# Teoría de Lenguajes

Primer cuatrimestre 2013

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

## TP Micro HTML Prettyprint



| Integrante | LU | Correo electrónico |
|---|---|---|
| Ezequiel Gutesman | 715/02 | egutesman@gmail.com |
| Mauricio Alfonso | 65/09 | mauricioalfonso88@gmail.com |
| Victor Hugo Montero | 707/98 | vico.walker@gmail.com |

# Contents

# 1 Primera Parte

## 1.1 Introducción

Cuando comenzamos a pensar en cómo encarar el problema del desarrollo de la gramática, pensamos que podríamos tratar cada caracter (letra, número, espacio, etc.) como un símbolo terminal diferente, pero terminaríamos con una cantidad inmanejable de símbolos no terminales y producciones, es decir con una gramática enorme. Por esa razón definimos una serie de Token Léxicos que al pasárselos a un analizador léxico transforman el texto de entrada de una serie de caracteres en una serie de tokens.

Definimos los tokens usando expresiones regulares. Primero definimos un token por cada tag de HTLM y por último un token para todo el texto que sobre. El analizador léxico tokeniza el archivo buscando los tokens en orden, de manera que todo el texto que halla entre dos tags queda tokenizado como `texto`.

El caso de la sección `script` de HTML nos presentó un problema: entre los tags `<script>` y `</script>` puede haber cualquier combinación de caracteres excepto el tag de cierre de script. Esto quiere decir que podría haber incluso otros tags HTML dentro de un script, que deberían ser ignorados y que podrían no formar HTML válido. Pensamos en un principio en tokenizar el texto interior de los scripts con el analizador léxico como `texto_sin_script` para tomar todo el texto que no tenga el tag `</script>` como un token. Sin embargo no es posible tokenizar el texto de esta manera, ya que no tiene sentido poner el token `texto_sin_script` antes ni después de los tags de HTML. Esto se debe a que si el analizador léxico busca primero el token `texto_sin_script` y luego los tags, el primer token estaría "absorviendo" todos los tags excepto `</script>` como parte de un script, incluso cuando no forman parte de un script; entonces no tokenizaría nunca los tags HTML. Si en cambio tokenizáramos primero los tags HTML y después `texto_sin_script`, los tags del interior de los scripts serían reconocidos y el tag `texto_sin_script` no cumpliría su función. Por esta razón decidimos tokenizar únicamente `texto` para todo el texto que sobre luego de haber reconocido los tags HTML, y reconocer los tags que pueda haber en el script desde la gramática y no desde el analizador léxico.

Además de tokenizar, el analizador léxico también se encarga de eliminar los comentarios y los espacios en blanco consecutivos, ya que estos podrían estar en cualquier parte del archivo recibido y debemos ignorarlos por completo.

En la gramática definimos un símbolo no terminal $S$ para todo el documento.

El símbolo no terminal $H$ contiene el interior del HTML, que a su vez puede tener un $HEAD$ y un $BODY$.

Los símbolos $HEAD$ y $BODY$ representan las secciones head y body de HTML respectivamente, y los símbolos $HE$ y $B$ representan el interior de dichas secciones.

Con el símbolo $SCS$ definimos una serie de cero o más scripts, que identificamos con el símbolo $SC$.

El símbolo $TSC$ representa el texto del interior de un script. Este es el símbolo con más producciones, ya que puede tener en su interior cualquier tag excepto `</script>` y además estos pueden ir de cualquier manera, sin formar HTML válido.

El interior de la sección body lo definimos recursivamente con el símbolo $B$, entendiendo que entre cada par de tags de apertura y cierre del mismo tipo puede haber más texto HTML válido.

Decidimos que sólo puede haber un elemento `<TITLE>...</TITLE>` dentro del head.

## 1.2 Tokens Léxicos

Los tokens léxicos están descritos por expresiones regulares. La siguiente tabla describe para cada token su expresión rgular correspondiente.

| *Token Léxico* | *Expresión Regular* |
|---|---|
| <HTML> | <html> |
| </HTML> | </html> |
| <HEAD> | <head> |
| </HEAD> | </head> |
| <BODY> | <body> |
| </BODY> | </body> |
| <TITLE> | <title> |
| </TITLE> | </title> |
| <SCRIPT> | <script> |
| </SCRIPT> | </script> |
| <DIV> |  |
| </DIV> |  |
| <P> |  |
| </P> |  |
| <BR> | <br> |
| texto | .* |

Antes de tokenizar, el analizador léxico elimina espacios consecutivos y comentarios. Los comentarios cumplen con la expresión regular `<!--.*-->`

### 1.2.1 Correcciones Para la implementación

Si bien la primer versión de los tokens no tenía ninguna corrección por parte del grupo docente, cambiamos la forma de tokenizar el token `texto` dadas las posibilidades que nos brindaba ANTLR[1]. La nueva versión del token quedó expresada como:

`texto [∼<]*`

Queriendo representar cualquier caracter menos el símbolo <

[1] www.antlr.org

## 1.3 Gramática

La siguiente gramática define el MicroHTML:

$$G =< N, \Sigma, P, S >$$

Donde:

$$N = \{S, H, HEAD, BODY, HE, SCS, SC, TSC, B\}$$
$$\Sigma = \{texto, < HTML >, < /HTML >, < HEAD >, < /HEAD >,$$
$$< TITLE >, < /TITLE >, < SCRIPT >, < /SCRIPT >,$$
$$< BODY >, < /BODY >, < H1 >, < /H1 >, < DIV >, < /DIV >,$$
$$< P >, < /P >, < BR >\}$$
$$S = S$$

Que de acuerdo a la clasificación de Chomsky es una gramática *tipo 0*, o *sin restricciones*. Las producciones ($P$) se detallan en la siguiente subsección.

## 1.4 Producciones - $P$

$S \rightarrow$ <HTML> $H$ </HTML>

$H \rightarrow HEAD\ BODY \mid HEAD \mid BODY \mid \lambda$

$HEAD \rightarrow$ <HEAD> $HE$ </HEAD>

$BODY \rightarrow$ <BODY> $B$ </BODY>

$HE \rightarrow SCS$ <TITLE> $texto$ </TITLE> $SCS$

$SCS \rightarrow SC\ SCS \mid \lambda$

$SC \rightarrow$ <SCRIPT> $TSC$ </SCRIPT>

$TSC \rightarrow$ <HTML> $TSC$ | </HTML> $TSC$ |
<HEAD> $TSC$ | </HEAD> $TSC$ |
<BODY> $TSC$ | </BODY> $TSC$ |
<TITLE> $TSC$ | </TITLE> $TSC$ |
<DIV> $TSC$ | </DIV> $TSC$ |
<H1> $TSC$ | </H1> $TSC$ |
<P> $TSC$ | </P> $TSC$ |
<SCRIPT> $TSC$ | <BR> $TSC$ | $texto\ TSC \mid \lambda$

$B \rightarrow texto\ B \mid$
<DIV> $B$ </DIV> $B$ |
<H1> $B$ </H1> $B$ |
<P> $B$ </P> $B$ |
<BR> $B$ |
$\lambda$

### 1.4.1 Correcciones Para la implementación

Debido a que ANTLR es un generador de parsers tipo LL(K) extendido, las producciones del símbolo no terminal H debieron ser cambiadas. En la gramática original dos de las producciones de H tienen el terminal `<HEAD>` como símbolo directriz, lo cual no está permitido en parsers LL. Las producciones de H fueron reemplazadas por una sola usando expresiones regulares de la siguiente manera:

$$H \rightarrow \textit{HEAD ? BODY ?}$$

Por otra parte, para evitar que ANTLR reconozca cadenas inválidas formadas por HTML válido seguido de otros caracteres (por ejemplo `<html></html>aaa`), fue necesario agregar el símbolo de fin de archivo (EOF en ANTLR) a la producción del símbolo distinguido S, quedando de la siguiente manera:

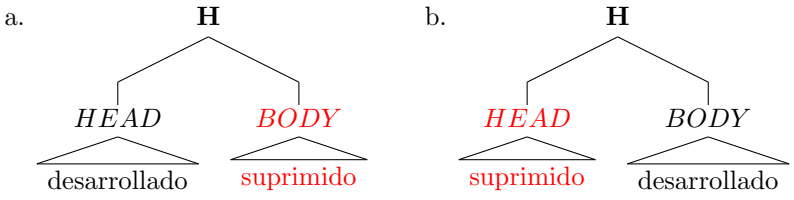
$$S \rightarrow \textit{<HTML> H </HTML> \$}$$

## 1.5 Arbol de derivación

A continuación presentamos el árbol de derivación para el siguiente ejemplo:

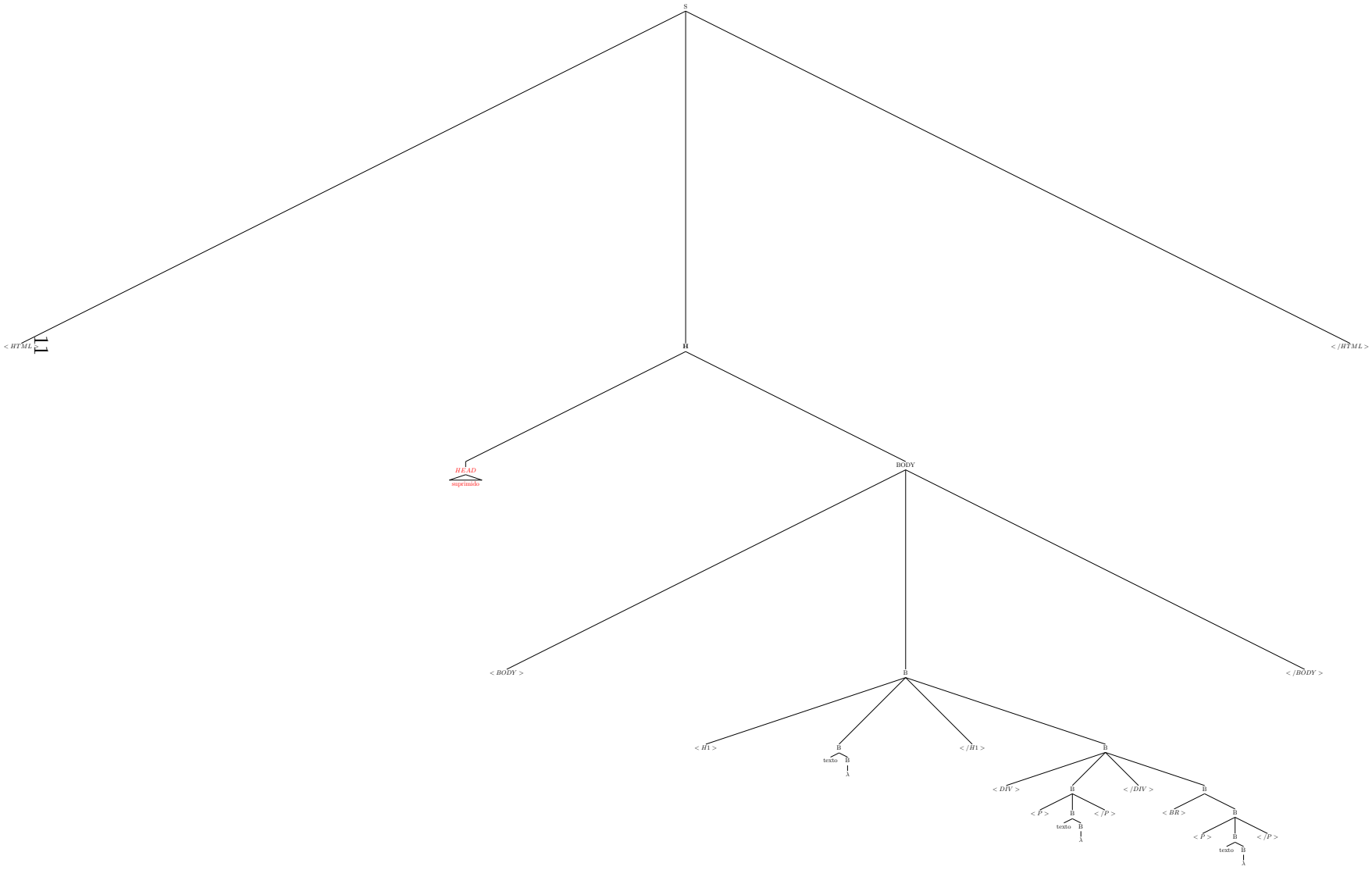
```
<HTML>
    <HEAD>
        <TITLE>Una pagina de ejemplo</TITLE>
        <SCRIPT>
        function unaFunc(){
            alert("esta funcion imprime un tag roto <TITLE>");
        }
        </SCRIPT>
        <SCRIPT></SCRIPT>
        <SCRIPT>alert("aca no aparece el cierre de SCRIPT")</SCRIPT>
    </HEAD>
    <BODY>
        <H1>Un heading</H1>
        <DIV>
            <P>Este texto es de prueba</P>
        </DIV>
        <BR>
        <P>Mas prueba</P>
    </BODY>
</HTML>
```

Debido al tamaño del ejemplo, el árbol de derivación tuvo que ser partido en dos partes. Nótese que del (único) nodo con el no terminal $H$ cuelgan 2 hijos con el no-terminal $HEAD$, y el no terminal $BODY$. El primer subárbol muestra los descendientes de $HEAD$ y el segundo los de $BODY$. En cada árbol, el no terminal cuyo subárbol está suprimido se encuentra marcado en rojo, y con un subárbol genérico (triangulo) simbolizando que en realidad continúa. Por ejemplo:




En la figura *a.* se suprime el subárbol corrspondiente al no terminal $BODY$ y en el *b.* el correspondiente al no terminal $HEAD$.

10


S
< HTML >
H
< /HTML >
HEAD
suprimido
BODY
< BODY >
B
< /BODY >
< H1 >
B
texto
B
λ
< /H1 >
B
< DIV >
B
< /DIV >
< P >
B
texto
B
λ
< /P >
B
< BR >
B
< P >
B
texto
B
λ
< /P >

# 2 Segunda Parte

## 2.1 Implementación

### 2.1.1 Descripción del problema

Se pidió en esta segunda parte implementar un parser que parsee texto según la gramática especificada en la sección 1.3. Esta implementación debería tomar como entrada una cadena de texto y en caso de poder parsearla, producir como salida un archivo en formato HTML que se pueda abrir desde un browser y que dentro de él, se encuentre el texto procesado de la entrada pero correctamente indentado y coloreado.

Por ejemplo, para la siguiente entrada:

```
<html><head><title>Título</title><script>print("hello")</script>
</head><body>texto<p>párrafo<h1><!--comentario--><p> más texto
</p></h1></p> <div>texto texto texto <br> mas texto texto texto
</div> </body> </html>
```

Se producirá la siguiente salida en un archivo HTML:

```
<html>
    <head>
        <title>Título</title>
        <script>
            print("hello")
        </script>
    </head>
    <body>
        texto
        <p>
            párrafo
            <h1>
                <p>
                    más texto
                </p>
            </h1>
        </p>
        <div>
            texto texto texto
            <br>
            mas texto texto texto
        </div>
    </body>
</html>
```

Figure 1: Salida para un ejemplo válido.

### 2.1.2 Detalles de implementación y limitaciones

La solución fue desarrollada con ANTLR[2] y tanto el lexer como el parser fueron generados en Java.

El parser no acepta el símbolo `<` dentro de un tag script. Esto podría tokenizarse mejor ya que una comparación dentro del script haría que falle el parsing.

En cuanto a la salida del parsing, esta es calculada en un atributo **sintetizado** llamado `texto`. Para manejar la indentación, utilizamos un único tag `<div class="bloque">` cuyo único estilo tiene un margen a izquierda, fijo. El efecto de ir generando estos divs a medida que se necesita generar un tag produce la indentación deseada, contemplando el nivel de encadenamiento de tags producto de ir procesando un tag dentro de otro.

Los símbolos terminales ya poseen un atributo llamado `text` al que se accede con los métodos `getText()` y `setText()`, y que modificamos reemplazando los caracteres `<` y `>` por símbolos de escape `&lt;` y `&gt;` respectivamente y rodeamos de un span para darle el color adecuado. En las producciones de los símbolos no terminales sintetizamos el atributo `texto` como la concatenación de los textos de los atributos en los que produce, junto con los tags `<div>` necesarios para que se intente y coloree el texto.

Al `texto` del símbolo distinguido `s` le agregamos también los tags `<html>`, `<head>` y `<body>` necesarios para que la salida sea un HTML válido, además de información de estilo en formato CSS.

Una modificación que debimos hacer para escribir la gramática fue pasar todos los nombres de los símbolos no terminales de mayúscula a minúscula y los de los terminales de minúscula a mayúscula. Esto se debe a que ATNLR decide si un símbolo es terminal o no terminal usando una convención inversa a la vista en clase.

Para ejecutar el programa, generamos una clase llamada `Main` que recibe por entrada estándar el HTML de entrada y genera el HTML de salida por salida estándar. Dicha clase genera un lexer usando la entrada estándar, y luego un parser usando dicho lexer. Por último ejecuta el método `s()` (correspondiente al símbolo distinguido) del parser e imprime pos salida estándar el atributo `texto`.

Un problema que tuvimos con la clase `Main` fue que si un componente interno del HTML era inválido, se reemplazaba su texto por `null`, pero el resto del archivo se imprimía normalmente. Como queremos que en caso de que la entrada no sea válida se descarte todo el HTML, decidimos sobreescribir los métodos `reportError()` del lexer y el parser. En nuestra implementación estos métodos reportan el error por `stderr` como se haría normalmente, pero además lanzan una excepción de tiempo de ejecución que interrumpe todo el programa. Modificar los métodos `reportError()` en en código de lexer y el parser traería problemas, ya que estos son autogenerados y cualquier cambio en la gramática borraría nuestra modificación; por esa razón sobreescribimos los métodos desde la clase `Main`, extendiendo el lexer y el parser.

Otro problema que tuvimos en un principio con la clase `Main` fue que ignoraba por completo los fines de línea, aceptando por ejemplo entradas donde un

[2] www.antlr.org

tag podía estar partido en varios renglones. Esto se debía a que se transformaba `stdin` en un `String`, y un bug ignoraba los fines de línea. Esto fue cambiado reemplazando el `String` por un `ANTLRInputStream` que lee directamente de la entrada.

### 2.1.3 Entradas de prueba válidas

Entrada 1:

```
<html><head><title>Título</title><script>print("hello")</script>
</head><body>texto<p>párrafo<h1><!--comentario--><p> más texto
</p></h1></p> <div>texto texto texto <br> mas texto texto texto
</div> </body> </html>
```

Salida 1:

```
<html>
    <head>
        <title>Título</title>
        <script>
            print("hello")
        </script>
    </head>
    <body>
        texto
        <p>
            párrafo
            <h1>
                <p>
                    más texto
                </p>
            </h1>
        </p>
        <div>
            texto texto texto
            <br>
            mas texto texto texto
        </div>
    </body>
</html>
```

Figure 2: Salida para un ejemplo válido.

Entrada 2:

```
<html>  <head> <script>print("a script")</script><title>Título</title>
 <script>print("hello")</script><script>print("world")</script>
</head><body>texto suelto<p>párrafo <h1><!-- comentario--><p> más texto</p></h1>
</p><div>texto texto texto <br> mas texto textotexto</div><div>
Un Div que adentro tiene otro<div>Dentro <div><p>de otro</p> con mas texto
</div></div></div>  </body>  </html>
```

Salida 2:

```
<html>
    <head>
        <script>
            print("a script")
        </script>
        <title>Título</title>
        <script>
            print("hello")
        </script>
        <script>
            print("world")
        </script>
    </head>
    <body>
        texto suelto
        <p>
            párrafo
            <h1>
                <p>
                    más texto
                </p>
            </h1>
        </p>
        <div>
            texto texto texto
            <br>
            mas texto texto texto
        </div>
        <div>
            Un Div que adentro tiene otro
            <div>
                Dentro
                <div>
                    <p>
                        de otro
                    </p>
                    con mas texto
                </div>
            </div>
        </div>
    </body>
</html>
```

Figure 3: Salida para un ejemplo válido.

### 2.1.4 Entradas de prueba inválidas

Entrada 3 (`<title>` sin abrir):

```
<html>  Título</title> <script>print("hello")</script><script>print("world")
</script></head><body>texto suelto<p>párrafo <h1><!-- comentario--><p>
más texto</p></h1></p><div>texto texto texto <br> mas texto texto texto
</div><div>Un Div que adentro tiene otro<div>Dentro <div><p>de otro
</p> con mas texto</div></div></div></body></html>
```

Salida 3: (stderr)

```
line 1:6 missing TK_C_HTML at ' Título' <html>
```

Entrada 4 (`<div>` sin cerrar):

```
<html>  <head> <script>print("a script")</script><title>Título</title>
<script>print("hello")</script><script>print("world")</script>
</head><body>texto suelto<p>párrafo <h1><!-- comentario--><p>
más texto</p></h1></p><div>texto texto texto <br> mas texto texto
texto<div>Un Div que adentro tiene otro<div>Dentro <div><p>de otro</p>
con mas texto</div></div></div></body></html>
```

Salida 4: (stderr)

```
line 6:0 mismatched input '<span class="body">&lt;/body&gt;</span>' expecting TK_C_DIV
```

## 2.2 Manual de Uso

### 2.2.1 Compilación

Desde la carpeta `prettyprinter`, (sin entrar a `src` ni `bin`) se puede compilar el programa con el comando:

```
javac -cp bin/antlrworks-1.5.jar -sourcepath src/ -d bin/ src/Main.java
```

El archivo `antlrworks-1.5.jar` debe estar previamente en la carpeta `bin`.

### 2.2.2 Uso

Dado que el programa lee de la entrada estándar, no necesita parámetros adicionales. Desde la carpeta `prettyprinter` se puede correr con el comando:

```
java -cp bin:bin/antlrworks-1.5.jar Main
```

Para leer desde archivos e imprimir en otro archivo en vez de la consola, se pueden usar `<` y `>` para pasar archivos, de la siguiente manera:

```
java -cp bin:bin/antlrworks-1.5.jar Main < ARCHIVO_DE_ENTRADA.html
> ARCHIVO_DE_SALIDA.html
```

## 2.3 Conclusiones

El problema que el presente trabajo resuelve sirvió como caso de estudio para el desarrollo, en la primera etapa, de una gramática *sin restricciones* (según la clasificación de Chomsky). En esa primera etapa tuvimos que tomar decisiones sobre cómo tokenizar la versión reducida del lenguaje HTML para poder generar, dada una cadena de texto, la versión correctamente coloreada e indentada, siempre y cuando esta formaba una entrada HTML válida.

La primer parte del trabajo no presentó mayores dificultades mas allá de la toma de decisiones referentes a qué parte del parsing sería manejado con producciones en la gramática y cuales con la tokenización.

En una segunda parte desarrollamos una implementación en el lenguaje Java, utilizando el generador de parsers *ANTLR*. La gramática desarrollada no tenía mayores problemas estructurales, por lo tanto procedimos a la sintetización de atributos dentro de *ANTRL*. Luego de una primera entrega, se reportaron errores atribuídos principalmente a dos problemas dentro de la implementación:

1. La salida del parsing era emitida como salida a medida que se sintetizaban los atributos y se parseaba la entrada, esto produjo que para entradas inválidas haya una salida parcial, interrumpida en el momento de emitir un error.

2. Permitir saltos de línea dentro de los tags HTML procesaba como válidos aquellas cadenas de texto donde los tags estaban separados entre 2 o más lineas.

Ambos errores fueron corregidos y la implementación entregada no posee estos problemas.

En cuanto a *ANTLR* resultó ser basntante intuitiva la manera en la que se sintetizan atributos a medida que se realiza el parsing de la entrada. El uso del lenguaje Java dentro de los bloques de código donde se realiza la sintetización otorgan una felixibilidad notable que imaginamos facilita el desarrollo de parsers de mayor complejidad.

Hubiera sido interesante comparar una implementación del mismo parser en tecnologías como *Bison*[3] o *Lex, YACC*[4]. Principalmente porque estos generan parsers LALR a diferencia de *ANTLR* que genera parsers LL(*). Podríamos haber tomado otras decisiones en la gramática y analizar su impacto en la performance de los parsers. Por ejemplo, una gramática recursiva a izquierda no habría sido soportada por *ANTLR* pero sí por *YACC* y *Bison*. Este tipo de comparación comprende un trabajo mayor tanto en la escritura de las gramáticas como en las implementaciones.

[3] http://www.gnu.org/software/bison/
[4] http://dinosaur.compilertools.net/

## 2.4 Apendice - Código

### 2.4.1 prettyprint.g

```
grammar prettyprinter;

options {
    language = Java;
    output = AST;
}


/* *********** PRODUCCIONES *********** */

s returns [String texto]
  : t1=TK_HTML h1=h t2=TK_C_HTML EOF {
      $texto =
          "<html><head>" +
          "<style type=\"text/css\">" +
          "div.bloque {margin-left: 2em;}" +
          "span.html {color:black;}" +
          "span.p {color:purple;}" +
          "span.head {color:blue;}" +
          "span.body {color:blue;}" +
          "span.title {color:red;}" +
          "span.script_tag {color:red;}" +
          "span.script {color:grey;}" +
          "span.h1 {color:fuchsia;}" +
          "span.div {color:green;}" +
          "span.br {color:orange;}" +
          "</style>" +
          "</head><body>" +
          $t1.getText() + $h1.texto + $t2.getText() +
          "</body></html>"
      ;
  };

h returns [String texto]
  : {$texto = "";}
    (h1=head {$texto += $h1.texto;})?
    (b1=body {$texto += $b1.texto;})?
  ;

head returns [String texto]
  : t1=TK_HEAD h1=he t2=TK_C_HEAD {
      $texto = "<div class=\"bloque\">" + $t1.getText() + $h1.texto
      + $t2.getText() + "</div>";
  };

body returns [String texto]
  : t1=TK_BODY b1=b t2=TK_C_BODY {
      $texto = "<div class=\"bloque\">" + $t1.getText() + $b1.texto
      + $t2.getText() + "</div>";
  };

he returns [String texto]
  : sc1=scs t1=TK_TITLE t2=TK_TEXTO t3=TK_C_TITLE scs2=scs {
      $texto = $sc1.texto + "<div class=\"bloque\">" + $t1.getText()
      + $t2.getText() + $t3.getText() + "</div>" + $scs2.texto;
  };

scs returns [String texto]
  : sc1=sc scs1=scs {$texto = $sc.texto + $scs1.texto;}
  | {$texto = "";}  //lambda
  ;

sc returns [String texto]
  : t1=TK_SCRIPT ts=tsc t2=TK_C_SCRIPT {
      $texto = "<div class=\"bloque\">" + $t1.getText() + "<div class=\"bloque\">"
      + "<span class=\"script\">" + $ts.texto + "</span>" + "</div>" + $t2.getText() + "</div>";
  };
```

```
tsc returns [String texto]
  : (
      tk=TK_HTML ts=tsc | tk=TK_C_HTML ts=tsc
    | tk=TK_HEAD ts=tsc | tk=TK_C_HEAD ts=tsc
    | tk=TK_BODY ts=tsc | tk=TK_C_BODY ts=tsc
    | tk=TK_TITLE ts=tsc | tk=TK_C_TITLE ts=tsc
    | tk=TK_DIV ts=tsc | tk=TK_C_DIV ts=tsc
    | tk=TK_H1 ts=tsc | tk=TK_C_H1 ts=tsc
    | tk=TK_P ts=tsc | tk=TK_C_P ts=tsc
    | tk=TK_SCRIPT ts=tsc
    | tk=TK_BR ts=tsc
    | tk=TK_TEXTO ts=tsc
   ) {$texto = $tk.getText() + $ts.texto;}
  | {$texto = "";} //lambda
  ;

b returns [String texto]
  : t1=TK_TEXTO b1=b {
        $texto = "<div class=\"bloque\">" + $t1.getText()  + "</div>" + $b1.texto;
    }
  | t1=TK_DIV b1=b t2=TK_C_DIV b2=b {
        $texto = "<div class=\"bloque\">" + $t1.getText() + $b1.texto
        + $t2.getText() + "</div>" + $b2.texto;
    }
  | t1=TK_H1 b1=b t2=TK_C_H1 b2=b {
        $texto = "<div class=\"bloque\">" + $t1.getText() + $b1.texto
        + $t2.getText() + "</div>" + $b2.texto;
    }
  | t1=TK_P b1=b t2=TK_C_P b2=b {
        $texto = "<div class=\"bloque\">" + $t1.getText() + $b1.texto
        + $t2.getText() + "</div>" + $b2.texto;
    }
  | t1=TK_BR b1=b {
        $texto = "<div class=\"bloque\">" + $t1.getText()  + "</div>" + $b1.texto;
    }
  | {
        $texto = "";  //lambda
  };


/* ********** TOKENS ********** */

WS : (' ' | '\t' | '\r' | '\n')+ {
    $channel = HIDDEN;  //para ignorar los blancos
};

COMM : '<!' .* '>' {
    $channel = HIDDEN;  //para ignorar comentarios
};

TK_HTML returns [String texto]: '<html>' {
    setText("<span class=\"html\">&lt;html&gt;</span>");
};

TK_C_HTML returns [String texto]: '</html>' {
    setText("<span class=\"html\">&lt;/html&gt;</span>");
};

TK_HEAD : '<head>' {
    setText("<span class=\"head\">&lt;head&gt;</span>");
};

TK_C_HEAD : '</head>' {
    setText("<span class=\"head\">&lt;/head&gt;</span>");
};

TK_TITLE : '<title>' {
    setText("<span class=\"title\">&lt;title&gt;</span>");
};

TK_C_TITLE : '</title>' {
    setText("<span class=\"title\">&lt;/title&gt;</span>");
};
```

```
TK_SCRIPT : '<script>' {
    setText("<span class=\"script_tag\">&lt;script&gt;</span>");
};

TK_C_SCRIPT : '</script>' {
    setText("<span class=\"script_tag\">&lt;/script&gt;</span>");
};

TK_BODY : '<body>' {
    setText("<span class=\"body\">&lt;body&gt;</span>");
};

TK_C_BODY : '</body>' {
    setText("<span class=\"body\">&lt;/body&gt;</span>");
};

TK_H1 : '<h1>' {
    setText("<span class=\"h1\">&lt;h1&gt;</span>");
};

TK_C_H1 : '</h1>' {
    setText("<span class=\"h1\">&lt;/h1&gt;</span>");
};

TK_DIV : '<div>' {
    setText("<span class=\"div\">&lt;div&gt;</span>");
};

TK_C_DIV : '</div>' {
    setText("<span class=\"div\">&lt;/div&gt;</span>");
};

TK_P : '<p>' {
    setText("<span class=\"p\">&lt;p&gt;</span>");
};

TK_C_P : '</p>' {
    setText("<span class=\"p\">&lt;/p&gt;</span>");
};

TK_BR : '<br>' {
    setText("<span class=\"br\">&lt;br&gt;</span>");
};

TK_TEXTO : (~('<'))+;  //todo menos <
```

### 2.4.2 Main.java

```
import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;

public class Main {

    public static void main(String[] args) throws Exception {
        prettyprinterLexer lexer = new prettyprinterLexer(new ANTLRInputStream(System.in)) {
            @Override
            public void reportError(RecognitionException e) {
                super.reportError(e);
                throw new RuntimeException();
            }
        };

        prettyprinterParser parser = new prettyprinterParser(new CommonTokenStream(lexer)) {
            @Override
            public void reportError(RecognitionException e) {
                super.reportError(e);
                throw new RuntimeException();
            }
        };

        try {
            System.out.println(parser.s().texto);
        } catch (RuntimeException e) {}
    }

}
```