# CI4251 - Programación Funcional Avanzada
# Tarea 5

Stefano De Colli
09-10203

Junio 27, 2015

## Parser

Se implentó una función de palabras a símbolos, para no tener que implementar la instancia Read.

```
stringToSym :: String → Symbol
stringToSym "true" = SymTrue
stringToSym "false" = SymFalse
stringToSym "and" = SymAnd
stringToSym "or"  = SymOr
stringToSym "xor" = SymXor
```

Luego, el cada expresión está separada por **;** y dejamos al otro combinador que se encarge de que es una expresión.

```
expresiones :: Parser [[Symbol]]
expresiones = do
            r ← expresion `endBy1` char ';'
            return $ map (map stringToSym) r
```

Las expresiones y pueden estar separadas por espacios y final de linea, así que necesitamos un parser que ignore eso.

```
sep :: String
sep = " λnλr"

garbage1 :: Parser ()
garbage1 = skipMany1 $ oneOf sep

garbage :: Parser ()
garbage = skipMany $ oneOf sep
```

Para parsear los operadores no se necesita hacer backtracking, así que el operador de opción de Parsec es suficiente.

```
value :: Parser String
value = string "true" <|>string "false"

operator :: Parser String
operator = string "and" <|>string "or" <|>string "xor"
```

Cada expresión puede tener basura al inicio, luego de la basura debería venir un valor True o False y por último cualquier cantidad de subexpresiones.

```
expresion :: Parser [String]
expresion = do
         garbage
         val ← value
         garbage
         comb ← many subExp
         garbage
         return $ val:(concat comb)
```

Cada expresión puede tener basura al inicio, luego de la basura debería venir un valor True o False y por último cualquier cantidad de subexpresiones.

```
subExp :: Parser [String]
subExp = do
         op ← operator
         garbage1
         val ← value
         garbage
         return [op, val]
```

## Algoritmo dinámico

En memoria se guarda en memoria un arreglo de caracteres, por lo tanto debemos mappear lo que recibamos a caracteres.

```
type Arr = UArray Int Char

symToChar :: Symbol → Char
symToChar SymTrue = 't'
symToChar SymFalse = 'f'
symToChar SymAnd = '&'
symToChar SymOr  = '|'
symToChar SymXor = '^'
```

El algorimto separa los símbolos en valores booleanos y operadores, así que se hizo una función que separa las listas, y dado que siempre las listas son de tamaño impar, entonces los operadores siempre terminarán en la primera posición de la tupla.

```
splitList :: [a] → ([a], [a])
splitList = foldl' (λ(a1, a2) b → (a2, b:a1)) ([], [])
```

Primero se mapean los Symbol a caracteres, y luego se separan con splitList, y se construyen los arreglos contiguos en memoria partiendo de las listas de operadores y valores. Finalmente se ejecuta el algoritmo dinámico. La solución esta en la posición (0, n), que significa de cuantas maneras se puede parentizar desde el principio hasta el final de la lista.

```
trueWays :: [Symbol] → Int
trueWays symbols = solve ! (0, n)
  where
    wsym = map symToChar symbols
    (op, sym) = splitList wsym
    n = length sym - 1
    sym' = array (0, n) $ [(i, x) |(i, x) ←zip [0..] (reverse sym)]
      :: Arr
    op' = array (0, (n - 1)) $ [(i, x) |(i, x) ←zip [0..] (reverse op)]
      :: Arr
    solve = solver sym' op' n
```

El algoritmo dinámico es básicamente una traducción del algoritmo iterativo con algunas diferencias. Primero se crean los arrego **T** y **F** y se proceden a llenar las diagonales con los valores bases. Luego en la iteración mas interna se leen todos los valores necesarios y se reescriben siguiendo el algoritmo iterativo.

```
solver :: Arr → Arr → Int → UArray (Int, Int) Int
solver sym op n = runSTUArray $ do
     t' ← newArray ((0, 0), (n, n)) 0 :: ST s (STUArray s (Int, Int) Int)
     f' ← newArray ((0, 0), (n, n)) 0 :: ST s (STUArray s (Int, Int) Int)
     forM_ [0..n] $ λi → do
                       let n = sym ! i
                       writeArray t' (i, i) $ if n = = 't' then 1 else 0
                       writeArray f' (i, i) $ if n = = 'f' then 1 else 0
     forM_ [1..n] $ λgap → do
       forM_ [gap..n] $ λj → do
         let i = j - gap
         forM_ [0..(gap - 1)] $ λg → do
           let k = i +g
           tt ← readArray t' (i, j)
           ff ← readArray f' (i, j)
           ttik ←readArray t' (i, k)
           ttkj ←readArray t' ((k +1), j)
           ffik ←readArray f' (i, k)
           ffkj ←readArray f' ((k +1), j)
           let cop = op ! k
               tik = ttik +ffik
               tkj = ttkj +ffkj

           if cop = = '&' then
             do
               writeArray t' (i, j) (tt +ttik ∗ttkj)
               writeArray f' (i, j) (ff +tik ∗tkj - ttik ∗ttkj)
           else
             if cop = = '|' then
               do
                 writeArray t' (i, j) (tt +tik ∗tkj - ffik∗ffkj)
                 writeArray f' (i, j) (ff +ffik ∗ffkj)
             else
               do
                 writeArray t' (i, j) (tt +ffik ∗ttkj +ttik ∗ffkj)
                 writeArray f' (i, j) (ff +ttik ∗ttkj +ffik ∗ffkj)
     return t'
```

## Arbitrary

Para no tener que escribir una instancia Random de Symbol, usaremos enteros y luego los mappeamos a Symbol.

```
intToSym :: Int → Symbol
intToSym 0 = SymTrue
intToSym 1 = SymFalse
intToSym 2 = SymAnd
intToSym 3 = SymOr
intToSym 4 = SymXor
```

Para la instancia de Arbitrary, creamos un nuevo tipo ya que arbitrary de por si instancia **[a]** y eso entra en conflicto con [Symbol].

```
newtype Expression = Expression [Symbol]
```

Para la instancia como tal, para un caso de tamaño **n** se generan **n + 1** valores booleanos y **n** operadores. Luego se intercalan para generar una instancia válida.

```
instance Arbitrary Expression where
arbitrary = sized $ λn → do
             values ←vectorOf (n +1) $ choose (0, 1)
             ops ← vectorOf n $ choose (2, 4)
             let mvalues = map intToSym values
                 mops = map intToSym ops
             return $ Expression $ concat $ transpose [mvalues, mops]
```