Functional Programming

Parsing

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Recall the expression language

Definition

```haskell
data Term = Con Integer
          | Bin Term Op Term
            deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
            deriving (Eq, Show)
```

Recall the expression language

Definition

```
data Term = Con Integer
          | Bin Term Op Term
            deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
            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

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

Combinator parsing

Primitive parsers

```
pempty :: Parser t r
succeed :: r -> Parser t r
satisfy :: (t -> Bool) -> Parser t t
msatisfy :: (t -> Maybe a) -> Parser t a
lit :: Eq t => t -> Parser t t
```

Combinator parsing II

Combination of parsers

```
palt :: Parser t r -> Parser t r -> Parser t r
pseq :: Parser t (s -> r) -> Parser t s -> Parser t r
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

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (io:ios) = do x <- io
                       xs <- sequence ios
                       return (x:xs)
```

Alternative way

```
sequence [] = return []
sequence (io:ios) = return (:) `ap` io `ap` sequence ios

return :: Monad m => a -> m a
ap :: Monad m => m (a -> b) -> m a -> m b
```

Applicative

Example 2: transposition

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

Rewrite

```
transpose [] = repeat []
transpose (xs:xss) = repeat (:) `zapp` xs `zapp` transpose xss

zapp :: [a -> b] -> [a] -> [b]
zapp fs xs = zipWith ($) fs xs
```

Applicative Interpreter

Standard interpretation

```
data Exp v
  = Var v
  | Val Int
  | Add (Exp v) (Exp v)

eval :: Exp v -> Env v -> Int
eval (Var v) env = fetch v env
eval (Val i) env = i
eval (Add e1 e2) env = eval e1 env + eval e2 env

type Env v = v -> Int
fetch :: v -> Env v -> Int
fetch v env = env v
```

Applicative Interpreter

Alternative implementation

```
eval' :: Exp v -> Env v -> Int
eval' (Var v) = fetch v
eval' (Val i) = const i
eval' (Add e1 e2) = const (+) `ess` (eval' e1) `ess` (eval' e2)

ess a b c = (a c) (b c)
```

Applicative

Extract the common structure

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

Applicative

Laws

- Identity

```
pure id <*> v == v
```

- Composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

- Homomorphism

```
pure f <*> pure x = pure (f x)
```

- Interchange

```
u <*> pure y = pure ($ y) <*> u
```

Parsers are Applicative!

```
instance Applicative (Parser' token) where
  pure = return
  (<*>) = ap

instance Alternative (Parser' token) where
  empty = mzero
  (<|>) = mplus
```

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