Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Parsers monádicos

Mónada de Parsing

```
newtype Parser a = P (String → [(a, String)])

runP :: Parser a → String → [(a, String)]
runP (P p) = p

instance Monad Parser where
  return a      = P $ λcs → [(a, cs)]
  (P p) >>= f = P $ λcs →
                  concat [runP (f a) cs' | (a, cs') ← p cs]
```

Parsing: combinadores básicos

```
pFail :: Parser a
pFail = P $ λcs → []

item :: Parser Char
item = P $ λcs → case cs of
                   ""     → []
                   (c : cs) → [(c, cs)]

pSat :: (Char → Bool) → Parser Char
pSat p = do c ← item
            if p c then return c else pFail

pSym :: Char → Parser Char
pSym c = pSat (== c)
```

Parsing: Alternativa

$$(<|>) :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a$$
$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to p\ cs +\!\!+ q\ cs$$

Parsing: Alternativa

$$(<|>) :: Parser\ a \to Parser\ a \to Parser\ a$$
$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to p\ cs +\!\!+ q\ cs$$

Otra forma de definir el operador de alternativa:

$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to \texttt{case}\ p\ cs +\!\!+ q\ cs\ \texttt{of}$$
$$\begin{array}{ll} [\,] & \to [\,] \\ (x : xs) & \to [x] \end{array}$$

many y some

p^* (many) cero o más veces p

```
pList :: Parser a → Parser [a]
pList p = do a ← p
             as ← pList p
             return (a : as)
          <|>
          return []
```

p^+ (some) una o más veces p

```
pListP :: Parser a → Parser [a]
pListP p = do a ← p
              as ← pList p
              return (a : as)
```

Ejemplo: digits

```
digit :: Parser Int
digit = do c ← pSat isDigit
           return (ord c − ord '0')

isDigit c = (c ⩾ '0') ∧ (c ⩽ '9')

digits :: Parser [Int]
digits = pListP digit

sumDigits :: Parser Int
sumDigits = do ds ← digits
               return (sum ds)
```

Ejemplo: `number`

```
number :: Parser Int
number = do d ← digit
            number' d

number' :: Int → Parser Int
number' n = do d ← digit
               number' (n * 10 + d)
            <|>
            return n
```

Ejemplo: `number`

```
number :: Parser Int
number = do d ← digit
            number' d

number' :: Int → Parser Int
number' n = do d ← digit
               number' (n * 10 + d)
            <|>
            return n
```

Esto equivale a la siguente definición:

```
number = do (d : ds) ← digits
            return (foldl (⊕) d ds)
         where
           n ⊕ d = n * 10 + d
```

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

$$\texttt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

```
data Expr = Val Int | Add Expr Expr
```

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

```
data Expr = Val Int | Add Expr Expr
```

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

Diverge! La recursividad a la izquierda hace que entre en loop

Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

El parser queda entonces de la siguiente forma:

```
expr :: Parser Expr
expr = do n ← number
          pSym '+'
          e ← expr
          return (Add (Val n) e)
       <|>
       do n ← number
          return (Val n)
```

Parsing y evaluación de expresiones

$$
\begin{aligned}
evalExpr = \texttt{do}\ & e \leftarrow expr \\
& return\ (eval\ e)
\end{aligned}
$$

Parsing y evaluación de expresiones

```
evalExpr = do e ← expr
              return (eval e)
```

Es posible fusionar las definiciones de *eval* y *expr* y obtener una definición de *evalExp* que computa directamente el valor de la expresión parseada sin generar el AST intermedio:

```
evalExpr :: Parser Int
evalExpr = do n ← number
              pSym '+'
              m ← evalExpr
              return (n + m)
           <|>
           number
```

Parser de un nano XML

```
data XML = Tag Char [XML]


xml :: Parser XML
xml = do    -- se parsea el tag de apertura
          pSym '<'
          name ← item
          pSym '>'
            -- se parsea la lista de XMLs internos
          xmls ← pList xml
            -- se parsea el tag de cierre
          pSym '<'
          pSym '/'
          pSym name    -- se utiliza nombre del tag de apertura
          pSym '>'
          return (Tag name xmls)
```