Kalkulačka

Vaším úkolem je napsat jednoduchou interaktivní symbolickou kalkulačku. Tato kalkulačka bude fungovat podobně jako interpret Haskellu `ghci` či běžný linuxový shell, tedy bude na příkazové řádce očekávat příkazy zadané uživatelem a bude na ně vhodně reagovat. Vaše kalkulačka *musí fungovat jako spustitelný soubor*, který s uživatelem komunikuje, ne jen jako modul s implementovanými funkcemi.

Symbolické kalkulačky obecně pracují s algebraickými výrazy, tedy matematickými výrazy s čísly i proměnnými. V našem případě se spokojíme s výrazy, které obsahují celá čísla, jedinou symbolickou proměnnou x a operace sčítání, odčítání, násobení a dělení. Váš program pak musí být schopen interaktivně takové výrazy načítat (v běžné infixové notaci), vypisovat, vyhodnocovat a symbolicky derivovat.

REPL

Požadujeme, aby bylo uživatelské rozhraní realizováno formou takzvaného REPLu (read-eval-print loop). Program tedy po spuštění čeká na zadání uživatelského příkazu na standardním vstupu, ten následně vyhodnotí, vypíše výsledek vyhodnocení na standardní výstup a opět čeká na následující příkaz.

Je třeba implementovat alespoň následující funkcionalitu:

1. Načtení nového výrazu a jeho uložení.
2. Vypsání současného (tedy uloženého) výrazu.
3. Vyhodnocení současného výrazu pro zadanou hodnotu proměnné x.
4. Vypsání symbolické derivace (vizte dále) současného výrazu.
5. Vyhodnocení symbolické derivace současného výrazu pro zadanou hodnotu x.

Příkladem konkrétní interakce tedy může být

```
./calc
> set (x * x + 1) / x
f(x) = (x * x + 1) / x
> get
f(x) = (x * x + 1) / x
> eval 5
f(5) = 26 / 5
> diff
f'(x) = ((1 * x + x * 1 + 0) * x - (x * x + 1) * 1) / x * x
> diff 5
f'(5) = 24 / 25
> set x / 2
f(x) = x / 2
> eval 10
f(10) = 5
```

kde text na řádku po znaku `>` je zadán uživatelem a řádky bez počátečního znaku `>` jsou výsledky vyhodnocení. [1]

Matematické výrazy

Vaše kalkulačka musí umět pracovat s infixovými matematickými výrazy. To znamená, že uživatel zadává výraz ve tvaru, který běžně znáte například z programovacích jazyků; tedy jako řetězec obsahující čísla (např. `42`), výskyty proměnné `x`, operátory `+`, `-`, `*` a `/`, závorky `(` a `)` a bílá místa (mezery) libovolně.

Operátory ve výrazu respektují běžné matematické konvence priority: násobení a dělení se aplikuje před sčítáním a odčítáním. Tedy vstup `1 + 2 * x` odpovídá výrazu $1+2x$, nikoliv $(1+2)x$. Setká-li se za sebou několik operátorů shodné priority (tedy skupina několika `+` a `-` nebo několika `*` a `/`), asociují operátory doleva. Vstup `1 - 2 * x + 3` tak značí výraz $(1-2x)+3$, nikoliv $1-(2x+3)$.[2]

Pro načtení vstupu v tomto formátu využijte knihovnu `Parsec`. Máte přitom vesměs dvě možnosti. Buď napíšete pomocí poskytovaných kombinátorů (v modulech `Text.Parsec`, `Text.Parsec.Char`,

[1] Toto je pouze ilustrační příklad. Nevynucujeme konkrétní podobu vašich příkazů, výstupu, či dalších detailů. Požadavkem je pouze implementovat zmíněnou funkcionalitu.

[2] Povšimněte si, že se tyto výrazy liší, neboť odečítání (a podobně dělení) není asociativní operace.

`Text.Parsec.Combinator` a `Text.Parsec.String`) takzvaný rekurzivně sestupný parser, nebo využijete toho, že `Parsec` přímo poskytuje funkcionalitu načítání vstupu s operátory se zadanou prioritou a asociativitou v modulu `Text.Parsec.Expr`.

První možnost je programátorsky obtížnější, ale nemusíte se při ní seznamovat s žádnými součástmi `Parsec`u nad rámec přednášky. Rozhodnete-li se pro ni, může se vám hodit následující bezkontextová gramatika pro matematické výrazy (až na možné mezery)[3], kde `number` odpovídá posloupnosti číslic:

```
expr   = term     | expr + term   | expr - term
term   = factor   | term * factor | term / factor
factor = atom     | - atom
atom   = ( expr ) | number        | x
```

Druhou možností (kterou doporučujeme), je nastudování si (poměrně malého) rozhraní modulu `Text.Parsec.Expr`, zejména funkce `buildExpressionParser`. Tento přístup spočívá v deklarativní specifikaci možných operátorů, jejich priorit a asociativit, parserů zpracovávajících jejich znaky a nakonec parseru pro atomické výrazy (které jsou v dokumentaci na Hackage poněkud matoucím způsobem označeny jako termy).

Potenciálním zádrhelem je zde typ `Operator s u m a`. Neděste, se, pro naše účely je možné typové parametry `s`, `u` a `m` ignorovat, přičemž `a` reprezentuje výsledný typ parseru (stejně jako místo `ParsecT s u m a` používáme pouze `Parser a`).

Při výpisu výrazů požadujeme jen to, aby výpis obsahoval nějaký infixový matematický výraz reprezentující "skutečný" algebraický výraz. Není potřeba provádět žádné zjednodušovací operace, jako je vyhodnocování konstantních podvýrazů či vynechávání zbytečných závorek.

Symbolické derivace

Někteří z vás se již mohli setkat s pojmem derivace funkce na střední škole. Pro účely tohoto příkladu stačí chápat derivaci jako čistě formální operaci manipulující s algebraickým výrazem, není potřeba nijak zacházet do definic a matematických podrobností.

Symbolickou derivací rozumíme operaci, která jako parametr dostane číselnou funkci jedné proměnné $f(x)$ a vyprodukuje funkci $f'(x)$ podle následující induktivní definice:

1. Pro $f(x) = n$, kde $n \in \mathbb{Z}$ je konstanta, máme $f'(x) = 0$.
2. Pro $f(x) = x$ máme $f'(x) = 1$.
3. Pro $f(x) = g(x) + h(x)$ máme $f'(x) = g'(x) + h'(x)$.
4. Pro $f(x) = g(x) - h(x)$ máme $f'(x) = g'(x) - h'(x)$.
5. Pro $f(x) = g(x) \cdot h(x)$ máme $f'(x) = g'(x) \cdot h(x) + g(x) \cdot h'(x)$.
6. Pro $f(x) = \frac{g(x)}{h(x)}$ máme $f'(x) = \frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{[h(x)]^2}$.

Tato pravidla zachycují všechny druhy výrazů, se kterými v tomto úkolu pracujeme, nemusíte se tedy zabývat ničím pokročilejším. Pokud i přesto chcete vědět více, vhodným zdrojem je Wikipedie.

Rady na závěr

Všechna čísla ve výrazech, se kterými pracujete, jsou celá. Přesto však může být výsledkem vyhodnocení číslo racionální, neboť ve výrazech umožňujeme dělení. Nepoužívejte nepřesné číselné typy s plovoucí desetinnou čárkou, jako je `Float` nebo `Double`. Místo toho se podívejte na modul `Data.Ratio` a jeho číselný typ `Rational`.

V implementaci zpracování uživatelského vstupu budete muset vyřešit, jak ignorovat mezery. Vzpomeňte si na funkci `spaces`. Ulehčíte si život, pokud budete mezery číst až po zpracování "užitečného vstupu", tedy jako `<váš parser> <* spaces`, nikoliv `spaces *> <váš parser>` (pro vysvětlení vizte například tuto odpověď na StackOverflow).

Budete si muset poradit s dělením nulou a neplatnými vstupy. Pouvažujte, jaké možnosti máte, abyste se vyhnuli repetitivnímu a duplikovanému kódu. Obecně se snažte zbytečně neduplikovat kód a používat rozumnou dekompozici s pomocnými funkcemi. S výhodou můžete využít funkcí z modulů `Data.Functor`,

[3] Dejte si však pozor, abyste skutečně zpracovávali operátory s asociací doleva!

`Control.Applicative` a `Control.Monad`. Celkově ošetření chyb necháváme na vás, ale váš program by rozhodně neměl při zadání neplatného vstupu spadnout.

Pro implementaci vlastního REPLu musíte pracovat s monádou `IO`. Mohla by se vám však hodit i monáda `State` (byť to není nijak nezbytné). Rozhodnete-li se ji však využít, budete muset tyto monády vhodně zkombinovat pomocí transformátoru monád `StateT` z balíku `mtl` (který jistě budete mít stažený, neboť na něm závisí balík `parsec`).

Nebojte se často kontrolovat dokumentaci používaných modulů na Hackage či využívat vyhledávač Hoogle. V případě problémů a dotazů se nebojte obrátit se na nás například na diskuzním fóru.