IB016 – 1. velký úkol – Nanohaskell

Termíny

- Neděle **4. dubna** 2021 23.59 – implementační část
- Neděle **11. dubna** 2021 23.59 – peer review

Tento úkol bude celý o Parsecu. Napíšeme si parser malinké podmnožiny Haskellu. Budeme bezostyšně ignorovat typový systém a odsazování; nebudeme podporovat pattern-matching či lokální definice a nebudeme se trápit prioritami a asociativitou operátorů.

Kostru řešení s testy naleznete ve studijních materiálech.

Vstup

Vstupem je malá podmnožina jazyka Haskell. Jeho syntaxi již důvěrně znáte; místo formálního popisu si zde proto jen vyjmenujeme, co má parser umět rozpoznat.

Můžete toho podporovat více, ale mělo by platit, že cokoli váš parser přijme, je, až na odsazení, syntakticky (ne nutně typově) platný Haskell. Ono to nepůjde vždycky 100%, ale snažte se nevybočovat z haskellových mantinelů.

Na úvod zapomeneme na jakékoli odsazování. Všechny prázdné znaky (mezery, tabulátory, zalomení řádků) budeme považovat za rovnocenné. Mimochodem, věděli jste, že v Haskellu jsou odsazené bloky jen syntaktický cukr, který se dá nahradit složenými závorkami a středníky? Takto:

```
do x <- foo     =>   do { x <- foo; let { y = x; z = y }; return z }
   let y = x
       z = y
   return z
```

Tím se ale trápit nebudeme; předpokládejme, že na vstupu bude program po této úpravě.

Deklarace

Parser bude na nejvyšší úrovni čekat posloupnost deklarací oddělených (a volitelně ukončených) středníky (na nejvyšší úrovni se složené závorky nepoužívají). Každá deklarace má potom tvar `jméno argument1 argument2 = výraz`, kde argumentů může být libovolně mnoho (vč. žádného).

Výraz

Parser má podporovat přinejmenším následující druhy výrazů:

- **Literál**
  - nezáporný celočíselný desítkový;
  - logický (`True` nebo `False`).
- **Identifikátor** – jméno proměnné nebo funkce (vizte sekci o jménech níže)
  - slovní identifikátor;
  - operátor uzavřený v kulatých závorkách.
- **Lambda-abstrakce** – s jedním nebo více argumenty.
- **Podmíněný výraz** – `if ... then ... else ...`.
- **Aplikace** výrazu na jeden nebo více argumentů:
  - **prefixová** – stejně jako v Haskellu:
    - `foo 1 2`, `(+) x y` i `(foo bar) (odd 1)`;
  - **infixová** – značně zjednodušená, ale stále validní Haskell:
    - operátory i slovní identifikátory: `1 + a` i `x `mod` 2`
    - bez priorit a asociativity – stačí umět přečíst `(x + y)`; výrazy jako `(x + y + z)` smíte zamítnout;
  - prefix má přednost před infixem jako v Haskellu, tudíž `f 1 + f 5` ≡ `((+) (f 1)) (f 5)`.
- **Podvýraz** – libovolný z výrazů výše uzavřený v kulatých závorkách.

Jména

Platná jména deklarovaných i používaných funkcí, konstant i argumentů musí být (s drobnými výjimkami) podmnožinou jmen haskellových:

- **Slovní identifikátor** začíná malým písmenem a smí obsahovat alfanumerické znaky, podtržítka a apostrofy.
  - Stačí rozsah ASCII, ale smíte přijímat i jiné znaky, které jsou v Haskellu přípustné (řecká písmena atd.).
  - Identifikátor nesmí být klíčové slovo. Nemusíte zjišťovat všechna klíčová slova Haskellu; stačí, když zamítnete ta, která mají zvláštní význam pro váš Nanohaskell. Takže typicky `if`, `then` a `else` (ale pokud byste si chtěli přidat třeba lokální definice, pak je nutné zamítanout i `let` a `in`).
- **Operátor** smí obsahovat znaky: `+-*/%<>=$^.|&:\`
  - Můžete přijímat i další, které Haskell dovoluje použít (dokonce i v ASCII ji ještě takových spousta zbývá).
  - Jméno operátoru nesmí začínat dvojtečkou (to jsou v Haskellu infixové konstruktory)
  - Jméno operátoru nesmí být žádná z následujících syntakticky významných sekvencí: `=`, `\`, `->`, `<-`, `|`, `--`, přetože poslední tři nemusí být významné pro váš Nanohaskell.
  - Kromě infixové aplikace je operátor vždy obalen kulatými závorkami.

Formální argumenty deklarovaných funkcí i lambd mohou být pouze identifikátory (nikoli literály). Překvapivě ale mohou být formálním argumentem i operátory: `\($$) -> ($$)` je platný haskellový (i nanohaskellový) výraz.

Další syntaktické kontroly

- **Kolize argumentů** – v rámci jedné deklarace a jedné lambdy musí být jména argumentů unikátní.
  - zamítnout: `(\x x -> x)`, `f x y x = y`
  - přijmout: `(\x -> \x -> x)`, `f x = \x -> x`
- **Kolize deklarací** – každá deklarace musí zavádět novou funkci.
  - zamítnout: `foo x = 4 ; foo = \x -> 5`

Jiné kontroly **neprovádějte** – zejména žádnou typovou kontrolu. Jakkoli je ten pohled bolestivý, `if 4 then True else 0` je platný nanohaskellový výraz.[1]

Nekontrolujte ani, zda jsou používané identifikátory deklarované. Tudíž i `foo x = y + 2` je syntakticky korektní, ačkoli `y` ani `(+)` v místě použití neexistuje.

Výstup

Výsledkem parsování je seznam deklarací výrazů ještě mnohem jednoduššího jazyka – mírně rozšířeného *netypovaného lambda-kalkulu*. Představte si ho jako Haskell, který má jen čísla, identifikátory, unární lambdy, unární aplikace a `if`.

Cílové datové typy jsou zadefinovány v kostře a vypadají následovně:

```
type Identifier = String

data Term = C Integer           -- 1, True
          | I Identifier        -- x
          | L Identifier Term   -- \x -> M
          | Term :$ Term        -- M N
          | Cond Term Term Term -- if M then N1 else N2
          deriving (Eq, Show)

data Decl = Identifier := Term  -- x = M
          deriving (Eq, Show)
```

[1] On ve skutečnosti může být v Haskellu i typově korektní.

Oproti tomuto má i náš skromný Nanohaskell hromadu syntaktického cukru, který musíme do prostší representace převést.

Nápověda: `f x y = y `x` 2` → `f = \x y -> x y 2` → `f = \x -> (\y -> (x y) 2)` ⟹ `"f" := L "x" (L "y" ((I "x" :$ I "y") :$ C 2))`

Další poznámky k převodu do `Decl` a `Term`:

- Konstanty `True` a `False` representujte pomocí `C 1` a `C 0`.
- Jména operátorů representujte bez obalovacích závorek, tedy třeba `I ">>="`.
- Neprovádějte žádné „chytré“ úpravy a optimalisace; například nevyhodnocujte aplikace funkcí, nepřejmenovávejte argumenty apod. Stejně tak zachovejte pořadí deklarací.
- Povšimněte si, že pro aplikaci a deklaraci používáme infixové hodnotové konstruktory `(:$)` a `(:=)` – můžete je psát jako běžné operátory.

K implementaci

```
nanoHsParser :: Parser [Decl]
```

Top-level parser nanohaskellových deklarací.

Příklad

```
fact n = if      n < 0  then False
         else if n == 0 then 1
         else n * fact ( n - 1 ) ;
negate = \n -> if n < 0 then (-) 0 n else n
```

Výstup:

```
["fact" := L "n" (Cond ((I "<" :$ I "n") :$ C 0) (C 0)
                 (Cond ((I "==" :$ I "n") :$ C 0) (C 1)
                 ((I "*" :$ I "n") :$ (I "fact" :$ ((I "-" :$ I "n") :$ C 1)))))
,"negate" := L "n" (Cond ((I "<" :$ I "n") :$ C 0) ((I "-" :$ C 0) :$ I "n") (I "n"))]
```

Odevzdávání

- Svou implementaci odevzdávejte do příslušné odevzdávárny v ISu.
- Na odevzdávání máte neomezeně mnoho pokusů, výsledky testů naleznete v poznámkových blocích, jak jste zvyklí z IB015.
- Po ukončení odevzdávání budou zveřejněna řešení všech studentů a zároveň vám budou přiděleni dva spolužáci, na jejichž implementaci následně napíšete peer review, tedy recenzi jejich kódu.
- Peer review se píší ve formě komentářů do recenzovaného zdrojového kódu a odevzdávají se na Aise. K tomuto ještě obdržíte e-mail s pokyny.

Technické požadavky

- Odevzdávejte jediný soubor s příponou `.hs`.
- Řešení musí být přeložitelné GHC 8.10. Můžete si to vyzkoušet na Aise s modulem `ghc`.
- Všechny globálně definované funkce musí mít typovou signaturu.
- Řešení nesmí obsahovat hlavičku modulu.

Moduly a rozšíření

- Můžete využít libovolné moduly z balíku `base`.
- Z balíku `parsec` si vystačíte s `Text.Parsec` a `Text.Parsec.String`.
- Řešení můžete okrášlit pomocí rozšíření `UnicodeSyntax` a balíku `base-unicode-symbols`.
- Povolenými rozšířeními jsou také `PatternGuards`, `LambdaCase` a `TupleSections`.

Hodnocení

Za úlohu můžete získat až čtyři body. Pro úspěšné absolvování předmětu potřebujete alespoň jeden bod (vizte IS pro podrobné informace). Doporučujeme tedy řešení úlohy zbytečně neodkládat.

V tomto semináři nerozlišujeme body za krásu a body za funkčnost, proto berte projití testy pouze jako potvrzení toho, že něco neděláte úplně špatně, ne jako signál, že máte 4 body v kapse. Řešení, která budou funkční, ale ošklivá, si plný počet nezaslouží. V žádném případě byste se neměli dopouštět prohřešků, před nimiž varujeme už v IB015.

Body vám udělí cvičící i se zpětnou vazbou až poté, co budou odevzdána i peer review.

Řešení tentokrát budou veřejná, takže se na ně po termínu odevzdání bude moci kdokoliv podívat. To proto, aby bylo jednodušší psát si vzájemně peer review, a částečně vás tím chceme motivovat k psaní hezčího kódu. Nebojte se pomoci si nástrojem HLint (vaši recenzenti to určitě udělají).

Poznámky a tipy

Konkrétně k tomuto zadání:

- Ne všechny výrazy jsou si rovny. Některé nemohou stát na místě funkce nebo argumentu bez obalení do závorek.
- Opatrně s `try`! Ujistěte se, že ho používáte pouze u jednoduchých, nerekursivních parserů.
  - Autorské řešení používá `try` pouze při parsování identifikátorů. Souvisí to s klíčovými slovy a s otvírací závorkou u operátorů.
  - Chcete-li použít `try` pro čtení infixové aplikace, pravděpodobně to je špatná cesta: načtený term je stejný, ať už je samostatný, nebo je operandem infixové aplikace. Nemá proto smysl ho zahazovat.
- Infixovou aplikaci poznáte až po načtení celého levého operandu.
- Při zpracování násobných argumentů si vzpomeňte na foldy, mohly by vám pomoci.
- Porozhlédněte se, jaké kombinátory vám Parsec dává k disposici. Ušetříte si tak spoustu práce. Na druhou stranu se nebojte psát si i vlastní, například na vypořádání se se skutečností, že mezery mohou být takřka kdekoli.
- Při parsování můžete dát pomocí `fail` najevo, že tudy cesta nevede. Hodí se to u kontroly vícenásobné deklarace či při použití klíčového slova jako identifikátoru.

A klasické obecné poznámky:

- Neduplikujte kód! Využívejte vlastních i knihovních funkcí. Pomoci vám v tom může Hoogle.
- Pokud vám není něco jasné, zeptejte se v diskuzním fóru nebo na ircovém/discordovém kanále.
- V případě, že přebíráte kód odjinud, uveďte zdroj.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.