

Laboratori DAT / 6

Índex

1	Introducció	2
2	Analitzadors	2
2.1	Caràcters	2
2.2	Enters	3
2.3	Com construir analitzadors ?	3
2.4	Més exercicis	4
3	Alternativa	4
4	Analitzar consecutivament	5
4.1	Espais	6
5	Analitzar Segments	6
5.1	Bits	6
5.2	Eliminar caràcters	6
5.3	Tipus de segment	7
5.4	Capçalera	7
5.5	Dades	8
5.6	Segments	8
5.7	Decodificació	8

1 Introducció

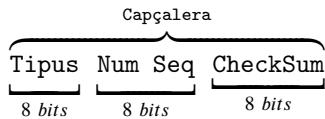
En aquesta pràctica es proposa un projecte on apareixen functors i aplicatius.

Un grup d'estudiants va fer un programa que convertia una llista dels segments TCP rebuts a la pràctica anterior a `String`. Malauradament, aquest programa no acaba d'anar del tot bé, i introduceix espais de manera aleatòria.

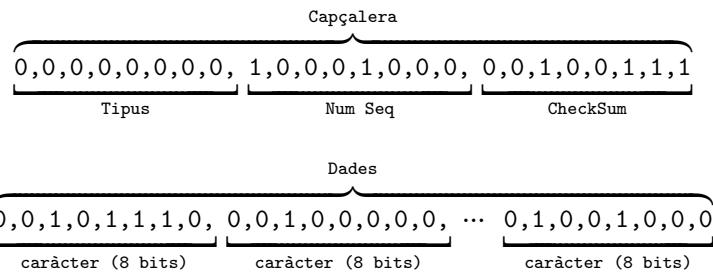
Recordar que el format dels segments era simplificat per poder fer més abordable el seu tractament. Concretament:



on



És a dir,



Un exemple d'`String` que dóna el programa que han fet els estudiants és:

```
< ACK 68 00000000 > { } < PSH 144 00010110 >
{ 0 11 011 100110001101 110100 01101 111 } < SYN 82 00000000
> { } < PSH 136 01001101 > { 1 0 1 0000 100 10 010 10010
01010 1001100 }
```

Es demana fer un programa que analitzi un `String` com l'anterior, el converteixi en una llista de `TCPSegments` (amb les definicions de tipus de la pràctica anterior) i n'extregui el missatge original.

2 Analitzadors

Un analitzador és un algorisme que té com a entrada dades no estructurades (per exemple un `String`) i dóna com a resultat dades estructurades (per exemple, tipus de dades del llenguatge o definits pel programador).

Per exemple, quan es carrega un fitxer Haskell a `ghci`, el primer que fa és analitzar el fitxer per convertir-lo d'un `String` a un tipus de dades `arbre` que representa el codi en una forma més estructurada.

Per nosaltres, un analitzador per a un valor de tipus `a` és una funció que té com a paràmetre un `String` que representa l'entrada a analitzar, i té èxit o fracassa: si té èxit, retorna el valor analitzat juntament amb la part de l'entrada que no ha utilitzat.

```
newtype Analitzador a = Analitzador {execAnalitzador :: String -> Maybe (a, String)}
```

2.1 Caràcters

Com a exemple anem a veure un analitzador pel tipus `Char`.

La funció `complirA` té com a parametre una funció que representa un predicat sobre un `Char` i té com a resultat un analitzador. Aquest analitzador:

- té èxit només si veu un Char que compleix el predicat (que alhora és el Char retornat per l'analitzador).
- si troba un caràcter que no compleix el predicat (o l'entrada és buida) llavors falla i la sortida és Nothing.

```
complirA :: (Char -> Bool) -> Analitzador Char
complirA predicat = Analitzador f
  where
    f [] = Nothing      -- falla si l'entrada es buida
    f (x:xs)
      -- Comprova si x compleix el predicat
      | predicat x       = Just (x, xs) -- si es que si
        -- retorna x i la resta de
        -- l'entrada (es a dir, xs)

      | otherwise = Nothing -- altrament, falla
```

Migançant complirA, es pot definir un analitzador que espera trobar exactament el caràcter que es passa a l'entrada i si no falla.

```
caracterA :: Char -> Analitzador Char
caracterA c = complirA (== c)
```

Exercici: Entendre la sortida de:

```
> execAnalitzador (complirA isUpper) "ABC"
> execAnalitzador (complirA isUpper) "abc"
> execAnalitzador (caracterA 'x') "xyz"
```

2.2 Enters

Observeu el següent analitzador. Si l'entrada comença per caracters numèrics els retorna tots fins que troba un caràcter no numèric.

```
enterPosA :: Analitzador Int
enterPosA = Analitzador f
  where
    f xs
      | null enters  = Nothing
      | otherwise = Just (read enters, elQueQueda)
        where (enters, elQueQueda) = primerTram isDigit xs
```

Exercici: Provar l'analitzador per diferents entrades, per exemple:

```
>execAnalitzador enterPosA "234abc"
```

2.3 Com construir analitzadors ?

Implementar analitzadors de manera explícita és llarg i és fàcil cometre errors per a qualsevol realització que vagi més enllà dels analitzadors primitius més bàsics. La idea és aconseguir tenir la capacitat de crear analitzadors complexos a partir de la combinació d'altre de més simples. Aquesta capacitat ens la donarà la classe Applicative.

Exercici:

Implementar una instància de Functor per Analitzador.

Exercici:

Implementar una instància d'Applicative per Analitzador.

1. pure a representa l'analitzador que no consumeix cap entrada i retorna amb èxit un resultat a.

2. $f1 \text{ } <*> \text{ } f2$ representa l'analitzador que executa primer $f1$ (que consumeix part de l'entrada i produeix una funció), després passa la part de l'entrada que no ha consumit a $f2$ (que consumeix més entrada i produeix algun valor), i finalment retorna el resultat d'aplicar la funció al valor.

Si $f1$ o $f2$ fallen, tot l'analitzador també falla (dit d'una altra manera, $f1 \text{ } <*> \text{ } f2$ només té èxit si tant $f1$ com $f2$ tenen èxit).

Exemple:

```
data Producte = P {codi :: Char, quantitat :: Int}
deriving Show

analitzarProducte :: Analitzador Producte
analitzarProducte = P <$> complirA isUpper <*> enterPosA

ghci> execAnalitzador analitzarProducte "F345"
Just (P {codi = 'F', quantitat = 345}, "")

ghci> execAnalitzador analitzarProducte "345F"
Nothing
```

2.4 Més exercicis

Els següents exercicis s'ha de realitzar tenint en compte que `Analitzador` és una instància d'`Applicative`.

- Realitzar un analitzador $abA :: \text{Analitzador} (\text{Char}, \text{Char})$ que espera trobar els caràcters ' a ' i ' b ' i els retorna com una tupla.
- Realitzar un analitzador $abcA :: \text{Analitzador} (\text{Char}, \text{Char}, \text{Char})$ que espera trobar els caràcters ' a ', ' b ' i ' c ' i els retorna com una tupla de tres elements.
- Realitzar un analitzador $abcLlistaA :: \text{Analitzador} [\text{Char}]$ que espera trobar els caràcters ' a ', ' b ' i ' c ' i els retorna com una llista.

Exemples de funcionament:

```
ghci> execAnalitzador abA "abcde"
Just ((‘a’, ‘b’), “cde”)

ghci> execAnalitzador abA "acde"
Nothing

ghci> execAnalitzador abcA "abcde"
Just ((‘a’, ‘b’, ‘c’), “de”)

ghci> execAnalitzador abcLlistaA "abcde"
Just (“abc”, “de”)
```

- Realitzar un analitzador $espaiEntreEspai :: \text{Analitzador} \text{Int}$ que espera trobar un espai, un enter positiu i un espai i retorna l'enter sense els espais.

3 Alternative

Com s'ha vist `Applicative` es pot utilitzar per fer analitzadors per a formats senzills i fixos. Però que passa quan un format implica triar (per exemple, “paraules que comencen per vocal”)? En aquest cas `Applicative` no és suficient. Per poder triar existeix la classe `Alternative` que està definida com,

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

Tenim que:

- `empty`: és l'analitzador que sempre falla i per tant sempre retorna `Nothing`.
- `f1 <|> f2`: és un analitzador que primer executa `f1`. Si `f1` té èxit llavors ignora `f2` i el resultat és el resultat de `f1`. Però si `f1` falla, llavors executa `f2` i aquest serà el resultat.

Exercici:

Implementar una instància d'`Alternative` per `Analitzador`.

Exercici:

Fer un analitzador `aeiouA :: Analitzador Char` que el primer que espera trobar és una vocal.

Exercici:

Fer un analitzador `espNum :: Analitzador ()` que el primer que espera trobar és un espai o un enter positiu. Observar el tipus de la funció.

Segurament us serà útil la següent funció:

```
(<$) :: a -> f b -> f a
Replace all locations in the input with the same value.
The default definition is fmap . const.
```

Exemple:

```
ghci> 'a' <$ (Just True)
Just 'a'
```

4 Analitzar consecutivament

Un alguns dels exercicis anteriors, s'ha proposat analitzar de manera repetida, i s'ha vist com la solució és poc elegant i no generalitzable. Per això es propone implementar dues funcions:

```
cap0mes :: Analitzador a -> Analitzador [a]
una0mes :: Analitzador a -> Analitzador [a]
```

1. `cap0mes`: que té com a entrada un analitzador i l'executa de manera consecutiva tantes vegades com sigui possible (que podria ser cap, si falla el primer cop), retornant una llista dels resultats. Observar que `cap0mes` zero o més sempre té èxit.
2. `una0mes`: és similar, excepte que requereix que l'analitzador d'entrada tingui èxit almenys una vegada. Si l'analitzador d'entrada falla immediatament `una0mes` també falla.

Exemples d'ús:

```
ghci> execAnalitzador (cap0mes enterPosA) "123abc"
Just ([123], "abc")
```

```
ghci> execAnalitzador (una0mes enterPosA) "123abc"
Just ([123], "abc")
```

Observar la diferència en el següent cas:

```
ghci> execAnalitzador (una0mes enterPosA) "abc123"
-- Nothing
ghci> execAnalitzador (cap0mes enterPosA) "abc123"
Just ([], "abc123")
```

Observacions:

- Per analitzar una o més execucions d'un analitzador, executar l'analitzador una vegada i després analitzar cap o més ocurrències de l'analitzador.
- Per analitzar cap o més execucions d'un analitzador, executar l'analitzador una vegada o més i si falla retornar la llista buida.

4.1 Espais

Serà molt útil disposar d'una funció `espaisA` que analitza un seguit de cap o més espais en blanc. Per la seva implementació pot utilitzar:

```
isSpace :: Char -> Bool
```

Exemples:

```
ghci> execAnalitzador espaisA "      abc"
Just ("      ","abc")
```

```
ghci> execAnalitzador espaisA "abc      "
Just ("","","abc      ")
```

Exercici: Implementar la funció

```
espaisA :: Analitzador String
```

5 Analitzar Segments

Finalment, podem començar a analitzar segments amb el format que s'ha vist a la Secció 1:

```
< ACK 68 00000000 > { } < PSH 144 00010110 > { }
{ 0 11 011 100110001101 > 110100 01101 111 } < SYN 82 00000000 > { }
< PSH 136 01001101 > { 1 0 1 0000 100 10 010 10010 01010 1001100 } }
```

5.1 Bits

Exercici: Implementar les següents funcions:

```
bitCharA :: Analitzador Char
bitCharA = undefined
```

```
bitA :: Analitzador Int
bitA = undefined
```

```
bitsA :: Analitzador [Int]
bitsA = undefined
```

1. `bitCharA`: analitza un caràcter i té èxit si es '0' o '1'. Altrament falla.

Exemples:

```
ghci> execAnalitzador bitCharA "1"
Just ('1',"")
```

```
ghci> execAnalitzador bitCharA "10"
Just ('1',"0")
```

```
ghci> execAnalitzador bitCharA " 1"
Nothing
```

2. `bitA`: Igual que `bitCharA` però en cas d'èxit el resultat de l'analitzador és un enter.

3. `bitsA`: L'analitzador té èxit si l'entrada és un `String` que comença per un seguit de '0's i/o '1's.

5.2 Eliminar caràcters

Segurament també serà molt útil una funció que donat un caràcter té com a resultat un analitzador que *elimina* espais abans i després del caràcter a l'entrada.

Exercici: Implementar la següent funció:

```
espCaracteResp :: Char -> Analitzador Char
```

Exemples:

```
ghci> execAnalitzador (espCaracteResp '*') "" * "
```

```
Just ('*', "")
```

Però observar que,

```
ghci> execAnalitzador (espCaracteResp '*') "" *a "
```

```
Just ('*', "a")
```

Observacions:

Per analitzar alguna *cosa* i ignorar el seu resultat es poden fer servir les funcions ($*>$) i ($<*$) :

```
(*>) :: Applicative f => f a -> f b -> f b
```

```
(<*) :: Applicative f => f a -> f b -> f a
```

Tenim que:

- $f1 *> f2$ executa primer $f1$ i llavors $f2$ però ignora el resultat de $f1$, i dóna com a resultat el resultat de $f2$.
- $f1 <* f2$ també executa primer $f1$, llavors $f2$ però dóna com a resultat el resultat de $f1$ (i ignora el resultat de $f2$)

Exemples:

```
ghci> execAnalitzador (caracterA 'a' *> enterPosA) "a123bc"
```

```
Just (123, "bc")
```

```
ghci> execAnalitzador (caracterA 'a' *> enterPosA) "b123cd"
```

```
Nothing
```

```
ghci> execAnalitzador (caracterA 'a' *> enterPosA) "123abc"
```

```
Nothing
```

```
ghci> execAnalitzador (caracterA 'a' <* enterPosA) "a123abc"
```

```
Just ('a', "abc")
```

```
ghci> execAnalitzador (caracterA 'a' <* enterPosA) "b123cd"
```

```
Nothing
```

5.3 Tipus de segment

Exercici: Implementar la següent funció:

```
tipusSegA :: Analitzador TipusSeg
```

que implementa un analitzador reconeix els Strings: "PSH", "ACK", "SYN" i "FIN".

5.4 Capçalera

El format de les possibles capçaleres a analitzar és:

```
espais '<' espais TipusSeg espais NumSeq espais CheckSum espais '>' espais
```

L'analitzador

```
capçaleraA :: Analitzador Capçalera
```

té èxit si es troba un string amb aquest format.

Exemple:

```
capçaleraACK456 = " < ACK 456 01010101 > "
```

```
ghci> execAnalitzador capçaleraA capçaleraACK456
```

```
Just (Cap ACK 456 [0,1,0,1,0,1,0,1], "")
```

5.5 Dades

El format de les possibles dades a analitzar és:

```
espais '{' espais 01101 espais 01101 espais 01101 espais '}' espais
```

L'analitzador

```
dadesA :: Analitzador Dades
```

té èxit si es troba un string amb aquest format.

Exemple:

```
dades01 = " { 101010010101 01010101001 000 111 0 0 0 1 1 1 } "
```

```
ghci> execAnalitzador dadesA dades01
Just (Dades [1,0,1,0,1,0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,1,0,0,0,1,1,1,0,0,0,1,1,1], "")
```

5.6 Segments

Implementar les funcions:

```
segmentA :: Analitzador TCPSegment
```

```
segmentsA :: Analitzador [TCPSegment]
```

La funció `segmentA` analitza un segment que té un format com el següent:

```
segmentPSH123 = " < PSH 123 01010101 > { 011001100 110100 101100 001 } "
```

Per altra banda la funció `segmentsA` intenta reconèixer strings que representen cap o més segments com l'anterior.

5.7 Decodificació

Un cop es disposa de la funció `segmentsA` es poden utilitzar les funcions de la pràctica anterior per decodificar l'string que es proporciona com a entrada en el fitxer .hs adjunt.