Domácí úkol 4: Bitové vektory

V tomto domácím úkolu si vyzkoušíte vytvořit o něco delší kód v Haskellu, odměnou vám za to mohou být až dva body do hodnocení. Vaším úkolem je naprogramovat několik funkcí pracujících s bitovými vektory. Bitový vektor je posloupnost `0` a `1` fixní délky, která reprezentuje číslo ve dvojkové soustavě. Délka bitového vektoru je často označována také jako bitová šířka. Většina operací s bitovými vektory, které budeme definovat, musí zachovávat délku bitového vektoru. Pokud toto pravidlo neplatí, tak je to u příslušné operace explicitně uvedeno. V této úloze budou bitové vektory reprezentovat nezáporná čísla bez znaménka (např. číslo 42 může být reprezentováno jako 6bitový vektor „101010“ nebo jako 8bitový „00101010“). Prázdný vektor reprezentuje číslo 0.

Kostru domácí úlohy si můžete stáhnout ze studijních materiálů.

Reprezentace bitových vektorů

Pro reprezentaci bitových vektorů si zvolte typ podle vašeho výběru splňující omezující podmínky popsané níže. Tuto reprezentaci (typ `Bitvec`) musíte definovat, jinak za celý úkol dostanete 0 bodů.

Pozor: K implementaci bitových vektorů **nesmíte** použít typ patřící do typové třídy `Num` (např: `Int` nebo `Integer`)! Toto omezení platí i pro implementaci jednotlivých funkcí, tedy nesmíte převádět `Bitvec` na instance typové třídy `Num` uvnitř žádné funkce (například sčítání bitových vektorů musíte implementovat po bitech, ne pomocí sčítání čísel). Tyto podmínky bude kontrolovat až cvičící. To znamená, že pokud použijete reprezentaci pomocí typu z třídy `Num` nebo převod `Bitvec` na instance této třídy použijete v některé z funkcí, ztratíte i body z automatizovaných testů (za celý domácí úkol tedy dostanete 0 bodů). Aritmetiku samozřejmě můžete používat pro záležitosti nesouvisející s hodnotou bitových vektorů, například můžete počítat s délkou bitových vektorů.

```
type Bitvec = {- Fill in here -}
```

Takto zadefinovaný typ `Bitvec` je zcela identický s vámi zvoleným typem. Něco podobného už znáte: typ `String` je také pouhým aliasem pro `[Char]` (definovaný jako `type String = [Char]`).

Můžete předpokládat, že funkce, které berou jako parametr `Bitvec`, nebudou testovány samostatně, ale vždy se budou využívat funkce `toBitvec` a `fromBitvec` (definované níže).

Funkce k implementaci

Následuje popis funkcí, které máte implementovat. V příkladech se držíme zavedeného formátu, v němž řádky začínající znakem „>“ označují vstup interpretu a ostatní řádky jeho výstup.

- `toBitvec :: String -> Bitvec`

Převede vstupní řetězec na `Bitvec` ve vámi zvolené reprezentaci (nejvyšší bit bitového vektoru je zadán jako první znak řetězce). Tedy například `"101"` reprezentuje 3bitový vektor s dekadickou hodnotou 5, `"00101010"` reprezentuje 8bitový vektor s dekadickou hodnotou 42.

Bitová šířka výsledného bitového vektoru je dána délkou zadaného řetězce. Řetězec tedy může začínat libovolným počtem nul. Můžete předpokládat, že vstupní řetězec bude obsahovat pouze znaky `0`, `1` a nic jiného.

- `fromBitvec :: Bitvec -> String`

Převede bitový vektor zpět na řetězec reprezentující binární zápis bitového vektoru dané délky.

Příklad:

```
> ( fromBitvec . toBitvec ) "0010"
"0010"
> ( fromBitvec . toBitvec ) ""
```

```
""
> ( fromBitvec . toBitvec ) "101010"
"101010"
```

- **`bvnot :: Bitvec -> Bitvec`**

Invertuje všechny bity v bitovém vektoru (všechny `0` změní na `1` a obráceně).

Příklad:

```
> ( fromBitvec . bvnot . toBitvec ) ""
""
> ( fromBitvec . bvnot . toBitvec ) "101010"
"010101"
> ( fromBitvec . bvnot . bvnot . toBitvec ) "00010"
"00010"
```

- **`bvand :: Bitvec -> Bitvec -> Bitvec`**

Aplikuje bitový **`and`** na vstupních parametrech. V případě, že vstupní bitové vektory nemají stejnou délku, bude výsledná délka odpovídat delšímu ze vstupních bitových vektorů. Kratší z bitových vektorů se pro výpočet „prodlouží“ na délku delšího, a to přidáním `0` na místo nejvyšších bitů (např: bitový vektor „0100“ se prodlouží na délku 8 tímto způsobem: „00000100“).

Příklad:

```
> fromBitvec ( bvand ( toBitvec "1100" ) ( toBitvec "1010" ) )
"1000"
> fromBitvec ( bvand ( toBitvec "1100" ) ( toBitvec "001010" ) )
"001000"
> fromBitvec ( bvand ( toBitvec "" ) ( toBitvec "1010" ) )
"0000"
> fromBitvec ( bvand ( toBitvec "" ) ( toBitvec "" ) )
""
```

- **`bvor :: Bitvec -> Bitvec -> Bitvec`**

Aplikuje bitový **`or`** na vstupních parametrech. V případě, že vstupní bitové vektory nemají stejnou délku, prodlužte kratší stejně jako u **`bvand`**.

Příklad:

```
> fromBitvec ( bvor ( toBitvec "1100" ) ( toBitvec "1010" ) )
"1110"
> fromBitvec ( bvor ( toBitvec "1100" ) ( toBitvec "001010" ) )
"001110"
> fromBitvec ( bvor ( toBitvec "" ) ( toBitvec "1010" ) )
"1010"
> fromBitvec ( bvor ( toBitvec "" ) ( toBitvec "" ) )
""
```

- **`bvxor :: Bitvec -> Bitvec -> Bitvec`**

Aplikuje bitový **`xor`** na vstupních parametrech (**`xor`** je jen jiné označení pro negaci ekvivalence). V případě, že vstupní bitové vektory nemají stejnou délku, prodlužte kratší stejně jako u **`bvand`**.

Příklad:

```
> fromBitvec ( bvxor ( toBitvec "1100" ) ( toBitvec "1010" ) )
"0110"
> fromBitvec ( bvxor ( toBitvec "1100" ) ( toBitvec "001010" ) )
"000110"
> fromBitvec ( bvxor ( toBitvec "" ) ( toBitvec "1010" ) )
"1010"
> fromBitvec ( bvxor ( toBitvec "" ) ( toBitvec "" ) )
""
```

- `resize :: Int -> Bitvec -> Bitvec`

Změní délku bitového vektoru na požadovanou délku. Přidá nuly na nejvyšší bity, zachová délku, nebo odstraní nejvyšší bity (podle potřeby). Argument bude nezáporný.

Příklad:

```
> ( fromBitvec . resize 5 . toBitvec ) ""
"00000"
> ( fromBitvec . resize 3 . toBitvec ) "101010"
"010"
> ( fromBitvec . resize 5 . toBitvec ) "00010"
"00010"
```

- `zero :: Int -> Bitvec`

Vytvoří bitový vektor zadané délky obsahující pouze nuly (můžete předpokládat, že zadaný argument bude nezáporný).

Příklad:

```
> ( fromBitvec . zero ) 5
"00000"
> ( fromBitvec . zero ) 0
""
```

- `rotRight :: Int -> Bitvec -> Bitvec`

Vykoná na bitovém vektoru bitovou rotaci doprava. Velikost rotace bude nezáporná, ale může být i větší než délka bitového vektoru.

Příklad:

```
> ( fromBitvec . rotRight 2 . toBitvec ) "1011100"
"0010111"
> ( fromBitvec . rotRight 0 . toBitvec ) "10110010"
"10110010"
> ( fromBitvec . rotRight 21 . toBitvec ) "0001"
"1000"
> ( fromBitvec . rotRight 0 . toBitvec ) ""
""
```

- `rotLeft :: Int -> Bitvec -> Bitvec`

Vykoná na bitovém vektoru bitovou rotaci doleva. Velikost rotace bude nezáporná, ale může být i větší než délka bitového vektoru.

Příklad:

```
> ( fromBitvec . rotLeft 2 . toBitvec ) "1011100"
"1110010"
> ( fromBitvec . rotLeft 0 . toBitvec ) "10110010"
"10110010"
> ( fromBitvec . rotLeft 21 . toBitvec ) "0001"
"0010"
> ( fromBitvec . rotLeft 0 . toBitvec ) ""
""
```

- **`shiftRight :: Int -> Bitvec -> Bitvec`**

Vykoná na bitovém vektoru bitový posun doprava. Velikost posunu bude nezáporná, ale může být i větší než délka bitového vektoru. Nezapomínejte, že délka bitového vektoru se nesmí změnit. Jedná se o logický bitový posun, tedy na místo nejvyšších bitů se „nasune“ nula.

Příklad:

```
> ( fromBitvec . shiftRight 2 . toBitvec ) "1011100"
"0010111"
> ( fromBitvec . shiftRight 0 . toBitvec ) "10110010"
"10110010"
> ( fromBitvec . shiftRight 21 . toBitvec ) "0001"
"0000"
```

- **`shiftLeft :: Int -> Bitvec -> Bitvec`**

Vykoná na bitovém vektoru bitový posun doleva. Velikost posunu bude nezáporná, ale může být i větší než délka bitového vektoru. Nezapomínejte, že délka bitového vektoru se nesmí změnit. Jedná se o logický bitový posun, tedy na místo nejnižších bitů se „nasune“ nula.

Příklad:

```
> ( fromBitvec . shiftLeft 2 . toBitvec ) "1011100"
"1110000"
> ( fromBitvec . shiftLeft 0 . toBitvec ) "10110010"
"10110010"
> ( fromBitvec . shiftLeft 21 . toBitvec ) "0001"
"0000"
```

- **`add :: Bitvec -> Bitvec -> Bitvec`**

Funkce vykoná aritmetický součet na zadaných bitových vektorech. V případě, že vstupní bitové vektory nemají stejnou délku, prodlužte kratší stejně jako u `bvand`.

Příklad:

```
> fromBitvec ( add ( toBitvec "1000" ) ( toBitvec "0010" ) )
"1010"
> fromBitvec ( add ( toBitvec "1010" ) ( toBitvec "0010" ) )
"1100"
> fromBitvec ( add ( toBitvec "1010" ) ( toBitvec "1010" ) )
"0100"
> fromBitvec ( add ( toBitvec "1010" ) ( toBitvec "001010" ) )
"010100"
> fromBitvec ( add ( toBitvec "" ) ( toBitvec "" ) )
""
```

Poznámky a tipy

- Směle využívejte funkcí z `Prelude`.
- Importovat smíte i jiné moduly z balíku `base`, pohodlně se bez nich ale obejdete.
- Nepište podobné funkce pořád dokola, ale pokuste se problém co nejvíc abstrahovat.
- Přebíráte-li kód odjinud, nezapomeňte uvést zdroj. V opačném případě bude na vaši práci pohlíženo jako na plagiát.
- Nezapomeňte, že **opisování** je **zakázáno** a bude postihováno podle studijního řádu.

Odevzdání a bodování

Domácí úkol se odevzdává přes odpovědník v ISu. Tento odpovědník obsahuje jediné pole, do kterého vložíte svou implementaci požadovaných funkcí, včetně importů a definic z kostry řešení. Své řešení stručně komentujte v kódu, tato úloha totiž podléhá také ruční kontrole cvičícími, při níž se přihlíží i k jasnosti a pochopitelnosti řešení.

Máte **dvě možnosti odevzdání** (a samozřejmě kontrolu syntaxe před odevzdáním). Dokud nekliknete na „odevzdat“, můžete odpovědník zavřít a znovu otevřít bez penalizace. Nezapomínejte na průběžné ukládání. Po každém odevzdání uvidíte výsledky automatických testů, věnujte však dostatek času vlastnímu testování.

Pokud vaše řešení projde všemi automatickými testy, obdržíte jeden bod. V případě, že se vám podařilo implementovat jen část funkcí, sice žádný bod automaticky nedostanete, ale cvičící vám podle míry vyřešenosti při ruční kontrole úlohy udělí část bodu. Za kvalitu kódu a jeho stručný popis vám cvičící mohou udělit další jeden bod nebo jeho část.

V součtu tedy můžete za úlohu získat až 2 body.