# Trabajo Final - Análisis de Lenguajes de Programación
# Web Scraping EDSL - Scrapell

Joaquín Manuel

## Descripción

### Problemática:

Gran parte de la información hoy en día se comparte mediante internet a través de las páginas Web. El inconveniente de esta situación es que la información en las páginas Web está dispuesta de manera especial para su visualización pero no se encuentra estructurada. En el caso de que se requiera almacenar esa información en una base de datos se necesitaría de una persona que analice la información, la extraiga y le de una estructura para poder almacenarla en la base de datos.

### Solución:

Web Scraping es el proceso de recopilar información de forma automática de la Web, enfocandose en la transformación de datos sin estructura (como el HTML) en datos estructurados que pueden ser almacenados y analizados en una base de datos. Ofrece técnicas de recopilación de información que permiten automatizar en parte el trabajo de una persona que examina manualmente páginas para obtener los datos deseados, haciendo de este trabajo un proceso más rápido y menos tedioso.

### Trabajo:

En este trabajo se intenta diseñar un Lenguaje de Dominio Específico embebido (EDSL) en Haskell destinado a realizar Web Scraping, que permita mediante operaciones lo más sencillas posibles, obtener datos concretos de páginas web estáticas para luego poder estructurarlos y utilizarlos.

## Instalación

El EDSL se puede instalar usando cabal-install. Se puede instalar localmente entrando a la carpeta del proyecto y usando el siguiente comando:

```
$ cabal install
```

O bien en una sandbox: (recomendado)

```
$ cabal sanbox init
$ cabal install
```

Luego para ejecutar los ejemplos se puede iniciar un intérprete limitado a la sandbox:

```
$ cabal repl

*Scrapell> :l Examples/Exaples.hs
*Examples> wiki
*Examples> imbd
```

# Manual de Uso

Al tratarse de un lenguaje embebido con solo importar el modulo "Scrapell" ya se pueden utilizar todas sus funciones.

## Ejemplo:

A continuación se explica el funcionamiento del ejemplo del archivo "Examples/Examples.hs" en el cual se busca para cada pelicula que se encuentra en una pagina de las top 10 mejores peliculas los actores que actuaron en ella y sus roles.

---

```
data Movie = Movie { mName::String, mActors::[(String,String)]} deriving Show

top10 = "https://www.imdb.com/list/ls003992425"
top10_offline = "Pages/imbd.html"

imbd :: IO [Movie]
imbd = do content <- openItem top10
          let Just trees = scrapAllForest (strToTree content) $ node $ "div" &
          ↪  ["class"==="lister-list"] --> "h3" & ["class"==="lister-item-header"]
              Just links = scrapAllForest trees $ attr "href" "a"
          content2 <- openItems (addDomain top10 links)
          let Just trees2 = scrapAllForest (strToTree content2) $ node "html"
          return $ applyToEveryTree movie trees2

movie :: Tree -> (Maybe Movie)
movie t = do name <- scrapFst t $ text ("div"&["class" === "title_wrapper"] --> "h1")
             nameA <- scrapAll t $ text ("div" & ["id" === "titleCast"] --> "td"&[notP
             ↪  (hasAttr "class")] --> "a" & [hasAttr "href"])
             role <- scrapAll t $ text ("div" & ["id" === "titleCast"] --> "td"&["class"
             ↪  === "character"] --> "a" & [hasAttr "href"])
             return $ Movie name (zip nameA role)
```

---

Creamos el tipo de dato que representará una película:

```
data Movie = Movie { mName::String, mActors::[(String,String)]}
```

Obtenemos todo el texto de la página en forma de string:

```
content <- openItem top10
```

Parseamos el html a un arbol para poder buscar en él:

```
strToTree content
```

Luego obtenemos la lista de nodos los cuales tienen forma de <h3 class = lister-item-header> y que tengan algún antecesor de la forma < div class = lister-list >, en estos nodos se encuentran el título y el link de cada película :

```
Just trees = scrapAllForest (strToTree content) $ node $ "div" & ["class"==="lister-list"]
↪  --> "h3" & ["class"==="lister-item-header"]
```

De cada uno obtenemos el link de cada película que se encuentra como valor del atributo href (<a href = link-de-pelicula >), ahora tenemos una lista de links en los cuales buscar información:

```
Just links = scrapAllForest trees $ attr "href" "a"
```

Descargamos cada página, extraemos su contenido y obtenemos una lista de árboles, en la cual cada árbol representa el html de una página:

```
content2 <- openItems (addDomain top10 links)
let Just trees2 = scrapAllForest (strToTree content2) $ node "html"
```

Luego a cada árbol se le aplica la función que dada una página obtiene el nombre de pelicula, actores y roles, retornando así una lista de películas:

```
return $ applyToEveryTree movie trees2
```

Buscamos el primer nodo que tienen forma de <h1 ... > y que tengan algún antecesor de la forma < div class = title_wrapper >, en este nodo se encuetra el nombre de la película con lo cual extraemos el texto de él:

```
name <- scrapFst t $ text ("div"&["class" === "title_wrapper"] --> "h1")
```

En este caso como puede haber muchos actores se buscan todos los nodos que cumplan con un cierto camino en el árbol, luego se extrae el texto de cada uno y se obtiene una lista de nombres. Lo mismo para los roles.

```
nameA <- scrapAll t $ text ("div" & ["id" === "titleCast"] --> "td"&[notP (hasAttr
↪  "class")] --> "a" & [hasAttr "href"])
role <- scrapAll t $ text ("div" & ["id" === "titleCast"] --> "td"&["class" ===
↪  "character"] --> "a" & [hasAttr "href"])
```

Se crea la película y se retorna:

```
return $ Movie name (zip nameA role)
```

### Aclaración:

La estructura de las páginas en los ejemplos pueden ser modificadas en cualquier momento generando errores en los códigos de ejemplos. Por ese motivo hay una copia de cada página en la carpeta "Exaples/-Pages", las cuales aseguran que los códigos de ejemplos funcionen correctamente.

## Organización del código

El código del EDSL se encuentra en la carpeta src. Se explica a continuación el contenido de cada módulo.

- **Types.hs** Aquí se definen todos los tipos de datos que se utilizaran como Attr, Selector, Selection, Scraper entre otros, y algunas clases que permitirán representar los tipos anteriores en forma de strings.
- **Attributes.hs** Funciones para el manejo de los atributos y para la creación de predicados sobre atributos.
- **Download.hs** Funciones para extraer el texto de un archivo o de páginas web que son descargadas.
- **Examples.hs** Contiene 4 ejemplos de uso del EDSL.
- **Scraper.hs** Funciones de búsqueda y extracción que trabajan con Scrapers y con árboles.
- **Selection.hs** Funciones para la creación de selecciones.
- **Extra.hs** Funciones últiles para modificar los datos luego de haberlos obtenido.
- **Tags.hs** Funciones que se aplican a Tags o listas de Tags.

# Diseño

El EDSL se creó en base a TagSoup, una librería de Haskell que permite principalmente parsear el html de una página a una lista de Tags, pero también permite parsear el html a un árbol el cual es la principal estructura utilizada en este EDSL.

```
data TagTree str
  = -- | A 'TagOpen'/'TagClose' pair with the 'Tag' values in between.
    TagBranch str [Attribute str] [TagTree str]
  | -- | Any leaf node
    TagLeaf (Tag str)
                deriving (Eq,Ord,Show)
```

Lo más importante para el diseño del lenguaje es el diseño de funciones de búsqueda y extracción sobre árboles (los cuales representan las páginas web) y el diseño de expresiones regulares para poder clasificar e identificar nodos.

## Predicados para Atributos:

Una etiqueta en html consiste de un nombre como por ejemplo "div" y varios atributos como por ejemplo "class" o "id" los cuales tienen valores. Por lo tanto un atributo se representa como una tupla (String,String). Para los predicados sobre atributos se creó un tipo de dato que contiene una función que toma una lista de atributos y retorna un booleano, ya que dada una etiqueta y un predicado para un solo atributo, con que se cumpla para una tupla es suficiente.

```
data AttrPredicate = AttrPredicate {runAttrPredicate :: ([Attr] -> Bool)}
```

Para poder crear predicados fácilmente se tienen las siguientes funciones:

- **fun** Permite dada una función que tome nombre y valor de un atributo y devuelva un booleano, crear un predicado que valga si esa función vale para algún atributo de la etiqueta.

  ```
  fun :: (String -> String -> Bool) -> AttrPredicate
  ```

- **notP** Crea el predicado que es resultado de negar el predicado pasado como parámetro.

  ```
  notP :: AttrPredicate -> AttrPredicate
  ```

- **startsWith** Verifica que el valor de un atributo determinado comience con un prefijo dado.

  ```
  startsWith :: (AttrRep a) => a -> String -> AttrPredicate
  ```

- **hasAttr** Verifica que la etiqueta tenga un atributo determinado.

  ```
  hasAttr :: (AttrRep a) => a -> AttrPredicate
  ```

- **hasString** Verifica que el valor de un atributo determinado tenga como substring a la string pasada como parámetro.

  ```
  hasString :: (AttrRep a) => a -> String -> AttrPredicate
  ```

- **===** Este operador verifica que la etiqueta/nodo tenga un atributo con un determinado nombre y valor.

```
(===) :: (AttrRep a) => a -> String -> AttrPredicate
```

El nombre de un atributo puede ser del tipo AttrName o puede ser una String, donde la "" representa que el atributo puede tener cualquier nombre.

## Selecciones:

Los **selectores** tienen toda la información para decidir si seleccionar un nodo/etiqueta o no, esta información se guarda en el tipo de dato:

```
data SelectNode = SelectNode TagName [AttrPredicate]
                | SelectText
                | SelectAny

type Selection = [SelectNode]
```

Necesitan el nombre de la etiqueta, y una lista de predicados sobre atributos (los cuales son creados como vimos anteriormente). Los selectores se crean con el operador (&) que toma un nombre de etiqueta el cual puede ser "" representando cualquier nombre y una lista de predicados para atributos. En el caso que no se requieran predicados para identificar el nodo con el nombre de la etiqueta es suficiente para crear el selector.

```
(&) :: (TagNameRep t) => t -> [AttrPredicate] -> Selection
```

Se crearían de la siguiente manera:

```
nombre_tag & [predicado_1 , predicado_2, ... ]

-- Ejemplo:
"div" & ["class" === "producto", "id" `startsWith` "prod"]
```

Una **selección** es una lista de **selectores** y representa el camino de nodos que tienen que ser seleccionados para llegar al nodo final en el cual estamos interesados. Para concatenar selecciones se utiliza el operador $(--)$ :

```
"div" & ["class" === "row itemList"] --> "div" & [hasAttr "data-product-id"] --> "a"
```

En este ejemplo estamos interesados en el nodo que tiene nombre de etiqueta "a", pero no en cualquiera sino en los que se encuentran al final del camino especificado.

## Scraper:

Un **Scraper** tiene la información de como encontrar el nodo (una selección) y de qué hacer con él luego que se encontró.

```
data Scraper a = Scraper {sSelection :: Selection , sToDo :: (Tree -> a)}
```

Para crearlos también existen una serie de funciones:

- **text** Dada una selección retorna un scraper que obtiene el texto del nodo final.

  ```
  text :: SelectionRep s => s -> Scraper (Maybe String)
  ```

- **textPred** Igual al anterior, solo que retorna el texto sólo si verifica un predicado pasado como parámetro.

```
textPred :: SelectionRep s => (String -> Bool) -> s -> Scraper (Maybe String)
```

- **attr** Retorna el valor de un atributo del nodo final.

```
attr :: SelectionRep s => String -> s -> Scraper (Maybe String)
```

- **node** Retorna el nodo final.

```
node :: SelectionRep s => s -> Scraper (Maybe Tree)
```

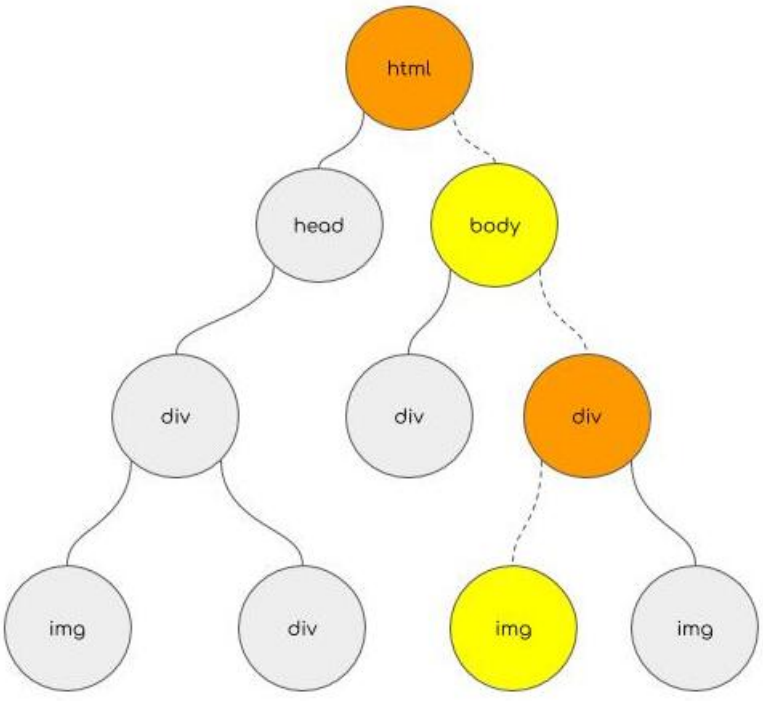
## Funciones de Búsqueda y Extracción:

Hay 3 funciones principales para este propósito:

- **scrapFst** Dado un árbol y un Scraper, busca el nodo objetivo siguiendo la ruta especificada en la selección, apenas encuentra un nodo que la cumpla aplica la función del Scraper en él y retorna su resultado. En el caso que no se encuentre el nodo o la función no obtenga ningún resultado retornará **Nothing**.

```
scrapFst :: Tree -> Scraper (Maybe a) -> Maybe a
```

En el siguiente ejemplo se marca en linea de puntos el camino que sigió la función para llegar al nodo final:

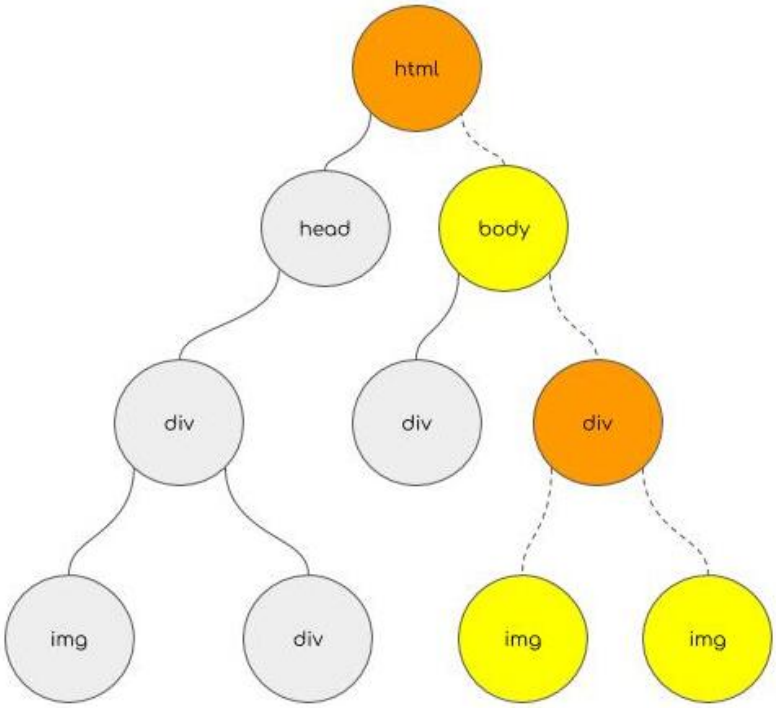



```
scrapFst arbol $ node $ "body" --> "img"
```

- **scrapAll** Realiza un procedimiento similar a la función anterior, solo que ésta retorna una lista con todos los resultados de aplicar la función del Scraper a todo nodo que cumpla con la selección.

```
scrapAll :: Tree -> Scraper (Maybe a) -> Maybe [a]
```

Aplicando al ejemplo anterior scrapAll:




```
scrapAll arbol $ node $ "body" --> "img"
```

- **scrapFstAfter / scrapAllAfter** A veces no es posible identificar un único nodo con una selección debido a que no existe información suficiente en los nombres de las etiquetas o en sus atributos como para distinguir uno de otro.

  Por ejemplo, en el siguiente caso si quisieramos obtener el número de documento no podríamos obtenerlo utilizando scrapFst, y en caso de scrapAll lo obtendríamos pero junto con todos los demás datos:

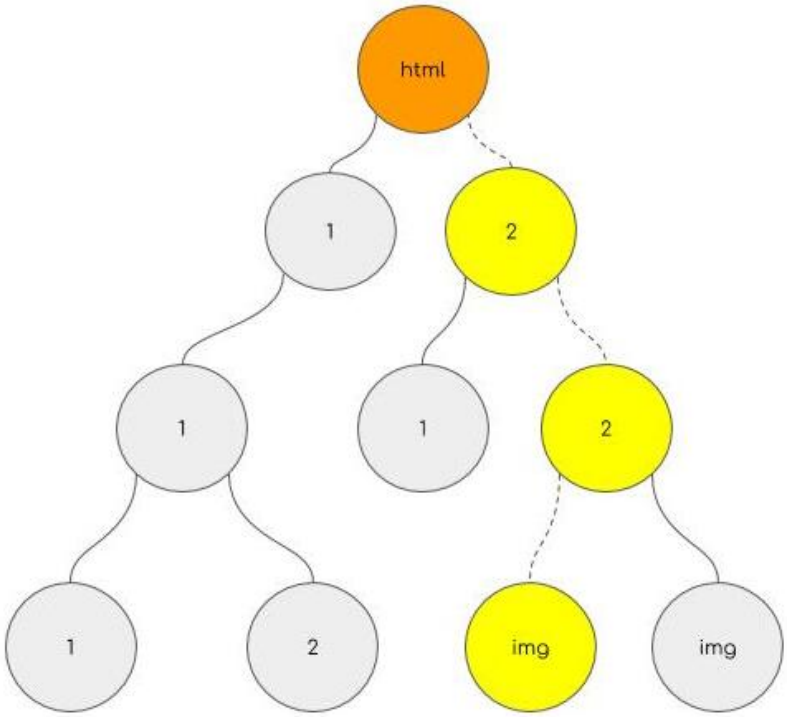
```
<tr>
    <td>
        <a > Nombre </a>
        <a > Apellido </a>
    </td>
    <td>
        <a > Calle </a>
    </td>
    <td>
        <a > Tipo Documento </a>
        <a > Número </a>
    </td>
</tr>
```

La información para llegar a un nodo específico se encuentra en la estructura del árbol, con lo cual si se especifica el camino exacto para llegar al nodo indicando paso a paso a que nodo hijo pasar no habría problema en llegar a cualquier nodo.

La función **scrapFstAfter / scrapAllAfter** toma los mismos argumentos que las anteriores pero además pide una lista de enteros que especifiquen paso a paso que nodos hijos se deben recorrer al principio de la ruta de selección, una vez que se termina de seguir esa secuencia la función se comporta igual que **scrapFst / scrapAll** aplicada en el nodo en el que se terminó la secuencia.

```
scrapFstAfter :: Tree -> [Int] -> Scraper (Maybe a) -> Maybe a
```

Aquí podemos ver un ejemplo:




```
scrapFstAfter arbol [2,2] $ attr "href" "img"
```

# Bibliografía


- Introducción al web Scraping:
  - Web Scraping with Python: Collecting Data from the Modern Web, `https://bit.ly/32kR3Y4`
  - An Overview On Web Scraping Techniques And Tools, `https://bit.ly/2SQWD0V`
- Trabajos relacionados hechos en Haskell:
  - TagSoup, una librería para parsear HTML/XML, `https://github.com/ndmitchell/tagsoup`
  - Scalpel, una librería para realizar Web Scraping, `https://github.com/fimad/scalpel`
- Otros:
  - Scraping Distributed Hierarchical Web Data, `https://bit.ly/37W2Mxr`