

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	3
1.1	Introducción	3
1.2	Tokens Léxicos	5
1.2.1	Correcciones Para la implementación	5
1.3	Gramática	6
1.4	Producciones - P	6
1.4.1	Correcciones Para la implementación	7
1.5	Arbol de derivación	8
2	Segunda Parte	12
2.1	Implementación	12
2.1.1	Descripción del problema	12
2.1.2	Detalles de implementación y limitaciones	13
2.1.3	Entradas de prueba válidas	13
2.1.4	Entradas de prueba inválidas	15
2.2	Apendice - Código	16

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 carácter (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 HTML 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 “absorbiendo” 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 regular correspondiente.

<i>Token Léxico</i>	<i>Expresión Regular</i>
<HTML>	<html>
</HTML>	</html>
<HEAD>	<head>
</HEAD>	</head>
<BODY>	<body>
</BODY>	</body>
<TITLE>	<title>
</TITLE>	</title>
<SCRIPT>	<script>
</SCRIPT>	</script>
<DIV>	<div>
</DIV>	</div>
<P>	<p>
</P>	</p>
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 ANTLR¹. La nueva versión del token quedó expresada como:

`texto [~<]*`

Queriendo representar cualquier caracter menos el símbolo `<`

¹[www.antlr.org](http://wwwantlr.org)

1.3 Gramática

La siguiente gramática define el MicroHTML:

$$G = \langle N, \Sigma, P, S \rangle$$

Donde:

$$\begin{aligned} N &= \{S, H, HEAD, BODY, HE, SCS, SC, TSC, B\} \\ \Sigma &= \{texto, \langle HTML \rangle, \langle /HTML \rangle, \langle HEAD \rangle, \langle /HEAD \rangle, \\ &\quad \langle TITLE \rangle, \langle /TITLE \rangle, \langle SCRIPT \rangle, \langle /SCRIPT \rangle, \\ &\quad \langle BODY \rangle, \langle /BODY \rangle, \langle H1 \rangle, \langle /H1 \rangle, \langle DIV \rangle, \langle /DIV \rangle, \\ &\quad \langle P \rangle, \langle /P \rangle, \langle BR \rangle\} \\ S &= S \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow \langle HTML \rangle H \langle /HTML \rangle \\ H &\rightarrow HEAD BODY \mid HEAD \mid BODY \mid \lambda \\ HEAD &\rightarrow \langle HEAD \rangle HE \langle /HEAD \rangle \\ BODY &\rightarrow \langle BODY \rangle B \langle /BODY \rangle \\ HE &\rightarrow SCS \langle TITLE \rangle texto \langle /TITLE \rangle SCS \\ SCS &\rightarrow SC SCS \mid \lambda \\ SC &\rightarrow \langle SCRIPT \rangle TSC \langle /SCRIPT \rangle \\ TSC &\rightarrow \langle HTML \rangle TSC \mid \langle /HTML \rangle TSC \mid \\ &\quad \langle HEAD \rangle TSC \mid \langle /HEAD \rangle TSC \mid \\ &\quad \langle BODY \rangle TSC \mid \langle /BODY \rangle TSC \mid \\ &\quad \langle TITLE \rangle TSC \mid \langle /TITLE \rangle TSC \mid \\ &\quad \langle DIV \rangle TSC \mid \langle /DIV \rangle TSC \mid \\ &\quad \langle H1 \rangle TSC \mid \langle /H1 \rangle TSC \mid \\ &\quad \langle P \rangle TSC \mid \langle /P \rangle TSC \mid \\ &\quad \langle SCRIPT \rangle TSC \mid \langle BR \rangle TSC \mid texto TSC \mid \lambda \\ B &\rightarrow texto B \mid \\ &\quad \langle DIV \rangle B \langle /DIV \rangle B \mid \\ &\quad \langle H1 \rangle B \langle /H1 \rangle B \mid \\ &\quad \langle P \rangle B \langle /P \rangle B \mid \\ &\quad \langle BR \rangle B \mid \\ &\quad \lambda \end{aligned}$$

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:

$$H \rightarrow HEAD \ ? \ BODY \ ?$$

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:

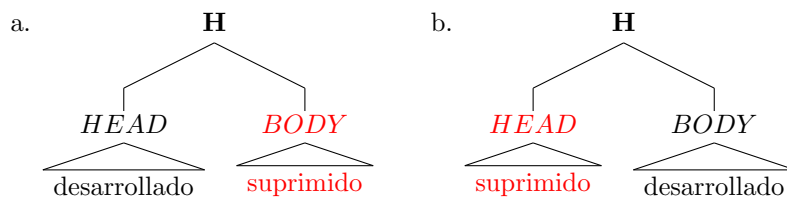
$$S \rightarrow <HTML> \ H \ </HTML> \ \$$$

1.5 Árbol de derivación

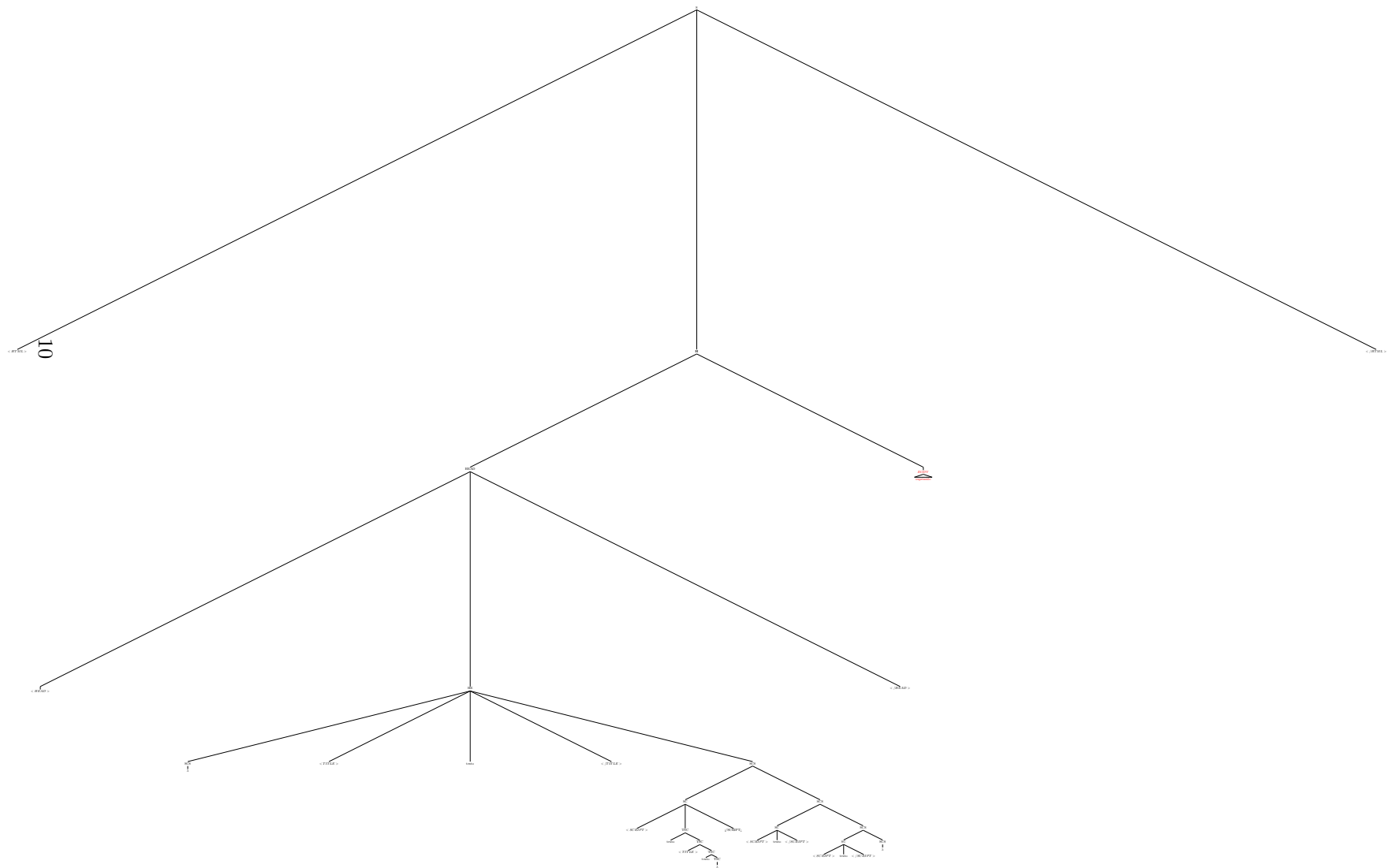
A continuación presentamos el árbol de derivación para el siguiente ejemplo:

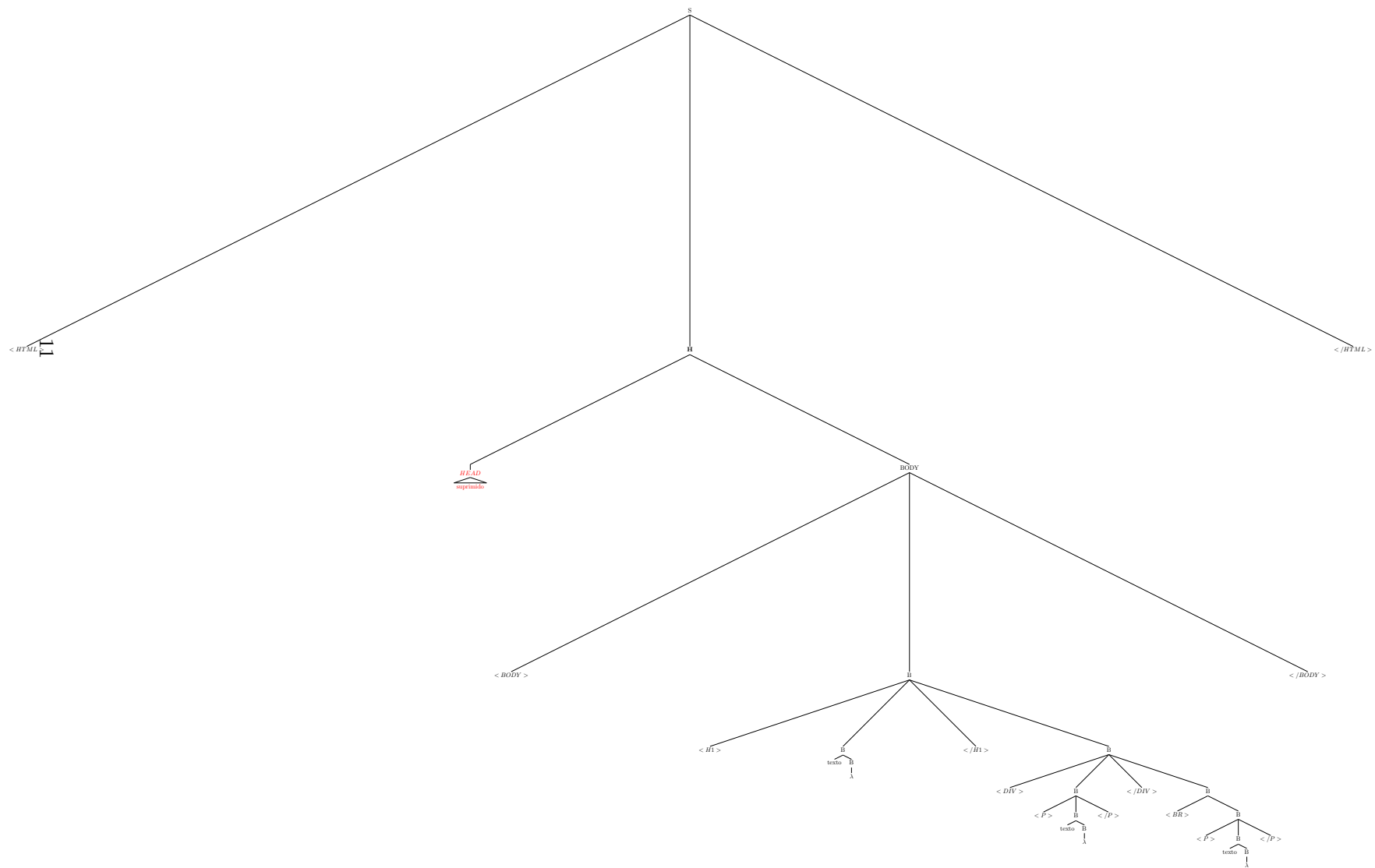
```
<HTML>
  <HEAD>
    <TITLE>Una pagina de ejemplo</TITLE>
    <SCRIPT>
      function unaFunc(){
        alert("esta funcion imprime un tag roto <TITLE>");
      }
    </SCRIPT>
    <SCRIPT></SCRIPT>
    <SCRIPT>alert("aca no aparece el cierre de SCRIPT")</SCRIPT>
  </HEAD>
  <BODY>
    <H1>Un heading</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 (triángulo) simbolizando que en realidad continúa. Por ejemplo:



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





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 ANTLR² 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.

En cuanto al coloreo de los tags, cada tag reconocido tiene su correspondiente código HTML con un estilo (CSS) que le otorga el color.

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.

²[www.antlr.org](http://wwwantlr.org)

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:

```
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:

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

2.2 Apendice - Código

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_TEXT0 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_TEXT0 ts=tsc
    ) {$texto = $tk.getText() + $ts.texto;}
| {$texto = "";} //lambda
;

b returns [String texto]
: t1=TK_TEXT0 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;
    }
    | ti=TK_P bi=b t2=TK_C_P b2=b {
        $texto = "<div class=\"bloque\">" + $t1.getText() + $b1.texto
        + $t2.getText() + "</div>" + $b2.texto;
    }
    | ti=TK_BR bi=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 <

```