

## 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 a zejména vyhodnocovat.

## 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  $x$ .

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
> set x / 2
f(x) = x / 2
> eval 10
f(10) = 5
```

kde první řádek se odehrává na vašem systémovém shellu, text na řádce po znaku `>` je zadán uživatelem a řádky bez počátečního znaku `>` jsou výsledky vyhodnocení.<sup>1</sup>

## 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  $1 - (2x + 3)$ .<sup>2</sup>

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`, `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 (a zajímavější!), ale nemusíte se při ní seznamovat s žádnými součástmi `Parsec` nad rámec přednášky. Rozhodnete-li se pro ni, může se vám hodit následující bez-

<sup>1</sup>Dejte si pozor, aby vaše kalkulačka uměla správně pracovat též se zápornými čísly!

<sup>2</sup>Povšimněte si, že se tyto výrazy liší, neboť odečítání (a podobně dělení) není asociativní operace.

kontextová gramatika (bez levé rekurze) pro matematické výrazy (až na možné mezery)<sup>3</sup>, kde `number` odpovídá posloupnosti číslic:

```
expr  = term      | term + expr  | term - expr
term  = factor    | factor * term | factor / term
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.<sup>4</sup>

## Rady na závěr

Všechna čísla ve výrazech, se kterými pracujete, jsou celá.<sup>5</sup> 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 čist 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`, `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`. Doporučujeme vám zamyslet se, jak rozdělit kód na čisté funkce, které s okolním světem skrze `IO` interagovat nemusí, a nečistou slupku, která `IO` používá. Dejte si zde pozor na správnou dekompozici a separaci; v Haskellu (ale čím dál více i jinde) je dobrým zvykem psát co nejvíce logiky v čistých funkcích, které o `IO` nic neví.

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.

---

<sup>3</sup>Dejte si však pozor, abyste skutečně zpracovávali operátory s asociací doleva!

<sup>4</sup>Napsat takové zjednodušování může být přesto zajímavý úkol. Pokud si to však chcete vyzkoušet, dejte si pozor, abyste nezabředli do komplikovaných a nehezských funkcí, což zde lehce jde.

<sup>5</sup>Dejte si pozor, aby vaše kalkulačka uměla správně pracovat též se zápornými čísly!