Functional Programming

Parsing

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Recall the expression language

Definition

```
1 data Term = Con Integer
2           | Bin Term Op Term
3             deriving (Eq, Show)
4
5 data Op = Add | Sub | Mul | Div
6             deriving (Eq, Show)
```

Recall the expression language

Definition

```
1 data Term = Con Integer
2           | Bin Term Op Term
3             deriving (Eq, Show)
4
5 data Op = Add | Sub | Mul | Div
6             deriving (Eq, Show)
```

Parsing expressions

- Read a string like 3+42/6
- Recognize it as a valid term
- Return `Bin (Con 3) Add (Bin (Con 42) Div (Con 6))`

Parsing

The type of a simple parser

```
1 type Parser token result = [token] -> [(result, [token])]
```

Combinator parsing

Primitive parsers

```
1 pempty :: Parser t r
2 succeed :: r -> Parser t r
3 satisfy :: (t -> Bool) -> Parser t t
4 msatisfy :: (t -> Maybe a) -> Parser t a
5 lit :: Eq t => t -> Parser t t
```

Combinator parsing II

Combination of parsers

```
1 palt :: Parser t r -> Parser t r -> Parser t r
2 pseq :: Parser t (s -> r) -> Parser t s -> Parser t r
3 pmap :: (s -> r) -> Parser t s -> Parser t r
```

A taste of compiler construction

A lexer

A lexer partitions the incoming list of characters into a list of tokens. A token is either a single symbol, an identifier, or a number. Whitespace characters are removed.

Underlying concepts

Parsers have a rich structure

- many concepts from category theory can be mapped to programming concepts
- parsing illustrates many of these concepts

Functors

The functor class

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Instances

List, Maybe, IO, ...

Functorial laws

```
fmap id_a == id_f_a
fmap (f . g) == fmap f . fmap g
```

Parsing is . . .

A functor

Check the functorial laws!

A monad

Check the monad laws!

Consequence

Can use do notation for parsing!

Applicative

Example 1: sequencing computation

```
1 sequence :: [IO a] -> IO [a]
2 sequence [] = return []
3 sequence (io:ios) = do x <- io
4                        xs <- sequence ios
5                        return (x:xs)
```

Alternative way

```
1 sequence [] = return []
2 sequence (io:ios) = return (:) `ap` io `ap` sequence ios
3
4 return :: Monad m => a -> m a
5 ap :: Monad m => m (a -> b) -> m a -> m b
```

Applicative

Example 2: transposition

```
1 transpose :: [[a]] -> [[a]]
2 transpose [] = repeat []
3 transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

Rewrite

```
1 transpose [] = repeat []
2 transpose (xs:xss) = repeat (:) `zapp` xs `zapp` transpose xss
3
4 zapp :: [a -> b] -> [a] -> [b]
5 zapp fs xs = zipWith ($) fs xs
```

Applicative Interpreter

Standard interpretation

```
1  data Exp v
2     = Var v
3     | Val Int
4     | Add (Exp v) (Exp v)
5
6  eval :: Exp v -> Env v -> Int
7  eval (Var v) env = fetch v env
8  eval (Val i) env = i
9  eval (Add e1 e2) env = eval e1 env + eval e2 env
10
11 type Env v = v -> Int
12 fetch :: v -> Env v -> Int
13 fetch v env = env v
```

Applicative Interpreter

Alternative implementation

```
1 eval' :: Exp v -> Env v -> Int
2 eval' (Var v) = fetch v
3 eval' (Val i) = const i
4 eval' (Add e1 e2) = const (+) `ess` (eval' e1) `ess` (eval' e2)
5
6 ess a b c = (a c) (b c)
```

Applicative

Extract the common structure

```
1 class Functor f => Applicative f where
2   pure :: a -> f a
3   (<*>) :: f (a -> b) -> f a -> f b
```

Wrapup

- what if there are multiple applicatives?

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation
- applicatives cannot express dependency

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation
- applicatives cannot express dependency
- enable more clever parsers