

# **PECL1 Procesadores del Lenguaje:**

Construcción de un analizador léxico.

Mario Ríos Muñoz  
09047759L  
15 Octubre 2014

---

## Uso del analizador:

Este analizador se ha concebido para ser usado en interfaces de líneas de comando. Para ser ejecutado, habría que emplear el comando:

```
java UpdComp (-i|ruta_al_fichero.upd).
```

En caso de optar por la opción `-i`, el analizador se ejecutaría en modo interactivo, siendo necesario pulsar la tecla 'enter' para que el analizador evaluase una cadena. Este modo fue creado para facilitar la tarea de depuración.

Para salir del modo interactivo será necesario interrumpir el programa (en Unix 'ctrl+c'), por lo que al terminar la ejecución no mostrará el contenido de la clase 'updClass'.

En caso de optar por la segunda opción, es necesario facilitar una ruta válida a un fichero válido.

Por fichero válido se entiende todo aquel que tenga una extensión '.upd' . En caso de no cumplir esta condición se mostrará un mensaje de error como el siguiente:

```
→ PECL1 git:(nuevo_concepto) ✖ java UpdComp fichero_no_valido
extension de archivo no reconocida
```

Nótese que el analizador compilado se llama 'UpdComp' y no 'Yylex' como sería por defecto. Esto se debe al uso de la directiva %class.

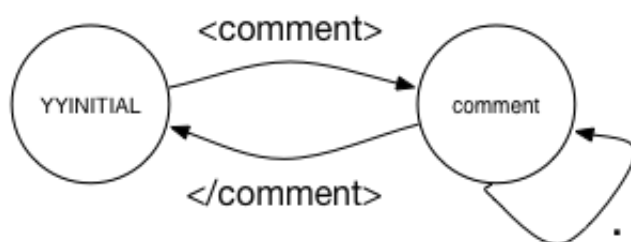
---

## Identificación de lexemas:

### *Acerca de los comentarios:*

Los comentarios son un caso especial, dado que no se consideran lexemas exactamente (no tienen significado sintáctico). Como lo único que nos interesa de los comentarios es eliminarlos a la hora de escanear, es necesario el uso de estados léxicos.

El diagrama de estados quedaría de la siguiente forma:



De esta forma, si estando en el estado inicial recibimos un token de apertura de comentario, la acción léxica es pasar a estado 'comment'.

Estando en ese estado de comentario, se ignora absolutamente todo lo que no sea una etiqueta de fin de comentario, y en caso de recibir dicha etiqueta se vuelve al estado inicial.

Nótese, que no se está teniendo en cuenta la expresión regular `[\\r\\n]` ( '.' acepta todo menos dicha expresión). Esto no supone un problema, dado que al final de toda la especificación léxica, se ha incluido una regla que ignora todos los espacios en blanco.

## ***Acerca de las palabras reservadas:***

Para esta práctica en concreto, las palabras reservadas no suponen ninguna acción léxica en particular. Simplemente tienen que ser aceptadas. Por ello, se ha optado por reconocer todas las palabras reservadas ('<ident>', '</ident>', '<date>', '</date>', ...) mediante una única expresión regular. En el caso de las etiquetas de apertura ('<name>'), la expresión que las captura es:

```
"<{marca}">"
```

donde marca designa cada uno de los posibles tipos que pueden darse (ident, date, bitSize...).

De la misma forma, para las etiquetas de cierre, la expresión:

```
"</{marca}">"
```

es la que las captura.

En ambos casos la acción léxica es nula.

Es evidente, que si este analizador tuviese que comunicarse con un pársers, para cada palabra reservada del lenguaje habría que definir una regla, con su respectiva acción léxica de devolver el tipo de token, pero como este no es el caso, se ha optado por reconocer a todas conjuntamente, para simplificar la implementación.

## ***Acerca de los identificadores:***

Partiendo de un estado inicial, se buscan los lexemas que coincidan con alguno de los identificadores requeridos (ident, fecha, nombre, uso ...). Una vez encontrado, se ejecuta la acción léxica pertinente.

En el caso del identificador ident, el comportamiento es el siguiente: se espera cualquier cadena que empiece por una letra y vaya seguido de cualquier caracter que no sea un espacio en blanco, una comilla doble (") o simple (') o e caracter '<'. El motivo de la última condición es para evitar que en el caso:

```
<ident>x86</ident>
```

se reconozca x86</ident> como identificador. La expresión regular queda de la siguiente forma:

```
{letra}[^\n\r\t \'"<']*
```

La acción léxica para ident es añadir el token identificado a la clase.

En el caso del identificador fecha, la expresión regular surgió de manera bastante directa, siendo la siguiente:

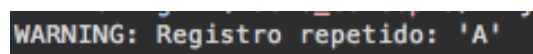
```
{dig}{dig}"/"{dig}{dig}"/"{dig}{dig}{dig}{dig}
```

La acción léxica para este token es similar a la de ident.

Para el identificador de nombre, se pueden tener una o dos letras delimitadas por comilla simple ('), por lo que la expresión resulta en:

```
"'"{letra}{letra}?"'
```

y la acción léxica consiste en introducir el token en la hashtable de la clase. En caso de que la función de inserción retorne 'false' se interpreta que ese registro ya existía y se notifica de inmediato con el siguiente mensaje:



```
WARNING: Registro repetido: 'A'
```

Para los identificadores de tipo uso, se tiene una regla léxica por cada posible uso. La acción a ejecutar es incrementar el número del uso respectivo.

En el caso del identificador de tipo bitSize, se tuvo que tener en cuenta un posible conflicto: Tanto bitSize como insBitCode tienen una parte que puede considerarse entera. Para no confundir una máscara de bits que comience con '0' con un entero, se decidió no permitir enteros que empezasen también por '0'. Por tanto la expresión regular queda como:

`[1-9][0-9]*`

En el caso de que la posible máscara de bits no empezase por '0' y sólo tuviese la parte fija (en caso de que fuese de la forma 1001001), no se consideraría como máscara de bits, sino como bitSize.

Para reconocer las máscaras de bits, se ha usado la expresión:

`[0-1]+(x+(yz*)*)*`

que obliga a una parte fija como cadena binaria y una parte opcional en la que solo hay Zs si hay Ys, y sólo hay Ys si hay Zs, aunque si se podría repetir el patrón xyzxyz

---

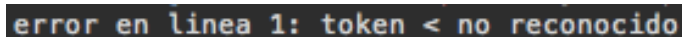
## Gestión de errores:

En caso de que ninguno de los patrones introducidos coincida con una expresión regular de la especificación léxica, se llegara a una regla final que acepta "todo lo demás".

Esta regla es de la forma:

`. {acción_lexica}`

La acción léxica que se ha elegido es imprimir un mensaje de error que informe del carácter no reconocido y la línea en la que se encuentra. Se puede apreciar a continuación:



```
error en línea 1: token < no reconocido
```

---

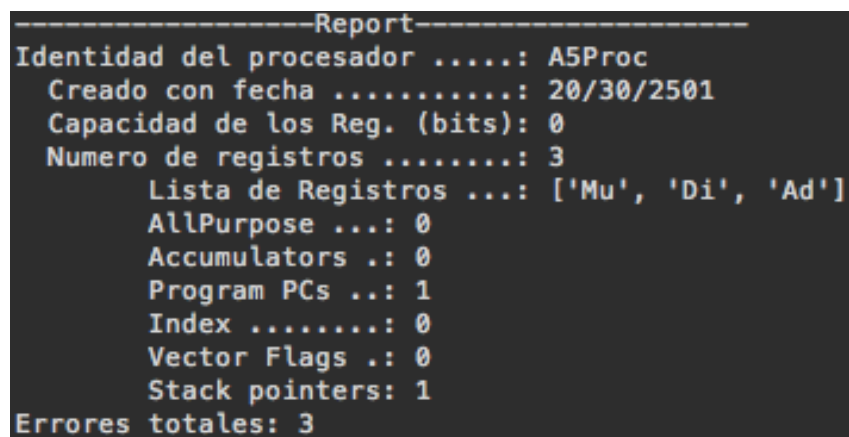
## Acciones finales:

Cuando se llega al final del fichero a analizar, hay un par de tareas que se tienen que realizar antes de dar el análisis por finalizado.

Una de ellas es realizar un volcado con la información de la clase.

La otra es imprimir un mensaje con el número de errores totales que se han encontrado.

Estas tareas están codificadas en un bloque %eof{ %eof}, y su resultado se puede apreciar en la siguiente imagen:



```
-----Report-----
Identidad del procesador .....: A5Proc
Creado con fecha .....: 20/30/2501
Capacidad de los Reg. (bits): 0
Numero de registros .....: 3
  Lista de Registros ...: ['Mu', 'Di', 'Ad']
  AllPurpose ...: 0
  Accumulators ..: 0
  Program PCs ..: 1
  Index .....: 0
  Vector Flags ..: 0
  Stack pointers: 1
Errores totales: 3
```

---

## Casos de prueba:

Además de los dos ficheros de prueba que se entregaban con la práctica, se ha construido uno adicional con más casos de prueba. Alguno de ellos son los siguientes:

Si se reciben mascararas de bits como las siguientes:

10y

10z

10xz

no se identifican como tal.

Varias fechas seguidas, o varios uses, names seguidos no causan conflicto:

20/20/202030/30/3030

'az"jk'

"ProgramPC""Accumulator"

Absolutamente todo lo que hay en un comentario es ignorado, incluso si eso incluye palabras reservadas.

Si un comentario no es cerrado, todo lo que vaya a continuación es ignorado hasta el fin del fichero, sin causar error.