Analizadores por combinación

á la Swierstra & Jeuring

Francisco José Cháves

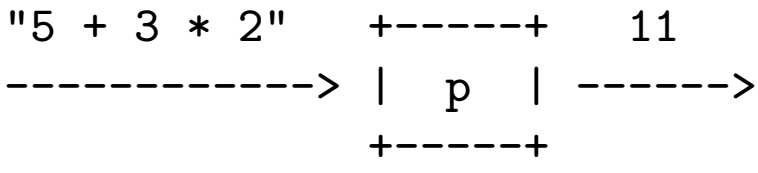
Bloom

4 de marzo de 2017

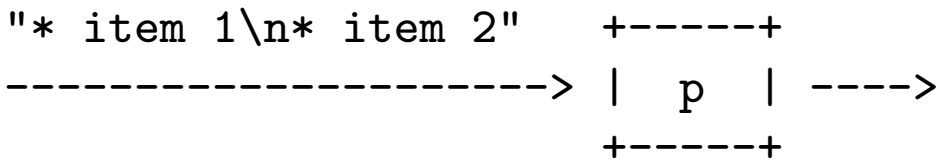
Parser

Problema: hacer un *algoritmo* "`p`" que tome un texto y lo *entienda*.

Ejemplo: una expresión aritmética



Ejemplo: un texto markdown:



- item 1
- item 2

Listas en Haskell

Dos constructores:

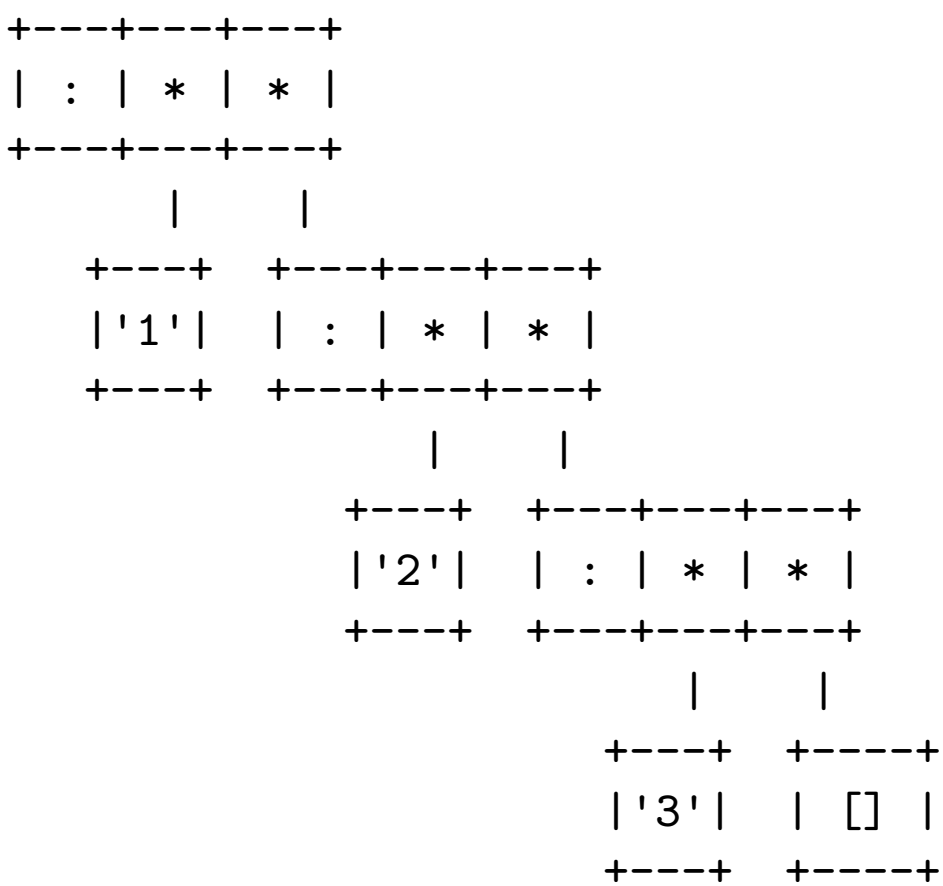
- la lista vacía (`Empty`),
- o un elemento seguido de una lista (`Prepend`)

```
data List a where
    Empty :: List a
    Prepend :: a -> List a -> List a

data [a] where
    [] :: [a]
    (:) :: a -> [a] -> [a]
```

El operador `(:)` asocia a derecha.

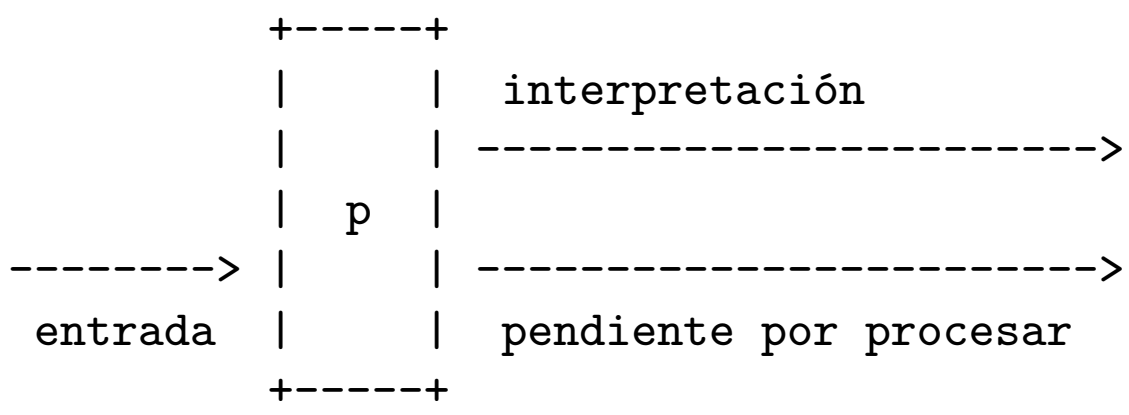
Ejemplo

- La lista `"123" = ['1','2','3'] = '1':('2':('3':[])) = '1':'2':'3':[]`.
- La representación en memoria es:



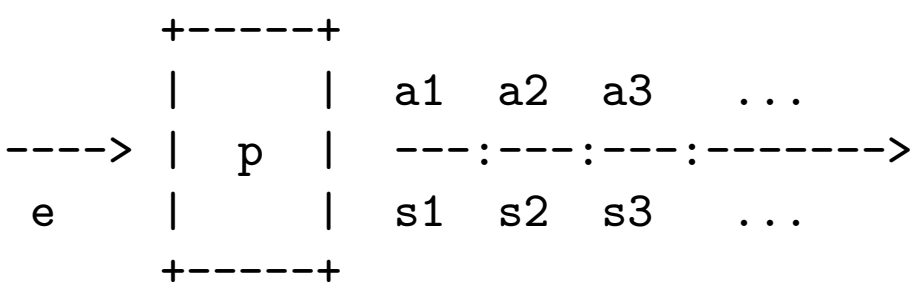
Análisis por combinación

Dividir y conquistar:



Parsing

En general puede haber varias interpretaciones, para cada interpretación hay una parte pendiente por procesar:




```
type Parser s a = [s] -> [(a,[s])]
```

- `a`: tipo de las interpretaciones
- `s`: tipo de los elementos de la secuencia de entrada (para texto `s = Char`)

Advertencia

En haskell no se pueden tener instancias de una clase con un type alias, hay que crear un nuevo tipo de datos:

```
newtype Parser s a = Parser ([s] -> [(a,[s])])
```

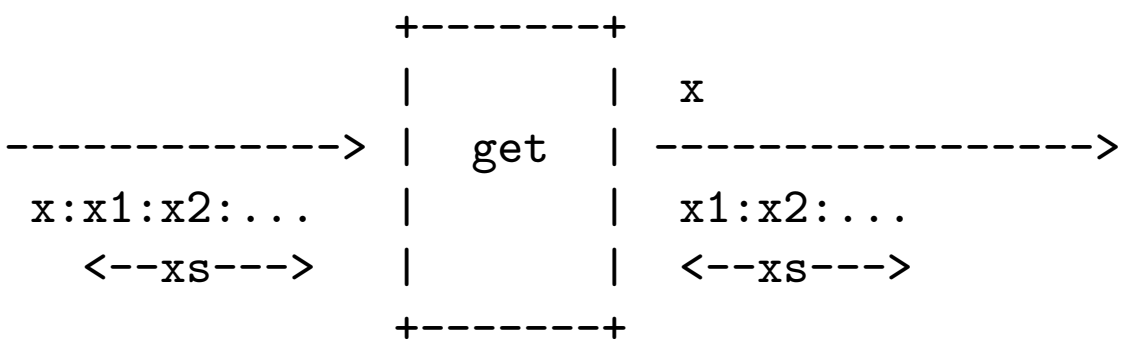
Usualmente se coloca en un record:

```
newtype Parser s a = Parser { runParse :: ([s] -> [(a,[s])]) }
```

- La definición con `newtype` crear un nuevo tipo de datos,
- restricción: un sólo constructor y un sólo parámetro,
- permite hacer instancias de clases;
- para esta presentación, por simplicidad, utilizamos el type alias definido anteriormente.

Analizadores básicos (get)

Reconocer un elemento en la entrada:

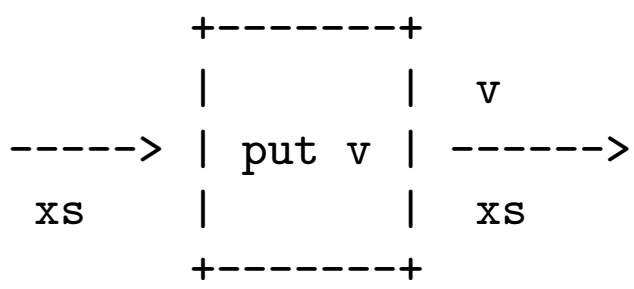


Una sola interpretación en el resultado

```haskell
get :: Parser s s
get     [] = [] -- No hay elementos en la entrada
get (x:xs) = [(x,xs)]
```

Analizadores básicos (put)

Poner un valor `v` en la interpretación sin procesar la entrada:




```
put :: a -> Parser s a
put v xs = [(v,xs)]
```

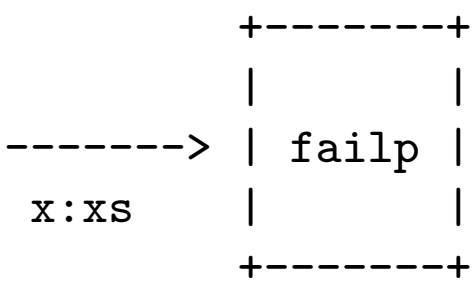
- Un alias de `put` es `succeed`:

  ```
  succeed :: a -> Parser s a
  succeed = put
  ```

- Note que `succeed` reconoce la secuencia vacía dando como resultado `v`

Analizadores básicos (`failp`)

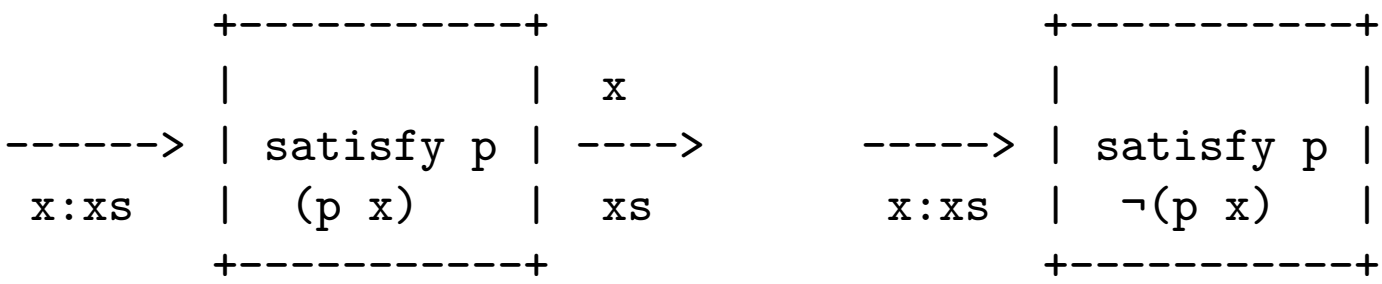
Fallar (elimina todas las interpretaciones):




```
failp :: Parser s a
failp xs = []
```

Analizador satisfy

Si se cumple "`p x`" se comporta como "`get`"; de lo contrario "`failp`"




```
satisfy :: (s -> Bool) -> Parser s s
satisfy p (x:xs) | p x = [(x,xs)]
satisfy p _ = []
```

Analizadores

```haskell
symbol :: Eq s => s -> Parser s s -- el siguiente elemento en
symbol x = satisfy (== x)         -- la entrada es x

digit :: Parser Char Char     -- dígito
digit = satisfy isDigit

space :: Parser Char Char     -- ' ','\t','\n','\r','\f','\v'
space = satisfy isSpace

letter :: Parser Char Char    -- letra
letter = satisfy isLetter

alphaNum :: Parser Char Char    -- alfanumérico
alphaNum = satisfy isAlphaNum


...
```

Analizadores

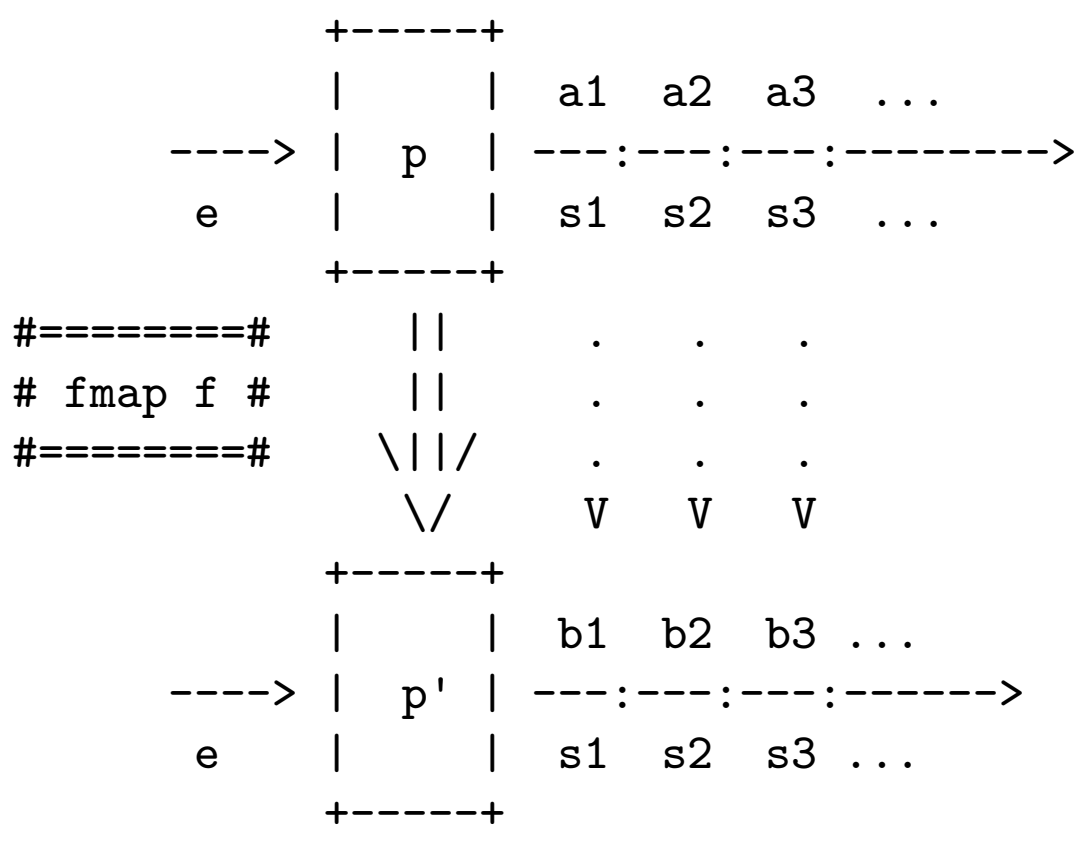
```
epsilon :: Parser s ()          -- secuencia vacía
epsilon = succeed ()

end :: Parser s ()              -- Fin de la entrada
end [] = [((),[])]
end xs  = []
```

Analizadores por combinación

Consejo: pensar en generalizar conceptos de funciones.

Transformación de analizadores (<$>)

Sea `f :: a -> b`,



Transformación de analizadores (<$>)

```
fmap :: (a -> b) -> Parser s a -> Parser s b
fmap f p = \e -> [ (f ai,si) | (ai,si) <- p e ]
```

- En forma de operador:

  ```
  infixl 4 <$>
  (<$>) :: (a -> b) -> Parser s a -> Parser s b
  (<$>) = fmap
  ```

- En la practica, es útil (<$), que reemplaza cada interpretación por `v`

  ```
  infixl 4 <$
  (<$) :: a -> Parser s b -> Paser s a
  (v <$ p) e = (\ _ -> v) <$> p e
  ```

Functor

Intuitivamente:

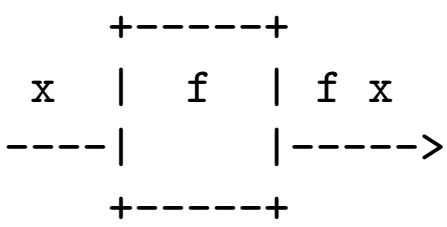
- Los elementos de tipo "`c a`", son elementos que "contienen" o "manipulan" elementos de tipo "`a`" (`[a]`, `Set a`, `(Parser s) a`, etc.).
- Si la función "`f`" transforma elementos de tipo "`a`" en elementos de tipo "`b`", "`fmap`", utilizando "`f`", transforma elementos de tipo "`c a`" en elementos de tipo "`c b`".

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

instance Functor (Parser s) where
  fmap = Parser (\e -> [ (f ai,si) | (ai,si) <- p e ])
```

Comparando:

```
f      ::              a ->             b
fmap f ::           c a ->          c b
fmap f :: (Parser s) a -> (Parser s) b
```

Aplicación funcional



haciendo visible la aplicación funcional:

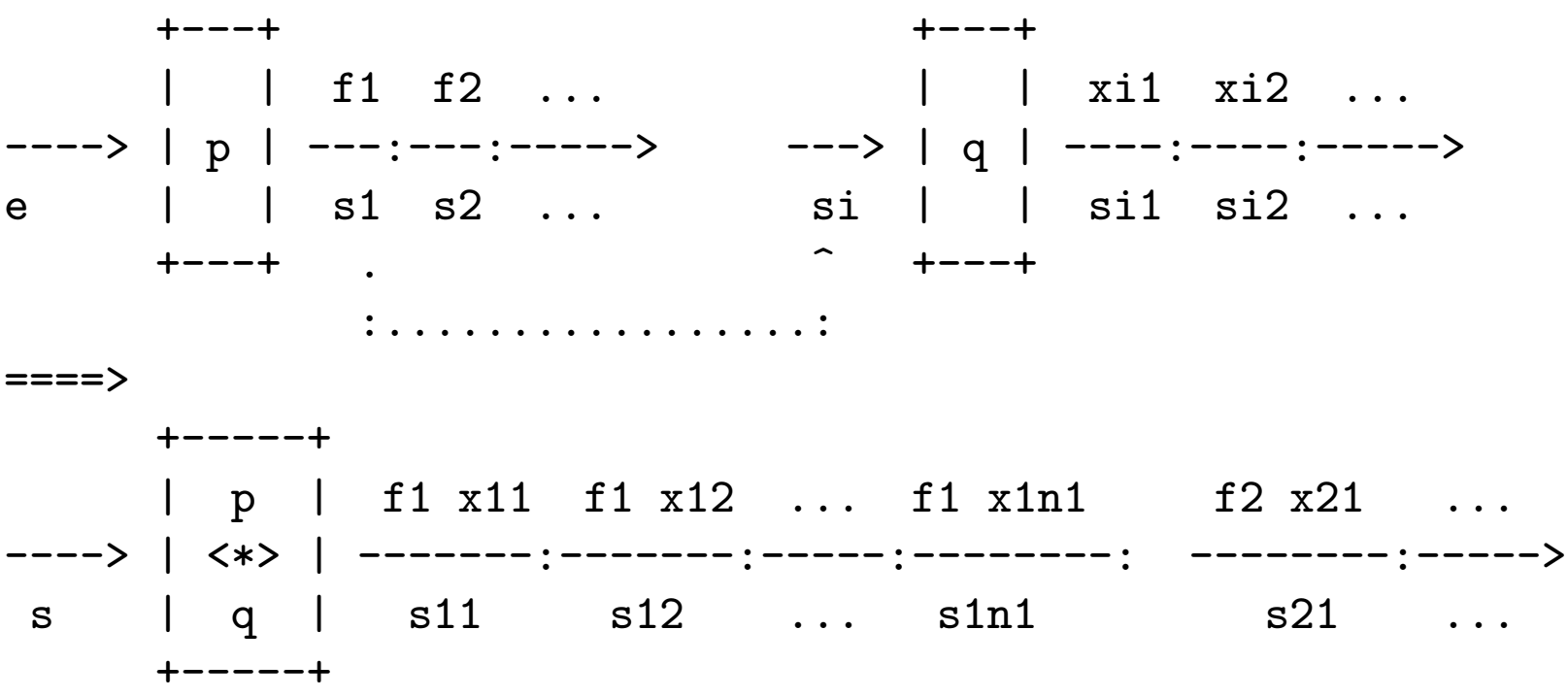
```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

algunos lenguajes definen la aplicación invertida:

```
infixl 0 |>
(|>) :: a -> (a -> b) -> b
x |> f = f x
```

Composición secuencial (<*>)

- También llamada producto,
- aplica el analizador "p" a la entrada "e", aplica "q" a cada salida "si" de "p",
- las interpretaciones de p son funciones,
- las interpretaciones de q son sus argumentos.



Composición secuencial

```
infixl 4 <*>
(<*>) :: Parser s (a -> b) -> Parser s a -> Parser s b
(p <*> q) e = [ (f x,s') | (f,s) <- p e, (x,s') <- q s ]
                          ----------------------^
```

- En la práctica también se utiliza (<*) y (*>)

```
infixl 4 <*
(<*) :: Parser s a -> Parser s b -> Parser s a
p <* q = (\a b -> a) <$> p <*> q

infixl 4 *>
(*>) :: Parser s a -> Parser s b -> Parser s b
p *> q = (\a b -> b) <$> p <*> q
```

Functor Aplicativo

Intuitivamente:

- En la expresión "`f $ x`" se aplica la función "`f :: a -> b`" al argumento "`x :: a`".
- El operador producto (<*>) aplica funciones de tipo "`a -> b`" a argumentos de tipo "`a`" que están dentro de elementos de tipo "`c`".

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

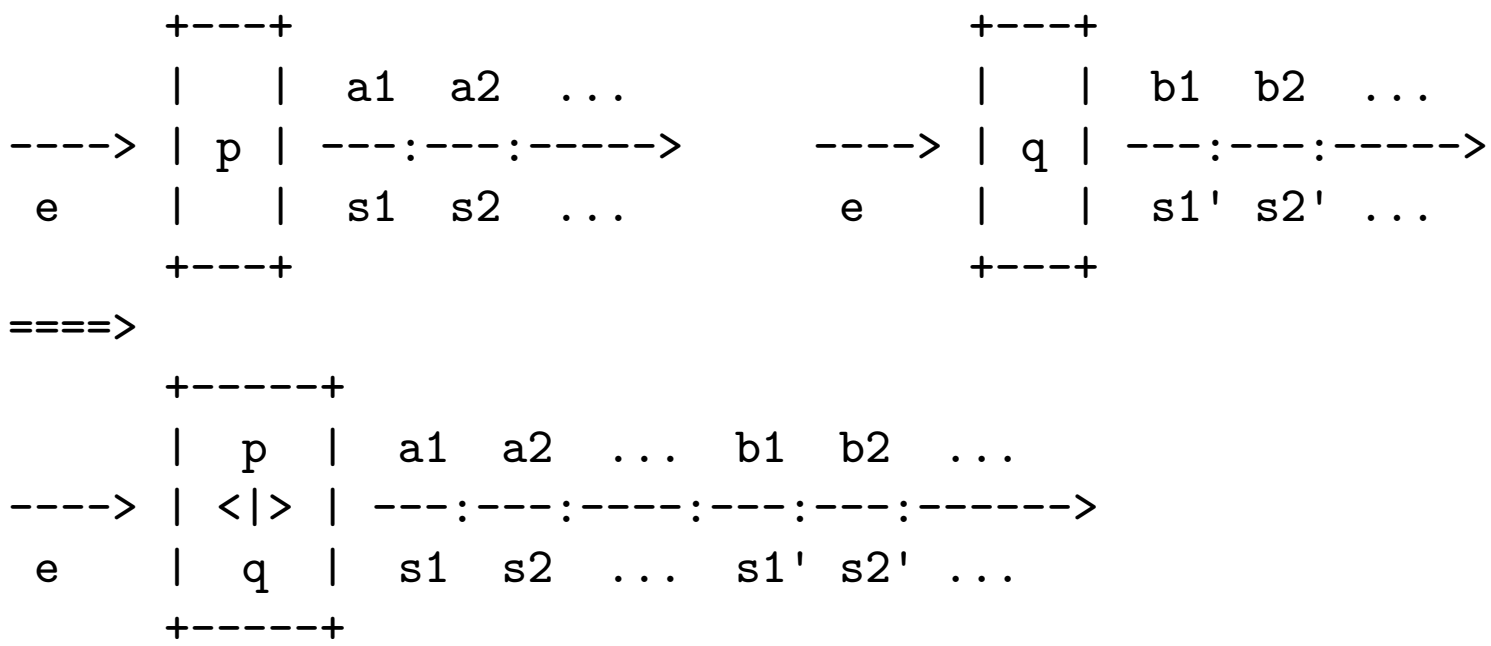
En la instancia de "`Parser`", "`pure`" es "`succeed`".

Comparando:

```
($)   ::             (a -> b) ->           a ->           b
(<*>) ::           c (a -> b) ->         c a ->         c b
(<*>) :: (Parser s) (a -> b) -> (Parser s) a -> (Parser s) b
```

Composición alternativa

- También llamada “ó”, son los posibles análisis de los parsers `p` y `q`.



Composición alternativa

```
infixr 3 <|>
(<|>) :: Parser s a -> Parser s a -> Parser s a
(p <|> q) e = p e ++ q e
```

Advertencia

En otras presentaciones (Hutton), si el análisis con `p` falla, "`p <|> q = q`", de lo contrario "`p <|> q = p`". Algunas implementaciones utilizan backtracking, otras son voraces.

Alternativos

Intuitivamente:

El concepto de monoide extendido a functores aplicativos.

```
class Applicative f => Alternative (f :: * -> *) where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

En la instancia "`empty = failp`"

Comparando

Recuerde que:

```
type Parser s a = [s] -> [(a,[s])]
```

```
[]    ::                [a]
failp :: [s] -> [(a,[s])]
```

```
(++)  ::                [a] ->                [a] ->                [a]
(<|>) :: [s] -> [(a,[s])] -> [s] -> [(a,[s])]  -> [s] -> [(a,[s])]
```

Mas analizadores

```haskell
many :: Parser s a -> Parser s [a] -- Cero o mas veces p
many p = some p <|> succeed []

some :: Parser s a -> Parser s [a] -- Una o mas veces p
some p = (:) <$> p <*> many p

digits :: Parser Char String   -- secuencia de dígitos
digits = some digit

identifier :: Parser Char String  -- identificador
identifier = (:) <$> letter <*> many alphaNum

spaces :: Parser Char () -- espacios
spaces = () <$ many space
```

Mas analizadores

```haskell
-- el analizador p delimitado por los analizadores open y close
between :: Parser s a -> Parser s b -> Parser s c -> Parser s c
between open close p = open *> p <* close

-- acepta la secuencia (x:xs)
accept :: Eq s => [s] -> Parser s [s]
accept [] = succeed []
accept (x:xs) = (:) <$> symbol x <*> accept xs

-- Una lista de p separados por sep
sepBy :: Parser s a -> Parser s b -> Parser s [a]
sepBy p sep = (:) <$> p <*> many (sep *> p)
            <|> succeed []

-- Cero o una vez p
option :: a -> Parser s a -> Parser s a
option x p = p <|> succeed x
```

Mas analizadores

```
string :: Parser Char String
string :: Parser Char String
string = between (symbol '"') (symbol '"')
                 (many ('"' <$ accept "\\\"" <|> satisfy (/= '"')))

number :: Parser Char Double  -- numero
number = read <$> pnum
   where neg  = "-" <$ symbol '-'
         sig  = "+" <$ symbol '+' <|> "-" <$ symbol '-'
         frac = (:) <$> symbol '.' <*> digits
         exp  = con4 <$> letE <*> sig <*> digits <*> put ""
         letE = "e" <$ (symbol 'e' <|> symbol 'E')
         nat  = accept "0"
              <|> (:) <$> satisfy (\d -> '0' < d && d <= '9')
                  <*> many digit
         pnum = con4 <$> option "" neg <*> pos <*> option "" frac
                <*> option "" exp
         con4 x1 x2 x3 x4 = concat [x1,x2,x3,x4]
```

Analizando JSON

```haskell
data JValue where
     JBool :: Bool -> JValue
     JNull :: JValue
     JString :: String -> JValue
     JNumber :: Double -> JValue
     JObject :: [(String, JValue)] -> JValue
     JArray  :: [JValue] -> JValue
     deriving (Eq, Ord, Show)
```

Analizador elemental JSON

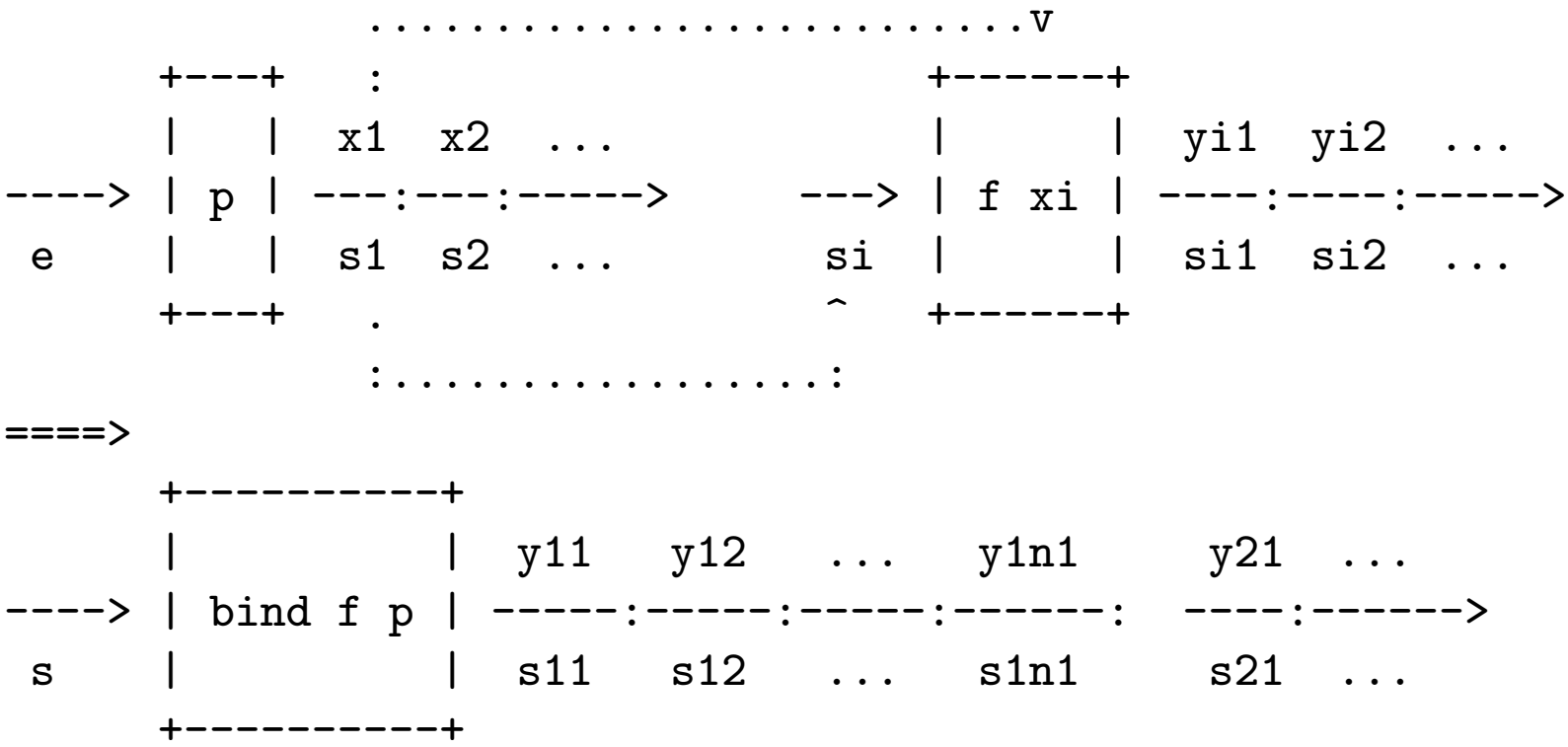
```haskell
jbool :: Parser Char JValue
jbool = JBool True <$ accept "true"
      <|> JBool False <$ accept "false"

jnull :: Parser Char JValue
jnull = JNull <$ accept "null"

jstring :: Parser Char JValue
jstring = JString <$> string

jnumber :: Parser Char JValue
jnumber = JNumber <$> number
```

Analizador elemental JSON

```
jarray :: Parser Char JValue
jarray = JArray <$> between (symbol '[') (symbol ']')
                    (sepBy json (symbol ','))

jobject :: Parser Char JValue
jobject = JObject <$> between (symbol '{') (symbol '}') jprops

jprops :: Parser Char [(String,JValue)]
jprops = sepBy jprop (symbol ',')

jprop :: Parser Char (String,JValue)
jprop = (,) <$ spaces <*> string
            <* spaces <* (symbol ':') <*> json

json :: Parser Char JValue
json = spaces *> (jbool <|> jnull <|> jstring <|> jnumber
                 <|> jobject <|> jarray) <* spaces
```

Demo

Composición monádica (>>=)



Composición monádica (>>=)

```
bind :: (a -> Parser s b) -> Parser s a -> Parser s b
bind f p e = [ r | (x,s) <- p e, r <- (f x) s]
                     :...................^..^

(>>=) ::  Parser s a -> (a -> Parser s b) -> Parser s b
p >>= f = bind f p
```

- Note que la función "`f :: a -> Parser s b`" es entre dos mundos.

Composición monádica

Intuitivamente

- La expresión "`x |> f`" utiliza "`x :: a`" como argumento de "`f :: a-> b`" para obtener un resultado de tipo "`b`"
- La expresión "`p >>= f`" toma los elementos dentro de "`p :: c a`", los utiliza como argumentos de "`f :: a -> c b`", combina todas las respuestas "`c b`" dentro de un solo elemento de tipo "`c b`"

```
class Applicative m => Monad (m :: * -> *) where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

En la instancia "`return = pure = succeed`".

Comparando

```
(|>)  ::              a -> (a ->            b) ->            b
(>>=) ::          c a -> (a ->        c b) ->        c b
(>>=) :: (Parser s) a -> (a -> (Parser s) b) -> (Parser s) b
```

Notación do

La notación "do" es un azúcar sintáctico para expresiones monádicas anidadas:

```
(p >>= (\x ->
 q >>= (\y ->
 ( ... return r))))
```

se escribe:

```
do x <- p
   y <- q
   ...
   return r
```

Aplicación de los analizadores monádicos

El lenguaje $a^n b^n c^n$ no es libre del contexto (lema de bombeo CFL).

```
anbncn :: Parser Char [Char]
anbncn = do
    an <- many (symbol 'a')
    let n = length an
    bn <- accept (replicate n 'b')
    cn <- accept (replicate n 'c')
    end
    return (an ++ bn ++ cn)
```

Note que esta estrategia puede ser útil para procesar lenguajes como haskell donde los bloques se deducen de la indentación.

Retrospectivamente

Comparando retrospectivamente:

- aplicación "($)",
- functor "(<$>)",
- functor aplicativo "(<*>)" y
- mónada "(>>=)":

```
($)    ::           (a -> b) ->          a ->          b
(<$>)  ::           (a -> b) -> Parser s a -> Parser s b
(<*>)  :: Parser s (a -> b) -> Parser s a -> Parser s b
bind   :: (a -> Parser s b) -> Parser s a -> Parser s b
```

Preguntas?