Typová třída Applicative, monadické parsery

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Funktory a monády – připomenutí

Functor:

- `fmap :: (a -> b) -> f a -> f b`:
- aplikace funkce na výsledek výpočtu

Monad:

- `return :: Monad f => a -> f a`
- triviální výpočet
- `(>>=) :: m a -> (a -> m b) -> m b`
- zřetězení dvou výpočtů; druhý výpočet může záviset na výsledku předchozího výpočtu
- `join :: m (m a) -> m a`
- lze spustit výpočet, který je výsledkem výpočtu

Typová třída Applicative

Motivace pro Applicative

Aplikativní funktory jsou na půli cesty mezi **`Functor`** a **`Monad`**.

- Umožňují z hodnoty vyrobit triviální výpočet:
  - `pure :: Applicative f => a -> f a`
  - doslova totéž co `return`, ale historické důvody

Motivace pro Applicative

Aplikativní funktory jsou na půli cesty mezi **`Functor`** a **`Monad`**.

- Umožňují z hodnoty vyrobit triviální výpočet:
  - `pure :: Applicative f => a -> f a`
  - doslova totéž co `return`, ale historické důvody
- Umožňují kombinovat výsledky nezávislých výpočtů:
  - `liftA2 ::...=> (a -> b -> c) -> f a -> f b -> f c`
  - `liftA3` obdobně pro ternární
  - `<*> ::...=> f (a -> b) -> f a -> f b`
  - Nelze měnit *efekt* (`f`) na základě *výsledků* (`a`, `b`). Např. `liftA2 op (Just 1) (Just 2)` neumí vrátit **`Nothing`**.

Existují aplikativní funktory, které nejsou monádami (**`ZipList`**).

Typová třída Applicative

```
class Functor f => Applicative f where
    pure   :: a -> f a
    (<*>)  :: f (a -> b) -> f a -> f b
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

- samotná třída v `Prelude`, více v `Control.Applicative`
- předepsaných funkcí je více, ale dají se odvodit
- stačí jedno z `(<*>)` a `liftA2`
- lze napsat instanci **Monad** a použít `liftA2 = liftM2`
- naopak `return` se automaticky zadefinuje jako `pure`
- libovolně-ární liftování: `f <$> mx <*> my <*> mz <*> ...`

Applicative vs. Monad

Z historických důvodů je spousta „aplikativních“ věcí v **`Monad`**:

- `pure` ≡ `return`
- `(<*>)` ≡ `ap`
- `(*>)` ≡ `(>>)`
- `liftA` ≡ `liftM` (≡ `fmap`)
- `liftA2` ≡ `liftM2`
- `liftA3` ≡ `liftM3`

`Applicative` má navíc `(<*)`, **`Monad`** zase až `liftM5`.

Pravidla pro třídu Applicative

Korektní instance **`Applicative`** musí splňovat:

- *identita*
  `(pure id) <*> x` ≡ `x`
- *kompozice*
  `(pure (.)) <*> f <*> g <*> x` ≡ `f <*> (g <*> x)`
- *homomorfismus*
  `(pure f) <*> (pure x)` ≡ `pure (f x)`
- *výměna*
  `u <*> (pure y)` ≡ `(pure ($ y)) <*> u`

Vsuvka: MonadFail

```
class Monad m => MonadFail m where
    fail :: String -> m a
```

- akce `fail` má přerušit probíhající výpočet
- historicky (ještě v GHC 8.6) součástí třídy **Monad**
- automaticky se používá při pattern-matchingu v **do**-bloku
  - `do (x:xs) <- lookup key tableOfLists`
- instance pro **[]**, **Maybe** a **IO**

Funktory a monády – shrnutí

Functor:

- `fmap :: (a -> b) -> f a -> f b`:
- aplikace funkce na výsledek výpočtu

Applicative:

- `pure :: a -> f a`
- triviální výpočet se zadaným výsledkem
- `liftA2 :: (a -> b -> c) -> f a -> f b -> f c`
- kombinace výsledků nezávislých výpočtů

Monad:

- `return :: Monad f => a -> f a`
- triviální výpočet
- `(>>=) :: m a -> (a -> m b) -> m b`
- zřetězení dvou výpočtů; druhý výpočet může záviset na výsledku předchozího výpočtu
- `join :: m (m a) -> m a`
- lze spustit výpočet, který je výsledkem výpočtu

Funktory a monády – „krabičkový pohled“

Functor:

- **někdy** odpovídá kontejnerům a krabičkám
- hodnoty uvnitř lze měnit bez změny struktury kontejneru

Applicative:

- umíme vyrobit „singleton“ s „neutrální strukturou“
- umíme zkombinovat dvě nezávislé krabičky:
  - jak kombinovat hodnoty uvnitř, říká dodaná čistá funkce
  - jak sloučit obaly (strukturu), říká instance

Monad:

- ??? . . . ne!
- `join` → umíme sloučit vnořené krabičky

Monadické parsování – Parsec

Syntaktické analyzátory (parsery)

Regulární výrazy často nestačí $\rightarrow$ využijeme syntaktické analyzátory (parsery)

- lexikální analyzátor **Alex** + syntaktický analyzátor **Happy** (podobné kombinaci *lex*/*flex* + *bison*/*yacc*)
- **Parsec** – knihovna založena na parserových kombinátorech, zvládá i tvorbu lexikálních analyzátorů, přívětivější chybové hlášky
- **Attoparsec** – další kombinátorová knihovna, rychlejší než Parsec – hlavně pro síťové protokoly a komplikované textové/binární formáty, chybí *transformátor*
- A mnoho dalších – Polyparse, Megaparsec, GLL...

Parsec

Myšlenka:

- definujme parser pomocí většího množství menších/jednodušších parserů

Parsec

Myšlenka:

- definujme parser pomocí většího množství menších/jednodušších parserů

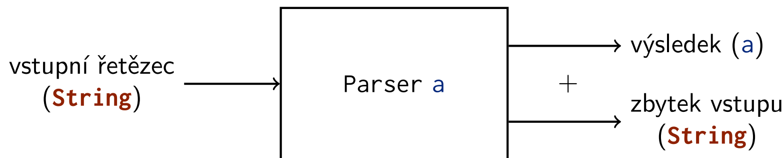
Technicky:

- balík parsec (není součástí standardní distribuce)
- nejpoužívanější funkce přímo v modulu `Text.Parsec`
- pro naše účely hlavně funkce z modulů `Text.Parsec.Char` a `Text.Parsec.Combinators`
- pro zjednodušení typování importujte i modul `Text.Parsec.String`

Datový typ Parser

```
myParser :: Parser a
```

- parser, který zpracuje vstup na hodnotu typu `a`
- instance pro třídy **`Functor`**, **`Applicative`** i **`Monad`**
- ve skutečnosti je to pouze typové synonymum pro obecnější parser, viz později



Aplikace parserů

```
parse :: Parser a -> SourceName -> String -> Either ParseError a
parse p name input:
```

- spustí parser `p` na vstupu `input`
- `name` se bude používat pouze na identifikaci v chybových hláškách
- výsledkem je buď zparsovaná hodnota, nebo chyba typu `ParseError`

Aplikace parserů

```
parse :: Parser a -> SourceName -> String -> Either ParseError a
parse p name input:
```

- spustí parser `p` na vstupu `input`
- `name` se bude používat pouze na identifikaci v chybových hláškách
- výsledkem je buď zparsovaná hodnota, nebo chyba typu `ParseError`

```
parseFromFile :: Parser a -> String -> IO (Either ParseError a)
parseFromFile p f
```

- spustí parser `p` na obsahu souboru `f`

Pro testování:

- `parseTest :: Parser a -> String -> IO ()`
- `parserTrace :: String -> Parser ()`

Základní znakový parser

```
satisfy :: (Char -> Bool) -> Parser Char
```

- parser `satisfy p` uspěje, jestliže načtený znak splňuje zadaný predikát (pak vrací tento znak)

Základní znakový parser

```
satisfy :: (Char -> Bool) -> Parser Char
```

- parser `satisfy p` uspěje, jestliže načtený znak splňuje zadaný predikát (pak vrací tento znak)

Existující užitečné parsery:

```
char :: Char -> Parser Char
digit, hexDigit, octDigit :: Parser Char
anyChar, upper, lower, alphaNum, letter :: Parser Char
newline, crlf, endOfLine, space, spaces, tab :: Parser Char
oneOf, noneOf :: [Char] -> Parser Char
```

Základní znakový parser

```
satisfy :: (Char -> Bool) -> Parser Char
```

- parser `satisfy p` uspěje, jestliže načtený znak splňuje zadaný predikát (pak vrací tento znak)

Existující užitečné parsery:

```
char :: Char -> Parser Char
digit, hexDigit, octDigit :: Parser Char
anyChar, upper, lower, alphaNum, letter :: Parser Char
newline, crlf, endOfLine, space, spaces, tab :: Parser Char
oneOf, noneOf :: [Char] -> Parser Char
```

Příklad: *02 Mar 2022*

- Které znakové parsery použít?
- Jak napsat znakové parsery pomocí `satisfy`?

Funkce a operátory z instancí

Zamyšlení: Jak budou fungovat instance **Functor**, **Monad**?

- ```
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap (: "!") (char 'A')
  ```
  ```

Funkce a operátory z instancí

Zamyšlení: Jak budou fungovat instance **Functor**, **Monad**?

- `fmap :: (a -> b) -> Parser a -> Parser b`
  `fmap (: "!") (char 'A')`

  `fmap` aplikuje funkci na vrácenou hodnotu
- `pure :: a -> Parser a`
  `pure 42`

Funkce a operátory z instancí

Zamyšlení: Jak budou fungovat instance **Functor**, **Monad**?

- ```
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap (: "!") (char 'A')
  ```
  fmap aplikuje funkci na vrácenou hodnotu
- ```
  pure :: a -> Parser a
  pure 42
  ```
  pure x nic nečte a vrátí hodnotu x
- ```
  twoChars = do
      char1 <- anyChar
      char2 <- anyChar
      pure [char1, char2]
  ```
  ```

Funkce a operátory z instancí

Zamyšlení: Jak budou fungovat instance **Functor**, **Monad**?

- `fmap :: (a -> b) -> Parser a -> Parser b`
  `fmap (: "!") (char 'A')`
  `fmap` aplikuje funkci na vrácenou hodnotu
- `pure :: a -> Parser a`
  `pure 42`
  `pure x` nic nečte a vrátí hodnotu `x`
- ```
  twoChars = do
      char1 <- anyChar
      char2 <- anyChar
      pure [char1, char2]
  ```
  operátor `>>=` předá zparsovanou hodnotu zleva doprava
  ```

Další užitečné operátory

- >>

```
spaces >> identifier
```

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`
- `<$>` (to samé jako `fmap`) — `(++ "!") <$> greeting`

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`
- `<$>` (to samé jako `fmap`) — `(++ "!") <$> greeting`
- `<$` — `"greeting here" <$ greeting`

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`
- `<$>` (to samé jako `fmap`) — `(++ "!") <$> greeting`
- `<$` — `"greeting here" <$ greeting`
- `<*>` — `(,) <$> key <*> value`

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`
- `<$>` (to samé jako `fmap`) — `(++ "!") <$> greeting`
- `<$` — `"greeting here" <$ greeting`
- `<*>` — `(,) <$> key <*> value`
- `<*` — `string "Hello" <* spaces <* name`

Další užitečné operátory

- `>>` — `spaces >> identifier`
- `liftAX` — `liftA2 (+) number number`
- `<$>` (to samé jako `fmap`) — `(++ "!") <$> greeting`
- `<$` — `"greeting here" <$ greeting`
- `<*>` — `(,) <$> key <*> value`
- `<*` — `string "Hello" <* spaces <* name`
- `*>` — `string "Hello" *> spaces *> name`

Mnemotechnická pomůcka: používají se ty hodnoty, na které ukazuje zobáček.

Sekvence parserů

Existující užitečné parsery:

```
string :: String -> Parser String
between :: Parser open -> Parser close -> Parser a -> Parser a
count :: Int -> Parser a -> Parser [a]
many, many1 :: Parser a -> Parser [a]
sepBy, sepBy1 :: Parser a -> Parser sep -> Parser [a]
choice :: [Parsec a] -> Parsec a
```

Sekvence parserů

Existující užitečné parsery:

```
string :: String -> Parser String
between :: Parser open -> Parser close -> Parser a -> Parser a
count :: Int -> Parser a -> Parser [a]
many, many1 :: Parser a -> Parser [a]
sepBy, sepBy1 :: Parser a -> Parser sep -> Parser [a]
choice :: [Parsec a] -> Parsec a
```

- Kolik vstupu spracuje `many`?
  `parse (many digit) "input name" "hello"`
- Jak zparsujeme datum a které parsery použijeme?
  Například: *02 Mar 2022*

Příklad kombinace parserů v do-bloku

Příklad: *02 Mar 2022*

```
data Date = Date { dDay :: Int, dMonth :: Month, dYear :: Int }
```

Příklad kombinace parserů v do-bloku

Příklad: *02 Mar 2022*

```
data Date = Date { dDay :: Int, dMonth :: Month, dYear :: Int }

dateParser :: Parser Date
dateParser = do
  day <- read <$> count 2 digit
  space
  month <- monthParser
  space
  year <- read <$> count 4 digit
  pure $ Date day month year
```

Příklad kombinace parserů v do-bloku

Příklad: *02 Mar 2022*

```
data Date = Date { dDay :: Int, dMonth :: Month, dYear :: Int }

dateParser :: Parser Date
dateParser = do
  day <- read <$> count 2 digit
  space
  month <- monthParser
  space
  year <- read <$> count 4 digit
  pure $ Date day month year
```

- Jak bude vypadat parser `monthParser`
- Jaká má parser omezení?

Příklad kombinace parserů v do-bloku – pokračování

Příklad: *02 Mar 2022*

```
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec

monthParser :: Parser Month
monthParser = choice [ string "Jan" >> pure Jan
                     , string "Feb" $> Feb
                     , Mar <$ string "Mar"
                     , string "Apr" *> pure Apr
                     , pure May <* string "May"
                     , ...
                     ]
```

Příklad kombinace parserů v do-bloku – pokračování

Příklad: *02 Mar 2022*

```
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec

monthParser :: Parser Month
monthParser = choice [ string "Jan" >> pure Jan
                     , string "Feb" $> Feb
                     , Mar <$ string "Mar"
                     , string "Apr" *> pure Apr
                     , pure May <* string "May"
                     , ...
                     ]
```

- Ve vlastním kódu si vyberte jen jeden styl a buďte konzistentní!

Příklad kombinace parserů v do-bloku – pokračování

Příklad: *02 Mar 2022*

```
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec

monthParser :: Parser Month
monthParser = choice [ string "Jan" >> pure Jan
                     , string "Feb" $> Feb
                     , Mar <$ string "Mar"
                     , string "Apr" *> pure Apr
                     , pure May <* string "May"
                     , ...
                     ]
```

- Ve vlastním kódu si vyberte jen jeden styl a buďte konzistentní!
- Je tento parser korektní?

Alternativa parserů

```haskell
<|> :: Parser a -> Parser a -> Parser a
```

- `p <|> q` se pokusí nejdříve použít parser `p` a jestli uspěje, vrátí jeho výsledek
- jestli `p` selže bez toho, aby spotřeboval nějaký vstup, použije se parser `q`
- Příklad: `string "spring" <|> string "autumn"`

Alternativa parserů

```
<|> :: Parser a -> Parser a -> Parser a
```

- `p <|> q` se pokusí nejdříve použít parser `p` a jestli uspěje, vrátí jeho výsledek
- jestli `p` selže bez toho, aby spotřeboval nějaký vstup, použije se parser `q`
- Příklad: `string "spring" <|> string "autumn"`

Existující užitečné parsery:

```
option :: a -> Parser a -> Parser a
optionMaybe :: Parser a -> Parser (Maybe a)
optional :: Parser a -> Parser ()
```

Alternativa parserů

```
season = string "spring" <|> string "summer"
```

Alternativa parserů

```
season = string "spring" <|> string "summer"
```

`season` na řetězci `"summer"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

Alternativa parserů

```
season = string "spring" <|> string "summer"
```

`season` na řetězci `"summer"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

```
try :: Parser a -> Parser a
```

- `try p` se chová stejně jako parser `p`, ale když `p` selže, vrátí se ve vstupu, jako by žádný nebyl parserem `p` spotřebován

Alternativa parserů

```
season = string "spring" <|> string "summer"
```

`season` na řetězci `"summer"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

```
try :: Parser a -> Parser a
```

- `try p` se chová stejně jako parser `p`, ale když `p` selže, vrátí se ve vstupu, jako by žádný nebyl parserem `p` spotřebován
- Pozor: používání `try` zvyšuje složitost parsování! (Ale na druhou stranu nám umožňuje mít libovolný *lookahead*.)

Příklad kombinace parserů v do-bloku – dokončení

Příklad: *02 Mar 2022*

```
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec

monthParser :: Parser Month
monthParser = choice [ try $ Jan <$ string "Jan"
                     ,       Feb <$ string "Feb"
                     , try $ Mar <$ string "Mar"
                     , try $ Apr <$ string "Apr"
                     ,       May <$ string "May"
                     , ...
                     ]
```

- `try` není potřeba všude

Užitečné chybové hlášky

Nevýhoda kombinátorových parserů: Ladění není snadné.
(Který parser selhal?)

Užitečné chybové hlášky

Nevýhoda kombinátorových parserů: Ladění není snadné.
(Který parser selhal?)

Zlepšení: Doplňme do parseru popis, co dělá:

```
"your input:" (line 1, column 1):
unexpected "L"
expecting digit
```

Užitečné chybové hlášky

Nevýhoda kombinátorových parserů: Ladění není snadné.
(Který parser selhal?)

Zlepšení: Doplňme do parseru popis, co dělá:

```
"your input:" (line 1, column 1):
unexpected "L"
expecting digit
```

```
(<?>) :: Parser a -> String -> Parser a
```

- mění chybovou hlášku („expected“)
- obzvlášť vhodné pro alternativy nebo `try`
- parser nesmí spotřebovat vstup

Ve skutečnosti je to obecnější…

Typ `Parser a` je pouze speciální případ parseru:

```
type Parser a = Parsec String () a
type Parsec s u a = ParsecT s u Identity a
```

`ParsecT s u m a` představuje parser, který

- zpracovává typ `s`
- udržuje si stav typu `u`
- pracuje v monádě `m`
- vrací výsledek typu `a`

Ve skutečnosti je to obecnější…

Typ `Parser a` je pouze speciální případ parseru:

```
type Parser a = Parsec String () a
type Parsec s u a = ParsecT s u Identity a
```

`ParsecT s u m a` představuje parser, který

- zpracovává typ `s`
- udržuje si stav typu `u`
- pracuje v monádě `m`
- vrací výsledek typu `a`

Typ funkce `parse` je pak následovný:

```
runParser :: Stream s Identity t =>
    Parsec s u a -> u -> SourceName -> s -> Either ParseError a
```

Příklady k procvičení

Napište parser pro:

- datum – `"2015-04-14"`
  rozpoznaný řetězec číslic můžete konvertovat funkcí `read`, výsledek ať je vámi definovaného typu **`Date`**, rozsah kontrolovat nemusíte
- čísla která můžou mít desetinnou část – `"12"`, `"12.54"`
  výsledek ať je typu **`Float`**
- seznam čísel – `"[ 12, 12.54, 42, 66, 3.14 ]"`
  využijte parser z předchozího cvičení
  výsledek ať je typu `[`**`Float`**`]`
- výraz s přirozenými čísly a binárními operacemi: `((2+5)*2)`
  každá aplikace operace musí být v závorkách, tedy precedenci operátorů neřešte
  parser ať výraz rovnou vyhodnotí, tj. ať je typu **`Parser Int`**