci>      3 9 2
29
ghci> let            =   *   +   in      3 4 2
14
ghci>
             1 0      in
```

Když je konstrukce let tak skvělá, proč bychom ji nemohli použít všude namísto where, ptáte se? No, protože je konstrukce let výraz a je poctivě lokální ve své působnosti, nemůže být použita mezi strážemi. Někteří lidé mají raději konstrukci where, protože definice následují za funkcí, ve které se používají. V tomto zápisu je tělo funkce blíže názvu funkce a typové deklaraci, takže to je pro některé čitelnější.

Podmíněný výraz case

Mnoho imperativních jazyků (C, C++, Java apod.) mají case syntaxi a pokud jste v nějakém z nich programovali, pravděpodobně víte, co to je. Funguje to tak, že se vezme proměnná a potom se provedou bloky kódu pro určenou hodnotu té proměnné a je možnost na konec přidat blok, který zachytí cokoliv, pro případ, že by proměnná nabyla hodnoty, se kterou jsme nepočítali.

Haskell bere tento koncept a rozšiřuje ho. Jak název napovídá, výrazy case jsou, no, výrazy, podobně jako výraz if a konstrukce let. Nejenom že umí vyhodnocovat výrazy podle možných případů, ve kterých proměnná nabývá určitých hodnot, můžeme také vyhodnocovat na základě vzorů. Hmmm, vzít proměnnou, ověřit podle vzoru, vyhodnotit kus kódu na podle jeho hodnoty, kde jsme to už slyšeli? No jasně, ověřování vzorů podle parametrů v definici funkce! Takže to je ve skutečnosti pouhý syntaktický cukr pro case výraz. Tyhle dva kusy kódy dělají tu stejnou věc a jsou zaměnitelné:

```
::     ->
   =       "Prázdný list nemá první prvek!"
       =
```

```
::     ->
   = case    of       ->       "Prázdný list nemá první prvek!"
                       ->
```

Jak můžete vidět, syntaxe výrazu case je pěkně jednoduchá:

```
case       of      ->
                   ->
                   ->
```

Obsah části `výraz` je ověřován vzory. Postup je stejný, jaký bychom čekali: je použit první vzor, který sedí. Jestliže pokračuje přes celé case a není nalezen vhodný vzor, nastane běhová chyba.

Zatímco vzory u parametrů funkcí mohou být použity pouze při definování těchto funkcí, case výrazy mohou být použity víceméně všude. Kupříkladu:

```
            ::     ->
               = "Seznam je " ++ case    of     -> "prázdný."
                                                 -> "jednoprvkový."
```

```
                                -> "víceprvkový."
```

To je užitečné pro ověřování něčeho uprostřed zápisu výrazu. Protože je ověřování v definici funkce syntaktický cukr pro výraz case, mohli bychom to rovněž definovat takto:

```
            ::      ->
               = "Seznam je " ++
    where            = "prázdný."
                     = "jednoprvkový."
                     = "víceprvkový."
```

Typy a typové třídy Obsah Rekurze

← Syntaxe ve funkcích Obsah Funkce vyššího řádu →

Rekurze

English version

Ahoj, rekurze!

Rekurzi jsme stručně zmínili v předchozí kapitole. V této kapitole se na ni podíváme zblízka, proč je v Haskellu tak důležitá a jak můžeme najít velice výstižná a elegantní řešení problémů díky rekurzivnímu myšlení.

Pokud stále ještě nevíte, co to rekurze je, přečtěte si tuhle větu. Ha, ha! Dělám si legraci! Rekurze je ve skutečnosti způsob definování funkce, ve kterém je použita tatáž funkce ve své vlastní definici. Definice v matematice jsou často zadány rekurzivně. Například fibonacciho posloupnost je definována rekurzivně. Nejprve si definujeme první dvě fibonacciho čísla nerekurzivně. Řekneme, že *F(0) = 0* a *F(1) = 1*, což znamená, že nulté a první fibonacciho číslo je nula a jednička (v tomto pořadí). Poté prohlásíme, že pro ostatní přirozená čísla je fibonacciho číslo součet dvou předchozích fibonacciho čísel. Takže *F(n) = F(n-1) + F(n-2)*. Tím pádem *F(3)* je *F(2) + F(1)*, což je *(F(1) + F(0)) + F(1)*. Protože jsme se dostali k nerekurzivně definovaným fibonacciho číslům, můžeme bezpečně říct, že *F(3)* se rovná 2. Nerekurzivně definované části rekurzivních definicí (stejně jako bylo tady *F(0)* a *F(1)*) se také říká **okrajová podmínka** a je dost důležitá, jestliže chceme, aby naše rekurzivní funkce někdy skončila. Kdybychom neměli definované *F(0)* a *F(1)* nerekurzivně, nikdy bychom nedostali řešení pro jakékoliv číslo, protože bychom po dosažení nuly šli do záporných čísel. Z ničeho nic bychom tvrdili, že *F(-2000)* je *F(-2001) + F(-2002)* a konec by byl stále v nedohlednu!

Rekurze je v Haskellu důležitá, protože narozdíl od imperativních jazyků provádíme výpočty deklarováním něčeho, jaké to *je*, místo deklarování *jak* to dostat. To je důvod, proč v Haskellu nejsou žádné while nebo for smyčky a místo toho jsme museli mnohokrát použít rekurzi pro deklaraci něčeho, jaké to je.

Maximální skvělost

Funkce `maximum` vezme seznam věcí, které se dají uspořádat (např. instance typové třídy `Ord`) a vrátí největší z nich. Přemýšlejte, jak byste to napsali imperativním stylem. Pravděpodobně byste si vytvořili proměnnou, aby uchovávala aktuální maximální hodnotu a poté byste postupně procházeli seznam a pokud by byl prvek v seznamu větší než současná nejvyšší hodnota, nahradili byste to tím prvkem. Maximální hodnota, která by zbyla na konci, by byla výsledek. Páni! To jsme použili celkem dost slov na popsání tak jednoduchého algoritmu!

A teď se podíváme, jak bychom to definovali rekurzivně. Mohli bychom nejprve vytvořit okrajovou podmínku a prohlásit, že maximum z jednoprvkového seznamu se rovná tomu prvku z něj. Pak můžeme říct, že maximum z delšího seznamu je první prvek, pokud je ten prvek větší než maximum ze zbytku. Jestliže je maximum ze zbytku větší, tak je výsledek maximum ze zbytku. Takhle to je! A teď si to napíšeme v Haskellu.

```
        ::          =>      ->
          =         "maximum z prázdného seznamu"
            =

    |   >           =
    |             =
    where           =
```

Jak můžete vidět, vzory se dobře snášejí s rekurzí! Většina imperativních jazyků neobsahuje vzory, takže musíte vytvořit hromadu if a else na otestování okrajových podmínek. Zde je jednoduše zahrneme do vzorů. Tedy první okrajová podmínka

říká, že jestliže je seznam prázdný, spadni! To dává smysl, protože co je maximum z prázdného seznamu? Nevím. Druhý vzor také zastává okrajovou podmínku. Říká, že jestliže to je jednoprvkový seznam, prostě vrať ten jediný prvek.

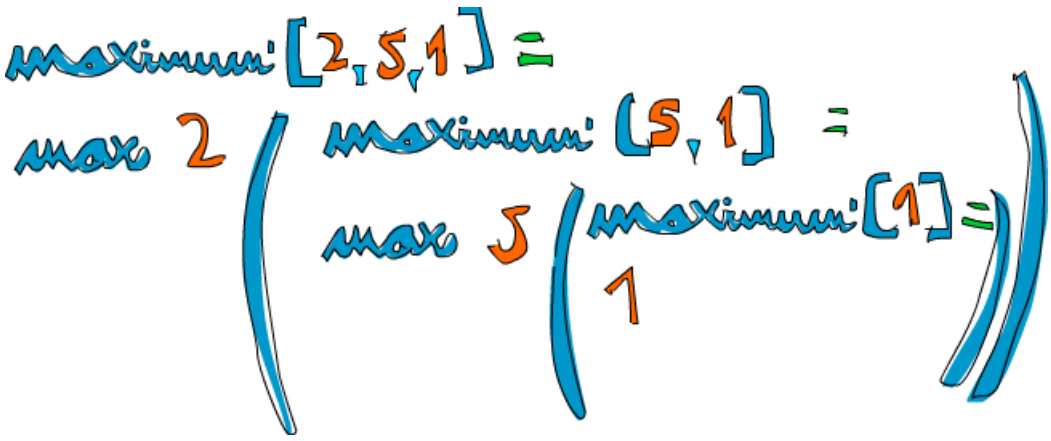
A třetí vzor je ten, ve kterém se všechno děje. Použili jsme vzor na rozdělení seznamu na první prvek a zbytek. Tohle je běžný postup při provádění rekurze na seznamech, na který je potřeba si zvyknout. Použijeme konstrukci *where* pro definování maxima zbytku seznamu `maxTail`. Poté zkontrolujeme, jestli je první prvek větší než maximum ze zbytku seznamu. Jestliže je, vrátíme ho. V opačném případě vrátíme maximum ze zbytku seznamu.

Zkusíme vzít na ukázku nějaký seznam čísel, třeba `[2,5,1]`, a zjistit, jak by to s ním mohlo fungovat. Jestliže na něj zavoláme funkci `maximum'`, první dva vzory nebudou sedět. Třetí ovšem ano a rozdělí seznam na `2` a `[5,1]`. Klauzule *where* chce znát maximum z `[5,1]`, takže následujme tuhle cestu. Znovu se to zastaví u třetího vzoru a `[5,1]` se rozdělí na `5` a `[1]`. Klauzule *where* chce znovu znát nejvyšší číslo ze seznamu `[1]`. Protože to je okrajová podmínka, vrátí `1`. Konečně! Takže když postoupíme o krok dál porovnáním čísla `5` s maximem z `[1]` (což je `1`), dostaneme zpátky `5`. Víme tedy, že maximum z `[5,1]` je `5`. Postoupíme zase o krok dál, kde máme `2` a `[5,1]`. Porovnáním čísla `2` s maximem z `[5,1]`, což je `5`, se rozhodneme pro číslo `5`.

Čistější způsob, jak napsat tuhle funkci, je použít `max`. Pokud si pamatujete, `max` je funkce, která vezme dvě čísla a vrátí větší z nich. Tady je ukázka, jak bychom mohli přepsat `maximum'` za použití funkce `max`:

```
::          =>     ->
  =         "maximum z prázdného seznamu"
   =
     =
```

To je ale elegantní! Maximum ze seznamu je v podstatě `max` z prvního prvku a maxima ze zbytku.



Několik dalších rekurzivních funkcí

A teď, když víme, jak se zhruba dá myslet rekurzivně, napišme si pár funkcí s použitím rekurze. Nejprve si implementujeme funkci `replicate`. Tato funkce vezme parametr typu `Int` a nějaký prvek a vrátí seznam, který má několik opakování toho stejného prvku. Kupříkladu `replicate 3 5` vrátí `[5,5,5]`. Zamysleme se nad okrajovou podmínkou. %00tip je, že okrajová podmínka je 0 nebo méně. Jestliže zkusíme opakovat něco nulakrát, měl by se vrátit prázdný seznam. Stejně pro záporná čísla, protože to nedává moc smysl.

```
    ::                 =>    ->    ->

|   <= 0     =
|           =                 1
```

Použili jsme stráže místo vzorů, protože testujeme booleovskou podmínku. Jestliže je `n` menší nebo rovno 0, vrátí se prázdný seznam. Jinak vrátí seznam, který má `x` jako první prvek a poté `x` opakované `n` mínus jedenkrát jako zbytek. Část s `(n-1)` způsobí, že naše funkce nakonec dosáhne okrajové podmínky.

Poznámka: třída `Num` není podtřídou `Ord`. Což znamená, že mohou existovat čísla, která nedodržují uspořádání. To je důvod, proč jsme museli specifikovat omezení třídami `Num` a `Ord` pro sčítání nebo odčítání a taktéž pro porovnávání.

Dále si vytvoříme funkci `take`. Ta vezme určitý počet prvků ze seznamu. Například výraz `take 3 [5,4,3,2,1]` vrátí seznam `[5,4,3]`. Jestliže zkusíme vzít 0 nebo méně prvků ze seznamu, dostaneme prázdný seznam. Stejně pokud zkusíme vybrat něco z prázdného seznamu, dostaneme prázdný seznam. Všimněte si, že tohle jsou dvě okrajové podmínky. Takže to zkusíme zapsat:

```
    ::                  =>    ->      ->

|   <= 0    =
           =
           =    :           1
```

První vzor uvádí, že pokud zkusíme vzít 0 nebo záporný počet prvků, dostaneme prázdný seznam. Všimněte si, že jsme použili `_` na ověření seznamu, protože nás v tomhle případě ten seznam nezajímá. Také si všimněte použití stráže bez části `otherwise`. Jestliže se tím pádem ukáže, že `n` je větší než 0, ověření selže a přejde se na další vzor. Druhý vzor naznačuje, že pokud chceme vzít cokoliv z prázdného sezamu, dostaneme prázdný seznam. Třetí vzor rozdělí seznam na první prvek a zbytek. A poté uvedeme, že když vezmeme `n` prvků ze seznamu, je to stejné jako sezam, který obsahuje první prvek `x` a poté seznam `n-1` prvků ze zbytku. Zkuste si vzít kus papíru, abyste si zapsali, jak by mohlo vypadat vyhodnocování, řekněme, tří prvků ze seznamu `[4,3,2,1]`.

Funkce `reverse` jednoduše převrátí seznam. Zamyslete se nad okrajovou podmínkou. Jaká je? No tak… je to prázdný seznam! Převrácený prázdný seznam je stejný jako prázdný seznam. Tak jo. A co s ostatními případy? No, můžeme říct, že když rozdělíme seznam na jeho první prvek a zbytek, převrácený seznam je stejný jako převrácený zbytek a k němu na konec přidaný první prvek.

```
    ::      ->
      =
            =                ++
```

A je to!

Protože Haskell podporuje nekonečné seznamy, naše rekurze nemusí mít okrajovou podmínku. Pokud by ji ale neobsahovala, tak by skončila v nekonečné smyčce nebo vyprodukovala nekonečnou datovou strukturu, jako třeba nekonečný seznam. Na nekonečných seznamech je dobré, že je můžeme useknout na jakémkoliv místě. Funkce `repeat` vezme prvek a vrátí nekonečný seznam, jež obsahuje jenom ten prvek. Dá se to napsat opravdu jednoduše, sledujte.

```
    ::    ->
      =
```

Zavolání `repeat 3` nám vytvoří seznam začínající číslem `3`, za kterým následuje nekonečné množství trojek. Takže by se zavolání `repeat 3` mohlo vyhodnotit jako `3:repeat 3`, což je `3:(3:repeat 3)`, poté `3:(3:(3:repeat 3))` a tak dále. Výraz

`repeat 3` nikdy neskončí vyhodnocování, zatímco `take 5 (repeat 3)` nám vytvoří seznam pěti trojek. To je v podstatě stejné jako `replicate 5 3`.

Funkce `zip` vezme dva seznamy a sepne je dohromady. Vyhodnocení `zip [1,2,3] [2,3]` vrátí `[(1,2),(2,3)]`, protože ořízne delší seznam, aby měl stejnou délku jako ten kratší. Co když sepneme něco s prázdným seznamem? No tak to pak dostaneme prázdný seznam. To je naše okrajová podmínka. Nicméně `zip` požaduje jako parametr dva seznamy, ve skutečnosti jsou tedy dvě okrajové podmínky.

```
      ::      ->      ->
          =
          =
                  =
```

První dva vzory zadávají, že pokud je první nebo druhý seznam prázdný, dostaneme prázdný seznam. Třetí udává, že se dva sepnuté seznamy rovnají spárování jejich prvních prvků a poté připojení jejich sepnutých zbytků. Při sepnutí seznamů `[1,2,3]` a `['a','b']` se pokusí sepnout `[3]` s `[]`. Uplatní se vzor s hraniční podmínkou a takže vznikne výsledek `(1,'a'):(2,'b'):[]`, což je úplně to stejné jako `[(1,'a'),(2,'b')]`.

Napíšeme si ještě jednu další standardní funkci — `elem`. Ta vezme prvek a seznam a podívá se, jestli se ten prvek vyskytuje v seznamu. Okrajová podmínka, jak to tak u seznamů většinou bývá, je prázdný seznam. Víme, že prázdný seznam neobsahuje žádné prvky, takže zcela určitě nebude obsahovat droidy, které hledáme.

```
      ::        =>    ->      ->
          =

      |   ==      =
      |           =    `elem'`
```

Poměrně jednoduché a očekávatelné. Jestliže není první prvek stejný jako hledaný, zkontrolujeme zbytek. Když narazíme na prázdný seznam, výsledek je `False`.

Rychle, řaď!

Máme seznam věcí, které mohou být seřazeny. Jejich typ je instancí typové třídy `Ord`. A teď bychom je chtěli seřadit! Existuje bezvadný řadící algoritmus, nazvaný quicksort. Je to velice chytrý způsob pro řazení věcí. Zatímco je potřeba více než 10 řádků pro napsání quicksortu v imperativních jazycích, implementace v Haskellu je mnohem kratší a elegantnější. Quicksort se stal haskellovou vábničkou. Tudíž si ho tady napíšeme, přestože je psaní quicksortu v Haskellu považováno za celkem podřadné, protože to všichni používají jako ukázku elegance Haskellu.

Takže typové omezení této funkce bude `quicksort :: (Ord a) => [a] -> [a]`. Žádné překvapení. Okrajová podmínka? Prázdný seznam, což jsme čekali. Seřazený prázdný seznam je prázdný seznam. A teď přichází základní algoritmus: **seřazený seznam je seznam, který má všechny hodnoty, jež jsou menší (nebo rovny) prvnímu prvku seznamu, na svém začátku (a tyto hodnoty jsou seřazeny), poté obsahuje první prvek a dále následují všechny hodnoty, které jsou větší než první prvek (jsou také seřazeny).** Všimněte si, že jsme v této definici dvakrát zmínili slovo *seřazeny*, takže pravděpodobně budeme muset udělat rekurzivní volání dvakrát! Také si povšimněte, že jsme to definovali použitím slovesa *je* pro definici algoritmu, místo abychom řekli *udělej tohle, udělej tamto, pak udělej to*… V tom spočívá krása funkcionálního programování! Jak zařídíme filtrování seznamu, abychom dostali pouze menší nebo větší prvky než je první prvek z našeho seznamu? Pomocí generátoru seznamu. Takže se do toho pustíme a definujeme si tu funkci.

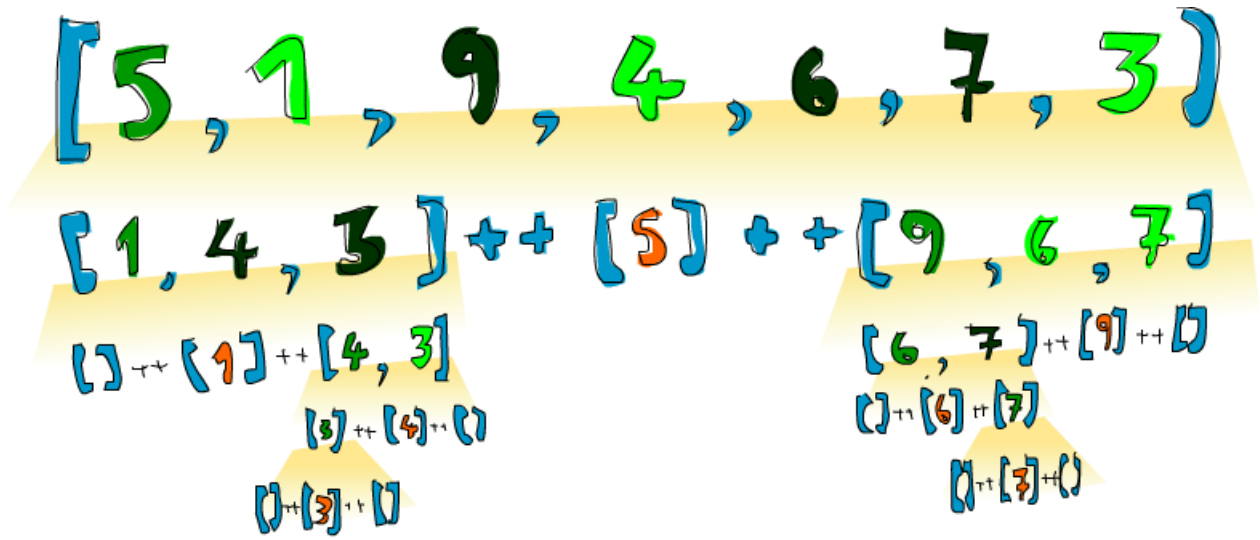
```
          ::          =>      ->
              =
                  =
```

```
    let                =                |   <-        <=
                      =                 |   <-        >
    in                 ++       ++
```

Zkusíme si malý testovací provoz, abychom viděli, zda-li to funguje bez chyb.

```
ghci>           10 2 5 3 1 6 7 4 2 3 4 8 9
 1 2 2 3 3 4 4 5 6 7 8 9 10
ghci>          "prilis zlutoucky kun upel dabelske ody"
"      abcddeeeiikkkllllnoopprsstuuuuyyz"
```

Hurá! Přesně o tomhle jsem mluvil! Takže když máme, řekněme, `[5,1,9,4,6,7,3]` a chceme tento seznam seřadit, algoritmus vezme první prvek, což je `5` a poté ho vloží doprostřed dvou seznamů, které jsou menší a větší než ten prvek. Takže v jednom okamžiku máme `[1,4,3] ++ [5] ++ [9,6,7]`. Víme, že až bude ten seznam celý seřazen, číslo `5` zůstane na čtvrtém místě, protože jsou v seznamu tři čísla menší a tři větší. A teď, jakmile seřadíme `[1,4,3]` a `[9,6,7]`, dostaneme seřazený seznam! Seřadíme ty dva seznamy použitím stejné funkce. Nakonec se to celé rozpadne tak, že dospějeme k prázdným seznamům a prázdné seznamy jsou už svým způsobem seřazené, vzhledem k tomu, že jsou prázdné. Tady je znázornění:



Prvek, který je na svém místě a nebudeme ho posouvat, je znázorněn **oranžově**. Jestliže je přečtete z leva do prava, uvidíte seřazený seznam. Ačkoliv jsme si vybrali na porovnávání první prvek, mohli jsme si vybrat jakýkoliv jiný. V quicksortu se prvek, který se používá na porovnávání, nazývá pivot. Je vyznačený **zeleně**. Vybrali jsme si první prvek, protože se jednoduše dá získat pomocí vzoru. Menší prvky než je pivot jsou zvýrazněny **světle zeleně** a větší prvky než je pivot jsou **tmavě zelené**. Nažloutlý přechod symbolizuje aplikaci quicksortu.

Myslíme rekurzivně

Udělali jsme toho dosud pomocí rekurze celkem dost a jak jste si zřejmě všimli, je v tom určité schéma. Obvykle si definujeme okrajový případ a poté funkci, která dělá něco s nějakým prvkem a funkcí aplikovanou na zbytek. Nezáleží na tom, jestli to je seznam, strom nebo nějaká jiná datová struktura. Součet je první prvek seznamu plus součet zbytku. Násobek seznamu je první prvek seznamu krát násobek zbytku. Délka seznamu je jednička plus délka zbytku seznamu. A tak dále, a tak dále…

Samozřejmě, tyhle funkce mají rovněž okrajové případy. Okrajový případ je obvykle nějaká možnost, u které aplikace rekurze nedává smysl. Když se pracujeme se seznamy, okrajovým případem často bývá prázdný seznam. Jestliže pracujeme se stromy, okrajovým případem obvykle je uzel, jež nemá žádné potomky.

Je to podobné, když rekurzivně pracujete s čísly. Obvykle to má co dělat s nějakým číslem a funkcí aplikovanou na to číslo s úpravou. Ukázali jsme si předtím funkci pro výpočet faktoriálu a to je násobek čísla a faktoriál toho čísla, zmenšeného o jedničku.

Tahle aplikace rekurze nedává smysl pro nulu, protože faktoriál je definován pouze pro kladná celá čísla. Často se ukáže, že okrajový případ je identita. Identita pro násobení je jednička, protože jestliže vynásobíte něco jedničkou, dostanete to nazpět. Stejně tak když sčítáme seznamy, definujeme přičtení prázdného seznamu jako nulu, protože nula je identita ve sčítání. U quicksortu je okrajový případ prázdný seznam a identita je také prázdný seznam, protože jestliže připojíme prázdný seznam k seznamu, získame ten stejný seznam.

Takže když se snažíte myslet rekurzivním způsobem při řešení úloh, zkuste přemýšlet nad případy, na které se rekurze nedá aplikovat a podívat se, jestli je můžete použít jako okrajový případ, přemýšlejte o identitách a přemýšlejte o případném rokladu parametrů funkce (například seznamy jsou obvykle rozkládány na první prvek a zbytek pomocí vzorů) a na jakou část použijete rekurzivní volání.

Syntaxe ve funkcích Obsah Funkce vyššího řádu

← Rekurze Obsah Moduly →

Funkce vyššího řádu

English version

Funkce v Haskellu mohou mít jako parametr jiné funkce a stejně tak mohou být funkce návratovou hodnotou. Funkce, která provádí obě tyhle věci se nazývá funkce vyššího řádu. Funkce vyššího řádu nejsou jenom součástí haskellové praxe, ony jsou v podstatě haskellovou praxí. Ukazuje se, že když chceme zadávat výpočty pomocí definování věcí jaké *jsou*, místo definování kroků, které změní nějaké stavy a možná pomocí cyklů, funkce vyššího řádu jsou nezbytné. Je to velice silný nástroj na řešení problémů a způsob myšlení o programech.

Curryfikované funkce

Každá funkce v Haskellu bere oficiálně pouze jeden parametr. Tak jak je možné, že jsme si předtím definovali a používali několik funkcí, jež braly více než jeden parametr? No, je to důmyslný trik! Všechny funkce, které předtím akceptovaly *několik parametrů*, byly **curryfikované funkce**. Co to znamená? Nejlépe to pochopíte na příkladu. Pojďme vzít naši dobrou známou, funkci `max`. Vypadá to, že vezme dva parametry a vrátí ten, který je větší. Při vyhodnocování `max 4 5` se nejprve vytvoří funkce, která vezme parametr a vrací buď `4` a nebo ten parametr, což závisí na tom, co je větší. Poté je aplikována hodnota `5` na tu funkci a funkce vyprodukuje náš požadovaný výsledek. Zní to jako jazykolam, ale je to ve skutečnosti vážně skvělý koncept. Následující dvě volání jsou ekvivalentní:

```
ghci>     4 5
5
ghci>      4  5
5
```

Poznámka překladatele: pojem *curryfikace* je pojmenován (stejně jako tento programovací jazyk) podle amerického matematika a logika Haskella Curryho. Protože nebyla ustálena pravopisná podoba slova, lze se setkat i s alternativními verzemi *currifikace* nebo *curryifikace*, které označují tu stejnou věc. Curryfikace se zdá být nejsprávnější variantou.

Vložená mezera mezi dvěma věcmi je jednoduše **aplikace funkce**. Mezera je něco jako operátor a má nejvyšší prioritu. Podívejme se na typ funkce `max`. Je `max :: (Ord a) => a -> a -> a`. To může být také zapsáno jako `max :: (Ord a) => a -> (a -> a)`. Což bychom mohli číst: `max` vezme `a` a vrátí (to je ta šipka `->`) funkci, která vezme nějaké `a` a vrátí `a`. To je důvod, proč jsou návratový typ a parametry funkce vždy odděleny pomocí šipek.

A jaký to má pro nás přínos? Jednoduše řečeno, jestliže zavoláme funkci s méně parametry, dostaneme zpátky **částečně aplikovanou** funkci, což znamená funkci, která požaduje tolik parametrů, kolik jich vynecháme. Použití částečné aplikace (zavolání funkce s méně parametry, jestli chcete) je prima cesta, jak vytvořit za běhu funkce, které můžeme poslat jiné funkci nebo je naplnit nějakými daty.

Podívejte se na tuhle až urážlivě jednoduchou funkci:

```
::          =>    ->    ->    ->
      =    *     *
```

Co se doopravdy děje, když provedeme `multThree 3 5 9` nebo `((multThree 3) 5) 9`? Nejprve je aplikován parametr `3` na funkci `multThree`, protože jsou odděleny mezerou. To vytvoří funkci, která vezme jeden parametr a vrátí funkci. Takže pak se na to aplikuje `5`, což vytvoří funkci, která vezme parametr a vynásobí ho 15. Poté se na tu funkci aplikuje `9` a výsledek je 135 nebo tak nějak. Zapamatujte si, že typ této funkce by se dal zapsat jako `multThree :: (Num a) => a -> (a -> (a -> a))`. Ta věc před první šipkou `->` je parametr, který funkce vezme a ta věc za ní je to co vrací. Takže naše funkce vezme a a vrátí funci typu `(Num a) => a -> (a -> a)`. Podobně tato funkce vezme a a vrátí funkci typu `(Num a) => a -> a`. A konečně tato funkce prostě %0000`a` a vrátí nějaké jiné `a`. Podívejte se na tohle:

```
ghci> let                  =           9
ghci>                  2 3
54
ghci> let                    =                 2
ghci>                   10
180
```

Zavoláním funkce s méně parametry, abych tak řekl, vytvoříme novou funkci za běhu. Co když chceme vytvořit funkci, která vezme číslo a porovná ho s číslem `100`? Mohli bychom to napsat nějak takhle:

```
::                  =>    ->
  =          100
```

Jestli že funkci zavoláme s parametrem `99`, vrátí `GT`. Jasná věc. Všimněte si, že `x` je na vpravo na obou stranách rovnice. A teď přemýšlejte, co vrací výraz `compare 100`. Vrací funkci, která vezme číslo a porovná ho s číslem `100`. Páni! Není to ta funkce, jakou jsme hledali? Můžeme to přepsat jako:

```
::                  =>    ->
=          100
```

Deklarace typu zůstává stejná, portože `compare 100` vrací funkci. Porovnání funkce má typ `(Ord a) => a -> (a -> Ordering)` a zavolání s parametrem `100` vrátí typ `(Num a, Ord a) => a -> Ordering`. Vkrade se tam další typové omezení, protože číslo `100` je součástí typové třídy `Num`.

Hej! Ujistěte se, že opravdu chápete, jak curryfikované funkce a částečná aplikace funguje, protože jsou to vážně důležitá témata!

Infixová funkce může být také částečně aplikovaná za použití řezu (sekce). K rozříznutí infixové funkce ji jednoduše obklopíme kulatými závorkami a dodáme parametr na jednu stranu. To vytvoří funkci, která vezme jeden parametr a ten aplikuje na tu stranu, kde chybí operand. Příklad rozříznuté triviální funkce:

```
::               =>    ->
=    10
```

Zavolání, řekněme, `divideByTen 200` je stejné jako provedení `200 / 10` nebo jako `(/10) 200`. Funkce, která ověřuj, zda je zadaný znak velké písmeno:

```
::       ->
=   `elem`   'A'   'Z'
```

Jedinou neobvyklou věcí u řezu je použití mínusu `-`. Z definice řezu by `(-4)` mohl být výsledek z funkce, která vezme číslo a odečte od něj čtyřku. Nicméně, z důvodů pohodlí, `(-4)` znamená mínus čtyři. Takže když budete chtít vytvořit funkci pro odečítání čtyřky od čísla, jež dostane jako parametr, částečně aplikujte funkci `subtract`, jako třeba: `(subtract 4)`.

Co se stane, pokud zkusíme zadat do GHCi `multThree 3 4`, bez toho, abychom výraz definovali pomocí konstrukce let nebo ho předali jiné funkci?

```
ghci>           3 4
             1 0
        instance              ->
                        of                          1 0 12
                        instance                          ->

           do
```

GHCi nám říká, že výraz vyprodukovaný funkcí má typ `a -> a`, ale neví, jako to vypsat na obrazovku. Funkce nejsou instancí typové třídy `Show`, takže nemůžeme získat hezkou textovou reprezentaci funkce. Když napíšeme, řekněme, `1 + 1` do příkazové řádky GHCi, nejprve se spočítá výsledek `2` a poté na něj zavolá funkce `show`, aby se získala textové reprezentace toho čísla. A textová reprezentace čísla `2` je řetězec `"2"`, který je pak vypsán na naši obrazovku.

Trocha vyššího řádu je v pořádku

Funkce mohou mít jako parametry funkce a také vracet funkce. Abychom si to objasnili, vytvoříme si funkci, která vezme další funkci a aplikuje ji na něco dvakrát!

```
             ::     ->     ->    ->
                =
```

Jako první si všimněte deklarace typu. Předtím jsme nepotřebovali závorky, protože šipka `->` je sama o sobě asociativní zprava. Nicméně tady jsou povinné. Ukazují, že první parametr je funkce, která něco vezme a vrátí tu stejnou věc. Druhým parametrem je něco stejného typu a co má návratovou hodnotu stejného typu. Mohli bychom číst tuhle typovou deklaci curryfikovaným způsobem, ale abychom se vyhnuli bolení hlavy, řekneme prostě, že tahle funkce vezme dva parametry a vrátí jednu věc. Prvním parametrem je nějaká funkce (typu `a -> a`) a druhý má jako typ to stejné `a`. První funkce může být třeba `Int -> Int` nebo `String -> String` nebo cokoliv jiného. Ale druhý parametr pak musí být stejného typu.

Poznámka: od teď si budeme říkat, že funkce vezme několik parametrů, přestože každá taková funkce ve skutečnosti bere pouze jeden parametr a vrací částečně aplikovanou funkci, dokud nevrátí solidní hodnotu. Takže v zájmu jednoduchosti budeme tvrdit, že `a -> a -> a` má dva parametry, i když víme, co se doopravdy odehrává pod kapotou.

Obsah této funkce je celkem srozumitelný. Prostě vezmeme parametr `f` jako funkci, aplikujeme na něj `x` pomocí oddělení mezerou a poté se výsledek aplikuje znovu na funkci `f`. Kromě toho ještě nějaké blbnutí s funkcemi:

```
ghci>              3  10
16
ghci>                " HAHA"   "HEJ"
"HEJ HAHA HAHA"
ghci>             "HAHA "      "HEJ"
"HAHA HAHA HEJ"
ghci>                        2 2  9
144
ghci>             3    1
 3 3 1
```

Skvělost a užitečnost částečné aplikace je zjevná. Jestliže naše funkce po nás chce, abychom ji předali funkci, která vezme pouze jeden parametr, můžeme částečně aplikovat funkci na to místo, kde bere ten jeden parametr, a poté ji předat.

A teď využijeme programování s částečnou aplikací na napsání vážně užitečné funkce, která je ve standardní knihovně. Je nazvaná `zipWith`. Vezme funkci a dva seznamy jako parametr a poté spojí ty dva seznamy aplikací funkce na korespondující elementy. Tady je ukázána naše implementace:

```
::     ->    ->     ->      ->      ->
       =
       =
                =       :
```

Podívejte se na typovou deklaraci. První parametr je funkce, která vezme dvě věci a vyprodukuje třetí. Nemusí být stejného typu, ale mohou. Druhý a třetí parametr jsou seznamy. Výsledek je také seznam. První musí být seznam typu `a`, protože spojovací funkce má první argument typu `a`. Druhý musí být seznam typu `b`, protože spojovací funkce má druhý argument také typu `b`. Výsledek je seznam věcí typu `c`. Jestliže typová deklarace funkce říká, že akceptuje funkci typu `a -> b -> c` jako parametr, bude také akceptovat funkci typu `a -> a -> a`, ale opačně to neplatí! Pamatujte si, že když vytváříme funkce, zvláště ty vyššího řádu, a nejsme si jistí jejich typem, můžeme jednoduše vynechat jejich typovou deklaraci a poté se podívat pomocí příkazu `:t` na to, co Haskell sám odvodí.

Činnost funkce je celkem podobná normální funkci `zip`. Okrajové podmínky jsou stejné, akorát je tam navíc jeden argument, spojovací funkce, ale na tom argumentu v okrajových podmínkách nezáleží, takže na to prostě použijeme podtržítko `_`. A část s posledním vzorem je rovněž podobná funkci `zip`, jenom nevytváří `(x,y)`, ale `f x y`. Jedna funkce vyššího řádu může být použita na spoustu odlišných úkolů, pokud je napsána dostatečně obecně. Tady je malá názorná ukázka různých věcí, co naše funkce `zipWith'` může provádět:

```
ghci>              4 2 5 6   2 6 2 3
 6 8 7 9
ghci>              6 3 2 1   7 3 1 5
 7 3 2 5
ghci>                "foo"  "bar"  "baz"    " fighters"  "man"  "mek"
 "foo fighters" "barman" "bazmek"
ghci>                        5 2   1
 2 4 6 8 10
ghci>                          1 2 3   3 5 6   2 3 4     3 2 2   3 4 5   5 4 3
  3 4 6   9 20 30   10 12 12
```

Jak můžete vidět, jediná funkce vyššího řádu může být použita k více účelům. Imperativní programování obvykle používá věci jako jsou for smyčky, while smyčky, přiřazení něčeho do proměnné, kontrolování stavu atd. pro dosažení nějakého chování a poté to obalí nějakým rozhraním, třeba funkcí. Funkcionální programování používá funkce vyššího řádu pro abstrahování běžných schémat, jako procházení dvou seznamů po dvojicích a zacházení s těmito dvojicemi nebo dostávání množin řešení a vylučování těch, které nepotřebujeme.

Vytvoříme si další funkci, vyskytující se ve standardní knihovně, nazvanou `flip`. Flip jednoduše vezme nějakou funkci a vrátí funkci, podobnou naší původní funkci, s tím rozdílem, že jsou první dva argumenty prohozeny. Můžeme ji napsat třeba jako:

```
    ::    ->    ->    ->    ->    ->
      =
  where       =
```

Když si přečteme typovou deklaraci, můžeme říct, že vezme funkci, která požaduje nějaké `a` a nějaké `b` a vrátí funkci, která požaduje nějaké `b` a nějaké `a`. Druhý pár závorek je opravdu nezbytný, protože funkce jsou implicitně curryfikované a protože je šipka `->` asociativní zprava. Typ `(a -> b -> c) -> (b -> a -> c)` je stejný jako `(a -> b -> c) -> (b -> (a -> c))`, což je totéž co `(a -> b -> c) -> b -> a -> c`. Napsali jsme, že `g x y = f y x`. Pokud to je pravda, pak také platí, že `f y x = g x y`, ne? Budeme-li to mít na paměti, můžeme funkci definovat dokonce ještě jednodušeji.

```
    ::    ->    ->    ->    ->    ->
```

```
              =
```

Tady jsme využili faktu, že jsou funkce curryfikované. Když zavoláme funkci `flip' f` bez dalších dvou parametrů, vrátí funkci `f`, která vezme tyhle dva parametry, ale zavolá je v opačném pořadí. Přestože funkce s prohozenými parametry jsou obvykle předány další funkci, můžeme rozmýšlet dopředu a využít curryfikace, když vytváříme funkce vyšších řádů, a napsat, jaký by mohl být jejich konečný výsledek, pokud jsou zavolány s plnou aplikací.

```
ghci>            1 2 3 4  "ahoj"
  'a' 1   'h' 2   'o' 3   'j' 4
ghci>                      2 2     10 8 6 4 2
 5 4 3 2 1
```

Mapy a filtry

vezme funkci a seznam a aplikuje tu funkci na každý prvek ze seznamu, což vyrobí nový seznam. Podívejme se, jaký má typ a jak je definována.

```
    ::    ->    ->     ->
        =
            =     :
```

Z typu zjistíme, že požaduje funkci, která vezme nějaké `a` a vrátí nějaké `b`, dále seznam věcí typu `a` a vrátí seznam věcí typu `b`. Je zajímavé, jak pouhým pohledem na typ funkce můžete občas říct, co funkce dělá. Funkce `map` je jedna z těch opravdu všestranných funkcí vyššího řádu, které mohou být použity na milión způsobů. Tady je v akci:

```
ghci>       3   1 5 3 1 6
 4 8 6 4 9
ghci>          "!"   "BUM"   "BÁC"   "PRÁSK"
 "BUM!" "BÁC!" "PRÁSK!"
ghci>                3   3  6
  3 3 3   4 4 4   5 5 5   6 6 6
ghci>            2     1 2   3 4 5 6   7 8
  1 4   9 16 25 36   49 64
ghci>           1 2   3 5   6 3   2 6   2 5
 1 3 6 2 2
```

Pravděpodobně jste si všimli, že by se všechny ty ukázky daly přepsat pomocí generátorů seznamu. Když napíšeme `map (+3) [1,5,3,1,6]`, tak to je stejné jako `[x+3 | x <- [1,5,3,1,6]]`. Nicméně použití funkce `map` je mnohem čitelnější pro případy, ve kterých pouze aplikujete nějakou funkci na prvek ze seznamu, zvláště pokud se vypořádáváte s mapováním mapování a pak by celá ta věc s hromadou závorek byla dost komplikovaná.

je funkce, která vezme predikát (predikát je funkce, která nám řekne, jestli je něco pravda nebo nepravda, takže v našem případě to je funkce vracející booleovskou hodnotu) a nějaký seznam a potom vrátí seznam prvků, které vyhovují predikátu. Typ a implementace vypadají nějak takhle:

```
      ::    ->       ->     ->
          =

    |           =   :
    |           =
```

Celkem jednoduchá věc. Jestliže se `p x` vyhodnotí jako `True`, prvek se zařadí do nového seznamu. Pokud ne, zahodí ho. Nějaké ukázky použití:

```
ghci>          3   1 5 3 2 1 6 4 3 2 1
 5 6 4
ghci>           3   1 2 3 4 5
 3
```

```
ghci>              1  10
 2 4 6 8 10
ghci> let           =              in                  1 2 3      3 4 5   2 2
  1 2 3   3 4 5   2 2
ghci>         `elem`  'a'  'z'    "SmĚjeŠ sE mI, pRoToŽe jSeM JinÝ."
"mjesmpooejein"
ghci>         `elem`  'A'  'Z'    "SmějU se váM, prOTožE jSte sTejní."
"SUMOTEST"
```

Tohle všechno by také dalo docílit pomocí generátorů seznamu s predikáty. Neexistuje žádná sada pravidel, kdy použít `map` a `filter` versus generátor seznamu, musíte se prostě v závislosti na kódu a kontextu rozhodnout, co je více čitelné. K aplikaci několika predikátů ve funkci `filter` je u generátoru seznamu ekvivalentní buď několikanásobné filtrování nebo spojení predikátů pomocí logické funkce `&&`.

Vzpomínáte si na naši funkci quicksort z předchozí kapitoly? Použili jsme generátor seznamu na odfiltrování prvků seznamu, které byly menší (nebo rovné) a větší než pivot. Můžeme docílit stejné funkčnosti a ještě lepší čitelnosti použitím funkce `filter`:

```
          ::          =>      ->
             =
                  =
      let               =
                        =
      in                 ++      ++
```

Mapování a filtrování jsou živobytím nářadí funkcionálního programátora. Hm. Nezáleží na tom, jestli budete využívat spíše funkce `map` a `filter` oproti generátoru seznamu. Vzpomeňte si, jak jsme řešili problém hledání pravoúhlých trojúhelníků s určitým obvodem. V imperativním programování bychom to mohli vyřešit zanořením tří smyček a následným testováním, jestli aktuální kombinace dá dohromady pravoúhlý trojúhelník a jestli má správný obvod. Pokud by to byl ten případ, mohli bychom to vytisknout na obrazovku nebo něco. Ve funkcionálním programování se tohoto schématu dá docílit mapováním a filtrováním. Vytvoříte si funkci, která vezme nějakou hodnotu a vyprodukuje nějaký výsledek. Namapujeme tuhle funkci na seznam hodnot a poté z něj odfiltrujeme vyhovující výsledky. I když něco namapujeme na seznam vícekrát a poté to několikrát odfiltrujeme projde to díky lenosti Haskellu tím seznamem pouze jednou.

Pojďme **najít největší číslo pod 100000, které je dělitelné číslem 3829**. Abychom toho docílili, jednoduše odfiltrujeme množinu možností, o kterých víme, že se mezi nimi nalézá řešení.

```
                ::              =>
                =                 100000 99999
      where     =    `mod` 3829 == 0
```

Nejprve si vytvoříme seznam všech čísel menší než 100000, sestupně. Poté ho odfiltrujeme naším predikátem a protože jsou čísla seřazena sestupně, největší číslo, které vyhovuje našemu predikátu, bude prvním prvkem našeho seznamu. Jako výchozí množinu jsme dokonce nemuseli použít konečný seznam. Opět je tu vidět lenost v akci. Protože jsme nakonec využili pouze první prvek z filtrovaného seznamu, nezáleží na tom, jestli je ten seznam konečný nebo ne. Vyhodnocování se zastaví, když se nalezne první přijatelné řešení.

Dále zkusíme **najít součet všech lichých čtverců, které jsou menší než 10000**. Nejprve si ale představíme funkci , protože ji využijeme v našem řešení. Vezme nějaký predikát a seznam a poté ten seznam prochází od začátku a vrací jeho prvky, dokud vyhovují predikátu. Jakmile se narazí na prvek, pro který predikát neplatí, procházení se zastaví. Pokud bychom chtěli získat první slovo z řetězce `"sloni si umí užívat"`, mohli bychom udělat něco jako `takeWhile (/= ' ') "sloni si umí užívat"` a to by nám vrátilo řetězec `"sloni"`. Dobře. Součet všech lichých čtverců, které jsou menší než 10000. Začneme namapováním funkce `(^2)` na nekonečný seznam `[1..]`. Poté to odfiltrujeme, abychom dostali jen ty liché. A

pak, vybereme prvky, které jsou menší než 10000. A nakonec to celé sečteme. Nemusíme si na to ani definovat funkci, můžeme to vyřešit jedním řádkem v GHCi:

```
ghci>                  10000                    2    1
166650
```

Skvělé! Začali jsme s nějakými počátečními daty (nekonečný seznam všech přirozených čísel) a poté jsme na něj namapovali funkci, odfiltrovali ho a zredukovali na část, která vyhovovala naším potřebám, a tu jsme prostě sečetli. Mohli jsme to také zapsat pomocí generátoru seznamu:

```
ghci>                  10000      |    <-    2 |    <-  1
166650
```

Je to otázka vkusu, který zápis se vám bude líbit více. Znovu, tohle celé umožňuje lenost v Haskellu. Můžeme namapovat a odfiltrovat nekonečný seznam, protože se to nebude ve skutečnosti mapovat a filtrovat hned, tyhle akce se odloží. Jenom když donutíme Haskell ukázat nám součet, funkce `sum` řekne funkci `takeWhile`, že potřebuje čísla. Funkce `takeWhile` přinutí filtry a mapy, aby se prováděly, ale jenom dokud nepotká číslo větší nebo rovné 10000.

V našem dalším problému se budeme vypořádávat s Collatzovou řadou. Vezmeme nějaké přirozené číslo. Jestliže je to číslo sudé, vydělíme ho dvěma. Jestliže je liché, vynásobíme ho třemi a přičteme k tomu jedničku. Na výsledné číslo aplikujeme stejný postup, což vyprodukuje nové číslo a tak dále. V podstatě dostaneme posloupnost čísel. Má se za to, že pro libovolné počáteční číslo skončí posloupnost jedničkou. Takže pokud vezmeme jako počáteční číslo 13, dostaneme tuhle řadu: *13, 40, 20, 10, 5, 16, 8, 4, 2, 1*. Výraz 13 * 3 + 1 se rovná 40. Číslo 40 vydělené 2 je 20, atd. Vidíme, že tato posloupnost má deset členů.

A teď chceme znát odpověď na tohle: **pro všechna čísla mezi 1 a 100, kolik posloupností je delších než 15?** Nejprve si napíšeme funkci, která vytvoří posloupnost:

```
        ::               =>    ->
       1 =  1

      |         =              `div` 2
      |         =              3 + 1
```

Protože posloupnost končí jedničkou, je to okrajový případ. To je celkem standardní rekurzivní funkce.

```
ghci>        10
 10 5 16 8 4 2 1
ghci>        1
 1
ghci>        30
 30 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

Jej! Vypadá to, že funguje korektně. A nyní přichází funkce, jež nám poskytne odpověď na naši otázku:

```
             ::
             =                                    1  100
    where           =           > 15
```

Namapujeme funkci `chain` na seznam `[1..100]`, abychom dostali seznam posloupností, které jsou znázorněny seznamem. Poté je odfiltrujeme pomocí predikátu, který se jednoduše podívá, jestli má seznam více než 15 prvků. Jakmile jsme hotoví s filtrováním, podíváme se, kolik posloupností zbylo ve výsledném seznamu.

Poznámka: tahle funkce je typu `numLongChains :: Int`, protože `length` vrací číslo typu `Int` místo `Num a` z historických důvodů. Kdybychom chtěli vrátit více obecné `Num a`, mohli bychom použít funkci `fromIntegral` na výsledný počet.

Použitím funkce `map` můžeme také vytvořit věci jako `map (*) [0..]`, minimálně z toho důvodu, abychom si ukázali, jak funguje curryfikace a že jsou (částečně aplikované) funkce opravdové hodnoty, které můžete posílat dalším funkcím nebo vkládat do seznamů (jenom je nemůžete změnit na řetězce). Zatím jsme pouze mapovali funkce, které použily jeden parametr na celý seznam, jako třeba `map (*2) [0..]`, abychom obdrželi seznam typu `(Num a) => [a]`, ale můžeme také bez problémů používat `map (*) [0..]`. Stane se zde to, že je číslo v seznamu aplikováno na funkci `*`, která má typ `(Num a) => a -> a -> a`. Aplikace pouze jednoho parametru na funkci, která vezme dva parametry, vrátí funkci, která požaduje jeden parametr. Jestliže ji namapujeme `*` na seznam `[0..]`, vrátí se nám seznam funkcí, které požadují pouze jeden parametr, takže je to typ `(Num a) => [a -> a]`. Když napíšeme `map (*) [0..]`, vytvoří to stejný seznam jako kdybychom napsali `[(*0),(*1),(*2),(*3),(*4),(*5)…]`.

```
ghci> let            =          0
ghci>                4  5
20
```

Získání prvku s indexem `4` z našeho seznamu vrátí funkci, která odpovídá `(*4)`. A poté jednoduše aplikujeme číslo `5` na tu funkci. Takže je to stejné jako napsání `(* 4) 5` nebo prostě `5 * 4`.

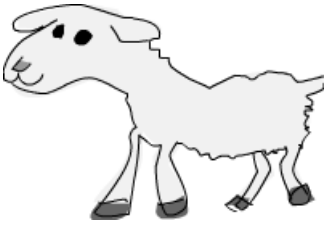
Lambdy

Lambdy jsou v zásadě anonymní funkce, které jsou používané, protože často potřebujeme nějakou funkci jenom jednou. Obvykle si vytváříme lambdu, abychom ji předali funkci vyššího řádu. Pro vytvoření lambdy napíšeme znak `\` (protože vypadá jako řecké písmeno lambda, když na něj pořádně zamžouráte) a poté napíšeme parametry, oddělené mezerami. Za tím následuje šipka `->` a tělo funkce. Obvykle to celé obklopíme kulatými závorkami, protože jinak to má sahá dál napravo.

Jestliže se podíváte o pětadvacet centimetrů nahoru, uvidíte, že jsme v naší funkci `numLongChains` použili konstrukci *where*, abychom si vytvořili funkci `isLong`, kterou jsme předali funkci `filter`. Takže místo toho můžeme použít lambdu:

```
::
=                    ->          > 15             1  100
```

Lambdy jsou výrazy, to je důvod, proč je můžeme jen tak předat. Výraz `(\xs -> length xs > 15)` vrátí funkci, která nám poví, jaké seznamy jsou delší než 15.



Lidé, jež nejsou důkladně obeznámeni s tím, jak curryfikace a částečná aplikace funguje, často používají lambdy tam, kde nemusí být. Kupříkladu výrazy `map (+3) [1,6,3,2]` a `map (\x -> x + 3) [1,6,3,2]` jsou ekvivalentní, protože `(+3)` a `(\x -> x + 3)` jsou obě funkce, co vezmou nějaké číslo a přičtou k němu trojku. Netřeba dodávat, že v tomhle případě je používat lambdu hloupé, jelikož je částečná aplikace mnohem čitelnější.

Lambdy mohou mít, stejně jako běžné funkce, libovolný počet parametrů:

```
ghci>              ->    * 30 + 3  /     5 4 3 2 1   1 2 3 4 5
 153.0 61.5 31.0 15.75 6.6
```

A můžete také, stejně jako v běžných funkcích, používat vzory. Jediný rozdíl je v tom, že nemůžete definovat více vzorů pro jeden parametr, jako napsání `[]` a `(x:xs)` na místo stejného parametru a poté ověřovat aktuální hodnotu. Jestliže vzory selžou u lambdy, dojde k běhové chybě, takže opatrně se vzory v lambda funkcích!

```
ghci>             ->    +       1 2    3 5    6 3    2 6    2 5
 3 8 9 8 7
```

Lambdy se normálně obklopují kulatými závorkami, pokud nechceme, aby sahaly dál napravo. Je to zajímavé: díky směru, v jakém jsou funkce automaticky curryfikované, jsou tyhle dva zápisy ekvivalentní:

```
          ::          =>    ->    ->    ->
                =     +     +
```

```
          ::          =>    ->    ->    ->
          =     ->      ->      ->    +     +
```

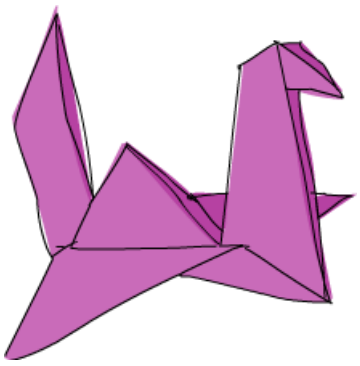
Jestliže definujeme takovou funkci, je zřejmé, proč má takovou typovou deklaraci, jako má. U typové deklarace i rovnice jsou tři šipky `->`. Samozřejmě, první způsob zápisu funkcí je mnohem čitelnější, druhý je spíše fígl na ukázání curryfikace.

Nicméně existují případy, ve který je použití této notace skvělé. Myslím si, že funkce `flip` je nejvíce přehledná, když je zapsaná nějak takhle:

```
        ::     ->    ->      ->    ->    ->
          =          ->
```

Ačkoliv to je stejné, jako když napíšeme `flip' f x y = f y x`, ozřejmíme tím, že to bude většinou použito pro vytvoření nové funkce. Nejobvyklejší použití funkce `flip` je zavolat ji pouze s jedním parametrem a předat výslednou funkci pomocí mapy nebo filtru. Takže používejte lambda funkce tímhle způsobem, když chcete zdůraznit, že je vaše funkce určená pro částečnou aplikaci a pro předání jiné funkci jako parametr.

Akumulační funkce fold



Když jsme se předtím zabývali rekurzí, všímali jsme si schématu u mnoha rekurzivních funkcí, které zacházejí se seznamy. Obvykle jsme měli okrajový případ pro prázdný seznam. Představili jsme si vzor `x:xs` a poté jsme prováděli určité akce týkající se prvku a zbytku seznamu. Ukázalo se, že to je velmi častý vzor, takže bylo zavedeno pár užitečných funkcí, které ho zapouzdřují. Těmto funkcím se říká foldy (skládače). Jsou podobné funkci `map`, akorát omezují seznam na nějakou jednu hodnotu.

Fold vezme binární funkci, nějakou počáteční hodnotu (rád ji říkám akumulátor) a seznam na poskládání. Binární funkce sama o sobě požaduje dva parametry. Je zavolána s akumulátorem a prvním (nebo posledním) prvkem a vytvoří nový akumulátor. Poté je tato binární funkce zavolána s novým akumulátorem a s novým prvním (nebo posledním) prvkem a tak dále. Jakmile projdeme celý seznam, zbyde nám pouze akumulátor, na který jsme seznam zredukovali.

Nejprve se podívejme na funkci , také nazvaná levý fold. Poskládá seznam z levé strany. Binární funkce je aplikována na počáteční hodnotu a první prvek ze seznamu. To vytvoří novou akumulační hodnotu a binární funkce je zavolána s tou hodnotou a dalším prvkem atd.

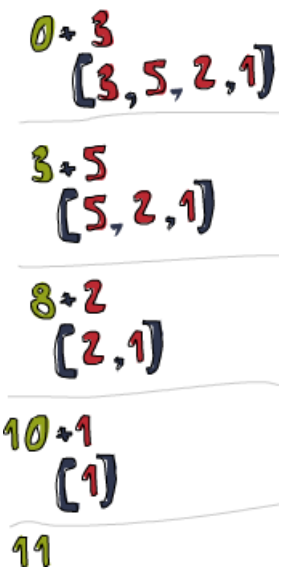
Pojďme si znovu implementovat funkci `sum`, jenom tentokrát použijeme fold místo explicitní rekurze.

```
        ::          =>      ->
```

```
=                ->     +    0
```

Zkouška, raz, dva, tři:

```
ghci>       3 5 2 1
11
```



Podíváme se do hloubky, co se při tomto vyhodnocování děje. Funkce `\acc x -> acc + x` je binární. Počáteční hodnota je tu `0` a `xs` je seznam, který bude poskládán. Na začátku je dosazeno číslo `0` na místo parametru `acc` v binární funkci a číslo `3` je (jako současný prvek) dosazeno na místo parametru `x`. Sečtení `0 + 3` vytvoří číslo `3` a to se stane novou akumulační hodnotou, dá-li se to tak říct. Dále je použité číslo `3` jako akumulační hodnota a číslo `5` jako aktuální prvek a tím pádem se číslo `8` stane novým akumulátorem. Pokračujeme, máme parametry `8` a `2`, nová akumulační hodnota je tedy `10`. A nakonec je použito číslo `10` jako akumulační hodnota a číslo `1` jako aktuální prvek, což vytvoří číslo `11`. Gratuluji, zvládli jsme skládání!

Tento odborný nákres nalevo objasňuje, co se ve foldu odehrává, krok za krokem (den za dnem!) . Zelenohnědé číslo je akumulační hodnota. Můžete vidět, jak je seznam takříkajíc požírán zleva akumulátorem. Mňamy, mňam, mňam! Jestliže vezmeme v úvahu, že funkce mohou být curryfikované, můžeme zapsat tuhle funkci ještě více stručněji, jako třeba:

```
::          =>      ->
=            0
```

Lambda funkce `(\acc x -> acc + x)` dělá to stejné co `(+)`. Můžeme vynechat `xs` jako parametr, protože zavolání `foldl (+) 0` vrátí funkci, která vezme seznam. Obecně, jestliže máte funkci jako `foo a = bar b a`, můžete ji díky curryfikaci přepsat jako `foo = bar b`.

Rozhodopádně si pojďme napsat vlastní funkci s levým foldem, než pokročíme k pravému. Jsem si jistý, že všichni víte, že funkce `elem` kontroluje, jestli se určitá hodnota vyskytuje v seznamu, takže to nebudu opakovat (jejda, zrovna jsem to udělal!). Pojďme si ji napsat pomocí levého foldu.

```
::         =>    ->      ->
     =                -> if    ==    then      else
```

Fajn, fajn, fajn, co to tady máme? Počáteční hodnota — a zároveň akumulátor — je zde booleovská hodnota. Typ akumulační hodnoty a typ výsledku je vždycky stejný, když se zabýváme s foldy. Pamatujte si, že pokud vůbec nevíte, co použít za počáteční hodnotu, dá vám určitou představu. Začneme s `False`. Dává to smysl použít `False` jako počáteční hodnotu. Předpokládáme, že tam ta hodnota není. Také když zavoláme fold na prázdný seznam, výsledkem bude počáteční hodnota. Pak zkontrolujeme, jestli se aktuální prvek rovná tomu, který hledáme. Jestliže je, nastavíme akumulátor na `True`. Jestliže není, jednoduše necháme akumulátor nezměněný. Pokud byl předtím `False`, zůstane stejný, protože se prvky nerovnají. Pokud byl `True`, necháme ho tak.

Pravý fold, , pracuje podobně jako levý, jenom s tím rozdílem, že akumulátor požírá hodnoty zprava. Také binární funkce pro levý fold má akumulátor jako první parametr a aktuální hodnotu jako druhý (tedy `\acc x -> …`), binární funkce pro pravý fold má aktuální hodnotu jako první parametr a akumulátor jako druhý (tedy `\x acc -> …`). Dává to celkem smysl, že pravý fold má akumulátor napravo, protože skládá z pravé strany.

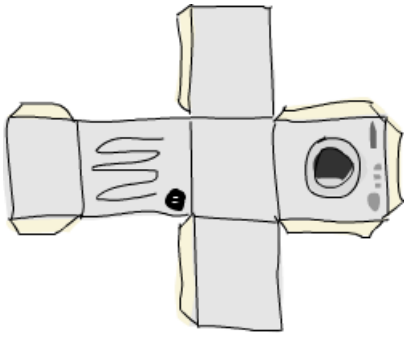
Hodnota akumulátoru (a tudíž i výsledek) ve foldu může být jakéhokoliv typu. Může to být číslo, booleovská proměnná nebo i seznam. Implementujeme si funkci map pomocí pravého foldu. Akumulátor bude seznam, budeme akumulovat mapovaný seznam prvek po prvku. Z toho je jasné, že počáteční prvek bude prázdný seznam.

```
::     ->     ->      ->
=                  ->      :
```

Jestliže namapujeme funkci `(+3)` na seznam `[1,2,3]`, přistoupíme k seznamu z pravé strany. Vezmeme poslední prvek, což je číslo `3`, a aplikujeme na něj funkci, z čehož se stane číslo `6`. Poté ho připojíme k akumulátoru, což nám dává `[]`. Výraz `6:[]` je `[6]` a to je současný akumulátor. Aplikujeme `(+3)` na číslo `2`, což je `5`, které připojíme `(:)` k akumulátoru, takže je akumulátor teď `[5,6]`. Aplikujeme `(+3)` na číslo `1` a to připojíme k akumulátoru, takže je konečná hodnota `[4,5,6]`.

Samozřejmě bychom si mohli napsat tuhle funkci pomocí levého foldu. Bylo by to `map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs`, ale problém je v tom, že funkce ++ mnohem náročnější než `:`, takže obvykle používáme pravé foldy na sestavování nových seznamů ze seznamu.

Když obrátíte seznam, můžete na něj použít pravý fold místo levého a naopak. Někdy ho ani nemusíte obracet. Funkce `sum` může být implementována prakticky stejně pomocí levého a pravého foldu. Hlavní rozdíl je v tom, že pravý fold může pracovat s nekonečnými seznamy, kdežto levý ne! Pro objasnění, pokud někdy vezmete nekonečný seznam od určitého místa a začnete ho skládat zprava, dostanete se časem na začátek toho seznamu. Nicméně pokud vezmete nekonečný seznam od určitého místa a zkusíte ho skládat zleva, nikdy se nedostanete na konec!



Foldy mohou být použity pro zápis funkcí, u kterých procházíme seznam jedenkrát, prvek po prvku, a poté vrátíme něco, co je na tom průchodu založené. Kdykoliv hodláte projít seznam, abyste něco vrátili, je pravděpodobné, že chcete fold. To je důvod, proč jsou foldy, spolu s mapami a filtry, jeden z nejužitečnějších typů funkcí ve funkcionálním programování.

Funkce a fungují podobně jako `foldl` a `foldr`, jenom jim nemusíte explicitně poskytnout počáteční hodnotu. Předpokládá se, že první (nebo poslední) prvek ze seznamu bude počáteční hodnota a ta se poté začne skládat s následujícími hodnotami. Budeme-li to mít na mysli, můžeme funkci `sum` napsat jako `sum = foldl1 (+)`. Protože záleží na tom, aby skládaný seznam obsahoval alespoň jeden prvek, vyhodí se běhová chyba, jestliže tyto funkce zavoláme na prázdný seznam. Na druhou stranu, funkce `foldl` and `foldr` fungují i s prázdnými seznamy. Když pracujete s foldy, popřemýšlejte o tom, jak by se měly chovat na prázdných seznamech. Jestliže nedává smysl pracovat s prázdným seznamem, můžete spíše použít funkci `foldl1` nebo `foldr1`.

Abych vám ukázal, jak jsou foldy užitečné, naprogramujeme si pár standardních funkcí za pomocí foldů:

```
::          =>      ->
=                   -> if    >      then    else

::      ->
=                ->    :

::          =>      ->
=

::     ->        ->      ->
  =                -> if      then    :      else

::      ->
=                ->

::      ->
=                ->
```

Funkci `head` je lepší napsat pomocí vzorů, ale takhle je ukázaný zapis pomocí foldů. Řekl bych, že naše definice funkce `reverse'` je celkem důmyslná. Vezmeme jako počáteční hodnotu prázdný seznam a poté postupujeme v našem seznamu zleva a připojujeme prvky k akumulátoru. Nakonec si vytvoříme obrácený seznam. Binární funkce `\acc x -> x : acc` vypadá podobně jako funkce `:`, jenom má přehozené parametry. To je důvod, proč bychom mohli zapsat naši funkci na obracení seznamu jako `foldl (flip (:)) []`.

Další způsob, jak si představit pravé a levé foldy je takový: řekněme, že máme pravý fold, binární funkce je `f` a počáteční funkce je `z`. Jestliže skládáme zprava seznam `[3,4,5,6]`, děláme v podstatě tohle: `f 3 (f 4 (f 5 (f 6 z)))`. Funkce `f` je zavolána na poslední prvek v seznamu a akumulátor, výsledná hodnota je předána jako akumulátor další hodnotě a tak dále. Jestliže funkce `f` bude + a počáteční akumulátor `0`, celý výraz bude `3 + (4 + (5 + (6 + 0)))`, což dává výsledek `18`. Nebo když napíšeme + jako prefixovou funkci, bude z toho `(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))`. Podobně, když skládáme zleva ten stejný seznam pomocí binární funkce `g` a akumulátorem `z`, vznikne nám výraz `g (g (g (g z 3) 4) 5) 6`. Jestliže použijeme jako binární funkci `flip (:)` a jako akumulátor `[]` (takže obracíme seznam), dostaneme z toho `flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6`. A samozřejmě, když vyhodnotíte tenhle výraz, dostanete seznam `[6,5,4,3]`.

Funkce a jsou stejné jako `foldl` a `foldr`, jenom vypisují všechny dílčí stavy akumulátoru ve formě seznamu. Existují také funkce `scanl1` a `scanr1`, jež jsou analogické k funkcím `foldl1` a `foldr1`.

```
ghci>           0  3 5 2 1
 0 3 8 10 11
ghci>           0  3 5 2 1
 11 8 3 1 0
ghci>               -> if   >     then   else       3 4 5 3 7 9 2 1
 3 4 5 5 7 9 9 9
ghci>                       3 2 1
     3   2 3   1 2 3
```

Když používáte funkci `scanl`, konečný výsledek bude poslední prvek z výsledného seznamu, kdežto `scanr` umístí výsledek na začátek.

Scany se používají na monitorování postupu funkce, která může být napsána pomocí foldu. Odpovězme si na otázku: „**Kolik prvků je potřeba pro součet druhých odmocnin všech přirozených čísel, aby přesáhl hodnotu 1000?**“ Abychom dostali odmocniny všech přirozených čísel, stačí napsat jenom `map sqrt [1..]`. A teď, pro získání součtu bychom mohli použít fold, ale protože nás zajímá, jak součet postupuje, použijeme scan. Jakmile jsme hotoví se scanem, podíváme se, kolik součtů je pod 1000. První součet v seznamu scanů bude normálně jednička. Druhý bude jedna plus druhá odmocnina ze dvou. Třetí bude to předtím plus druhá odmocnina ze tří. Jestliže existuje X součtů menších než 1000, bude potřeba X + 1 prvků pro součet, přesahující 1000.

```
        ::
        =                   1000                    1       + 1
```

```
ghci>
131
ghci>               1  131
1005.0942035344083
ghci>               1  130
993.6486803921487
```

Použijeme zde funkci `takeWhile` místo `filter`, protože funkce `filter` neumí pracovat s nekonečnými seznamy. I když víme, že je seznam vzrůstající, `filter` to neví, takže použijeme `takeWhile`, abychom generování seznamu scanů přerušili při prvním výskytu součtu větším než 1000.

Aplikace funkce pomocí $

Dále se podíváme na funkci `$`, která se také nazývá *aplikace funkce*. Nejprve se mrkneme, jak je definována:

```
  ::     ->     ->   ->
$   =
```

Co to sakra je? Co je tohle za neužitečný operátor? Je to jenom aplikovaná funkce! No skoro, ale nejen to! Zatímco obyčejná aplikace funkce (vložení mezery mezi dvě věci) má celkem vysokou prioritu, funkce `$` má prioritu nejnižší. Aplikace funkce pomocí mezery je asociativní zleva (takže `f a b c` je to stejné co `((f a) b) c)`), aplikace funkce pomocí `$` je asociativní zprava.

To je sice skvělé, ale jak nám tohle pomůže? Většinou ji využijeme pro konvenci, abychom nemuseli psát tolik závorek. Předpokládejme výraz `sum (map sqrt [1..130])`. Protože má funkce `$` tak nízkou prioritu, můžeme ho přepsat na `sum $ map sqrt [1..130]`, čímž jsme si ušetřili drahocenné úhozy na klávesnici! Když se narazí na operátor `$`, výraz napravo je aplikován jako parametr funkci nalevo. A co takhle `sqrt 3 + 4 + 9`? Tenhle výraz sečte devítku, čtyřku a druhou odmocninu trojky. Kdybychom chtěli druhou odmocninu z *3 + 4 + 9*, museli bychom napsat `sqrt (3 + 4 + 9)` nebo s použitím operátoru `$` můžeme napsat `sqrt $ 3 + 4 + 9`, protože operátor `$` má nižší prioritu než všechny ostatní. To je důvod, proč si můžete představit funkci `$` jako obdobu toho, když napíšeme otevírací závorku a poté někam hodně daleko na pravou stranu výrazu závorku uzavírací.

A co třeba `sum (filter (> 10) (map (*2) [2..10]))`? No protože je operátor `$` asociativní zprava, výraz `f (g (z x))` odpovídá výrazu `f $ g $ z x`. A tak můžeme přepsat `sum (filter (> 10) (map (*2) [2..10]))` na `sum $ filter (> 10) $ map (*2) [2..10]`.

Kromě zbavování se závorek nám operátor `$` umožňuje zacházet s aplikací funkce jako s jinými funkcemi. Tímto způsobem můžeme kupříkladu namapovat aplikaci funkce na seznam funkcí.

```
ghci>        3    4     10      2
 7.0 30.0 9.0 1.7320508075688772
```

Skládání funkcí

V matematice je skládání funkcí (kompozice) definováno takto: $(f \circ g)(x) = f(g(x))$, což znamená, že skládání dvou funkcí vytvoří novou funkci, kterou když zavoláme s parametrem, řekněme *x*, je ekvivalentní zavolání funkce *g* s parametrem *x* a poté zavolání funkce *f* na výsledek.

V Haskellu je skládání funkcí v podstatě stejná věc. Pro skládání funkcí používáme funkci `.`, jež je definována následovně:

```
  ::     ->     ->     ->     ->   ->
.   =     ->
```

Popřemýšlejte nad typovou deklarací. Funkce `f` musí mít jako parametr hodnotu se stejným typem jako je typ návratové hodnoty `g`. Takže výsledná funkce vezme parametr stejného typu jako požaduje `g` a vrátí hodnotu shodného typu, jako vrací `f`. Výraz `negate . (* 3)` vrací funkci, která vezme nějaké číslo, vynásobí ho trojkou a poté zneguje.

Jedno z použití skládání funkcí je vytvoření funkcí za chodu pro předání jiným funkcím. Jasně, můžeme na to použít lambdu, ale častokrát je skládání funkcí čistější a stručnější. Řekněme, že máme seznam čísel a chceme je všechny převést na záporná čísla. Jeden ze způsobů, jak to provést, by mohl být negace absolutní hodnoty, jako tady:

```
ghci>          ->                  5  3  6 7  3 2  19 24
```

```
  5  3  6  7  3  2  19  24
```

Všimněte si lambdy, že vypadá jako výsledek skládání funkcí. S využitím skládání funkcí to můžeme přepsat na:

```
ghci>            .       5  3  6 7  3 2  19 24
  5  3  6  7  3  2  19  24
```

Báječné! Skládání funkcí je asociativní zprava, takže můžeme skládat více funkcí najednou. Výraz `f (g (z x))` je stejný jako `(f . g . z) x`. Pokud tohle vezmeme v úvahu, můžeme převést výraz:

```
ghci>          ->                          1  5   3  6   1  7
  14  15  27
```

na výraz:

```
ghci>             .     .         1  5   3  6   1  7
  14  15  27
```

Ale co funkce s více parametry? Když je chceme použít při skládání funkcí, obvykle je musíme částečně aplikovat natolik, abychom dostali funkci, která bere pouze jeden parametr. Například `sum (replicate 5 (max 6.7 8.9))` může být zapsáno jako `(sum . replicate 5 . max 6.7) 8.9` nebo jako `sum . replicate 5 . max 6.7 $ 8.9`. O co tady jde je tohle: je vytvořena funkce, která vezme `max 6.7` a aplikuje na to `replicate 5`. Poté se vytvoří funkce, co vezme předchozí výsledek a provede součet. A nakonec je tato funkce zavolána na číslo `8.9`. Ale obvykle to čteme takto: číslo `8.9` se aplikuje na výraz `max 6.7`, poté se na to aplikuje výraz `replicate 5` a nakonec se na to aplikuje funkce `sum`. Jestliže chcete zapsat výraz s hromadou závorek za pomocí skládání funkcí, můžete začít vložením posledního parametru nejvnitřnější funkce za operátor `$` a poté prostě skládat všechny ostatní volání funkcí, zapsat je bez jejich posledního parametru a vkládat mezi ně tečky. Pokud máte výraz `replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))`, můžete ho zapsat jako `replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]`. Jestliže výraz končí třemi kulatými závorkami, je pravděpodobné, že po předělání na skládání funkcí bude obsahovat tři operátory skládání.

Další obvyklé použití skládání funkcí je definování funkce v takzvaném pointfree stylu (také se mu říká point*less* styl). Vezmeme si pro příklad funkci, kterou jsme si napsali dříve:

```
    ::          =>      ->
      =             0
```

Část `xs` je přítomna na obou pravých stranách. Díky curryfikaci můžeme `xs` vynechat, protože zavolání `foldl (+) 0` vytvoří funkci, která požaduje seznam. Zápisu `sum' = foldl (+) 0` takové funkce se říká zápis v pointfree stylu. Jak bychom mohli napsat tohle v pointfree stylu?

```
    =                                 50
```

Nemůžeme se jednoduše zbavit `x` na obou stranách. Parametr `x` je v těle funkce uzávorkovaný. Výraz `cos (max 50)` by nedával smysl. Nemůžete dostat kosinus funkce. Uděláme to, že vyjádříme `fn` jako skládání funkcí.

```
  =         .         .      .      .      50
```

Skvělé! Pointfree styl bývá často čitelnější a stručnější, protože nás nutí přemýšlet o funkcích a o výsledcích skládání funkcí místo přemýšlení nad daty a jejich přesouvání. Můžete vzít jednoduchou funkci a použít skládání jako lepidlo na vytvarování složitějších funkcí. Nicméně někdy může být zápis funkce v pointfree stylu méně čitelný, pokud je funkce příliš komplikovaná. To

je důvod, proč se nedoporučuje dělat dlouhé řetězce poskládaných funkcí, ačkoliv se přiznávám, že se občas ve skládání funkcí vyžívám. Preferovaný styl je používat konstrukci *let* pro označování mezivýsledků nebo rozdělení problému na podproblémy a poté je dát dohromady, místo vytváření dlouhých řetězců funkcí, aby to dávalo smysl i případnému čtenáři.

V sekci o mapách a filtrech jsme řešili problém hledání součtu všech lichých čtverců, které byly menší než 10000. Takto vypadá řešení, když se vloží do funkce.

```
::
=                  10000                    2    1
```

Jako fanoušek skládání funkcí bych to nejspíš zapsal takhle:

```
::
=     .            10000  .           .       2  $  1
```

Nicméně kdyby byla šance, že někdo jiný bude číst můj kód, napsal bych to asi takhle:

```
          ::
          =
let             =            $       2    1
                =             10000
in
```

Nevyhrál bych s tím žádný programátorský turnaj, ale člověk, co by to četl, by to patrně považoval za čitelnější než řetězec poskládaných funkcí.

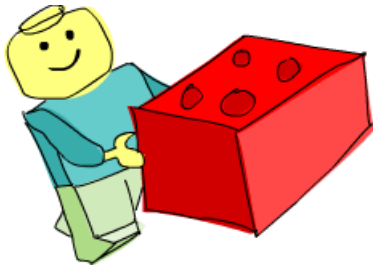
Rekurze Obsah Moduly

← Funkce vyššího řádu Obsah Vytváříme si své typy a typové třídy →

Moduly

English version

Načítání modulů



Modul je v Haskellu kolekce souvisejících funkcí, typů a typových tříd. Program je v Haskellu kolekce modulů, kde hlavní modul načte ostatní moduly a poté použije funkce, které jsou v nich definované a něco pomocí nic udělá. Rozdělení kódu do několika modulů má celkem dost výhod. Jestliže je modul dostatečně obecný, funkce v něm mohou být použity ve velkém množství odlišných programů. Pokud je váš kód oddělený do samostatných modulů, které na sobě příliš nezávisí (říkáme jim také volně vázané), můžeme je později použít znovu. To dělá celou záležitost psaní kódu zvládnutelnější, když je kód rozdělený do více částí a každá část má svůj účel.

Haskellová standardní knihovna je rozdělená do modulů a každý z nich obsahuje funkce a typy, které spolu souvisí a slouží ke společnému účelu. Existuje modul pro zacházení se seznamy, modul pro souběžné programování, modul, jež se zabývá komplexními čísly atd. Všechny funkce, typy a typové třídy, se kterými jsme se zatím potkali, byly součástí modulu `Prelude`, jenž se importuje automaticky. V této kapitole prozkoumáme několik užitečných modulů a v nich obsažených funkcí. Ale nejprve se podíváme na importování modulů.

Syntax pro importování modulů v haskellovém skriptu je `import <název modulu>`. Musí se to napsat před definice jakýchkoliv funkcí, takže se importy většinou nachází na začátku souboru. Jeden skript může importovat samozřejmě více modulů — stačí každý import umístit na samostatný řádek. Naimportujme si modul `Data.List` obsahující několik užitečných funkcí pro práci se seznamy a použijme funkci, kterou exportuje, na vytvoření funkce, co nám řekne, kolik unikátních prvků seznam má.

```
import

          ::         =>      ->
          =         .
```

Když do nějakého souboru napíšete `import Data.List`, všechny funkce, které exportuje modul `Data.List`, se stanou dostupné v globálním jmenném prostoru, což znamená, že je můžete zavolat kdekoliv v daném skriptu. Funkce `nub` je definovaná v `Data.List` tak, že vezme seznam a vyřadí z něj duplicitní prvky. Složením funkcí `length` a `nub` napsáním `length . nub` vytvoří funkci, jež je ekvivalentní funkci `\xs -> length (nub xs)`.

Můžete také vložit funkce z modulů do globálního jmenného prostoru GHCi. Když máte spuštěné GHCi a chcete mít možnost zavolat funkce, které exportuje modul `Data.List`, napište tohle:

```
ghci>    +
```

Jestliže chceme načíst více modulů v GHCi, nemusíme psát `:m +` několikrát, můžeme prostě načíst několik modulů najednou.

```
ghci>    +
```

Nicméně pokud načítáte skript, který už importuje nějaké moduly, nemusíte používat `:m +`, abyste k nim přistupovali.

Pokud potřebujete jenom pár funkcí z modulu, můžete si selektivně importovat jenom tyto funkce. Kdybychom chtěli importovat pouze funkce `nub` a `sort` z modulu `Data.List`, udělali bychom tohle:

```
import
```

Můžete také chtít importovat všechny funkce z modulu kromě několika vybraných. To je často užitečné, když několik modulů exportuje funkci se stejným názvem a vy se chcete zbavit těch nepotřebných. Řekněme, že už máme svou vlastní funkci nazvanou `nub` a budeme chtít importovat všechny funkce z modulu `Data.List` kromě funkce `nub`:

```
import           hiding
```

Jiný způsob, jak se vypořádávat s kolizemi, je používat kvalifikované importy. Modul `Data.Map`, nabízející datovou strukturu pro vyhledávání hodnot podle klíče, exportuje hromadu funkcí se stejným názvem jako mají funkce v `Prelude`, jako třeba `filter` nebo `null`. Takže jakmile importujeme `Data.Map` a poté zavoláme funkci `filter`, Haskell si nebude jistý, kterou z těch dvou funkcí použít. Tady je ukázané, jak to vyřešit:

```
import %0000entd
```

Tohle zajistí, že když budeme chtít použít funkci `filter` z modulu `Data.Map`, musíme napsat `Data.Map.filter`, kdežto pouhé `filter` stále odkazuje na normální funkci `filter`, jak ji známe a milujeme. Vypisování `Data.Map` před každou funkci z toho modulu je celkem jednotvárné. To je důvod proč máme možnost přejmenovat kvalifikovaný import na něco kratšího:

```
import qualified          as
```

Teď stačí pro použití funkce `filter` z modulu `Data.Map` napsat `M.filter`.

Rozhodně nahlédněte do [této užitečné referenční dokumentace](), abyste viděli, které moduly jsou ve standardní knihovně. Dobrý způsob, jak pochytit nové znalosti o Haskellu, je proklikávat se dokumentací a prozkoumávat moduly a jejich funkce. Můžete si také prohlížet zdrojový kód jednotlivých modulů Haskellu. Čtení kódu některých modulů je vážně dobrý způsob jak se naučit Haskell a získat pro něj náležitý cit.

Na hledání funkcí nebo zjišťování, kde jsou umístěné, používejte [Hoogle](). Je to opravdu skvělý haskellový vyhledávač. Můžete hledat podle názvu funkcí, modulů, nebo dokonce podle typu funkce.

Data.List

Modul `Data.List` se kupodivu celý věnuje seznamům. Poskytuje několik velice užitečných funkcí pro zacházení s nimi. Některé funkce jsme už potkali (jako `map` a `filter`), protože modul `Prelude` nechává příhodně exportovat některé funkce z `Data.List`. Nemusíte importovat `Data.List` kvalifikovaně, protože nekoliduje s žádným názvem z modulu `Prelude`, kromě těch, které `Prelude` už nakradlo z modulu `Data.List`. Pojďme se podívat na nějaké z funkcí se kterými jsme se zatím nesetkali.

Funkce vezme prvek a seznam a poté vloží ten prvek mezi každý pár prvků v seznamu. Tady je ukázka:

```
ghci>             '.' "OPIČÁK"
"O.P.I.Č.Á.K"
ghci>             0  1 2 3 4 5 6
 1 0 2 0 3 0 4 0 5 0 6
```

Funkce vezme seznam seznamů a seznam. Ten poté vloží mezi všechny ty seznamy a poté zarovná výsledek.

```
ghci>             " "  "hej" "nazdar" "kluci"
```

```
"hej nazdar kluci"
ghci>              0 0 0    1 2 3   4 5 6   7 8 9
 1 2 3 0 0 0 4 5 6 0 0 0 7 8 9
```

Funkce prohodí (transponuje) seznam seznamů. Pokud si představíte seznam seznamů jako 2D matici, řádky se stanou sloupci a naopak.

```
ghci>             1 2 3   4 5 6   7 8 9
  1 4 7   2 5 8   3 6 9
ghci>            "hej" "nazdar" "kluci"
 "hnk" "eal" "jzu" "dc" "ai" "r"
```

Řekněme, že máme polynomy $3x^2 + 5x + 9$, $10x^3 + 9$ a $8x^3 + 5x^2 + x - 1$ a chceme je spojit dohromady. Můžeme je v Haskellu reprezentovat pomocí seznamů `[0,3,5,9]`, `[10,0,0,9]` a `[8,5,1,-1]`. A teď, abychom je sečetli, jediné, co musíme udělat, je tohle:

```
ghci>        $             0 3 5 9   10 0 0 9   8 5 1  1
 18 8 6 17
```

Když prohazujeme tyhle tři seznamy, třetí mocniny jsou pak v prvním řádku, druhé ve druhém a tak dále. Namapováním funkce `sum` na tento transponovaný seznam seznamů dosáhneme požadovaného výsledku.

Funkce , , a jsou striktní verze svých příslušných líných podob. Když použijeme líný fold na opravdu velký seznam, může nastat chyba přetečení zásobníku. Viník této chyby je líná podstata foldů, protože hodnota akumulátoru se ve skutečnosti při skládání neaktualizuje. Co se děje ve skutečnosti je to, že akumulátor tak nějak slibuje, že spočítá svou hodnotu, jakmile se to po něm bude chtít. Takhle je to u každého akumulátoru s mezivýsledkem a všechny tyhle nahromaděné sliby přetečou váš zásobník. Striktní foldy nejsou líní lemplové a ve skutečnosti vypočítají mezivýsledky hned jak k nim přicházejí místo aby plnily zásobník sliby. Takže jestliže někdy dostanete chybu přetečení zásobníku při skládání seznamů, zkuste přejít na striktní verzi foldů.

Funkce (zřetězení) zarovná seznam seznamů na prostý seznam prvků.

```
ghci>        "foo" "bar" "baz"
"foobarbaz"
ghci>          3 4 5   2 3 4   2 1 1
 3 4 5 2 3 4 2 1 1
```

Jenom to prostě odstraní jednu úroveň zanoření. Takže pokud chceme zarovnat výraz `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`, který je seznam seznamů seznamů, musíme použít funkci dvakrát.

Napsáním funkce uděláme stejnou věc, jako kdybychom nejprve namapovali funkci na seznam a poté ji zřetězili pomocí funkce `concat`.

```
ghci>                    4   1  3
 1 1 1 1 2 2 2 2 3 3 3 3
```

Funkce vezme seznam booleovských hodnot a vrátí `True` pouze pokud jsou *všechny* hodnoty v seznamu rovny `True`.

```
ghci>     $       4   5 6 7 8

ghci>     $        4   4 4 4 3 4
```

Funkce je podobná funkci `and`, jenom vrátí `True` pokud *nějaká* z booleovských hodnot v seznamu je `True`.

```
ghci>    $        4   2 3 4 5 6 1
ghci>    $       4   1 2 3
```

Funkce a vezmou predikát a poté zkontrolují zda-li některý nebo všechny prvky v seznamu vyhovují predikátu, v tomto pořadí. Obvykle použijeme tyhle dvě funkce místo abychom na seznam namapovali funkce `and` nebo `or`.

```
ghci>        4   2 3 5 6 1 4
ghci>       4   6 9 10
ghci>      `elem`  'A'  'Z'   "HEJKLUCIjakje"
ghci>      `elem`  'A'  'Z'   "HEJKLUCIjakje"
```

Funkce vezme funkci a počáteční hodnotu. Následně aplikuje funkci na počáteční hodnotu, poté aplikuje funkci na výsledek té aplikace funkce, poté opět funkci na výsledek předchozí aplikace atd. Vrátí všechny výsledky ve formě nekonečného seznamu.

```
ghci>      10 $           2  1
 1 2 4 8 16 32 64 128 256 512
ghci>      3 $             "haha"  "haha"
 "haha" "hahahaha" "hahahahahaha"
```

Funkce vezme číslo a seznam. Potom rozdělí seznam na pozici, kterou zadává číslo, a jako výsledek vrátí dva seznamy v n-tici.

```
ghci>         3 "hejchlape"
 "hej" "chlape"
ghci>         100 "hejchlape"
 "hejchlape" ""
ghci>           3  "hejchlape"
 "" "hejchlape"
ghci> let       =         3 "foobar" in   ++
"barfoo"
```

Funkcička je opravdu užitečná. Postupně tahá prvky ze seznamu, zatímco platí predikát, a poté, když narazí na prvek, jež nevyhovuje predikátu, je tahání přerušeno. To se ukázalo být velmi užitečné.

```
ghci>            3   6 5 4 3 2 1 2 3 4 5 4 3 2 1
 6 5 4
ghci>             ' '  "Tohle je věta."
"Tohle"
```

Řekněme, že chceme znát součet všech třetích mocnin (přirozených čísel) menších než 10000. Nemůžeme namapovat výraz `(^3)` na seznam `[1..]`, aplikovat nějaký filtr a poté to celé zkusit sečíst, protože filtrování nekonečného seznamu nikdy neskončí. Možná víte, že tahle řada prvků narůstá, ale Haskell ne. To je důvod, proč můžeme udělat tohle:

```
ghci>    $            10000  $       3   1
53361
```

Aplikujeme výraz `(^3)` na nekonečný seznam a hned poté co narazíme na prvek, který je větší nebo rovný číslu 10000, seznam je oříznut. Pak tedy můžeme výsledek jednoduše sečíst.

Funkce je podobná, jenom zahazuje všechny prvky, dokud je predikát pravdivý. Jakmile se jednou predikát vyhodnotí jako `False`, vrátí se zbytek seznamu. Extrémně užitečná a původná funkce!

```
ghci>              ' '  "Tohle je věta."
" je věta."
ghci>            3   1 2 2 2 3 4 5 4 3 2 1
 3 4 5 4 3 2 1
```

Byl nám zadán seznam představující hodnotu akcií k určitému datu. Tento seznam je vytvořený z n-tic jejichž první složka je hodnota akcií, druhá je rok, třetí měsíc a čtvrtá den. Chtěli bychom vědět, kdy hodnota akcií poprvé přesáhla tisíc dolarů!

```
ghci> let       =   994.4 2008 9 1   995.2 2008 9 2   999.2 2008 9 3   1001.4 2008 9 4
 998.3 2008 9 5
ghci>                                ->     < 1000
 1001.4 2008 9 4
```

Funkce je podobná funkci `takeWhile`, jenom vrací dvojici seznamů. První seznam obsahuje všechno co by obsahovalo zavolání funkce `takeWhile` na stejný predikát a seznam. Druhý seznam obsahuje část seznamu, která by se zahodila.

```
ghci> let           =         ' '  "Tohle je věta." in "První slovo: " ++    ++ ", zbytek:" +
+
"První slovo: Tohle, zbytek: je věta."
```

Zatímco funkce `span` rozdělí seznam za místem, kde predikát platí, funkce ho roztrhne tam, kde platí poprvé. Když napíšeme `break p`, je to stejné jako bychom napsali `span (not . p)`.

```
ghci>          4   1 2 3 4 5 6 7
  1 2 3   4 5 6 7
ghci>         4   1 2 3 4 5 6 7
  1 2 3   4 5 6 7
```

Po použití funkce `break` bude druhý výsledný seznam začínat prvním prvkem, který vyhovuje predikátu.

Funkce jednoduše seřadí seznam. Typ prvků v seznamu musí patřit do typové třídy `Ord`, protože pokud prvky v seznamu nemají stanovené uspořádání, seznam nemůže být seřazen.

```
ghci>       8 5 3 2 1 6 4 2
 1 2 2 3 4 5 6 8
ghci>      "Tohle bude brzy serazeno"
"   Tabbdeeeehlnoorrsuyzz"
```

Funkce vezme seznam a seskupí sousedící prvky, které jsou si rovny, do dalšího seznamu.

```
ghci>        1 1 1 1 2 2 2 2 3 3 2 2 2 5 6 7
  1 1 1 1   2 2 2 2   3 3   2 2 2   5   6   7
```

Jestliže seřadíme seznam před seskupením, můžeme zjistit, kolikrát se v tom seznamu jednotlivé prvky vyskytují.

```
ghci>              ->             .       .      $  1 1 1 1 2 2 2 2 3 3 2 2 2 5 6 7
  1 4   2 7   3 2   5 1   6 1   7 1
```

Funkce a jsou podobné funkcím `init` a `tail`, jenom s tím rozdílem, že se aplikují rekurzivně na seznam tak dlouho dokud něco ještě zbývá. Sledujte.

```
ghci>       "w00t"
 "" "w" "w0" "w00" "w00t"
ghci>       "w00t"
 "w00t" "00t" "0t" "t" ""
ghci> let   = "w00t" in
  "" "w00t"   "w" "00t"   "w0" "0t"   "w00" "t"   "w00t" ""
```

Zkusíme použít fold pro implementaci hledání částí seznamu.

```
       ::        =>     ->     ->
                      =
    let      =
    in               -> if            ==        then      else
```

Nejprve zavoláme funkci `tails` na seznam ve kterém hledáme. Poté projdeme každý zbytek a zjistíme, jestli začiná tím stejným co hledáme.

Takhle jsme vlastně vytvořili funkci která se chová stejně jako . Funkce `isInfixOf` hledá v seznamu jeho část a vrací `True` jestliže se hledaná část vyskytuje někde v cílovém seznamu.

```
ghci> "zloděj" `isInfixOf` "jsem zloděj koček"
ghci> "Zloděj" `isInfixOf` "jsem zloděj koček"
ghci> "zloději" `isInfixOf` "jsem zloděj koček"
```

Funkce a hledají část seznamu na jeho začátku a konci, v tomto pořadí.

```
ghci> "hej" `isPrefixOf` "hej ty tam!"
ghci> "hej" `isPrefixOf` "sakra, hej ty tam!"
ghci> "tam!" `isSuffixOf` "hej ty tam!"
ghci> "tam!" `isSuffixOf` "hej ty tam"
```

Funkce a zjišťují, jestli prvek je nebo není v daném seznamu.

Funkce vezme nějaký seznam a predikát a vrátí dvojici seznamů. První seznam ve výsledku obsahuje všechny prvky vyhovující predikátu, druhý obsahuje všechny nevyhovující.

```
ghci>           `elem`  'A'  'Z'   "BOBsidneyMORGANeddy"
 "BOBMORGAN" "sidneyeddy"
ghci>            3   1 3 5 6 3 2 1 0 3 7
  5 6 7   1 3 3 2 1 0 3
```

Je důležité porozumět tomu, jak se tohle liší oproti funkcím `span` a `break`:

```
ghci>       `elem`  'A'  'Z'   "BOBsidneyMORGANeddy"
 "BOB" "sidneyMORGANeddy"
```

Zatímco funkce `span` a `break` jsou hotovy jakmile narazí na první prvek který nevyhovuje nebo vyhovuje predikátu, funkce `partition` prochází celý seznam a rozděluje ho podle predikátu.

Funkce vezme nějaký seznam a predikát a vrací první prvek který vyhovuje tomu predikátu. Avšak vrátí nám ho zabalený v datovém typu `Maybe`. Algebraické datové typy probereme více do hloubky v následující kapitole, ale prozatím nám stačí vědět, že hodnota datového typu `Maybe` může být buď `Just něco` nebo `Nothing`. Podobně jako seznam může být buď prázdný seznam nebo seznam obsahující nějaké prvky, `Maybe` je možná prázdné (neobsahuje nic) nebo možná obsahuje právě jeden prvek. A jako je typ seznamu kupříkladu celých čísel `[Int]`, typ, který možná obsahuje celé číslo, je `Maybe Int`. V každém případě, pojďme si vzít naši funkci `find` na projížďku.

```
ghci>        4   1 2 3 4 5 6
     5
ghci>        9   1 2 3 4 5 6

ghci>
     ::    ->       ->     ->
```

Všimněte si typu funkce `find`. Její výsledek je `Maybe a`. To je podobné jako když máme typ `[a]`, akorát hodnota typu `Maybe` může obsahovat buď žádné prvky nebo jeden prvek, zatímco seznam může obsahovat žádné prvky, jeden prvek, nebo více prvků.

Vzpomeňte si jak jsme určovali, kdy poprvé hodnota našich akcií přesáhla tisíc dolarů. Vytvořili jsme výraz `head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`. Vzpomeňte si, že funkce `head` rozhodně není bezpečná. Co by se stalo, kdyby hodnota našich akcií nikdy nepřekonala tisíc dolarů? Naše aplikace funkce `dropWhile` by vrátila prázdný seznam a pokus o získání počátku toho seznamu by skončil běhovou chybou. Nicméně pokud to přepíšeme na `find (\(val,y,m,d) -> val > 1000) stock`, bude to mnohem bezpečnější. Jestliže naše akcie nikdy nepřesáhly tisíc dolarů (tedy jestliže žádný prvek nevyhovoval predikátu), dostali bychom hodnotu `Nothing`. Ale ten seznam obsahoval platnou odpověď, dostali bychom něco jako `Just (1001.4,2008,9,4)`.

Funkce je něco jako `elem`, jenom nevrací booleovskou hodnotu. Možná vrátí index prvku, který hledáme. Jestliže se v našem seznamu prvek nevyskytuje, vrátí se `Nothing`.

```
ghci>
          ::        =>   ->     ->
ghci> 4 `elemIndex`  1 2 3 4 5 6
     3
ghci> 10 `elemIndex`  1 2 3 4 5 6
```

Funkce je podobná funkci `elemIndex`, jenom vrátí seznam indexů určujících vícenásobný výskyt prvku v našem seznamu. Jelikož používáme seznam pro znázornění indexů, nepotřebujeme typ `Maybe`, protože neúspěch se dá vyjádřit prázdným seznamem, který je synonymní s `Nothing`.

```
ghci> ' ' `elemIndices` "Kde tu jsou mezery?"
 3 6 11
```

Funkce je podobná funkci `find`, ale jenom možná vrátí index prvního prvku, který vyhovuje predikátu. Podobně tak funkce vrátí indexy všech prvků, které vyhovují predikátu, ve formě seznamu.

```
ghci>             4   5 3 2 1 6 4
     5
ghci>             7   5 3 2 1 6 4

ghci>             `elem`  'A'  'Z'   "Kde Tu Jsou Verzálky?"
```

```
0 4 7 12
```

Funkcemi `zip` a `zipWith` jsme se už zabývali. Vysvětlili jsme si, že tyto funkce sepnou dva seznamy, buď jako dvojici, nebo pomocí binární funkce (což je funkce se dvěma parametry). Ale co když chceme dát dohromady tři seznamy? Nebo sepnout tři seznamy pomocí funkce se třemi parametry? Tak k tomu nám slouží funkce , atd. a funkce , atd. Takhle to pokračuje až k číslu 7. I když se to může zdát nedostatečné, úplně to stačí, protože se nestává moc často, že byste potřebovali dávat dohromady osm a víc seznamů. Mimo to existuje velmi chytrý způsob, jak sepnout nekonečný počet seznamů, ale zatím jsme nepokročili natolik, abychom si to mohli ukázat.

```
ghci>                 ->   +   +     1 2 3   4 5 2 2   2 2 3
 7 9 8
ghci>       2 3 3   2 2 2   5 5 3   2 2 2
  2 2 5 2   3 2 5 2   3 2 3 2
```

Stejně jako u normálního spínání se seznamy, které jsou delší než nejkratší seznam z těch spínaných, ořízne na jeho velikost.

Funkce je užitečná, pokud se snažíme zpracovat soubory nebo vstup odněkud. Vezme řetězec a vrátí každý řádek toho řetězce jako položku seznamu.

```
ghci>        "první řádek\ndruhý řádek\ntřetí řádek"
 "první řádek" "druhý řádek" "třetí řádek"
```

Znak `'\n'` znázorňuje ukončení řádku v UNIXu. Zpětné lomítko má v haskellových řetězcích a znacích zvláštní význam.

Funkce je opačná funkce k funkci `lines`. Vezme seznam řetězců a spojí je pomocí znaku `'\n'`.

```
ghci>          "první řádek"  "druhý řádek"  "třetí řádek"
"první řádek\ndruhý řádek\ntřetí řádek\n"
```

Funkce a jsou určené na rozdělování řádku textu na slova a na spojování seznamu slov do textu. Jsou dost užitečné.

```
ghci>       "hej tohle jsou slova v téhle větě"
 "hej" "tohle" "jsou" "slova" "v" "téhle" "větě"
ghci>       "hej tohle        jsou   slova v téhle\nvětě"
 "hej" "tohle" "jsou" "slova" "v" "téhle" "větě"
ghci>          "hej"  "nazdar"  "kámo"
"hej nazdar kámo"
```

Funkci jsme si již také zmínili. Vezme seznam a vytřídí z něj duplikátní prvky, což nám vrátí seznam, jehož každý prvek je sněhová unikátní vločka! Tahle funkce má celkem divné jméno. Zjistil jsem, že „nub“ znamená v angličtině malou hrudku nebo nezbytnou část něčeho. Podle mého názoru by měli používat opravdová slova pro názvy funkcí místo archaických.

```
ghci>      1 2 3 4 3 2 1 2 3 4 3 2 1
 1 2 3 4
ghci>     "Hrozně moc slov a tak"
"Hrozně mcslvatk"
```

Funkce vezme nějaký prvek a k němu seznam a odstraní první výskyt toho prvku v tom seznamu.

```
ghci>        'h' "hej ty sthará bandho!"
"ej ty sthará bandho!"
ghci>        'h' .        'h' $ "hej ty sthará bandho!"
"ej ty stará bandho!"
ghci>        'h' .        'h' .        'h' $ "hej ty sthará bandho!"
```

```
"ej ty stará bando!"
```

Operátor je rozdílová funkce pro seznamy. Chová se v zásadě jako rozdíl množin. Za každý prvek v pravém seznamu odstraní výskyt odpovídajícího prvku v levém.

```
ghci>  1  10      2 5 9
 1 3 4 6 7 8 10
ghci> "Já, velké dítě"    "velké"
"Já,  dítě"
```

Napsání `[1..10] \\ [2,5,9]` je stejné jako `delete 2 . delete 5 . delete 9 $ [1..10]`.

Funkce se také chová podobně jako množinová funkce. Vrátí sjednocení dvou seznamů. V podstatě projde všechny prvky ve druhém seznamu a připojí je k prvnímu, jestliže tam už nejsou obsaženy. Dávejte si ovšem pozor, že duplikáty z druhého seznamu zmizí!

```
ghci> "hej chlape" `union` "chlape jak je"
"hej chlapek"
ghci>  1  7  `union`  5  10
 1 2 3 4 5 6 7 8 9 10
```

Funkce funguje jako průnik množin. Vrátí pouze ty prvky, které jsou nalezeny v obou seznamech.

```
ghci>  1  7  `intersect`  5  10
 5 6 7
```

Funkce vezme nějaký prvek seznam prvků, které mohou být porovnány, a vloží ho do toho seznamu. Vloží ho tam tak, aby všechny prvky, jež jsou větší nebo rovné, byly napravo od něj. Jestliže použijeme funkci `insert` pro vložení prvku do seřazeného seznamu, tento seznam zůstane seřazený.

```
ghci>       4  1 2 3 5 6 7
 1 2 3 4 5 6 7
ghci>       'g' $  'a'  'f'  ++  'h'  'z'
"abcdefghijklmnopqrstuvwxyz"
ghci>       3  1 2 4 3 2 1
 1 2 3 4 3 2 1
```

Funkce `length`, `take`, `drop`, `splitAt`, `!!` a `replicate` mají společnou vlastnost, že vezmou nějakou hodnotu typu `Int` jako jeden z jejich parametrů, přestože by mohlo být mnohem obecnější a použitelnější, kdyby prostě akceptovali jakýkoliv typ, který je součástí typových tříd `Integral` nebo `Num` (záleží na jednotlivých funkcích). Je tomu tak z historických důvodů. Nicméně, po opravě tohohle by přestalo fungovat velké množství existujícího kódu. To je důvod, proč modul `Data.List` obsahuje obecnější protějšky těchhle funkcí, pojmenované , , , , a . Kupříkladu funkce `length` má typ `length :: [a] -> Int`. Když si zkusíme vypočítat průměr seznamu čísel napsáním `let xs = [1..6] in sum xs / length xs`, dostaneme typovou chybu, protože nemůžeme použít operátor `/` k celočíselnému dělení. Funkce `genericLength` má, na druhou stranu, typ `genericLength :: (Num a) => [b] -> a`. Protože se `Num` může chovat jako desetinné číslo, průměr se napsáním `let xs = [1..6] in sum xs / genericLength xs` spočítá bez problémů.

Každá z funkcí `nub`, `delete`, `union`, `intersect` a `group` má svůj obecnější protějšek nazvaný , , , a . Rozdíl mezi nimi je v tom, že první skupina funkcí používá operátor `==` pro testování rovnosti, zatímco ty s *By* vezmou porovnávací funkci a poté pomocí ní testují prvky v seznamu. Funkce `group` tedy odpovídá `groupBy (==)`.

Kupříkladu si vezměme seznam popisující chování funkce každou sekundu. Chtěli bychom ho rozdělit do podseznamů podle toho, jestli hodnoty byly menší nebo větší než nula. Kdybychom použili obyčejné `group`, museli bychom seskupit pouze sousední hodnoty, které se rovnají. Ale my je chceme seskupit na základě toho, zda-li jsou záporné nebo ne. Proto na scénu vstupuje funkce `groupBy`! Porovnávací funkce poskytovaná funkci s *By* by měla vzít dva prvky stejného typu a vrátit `True`, jestliže je dle svých standardů považuje za rovné.

```
ghci> let        =   4.3   2.4   1.2  0.4  2.3  5.9  10.5  29.1  5.3   2.4   14.5  2.9  2.3
ghci>                ->    > 0  ==    > 0
   4.3  2.4  1.2   0.4 2.3 5.9 10.5 29.1 5.3    2.4  14.5   2.9 2.3
```

Z příkladu jasně vidíme, které sekce jsou kladné a které záporné. Poskytnutá porovnávací funkce vezme dva prvky a poté vrátí `True` pouze pokud jsou oba záporné nebo kladné. Tato funkce může býta také zapsána jako `\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`, ačkoliv si myslím, že ten první způsob je mnohem čitelnější. Ještě přehlednější způsob jak zapsat porovnávající funkce pro funkce s *By* je importování funkce z modulu `Data.Function`. Tato funkce je definována jako:

```
   ::    ->   ->    ->    ->    ->   ->   ->
`on`   =      ->
```

Takže výraz `(==) `on` (> 0)` vrátí porovnávací funkci, která odpovídá funkci `\x y -> (x > 0) == (y > 0)`. Funkce on se hodně používá s funkcemi s *By*, protože pomocí ní můžeme napsat:

```
ghci>              `on`    0
   4.3  2.4  1.2   0.4 2.3 5.9 10.5 29.1 5.3    2.4  14.5   2.9 2.3
```

To je opravdu hodně čitelné! Dá to i přečíst nahlas: seskup hodnoty pomocí porovnání jestli jsou prvky větší než nula.

Podobně také funkce `sort`, `insert`, `maximum` a `minimum` mají své obecnější protějšky. Funkce jako `groupBy` vezmou funkci, která určuje, kdy se dva prvky rovnají. Funkce , , a vezmou funkci, která určí, kdy je jeden prvek větší, menší nebo rovný druhému. Typ funkce `sortBy` je `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. Jestli si vzpomínáte z dřívějška, typ `Ordering` může nabývat hodnoty `LT`, `EQ`, nebo `GT`. Funkce `sort` odpovídá funkci `sortBy compare`, protože funkce `compare` prostě vezme dva prvky jejichž typ patří do typové třídy `Ord`, a vrátí jejich uspořádání.

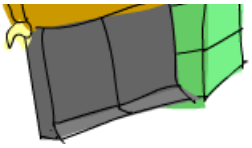
Seznamy mohou být porovnávány, a když na to dojde, tak jde o lexikografické porovnání. Co když máme nějaký seznam seznamů a chtěli bychom ho seřadit ne podle obsahu vnitřních seznamů, ale podle jejich délky? No, pravděpodobně jste odhadli, že bychom na to použili funkci `sortBy`.

```
ghci> let    =   5 4 5 4 4   1 2 3   3 5 4 3      2   2 2
ghci>                `on`
     2   2 2   1 2 3   3 5 4 3   5 4 5 4 4
```

Paráda! Část `compare `on` length` je v podstatě normální angličtina! Jestli si nejste jistí, k čemu tady funkce on je, tak výraz `compare `on` length` funguje stejně jako `\x y -> length x `compare` length y`. Když se zabýváte funkcemi s *By* využívajícími porovnávací funkce, obvykle napíšete `(==) `on` něco`, a když se zabýváte funkcemi s *By* využívajícími porovnávací funkce, obvykle napíšete `compare `on` něco`.

Data.Char

Modul `Data.Char` dělá přesně to, co jeho název napovídá. Exportuje funkce zabývajícími se znaky. Také je užitečný při filtrování a mapování řetězců, protože to jsou vlastně jenom seznamy znaků.



Mimo jiné modul `Data.Char` exportuje znakové predikáty. To jsou funkce, které vezmou nějaký znak a řeknou nám o něm, jestli určité předpoklady platí nebo ne. Tady je máme:

Funkce zkontroluje, zda-li se jedná o řídící znak.

Funkce zkontroluje, zda-li se jedná o prázdné znaky. To zahrnuje mezery, tabulátory, nové řádky apod.

Funkce zkontroluje, zda-li se jedná o minusku (malé písmeno).

Funkce zkontroluje, zda-li se jedná o verzálku (velké písmeno).

Funkce zkontroluje, zda-li se jedná o písmeno.

Funkce zkontroluje, zda-li se jedná o písmeno nebo číslici.

Funkce zkontroluje, zda-li se jedná o tisknutelný znak. Kupříkladu řídící znaky nejsou tisknutelné.

Funkce zkontroluje, zda-li se jedná o číslici.

Funkce zkontroluje, zda-li se jedná o oktalovou (osmičkovou) číslici.

Funkce zkontroluje, zda-li se jedná o hexadecimální (šestnáctkovou) číslici.

Funkce zkontroluje, zda-li se jedná o písmeno. (Je totožná s funkcí `isAlpha`.)

Funkce zkontroluje, zda-li se jedná o unikódové diakritické znaménko. To jsou znaky, které se kombinují s předcházejícími písmeny, aby vytvořily znak s diakritikou. Používejte tohle, pokud jste Francouz.

Funkce zkontroluje, zda-li se jedná o číslici. (Je totožná s funkcí `isDigit`.)

Funkce zkontroluje, zda-li se jedná o interpunkci.

Funkce zkontroluje, zda-li se jedná o elegantní matematický symbol nebo označení měny.

Funkce zkontroluje, zda-li se jedná o unikódovou mezeru nebo oddělovač.

Funkce zkontroluje, zda-li se znak nachází mezi prvními 128 pozicemi znakové sady.

Funkce zkontroluje, zda-li se znak nachází mezi prvními 256 pozicemi znakové sady.

Funkce zkontroluje, zda-li se jedná o ASCII verzálku.

Funkce zkontroluje, zda-li se jedná o ASCII minusku.

Všechny tyhle predikáty jsou typu `Char -> Bool`. Většinou je budete používat k filtrování řetězců nebo k podobnému účelu. Kupříkladu řekněme, že vytváříme program, který požaduje uživatelské jméno a tohle uživatelské jméno se může sestávat pouze z alfanumerických znaků. Můžeme použít funkci `all` z modulu `Data.List` v kombinaci s predikátem z modulu `Data.Char`, abychom stanovili, zda-li je uživatelské jméno v pořádku.

```
ghci>                "bobby283"
ghci>                "eddy ryba!"
```

Bezva. Pro případ že si nevzpomínáte, funkce `all` vezme predikát a nějaký seznam a vrátí hodnotu `True` pouze tehdy, když predikát vyhovuje každému prvku z daného seznamu.

Také můžeme použít funkci `isSpace` pro simulaci funkce `words` z modulu `Data.List`.

```
ghci>       "hej kluci to jsem já"
 "hej" "kluci" "to" "jsem" "já"
ghci>              `on`          "hej kluci to jsem já"
 "hej" " " "kluci" " " "to" " " "jsem" " " "já"
ghci>
```

Hmmm, no, dělá to něco podobného co funkce `words`, ale zůstanou nám prvky obsahující mezery. Hmm, co bychom s tím mohli dělat? Já vím, odfiltrujeme ty zmetky.

```
ghci>            .             .              `on`          $ "hej kluci to jsem já"
 "hej" "kluci" "to" "jsem" "já"
```

Ach.

Modul `Data.Char` také exportuje datový typ podobající se typu `Ordering`. Typ `Ordering` může nabývat hodnoty `LT`, `EQ` nebo `GT`. Je to jakýsi výčet. Popisuje několik možných výsledků, jež mohou nastat při porovnávání dvou prvků. Typ `GeneralCategory` je rovněž výčtový. Poskytuje nám několik možných kategorií, do kterých znak může spadat. Hlavní funkce pro získání obecné kategorie se nazývá `generalCategory`. Její typ je `generalCategory :: Char -> GeneralCategory`. Existuje nějakých 31 kategorií, takže si je sem všechny vypisovat nebudeme, ale pohrajeme si s touhle funkcí.

```
ghci>               ' '
ghci>               'A'
ghci>               'a'
ghci>               '.'
ghci>               '9'
ghci>                   " \t\nA9?|"
```

Vzhledem k tomu, že typ `GeneralCategory` je součástí typové třídy `Eq`, můžeme také testovat věci jako třeba `generalCategory c == Space`.

Funkce převede znak na verzálku. Mezery, čísla a podobné znaky zůstanou nezměněny.

Funkce převede znak na minusku.

Funkce převede znak na titulkovou velikost. Pro většinu znaků odpovídá titulková velikost verzálce.

Funkce převede znak na typ `Int`. Aby byl převod úspěšný, převáděný znak musí být v rozsazích `'0'..'9'`, `'a'..'f'` nebo `'A'..'F'`.

```
ghci>               "34538"
 3 4 5 3 8
ghci>               "FF85AB"
 15 15 8 5 10 11
```

Opačná funkce k funkci `digitToInt` je . Vezme hodnotu typu `Int` v rozsahu `0..15` a převede ho na číslici nebo minusku.

```
ghci>            15
'f'
ghci>            5
'5'
```

Funkce a `chr` převedou znak na jeho odpovídající číselnou hodnotu a naopak:

```
ghci>     'a'
97
ghci>     97
'a'
ghci>         "abcdefgh"
 97 98 99 100 101 102 103 104
```

Rozdíl mezi hodnotami `ord` dvou znaků se rovná jejich vzdálenosti v tabulce Unicode.

Caesarova šifra je primitivní metoda pro zakódování zpráv, která posouvá každý znak o stanovený počet pozic v abecedě. Můžeme si sami jednoduše vytvořit obměnu Ceasarovy šifry, jenom se nebudeme omezovat na abecedu.

```
       ::     ->        ->
                =
    let      =
               =
    in
```

Zde nejprve převedeme řetězec na seznam čísel. Poté přidáme posunutí každého čísla před převedením seznamu čísel zpátky na znaky. Jestliže jste skládací kovbojové, mohli byste zapsat tělo funkce jako `map (chr . (+ shift) . ord) msg`. Zkusme zkusit zakódovat několik zpráv.

```
ghci>        3 "Heeeeej"
"Khhhhhm"
ghci>        4 "Heeeeej"
"Liiiiin"
ghci>        1 "abcd"
"bcde"
ghci>        5 "Veselé Vánoce! Ho, ho, ho!"
"[jxjqi%[æsthj&%Mt1%mt1%mt&"
```

Zakódovalo se to bez problémů. Rozkódování zprávy je v podstatě pouhé posunutí o stejný počet míst jako se posouvalo při zakódování, jenom na druhou stranu.

```
       ::     ->        ->
                =
```

```
ghci>        3 "Jsem čajová konvička."
"Mvhp#Ðdmryä#nrqylÐnd1"
ghci>        3 "Mvhp#Ðdmryä#nrqylÐnd1"
"Jsem čajová konvička."
ghci>        5 .        5 $ "Tohle je věta."
"Tohle je věta."
```

Data.Map

Asociační seznamy (taktéž nazývané slovníky) jsou seznamy používané pro ukládání dvojic klíč-hodnota, u kterých nezáleží na pořadí. Kupříkladu můžeme použít asociační seznam na ukládání telefonních čísel, kde by telefonní čísla byla hodnotami a

jména lidí by byla klíči. Nestaráme se o pořadí, v jakém jsou uloženy, stačí nám jenom získat správné telefonní číslo pro určitou osobu.

Nejzřejmější způsob znázornění asociačních seznamů v Haskellu by bylo mít seznam dvojic. První složka seznamu by byla klíč, druhá složka hodnota. Tady máme příklad asociačního seznamu s telefonními čísly:

```
    =
"betty" "555-2938"
"bonnie" "452-2928"
"patsy" "493-2928"
"lucille" "205-2928"
"wendy" "939-8282"
"penny" "853-2492"
```

I přes zdánlivě podivné odsazení je tohle seznam dvojic řetězců. Nejběžnější úloha pro práci s asociačními seznamy je vyhledání nějaké hodnoty pomocí klíče. Napišme si funkci, která najde hodnotu podle jejího klíče.

```
::        =>   ->         ->
      =     .      .               ->     ==    $
```

Celkem jednoduché. Tahle funkce vezme klíč a seznam, profiltruje seznam, takže zůstanou pouze odpovídající klíče, vybere z nich první dvojici a vrátí danou hodnotu. Ale co se stane když klíč který hledáme není přítomen v asociačním seznamu? Hmm. V tomhle případě skončíme u pokusu získat první prvek z prázdného seznamu, což vyhodí běhovou chybu. Rozhodně bychom se měli vyhnout vytváření tak lehce havarovatelných programů, takže zkusme použít datový typ `Maybe`. Jestliže nenalezneme klíč, vrátíme hodnotu `Nothing`. Pokud ho ale nalezneme, vrátíme `Just něco`, kde něco je hodnota odpovídající klíči.

```
::        =>   ->         ->
      =
            = if     ==
                  then
                  else
```

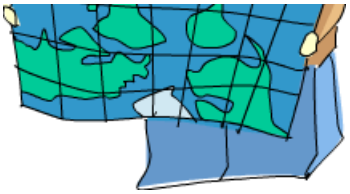
Podívejte se na typ funkce. Ta vezme klíč jež se dá porovnat, k němu nějaký asociační seznam a možná vyprodukuje nějakou hodnotu. Zní to celkem dobře.

Tohle je učebnicový příklad rekurzivní funkce pracující se seznamem. Okrajový případ, rozdělení seznamu na první prvek a zbytek, rekurzivní volání, je to tam všechno. Tohle je klasické skládání, takže se pojďme podívat, jak by se to dalo přepsat pomocí foldu.

```
::        =>   ->         ->
   =                  -> if     ==   then       else
```

Poznámka: je obvykle lepší použít foldy pro tuhle obyčejnou seznamovou rekurzi namísto explicitního rekurze, protože se jednodušeji čtou a rozeznávají. Každý ví, že se jedná o skládání, když vidí volání funkce `foldr`, ale k rozpoznání explicitní rekurze je potřeba více přemýšlení.

```
ghci>        "penny"
     "853-2492"
ghci>        "betty"
     "555-2938"
ghci>        "wilma"
```



Funguje to jedna radost! Jestliže máme v našem seznamu telefonní číslo holky, vybereme právě to číslo, jinak nic nevrátíme.

Právě jsme si vlastně napsali funkci `lookup` z modulu `Data.List`. Jestliže chceme najít odpovídající hodnotu ke klíči, musíme projít všechny prvky ze seznamu než na něj narazíme. Modul `Data.Map` nabízí asociační seznamy, jež jsou mnohem rychlejší (protože jsou vnitřně implementovány pomocí stromů), a také poskytuje velké množství užitečných funkcí. Odteď budeme říkat, že pracujeme s mapami místo s asociačními seznamy.

Protože modul `Data.Map` exportuje funkce, které by kolidovaly s těmi z modulů `Prelude` a `Data.List`, vyřešíme to kvalifikovaným importem.

```
import qualified          as
```

Vložte tuhle deklaraci importu do skriptu a poté ho načtěte do GHCi.

Pokročíme dál a podíváme se, co pro nás modul `Data.Map` může uložit! Tady je základní přehled jeho funkcí.

Funkce vezme asociační seznam (ve formě seznamu dvojic) a vrátí mapu těchto asociací.

```
ghci>                "betty" "555-2938"   "bonnie" "452-2928"   "lucille" "205-2928"
          "betty" "555-2938"   "bonnie" "452-2928"   "lucille" "205-2928"
ghci>                1 2   3 4   3 2   5 5
          1 2   3 2   5 5
```

Pokud existují duplikátní klíče v originálním asociačním seznamu, jsou zahozeny. Takhle vypadá typ funkce `fromList`:

```
              ::          =>          ->
```

Říká, že vezme nějaký seznam dvojic typů `k` a `v` a vrátí mapu která zobrazí klíče typu `k` na hodnoty typu `v`. Všimněte si, že když jsme vyráběli asociační seznamy z běžných seznamů, klíče musely být porovnatelné (jejich typ patřil do typové třídy `Eq`), ale teď musí být uspořádatelné. To je podstatné omezení modulu `Data.Map`. Potřebuje, aby klíče byly uspořádatelné, a mohl je tak uspořádat do stromu.

Měli byste vždycky použít modul `Data.Map` pro asociační seznamy, kromě případu, kdy byste měli klíče, které nejsou součástí typové třídy `Ord`.

Nulární funkce reprezentuje prázdnou mapu. Nepožaduje žádné argumenty, pouze vrátí prázdnou mapu.

```
ghci>
```

Funkce vezme klíč, hodnotu a mapu a vrátí novou mapu, která je stejná jako ta stará, jenom má v sobě navíc vložený klíč s hodnotou.

```
ghci>
ghci>           3 100
          3 100
ghci>           5 600           4 200           3 100
          3 100   4 200   5 600
ghci>           5 600 .           4 200 .           3 100 $
```

```
            3 100   4 200   5 600
```

Můžeme si napsat svou vlastní funkci `fromList` za použití prázdné mapy, funkce `insert` a foldu. Sledujte:

```
          ::          =>          ->
          =                    ->
```

Je to celkem nekomplikovaný fold. Začínáme s prázdnou mapou, kterou skládáme zprava a průběžně vkládáme dvojice klíčů a hodnot do akumulátoru.

Funkce ověří, jestli je mapa prázdná.

```
ghci>
ghci>          $               2 3   5 5
```

Funkce nahlásí velikost mapy.

```
ghci>
0
ghci>          $               2 4   3 3   4 2   5 4   6 4
5
```

Funkce vezme klíč a hodnotu a vytvoří mapu o velikosti jedna.

```
ghci>              3 9
          3 9
ghci>           5 9 $              3 9
          3 9   5 9
```

Funkce funguje podobně jako ta z modulu `Data.List`, jenom pracuje s mapami. Vrátí `Just něco`, pokud nalezne pro něco klíč, a `Nothing`, pokud ne.

Funkce je predikát, který vezme klíč a mapu a informuje, zda-li se klíč v mapě vyskytuje nebo ne.

```
ghci>           3 $               3 6   4 3   6 9
ghci>           3 $               2 5   4 5
```

Funkce a fungují skoro stejně jako jejich seznamové protějšky.

```
ghci>          100  $              1 1   2 4   3 9
          1 100   2 400   3 900
ghci>                  $              1 'a'   2 'A'   3 'b'   4 'B'
          2 'A'   4 'B'
```

Funkce je opakem funkce `fromList`.

```
ghci>           .           9 2 $              4 3
  4 3   9 2
```

Funkce a vrátí seznam klíčů, respektive hodnot. Funkce `keys` dělá totéž co `map fst . Map.toList` a funkce `elems` totéž co `map snd . Map.toList`.

Funkce je senzační funkcička. Chová se stejně jako funkce `fromList`, jenom nezahazuje duplicitní klíče, ale využije zadanou funkci pro rozhodnutí, jak s nimi naložit. Řekněme že holky mohou mít několik telefonních čísel a my máme zadán následující asociační seznam.

```
    =
"betty" "555-2938"
"betty" "342-2492"
"bonnie" "452-2928"
"patsy" "493-2928"
"patsy" "943-2929"
"patsy" "827-9162"
"lucille" "205-2928"
"wendy" "939-8282"
"penny" "853-2492"
"penny" "555-2111"
```

Když bychom teď na vytvoření mapy použili funkci `fromList`, ztratili bychom několik čísel! Takže uděláme tohle:

```
          ::         =>              ->
            =                                ->         ++ ", " ++
```

```
ghci>           "patsy" $
"827-9162, 943-2929, 493-2928"
ghci>           "wendy" $
"939-8282"
ghci>           "betty" $
"342-2492, 555-2938"
```

Jestliže je nalezen duplicitní klíč, naše funkce je použita pro sloučení hodnot těchto klíčů do jiné hodnoty. Můžeme taktéž nejprve všechny hodnoty přetvořit na jednoprvkový seznam a poté použít operátor ++ na přidávání dalších čísel.

```
          ::         =>          ->
            =                    $            ->
```

```
ghci>           "patsy" $
 "827-9162" "943-2929" "493-2928"
```

Nádhera! Další možné použití je v případě, kdy vytváříme mapu z asociačního seznamu čísel, u kterého si chceme uchovat pouze největší z hodnot, jakmile je nalezen duplicitní klíč.

```
ghci>                   2 3   2 5   2 100   3 29   3 22   3 11   4 22   4 15
          2 100   3 29   4 22
```

Nebo bychom si hodnoty odpovídajících klíčů mohli přát sečíst.

```
ghci>                   2 3   2 5   2 100   3 29   3 22   3 11   4 22   4 15
          2 108   3 62   4 37
```

Funkce je ve stejném vztahu s funkcí `insert`, jako je funkce `fromListWith` s funkcí `fromList`. Vloží dvojici klíč-hodnota do mapy, ale pokud tato mapa již obsahuje daný klíč, zadaná funkce stanoví, co se má provést.

```
ghci>                 3 100 $              3 4   5 103   6 339
```

```
3 104   5 103   6 339
```

Tohle je pouze několik funkcí z modulu `Data.Map`. V dokumentaci můžete nalézt jejich kompletní seznam.

Data.Set

Modul `Data.Set` nám poskytuje množiny podobající se těm matematickým. Množiny jsou něco mezi seznamy a mapami. Všechny prvky v množině jsou unikátní. A protože jsou interně implementovány pomocí stromů (podobně jako mapy z modulu `Data.Map`), jsou taktéž seřazeny. Zjišťování příslušnosti, vkládání, mazání atd. je mnohem rychlejší než provádění stejných akcí se seznamy. Nejobvyklejšími operacemi na množinách jsou vkládání do množiny, zjišťování příslušnosti a převádění množiny na seznam.

Protože názvy funkcí z modulu `Data.Set` hodně kolidují s názvy v modulech `Prelude` a `Data.List`, musíme provést kvalifikovaný import.

Vložte tento příkaz do skriptu:

```
import qualified          as
```

… a poté tento skript načtěte přes GHCi.

Řekněme že máme dva různé texty. Chceme zjistit, jaké znaky jsou použity v obou dvou.

```
      = "Měl jsem zrovna anime sen. Anime... Skutečnost... Jak moc se liší?"
      = "Stařík převrhl popelnici a teď jsou odpadky rozházené po celém mém trávníku!"
```

Funkce funguje zhruba tak jak byste čekali. Vezme libovolný seznam a převede ho na množinu.

```
ghci> let      =
ghci> let      =
ghci>
        " .?AJMSaceijklmnorstuvzíčěš"
ghci>
        " !Sacdehijklmnoprstuvyzáéídř"
```

Jak můžete vidět, položky jsou seřazeny a každý prvek je unikátní. A teď se použijeme funkci , abychom se podívali, které prvky mají oba texty společné.

```
ghci>
        " Saceijklmnorstuvzí"
```

Můžeme také využít funkci na zjištění, která písmena jsou obsažena v první množině, ale ne ve druhé, a naopak.

```
ghci>
        ".?AJMčěš"
ghci>
        "!dhpyáédř"
```

Nebo si můžeme nechat vypsat všechny unikátní znaky v obou řetězcích pomocí funkce .

```
ghci>
```

```
" !.?AJMSacdehijklmnoprstuvyzáéíčďěřš"
```

Funkce , , , , , a fungují přesně tak jak byste čekali.

```
ghci>
ghci>          $              3 4 5 5 4 3
ghci>          $              3 4 5 3 4 5
3
ghci>              9
9
ghci>            4 $              9 3 8 1
1 3 4 8 9
ghci>            8 $              5  10
5 6 7 8 9 10
ghci>            4 $              3 4 5 4 3 4 5
3 5
```

Také můžeme zjišťovat, zda-li je určitá množina podmnožinou či vlastní podmnožinou dané množiny. Množina A je podmnožinou množiny B, jestliže B obsahuje všechny prvky z množiny A. Množina A je vlastní podmnožina množiny B, jestliže B obsahuje všechny prvky z množiny A, ale v množině B se musí nacházet více prvků než v A.

```
ghci>              2 3 4                          1 2 3 4 5
ghci>              1 2 3 4 5                        1 2 3 4 5
ghci>              1 2 3 4 5                              1 2 3 4 5
ghci>              2 3 4 8                          1 2 3 4 5
```

Rovněž můžeme v množinách využívat funkce a .

```
ghci>               $              3 4 5 6 7 2 3 4
3 5 7
ghci>            1  $              3 4 5 6 7 2 3 4
3 4 5 6 7 8
```

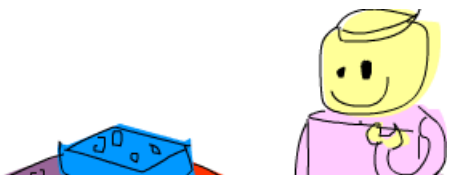
Množiny jsou často používány pro vyčištění seznamu od duplikátů, a to tak, že se seznam převede na množinu pomocí funkce `fromList`, která se následně převede zpět na seznam pomocí funkce . Funkce `nub` z modulu `Data.List` tohle umí také, ale vyřazení duplikátů z většího seznamů je mnohem rychlejši, když ho nacpeme do množiny a poté ho přeměníme zpátky na seznam, než abychom použili funkci `nub`. Jenže funkci `nub` stačí, aby prvky daného seznamu byly součástí typové třídy `Eq`, kdežto pokud chcete nacpat nějaké prvky do množiny, musí patřit do typové třídy `Ord`.

```
ghci> let          =            $
ghci>        "JAK TI DUPOU KRÁLÍCI?"
" ?ACDIJKLOPRTUAÍ"
ghci>     "JAK TI DUPOU KRÁLÍCI?"
"JAK TIDUPORÁLÍC?"
```

Naše funkce `setNub` je na větších seznamech obecně rychlejší než funkce `nub`, ale jak můžete vidět, tak funkce `nub` zachovává pořadí prvků v seznamu, zatímco funkce `setNub` ne.

Vytváření vlastních modulů

Zkoumali jsme zatím pár skvělých modulů, ale jak si vytvoříme vlastní? Téměř každý programovací jazyk nám umožňuje rozdělit náš kód do více souborů a v Haskellu tomu není jinak. Při psaní programů je dobrým zvykem vzít všechny funkce a data, jež slouží k



podobnému účelu, a vložit do modulu. Takto můžete jednoduše opětovně použít tyto funkce v jiných programech pouhým importováním daného modulu.

Podívejme se na to, jak bychom si mohli napsat naše vlastní moduly vytvořením malého modulu, který poskytuje funkce pro výpočet objemu a povrchu několika geometrických objektů. Začneme vytvořením souboru nazvaného `Geometry.hs`.

Řekněme, že modul *exportuje* funkce. Co to znamená je to, že když importuji modul, mohu používat funkce, které exportuje. Mohou v něm být definovány funkce volající jiné vnitřní funkce, avšak my uvidíme a budeme používat pouze funkce exportované modulem.

Na začátku modulu si stanovíme jeho název. Jestliže máme soubor s názvem `Geometry.hs`, měli bychom náš modul pojmenovat `Geometry`. Poté určíme funkce, které budou exportovány, a pak začneme psát funkce. Takže začneme tímhle.

```
module



   where
```

Jak můžete vidět, budeme počítat povrchy a objemy koulí, krychlí a kvádrů. Pustíme se tedy do toho a definujeme si naše funkce:

```
module



   where
            ::       ->
                  =  4.0 / 3.0  *    *         ^ 3

          ::       ->
                = 4 *    *         ^ 2

          ::       ->
               =

        ::       ->
             =

           ::       ->       ->       ->
                 =                  *

          ::       ->       ->       ->
               =                  * 2 +                  * 2 +                  * 2

            ::       ->       ->
                 =    *
```

Máme tu celkem standardní geometrii, i když je tu pár věcí k povšimnutí. Protože je krychle speciální případ kvádru, definovali jsme si jeho povrch a objem tím, že jsme s ní nakládali jako s kvádrem jehož strany mají stejnou délku. Taktéž jsme si definovali pomocnou funkci nazvanou `rectangleArea`, která vypočítá obsah obdelníku v závislosti na délce jeho stran. Je to poměrně triviální, protože to je pouhé násobení. Všimněte si, že jsme tuto funkci využili v definicích jiných funkcí (jmenovitě `cuboidArea` a `cuboidVolume`), ale neexportovali jsme ji! Protože chceme, aby náš modul poskytoval pouze funkce pro zpracovávání trojdimenzionálních objektů, použili jsme funkci `rectangleArea`, ale neexportujeme ji.

Když vytváříme modul, obvykle exportujeme pouze ty funkce, které se chovají trochu jako rozhraní k našemu modulu, takže je samotná implementace skrytá. Jestliže někdo hodlá použít náš modul `Geometry`, nebude se muset zabývat funkcemi jež neexportujeme. Můžeme se rozhodnout tyto funkce úplně změnit nebo je odstranit v novější verzi (mohli bychom odstranit funkci `rectangleArea` a místo ní použít operátor násobení) a nikomu by to nevadilo, protože je vůbec neposkytujeme.

Pro použití našeho modulu stačí napsat:

```
import
```

Přičemž soubor `Geometry.hs` musí být ve stejném adresáři jako je program, který ho importuje.

Moduly také mohou být strukturovány hierarchicky. Každý modul může mít několik podmodulů a ty mohou mít své vlastní podmoduly. Oddělíme si naše funkce, aby `Geometry` byl modul mající tři podmoduly, každý pro jiný typ objektů.

Nejprve si vytvoříme adresář nazvaný `Geometry`. Dávejte pozor na velké písmeno G. V tomto adresáři si vytvoříme tři soubory: `sphere.hs`, `cuboid.hs` a `cube.hs`. Tady je výpis toho, co tyto soubory mají obsahovat:

`sphere.hs`

```
module

  where

      ::       ->
            =  4.0 / 3.0  *    *         ^ 3

    ::       ->
          = 4 *    *         ^ 2
```

`cuboid.hs`

```
module

  where

      ::       ->       ->       ->
            =                   *

    ::       ->       ->       ->
          =                   * 2 +                   * 2 +                   * 2

             ::       ->       ->
                =    *
```

`cube.hs`

```
module

  where
import qualified                as

      ::       ->
           =

    ::       ->
         =
```

Prima! Takže první je modul `Geometry.Sphere`. Všimněte si, že je uložen v adresáři nazvaném `Geometry` a v modulu je pojmenován jako `Geometry.Sphere`. Totéž jsme udělali s kvádrem. Taktéž si všimněte, že jsme ve všech třech podmodulech definovali funkce se stejnými názvy. Můžeme si to dovolit, protože se jedná o oddělené moduly. Dále chceme použít v modulu `Geometry.Cube` funkce z modulu `Geometry.Cuboid`, ale nemůžeme toho docílit obyčejným příkazem `import Geometry.Cuboid`, protože by se exportovaly funkce se stejnými názvy jako jsou v `Geometry.Cube`. To je důvod, proč jsme použili kvalifikovaný import a všechno je v pořádku.

Tak když teď založíme soubor nacházející se ve stejné úrovni jako je adresář `Geometry`, můžeme napsat, řekněme:

```
import
```

Poté zavoláme funkce `area` a `volume` a ty nám vypočítají povrch a objem koule. Když bychom si chtěli pohrát se dvěma nebo více těmito moduly, musíme je importovat kvalifikovaně, protože exportují funkce se stejnými názvy. Takže napíšeme něco takového:

```
import qualified                as
import qualified                as
import qualified              as
```

Pak máme možnost zavolat funkce `Sphere.area`, `Sphere.volume`, `Cuboid.area` atd. a každá nám vypočítá povrch nebo objem odpovídajícího objektu.

Až budete příště vytvářet soubor, který je opravdu velký a obsahuje hodně funkcí, zkuste se podívat, které funkce slouží společnému účelu a zkuste je oddělit do vlastního modulu. Jakmile bude váš program potřebovat nějakou funkci z tohoto modulu, bude pak pouze stačit ho naimportovat.

Funkce vyššího řádu Obsah Vytváříme si své typy a typové třídy

← Moduly Obsah Input and Output →

Vytváříme si své typy a typové třídy

English version

Předchozí kapitoly se týkaly několika existujících haskellových typů a typových tříd. V této kapitole se naučíme vytvářet si naše vlastní a jak s nimi pracovat!

Tady překladatel prozatím skončil. Můžete navštívit IRC kanál #haskell.cz a povzbudit ho.

Úvod do algebraických datových typů

So far, we've run into a lot of data types. `Bool`, `Int`, `Char`, `Maybe`, etc. But how do we make our own? Well, one way is to use the **data** keyword to define a type. Let's see how the `Bool` type is defined in the standard library.

```
data      =       |
```

`data` means that we're defining a new data type. The part before the `=` denotes the type, which is `Bool`. The parts after the `=` are **value constructors**. They specify the different values that this type can have. The `|` is read as *or*. So we can read this as: the `Bool` type can have a value of `True` or `False`. Both the type name and the value constructors have to be capital cased.

In a similar fashion, we can think of the `Int` type as being defined like this:

```
data     =  2147483648 |  2147483647 |     |  1 | 0 | 1 | 2 |     | 2147483647
```

The first and last value constructors are the minimum and maximum possible values of `Int`. It's not actually defined like this, the ellipses are here because we omitted a heapload of numbers, so this is just for illustrative purposes.

Now, let's think about how we would represent a shape in Haskell. One way would be to use tuples. A circle could be denoted as `(43.1, 55.0, 10.4)` where the first and second fields are the coordinates of the circle's center and the third field is the radius. Sounds OK, but those could also represent a 3D vector or anything else. A better solution would be to make our own type to represent a shape. Let's say that a shape can be a circle or a rectangle. Here it is:

```
data      =                       |
```

Now what's this? Think of it like this. The `Circle` value constructor has three fields, which take floats. So when we write a value constructor, we can optionally add some types after it and those types define the values it will contain. Here, the first two fields are the coordinates of its center, the third one its radius. The `Rectangle` value constructor has four fields which accept floats. The first two are the coordinates to its upper left corner and the second two are coordinates to its lower right one.

Now when I say fields, I actually mean parameters. Value constructors are actually functions that ultimately return a value of a data type. Let's take a look at the type signatures for these two value constructors.

```
ghci>
        ::        ->        ->        ->
ghci>
          ::        ->        ->        ->        ->
```

Cool, so value constructors are functions like everything else. Who would have thought? Let's make a function that takes a shape and returns its surface.

```
         ::        ->
                    =     *    ^ 2
                          =      $     -     *      $     -
```

The first notable thing here is the type declaration. It says that the function takes a shape and returns a float. We couldn't write a type declaration of `Circle -> Float` because `Circle` is not a type, `Shape` is. Just like we can't write a function with a type declaration of `True -> Int`. The next thing we notice here is that we can pattern match against constructors. We pattern matched against constructors before (all the time actually) when we pattern matched against values like `[]` or `False` or `5`, only those values didn't have any fields. We just write a constructor and then bind its fields to names. Because we're interested in the radius, we don't actually care about the first two fields, which tell us where the circle is.

```
ghci>          $         10 20 10
314.15927
ghci>          $            0 0 100 100
10000.0
```

Yay, it works! But if we try to just print out `Circle 10 20 5` in the prompt, we'll get an error. That's because Haskell doesn't know how to display our data type as a string (yet). Remember, when we try to print a value out in the prompt, Haskell first runs the `show` function to get the string representation of our value and then it prints that out to the terminal. To make our `Shape` type part of the `Show` typeclass, we modify it like this:

```
data        =                          |                                  deriving
```

We won't concern ourselves with deriving too much for now. Let's just say that if we add `deriving (Show)` at the end of a *data* declaration, Haskell automagically makes that type part of the `Show` typeclass. So now, we can do this:

```
ghci>         10 20 5
        10.0 20.0 5.0
ghci>            50 230 60 90
           50.0 230.0 60.0 90.0
```

Value constructors are functions, so we can map them and partially apply them and everything. If we want a list of concentric circles with different radii, we can do this.

```
ghci>              10 20   4 5 6 6
         10.0 20.0 4.0         10.0 20.0 5.0         10.0 20.0 6.0         10.0 20.0 6.0
```

Our data type is good, although it could be better. Let's make an intermediate data type that defines a point in two-dimensional space. Then we can use that to make our shapes more understandable.

```
data        =                    deriving
```

```
data        =                    |                        deriving
```

Notice that when defining a point, we used the same name for the data type and the value constructor. This has no special meaning, although it's common to use the same name as the type if there's only one value constructor. So now the `Circle` has two fields, one is of type `Point` and the other of type `Float`. This makes it easier to understand what's what. Same goes for the rectangle. We have to adjust our `surface` function to reflect these changes.

```
          ::        ->
                =     *    ^ 2
                                                 =      $     -      *      $     -
```

The only thing we had to change were the patterns. We disregarded the whole point in the circle pattern. In the rectangle pattern, we just used a nested pattern matching to get the fields of the points. If we wanted to reference the points themselves for some reason, we could have used as-patterns.

```
ghci>                            0 0          100 100
10000.0
ghci>                         0 0   24
1809.5574
```

How about a function that nudges a shape? It takes a shape, the amount to move it on the x axis and the amount to move it on the y axis and then returns a new shape that has the same dimensions, only it's located somewhere else.

```
        ::        ->        ->        ->
                                      =
                                                        =
```

Pretty straightforward. We add the nudge amounts to the points that denote the position of the shape.

```
ghci>                       34 34   10   5 10
              39.0 44.0   10.0
```

If we don't want to deal directly with points, we can make some auxilliary functions that create %0000œof some size at the zero coordinates and then nudge those.

```
             ::        ->
               =                 0 0

           ::        ->        ->
                       =                    0 0


ghci>                   40 100   60 23
                  60.0 23.0          100.0 123.0
```

You can, of course, export your data types in your modules. To do that, just write your type along with the functions you are exporting and then add some parentheses and in them specify the value constructors that you want to export for it, separated by commas. If you want to export all the value constructors for a given type, just write `...`.

If we wanted to export the functions and types that we defined here in a module, we could start it off like this:

```
module
```

```
    where
```

By doing `Shape(..)`, we exported all the value constructors for `Shape`, so that means that whoever imports our module can make shapes by using the `Rectangle` and `Circle` value constructors. It's the same as writing `Shape (Rectangle, Circle)`.

We could also opt not to export any value constructors for `Shape` by just writing `Shape` in the export statement. That way, someone importing our module could only make shapes by using the auxilliary functions `baseCircle` and `baseRect`. `Data.Map` uses that approach. You can't create a map by doing `Map.Map [(1,2),(3,4)]` because it doesn't export that value constructor. However, you can make a mapping by using one of the auxilliary functions like `Map.fromList`. Remember, value constructors are just functions that take the fields as parameters and return a value of some type (like `Shape`) as a result. So when we choose not to export them, we just prevent the person importing our module from using those functions, but if some other functions that are exported return a type, we can use them to make values of our custom data types.

Not exporting the value constructors of a data types makes them more abstract in such a way that we hide their implementation. Also, whoever uses our module can't pattern match against the value constructors.

Záznamy

OK, we've been tasked with creating a data type that describes a person. The info that we want to store about that person is: first name, last name, age, height, phone number, and favorite ice-cream flavor. I don't know about you, but that's all I ever want to know about a person. Let's give it a go!

```
data        =                                          deriving
```

O-kay. The first field is the first name, the second is the last name, the third is the age and so on. Let's make a person.

```
ghci> let     =        "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci>
       "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

That's kind of cool, although slightly unreadable. What if we want to create a function to get seperate info from a person? A function that gets some person's first name, a function that gets some person's last name, etc. Well, we'd have to define them kind of like this.

```
          ::         ->
                                  =

         ::         ->
                                 =

    ::         ->
                           =

       ::         ->
                                 =

           ::         ->
                                 =

       ::         ->
                                 =
```

Whew! I certainly did not enjoy writing that! Despite being very cumbersome and BORING to write, this method works.

```
ghci> let     =        "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci>
"Buddy"
ghci>
184.2
ghci>
"Chocolate"
```

There must be a better way, you say! Well no, there isn't, sorry.

Just kidding, there is. Hahaha! The makers of Haskell were very smart and anticipated this scenario. They included an alternative way to write data types. Here's how we could achieve the above functionality with record syntax.

```
data        =                    ::
                                 ::
                            ::
                               ::
                                     ::
                               ::
                         deriving
```

So instead of just naming the field types one after another and separating them with spaces, we use curly brackets. First we write the name of the field, for instance, `firstName` and then we write a double colon `::` (also called Paamayim Nekudotayim, haha) and then we specify the type. The resulting data type is exactly the same. The main benefit of this is that it creates functions that lookup fields in the data type. By using record syntax to create this data type, Haskell automatically made these functions: `firstName`, `lastName`, `age`, `height`, `phoneNumber` and `flavor`.

```
ghci>
       ::        ->
ghci>
          ::        ->
```

There's another benefit to using record syntax. When we derive `Show` for the type, it displays it differently if we use record syntax to define and instantiate the type. Say we have a type that represents a car. We want to keep track of the company that made it, the model name and its year of production. Watch.

```
data     =                     deriving


ghci>     "Ford" "Mustang" 1967
    "Ford" "Mustang" 1967
```

If we define it using record syntax, we can make a new car like this.

```
data     =             ::              ::              ::      deriving


ghci>              "Ford"         "Mustang"       1967
            = "Ford"        = "Mustang"       = 1967
```

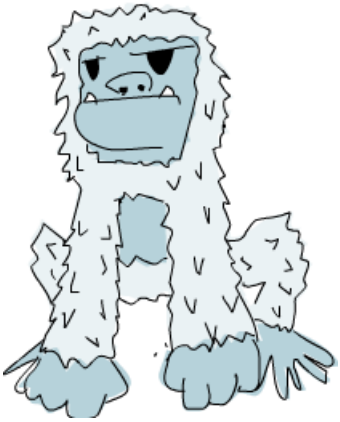
When making a new car, we don't have to necessarily put the fields in the proper order, as long as we list all of them. But if we don't use record syntax, we have to specify them in order.

Use record syntax when a constructor has several fields and it's not obvious which field is which. If we make a 3D vector data type by doing `data Vector = Vector Int Int Int`, it's pretty obvious that the fields are the components of a vector. However, in our `Person` and `Car` types, it wasn't so obvious and we greatly benefited from using record syntax.

Typové parametry

A value constructor can take some values parameters and then produce a new value. For instance, the `Car` constructor takes three values and produces a car value. In a similar manner, **type constructors** can take types as parameters to produce new types. This might sound a bit too meta at first, but it's not that complicated. If you're familiar with templates in C++, you'll see some parallels. To get a clear picture of what type parameters work like in action, let's take a look at how a type we've already met is implemented.

```
data          =          |
```



The `a` here is the type parameter. And because there's a type parameter involved, we call `Maybe` a type constructor. Depending on what we want this data type to hold when it's not `Nothing`, this type constructor can end up producing a type of `Maybe Int`, `Maybe Car`, `Maybe String`, etc. No value can have a type of just `Maybe`, because that's not a type per se, it's a type constructor. In order for this to be a real type that a value can be part of, it has to have all its type parameters filled up.

So if we pass `Char` as the type parameter to `Maybe`, we get a type of `Maybe Char`. The value `Just 'a'` has a type of `Maybe Char`, for example.

You might not know it, but we used a type that has a type parameter before we used `Maybe`. That type is the list type. Although there's some syntactic sugar in play, the list type takes a parameter to produce a concrete type. Values can have an `[Int]` type, a `[Char]` type, a `[[String]]` type, but you can't have a value that just has a type of `[]`.

Let's play around with the `Maybe` type.

```
ghci>       "Haha"
      "Haha"
ghci>       84
      84
ghci>           "Haha"
      "Haha" ::
ghci>           84
      84 ::           =>
ghci>
          ::
ghci>       10 ::
      10.0
```

Type parameters are useful because we can make different types with them depending on what kind of types we want contained in our data type. When we do `:t Just "Haha"`, the type inference engine figures it out to be of the type `Maybe [Char]`, because if the `a` in the `Just a` is a string, then the `a` in `Maybe a` must also be a string.

Notice that the type of `Nothing` is `Maybe a`. Its type is polymorphic. If some function requires a `Maybe Int` as a parameter, we can give it a `Nothing`, because a `Nothing` doesn't contain a value anyway and so it doesn't matter. The `Maybe a` type can act like a `Maybe Int` if it has to, just like `5` can act like an `Int` or a `Double`. Similarly, the type of the empty list is `[a]`. An empty list can act like a list of anything. That's why we can do `[1,2,3] ++ []` and `["ha","ha","ha"] ++ []`.

Using type parameters is very beneficial, but only when using them makes sense. Usually we use them when our data type would work regardless of the type of the value it then holds inside it, like with our `Maybe a` type. If our type acts as some kind of box, it's good to use them. We could change our `Car` data type from this:

```
data      =                ::
                         ::
                       ::
```

```
                deriving
```

To this:

```
data           =               ::
                             ::
                           ::
                      deriving
```

But would we really benefit? The answer is: probably no, because we'd just end up defining functions that only work on the `Car String String Int` type. For instance, given our first definition of `Car`, we could make a function that displays the car's properties in a nice little text.

```
        ::      ->
                    =          =          =      = "This " ++    ++ " " ++    ++ " was made in " +
+
```

```
ghci> let       =              "Ford"         "Mustang"       1967
ghci>
"This Ford Mustang was made in 1967"
```

A cute little function! The type declaration is cute and it works nicely. Now what if `Car` was `Car a b c`?

```
        ::          =>                  ->
                      =          =          =      = "This " ++    ++ " " ++    ++ " was made in " +
+
```

We'd have to force this function to take a `Car` type of `(Show a) => Car String String a`. You can see that the type signature is more complicated and the only benefit we'd actually get would be that we can use any type that's an instance of the `Show` typeclass as the type for `c`.

```
ghci>              "Ford" "Mustang" 1967
"This Ford Mustang was made in 1967"
ghci>              "Ford" "Mustang" "nineteen sixty seven"
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci>         "Ford" "Mustang" 1967
    "Ford" "Mustang" 1967 ::          =>
ghci>         "Ford" "Mustang" "nineteen sixty seven"
    "Ford" "Mustang" "nineteen sixty seven" ::
```

In real life though, we'd end up using `Car String String Int` most of the time and so it would seem that parameterizing the `Car` type isn't really worth it. We usually use type parameters when the type that's contained inside the data type's various value constructors isn't really that important for the type to work. A list of stuff is a list of stuff and it doesn't matter what the type of that stuff is, it can still work. If we want to sum a list of numbers, we can specify later in the summing function that we specifically want a list of numbers. Same goes for `Maybe`. `Maybe` represents an option of either having nothing or having one of something. It doesn't matter what the type of that something is.

Another example of a parameterized type that we've already met is `Map k v` from `Data.Map`. The `k` is the type of the keys in a map and the `v` is the type of the values. This is a good example of where type parameters are very useful. Having maps parameterized enables us to have mappings from any type to any other type, as long as the type of the key is part of the `Ord` typeclass. If we were defining a mapping type, we could add a typeclass constraint in the *data* declaration:

```
data          =>          =
```

However, it's a very strong convention in Haskell to **never add typeclass constraints in data declarations.** Why? Well, because we don't benefit a lot, but we end up writing more class constraints, even when we don't need them. If we put or don't put the `Ord k` constraint in the *data* declaration for `Map k v`, we're going to have to put the constraint into functions that assume the keys in a map can be ordered. But if we don't put the constraint in the data declaration, we don't have to put `(Ord k) =>` in the type declarations of functions that don't care whether the keys can be ordered or not. An example of such a function is `toList`, that just takes a mapping and converts it to an associative list. Its type signature is `toList :: Map k a -> [(k, a)]`. If `Map k v` had a type constraint in its *data* declaration, the type for `toList` would have to be `toList :: (Ord k) => Map k a -> [(k, a)]`, even though the function doesn't do any comparing of keys by order.

So don't put type constraints into *data* declarations even if it seems to make sense, because you'll have to put them into the function type declarations either way.

Let's implement a 3D vector type and add some operations for it. We'll be using a parameterized type because even though it will usually contain numeric types, it will still support several of them.

```
data          =              deriving

      ::          =>          ->          ->
            `vplus`                 =

         ::          =>          ->    ->
               `vectMult`    =

            ::          =>          ->          ->
                 `scalarMult`                =      +      +
```

`vplus` is for adding two vectors together. Two vectors are added just by adding their corresponding components. `scalarMult` is for the scalar product of two vectors and `vectMult` is for multiplying a vector with a scalar. These functions can operate on types of `Vector Int`, `Vector Integer`, `Vector Float`, whatever, as long as the `a` from `Vector a` is from the `Num` typeclass. Also, if you examine the type declaration for these functions, you'll see that they can operate only on vectors of the same type and the numbers involved must also be of the type that is contained in the vectors. Notice that we didn't put a `Num` class constraint in the *data* declaration, because we'd have to repeat it in the functions anyway.

Once again, it's very important to distinguish between the type constructor and the value constructor. When declaring a data type, the part before the `=` is the type constructor and the constructors after it (possibly separated by `|`'s) are value constructors. Giving a function a type of `Vector t t t -> Vector t t t -> t` would be wrong, because we have to put types in type declaration and the vector **type** constructor takes only one parameter, whereas the value constructor takes three. Let's play around with our vectors.

```
ghci>        3 5 8 `vplus`        9 2 8
       12 7 16
ghci>        3 5 8 `vplus`        9 2 8 `vplus`        0 2 3
       12 9 19
ghci>        3 9 7 `vectMult` 10
       30 90 70
ghci>        4 9 5 `scalarMult`        9.0 2.0 4.0
74.0
ghci>        2 9 3 `vectMult`         4 9 5 `scalarMult`        9 2 4
       148 666 222
```

Odvozené instance

In the Typeclasses 101 section, we explained the basics of typeclasses. We explained that a typeclass is a sort of an interface that defines some behavior. A type can be made an **instance** of a typeclass if it supports that behavior. Example: the `Int` type is an instance of the `Eq` typeclass because the `Eq` typeclass defines behavior for stuff that can be equated. And because integers can be equated, `Int` is a part of the

`Eq` typeclass. The real usefulness comes with the functions that act as the interface for `Eq`, namely `==` and `/=`. If a type is a part of the `Eq` typeclass, we can use the `==` functions with values of that type. That's why expressions like `4 == 4` and `"foo" /= "bar"` typecheck.

We also mentioned that they're often confused with classes in languages like Java, Python, C++ and the like, which then baffles a lot of people. In those languages, classes are a blueprint from which we then create objects that contain state and can do some actions. Typeclasses are more like interfaces. We don't make data from typeclasses. Instead, we first make our data type and then we think about what it can act like. If it can act like something that can be equated, we make it an instance of the `Eq` typeclass. If it can act like something that can be ordered, we make it an instance of the `Ord` typeclass.

In the next section, we'll take a look at how we can manually make our types instances of typeclasses by implementing the functions defined by the typeclasses. But right now, let's see how Haskell can automatically make our type an instance of any of the following typeclasses: `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`. Haskell can derive the behavior of our types in these contexts if we use the *deriving* keyword when making our data type.

Consider this data type:

```
data        =                    ::
                                 ::
                        ::
```

It describes a person. Let's assume that no two people have the same combination of first name, last name and age. Now, if we have records for two people, does it make sense to see if they represent the same person? Sure it does. We can try to equate them and see if they're equal or not. That's why it would make sense for this type to be part of the `Eq` typeclass. We'll derive the instance.

```
data        =                    ::
                                 ::
                          ::
                    deriving
```

When we derive the `Eq` instance for a type and then try to compare two values of that type with `==` or `/=`, Haskell will see if the value constructors match (there's only one value constructor here though) and then it will check if all the data contained inside matches by testing each pair of fields with `==`. There's only one catch though, the types of all the fields also have to be part of the `Eq` typeclass. But since both `String` and `Int` are, we're OK. Let's test our `Eq` instance.

```
ghci> let       =                  = "Michael"          = "Diamond"      = 43
ghci> let        =                  = "Adam"           = "Horovitz"      = 41
ghci> let     =                   = "Adam"           = "Yauch"      = 44
ghci>     ==

ghci>       ==

ghci>       ==

ghci>       ==                   = "Michael"          = "Diamond"      = 43
```

Of course, since `Person` is now in `Eq`, we can use it as the `a` for all functions that have a class constraint of `Eq a` in their type signature, such as `elem`.

```
ghci> let             =
ghci>       `elem`
```

The `Show` and `Read` typeclasses are for things that can be converted to or from strings, respectively. Like with `Eq`, if a type's constructors have fields, their type has to be a part of `Show` or `Read` if we want to make our type an instance of them. Let's make our `Person` data type a part of `Show` and `Read` as well.

```
data         =                    ::
                                  ::
                          ::
                     deriving
```

Now we can print a person out to the terminal.

```
ghci> let        =                   = "Michael"            = "Diamond"       = 43
ghci>
                  = "Michael"            = "Diamond"       = 43
ghci> "mikeD is: " ++
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

Had we tried to print a person on the terminal before making the `Person` data type part of `Show`, Haskell would have complained at us, claiming it doesn't know how to represent a person as a string. But now that we've derived a `Show` instance for it, it does know.

`Read` is pretty much the inverse typeclass of `Show`. `Show` is for converting values of our a type to a string, `Read` is for converting strings to values of our type. Remember though, when we use the `read` function, we have to use an explicit type annotation to tell Haskell which type we want to get as a result. If we don't make the type we want as a result explicit, Haskell doesn't know which type we want.

```
ghci>       "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" ::
                  = "Michael"            = "Diamond"       = 43
```

If we use the result of our `read` later on in a way that Haskell can infer that it should read it as a person, we don't have to use type annotation.

```
ghci>       "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" ==
```

We can also read parameterized types, but we have to fill in the type parameters. So we can't do `read "Just 't'" :: Maybe a`, but we can do `read "Just 't'" :: Maybe Char`.

Derived instances of `Ord` work like expected. First the constructors are compared lexicographically and if the values of two constructors are the same, their fields are compared, provided that the types of the fields are also instances of `Ord`. The `Bool` type can have a value of either `False` or `True`. For the purpose of seeing how it behaves when compared, we can think of it as being implemented like this:

```
data      =       |       deriving
```

Because the `False` value constructor is specified first and the `True` value constructor is specified after it, we can consider `True` as greater than `False`.

```
ghci>       `compare`
ghci>       >
```

```
ghci>      <
```

In the `Maybe a` data type, the `Nothing` value constructor is specified before the `Just` value constructor, so a value of `Nothing` is always smaller than a value of `Just something`, even if that something is minus one billion trillion. But if we compare two `Just` values, then it goes to compare what's inside them.

```
ghci>         <      100
ghci>         >        49999
ghci>      3 `compare`      2
ghci>      100 >      50
```

But we can't do something like `Just (*3) > Just (*2)`, because `(*3)` and `(*2)` are functions, which aren't instances of `Ord`.

We can easily use algebraic data types to make enumerations and the `Enum` and `Bounded` typeclasses help us with that. Consider the following data type:

```
data     =        |         |          |          |        |          |
```

Because all the value constructors are nullary (take no parameters, i.e. fields), we can make it part of the `Enum` typeclass. The `Enum` typeclass is for things that have predecessors and successors. We can also make it part of the `Bounded` typeclass, which is for things that have a lowest possible value and highest possible value. And while we're at it, let's also make it an instance of all the other derivable typeclasses and see what we can do with it.

```
data     =        |         |          |          |        |          |
           deriving
```

Because it's part of the `Show` and `Read` typeclasses, we can convert values of this type to and from strings.

```
ghci>
ghci>
"Wednesday"
ghci>      "Saturday" ::
```

Because it's part of the `Eq` and `Ord` typeclasses, we can compare or equate days.

```
ghci>         ==
ghci>         ==
ghci>         >
ghci>       `compare`
```

It's also part of `Bounded`, so we can get the lowest and highest day.

```
ghci>         ::
ghci>         ::
```

It's also an instance of `Enum`. We can get predecessors and successors of days and we can make list ranges from them!

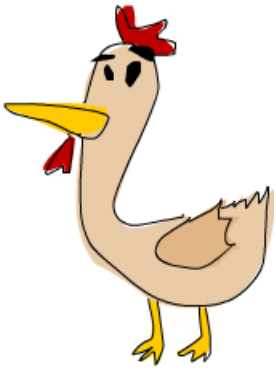
```
ghci>
ghci>
ghci>
ghci>                       ::
```

That's pretty awesome.

Typová synonyma

Previously, we mentioned that when writing types, the `[Char]` and `String` types are equivalent and interchangeable. That's implemented with **type synonyms**. Type synonyms don't really do anything per se, they're just about giving some types different names so that they make more sense to someone reading our code and documentation. Here's how the standard library defines `String` as a synonym for `[Char]`.

```
type        =
```



We've introduced the *type* keyword. The keyword might be misleading to some, because we're not actually making anything new (we did that with the *data* keyword), but we're just making a synonym for an already existing type.

If we make a function that converts a string to uppercase and call it `toUpperString` or something, we can give it a type declaration of `toUpperString :: [Char] -> [Char]` or `toUpperString :: String -> String`. Both of these are essentially the same, only the latter is nicer to read.

When we were dealing with the `Data.Map` module, we first represented a phonebook with an association list before converting it into a map. As we've already found out, an association list is a list of key-value pairs. Let's look at a phonebook that we had.

```
          ::
          =
    "betty" "555-2938"
    "bonnie" "452-2928"
    "patsy" "493-2928"
    "lucille" "205-2928"
    "wendy" "939-8282"
    "penny" "853-2492"
```

We see that the type of `phoneBook` is `[(String,String)]`. That tells us that it's an association list that maps from strings to strings, but not much else. Let's make a type synonym to convey some more information in the type declaration.

```
type            =
```

Now the type declaration for our phonebook can be `phoneBook :: PhoneBook`. Let's make a type synonym for `String` as well.

```
type              =
type       =
type            =
```

Giving the `String` type synonyms is something that Haskell programmers do when they want to convey more information about what strings in their functions should be used as and what they represent.

So now, when we implement a function that takes a name and a number and sees if that name and number combination is in our phonebook, we can give it a very pretty and descriptive type declaration.

```
::       ->              ->            ->
                 =                  `elem`
```

If we decided not to use type synonyms, our function would have a type of `String -> String -> [(String,String)] -> Bool`. In this case, the type declaration that took advantage of type synonyms is easier to understand. However, you shouldn't go overboard with them. We introduce type synonyms either to describe what some existing type represents in our functions (and thus our type declarations become better documentation) or when something has a long-ish type that's repeated a lot (like `[(String,String)]`) but represents something more specific in the context of our functions.

Type synonyms can also be parameterized. If we want a type that represents an association list type but still want it to be general so it can use any type as the keys and values, we can do this:

```
type                =
```

Now, a function that gets the value by a key in an association list can have a type of `(Eq k) => k -> AssocList k v -> Maybe v`. `AssocList` is a type constructor that takes two types and produces a concrete type, like `AssocList Int String`, for instance.

> **Fonzie says:** Aaay! When I talk about *concrete types* I mean like fully applied types like `Map Int String` or if we're dealin' with one of them polymorphic functions, `[a]` or `(Ord a) => Maybe a` and stuff. And like, sometimes me and the boys say that `Maybe` is a type, but we don't mean that, cause every idiot knows `Maybe` is a type constructor. When I apply an extra type to `Maybe`, like `Maybe String`, then I have a concrete type. You know, values can only have types that are concrete types! So in conclusion, live fast, love hard and don't let anybody else use your comb!

Just like we can partially apply functions to get new functions, we can partially apply type parameters and get new type constructors from them. Just like we call a function with too few parameters to get back a new function, we can specify a type constructor with too few type parameters and get back a partially applied type constructor. If we wanted a type that represents a map (from `Data.Map`) from integers to something, we could either do this:

```
type          =
```

Or we could do it like this:

```
type        =
```

Either way, the `IntMap` type constructor takes one parameter and that is the type of what the integers will point to.

> **Oh yeah**. If you're going to try and implement this, you'll probably going to do a qualified import of `Data.Map`. When you do a qualified import, type constructors also have to be preceeded with a module name. So you'd write `type IntMap = Map.Map Int`.

Make sure that you really understand the distinction between type constructors and value constructors. Just because we made a type synonym called `IntMap` or `AssocList` doesn't mean that we can do stuff like `AssocList [(1,2),(4,5),(7,9)]`. All it

means is that we can refer to its type by using different names. We can do `[(1,2),(3,5),(8,9)] :: AssocList Int Int`, which will make the numbers inside assume a type of `Int`, but we can still use that list as we would any normal list that has pairs of integers inside. Type synonyms (and types generally) can only be used in the type portion of Haskell. We're in Haskell's type portion whenever we're defining new types (so in *data* and *type* declarations) or when we're located after a `::`. The `::` is in type declarations or in type annotations.

Another cool data type that takes two types as its parameters is the `Either a b` type. This is roughly how it's defined:

```
data            =         |         deriving
```

It has two value constructors. If the `Left` is used, then its contents are of type `a` and if `Right` is used, then its contents are of type `b`. So we can use this type to encapsulate a value of one type or another and then when we get a value of type `Either a b`, we usually pattern match on both `Left` and `Right` and we different stuff based on which one of them it was.

```
ghci>       20
      20
ghci>      "w00t"
     "w00t"
ghci>           'a'
      'a' ::
ghci>
          ::
```

So far, we've seen that `Maybe a` was mostly used to represent the results of computations that could have either failed or not. But somtimes, `Maybe a` isn't good enough because `Nothing` doesn't really convey much information other than that something has failed. That's cool for functions that can fail in only one way or if we're just not interested in how and why they failed. A `Data.Map` lookup fails only if the key we were looking for wasn't in the map, so we know exactly what happened. However, when we're interested in how some function failed or why, we usually use the result type of `Either a b`, where `a` is some sort of type that can tell us something about the possible failure and `b` is the type of a successful computation. Hence, errors use the `Left` value constructor while results use `Right`.

An example: a high-school has lockers so that students have some place to put their Guns'n'Roses posters. Each locker has a code combination. When a student wants a new locker, they tell the locker supervisor which locker number they want and he gives them the code. However, if someone is already using that locker, he can't tell them the code for the locker and they have to pick a different one. We'll use a map from `Data.Map` to represent the lockers. It'll map from locker numbers to a pair of whether the locker is in use or not and the locker code.

```
import qualified          as
data             =       |      deriving
type      =
type           =
```

Simple stuff. We introduce a new data type to represent whether a locker is taken or free and we make a type synonym for the locker code. We also make a type synonym for the type that maps from integers to pairs of locker state and code. And now, we're going to make a function that searches for the code in a locker map. We're going to use an `Either String Code` type to represent our result, because our lookup can fail in two ways — the locker can be taken, in which case we can't tell the code or the locker number might not exist at all. If the lookup fails, we're just going to use a `String` to tell what's happened.

```
           ::      ->           ->
                          =
     case                              of
               ->      $ "Locker number " ++                ++ " doesn't exist!"
                          -> if       /=
                               then
```

```
                                    else       $ "Locker " ++                  +
  + " is already taken!"
```

We do a normal lookup in the map. If we get a `Nothing`, we return a value of type `Left String`, saying that the locker doesn't exist at all. If we do find it, then we do an additional check to see if the locker is taken. If it is, return a `Left` saying that it's already taken. If it isn't, then return a value of type `Right Code`, in which we give the student the correct code for the locker. It's actually a `Right String`, but we introduced that type synonym to introduce some additional documentation into the type declaration. Here's an example map:

```
          ::
          =
        100         "ZD39I"
        101        "JAH3I"
        103        "IQSA9"
        105        "QOTSA"
        109         "893JJ"
        110         "99292"
```

Now let's try looking up some locker codes.

```
  ghci>              101
       "JAH3I"
  ghci>              100
       "Locker 100 is already taken!"
  ghci>              102
       "Locker number 102 doesn't exist!"
  ghci>              110
       "Locker 110 is already taken!"
  ghci>              105
       "QOTSA"
```

We could have used a `Maybe a` to represent the result but then we wouldn't know why we couldn't get the code. But now, we have information about the failure in our result type.

Rekurzivní datové struktury

As we've seen, a constructor in an algebraic data type can have several (or none at all) fields and each field must be of some concrete type. With that in mind, we can make types whose constructors have fields that are of the same type! Using that, we can create recursive data types, where one value of some type contains values of that type, which in turn contain more values of the same type and so on.

Think about this list: `[5]`. That's just syntactic sugar for `5:[]`. On the left side of the `:`, there's a value and on the right side, there's a list. And in this case, it's an empty list. Now how about the list `[4,5]`? Well, that desugars to `4:(5:[])`. Looking at the first `:`, we see that it also has an element on its left side and a list (`5:[]`) on its right side. Same goes for a list like `3:(4:(5:6:[])`), which could be written either like that or like `3:4:5:6:[]` (because `:` is right-associative) or `[3,4,5,6]`.

We could say that a list can be an empty list or it can be an element joined together with a `:` with another list (that can be either the empty list or not).

Let's use algebraic data types to implement our own list then!

```
  data        =       |                deriving
```

This reads just like our definition of lists from one of the previous paragraphs. It's either an empty list or a combination of a head with some value and a list. If you're confused about this, you might find it easier to understand in record syntax.

```
data        =       |                 ::              ::          deriving
```

You might also be confused about the `Cons` constructor here. *cons* is another word for `:`. You see, in lists, `:` is actually a constructor that takes a value and another list and returns a list. We can already use our new list type! In other words, it has two fields. One field is of the type of `a` and the other is of the type `[a]`.

```
ghci>

ghci> 5 `Cons`
     5
ghci> 4 `Cons`  5 `Cons`
     4       5
ghci> 3 `Cons`  4 `Cons`  5 `Cons`
     3       4       5
```

We called our `Cons` constructor in an infix manner so you can see how it's just like `:`. `Empty` is like `[]` and `4 `Cons` (5 `Cons` Empty)` is like `4:(5:[])`.

We can define functions to be automatically infix by making them comprised of only special characters. We can also do the same with constructors, since they're just functions that return a data type. So check this out.

```
infixr 5
data        =       |                deriving
```

First off, we notice a new syntactic construct, the fixity declarations. When we define functions as operators, we can use that to give them a fixity (but we don't have to). A fixity states how tightly the operator binds and whether it's left-associative or right-associative. For instance, `*`'s fixity is `infixl 7 *` and `+`'s fixity is `infixl 6`. That means that they're both left-associative (`4 * 3 * 2` is `(4 * 3) * 2`) but `*` binds tighter than `+`, because it has a greater fixity, so `5 * 4 + 3` is `(5 * 4) + 3`.

Otherwise, we just wrote `a :-: (List a)` instead of `Cons a (List a)`. Now, we can write out lists in our list type like so:

```
ghci> 3     4     5
      3       4       5
ghci> let   = 3     4     5
ghci> 100
      100       3       4       5
```

When deriving `Show` for our type, Haskell will still display it as if the constructor was a prefix function, hence the parentheses around the operator (remember, `4 + 3` is `(+) 4 3`).

Let's make a function that adds two of our lists together. This is how `++` is defined for normal lists:

```
infixr 5  ++
     ::     ->     ->
       ++    =
       ++    =   :     ++
```

So we'll just steal that for our own list. We'll name the function `.++`.

```
infixr 5
      ::        ->        ->
             =
```

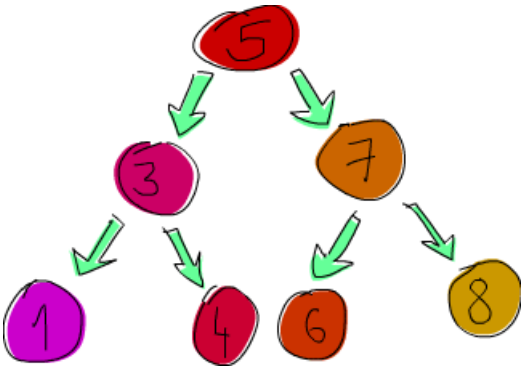
```
                =
```

And let's see if it works ...

```
ghci> let    = 3      4      5
ghci> let    = 6      7
ghci>
       3         4         5         6         7
```

Nice. Is nice. If we wanted, we could implement all of the functions that operate on lists on our own list type.

Notice how we pattern matched on `(x :-: xs)`. That works because pattern matching is actually about matching constructors. We can match on `:-:` because it is a constructor for our own list type and we can also match on `:` because it is a constructor for the built-in list type. Same goes for `[]`. Because pattern matching works (only) on constructors, we can match for stuff like that, normal prefix constructors or stuff like `8` or `'a'`, which are basically constructors for the numeric and character types, respectively.



Now, we're going to implement a **binary search tree**. If you're not familiar with binary search trees from languages like C, here's what they are: an element points to two elements, one on its left and one on its right. The element to the left is smaller, the element to the right is bigger. Each of those elements can also point to two elements (or one, or none). In effect, each element has up to two sub-trees. And a cool thing about binary search trees is that we know that all the elements at the left sub-tree of, say, 5 are going to be smaller than 5. Elements in its right sub-tree are going to be bigger. So if we need to find if 8 is in our tree, we'd start at 5 and then because 8 is greater than 5, we'd go right. We're now at 7 and because 8 is greater than 7, we go right again. And we've found our element in three hops! Now if this were a normal list (or a tree, but really unbalanced), it would take us seven hops instead of three to see if 8 is in there.

Sets and maps from `Data.Set` and `Data.Map` are implemented using trees, only instead of normal binary search trees, they use balanced binary search trees, which are always balanced. But right now, we'll just be implementing normal binary search trees.

Here's what we're going to say: a tree is either an empty tree or it's an element that contains some value and two trees. Sounds like a perfect fit for an algebraic data type!

```
data         =              |                          deriving
```

Okay, good, this is good. Instead of manually building a tree, we're going to make a function that takes a tree and an element and inserts an element. We do this by comparing the value we want to insert to the root node and then if it's smaller, we go left, if it's larger, we go right. We do the same for every subsequent node until we reach an empty tree. Once we've reached an empty tree, we just insert a node with that value instead of the empty tree.

In languages like C, we'd do this by modifying the pointers and values inside the tree. In Haskell, we can't really modify our tree, so we have to make a new sub-tree each time we decide to go left or right and in the end the insertion function returns a completely new tree, because Haskell doesn't really have a concept of pointer, just values. Hence, the type for our insertion function is going to be something like `a -> Tree a - > Tree a`. It takes an element and a tree and returns a new tree that has that element inside. This might seem like it's inefficient but laziness takes care of that problem.

So, here are two functions. One is a utility function for making a singleton tree (a tree with just one node) and a function to insert an element into a tree.

```
          ::   ->
            =

          ::          =>   ->         ->
                       =

    |   ==   =
    |   <    =
    |   >    =
```

The `singleton` function is just a shortcut for making a node that has something and then two empty sub-trees. In the insertion function, we first have the edge condition as a pattern. If we've reached an empty sub-tree, that means we're where we want and instead of the empty tree, we put a singleton tree with our element. If we're not inserting into an empty tree, then we have to check some things. First off, if the element we're inserting is equal to the root element, just return a tree that's the same. If it's smaller, return a tree that has the same root value, the same right sub-tree but instead of its left sub-tree, put a tree that has our value inserted into it. Same (but the other way around) goes if our value is bigger than the root element.

Next up, we're going to make a function that checks if some element is in the tree. First, let's define the edge condition. If we're looking for an element in an empty tree, then it's certainly not there. Okay. Notice how this is the same as the edge condition when searching for elements in lists. If we're looking for an element in an empty list, it's not there. Anyway, if we're not looking for an element in an empty tree, then we check some things. If the element in the root node is what we're looking for, great! If it's not, what then? Well, we can take advantage of knowing that all the left elements are smaller than the root node. So if the element we're looking for is smaller than the root node, check to see if it's in the left sub-tree. If it's bigger, check to see if it's in the right sub-tree.

```
          ::          =>   ->         ->
                       =

    |   ==   =
    |   <    =
    |   >    =
```

All we had to do was write up the previous paragraph in code. Let's have some fun with our trees! Instead of manually building one (although we could), we'll use a fold to build up a tree from a list. Remember, pretty much everything that traverses a list one by one and then returns some sort of value can be implemented with a fold! We're going to start with the empty tree and then approach a list from the right and just insert element after element into our accumulator tree.

```
ghci> let      =  8 6 4 1 7 3 5
ghci> let          =
ghci>
     5       3       1                       4
       7       6                       8
```

In that `foldr`, `treeInsert` was the folding function (it takes a tree and a list element and produces a new tree) and `EmptyTree` was the starting accumulator. `nums`, of course, was the list we were folding over.

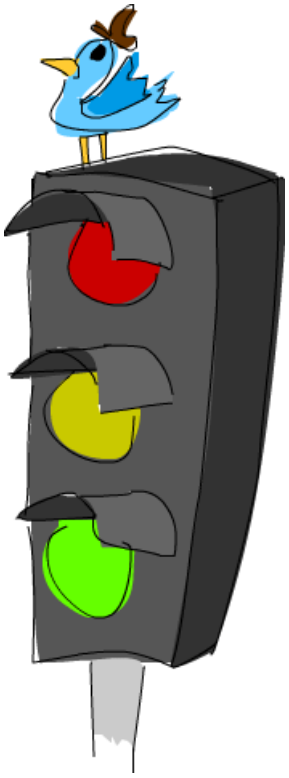
When we print our tree to the console, it's not very readable, but if we try, we can make out its structure. We see that the root node is 5 and then it has two sub-trees, one of which has the root node of 3 and the other a 7, etc.

```
ghci> 8 `treeElem`

ghci> 100 `treeElem`

ghci> 1 `treeElem`

ghci> 10 `treeElem`
```

Checking for membership also works nicely. Cool.

So as you can see, algebraic data structures are a really cool and powerful concept in Haskell. We can use them to make anything from boolean values and weekday enumerations to binary search trees and more!

Typové třídy pro pokročilé



So far, we've learned about some of the standard Haskell typeclasses and we've seen which types are in them. We've also learned how to automatically make our own types instances of the standard typeclasses by asking Haskell to derive the instances for us. In this section, we're going to learn how to make our own typeclasses and how to make types instances of them by hand.

A quick recap on typeclasses: typeclasses are like interfaces. A typeclass defines some behavior (like comparing for equality, comparing for ordering, enumeration) and then types that can behave in that way are made instances of that typeclass. The behavior of typeclasses is achieved by defining functions or just type declarations that we then implement. So when we say that a type is an instance of a typeclass, we mean that we can use the functions that the typeclass defines with that type.

Typeclasses have pretty much nothing to do with classes in languages like Java or Python. This confuses many people, so I want you to forget everything you know about classes in imperative languages right now.

For example, the `Eq` typeclass is for stuff that can be equated. It defines the functions `==` and `/=`. If we have a type (say, `Car`) and comparing two cars with the equality function `==` makes sense, then it makes sense for `Car` to be an instance of `Eq`.

This is how the `Eq` class is defined in the standard prelude:

```
class      where
          ::    ->    ->
          ::    ->    ->
       ==    =           /=
       /=    =           ==
```

Woah, woah, woah! Some new strange syntax and keywords there! Don't worry, this will all be clear in a second. First off, when we write `class Eq a where`, this means that we're defining a new typeclass and that's called `Eq`. The `a` is the type variable and it means that `a` will play the role of the type that we will soon be making an instance of `Eq`. It doesn't have to be called `a`, it doesn't even have to be one letter, it just has to be a lowercase word. Then, we define several functions. It's not mandatory to implement the function bodies themselves, we just have to specify the type declarations for the functions.

> Some people might understand this better if we wrote `class Eq equatable where` and then specified the type declarations like `(==) :: equatable -> equatable -> Bool`.

Anyway, we *did* implement the function bodies for the functions that `Eq` defines, only we defined them in terms of mutual recursion. We said that two instances of `Eq` are equal if they are not different and they are different if they are not equal. We didn't have to do this, really, but we did and we'll see how this helps us soon.

> If we have say `class Eq a where` and then define a type declaration within that class like `(==) :: a -> -a -> Bool`, then when we examine the type of that function later on, it will have the type of `(Eq a) => a -> a -> Bool`.

So once we have a class, what can we do with it? Well, not much, really. But once we start making types instances of that class, we start getting some nice functionality. So check out this type:

```
data              =      |         |
```

It defines the states of a traffic light. Notice how we didn't derive any class instances for it. That's because we're going to write up some instances by hand, even though we could derive them for types like `Eq` and `Show`. Here's how we make it an instance of `Eq`.

```
instance                  where
         ==      =
            ==         =
              ==           =
        ==    =
```

We did it by using the *instance* keyword. So *class* is for defining new typeclasses and *instance* is for making our types instances of typeclasses. When we were defining `Eq`, we wrote `class Eq a where` and we said that `a` plays the role of whichever type will be made an instance later on. We can see that clearly here, because when we're making an instance, we write `instance Eq TrafficLight where`. We replace the `a` with the actual type.

Because `==` was defined in terms of `/=` and vice versa in the *class* declaration, we only had to overwrite one of them in the instance declaration. That's called the minimal complete definition for the typeclass — the minimum of functions that we have to implement so that our type can behave like the class advertises. To fulfill the minimal complete definition for `Eq`, we have to overwrite either one of `==` or `/=`. If `Eq` was defined simply like this:

```
class       where
          ::    ->    ->
          ::    ->    ->
```

we'd have to implement both of these functions when making a type an instance of it, because Haskell wouldn't know how these two functions are related. The minimal complete definition would then be: both `==` and `/=`.

You can see that we implemented `==` simply by doing pattern matching. Since there are many more cases where two lights aren't equal, we specified the ones that are equal and then just did a catch-all pattern saying that if it's none of the previous combinations, then two lights aren't equal.

Let's make this an instance of `Show` by hand, too. To satisfy the minimal complete definition for `Show`, we just have to implement its `show` function, which takes a value and turns it into a string.

```
instance                    where
              = "Red light"
                 = "Yellow light"
                = "Green light"
```

Once again, we used pattern matching to achieve our goals. Let's see how it works in action:

```
ghci>      ==
ghci>      ==
ghci>      `elem`
ghci>
```

Nice. We could have just derived `Eq` and it would have had the same effect (but we didn't for educational purposes). However, deriving `Show` would have just directly translated the value constructors to strings. But if we want lights to appear like `"Red light"`, then we have to make the instance declaration by hand.

You can also make typeclasses that are subclasses of other typeclasses. The *class* declaration for `Num` is a bit long, but here's the first part:

```
class        =>       where
```

As we mentioned previously, there are a lot of places where we can cram in class constraints. So this is just like writing `class Num a where`, only we state that our type `a` must be an instance of `Eq`. We're essentially saying that we have to make a type an instance of `Eq` before we can make it an instance of `Num`. Before some type can be considered a number, it makes sense that we can determine whether values of that type can be equated or not. That's all there is to subclassing really, it's just a class constraint on a *class* declaration! When defining function bodies in the *class* declaration or when defining them in *instance* declarations, we can assume that `a` is a part of `Eq` and so we can use `==` on values of that type.

But how are the `Maybe` or list types made as instances of typeclasses? What makes `Maybe` different from, say, `TrafficLight` is that `Maybe` in itself isn't a concrete type, it's a type constructor that takes one type parameter (like `Char` or something) to produce a concrete type (like `Maybe Char`). Let's take a look at the `Eq` typeclass again:

```
class      where
          ::    ->    ->
          ::    ->    ->
       ==    =          /=
       /=    =          ==
```

From the type declarations, we see that the `a` is used as a concrete type because all the types in functions have to be concrete (remember, you can't have a function of the type `a -> Maybe` but you can have a function of `a -> Maybe a` or `Maybe Int -> Maybe String`). That's why we can't do something like

```
instance          where
```

Because like we've seen, the `a` has to be a concrete type but `Maybe` isn't a concrete type. It's a type constructor that takes one parameter and then produces a concrete type. It would also be tedious to write `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where`, etc. for every type ever. So we could write it out like so:

```
instance              where
            ==         =    ==
              ==          =
       ==    =
```

This is like saying that we want to make all types of the form `Maybe something` an instance of `Eq`. We actually could have written `(Maybe something)`, but we usually opt for single letters to be true to the Haskell style. The `(Maybe m)` here plays the role of the `a` from `class Eq a where`. While `Maybe` isn't a concrete type, `Maybe m` is. By specifying a type parameter (`m`, which is in lowercase), we said that we want all types that are in the form of `Maybe m`, where `m` is any type, to be an instance of `Eq`.

There's one problem with this though. Can you spot it? We use `==` on the contents of the `Maybe` but we have no assurance that what the `Maybe` contains can be used with `Eq`! That's why we have to modify our *instance* declaration like this:

```
instance        =>              where
            ==         =    ==
```

```
          ==          =
      ==    =
```

We had to add a class constraint! With this *instance* declaration, we say this: we want all types of the form `Maybe m` to be part of the `Eq` typeclass, but only those types where the `m` (so what's contained inside the `Maybe`) is also a part of `Eq`. This is actually how Haskell would derive the instance too.

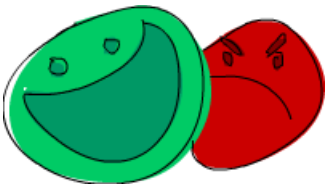
Most of the times, class constraints in *class* declarations are used for making a typeclass a subclass of another typeclass and class constraints in *instance* declarations are used to express requirements about the contents of some type. For instance, here we required the contents of the `Maybe` to also be part of the `Eq` typeclass.

When making instances, if you see that a type is used as a concrete type in the type declarations (like the `a` in `a -> a -> Bool`), you have to supply type parameters and add parentheses so that you end up with a concrete type.

> Take into account that the type you're trying to make an instance of will replace the parameter in the *class* declaration. The `a` from `class Eq a where` will be replaced with a real type when you make an instance, so try mentally putting your type into the function type declarations as well. `(==) :: Maybe -> Maybe -> Bool` doesn't make much sense but `(==) :: (Eq m) => Maybe m -> Maybe m -> Bool` does. But this is just something to think about, because `==` will always have a type of `(==) :: (Eq a) => a -> a -> Bool`, no matter what instances we make.

Ooh, one more thing, check this out! If you want to see what the instances of a typeclass are, just do `:info YourTypeClass` in GHCI. So typing `:info Num` will show which functions the typeclass defines and it will give you a list of the types in the typeclass. `:info` works for types and type constructors too. If you do `:info Maybe`, it will show you all the typeclasses that `Maybe` is an instance of. Also `:info` can show you the type declaration of a function. I think that's pretty cool.

Typová třída ano/ne



In JavaScript and some other weakly typed languages, you can put almost anything inside an if expression. For example, you can do all of the following: `if (0) alert("YEAH!") else alert("NO!")`, `if ("") alert ("YEAH!") else alert("NO!")`, `if (false) alert("YEAH") else alert("NO!")`, etc. and all of these will throw an alert of `NO!`. If you do `if ("WHAT") alert ("YEAH") else alert("NO!")`, it will alert a `"YEAH!"` because JavaScript considers non-empty strings to be a sort of true-ish value.

Even though strictly using `Bool` for boolean semantics works better in Haskell, let's try and implement that JavaScript-ish behavior anyway. For fun! Let's start out with a *class* declaration.

```
class          where
          ::    ->
```

Pretty simple. The `YesNo` typeclass defines one function. That function takes one value of a type that can be considered to hold some concept of true-ness and tells us for sure if it's true or not. Notice that from the way we use the `a` in the function, `a` has to be a concrete type.

Next up, let's define some instances. For numbers, we'll assume that (like in JavaScript) any number that isn't 0 is true-ish and 0 is false-ish.

```
instance            where
          0 =
            =
```

Empty lists (and by extensions, strings) are a no-ish value, while non-empty lists are a yes-ish value.

```
instance           where
             =
           =
```

Notice how we just put in a type parameter `a` in there to make the list a concrete type, even though we don't make any assumptions about the type that's contained in the list. What else, hmm ... I know, `Bool` itself also holds true-ness and false-ness and it's pretty obvious which is which.

```
instance            where
          =
```

Huh? What's `id`? It's just a standard library function that takes a parameter and returns the same thing, which is what we would be writing here anyway.

Let's make `Maybe a` an instance too.

```
instance                 where
                  =
                =
```

We didn't need a class constraint because we made no assumptions about the contents of the `Maybe`. We just said that it's true-ish if it's a `Just` value and false-ish if it's a `Nothing`. We still had to write out `(Maybe a)` instead of just `Maybe` because if you think about it, a `Maybe -> Bool` function can't exist (because `Maybe` isn't a concrete type), whereas a `Maybe a -> Bool` is fine and dandy. Still, this is really cool because now, any type of the form `Maybe something` is part of `YesNo` and it doesn't matter what that `something` is.

Previously, we defined a `Tree a` type, that represented a binary search tree. We can say an empty tree is false-ish and anything that's not an empty tree is true-ish.

```
instance               where
                 =
             =
```

Can a traffic light be a yes or no value? Sure. If it's red, you stop. If it's green, you go. If it's yellow? Eh, I usually run the yellows because I live for adrenaline.

```
instance                   where
            =
          =
```

Cool, now that we have some instances, let's go play!

```
ghci>       $
ghci>       "haha"
ghci>       ""
ghci>       $      0
ghci>
ghci>
ghci>
```

```
ghci>        0 0 0

ghci>
      ::           =>   ->
```

Right, it works! Let's make a function that mimics the if statement, but it works with `YesNo` values.

```
        ::           =>   ->   ->   ->
                                 = if              then          else
```

Pretty straightforward. It takes a yes-no-ish value and two things. If the yes-no-ish value is more of a yes, it returns the first of the two things, otherwise it returns the second of them.

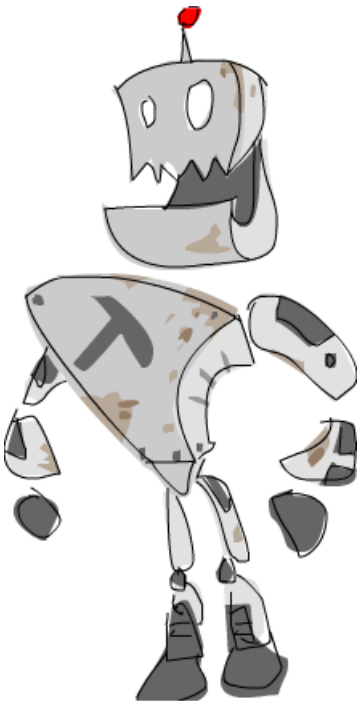
```
ghci>            "YEAH!" "NO!"
"NO!"
ghci>          2 3 4  "YEAH!" "NO!"
"YEAH!"
ghci>              "YEAH!" "NO!"
"YEAH!"
ghci>               500  "YEAH!" "NO!"
"YEAH!"
ghci>                 "YEAH!" "NO!"
"NO!"
```

Typová třída funktor

So far, we've encountered a lot of the typeclasses in the standard library. We've played with `Ord`, which is for stuff that can be ordered. We've palled around with `Eq`, which is for things that can be equated. We've seen `Show`, which presents an interface for types whose values can be displayed as strings. Our good friend `Read` is there whenever we need to convert a string to a value of some type. And now, we're going to take a look at the typeclass, which is basically for things that can be mapped over. You're probably thinking about lists now, since mapping over lists is such a dominant idiom in Haskell. And you're right, the list type is part of the `Functor` typeclass.

What better way to get to know the `Functor` typeclass than to see how it's implemented? Let's take a peek.

```
class          where
        ::    ->    ->     ->
```



Alright. We see that it defines one function, `fmap`, and doesn't provide any default implementation for it. The type of `fmap` is interesting. In the definitions of typeclasses so far, the type variable that played the role of the type in the typeclass was a concrete type, like the `a` in `(==) :: (Eq a) => a -> a -> Bool`. But now, the `f` is not a concrete type (a type that a value can hold, like `Int`, `Bool` or `Maybe String`), but a type constructor that takes one type parameter. A quick refresher example: `Maybe Int` is a concrete type, but `Maybe` is a type constructor that takes one type as the parameter. Anyway, we see that `fmap` takes a function from one type to another and a functor applied with one type and returns a functor applied with another type.

If this sounds a bit confusing, don't worry. All will be revealed soon when we check out a few examples. Hmm, this type declaration for `fmap` reminds me of something. If you don't know what the type signature of `map` is, it's: `map :: (a -> b) -> [a] -> [b]`.

Ah, interesting! It takes a function from one type to another and a list of one type and returns a list of another type. My friends, I think we have ourselves a functor! In fact, `map`

is just a `fmap` that works only on lists. Here's how the list is an instance of the `Functor` typeclass.

```
instance            where
         =
```

That's it! Notice how we didn't write `instance Functor [a] where`, because from `fmap :: (a -> b) -> f a -> f b`, we see that the `f` has to be a type constructor that takes one type. `[a]` is already a concrete type (of a list with any type inside it), while `[]` is a type constructor that takes one type and can produce types such as `[Int]`, `[String]` or even `[[String]]`.

Since for lists, `fmap` is just `map`, we get the same results when using them on lists.

```
    ::     ->     ->      ->
ghci>         2    1   3
 2 4 6
ghci>        2    1   3
 2 4 6
```

What happens when we `map` or `fmap` over an empty list? Well, of course, we get an empty list. It just turns an empty list of type `[a]` into an empty list of type `[b]`.

Types that can act like a box can be functors. You can think of a list as a box that has an infinite amount of little compartments and they can all be empty, one can be full and the others empty or a number of them can be full. So, what else has the properties of being like a box? For one, the `Maybe a` type. In a way, it's like a box that can either hold nothing, in which case it has the value of `Nothing`, or it can hold one item, like `"HAHA"`, in which case it has a value of `Just "HAHA"`. Here's how `Maybe` is a functor.

```
instance                where
                    =
                   =
```

Again, notice how we wrote `instance Functor Maybe where` instead of `instance Functor (Maybe m) where`, like we did when we were dealing with `Maybe` and `YesNo`. `Functor` wants a type constructor that takes one type and not a concrete type. If you mentally replace the `f`s with `Maybe`s, `fmap` acts like a `(a -> b) -> Maybe a -> Maybe b` for this particular type, which looks OK. But if you replace `f` with `(Maybe m)`, then it would seem to act like a `(a -> b) -> Maybe m a -> Maybe m b`, which doesn't make any damn sense because `Maybe` takes just one type parameter.

Anyway, the `fmap` implementation is pretty simple. If it's an empty value of `Nothing`, then just return a `Nothing`. If we map over an empty box, we get an empty box. It makes sense. Just like if we map over an empty list, we get back an empty list. If it's not an empty value, but rather a single value packed up in a `Just`, then we apply the function on the contents of the `Just`.

```
ghci>           " HEY GUYS IM INSIDE THE JUST"         "Something serious."
     "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci>           " HEY GUYS IM INSIDE THE JUST"

ghci>         2         200
     400
ghci>         2
```

Another thing that can be mapped over and made an instance of `Functor` is our `Tree a` type. It can be thought of as a box in a way (holds several or no values) and the `Tree` type constructor takes exactly one type parameter. If you look at `fmap` as if it were a function made only for `Tree`, its type signature would look like `(a -> b) -> Tree a -> Tree b`. We're going to use recursion on this one. Mapping over an empty tree will produce an empty tree. Mapping over a non-empty tree will be a tree

consisting of our function applied to the root value and its left and right sub-trees will be the previous sub-trees, only our function will be mapped over them.

```
instance              where
                     =
                                  =


ghci>        2

ghci>        4                              5 7 3 2 1 7
     28       4               8                12                20
```

Nice! Now how about `Either a b`? Can this be made a functor? The `Functor` typeclass wants a type constructor that takes only one type parameter but `Either` takes two. Hmmm! I know, we'll partially apply `Either` by feeding it only one parameter so that it has one free parameter. Here's how `Either a` is a functor in the standard libraries:

```
instance                    where
                   =
                  =
```

Well well, what did we do here? You can see how we made `Either a` an instance instead of just `Either`. That's because `Either a` is a type constructor that takes one parameter, whereas `Either` takes two. If `fmap` was specifically for `Either a`, the type signature would then be `(b -> c) -> Either a b -> Either a c` because that's the same as `(b -> c) -> (Either a) b -> (Either a) c`. In the implementation, we mapped in the case of a `Right` value constructor, but we didn't in the case of a `Left`. Why is that? Well, if we look back at how the `Either a b` type is defined, it's kind of like:

```
data            =         |
```

Well, if we wanted to map one function over both of them, `a` and `b` would have to be the same type. I mean, if we tried to map a function that takes a string and returns a string and the `b` was a string but the `a` was a number, that wouldn't really work out. Also, from seeing what `fmap`'s type would be if it operated only on `Either` values, we see that the first parameter has to remain the same while the second one can change and the first parameter is actualized by the `Left` value constructor.

This also goes nicely with our box analogy if we think of the `Left` part as sort of an empty box with an error message written on the side telling us why it's empty.

Maps from `Data.Map` can also be made a functor because they hold values (or not!). In the case of `Map k v`, `fmap` will map a function `v -> v'` over a map of type `Map k v` and return a map of type `Map k v'`.

> Note, the `'` has no special meaning in types just like it doesn't have special meaning when naming values. It's used to denote things that are similar, only slightly changed.

Try figuring out how `Map k` is made an instance of `Functor` by yourself!

With the `Functor` typeclass, we've seen how typeclasses can represent pretty cool higher-order concepts. We've also had some more practice with partially applying types and making instances. In one of the next chapters, we'll also take a look at some laws that apply for functors.

> **Just one more thing!** Functors should obey some laws so that they may have some properties that we can depend on and not think about too much. If we use `fmap (+1)` over the list `[1,2,3,4]`, we expect the result to be `[2,3,4,5]` and not its reverse, `[5,4,3,2]`. If we use `fmap (\a -> a)` (the identity function, which just returns its parameter) over some list, we

expect to get back the same list as a result. For example, if we gave the wrong functor instance to our `Tree` type, using `fmap` over a tree where the left sub-tree of a node only has elements that are smaller than the node and the right sub-tree only has nodes that are larger than the node might produce a tree where that's not the case. We'll go over the functor laws in more detail in one of the next chapters.

Druhy a nějaké to typové kung-fu

Type constructors take other types as parameters to eventually produce concrete types. That kind of reminds me of functions, which take values as parameters to produce values. We've seen that type constructors can be partially applied (`Either String` is a type that takes one type and produces a concrete type, like `Either String Int`), just like functions can. This is all very interesting indeed. In this section, we'll take a look at formally defining how types are applied to type constructors, just like we took a look at formally defining how values are applied to functions by using type declarations. **You don't really have to read this section to continue on your magical Haskell quest** and if you don't understand it, don't worry about it. However, getting this will give you a very thorough understanding of the type system.

So, values like `3`, `"YEAH"` or `takeWhile` (functions are also values, because we can pass them around and such) each have their own type. Types are little labels that values carry so that we can reason about the values. But types have their own little labels, called **kinds**. A kind is more or less the type of a type. This may sound a bit weird and confusing, but it's actually a really cool concept.

What are kinds and what are they good for? Well, let's examine the kind of a type by using the `:k` command in GHCI.

```
ghci>
    ::
```

A star? How quaint. What does that mean? A `*` means that the type is a concrete type. A concrete type is a type that doesn't take any type parameters and values can only have types that are concrete types. If I had to read `*` out loud (I haven't had to do that so far), I'd say *star* or just *type*.

Okay, now let's see what the kind of `Maybe` is.

```
ghci>
      ::   ->
```

The `Maybe` type constructor takes one concrete type (like `Int`) and then returns a concrete type like `Maybe Int`. And that's what this kind tells us. Just like `Int -> Int` means that a function takes an `Int` and returns an `Int`, `* -> *` means that the type constructor takes one concrete type and returns a concrete type. Let's apply the type parameter to `Maybe` and see what the kind of that type is.

```
ghci>
         ::
```

Just like I expected! We applied the type parameter to `Maybe` and got back a concrete type (that's what `* -> *` means. A parallel (although not equivalent, types and kinds are two different things) to this is if we do `:t isUpper` and `:t isUpper 'A'`.

`isUpper` has a type of `Char -> Bool` and `isUpper 'A'` has a type of `Bool`, because its value is basically `False`. Both those types, however, have a kind of `*`.

We used `:k` on a type to get its kind, just like we can use `:t` on a value to get its type. Like we said, types are the labels of values and kinds are the labels of types and there are parallels between the two.

Let's look at another kind.

```
ghci>
       ::   ->   ->
```

Aha, this tells us that `Either` takes two concrete types as type parameters to produce a concrete type. It also looks kind of like a type declaration of a function that takes two values and returns something. Type constructors are curried (just like functions), so we can partially apply them.

```
ghci>
             ::   ->
ghci>
                  ::
```

When we wanted to make `Either` a part of the `Functor` typeclass, we had to partially apply it because `Functor` wants types that take only one parameter while `Either` takes two. In other words, `Functor` wants types of kind `* -> *` and so we had to partially apply `Either` to get a type of kind `* -> *` instead of its original kind `* -> * -> *`. If we look at the definition of `Functor` again

```
class           where
         ::    ->    ->     ->
```

we see that the `f` type variable is used as a type that takes one concrete type to produce a concrete type. We know it has to produce a concrete type because it's used as the type of a value in a function. And from that, we can deduce that types that want to be friends with `Functor` have to be of kind `* -> *`.

Now, let's do some type-foo. Take a look at this typeclass that I'm just going to make up right now:

```
class        where
         ::     ->
```

Man, that looks weird. How would we make a type that could be an instance of that strange typeclass? Well, let's look at what its kind would have to be. Because `j a` is used as the type of a value that the `tofu` function takes as its parameter, `j a` has to have a kind of `*`. We assume `*` for `a` and so we can infer that `j` has to have a kind of `* -> *`. We see that `t` has to produce a concrete value too and that it takes two types. And knowing that `a` has a kind of `*` and `j` has a kind of `* -> *`, we infer that `t` has to have a kind of `* -> (* -> *) -> *`. So it takes a concrete type (`a`), a type constructor that takes one concrete type (`j`) and produces a concrete type. Wow.

OK, so let's make a type with a kind of `* -> (* -> *) -> *`. Here's one way of going about it.

```
data            =                   ::      deriving
```

How do we know this type has a kind of `* -> (* -> *) - > *`? Well, fields in ADTs are made to hold values, so they must be of kind `*`, obviously. We assume `*` for `a`, which means that `b` takes one type parameter and so its kind is `* -> *`. Now we know the kinds of both `a` and `b` and because they're parameters for `Frank`, we see that `Frank` has a kind of `* -> (* -> *) -> *` The first `*` represents `a` and the `(* -> *)` represents `b`. Let's make some `Frank` values and check out their types.

```
ghci>                    =       "HAHA"
                 =       "HAHA"  ::
ghci>                    =       'a'
                 =       'a'                     ::
ghci>                    = "YES"
                 = "YES"  ::
```

Hmm. Because `frankField` has a type of form `a b`, its values must have types that are of a similar form as well. So they can be `Just "HAHA"`, which has a type of `Maybe [Char]` or it can have a value of `['Y','E','S']`, which has a type of `[Char]` (if we used our own list type for this, it would have a type of `List Char`). And we see that the types of the `Frank` values correspond with the kind for `Frank`. `[Char]` has a kind of `*` and `Maybe` has a kind of `* -> *`. Because in order to have a value, it has to be a concrete type and thus has to be fully applied, every value of `Frank blah blaah` has a kind of `*`.

Making `Frank` an instance of `Tofu` is pretty simple. We see that `tofu` takes a `j a` (so an example type of that form would be `Maybe Int`) and returns a `t a j`. So if we replace `Frank` with `j`, the result type would be `Frank Int Maybe`.

```
instance            where
          =
```

```
ghci>            'a'  ::
                 =       'a'
ghci>       "HELLO"   ::
                 =  "HELLO"
```

Not very useful, but we did flex our type muscles. Let's do some more type-foo. We have this data type:

```
data             =               ::          ::
```

And now we want to make it an instance of `Functor`. `Functor` wants types of kind `* -> *` but `Barry` doesn't look like it has that kind. What is the kind of `Barry`? Well, we see it takes three type parameters, so it's going to be `something -> something -> something -> *`. It's safe to say that `p` is a concrete type and thus has a kind of `*`. For `k`, we assume `*` and so by extension, `t` has a kind of `* -> *`. Now let's just replace those kinds with the *somethings* that we used as placeholders and we see it has a kind of `(* -> *) -> * -> * -> * -> *`. Let's check that with GHCI.

```
ghci>
      ::    ->    ->    ->    ->
```

Ah, we were right. How satisfying. Now, to make this type a part of `Functor` we have to partially apply the first two type parameters so that we're left with `* -> *`. That means that the start of the instance declaration will be: `instance Functor (Barry a b) where`. If we look at `fmap` as if it was made specifically for `Barry`, it would have a type of `fmap :: (a -> b) -> Barry c d a -> Barry c d b`, because we just replace the `Functor`'s `f` with `Barry c d`. The third type parameter from `Barry` will have to change and we see that it's conviniently in its own field.

```
instance                    where
                    =          =     =              =            =
```

There we go! We just mapped the `f` over the first field.

In this section, we took a good look at how type parameters work and kind of formalized them with kinds, just like we formalized function parameters with type declarations. We saw that there are interesting parallels between functions and type constructors. They are, however, two completely different things. When working on real Haskell, you usually won't have to mess with kinds and do kind inference by hand like we did now. Usually, you just have to partially apply your own type to `* -> *` or `*` when making it an instance of one of the standard typeclasses, but it's good to know how and why that actually works. It's also

interesting to see that types have little types of their own. Again, you don't really have to understand everything we did here to read on, but if you understand how kinds work, chances are that you have a very solid grasp of Haskell's type system.

Moduly Obsah Input and Output

← Vytváříme si své typy a typové třídy Obsah Functionally Solving Problems →

Input and Output

We've mentioned that Haskell is a purely functional language. Whereas in imperative languages you usually get things done by giving the computer a series of steps to execute, functional programming is more of defining what stuff is. In Haskell, a function can't change some state, like changing the contents of a variable (when a function changes state, we say that the function has *side-effects*). The only thing a function can do in Haskell is give us back some result based on the parameters we gave it. If a function is called two times with the same parameters, it has to return the same result. While this may seem a bit limiting when you're coming from an imperative world, we've seen that it's actually really cool. In an imperative language, you have no guarantee that a simple function that should just crunch some numbers won't burn down your house, kidnap your dog and scratch your car with a potato while crunching those numbers. For instance, when we were making a binary search tree, we didn't insert an element into a tree by modifying some tree in place. Our function for inserting into a binary search tree actually returned a new tree, because it can't change the old one.

While functions being unable to change state is good because it helps us reason about our programs, there's one problem with that. If a function can't change anything in the world, how is it supposed to tell us what it calculated? In order to tell us what it calculated, it has to change the state of an output device (usually the state of the screen), which then emits photons that travel to our brain and change the state of our mind, man.

Do not despair, all is not lost. It turns out that Haskell actually has a really clever system for dealing with functions that have side-effects that neatly separates the part of our program that is pure and the part of our program that is impure, which does all the dirty work like talking to the keyboard and the screen. With those two parts separated, we can still reason about our pure program and take advantage of all the things that purity offers, like laziness, robustness and modularity while efficiently communicating with the outside world.

Hello, world!

Up until now, we've always loaded our functions into GHCI to test them out and play with them. We've also explored the standard library functions that way. But now, after eight or so chapters, we're finally going to write our first *real* Haskell program! Yay! And sure enough, we're going to do the good old `"hello, world"` schtick.

Hey! For the purposes of this chapter, I'm going to assume you're using a unix-y environment for learning Haskell. If you're in Windows, I'd suggest you download Cygwin, which is a Linux-like environment for Windows, A.K.A. just what you need.

So, for starters, punch in the following in your favorite text editor:

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

We just defined a name called `main` and in it we call a function called `putStrLn` with the parameter `"hello, world"`. Looks pretty much run of the mill, but it isn't, as we'll see in just a few moments. Save that file as `helloworld.hs`.

And now, we're going to do something we've never done before. We're actually going to compile our program! I'm so excited! Open up your terminal and navigate to the directory where `helloworld.hs` is located and do the following:

Okay! With any luck, you got something like this and now you can run your program by doing `./helloworld`.

And there we go, our first compiled program that printed out something to the terminal. How extraordinarily boring!

Let's examine what we wrote. First, let's look at the type of the function `putStrLn`.

```
ghci>
        ::         ->
ghci>              "hello, world"
        "hello, world" ::
```

We can read the type of `putStrLn` like this: `putStrLn` takes a string and returns an **I/O action** that has a result type of `()` (i.e. the empty tuple, also know as unit). An I/O action is something that, when performed, will carry out an action with a side-effect (that's usually either reading from the input or printing stuff to the screen) and will also contain some kind of return value inside it. Printing a string to the terminal doesn't really have any kind of meaningful return value, so a dummy value of `()` is used.

The empty tuple is a value of `()` and it also has a type of `()`.

So, when will an I/O action be performed? Well, this is where `main` comes in. An I/O action will be performed when we give it a name of `main` and then run our program.

Having your whole program be just one I/O action seems kind of limiting. That's why we can use *do* syntax to glue together several I/O actions into one. Take a look at the following example:

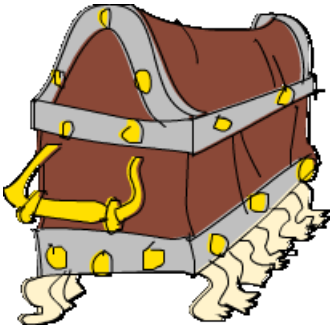
```
    = do
          "Hello, what's your name?"
      <-
           "Hey " ++      ++ ", you rock!"
```

Ah, interesting, new syntax! And this reads pretty much like an imperative program. If you compile it and try it out, it will probably behave just like you expect it to. Notice that we said *do* and then we laid out a series of steps, like we would in an imperative program. Each of these steps is an I/O action. By putting them together with *do* syntax, we glued them into one I/O action. The action that we got has a type of `IO ()`, because that's the type of the last I/O action inside.

Because of that, `main` always has a type signature of `main :: IO something`, where `something` is some concrete type. By convention, we don't usually specify a type declaration for `main`.

%0interesting thing that we haven't met before is the third line, which states `name <- getLine`. It looks like it reads a line from the input and stores it into a variable called `name`. Does it really? Well, let's examine the type of `getLine`.

```
ghci>
        ::
```



Aha, o-kay. `getLine` is an I/O action that contains a result type of `String`. That makes sense, because it will wait for the user to input something at the terminal and then that something will be represented as a string. So what's up with `name <- getLine` then? You can read that piece of code like this: **perform the I/O action `getLine` and then bind its result value to `name`**. `getLine` has a type of `IO String`, so `name` will have a type of `String`. You can think of an I/O action as a box with little feet that will go out into the real world and do something there (like write some graffiti on a wall) and maybe bring back some data. Once it's fetched that data for you, the only way to open the box and get the data inside it is to use the `<-` construct. And if we're taking data out of an I/O action, we can only take it out when we're inside another I/O action. This is how Haskell manages to neatly separate the pure and impure parts of our code. `getLine` is in a sense impure because its result value is not guaranteed to be the same when performed twice. That's why it's sort of *tainted* with the `IO` type constructor and we can only get that data out in I/O code. And because I/O code is tainted too, any computation that depends on tainted I/O data will have a tainted result.

When I say *tainted*, I don't mean tainted in such a way that we can never use the result contained in an I/O action ever again in pure code. No, we temporarily *un-taint* the data inside an I/O action when we bind it to a name. When we do `name <- getLine`, `name` is just a normal string, because it represents what's inside the box. We can have a really complicated function that, say, takes your name (a normal string) as a parameter and tells you your fortune and your whole life's future based on your name. We can do this:

```
    = do
          "Hello, what's your name?"
      <-
          $ "Read this carefully, because this is your future: " ++
```

and `tellFortune` (or any of the functions it passes `name` to) doesn't have to know anything about I/O, it's just a normal `String -> String` function!

Take a look at this piece of code. Is it valid?

```
        = "Hello, my name is " ++
```

If you said no, go eat a cookie. If you said yes, drink a bowl of molten lava. Just kidding, don't! The reason that this doesn't work is that `++` requires both its parameters to be lists over the same type. The left parameter has a type of `String` (or `[Char]` if you will), whilst `getLine` has a type of `IO String`. You can't concatenate a string and an I/O action. We first have to get the result out of the I/O action to get a value of type `String` and the only way to do that is to say something like `name <- getLine` inside some other I/O action. If we want to deal with impure data, we have to do it in an impure environment. So the taint of impurity spreads around much like the undead scourge and it's in our best interest to keep the I/O parts of our code as small as possible.

Every I/O action that gets performed has a result encapsulated within it. That's why our previous example program could also have been written like this:

```
    = do
      <-          "Hello, what's your name?"
       <-
```

```
                "Hey " ++       ++ ", you rock!"
```

However, `foo` would just have a value of `()`, so doing that would be kind of moot. Notice that we didn't bind the last `putStrLn` to anything. That's because in a *do* block, **the last action cannot be bound to a name** like the first two were. We'll see exactly why that is so a bit later when we venture off into the world of monads. For now, you can think of it in the way that the *do* block automatically extracts the value from the last action and binds it to its own result.

Except for the last line, every line in a *do* block that doesn't bind can also be written with a bind. So `putStrLn "BLAH"` can be written as `_ <- putStrLn "BLAH"`. But that's useless, so we leave out the `<-` for I/O actions that don't contain an important result, like `putStrLn` *`something`*.

Beginners sometimes think that doing

```
      =
```

will read from the input and then bind the value of that to `name`. Well, it won't, all this does is give the `getLine` I/O action a different name called, well, `name`. Remember, to get the value out of an I/O action, you have to perform it inside another I/O action by binding it to a name with `<-`.

I/O actions will only be performed when they are given a name of `main` or when they're inside a bigger I/O action that we composed with a *do* block. We can also use a *do* block to glue together a few I/O actions and then we can use that I/O action in another *do* block and so on. Either way, they'll be performed only if they eventually fall into `main`.

Oh, right, there's also one more case when I/O actions will be performed. When we type out an I/O action in GHCI and press return, it will be performed.

```
ghci>          "HEEY"
```

Even when we just punch out a number or call a function in GHCI and press return, it will evaluate it (as much as it needs) and then call `show` on it and then it will print that string to the terminal using `putStrLn` implicitly.

Remember *let* bindings? If you don't, refresh your memory on them by reading this section. They have to be in the form of `let` *`bindings`* `in` *`expression`*, where *`bindings`* are names to be given to expressions and *`expression`* is the expression that is to be evaluated that sees them. We also said that in list comprehensions, the *in* part isn't needed. Well, you can use them in *do* blocks pretty much like you use them in list comprehensions. Check this out:

```
import

     = do
          "What's your first name?"
           <-
          "What's your last name?"
          <-
    let              =
                     =
          $ "hey " ++             ++ " " ++             ++ ", how are you?"
```

See how the I/O actions in the *do* block are lined up? Also notice how the *let* is lined up with the I/O actions and the names of the *let* are lined up with each other? That's good practice, because indentation is important in Haskell. Now, we did `map toUpper firstName`, which turns something like `"John"` into a much cooler string like `"JOHN"`. We bound that uppercased string to a name and then used it in a string later on that we printed to the terminal.

You may be wondering when to use `<-` and when to use *let* bindings? Well, remember, `<-` is (for now) for performing I/O actions and binding their results to names. `map toUpper firstName`, however, isn't an I/O action. It's a pure expression in Haskell. So use `<-` when you want to bind results of I/O actions to names and you can use *let* bindings to bind pure expressions to names. Had we done something like `let firstName = getLine`, we would have just called the `getLine` I/O action a different name and we'd still have to run it through a `<-` to perform it.

Now we're going to make a program that continuously reads a line and prints out the same line with the words reversed. The program's execution will stop when we input a blank line. This is the program:

```
 = do
    <-
if
    then
    else do
                $


        ::        ->
        =         .             .
```

To get a feel of what it does, you can run it before we go over the code.

Protip: To run a program you can either compile it and then run the produced executable file by doing `ghc --make helloworld` and then `./helloworld` or you can use the `runhaskell` command like so: `runhaskell helloworld.hs` and your program will be executed on the fly.

First, let's take a look at the `reverseWords` function. It's just a normal function that takes a string like `"hey there man"` and then calls `words` with it to produce a list of words like `["hey","there","man"]`. Then we map `reverse` on the list, getting `["yeh","ereht","nam"]` and then we put that back into one string by using `unwords` and the final result is `"yeh ereht nam"`. See how we used function composition here. Without function composition, we'd have to write something like `reverseWords st = unwords (map reverse (words st))`.

What about `main`? First, we get a line from the terminal by performing `getLine` call that line `line`. And now, we have a conditional expression. Remember that in Haskell, every *if* must have a corresponding *else* because every expression has to have some sort of value. We make the *if* so that when a condition is true (in our case, the line that we entered is blank), we perform one I/O action and when it isn't, the I/O action under the *else* is performed. That's why in an I/O *do* block, *if*s have to have a form of `if` *`condition`* `then` *`I/O action`* `else` *`I/O action`*`.`

Let's first take a look at what happens under the *else* clause. Because, we have to have exactly one I/O action after the *else*, we use a *do* block to glue together two I/O actions into one. You could also write that part out as:

```
else  do
            $
```

This makes it more explicit that the *do* block can be viewed as one I/O action, but it's uglier. Anyway, inside the *do* block, we call `reverseWords` on the line that we got from `getLine` and then print that out to the terminal. After that, we just perform `main`. It's called recursively and that's okay, because `main` is itself an I/O action. So in a sense, we go back to the start of the program.

Now what happens when `null line` holds true? What's after the *then* is performed in that case. If we look up, we'll see that it says `then return ()`. If you've done imperative languages like C, Java or Python, you're probably thinking that you know what this `return` does and chances are you've already skipped this really long paragraph. Well, here's the thing: **the `return` in Haskell is really nothing like the `return` in most other languages!** It has the same name, which confuses a lot of people, but in reality it's quite different. In imperative languages, `return` usually ends the execution of a method or subroutine and makes it

report some sort of value to whoever called it. In Haskell (in I/O actions specifically), it makes an I/O action out of a pure value. If you think about the box analogy from before, it takes a value and wraps it up in a box. The resulting I/O action doesn't actually do anything, it just has that value encapsulated as its result. So in an I/O context, `return "haha"` will have a type of `IO String`. What's the point of just transforming a pure value into an I/O action that doesn't do anything? Why taint our program with `IO` more than it has to be? Well, we needed some I/O action to carry out in the case of an empty input line. That's why we just made a bogus I/O action that doesn't do anything by writing `return ()`.

Using `return` doesn't cause the I/O *do* block to end in execution or anything like that. For instance, this program will quite happily carry out all the way to the last line:

```
= do

    "HAHAHA"
  <-
    "BLAH BLAH BLAH"
    4
```

All these `return`s do is that they make I/O actions that don't really do anything except have an encapsulated result and that result is thrown away because it isn't bound to a name. We can use `return` in combination with `<-` to bind stuff to names.

```
= do
 <-       "hell"
 <-       "yeah!"
      $   ++ " " ++
```

So you see, `return` is sort of the opposite to `<-`. While `return` takes a value and wraps it up in a box, `<-` takes a box (and performs it) and takes the value out of it, binding it to a name. But doing this is kind of redundant, especially since you can use *let* bindings in *do* blocks to bind to names, like so:

```
 = do
let   = "hell"
      = "yeah"
        $   ++ " " ++
```

When dealing with I/O *do* blocks, we mostly use `return` either because we need to create an I/O action that doesn't do anything or because we don't want the I/O action that's made up from a *do* block to have the result value of its last action, but we want it to have a different result value, so we use `return` to make an I/O action that always has our desired result contained and we put it at the end.

A *do* block can also have just one I/O action. In that case, it's the same as just writing the I/O action. Some people would prefer writing `then do return ()` in this case because the *else* also has a *do*.

Before we move on to files, let's take a look at some functions that are useful when dealing with I/O.

is much like `putStrLn` in that it takes a string as a parameter and returns an I/O action that will print that string to the terminal, only `putStr` doesn't jump into a new line after printing out the string while `putStrLn` does.

```
= do          "Hey, "
              "I'm "
                "Andy!"
```

Its type signature is `putStr :: String -> IO ()`, so the result encapsulated within the resulting I/O action is the unit. A dud value, so it doesn't make sense to bind it.

takes a character and returns an I/O action that will print it out to the terminal.

```
= do           't'
               'e'
               'h'
```

`putStr` is actually defined recursively with the help of `putChar`. The edge condition of `putStr` is the empty string, so if we're printing an empty string, just return an I/O action that does nothing by using `return ()`. If it's not empty, then print the first character of the string by doing `putChar` and then print of them using `putStr`.

```
::        ->
   =
       = do
```

See how we can use recursion in I/O, just like we can use it in pure code. Just like in pure code, we define the edge case and then think what the result actually is. It's an action that first outputs the first character and then outputs the rest of the string.

takes a value of any type that's an instance of `Show` (meaning that we know how to represent it as a string), calls `show` with that value to stringify it and then outputs that string to the terminal. Basically, it's just `putStrLn . show`. It first runs `show` on a value and then feeds that to `putStrLn`, which returns an I/O action that will print out our value.

```
= do
              2
              "haha"
              3.2
               3 4 3
```

```
2
"haha"
3.2
 3 4 3
```

As you can see, it's a very handy function. Remember how we talked about how I/O actions are performed only when they fall into `main` or when we try to evaluate them in the GHCI prompt? When we type out a value (like `3` or `[1,2,3]`) and press the return key, GHCI actually uses `print` on that value to display it on our terminal!

```
ghci> 3
3
ghci>       3
3
ghci>         "!"   "hey" "ho" "woo"
 "hey!" "ho!" "woo!"
ghci>               "!"   "hey" "ho" "woo"
 "hey!" "ho!" "woo!"
```

When we want to print out strings, we usually use `putStrLn` because we don't want the quotes around them, but for printing out values of other types to the terminal, `print` is used the most.

is an I/O action that reads a character from the input. Thus, its type signature is `getChar :: IO Char`, because the result contained within the I/O action is a `Char`. Note that due to buffering, reading of the characters won't actually happen until the user mashes the return key.

```
 = do
   <-
if   /= ' '
    then do

    else
```

This program looks like it should read a character and then check if it's a space. If it is, halt execution and if it isn't, print it to the terminal and then do the same thing all over again. Well, it kind of does, only not in the way you might expect. Check this out:

The second line is the input. We input `hello sir` and then press return. Due to buffering, the execution of the program will begin only when after we've hit return and not after every inputted character. But once we press return, it acts on what we've been putting in so far. Try playing with this program to get a feel for it!

The function is found in `Control.Monad` (to get access to it, do `import Control.Monad`). It's interesting because in a *do* block it looks like a control flow statement, but it's actually a normal function. It takes a boolean value and an I/O action if that boolean value is `True`, it returns the same I/O action that we supplied to it. However, if it's `False`, it returns the `return ()`, action, so an I/O action that doesn't do anything. Here's how we could rewrite the previous piece of code with which we demonstrated `getChar` by using `when`:

```
import

     = do
       <-
            /= ' '  $ do
```

So as you can see, it's useful for encapsulating the `if` *something* `then do` *some I/O action* `else return ()` pattern.

takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other. The result contained in that I/O action will be a list of the results of all the I/O actions that were performed. Its type signature is `sequence :: [IO a] -> IO [a]`. Doing this:

```
 = do
   <-
   <-
   <-
```

Is exactly the same as doing this:.

```
 = do
   <-
```

So `sequence [getLine, getLine, getLine]` makes an I/O action that will perform `getLine` three times. If we bind that action to a name, the result is a list of all the results, so in our case, a list of three things that the user entered at the prompt.

A common pattern with `sequence` is when we map functions like `print` or `putStrLn` over lists. Doing `map print [1,2,3,4]` won't create an I/O action. It will create a list of I/O actions, because that's like writing `[print 1, print 2, print 3, print 4]`. If we want to transform that list of I/O actions into an I/O action, we have to sequence it.

```
ghci>                    1 2 3 4 5
1
2
3
4
5
```

What's with the `[(),(),(),(),()]` at the end? Well, when we evaluate an I/O action in GHCI, it's performed and then its result is printed out, unless that result is `()`, in which case it's not printed out. That's why evaluating `putStrLn "hehe"` in GHCI just prints out `hehe` (because the contained result in `putStrLn "hehe"` is `()`). But when we do `getLine` in GHCI, the result of that I/O action is printed out, because `getLine` has a type of `IO String`.

Because the mapping a function that returns an I/O action over a list and then sequencing it so common, the utility functions and were introduced. `mapM` takes a function and a list, maps the function over the list and then sequences it. `mapM_` does the same, only it throws away the result later. We usually use `mapM_` when we don't care what result our sequenced I/O actions have.

```
ghci>             1 2 3
1
2
3

ghci>              1 2 3
1
2
3
```

takes an I/O action and returns an I/O action that just repeats the I/O action it got forever. It's located in `Control.Monad`. This little program will indefinitely ask the user for some input and spit it back to him, CAPSLOCKED:

```
import
import

     =         $ do
          "Give me some input: "
      <-
            $
```

(located in `Control.Monad`) is like `mapM`, only that it has its parameters switched around. The first parameter is the list and the second one is the function to map over that list, which is then sequenced. Why is that useful? Well, with some creative use of lambdas and *do* notation, we can do stuff like this:

```
import

     = do
          <-       1 2 3 4      -> do
               $ "Which color do you associate with the number " ++       ++ "?"
            <-

            "The colors that you associate with 1, 2, 3 and 4 are: "
```

The `(\a -> do ... )` is a function that takes a number and returns an I/O action. We have to surround it with parentheses, otherwise the lambda thinks the last two I/O actions belong to it. Notice that we do `return color` in the inside *do* block. We do that so that the I/O action which the *do* block defines has the result of our color contained within it. We actually didn't have to do that, because `getLine` already has that contained within it. Doing `color <- getLine` and then `return color` is just unpacking the result from `getLine` and then repackaging it again, so it's the same as just doing `getLine`. The `forM` (called with its two parameters) produces an I/O action, whose result we bind to `colors`. `colors` is just a normal list that holds strings. At the end, we print out all those colors by doing `mapM putStrLn colors`.

You can think of `forM` as meaning: make an I/O action for every element in this list. What each I/O action will do can depend on the element that was used to make the action. Finally, perform those actions and bind their results to something. We don't have to bind it, we can also just throw it away.

We could have actually done that without `forM`, only with `forM` it's more readable. Normally we write `forM` when we want to map and sequence some actions that we define there on the spot using *do* notation. In the same vein, we could have replaced the last line with `forM colors putStrLn`.

In this section, we learned the basics of input and output. We also found out what I/O actions are, how they enable us to do input and output and when they are actually performed. To reiterate, I/O actions are values much like any other value in Haskell. We can pass them as parameters to functions and functions can return I/O actions as results. What's special about them is that if they fall into the `main` function (or are the result in a GHCI line), they are performed. And that's when they get to write stuff on your screen or play Yakety Sax through your speakers. Each I/O action can also encapsulate a result with which it tells you what it got from the real world.

Don't think of a function like `putStrLn` as a function that takes a string and prints it to the screen. Think of it as a function that takes a string and returns an I/O action. That I/O action will, when performed, print beautiful poetry to your terminal.

Files and streams

`getChar` is an I/O action that reads a single character from the terminal. `getLine` is an I/O action that reads a line from the terminal. These two are pretty straightforward and most programming languages have some functions or statements that are parallel to them. But now, let's meet . `getContents` is an I/O action that reads everything from the standard input until it encounters an end-of-file character. Its type is `getContents :: IO String`. What's cool about `getContents` is that it does lazy I/O. When we do `foo <- getContents`, it doesn't

read all of the input at once, store it in memory and then bind it to `foo`. No, it's lazy! It'll say: *"Yeah yeah, I'll read the input from the terminal later as we go along, when you really need it!"*.

`getContents` is really useful when we're piping the output from one program into the input of our program. In case you don't know how piping works in unix-y systems, here's a quick primer. Let's make a text file that contains the following little haiku:

Yeah, the haiku sucks, what of it? If anyone knows of any good haiku tutorials, let me know.

Now, recall the little program we wrote when we were introducing the `forever` function. It prompted the user for a line, returned it to him in CAPSLOCK and then did that all over again, indefinitely. Just so you don't have to scroll all the way back, here it is again:

```
import
import

     =          $ do
           "Give me some input: "
       <-
             $
```

We'll save that program as `capslocker.hs` or something and compile it. And then, we're going to use a unix pipe to feed our text file directly to our little program. We're going to use the help of the GNU *cat* program, which prints out a file that's given to it as an argument. Check it out, booyaka!

As you can see, piping the output of one program (in our case that was *cat*) to the input of another (*capslocker*) is done with the `|` character. What we've done is pretty much equivalent to just running *capslocker*, typing our haiku at the terminal and then issuing an end-of-file character (that's usually done by pressing Ctrl-D). It's like running *cat haiku.txt* and saying: “Wait, don't print this out to the terminal, tell it to *capslocker* instead!”.

So what we're essentially doing with that use of `forever` is taking the input and transforming it into some output. That's why we can use `getContents` to make our program even shorter and better:

```
import

     = do
           <-
```

We run the `getContents` I/O action and name the string it produces `contents`. Then, we map `toUpper` over that string and print that to the terminal. Keep in mind that because strings are basically lists, which are lazy, and `getContents` is I/O lazy, it won't try

to read the whole content at once and store it into memory before printing out the capslocked version. Rather, it will print out the capslocked version as it reads it, because it will only read a line from the input when it really needs to.

Cool, it works. What if we just run *capslocker* and try to type in the lines ourselves?

We got out of that by pressing Ctrl-D. Pretty nice! As you can see, it prints out our capslocked input back to us line by line. When the result of `getContents` is bound to `contents`, it's not represented in memory as a real string, but more like a promise that it will produce the string eventually. When we map `toUpper` over `contents`, that's also a promise to map that function over the eventual contents. And finally when `putStr` happens, it says to the previous promise: *"Hey, I need a capslocked line!"*. It doesn't have any lines yet, so it says to `contents`: *"Hey, how about actually getting a line from the terminal?"*. So that's when `getContents` actually reads from the terminal and gives a line to the code that asked it to produce something tangible. That code then maps `toUpper` over that line and gives it to `putStr`, which prints it. And then, `putStr` says: *"Hey, I need the next line, come on!"* and this repeats until there's no more input, which is signified by an end-of-file character.

Let's make program that takes some input and prints out only those lines that are shorter than 10 characters. Observe:

```
 = do
        <-


          ::        ->
               =
let          =
             =              ->            < 10
          =
in
```

We've made our I/O part of the program as short as possible. Because our program is supposed to take some input and print out some output based on the input, we can implement it by reading the input contents, running a function on them and then printing out what the function gave back.

The `shortLinesOnly` function works like this: it takes a string, like `"short\nlooooooooooooooong\nshort again"`. That string has three lines, two of them are short and the middle one is long. It runs the `lines` function on that string, which converts it to `["short", "looooooooooooooong", "short again"]`, which we then bind to the name `allLines`. That list of string is then filtered so that only those lines that are shorter than 10 characters remain in the list, producing `["short", "short again"]`. And finally, `unlines` joins that list into a single newline delimited string, giving `"short\nshort again"`. Let's give it a go.

We pipe the contents of *shortlines.txt* into the output of *shortlinesonly* and as the output, we only get the short lines.

This pattern of getting some string from the input, transforming it with a function and then outputting that is so common that there exists a function which makes that even easier, called . `interact` takes a function of type `String -> String` as a parameter and returns an I/O action that will take some input, run that function on it and then print out the function's result. Let's modify our program to use that.

```
 =

          ::       ->
              =
let         =
              =              ->            < 10
          =
in
```

Just to show that this can be achieved in much less code (even though it will be less readable) and to demonstrate our function composition skill, we're going to rework that a bit further.

```
 =         $         .          10  .        .
```

Wow, we actually reduced that to just one line, which is pretty cool!

`interact` can be used to make programs that are piped some contents into them and then dump some result out or it can be used to make programs that appear to take a line of input from the user, give back some result based on that line and then take another line and so on. There isn't actually a real distinction between the two, it just depends on how the user is supposed to use them.

Let's make a program that continuously reads a line and then tells us if the line is a palindrome or not. We could just use `getLine` to read a line, tell the user if it's a palindrome and then run `main` all over again. But it's simpler if we use `interact`. When using `interact`, think about what you need to do to transform some input into the desired output. In our case, we have to replace each line of the input with either `"palindrome"` or `"not a palindrome"`. So we have to write a function that transforms something like `"elephant\nABCBA\nwhatever"` into `"not a palindrome\npalindrome\nnot a palindrome"`. Let's do this!

```
                  =                  -
> if              then "palindrome" else "not a palindrome"
  where                =    ==
```

Let's write this in point-free.

```
                =         .          -
> if              then "palindrome" else "not a palindrome"  .
  where                =    ==
```

Pretty straightforward. First it turns something like `"elephant\nABCBA\nwhatever"` into `["elephant", "ABCBA", "whatever"]` and then it maps that lambda over it, giving `["not a palindrome", "palindrome", "not a palindrome"]` and then `unlines` joins that list into a single, newline delimited string. Now we can do

=

Let's test this out:

Even though we made a program that transforms one big string of input into another, it acts like we made a program that does it line by line. That's because Haskell is lazy and it wants to print the first line of the result string, but it can't because it doesn't have the first line of the input yet. So as soon as we give it the first line of input, it prints the first line of the output. We get out of the program by issuing an end-of-line character.

We can also use this program by just piping a file into it. Let's say we have this file:

and we save it as `words.txt`. This is what we get by piping it into our program:

Again, we get the same output as if we had run our program and put in the words ourselves at the standard input. We just don't see the input that `palindromes.hs` because the input came from the file and not from us typing the words in.

So now you probably see how lazy I/O works and how we can use it to our advantage. You can just think in terms of what the output is supposed to be for some given input and write a function to do that transformation. In lazy I/O, nothing is eaten from the input until it absolutely has to be because what we want to print right now depends on that input.

So far, we've worked with I/O by printing out stuff to the terminal and reading from it. But what about reading and writing files? Well, in a way, we've already been doing that. One way to think about reading from the terminal is to imagine that it's like reading from a (somewhat special) file. Same goes for writing to the terminal, it's kind of like writing to a file. We can call these two files `stdout` and `stdin`, meaning *standard output* and *standard input*, respectively. Keeping that in mind, we'll see that writing to and reading from files is very much like writing to the standard output and reading from the standard input.

We'll start off with a really simple program that opens a file called *girlfriend.txt*, which contains a verse from Avril Lavigne's #1 hit *Girlfriend*, and just prints out out to the terminal. Here's *girlfriend.txt*:

And here's our program:

```
import

    = do
          <-          "girlfriend.txt"
            <-
```

Running it, we get the expected result:

Let's go over this line by line. The first line is just four exclamations, to get our attention. In the second line, Avril tells us that she doesn't like our current romantic partner. The third line serves to emphasize that disapproval, whereas the fourth line suggests we should seek out a new girlfriend.

Let's also go over the program line by line! Our program is several I/O actions glued together with a *do* block. In the first line of the *do* block, we notice a new function called . This is its type signature: `openFile :: FilePath -> IOMode -> IO Handle`. If you read that out loud, it states: `openFile` takes a file path and an `IOMode` and returns an I/O action that will open a file and have the file's associated handle encapsulated as its result.

`FilePath` is just a type synonym for `String`, simply defined as:

```
type          =
```

`IOMode` is a type that's defined like this:

```
data       =          |          |          |
```

Just like our type that represents the seven possible values for the days of the week, this type is an enumeration that represents what we want to do with our opened file. Very simple. Just note that this type is `IOMode` and not `IO Mode`. `IO Mode` would be the type of an I/O action that has a value of some type `Mode` as its result, but `IOMode` is just a simple enumeration.

Finally, it returns an I/O action that will open the specified file in the specified mode. If we bind that action to something we get a `Handle`. A value of type `Handle` represents where our file is. We'll use that handle so we know which file to read from. It would be stupid to read a file but not bind that read to a handle because we wouldn't be able to do anything with the file. So in our case, we bound the handle to `handle`.

In the next line, we see a function called . It takes a `Handle`, so it knows which file to get the contents from and returns an `IO String` — an I/O action that holds as its result the contents of the file. This function is pretty much like `getContents`. The only difference is that `getContents` will automatically read from the standard input (that is from the terminal), whereas `hGetContents` takes a file handle which tells it which file to read from. In all other respects, they work the same. And just like `getContents`, `hGetContents` won't attempt to read the file at once and store it in memory, but it will read it as needed. That's really cool because we can treat `contents` as the whole contents of the file, but it's not really loaded in memory. So if this

were a really huge file, doing `hGetContents` wouldn't choke up our memory, but it would read only what it needed to from the file, when it needed to.

Note the difference between the handle used to identify a file and the contents of the file, bound in our program to `handle` and `contents`. The handle is just something by which we know what our file is. If you imagine your whole file system to be a really big book and each file is a chapter in the book, the handle is a bookmark that shows where you're currently reading (or writing) a chapter, whereas the contents are the actual chapter.

With `putStr contents` we just print the contents out to the standard output and then we do , which takes a handle and returns an I/O action that closes the file. You have to close the file yourself after opening it with `openFile`!

Another way of doing what we just did is to use the function, which has a type signature of `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`. It takes a path to a file, an `IOMode` and then it takes a function that takes a handle and returns some I/O action. What it returns is an I/O action that will open that file, do something we want with the file and then close it. The result encapsulated in the final I/O action that's returned is the same as the result of the I/O action that the function we give it returns. This might sound a bit complicated, but it's really simple, especially with lambdas, here's our previous example rewritten to use `withFile`:

```
import

     = do
          "girlfriend.txt"                    -> do
              <-
```

As you can see, it's very similar to the previous piece of code. `(\handle -> ... )` is the function that takes a handle and returns an I/O action and it's usually done like this, with a lambda. The reason it has to take a function that returns an I/O action instead of just taking an I/O action to do and then close the file is because the I/O action that we'd pass to it wouldn't know on which file to operate. This way, `withFile` opens the file and then passes the handle to the function we gave it. It gets an I/O action back from that function and then makes an I/O action that's just like it, only it closes the file afterwards. Here's how we can make our own `withFile` function:

```
::          ->        ->        ->       ->
            = do
 <-
 <-
```

We know the result will be an I/O action so we can just start off with a *do*. First we open the file and get a handle from it. Then, we apply `handle` to our function to get back the I/O action that does all the work. We bind that action to `result`, close the handle and then do `return result`. By returning the result encapsulated in the I/O action that we got from `f`, we make it so that our I/O action encapsulates the same result as the one we got from `f handle`. So if `f handle` returns an action that will read a number of lines from the standard input and write them to a file and have as its result encapsulated the number of lines it read, if we used that with `withFile'`, the resulting I/O action would also have as its result the number of lines read.

Just like we have `hGetContents` that works like `getContents` but for a specific file, there's also , , , , etc. They work just like their counterparts without the *h*, only they take a handle as a parameter and operate on that specific file instead of operating on standard input or standard output. Example: `putStrLn` is a function that takes a string and returns an I/O action that will

print out that string to the terminal and a newline after it. `hPutStrLn` takes a handle and a string and returns an I/O action that will write that string to the file associated with the handle and then put a newline after it. In the same vein, `hGetLine` takes a handle and returns an I/O action that reads a line from its file.

Loading files and then treating their contents as strings is so common that we have these three nice little functions to make our work even easier:

has a type signature of `readFile :: FilePath -> IO String`. Remember, `FilePath` is just a fancy name for `String`. `readFile` takes a path to a file and returns an I/O action that will read that file (lazily, of course) and bind its contents to something as a string. It's usually more handy than doing `openFile` and binding it to a handle and then doing `hGetContents`. Here's how we could have written our previous example with `readFile`:

```
import

     = do
          <-          "girlfriend.txt"
```

Because we don't get a handle with which to identify our file, we can't close it manually, so Haskell does that for us when we use `readFile`.

has a type of `writeFile :: FilePath -> String -> IO ()`. It takes a path to a file and a string to write to that file and returns an I/O action that will do the writing. If such a file already exists, it will be stomped down to zero length before being written on. Here's how to turn *girlfriend.txt* into a CAPSLOCKED version and write it to *girlfriendcaps.txt*:

```
import
import

     = do
          <-          "girlfriend.txt"
           "girlfriendcaps.txt"
```

has a type signature that's just like `writeFile`, only `appendFile` doesn't truncate the file to zero length if it already exists but it appends stuff to it.

Let's say we have a file *todo.txt* that has one task per line that we have to do. Now let's make a program that takes a line from the standard input and adds that to our to-do list.

```
import

     = do
          <-
            "todo.txt"           ++ "\n"
```

We needed to add the `"\n"` to the end of each line because `getLine` doesn't give us a newline character at the end.

Ooh, one more thing. We talked about how doing `contents <- hGetContents handle` doesn't cause the whole file to be read at once and stored in-memory. It's I/O lazy, so doing this:

```
= do
        "something.txt"                 -> do
            <-
```

is actually like connecting a pipe from the file to the output. Just like you can think of lists as streams, you can also think of files as streams. This will read one line at a time and print it out to the terminal as it goes along. So you may be asking, how wide is this pipe then? How often will the disk be accessed? Well, for text files, the default buffering is line-buffering usually. That means that the smallest part of the file to be read at once is one line. That's why in this case it actually reads a line, prints it to the output, reads the next line, prints it, etc. For binary files, the default buffering is usually block-buffering. That means that it will read the file chunk by chunk. The chunk size is some size that your operating system thinks is cool.

You can control how exactly buffering is done by using the `hSetBuffering` function. It takes a handle and a `BufferMode` and returns an I/O action that sets the buffering. `BufferMode` is a simple enumeration data type and the possible values it can hold are: `NoBuffering`, `LineBuffering` or `BlockBuffering (Maybe Int)`. The `Maybe Int` is for how big the chunk should be, in bytes. If it's `Nothing`, then the operating system determines the chunk size. `NoBuffering` means that it will be read one character at a time. `NoBuffering` usually sucks as a buffering mode because it has to access the disk so much.

Here's our previous piece of code, only it doesn't read it line by line but reads the whole file in chunks of 2048 bytes.

```
= do
        "something.txt"                 -> do
                    $                       2048
            <-
```

Reading files in bigger chunks can help if we want to minimize disk access or when our file is actually a slow network resource.

We can also use , which is a function that takes a handle and returns an I/O action that will flush the buffer of the file associated with the handle. When we're doing line-buffering, the buffer is flushed after every line. When we're doing block-buffering, it's after we've read a chunk. It's also flushed after closing a handle. That means that when we've reached a newline character, the reading (or writing) mechanism reports all the data so far. But we can use `hFlush` to force that reporting of data that has been read so far. After flushing, the data is available to other programs that are running at the same time.

Think of reading a block-buffered file like this: your toilet bowl is set to flush itself after it has one gallon of water inside it. So you start pouring in water and once the gallon mark is reached, that water is automatically flushed and the data in the water that you've poured in so far is read. But you can flush the toilet manually too by pressing the button on the toilet. This makes the toilet flush and all the water (data) inside the toilet is read. In case you haven't noticed, flushing the toilet manually is a metaphor for `hFlush`. This is not a very great analogy by programming analogy standards, but I wanted a real world object that can be flushed for the punchline.

We already made a program to add a new item to our to-do list in *todo.txt*, now let's make a program to remove an item. I'll just paste the code and then we'll go over the program together so you see that it's really easy. We'll be using a few new functions from `System.Directory` and one new function from `System.IO`, but they'll all be explained.

Anyway, here's the program for removing an item from *todo.txt*:

```
import
import
import

     = do
          <-          "todo.txt"
                        <-              "." "temp"
            <-
    let            =
                     =                    ->        ++ " - " ++        0
            "These are your TO-DO items:"
          $
            "Which one do you want to delete?"
              <-
    let        =
                 =
                   $


              "todo.txt"
                       "todo.txt"
```

At first, we just open *todo.txt* in read mode and bind its handle to `handle`.

Next up, we use a function that we haven't met before which is from `System.IO` — . Its name is pretty self-explanatory. It takes a path to a temporary directory and a template name for a file and opens a temporary file. We used `"."` for the temporary directory, because `.` denotes the current directory on just about any OS. We used `"temp"` as the template name for the temporary file, which means that the temporary file will be named *temp* plus some random characters. It returns an I/O action that makes the temporary file and the result in that I/O action is a pair of values: the name of the temporary file and a handle. We could just open a normal file called *todo2.txt* or something like that but it's better practice to use `openTempFile` so you know you're probably not overwriting anything.

The reason we used `getCurrentDirectory` to get the current directory and then passed it to `openTempFile` instead of just passing `"."` to `openTempFile` is because `.` refers to the current directory on unix-like system and Windows

Next up, we bind the contents of *todo.txt* to `contents`. Then, split that string into a list of strings, each string one line. So `todoTasks` is now something like `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]`. We zip the numbers from 0 onwards and that list with a function that takes a number, like 3, and a string, like `"hey"` and returns `"3 - hey"`, so `numberedTasks` is `["0 - Iron the dishes", "1 - Dust the dog" ...`. We join that list of strings into a single newline delimited string with `unlines` and print that string out to the terminal. Note that instead of doing that, we could have also done `mapM putStrLn numberedTasks`

We ask the user which one they want to delete and wait for them to enter a number. Let's say they want to delete number `1`, which is `Dust the dog`, so they punch in `1`. `numberString` is now `"1"` and because we want a number, not a string, we run `read` on that to get `1` and bind that to `number`.

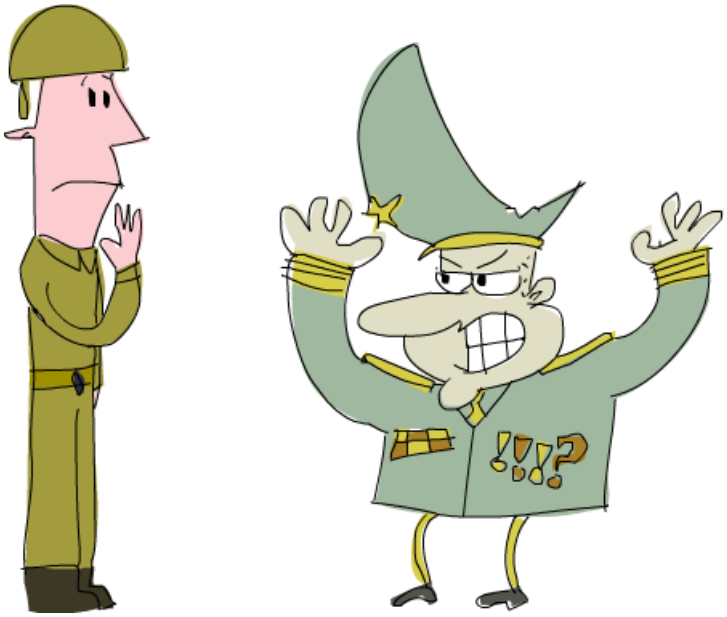
Remember the `delete` and `!!` functions from `Data.List`. `!!` returns an element from a list with some index and `delete` deletes the first occurence of an element in a list and returns a new list without that occurence. `(todoTasks !! number)` (number is now `1`) returns `"Dust the dog"`. We bind `todoTasks` without the first occurence of `"Dust the dog"` to `newTodoItems` and then join that into a single string with `unlines` before writing it to the temporary file that we opened. The old file is now unchanged and the temporary file contains all the lines that the old one does, except the one we deleted.

After that we close both the original and the temporary files and then we remove the original one with , which, as you can see, takes a path to a file and deletes it. After deleting the old *todo.txt*, we use to rename the temporary

file to *todo.txt*. Be careful, `removeFile` and `renameFile` (which are both in `System.Directory` by the way) take file paths as their parameters, not handles.

And that's that! We could have done this in even fewer lines, but we were very careful not to overwrite any existing files and politely asked the operating system to tell us where we can put our temporary file. Let's give this a go!

Command line arguments



Dealing with command line arguments is pretty much a necessity if you want to make a script or application that runs on a terminal. Luckily, Haskell's standard library has a nice way of getting command line arguments of a program.

In the previous section, we made one program for adding a to-do item to our to-do list and one program for removing an item. There are two problems with the approach we took. The first one is that we just hardcoded the name of our to-do file in our code. We just decided that the file will be named *todo.txt* and that the user will never have a need for managing several to-do lists.

One way to solve that is to always ask the user which file they want to use as their to-do list. We used that approach when we wanted to know which item the user wants to delete. It works, but it's not so good, because it requires the user to run the program, wait for the program to ask something and then tell that to the program. That's called an interactive program and the difficult bit with interactive command line programs is this — what if you want to automate the execution of that program, like with a batch script? It's harder to make a batch script that interacts with a program than a batch script that just calls one program or several of them.

That's why it's sometimes better to have the user tell the program what they want when they run the program, instead of having the program ask the user once it's run. And what better way to have the user tell the program what they want it to do when they run it than via command line arguments!

The `System.Environment` module has two cool I/O actions. One is , which has a type of `getArgs :: IO [String]` and is an I/O action that will get the arguments that the program was run with and have as its contained result a list with the arguments. has a type of `getProgName :: IO String` and is an I/O action that contains the program name.

Here's a small program that demonstrates how these two work:

```
import
import

     = do
        <-
            <-
            "The arguments are:"

            "The program name is:"
```

We bind `getArgs` and `progName` to `args` and `progName`. We say `The arguments are:` and then for every argument in `args`, we do `putStrLn`. Finally, we also print out the program name. Let's compile this as `arg-test`.

Nice. Armed with this knowledge you could create some cool command line apps. In fact, let's go ahead and make one. In the previous section, we made a separate program for adding tasks and a separate program for deleting them. Now, we're going to join that into one program, what it does will depend on the command line arguments. We're also going to make it so it can operate on different files, not just *todo.txt*.

We'll call it simply *todo* and it'll be able to do (haha!) three different things:

- View tasks
- Add tasks
- Delete tasks

We're not going to concern ourselves with possible bad input too much right now.

Our program will be made so that if we want to add the task `Find the magic sword of power` to the file *todo.txt*, we have to punch in `todo add todo.txt "Find the magic sword of power"` in our terminal. To view the tasks we'll just do `todo view todo.txt` and to remove the task with the index of 2, we'll do `todo remove todo.txt 2`.

We'll start by making a dispatch association list. It's going to be a simple association list that has command line arguments as keys and functions as their corresponding values. All these functions will be of type `[String] -> IO ()`. They're going to take the argument list as a parameter and return an I/O action that does the viewing, adding, deleting, etc.

```
import
import
import
import

        ::                    ->
        =     "add"
              "view"
              "remove"
```

We have yet to define `main`, `add`, `view` and `remove`, so let's start with `main`:

```
 = do
              <-
let             =
```

First, we get the arguments and bind them to `(command:args)`. If you remember your pattern matching, this means that the first argument will get bound to `command` and the rest of them will get bound to `args`. If we call our program like `todo add todo.txt "Spank the monkey"`, `command` will be `"add"` and `args` will be `["todo.xt", "Spank the monkey"]`.

In the next line, we look up our command in the dispatch list. Because `"add"` points to `add`, we get `Just add` as a result. We use pattern matching again to extract our function out of the `Maybe`. What happens if our command isn't in the dispatch list? Well then the lookup will return `Nothing`, but we said we won't concern ourselves with failing gracefully too much, so the pattern matching will fail and our program will throw a fit.

Finally, we call our `action` function with the rest of the argument list. That will return an I/O action that either adds an item, displays a list of items or deletes an item and because that action is part of the `main` *do* block, it will get performed. If we follow our concrete example so far and our `action` function is `add`, it will get called with `args` (so `["todo.txt", "Spank the monkey"]`) and return an I/O action that adds `Spank the monkey` to *todo.txt*.

Great! All that's left now is to implement `add`, `view` and `remove`. Let's start with `add`:

```
::          ->
                =                                ++ "\n"
```

If we call our program like `todo add todo.txt "Spank the monkey"`, the `"add"` will get bound to `command` in the first pattern match in the `main` block, whereas `["todo.txt", "Spank the monkey"]` will get passed to the function that we get from the dispatch list. So, because we're not dealing with bad input right now, we just pattern match against a list with those two elements right away and return an I/O action that appends that line to the end of the file, along with a newline character.

Next, let's implement the list viewing functionality. If we want to view the items in a file, we do `todo view todo.txt`. So in the first pattern match, `command` will be `"view"` and `args` will be `["todo.txt"]`.

```
 ::          ->
           = do
         <-
let            =
                =                  ->        ++ " - " ++        0
       $
```

We already did pretty much the same thing in the program that only deleted tasks when we were displaying the tasks so that the user can choose one for deletion, only here we just display the tasks.

And finally, we're going to implement `remove`. It's going to be very similar to the program that only deleted the tasks, so if you don't understand how deleting an item here works, check out the explanation under that program. The main difference is that we're not hardcoding *todo.txt* but getting it as an argument. We're also not prompting the user for the task number to delete, we're getting it as an argument.

```
  ::          ->
                        = do
     <-
                  <-              "." "temp"
```

```
            <-
    let       =
                =
                  =
                    $
```

We opened up the file based on `fileName` and opened a temporary file, deleted the line with the index that the user wants to delete, wrote that to the temporary file, removed the original file and renamed the temporary file back to `fileName`.

Here's the whole program at once, in all its glory!

```
import
import
import
import

        ::                    ->
        =     "add"
              "view"
              "remove"


    = do
              <-
    let           =

    ::          ->
                  =                          ++ "\n"

     ::          ->
              = do
          <-
    let           =
                =                  ->       ++ " - " ++       0
        $

      ::          ->
                        = do
          <-
                    <-             "." "temp"
          <-
    let       =
                =
                  =
                    $
```

To summarize our solution: we made a dispatch association that maps from commands to functions that take some command line arguments and return an I/O action. We see what the command is and based on that we get the appropriate function from the dispatch list. We call that function with the rest of the command line arguments to get back an I/O action that will do the appropriate thing and then just perform that action!

In other languages, we might have implemented this with a big switch case statement or whatever, but using higher order functions allows us to just tell the dispatch list to give us the appropriate function and then tell that function to give us an I/O action for some command line arguments.

Let's try our app out!

Another cool thing about this is that it's easy to add extra functionality. Just add an entry in the dispatch association list and implement the corresponding function and you're laughing! As an exercise, you can try implementing a `bump` function that will take a file and a task number and return an I/O action that bumps that task to the top of the to-do list.

You could make this program fail a bit more gracefully in case of bad input (for example, if someone runs `todo UP YOURS HAHAHAHA`) by making an I/O action that just reports there has been an error (say, `errorExit :: IO ()`) and then check for possible erronous input and if there is erronous input, perform the error reporting I/O action. Another way is to use exceptions, which we will meet soon.

Randomness

Many times while programming, you need to get some random data. Maybe you're making a game where a die needs to be thrown or you need to generate some test data to test out your program. There are a lot of uses for random data when programming. Well, actually, pseudo-random, because we all know that the only true source of randomness is a monkey on a unicycle with a cheese in one hand and its butt in the other. In this section, we'll take a look at how to make Haskell generate seemingly random data.

In most other programming languages, you have functions that give you back some random number. Each time you call that function, you get back a (hopefully) different random number. How about Haskell? Well, remember, Haskell is a pure functional language. What that means is that it has referential transparency. What THAT means is that a function, if given the same parameters twice, must produce the same result twice. That's really cool because it allows us to reason differently about programs and it enables us to defer evaluation until we really need it. If I call a function, I can be sure that it won't do any funny stuff before giving me the results. All that matters are its results. However, this makes it a bit tricky for getting random numbers. If I have a function like this:

```
::          =>
= 4
```

It's not very useful as a random number function because it will always return `4`, even though I can assure you that the 4 is completely random, because I used a die to determine it.

How do other languages make seemingly random numbers? Well, they take various info from your computer, like the current time, how much and where you moved your mouse and what kind of noises you made behind your computer and based on that,

give a number that looks really random. The combination of those factors (that randomness) is probably different in any given moment in time, so you get a different random number.

Ah. So in Haskell, we can make a random number then if we make a function that takes as its parameter that randomness and based on that returns some number (or other data type).

Enter the `System.Random` module. It has all the functions that satisfy our need for randomness. Let's just dive into one of the functions it exports then, namely . Here's its type: `random :: (RandomGen g, Random a) => g -> (a, g)`. Whoa! Some new typeclasses in this type declaration up in here! The typeclass is for types that can act as sources of randomness. The typeclass is for things that can take on random values. A boolean value can take on a random value, namely `True` or `False`. A number can also take up a plethora of different random values. Can a function take on a random value? I don't think so, probably not! If we try to translate the type declaration of `random` to English, we get something like: it takes a random generator (that's our source of randomness) and returns a random value and a new random generator. Why does it also return a new generator as well as a random value? Well, we'll see in a moment.

To use our `random` function, we have to get our hands on one of those random generators. The `System.Random` module exports a cool type, namely that is an instance of the `RandomGen` typeclass. We can either make a `StdGen` manually or we can tell the system to give us one based on a multitude of sort of random stuff.

To manually make a random generator, use the function. It has a type of `mkStdGen :: Int -> StdGen`. It takes an integer and based on that, gives us a random generator. Okay then, let's try using `random` and `mkStdGen` in tandem to get a (hardly random) number.

```
ghci>                  100
```

What's this? Ah, right, the `random` function can return a value of any type that's part of the `Random` typeclass, so we have to inform Haskell what kind of type we want. Also let's not forget that it returns a random value and a random generator in a pair.

```
ghci>                  100  ::
  1352021624 651872571 1655838864
```

Finally! A number that looks kind of random! The first component of the tuple is our number whereas the second component is a textual representation of our new random generator. What happens if we call `random` with the same random generator again?

```
ghci>                  100  ::
  1352021624 651872571 1655838864
```

Of course. The same result for the same parameters. So let's try giving it a different random generator as a parameter.

```
ghci>                  949494  ::
 539963926 466647808 1655838864
```

Alright, cool, great, a different number. We can use the type annotation to get different types back from that function.

```
ghci>                  949488  ::
 0.8938442 1597344447 1655838864
ghci>                  949488  ::
```

```
       1485632275 40692
ghci>                  949488  ::
 1691547873 1597344447 1655838864
```

Let's make a function that simulates tossing a coin three times. If `random` didn't return a new generator along with a random value, we'd have to make this function take three random generators as a parameter and then return coin tosses for each of them. But that sounds wrong because if one generator can make a random value of type `Int` (which can take on a load of different values), it should be able to make three coin tosses (which can take on precisely eight combinations). So this is where `random` returning a new generator along with a value really comes in handy.

We'll represent a coin with a simple `Bool`. `True` is tails, `False` is heads.

```
           ::        ->
              =
    let                    =
                             =
                             =
    in
```

We call `random` with the generator we got as a parameter to get a coin and a new generator. Then we call it again, only this time with our new generator, to get the second coin. We do the same for the third coin. Had we called it with the same generator every time, all the coins would have had the same value and we'd only be able to get `(False, False, False)` or `(True, True, True)` as a result.

```
ghci>                    21

ghci>                    22

ghci>                    943

ghci>                    944
```

Notice that we didn't have to do `random gen :: (Bool, StdGen)`. That's because we already specified that we want booleans in the type declaration of the function. That's why Haskell can infer that we want a boolean value in this case.

So what if we want to flip four coins? Or five? Well, there's a function called that takes a generator and returns an infinite sequence of values based on that generator.

```
ghci>      5 $                  11  ::
  1807975507 545074951  1015194702  1622477312  502893664
ghci>      5 $                  11  ::

ghci>      5 $                  11  ::
 7         2 0.62691015 0.26363158 0.12223756 0.38291094
```

Why doesn't `randoms` return a new generator as well as a list? We could implement the `randoms` function very easily like this:

```
          ::                          =>   ->
             = let                   =           in
```

A recursive definition. We get a random value and a new generator from the current generator and then make a list that has the value as its head and random numbers based on the new generator as its tail. Because we have to be able to potentially generate an infinite amount of numbers, we can't give the new random generator back.

We could make a function that generates a finite stream of numbers and a new generator like this:

```
            ::                                  =>    ->    ->
            0      =
                   =
    let                 =
                                =                    1
    in
```

Again, a recursive definition. We say that if we want 0 numbers, we just return an empty list and the generator that was given to us. For any other number of random values, we first get one random number and a new generator. That will be the head. Then we say that the tail will be *n - 1* numbers generated with the new generator. Then we return the head and the rest of the list joined and the final generator that we got from getting the *n - 1* random numbers.

What if we want a random value in some sort of range? All the random integers so far were outrageously big or small. What if we want to to throw a die? Well, we use for that purpose. It has a type of `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`, meaning that it's kind of like `random`, only it takes as its first parameter a pair of values that set the lower and upper bounds and the final value produced will be within those bounds.

```
ghci>          1 6             359353
 6 1494289578 40692
ghci>          1 6             35935335
 3 1250031057 40692
```

There's also , which produces a stream of random values within our defined ranges. Check this out:

```
ghci>      10 $           'a' 'z'            3   ::
"ndkxbvmomg"
```

Nice, looks like a super secret password or something.

You may be asking yourself, what does this section have to do with I/O anyway? We haven't done anything concerning I/O so far. Well, so far we've always made our random number generator manually by making it with some arbitrary integer. The problem is, if we do that in our real programs, they will always return the same random numbers, which is no good for us. That's why `System.Random` offers the I/O action, which has a type of `IO StdGen`. When your program starts, it asks the system for a good random number generator and stores that in a so called global generator. `getStdGen` fetches you that global random generator when you bind it to something.

Here's a simple program that generates a random string.

```
import

     = do
        <-
           $      20            'a' 'z'
```

Be careful though, just performing `getStdGen` twice will ask the system for the same global generator twice. If you do this:

```
import

     = do
```

```
        <-
            $       20            'a' 'z'
         <-
           $       20            'a' 'z'
```

you will get the same string printed out twice! One way to get two different strings of length 20 is to set up an infinite stream and then take the first 20 characters and print them out in one line and then take the second set of 20 characters and print them out in the second line. For this, we can use the `splitAt` function from `Data.List`, which splits a list at some index and returns a tuple that has the first part as the first component and the second part as the second component.

```
import
import

     = do
        <-
    let             =           'a' 'z'
                        =         20
                      =         20
```

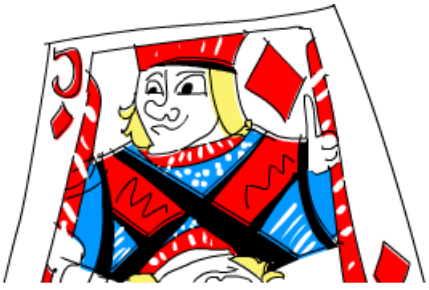
Another way is to use the action, which splits our current random generator into two generators. It updates the global random generator with one of them and encapsulates the other as its result.

```
import

     = do
        <-
            $       20            'a' 'z'
         <-
           $       20            'a' 'z'
```

Not only do we get a new random generator when we bind `newStdGen` to something, the global one gets updated as well, so if we do `getStdGen` again and bind it to something, we'll get a generator that's not the same as `gen`.

Here's a little program that will make the user guess which number it's thinking of.

```
import
import

     = do
        <-


          ::         ->
             = do
    let                    =          1 10      ::
          "Which number in the range from 1 to 10 am I thinking of? "
             <-
           $                    $ do
        let        =
        if             ==
            then           "You are correct!"
            else         $ "Sorry, it was " ++
```



We make a function `askForNumber`, which takes a random number generator and returns an I/O action that will prompt the user for a number and tell him if he guessed it right. In that function, we first generate a random number and a new generator based on the generator that we got as a parameter and call them `randNumber` and `newGen`. Let's say that the number generated was `7`. Then we tell the user to guess which number we're thinking of. We perform `getLine` and bind its result to

`numberString`. When the user enters `7`, `numberString` becomes `"7"`. Next, we use `when` to check if the string the user entered is an empty string. If it is, an empty I/O action of `return ()` is performed, which effectively ends the program. If it isn't, the action consisting of that *do* block right there gets performed. We use `read` on `numberString` to convert it to a number, so `number` is now `7`.

Excuse me! If the user gives us some input here that `read` can't read (like `"haha"`), our program will crash with an ugly error message. If you don't want your program to crash on erronous input, use , which returns an empty list when it fails to read a string. When it succeeds, it returns a singleton list with a tuple that has our desired value as one component and a string with what it didn't consume as the other.

We check if the number that we entered is equal to the one generated randomly and give the user the appropriate message. And then we call `askForNumber` recursively, only this time with the new generator that we got, which gives us an I/O action that's just like the one we performed, only it depends on a different generator and we perform it.

`main` consists of just getting a random generator from the system and calling `askForNumber` with it to get the initial action.

Here's our program in action!

Another way to make this same program is like this:

```
import
import

     = do
        <-
    let                 =          1 10      ::
          "Which number in the range from 1 to 10 am I thinking of? "
              <-
           $                    $ do
        let        =
        if             ==
            then           "You are correct!"
            else          $ "Sorry, it was " ++
```

It's very similar to the previous version, only instead of making a function that takes a generator and then calls itself recursively with the new updated generator, we do all the work in `main`. After telling the user whether they were correct in their guess or not, we update the global generator and then call `main` again. Both approaches are valid but I like the first one more since it does less stuff in `main` and also provides us with a function that we can reuse easily.

Bytestrings

Lists are a cool and useful data structure. So far, we've used them pretty much everywhere. There are a multitude of functions that operate on them and Haskell's laziness allows us to exchange the for and while loops of other languages for filtering and mapping over lists, because evaluation will

only happen once it really needs to, so things like infinite lists (and even infinite lists of infinite lists!) are no problem for us. That's why lists can also be used to represent streams, either when reading from the standard input or when reading from files. We can just open a file and read it as a string, even though it will only be accessed when the need arises.

However, processing files as strings has one drawback: it tends to be slow. As you know, `String` is a type synonym for `[Char]`. `Char`s don't have a fixed size, because it takes several bytes to represent a character from, say, Unicode. Furthemore, lists are really lazy. If you have a list like `[1,2,3,4]`, it will be evaluated only when completely necessary. So the whole list is sort of a promise of a list. Remember that `[1,2,3,4]` is syntactic sugar for `1:2:3:4:[]`. When the first element of the list is forcibly evaluated (say by printing it), the rest of the list `2:3:4:[]` is still just a promise of a list, and so on. So you can think of lists as promises that the next element will be delivered once it really has to and along with it, the promise of the element after it. It doesn't take a big mental leap to conclude that processing a simple list of numbers as a series of promises might not be the most efficient thing in the world.

That overhead doesn't bother us so much most of the time, but it turns out to be a liability when reading big files and manipulating them. That's why Haskell has **bytestrings**. Bytestrings are sort of like lists, only each element is one byte (or 8 bits) in size. The way they handle laziness is also different.

Bytestrings come in two flavors: strict and lazy ones. Strict bytestrings reside in `Data.ByteString` and they do away with the laziness completely. There are no promises involved; a strict bytestring represents a series of bytes in an array. You can't have things like infinite strict bytestrings. If you evaluate the first byte of a strict bytestring, you have to evaluate it whole. The upside is that there's less overhead because there are no thunks (the technical term for *promise*) involved. The downside is that they're likely to fill your memory up faster because they're read into memory at once.

The other variety of bytestrings resides in `Data.ByteString.Lazy`. They're lazy, but not quite as lazy as lists. Like we said before, there are as many thunks in a list as there are elements. That's what makes them kind of slow for some purposes. Lazy bytestrings take a different approach — they are stored in chunks (not to be confused with thunks!), each chunk has a size of 64K. So if you evaluate a byte in a lazy bytestring (by printing it or something), the first 64K will be evaluated. After that, it's just a promise for the rest of the chunks. Lazy bytestrings are kind of like lists of strict bytestrings with a size of 64K. When you process a file with lazy bytestrings, it will be read chunk by chunk. This is cool because it won't cause the memory usage to skyrocket and the 64K probably fits neatly into your CPU's L2 cache.

If you look through the documentation for `Data.ByteString.Lazy`, you'll see that it has a lot of functions that have the same names as the ones from `Data.List`, only the type signatures have `ByteString` instead of `[a]` and `Word8` instead of `a` in them. The functions with the same names mostly act the same as the ones that work on lists. Because the names are the same, we're going to do a qualified import in a script and then load that script into GHCI to play with bytestrings.

```
import qualified                    as
import qualified               as
```

`B` has lazy bytestring types and functions, whereas `S` has strict ones. We'll mostly be using the lazy version.

The function has the type signature `pack :: [Word8] -> ByteString`. What that means is that it takes a list of bytes of type `Word8` and returns a `ByteString`. You can think of it as taking a list, which is lazy, and making it less lazy, so that it's lazy only at 64K intervals.

What's the deal with that `Word8` type? Well, it's like `Int`, only that it has a much smaller range, namely 0-255. It represents an 8-bit number. And just like `Int`, it's in the `Num` typeclass. For instance, we know that the value `5` is polymorphic in that it can act like any numeral type. Well, it can also take the type of `Word8`.

```
ghci>          99 97 110
       "can"
ghci>          98  120
       "bcdefghijklmnopqrstuvwx"
```

As you can see, you usually don't have to worry about the `Word8` too much, because the type system can makes the numbers choose that type. If you try to use a big number, like `336` as a `Word8`, it will just wrap around to `80`.

We packed only a handful of values into a `ByteString`, so they fit inside one chunk. The `Empty` is like the `[]` for lists.

is the inverse function of `pack`. It takes a bytestring and turns it into a list of bytes.

takes a list of strict bytestrings and converts it to a lazy bytestring. takes a lazy bytestring and converts it to a list of strict ones.

```
ghci>                        40 41 42           43 44 45           46 47 48
       "()*"         "+,-"          "./0"
```

This is good if you have a lot of small strict bytestrings and you want to process them efficiently without joining them into one big strict bytestring in memory first.

The bytestring version of `:` is called It takes a byte and a bytestring and puts the byte at the beginning. It's lazy though, so it will make a new chunk even if the first chunk in the bytestring isn't full. That's why it's better to use the strict version of `cons`, if you're going to be inserting a lot of bytes at the beginning of a bytestring.

```
ghci>         85 $          80 81 82 84
       "U"         "PQRT"
ghci>          85 $          80 81 82 84
       "UPQRT"
ghci>                        50  60
       "2"         "3"         "4"         "5"         "6"         "7"         "8"         "9"         ":"
       ";"         "<"

ghci>                         50  60
       "23456789:;<"
```

As you can see makes an empty bytestring. See the difference between `cons` and `cons'`? With the `foldr`, we started with an empty bytestring and then went over the list of numbers from the right, adding each number to the beginning of the bytestring. When we used `cons`, we ended up with one chunk for every byte, which kind of defeats the purpose.

Otherwise, the bytestring modules have a load of functions that are analogous to those in `Data.List`, including, but not limited to, `head`, `tail`, `init`, `null`, `length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

It also has functions that have the same name and behave the same as some functions found in `System.IO`, only `Strings` are replaced with `ByteStrings`. For instance, the `readFile` function in `System.IO` has a type of `readFile :: FilePath -> IO String`, while the from the bytestring modules has a type of `readFile :: FilePath -> IO ByteString`. Watch out, if you're using strict bytestrings and you attempt to read a file, it will read it into memory at once! With lazy bytestrings, it will read it into neat chunks.

Let's make a simple program that takes two filenames as command-line arguments and copies the first file into the second file. Note that `System.Directory` already has a function called `copyFile`, but we're going to implement our own file copying function and program anyway.

```
import
import qualified                    as
```

```
= do
                    <-


::          ->          ->
            = do
      <-
```

We make our own function that takes two `FilePaths` (remember, `FilePath` is just a synonym for `String`) and returns an I/O action that will copy one file into another using bytestring. In the `main` function, we just get the arguments and call our function with them to get the I/O action, which is then performed.

Notice that a program that doesn't use bytestrings could look just like this, the only difference is that we used `B.readFile` and `B.writeFile` instead of `readFile` and `writeFile`. Many times, you can convert a program that uses normal strings to a program that uses bytestrings by just doing the necessary imports and then putting the qualified module names in front of some functions. Sometimes, you have to convert functions that you wrote to work on strings so that they work on bytestrings, but that's not hard.

Whenever you need better performance in a program that reads a lot of data into strings, give bytestrings a try, chances are you'll get some good performance boosts with very little effort on your part. I usually write programs by using normal strings and then convert them to use bytestrings if the performance is not satisfactory.

Exceptions

All languages have procedures, functions, and pieces of code that might fail in some way. That's just a fact of life. Different languages have different ways of handling those failures. In C, we usually use some abnormal return value (like `-1` or a null pointer) to indicate that what a function returned shouldn't be treated like a normal value. Java and C#, on the other hand, tend to use exceptions to handle failure. When an exception is thrown, the control flow jumps to some code that we've defined that does some cleanup and then maybe re-throws the exception so that some other error handling code can take care of some other stuff.

Haskell has a very good type system. Algebraic data types allow for types like `Maybe` and `Either` and we can use values of those types to represent results that may be there or not. In C, returning, say, `-1` on failure is completely a matter of convention. It only has special meaning to humans. If we're not careful, we might treat these abnormal values as ordinary ones and then they can cause havoc and dismay in our code. Haskell's type system gives us some much-needed safety in that aspect. A function `a -> Maybe b` clearly indicates that it it may produce a `b` wrapped in `Just` or that it may return `Nothing`. The type is different from just plain `a -> b` and if we try to use those two functions interchangeably, the compiler will complain at us.

Despite having expressive types that support failed computations, Haskell still has support for exceptions, because they make more sense in I/O contexts. A lot of things can go wrong when dealing with the outside world because it is so unreliable. For instance, when opening a file, a bunch of things can go wrong. The file might be locked, it might not be there at all or the hard disk drive or something might not be there at all. So it's good to be able to jump to some error handling part of our code when such an error occurs.

Okay, so I/O code (i.e. impure code) can throw exceptions. It makes sense. But what about pure code? Well, it can throw exceptions too. Think about the `div` and `head` functions. They have types of `(Integral a) => a -> a -> a` and `[a] -> a`, respectively. No `Maybe` or `Either` in their return type and yet they can both fail! `div` explodes in your face if you try to divide by zero and `head` throws a tantrum when you give it an empty list.

```
ghci> 4 `div` 0
ghci>
```

Pure code can throw exceptions, but it they can only be caught in the I/O part of our code (when we're inside a *do* block that goes into `main`). That's because you don't know when (or if) anything will be evaluated in pure code, because it is lazy and doesn't have a well-defined order of execution, whereas I/O code does.

Earlier, we talked about how we should spend as little time as possible in the I/O part of our program. The logic of our program should reside mostly within our pure functions, because their results are dependant only on the parameters that the functions are called with. When dealing with pure functions, you only have to think about what a function returns, because it can't do anything else. This makes your life easier. Even though doing some logic in I/O is necessary (like opening files and the like), it should preferably be kept to a minimum. Pure functions are lazy by default, which means that we don't know when they will be evaluated and that it really shouldn't matter. However, once pure functions start throwing exceptions, it matters when they are evaluated. That's why we can only catch exceptions thrown from pure functions in the I/O part of our code. And that's bad, because we want to keep the I/O part as small as possible. However, if we don't catch them in the I/O part of our code, our program crashes. The solution? Don't mix exceptions and pure code. Take advantage of Haskell's powerful type system and use types like `Either` and `Maybe` to represent results that may have failed.

That's why we'll just be looking at how to use I/O exceptions for now. I/O exceptions are exceptions that are caused when something goes wrong while we are communicating with the outside world in an I/O action that's part of `main`. For example, we can try opening a file and then it turns out that the file has been deleted or something. Take a look at this program that opens a file whose name is given to it as a command line argument and tells us how many lines the file has.

```
import
import

     = do              <-
                  <-
                  $ "The file has " ++                    ++ " lines!"
```

A very simple program. We perform the `getArgs` I/O action and bind the first string in the list that it yields to `fileName`. Then we call the contents of the file with that name `contents`. Lastly, we apply `lines` to those contents to get a list of lines and then we get the length of that list and give it to `show` to get a string representation of that number. It works as expected, but what happens when we give it the name of a file that doesn't exist?

Aha, we get an error from GHC, telling us that the file does not exist. Our program crashes. What if we wanted to print out a nicer message if the file doesn't exist? One way to do that is to check if the file exists before trying to open it by using the function from `System.Directory`.

```
import
```

```
import
import

     = do              <-
                     <-
          if
              then do          <-
                               $ "The file has " ++                              ++ " lines!"
              else do          "The file doesn't exist!"
```

We did `fileExists <- doesFileExist fileName` because `doesFileExist` has a type of `doesFileExist :: FilePath -> IO Bool`, which means that it returns an I/O action that has as its result a boolean value which tells us if the file exists. We can't just use `doesFileExist` in an *if* expression directly.

Another solution here would be to use exceptions. It's perfectly acceptable to use them in this context. A file not existing is an exception that arises from I/O, so catching it in I/O is fine and dandy.

To deal with this by using exceptions, we're going to take advantage of the function from `System.IO.Error`. Its type is `catch :: IO a -> (IOError -> IO a) -> IO a`. It takes two parameters. The first one is an I/O action. For instance, it could be an I/O action that tries to open a file. The second one is the so-called handler. If the first I/O action passed to `catch` throws an I/O exception, that exception gets passed to the handler, which then decides what to do. So the final result is an I/O action that will either act the same as the first parameter or it will do what the handler tells it if the first I/O action throws an exception.

If you're familiar with *try-catch* blocks in languages like Java or Python, the `catch` function is similar to them. The first parameter is the thing to try, kind of like the stuff in the *try* block in other, imperative languages. The second parameter is the handler that takes an exception, just like most *catch* blocks take exceptions that you can then examine to see what happened. The handler is invoked if an exception is thrown.

The handler takes a value of type `IOError`, which is a value that signifies that an I/O exception occurred. It also carries information regarding the type of the exception that was thrown. How this type is implemented depends on the implementation of the language itself, which means that we can't inspect values of the type `IOError` by pattern matching against them, just like we can't pattern match against values of type `IO something`. We can use a bunch of useful predicates to find out stuff about values of type `IOError` as we'll learn in a second.

So let's put our new friend `catch` to use!

```
import
import
import

     =       `catch`

      ::
      = do             <-
                  <-
                  $ "The file has " ++                          ++ " lines!"

        ::         ->
          =          "Whoops, had some trouble!"
```

First of all, you'll see that put backticks around it so that we can use it as an infix function, because it takes two parameters. Using it as an infix function makes it more readable. So `toTry `catch` handler` is the same as `catch toTry handler`,

which fits well with its type. `toTry` is the I/O action that we try to carry out and `handler` is the function that takes an `IOError` and returns an action to be carried out in case of an exception.

Let's give this a go:

In the handler, we didn't check to see what kind of `IOError` we got. We just say `"Whoops, had some trouble!"` for any kind of error. Just catching all types of exceptions in one handler is bad practice in Haskell just like it is in most other languages. What if some other exception happens that we don't want to catch, like us interrupting the program or something? That's why we're going to do the same thing that's usually done in other languages as well: we'll check to see what kind of exception we got. If it's the kind of exception we're waiting to catch, we do our stuff. If it's not, we throw that exception back into the wild. Let's modify our program to catch only the exceptions caused by a file not existing.

```
import
import
import

     =       `catch`

      ::
      = do              <-
                    <-
                    $ "The file has " ++                              ++ " lines!"

        ::          ->

    |                       =          "The file doesn't exist!"
    |           =
```

Everything stays the same except the handler, which we modified to only catch a certain group of I/O exceptions. Here we used two new functions from `System.IO.Error` — and . `isDoesNotExistError` is a predicate over `IOErrors`, which means that it's a function that takes an `IOError` and returns a `True` or `False`, meaning it has a type of `isDoesNotExistError :: IOError -> Bool`. We use it on the exception that gets passed to our handler to see if it's an error caused by a file not existing. We use guard syntax here, but we could have also used an *if else*. If it's not caused by a file not existing, we re-throw the exception that was passed by the handler with the `ioError` function. It has a type of `ioError :: IOException -> IO a`, so it takes an `IOError` and produces an I/O action that will throw it. The I/O action has a type of `IO a`, because it never actually yields a result, so it can act as `IO` *`anything`*.

So the exception thrown in the `toTry` I/O action that we glued together with a *do* block isn't caused by a file existing, `toTry` `` `catch` `` `handler` will catch that and then re-throw it. Pretty cool, huh?

There are several predicates that act on `IOError` and if a guard doesn't evaluate to `True`, evaluation falls through to the next guard. The predicates that act on `IOError` are:

-
-
-
-
-
-
-

-

Most of these are pretty self-explanatory. `isUserError` evaluates to `True` when we use the function to make the exception, which is used for making exceptions from our code and equipping them with a string. For instance, you can do `ioError $ userError "remote computer unplugged!"`, although It's prefered you use types like `Either` and `Maybe` to express possible failure instead of throwing exceptions yourself with `userError`.

So you could have a handler that looks something like this:

```
        ::         ->

    |                    =          "The file doesn't exist!"
    |               =
    |                    =
    |           =
```

Where `notifyCops` and `freeSomeSpace` are some I/O actions that you define. Be sure to re-throw exceptions if they don't match any of your criteria, otherwise you're causing your program to fail silently in some cases where it shouldn't.

`System.IO.Error` also exports functions that enable us to ask our exceptions for some attributes, like what the handle of the file that caused the error is, or what the filename is. These start with `ioe` and you can see a full list of them in the documentation. Say we want to print the filename that caused our error. We can't print the `filaName` that we got from `getArgs`, because only the `IOError` is passed to the handler and the handler doesn't know about anything else. A function depends only on the parameters it was called with. That's why we can use the function, which has a type of `ioeGetFileName :: IOError -> Maybe FilePath`. It takes an `IOError` as a parameter and maybe returns a `FilePath` (which is just a type synonym for `String`, remember, so it's kind of the same thing). Basically, what it does is it extracts the file path from the `IOError`, if it can. Let's modify our program to print out the file path that's responsible for the exception occurring.

```
import
import
import

     =       `catch`

      ::
      = do              <-
                  <-
                  $ "The file has " ++                          ++ " lines!"

        ::         ->

    |                      =
       case                  of          ->          $ "Whoops! File does not exist at: " ++

                                        -
>          "Whoops! File does not exist at unknown location!"
    |           =
```

In the guard where `isDoesNotExistError` is `True`, we used a *case* expression to call `ioeGetFileName` with `e` and then pattern match against the `Maybe` value that it returned. Using *case* expressions is commonly used when you want to pattern match against something without bringing in a new function.

You don't have to use one handler to `catch` exceptions in your whole I/O part. You can just cover certain parts of your I/O code with `catch` or you can cover several of them with `catch` and use different handlers for them, like so:

```
      = do       `catch`
                       `catch`
```

Here, `toTry` uses `handler1` as the handler and `thenTryThis` uses `handler2`. `launchRockets` isn't a parameter to `catch`, so whichever exceptions it might throw will likely crash our program, unless `launchRockets` uses `catch` internally to handle its own exceptions. Of course `toTry`, `thenTryThis` and `launchRockets` are I/O actions that have been glued together using *do* syntax and hypothetically defined somewhere else. This is kind of similar to *try-catch* blocks of other languages, where you can surround your whole program in a single *try-catch* or you can use a more fine-grained approach and use different ones in different parts of your code to control what kind of error handling happens where.

Now you know how to deal with I/O exceptions! Throwing exceptions from pure code and dealing with them hasn't been covered here, mainly because, like we said, Haskell offers much better ways to indicate errors than reverting to I/O to catch them. Even when glueing together I/O actions that might fail, I prefer to have their type be something like `IO (Either a b)`, meaning that they're normal I/O actions but the result that they yield when performed is of type `Either a b`, meaning it's either `Left a` or `Right b`.

Vytváříme si své typy a typové třídy Obsah Functionally Solving Problems

← Input and Output Obsah Functors, Applicative Functors and Monoids →

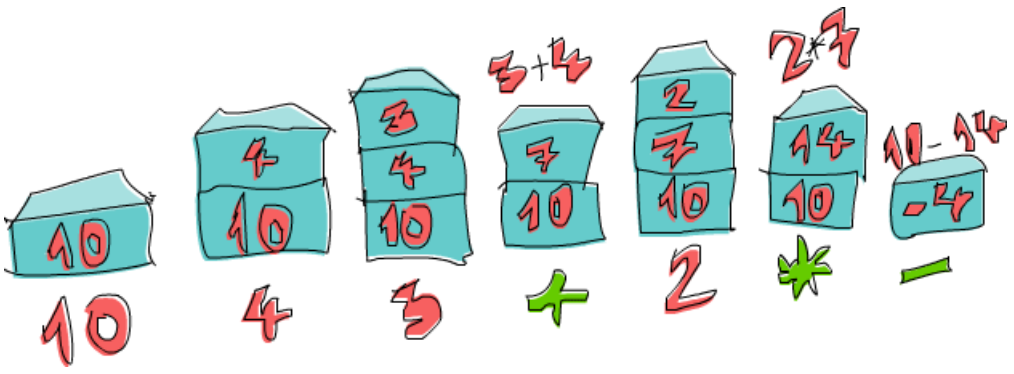
Functionally Solving Problems

In this chapter, we'll take a look at a few interesting problems and how to think functionally to solve them as elegantly as possible. We probably won't be introducing any new concepts, we'll just be flexing our newly acquired Haskell muscles and practicing our coding skills. Each section will present a different problem. First we'll describe the problem, then we'll try and find out what the best (or least worst) way of solving it is.

Reverse Polish notation calculator

Usually when we write mathematical expressions in school, we write them in an infix manner. For instance, we write `10 - (4 + 3) * 2`. `+`, `*` and `-` are infix operators, just like the infix functions we met in Haskell (`+`, `` `elem` ``, etc.). This makes it handy because we, as humans, can parse it easily in our minds by looking at such an expression. The downside to it is that we have to use parentheses to denote precedence.

Reverse Polish notation is another way of writing down mathematical expressions. Initially it looks a bit weird, but it's actually pretty easy to understand and use because there's no need for parentheses and it's very easy to punch into a calculator. While most modern calculators use infix notation, some people still swear by RPN calculators. This is what the previous infix expression looks like in RPN: `10 4 3 + 2 * -`. How do we calculate what the result of that is? Well, think of a stack. You go over the expression from left to right. Every time a number is encountered, push it on to the stack. When we encounter an operator, take the two numbers that are on top of the stack (we also say that we *pop* them), use the operator and those two and then push the resulting number back onto the stack. When you reach the end of the expression, you should be left with a single number if the expression was well-formed and that number represents the result.



Let's go over the expression `10 4 3 + 2 * -` together! First we push `10` on to the stack and the stack is now `10`. The next item is `4`, so we push it to the stack as well. The stack is now `10, 4`. We do the same with `3` and the stack is now `10, 4, 3`. And now, we encounter an operator, namely `+`! We pop the two top numbers from the stack (so now the stack is just `10`), add those numbers together and push that result to the stack. The stack is now `10, 7`. We push `2` to the stack, the stack for now is `10, 7, 2`. We've encountered an operator again, so let's pop `7` and `2` off the stack, multiply them and push that result to the stack. Multiplying `7` and `2` produces a `14`, so the stack we have now is `10, 14`. Finally, there's a `-`. We pop `10` and `14` from the stack, subtract `14` from `10` and push that back. The number on the stack is now `-4` and because there are no more %0000enor operators in our expression, that's our result!

Now that we know how we'd calculate any RPN expression by hand, let's think about how we could make a Haskell function that takes as its parameter a string that contains a RPN expression, like `"10 4 3 + 2 * -"` and gives us back its result.

What would the type of that function be? We want it to take a string as a parameter and produce a number as its result. So it will probably be something like `solveRPN :: (Num a) => String -> a`.

Protip: it really helps to first think what the type declaration of a function should be before concerning ourselves with the implementation and then write it down. In Haskell, a function's type declaration tells us a whole lot about the function, due to the very strong type system.

Cool. When implementing a solution to a problem in Haskell, it's also good to think back on how you did it by hand and maybe try to see if you can gain any insight from that. Here we see that we treated every number or operator that was separated by a space as a single item. So it might help us if we start by breaking a string like `"10 4 3 + 2 * -"` into a list of items like `["10","4","3","+","2","*","-"]`.

Next up, what did we do with that list of items in our head? We went over it from left to right and kept a stack as we did that. Does the previous sentence remind you of anything? Remember, in the section about folds, we said that pretty much any function where you traverse a list from left to right or right to left one element by element and build up (accumulate) some result (whether it's a number, a list, a stack, whatever) can be implemented with a fold.

In this case, we're going to use a left fold, because we go over the list from left to right. The accumulator value will be our stack and hence, the result from the fold will also be a stack, only as we've seen, it will only have one item.

One more thing to think about is, well, how are we going to represent the stack? I propose we use a list. Also I propose that we keep the top of our stack at the head of the list. That's because adding to the head (beginning) of a list is much faster than adding to the end of it. So if we have a stack of, say, `10, 4, 3`, we'll represent that as the list `[3,4,10]`.

Now we have enough information to roughly sketch our function. It's going to take a string, like, `"10 4 3 + 2 * -"` and break it down into a list of items by using `words` to get `["10","4","3","+","2","*","-"]`. Next, we'll do a left fold over that list and end up with a stack that has a single item, so `[-4]`. We take that single item out of the list and that's our final result!

So here's a sketch of that function:

```
import

        ::          =>        ->
                    =
    where                           =
```

We take the expression and turn it into a list of items. Then we fold over that list of items with the folding function. Mind the `[]`, which represents the starting accumulator. The accumulator is our stack, so `[]` represents an empty stack, which is what we start with. After getting the final stack with a single item, we call `head` on that list to get the item out and then we apply `read`.

So all that's left now is to implement a folding function that will take a stack, like `[4,10]`, and an item, like `"3"` and return a new stack `[3,4,10]`. If the stack is `[4,10]` and the item `"*"`, then it will have to return `[40]`. But before that, let's turn our function into point-free style because it has a lot of parentheses that are kind of freaking me out:

```
import

        ::          =>        ->
        =       .                       .
```

```
    where                           =
```

Ah, there we go. Much better. So, the folding function will take a stack and an item and return a new stack. We'll use pattern matching to get the top items of a stack and to pattern match against operators like `"*"` and `"-"`.

```
        ::                  =>          ->
        =       .                               .
    where                                   "*" =     *
                                            "+" =     +
                                            "-" =     -
                                                =
```

We laid this out as four patterns. The patterns will be tried from top to bottom. First the folding function will see if the current item is `"*"`. If it is, then it will take a list like `[3,4,9,3]` and call its first two elements `x` and `y` respectively. So in this case, `x` would be `3` and `y` would be `4`. `ys` would be `[9,3]`. It will return a list that's just like `ys`, only it has `x` and `y` multiplied as its head. So with this we pop the two topmost numbers off the stack, multiply them and push the result back on to the stack. If the item is not `"*"`, the pattern matching will fall through and `"+"` will be checked, and so on.

If the item is none of the operators, then we assume it's a string that represents a number. If it's a number, we just call `read` on that string to get a number from it and return the previous stack but with that number pushed to the top.

And that's it! Also noticed that we added an extra class constraint of `Read a` to the function declaration, because we call `read` on our string to get the number. So this declaration means that the result can be of any type that's part of the `Num` and `Read` typeclasses (like `Int`, `Float`, etc.).

For the list of items `["2","3","+"]`, our function will start folding from the left. The intial stack will be `[]`. It will call the folding function with `[]` as the stack (accumulator) and `"2"` as the item. Because that item is not an operator, it will be `read` and the added to the beginning of `[]`. So the new stack is now `[2]` and the folding function will be called with `[2]` as the stack and `["3"]` as the item, producing a new stack of `[3,2]`. Then, it's called for the third time with `[3,2]` as the stack and `"+"` as the item. This causes these two numbers to be popped off the stack, added together and pushed back. The final stack is `[5]`, which is the number that we return.

Let's play around with our function:

```
ghci>           "10 4 3 + 2 * -"
 4
ghci>           "2 3 +"
5
ghci>           "90 34 12 33 55 66 + * - +"
 3947
ghci>           "90 34 12 33 55 66 + * - + -"
4037
ghci>           "90 34 12 33 55 66 + * - + -"
4037
ghci>           "90 3 -"
87
```

Cool, it works! One nice thing about this function is that it can be easily modified to support various other operators. They don't even have to be binary operators. For instance, we can make an operator `"log"` that just pops one number off the stack and pushes back its logarithm. We can also make a ternary operators that pop three numbers off the stack and push back a result or operators like `"sum"` which pop off all the numbers and push back their sum.

Let's modify our function to take a few more operators. For simplicity's sake, we'll change its type declaration so that it returns a number of type `Float`.

```
  import
```

```
      ::         ->
      =       .
  where                          "*" =     *
                                 "+" =     +
                                 "-" =     -
                                 "/" =     /
                                 "^" =
                               "ln" =
                            "sum" =
                                  =
```

Wow, great! `/` is division of course and `**` is floating point exponentiation. With the logarithm operator, we just pattern match against a single element and the rest of the stack because we only need one element to perform its natural logarithm. With the sum operator, we just return a stack that has only one element, which is the sum of the stack so far.

```
ghci>           "2.7 ln"
0.9932518
ghci>           "10 10 10 10 sum 4 /"
10.0
ghci>           "10 10 10 10 10 sum 4 /"
12.5
ghci>           "10 2 ^"
100.0
```

Notice that we can include floating point numbers in our expression because `read` knows how to read them.

```
ghci>           "43.2425 0.5 ^"
6.575903
```

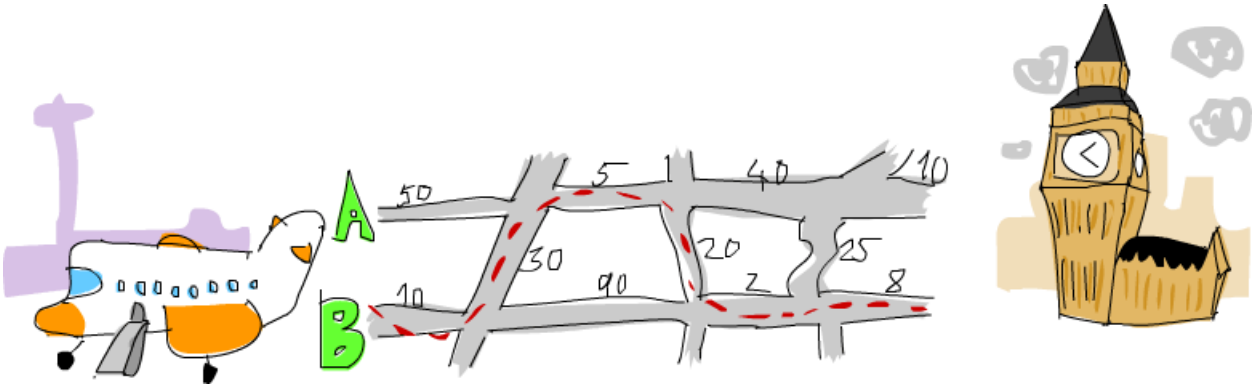
I think that making a function that can calculate arbitrary floating point RPN expressions and has the option to be easily extended in 10 lines is pretty awesome.

One thing to note about this function is that it's not really fault tolerant. When given input that doesn't make sense, it will just crash everything. We'll make a fault tolerant version of this with a type declaration of `solveRPN :: String -> Maybe Float` once we get to know monads (they're not scary, trust me!). We could make one right now, but it would be a bit tedious because it would involve a lot of checking for `Nothing` on every step. If you're feeling up to the challenge though, you can go ahead and try it! Hint: you can use `reads` to see if a read was successful or not.

Heathrow to London

Our next problem is this: your plane has just landed in England and you rent a car. You have a meeting really soon and you have to get from Heathrow Airport to London as fast as you can (but safely!).

There are two main roads going from Heathrow to London and there's a number of regional roads crossing them. It takes you a fixed amount of time to travel from one crossroads to another. It's up to you to find the optimal path to take so that you get to London as fast as you can! You start on the left side and can either cross to the other main road or go forward.



As you can see in the picture, the shortest path from Heathrow to London in this case is to start on main road B, cross over, go forward on A, cross over again and then go forward twice on B. If we take this path, it takes us 75 minutes. Had we chosen any other path, it would take more than that.

Our job is to make a program that takes input that represents a road system and print out what the shortest path across it is. Here's what the input would look like for this case:

To mentally parse the input file, read it in threes and mentally split the road system into sections. Each section is comprised of a road A, road B and a crossing road. To have it neatly fit into threes, we say that there's a last crossing section that takes 0 minutes to drive over. That's because we don't care where we arrive in London, as long as we're in London.

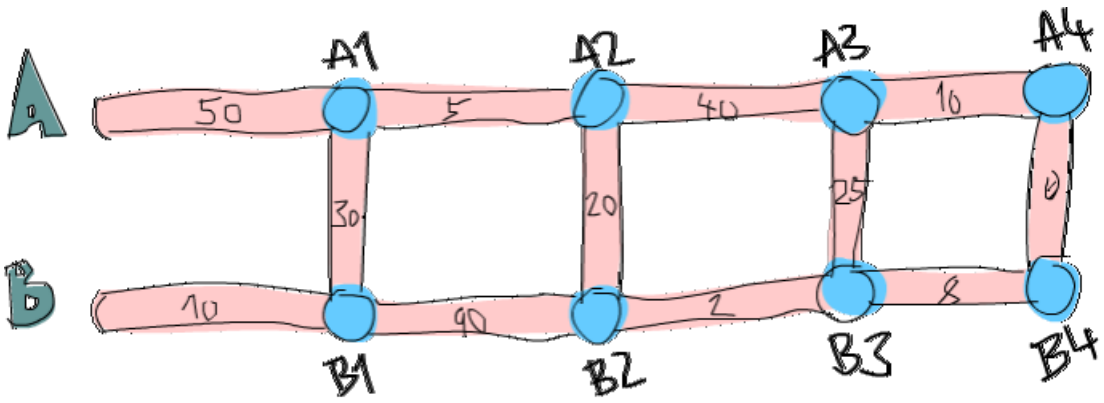
Just like we did when solving the RPN calculator problem, we're going to solve this problem in three steps:

- Forget Haskell for a minute and think about how we'd solve the problem by hand
- Think about how we're going to represent our data in Haskell
- Figure out how to operate on that data in Haskell so that we produce at a solution

In the RPN calculator section, we first figured out that when calculating an expression by hand, we'd keep a sort of stack in our minds and then go over the expression one item at a time. We decided to use a list of strings to represent our expression. Finally, we used a left fold to walk oevr the list of strings while keeping a stack to produce a solution.

Okay, so how would we figure out the shortest path from Heathrow to London by hand? Well, we can just sort of look at the whole picture and try to guess what the shortest path is and hopefully we'll make a guess that's right. That solution works for very small inputs, but what if we have a road that has 10,000 sections? Yikes! We also won't be able to say for certain that our solution is the optimal one, we can just sort of say that we're pretty sure.

That's not a good solution then. Here's a simplified picture of our road system:



Alright, can you figure out what the shortest path to the first crossroads (the first blue dot on A, marked *A1*) on road A is? That's pretty trivial. We just see if it's shorter to go directly forward on A or if it's shorter to go forward on B and then cross over. Obviously, it's cheaper to go forward via B and then cross over because that takes 40 minutes, whereas going directly via A

takes 50 minutes. What about crossroads *B1*? Same thing. We see that it's a lot cheaper to just go directly via B (incurring a cost of 10 minutes), because going via A and then crossing over would take us a whole 80 minutes!

Now we know what the cheapest path to *A1* is (go via B and then cross over, so we'll say that's `B, C` with a cost of 40) and we know what the cheapest path to *B1* is (go directly via B, so that's just `B`, going at 10). Does this knowledge help us at all if we want to know the cheapest path to the next crossroads on both main roads? Gee golly, it sure does!

Let's see what the shortest path to *A2* would be. To get to *A2*, we'll either go directly to *A2* from *A1* or we'll go forward from *B1* and then cross over (remember, we can only move forward or cross to the other side). And because we know the cost to *A1* and *B1*, we can easily figure out what the best path to *A2* is. It costs 40 to get to *A1* and then 5 to get from *A1* to *A2*, so that's `B, C, A` for a cost of 45. It costs only 10 to get to *B1*, but then it would take an additional 110 minutes to go to *B2* and then cross over! So obviously, the cheapest path to *A2* is `B, C, A`. In the same way, the cheapest way to *B2* is to go forward from *A1* and then cross over.

> **Maybe you're asking yourself**: but what about getting to *A2* by first crossing over at *B1* and then going on forward? Well, we already covered crossing from *B1* to *A1* when we were looking for the best way to *A1*, so we don't have to take that into account in the next step as well.

Now that we have the best path to *A2* and *B2*, we can repeat this indefinitely until we reach the end. Once we've gotten the best paths for *A4* and *B4*, the one that's cheaper is the optimal path!

So in essence, for the second section, we just repeat the step we did at first, only we take into account what the previous best paths on A and B. We could say that we also took into account the best paths on A and on B in the first step, only they were both empty paths with a cost of 0.

Here's a summary. To get the bast path from Heathrow to London, we do this: first we see what the best path to the next crossroads on main road A is. The two options are going directly forward or starting at the opposite road, going forward and then crossing over. We remember the cost and the path. We use the same method to see what the best path to the next crossroads on main road B is and remember that. Then, we see if the path to the next crossroads on A is cheaper if we go from the previous A crossroads or if we go from the previous B crossroads and then cross over. We remember the cheaper path and then we do the same for the crossroads opposite of it. We do this for every section until we reach the end. Once we've reached the end, the cheapest of the two paths that we have is our optimal path!

So in essence, we keep one shortest path on the A road and one shortest path on the B road and when we reach the end, the shorter of those two is our path. We now know how to figure out the shortest path by hand. If you had enough time, paper and pencils, you could figure out the shortest path through a road system with any number of sections.

Next step! How do we represent this road system with Haskell's data types? One way is to think of the starting points and crossroads as nodes of a graph that point to other crossroads. If we imagine that the starting points actually point to each other with a road that has a length of one, we see that every crossroads (or node) points to the node on the other side and also to the next one on its side. Except for the last nodes, they just point to the other side.

```
data      =
data      =
```

A node is either a normal node and has information about the road that leads to the other main road and the road that leads to the next node or an end node, which only has information about the road to the other main road. A road keeps information about how long it is and which node it points to. For instance, the first part of the road on the A main road would be `Road 50 a1` where `a1` would be a node `Node x y`, where `x` and `y` are roads that point to *B1* and *A2*.

Another way would be to use `Maybe` for the road parts that point forward. Each node has a road part that point to the opposite road, but only those nodes that aren't the end ones have road parts that point forward.

```
data      =
data      =
```

This is an alright way to represent the road system in Haskell and we could certainly solve this problem with it, but maybe we could come up with something simpler? If we think back to our solution by hand, we always just checked the lengths of three road parts at once: the road part on the A road, its opposite part on the B road and part C, which touches those two parts and connects them. When we were looking for the shortest path to *A1* and *B1*, we only had to deal with the lengths of the first three parts, which have lengths of 50, 10 and 30. We'll call that one section. So the road system that we use for this example can be easily represented as four sections: `50, 10, 30`, `5, 90, 20`, `40, 2, 25`, and `10, 8, 0`.

It's always good to keep our data types as simple as possible, although not any simpler!

```
data         =               ::          ::          ::       deriving
type           =
```

This is pretty much perfect! It's as simple as it goes and I have a feeling it'll work perfectly for implementing our solution. `Section` is a simple algebraic data type that holds three integers for the lenghts of its three road parts. We introduce a type synonym as well, saying that `RoadSystem` is a list of sections.

> We could also use a triple of `(Int, Int, Int)` to represent a road section. Using tuples instead of making your own algebraic data types is good for some small localized stuff, but it's usually better to make a new type for things like this. It gives the type system more information about what's what. We can use `(Int, Int, Int)` to represent a road section or a vector in 3D space and we can operate on those two, but that allows us to mix them up. If we use `Section` and `Vector` data types, then we can't accidentally add a vector to a section of a road system.

Our road system from Heathrow to London can now be represented like this:

```
            ::
            =          50 10 30          5 90 20          40 2 25          10 8 0
```

All we need to do now is to implement the solution that we came up with previously in Haskell. What should the type declaration for a function that calculates a shortest path for any given road system be? It should take a road system as a parameter and return a path. We'll represent a path as a list as well. Let's introduce a `Label` type that's just an enumeration of either `A`, `B` or `C`. We'll also make a type synonym: `Path`.

```
data      =   |   |   deriving
type      =
```

Our function, we'll call it `optimalPath` should thus have a type declaration of `optimalPath :: RoadSystem -> Path`. If called with the road system `heathrowToLondon`, it should return the following path:

```
    10    30    5    20    2    8
```

We're going to have to walk over the list with the sections from left to right and keep the optimal path on A and optimal path on B as we go along. We'll accumulate the best path as we walk over the list, left to right. What does that sound like? Ding, ding, ding! That's right, A LEFT FOLD!

When doing the solution by hand, there was a step that we repeated over and over again. It involed checking the optimal paths on A and B so far and the current section to produce the new optimal paths on A and B. For instance, at the beginning the optimal paths were `[]` and `[]` for A and B respectively. We examined the section `Section 50 10 30` and concluded that the new optimal path to *A1* is `[(B,10),(C,30)]` and the optimal path to *B1* is `[(B,10)]`. If you look at this step as a function, it takes a pair of paths and a section and produces a new pair of paths. The type is `(Path, Path) -> Section -> (Path, Path)`. Let's go ahead and implement this function, because it's bound to be useful.

Hint: it will be useful because `(Path, Path) -> Section -> (Path, Path)` can be used as the binary function for a left fold, which has to have a type of `a -> b -> a`

```
      ::                ->           ->
                                       =
  let         =     $
              =     $
                        =          +
                      =          +    +
                        =          +
                      =          +    +
                  = if                     <=
                       then
                       else
                  = if                     <=
                       then
                       else
  in
```

What's going on here? First, calculate the optimal price on road A based on the best so far on A and we do the same for B. We do `sum $ map snd pathA`, so if `pathA` is something like `[(A,100),(C,20)]`, `priceA` becomes `120`. `forwardPriceToA` is the price that we would pay if we went to the next crossroads on A if we went there directly from the previous crossroads on A. It equals the best price to our previous A, plus the length of the A part of the current section. `crossPriceToA` is the price that we would pay if we went to the next A by going forward from the previous B and then crossing over. It's the best price to the previous B so far plus the B length of the section plus the C length of the section. We determine `forwardPriceToB` and `crossPriceToB` in the same manner.

Now that we know what the best way to A and B is, we just need to make the new paths to A and B based on that. If it's cheaper to go to A by just going forwards, we set `newPathToA` to be `(A,a):pathA`. Basically we prepend the `Label` `A` and the section length `a` to the optimal path path on A so far. Basically, we say that the best path to the next A crossroads is the path to the previous A crossroads and then one section forward via A. Remember, `A` is just a label, whereas `a` has a type of `Int`. Why do we prepend instead of doing `pathA ++ [(A,a)]`? Well, adding an element to the beginning of a list (also known as consing) is much faster than adding it to the end. This means that the path will be the wrong way around once we fold over a list with this function, but it's easy to reverse the list later. If it's cheaper to get to the next A crossroads by going forward from road B and then crossing over, then `newPathToA` is the old path to B that then goes forward and crosses to A. We do the same thing for `newPathToB`, only everything's mirrored.

Finally, we return `newPathToA` and `newPathToB` in a pair.

Let's run this function on the first section of `heathrowToLondon`. Because it's the first section, the best paths on A and B parameter will be a pair of empty lists.

```
ghci>
     30     10       10
```

Remember, the paths are reversed, so read them from right to left. From this we can read that the best path to the next A is to start on B and then cross over to A and that the best path to the next B is to just go directly forward from the starting point at B.

Optimization tip: when we do `priceA = sum $ map snd pathA`, we're calculating the price from the path on every step. We wouldn't have to do that if we implemented `roadStep` as a `(Path, Path, Int, Int) -> Section -> (Path, Path, Int, Int)` function where the integers represent the best price on A and B.

Now that we have a function that takes a pair of paths and a section and produces a new optimal path, we can just easily do a left fold over a list of sections. `roadStep` is called with `([],[])` and the first section and returns a pair of optimal paths to that section. Then, it's called with that pair of paths and the next section and so on. When we've walked over all the sections, we're left with a pair of optimal paths and the shorter of them is our answer. With this in mind, we can implement `optimalPath`.

```
          ::            ->
                    =
    let                       =
    in  if                        <=
          then
          else
```

We left fold over `roadSystem` (remember, it's a list of sections) with the starting accumulator being a pair of empty paths. The result of that fold is a pair of paths, so we pattern match on the pair to get the paths themselves. Then, we check which one of these was cheaper and return it. Before returning it, we also reverse it, because the optimal paths so far were reversed due to us choosing consing over appending.

Let's test this!

```
ghci>
    10     30     5     20     2     8     0
```

This is the result that we were supposed to get! Awesome! It differs from our expected result a bit because there's a step `(C,0)` at the end, which means that we cross over to the other road once we're in London, but because that crossing doesn't cost anything, this is still the correct result.

We have the function that finds an optimal path based on, now we just have to read a textual representation of a road system from the standard input, convert it into a type of `RoadSystem`, run that through our `optimalPath` function and print the path.

First off, let's make a function that takes a list and splits it into groups of the same size. We'll call it `groupsOf`. For a parameter of `[1..10]`, `groupsOf 3` should return `[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
          ::      ->      ->
          0   =
                =
                =             :
```

A standard recursive function. For an `xs` of `[1..10]` and an `n` of `3`, this equals `[1,2,3] : groupsOf 3 [4,5,6,7,8,9,10]`. When the recursion is done, we get our list in groups of three. And here's our `main` function, which reads from the standard input, makes a `RoadSystem` out of it and prints out the shortest path:

```
import

     = do
          <-
    let     =          3          $
                =               ->
          =
                =        $          .
               =     $
          $ "The best path to take is: " ++
          $ "The price is: " ++
```

First, we get all the contents from the standard input. Then, we call `lines` with our contents to convert something like `"50\n10\n30\n...` to `["50","10","30"..` and then we map `read` to that to convert it to a list of numbers. We call `groupsOf 3` on it so that we turn it to a list of lists of length 3. We map the lambda `(\[a,b,c] -> Section a b c)` over that list of lists. As you can see, the lambda just takes a list of length 3 and turns it into a section. So `roadSystem` is now our system of roads and it even has the correct type, namely `RoadSystem` (or `[Section]`). We call `optimalPath` with that and then get the path and the price in a nice textual representation and print it out.

We save the following text

in a file called `paths.txt` and then feed it to our program.

Works like a charm! You can use your knowledge of the `Data.Random` module to generate a much longer system of roads, which you can then feed to what we just wrote. If you get stack overflows, try using `foldl'` instead of `foldl`, because `foldl'` is strict.

Input and Output Obsah Functors, Applicative Functors and Monoids

Functionally Solving Problems Obsah

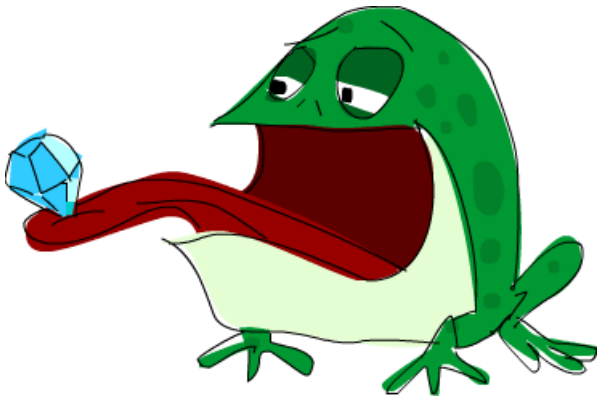
Functors, Applicative Functors and Monoids

Haskell's combination of purity, higher order functions, parameterized algebraic data types, and typeclasses allows us to implement polymorphism on a much higher level than possible in other languages. We don't have to think about types belonging to a big hierarchy of types. Instead, we think about what the types can act like and then connect them with the appropriate typeclasses. An `Int` can act like a lot of things. It can act like an equatable thing, like an ordered thing, like an enumerable thing, etc.

Typeclasses are open, which means that we can define our own data type, think about what it can act like and connect it with the typeclasses that define its behaviors. Because of that and because of Haskell's great type system that allows us to know a lot about a function just by knowing its type declaration, we can define typeclasses that define behavior that's very general and abstract. We've met typeclasses that define operations for seeing if two things are equal or comparing two things by some ordering. Those are very abstract and elegant behaviors, but we just don't think of them as anything very special because we've been dealing with them for most of our lives. We recently met functors, which are basically things that can be mapped over. That's an example of a useful and yet still pretty abstract property that typeclasses can describe. In this chapter, we'll take a closer look at functors, along with slightly stronger and more useful versions of functors called applicative functors. We'll also take a look at monoids, which are sort of like socks.

Functors redux

We've already talked about functors in their own little section. If you haven't read it yet, you should probably give it a glance right now, or maybe later when you have more time. Or you can just pretend you read it.



Still, here's a quick refresher: Functors are things that can be mapped over, like lists, `Maybe`s, trees, and such. In Haskell, they're described by the typeclass `Functor`, which has only one typeclass method, namely `fmap`, which has a type of `fmap :: (a -> b) -> f a -> f b`. It says: give me a function that takes an `a` and returns a `b` and a box with an `a` (or several of them) inside it and I'll give you a box with a `b` (or several of them) inside it. It kind of applies the function to the element inside the box.

> **A word of advice.** Many times the box analogy is used to help you get some intuition for how functors work, and later, we'll probably use the same analogy for applicative functors and monads. It's an okay analogy that helps people understand functors at first, just don't take it too literally, because for some functors the box analogy has to be stretched really thin to still hold some truth. A more correct term for what a functor is would be *computational context*. The context might be that the computation can have a value or it might have failed (`Maybe` and `Either a`) or that there might be more values (lists), stuff like that.

If we want to make a type constructor an instance of `Functor`, it has to have a kind of `* -> *`, which means that it has to take exactly one concrete type as a type parameter. For example `Maybe` can be made an instance because it takes one type parameter to produce a concrete type, like `Maybe Int` or `Maybe String`. If a type constructor takes two parameters, like

,

`Either`, we have to partially apply the type constructor until it only takes one type parameter. So we can't write `instance Functor Either where`, but we can write `instance Functor (Either a) where` and then if we imagine that `fmap` is only for `Either a`, it would have a type declaration of `fmap :: (b -> c) -> Either a b -> Either a c`. As you can see, the `Either a` part is fixed, because `Either a` takes only one type parameter, whereas just `Either` takes two so `fmap :: (b -> c) -> Either b -> Either c` wouldn't really make sense.

We've learned by now how a lot of types (well, type constructors really) are instances of `Functor`, like `[]`, `Maybe`, `Either a` and a `Tree` type that we made on our own. We saw how we can map functions over them for great good. In this section, we'll take a look at two more instances of functor, namely `IO` and `(->) r`.

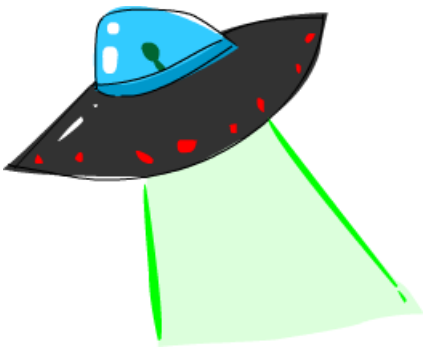
If some value has a type of, say, `IO String`, that means that it's an I/O action that, when performed, will go out into the real world and get some string for us, which it will yield as a result. We can use `<-` in *do* syntax to bind that result to a name. We mentioned that I/O actions are like boxes with little feet that go out and fetch some value from the outside world for us. We can inspect what they fetched, but after inspecting, we have to wrap the value back in `IO`. By thinking about this box with little feet analogy, we can see how `IO` acts like a functor.

The `IO a` type doesn't expose any value constructors, because I/O is implementation dependent. This forces us to keep our I/O code and our pure code separated. That's why we can't see exactly how `IO` is an instance of `Functor`, but we can play around with it to gain some intuition. It's pretty simple really. Check out this piece of code:

```
= do      <-
    let       =
            "You said " ++       ++ " backwards!"
            "Yes, you really said" ++       ++ " backwards!"
```

The user is prompted for a line and we give it back to the user, only reversed. Here's how to rewrite this by using `fmap`:

```
= do      <-
            "You said " ++      ++ " backwards!"
            "Yes, you really said" ++      ++ " backwards!"
```



Just like when we `fmap reverse` over `Just "blah"` to get `Just "halb"`, we can `fmap reverse` over `getLine`. `getLine` is an I/O action that has a type of `IO String` and mapping `reverse` over it gives us an I/O action that will go out into the real world and get a line and then apply `reverse` to its result. Like we can apply a function to something that's inside a `Maybe` box, we can apply a function to what's inside an `IO` box, only it has to go out into the real world to get something. Then when we bind it to a name by using `<-`, the name will reflect the result that already has `reverse` applied to it.

If we look at what `fmap`'s type would be if it were limited to `IO`, it would be `fmap :: (a -> b) -> IO a -> IO b`. `fmap` takes a function and an I/O action and returns a new I/O action that's like the old one, except that the function is applied to its contained result.

If you ever find yourself binding the result of an I/O action to a name, only to apply a function to that and call that something else, consider using `fmap`, because it looks prettier. If you want to apply multiple transformations to some data inside a functor, you can declare your own function at the top level, make a lambda function or ideally, use function composition:

```
import
import

    = do      <-                  '-' .         .
```

As you probably know, `intersperse '-' . reverse . map toUpper` is a function that takes a string, maps `toUpper` over it, the applies `reverse` to that result and then applies `intersperse '-'` to that result. It's like writing `(\xs -> intersperse '-' (reverse (map toUpper xs)))`, only prettier.

Another instance of `Functor` that we've been dealing with all along but didn't know was a `Functor` is `(->) r`. You're probably slightly confused now, since what the heck does `(->) r` mean? The function type `r -> a` can be rewritten as `(->) r a`, much like we can write `2 + 3` as `(+) 2 3`. When we look at it as `(->) r a`, we can see `(->)` in a slighty different light, because we see that it's just a type constructor that takes two type parameters, just like `Either`. But remember, we said that a type constructor has to take exactly one type parameter so that it can be made an instance of `Functor`. That's why we can't make `(->)` an instance of `Functor`, but if we partially apply it to `(->) r`, it doesn't pose any problems. If the syntax allowed for type constructors to be partially applied with sections (like we can partially apply + by doing `(2+)`, which is the same as `(+) 2`), you could write `(->) r` as `(r ->)`. How are functions functors? Well, let's take a look at the implementation, which lies in `Control.Monad.Instances`

> We usually mark functions that take anything and return anything as `a -> b`. `r -> a` is the same thing, we just used different letters for the type variables.

```
instance                  where
             =      ->
```

If the syntax allowed for it, it could have been written as

```
instance                 where
             =      ->
```

But it doesn't, so we have to write it in the former fashion.

First of all, let's think about `fmap`'s type. It's `fmap :: (a -> b) -> f a -> f b`. Now what we'll do is mentally replace all the `f`'s, which are the role that our functor instance plays, with `(->) r`'s. We'll do that to see how `fmap` should behave for this particular instance. We get `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)`. Now what we can do is write the `(->) r a` and `(-> r b)` types as infix `r -> a` and `r -> b`, like we normally do with functions. What we get now is `fmap :: (a -> b) -> (r -> a) -> (r -> b)`.

Hmmm OK. Mapping one function over a function has to produce a function, just like mapping a function over a `Maybe` has to produce a `Maybe` and mapping a function over a list has to produce a list. What does the type `fmap :: (a -> b) -> (r -> a) -> (r -> b)` for this instance tell us? Well, we see that it takes a function from `a` to `b` and a function from `r` to `a` and returns a function from `r` to `b`. Does this remind you of anything? Yes! Function composition! We pipe the output of `r -> a` into the input of `a -> b` to get a function `r -> b`, which is exactly what function composition is about. If you look at how the instance is defined above, you'll see that it's just function composition. Another way to write this instance would be:

```
instance                  where
         =
```

This makes the revelation that using `fmap` over functions is just composition sort of obvious. Do `:m + Control.Monad.Instances`, since that's where the instance is defined and then try playing with mapping over functions.

```
ghci>           3    100
```

```
        3    100  ::          =>   ->
ghci>        3    100  1
303
ghci>   3  `fmap`   100  $ 1
303
ghci>   3  .   100  $ 1
303
ghci>            .   3     100  1
"300"
```

We can call `fmap` as an infix function so that the resemblance to `.` is clear. In the second input line, we're mapping `(*3)` over `(+100)`, which results in a function that will take an input, call `(+100)` on that and then call `(*3)` on that result. We call that function with `1`.

How does the box analogy hold here? Well, if you stretch it, it holds. When we use `fmap (+3)` over `Just 3`, it's easy to imagine the `Maybe` as a box that has some contents on which we apply the function `(*3)`. But what about when we're doing `fmap (*3) (+100)`? Well, you can think of the function `(+100)` as a box that contains its eventual result. Sort of like how an I/O action can be thought of as a box that will go out into the real world and fetch some result. Using `fmap (*3)` on `(+100)` will create another function that acts like `(+100)`, only before producing a result, `(*3)` will be applied to that result. Now we can see how `fmap` acts just like `.` for functions.

The fact that `fmap` is function composition when used on functions isn't so terribly useful right now, but at least it's very interesting. It also bends our minds a bit and let us see how things that act more like computations than boxes (`IO` and `(->) r`) can be functors. The function being mapped over a computation results in the same computation but the result of that computation is modified with the function.

Before we go on to the rules that `fmap` should follow, let's think about the type of `fmap` once more. Its type is `fmap :: (a -> b) -> f a -> f b`. We're missing the class constraint `(Functor f) =>`, but we left it out here for brevity, because we're talking about functors anyway so we know what the `f` stands for. When we first learned about curried functions, we said that all Haskell functions actually take one parameter. A function `a -> b -> c` actually takes just one parameter of type `a` and then returns a function `b -> c`, which takes one parameter and returns a `c`. That's how if we call a function with too few parameters (i.e. partially apply it), we get back a function that takes the number of parameters that we left out (if we're thinking about functions as taking several parameters again). So `a -> b -> c` can be written as `a -> (b -> c)`, to make the currying more apparent.

In the same vein, if we write `fmap :: (a -> b) -> (f a -> f b)`, we can think of `fmap` not as a function that takes one function and a functor and returns a functor, but as a function that takes a function and returns a new function that's just like the old one, only it takes a functor as a parameter and returns a functor as the result. It takes an `a -> b` function and returns a function `f a -> f b`. This is called *lifting* a function. Let's play around with that idea by using GHCI's `:t` command:

```
ghci>          2
       2  ::                  =>     ->
ghci>                   3
              3  ::             =>     ->
```

The expression `fmap (*2)` is a function that takes a functor `f` over numbers and returns a functor over numbers. That functor can be a list, a `Maybe` , an `Either String`, whatever. The expression `fmap (replicate 3)` will take a functor over any type and return a functor over a list of elements of that type.

> When we say *a functor over numbers*, you can think of that as *a functor that has numbers in it*. The former is a bit fancier and more technically correct, but the latter is usually easier to get.

This is even more apparent if we partially apply, say, `fmap (++"!")` and then bind it to a name in GHCI.

You can think of `fmap` as either a function that takes a function and a functor and then maps that function over the functor, or you can think of it as a function that takes a function and lifts that function so that it operates on functors. Both views are correct and in Haskell, equivalent.

The type `fmap (replicate 3) :: (Functor f) => f a -> f [a]` means that the function will work on any functor. What exactly it will do depends on which functor we use it on. If we use `fmap (replicate 3)` on a list, the list's implementation for `fmap` will be chosen, which is just `map`. If we use it on a `Maybe a`, it'll apply `replicate 3` to the value inside the `Just`, or if it's `Nothing`, then it stays `Nothing`.

```
ghci>                3   1 2 3 4
  1 1 1   2 2 2   3 3 3   4 4 4
ghci>                3       4
     4 4 4
ghci>                3         "blah"
      "blah" "blah" "blah"
ghci>                3

ghci>                3       "foo"
     "foo"
```

Next up, we're going to look at the **functor laws**. In order for something to be a functor, it should satisfy some laws. All functors are expected to exhibit certain kinds of functor-like properties and behaviors. They should reliably behave as things that can be mapped over. Calling `fmap` on a functor should just map a function over the functor, nothing more. This behavior is described in the functor laws. There are two of them that all instances of `Functor` should abide by. They aren't enforced by Haskell automatically, so you have to test them out yourself.

The first functor law states that if we map the `id` function over a functor, the functor that we get back should be the same as the original functor. If we write that a bit more formally, it means that `fmap id = id` . So essentially, this says that if we do `fmap id` over a functor, it should be the same as just calling `id` on the functor. Remember, `id` is the identity function, which just returns its parameter unmodified. It can also be written as `\x -> x`. If we view the functor as something that can be mapped over, the `fmap id = id` law seems kind of trivial or obvious.

Let's see if this law holds for a few values of functors.

```
ghci>              3
     3
ghci>          3
     3
ghci>          1  5
 1 2 3 4 5
ghci>     1  5
 1 2 3 4 5
ghci>

ghci>
```

If we look at the implementation of `fmap` for, say, `Maybe`, we can figure out why the first functor law holds.

```
instance                where
                    =
                   =
```

We imagine that `id` plays the role of the `f` parameter in the implementation. We see that if wee `fmap id` over `Just x`, the result will be `Just (id x)`, and because `id` just returns its parameter, we can deduce that `Just (id x)` equals `Just x`. So now we know that if we map `id` over a `Maybe` value with a `Just` value constructor, we get that same value back.

Seeing that mapping `id` over a `Nothing` value returns the same value is trivial. So from these two equations in the implementation for `fmap`, we see that the law `fmap id = id` holds.

The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one. Formally written, that means that `fmap (f . g) = fmap f . fmap g`. Or to write it in another way, for any functor *F*, the following should hold: `fmap (f . g) F = fmap f (fmap g F)`.

If we can show that some type obeys both functor laws, we can rely on it having the same fundamental behaviors as other functors when it comes to mapping. We can know that when we use `fmap` on it, there won't be anything other than mapping going on behind the scenes and that it will act like a thing that can be mapped over, i.e. a functor. You figure out how the second law holds for some type by looking at the implementation of `fmap` for that type and then using the method that we used to check if `Maybe` obeys the first law.

If you want, we can check out how the second functor law holds for `Maybe`. If we do `fmap (f . g)` over `Nothing`, we get `Nothing`, because doing a `fmap` with any function over `Nothing` returns `Nothing`. If we do `fmap f (fmap g Nothing)`, we get `Nothing`, for the same reason. OK, seeing how the second law holds for `Maybe` if it's a `Nothing` value is pretty easy, almost trivial.

How about if it's a `Just` *`something`* value? Well, if we do `fmap (f . g) (Just x)`, we see from the implementation that it's implemented as `Just ((f . g) x)`, which is, of course, `Just (f (g x))`. If we do `fmap f (fmap g (Just x))`, we see from the implementation that `fmap g (Just x)` is `Just (g x)`. Ergo, `fmap f (fmap g (Just x))` equals `fmap f (Just (g x))` and from the implementation we see that this equals `Just (f (g x))`.

If you're a bit confused by this proof, don't worry. Be sure that you understand how function composition works. Many times, you can intuitively see how these laws hold because the types act like containers or functions. You can also just try them on a bunch of different values of a type and be able to say with some certainty that a type does indeed obey the laws.

Let's take a look at a pathological example of a type constructor being an instance of the `Functor` typeclass but not really being a functor, because it doesn't satisfy the laws. Let's say that we have a type:

```
data          =           |             deriving
```

The C here stands for *counter*. It's a data type that looks much like `Maybe a`, only the `Just` part holds two fields instead of one. The first field in the `CJust` value constructor will always have a type of `Int`, and it will be some sort of counter and the second

field is of type `a`, which comes from the type parameter and its type will, of course, depend on the concrete type that we choose for `CMaybe a`. Let's play with our new type to get some intuition for it.

```
ghci>
ghci>       0 "haha"
      0 "haha"
ghci>
       ::
ghci>          0 "haha"
      0 "haha" ::
ghci>       100  1 2 3
      100  1 2 3
```

If we use the `CNothing` constructor, there are no fields, and if we use the `CJust` constructor, the first field is an integer and the second field can be any type. Let's make this an instance of `Functor` so that everytime we use `fmap`, the function gets applied to the second field, whereas the first field gets increased by 1.

```
instance                where
                    =
                              =                1
```

This is kind of like the instance implementation for `Maybe`, except that when we do `fmap` over a value that doesn't represent an empty box (a `CJust` value), we don't just apply the function to the contents, we also increase the counter by 1. Everything seems cool so far, we can even play with this a bit:

```
ghci>          "ha"          0 "ho"
      1 "hoha"
ghci>          "he"            "ha"          0 "ho"
      2 "hohahe"
ghci>          "blah"
```

Does this obey the functor laws? In order to see that something doesn't obey a law, it's enough to find just one counter-example.

```
ghci>                0 "haha"
      1 "haha"
ghci>           0 "haha"
      0 "haha"
```

Ah! We know that the first functor law states that if we map `id` over a functor, it should be the same as just calling `id` with the same functor, but as we've seen from this example, this is not true for our `CMaybe` functor. Even though it's part of the `Functor` typeclass, it doesn't obey the functor laws and is therefore not a functor. If someone used our `CMaybe` type as a functor, they would expect it to obey the functor laws like a good functor. But `CMaybe` fails at being a functor even though it pretends to be one, so using it as a functor might lead to some faulty code. When we use a functor, it shouldn't matter if we first compose a few functions and then map them over the functor or if we just map each function over a functor in succession. But with `CMaybe`, it matters, because it keeps track of how many times it's been mapped over. Not cool! If we wanted `CMaybe` to obey the functor laws, we'd have to make it so that the `Int` field stays the same when we use `fmap`.

At first, the functor laws might seem a bit confusing and unnecessary, but then we see that if we know that a type obeys both laws, we can make certain assumptions about how it will act. If a type obeys the functor laws, we know that calling `fmap` on a value of that type will only map the function over it, nothing more. This leads to code that is more abstract and extensible, because we can use laws to reason about behaviors that any functor should have and make functions that operate reliably on any functor.

All the `Functor` instances in the standard library obey these laws, but you can check for yourself if you don't believe me. And the next time you make a type an instance of `Functor`, take a minute to make sure that it obeys the functor laws. Once you've dealt with enough functors, you kind of intuitively see the properties and behaviors that they have in common and it's not hard to intuitively see if a type obeys the functor laws. But even without the intuition, you can always just go over the implementation line by line and see if the laws hold or try to find a counter-example.

We can also look at functors as things that output values in a context. For instance, `Just 3` outputs the value `3` in the context that it might or not output any values at all. `[1,2,3]` outputs three values—`1`, `2`, and `3`, the context is that there may be multiple values or no values. The function `(+3)` will output a value, depending on which parameter it is given.

If you think of functors as things that output values, you can think of mapping over functors as attaching a transformation to the output of the functor that changes the value. When we do `fmap (+3) [1,2,3]`, we attach the transformation `(+3)` to the output of `[1,2,3]`, so whenever we look at a number that the list outputs, `(+3)` will be applied to it. Another example is mapping over functions. When we do `fmap (+3) (*3)`, we attach the transformation `(+3)` to the eventual output of `(*3)`. Looking at it this way gives us some intuition as to why using `fmap` on functions is just composition (`fmap (+3) (*3)` equals `(+3) . (*3)`, which equals `\x -> ((x*3)+3)`), because we take a function like `(*3)` then we attach the transformation `(+3)` to its output. The result is still a function, only when we give it a number, it will be multiplied by three and then it will go through the attached transformation where it will be added to three. This is what happens with composition.

Applicative functors

In this section, we'll take a look at applicative functors, which are beefed up functors, represented in Haskell by the `Applicative` typeclass, found in the `Control.Applicative` module.

As you know, functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on. If a function is of type `a -> b -> c`, we usually say that it takes two parameters and returns a `c`, but actually it takes an `a` and returns a function `b -> c`. That's why we can call a function as `f x y` or as `(f x) y`. This mechanism is what enables us to partially apply functions by just calling them with too few parameters, which results in functions that we can then pass on to other functions.

So far, when we were mapping functions over functors, we usually mapped functions that take only one parameter. But what happens when we map a function like `*`, which takes two parameters, over a functor? Let's take a look at a couple of concrete examples of this. If we have `Just 3` and we do `fmap (*) (Just 3)`, what do we get? From the instance implementation of `Maybe` for `Functor`, we know that if it's a `Just` *`something`* value, it will apply the function to the *`something`* inside the `Just`. Therefore, doing `fmap (*) (Just 3)` results in `Just ((*) 3)`, which can also be written as `Just (3 *)` if we use sections. Interesting! We get a function wrapped in a `Just`!

```
ghci> "hey"
"hey"  ::              ->
ghci> 'a'
'a'  ::            ->
ghci> "A LIST OF CHARS"
"A LIST OF CHARS" ::       ->
ghci>          ->   +   /     3 4 5 6
->   +   /     3 4 5 6  ::                =>    ->   ->
```

If we map `compare`, which has a type of `(Ord a) => a -> a -> Ordering` over a list of characters, we get a list of functions of type `Char -> Ordering`, because the function `compare` gets partially applied with the characters in the list. It's not a list of

`(Ord a) => a -> Ordering` function, because the first `a` that got applied was a `Char` and so the second `a` has to decide to be of type `Char`.

We see how by mapping "multi-parameter" functions over functors, we get functors that contain functions inside them. So now what can we do with them? Well for one, we can map functions that take these functions as parameters over them, because whatever is inside a functor will be given to the function that we're mapping over it as a parameter.

```
ghci> let   =           1 2 3 4
ghci>
  ::          ->
ghci>          ->   9
 9 18 27 36
```

But what if we have a functor value of `Just (3 *)` and a functor value of `Just 5` and we want to take out the function from `Just (3 *)` and map it over `Just 5`? With normal functors, we're out of luck, because all they support is just mapping normal functions over existing functors. Even when we mapped `\f -> f 9` over a functor that contained functions inside it, we were just mapping a normal function over it. But we can't map a function that's inside a functor over another functor with what `fmap` offers us. We could pattern-match against the `Just` constructor to get the function out of it and then map it over `Just 5`, but we're looking for a more general and abstract way of doing that, which works across functors.

Meet the `Applicative` typeclass. It lies in the `Control.Applicative` module and it defines two methods, `pure` and `<*>`. It doesn't provide a default implementation for any of them, so we have to define them both if we want something to be an applicative functor. The class is defined like so:

```
class             =>              where
         ::   ->
          ::      ->    ->     ->
```

This simple three line class definition tells us a lot! Let's start at the first line. It starts the definition of the `Applicative` class and it also introduces a class constraint. It says that if we want to make a type constructor part of the `Applicative` typeclass, it has to be in `Functor` first. That's why if we know that if a type constructor is part of the `Applicative` typeclass, it's also in `Functor`, so we can use `fmap` on it.

The first method it defines is called `pure`. Its type declaration is `pure :: a -> f a`. `f` plays the role of our applicative functor instance here. Because Haskell has a very good type system and because everything a function can do is take some parameters and return some value, we can tell a lot from a type declaration and this is no exception. `pure` should take a value of any type and return an applicative functor with that value inside it. When we say *inside it*, we're using the box analogy again, even though we've seen that it doesn't always stand up to scrutiny. But the `a -> f a` type declaration is still pretty descriptive. We take a value and we wrap it in an applicative functor that has that value as the result inside it.

A better way of thinking about `pure` would be to say that it takes a value and puts it in some sort of default (or pure) context—a minimal context that still yields that value.

The `<*>` function is really interesting. It has a type declaration of `f (a -> b) -> f a -> f b`. Does this remind you of anything? Of course, `fmap :: (a -> b) -> f a -> f b`. It's a sort of a beefed up `fmap`. Whereas `fmap` takes a function and a functor and applies the function inside the functor, `<*>` takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one. When I say *extract*, I actually sort of mean *run* and then extract, maybe even *sequence*. We'll see why soon.

Let's take a look at the `Applicative` instance implementation for `Maybe`.

```
instance                  where
         =
            <*>    =
```

```
<*>           =
```

Again, from the class definition we see that the `f` that plays the role of the applicative functor should take one concrete type as a parameter, so we write `instance Applicative Maybe where` instead of writing `instance Applicative (Maybe a) where`.

First off, `pure`. We said earlier that it's supposed to take something and wrap it in an applicative functor. We wrote `pure = Just`, because value constructors like `Just` are normal functions. We could have also written `pure x = Just x`.

Next up, we have the definition for `<*>`. We can't extract a function out of a `Nothing`, because it has no function inside it. So we say that if we try to extract a function from a `Nothing`, the result is a `Nothing`. If you look at the class definition for `Applicative`, you'll see that there's a `Functor` class constraint, which means that we can assume that both of `<*>`'s parameters are functors. If the first parameter is not a `Nothing`, but a `Just` with some function inside it, we say that we then want to map that function over the second parameter. This also takes care of the case where the second parameter is `Nothing`, because doing `fmap` with any function over a `Nothing` will return a `Nothing`.

So for `Maybe`, `<*>` extracts the function from the left value if it's a `Just` and maps it over the right value. If any of the parameters is `Nothing`, `Nothing` is the result.

OK cool great. Let's give this a whirl.

```
ghci>        3  <*>      9
     12
ghci>        3  <*>      10
     13
ghci>        3  <*>      9
     12
ghci>         "hahah"  <*>

ghci>         <*>      "woot"
```

We see how doing `pure (+3)` and `Just (+3)` is the same in this case. Use `pure` if you're dealing with `Maybe` values in an applicative context (i.e. using them with `<*>`), otherwise stick to `Just`. The first four input lines demonstrate how the function is extracted and then mapped, but in this case, they could have been achieved by just mapping unwrapped functions over functors. The last line is interesting, because we try to extract a function from a `Nothing` and then map it over something, which of course results in a `Nothing`.

With normal functors, you can just map a function over a functor and then you can't get the result out in any general way, even if the result is a partially applied function. Applicative functors, on the other hand, allow you to operate on several functors with a single function. Check out this piece of code:

```
ghci>          <*>      3 <*>      5
     8
ghci>          <*>      3 <*>

ghci>          <*>         <*>      5
```

What's going on here? Let's take a look, step by step. `<*>` is left-associative, which means that `pure (+) <*> Just 3 <*> Just 5` is the same as `(pure (+) <*> Just 3) <*> Just 5`. First, the `+` function is put in a functor, which is in this case a `Maybe` value that contains the function. So at first, we have `pure (+)`, which is `Just (+)`. Next, `Just (+) <*> Just 3` happens. The result of this is `Just (3+)`. This is because of partial application. Only applying `3` to the `+` function results in a function that takes one parameter and adds 3 to it. Finally, `Just (3+) <*> Just 5` is carried out, which results in a `Just 8`.

Isn't this awesome?! Applicative functors and the applicative style of doing `pure f <*> x <*> y <*> ...` allow us to take a function that expects parameters that aren't necessarily wrapped in functors and use that function to operate on several values that are in functor contexts. The function can take as many parameters as we want, because it's always partially applied step by step between occurences of `<*>`.

This becomes even more handy and apparent if we consider the fact that `pure f <*> x` equals `fmap f x`. This is one of the applicative laws. We'll take a closer look at them later, but for now, we can sort of intuitively see that this is so. Think about it, it makes sense. Like we said before, `pure` puts a value in a default context. If we just put a function in a default context and then extract and apply it to a value inside another applicative functor, we did the same as just mapping that function over that applicative functor. Instead of writing `pure f <*> x <*> y <*> ...`, we can write `fmap f x <*> y <*> ...`. This is why `Control.Applicative` exports a function called `<$>`, which is just `fmap` as an infix operator. Here's how it's defined:

```
    ::              =>     ->     ->      ->
<$>   =
```

Yo! Quick reminder: type variables are independent of parameter names or other value names. The `f` in the function declaration here is a type variable with a class constraint saying that any type constructor that replaces `f` should be in the `Functor` typeclass. The `f` in the function body denotes a function that we map over `x`. The fact that we used `f` to represent both of those doesn't mean that they somehow represent the same thing.

By using `<$>`, the applicative style really shines, because now if we want to apply a function `f` between three applicative functors, we can write `f <$> x <*> y <*> z`. If the parameters weren't applicative functors but normal values, we'd write `f x y z`.

Let's take a closer look at how this works. We have a value of `Just "johntra"` and a value of `Just "volta"` and we want to join them into one `String` inside a `Maybe` functor. We do this:

```
ghci>       <$>       "johntra" <*>       "volta"
     "johntravolta"
```

Before we see how this happens, compare the above line with this:

```
ghci>       "johntra" "volta"
"johntravolta"
```

Awesome! To use a normal function on applicative functors, just sprinkle some `<$>` and `<*>` about and the function will operate on applicatives and return an applicative. How cool is that?

Anyway, when we do `(++) <$> Just "johntra" <*> Just "volta"`, first `(++)`, which has a type of `(++) :: [a] -> [a] -> [a]` gets mapped over `Just "johntra"`, resulting in a value that's the same as `Just ("johntra"++)` and has a type of `Maybe ([Char] -> [Char])`. Notice how the first parameter of `(++)` got eaten up and how the as turned into `Char`s. And now `Just ("johntra"++) <*> Just "volta"` happens, which takes the function out of the `Just` and maps it over `Just "volta"`, resulting in `Just "johntravolta"`. Had any of the two values been `Nothing`, the result would have also been `Nothing`.

So far, we've only used `Maybe` in our examples and you might be thinking that applicative functors are all about `Maybe`. There are loads of other instances of `Applicative`, so let's go and meet them!

Lists (actually the list type constructor, `[]`) are applicative functors. What a suprise! Here's how `[]` is an instance of `Applicative`:

```
instance               where
```

```
          =
<*>     =       |    <-        <-
```

Earlier, we said that `pure` takes a value and puts it in a default context. Or in other words, a minimal context that still yields that value. The minimal context for lists would be the empty list, `[]`, but the empty list represents the lack of a value, so it can't hold in itself the value that we used `pure` on. That's why `pure` takes a value and puts it in a singleton list. Similarly, the minimal context for the `Maybe` applicative functor would be a `Nothing`, but it represents the lack of a value instead of a value, so `pure` is implemented as `Just` in the instance implementation for `Maybe`.

```
ghci>       "Hey" ::
 "Hey"
ghci>       "Hey" ::
     "Hey"
```

What about `<*>`? If we look at what `<*>`'s type would be if it were limited only to lists, we get `(<*>) :: [a -> b] -> [a] -> [b]`. It's implemented with a list comprehension. `<*>` has to somehow extract the function out of its left parameter and then map it over the right parameter. But the thing here is that the left list can have zero functions, one function, or several functions inside it. The right list can also hold several values. That's why we use a list comprehension to draw from both lists. We apply every possible function from the left list to every possible value from the right list. The resulting list has every possible combination of applying a function from the left list to a value in the right one.

```
ghci>    0    100    2   <*>  1 2 3
 0 0 0 101 102 103 1 4 9
```

The left list has three functions and the right list has three values, so the resulting list will have nine elements. Every function in the left list is applied to every function in the right one. If we have a list of functions that take two parameters, we can apply those functions between two lists.

```
ghci>           <*>  1 2  <*>  3 4
 4 5 5 6 3 4 6 8
```

Because `<*>` is left-associative, `[(+),(*)] <*> [1,2]` happens first, resulting in a list that's the same as `[(1+),(2+),(1*),(2*)]`, because every function on the left gets applied to every value on the right. Then, `[(1+),(2+),(1*),(2*)] <*> [3,4]` happens, which produces the final result.

Using the applicative style with lists is fun! Watch:

```
ghci>       <$>  "ha" "heh" "hmm"  <*>  "?" "!" "."
 "ha?" "ha!" "ha." "heh?" "heh!" "heh." "hmm?" "hmm!" "hmm."
```

Again, see how we used a normal function that takes two strings between two applicative functors of strings just by inserting the appropriate applicative operators.

You can view lists as non-deterministic computations. A value like `100` or `"what"` can be viewed as a deterministic computation that has only one result, whereas a list like `[1,2,3]` can be viewed as a computation that can't decide on which result it wants to have, so it presents us with all of the possible results. So when you do something like `(+) <$> [1,2,3] <*> [4,5,6]`, you can think of it as adding together two non-deterministic computations with `+`, only to produce another non-deterministic computation that's even less sure about its result.

Using the applicative style on lists is often a good replacement for list comprehensions. In the second chapter, we wanted to see all the possible products of `[2,5,10]` and `[8,10,11]`, so we did this:

```
ghci>        |    <-   2 5 10      <-   8 10 11
 16 20 22 40 50 55 80 100 110
```

We're just drawing from two lists and applying a function between every combination of elements. This can be done in the applicative style as well:

```
ghci>      <$>   2 5 10   <*>   8 10 11
 16 20 22 40 50 55 80 100 110
```

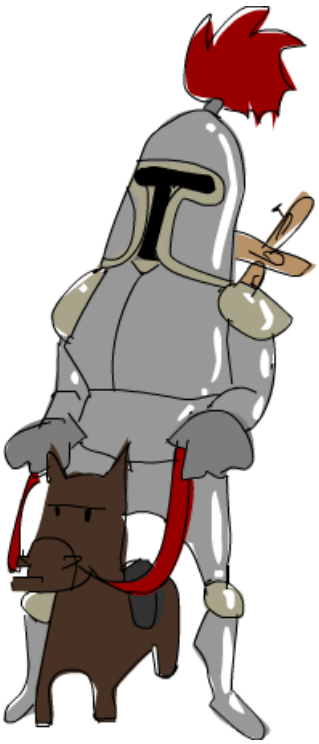
This seems clearer to me, because it's easier to see that we're just calling `*` between two non-deterministic computations. If we wanted all possible products of those two lists that are more than 50, we'd just do:

```
ghci>           50   $      <$>   2 5 10   <*>   8 10 11
 55 80 100 110
```

It's easy to see how `pure f <*> xs` equals `fmap f xs` with lists. `pure f` is just `[f]` and `[f] <*> xs` will apply every function in the left list to every value in the right one, but there's just one function in the left list, so it's like mapping.

Another instance of `Applicative` that we've already encountered is `IO`. This is how the instance is implemented:

```
instance                  where
         =
       <*>    = do
            <-
            <-
```



Since `pure` is all about putting a value in a minimal context that still holds it as its result, it makes sense that `pure` is just `return`, because `return` does exactly that; it makes an I/O action that doesn't do anything, it just yields some value as its result, but it doesn't really do any I/O operations like printing to the terminal or reading from a file.

If `<*>` were specialized for `IO` it would have a type of `(<*>) :: IO (a -> b) -> IO a -> IO b`. It would take an I/O action that yields a function as its result and another I/O action and create a new I/O action from those two that, when performed, first performs the first one to get the function and then performs the second one to get the value and then it would yield that function applied to the value as its result. We used *do* syntax to implement it here. Remember, *do* syntax is about taking several I/O actions and gluing them into one, which is exactly what we do here.

With `Maybe` and `[]`, we could think of `<*>` as simply extracting a function from its left parameter and then sort of applying it over the right one. With `IO`, extracting is still in the game, but now we also have a notion of *sequencing*, because we're taking two I/O actions and we're sequencing, or gluing, them into one. We have to extract the function from the first I/O action, but to extract a result from an I/O action, it has to be performed.

Consider this:

```
        ::
        = do
      <-
      <-
           $    ++
```

This is an I/O action that will prompt the user for two lines and yield as its result those two lines concatenated. We achieved it by gluing together two `getLine` I/O actions and a `return`, because we wanted our new glued I/O action to hold the result of `a ++ b`. Another way of writing this would be to use the applicative style.

```
::
=       <$>          <*>
```

What we were doing before was making an I/O action that applied a function between the results of two other I/O actions, and this is the same thing. Remember, `getLine` is an I/O action with the type `getLine :: IO String`. When we use `<*>` between two applicative functors, the result is an applicative functor, so this all makes sense.

If we regress to the box analogy, we can imagine `getLine` as a box that will go out into the real world and fetch us a string. Doing `(++) <$> getLine <*> getLine` makes a new, bigger box that sends those two boxes out to fetch lines from the terminal and then presents the concatenation of those two lines as its result.

The type of the expression `(++) <$> getLine <*> getLine` is `IO String`, which means that this expression is a completely normal I/O action like any other, which also holds a result value inside it, just like other I/O actions. That's why we can do stuff like:

```
= do
  <-       <$>          <*>
         $ "The two lines concatenated turn out to be: " ++
```

If you ever find yourself binding some I/O actions to names and then calling some function on them and presenting that as the result by using `return`, consider using the applicative style because it's arguably a bit more concise and terse.

Another instance of `Applicative` is `(->) r`, so functions. They are rarely used with the applicative style outside of code golf, but they're still interesting as applicatives, so let's take a look at how the function instance is implemented.

If you're confused about what `(->) r` means, check out the previous section where we explain how `(-> r)` is a functor.

```
instance                        where
         =      ->
      <*>   =     ->
```

When we wrap a value into an applicative functor with `pure`, the result it yields always has to be that value. A minimal default context that still yields that value as a result. That's why in the function instance implementation, `pure` takes a value and creates a function that ignores its parameter and always returns that value. If we look at the type for `pure`, but specialized for the `(->) r` instance, it's `pure :: a -> (r -> a)`.

```
ghci>       3  "blah"
3
```

Because of currying, function application is left-associative, so we can omit the parentheses.

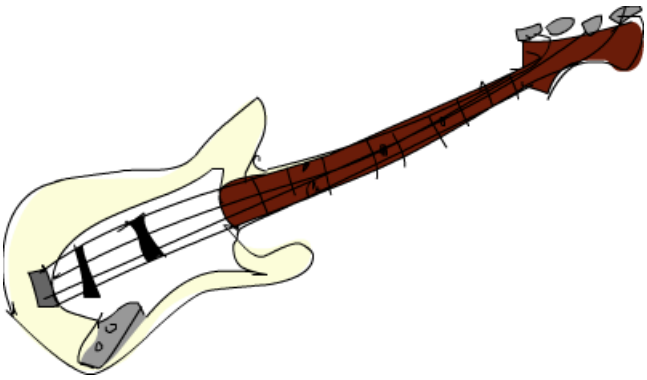
```
ghci>      3 "blah"
3
```

The instance implementation for `<*>` is a bit cryptic, so it's best if we just take a look at how to use functions as applicative functors in the applicative style.

```
ghci>        <$>   3  <*>   100
    <$>   3  <*>   100  ::          =>   ->
ghci>     <$>   3  <*>   100  $ 5
508
```

Calling `<*>` with two applicative functors results in an applicative functor, so if we use it on two functions, we get back a function. So what goes on here? When we do `(+) <$> (+3) <*> (*100)`, we're making a function that will use `+` on the results of `(+3)` and `(*100)` and return that. To demonstrate on a real example, when we did `(+) <$> (+3) <*> (*100) $ 5`, the `5` first got applied to `(+3)` and `(*100)`, resulting in `8` and `500`. Then, `+` gets called with `8` and `500`, resulting in `508`.

```
ghci>         ->          <$>   3  <*>   2  <*>   2  $ 5
 8.0 10.0 2.5
```



Same here. We create a function that will call the function `\x y z -> [x,y,z]` with the eventual results from `(+3)`, `(*2)` and `(/2)`. The `5` gets fed to each of the three functions and then `\x y z -> [x, y, z]` gets called with those results.

You can think of functions as boxes that contain their eventual results, so doing `k <$> f <*> g` creates a function that will call `k` with the eventual results from `f` and `g`. When we do something like `(+) <$> Just 3 <*> Just 5`, we're using `+` on values that might or might not be there, which also results in a value that might or might not be there. When we do `(+) <$> (+10) <*> (+5)`, we're using `+` on the future return values of `(+10)` and `(+5)` and the result is also something that will produce a value only when called with a parameter.

We don't often use functions as applicatives, but this is still really interesting. It's not very important that you get how the `(->) r` instance for `Applicative` works, so don't despair if you're not getting this right now. Try playing with the applicative style and functions to build up an intuition for functions as applicatives.

An instance of `Applicative` that we haven't encountered yet is `ZipList`, and it lives in `Control.Applicative`.

It turns out there are actually more ways for lists to be applicative functors. One way is the one we already covered, which says that calling `<*>` with a list of functions and a list of values results in a list which has all the possible combinations of applying functions from the left list to the values in the right list. If we do `[(+3),(*2)] <*> [1,2]`, `(+3)` will be applied to both `1` and `2` and `(*2)` will also be applied to both `1` and `2`, resulting in a list that has four elements, namely `[4,5,2,4]`.

However, `[(+3),(*2)] <*> [1,2]` could also work in such a way that the first function in the left list gets applied to the first value in the right one, the second function gets applied to the second value, and so on. That would result in a list with two values, namely `[4,4]`. You could look at it as `[1 + 3, 2 * 2]`.

Because one type can't have two instances for the same typeclass, the `ZipList a` type was introduced, which has one constructor `ZipList` that has just one field, and that field is a list. Here's the instance:

```
instance                    where
              =
                  <*>           =                       ->
```

`<*>` does just what we said. It applies the first function to the first value, the second function to the second value, etc. This is done with `zipWith (\f x -> f x) fs xs`. Because of how `zipWith` works, the resulting list will be as long as the shorter of the two lists.

`pure` is also interesting here. It takes a value and puts it in a list that just has that value repeating indefinitely. `pure "haha"` results in `ZipList ["haha","haha","haha"...`. This might be a bit confusing since we said that `pure` should put a value in a minimal context that still yields that value. And you might be thinking that an infinite list of something is hardly minimal. But it makes sense with zip lists, because it has to produce the value on every position. This also satisfies the law that `pure f <*> xs` should equal `fmap f xs`. If `pure 3` just returned `ZipList [3]`, `pure (*2) <*> ZipList [1,5,10]` would result in `ZipList [2]`, because the resulting list of two zipped lists has the length of the shorter of the two. If we zip a finite list with an infinite list, the length of the resulting list will always be equal to the length of the finite list.

So how do zip lists work in an applicative style? Let's see. Oh, the `ZipList a` type doesn't have a `Show` instance, so we have to use the function to extract a raw list out of a zip list.

```
ghci>            $      <$>          1 2 3   <*>          100 100 100
 101 102 103
ghci>            $      <$>          1 2 3   <*>          100 100
 101 102 103
ghci>            $      <$>          1 2 3 4 5 3   <*>          5 3 1 2
 5 3 3 4
ghci>            $       <$>         "dog" <*>          "cat" <*>          "rat"
  'd' 'c' 'r'   'o' 'a' 'a'   'g' 't' 't'
```

The `(,,)` function is the same as `\x y z -> (x,y,z)`. Also, the `(,)` function is the same as `\x y -> (x,y)`.

Aside from `zipWith`, the standard library has functions such as `zipWith3`, `zipWith4`, all the way up to 7. `zipWith` takes a function that takes two parameters and zips two lists with it. `zipWith3` takes a function that takes three parameters and zips three lists with it, and so on. By using zip lists with an applicative style, we don't have to have a separate zip function for each number of lists that we want to zip together. We just use the applicative style to zip together an arbitrary amount of lists with a function, and that's pretty cool.

`Control.Applicative` defines a function that's called , which has a type of `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c` . It's defined like this:

```
        ::                  =>     ->    ->     ->      ->      ->
              =    <$>    <*>
```

Nothing special, it just applies a function between two applicatives, hiding the applicative style that we've become familiar with. The reason we're looking at it is because it clearly showcases why applicative functors are more powerful than just ordinary functors. With ordinary functors, we can just map functions over one functor. But with applicative functors, we can apply a function between several functors. It's also interesting to look at this function's type as `(a -> b -> c) -> (f a -> f b -> f c)`. When we look at it like this, we can say that `liftA2` takes a normal binary function and promotes it to a function that operates on two functors.

Here's an interesting concept: we can take two applicative functors and combine them into one applicative functor that has inside it the results of those two applicative functors in a list. For instance, we have `Just 3` and `Just 4`. Let's assume that the second one has a singleton list inside it, because that's really easy to achieve:

```
ghci>           ->             4
      4
```

OK, so let's say we have `Just 3` and `Just [4]`. How do we get `Just [3,4]`? Easy.

```
ghci>                   3          4
      3 4
ghci>      <$>       3 <*>        4
```

```
3 4
```

Remember, `:` is a function that takes an element and a list and returns a new list with that element at the beginning. Now that we have `Just [3,4]`, could we combine that with `Just 2` to produce `Just [2,3,4]`? Of course we could. It seems that we can combine any amount of applicatives into one applicative that has a list of the results of those applicatives inside it. Let's try implementing a function that takes a list of applicatives and returns an applicative that has a list as its result value. We'll call it `sequenceA`.

```
::                  =>        ->
  =
      =      <$>    <*>
```

Ah, recursion! First, we look at the type. It will transform a list of applicatives into an applicative with a list. From that, we can lay some groundwork for an edge condition. If we want to turn an empty list into an applicative with a list of results, well, we just put an empty list in a default context. Now comes the recursion. If we have a list with a head and a tail (remember, `x` is an applicative and `xs` is a list of them), we call `sequenceA` on the tail, which results in an applicative with a list. Then, we just prepend the value inside the applicative `x` into that applicative with a list, and that's it!

So if we do `sequenceA [Just 1, Just 2]`, that's `(:) <$> Just 1 <*> sequenceA [Just 2]` . That equals `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])`. Ah! We know that `sequenceA []` ends up as being `Just []`, so this expression is now `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])`, which is `(:) <$> Just 1 <*> Just [2]`, which is `Just [1,2]`!

Another way to implement `sequenceA` is with a fold. Remember, pretty much any function where we go over a list element by element and accumulate a result along the way can be implemented with a fold.

```
::                  =>        ->
=
```

We approach the list from the right and start off with an accumulator value of `pure []`. We do `liftA2 (:)` between the accumulator and the last element of the list, which results in an applicative that has a singleton in it. Then we do `liftA2 (:)` with the now last element and the current accumulator and so on, until we're left with just the accumulator, which holds a list of the results of all the applicatives.

Let's give our function a whirl on some applicatives.

```
ghci>                3       2       1
     3 2 1
ghci>                3               1

ghci>              3    2    1   3
 6 5 4
ghci>            1 2 3   4 5 6
  1 4   1 5   1 6   2 4   2 5   2 6   3 4   3 5   3 6
ghci>            1 2 3   4 5 6   3 4 4
```

Ah! Pretty cool. When used on `Maybe` values, `sequenceA` creates a `Maybe` value with all the results inside it as a list. If one of the values was `Nothing`, then the result is also a `Nothing`. This is cool when you have a list of `Maybe` values and you're interested in the values only if none of them is a `Nothing`.

When used with functions, `sequenceA` takes a list of functions and returns a function that returns a list. In our example, we made a function that took a number as a parameter and applied it to each function in the list and then returned a list of results. `sequenceA [(+3),(+2),(+1)] 3` will call `(+3)` with `3`, `(+2)` with `3` and `(+1)` with `3` and present all those results as a list.

Doing `(+) <$> (+3) <*> (*2)` will create a function that takes a parameter, feeds it to both `(+3)` and `(*2)` and then calls `+` with those two results. In the same vein, it makes sense that `sequenceA [(+3),(*2)]` makes a function that takes a parameter and feeds it to all of the functions in the list. Instead of calling `+` with the results of the functions, a combination of `:` and `pure []` is used to gather those results in a list, which is the result of that function.

Using `sequenceA` is cool when we have a list of functions and we want to feed the same input to all of them and then view the list of results. For instance, we have a number and we're wondering whether it satisfies all of the predicates in a list. One way to do that would be like so:

```
ghci>         ->   7     4    10
ghci>     $         ->   7     4    10
```

Remember, `and` takes a list of booleans and returns `True` if they're all `True`. Another way to achieve the same thing would be with `sequenceA`:

```
ghci>              4    10       7
ghci>     $              4    10       7
```

`sequenceA [(>4),(<10),odd]` creates a function that will take a number and feed it to all of the predicates in `[(>4),(<10),odd]` and return a list of booleans. It turns a list with the type `(Num a) => [a -> Bool]` into a function with the type `(Num a) => a -> [Bool]`. Pretty neat, huh?

Because lists are homogenous, all the functions in the list have to be functions of the same type, of course. You can't have a list like `[ord, (+3)]`, because `ord` takes a character and returns a number, whereas `(+3)` takes a number and returns a number.

When used with `[]`, `sequenceA` takes a list of lists and returns a list of lists. Hmm, interesting. It actually creates lists that have all possible combinations of their elements. For illustration, here's the above done with `sequenceA` and then done with a list comprehension:

```
ghci>             1 2 3   4 5 6
  1 4   1 5   1 6   2 4   2 5   2 6   3 4   3 5   3 6
ghci>         |   <-  1 2 3     <-  4 5 6
  1 4   1 5   1 6   2 4   2 5   2 6   3 4   3 5   3 6
ghci>             1 2   3 4
  1 3   1 4   2 3   2 4
ghci>         |   <-  1 2     <-  3 4
  1 3   1 4   2 3   2 4
ghci>             1 2   3 4   5 6
  1 3 5   1 3 6   1 4 5   1 4 6   2 3 5   2 3 6   2 4 5   2 4 6
ghci>           |   <-  1 2     <-  3 4     <-  5 6
  1 3 5   1 3 6   1 4 5   1 4 6   2 3 5   2 3 6   2 4 5   2 4 6
```

This might be a bit hard to grasp, but if you play with it for a while, you'll see how it works. Let's say that we're doing `sequenceA [[1,2],[3,4]]`. To see how this happens, let's use the `sequenceA (x:xs) = (:) <$> x <*> sequenceA xs` definition of `sequenceA` and the edge condition `sequenceA [] = pure []`. You don't have to follow this evaluation, but it might help you if have trouble imagining how `sequenceA` works on lists of lists, because it can be a bit mind-bending.

- We start off with `sequenceA [[1,2],[3,4]]`
- That evaluates to `(:) <$> [1,2] <*> sequenceA [[3,4]]`
- Evaluating the inner `sequenceA` further, we get `(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])`
- We've reached the edge condition, so this is now `(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])`

- Now, we evaluate the `(:) <$> [3,4] <*> [[]]` part, which will use `:` with every possible value in the left list (possible values are `3` and `4`) with every possible value on the right list (only possible value is `[]`), which results in `[3:[], 4:[]]`, which is `[[3],[4]]`. So now we have `(:) <$> [1,2] <*> [[3],[4]]`
- Now, `:` is used with every possible value from the left list (`1` and `2`) with every possible value in the right list (`[3]` and `[4]`), which results in `[1:[3], 1:[4], 2:[3], 2:[4]]`, which is `[[1,3],[1,4],[2,3],[2,4]`

When used with I/O actions, `sequenceA` is the same thing as `sequence`! It takes a list of I/O actions and returns an I/O action that will perform each of those actions and have as its result a list of the results of those I/O actions. That's because to turn an `[IO a]` value into an `IO [a]` value, to make an I/O action that yields a list of results when performed, all those I/O actions have to be sequenced so that they're then performed one after the other when evaluation is forced. You can't get the result of an I/O action without performing it.

```
ghci>


 "heyh" "ho" "woo"
```

Like normal functors, applicative functors come with a few laws. The most important one is the one that we already mentioned, namely that `pure f <*> x = fmap f x` holds. As an exercise, you can prove this law for some of the applicative functors that we've met in this chapter.The other functor laws are:

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

We won't go over them in detail right now because that would take up a lot of pages and it would probably be kind of boring, but if you're up to the task, you can take a closer look at them and see if they hold for some of the instances.

In conclusion, applicative functors aren't just interesting, they're also useful, because they allow us to combine different computations, such as I/O computations, non-deterministic computations, computations that might have failed, etc. by using the applicative style. Just by using `<$>` and `<*>` we can use normal functions to uniformly operate on any number of applicative functors and take advantage of the semantics of each one.

Newtype

Functionally Solving Problems Obsah