IB015 – Domácí úkol 10: Aritmetické výrazy

Termín: 20. 11. 2022; 5 pokusů odevzdání; způsob odevzdání je podrobně popsán níže.

Úvod

V tomto úkolu budete pracovat s datovou strukturou reprezentující aritmetické výrazy. S podobnou strukturou jste se setkali už v příkladech 4.3.2 a 4.3.3 ve sbírce. Pro tuto úlohu ji ale rozšíříme o možnost reprezentovat i libovolné proměnné a zavádět lokální definice výrazů.

Představení reprezentace

Aritmetické výrazy budeme reprezentovat následující datovou strukturou, kterou si představíme řádek po řádku:

```
data OpType = Add | Mul deriving Show

data Expr a = Con a
            | Var String
            | Op OpType [Expr a]
            | Let String (Expr a) (Expr a)
            deriving Show
```

- Hodnotový konstruktor `Con :: a -> Expr a` slouží pro reprezentaci konstanty. Například číslo `42` lze reprezentovat hodnotou `Con 42`.
- Hodnotový konstruktor `Var :: String -> Expr a` slouží k reprezentaci proměnné pojmenované daným řetězcem. Například proměnnou `x` lze reprezentovat hodnotou `Var "x"`.
- Hodnotový konstruktor `Op :: OpType -> [Expr a] -> Expr a`, kde `OpType` je `Add` nebo `Mul`, reprezentuje *n*ární součet nebo součin jiných výrazů.

  Výraz `1 + 2 + 3` lze reprezentovat například jako `Expr Add [Con 1, Con 2, Con 3]` nebo `Expr Add [Con 1, Expr Add [Con 2, Con 3]]`, `Expr Add [Expr Add [Con 1], Con 2, Con 3]` a podobně. Každá taková reprezentace samozřejmě odpovídá jinému uzávorkování, ale jelikož jde o asociativní operace, tak nám to nevadí.

  Rozmyslete si, na jakou hodnotu dává smysl, aby se vyhodnotily výrazy `Expr Add []` a `Expr Mul []`, bude se vám to hodit v pozdějších příkladech.
- Hodnotový konstruktor `Let :: String -> Expr a -> Expr a -> Expr a` představuje lokální vazbu výrazu na jméno (podobně jako `let` označuje lokální definici v Haskellu). Předaný řetězec označuje název nové proměnné, první výraz pak její hodnotu a druhý výraz je ten, který tuto novou proměnnou může využívat a určuje hodnotu celého `let` výrazu. `let` výraz tedy lze přečíst stejně jako `let <řetězec> = <první výraz> in <druhý výraz>`.

  Výraz `x + 42` kde `x` má hodnotu `28` (tedy `let x = 28 in x + 42`) lze reprezentovat hodnotou `Let "x" (Con 28) (Op Add [Var "x", Con 42])`.

  Toto `let` není rekurzivní (na rozdíl od Haskellu) – pokud se v definici pro jméno `x` objeví opět `x`, neodkazuje se na právě definovanou hodnotu. Výraz `let x = 0 in let x = x + 1 in x + 2`, neboli

  ```
  Let "x" (Con 0) (Let "x" (Op Add [Var "x", Con 1]) (Op Add [Var "x", Con 2])
  ```

  obsahuje dvě vazby pro `x`: vnější, kde se `x` váže na `0`, a vnitřní, kde se `x` váže na `x + 1`. Toto `x` z podvýrazu `x + 1` se odkazuje na ono vnější `x`, ne na právě definované vnitřní `x`. Proměnná `x` v podvýrazu `x + 2` se pak odkazuje na ono vnitřní `x`. Tento výraz by se tedy u nás vyhodnotil na 3.

Žádný z datových typů neupravujte, kromě případného přidávání instancí typových tříd.

Funkce k implementaci

Řada funkcí, které máte implementovat, bere výraz typu `Expr a` a vrací ho upravený. V takových případech požadujeme, aby jediné provedené úpravy byly ty popsané zadáním. Neprovádějte žádné vlastní dodatečné

optimalizace. Speciálně tedy pokud je tedy na vstupu funkce výraz, který neobsahuje žádný problém, který má funkce odstranit, musí funkce vrátit výraz nezměněný.

- `freeVars :: Expr a -> [String]`

  Tato funkce vrací seznam všech proměnných, které jsou volné, t.j. které mají nějaký výskyt, který není vázaný. Daný výskyt proměnné `x` ve výrazu `expr` je vázaný, pokud `expr` obsahuje podvýraz `Let "x" subExprDef subExprIn` a uvažovaný výskyt `x` je v `subExprIn`. Seznam nesmí obsahovat duplicity, na pořadí prvků ovšem nezáleží.

  Tip: Modul `Data.List` obsahuje funkci pro deduplikaci seznamu.

```
freeVars (Var "x") ~>* ["x"]
freeVars (Op Add [Var "x", Var "x"]) ~>* ["x"]
freeVars (Let "x" (Con 28) (Op Add [Var "x", Con 42])) ~>* []
freeVars (Let "x" (Con 28) (Op Add [Con 42])) ~>* []
freeVars (Let "x" (Var "x") (Op Add [Var "x", Con 42])) ~>* ["x"]
freeVars (Let "x" (Var "x") (Op Add [Var "x", Var "y"])) ~>* ["x", "y"]
freeVars (Op Add [Var "x", Let "x" (Con 28) (Op Add [Var "x", Var "y"])])
      ~>* ["x", "y"]
```

- `simplifyArity01 :: Num a => Expr a -> Expr a`

  Tato funkce vrátí nový výraz vytvořený z původního odstraněním operací arity 0 a 1 (například `Op Add []`, `Op Mul [Con 42]`). Výrazy představující operátory s prázdným seznamem argumentů se vyhodnotí na neutrální prvek vzhledem k dané operaci. Výraz, který `simplifyArity01` vrátí, nesmí jít pomocí `simplifyArity01` dále zjednodušit.

```
simplifyArity01 (Op Mul [Con 4]) ~>* Con 4
simplifyArity01 (Op Mul []) ~>* Con 1
simplifyArity01 (Op Add []) ~>* Con 0
simplifyArity01 (Op Mul [Op Mul [], Op Mul []]) ~>* Op Mul [Con 1, Con 1]
simplifyArity01 (Op Mul [Op Add [Op Mul []]]) ~>* Con 1
```

- `eval :: Num a => Expr a -> [(String, a)] -> Maybe a`

  Tato funkce vrací vyhodnocený výraz, předaný v prvním parametru, přičemž hodnoty volných proměnných jsou dány druhým parametrem. Pokud hodnota kterékoli volné proměnné ve výrazu není poskytnutá, funkce vrací `Nothing`. Pokud má některá proměnná volný i vázaný výskyt nebo vícero vnořených vázaných výskytů, použije se vždy ten syntakticky nejbližší.

  Tip pro fajnšmekry: Může se vám hodit funkce `mapM` z `Control.Monad` a `fmap` nebo `>>=`. Pro naše účely si můžeme představit, že jejich typy jsou:

```
mapM :: (a -> Maybe b) -> [a] -> Maybe [b]
fmap :: (a -> b) -> Maybe a -> Maybe b
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
eval (Con 42) [] ~>* Just 42
eval (Con 42) [("x", 28)] ~>* Just 42
eval (Op Add [Con 28, Op Mul [Var "x", Var "y"]]) [("x", 7), ("y", 2)] ~>* Just 42
eval (Op Add [Con 28, Op Mul [Var "x", Var "y"]]) [("x", 7)] ~>* Nothing
eval (Let "x" (Con 42) (Var "x")) [("x", 0)] ~>* Just 42
eval (Op Add [Var "x", Let "x" (Con 42) (Var "x")]) [("x", 7)] ~>* Just 49
eval (Op Add [Let "x" (Con 7) (Var "x"), Let "x" (Con 42) (Var "x")]) [("x", 100)]
      ~>* Just 49
eval (Let "x" (Con 7) (Let "x" (Con 42) (Var "x"))) [("x", 100)] ~>* Just 42
eval (Let "x" (Op Add [Con 3, Var "y"]) (Op Mul [Con 6, Var "x"])) [("y", 4)]
      ~>* Just 42
```

- `flatten :: Expr a -> Expr a`

  Tato funkce ze vstupního výrazu odstraní všechny zbytečně vnořené `Op Add` a `Op Mul`. Funkce zachovává pořadí argumentů operací. Výraz, který `flatten` vrátí, nesmí jít pomocí `flatten` dále zjednodušit.

Tip: Zamyslete se nad pořadím, ve kterém zpracováváte daný výraz a jeho případné podvýrazy. Nemělo by se vám stát, že voláte funkci `flatten` na stejném (pod)výraze vícekrát.

```
flatten (Op Mul [Op Mul [Var "x", Con 4], Con 2, Op Add []])
    ~>* Op Mul [Var "x", Con 4, Con 2, Op Add []]
flatten (Op Mul [Op Mul [Var "x", Con 4], Con 2, Op Add [Op Add []]])
    ~>* Op Mul [Var "x", Con 4, Con 2, Op Add []]
flatten (Op Mul [Con 3, Op Add [Con 4]]) ~>* Op Mul [Con 3, Op Add [Con 4]]
flatten (Let "x" (Op Add [Op Add [Con 4]]) (Op Mul [Op Mul [Con 2]]))
    ~>* Let "x" (Op Add [Con 4]) (Op Mul [Con 2])
```

- `simplifyIdentity :: (Eq a, Num a) => Expr a -> Expr a`

  Tato funkce z `Op Add` a `Op Mul` odstraní operandy s neutrálními prvky vzhledem k dané operaci. Funkce **nezohledňuje** situace, kdy by se nějaký výraz vyhodnotil na neutrální prvek vzhledem k dané operaci. Funkce neodstraňuje ani proměnné, které představují neutrální prvek. Jde nám tedy pouze o případy, kdy je neutrální prvek reprezentován konstantou. Výraz, který `simplifyIdentity` vrátí, nesmí jít pomocí `simplifyIdentity` dále zjednodušit.

  ```
  simplifyIdentity (Op Add [Con 0, Con 42]) ~>* Op Add [Con 42]
  simplifyIdentity (Op Mul [Con 42, Con 1]) ~>* Op Mul [Con 42]
  simplifyIdentity (Op Add [Con 42, Con 1]) ~>* Op Add [Con 42, Con 1]
  simplifyIdentity (Op Mul [Con 1, Con 1]) ~>* Op Mul []
  simplifyIdentity (Op Mul [Con 1, Op Mul []]) ~>* Op Mul [Op Mul []]
  simplifyIdentity (Let "x" (Con 0) (Op Add [Var "x", Con 0, Con 42]))
      ~>* Let "x" (Con 0) (Op Add [Var "x", Con 42])
  ```

- `simplifyAnnihilating :: (Eq a, Num a) => Expr a -> Expr a`

  Tato funkce z výrazu odstraní všechny násobící operace, které obsahují absorpční prvek (0 vzhledem k násobení), a nahradí je konstantou odpovídající hodnotě `0` tedy `Con 0`. Sčítání žádný absorpční prvek nemá. Výraz, který `simplifyAnnihilating` vrátí, nesmí jít pomocí `simplifyAnnihilating` dále zjednodušit.

  ```
  simplifyAnnihilating (Op Mul [Con 0]) ~>* Con 0
  simplifyAnnihilating (Op Mul [Op Add []]) ~>* Op Mul [Op Add []]
  simplifyAnnihilating (Op Mul [Op Add [Con 1, Var "x"], Con 0, Op Mul [Con 42]])
      ~>* Con 0
  simplifyAnnihilating (Op Add [Op Mul [Con 0, Var "x"], Con 0, Op Mul [Con 0]])
      ~>* Op Add [Con 0, Con 0, Con 0]
  simplifyAnnihilating (Op Mul [Con 1, Op Mul [Con 2, Op Mul [Con 3, Con 0]]])
      ~>* Con 0
  ```

- `simplifyConstants :: Num a => Expr a -> Expr a`

  Tato funkce maximálně vyhodnotí všechny konstantní (pod)výrazy, a to včetně substituce za lokálně definované proměnné, které se vyhodnotí na konstantu.

  Konkrétně tedy funkce vyhodnotí všechny operace, kde jsou **všechny** operandy konstantní podvýrazy, a ve všech `let` výrazech, kde se jimi definovaná proměnná vyhodnotí na konstantu, se tato konstanta nahradí za výskyty dané proměnné. Výraz, který `simplifyConstants` vrátí, nesmí jít pomocí `simplifyConstants` dále zjednodušit.

  Funkce musí zachovávat relativní pořadí výrazů i v případě, že některé budou zjednodušeny. Například, pokud lze druhý výraz v seznamu zjednodušit na konstantu, tato konstanta bude v daném seznamu stále druhá v pořadí.

  Tip: V řešení můžete s výhodou použít funkce `sum` a `product` z `Prelude`. Prozkoumejte jejich chování na prázdném seznamu.

  ```
  simplifyConstants (Op Add [Con 28, Op Mul [Con 2, Con 7]]) ~>* Con 42
  simplifyConstants (Op Add [Con 28, Op Mul []]) ~>* Con 29
  simplifyConstants (Op Add [Var "x", Op Mul []]) ~>* Op Add [Var "x", Con 1]
  simplifyConstants (Op Add [Var "x", Op Mul [Con 2, Con 7]])
      ~>* Op Add [Var "x", Con 14]
  ```

```
simplifyConstants (Op Mul [Var "x", Con 0, Con 1])
    ~>* Op Mul [Var "x", Con 0, Con 1]
simplifyConstants (Let "x" (Con 28) (Op Add [Var "x", Op Mul [Con 2, Con 7]]))
    ~>* Con 42
simplifyConstants (Let "x" (Op Add [Con 20, Con 8])
                           (Op Add [Var "x", Op Mul [Con 2, Con 7]]))
    ~>* Con 42
simplifyConstants (Let "x" (Con 0) (Let "x" (Var "p") (Var "x")))
    ~>* Let "x" (Var "p") (Var "x")
```

Poznámky a tipy

- Importovat smíte moduly z balíku `base`.[1]
- Neduplikujte kód! Snažte se vždy využít funkce, které jste již naprogramovali. Pokud to nejde přímo, ale přesto vidíte v řešení podobu, vytkněte podobnou část do pomocné funkce.
- Nevynalézejte znovu kolo! Snažte se využívat knihovních funkcí, především nemá smysl je zbytečně reimplementovat. Důrazně doporučujeme použít vyhledávač Hoogle ve fakultní verzi, která automaticky vyhledává pouze v `base`.
- Pomocné funkce definujte lokálně, pokud jsou využívány jedinou funkcí.
- V řešení se vám může hodit pomocná funkce s akumulátorem (parametrem navíc, který ponese například aktuální hodnoty lokálních proměnných, nebo jinou pomocnou informaci).
- Funkce jsou v kostře zadefinovány jako `undefined`, takže projdou překladem, ale jejich zavolání způsobí chybu.
- Nejste-li si jisti nějakou částí zadání, zeptejte se v diskusním fóru.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.
- Přebíráte-li kód odjinud, uveďte zdroj, jinak bude na vaši práci pohlíženo jako na plagiát.
- Než řešení odevzdáte, **pečlivě si přečtěte následující sekci** a ujistěte se, že váš kód splňuje všechny náležitosti. Neztrácejte body jen kvůli nepozornému čtení pokynů.

Odevzdání a hodnocení

Na úlohu máte opět 5 pokusů. Úloha je plně automaticky testovaná.

Doporučujeme vám věnovat pozornost tipům k psaní hezkého kódu, které naleznete ve sbírce. Výhodou hezkého kódu je, že většinou obsahuje méně chyb a pokud už nějaké obsahuje, snáze se hledají.

Funkcionalita	Body
`freeVars`	0.1
`simplifyArity01`	0.2
`eval`	0.6
`flatten`	0.4
`simplifyIdentity`	0.2
`simplifyAnnihilating`	0.4
`simplifyConstants`	0.6

Odevzdání

Tento domácí úkol se neodevzdává přes odpovědník, nýbrž přes příslušnou odevzdávárnu v Informačním systému.

- Do odevzdávárny vkládejte **jediný** soubor s příponou `.hs` obsahující vaši implementaci **všech** požadovaných funkcí.
  - Pokud chcete odevzdat znovu, můžete soubor přepsat či přidat nový, nezáleží na tom – vyhodnocuje se vždy nejnovější odevzdaný soubor.
  - Žádné jiné soubory nevkládejte.
  - Již vložené soubory nepřejmenovávejte, mohou se pak vyhodnotit dvakrát.

[1] Výjimku tvoří moduly, které jsou „Unsafe“, ty však určitě nebudete potřebovat.

- Vaše řešení **musí zachovat všechny definice datových typů tak, jak jsou v zadání**, jediná povolená modifikace je přidání instance typové třídy.
- Řešení musí jít přeložit překladačem GHC 9.2.1. Je možné, že na svých počítačích máte starší verzi, což by nemělo vadit, jde-li o verzi 8.4 a vyšší. I přesto vám doporučujeme, abyste si před odevzdáním svůj kód zkusili zkompilovat a spustit na Aise (nezapomeňte přidat modul s novým GHC), kde máte k dispozici GHC ve verzi 9.2.1. V případě selhání testů už při kompilaci nepřijdete o žádný pokus.
- **Všechny globální funkce musí mít typovou signaturu**.
- V odevzdaném souboru neuvádějte hlavičku `module` (pokud nevíte, o co se jedná, vůbec to nevadí).

Vyhodnocování

- Vyhodnocení po nahrání souboru **není** okamžité.
  - Automatický testovací nástroj kontroluje soubory v odevzdávárně v pravidelných intervalech několika minut. Podle času odevzdání a vytížení vyhodnocovacího serveru může vyhodnocení trvat několik desítek minut.
  - Pokud se vám výsledky neobjevily cca do půl hodiny a neblíží se zatím čas deadline, dejte nám vědět ve fóru.
- Po vyhodnocení se získané body a případný výpis testů, které na vašem řešení selhaly, **objeví v poznámkovém bloku**.
  - U nesprávně implementovaných funkcí se dozvíte příklad vstupu, na němž se váš výsledek neshoduje s očekávaným.
- Máte **pět možností odevzdání**, započítává se nejlepší z nich.
  - Pokud při odevzdání neprojde kontrola syntaxe, tak se do tohoto limitu nepočítá. Pokud však kontrola projde, je automaticky započítáno a nelze to zvrátit.
- Další odevzdání provedete tak, že do odevzdávárny nahrajete novou verzi.
- Vzhledem k prodlevám při vyhodnocování neodkládejte práci na poslední chvíli, ať možnost vícenásobného odevzdání v případě potřeby vůbec stihnete využít.
  - S blížícím se termínem uzavření odevzdáváren očekávejte větší (i několikahodinové) prodlevy.
  - Není žádná garance rychlosti vyhodnocování (může se tedy stát, že výsledky řešení odevzdaného v neděli ve 21.00 neuvidíte do konce deadline).
  - Na jakákoli odevzdání po termínu nebude brán zřetel.

S odevzdávárnou zacházejte s rozvahou, abyste nepřišli o možnosti odevzdání. I když nahrajete nové řešení ještě před zveřejněním výsledku v poznámkovém bloku, vyhodnocovací nástroj už může mít (a pravděpodobně má) vaše dřívější odevzdání ve frontě. Z jeho pohledu tak došlo ke dvěma odevzdáním a vy si vyplýtváte jeden pokus. Podobně se vám mohou započítat odevzdání navíc, pokud do odevzdávárny omylem vložíte více než jeden soubor nebo pokud soubor přejmenujete (každý soubor se vezme jako samostatné odevzdání).