Parsers & Monads

Kim Bruce
for Michael Greenberg
Recall Parser

• newtype Parser a = Parser{parse :: String -> Maybe (a,String)}

• letter :: Char -> Parser Char
  letter c = Parser $ \s ->
    case s of
      c':s' | c == c' -> Just (c,s')
      _ -> Nothing

• letters :: String -> Parser String
  letters str = Parser $ \s ->
    if take (length str) s == str
      then Just (str,drop (length str) s)
      else Nothing
Easy to Use

• parse (letter ‘c’) “chocolate”
  => Just (‘c’, ‘hocolate’)

• parse (letters "choco") "chocolate"
  => Just ("choco","late")
Functors & Applicative

• Recall:

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

  • class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b  — application inside a context
Parsers are …

Functor & Applicative

```haskell
instance Functor Parser where
  fmap f p = Parser $ \s ->
    case parse p s of
      Nothing -> Nothing
      Just (v,s') -> Just (f v, s')

instance Applicative Parser where
  pure a = Parser $ \s -> Just (a,s)
  f <*> a = Parser $ \s ->
    case parse f s of
      Nothing -> Nothing
      Just (g,s') -> parse (fmap g a) s'
```
Alternative

is left-biased choice

instance Alternative Maybe where
  -- empty :: f a
  empty = Nothing

  -- (<|>) :: f a -> f a -> f a
  Just x <|> _ = Just x
  Nothing <|> r = r
  -- empty <|> f == f
  -- f <|> empty == f

instance Alternative [] where
  empty = []
  [] <|> r = r
  _ <|> r = r
  l <|> _ = l
Parsers are ...

... also Alternative

instance Alternative Parser where
  empty = Parser $ \s -> Nothing
  p1 <|> p2 = Parser $ \s ->
    case parse p1 s of
      Just (a,s') -> Just (a,s')
      Nothing    -> parse p2 s

If p1 succeeds on s, give its result,
otherwise return result of parsing with p2
Parsing Arithmetic

Look at file lec11-setup.hs

*Recall:*

— throw away second result

\[(<*) :: f \ a \to f \ b \to f \ a\]

\[fa <*> fb = pure (\_ \_ \to x) <> fa <*> fb\]

— throw away first result

\[(*) :: f \ a \to f \ b \to f \ b\]

\[fa *> fb = pure (\_ y \to y) <> fa <*> fb\]
Toward Monads

- Parsers were both Functors and Applicative
  - newtype Parser a = Parser{parse :: String -> Maybe (a,String)}
  - Powerful operations and laws
  - Can even do:
    ```haskell
    data Foo = Bar Int Int Char
    parseFoo :: Parser Foo
    parseFoo = Bar <$> parseInt <*> parseInt <*> parseChar
    ```
- But they can’t handle everything
Hard Case

- Parse 4 78 19 3 44 3 1 7 5 2 3 2, where first number tells length of first group, next length of next, etc.

- [78, 19, 3, 44], [1, 7, 5], [3, 2]

- Want parseFile :: Parser [[[Int]]]
  - but Applicative not strong enough!
class Monad m where
  return :: a -> m a

  (>>>=) :: m a -> (a -> m b) -> m b

  (>>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>>= \_ -> m2

Last is definable from >>>=

return is like pure
Bind

• Recall monad is a context for computation

• Power is from >>= (called “bind”)
  
  • (>>=) :: m a -> (a -> m b) -> m b

  • Idea: choose second computation (via second argument) based on value of first.
Maybe Monad

instance Monad Maybe where
  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
  return x = Just x

>>= preserves “Nothing”,

>>= unwraps argument to compute w/ a Just’ed value
  Just x >>= f  gives  f x

Second arg of >>= is function to be applied to unwrapped value

Abbreviate compu >>= \x → exp  as
do x <- compu
  exp


An example

- dormRooms = ["Jack",10),("Jill",20),("Ann",20)]

- phonesForRooms = [(10,23434),(20,23435),(30,23438)]

- getDormFor name [] = Nothing
  -- 2nd arg is name-room pairs
  getDormFor name ((nm,rm):rest) = if nm == name
    then Just rm
    else getDormFor name rest

- getPhoneForRoom rm [] = Nothing
  getPhoneForRoom rm ((rmnum,phone):rest) =
    if rm == rmnum then Just phone
    else getPhoneForRoom rm rest
An example

- `getPhoneForName name rooms phones =
  case getDormFor name rooms of
    Nothing -> Nothing
    Just rm -> getPhoneForRoom rm phones`

What if we could ignore Nothing!

- `getPhoneForName name rooms phones =
  getDormFor name rooms >>=
    (\rm -> getPhoneForRoom rm phones)`

Even nicer:

- `getPFN name rooms phones =
  do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
  return num`