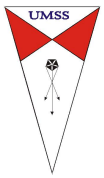


UNIVERSIDAD MAYOR DE SAN SIMÓN
FACULTAD DE CIENCIAS Y TECNOLOGÍA
CARRERA DE LICENCIATURA EN INFORMÁTICA

DESARROLLO DE NAVEGADOR WEB CON HASKELL

Proyecto de Grado, Presentado Para Optar al Diploma Académico
de Licenciatura en Informática.

Presentado por : GOMEZ AJHUACHO DIEGO CARLOS
Tutor : M.Sc. Costas Jáuregui Vladimir Abel

Cochabamba - Bolivia
Julio, 2011

“*Dedicado a mis padres Daniel y Alicia*”

Agradecimientos

No hubiera sido posible llegar a la culminación de este proyecto sin la colaboración de muchas personas, docentes, entidades, y amigos.

En primer lugar quiero agradecer a Dios, en quien he puesto mi fé, quien merece toda la Gloria y Honra, por haberme guiado y ayudado en todo este proyecto.

También agradecer a mis padres, Daniel Gomez y Alicia Ajhuacho, por su amor y apoyo incondicional. A mis hermanos Franz, Lucia, Dorca, David, Eulalia, Martha, Mercedez, y cuñados Gonzalo, Rene y Jaime, y a todos mi sobrinos, especialmente a Jhonatan, Richard, Verito, Tania y Chana. Agradecer a toda mi famifia por su apoyo, paciencia, comprensión, confianza, alegría y sobre todo consejos.

Asimismo, quiero agradecer a Compassion International en Bolvia, por la confianza depositada. A la Familia Greanias por su amor, confianza y apoyo económico.
Y a todos mis amigos del LDP, por brindarme su apoyo moral.

De igual manera, agradecer a las personas que hicieron posible este proyecto, quienes depositaron su confianza en mi persona, me brindaron el apoyo académico y me ofrecieron sus valiosos consejos. Agradecer a mi tutor M.Sc. Vladimir Costas y al Dr. Pablo Azero.

Igualmente, agradecer al MEMI, por abrirme las puertas y recibirme como uno de sus amigos, por brindarme el compañerismo y la confraternidad. Agradecer a la Lic. Leticia Blanco, al Lic. Marcelo Flores, y a la Lic. Claudia Ureña por las importantes recomendaciones proporcionadas.
Además, agradecer a la Comunidad Haskell, Comunidad WxWidgets y CHSS por brindarme su colaboración en detalles técnicos.

Por ultimo, también estoy en deuda con mis amigos y compañeros de estudio. Aquellos que con su alegría, atención y consejos, ayudaron en la culminación de este proyecto: Angie Romero, Lourdes Villca, Thelma Caceres, Antonio Mamani, Richard Jaldin, Lizbet Leaños, Zulma Cabezas, Pamela, Gladys Mamani, Maria Canaviri, Maria Luz Yavi y Armando Mollo.

Resumen

El desarrollo de un Navegador Web funcional es un proyecto gigante, porque se debe desarrollar varios módulos, dar soporte a varias versiones de HTML/CSS e implementar gran cantidad de funcionalidad.
Los actuales Navegadores Web, tales como Firefox, Internet Explorer, Chrome, Safari, son programas muy sofisticados, con más de 10 años de desarrollo y madurez. Estos han sido desarrollados con lenguajes de programación *imperativos*. En el desarrollo de estos programas se necesita que el código sea modular, fácil de comprender, expresivo, mantenible, flexible a cambios, eficiente, etc.

En este proyecto se ha desarrollado un Navegador Web con *Haskell*, un lenguaje de programación *funcional*, que incorpora muchas de las innovaciones recientes del diseño de lenguajes de programación. Se ha implementado un sub-conjunto de la gramática de HTML y CSS, se dio soporte a 48 propiedades de CSS.
En el desarrollo se ha utilizado varias herramientas y librerías de Haskell. Por ejemplo:

- El parser de HTML y CSS fue desarrollado utilizando la librería *uu-parsinglib*, la cual ha beneficiado con un código simple, fácil de entender, expresivo y sobre todo robusto.
- Para la mayor parte del comportamiento de HTML y CSS, se ha utilizado la herramienta *UUAGC*. Esta herramienta ha permitido que se escriba un código simple, comprensible y compacto.
- También se ha utilizado la librería *WxHaskell* para el desarrollo de la interfaz gráfica de usuario y renderización de páginas Web.
- Por último, se ha utilizado la librería *libcurl* para descargar recursos de la Web.

En conclusión, se encontró que el lenguaje de programación funcional Haskell es apropiado y maduro para el desarrollo de un Navegador Web. Las herramientas y librerías utilizadas han jugado un rol importante en la simplificación de la complejidad en el desarrollo del proyecto.

A pesar de que normalmente se utilizan, para el desarrollo de este tipo de programas, lenguajes convencionales e imperativos, Haskell ha sido de bastante utilidad, beneficiando al proyecto con varias de sus características, entre las más importantes: código modular, funciones de alto-orden, evaluación no estricta y emparejamiento de patrones.

Índice general

Índice de códigos Haskell

Índice de códigos UUAGC

Índice de descripciones

Índice de figuras

Capítulo 1

Introducción General

El presente documento contiene la descripción de la implementación de un Navegador Web con Haskell.

Desde sus inicios el Internet ha sido muy prometedora, permitiendo la comunicación entre computadoras al rededor del mundo. Sin embargo, la Internet ha sido aun más exitosa con la creación de un Sistema de Acceso a Documentos, la WWW (World Wide Web) o Web.

De ahí en adelante han adquirido un cierto grado de importancia los Navegadores Web (Web Browsers) y Servidores Web.
De manera que, los documentos almacenados en los Servidores Web son requeridos por los Navegadores Web.

Los Navegadores Web(Sebesta, 2006, p. 7), llamados así porque permiten al usuario navegar por la Web o buscar alguna información almacenada en el servidor, son programas que se ejecutan en la máquina del cliente, los cuales actúan como intermediarios entre la comunicación del usuario y los servidores de páginas Web.

Los actuales Navegadores Web, tales como Firefox, Internet Explorer, Opera, Safari, Chrome, son programas muy sofisticados que están implementados en lenguajes *imperativos*; denominados así debido a que consisten de una secuencia de comandos estrictamente ejecutados uno después del otro(Peyton Jones, s.f.).

Dada la variedad de versiones para tecnologías y estándares Web tanto para HTML, CSS (Cascading Style Sheet), DOM (Document Object Model) y JavaScript, los navegadores Web deben estar implementados con un alto nivel de modularidad, código mantenible y fácil de comprender.

Sin lugar a dudas, estas características existen en los lenguajes imperativos. Sin embargo, el costo de implementar una aplicación con esas características es elevado tanto en tiempo de desarrollo como en el de mantenimiento.

Tener una aplicación con un costo elevado puede afectar directamente a la cantidad de código que se escribe y al tamaño en bytes de la aplicación. También puede dificultar el mantenimiento a la aplicación, dado que las actividades de mantenimiento implican mejorar los productos de software, adaptarlos a nuevos ambientes, y corregir problemas(Fairley, 1987,

cap. 9, p. 334); es necesario comprender bien el código, incluso si se tiene una documentación.

También afecta al costo económico de desarrollo, pues a pesar de que la mayoría de los Navegadores Web son de distribución gratuita, éstos demandan un alto esfuerzo en su desarrollo.

Haskell, según (Peyton Jones, 2002, p. 3), es un Lenguaje de Programación Funcional puro, con evaluación perezosa y de propósito general que incorpora muchas de las innovaciones recientes del diseño de lenguajes de programación.

Haskell provee funciones de alto-orden, semántica no estricta, tipado polimórfico estático, tipos de datos algebraicos definidos por el usuario, emparejamiento de patrones, listas por comprensión, un sistema modular, un sistema monádico I/O, y un conjunto rico de tipos de datos primitivos que incluyen listas, arrays, números enteros de precisión fija y arbitraria, y números de punto flotante.

También se puede observar que Haskell ha crecido y madurado lo necesario como para hacer aplicaciones no sólo en el ámbito académico, sino también comercial (*Haskell Community and Activities Report*, s.f.; *Base de Datos de aplicaciones y librerias de Haskell*, s.f.).

Es por ello que con el presente proyecto se pretende experimentar las capacidades de Haskell mediante el desarrollo de un Navegador Web.

A continuación se describe los objetivos, el alcance, la justificación y descripción general del proyecto.

1.1. Objetivos

1.1.1. Objetivo General

Desarrollar un Navegador Web con el lenguaje de programación funcional Haskell.

1.1.2. Objetivos Específicos

1. Desarrollar el módulo de comunicación entre el Navegador Web y los Protocolos HTTP y modelo TCP/IP.
2. Desarrollar un intérprete de la información HTML.
3. Desarrollar los algoritmos que nos permitirán mostrar la información en la pantalla del Navegador Web.
4. Desarrollar la Interfaz Gráfica de Usuario (GUI).
5. Desarrollar un módulo que de soporte a CSS (Style Sheet Cascade).

1.2. Alcance

Dada la cantidad de versiones de código HTML, XHTML y CSS para el desarrollo, sólo se tomará en cuenta un subconjunto de las versiones estándar. Asimismo, no se considerará los DTD (Document Type Definition) de HTML/XHTML.

Finalmente, tampoco se tomará en cuenta las extensiones (plugins) de un Navegador Web tanto de Java (soporte para Applets de JVM) como de Flash (Flash Player) entre otros.

1.3. Justificación

El desarrollo de este proyecto mostrará cómo las características de la programación funcional pueden colaborar a reducir los costos de desarrollo y mantenimiento, permitiendo un código compacto, modular y fácil de entender.

1.4. Descripción General

El proceso general para renderizar una página puede ser descrito a través de la Figura 1.1.

Figura 1.1: Proceso general de renderización

1.4.1. Obteniendo entradas

Para renderizar una página Web, lo primero que se realiza es obtener la página Web que se quiere renderizar.

Esto se realiza utilizando la dirección *URL* de la página Web y proveyendo el *URL* a la librería *libcurl*, la cual se encarga de comunicarse con el protocolo y devolver un *String* con el contenido de la página Web.

Por otra parte, si el documento contiene imágenes o archivos de hojas de estilo, entonces también se obtienen esos recursos utilizando la librería *libcurl*.

En la Sección 10.2, página 124 se describe detalladamente el proceso de descargar recursos de la Web.

1.4.2. Parseando el Documento

Lo siguiente es analizar la entrada sintácticamente para obtener el árbol de sintaxis abstracta.

Para realizar el análisis sintáctico se utiliza la librería *uu-parsinglib*. El lector interesado en un tutorial simple para la librería *uu-parsinglib* puede revisar el Apéndice A.

En el capítulo 3, 4 y 5 se describe el proceso de análisis sintáctico.

1.4.3. Formateando la Estructura

Para formatear la estructura, se sigue el siguiente proceso:

1. Una vez que se tiene el *NTree*(resultado del análisis sintáctico), se obtienen todas las declaraciones de estilos y luego se asigna un valor a cada propiedad de CSS.

   El Capítulo 6 describe esta parte del proyecto.

2. Luego se genera la estructura de formato. El procesamiento de la estructura de formato se ha dividido en 2 fases:

   - La primera fase, se encarga principalmente de 2 cosas:
     - Generar líneas de *boxes* dependientes de un ancho.
     - Concretizar los valores de algunas propiedades que dependen de otros valores (por ejemplo, porcentajes que dependen del ancho del contenedor).
   - La segunda fase, se encarga de generar ventanas (*Window*) con su respectiva posición, dimensión, funciones de eventos y propiedades de CSS renderizables.

   El Capítulo 7 describe en detalle cada parte del proceso de formatear la estructura de fase 1 y 2.

Para formatear la estructura se hace uso de la herramienta *uuagc*, el lector interesado en un tutorial simple para la herramienta *uuagc* puede revisar el Apéndice B.

1.4.4. Renderizando la Estructura de Formato

La renderización consiste en pintar o dibujar el texto e imágenes con las propiedades de CSS especificadas para cada ventana o *box*.

El Capítulo 8, 9 y 10 describen este proceso.

Capítulo 2

Marco Teórico

En el presente capítulo se presentará los conceptos generales a utilizar en el desarrollo del proyecto, los cuales permitirán tener un buen soporte teórico para el entendimiento de la misma.

2.1. Conceptos Generales

2.1.1. El Internet

El *Internet* es una colección inmensa de equipos conectados a una red de comunicación (Sebesta, 2006, cap. 1, pag. 3). Los equipos conectados pueden ser *routers*, *switches*, *hubs*, *impresoras*, *computadoras*, etc. La comunicación entre estos equipos es realizada gracias a un conjunto de protocolos denominado *TCP/IP*.

El Internet simplemente permite la comunicación entre equipos al rededor del mundo, sin embargo, una tecnología que la hizo mucho más útil fue la aparición de la *World Wide Web (WWW)*.

2.1.2. La WWW o Web

La *World Wide Web* (*WWW*) o simplemente la *Web*, es sistema de acceso a documentos que permite a cualquier usuario, conectado a la Internet a través de una computadora, buscar y recuperar documentos de una computadora que hace el servicio de almacenar esos documentos.

En otras palabras, existe un usuario que juega el papel de *cliente* que necesita un documento. Y también existe un *servidor* que almacena todos los documentos que un cliente puede requerir.

Según (W3C, 1999), la Web provee de 3 mecanismos para permitir la comunicación entre cliente-servidor:

1. URL, un esquema de nombramiento uniforme para la localización de recursos/documentos en la Web.
2. HTTP, un protocolo para acceder a los recursos/documentos en la Web.
3. HTML, un lenguaje de marcado con hipertexto para la navegación entre los recursos de la Web.

La comunicación entre cliente-servidor es realizada utilizando el protocolo *HTTP* el cual se encarga de comunicarse con el servidor a través del conjunto de protocolos TCP/IP y de devolver un resultado al cliente.

En realidad, el cliente puede ser cualquier dispositivo electrónico o software que utiliza el protocolo *HTTP* para buscar o recuperar documentos. Por ejemplo, el Navegador Web es el cliente más utilizado por un usuario para buscar o recuperar documentos de la Web.

Los documentos que un cliente puede requerir pueden ser de distintos tipos (texto, imágenes, etc), pero el más común es el *hipertexto*, el cual es simplemente texto que contiene enlaces a otros documentos. El formato de un documento *hipertexto* está definido por un lenguaje de marcado como: HTML, XHTML.

En conclusión, la Web es una colección amplia de documentos, de los cuales algunos están conectados con enlaces. Esos documentos, que son proveídos por un Servidor Web, son comúnmente accedidos con un Navegador Web.

HTTP

HTTP, significa *Hypertext Tansfer Protocol*, es el protocolo que permite la comunicación cliente-servidor para obtener documentos (Brian Totty, s.f.). El protocolo HTTP interactúa con los protocolos TCP/IP para ejecutar las operaciones entre cliente-servidor.

Las operaciones que HTTP provee son:

- GET: permite al cliente recuperar un documento del servidor.
- PUT: permite al cliente almacenar un documento en el servidor.
- DELETE: permite al cliente eliminar un documento del servidor.
- POST: permite al cliente enviar información a una aplicación del servidor.
- HEAD: permite al cliente enviar solo las cabeceras de la respuesta al servidor.

En la siguiente figura se muestra un ejemplo del proceso de obtener una imagen para un cliente desde un servidor:

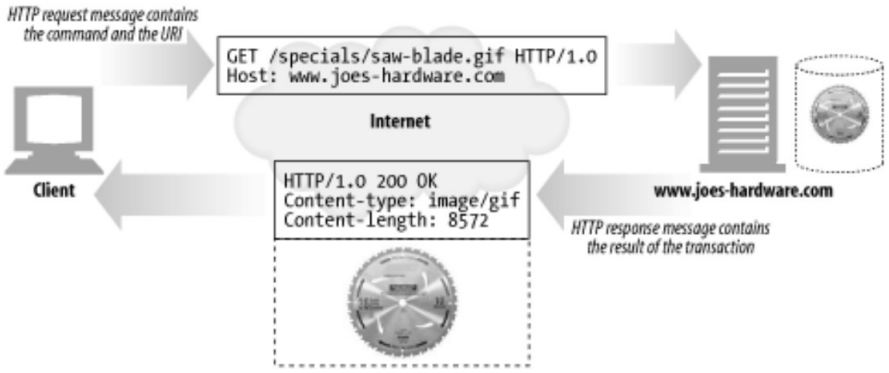


Figura 2.1: Proceso de obtener una imagen con HTTP, Fuente: "HTTP, The definitive guide" (Brian Totty, s.f., cap. 1)

URL

El URL, significa *Uniform Resource Locator*, es la dirección específica que un recurso o documento tiene en el servidor. Por ejemplo, la dirección URL de la imagen que se obtiene del servidor en la Figura 2.1 es: `www.joes-hardware.com/specials/saw-blade.gif`. Con esa dirección URL, el usuario puede obtener la imagen desde el servidor.

El formato de una dirección URL es:

```
<protocolo>://<usuario>:<contrasenia>@<equipo>
           :<puerto>/<dirección>:<parámetros>?<consulta>#<sección>
```

Descripción 1 Formato de una dirección URL

No todos los campos de la Descripción 1 son necesarios para la URL, pero los más importantes son: protocolo, equipo y dirección. El campo *protocolo* puede ser: HTTP, File, FTP, etc. El campo *equipo* puede ser una dirección IP o también una dirección IP textual del equipo. Y finalmente el campo *dirección* corresponde a la dirección del documento que se quiere obtener en el servidor.

2.1.3. Navegador Web

Un Navegador Web es un programa informático que se encarga de renderizar documentos. Los tipos de documentos que puede renderizar depende del soporte con el que se ha implementado, pero los más comunes son: HTML, texto e imágenes.

En su forma más simple, un Navegador Web puede trabajar con el protocolo File y renderizar páginas Web que se encuentran en la misma computadora donde se ejecuta el programa. Pero normalmente trabaja con varios protocolos, por ejemplo: HTTP, FTP. Cuando el Navegador Web trabaja con el protocolo HTTP, el Navegador Web juega el papel de cliente en la comunicación cliente-servidor. El usuario requiere un documento, luego el Navegador Web interactúa con el servidor para obtenerlo y finalmente renderizarlo para el usuario.

2.2. HTML/XHTML

HTML/XHTML son lenguajes de marcado que describen la forma y esquema de los documentos que serán mostrados por un Navegador Web (Sebesta, 2006).

Los documentos descritos por un lenguaje de marcado están compuestos de elementos y contenido. Los elementos son descritos por las etiquetas de HTML/XHTML, los cuales son utilizados para delimitar partes del contenido. Por ejemplo, el elemento de párrafo es descrito por una etiqueta de inicio '`<p>`' y una etiqueta de fin '`</p>`', de manera que todo lo que está encerrado dentro de las etiquetas llega a formar parte del contenido del elemento.

Las etiquetas para un elemento tienen un formato especial. La etiqueta de inicio está compuesto por 2 símbolos (`<>`) y el nombre de la etiqueta. Por ejemplo, la etiqueta de inicio para el elemento de HTML es: `<html>`. A diferencia de la etiqueta de inicio, la etiqueta de fin está compuesto de 3 símbolos (`</>`) y el nombre de la etiqueta; por ejemplo, la etiqueta de fin para el elemento HTML es: `</html>`.

Existen 2 formas de definir un elemento: *normal* y *especial*. La forma normal es utiliza una etiqueta de inicio y otra de fin. Sin embargo, la forma especial solo utiliza una etiqueta especial de inicio, es decir, no tiene una etiqueta de fin, ni tampoco tiene contenido. Las etiquetas de inicio especiales son definidos con 3 símbolos (`</>`) y el nombre de la etiqueta. Por ejemplo, la etiqueta especial para el elemento IMG es: `<img />`.

Opcionalmente, los elementos pueden contener definiciones de atributos. Básicamente, los atributos proveen información adicional para el Navegador Web (Sebesta, 2006). Por ejemplo, la Descripción 2 muestra que el atributo 'src' del elemento IMG provee la dirección URL del contenido:

```
<img src = "image.jpg" />
```

Descripción 2 Atributo 'src' del elemento IMG

El formato para especificar un atributo es: nombre del atributo, símbolo '=' y contenido del atributo encerrado entre comillas.

2.2.1. DTD

Un DTD (*Document Type Definition*) define la estructura del documento para un lenguaje de marcado específico. Por ejemplo, HTML define 3 tipos de DTD:

- Strict
- Transitional
- Framed

De manera general, el DTD para HTML/XHTML define varios grupos de elementos de los cuales los más importantes son: *inline* y *block*. Los elementos *inline* son los que se renderizan horizontalmente uno seguido del otro; pero los de *block* se renderizan verticalmente uno debajo del otro. Por ejemplo, algunos elementos *block* son: $p, h1 - h6, div, blockquote, etc.$ De la misma manera, algunos elementos *inline* son: $big, small, em, img, etc.$

Otros grupos de elementos *inline* que define el DTD son: *fontstyle* (tt, i, b, big, small), *phrase* (em, strong, q, etc.), *special* (a, img, br).

2.2.2. Árbol del documento

El árbol del documento es una representación abstracta en forma de árbol de la información que puede ser renderizada en un Navegador Web.

Por ejemplo, en la Figura 2.2 se muestra el árbol del documento de la Descripción 3, donde *text* representa el contenido de un elemento que contiene solo texto:

```
<html>
    <head>
        <style>
            p {color: red}
        </style>
    </head>
    <body>
        <p> texto1 </p>
        <p> texto2 </p>
    </body>
</html>
```

Descripción 3 Ejemplo de HTML

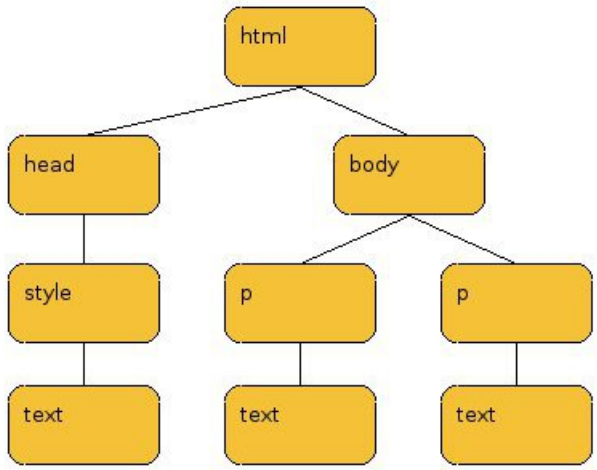


Figura 2.2: Árbol del documento

2.3. La especificación de CSS

La especificación de CSS (*Cascading StyleSheet*) provee 3 cosas importantes:

- Modelo y Comportamiento para la renderización de un documento
- Lenguaje para hojas de estilo
- Propiedades de CSS

2.3.1. Modelo y Comportamiento

A continuación se presenta los distintos modelos que la especificación de CSS define:

Modelo en Cascada

La especificación de CSS define el modelo en Cascada, la cual permite definir las hojas de estilo de 3 formas (o niveles) y por 3 tipos de usuario diferentes, los cuales determinan el

origen de donde proviene la hoja de estilo.

Las 3 formas de definición son:

1. *Declaraciones de Estilo*.- Las declaraciones de estilo pueden encontrarse en el atributo '*style*' de un elemento. En el siguiente ejemplo se muestra las declaraciones de estilo del elemento *span*:

```
<span style = "font-size: 12pt; font-style: italic">
    contenido del elemento span
</span>
```

Descripción 4 Declaraciones de Estilo en un atributo

2. *Hojas de Estilo Internas*.- Las hojas de estilo son especificadas dentro (*interna*) del documento que se va a renderizar. Se utiliza el elemento *style* para especificar las hojas de estilo. En el siguiente ejemplo se muestra las hojas de estilo internas de un documento de HTML:

```
<html>
    <head>
        <style>
            span { font-size: 12pt
                 ; font-style: italic
                 }
        </style>
    </head>
    <body>
        <p>
            Esto es un <span>ejemplo</span>
            de hoja de estilo <span>interna.</span>
    </body>
</html>
```

Descripción 5 Ejemplo de Hojas de Estilo Internas

3. *Hojas de Estilo Externas*.- Las hojas de estilo se encuentran en un archivo externo, pero son especificadas en el documento a renderizar a través del elemento *link*. El siguiente ejemplo ilustra la forma de especificar las hojas de estilo externas:

```
<html>
    <head>
        <link rel="stylesheet" type="text/css" href="estilo.css"/>
    </head>
    <body>
        <p>
            Esto es un <span>ejemplo</span>
            de hoja de estilo <span>interna.</span>
    </body>
</html>
```

Descripción 6 Ejemplo de Hojas de Estilo Externas

Además de las 3 formas de definición de hojas de estilo, también existe 3 tipos de usuarios que pueden definir hojas de estilo en alguna de las 3 formas:

1. *Author*. Hace referencia al creador del documento a renderizarse, el cual puede definir hojas estilos en cualquiera de las 3 formas.

2. *User*. Hace referencia al usuario que interactua con el documento a través de un Navegador Web. Este tipo de usuario solo puede definir una hoja de estilo externa, la cual afectará la renderización final del documento en el Navegador Web.

3. *User Agent*. Se refiere al Navegador Web. El Navegador Web tiene una hoja de estilo la cual es utilizada cuando los otros usuarios no definen una hoja de estilo.

Modelo de Procesamiento

La especificación de CSS define un posible modelo de procesamiento de documentos que podría ser adoptado por un Navegador Web. A continuación se muestra la secuencia de pasos que define el modelo:

1. Parsear el documento fuente y crear el árbol del documento.

2. Identificar el tipo de media destino. Para el proyecto, el tipo de media siempre será la pantalla, porque los documentos son renderizados en la pantalla.

3. Recuperar todas las hojas de estilo asociados con el documento y que son especificados para el tipo de media destino.

4. Para cada elemento del árbol del documento, asignar un valor a cada propiedad que es aplicable al tipo de media destino.

5. Generar la estructura de formato desde el árbol del documento.

6. Convertir la estructura de formato al tipo de media destino. Para el proyecto, esto implica renderizar la estructura de formato en la pantalla.

Algoritmo en Cascada para la asignación de valores a propiedades

La especificación de CSS también define un algoritmo para la asignación de un valor a una propiedad de CSS. El algoritmo es especificado como una secuencia de pasos:

1. Encontrar las declaraciones de estilo que aplican al elemento y propiedad en cuestión. Las declaraciones aplican si el selector empareja con el elemento en cuestión.

2. Ordenar la lista de declaraciones de acuerdo a la importancia (*normal* o *important*) y usuario (*Author*, *User*, *UserAgent*) en forma ascendente:

   a) Declaraciones *UserAgent*

   b) Declaraciones *Normal* de *User*

   c) Declaraciones *Normal* de *Author*

   d) Declaraciones *Important* de *Author*

   e) Declaraciones *Important* de *User*

3. Ordenar todas las declaraciones que tienen la misma importancia y usuario de acuerdo a la especificidad del selector.

   La especificidad de un selector es un número de 4 dígitos: *abcd*, donde $a = 1, b = c = d = 0$ si se trata de una declaración de estilo, caso contrario $a = 0$, b es igual a la cantidad de atributos *ID* que aparecen en el selector, c es la cantidad de atributos diferentes a *ID*, y d es el número de nombres de elementos que aparecen en el selector.

4. Si aún no se puede determinar que declaración se va a utilizar, entonces se ordena de acuerdo al orden en que han sido especificados en la hoja de estilos fuente. El último que ha sido especificado es el que se selecciona.

Modelo de valores para las propiedades de CSS

CSS define 4 tipos de valores para cada propiedad de CSS:

- *specified-value*: Corresponde al valor especificado en las hojas de estilo. El algoritmo en Cascada (Sección 2.3.1, página 12) se encarga de asignar un valor para este campo de la propiedad.

- *computed-value*: Es un valor preparado para ser heredado por otras propiedades o elementos.

- *used-value*: Se constituye en el valor que no tiene dependencias de otras propiedades de CSS. Por ejemplo, es aquí donde un valor con porcentaje es convertido a un valor concreto, es decir a un valor que no tiene porcentaje.

- *actual-value*: Corresponde al valor sin ningún tipo de limitaciones. Por ejemplo, si por motivos de la librería gráfica, el Navegador Web no tiene soporte para una característica, entonces el valor para este campo de la propiedad es reemplazado por un valor por defecto.

  El *actual-value* es el valor utilizado por un Navegador Web para la renderización.

Modelo Box de CSS

La especificación de CSS define el modelo Box, el cual es utilizado para representar todos los elementos que se renderizan en el Navegador Web. Un Box es una caja rectangular compuesto de 4 áreas:

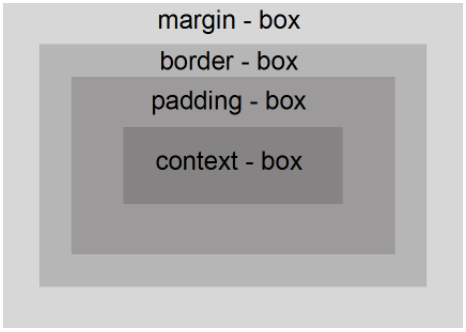


Figura 2.3: Modelo Box de CSS

- `Content-Box`. Es el área donde se renderiza el contenido del *box*, éste puede ser una imagen o también un texto. Sus dimensiones están fijadas por las propiedades *width* y *height* de CSS.
- `Padding-Box`. Es la distancia opcional que separa el `content-box` del borde. La distancia de separación es controlada por el conjunto de propiedades de *padding*.
- `Border-Box`. Es el área opcional que representa el borde del *box*. El ancho del *Border-Box* es fijado por el conjunto de propiedades de *border*.
- `Margin-Box`. Es el área opcional que representa el margen externo del *box*. El ancho del *Margin-Box* es fijado por el conjunto de propiedades de *margin*.

2.3.2. Lenguaje para hojas de estilos

CSS también provee un lenguaje para describir las hojas de estilos de un documento que va a ser renderizado en un Navegador Web.

El capítulo 4 de la especificación de CSS (W3C, 2009) define la sintaxis y tipos de datos básicos del lenguaje para las hojas de estilos de CSS.

2.3.3. Propiedades de CSS

Las propiedades de CSS se encargan de guiar la renderización de un documento en un Navegador Web. CSS ha creado como 98 propiedades, que se aplican a distintos tipos de media destino. Por ejemplo, para el tipo de media *visual* o *pantalla*, CSS ha definido como 76 propiedades.

Cada propiedad de CSS tiene un nombre, sintaxis para sus valores, un valor por defecto, un conjunto de elementos a los cuales se aplica la propiedad, un campo para especificar si la propiedad es heredable, un campo para indicar la forma de procesar los valores que tienen porcentajes, un campo para indicar el tipo de media destino, y finalmente un campo para indicar la forma de procesar el *computed-value* de una propiedad.

Por ejemplo, a continuación se presenta la descripción de la propiedad *font-size* de CSS:

```
'font-size'
        Value           : <absolute-size> |  <relative-size> | <length>
                        | <percentage>    |  inherit
        Initial         : medium
        Applies to      : all elements
        Inherited       : yes
        Percentages     : refer to inherited font size
        Media           : visual
        Computed value  : absolute length

<absolute-size> ::=  xx-small   |   x-small | small | medium | large
                   | x-large   |   xx-large

<relative-size>  ::= larger | smaller
<length>         ::= pixel  | point | em
<percentage>     ::= Number%
<pixel>          ::= Number‘px’
<point>          ::= Number‘pt’
<em>             ::= Number‘em’
```

Descripción 7 Propiedad font-size de CSS, Fuente: Especificación de CSS

2.4. Haskell

Haskell, según (Peyton Jones, 2002, p. 3), es un Lenguaje de Programación Funcional puro, con evaluación perezosa y de propósito general que incorpora muchas de las innovaciones recientes del diseño de lenguajes de programación.

Haskell provee funciones de alto-orden, semántica no estricta, tipado polimórfico estático, tipos de datos algebraicos definidos por el usuario, emparejamiento de patrones, listas por comprensión, un sistema modular, un sistema monádico I/O, y un conjunto rico de tipos de datos primitivos que incluyen listas, arrays, números enteros de precisión fija y arbitraria, y números de punto flotante.

Existe una amplia cantidad de librerías y herramientas para Haskell. Sin embargo, 3 de las principales librerías y herramientas que se utiliza en el proyecto son:

- Librería *uu-parsinglib*(Swierstra, s.f.-b). Es una librería para Haskell que permite describir la gramática de un lenguaje. Ésta es utilizada para el analizador sintáctico (Parser) del proyecto. En el Apéndice A se tiene un tutorial básico para la utilización de la librería.

- Herramienta *UUAGC*(Swierstra, s.f.-a). Es una herramienta que genera código Haskell a través de una descripción de gramática de atributos del comportamiento. Ésta es utiliza para la especificación del comportamiento de la mayor parte del proyecto. Se tiene un tutorial básico en el Apéndice B.

- Librería *WxHaskell*(Leijen, 2004). Es una librería para Haskell que permite implementar

la Interfaz Gráfica de Usuario de un programa. Ésta es utilizada para implementar la parte gráfica del proyecto.

2.5. Trabajos relacionados

Uno de los documentos importantes para el proyecto es la especificación de CSS (Sección 2.3, página 9), la cual detalla los aspectos más relevantes para la implementación de un Navegador Web.

Sin embargo, no se encontró muchos documentos técnicos sobre la forma de implementación de la especificación de CSS en los Navegadores Web actuales (Internet Explorer, Chrome).

2.5.1. Firefox

En la página de *David Baron* (Ingeniero en Desarrollo de Software de Firefox y miembro del W3C) se encontró algunos vídeos y documentos técnicos que describen partes generales y algunas partes concretas sobre Firefox, uno de los Navegadores Web actuales que tiene varios años de desarrollo.

La forma de procesamiento de hojas de estilos de Firefox es realizada de forma optimizada, Firefox construye un árbol lexicográfico de las reglas de estilo, lo cual sería interesante implementar en el proyecto.

2.5.2. WWWBrowser

También se encontró un proyecto de un Navegador Web (WWWWBrowser) implementado con Haskell utilizando la librería gráfica *Fudgets* (Carlsson y Hallgren).

El proyecto WWWBrowser fue implementado por los años 90 y dio soporte para las versiones 2 y 3.2 de HTML (actualmente HTML se encuentra en la versión 4 y queriendo avanzar a la versión 5). Implementó la mayor parte de HTML incluyendo imágenes, tablas y formularios. Utilizaron la librería *uuparsing* de la Universidad de Utrecht para el parser de HTML y también utilizaron programación paralela para el cargado de múltiples imágenes.

Algo interesante del proyecto WWBrowser es que la renderización de un documento fue realizada utilizando mecanismos de acomodación (*layout*) de la librería Fudget, es decir, que el posicionamiento y redimensionamiento no fue realizado de forma manual, sino con las propias funciones de la librería *Fudgets*.

2.5.3. HXT

Existen varias librerías para Haskell que trabajan con XML/HTML, entre ellas, una de las más utilizadas es la librería HXT (Haskell Xml Toolbox).

HXT tiene soporte para XML/HTML/XHTML y DTD. Su parser está implementado utilizando la librería *parsec* de Haskell, también utiliza la librería *Arrow* para la implementación de la funcionalidad que provee la librería.
Uno de los aspectos importantes de la librería HXT es su estructura *NTree* la cual utiliza varios tipos de nodos para almacenar varios tipos de información que es reconocida por el parser.

Capítulo 3

Sintaxis Concreta y Abstracta

La primera tarea a realizar en el desarrollo de un Navegador Web es entender y representar la entrada para el Navegador Web. Ésta entrada es una página Web, que está descrita a través de un lenguaje de marcado como HTML, que puede contener, dentro del archivo HTML, reglas de hojas de estilo que son descritas a través del lenguaje para hojas de estilos de CSS. Entonces, la entrada para el Navegador Web está descrita a través de dos lenguajes, uno de marcado (HTML) y otro de estilos (CSS).

Los lenguajes son descritos a través de una sintaxis concreta, al mismo tiempo, una sintaxis concreta puede ser representada utilizando una notación EBNF.
Para entender un lenguaje, básicamente, se necesita conocer su sintaxis concreta. Y para representarlo, se necesita una sintaxis abstracta.
La importancia de representar un lenguaje en Haskell radica en la necesidad de aplicar futuras operaciones sobre la entrada, de manera que ésta pueda ser renderizada como una página Web.

En las siguientes secciones del capítulo, se describirá en más detalle la sintaxis concreta y abstracta de un lenguaje. También se mostrará la sintaxis concreta y abstracta para un lenguaje de marcado y de estilos.

3.1. Sintaxis concreta y abstracta de un lenguaje

Una sintaxis concreta es descrita a través de una gramática(Jeuring y Swierstra, 2000, p. 18), la cual está compuesta de un conjunto de reglas de producción, símbolos terminales, símbolos no terminales y un símbolo de inicio. Estos 4 componentes definen la sintaxis para la gramática de un lenguaje.

La sintaxis abstracta también puede tener los 4 componentes de una gramática, pero normalmente sólo guarda la información que es importante para el lenguaje (por ejemplo, no guarda los símbolos terminales de la gramática). En otras palabras, una sintaxis abstracta *abstrae* la información importante de la gramática.

3.1.1. Notación BNF y EBNF

BNF, la versión no extendida de **EBNF**, provee un formalismo para describir la sintaxis de un lenguaje. El formalismo de **BNF** consiste de un conjunto de reglas de producción, que están compuestos de *Terminales y No-Terminales*.

Un *No-Terminal* es simplemente un nombre que hace referencia a una regla de producción. Y un *Terminal* es un símbolo o elemento del lenguaje (un *Terminal* no hace referencia a una regla de producción). También se tiene un *Terminal* especial denominado *épsilon*, que significa que no produce nada.

Concretamente, una regla de producción consiste de un *No-Terminal* en el lado izquierdo y un conjunto de producciones en el lado derecho. Las producciones del lado derecho de una regla pueden contener *Terminales y No-Terminales*, si se tiene dos o más producciones en el lado derecho, deben estar separadas por el símbolo '`|`', que indica que es una producción alternativa.
Además, el lado izquierdo y derecho están separados por un símbolo '`::=`' que significa que el *No-Terminal* de la izquierda genera las reglas de producción de la derecha.

A continuación se muestra la gramática de un lenguaje que produce palíndromos para los caracteres 'a','b' y 'c':

```
Pal ::= epsilon
     |  'a'
     |  'b'
     |  'c'
     |  'a' Pal 'a'
     |  'b' Pal 'b'
     |  'c' Pal 'c'
```

Descripción 8 Sintaxis concreta para palíndromos

La versión extendida de **BNF** (**EBNF**) añade características para hacer la gramática del lenguaje más legible:

- **Paréntesis de Agrupación**
  Permite agrupar *Terminales y No-Terminales* entre paréntesis.
- **Ocurrencia Opcional (?)**
  Hace que el símbolo al que se refiere tenga una ocurrencia de uno o cero.
- **Ocurrencia de Lista (+)**
  Hace que el símbolo al que se refiere tenga una ocurrencia de uno o más símbolos.
- **Ocurrencia de Lista (∗)**
  Hace que el símbolo al que se refiere tenga una ocurrencia de cero o más símbolos.

Un símbolo puede ser tanto *Terminales, No-Terminales o agrupaciones.*

3.1.2. Ejemplo de sintaxis concreta y abstracta

Para una mejor comprensión de la relación entre sintaxis concreta y abstracta se revisará en detalle el ejemplo de la Descripción 8.

La gramática para la Descripción 8 sería:

```
Gramatica para Pal:
    produccion
        -> Pal
    simbolos terminales
        -> {‘a’, ‘b’, ‘c’}
    simbolos no-terminales
        -> {Pal}
    simbolo de inicio
        -> {Pal}
```

Descripción 9 Gramática para palíndromos

La sintaxis abstracta es similar a la sintaxis concreta, pero sólo representa lo más importante de la sintaxis concreta. Por ejemplo, para el caso de los palíndromos se puede usar el tipo *Char* de Haskell para representar ‘a’, ‘b’, ‘c’; se puede usar un constructor especial *NoPal* para representar `épsilon`; Y para las distintas versiones de `‘a’ Pal ‘a’`, `‘b’ Pal ‘b’`, `‘c’ Pal ‘c’` se puede usar el tipo *Char* para el carácter que se repite en ambos lados y una referencia a otro palíndromo:

```
data Pal
  = NoPal
  | SimplePal    Char
  | ComplexPal Char Pal
```

Código Haskell 1: Sintaxis Abstracta para la sintaxis concreta de palíndromos

En el siguiente ejemplo se muestra la sintaxis abstracta para el palíndromo `"cbaabc"`:

```
test1 = ComplexPal 'c' (ComplexPal 'b' (ComplexPal 'a' NoPal))
```

Código Haskell 2: Ejemplo de sintaxis abstracta

3.2. Lenguaje de Marcado genérico

En está sección se definirá la sintaxis concreta y abstracta para un lenguaje de marcado genérico.

3.2.1. Sintaxis Concreta para un *Lenguaje de Marcado* Genérico

Un *Lenguaje de Marcado* genérico es un lenguaje que utiliza marcas o etiquetas para estructurar un documento o texto. Éstas etiquetas no necesariamente corresponden a las eti-

quetas de *HTML*, sino que también pueden ser etiquetas de *XML*, de ahí su nombre genérico.

Las etiquetas del lenguaje genérico tienen una única restricción dentro del documento, que el nombre de la etiqueta de inicio sea el mismo que la etiqueta final (si es que tiene etiqueta final).

A continuación se presenta la sintaxis concreta para el lenguaje de marcado genérico:

```
Marcado ::= '<' Identificador Atributo* '>'  Elemento*  '<' '/' Identificador '>'
         |  '<' Identificador Atributo* '/' '>'
Elemento ::= Marcado
          |  Texto
Atributo ::= Identificador '=' Valor
Valor ::= '"' ContenidoAtributoValor '"'
Identificador ::= AlphaNum+
Valor ::= AlphaNum+ | ' ' | '\t' | '\r'
Texto ::= ...   Cualquier texto excepto ''<>/''
AlphaNum ::= Alpha
          |  Num
Alpha ::= 'a' | 'b' | 'c' | .. | 'z'
        | 'A' | 'B' | 'C' | .. | 'Z'
Num ::= '0' | '1' | .. | '9'
```

Descripción 10 *Sintaxis Concreta* para un lenguaje de marcado genérico (Fuente: Elaboración propia)

Con la sintaxis concreta de la Descripción 10 se puede describir el siguiente ejemplo de *HTML*:

```
<html>
    <head>
        <title> Ejemplo 1: El Saludo </title>
    </head>
    <body>
        <p name="saludo"> Hola Mundo Cruel. </p>
        <img src="saludo.png" />
    </body>
</html>
```

Descripción 11 Ejemplo HTML

3.2.2. Gramática Abstracta para el *Lenguaje de Marcado*

Revisando la bibliografía (Bird, 2000; Ohlendorf, 2007), normalmente se utiliza una estructura *Rosadelfa* para representar un *Lenguaje de Marcado* (por ejemplo **HXT** (Haskell Xml Toolbox) utiliza una estructura *Rosadelfa* llamada **NTree**(Ohlendorf, 2007, cap. 2, p. 8)). En este proyecto también se utilizará una estructura *Rosadelfa*.

Una *Rosadelfa* o *Árbol Rosa* (Bird, 2000, cap. 6, p. 167) es una estructura para describir árboles que tienen una ramificación múltiple, cada ramificación tiene un nodo, que es utilizado para guardar la información del árbol.

A continuación se muestra la implementación de una estructura Rosadelfa en Haskell:

```
data ArbolRosa
   = ArbolRosa Nodo [ArbolRosa]
data Nodo
   = NTag Nombre Atributos
   | NTexto String
type Atributos = [(Nombre, Valor)]
type Nombre = String
type Valor = String
```

Código Haskell 3: Estructura Rosadelfa

En el tipo de dato *Nodo* del Código Haskell 3 se ha definido dos constructores: *NTag* y *NTexto*. El constructor *NTag* guarda el nombre y atributos de una etiqueta y el nodo *NTexto* sólo guarda el texto (*String*).

Antes de representar el ejemplo *HTML* del Descripción 11, considere el siguiente gráfico:

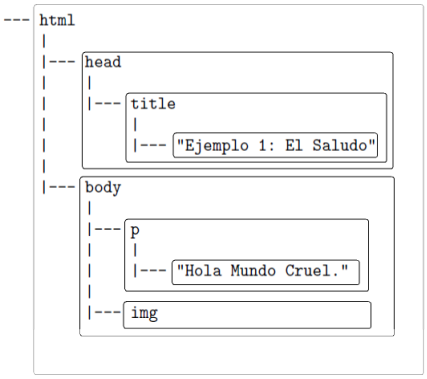


Figura 3.1: Gráfico representativo de un árbol rosa

Cada rectángulo en la Figura 3.1 es un *ArbolRosa* y las líneas dirigen a cada elemento de una lista de *Árboles Rosa* (Porque la estructura dice que un *ArbolRosa* contiene un *Nodo* y una lista de *árboles Rosa).*

Así, una etiqueta especial como el *img*, es un Árbol Rosa, con la diferencia de que no tiene ramificaciones; un *texto*, también es un Árbol Rosa, que tampoco tiene ramificaciones. Se debe notar que la diferencia entre una *etiqueta* y un *texto* está en el tipo de nodo.

A continuacion se muestra la representación de la Descripción 11 en el Código Haskell 4.

```
texto1 :: ArbolRosa
texto1 = ArbolRosa (NTexto "Ejemplo 1: El Saludo") []
texto2 :: ArbolRosa
texto2 = ArbolRosa (NTexto "Hola Mundo Cruel.") []
title :: ArbolRosa
title = ArbolRosa (NTag "title" []) [texto1]
jead :: ArbolRosa
jead = ArbolRosa (NTag "head" []) [title]
p :: ArbolRosa
p = ArbolRosa (NTag "p" [("name","saludo")]) [texto2]
img :: ArbolRosa
img = ArbolRosa (NTag "img" [("src","saludo.png")]) []
body :: ArbolRosa
body = ArbolRosa (NTag "body" []) [p, img]
html :: ArbolRosa
html = ArbolRosa (NTag "html" []) [jead, body]
```

Código Haskell 4: Ejemplos de arboles rosa

3.3. Lenguaje para hojas de estilos CSS

En está sección se definirá la sintaxis concreta y abstracta para CSS.

3.3.1. Sintaxis Concreta para *CSS*

Cascading Style Sheets (CSS) (W3C, 2009), que traducido al castellano es *Hojas de Estilo en Cascada*, es un lenguaje de hojas de estilos que permite a los usuarios adjuntar un estilo para documentos estructurados (como ser *HTML*, *XML*).

La especificación de *CSS* (W3C, 2009) define la sintaxis para el lenguaje de estilos de CSS, su sintaxis es amplia y con soporte para versiones anteriores. En este proyecto sólo se considera la parte central de CSS, cuya sintaxis se define en la Descripción 12 (note que no se considera todas las declaraciones de *pseudo-selectores*, ni tampoco las declaraciones de *charset, import, media y page*).

```
HojaEstilo :: Regla*
Regla ::= Selectores ‘{’ Declaraciones ‘}’
Selectores ::= Selector
            |  Selector (‘,’ Selector)+
Selector ::= Simple_Selector
          |  Simple_Selector Operador Selector
Simple_Selector ::= Nombre_Selector Atributo* Pseudo?
                  |                 Atributo+ Pseudo?
                  |                           Pseudo
Nombre_Selector ::= Identificador
                  |  ‘*’
Operador ::= ‘ ’ | ‘>’ | ‘+’
Atributo ::= ‘#’ Identificador
          |  ‘.’ Identificador
          |  ‘[’ Identificador ‘]’
          |  ‘[’ Identificador OperadorAT ValorAT ‘]’
OperadorAT ::= ‘~=’ | ‘=’
ValorAT ::= Cadena
Pseudo ::= ‘:’ ‘‘before’’
        |  ‘:’ ‘‘after’’
Declaraciones ::= Declaracion
               |  Declaracion (‘;’ Declaracion)+
Declaracion ::= Identificador ‘:’ ValorPropiedad
ValorPropiedad ::= Numero‘px’
                |  Numero‘em’
                |  Numero‘pt’
                |  Numero‘%’
                |  PalabraReservada
                |  Cadena
                |  Lista ValorPropiedad*
Identificador ::= AlphaNum+
AlphaNum ::= Alpha
          |  Num
Alpha ::= ‘a’ | ‘b’ | ‘c’ | .. | ‘z’
        | ‘A’ | ‘B’ | ‘C’ | .. | ‘Z’
Num ::= ‘0’ | ‘1’ | .. | ‘9’
Cadena ::= ‘’’ ContenidoCadena1 ‘’’
         | ‘"’ ContenidoCadena2 ‘"’
ContentidoCadena1 ::= ... Cualquier texto excepto ‘’’
ContentidoCadena2 ::= ... Cualquier texto excepto ‘"’
```

Descripción 12 *Sintaxis Concreta* para CSS (Fuente: Elaboración propia)

La sintaxis de la Descripción 12 corresponde al núcleo de *CSS*. La parte que no está especificada en esta sintaxis es la parte de la *Declaracion*. Porque cada propiedad de *CSS* tiene su propia sintaxis para el valor de la propiedad. Se tratará esta parte con más detalle en el Capítulo 5. Sin embargo, la estructura general de una *Declaracion* no cambia.

3.3.2. Sintaxis Abstracta para *CSS*

La *Sintaxis Abstracta* para CSS se deriva casi directamente de la *Sintaxis Concreta* de la Descripción 12. En esta sección, se presentará, parte por parte, los tipos de datos para la

sintaxis abstracta de CSS.

Viendo la *Sintaxis Concreta*, se puede decir sencillamente que una *Hoja de Estilo* es una lista de *Reglas* y que cada *Regla* tiene *Selectores y Declaraciones*.

Sin embargo, la especificación de CSS define el modelo en Cascada (Sección 2.3.1, página 9) a través de 3 tipos y 3 orígenes de las hojas de estilo. De manera que una *Regla* tiene un tipo y origen determinado.

Entonces, se inicia definiendo el *Tipo* y *Origen* en la parte de *Regla*, porque un documento estructurado puede tener una *Hoja de Estilo* que está compuesto de varios tipos y orígenes.

```
type HojaEstilo = [Regla]
type Regla = (Tipo, Origen, Selectores, Declaraciones)
data Tipo = HojaExterna | HojaInterna | EstiloAtributo
  deriving Show
data Origen = UserAgent | User | Author
  deriving Show
```

Código Haskell 5: *Sintaxis Abstracta* para *CSS*, regla, tipo y origen

Por otro lado, también se tiene la lista de selectores, donde cada *Selector* puede ser *Simple* o *Compuesto*. Así un *Selector Compuesto* es una lista de 2 o más selectores simples separados por un operador (ese operador es almacenado en un tipo de dato *String* de Haskell).

Si es un *Selector Simple*, puede ser un Selector Tipo (*TypeSelector*), que tiene *Nombre*, *Atributos* y *PseudoSelector*, o también puede ser un Selector Universal (*UnivSelector*) que también puede tener *Atributos* o un *PseudoSelector*, pero no puede tener *Nombre*.

```
type Selectores = [Selector]
data Selector
   = SimpSelector  SSelector
   | CompSelector SSelector Operador Selector
  deriving Show
type Operador = String
data SSelector
   = TypeSelector  Nombre Atributos Pseudo
   | UnivSelector         Atributos Pseudo
  deriving Show
type Nombre = String
```

Código Haskell 6: *Sintaxis Abstracta* para *CSS*, selectores

Para la parte de los atributos, *CSS* dice que el atributo `.nombreClase` corresponde a un atributo tipo operador `[class ~= "nombreClase"]`, entonces sólo se tiene 3 tipos de atributos, estos son:

```
type Atributos = [Atributo]
data Atributo
   = AtribID        ID
   | AtribNombre Nombre
   | AtribTipoOp  Nombre TipoOp AtributoValor
   deriving Show
type ID             = String
type TipoOp         = String
type AtributoValor = String
```

Código Haskell 7: *Sintaxis Abstracta* para *CSS*, atributos

Los *Pseudo* selectores, que aparecen al final de un selector simple, se encargan de marcar a un selector como *PseudoSelector*.

Para su representación se utiliza el tipo *Maybe* de Haskell, el cual guarda la información en el constructor *Just* y utiliza *Nothing* para referirse a la no existencia de la información.

```
type Pseudo = Maybe PseudoElemento
data PseudoElemento
   = PseudoBefore
   | PseudoAfter
   deriving Show
```

Código Haskell 8: *Sintaxis Abstracta* para *CSS*, pseudo selectores

Y por último las *Declaraciones*, cada declaración es simplemente una declaración de propiedad de *CSS*. Entonces las declaraciones son simplemente una lista declaración con nombre, valor e importancia.

Cada *Declaracion* tiene un *Nombre* de propiedad (*String*), un *Valor* asignado y un nivel de *Importancia* (*Bool*).

```
type Declaraciones = [Declaracion]
data Declaracion
   = Declaracion Nombre Valor Importancia
   deriving Show
type Nombre      = String
type Importancia = Bool
data Valor
   = NumeroPixel  Float
   | NumeroPoint Float
   | NumeroEm    Float
   | Porcentage    Float
   | ValorClave    String
   | ValorString   String
   | NoEspecificado
   deriving Show
```

Código Haskell 9: *Sintaxis Abstracta* para *CSS*, declaraciones

Así como se mencionó anteriormente, esta sintaxis corresponde a la parte central de *CSS*. La otra parte que no está detallado aquí son las más de 80 propiedades de CSS.

Cada propiedad de CSS define su propia sintaxis para asignar su valor, de manera que, cada sintaxis de una propiedad define sus propias palabras claves, números y valores. La forma en que se representa esta parte, es definiendo valores genéricos para cada propiedad.

Por ejemplo, la especificación de CSS define la propiedad **font-size** de la siguiente manera:

```
'font-size'
        Value           : <absolute-size> |  <relative-size> | <length>
                        | <percentage>    |  inherit
        Initial         : medium
        Applies to      : all elements
        Inherited       : yes
        Percentages     : refer to inherited font size
        Media           : visual
        Computed value  : absolute length

<absolute-size> ::=  xx-small   |   x-small | small | medium | large
                   | x-large   |   xx-large

<relative-size>  ::= larger | smaller
<length>         ::= pixel  | point | em
<percentage>     ::= Number%
<pixel>          ::= Number‘px’
<point>          ::= Number‘pt’
<em>             ::= Number‘em’
```

Descripción 13 Propiedad font-size de CSS (Fuente: Especificación de la propiedad font-size de CSS)

Ésta propiedad indica que el valor para ‘font-size’ puede ser *absoluto*, *relativo*, *length*, *porcentage* o la palabra clave ‘inherit’. Además, *absoluto* y *relativo* definen su propio conjunto de palabras clave. También se tiene que los *porcentajes* y números *length* pueden ser *Pixel*, *Point* o *Em*.

La forma de representar los valores de la propiedad ‘font-size’ es:

- se utiliza el constructor **PalabraClave String** para todas las palabras claves de la sintaxis (*inherit, small, medium, large, larger, smaller*, etc)
- se utiliza el constructor **Porcentage Float** para representar todos los número porcentaje.
- y para los otros números se utiliza el constructor **NumeroPixel Float, NumeroPoint Float, NumeroEm Float**.

La sintaxis de todas la propiedades se describirán o implementarán a nivel del parser para cada propiedad. Se hablará más de este tema en los capítulos posteriores.

Capítulo 4

Parser para *Lenguaje de Marcado*

En el anterior capítulo se mencionó que la primera tarea era entender y representar la entrada para el Navegador Web. Como resultado, se definió la sintaxis concreta y abstracta para un *Lenguaje de Marcado* (Sección 3.2, página 18), lo siguiente es reconocer un ejemplo particular descrito por la sintaxis concreta y representarlo con la sintaxis abstracta. En otras palabras, se necesita un parser que reconozca la sintaxis concreta para el lenguaje de marcado y genere como resultado la sintaxis abstracta del mismo.

No sería buena idea obligar al usuario a escribir ejemplos particulares de la estructura *Rosadelfa* directamente en *Haskell*. Si ése fuera el caso, no se tendría la necesidad de escribir un parser para la estructura *Rosadelfa*. Sin embargo, cuando el usuario escribe un documento con algún *Lenguaje de Marcado* ni siquiera se da cuenta, ni le interesa, que se está usando una estructura *Rosadelfa*. Peor aún, el usuario es capaz de escribir cualquier cosa, pero menos algo que se sujete a un *Lenguaje de Marcado*.
En estos casos la librería *uu-parsinglib*, que se utiliza para desarrollar el parser, ayuda a construir un parser robusto, en el sentido de que la librería es capaz de aplicar correcciones en la entrada, de manera que siempre se obtenga entradas correctas.

En este capítulo se mostrará el desarrollo de un *Parser* que reconozca la sintaxis concreta de un *Lenguaje de Marcado* genérico y genere como resultado la sintaxis abstracta para el *Lenguaje de Marcado*.
La sintaxis concreta y abstracta que se utilizará para el parser está descrita en Sección 3.2, página 18 del Capítulo 3.

Se utilizará la librería *uu-parsinglib* (versión 2.5.5) como herramienta para "parsear" la entrada. También se utilizará el módulo *CombinadoresBasicos* del Apéndice A.

En las siguientes secciones, se comenzará definiendo combinadores elementales para un *Lenguaje de Marcado* y sucesivamente se desarrollará combinadores que permitan reconocer partes de la estructura, para luego reconocer todo un *Lenguaje de Marcado*.

4.1. Combinadores elementales

4.1.1. Parser para las Marcas o Etiquetas

Se inicia esta sección escribiendo un parser para uno de los principales elementos de un *Lenguaje de Marcado*: las marcas o etiquetas.

```
<html>
    <head>
        <title> Ejemplo 1: El Saludo </title>
    </head>
    <body>
        <p name="saludo"> Hola Mundo Cruel. </p>
        <img src="saludo.png" />
    </body>
</html>
```

Descripción 14 Ejemplo de HTML

En la Descripción 14 se puede distinguir 3 tipos de etiquetas: etiqueta de inicio `<html>`, etiqueta de fin `</body>` y una etiqueta especial `<img/>`.

Utilizando el módulo de *CombinadoresBasicos* del Apéndice A, el Código Haskell 10 define un parser para cada una de las etiquetas:

$$
\begin{aligned}
&pTagInicio :: Parser\ String \\
&pTagInicio = pSimbolo\ \texttt{"<"} *> pPalabra <* pSimbolo\ \texttt{">"} \\
&pTagFin :: Parser\ String \\
&pTagFin = pSimbolo\ \texttt{"</"} *> pPalabra <* pSimbolo\ \texttt{">"} \\
&pTagEspecial :: Parser\ String \\
&pTagEspecial = pSimbolo\ \texttt{"<"} *> pPalabra <* pSimbolo\ \texttt{"/>"}
\end{aligned}
$$

Código Haskell 10: Parser para Etiquetas, versión 1

Seguidamente, se muestra algunos ejemplos para probar el parser para las etiquetas de inicio, fin y especiales:

```
*Parser> parseString pTagInicio "<html>"
"html"
*Parser> parseString pTagInicio "<h1>"
"h1"
*Parser> parseString pTagFin "</body>"
"body"
*Parser> parseString pTagEspecial "<img/>"
"img"
*Parser> parseString pTagInicio "< div >"
-- >    Deleted  ' ' at position (0,1) expecting isAlphaNum
-- >    Deleted  ' ' at position (0,5) expecting one of [isAlphaNum, '>']
"div"
```

Descripción 15 Ejemplos de prueba para el parser de etiquetas

Como se puede ver, un error inesperado ocurrió en el último ejemplo de prueba. Aunque la librería *uu-parsinglib* hizo las correcciones correspondientes para devolver un resultado correcto, es algo que no debe ocurrir. Estos errores se deben a que no se está considerando la existencia de espacios entre los símbolos y el nombre de la etiqueta.

Entonces, para corregir el error, se puede decir que una etiqueta puede tener espacios en los lados internos de los símbolos delimitadores. Así, se modifica el Código Haskell 10 de la siguiente manera:

```
pTagInicio :: Parser String
pTagInicio = pSimboloDer "<" *> pPalabra <* pSimboloIzq ">"
pTagFin :: Parser String
pTagFin = pSimbolo "<" *> pSimboloAmb "/" *> pPalabra <* pSimboloIzq ">"
pTagEspecial :: Parser String
pTagEspecial = pSimboloDer "<" *> pPalabra <* pSimboloAmb "/" <* pSimbolo ">"
```

Código Haskell 11: Parser para Etiquetas en Haskell, versión 2

4.1.2. Parser para el Texto

Otro de los elementos importantes de un lenguaje de marcado es el texto. El texto puede estar entre las etiquetas de inicio y fin.

En un primer intento es posible decir que el parser podría reconocer sencillamente una lista de uno o más caracteres alfanuméricos. Pero no se estaría considerando los espacios y saltos de línea.

Luego se podría decir que no solo son los espacios y saltos de línea, sino que también pueden ser otros símbolos como `+(),._ etc`. Pero la lista de símbolos a reconocer puede seguir creciendo.

Sin embargo, hay una excepción, el conjunto de símbolos no puede incluir los símbolos `</>`, porque están reservados para construir una nueva etiqueta. Entonces, el conjunto válido de símbolos es cualquier símbolo diferente a: `</>`.

Utilizando el módulo de *CombinadoresBasicos* del Apéndice A, su implementación podría ser así:

```
pTexto :: Parser String
pTexto = pTextoRestringido "</>"
```

Código Haskell 12: Parser para un texto

Y finalmente se muestra una prueba para reconocer un texto cualquiera:

```
*Parser> parseString pTexto "Ejemplo 1: El Saludo\n"
"Ejemplo 1: El Saludo\n"
```

Descripción 16 Ejemplo de prueba para el parser de un texto

4.2. Atributos de un *Lenguaje de Marcado*

Continuando con la implementación de parsers, lo siguiente es implementar los atributos que la etiqueta del lenguaje de marcado puede tener.

Recuerde que la estructura para guardar los atributos es:

```
type Atributos = [(Nombre, Valor)]
type Nombre    = String
type Valor     = String
```

Código Haskell 13: Tipos de datos para guardar los Atributos de una etiqueta

Según la *Sintaxis Concreta*, un atributo es simplemente un identificador seguido de un símbolo '=' y una cadena delimitada por comillas dobles.

Y una lista de atributos son los mismos atributos separados por caracteres especiales tales como *salto de línea, tabulador y espacio.*

El Código Haskell 14 muestra la implementación para los atributos:

```
pAtributo :: Parser (String, String)
pAtributo = (,) <$> pPalabra <* pSimboloAmb "=" <*> pValor

pValor :: Parser String
pValor = pDeLimitadoCon (pSym '"') (pTextoRestringido "\"")

pAtributos :: Parser [(String, String)]
pAtributos = pListSep_ng pInutil1 pAtributo
```

Código Haskell 14: Parser para atributos

4.3. Parser para la Estructura *Rosadelfa*

Hasta esta parte ya se sabe reconocer elementos básicos de la estructura *Rosadelfa*, lo siguiente es construir el parser para toda la estructura *Rosadelfa*.

Para simplificar el desarrollo de esta parte, se definirá una nueva estructura *Rosadelfa* más sencilla que no incluya atributos.

Los tipos de datos para la nueva estructura *Rosadelfa* simple (sin atributos) son:

```
data SRosa = SimpleRosa SNodo [SRosa]
  deriving Show

data SNodo
  = SimpleTag String
  | SimpleTexto String
  deriving Show
```

Código Haskell 15: Tipos de datos para la estructura *Rosadelfa* simple

La única restricción para escribir el parser del Código Haskell 15 es que el nombre de la etiqueta de inicio debe ser el mismo que el nombre de la etiqueta final. Imagínese si fueran

nombres diferentes, no se sabría con que nombre identificar la etiqueta. Pero si los nombres son diferentes, se puede devolver un error o hacer algo al respecto.

También se debe considerar que existe 3 formas de crear una estructura *Rosadelfa*, la más sencilla es que puede ser simplemente texto. La otra es que puede ser una etiqueta especial sin ramificaciones, como el *img*. Y el último puede ser una etiqueta normal (inicio y fin) y con ramificaciones que pueden volver a ser cualquiera de las 3 opciones.

A continuación se muestra la implementación de la primera versión:

```
pSimpleRosa :: Parser SRosa
pSimpleRosa
    =  rosaTag <$> pTagInicio
                   <*> pList_ng pSimpleRosa <*>
               pTagFin
  <|> espeTag <$> pTagEspecial
  <|> rosaText <$> pTexto
rosaTag :: String → [SRosa] → String → SRosa
rosaTag tinicio tags tfin
    =  if tinicio ≡ tfin
       then SimpleRosa (SimpleTag tinicio) tags
       else error "Nombres diferentes."
espeTag :: String → SRosa
espeTag tag = rosaTag tag [] tag
rosaText :: String → SRosa
rosaText str = SimpleRosa (SimpleTexto str) []
```

Código Haskell 16: Parser para la estructura Rosadelfa simple

También se muestra algunos ejemplos de prueba:

```
*Parser> parseString pSimpleRosa "Ejemplo 1: El Saludo\n"
SimpleRosa (SimpleTexto "Ejemplo 1: El Saludo\n") []
*Parser> parseString pSimpleRosa "< img />"
SimpleRosa (SimpleTag "img") []
*Parser> parseString pSimpleRosa "<p><img /> Esto es un texto </p>"
SimpleRosa (SimpleTag "p") [SimpleRosa (SimpleTag "img") []
                           ,SimpleRosa (SimpleTexto " Esto es un texto ") []
                           ]
```

Descripción 17 Ejemplos de prueba del parser para la estructura Rosadelfa simple

Hasta aquí todo parece estar bien, pero solo hasta que se ve el siguiente error:

```
*Parser> parseString pSimpleRosa "<head> cabeza </jead>"
*** Exception: Nombres diferentes.
```

Descripción 18 Error generado cuando las etiquetas son diferentes

El resultado de la Descripción 18 es correcto, pero no el deseado. Porque al principio del capítulo se mencionó que la librería *uu-parsinglib* hace correcciones. Pero el anterior código no

muestra ninguna corrección. En realidad la librería no tiene nada que corregir, porque quien genera la excepción es el código que se ha escrito y no el de la librería. Sin embargo, no es el resultado deseado.

Algo que se puede hacer en la función *tagRosa* del Código Haskell 16 es que si las etiquetas son diferentes, se podría devolver un tipo de dato *SRosa* con la primera etiqueta, pero obviando la segunda:

```
rosaTag2 :: String → [SRosa] → String → SRosa
rosaTag2 tinicio tags tfin
  = if tinicio ≡ tfin
    then SimpleRosa (SimpleTag tinicio) tags
    else SimpleRosa (SimpleTag tinicio) tags
```

Código Haskell 17: Segunda versión para la función *tagRosa*

Haciendo la última corrección en el código, ya de nada sirve hacer una comparación entre etiquetas, porque su código es el mismo. Entonces, esta solución no es buena, aunque es válida.

Sería interesante si se pudiera hacerle responsable a la librería *uu-parsinglib* de la corrección de la entrada. Pero antes se necesita entender mejor la forma en que la librería *uu-parsinglib* hace las correcciones.

4.3.1. Las correcciones que realiza la librería *uu-parsinglib*

Para entender la forma en que la librería las correcciones, considere el siguiente *parser* que reconoce caracteres dígitos:

```
pDigito :: Parser Char
pDigito = pSym (isDigit, "isDigit", '0')
```

A continuación se muestra algunos ejemplo de prueba para el *parser* de dígitos:

```
*Parser> parseString pDigito "1"
'1'
*Parser> parseString pDigito "2"
'2'
*Parser> parseString pDigito "0"
'0'
*Parser> parseString pDigito "a"
-- >    Deleted  'a' at position (0,0) expecting isDigit
-- >    Inserted '0' at position (0,1) expecting isDigit
'0'
```

Descripción 19 Ejemplos de prueba para el parser de dígitos

El primer parámetro de *pSym* es una función booleana[1] que determina que carácter será aceptado, el segundo y tercer parámetro están relacionados con la corrección de erro-

[1]Que devuelve un resultado de tipo *Bool* (*False*, *True*)

res. Por ejemplo, la última prueba a *pDigito*, indica (primer mensaje) que se borró el carácter 'a' (`Deleted 'a'`) porque no se logro satisfacer la función *isDigit*. En la segunda, se indica que se insertó el carácter '0' (`Inserted '0'`) porque es el carácter por defecto del parser.

Entonces, el segundo parámetro determina el nombre de lo que se está esperando (String) y el tercero es el carácter por defecto que se insertará en caso de errores.

Por ejemplo, si se cambia el parser para $pDigito = pSym$ (*isDigit*, `"digito"`, `'9'`), entonces se insertará un 9 en vez de un 0 y el mensaje será "digito" y no "isDigit":

```
*Parser> parseString pDigito "a"
-- >    Deleted  'a' at position (0,0) expecting digito
-- >    Inserted '9' at position (0,1) expecting digito
'9'
```

Descripción 20 Ejemplo de prueba para el parser de dígitos

Entonces, las correcciones de la librería *uu-parsinglib* se realizan mediante las funciones que determinan los valores que se está esperando en la entrada. Así, si la entrada no es correcta, se inserta o elimina valores para que sea correcta.

Existen varias formas de reconocer un carácter simple, por ejemplo:

$pa' = pSym\ (\equiv \texttt{'a'}, \texttt{"letra a"}, \texttt{'a'})$
$pa\ = pSym\ \texttt{'a'}$

En la última definición, ambas versiones hacen lo mismo. De la misma forma, si se quiere reconocer una lista de caracteres, se puede definir el combinador *pToken*:

$pal = pToken\ \texttt{"carlos"}$
$pToken\ [\,] \quad = pReturn\ [\,]$
$pToken\ (a : as) = (:) <\$> pSym\ a <*> pToken\ as$

El combinador *pToken*, la cual no es necesario redefinirla porque se encuentra dentro de la librería, reconoce una lista de símbolos con el combinador *pSym*.

Entonces, la función *pSym* es una de las funciones principales de la librería. Se puede decir que las demás funciones de la librería trabajan en base a *pSym*.

Ahora, todo esto tiene sentido cuando se crea un *Parser* que reconozca un conjunto de caracteres, por ejemplo el nombre de una etiqueta. De manera que la librería se encargue de reconocer el nombre de una etiqueta y si no lo encuentra, aplique correcciones en la entrada. Esta función puede ser definida de la siguiente manera:

$pNombreTag :: String \to Parser\ String$
$pNombreTag = pToken$

Código Haskell 18: Parser para el nombre de una etiqueta en Haskell

A continuación se muestra un ejemplo de prueba para el parser del Código Haskell 18:

```
*Parser> parseString (pNombreTag "html") "html"
"html"
*Parser> parseString (pNombreTag "html") "body"
-- >    Deleted  'b' at position (0,0) expecting "html"
-- >    Deleted  'o' at position (0,1) expecting "html"
-- >    Deleted  'd' at position (0,2) expecting "html"
-- >    Deleted  'y' at position (0,3) expecting "html"
-- >    Inserted "html" at position (0,4) expecting "html"
"html"
```

Descripción 21 Ejemplo de prueba para el parser *pNombreTag*

Entonces, si se envía a *pNombreTag* `"html"` una entrada incorrecta, la librería realizará las correcciones necesarias hasta obtener lo que se desea.

Con todo esto, es posible decirle a la librería que se está esperando ciertos nombres de etiquetas y si no los encuentra que aplique correcciones. Pero para aplicar la idea anterior, se necesita conocer al menos el nombre de la etiqueta que se desea esperar. Es decir, se necesita obtener el nombre de la etiqueta de inicio, para luego decirle a la librería que se está esperando una etiqueta final con el mismo nombre.

Se necesita algo como lo siguiente:

```
pSimpleRosa
    =  let nombreTagInicio = pTagInicio2
           ramificaciones  = pList_ng pSimpleRosa
           nombreTagFinal  = pNombreTag nombreTagInicio
       in rosaTag nombreTagInicio ramificaciones nombreTagFinal
```

Descripción 22 Ejemplo Ideal para el parser de elementos

El código que se acaba de mostrar es incorrecto porque no se tiene acceso a la información de parsing intermedia. Pero la librería *uu-parsinglib* provee la interfaz monádica para resolver este problema.

4.3.2. Interfaz Monádica de *uu-parsinglib*

Una extensión interesante que permite tener éxito con el anterior problema es utilizar la interfaz monádica de *uu-parsinglib*.

La interfaz monádica permite acceder a la información que se está revisando en el proceso. De manera que es posible utilizar esta información para redirigir el proceso de parsing.

Por ejemplo, si se quiere construir una pequeña función para construir etiquetas para un lenguaje de marcado, primeramente se define una función que recibe el nombre de una etiqueta y su cuerpo, luego se podría usar la interfaz monádica de Haskell para guardar el nombre de la etiqueta inicio y fin para usarlos luego en la construcción de la etiqueta. Su implementación podría ser de la siguiente manera:

```
crearITag :: String → IO String
crearITag tag = return ("<" ++ tag ++ ">")
crearFTag :: String → IO String
crearFTag tag = return ("</" ++ tag ++ ">")
tagged :: String → String → IO String
tagged tag cuerpo = do tag1 ← crearITag tag
                       tag2 ← crearFTag tag
                       return (tag1 ++ cuerpo ++ tag2)
```

Código Haskell 19: Ejemplo simple utilizando mónadas

Vea que la notación **do** se parece a la notación **let in** de *Haskell*, básicamente, porque se está creando variables dentro la notación **do**, las cuales son utilizadas posteriormente para construir una etiqueta.

Se puede utilizar la misma idea para construir el parser para etiquetas. Por ejemplo, se puede usar variables para almacenar el resultado de un parser y luego usarlas para redirigir el proceso de parsing.

A continuación se muestra la implementación de parser para etiquetas utilizando la interfaz monádica de *uu-parsinglib*:

```
pFinalTag :: String → Parser String
pFinalTag tag = pSimbolo "<" *> pSimboloAmb '/' *> pToken tag <* pSimboloIzq '>'
pMTag :: Parser SRosa
pMTag  =  do itag  ← pTagInicio2
             rami ← pList_ng pMTag
             ftag  ← pFinalTag itag
             return (rosaTag itag rami ftag)
      <|> rosaText <$> pTexto
```

Código Haskell 20: Ejemplo con Interfaz Monádica

En el Código Haskell 20 se ha definido el combinador *pFinalTag* que reconoce una determinada etiqueta (*String*) con el parámetro que recibe. Este combinador es el que se encarga de corregir la entrada en caso de errores.

También se tiene la función *pMTag* que hace uso de la interfaz monádica, el cual crea una variable para guardar la etiqueta de inicio y enviarlo a *pFinalTag* para construir la estructura final correcta.

4.4. Parser final para la estructura *Rosadelfa* y sus optimizaciones

Antes de mostrar la versión final del *Parser Rosadelfa* que se ha desarrollado, se corregirá un error que se produce en el parser del Código Haskell 20, lo cual puede considerarse como una optimización al parser que se ha definido.

Si se añade soporte para etiquetas especiales al parser del Código Haskell 20, entonces se tendría:

```
pMTag2 :: Parser SRosa
pMTag2   =   do itag  ← pTagInicio2
                rami ← pList_ng pMTag
                ftag  ← pFinalTag itag
                return (rosaTag itag rami ftag)
           <|> espeTag  <$> pTagEspecial2
           <|> rosaText <$> pTexto
```

Código Haskell 21: Ejemplo con Interfaz Monádica y etiquetas especiales

Luego, si se prueba el parser del Código Haskell 21 con un ejemplo, entonces se produce el siguiente error:

```
*Parser> parseString pMTag2 "<html> hola </html>"
*** Exception: cannot compute minimal length of right hand side of monadic parser
```

Descripción 23 Error que produce el parser del Código Haskell 21

Para corregir el error de la Descripción 23 se ha convenido en optimizar la definición del parser con una factorización de las partes comunes de *pTagInicio2* y *pTagEspecial2*.

4.4.1. Optimizaciones

Hay una pequeña optimización que se puede hacer al código. Fíjese en detalle lo que las siguientes definiciones de parsers tienen en común:

```
pTagInicio :: Parser String
pTagInicio    = pSimboloDer "<" *> pPalabra                          <* pSimboloIzq ">"
pTagEspecial :: Parser String
pTagEspecial = pSimboloDer "<" *> pPalabra <* pSimboloAmb "/" <* pSimbolo     ">"
```

Código Haskell 22: Parser semi comunes

Se puede decir que ambos parsers tienen algo en común, básicamente esto es:
pSimboloDer `"<"` $*>$ *pPalabra*. No es buena idea permitir repetir libremente lo que tiene en común, porque podría ser la causa del error ya que habría más consumo de memoria y procesamiento innecesario.

Por lo tanto, se podría optimizar lo siguiente:

```
pSimpleRosa :: Parser SRosa
pSimpleRosa
   =     rosaTag <$> pTagInicio
                       <*> pList_ng pSimpleRosa <*>
                     pTagFin
   <|> espeTag <$> pTagEspecial
   <|> rosaText <$> pTexto
```

Código Haskell 23: Parser Rosadelfa con 3 alternativas

Fíjese que se tiene 3 alternativas. Si se separa la versión común, se puede obtener una nueva versión con sólo 2 alternativas:

```
pSimpleRosa2 :: Parser SRosa
pSimpleRosa2
   =     rosaTag2   <$> pComunTagInicio <*> pRestoTagInicio
   <|> rosaText   <$> pTexto

pComunTagInicio :: Parser String
pComunTagInicio = pSimboloDer "<" *> pPalabra

pRestoTagInicio :: Parser [SRosa]
pRestoTagInicio = id <$> pList_ng pSimpleRosa <* pTagFin
   <|>              []   <$  pSimboloAmb "/" *> pSimbolo ">"
```

Código Haskell 24: Parser Rosadelfa con 2 alternativas

Otra forma de optimizar podría ser haciendo uso de la interfaz monádica. Esta forma es la que se utilizó en la versión final del *Parser Rosadelfa*.

4.4.2. Versión Final

En está versión se añadirá la parte de los atributos y se hará la versión monádica de la optimización.

Primeramente se define las funciones constructoras de estructuras *Rosadelfa*. Una para el texto y otra para las etiquetas (que recibe el nombre de la etiqueta, sus atributos y el cuerpo de la etiqueta).

```
rosaTexto :: String → ArbolRosa
rosaTexto str = ArbolRosa (NTexto str) []

rosaTagged :: String → [(String, String)] → [ArbolRosa] → ArbolRosa
rosaTagged nm ats rms = ArbolRosa (NTag nm ats) rms
```

Código Haskell 25: Parser para *Rosadelfa*, funciones constructoras

Un *Parser* para la estructura *Rosadelfa* es básicamente un texto o un *tagged* (una etiqueta normal o una etiqueta especial):

```
pRosadelfa :: Parser ArbolRosa
pRosadelfa  =     pRosaTagged
            <|> pRosaTexto

pRosaTexto :: Parser ArbolRosa
pRosaTexto = rosaTexto <$> pTexto
```

Código Haskell 26: Parser para *Rosadelfa*, elementos básicos

Para implementar la versión optimizada, se debe separar la etiqueta de inicio en parte común y la parte restante. Luego se debe preguntar a la parte restante, si es una etiqueta especial o una etiqueta normal y se continua reconociendo la parte que falta. Si la parte restante es una etiqueta especial, entonces significa que no tiene cuerpo y se devuelve una lista vacía, caso contrario se debe procesar el cuerpo seguido de la etiqueta final.

```
pRosaTagged :: Parser ArbolRosa
pRosaTagged
   = do (itag, mp) ← pComunTag
        bool       ← pRestoTagInicio
        ramif      ← if bool
                     then return []
                     else pList_ng pRosadelfa <* pFinalTag itag
        return (rosaTagged itag mp ramif)
pComunTag :: Parser (String, [(String, String)])
pComunTag = (,) <$> pSimboloDer '<' <*> pPalabra2 <* pInutil <*> pAtributos
pRestoTagInicio :: Parser Bool
pRestoTagInicio  =  False <$ pSimboloIzq '>'
                 <|> True <$ pSimbolo '/' <* pSym '>'
pFinalTag :: String → Parser String
pFinalTag tag = pSym '<' *> pSimbolo '/' *> pToken tag <* pSimboloIzq '>'
```

Código Haskell 27: Parser para *Rosadelfa*, versión monádica

Note que se esta integrando el combinador para atributos en la parte común de la etiqueta.

Capítulo 5

Parser para *Cascading Style Sheets (CSS)*

Al igual que para el capítulo del Parser para HTML (Capítulo 4) también se necesita desarrollar una parser para el lenguaje de hojas de estilos CSS. Este parser debe reconocer un ejemplo particular de la sintaxis concreta de CSS y generar una sintaxis abstracta del lenguaje de hojas de estilos CSS.

Entonces, en este capítulo se desarrollará un *Parser* que reconozca la sintaxis concreta del lenguaje de hojas de estilo CSS y genere como resultado la sintaxis abstracta del mismo lenguaje. La sintaxis concreta y abstracta (que se utilizará para el parser de CSS) está descrita en la Sección 3.3, página 21 del Capítulo 3.

Se utilizará la librería *uu-parsinglib* (versión 2.5.5) como herramienta para "parsear" la entrada. También se utilizará el módulo *CombinadoresBasicos* del Apéndice A.

En las siguientes secciones del capítulo se desarrollará el parser de manera acumulativa, se inicia desarrollando el parser para las reglas de estilo, luego para los distintos tipos de selectores y finalmente para las declaraciones de estilos.

5.1. Combinadores para *Hojas de Estilo* y Reglas

Se inicia esta sección escribiendo los combinadores para las *Hojas de Estilo*.

Las *Hojas de Estilo* son sencillamente una lista de Reglas que pueden pertenecer a un determinado tipo y origen. Pero, no existe una sintaxis para reconocer el tipo y origen.

La información de tipo y origen es determinada de acuerdo al contexto en que es utilizado. Por ejemplo, si se quiere definir una *Hoja de estilos* con el usuario (origen) *User*, el tipo de la hoja de estilos siempre será *Externo*, porque no hay forma de definir una hoja de estilos *Interna* o de *Atributos*. Este es el mismo caso cuando se quiere definir hojas de estilo con el usuario (origen) *UserAgent*.

Entonces, el único usuario que puede definir hojas de estilo en cualquiera de los 3 tipos es *Author*. Más adelante (Sección 5.4, página 46) se mostrará las funciones que permitan definir las hojas de estilo de acuerdo al contexto.

Por ahora, simplemente se necesita saber que para reconocer una hoja de estilo se debe recibir 2 parámetros: tipo y origen.

```
pHojaEstilo :: Tipo → Origen → Parser HojaEstilo
pHojaEstilo tp org = pList (pRegla tp org)
```

Las hojas de estilo son simplemente una lista de reglas, donde los parámetros que recibe para su parser no le son importantes, de manera que son nuevamente enviados, como argumentos, al parser para reconocer una regla.

Entonces el parser para reconocer una *Regla* recibe 2 parámetros. Además, su parser está compuesto por selectores y seguido de declaraciones, las cuales están agrupadas entre llaves. La forma en que se guarda una *Regla*, según la *Sintaxis Abstracta*, es usando una tupla de 4: el tipo, origen, selectores y declaraciones.

```
pRegla :: Tipo → Origen → Parser Regla
pRegla tp org = (,,,) tp org <$> pSelectores <* pSimboloAmb "{"
                                             <*> pDeclaraciones
                                             <* pSimboloAmb "}"
```

5.2. Combinadores para Selectores

Los Selectores son una lista de uno o más Selectores separados por comas.

```
pSelectores = pList1Sep_ng (pSimboloAmb ",") pSelector
```

Para implementar el parser para *Selector*, primeramente se realizara el parser/combinador para los atributos y Pseudo-Selectores.

5.2.1. Combinadores para atributos

La sintaxis concreta para los atributos (Descripción 12) define 4 tipos de atributos: Atributo ID, atributo clase, atributo nombre y atributo operador. Pero en la sintaxis abstracta (Código Haskell 7) sólo define los tipo de datos para 3 de ellos. Esto es así, porque el atributo clase es representado a través del tipo de dato para del atributo operador, donde su nombre es 'class' y su operador es '$\sim=$'.

A continuación se muestra el parser para los atributos:

```
pAtributo :: Parser Atributo
pAtributo  =  AtribID                          <$ pSimbolo "#" <* > pPalabra
          <|> AtribNombre                      <$ pSimboloDer "["
                                                  <* > pPalabra
                                               <* pSimboloIzq "]"
          <|> AtribTipoOp "class" " ~= "       <$ pSimbolo "." <* > pPalabra
          <|> AtribTipoOp                      <$ pSimboloDer "["
                                                  <* > pPalabra
                                                  <* > pTipoOp
                                                  <* > pSimpleString
                                               <* pSimboloIzq "]"
pTipoOp :: Parser String
pTipoOp = pSimboloAmb "=" <|> pSimboloAmb " ~= "
```

5.2.2. Combinadores para pseudo-selectores

Los pseudo-selectores son opcionales y pueden estar declarados al final de un selector simple. Pero existe un caso especial donde el pseudo-selector no puede ser opcional, esto es cuando todos los demás campos de un selector simple no están presentes, si eso ocurre, el pseudo-selector no puede ser opcional.

Para su implementación se construye dos versiones de un pseudo-selector. Uno que es opcional y otro que no puede ser opcional.

```
pMaybePseudo :: Parser Pseudo
pMaybePseudo = pMaybe pPseudoElemento

pMaybeJustPseudo :: Parser Pseudo
pMaybeJustPseudo = Just <$> pPseudoElemento

pPseudoElemento :: Parser PseudoElemento
pPseudoElemento  =  PseudoBefore <$ pSimbolo ":" <* pToken "before"
                <|> PseudoAfter  <$ pSimbolo ":" <* pToken "after"
```

El parser *pMaybePseudo* puede ser opcional y el parser *pMaybeJustPseudo* no puede ser opcional.

5.2.3. Combinadores para selector

Ahora que se tiene el parser para las partes de un selector, se definirá el combinador para un selector simple.

Un Selector Simple puede ser de dos tipos: *TypeSelector* que tiene un nombre o *UnivSelector* identificado por el caracter '∗' el cual no tiene nombre. En ambos casos se puede tener una lista de atributos y un pseudo-selector opcional.

El Selector Universal es un caso especial, porque puede tener directamente una lista de atributos y un pseudo-selector opcional. O también puede no tener ninguna lista de atributos, pero si o si un pseudo-selector no opcional:

```
pSSelector :: Parser SSelector
pSSelector
    =  TypeSelector   <$> pPalabra      <*> pList  pAtributo <*> pMaybePseudo
   <|> UnivSelector    <$  pSimbolo "*" <*> pList  pAtributo <*> pMaybePseudo
   <|> UnivSelector   <$> pList1 pAtributo <*> pMaybePseudo
   <|> UnivSelector []<$>                      pMaybeJustPseudo
```

Continuando con el parser para el selector, la *Sintaxis Concreta* indica que se tiene 2 tipos de selectores: uno simple y otro compuesto. El simple es el parser para *SSelector*, pero el compuesto es una lista de *SSelector* separados por operadores.

```
pSelector :: Parser Selector
pSelector  =  SimpSelector  <$> pSSelector
          <|> CompSelector <$> pSSelector <*> pOperador <*> pSelector
```

El operador del selector compuesto debe reconocer los caracteres >, + y ' '. Reconocer el caracter de espacio es un caso especial, porque en primer lugar, puede ser una lista de al menos un espacio acompañado de caracteres como: *nueva linea, retornos de carro o tabuladores*. En segundo lugar, puede no haber un espacio, pero debe al menos tener un caracter de *nueva linea, retorno de carro o tabulador*.

```
pOperador :: Parser String
pOperador = pSimboloAmb ">" <|> pSimboloAmb "+" <|> pEspacioEspecial
pEspacioEspecial :: Parser String
pEspacioEspecial
   =    " " <$ stuff  <* pList1 (pSym ' ') <* stuff
   <|> " " <$ stuff1 <* pList  (pSym ' ') <* stuff
  where stuff  = pList  (pAnySym "\t\r\n")
        stuff1 = pList1 (pAnySym "\t\r\n")
```

5.3. Combinadores para Declaraciones

Las declaraciones de CSS son una lista de declaraciones separadas por el símbolo ';', cada declaración consiste en la asignación de uno o más valores a una o más propiedades. Por ejemplo, cuando la declaración es simple, se asigna un sólo valor a una propiedad; pero cuando se trata de una propiedad *shorthand*, se asigna varios valores a varias propiedades.

Es por eso que *pDeclaracion* debe devolver una lista de *Declaraciones* y que una vez reconocidas, deben concatenarse para obtener una simple lista de declaraciones y no una lista de listas de declaraciones.

```
pDeclaraciones :: Parser Declaraciones
pDeclaraciones = concat <$> pList1Sep_ng (pSimboloAmb ";") pDeclaracion
```

Para facilitar la construcción de una *Declaracion*, se definirá la función *construirDeclaracion*, que construye una declaración simple. Esta función recibirá, como argumento, una tupla donde el primer valor será el nombre de la propiedad y el segundo valor será un parser para sus valores y devolverá una lista con un elemento (la declaración reconocida):

```
construirDeclaracion :: (String, Parser Value) → Parser Declaraciones
construirDeclaracion (nm, pValor)
    =   (λnm vl imp → [Declaracion nm vl imp])
  <$> pToken nm <* pSimboloAmb ":" <*> pValor <*> pImportancia

pImportancia :: Parser Importancia
pImportancia = (True <$ pSimboloAmb "!" <* pToken "important") `opt` False
```

Con ayuda de la función *construirDeclaracion* se puede implementar el parser para *pDeclaracion*:

```
pDeclaracion :: Parser Declaracion
pDeclaracion
  = construirDeclaracion
      ("display"
      , pValoresClave ["inline", "block", "list-item", "none", "inherit"])
  <|> ...
```

En la definición de *pDeclaracion* se debe especificar todas las declaraciones de las propiedades de CSS que se implementará en el proyecto. Sería sencillo si sólo fueran 10 o 20 propiedades, pero en realidad son más de 40 propiedades a las que se dará soporte. Dejarlo todo en un mismo módulo es voluminoso y propenso a errores, de manera que se ha optado por separarlos en distintos módulos.

El hecho de separarlos en distintos módulos implica tener una lista externa de parsers para las propiedades de CSS y que la lista externa debe recibirse como un argumento en el parser para *pDeclaraciones*. En otras palabras, implica que se debe modificar el parser desarrollado hasta ahora.

Para la nueva modificación se ha decidido crear un módulo denominado *BasicCssParser* para almacenar los combinadores comunes para los valores de propiedades CSS. También se ha creado otro módulo denominado *PropiedadesCSS* para guardar la lista de propiedades de CSS a las que se dará soporte. Los elementos de esta lista serán por ahora tuplas de 2 valores, donde el primer valor contendrá el nombre de la propiedad y el segundo será el parser que reconoce los valores para la propiedad.

Para construir una lista de declaraciones se usara la función *construirDeclaracion*:

```
lista_valor_parser :: [Parser Declaraciones]
lista_valor_parser = map construirDeclaracion cssPropiedades
```

Antes de describir los nuevos módulos mencionados, se verá como cambia el parser desarrollado hasta ahora para las declaraciones y funciones relacionadas.

El parser para *pDeclaraciones* ahora recibe una lista de declaraciones de CSS, donde cada declaración devuelve otra vez una lista de declaraciones. Lo único que queda por hacer, es combinar todos los elementos de la lista con el combinador alternativo de la libreria *uu-parsinglib* `<|>`:

```
pDeclaraciones :: [Parser Declaraciones] → Parser Declaraciones
pDeclaraciones pDeclaracion
   = concat <$> pList1Sep_ng (pSimboloAmb ";") (foldr (<|>) pFail pDeclaracion)
```

Este cambio afecta a las funciones relacionadas con *pDeclaraciones*, de manera que *pRegla* y *pHojaEstilo* deben recibir un parámetro más, la lista de declaraciones:

```
pHojaEstilo :: Tipo → Origen → [Parser Declaraciones] → Parser HojaEstilo
pHojaEstilo tp org props = pList (pRegla tp org props)

pRegla :: Tipo → Origen → [(String, Parser Valor)] → Parser Regla
pRegla tp org props   =   (,,) tp org
                      <$> pSelectores <* pSimboloAmb "{"
                                      <*> pDeclaraciones props
                                      <* pSimboloAmb "}"
```

5.3.1. Separando Propiedades *CSS*

En esta sub-sección se describirá los módulos *BasicCssParser* y *PropiedadesCSS*.

Módulo BasicCssParser

Los valores comunes que se puede encontrar en un valor de una propiedad de CSS son:

- **Valores Clave.** Lo más básico es reconocer una palabra clave. Luego se puede usar la misma función para reconocer un *ValorClave* y usarlo para reconocer una lista de valores clave.

```
pPalabraClave :: String → Parser String
pPalabraClave = pToken

pValorClave :: String → Parser Valor
pValorClave str = ValorClave <$> pPalabraClave str

pValoresClave :: [String] → Parser Valor
pValoresClave = pAny pValorClave
```

- **Valores Length.** Estos combinadores construyen números *pixel, point y em.* Se tiene dos variaciones: uno que puede ser positivo o negativo, y el otro que sólo puede ser positivo.

```
pLength      =   NumeroPixel  <$> pNumeroFloat <* pToken "px"
             <|> NumeroPoint  <$> pNumeroFloat <* pToken "pt"
             <|> NumeroEm     <$> pNumeroFloat <* pToken "em"
pLengthPos   =   NumeroPixel  <$> pNumeroFloatPos <* pToken "px"
             <|> NumeroPoint  <$> pNumeroFloatPos <* pToken "pt"
             <|> NumeroEm     <$> pNumeroFloatPos <* pToken "em"
```

- **Valores Porcentajes.** Se construye números porcentajes de dos tipos: que pueden ser positivos o negativos, o solamente positivos.

```
pPorcentage
  = Porcentage < $ > pNumeroFloat < * pSimbolo "%"
pPorcentagePos
  = Porcentage < $ > pNumeroFloatPos < * pSimbolo "%"
```

- **Colores Claves.** El valor *ColorClave* recibe una tupla de 3 enteros, que representan los valores RGB. Estos valores pueden presentarse en 3 distintas formas:

```
pColor = pColorComun < | > pColorHexadecimal < | > pColorFuncion
```

  - Los colores comunes están definidos a través de palabras claves, los cuales son reescritos en sus valores RGB correspondientes.

```
pColorComun
  =    ColorClave (0 x80, 0 x00, 0 x00) < $ pPalabraClave "maroon"
  < | > ColorClave (0 xff, 0 x00, 0 x00) < $ pPalabraClave "red"
  < | > ColorClave (0 xff, 0 xa5, 0 x00) < $ pPalabraClave "orange"
  < | > ColorClave (0 xff, 0 xff, 0 x00) < $ pPalabraClave "yellow"
   ....
```

  - Los colores hexadecimales pueden presentarse de dos formas: pueden contener 3 caracteres o 6 caracteres. Si son de 3 caracteres, cada caracter hexadecimal se repite y se lo convierte a un entero para formar los valores RGB. Pero si son de 6 caracteres, se agrupa cada 2 caracteres para convertirlos a un número entero para los valores RGB.

    El combinador *pSimpleHex* se usa cuando son de 3 caracteres y *pDoubleHex* cuando son de 6 caracteres.

```
pSimpleHex = (λh → "0x" ++ [h, h]) < $ > pHex
pDoubleHex = (λh1 h2 → "0x" ++ [h1, h2]) < $ > pHex < * > pHex
```

    Las dos formas de color hexadecimal pueden ser implementadas se la siguiente manera:

```
pColorHexadecimal
    =  (λr g b → ColorClave (r, g, b))
          < $ pSimbolo "#" < * > pSimpleHex < * > pSimpleHex < * > pSimpleHex
  < | > (λr g b → ColorClave (r, g, b))
          < $ pSimbolo "#" < * > pDoubleHex < * > pDoubleHex < * > pDoubleHex
```

  - Por último, color función que tiene la siguiente forma *rgb* (r, g, b), donde los valores RGB pueden ser números o porcentajes. Si es un número, debe estar en un rango de [0,255] y si es porcentaje, debe estar entre [0,100]. La función *fixedRange* del

combinador *pNumeroColor* se encarga de controlar el rango:

```
pColorFuncion
  = (λr g b → ColorClave (r, g, b))
      <$ pPalabraClave "rgb" <* pSimboloAmb "(" <*> pNumeroColor
                             <* pSimboloAmb "," <*> pNumeroColor
                             <* pSimboloAmb "," <*> pNumeroColor
                             <* pSimboloAmb ")"
pNumeroColor  =  fixedRange 0 255   <$> pEnteroPos
              <|> fixedRange 0 100   <$> pEnteroPos <* pSimbolo "%"
  where fixedRange start end val
          =  if val < start then start
             else if val > end then end
                  else val
```

Módulo PropiedadesCSS

En este módulo se especifica la lista de propiedades de *CSS* a las que se dará soporte en el proyecto. La forma de especificar es una lista de tuplas de 2 valores: el primero guarda el nombre de la propiedad y el segundo el parser para reconocer el valor de la propiedad.

A medida que se vaya avanzando con el desarrollo de proyecto se aumentará la lista de propiedades a las que se dará soporte y se hará la lista más completa. Por ahora se mostrará sólo un ejemplo de como podría ser esta lista:

```
cssPropiedades
  = [("display"  , display      )
    ,("position", position     )
    ,("top"      , offset_value)
    ,("right"    , offset_value)
    ,("bottom"   , offset_value)
    ,("left"     , offset_value)
    ,("float"    , float        )
    ]
display
  = pValoresClave ["inline","block","list-item","none","inherit"]
position
  = pValoresClave ["static","relative","absolute","fixed","inherit"]
offset_value
  =    pLength
  <|> pPorcentagePos
  <|> pValoresClave ["auto","inherit"]
float
  = pValoresClave ["left","right","none","inherit"]
```

5.4. Interfaces para el parser de CSS

En la Sección 5.1, página 38 se mencionó que las *Hojas de Estilo* son sencillamente una lista de Reglas que pueden pertenecer a un determinado tipo y origen. Pero, no existe una sintaxis para reconocer el tipo y origen, porque son determinados de acuerdo al contexto en que son utilizados.

Entonces, se definirán funciones interfaces las cuales serán llamadas en el contexto adecuado. Estas funciones deben comunicarse con el parser de las hojas de estilo y manejar valores de tipo y origen que sean adecuados al contexto.

En el Código Haskell 28 se define las funciones *parseUserAgent y parseUser* para el usuario *UserAgent* y *User* de manera correspondiente, estas funciones, así como se había mencionado, sólo soportan el tipo de origen *HojaExterna*. Las demás funciones que se han definido en el Código Haskell 28 y Código Haskell 29 son para el usuario *Author*, el cual puede definir hojas de estilo en los distintos 3 tipos de hojas de estilo.

```
parseUserAgent input
   = parseString (pHojaEstilo HojaExterna UserAgent lista_valor_parser) input
parseUser input
   = parseString (pHojaEstilo HojaExterna User lista_valor_parser) input
parseHojaInterna input
   = parseString (pHojaEstilo HojaInterna Author lista_valor_parser) input
parseHojaExterna input
   = parseString (pHojaEstilo HojaExterna Author lista_valor_parser) input
```

Código Haskell 28: Funciones interfaces para el parser de CSS, parte 1

Las funciones definidas en Código Haskell 28 son todas similares, el único que difiere es la función para el estilo *Atributo*, que está definida en el Código Haskell 29. El estilo de un atributo se encuentra en el atributo *style* de una etiqueta.

La restricción para esta parte es, que no se puede tener selectores, sólo se puede tener declaraciones de estilos. Así, los estilos que se encuentren en el atributo se aplican directamente a la etiqueta donde han sido declarados. Es decir, que su selector siempre será un *SelectorSimple* el cual estaría compuesto solamente del nombre de la etiqueta.

Entonces, para construir un estilo atributo, se necesita recibir el nombre de la etiqueta y la entrada. Luego se obtiene las declaraciones con la entrada y se construye un selector simple con el nombre de la etiqueta. Finalmente se construye la regla de estilo atributo con el usuario *Author* y los datos obtenidos:

```
parseEstiloAtributo tag input
   = do decls ← parseString (pDeclaraciones lista_valor_parser) input
        let sel = SimpSelector (TypeSelector tag [] Nothing)
        return (EstiloAtributo, Author, [sel], decls)
```

Código Haskell 29: Funciones interfaces para el parser de CSS, parte 2

Capítulo 6

Asignación de valores a Propiedades de CSS

Hasta esta parte ya se ha trabajado en reconocer la entrada del Navegador Web, lo que ahora se debe hacer es procesar la entrada para su renderización. Así, en este capítulo y el siguiente se describirá el proceso de renderización.

La renderización es guiada a través de las propiedades de CSS. Por ejemplo, la propiedad *display* determina la forma en que se mostrará el elemento o nodo en la pantalla, si su valor es *none*, entonces no se mostrará nada, pero si su valor es *block*, se mostrará un elemento que se renderice como un bloque en la pantalla.
De esa manera, las propiedades de CSS toman un rol importante en el proceso de renderización. Pero, para poder disponer de un valor para una propiedad se debe primeramente asignar un valor a la propiedad.

La asignación de un valor a una propiedad, lamentablemente, no es una tarea trivial, más al contrario es un proceso complejo.
El caso más sencillo es cuando un autor especifica en las hojas de estilo el valor para una propiedad, pero se va complicando cuando se especifica más de dos veces valores diferentes para la misma propiedad, aún peor es cuando el Navegador Web tiene sus propios valores por defecto, o cuando el autor especifica un valor para un subconjunto de elementos.
Todo este proceso de asignar una valor a una propiedad está definido en el algoritmo Cascada (W3C, 2009, cap. 6) de la especificación de CSS. Básicamente, la especificación dice que una vez que se ha recolectado todas las reglas de estilo de la entrada, se debe asignar un valor a cada propiedad a través de un emparejamiento entre el nombre del nodo o etiqueta y el selector de la regla de estilo.

Por lo tanto, en este capítulo se presentará la secuencia de desarrollo del proceso de asignación de un valor a cada propiedad de CSS. En las siguientes secciones, primeramente se recolectará u obtendrá todas las hojas de estilo de la entrada, luego se procederá con el emparejamiento de selectores y finalmente se construirá la lista de propiedades de CSS a las cuales se ha asignado un valor.

Para especificar la semántica del proceso se hará uso de la herramienta UUAGC (Swierstra, s.f.-a), la cual requiere que se reescriban los tipos de datos del lenguaje de marcado a una

sintaxis conocida por UUAGC. Este cambio no afectará los tipos de datos definidos anteriormente. En el Código UUAGC 1 se presenta los tipos de datos del lenguaje de marcado en la sintaxis de UUAGC.

```
DATA Root
  | Root ntree : NTree
DATA NTree
  | NTree Node ntrees : NTrees
TYPE NTrees = [NTree]
DATA Node
  | NText text                 : String
  | NTag  name                 : String
          iamReplacedElement   : Bool
          attributes           : {Map.Map String String}
DERIVING * : Show
```

Código UUAGC 1: Tipos de datos para el lenguaje de marcado

6.1. Obtener las Hojas de Estilo

En esta sección se describirá la forma de obtener todas las hojas de estilo y hacerlas disponibles para su utilización en cada nodo de la estructura *Rosadelfa* o *NTree*.

6.1.1. El tipo *MapSelector*

Las hojas de estilo que se obtienen deben guardarse en alguna estructura o tipo de dato. Para esto, se utiliza la estructura *Map* de Haskell con el tipo *MapSelector*:

```
type MapSelector = Map.Map Selector [(Tipo, Origen, Declaraciones, Int)]
```

La estructura *Map* guarda la información en forma de clave-valor, donde la clave es el *Selector* y su valor es una lista de tuplas compuesto de: *Tipo*, *Origen*, *Declaraciones* y especificidad del selector (*Int*).

En el Apéndice C se muestra la documentación de las funciones más importantes de la librería Map de Haskell.

6.1.2. Obtener las hojas de estilos

Las hojas de estilo pueden ser definidas de 3 formas y por 3 usuarios distintos. Esta información esta descrita en la Sección 2.3.1, página 9 del Marco Teórico.

Para comenzar, en el Código UUAGC 2 se define el atributo heredado *tagEstilo* de tipo *Bool* para verificar si una etiqueta tiene nombre “style”:

```
ATTR NTree NTrees [tagEstilo : Bool | | ]
SEM NTree
  | NTree ntrees.tagEstilo = case @nodo.self of
                               NTag "style" _ _ → True
                               otherwise        → False
```

Código UUAGC 2: Atributo *tagEstilo*

Se ha definido un atributo heredado porque esa información debe ser compartida con todos los nodos hijos de cada nodo, especialmente con los nodos de texto, porque es de esa manera en que *HTML* guarda la información de estilos.

Como se está utilizando un atributo heredado, debe ser inicializado en el nodo *Root*:

```
SEM NRoot
  | NRoot ntree.tagEstilo = False
```

En las siguientes sub-secciones se procederá a obtener las hojas de estilo.

Obtener la Hoja de estilo interna y externa del autor *Author*

Para obtener las hojas de estilo interna y externa del usuario *Author* se realiza las siguientes verificaciones:

- Si es un nodo *NTexto*, se comprueba si su padre es un nodo *NTag* con nombre 'style';
- Sino, se verifica que sea un nodo *NTag* 'link' correcto.

En el Código UUAGC 3 se describe la forma de obtener las hojas de estilo para el usuario *Author*. Se ha definido un atributo sintetizado *reglas* de tipo *MapSelector* para almacenar las hojas de estilo.

```
ATTR NTree NTrees [ | | reglas : {MapSelector} ]
SEM NTree
  | NTree lhs.reglas
    = let nsel = case @nodo.self of
          NTexto str      → if @lhs.tagEstilo
                            then parseHojaInterna str
                            else Map.empty
          NTag nm _ at → if nm ≡ "link" ∧ verificarLinkAtributos at
                            then getHojaExterna at
                            else Map.empty
      in Map.unionWith (++) nsel @ntrees.reglas
{
getHojaExterna atProps = let url   = atPropsMap. ! "href"
                             path = getStylePath url
                         in parseHojaExterna path
}
```

Código UUAGC 3: Obtener las Hojas de estilo

Las declaraciones de un selector son concatenadas con las de sus hijos. Las reglas de los hijos también son concatenadas en una sola lista. Para esto, se utiliza la función (++) y *unionWith* de la estructura *Map*:

```
SEM NTrees
  | Cons lhs.reglas = Map.unionWith (++) @hd.reglas @tl.reglas
  | Nil  lhs.reglas = Map.empty
```

Para verificar si los valores de los nodos son adecuados, no sólo se verifica su nombre, sino también que se disponga de los atributos correctos. La función *verificarLinkAtributos* del Código Haskell 30 se encarga de verificar la existencia de los atributos *href*, *rel* y **type** en el nodo *link*:

```
verificarLinkAtributos at = href ∧ rel ∧ tipo
  where href = Map.member "href" at
        rel  = maybe False (≡ "stylesheet") $ Map.lookup "rel" at
        tipo = maybe False (≡ "text/css")   $ Map.lookup "type" at
```

Código Haskell 30: Función para verificar los atributos del elemento 'link'

Obtener el estilo de los atributos para el usuario Author

Las hojas de estilo también pueden encontrarse en los atributos de un nodo, para esto se debe buscar si existe un atributo *style* en los atributos del nodo, si el atributo no se encuentra, simplemente se devuelve una estructura *Map* vacía (*empty*).

En el Código UUAGC 4 se describe la forma de obtener los estilos desde los atributos de un nodo con etiqueta. Se ha definido la variable local *atributoEstilo* para guardar los estilos de un atributo.

```
SEM NTree
  | NTree loc.atributoEstilo
      = case @nodo.self of
        NTag name _ attrs → maybe Map.empty
                                  (parseEstiloAtributo name)
                                  (Map.lookup "style" attrs)
        otherwise         → Map.empty
```

Código UUAGC 4: Obtener las Hojas de estilo atributo

En todas las formas para obtener los estilos se ha utilizado las funciones correspondientes de la Sección 5.4, página 46 del Parser de CSS.

Hojas de Estilo externas del usuario UserAgent y User

En el Código 5 se define un atributo heredado para obtener las hojas de estilo del autor *UserAgent* y *User*.

```
ATTR NRoot [defaultcss4html : {MapSelector} | | ]
ATTR NRoot [usercss4html : {MapSelector} | | ]
```

Código UUAGC 5: Obtener las Hojas de estilo para el *UserAgent* y *User*

6.1.3. Distribución de Hojas de Estilo

Una vez que se obtienen todas las hojas de estilo, se debe recolectarlas y distribuirlas a cada nodo del *NTree*. Para esto, primeramente se concatena las reglas del *UserAgent* y *User* con las reglas definidas por *Author* y luego son distribuidas con el atributo heredado *css*:

```
ATTR NTrees NTree [css : {MapSelector} | | ]
SEM NRoot
  | NRoot ntree.css = let reglas = Map.unionWith (++)
                                   @lhs.defaultcss4html
                                   @lhs.usercss4html
                      in Map.unionWith (++) reglas @ntree.reglas
```

6.1.4. Variable local *misHojasEstilo*

Finalmente, se ha definido la variable local *misHojasEstilo* en cada nodo del *NTree*. Esta variable contiene las hojas de estilo atributo del nodo y las hojas de estilo de todo el *NTree*:

```
SEM NTree
  | NTree loc.misHojasEstilo = Map.unionWith (++)
                                              @loc.atributoEstilo
                                              @lhs.css
```

Código UUAGC 6: La variable local *misHojasEstilo*

6.2. Emparejando Selectores

Una vez que se tiene todas las hojas de estilo en cada nodo, se debe comenzar con el proceso de asignación de un valor a cada propiedad.

Este proceso inicia con el emparejamiento de selectores, que se encarga de comprobar si una regla de estilo se puede aplicar a un nodo o etiqueta. Esta comprobación es realizada a través de un emparejamiento entre el selector y la etiqueta.

En esta sección se describirá el proceso de emparejamiento de selectores.

6.2.1. Funciones básicas para el emparejamiento

Emparejando Atributos

Se inicia definiendo una función que recibe 3 valores de tipo *String*, de los cuales 2 de ellos son valores y el último es un operador. Esta función hace la comparación y devuelve un valor *Bool*:

```
funOp :: String → String → String → Bool
funOp val1 op val2
  = case op of
        "="     → val1 ≡ val2
        " ∼= " → any (≡ val1) $ words val2
```

Código Haskell 31: Función de comparación para atributos

La función definida en el Código Haskell 31 es usada para comparar los atributos de los selectores con los de HTML. Si el operador es '=', ambos valores deben ser iguales. Si el operador es '∼=', el primer valor debe pertenecer a alguna palabra de la lista de palabras separadas por espacios del segundo valor.

También se define, en el Código Haskell 32, una función para testear o verificar los atributos de CSS y HTML, la cual utiliza la función *funOp* del Código Haskell 31. Se recibe una lista de atributos de *HTML*, un atributo *CSS* y se devuelve un valor *Bool*, que significa si el atributo *CSS* empareja la lista de atributos de *HTML*.

La implementación trabaja en base al atributo de CSS, por ejemplo, si se recibe un *AtribNombre*, simplemente se verifica su existencia en la lista de atributos de *HTML*. Pero, si se trata de un atributo *AtribTipoOp* o *AtribID*, entonces primeramente se obtiene el atributo que se quiere de la lista de atributos de *HTML*, si el atributo no se encuentra, se

devuelve directamente el valor *False*, pero si se encuentra, se llama a la función *funOp* del Código Haskell 31.

```
testAttribute :: Map.Map String String → Atributo → Bool
testAttribute htmlAttrs at
  = case at of
        AtribID value
          → maybe False (funOp value " ~= ") (Map.lookup "id" htmlAttrs)
        AtribNombre name
          → Map.member name htmlAttrs
        AtribTipoOp name op value
          → maybe False (funOp value op)      (Map.lookup name htmlAttrs)
```

Código Haskell 32: Función para testear un atributo

Para terminar esta parte, si se quiere testear una lista de atributos, simplemente se utiliza la función *all* del preludio de *Haskell* con la función *testAtribute* y la lista de atributos de *CSS*:

```
testAttributes :: Map.Map String String → Atributos → Bool
testAttributes htmlAttrs = all (testAttribute htmlAttrs)
```

Código Haskell 33: Función para testear varios atributos

Emparejando Pseudo elementos

Lo siguiente es verificar un pseudo elemento. Para esto se recibe un valor *Bool* y el pseudo elemento. Si el valor *Bool* es verdadero, entonces se debe verificar que exista un pseudo elemento, caso contrario se debe verificar su no existencia:

```
testPseudo :: Bool → Maybe PseudoElemento → Bool
testPseudo bool pse = if bool then isJust pse else isNothing pse
```

Código Haskell 34: Función para testear pseudo-elementos

6.2.2. Emparejar un Selector Simple

Con las funciones definidas hasta ahora es posible implementar una función que verifique o compruebe un selector simple.

En el Código Haskell 35 se define la función *testSimpleSelector*, que recibe el selector simple, el nodo con el que se quiere verificar y un valor *Bool* que indica si se debe verificar el pseudo elemento. Y se devuelve un valor *Bool* para indicar si se emparejo el selector simple y el nodo.

Los selectores sólo se aplican a los nodos con etiquetas (*NTag*), no así a los nodos de texto. Entonces, si el nodo es diferente a un *NTag*, directamente se devuelve *False*.

Caso contrario, se trabaja de acuerdo al selector, si el selector es universal (*UnivSelector*) se verifica sus atributos y el pseudo elemento, pero si es un tipo selector (*TypeSelector*) además de lo anterior, se verifica su nombre.

```
testSimpleSelector :: SSelector → Nodo → Bool → Bool
testSimpleSelector ssel nd bool
  = case nd of
      NTag nm1 _ attrs → case ssel of
                           TypeSelector nm2 atsel pse
                             → (nm1 ≡ nm2)
                               ∧ testAttributes attrs atsel
                               ∧ testPseudo bool pse
                           UnivSelector atsel pse
                             → testAttributes attrs atsel
                               ∧ testPseudo bool pse
      otherwise          → False
```

Código Haskell 35: Función para verificar un Selector Simple

6.2.3. Emparejar Selectores compuestos

A continuación se describirán las funciones de emparejamiento para cada uno de los tipos del selector compuesto.

La función matchSelector

La función *matchSelector* definida en el Código Haskell 36 es una función que empareja de acuerdo al tipo de selector. Esta función consume elemento por elemento la lista de *ESelector*.

Si *Selector* es una lista vacía, entonces se devuelve el valor *True*, que significa que el selector tuvo éxito en emparejar el nodo. Caso contrario, se continua llamando a la función correspondiente para emparejar el primer elemento de la lista.

```
type TypeMatchSelector
  = Nodo → [(Nodo, [Nodo])] → [Nodo] → [(Nodo, [Nodo])] → Int → Selector → Bool → Bool
matchSelector :: TypeMatchSelector
matchSelector _ _ _ _ _ [] _
  = True
matchSelector nd fathers siblings before level (sel : nextSel) pseudo
  = case sel of
      SimpSelector s
        → applySimplSelector nd fathers siblings before level s nextSel pseudo
      DescSelector s
        → applyDescdSelector nd fathers siblings before level s nextSel pseudo
      ChilSelector s
        → applyChildSelector nd fathers siblings before level s nextSel pseudo
      SiblSelector s
        → applySiblnSelector nd fathers siblings before level s nextSel pseudo
```

Código Haskell 36: Función matchSelector

La función *matchSelector* recibe un nodo de tipo *Nodo*, una lista de padres (que también incluye a los hermanos de cada padre) de tipo $[(Nodo, [Nodo])]$, una lista de hermanos de tipo $[Nodo]$, una lista de padres anteriores (los nodos padres que ya se reviso) de tipo $[(Nodo, [Nodo])]$, un contador de tipo *Int* que indica el nivel del árbol donde se encuentra el proceso de emparejamiento y un valor *Bool* que indica si se va a revisar el pseudo elemento. Finalmente, la función devuelve un valor *Bool* el cual indica si se emparejo el selector con el nodo.

Emparejar un Simple Selector

En el Código Haskell 37 se define la función *applySimplSelector*, que empareja un selector simple. Para emparejar un selector simple, se llama a la función *testSimpleSelector* definida en Código Haskell 35, y luego se continua emparejando los siguientes selectores.

```
applySimplSelector nd fathers siblings before level s nextSel pseudo
  = testSimpleSelector s nd pseudo
  ∧ matchSelector nd fathers siblings before level nextSel False
```

Código Haskell 37: Emparejar un SimplSelector

Note que no se modifica las variables 'before' y 'level' porque se está manteniendo en el mismo nivel.

Emparejar un Selector Descendiente

Para emparejar un selector descendiente, primeramente se verifica que la lista de padres no sea vacía, si es así, se devuelve directamente *False*, porque no se puede aplicar este tipo de selector.

En otro caso, cuando la lista de padres no es vacía, se crea 2 opciones de emparejamiento con el operador OR `(||)` de Haskell. La primera opción empareja el primer padre y el selector y continua emparejando los siguientes selectores añadiendo el primer padre a la lista de anteriores e incrementando el contador de nivel *level*. Si la primera opción falla en el emparejamiento, se utiliza la segunda opción.

La segunda opción vuelve a llamar a la función para emparejar selectores descendientes con la lista restante de padres.

El uso del operador *OR* de Haskell permite implementar el algoritmo de *backtraking* de los selectores descendientes.

En el Código Haskell 38 se describe la implementación de la función *applyDescdSelector*.

```
applyDescdSelector _ []      _      _      _    _ _      _
  = False
applyDescdSelector nd (f : fs) siblings before level s nextSel pseudo
  = (testSimpleSelector s (fst f) pseudo
      ∧ matchSelector nd fs siblings (f : before) (level + 1) nextSel False)
  ∨ applyDescdSelector nd fs siblings before level s nextSel False
```

Código Haskell 38: Emparejar un DescdSelector

Emparejar un Selector Hijo

El selector hijo es similar al selector descendiente, la única diferencia entre ambos es que el selector hijo simplemente revisa el primer nivel, en otras palabras, no necesita hacer backtracking. En el Código Haskell 39 describe la implementación de la función *applyChildSelector*.

```
applyChildSelector _ []      _      _      _    _ _      _
  = False
applyChildSelector nd (f : fs) siblings before level s nextSel pseudo
  = testSimpleSelector s (fst f) pseudo
  ∧ matchSelector nd fs siblings (f : before) (level + 1) nextSel False
```

Código Haskell 39: Emparejar un ChildSelector

Note que de la misma manera que para el selector descendiente, se incrementa el nivel *level*.

Emparejar un Selector Hermano

Para implementar el Selector Hermano, se inicia definiendo la función *getNextvalidTag* del Código Haskell 40, que se encarga de encontrar un hermano válido dada una lista de hermanos.

Un hermano válido es el primer nodo o etiqueta que se encuentra en la lista. Note que este nodo no puede ser comentario o texto. Se retorna una tupla, donde el primer elemento es un 'Bool' que indica si se encontró un hermano válido, y el segundo elemento es una lista de nodos, donde el primer elemento es el hermano válido. Sino se encuentra un hermano válido, se devuelve lista vacía.

```
getNextValidTag :: [Nodo]              → (Bool, [Nodo])
getNextValidTag   []                   = (False, [])
getNextValidTag   l@(NTag _ _ _: _)    = (True, l)
getNextValidTag   (_ : xs)             = getNextValidTag xs
```

Código Haskell 40: Función para encontrar un hermano válido

Para el selector hermano (Sibling Selector) se debe considerar a los nodos hermanos del nodo actual. Los nodos hermanos pueden aparecer en 2 lugares: en la variable *siblings* si se

encuentra en el nivel 0, o en la lista de nodos padres y hermanos *before* si se encuentra en un nivel mayor a 0.

Cuando se encuentra en un nivel mayor a 0, la lista de hermanos es el segundo elemento de la lista de tuplas *before*.

En el Código Haskell 41 se define la función *applySiblnSelector*, esta función define la variable *brothers* que guarda la lista de hermanos de nivel 0 o superior.

Luego, dependiendo de la función *getNetValidTag*, se procede a construir la nueva lista de hermanos y a procesar los siguientes selectores.

Si existe un hermano válido, entonces se debe comprobar que sea el hermano que se esta buscando (para esto, se llama a función *testSimpleSelector* definida en Código Haskell 35), caso contrario se retorna *False* y termina la función.

Si el nodo hermano empareja con el selector, entonces se debe construir una nueva lista de hermanos, porque podría existir otro selector hermano el cual debe trabajar con la nueva lista de hermanos. **Nota:** Solo se modifica la lista de hermanos, porque la otra información puede ser importante para los selectores restantes.

```
applySiblnSelector nd fathers siblings before level s nextSel pseudo
  = let brothers              = if level ≡ 0 then siblings else snd $ head before
        (bool, ts)            = getNextValidTag brothers
        (ntest, rsibl)        = if bool
                                then (testSimpleSelector s (head ts) pseudo, tail ts)
                                else (False                                , [] )
        (newSibl, newBefo)    = if level ≡ 0
                                then (rsibl, before)
                                else let (f, _)      = head before
                                         newBefore   = (f, rsibl) : tail before
                                     in (siblings, newBefore)
    in ntest ∧ matchSelector nd fathers newSibl newBefo level nextSel False
```

Código Haskell 41: Emparejar un SiblnSelector

Vea que no se modifica el nivel, ni tampoco la lista *before*, porque se está manteniendo en el mismo nivel.

6.2.4. La función emparejarSelector

Finalmente, en el Código Haskell 42 se define la función 'emparejarSelector', la cual inicializa los valores por defecto para llamar a la función 'matchSelector'.

```
emparejarSelector :: Nodo → [(Nodo, [Nodo])] → [Nodo] → Selector → Bool → Bool
emparejarSelector nd fths sbls = matchSelector nd fths sbls [] 0
```

Código Haskell 42: La función emparejarSelector

Con todo esto, la función *emparejarSelector* recibe un *Nodo* con el cual emparejar, una lista de nodos padres junto con sus hermanos, otra lista de nodos hermanos, el selector a

emparejar y un valor *Bool* que indica si se desea emparejar el elemento pseudo. Y como resultado se devuelve un *Bool* que indica verdadero si el selector empareja con el nodo o falso en caso contrario.

6.2.5. Usando UUAGC para recolectar información

Ahora se utilizará la herramienta UUAGC para recolectar la información que la función *emparejarSelector* necesita.

Se inicia definiendo el atributo sintetizado *self* sobre *Nodo*, el cual guarda la información de sí mismo:

```
ATTR Nodo [ | | self : SELF ]
```

También se define otro atributo sintetizado *nd* en cada *NTree*, este atributo almacena el nodo *self*:

```
ATTR NTree [ | | nd : Nodo ]
SEM NTree
  | NTree lhs.nd = @nodo.self
```

Ahora, se construirá la lista de padres y hermanos de cada padre. Se define un atributo heredado *fathers* el cual es una lista de tuplas, donde el primer elemento es el nodo padre y el segundo elemento es la lista de los hermanos del nodo padre:

```
ATTR NTrees NTree [fathers : {[(Nodo, [Nodo])]} | | ]
SEM NTree
  | NTree ntrees.fathers = (@nodo.self, @loc.siblings) : @lhs.fathers
          loc.fathers    = @lhs.fathers
```

También se define una variable local *fathers* que almacena la lista de padres que se ha recolectado hasta el nivel del *NTree* donde es invocado.

Al momento de construir la tupla, se hace referencia a **loc**.*siblings*. La variable local *siglings* es similar a la variable local *fathers* con la diferencia de que *siblings* guarda la lista de hermanos que se ha recolectado hasta el nivel de *NTree* donde es invocado.

La lista de hermanos está definido a través de un atributo heredado *siblings*, que colecciona los nodos de los *NTrees* y comparte la lista de hermanos con el *NTree*.

```
ATTR NTrees NTree [siblings : {[Nodo]} | | ]
SEM NTrees
  | Cons tl.siblings  = @hd.nd : @lhs.siblings
       hd.siblings = @lhs.siblings
```

Cada lista de hermanos es inicializada en cada *NTree* con una lista vacía. También se define la variable local *siblings* (la cual es utilizada en el atributo *fathers*) que almacena la lista de hermanos del nodo:

```
SEM NTree
  | NTree ntrees.siblings = []
         loc.siblings    = @lhs.siblings
```

Se necesita inicializar los atributos *fathers* y *siblings* en el nodo *Root*:

```
SEM NRoot
  | NRoot ntree.fathers  = []
          ntree.siblings = []
```

6.2.6. La variable local *reglasEmparejadas*

Finalmente, en el Código UUAGC 7, se define la variable local *reglasEmparejadas* que almacena la lista de reglas CSS que emparejan con el nodo.

Para emparejar las reglas se hace uso de la función *emparejarSelector* definida en el Código Haskell 42:

```
SEM NTree
  | NTree loc.reglasEmparejadas
    = let applyMatchSelector selector = emparejarSelector @nodo.self
                                                          @loc.fathers
                                                          @loc.siblings
                                                          selector
                                                          False
          obtenerReglas selector r1 r2 = if applyMatchSelector selector
                                         then r1 ++ r2
                                         else r2
      in Map.foldWithKey obtenerReglas [] @loc.misHojasEstilo
```

Código UUAGC 7: La variable local reglasEmparejadas

6.3. Propiedades de CSS

En esta sección se continua con la asignación de un valor a una propiedad. Primeramente se definirá el tipo de dato para representar una propiedad de CSS en Haskell, luego se describirá las funciones que se encarguen de asignar un valor a una propiedad.

6.3.1. El tipo de dato Property

Uno de los tipos de datos importantes del proyecto es el tipo de dato *Property*. Este tipo representa a una propiedad de CSS (descrita en Sección 2.3.3, página 13), por consiguiente,

tiene un nombre, un valor *Bool* para representar si es heredable, un valor inicial, un parser para sus valores, el valor de la propiedad de tipo *PropertyValue* y dos funciones para generar valores de tipo computed y used respectivamente.

En el Código Haskell 43 se muestra la definición del tipo de dato *Property*.

```
data Property
  = Property { nombre            :: String
             , inherited         :: Bool
             , initial           :: Valor
             , valor             :: Parser Valor
             , propertyValue     :: PropertyValue
             , fnComputedValue :: FunctionComputed
             , fnUsedValue       :: FunctionUsed
             }
data PropertyValue
  = PropertyValue { specifiedValue :: Valor
                  , computedValue :: Valor
                  , usedValue     :: Valor
                  , actualValue   :: Valor
                  }
```

Código Haskell 43: El tipo de dato *Property*

6.3.2. Funciones útiles para Property

En esta sub sección se define algunas funciones útiles para *Property*.

- Obtener el nombre de la propiedad:

```
getPropertyName = nombre
```

Código Haskell 44: Obtener el nombre de una propiedad

- Obtener el *PropertyValue* de una propiedad. Recibe una estructura *Map* y una clave y devuelve el *PropertyValue* de la propiedad donde apunta la clave en la estructura *Map*.

```
get :: Map.Map String Property → String → PropertyValue
get map k = propertyValue $ map Map.! k
```

Código Haskell 45: Obtener el *PropertyValue* de una propiedad

- Obtener el *PropertyValue* de una propiedad encapsulado en un tipo *Maybe*. Al igual que el anterior, se recibe una estructura *Map* y una clave, si la clave no se encuentra en la estructura *Map* se devuelve *Nothing*, caso contrario se devuelve el *PropertyValue* (encapsulado en el tipo *Just*) de la propiedad donde apunta la clave.

```
getM :: Map.Map String Property → String → Maybe PropertyValue
getM map k = maybe Nothing (Just ∘ propertyValue) $ Map.lookup k map
```

Código Haskell 46: Obtener el *PropertyValue* de una propiedad encapsulado en *Maybe*

- Modificar el *PropertyValue* de una propiedad. Se recibe una función que modifique el *PropertyValue*, la propiedad y se retorna la propiedad modificada.

```
adjustPropertyValue :: (PropertyValue → PropertyValue) → Property → Property
adjustPropertyValue fpv prop@(Property _ _ _ _ pv _ _)
  = prop { propertyValue = fpv pv }
```

Código Haskell 47: Función para modificar el *PropertyValue* de una propiedad

- Comparar un *ValorClave* con un *String*. Se recibe una función para comparar, el valor y el *String*. Si el valor que se recibe no es *ValorClave*, entonces se retorna directamente *False*, caso contrario se aplica la función que se recibe.

```
compareKeyPropertyValueWith fcmp val str
  = case val of
      ValorClave str' → fcmp str' str
      _               → False
```

Código Haskell 48: Función genérica para comparar el valor de una propiedad

Con la ultima definición, se puede crear una función que realice una comparación con la función de igualdad:

```
compareKeyPropertyValue :: Valor → String → Bool
compareKeyPropertyValue = compareKeyPropertyValueWith (≡)
```

Código Haskell 49: Función para comparar el valor de una propiedad con la igualdad

- Obtener el *ValorClave* envuelto en el tipo de dato *Valor*.

```
unKeyUsedValue      = (λ(ValorClave v) → v) ∘ usedValue
unKeyComputedValue  = (λ(ValorClave v) → v) ∘ computedValue
unKeySpecifiedValue = (λ(ValorClave v) → v) ∘ specifiedValue
```

Código Haskell 50: Funciones que retornan el valor almacenado por el constructor *ValorClave*

- Obtener el *ColorClave* envuelto en el tipo de dato *Valor*.

```
unKeyUsedColor      = (λ(ColorClave v) → v) ∘ usedValue
unKeyComputedColor  = (λ(ColorClave v) → v) ∘ computedValue
unKeySpecifiedColor = (λ(ColorClave v) → v) ∘ specifiedValue
```

Código Haskell 51: Funciones que retornan el color almacenado por el constructor *ColorClave*

- Obtener el *NumeroPixel* envuelto en el tipo de dato *Valor*.

```
unPixelUsedValue      = (λNumeroPixel px → px) ∘ usedValue
unPixelComputedValue  = (λNumeroPixel px → px) ∘ computedValue
unPixelSpecifiedValue = (λNumeroPixel px → px) ∘ specifiedValue
```

Código Haskell 52: Funciones que retornan el número *pixel* almacenado por el constructor *NumeroPixel*

- Verificar que el *ValorClave* de una propiedad sea el mismo que el que recibe como argumento.

```
verifyProperty :: String → String → Map.Map String Property → Bool
verifyProperty nm val props
  = let pval = computedValue $ props `get` nm
    in compareKeyPropertyValue pval val
```

Código Haskell 53: Función para comparar el *ValorClave* de una propiedad

6.3.3. SpecifiedValue de CSS

El SpecifiedValue de CSS corresponde al valor especificado en las hojas de estilo, el cual se obtiene como resultado de aplicar el algoritmo cascada de CSS (Sección 2.3.1, página 12).

El algoritmo en Cascada

El algoritmo en cascada (Sección 2.3.1, página 12) indica que una vez que se tiene las hojas de estilo que emparejan con el nodo, lo primero que debe hacer es obtener todas las declaraciones de estilo para la propiedad, para la cual se quiere encontrar su valor. Luego se aplica el algoritmo de ordenamiento en cascada a la lista resultante y finalmente se selecciona el primer valor de la lista resultado.

En el Código Haskell 54 se muestra la definición de la función *doSpecifiedValue*, que se encarga de obtener las declaraciones, aplicar el algoritmo cascada y seleccionar un valor.

Se utiliza *parttern matching* en la definición de la función, si el valor de la propiedad es *NoEspecificado*, entonces se aplica el algoritmo, pero si existe un valor, simplemente se devuelve el valor especificado.

```
doSpecifiedValue :: Map.Map String Property
                  → Bool
                  → [(Tipo, Origen, Declaraciones, Int)]
                  → Property
                  → Property
doSpecifiedValue father
                 isRoot
                 rules
                 prop@(Property nm inh defval _ NoEspecificado _ _ _ _ _)
  = selectValue ∘ applyCascadingSorting $ getPropertyDeclarations nm rules
  where applyCascadingSorting
          = head' ∘ dropWhile null ∘ cascadingSorting
        selectValue rlist
          = if null rlist
            then if inh ∧ ¬ isRoot
                   then let sv = getComputedValue $ father `get` nm
                            in prop {propertyValue = pv {specifiedValue = sv}}
                   else prop {propertyValue = pv {specifiedValue = defval}}
            else  let (_, _, Declaracion _ val _, _, _) = head rlist
                  in if compareKeyPropertyValue val "inherit"
                     then if isRoot
                            then prop {propertyValue = pv {specifiedValue = defval}}
                            else let sv = computedValue $ father `get` nm
                                     in prop {propertyValue = pv {specifiedValue = sv}}
                     else prop {propertyValue = pv {specifiedValue = val}}
doSpecifiedValue _ _ _ p
  = p
```

Código Haskell 54: La función *doSpecifiedValue*

La función *applyCascadingSorting* devuelve el primer elemento del resultado de eliminar todas listas vacías de la llamada a la función *cascadingSorting*. Si la lista que se aplica a *head'* es vacía, se devuelve lista vacía.

Seleccionar un valor

La función *selectValue* que está definida en la función *doSpecifiedValue*, selecciona un valor para una propiedad de CSS.

Si la lista de declaraciones que se recibe es vacía, significa que no se encontró un valor en las hojas de estilo, en este caso, se puede hacer 2 cosas: (a) heredar el valor del nodo padre o (b) devolver el valor por defecto de la propiedad.

Para el primer caso, se debe verificar que la propiedad sea heredable y que no sea el nodo *Root* (porque el nodo *Root* no tiene nodo padre).

En el caso de que la lista que se recibe no es vacía, significa que existe al menos una declaración para la propiedad en las hojas de estilo, entonces se obtiene la primera declaración de la lista. Luego se verifica si el valor que se ha obtenido es *inherit*, si es así, se debe obtener el valor del padre con la consideración de no encontrarse en el nodo *Root*, pero si se encuentra

en el nodo *Root* se debe usar el valor por defecto de la propiedad.
Si el valor de la primera declaración no es "inherit", se utiliza su valor para construir la propiedad.

Obteniendo las declaraciones para una propiedad específica

En el Código Haskell 55 se define la función *getPropertyDeclarations* que recibe el nombre de una propiedad, una lista de declaraciones y devuelve todas las declaraciones para el nombre de la propiedad.

Como una tarea extra, se expande la lista de listas de declaraciones a una lista simple de declaraciones.

```
getPropertyDeclarations :: String → [(Tipo, Origen, Declaraciones, Int)]
                                  → [(Tipo, Origen, Declaracion, Int)]
getPropertyDeclarations nm1 = foldr fConcat []
  where fConcat (tipo, origen, declaraciones, spe) r2
          = let r0 = filter (λ(Declaracion nm2 _ _) → nm1 ≡ nm2) declaraciones
            in if null r0
               then r2
               else let r1 = map (λdecl → (tipo, origen, decl, spe)) r0
                    in r1 ++ r2
```

Código Haskell 55: Obtener todas las declaraciones para una propiedad

Ordenamiento en Cascada

En la Sección 2.3.1, página 12 se describe el algoritmo de ordenamiento en cascada, el cual es implementado en la función *cascadingSorting* definida en el Código Haskell 56. Esta función recibe una lista de tuplas de *Tipo*, *Origen*, *Declaracion* y *Especificidad* del selector que corresponde al último valor de la tupla. La especificidad es calculado después del proceso de análisis sintáctico (parsing), pero se explicará como se encuentra la especificidad de un selector en la siguiente sección.

Lo primero que se hace en la función *cascadingSorting* es asignar una posición a cada declaración (esto para ordenar de acuerdo a la especificación). Luego, se construye la lista resultado donde la primera sub-lista son las declaraciones *User Important*, posteriormente están los de *Author Important* y así sucesivamente.

La función *getDeclarations* obtiene las declaraciones de acuerdo al origen e importancia. Luego, se llama a la función *sortBy* para que ordene la lista de acuerdo a la especificidad y posición.

```
cascadingSorting :: [(Tipo, Origen, Declaracion, Int)] → [[(Tipo, Origen, Declaracion, Int, Int)]]
cascadingSorting lista1
  = let  lista2 = myZip lista1 [1..]
         lst1   = sortBy fsort $ getDeclarations User      True  lista2
         lst2   = sortBy fsort $ getDeclarations Author    True  lista2
         lst3   = sortBy fsort $ getDeclarations Author    False lista2
         lst4   = sortBy fsort $ getDeclarations User      False lista2
         lst5   = sortBy fsort $ getDeclarations UserAgent False lista2
    in [lst1, lst2, lst3, lst4, lst5]
  where myZip []           _      = []
        myZip ((a, b, c, d) : next) (f : fs) = (a, b, c, d, f) : myZip next fs
        getDeclarations origin important
          = filter (λ(_, org, Declaracion _ _ imp, _, _) → origin ≡ org ∧ important ≡ imp)
        fsort (_, _, _, v1, v3) (_, _, _, v2, v4)
          | v1 > v2              = LT
          | v1 < v2              = GT
          | v1 ≡ v2 ∧ v3 > v4    = LT
          | v1 ≡ v2 ∧ v3 < v4    = GT
          | otherwise            = EQ
```

Código Haskell 56: Algoritmo *cascadingSorting*

Especificidad de un Selector

Este valor corresponde al 4to elemento de la tupla de la lista de declaraciones. Se obtiene este valor después de hacer el análisis sintáctico (parsing).

La forma de obtener la especificidad de un selector esta descrita en la Sección 2.3.1, página 12. Básicamente, la especificidad de un selector es un número de 4 cifras: 'abcd'.

En cada cifra se cuenta las ocurrencias de un cierto tipo. Por ejemplo, en la cifra 'd' se cuenta todas las ocurrencias de los elementos pseudo. Para esto se ha definido un atributo sintetizado 'd':

```
ATTR MaybePseudo [ | | d : Int]
SEM MaybePseudo
  | Just    lhs.d = 1   -- tiene un pseudo
  | Nothing lhs.d = 0   -- no tiene pseudo
```

También se cuenta todos los atributos *ID* en 'b' y todos los atributos que no son *ID* 'c'. Del mismo modo, se ha definido un atributo sintetizado para cada contador.

Como se puede tener una lista de atributos, se suma todas las ocurrencias de 'b' y 'c'.

```
ATTR Atributos Atributo [ | | b, c USE {+} {0} : Int]
SEM Atributo
  | AtribID lhs.b = 1
            lhs.c = 0          -- 0 (cero) porque no tiene otros atributos
  | AtribNombre lhs.b = 0      -- 0 (cero) porque no es ID
            lhs.c = 1
  | AtribTipoOp lhs.b = 0      -- 0 (cero) porque no es ID
            lhs.c = 1
```

También se debe contar los nombres de los elementos en 'd'. Vea que sólo el *TypeSelector* tiene nombre, el *UnivSelector* no tiene nombre y por consecuente no se cuenta.

Como puede haber una lista de *ESelector*, se suma todas las ocurrencias de estas cifras:

```
ATTR Selector ESelector SSelector [ | | b, c, d USE {+} {0} : Int]
SEM SSelector
  | TypeSelector lhs.d = @maybePseudo.d + 1
```

La cifra 'a' es un caso especial, porque si se está frente a un *EstiloAtributo*, la especificidad es igual a '1000' y se omiten las otras cifras. Es decir $a = 1$ y $b = 0, c = 0, d = 0$. Caso contrario, $a = 0$ y no se omiten las demás cifras.

```
ATTR Regla [ | | output : {(Tipo, Origen, Selector, Declaraciones, Int)}]
SEM Regla
  | Tuple lhs.output = let especificidad
                         = if @x1.self ≡ EstiloAtributo
                           then 1000
                           else @x3.b ∗ (10 ↑ 3) +
                                @x3.c ∗ (10 ↑ 2) +
                                @x3.d ∗ 10
                      in (@x1.self, @x2.self, @x3.self, @x4.self, especificidad)
```

Las referencias a atributos 'self' son atributos sintetizados que contienen su mismo valor y tipo:

```
SET All = ∗ − SRoot
ATTR All [ | | self : SELF]
```

Una vez que se obtiene la especificidad de un selector, se recoge todas las reglas en una lista:

```
ATTR HojaEstilo [ | | output USE {:} {[]} : {[(Tipo, Origen, Selector, Declaraciones, Int)]}]
```

Finalmente se construye la estructura *Map* con el tipo *MapSelector* (descrito en Sección 6.1.1, página 48) y se aplica la función *reverse* de Haskell sobre los selectores para que el proceso de emparejamiento sea más sencillo.

```
ATTR SRoot [ | | output2 : { MapSelector } ]
SEM SRoot
  | SRoot lhs.output2 = let fMap (t, o, s, d, e) map'
                                = Map.insertWith (++) (reverse s) [(t, o, d, e)] map'
                        in foldr fMap Map.empty @he.output
```

Código UUAGC 8: Construir la estructura *MapSelector*

6.3.4. ComputedValue, UsedValue y ActualValue de CSS

La función para encontrar el *specifiedValue* es el mismo para todas las propiedades, pero las funciones para encontrar los otros valores (computedValue, usedValue y actualValue) pueden ser diferentes para cada propiedad.

Es por eso que se ha definido funciones 'fnComputedValue' y 'fnUsedValue' en el tipo de dato *Property*. No se ha definido una función para el *actualValue* porque no se está utilizando en todo el proyecto.

ComputedValue

En el Código Haskell 57 se muestra el tipo de la función 'fnComputedValue'.

```
type FunctionComputed = Bool                        -- soy el root?
                      → Map.Map String Property     -- father props
                      → Map.Map String Property     -- local props
                      → Maybe Bool                  -- soy replaced ?
                      → Bool                        -- revizare el pseudo?
                      → String                      -- Nombre
                      → PropertyValue               -- PropertyValue
                      → Valor
```

Código Haskell 57: El tipo de la función *fnComputedValue*

Para obtener el valor *computedValue* de una propiedad se debe llamar a la función *doComputedValue* con los parámetros que necesita la función *fnComputedValue* de la propiedad.

El Código Haskell 58 muestra la definición de la función *doComputedValue*. Se está usando *Pattern matching* sobre la propiedad para definir la función. Si el valor *computedValue* de la propiedad es *NoEspecificado* entonces se obtiene la función *fnComputedValue* de la propiedad y se aplica con los parámetros con que se le envía a *doComputedValue*, con ese resultado se construye una nueva propiedad.

Pero, si el valor *computedValue* de la propiedad es diferente a *NoEspecificado*, significa que se aplicó la función *fnComputedValue*, entonces simplemente se devuelve la propiedad:

```
doComputedValue :: Bool
                -> Map.Map String Property
                -> Map.Map String Property
                -> Maybe Bool
                -> Bool
                -> Property
                -> Property
doComputedValue iamtheroot
                fatherProps
                locProps
                iamreplaced
                iamPseudo
                prop@(Property nm _ _ _ pv@(PropertyValue _ NoEspecificado _ _) fnc _)
  = let cv = fnc iamtheroot fatherProps locProps iamreplaced iamPseudo nm pv
    in prop {propertyValue = pv {computedValue = cv}}
doComputedValue _ _ _ _ _ p
  = p
```

Código Haskell 58: La función *doComputedValue*

Muchas veces el valor *computedValue* es el mismo que *specifiedValue*, para estos casos, se ha definido un función 'computed_asSpecified' el cual copia el valor del *specifiedValue* al *computedValue*.

```
computed_asSpecified :: FunctionComputed
computed_asSpecified _ _ _ _ _ _ = specifiedValue
```

Código Haskell 59: La función *computed_asSpecified*

UsedValue

En el Código Haskell 60 se muestra el tipo de la función 'fnUsedValue'.

```
type FunctionUsed = Bool                        -- soy el root?
                  -> (Float, Float)             -- dimenciones del root
                  -> Map.Map String Property    -- father props
                  -> Map.Map String Property    -- local props
                  -> Map.Map String String      -- atributos
                  -> Bool                       -- soy replaced?
                  -> String                     -- Nombre
                  -> PropertyValue              -- PropertyValue
                  -> Valor
```

Código Haskell 60: El tipo de la función *fnUsedValue*

Al igual que *doComputedValue*, en el Código Haskell 61 se define la función *doUsedValue* para aplicar la función y encontrar el valor de *usedValue*. Esta función está definido usando

pattern matching sobre el valor de *usedValue*, si este es *NoEspecificado*, se aplica la función, caso contrario se devuelve simplemente la propiedad.

```
doUsedValue :: Bool
            → (Float, Float)
            → Map.Map String Property
            → Map.Map String Property
            → Map.Map String String
            → Bool
            → Property
            → Property
doUsedValue iamtheroot
            icbsize
            fatherProps
            locProps
            attrs
            iamreplaced
            prop@(Property nm _ _ _ pv@(PropertyValue _ _ NoEspecificado _) _ fnu)
  = let uv = fnu iamtheroot icbsize fatherProps locProps attrs iamreplaced nm pv
    in prop {propertyValue = pv {usedValue = uv}}
doUsedValue _ _ _ _ _ _ p = p
```

Código Haskell 61: La función *doUsedValue*

También se define una función 'used_asComputed' el cual copia el valor de *computedValue* en el de *usedValue*:

```
used_asComputed :: FunctionUsed
used_asComputed _ _ _ _ _ _ _ _ = computedValue
```

Código Haskell 62: La función *used_asComputed*

6.3.5. La lista de Propiedades

En la Sección 5.3.1, página 45 se ha definido una lista de propiedades CSS como una lista de tuplas, donde el primer valor era el nombre de la propiedad y el segundo valor era el parser para los valores de la propiedad. En esta sección se modificará esa lista de propiedades, de manera que la nueva lista debe almacenar el tipo de dato *Property*.

Para crear un valor de tipo *Property* no se necesita que todos los parámetros sean escritos explícitamente, por ejemplo, los valores *specifiedValue*, *computedValue*, *usedValue* y *actualValue* no necesitan ser descritos al momento inicial. Así, en el Código Haskell 63 se ha definido la función *mkProp*, que recibe sólo los argumentos que necesarios.

```
mkProp (nm, bool, init, pval, fnc, fnu)
  = Property nm
             bool
             init
             pval
             defaultPropertyValue
             fnc
             fnu
defaultPropertyValue
  = PropertyValue {specifiedValue  = NoEspecificado
                  , computedValue = NoEspecificado
                  , usedValue     = NoEspecificado
                  , actualValue   = NoEspecificado
                  }
```

Código Haskell 63: La función *mkProp*

A continuación, se muestra un ejemplo de la nueva lista de propiedades utilizando la función *mkProp*:

```
propiedadesCSS :: [Property]
propiedadesCSS
 = [mkProp ("display", False, (ValorClave "inline"), display, cdisplay, udisplay)]
display :: Parser Valor
display = pValoresClave ["inline"
                        ,"block"
                        ,"list-item"
                        ,"none"
                        ,"inherit"]
cdisplay = computed_asSpecified
udisplay = used_asComputed
```

En la anterior versión (Sección 5.3.1, página 45) se necesitaba tener una lista de tuplas con nombre y parser del valor. Se puede definir la función *propertyParser* para obtener el nombre y parser de una propiedad:

```
propertyParser :: Property → (String, Parser Valor)
propertyParser (Property nm _ _ pr _ _ _) = (nm, pr)
```

Código Haskell 64: Obtener el nombre y parser de una propiedad

Luego, la función para construir la declaración cambiaría de la siguiente manera:

```
lista_valor_parser :: [Parser Declaraciones]
lista_valor_parser = map (construirDeclaracion ∘ propertyParser) propiedadesCSS
```

6.3.6. Encontrar los valores de SpecifiedValue y ComputedValue

Para encontrar los valores *specifiedValue* de cada propiedad de la lista de propiedades se debe primeramente recolectar los argumentos que se necesita para llamar a la función *doSpecifiedValue*.

Para ello, se necesita saber si el nodo donde se encuentra, es el nodo *Root*. Se define el atributo heredado *iamtheroot* de tipo *Bool* para identificar al nodo *Root*:

```
ATTR NTree NTrees [iamtheroot : Bool | | ]
SEM NRoot
  | NRoot ntree.iamtheroot = True
SEM NTree
  | NTree ntrees.iamtheroot = False
```

Con esta información, se puede llamar a la función *doSpecifiedValue* de la siguiente forma:

```
SEM NTree
  | NTree loc.specifiedValueProps
    = let propsTupla = map (λp → (getPropertyName p
                                 , doSpecifiedValue @lhs.propsFather
                                                    @lhs.iamtheroot
                                                    @loc.reglasEmparejadas
                                                    p
                                 )
                           ) propiedadesCSS
      in Map.fromList propsTupla
```

Código UUAGC 9: Llamando a la función *doSpecifiedValue*

En la definición del Código UUAGC 9, se construye una lista de tuplas, donde el primer valor es el nombre de la propiedad y el segundo valor es el resultado de llamar a *doSpecifiedValue*. Con esta lista, se construye la estructura 'Map String Property' y se guarda en la variable local *specifiedValueProps*.

Los parámetros con los que se llama a la función *doSpecifiedValue* son: un valor heredado *propsFather* el cual es definido una vez que se obtiene el *computedValue* de una propiedad, el valor heredado *iamtheroot*, un valor local *reglasEmparejadas* el cual es definido en el proceso de emparejar los selectores con el nodo, y una propiedad '*p*' que se recibe de la lista de propiedades.

Para encontrar el *computedValue* de una propiedad, se necesita casi los mismos valores que para *specifiedValue*. Lo que falta es encontrar el valor *replaced* de un nodo.

Un nodo es *replaced* sólo si el nodo con etiqueta es *replaced*, caso contrario es *Nothing* (del tipo de dato *Maybe*):

```
ATTR Nodo [ | | replaced : { Maybe Bool } ]
SEM Nodo
  | NTag   lhs.replaced = Just @replaced
  | NTexto lhs.replaced = Nothing
```

Con esta información se puede llamar a la función *doComputedValue*:

```
SEM NTree
  | NTree loc.computedValueProps = Map.map (doComputedValue @lhs.iamtheroot
                                                            @lhs.propsFather
                                                            @loc.specifiedValueProps
                                                            @nodo.replaced
                                                            False
                                            ) @loc.specifiedValueProps
```

Código UUAGC 10: Llamando a la función *doComputedValue*

La información que se enviará a *doComputedValue* es: un valor heredado que representa si el nodo es *Root*, un valor heredado que representa la lista de propiedades del nodo padre, un valor local que representa la lista de propiedades locales, un valor sintetizado que representa si el nodo es *replaced* y un valor *Bool* que indica si se revisará los elementos pseudo.

Vea que se está utilizando la función *map* de la estructura *Map*, para construir la nueva estructura *Map* de propiedades. Esta nueva estructura es almacenada en la variable local *computedValueProps*.

Una vez que se tiene la nueva lista de propiedades, se debe compartir con los nodos hijos, para esto, se define el atributo heredado *propsFather*:

```
ATTR NTree NTrees [propsFather : { Map.Map String Property } | | ]
SEM NTree
  | NTree ntrees.propsFather = @loc.computedValueProps
SEM NRoot
  | NRoot ntree.propsFather = Map.empty
```

Entonces, en cada nodo *NTree*, se comparte la nueva estructura (*computedValueProps*) con los nodos hijos (*ntrees*). Pero para el caso del nodo *Root*, la lista de propiedades del padre es una estructura *Map* vacía.

Capítulo 7

Estructura de Formato

Cada elemento de la estructura Rosadelfa/NTree, dependiendo de la propiedad *display* de CSS, genera un *box* para que sea renderizado.

Luego, el *box* generado debe ser acomodado de acuerdo al posicionamiento estático de CSS. Una vez acomodado, se debe calcular la posición y dimensión correspondiente para cada *box*, de manera que se pueda usar esa información para su renderización.

El cálculo de la dimensión de cada *box* es una tarea definida en la especificación de CSS. Esta tarea depende de varias propiedades de CSS, entre ellas: *line-height*, *vertical-align*, *width* y *height*.
De la misma manera, el cálculo de la posición para cada *box* depende de un contexto de formato, el cual esta definido en el posicionamiento estático de CSS. Básicamente, antes de calcular las posiciones se deben formar bloques que contengan líneas de *boxes*.

En este capítulo se ha denominado *Estructura de Formato* (*Formatting Structure* [*FSTree*]) a los *boxes* generados. El cálculo de la posición y dimensión se ha dividido en dos fases, de manera que también se tiene dos estructuras de formato: *FSTreeFase1* y *FSTreeFase2*.
En las siguientes secciones se inicia describiendo los tipos de datos para las dos fases de la estructura de formato. Luego se describe la generación de la estructura de formato desde la estructura NTree y finalmente se describe cada fase de la estructura de formato.

En este capítulo se utilizará la herramienta *UUAGC*(Swierstra, s.f.-a) para describir los atributos y semánticas. También se utilizará la librería *WxHaskell* para la parte de generación de ventanas.

7.1. Tipos de datos

En el Código UUAGC 1 del Capítulo 6 se muestra los tipos de datos para la estructura *NTree*. A continuación, sólo se mostrará los tipos de datos para la estructura de formato.

7.1.1. FSTreeFase1

Los tipos de datos para la estructura de formato de fase 1 son similares a los del *NTree*. Básicamente porque ambos guardan la misma información.

La nueva estructura *BoxTree* descrita en el Código UUAGC 11, ya no tiene nodos, pero guarda la información del nodo directamente en el árbol.

Se tiene dos tipos de *BoxTree* en la nueva estructura. Para representar el texto se tiene a *BoxText*, el cual guarda un nombre, la lista de propiedades, la lista de atributos y el texto. Se utiliza este tipo no sólo para representar los nodos de texto de un *NTree*, sino también para representar los nodos etiquetas que sólo contienen texto, es por eso que se tiene una lista de propiedades y atributos.

Para representar los nodos etiquetas se tiene a *BoxContainer*, el cual guarda el nombre de la etiqueta, el contexto del contenedor, la lista de propiedades, el tipo del contenedor, la lista de atributos y los hijos del contenedor.

```
DATA BoxTree
  | BoxContainer name : String
                 fcnxt  : {FormattingContext}
                 props  : {Map.Map String Property}
                 bRepl  : Bool
                 attrs  : {Map.Map String String}
                 boxes  : Boxes
  | BoxText name : String
             props : {Map.Map String Property}
             attrs : {Map.Map String String}
             text  : String
TYPE Boxes = [BoxTree]
```

Código UUAGC 11: Tipo de dato para el *FSTreeFase1*

La información que se guarda en este tipo es casi la misma que en los del *NTree*, con la excepción del formato de contexto del contenedor, descrita en el Código Haskell 65.

```
data FormattingContext
  = InlineContext | BlockContext | NoContext
    deriving Show
```

Código Haskell 65: Tipo de dato para representar el contexto de formato

7.1.2. FSTreeFase2

La nueva estructura para fase 2 descrita en el Código UUAGC 13, tiene grandes cambios con respecto a la de fase 1. En fase 1 se manejaba una lista de *boxes* como hijos de un contenedor, pero en fase 2 se tiene o bloques de ventanas que se acomodan uno debajo del otro o líneas de ventanas que se acomodan una seguida de la otra. El tipo de dato *Element* del Código UUAGC 12 muestra este comportamiento.

```
DATA Element
  | EWinds winds : WindowTrees
  | ELines  lines  : Lines
  | ENothing
TYPE WindowTrees = [WindowTree]
TYPE Lines       = [Line]
DATA Line
  | Line winds     : WindowTrees
```

Código UUAGC 12: Tipo de dato Element

Además, un *BoxTree* puede ser dividido en varias partes para ser renderizado en varias líneas. Así, para representar a que parte pertenece una ventana, se utiliza el tipo de dato *TypeContinuation* del Código Haskell 66.

```
data TypeContinuation
  = Full | Init | Medium | End
    deriving (Show, Eq)
```

Código Haskell 66: Tipo de dato *TypeContinuation*

```
DATA WindowTree
  | WindowContainer name : String
                    fcnxt : {FormattingContext}
                    props : {Map.Map String Property}
                    attrs : {Map.Map String String}
                    tCont : {TypeContinuation}
                    bRepl : Bool
                    elem  : Element
  | WindowText name  :  String
               props :  {Map.Map String Property}
               attrs :  {Map.Map String String}
               tCont :  {TypeContinuation}
               text  :  String
```

Código UUAGC 13: Tipo de dato para el *FSTreeFase2*

7.2. Generar resultado para Fase 1

Una vez que se procesa las hojas de estilo y se asigna valores a *specifiedValue* y *computedValue* en el *NTree*, se debe proceder con la fase 1. Para esto, se debe generar una estructura *FSTreeFase1* desde el *NTree*.

7.2.1. Generar un *BoxText*

Lo más sencillo es generar un *BoxText* desde un *NText*. Se recibe la lista de propiedades y el contenido y se construye el *BoxText* con una lista vacía de atributos:

```
genTextBox props str
  = Just $ BoxText "text" props Map.empty str
```

7.2.2. Generar un *ReplacedBox*

Si se tiene un nodo que es *replaced*, se genera un *BoxContainer* que no tiene contexto, ni tampoco hijos, porque el contenido de este elemento es externo:

```
genReplacedBox nm attrs props
  = Just $ BoxContainer nm NoContext props True attrs []
```

7.2.3. Generar un *InlineBox*

Para el caso de generar un *InlineBox* se debe realizar 2 comprobaciones:

- Que ninguno de sus hijos tenga *display* = `"block"`. Esto porque no se está dando soporte al manejo de bloques en elementos *inline*, es más no se esta dando soporte a *display* = *"inline-block"*.

  Para verificar este valor, se ha creado la función *isThereBlockDisplay*, que verifica si algún elemento tiene *display* = `"block"`:

```
isThereBlockDisplay bx
  = case bx of
      BoxContainer _ _ props _ _ _ → verifyProperty "display" "block" props
      BoxText      _   props _ _   → verifyProperty "display" "block" props
```

- También se debe comprobar si el hijo del contenedor es simplemente texto, si es así, se puede crear un *BoxText* extrayendo el contenido del texto.

Con estas consideraciones, se muestra la implementación para crear un *inlineBox*:

```
genInlineBox nm attrs props boxes
  = case boxes of
      [BoxText "text" _ _ str]
        → mkBoxText str
      otherwise
        → if any isThereBlockDisplay boxes
          then error $ "Unsupported feature: inline block at node:" ++ nm
          else  mkBoxContainer InlineContext
  where mkBoxContainer context
          = Just $ BoxContainer nm context props False attrs boxes
        mkBoxText str
          = Just $ BoxText nm props attrs str
```

Vea que en la implementación de la función *genInlineBox*, se consideró las 2 comprobaciones.

7.2.4. Generar un *BlockBox*

Para el caso del *BlockBox* el contexto puede ser *InlineContext* o *BlockContext*. Para que sea *InlineContext* todos los hijos deben tener *display* = `"inline"` y de la misma forma, para que sea *BlockContext* todos los hijos deben tener *display* = `"block"`.

Muchas veces, no todos los hijos del contenedor tienen *display* = `"block"`, en esos casos se debe agrupar todos los elementos que son *inline* e insertarlos como hijos en un nuevo contenedor *block*.

Se utiliza la función *funGroupCompare* para agrupar todos los elementos *inline*:

```
funGroupCompare = (∧) `on` (λbx → verifyProperty "display" "inline" (getProps bx))
  where getProps bx = case bx of
                        BoxContainer _ _ props _ _ _ → props
                        BoxText      _   props _ _   → props
```

Y se utiliza la función *toBoxContainer* para convertirlos en bloques:

```
toBoxContainer broot sprops replaced lst@(bx : bxs)
  = case bx of
      BoxContainer nm _ props _ _ _
        → if verifyProperty "display" "block" props then bx
          else BoxContainer (nm ++ "???") InlineContext
                            (doInheritance broot propiedadesCSS sprops replaced)
                            ...
      BoxText nm props _ _
        → if verifyProperty "display" "block" props then bx
          else BoxContainer ...
```

La función *doInheritance* se encarga de generar una nueva lista de propiedades para el nuevo contenedor. La forma de generar las propiedades es obteniendo las propiedades heredables del contenedor donde se encuentra.

```
doInheritance broot listProps fatherProps replaced
  = let lprops      = map (λp → (getPropertyName p
                                , applyInheritance broot fatherProps p
                                )
                           ) listProps
        inhProps    = Map.fromList lprops
        blockProps  = Map.adjust adjustFunction "display" inhProps
    in Map.map (doComputedValue broot fatherProps blockProps replaced False) blockProps
  where adjustFunction
          = adjustPropertyValue (λpv → pv {specifiedValue = KeyValue "block"})
```

La función *applyInheritance* es similar a seleccionar un valor para el *specifiedValue* de una propiedad. Esta función esta definida en el módulo *Property*.

Con todo esto, la función *genBlockBox* es de la siguiente manera:

```
genBlockBox nm attrs props boxes broot
  = if any isThereBlockDisplay boxes
    then let listGrouped       = groupBy funGroupCompare boxes
             listBoxContainer = map (toBoxContainer broot props Nothing) listGrouped
         in mkBoxContainer BlockContext listBoxContainer
    else mkBoxContainer InlineContext boxes
  where mkBoxContainer context children
          = Just $ BoxContainer nm context props False attrs children
        mkBoxText str
          = Just $ BoxText nm props attrs str
```

Lo siguiente es crear los *boxes* de acuerdo a la propiedad *display* y el nodo. Si *display* = `"none"`, entonces no se genera nada, caso contrario, se va generando los *boxes* de acuerdo al tipo. El Código Haskell 67 describe la función para generar *boxes*.

```
genBox (NText str) props boxes _
  = genTextBox props str
genBox (NTag nm replaced attrs) props boxes broot
  = if replaced then genReplacedBox nm attrs props
    else case computedValue (props `get` "display") of
           KeyValue "none"   → Nothing
           KeyValue "inline" → genInlineBox nm attrs props boxes
           KeyValue "block"  → genBlockBox nm attrs props boxes broot
```

Código Haskell 67: Generación de Boxes

Finalmente, se utiliza UUAGC para llamar a las funciones que se ha definido. Se ha creado el atributo sintetizado *fstree* para recolectar todos los *boxes* generados:

```
ATTR NRoot NTree [ | | fstree : {Maybe BoxTree} ]
SEM NTree
  | NTree lhs.fstree
    = let boxes = catMaybes@ntrees.fstree
      in genBox@node.self @loc.computedValueProps boxes @lhs.iamtheroot
ATTR NTrees [ | | fstree USE {:} {[]} : {[Maybe BoxTree]} ]
```

7.3. Estructura de Formato, Fase 1

Ahora que se ha generado el *FSTreeFase1* se comenzará a procesar la estructura de formato de fase 1.

Esta fase se encarga básicamente de construir líneas de *boxes* para todos los contenedores que tengan un contexto de *InlineContext*. Para generar las líneas se necesita saber las dimensiones de todos los *boxes*, especialmente del contenedor que tiene el contexto deseado.

Y para conocer las dimensiones de un *box* se necesita avanzar un nivel más en los valores de la propiedad. Es decir, se necesita obtener el *usedValue* de una propiedad.

En esta sección, primeramente se construirá el *usedValue* para todas las propiedades, luego se generará las líneas y finalmente se generará el tipo de dato *FSTreeFase2* para continuar con el proceso de formatear la estructura.

7.3.1. Construir el *usedValue*

Para construir el *usedValue* se utiliza la función *doUsedValue* (Sección 6.3.4, página 67) definida en el módulo *Property*.

Primero, se debe definir un atributo heredado *iamtheroot* de tipo *Bool* para determinar quien es el nodo *Root*:

```
ATTR BoxTree Boxes [iamtheroot : Bool | | ]
SEM BoxRoot
  | BoxRoot boxtree.iamtheroot = True
SEM BoxTree
  | BoxItemContainer boxes.iamtheroot = False
  | BoxContainer     boxes.iamtheroot = False
```

Segundo, se debe llamar a la función *doUsedValue* y guardar el resultado en la variable local *usedValueProps*. El Código UUAGC 14 describe la forma de calcular el *usedValue*.

```
SEM BoxTree
  | BoxContainer
      loc.usedValueProps
        = Map.map (doUsedValue @lhs.iamtheroot
                               (toTupleFloat (@lhs.cbSize))
                               @lhs.propsFather
                               @props
                               @attrs
                               @bRepl) @props
  | BoxText
      loc.usedValueProps
        = Map.map (doUsedValue @lhs.iamtheroot
                               (toTupleFloat (@lhs.cbSize))
                               @lhs.propsFather
                               @props
                               @attrs
                               False) @props
```

Código UUAGC 14: Calcular el *usedValue* de una Propiedad

En el Código UUAGC 14 se hace uso del atributo sintetizado *cbSize* que contiene las dimensiones (*Size*) del contenedor *box*, el cual se recibe desde afuera y es compartido con todo el árbol *FSTreeFase1*:

```
ATTR BoxRoot BoxTree Boxes [cbSize : {(Int, Int)} | | ]
```

Finalmente, las propiedades que se obtiene deben ser compartidas con los nodos hijos, para esto se ha creado el atributo heredado *propsFather*, que es inicializado en el *BoxRoot*:

```
ATTR BoxTree Boxes [propsFather : { Map.Map String Property} | | ]
SEM BoxTree
  | BoxItemContainer boxes.propsFather = @loc.usedValueProps
  | BoxContainer     boxes.propsFather = @loc.usedValueProps
SEM BoxRoot
  | BoxRoot boxtree.propsFather = Map.empty
```

7.3.2. Construir líneas

Otra de las tareas importantes de esta fase es la construcción de líneas de ventanas en contenedores *block* que tengan un contexto de formato *InlineContext*.

La forma de construir líneas es convirtiendo el árbol de *boxes* en una lista de elementos, donde cada elemento es atómico (que no se puede dividir y que se debe renderizar como tal), luego se debe aplicar un algoritmo que acomode los elementos en líneas de un determinado tamaño. Una vez que se tiene las líneas, se debe volver a convertir cada línea (lista de elementos)

a un árbol de ventanas.

Por ejemplo, sí se tiene un elemento párrafo (‘*p*’) que es *block* y tiene un contexto de *InlineContext*, así como en el siguiente ejemplo:

```
<p>
    <span> uno dos </span>
    tres cuatro cinco
    <span> seis </span>
</p>
```

Puede ser representado en forma de árbol:

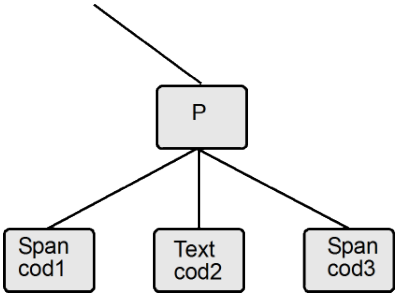


Figura 7.1: Representación en forma de árbol

En la Figura 7.1 se muestra que se asigna un código a cada nodo, ese código es importante para volver a construir el árbol después de convertirlos en líneas.

Si se convierte el árbol en una lista de elementos atómicos, la Figura 7.1 cambiaría a:

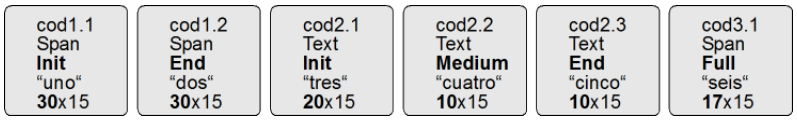


Figura 7.2: Lista de elementos atómicos

Ahora, cada nodo ha sido separado en partes atómicas, por ejemplo, en la Figura 7.2, el elemento *span* ha sido separado en 2 partes y el *texto* ha sido separado en 3 partes.

Además, cada elemento tiene un nuevo código y un valor que identifica la posición del elemento con respecto al contenedor (esto es el *TypeContinuation*). También se tiene las dimensiones que ocupará cada elemento atómico en el contenedor.

Luego se aplica un algoritmo para acomodar cada elemento en una línea de un tamaño fijo. Por ejemplo, sí el tamaño es 30, el ejemplo se vería de la siguiente manera:

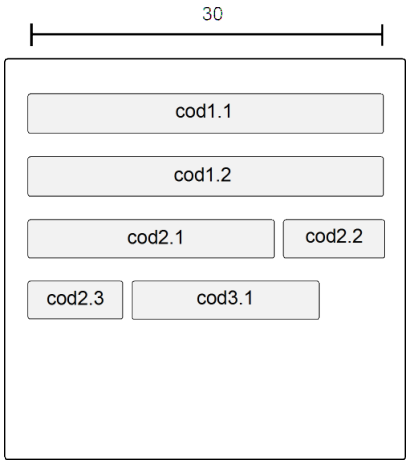


Figura 7.3: Acomodar los elementos en líneas

El algoritmo para acomodar los elementos es relativamente sencillo, simplemente se va construyendo líneas hasta que ya no haya más elementos.

En el Código Haskell 68 se muestra la definicion de la función *inlineFormatting*, la cual recibe una lista de elementos, el tamaño fijo de la línea (*width*), un valor entero que representa la longitud de sangría para la primera línea y el tamaño que ocupa un espacio.

Si la lista de elementos es vacía, termina el algoritmo, caso contrario, se va construyendo las líneas de manera acumulativa.

```
inlineFormatting :: [ElementList] → Int → Int → Int    → [[ElementList]]
inlineFormatting   []             _     _     _      = []
inlineFormatting   lst            width indent space = doInline width indent space lst
```

Código Haskell 68: Algoritmo para acomodar los elementos en líneas, 1

La función *doInline* del Código Haskell 68 construye las líneas de manera recursiva llamando a la función *buildLine* el cual devuelve una tupla con la línea y la lista de elementos que aún no se ha consumido. El Código Haskell 69 muestra su implementación.

La primera vez que se llama a la función *doInline* del Código Haskell 69 se crea una nueva longitud para la línea restando la cantidad de que ocupa la sangría, para emular el comportamiento de la sangría en el contenido de la línea. Una vez que se devuelve una línea, el valor de la sangría para todas las siguientes líneas cambia a 0, porque la sangría sólo se aplica a la primera línea.

```
doInline w indent s [] = []
doInline w indent s list = let newWidth = w − indent
                               (line, rest) = buildLine list (newWidth + s) 0 s
                           in line : doInline w 0 s rest
```

Código Haskell 69: Algoritmo para acomodar los elementos en líneas, 2

La función *buildLine* del Código Haskell 69 construye una línea de manera acumulativa, recibe la lista de elementos, la longitud de la línea, una longitud temporal (que inicialmente es cero) y la longitud que ocupa un espacio. El Código Haskell 70 muestra su implementación.

Si la lista de elementos es vacía, se termina todo con lista vacía, caso contrario se debe verificar sí el primer elemento de la lista puede acomodarse en la línea actual, si es así se inserta el elemento en la línea y se vuelve a llamar a *buildLine* con la nueva longitud temporal incrementada; pero, sí no se acomoda, entonces se devuelve una lista vacía y la lista de elementos sin modificar.

```
buildLine [] _ _ _
  = ([], [])
buildLine nlst@(e : es) w wt space
  = let len = wt + getLength e + space
    in if len ⩽ w
       then let (ln, rs) = buildLine es w len space
            in (e : ln, rs)
       else ([], nlst)
```

Código Haskell 70: Algoritmo para acomodar los elementos en líneas, 3

El algoritmo presentado, puede ingresar en un bucle recursivo si el tamaño de la línea no es suficiente para un elemento. En el código del proyecto se ha modificado el algoritmo haciendo una verificación de este caso antes de llamar a la función *inlineFormatting*. De manera que, si alguno de los elementos es más largo que la longitud de la línea, entonces se modifica la longitud de la línea para abarcar al elemento más largo.

Otra modificación al algoritmo es el de tener elementos que obliguen a hacer el rompimiento de línea. Cuando ocurra este tipo de elementos en la lista, simplemente se debe terminar la construcción de la línea y comenzar a construir una nueva línea.

Estos elementos son necesarios para dar soporte a elementos como el *br* o a la propiedad *whitespace* de CSS.

Finalmente, se debe volver a reconstruir el árbol, pero en vez de reconstruir el mismo árbol (*Fase1*) se debe reconstruirlo directamente para *Fase2*.

Para reconstruir el árbol se utiliza los códigos que se había puesto antes de dividirlos en partes. A continuación se muestra la siguiente figura que ejemplifica la reconstrucción para fase 2 del ejemplo de la Figura 7.3:

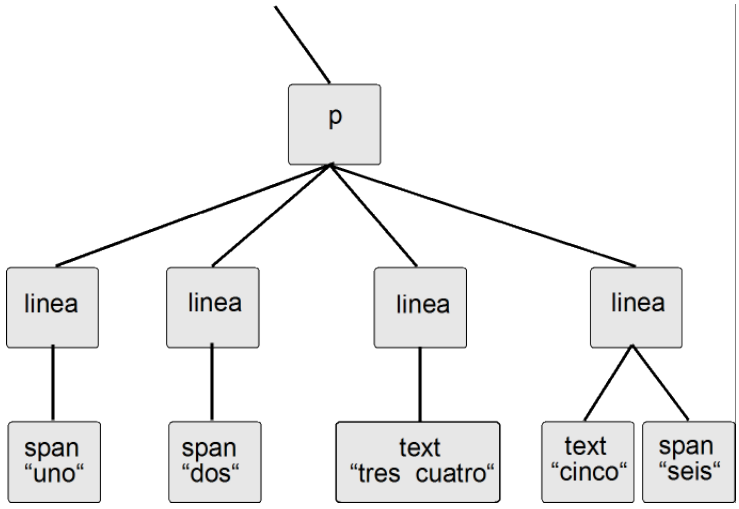


Figura 7.4: Ejemplo de reconstrucción del árbol para fase 2

Para terminar, se muestra la parte del código con UUAGC el cual hace el trabajó de llamar a la función *applyWrap*.

La función *applyWrap* llama a la función *inlineFormatting* la cual esta definido en el Código Haskell 68.

```
SEM BoxTree
  | BoxContainer loc.lines
      = let indent = toInt $ unPixelUsedValue $ @loc.usedValueProps `get` "text-indent"
        in applyWrap @loc.width indent 6 @boxes.elements
```

Código UUAGC 15: Aplicar el algoritmo para acomodar los elementos en líneas

Esta función se aplica en cada *BoxContainer* del árbol. Se podría solamente aplicar en los elementos que son *block* con *InlineContext* pero se deja el trabajó al lenguaje con evaluación perezosa, el cual se encarga de evaluar sólo lo que es necesario.

7.3.3. Generar resultado para Fase 2

Finalmente, como la última tarea de *Fase1*, se debe generar un resultado para continuar con la *Fase2*.

Para esta parte se debe hacer una generación que dependa del valor de la propiedad *display* y del contexto de un contenedor. Por ejemplo, sí se tiene un elemento que es *block* que tiene *InlineContext*, se debe generar un contenedor que tenga como hijos a las líneas que se ha construido en la anterior sub-sección. Pero si se tiene un contexto *BlockContext* sus hijos serán una lista de ventanas y no de líneas.

Se crea el atributo sintetizado *boxtree* que inspecciona la propiedad *display* y el contexto del contenedor. Y sí es *block* e *InlineContext* se usa las líneas (variable local **loc**.*lines*), pero sí es

BlockContext se usa las ventanas (atributo sintetizado *boxes.boxtree*). El Código UUAGC 16 muestra la implementación para esta parte.

```
ATTR BoxRoot BoxTree [ | | boxtree : { WindowTree } ]
SEM BoxTree
  | BoxContainer lhs.boxtree
      = case computedValue (@loc.usedValueProps 'get' "display") of
          KeyValue "block"
            → case @fcnxt of
                InlineContext → WindowContainer @name
                                                InlineContext
                                                @loc.usedValueProps
                                                @attrs Full
                                                @bRepl
                                                (ELines @loc.lines)
                otherwise     → WindowContainer @name
                                                @fcnxt
                                                @loc.usedValueProps
                                                @attrs Full
                                                @bRepl
                                                (EWinds @boxes.boxtree)
          KeyValue "inline"
            → case @fcnxt of
                InlineContext → WindowContainer @name
                                                InlineContext
                                                @loc.usedValueProps
                                                @attrs Full
                                                @bRepl
                                                (EWinds @boxes.boxtree)
                otherwise     → error $ " error ..."
  | BoxText lhs.boxtree
      = WindowText @name @loc.usedValueProps @attrs Full @text
```

Código UUAGC 16: Generar resultado para Fase 2

7.4. Estructura de Formato, Fase 2

La tarea para formatear la estructura de fase 2 es generar una ventana renderizable que tenga dimensiones y posiciones de acuerdo al contexto del contenedor.

Se inicia esta sección generando las dimensiones para cada ventana, luego se asigna posiciones y finalmente se genera las ventanas renderizables.

7.4.1. Generar dimensiones para cada ventana

En esta sección se procederá a calcular el ancho (*width*) y alto (*height*) con los que una ventana se va a renderizar.

Se ha definido una lista de consideraciones los cuales guían la implementación para esta parte del proyecto:

1. Cada ventana tiene una lista de propiedades de CSS, de las cuales son importantes las propiedades *width* y *height*.

   El valor *usedValue* de cada una de estas propiedades puede tener uno de los 2 valores correctos (sí existe algún otro tipo de valor, significa que hubo algún error.):

   - *KeyValue* `"auto"`. Significa que el valor debe ser asignado por el Navegador Web.
   - *PixelNumber px*. Significa que existe una dimensión concreta para la ventana, la cual debe ser utilizada.

2. Los valores de las propiedades *width* y *height* corresponden a las dimensiones del `content-box` de un *Box* de CSS y no así a las dimensiones de todo el *box*. Para calcular la dimensión de todo el *box* se debe sumar todas las áreas del *box*.

   En el Capítulo 9 se describirá más del modelo *Box* de CSS.

3. Para sumar el ancho (width) de las dimensiones externas (`padding-box`, `border-box` y `margin-box`) con el ancho del `content-box` se debe considerar el tipo *TypeContinuation* de cada ventana. Por ejemplo, la siguiente figura, muestra un contenedor que tiene 2 líneas:

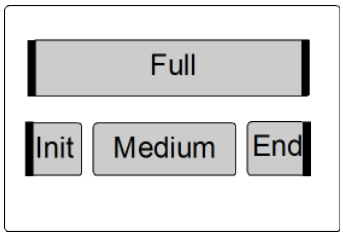


Figura 7.5: Ejemplo para *TypeContinuation*

   El *TypeContinuation* afecta el ancho de un *box*, de manera que:

   - Si es *Full*, se considera ambos lados del ancho (*width*) de la dimensión externa.
   - Si es *Init*, sólo se considera el lado izquierdo del ancho de la dimensión externa.
   - Si es *Medium*, no considera ninguno de los lados del ancho de la dimensión externa.
   - Si es *End*, sólo considera el lado derecho del ancho de la dimensión externa.

4. Para obtener el ancho y alto de un *WindowContainer* se debe considerar el tipo del contenedor de acuerdo a la propiedad *display*:

   - Si el contenedor es *Block*, el ancho del contenedor estará determinado por el valor de la propiedad *width* que siempre estará en *pixels*.

     Esto es así porque las propiedades de CSS siempre calculan su valor con respecto al ancho del contenedor donde se encuentra el elemento.

     Entonces, el ancho del contenedor sería simplemente sumar el valor de la propiedad *width* con las dimensiones externas.

Para el caso de la altura del contenedor, la propiedad *height* puede ser `"auto"` o un valor en *pixels*. Si es un valor en *pixels*, entonces simplemente se suma el valor con las dimensiones externas.

Pero sí es `"auto"`, se debe calcular la altura del contenido y sumarlo con las dimensiones externas. La altura del contenido depende del formato de contexto del contenedor.

Por ejemplo, en el Figura 7.6 se tiene un contenedor con *BlockContext* y en la Figura 7.7 se tiene un contenedor con *InlineContext*.

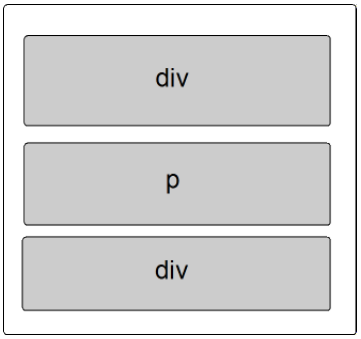


Figura 7.6: Contenedor con *BlockContext*

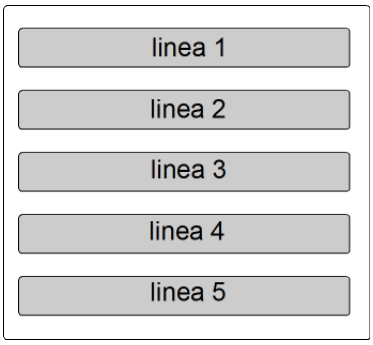


Figura 7.7: Contenedor con *InlineContext*

Entonces, el formato de contexto afecta la altura del contenedor, de manera que:

- Si es *BlockContext*, la altura del contenido es la suma de todas las alturas de las *ventanas* contenedor.
- Si es *InlineContext*, la altura del contenido es la suma de todas las alturas de las *líneas* del contenedor.

- Si el contenedor es *Inline*, entonces siempre se calculará las dimensiones, porque los valores de las propiedades *width* y *height* siempre serán `"auto"`.

  Para calcular las dimensiones del contenedor *inline*, se debe considerar el tipo del contenedor:

  - Si el elemento es *replaced*[1], se debe obtener las dimensiones del contenido externo y sumarlas con las dimensiones externas.

[1]replaced: El único elemento que se considera como *replaced* es el etiqueta *img*.

Para encontrar las dimensiones del contenido, se debe verificar las propiedades *width* y *height* del elemento. Si sus valores están en *pixels*, se utiliza como las dimensiones del contenido, caso contrario, se obtiene las dimensiones del objeto externo.

- Si el elemento no es *replaced*, entonces se trata de un elemento *inline* que tiene otros elementos *inline* como hijos. El ancho (*width*) del contenedor es la sumatoria de todos los anchos (*widths*) de los elementos hijos. Y la altura (*height*) del contenedor es la altura máxima de todos los elementos hijos.

5. Para obtener el ancho y alto de un *WindowText*, se hace lo mismo que con *WindowContainer*, pero con las suposiciones de que el formato de contexto siempre es *InlineContext* y que no existen elementos *replaced* en *WindowText*.

7.4.2. La altura de una línea

La altura (height) de una línea es calculado con respecto al *fontMetrics* y *logicalMetrics*. Estos valores son calculados usando algunas propiedades de CSS como: `font-size`, `line-height` y `vertical-align`.

El *fontMetrics* es básicamente el `font-size` de un elemento, es decir el área que ocupa un texto. Tiene un *fontTop* y *fontBottom*. El *fontTop* es la distancia desde la línea base del texto hacia arriba. Y el *fontBottom* es la distancia desde la línea base hacia abajo.

Por ejemplo, sí se tiene la siguiente entrada:

```
<p>
    Muy <big>gigante</big> texto
</p>
```

Descripción 24 Ejemplo simple

Que tiene el siguiente estilo:

```
<style>
    p { font-size: 30px
      ; line-height: 30px}

    big { font-size: 70px}
</style>
```

El *fontMetrics* de cada elemento de la Descripción 24 se muestra en la Figura 7.8.

Figura 7.8: Métricas para el ejemplo de la Descripción 24

El *logicalMetrics* corresponde a la altura de una línea. También tiene un *logicalTop* y *logicalBottom*. Estas variables son calculadas con respecto al `half-leading`, `fontMetrics` y la propiedad `vertical-align`.

El `half-leading` es calculado con respecto a las propiedades `line-height` y `font-size`. El Descripción 25 muestra la ecuación para calcular el `half-leading`, *logicalTop* y *logicalBottom*.

```
half-leading  = (line-height - font-size) / 2

logicalTop    = fontTop     + half-leading + vertical-aling

logicalBottom = fontBottom  + half-leading - vertical-align
```

Descripción 25 Ecuaciones para encontrar el *half-leading*, *logicalTop* y *logicalBottom*

Para que el valor del `vertical-align` sea utilizable en la ecuación del Descripción 25, se debe convertirlo a un número entero:

- `baseline`. Se convierte a 0 (cero).
- `super`. Se convierte a +10. Significa que el elemento se eleva 10 *px* sobre el *baseline*.
- `sub`. Se convierte a −10. Significa que el elemento baja 10 *px* debajo del *baseline*.
- `text-top`. Esta propiedad indica que se debe elevar el elemento hasta que toque la parte máxima (*top*) del texto de la línea. Para esto, se necesita saber la altura (*fontTop*) máxima y restarlo con la altura actual (*fontTop*) del elemento, el resultado de esa resta será la cantidad que se debe elevar el elemento.
- `text-bottom`. Al igual que el anterior, el elemento debe ser colocado en la parte más inferior del texto de la línea. Entonces, se resta el máximo *fontBottom* con el *fontBottom* actual del elemento, su resultado será la cantidad que se debe bajar el elemento.
- `PixelNumber`. El valor también puede ser un número pixel (positivo o negativo), donde simplemente se eleva o baja la cantidad especificada.

Como ejemplo, se procederá a calcular el *logicalTop* y *logicalBottom* para cada elemento del ejemplo de la Descripción 24:

Para `Muy`:

```
half-leading = ( 30 - 30 ) / 2 = 0

logicalTop    = 20 + 0 + 0 = 20
logicalBottom = 10 + 0 - 0 = 10
```

Para `gigante`:

```
half-leading = ( 30 - 70 ) / 2 = -40 / 2 = -20

logicalTop    = 50 + (-20) + 0 = 30
logicalBottom = 20 + (-20) - 0 = 0
```

Para `texto`:

```
half-leading = ( 30 - 30 ) / 2 = 0

logicalTop    = 20 + 0 + 0 = 20
logicalBottom = 10 + 0 - 0 = 10
```

Con el *fontMetrics* y *logicalMetrics* para cada elemento, se puede calcular la *altura* de cada línea. La altura de una línea es el máximo *logicalMetrics* de la lista de ventanas de una línea. Es decir, la altura de una línea, que corresponde al máximo *logicalMetrics*, es la sumatoria del máximo *logicalTop* y máximo *logicalBottom*.

El cálculo del valor máximo de *logicalTop* y *logicalBottom* debe hacerse de manera independiente, es decir que puede corresponder a diferentes elementos.

7.4.3. Generar posiciones para las ventanas

En esta sección se generará una posición (x, y) para cada ventana. Esta posición se utilizará para posicionar a la ventana con respecto a su contenedor.

La asignación de posiciones se realiza de acuerdo al contexto de formato, si el contexto es *inline*, se incrementa la variable x de la posición con el ancho (*width*) de cada ventana, pero si el contexto es *block*, se incrementa la variable y de la posición con la altura de cada ventana.

Se ha definido, en el Código UUAGC 17, el atributo heredado *statePos* que representa el estado de la posición. Se debe llevar este atributo a todos los lugares donde se quiera asignar posiciones:

SET *WPos* = *WindowTree Element WindowTrees Lines Line*
ATTR *WPos* [*statePos* : {(*FormattingContext*, (*Int*, *Int*))} | |]

Código UUAGC 17: Atributo para el estado de una posición

Inicialmente, el atributo *statePos* no tiene contexto y se inicializa en la posición $(0, 0)$:

```
SEM WindowRoot
  | WindowRoot windowTree.statePos = (NoContext, (0, 0))
```

Posiciones de acuerdo al contexto

Si se tiene una lista de ventanas, las posiciones para cada ventana se logra sumando las dimensiones correspondientes de acuerdo al contexto.

Si el contexto es *InlineContext*, se incrementa la variable x de la posición con el ancho (*width*) de cada ventana. Pero sí el contexto es *BlockContext*, entonces se incrementa la variable y de la posición con la altura (*height*) de cada ventana. En el Código UUAGC 18 se describe este comportamiento.

```
SEM WindowTrees
  | Cons hd.statePos = @lhs.statePos
       tl.statePos  = case fst @lhs.statePos of
                        InlineContext → let (x, y) = snd @lhs.statePos
                                        in (InlineContext, ((fst @hd.size) + x + 6, y))
                        BlockContext  → let (x, y) = snd@lhs.statePos
                                        in (BlockContext, (x, (snd @hd.size) + y))
```

Código UUAGC 18: Asignar posiciones de acuerdo al contexto

El atributo sintetizado @*hd.size* del Código UUAGC 18 es una tupla que guarda las dimensiones de la ventana, el primero es el ancho y el segundo es la altura. Y el número 6, es la cantidad que un espacio ocupa en *pixels*.

Posiciones en una lista de líneas

Si se tiene una lista de líneas, se debe incrementar la variable y de la posición con la altura de cada línea. El Código UUAGC 19 muestra esta parte de la implementación.

```
SEM Lines
  | Cons hd.statePos =  @lhs.statePos
       tl.statePos  = let (x, y)  = snd @lhs.statePos
                          newY = y +  @hd.lineHeight
                      in (fst @lhs.statePos, (x, newY))
```

Código UUAGC 19: Asignar posiciones a una lista de líneas

En el Código UUAGC 19, @*hd.lineHeight* corresponde a la altura de cada línea, que es la sumatoria de: *maxLogicalTop* y *maxLogicalBottom*.

Asignar posiciones a ventanas

Para guardar la posición de una ventana se ha creado la variable local *position*, que almacena una tupla que representa las posiciones (x, y) de la ventana.

Para un *WindowText*, se recibe el atributo heredado *statePos*, del cual sólo se necesita las variables (x, y). La variable y puede ser afectada por la propiedad `vertical-align` de CSS, la cual es aplicada sólo a elementos *inline*.

La nueva posición y es calculada a través de la ecuación de la Descripción 26. La implementación para calcular la nueva posición y esta descrita en el Código UUAGC 20.

```
y2 = maxLogicalTop - (vertical-align + fontTop + externalSizeTop)
```

Descripción 26 Formula para calcular la posición y en un elemento de texto inline

```
| WindowText
    loc.position
      = let (x, y1) = snd @lhs.statePos
            y2      = case @loc.vdisplay of
                        "inline" → fst @lhs.maxLogicalMetrics
                                   − ( @loc.verticalAlign
                                     + @loc.fontTop
                                     + (thd4 @loc.extSize)
                                     )
                        otherwise → 0
        in (x, y1 + y2)
```

Código UUAGC 20: Asignar posición a un *WindowText*

Para el caso del *WindowContainer*, se utiliza directamente la posición que se recibe del *statePos*. Como el contenedor puede tener hijos, se debe generar un nuevo *statePos* para los hijos.

El nuevo *statePos* debe tener el contexto del contenedor y la posición donde se comienza a dibujar en el contenedor. La implementación para esta parte se describe en el Código UUAGC 21.

```
| WindowContainer
    loc.position    = snd @lhs.statePos
    elem.statePos = let pointContent = getTopLeftContentPoint @tCont @props
                    in (@fcnxt, pointContent)
```

Código UUAGC 21: Asignar posición a un *WindowContainer*

La función *getTopLeftContentPoint* del Código UUAGC 21 obtiene la nueva posición donde se comienza a dibujar. Esta posición corresponde a la esquina superior izquierda del

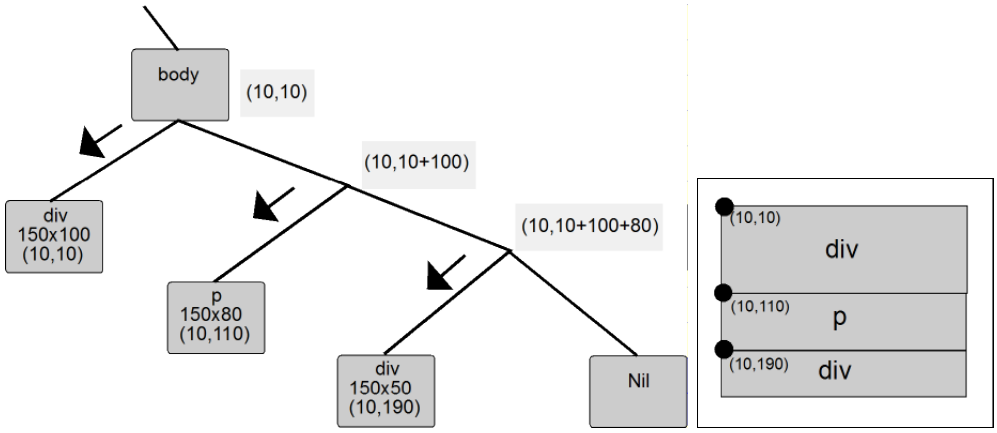


Figura 7.9: Posicionamiento con *BlockContext*

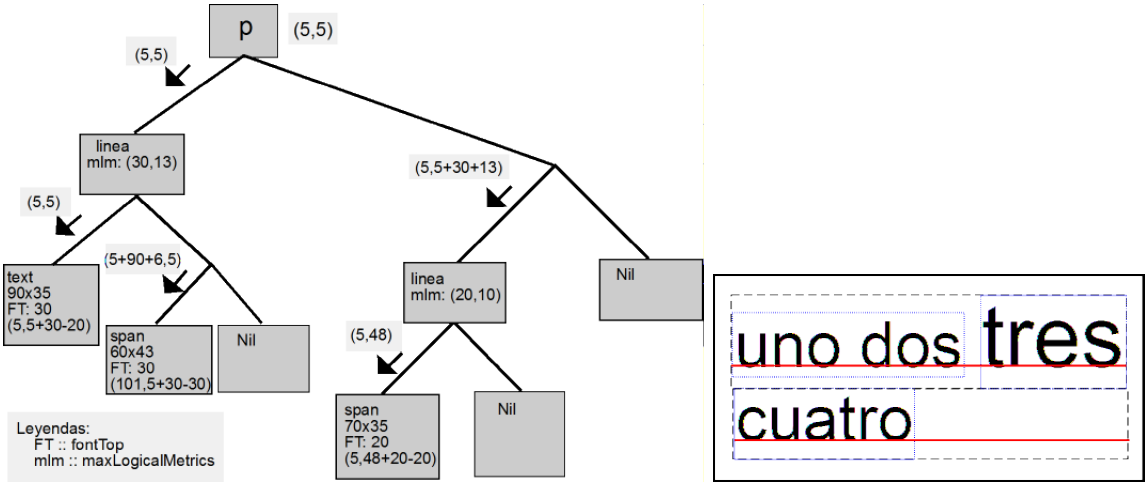


Figura 7.10: Posicionamiento con *InlineContext*

`content-box` del contenedor.

Para terminar esta parte, se presenta las figuras: Figura 7.9 y Figura 7.10, que ejemplifican la asignación de posiciones.

7.4.4. Generar ventanas renderizables

Finalmente, como resultado de formatear toda la estructura, se debe generar las ventanas renderizables.

Cada elemento de *FSTreeFase2* genera una ventana renderizable, es decir un *box*. En el Capítulo 9 se dará más detalles del *box*. Por ahora sólo se conoce que se dispone de las funciones *boxContainer* y *box* para la creación de un *box*. En el Código Haskell 71 se muestra los tipos de las funciones *boxContainer* y *box*.

```
boxContainer :: Window a                    -- ventana padre
             → (Int, Int)                   -- posicion
             → (Int, Int)                   -- dimencion
             → TypeContinuation
             → Map.Map String Property      -- propiedades de CSS
             → Map.Map [Char] String        -- atributos
             → Bool                         -- replaced
             → IO (ScrolledWindow ())
box :: String                               -- contenido
    → Window a                              -- ventana padre
    → (Int, Int)                            -- posicion
    → (Int, Int)                            -- dimencion
    → TypeContinuation
    → Map.Map String Property               -- propiedades de CSS
    → Map.Map [Char] String                 -- atributos
    → Bool                                  -- replaced
    → IO (ScrolledWindow ())
```

Código Haskell 71: Funciones para construir un *box*

Cada *WindowContainer* genera un *boxContainer* y cada *WindowText* genera un *box*. La ventana padre de ambos *boxes* es el contenedor padre donde se encuentra el elemento.

La idea para generar los *boxes* es generar funciones que esperen una ventana y produzcan acciones de renderización. Para esto, se ha definido el atributo sintetizado *result*, que tiene el tipo mencionado:

```
ATTR WindowRoot WindowTree [ | | result : { WX.Window a → IO () } ]
```

El Código UUAGC 22 describe la forma de generar un *box* para un *WindowText*.

```
| WindowText
    lhs.result = λcb → do box @text cb
                           @loc.position
                           @loc.size
                           @tCont
                           @props
                           @attrs
                           False
                       return ()
```

Código UUAGC 22: Generar un *box* para *WindowText*

En el Código UUAGC 22 se crea una función *lambda* que recibe un '*cb*' (*Container Box*) y devuelve la acción para renderizar el *WindowText*.

El Código UUAGC 23 describe la forma de generar un *box* para el *WindowContainer*.

```
| WindowContainer
    lhs.result = λcb → do cbox ← boxContainer cb @loc.position
                                                 @loc.size
                                                 @tCont
                                                 @props
                                                 @attrs
                                                 @bRepl
                          mapM_ (λf → f cbox) @elem.result
```

Código UUAGC 23: Generar un *box* para *WindowContainer*

En el Código UUAGC 23 el *WindowContainer* es padre de sus hijos, esto significa que se debe compartir la ventana creada con todos los hijos. Se utiliza la función $mapM\_$ de Haskell para compartir la ventana creada por el contenedor con todos los hijos.

También se define el atributo sintetizado *result* para recolectar todas las acciones de la lista de hijos:

```
ATTR Element Lines Line [ | | result USE {++} {[]} : {[WX.Window a → IO ()]}]
ATTR WindowTrees [ | | result USE {:} {[]} : {[WX.Window a → IO ()]}]
```

Con todo esto, ya se tiene generado todas las ventanas para su renderización.

Finalmente, un detalle que no se ha mostrado hasta ahora, es la generación de eventos de *clic* para un contenedor. Los eventos son generados utilizando las funciones de la librería WxHaskell. Por ejemplo, si se quiere generar un evento de *clic* de un URL, se debe disponer de una función que va a procesar el *clic*:

```
WX.set cbox [on focus := onClick @lhs.goToURL url]
```

Código UUAGC 24: Generación de eventos de *clic*

La función que procesa el *clic* es la función *onClick*, que llama a la función *goToURL* (su argumento) con el nuevo *url* que se quiere renderizar:

```
onClick function url bool
  = if bool
    then function url
    else return ()
```

Código Haskell 72: Función *onClick*

Capítulo 8

Descripción de la Implementación de las Propiedades CSS

Las propiedades de CSS son importantes porque guían la renderización de un documento. En este capítulo se describirá algunos detalles de la implementación de las propiedades de CSS a las que se dá soporte en el proyecto.

8.1. Programando las propiedades de CSS

En este proyecto, se ha desarrollado un pequeño lenguaje de dominio específico dentro de Haskell (*Embeded Domain Specific Language, EDSL*) para describir el comportamiento de las propiedades de CSS.

El EDSL permite describir el comportamiento de las propiedades de CSS a través del tipo de dato *Property*. Para describir, se utiliza la función *mkProp*, que crea un *Property* sólo con la información necesaria.

La función *mkProp* (Sección 6.3.5, página 69) recibe 6 parámetros: nombre de la propiedad (de tipo *String*), información de herencia (de tipo *Bool*), valor inicial o por defecto (de tipo *Valor*), parser para los valores de la propiedad (de tipo *Parser Valor*), función para procesar el *computedValue* y función para procesar el *usedValue* de una propiedad.

3 de los parámetros de la función *mkProp*, son sencillos de declarar, pero no así el parser para los valores y las 2 funciones para *computedValue* y *usedValue*.

8.1.1. Parser para los valores de una Propiedad

Para describir el parser para los valores de una propiedad se debe utilizar la librería *uu-parsinglib*, que permite describir el parser para una gramática directamente en Haskell.

Una de las ventajas de esta forma, es que se puede utilizar toda la capacidad de la librería para describir el parser. Por ejemplo, se podría utilizar los combinadores especiales de permutaciones para describir las propiedades *shorthand* de CSS.

8.1.2. Función para el *computedValue*

También se debe especificar una función para procesar el *computedValue* de una propiedad. Esta función debe tener el tipo *FuncionComputed* (descrita en el Código Haskell 57 de la Sección 6.3.4, página 67) que indica los parámetros que recibe y el valor que debe retornar.

Para la implementación de la función se puede utilizar la información de los parámetros que se recibe, como también toda la funcionalidad de Haskell. A continuación se describe los parámetros que recibe la función:

- El primer parámetro describe si el elemento es el nodo *Root*. El valor es indicado con el tipo *Bool*.
- El segundo parámetro contiene la lista de propiedades (de tipo *Map*) del nodo padre del elemento en el que se encuentra al procesar esta propiedad.

  Se puede confiar que todos los valores de propiedad para este parámetro, tienen procesado hasta el *computedValue*. Es decir, que los valores para todas las propiedades de este parámetro tienen asignado los valores para *specifiedValue* y *computedValue*, pero no tiene asignado el *usedValue*, ni *actualValue*.

  Junto con este parámetro, se puede utilizar todas las funciones de la librería *Map* de Haskell (su documentación esta disponible en el Apéndice C). Además, también se puede utilizar la funciones definidas en la Sección 6.3.2, página 60.
- El tercer parámetro contiene la lista de propiedades (de tipo *Map*) del elemento donde se procesa esta propiedad.

  La restricción para este parámetro, es que sólo se tiene disponible el *specifiedValue* para todas las propiedades de la lista. Al igual que para el segundo parámetro, también se puede utilizar las funciones de *Map*.
- El cuarto parámetro indica si la propiedad es *replaced*, además indica si se trata de un elemento de texto (Nothing) o de uno con etiqueta (Just).
- El quinto parámetro indica si se trata de un *pseudo-elemento*, porque existen algunas propiedades que sólo se aplican a *pseudo-elementos*.
- El sexto parámetro indica el nombre la propiedad.
- El séptimo parámetro indica el *PropertyValue* de la propiedad.

Con todos los parámetros descritos, se puede implementar el comportamiento para procesar el *computedValue* de una propiedad. Cuando el *computedValue* es el mismo que el *specifiedValue*, se puede utilizar la función *computed_asSpecified*.

8.1.3. Función para el *usedValue*

Al igual que para el *computedValue*, también se debe especificar una función para procesar el *usedValue* de una propiedad. Esta función debe tener el tipo *FunctionUsed* (descrita en el Código Haskell 60 de la Sección 6.3.4, página 68).

La función para el *usedValue* recibe casi los mismo parámetros que la función para el *computedValue*, a continuación sólo se describe los parámetros que no se describieron en la anterior sub-sección.

- El segundo parámetro corresponde a las dimensiones del contenedor inicial (*Initial Container Box, icb*). Normalmente, estas dimensiones son necesarias cuando se encuentra un elemento que es *Root*.

- El quinto parámetro corresponde a la lista de atributos del elemento, que esta contenida en una estructura *Map*, lo que significa que se puede utilizar cualquiera de las funciones descritas en el Apéndice C.
- El sexto parámetro indica el tipo del elemento (*replaced*) a través de un tipo *Bool*.

8.2. Descripción de la implementación

Describir una propiedad utilizando el EDSL de la anterior sección, sólo garantiza el comportamiento básico. Para que una propiedad sea implementada por completo, se necesita implementar su forma de renderización; aunque algunas propiedades no necesitan ser renderizadas, porque ayudan en la renderización de otras propiedades, en la mayoría de los casos se necesita modificar algunas partes del proyecto.

En esta sección se mostrará los detalles más importantes de la implementación de las propiedades de CSS.

8.2.1. Propiedad display

Se tiene soporte para los siguientes valores de la propiedad *display*: *inline, block, list-item, none, inherit.*

El *computedValue* de la propiedad *display* es el mismo que el *specifiedValue*, a menos que se encuentre en el nodo *Root*.

Si se encuentra en el nodo *Root*, se utiliza la siguiente tabla de conversión:

```
Specified Value   Computed Value
"inline"       -> "block"
"run-in"       -> "block"
"inline-block" -> "block"
en otro caso   -> el valor especificado
```

El *usedValue* es el mismo que el *computedValue*.

8.2.2. Propiedades para el formato horizontal

Las propiedades para el formato horizontal son 7: *margin-left*, *border-width-left*, *padding-left*, *width*, *padding-right*, *border-width-right* y *margin-right.*

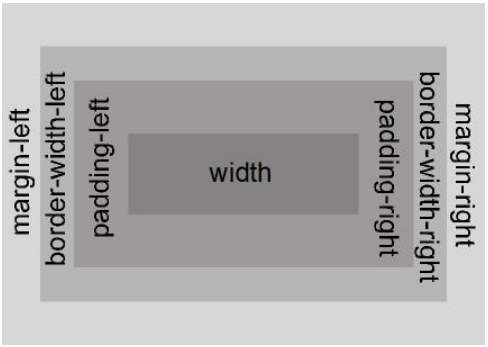


Figura 8.1: 7 Propiedades para el formato horizontal

De manera general, cada una de estas propiedades puede ser especificado con una dimensión: valor *length* o porcentaje. Existen ciertas excepciones: el *border-width* sólo puede ser *length*, el *margin-left*, *margin-right* y *width* también pueden ser especificados con un valor *auto*.

La funciones para obtener el *computedValue* de cada propiedad convierten el valor *length* a un valor *length* en *pixels*, pero si el valor del *specifiedValue* es *auto* o porcentaje, se copia directamente el valor del *specifiedValue* al *computedValue* para que las funciones del *usedValue* la procesen.

Las funciones para obtener el *usedValue* calculan el ancho en *pixels* que cada propiedad ocupará. De manera general, si el valor del *computedValue* está en *pixels*, básicamente se copia el valor al *usedValue*, pero sí está en porcentajes, entonces se convierte a valores en *pixels*. Existe un comportamiento diferente para las 3 propiedades que pueden ser *auto*, esto es: *margin-left*, *margin-right* y *width*.

Cuando el valor de alguna de las 3 propiedades es *auto*, significa que el Navegador Web se encarga de asignar un valor en *pixels* para esa propiedad.

La especificación de CSS (W3C, 2009, cap. 10) indica que existe una ecuación para asignar un valor cuando alguna de las propiedades es *auto*:

```
'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' +
'border-right-width' + 'margin-right' = width of containing block
```

Descripción 27 Ecuación para el formato horizontal

La Descripción 27 muestra la ecuación para encontrar el valor cuando alguna de las propiedades es *auto*.

Cuando sólo una de la propiedades es *auto* se puede utilizar la ecuación, para encontrar el valor para la propiedad que tiene *auto*, pero cuando dos o más propiedades tienen un valor de *auto*, o incluso cuando ninguno tiene un valor de *auto* no es posible utilizar la ecuación. Para resolver ese tipo de problemas la especificación de CSS ha definido las acciones a realizar para aplicar la ecuación de la Descripción 27. En la Descripción 28 se muestra la tabla la cual guía la implementación.

margin-left	width	margin-right	accion
auto	auto	auto	margin-left = margin-right = 0, y se utiliza la ecuacion para width
auto	auto	----	se hace margin-left = 0, y se utiliza la ecuacion para width
auto	----	auto	se utiliza la ecuacion y la cantidad obtenida se divide entre los ambos margenes
auto	----	----	se utiliza la ecuacion
----	auto	auto	se hace margin-right = 0, y se utiliza la ecuacion para width
----	auto	----	se utiliza la ecuacion
----	----	auto	se utiliza la ucuacion
----	----	----	overconstrained, se obliga a utilizar la ecuacion para margin-right

Descripción 28 Valores *auto* para el formato horizontal

8.2.3. Propiedades para el formato vertical

Al igual que para el formato horizontal, se tiene 7 propiedades para el formato vertical: *margin-top*, *border-width-top*, *padding-top*, *height*, *padding-bottom*, *border-width-bottom* y *margin-bottom*.

Así como en el formato horizontal, cada una de estas propiedades puede ser especificado con una dimensión: valor *length* o porcentaje. Existen ciertas excepciones: el *border-width* sólo puede ser *length*, el *margin-top*, *margin-bottom* y *height* también pueden ser especificados con un valor *auto*.

Las funciones para encontrar el *computedValue* para estas propiedades convierten los valores del *specifiedValue* a valores *pixel*, con la excepción de los valores *auto* y porcentaje, que son copiados directamente al *computedValue*.

Para el caso del *usedValue*, si el valor del *computedValue* está en *pixels*, se copia directamente al *usedValue*, pero sí está en porcentajes, se convierte a valores en *pixels*. Cuando el valor de alguna de las 3 propiedades (*margin-top*, *margin-bottom* y *height*) es *auto*, se tiene un comportamiento que es diferente al formato horizontal.

La especificación de CSS (W3C, 2009, cap 10) dice que cuando el valor de *margin-top*, *margin-bottom* es *auto*, el valor para el *usedValue* se convierte directamente a 0. Pero cuando el valor de la propiedad *height* es *auto*, se lo deja como *auto*, porque su altura será determinado por el contenido del contenedor.

8.2.4. Propiedades para especificar el borde de un box

Las propiedades de *border* especifican el ancho, color y estilo para cada borde de cada lado de un *box*. En las anteriores secciones de formato horizontal y vertical se describieron las propiedades para el ancho del borde (*border-width*). Ahora se describirá las propiedades del borde para el color y estilo.

La Sección 9.1.3, página 112 describe en detalle como se renderizan estas propiedades.

Propiedad border-color

Se tiene cuatro propiedades que especifican el color para cada lado: *border-color-top*, *border-color-right*, *border-color-bottom* y *border-color-left*.

Cada una de estas propiedades puede ser especificada con un *color* o *inherit*. En la Sección 5.3.1, página 43 se describe el *parser* para reconocer un color.

Propiedad border-style

Se tiene cuatro propiedades que especifican el estilo para cada lado: *border-style-top*, *border-style-right*, *border-style-bottom* y *border-style-left*.

El estilo de cada borde puede ser especificado con los valores: *hidden*, *dotted*, *dashed*, *solid*, *none* o *inherit*.

8.2.5. Propiedades para las fuentes de texto

Las propiedades para especificar las fuentes de texto son: *font-weight*, *font-style*, *font-family* y *font-size*. Estas propiedades son utilizadas al momento de renderizar el texto de un *box*, la Sección 9.3.1, página 114 describe en detalle esta parte.

En la Sección 8.2.8, página 102 se describe la propiedad *font-size*, a continuación se describirá algunos detalles de las otras propiedades para las fuentes de texto.

Propiedad font-weight

Se tiene soporte para los siguientes valores de la propiedad *font-weight*: *normal*, *bold*, *inherit*.

Propiedad font-style

Se tiene soporte para los siguientes valores de la propiedad *font-style*: *normal*, *italic*, *oblique*, *inherit*.

Propiedad font-family

Se tiene soporte para los siguientes valores de la propiedad *font-family*: fuente *string*, *serif*, *sans-serif*, *cursive*, *fantasy*, *monospace*, *inherit*.

La fuente *string* especifica una fuente de texto concreta, pero las demás (*serif*, *sans-serif*, *cursive*, *fantasy*, *monospace*) especifican una fuente de texto genérica.

Nota.- Una característica de la propiedad *font-family* es la capacidad de especificar una lista de fuentes de texto de manera que si una fuente no está disponible, se puede utilizar alguna de las restantes, e incluso llegar a utilizar una por defecto.

La librería *WxHaskell* no permite saber si una fuente de texto está disponible, esto porque existe un error en el mapeo de funciones a la librería *WxWidget*. Entonces, no es posible implementar toda la funcionalidad de la propiedad *font-family*.

8.2.6. Posicionamiento estático y relativo

La propiedad *position* de CSS permite especificar el tipo de posicionamiento que se utilizara al renderizar la página Web. Se tiene soporte sólo para posicionamiento estático y relativo, es decir: *static*, *relative* e *inherit*.

Cuando el posicionamiento es estático, el Navegador Web se encarga de asignar las posiciones, pero cuando es relativo, el Navegador Web asigna posiciones las cuales pueden ser modificadas por el autor de la página Web. Para modificar la posición asignada por el Navegador Web, el autor utiliza las propiedades *top*, *right*, *bottom*, *left* de CSS.

Las 4 propiedades especifican la longitud a mover en relación (relativo) a la posición asignada por el Navegador Web.

Entonces, las 4 propiedades sólo son utilizadas cuando el posicionamiento es relativo. Los valores de estas 4 propiedades pueden ser: un valor *lenght*, porcentaje positivo, *auto*, o *inherit*.

8.2.7. Propiedad color

La propiedad *color* define el color con que se va a renderizar el texto de un *box*. Sus valores pueden ser: *color* o *inherit*.

En la Sección 5.3.1, página 43 se describe el *parser* para reconocer un *color*.

8.2.8. Propiedades font-size, line-height y vertical-align

Las propiedades *font-size*, *line-height* y *vertical-align* son utilizadas para calcular las dimensiones de una línea, en la Sección 7.4.2, página 88 se da más detalles de la implementación de estas propiedades.

Propiedad font-size

Se tiene soporte para los siguientes valores de la propiedad *font-size*: *valor absoluto*, *valor relativo*, valor *lenght* o un valor porcentaje positivo. El *valor absoluto* puede ser: *xx-small*, *x-small*, *small*, *medium*, *large*, *x-large*, *xx-large*. El *valor relativo* puede ser: *smaller* o *larger*.

Se ha definido constantes para representar los valores absolutos del *font-size*, pero para los valores relativos se incrementa o decrementa una constante de 0,2 del valor del *font-size* para elemento anidado.

Propiedad line-height

Se tiene soporte para los siguientes valores de la propiedad *line-height*: valor *lenght* positivo, porcentaje positivo, o *inherit*.

Propiedad vertical-align

Se tiene soporte para los siguientes valores de la propiedad *vertical-align*: valor *lenght*, porcentaje, *baseline*, *sub*, *super*, *text-top*, *text-bottom* o *inherit*.

8.2.9. Generación de contenidos

La generación de contenidos sólo se aplica a *pseudo-elementos* que sean *before* / *after*:

```
p:before { content: ...}
p:after  { content: ...}
```

Descripción 29 Generación de contenidos con *pseudo-elementos*

Para especificar el contenido se utiliza la propiedad *content* de CSS.

Se tiene soporte para los siguientes valores de la propiedad *content*: *string*, *counter*, *counters*, *open-quote*, *close-quote*, *no-open-quote*, *no-close-quote*, *normal*, *none* e *inherit*.

Una vez que se describe el contenido con la propiedad *content*, el *pseudo-selector* define donde insertar el contenido generado. Por ejemplo, si el *pseudo-selector* es 'before', el contenido se insertará antes del elemento, pero sí es 'after', el contenido se insertará después del elemento.

La generación de contenidos se realiza antes de generar la estructura de formato de fase 1. Por ejemplo, si el contenido es:

- **String**, se genera un *BoxText* con el contenido especificado.
- **counter/counters**, el *box* a generar depende del estilo del *counter* / *counters*. Por ejemplo, si el estilo es *decimal*, *upper-roman* o *lower-roman*, entonces se genera un *BoxText* con el número generado por el contador. Pero si el estilo es *disc*, *circle* o *square*, entonces se genera un *BoxContainer* que es *replaced*, de manera que se renderize en una imagen.

  Tanto para *counter* y *counters* se debe especificar una variable que representa el contador. El contador es controlado por dos propiedades de CSS:

  - *counter-reset*: se encarga de inicializar una o más variables en cero u opcionalmente en un valor especificado.
  - *counter-increment*: se encarga de incrementar una o más variables en uno u opcionalmente en un valor especificado.

  Entonces, cada vez que se encuentre un *counter-reset* en las propiedades de un nodo, se inicializa un contador, y de igual manera, cada vez que se encuentre un *counter-increment* en un nodo, se incrementa en un valor al contador.

- **open-quote/close-quote/no-open-quote/no-close-quote**, dependiendo del tipo de *quote*, se genera un *BoxText* con el *quote* correspondiente o no se genera nada.

  El *quote* a insertar puede ser especificado por la propiedad *quotes* de CSS, el cual define lo que se va a insertar en cada nivel anidado de un elemento, por ejemplo:

  ```
  q {quotes: '"' '"' "'" "'" "<" ">"}
  ```

define 3 niveles de anidamiento de *quote* para el elemento *q*. Cada uno de estos *quotes* pueden ser insertados utilizando *open-quote* y *close-quote* de la propiedad *content*.

Los valores *no-open-quote* y *no-close-quote* no insertan ningún contenido, pero sirven para emular el comportamiento de colocar un *quote* para varios elementos, así como el elemento *blockquote* de HTML.

- **normal/none**, no se genera ningún contenido.

8.2.10. Listas

Para dar soporte a listas de HTML se ha creado un nuevo constructor en la estructura de formato de fase 1 y 2, esto es: *BoxItemContainer* del Código UUAGC 25 para fase 1, y *WindowItemContainer* del Código UUAGC 26 para fase 2.

```
DATA BoxTree
  | BoxItemContainer
      props : {Map.Map String Property}
      attrs : {Map.Map String String}
      boxes : Boxes
  | BoxContainer
      name : String
      fcnxt : {FormattingContext}
        ...
```

Código UUAGC 25: Tipo de dato para el ítem de las listas en fase 1

```
DATA WindowTree
  | WindowItemContainer
      marker     : {ListMarker}
      sizeMarker : {(Int, Int)}
      elem       : Element
      props      : {Map.Map String Property}
      attrs      : {Map.Map String String}
  | WindowContainer
      name : String
        ...
```

Código UUAGC 26: Tipo de dato para el ítem de las listas en fase 2

Cuando el valor de la propiedad *display* es *list-item* se genera un *BoxItemContainer* para fase 1. El comportamiento de un *BoxItemContainer* es similar a un *BoxContainer*, incluso es más sencillo. Por ejemplo el contexto de un *BoxItemContainer* siempre es *block*, así, no es necesario verificar si puede ser *inline*.

Para fase 2, además del contenido del *box*, se debe generar un nuevo *box* que represente el tipo del ítem. El tipo del ítem está controlado por la propiedad *list-style-type*, que tiene

soporte para los siguientes valores: *disc*, *circle*, *square*, *decimal*, *lower-roman*, *upper-roman*, *none*.

La forma de representar el nuevo *box* que representa el ítem en fase 2 es a través del campo *marker* (tipo de ítem) y *sizeMarker* (dimensiones del ítem) del Código UUAGC 26. El tipo del campo *marker* está descrito en el Código UUAGC 27.

```
DATA ListMarker
  | Glyph     name : String
  | Numering value : String
  | NoMarker
```

Código UUAGC 27: Tipo de dato que representa el tipo de un ítem

El Código UUAGC 28 muestra la generación del campo *marker* y sus dimensiones para un *WindownItemContainer*. Si el valor de la propiedad *list-style-type* es *disc*, *circle* o *square* se genera un tipo de dato *Glyph* con el nombre del tipo y una dimensión por defecto de 14x14 *pixels*, pero sí el tipo es *decimal*, *lower-roman* o *upper-roman* se genera un tipo de dato *Numering* con el valor *String* en el formato del tipo especificado, también se calcula las dimensiones que el *String* ocupará usando la función *getSizeBox*.

En el Código UUAGC 28 se hace referencia a un atributo heredado *counterItem* que es el número que representa el ítem en la lista de ítem del contenedor padre.

Las funciones *toRomanLower* y *toRomanUpper* del Código UUAGC 28 son funciones que convierten un número decimal a un número romano.

```
SEM BoxTree
  | BoxItemContainer
      loc.marker
        = case computedValue (@props `get` "list-style-type") of
            KeyValue "none"    → (NoMarker     , (0, 0))
            KeyValue "disc"    → (Glyph "disc", (14, 14))
            KeyValue "circle"  → (Glyph "circle", (14, 14))
            KeyValue "square"  → (Glyph "square", (14, 14))
            KeyValue "decimal"
              → let str = show @lhs.counterItem ++ "."
                (w, h, _, _) = unsafePerformIO $ getSizeBox str@lhs.cb @loc.usedValueProps
                in (Numering str, (w, h))
            KeyValue "lower-roman"
              → let str = toRomanLower @lhs.counterItem ++ "."
                (w, h, _, _) = unsafePerformIO $ getSizeBox str @lhs.cb@loc.usedValueProps
                in (Numering str, (w, h))
            KeyValue "upper-roman"
              → let str = toRomanUpper@lhs.counterItem ++ "."
                (w, h, _, _) = unsafePerformIO $ getSizeBox str @lhs.cb@loc.usedValueProps
                in (Numering str, (w, h))
```

Código UUAGC 28: Generación del tipo y dimensiones del ítem

Además de generar el tipo y dimensión para el campo *marker* y *sizeMarker* respectivamente, también se debe generar una posición para el campo *marker*. Generar una posición para el campo *marker* afecta el posicionamiento del contenido, de manera que el contenido debe recorrerse para no estar posicionado encima del campo *marker*. El Código UUAGC 29 muestra la asignación de posiciones para el campo *marker* y contenido.

Se crea una variable local *markerPosition* para guardar la posición del campo *marker*.

```
SEM WindowTree
  | WindowItemContainer
      loc.markerPosition
        = let (x, y) = snd @lhs.statePos
          in (x, y + 2)   -- para nivelar los bordes
      loc.position
        = let cposition = computedValue $ @props `get` "position"
              (x, y)    = let pos = snd @lhs.statePos
                          in (fst @sizeMarker + fst pos + 6, snd pos)
          in case cposition of
               KeyValue "static"   → (x, y)
               KeyValue "relative" → let (xdespl, ydespl) = getDesplazamiento@props
                                     in (x + xdespl, y + ydespl)
      elem.statePos = let pointContent = getTopLeftContentPoint Full@props
        in (BlockContext, pointContent)
```

Código UUAGC 29: Asignación de posiciones para *WindowItemContainer*

Finalmente, para renderizar un *WindowItemContainer* se genera dos *boxes*, uno para representar el *marker* y otro para el contenido. El Código UUAGC 30 muestra la generación de *boxes*.

```
SEM WindowTree
  | WindowItemContainer
      lhs.result = λcb →
        do case @marker of
             Numering str
               → box str cb @loc.markerPosition @sizeMarker Full @props @attrs False
             Glyph name
               → let attrs′ = Map.insert "src" (name ++ ".png")@attrs
                 in box "" cb @loc.markerPosition @sizeMarker Full @props attrs′ True
             NoMarker
               → return ()
           cbox ← boxContainer cb @loc.position @loc.size Full @props @attrs False
           mapM_ (λf → f cbox)@elem.result
```

Código UUAGC 30: Generación de *boxes* para *WindowItemContainer*

Si el tipo del *marker* es *Numering*, se genera un *box* con el *String* que contiene el *Numering*, posición y dimensión. Pero si es un *Glyph*, se genera un elemento *replaced* que contiene una imagen, posición y dimensión. (Para cada tipo del *marker* se ha generado una imagen de 14x14 *pixels*).

8.2.11. Propiedad background-color

El valor para la propiedad *background-color* puede ser cualquier color, *transparent* o *inherit*.

Esta propiedad se aplica al momento de renderizar el *box*. La Sección 9.3.1, página 117 describe la forma en que se implementa esta propiedad. Y la Sección 5.3.1, página 43 describe el *parser* para especificar un *color*.

8.2.12. Propiedad text-indent

El valor para la propiedad *text-indent* puede ser cualquier valor *length*, porcentage o *inherit*. Esta propiedad sólo se aplica a elementos *block* que tengan un contexto de formato *inline*.

El algoritmo desarrollado en el (Código UUAGC 15) de la Sección 7.3.2, página 80, que tiene soporte para un valor de sangria, muestra como se obtiene el valor de esta propiedad, el cual al mismo tiempo, es enviado como argumento a la función *applyWrap*.

En la estructura de formato de fase 2, se generan las posiciones para cada línea. El valor de esta propiedad sólo afecta a la posición x de la primera línea de un contenedor. En el Código UUAGC 31 se muestra como se modifica la posición x de la primera línea.

```
SEM Lines
  | Cons hd.statePos = let (x, y) = snd @lhs.statePos
                         newX = if @lhs.amifirstline
                                then x + @lhs.indent
                                else x
                     in (fst @lhs.statePos, (newX, y))
```

Código UUAGC 31: Aplicando el valor de *text-indent* a la primera línea

8.2.13. Propiedad text-align

Se tiene soporte para los siguientes valores de la propiedad *text-align*: *left*, *right*, *center*, *inherit*.

Esta propiedad sólo se aplica a elementos *block* que tienen un contexto de formato *inline*. La misma que se implementa al momento de generar las posiciones para cada *box*, es decir en la estructura de formato de fase 2 (Sección 7.4, página 85). Si el valor de la propiedad es *left*, se asigna las posiciones x de manera normal (desde la izquierda).

Si el valor de la propiedad es *right*, se calcula la longitud de espacio sobrante (la resta entre el ancho del contenedor y línea) y se suma a la posición x, como si se tratara de un valor de sangría, y luego se asigna posiciones de manera normal.

Si su valor es *center*, entonces también se calcula la longitud del espacio sobrante, pero en vez de sumarle a x todo el valor, sólo se suma la mitad de la longitud restante, y se continua asignando las posiciones. El Código UUAGC 32 muestra la parte de asignación de posiciones para esta propiedad.

```
SEM Line
  | Line  winds.statePos
    = case @lhs.align of
        "left"   → @lhs.statePos
        "right"  → let (_, y)  = snd@lhs.statePos
                       newX = @lhs.width − @winds.innerLineWidth
                   in (fst @lhs.statePos, (newX, y))
        "center" → let (_, y)  = snd @lhs.statePos
                       newX = (@lhs.width − @winds.innerLineWidth) `div` 2
                   in (fst @lhs.statePos, (newX, y))
```

Código UUAGC 32: Asignación de posiciones para la propiedad *text-align*

8.2.14. Propiedad text-decoration

Se tiene soporte para los siguientes valores de la propiedad *text-decoration*: *underline*, *overline*, *line-through*, *none*, *inherit*.

Esta propiedad se aplica al momento de renderizar el texto de un *box*. Si el valor es *none*, no se realiza nada, pero sí el valor es diferente a *none*, entonces se debe dibujar una línea de acuerdo al tipo de valor. Si el valor es *underline*, se dibuja una línea por debajo del *baseline* del texto, si el valor es *overline*, se dibuja una línea en la parte superior del texto, pero sí el valor es *line-through*, se dibuja una línea por el medio del texto.

El Código UUAGC 73 muestra la implementación del dibujado de líneas.

```
doDecoration dc (Point x y) txtColor (Size width height, baseline, a) value
  = case value of
      KeyValue "underline"
        → do let yb = height − baseline + 2
              line dc (pt 0 yb) (pt width yb) [penColor := txtColor]
      KeyValue "overline"
        → do let yb = y
              line dc (pt 0 yb) (pt width yb) [penColor := txtColor]
      KeyValue "line-through"
        → do let yb = height − baseline − ((height − baseline) `div` 3)
              line dc (pt 0 yb) (pt width yb) [penColor := txtColor]
```

Código Haskell 73: Implementación del comportamiento para la propiedad *text-decoration*

8.2.15. Propiedad text-transform

Se tiene soporte para los siguientes valores de la propiedad *text-transform*: *capitalize*, *uppercase*, *lowercase*, *none*, *inherit*.

Esta propiedad se aplica al momento de renderizar el texto de un *box*. Se ha utilizado funciones *toUpper* y *toLower* del módulo *Char* de Haskell para implementar el comportamiento de los valores de esta propiedad. El Código Haskell 74 muestra la implementación del comportamiento para esta propiedad.

```
applyTextTransform props str
  = case usedValue (props `get` "text-transform") of
      KeyValue "none"
        → str
      KeyValue "capitalize"
        → let newStr       = words str
              fcap (c : cs) = toUpper c : cs
          in unwords $ map fcap newStr
      KeyValue "uppercase"
        → map toUpper str
      KeyValue "lowercase"
        → map toLower str
```

Código Haskell 74: Implementación del comportamiento para la propiedad *text-transform*

8.2.16. Propiedad white-space

Se tiene soporte para los siguientes valores de la propiedad *white-space*: *normal*, *pre*, *nowrap*, *pre-wrap*, *pre-line*, *inherit*.

En (Meyer, 2004, cap. 6) se tiene una tabla que resume el comportamiento de cada valor para esta propiedad, la cual se muestra en la Descripción 30.

La columna de *whitespace* hace referencia a los espacios y tabs, si es *Collapsed* significa que se debe eliminar todos los espacios y tabs, pero sí es *Preserved* se debe conservar todos los espacios y tabs.

La columna de *Linefeeds* hace referencia a los saltos de línea, si su valor es *Ignored*, se debe eliminar todos los saltos de línea, pero sí es *Honored*, se debe conservar todos los saltos de línea.

La columna *Auto line wrapping* hace referencia a la forma automática de acomodar los elementos en líneas, por ejemplo si el valor es *Allowed*, significa que el Navegador Web debe insertar saltos de línea para que los elementos se acomoden de acuerdo al ancho del contenedor, pero sí el valor es *Prevented*, el Navegador Web no insertará saltos de línea a menos que sean explícitamente declarador por el autor de la página web.

Valor	Whitespace	Linefeeds	Auto line wrapping
normal	Collapsed	Ignored	Allowed
nowrap	Collapsed	Ignored	Prevented
pre	Preserved	Honored	Prevented
pre-wrap	Preserved	Honored	Allowed
pre-line	Collapsed	Honored	Allowed

Descripción 30 Comportamiento de la propiedad *white-space*

La propiedad *white-space* se aplica sólo a elementos que contienen texto. Se ha utilizado funciones del módulo *List* de Haskell para implementar el comportamiento de esta propiedad. El Código Haskell 75 muestra la implementación para las columnas *Whitespace* y *Linefeeds* de la Descripción 30.

```
processString isSpaceCollapsed isLineFeedIgnored input
  = case (isSpaceCollapsed, isLineFeedIgnored) of
      (True, True)  → unwords ∘ words $ input   -- case of normal and nowrap
      (True, False) → unlines
                    ∘ map unwords
                    ∘ map words
                    ∘ lines $ input             -- case of pre-line
      otherwise     → input                     -- case of pre and pre-wrap
```

Código Haskell 75: Implementación de *Whitespace* y *Linefeed*

Para la columna *Auto line wrapping* se utiliza la función *lines* del módulo *List* de Haskell.

La 1ra y 2da columna se aplican antes de generar la estructura de formato de fase 1 (Sección 7.2, página 75), pero la 3ra columna se aplica antes de generar los elementos atómicos en la estructura de formato de fase 1(Sección 7.3, página 79).

Capítulo 9

El modelo Box de CSS

El resultado de fase 2 de la estructura de formato del Capítulo 7 era generar los *boxes* de renderización. Estos *boxes*, que son ventanas de *WxHaskell*, son utilizados para renderizar los elementos de una página Web. En otras palabras, cada elemento que se renderiza en la pantalla es un *box*.

Las características de un *box* están definidas por el modelo *Box* de la especificación de CSS (Sección 2.3.1, página 13). Sus dimensiones son calculados utilizando propiedades de CSS.

En éste capítulo se describirá la representación y renderización del modelo *box* de CSS.

9.1. Propiedades del *Box* de CSS

Un *box* de CSS es una caja rectangular con 4 áreas: *content*, *padding*, *border* y *margin*. Cada área tiene sus propiedades de CSS que determinan sus características.

En esta sección se definirá funciones para obtener los valores de las propiedades de cada área del *box*.

9.1.1. Propiedades del `margin-box`

Las propiedades de `margin` son:

- `margin-top`. Determina el ancho que ocupa el margen superior de un *box*.
- `margin-right`. Determina el ancho que ocupa el margen derecho de un *box*.
- `margin-bottom`. Determina el ancho que ocupa el margen inferior de un *box*.
- `margin-left`. Determina el ancho que ocupa el margen izquierdo de un *box*.

Para obtener el ancho que ocupa cada margen, se ha definido una función que obtenga los márgenes y los devuelva en una lista:

```
getMarginProperties props
  = map toInt [maybe 0 unPixelUsedValue (props `getM` "margin-top"    )
              , maybe 0 unPixelUsedValue (props `getM` "margin-right"  )
              , maybe 0 unPixelUsedValue (props `getM` "margin-bottom")
              , maybe 0 unPixelUsedValue (props `getM` "margin-left"   )
              ]
```

Código Haskell 76: Obtener las propiedades del área de *margin*

9.1.2. Propiedades del `padding-box`

Las propiedades del `padding` son:

- `padding-top`. Determina el ancho que ocupa el padding superior de un *box*.
- `padding-right`. Determina el ancho que ocupa el padding derecho de un *box*.
- `padding-bottom`. Determina el ancho que ocupa el padding inferior de un *box*.
- `padding-left`. Determina el ancho que ocupa el padding izquierdo de un *box*.

Al igual que en el Código Haskell 76, se ha definido una función que obtenga las dimensiones del área de *padding*:

```
getPaddingProperties props
  = map toInt [maybe 0 unPixelUsedValue (props `getM` "padding-top"    )
              , maybe 0 unPixelUsedValue (props `getM` "padding-right"  )
              , maybe 0 unPixelUsedValue (props `getM` "padding-bottom")
              , maybe 0 unPixelUsedValue (props `getM` "padding-left"   )
              ]
```

Código Haskell 77: Obtener las propiedades del área de *padding*

9.1.3. Propiedades del `border-box`

Las propiedades del `border-box`, a diferencia de las anteriores que sólo tenían un ancho, están compuestas de: ancho, color y estilo para cada lado del borde.

Para el caso del *color* se ha definido una función que obtiene las propiedades de color para el área del *border*:

```
getBorderColorProperties props
  = [maybe (0,0,0) unKeyComputedColor (props `getM` "border-top-color"    )
    , maybe (0,0,0) unKeyComputedColor (props `getM` "border-right-color"  )
    , maybe (0,0,0) unKeyComputedColor (props `getM` "border-bottom-color")
    , maybe (0,0,0) unKeyComputedColor (props `getM` "border-left-color"   )
    ]
```

Código Haskell 78: Obtener las propiedades de color para el área del *border*

Para el caso del *estilo*, también se ha definido otra función que obtiene las propiedades de estilo para el área del *border*:

```
getBorderStyleProperties props
  = [maybe "none" unKeyComputedValue (props `getM` "border-top-style"    )
    , maybe "none" unKeyComputedValue (props `getM` "border-right-style" )
    , maybe "none" unKeyComputedValue (props `getM` "border-bottom-style")
    , maybe "none" unKeyComputedValue (props `getM` "border-left-style"  )
    ]
```

Código Haskell 79: Obtener las propiedades de estilo para el área del *border*

Finalmente, se ha definido una función para obtener el ancho del área de un *border*:

```
getBorderProperties props
  = let bst  = getBorderStyleProperties props
        bwd = map toInt [maybe 0 unPixelUsedValue (props `getM` "border-top-width"    )
                        , maybe 0 unPixelUsedValue (props `getM` "border-right-width" )
                        , maybe 0 unPixelUsedValue (props `getM` "border-bottom-width")
                        , maybe 0 unPixelUsedValue (props `getM` "border-left-width"  )
                        ]
    in zipWith (λstr wd → if str ≡ "none" then 0 else wd) bst bwd
```

Código Haskell 80: Obtener las propiedades de ancho para el área del *border*

El ancho de un borde depende en cierto modo del estilo, por ejemplo: si el estilo del borde es `"none"`, entonces sin importar el ancho del borde, el ancho siempre sera 0.

9.1.4. Propiedades del `content-box`

Las propiedades del `content-box` son: *width* y *height*.

Estas propiedades no se definen para cada lado. De manera que para obtener su valor, simplemente se busca el nombre en la lista de propiedades de CSS y se obtiene el valor en *pixels*, por ejemplo:

```
valor = unPixelUsedValue (props `get` "width")
```

9.2. Representación del Modelo *Box* de CSS

Para representar el modelo *box* de CSS se utiliza una ventana rectangular de *WxHaskell*. Específicamente se utiliza un *ScrolledWindow*.

Para representar las áreas de un *box* se realiza cálculos para encontrar las dimensiones de cada área.

El *box* que se genera para su renderización debe tener básicamente: un contenido (si es texto), una posición, sus dimensiones, la lista de propiedades y la lista de atributos. También se necesita saber el tipo de *box* (*replaced*) y el *TypeContinuation* del *box*.

Para crear un *box* se ha definido una función que se encargue de crear el *scrolledWindow* y aplicar algunas propiedades. El Código Haskell 81 muestra la función que se encarga de crear el *box*:

```
box cnt wn (x, y) (w, h) continuation props attrs amireplaced
  = do pnl ← scrolledWindow wn [size := sz w h
                               , on paint := onBoxPaint cnt
                                                        continuation
                                                        props
                                                        attrs
                                                        amireplaced
                               ]
       windowMove pnl (pt x y)
       return pnl
```

Código Haskell 81: Función para crear un *box*

Con la definición del Código Haskell 81 se puede representar los *boxes* que generan un *WindowText* y un *WindowContainer*:

```
boxText      = box
boxContainer = box ""
```

La única diferencia entre un *boxText* y *boxContainer* esta en el contenido del *boxContainer* (es un *String* vacío).

9.3. Renderización de un Box

En el Código Haskell 81 se ha mostrado la función que crea el *scrolledWindow*. Al momento de crear el *scrolledWindow* se configura las dimensiones de la ventana con las dimensiones que se recibe como argumento, se utilizó la propiedad *size* de *WxHaskell* para configurar la dimensión de la ventana.

También se configura la posición de la ventana con la función *windowMove* de *WxHaskell*.

Finalmente, se configura la función de renderización del *box* utilizando la función *onBoxPaint* y el evento 'on paint' de *WxHaskell*. En las siguientes sub-secciones, se describirá en más detalle la función *onBoxPaint*.

9.3.1. La función de pintado de un *box*

A continuación se describirá paso a paso la función *onBoxPaint*.

Para empezar, la función *onBoxPaint* recibe: una cadena que es el contenido, el *TypeContinuation* del *box*, la lista de propiedades de CSS, la lista de atributos y el tipo del elemento (*replaced*). También recibe el *device context*, que se utiliza para pintar y dibujar en el *box* y finalmente recibe un rectángulo que representa la dimensión de la ventana:

```
onBoxPaint cnt tp props attrs replaced dc rt@(Rect x y w h)
  = do ...
```

Configurando las fuentes

Primeramente se debe configurar las fuentes de texto para renderizar el contenido:

```
let myFont = buildFont props
dcSetFontStyle dc myFont
```

El Código Haskell 82 muestra la definición de la función *buildFont* que construye el estilo de la fuente utilizando la lista de propiedades de CSS.

```
buildFont props
  = let fsze
          = toInt $ (λvp → vp / 1,6)
                $ unPixelUsedValue (props `get` "font-size")
        fwgt
          = toFontWeight $ computedValue (props `get` "font-weight")
        fstl
          = toFontStyle $ computedValue (props `get` "font-style")
        (family, face)
          = getFont_Family_Face (computedValue (props `get` "font-family"))
    in fontDefault { _fontSize   = fsze
                   , _fontWeight = fwgt
                   , _fontShape  = fstl
                   , _fontFamily = family
                   , _fontFace   = face
                   }
```

Código Haskell 82: Construir la fuente del texto

Las funciones *toFontWeight*, *toFontStyle* y *getFont_Family_Face* (definidas en el Código Haskell 83) convierten valores de CSS a los correspondientes valores utilizables por *WxHaskell*.

```
toFontWeight w = case w of KeyValue "bold" → WeightBold
                           otherwise       → WeightNormal
toFontStyle s = case s of KeyValue "italic"  → ShapeItalic
                          KeyValue "oblique" → ShapeSlant
                          otherwise          → ShapeNormal
getFont_Family_Face fn
  = case fn of
      ListValue list → case head list of
                         StringValue str          → (FontDefault, str)
                         KeyValue "serif"         → (FontRoman, "")
                         KeyValue "sans-serif"    → (FontSwiss, "")
                         KeyValue "cursive"       → (FontScript, "")
                         KeyValue "fantasy"       → (FontDecorative, "")
                         KeyValue "monospace"     → (FontModern, "")
                         otherwise                → (FontDefault, "")
      otherwise      → (FontDefault, "")
```

Código Haskell 83: Funciones de conversión para renderizar las propiedades de la fuente de un texto

Obteniendo las propiedades del *box*

Para obtener las propiedades del *box* se utiliza las funciones definidas en las secciones anteriores:

```
let [mt, mr, mb, ml]   = checkWithTypeElement tp $ getMarginProperties   props
let [bt, br, bb, bl]   = checkWithTypeElement tp $ getBorderProperties   props
let [ppt, ppr, ppb, ppl] = checkWithTypeElement tp $ getPaddingProperties props
let [bct, bcr, bcb, bcl] = map toColor $ getBorderColorProperties props
let [bst, bsr, bsb, bsl] = getBorderStyleProperties props
```

Código Haskell 84: Obtener los valores de las propiedades de un box

La función *checkWithTypeElement* (definida en el Código Haskell 85) modifica cada lado de las dimensiones de acuerdo al *TypeContinuation* del *box*:

```
checkWithTypeElement tp lst@(wt : wr : wb : wl : [])
  = case tp of
      Full   → lst                -- se considera todas la dimensiones
      Init   → [wt, 0, wb, wl]    -- no se considera el lado derecho
      Medium → [wt, 0, wb, 0]     -- no se considera ninguno de los lados
      End    → [wt, wr, wb, 0]    -- no se considera el lado izquierdo
```

Código Haskell 85: Verificar el *TypeContinuation* de un box

También se utiliza funciones para convertir colores y estilos a valores utilizables por *WxHaskell*. El Código Haskell 86 muestra las funciones correspondientes.

```
toColor (r, g, b) = rgb r g b
toPenStyle s
  = case s of
      "hidden" → PenTransparent
      "dotted" → PenDash DashDot
      "dashed" → PenDash DashLong
      otherwise → PenSolid
```

Código Haskell 86: Otras funciones de conversión para la renderización

Calculando las dimensiones de las áreas

Es posible obtener las esquinas de cada área haciendo operaciones de suma y resta sobre los lados de cada área.

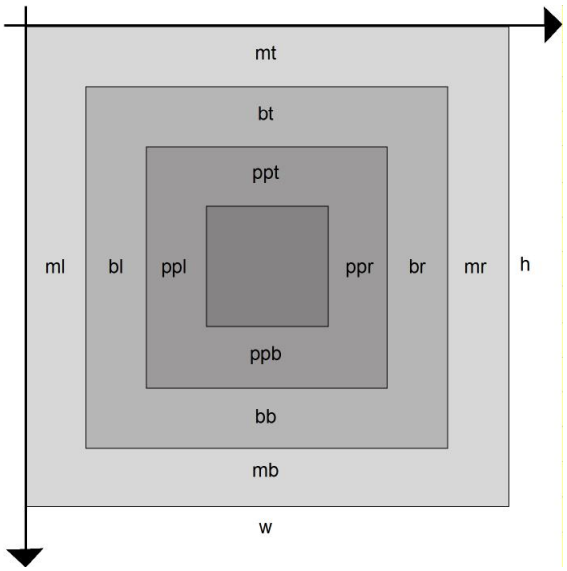


Figura 9.1: Los lados de un box

Con ayuda de la Figura 9.1, se puede calcular los siguientes puntos:

let $(bx1, bx2) = (ml, w - mr - 1)$
let $(by1, by2) = (mt, h - mb - 1)$
let $ptContent = pt\ (ml + bl + ppl)\ (mt + bt + ppt)$

Pintando el background

El *background* (*estilo de fondo del box*) se pinta sobre todo el `content-box`, `padding-box` y `border-box`. El Código Haskell 87 describe la forma de obtener el valor de la propiedad `background-color`.

```
let bkgBrush = case props `getM` "background-color" of
                 Just p   → case computedValue p of
                              KeyValue "transparent" → brushTransparent
                              KeyColor value         → brushSolid (toColor value)
                 Nothing → error "unexpected value at background-color property"
```

Código Haskell 87: Obtener el valor de la propiedad *background-color*

Finalmente, para pintar con la propiedad *background-color*, se debe calcular la dimensión del rectángulo donde se debe pintar:

```
let bkgRect = rect (pt bx1 by1) (sz (bx2 − bx1 + 1) (by2 − by1 + 1))
drawRect dc bkgRect [brush := bkgBrush, pen := penTransparent]
```

Nota.- El WxHaskell para Gnu/Linux no tiene soporte para el estilo transparente de una ventana, esto limita la renderización de ventanas transparentes, especialmente el área del margen de un *box*.

Dibujando los bordes

Para dibujar los bordes se ha definido la función *paintLine* en el Código Haskell 88. Esta función dibuja un borde de un ancho específico y en una dirección dada, recibe el *device context*, el punto inicial y final de línea, el ancho de la línea, el tipo de línea (horizontal o vertical) y la dirección con la que se dibujará la línea (de arriba a abajo o de abajo a arriba).

Si el ancho de la línea es 0, no se dibuja nada:

```
paintLine _ _          _          0      _    _   _
  = return ()
paintLine dc (x1, y1) (x2, y2) width kind dir style
  = do  line dc (pt x1 y1) (pt x2 y2) style
        if kind
          then do let (y3, y4) = if dir
                                 then (y1 + 1, y2 + 1)
                                 else (y1 − 1, y2 − 1)
                  paintLine dc (x1, y3) (x2, y4) (width − 1) kind dir style
          else do let (x3, x4) = if dir
                                 then (x1 + 1, x2 + 1)
                                 else (x1 − 1, x2 − 1)
                  paintLine dc (x3, y1) (x4, y2) (width − 1) kind dir style
```

Código Haskell 88: Función para dibujar el borde un box

Utilizando la función *paintLine*, se dibuja los bordes de cada lado del *box*:

```
paintLine dc (bx1, by1) (bx2, by1) bt True  True  [penWidth := 1
                                                   , penColor := bct
                                                   , penKind  := toPenStyle bst
                                                   ]
paintLine dc (bx2, by1) (bx2, by2) br False False [penWidth := 1
                                                   , penColor := bcr
                                                   , penKind  := toPenStyle bsr
                                                   ]
paintLine dc (bx2, by2) (bx1, by2) bb True  False [penWidth := 1
                                                   , penColor := bcb
                                                   , penKind  := toPenStyle bsb
                                                   ]
paintLine dc (bx1, by2) (bx1, by1) bl False True  [penWidth := 1
                                                   , penColor := bcl
                                                   , penKind  := toPenStyle bsl
                                                   ]
```

Dibujando el contenido

Finalmente, para dibujar el contenido de un *box* se debe considerar el tipo del *box*.

Si el *box* es *replaced*, se procede a dibujar la imagen (el único elemento *replaced* que se reconoce es el nodo etiqueta *img*), caso contrario, se dibuja el contenido del *box*.

Para dibujar una imagen, se obtiene el *path* de la imagen, se calcula sus dimensiones y se escala de acuerdo a las dimensiones.

Cuando la ventana no es *replaced*, se verifica el tipo del elemento de acuerdo a la propiedad *display*. Si el tipo es *block*, no se dibuja nada porque el contenido se encuentra en los hijos, pero si es *inline*, se dibuja el contenido del *box* con el color de texto correspondiente a la propiedad *color* de CSS.

```
if replaced
  then do path ← getImagePath (getAttribute "src" attrs)
           let szimg = sz (w − mr − br − ppr − 1 − ml − bl − ppl)
                          (h − mb − bb − ppb − 1 − mt − bt − ppt)
           img1 ← imageCreateFromFile path
           img2 ← imageScale img1 szimg
           drawImage dc img2 ptContent []
           return ()
  else case usedValue (props `get` "display") of
         KeyValue "block"
           → return ()
         KeyValue "inline"
           → do let txtColor = toColor $ maybe (0, 0, 0)
                                               unKeyComputedColor
                                               (props `getM` "color")
                drawText dc cnt ptContent [color := txtColor]
```

Capítulo 10

Interfaz Gráfica de Usuario (GUI)

En este capítulo se mostrará el desarrollo de la interfaz gráfica de usuario (GUI) para el Navegador Web.

La interfaz gráfica de usuario, en primer lugar, se encarga de integrar todos los módulos que se ha desarrollado en el proyecto.

Además, la interfaz gráfica de usuario provee los mecanismos de interacción entre el usuario y la página Web. Por ejemplo, si el usuario quiere navegar por una página Web, el GUI debe proveer un lugar donde el usuario pueda escribir la dirección URL de la página Web. Y si desde una página Web, el usuario desea seguir un enlace para ir a otra página Web, el GUI también debe proveer la forma de hacer un clic en un enlace y cargar la nueva página Web. Finalmente si el usuario desea volver a una página Web en la que navegó anteriormente, entonces el GUI también debe proveer alguna forma de recargar alguna página ya visitada.

En el Capítulo 6 (Sección 6.1.2, página 48), cuando se obtenía todas las hojas se estilo, se mencionó la existencia de 3 autores: *UserAgent*, *User y Author*. El autor *User*, que corresponde al que maneja el Navegador Web, debe tener la capacidad de añadir reglas de estilo. Entonces, además de la interacción usuario-página Web, el GUI también debe proveer una forma de añadir reglas de estilo para el usuario *User*.

De esa manera, en el presente capítulo se presenta del desarrollo de la Interfaz Gráfica de Usuario para el proyecto.

Así mismo, se utilizará la librería *WxHaskell* (Leijen, 2004) para el desarrollo del GUI.

Se inicia el capítulo con el desarrollo una interfaz gráfica de usuario sin acciones y a medida que se vaya avanzando, se añadirá acciones para los botones y menús.

10.1. Interfaz Gráfica de Usuario Básica

En esta sección se definirá una interfaz gráfica de usuario sin acciones.

Para ello se define las funciones principales para *WxHaskell* e interfaz gráfica de usuario. El Código Haskell 89 muestra las funciones principales.

```
main :: IO ()
main = start browser
browser :: IO ()
browser
  = do ...
```

Código Haskell 89: Funciones principales del GUI

En la función *browser* del Código Haskell 89 se definirá todos los componentes del GUI.

10.1.1. Variables de WxHaskell

Se utiliza las variables de *WxHaskell* para almacenar información relevante para el Navegador Web.

Por ejemplo ha definido las siguientes variables:

- Variable *sfiles*: guarda la lista de nombres de archivos de hojas de estilo.

```
lfiles ← readConfigFile
sfiles ← variable [value := lfiles]
```

- Variable *varfstree*: guarda la estructura *FSTressFase1* en un tipo *Maybe*. Inicialmente es *Nothing*.

```
varfstree ← variable [value := Nothing]
```

- Variable *varzipper*: guarda a lista de páginas de navegación (necesario para ir adelante y atrás).

```
varzipper ← variable [value := initZipperList]
```

- Variable *varbaseurl*: guarda el *url* base de una página Web. Inicialmente es `""`.

```
varbaseurl ← variable [value := ""]
```

- Variable *varDefaultCSS4html*: guarda la hoja de estilos del *UserAgent*.

```
defaultcss4html      ← parseFileUserAgent $ maybe "" id $ Map.lookup
                                                    "User_Agent_Stylesheet"
                                                    lfiles
varDefaultCSS4Html ← variable [value := defaultcss4html]
```

10.1.2. Ventanas y Botones

La ventana principal del Navegador Web es una ventana de tipo *Frame*, que es definida con la función *frame* de WxHaskell:

```
f ← frame [text := "Simple San Simon Functional Web Browser"]
```

También se define la ventana donde se renderizará las páginas Web, este tiene un tamaño virtual de $800 * 600$:

```
pnl ← scrolledWindow f [virtualSize := sz 800 600]
```

Luego se define los botones principales con los que interactuará el usuario: un widget *inp* donde se escribirá el *url*, un botón *get* para obtener la página Web, un botón *upd* para actualizar la página Web, y los botones *goForward* y *goBackward* para la navegación entre páginas.

```
inp         ← entry f   [text := "file:///home/carlos/fwb/test1.html"]
get         ← button f [text := "Get"]
upd         ← button f [text := "Update"]
goForward   ← button f [text := " >> ", size := sz 50 (-1)]
goBackward ← button f [text := " << ", size := sz 50 (-1)]
```

10.1.3. El menú principal

Para crear el menú principal, se ha definido la función *createMenus*, que se encarga de crear todos los menús del Navegador Web:

```
createMenus f sfiles varzipper inp pnl varfstree varDefaultCSS4Html varbaseurl
```

La función *createMenus* está definido de la siguiente manera:

```
createMenus f sfiles varzipper inp pnl varfstree varDefaultCSS4Html varbaseurl
  = do    -- panel browser
       pbrowser  ← menuPane [text := "Browser"]
       mgetPage ← menuItem pbrowser [text := "Get a Web Page\tCtrl+g"
                                    , on command := windowSetFocus inp
                                    ]
       mgoFord  ← menuItem pbrowser [text := "Go Forward\tCtrl+f"]
       mgoBack  ← menuItem pbrowser [text := "Go Backward\tCtrl+b"]
       menuLine pbrowser
       mclose     ← menuQuit pbrowser [text := "Close", on command := close f]
         -- panel settings
       psettings  ← menuPane [text := "Settings"]
       muserS    ← menuItem psettings [text := "User Stylesheet"]
       magentS   ← menuItem psettings [text := "User Agent Stylesheet"]
```

```
  -- panel help
phelp ← menuPane [text := "Help"]
menuAbout phelp [text := "About", on command := infoDialog f "About" about]
  -- set the menus on the frame
set f [menuBar := [pbrowser, psettings, phelp]]
```

10.1.4. El *layout* del Navegador Web

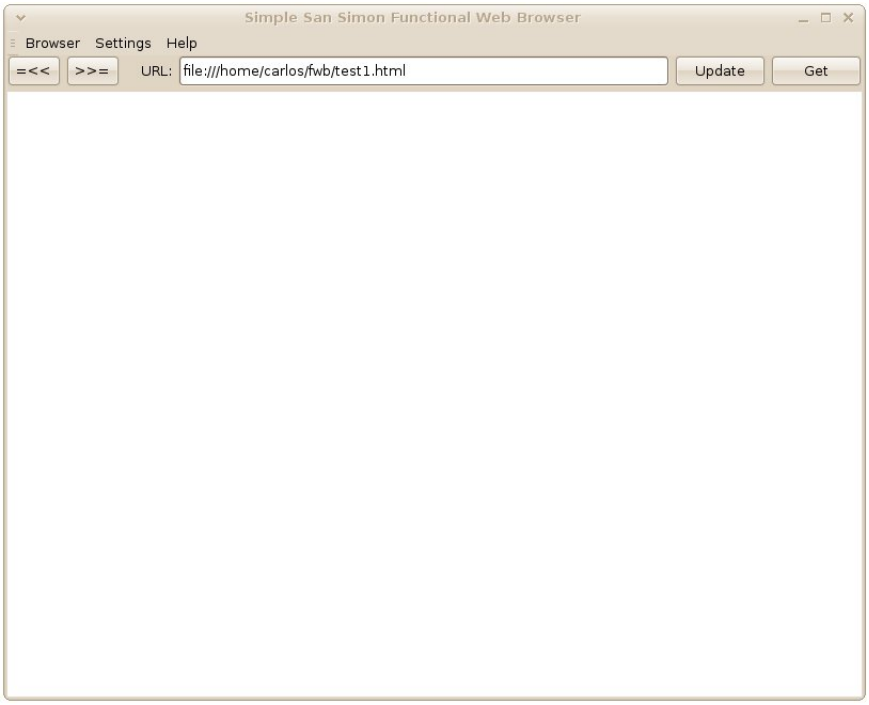
El esquema para la ventana principal del Navegador Web tiene dos partes:

- Panel de interacción: es el lugar donde están los botones principales para interactuar con el Navegador Web. Por ejemplo se tiene al widget donde se coloca el *url*, los botones para obtener y actualizar la página Web.
- Panel del renderización: es el lugar donde se renderizan las páginas Web. Ocupa la mayor parte de la ventana principal.

La implementación del esquema (*layout*) es de la siguiente manera:

```
set f [layout := column 5 [row 5 [widget goBackward, widget goForward, hspace 10
                                 , centre $ label "URL:", hfill $ widget inp
                                 , widget up, widget go
                                 ]
                           , fill $ widget pnl
                           ]
      ]
```

A continuación se muestra la imagen de la GUI desarrollada hasta esta parte:



10.2. Descargar Recursos de la Web

En esta sección se desarrollará un módulo que interactué con los protocolos HTTP y File para descargar recursos de la Web.

Los recursos que se necesita descargar son: archivos HTML, hojas de estilo e imágenes.

Para descargar los recursos de la Web se está utilizando la librería *libcurl*. LibCurl (Finne, s.f.) es una librería que permite interactuar con varios protocolos a través de una dirección *URL*. Entre los protocolos con los que interactúa están los protocolos HTTP y File, los cuales son necesarios para el proyecto.

10.2.1. Descargar un documento HTML

Para descargar un archivo HTML se ha definido la función *getContenidoURL* del Código Haskell 90 que recibe un *URL* en un *String* y devuelve una tupla, donde el primer elemento es el *URL* y el segundo elemento es un *String* que representa el contenido de la página Web.

Para descargar el archivo se está utilizando la función *curlGetResponse_* de la librería LibCurl.

Si ocurre algún error en la descarga, se devuelve la dirección *URL* y una página HTML que muestre el error. Pero si la descarga tiene éxito, entonces se crea un nuevo proceso para descargar tanto las imágenes y hojas de estilo. Luego se devuelve la dirección *url* y el contenido HTML que se ha descargado.

```
getContenidoURL :: String → IO (String, String)
getContenidoURL url
   = do (CurlResponse cCode _ _ _ content fvalue) ← getResponse url
        putStrLn $ show cCode ++ " at " ++ url
        (IString eurl) ← fvalue EffectiveUrl
        if cCode ≡ CurlOK
          then do let base = URL.getBaseUrl eurl
                  forkIO (downloadImages base content)
                  forkIO (downloadHTMLStyleSheet base content)
                  return (eurl, content)
          else return $ (url, pageNoDisponible (show cCode) url)
getResponse :: String → IO (CurlResponse_ [(String, String)] String)
getResponse url = curlGetResponse_ url []
pageNoDisponible :: String → String → String
pageNoDisponible error link
   = "<html> error page ... "
```

Código Haskell 90: Función para descargar el contenido de una dirección URL

10.2.2. Descargar imágenes

En la anterior sección, después que la descarga tenía éxito, se procedía a crear un nuevo proceso para descargar las imágenes, en ese nuevo proceso se llamaba a la función *downloadImages*, la cual recibía el *URL* base de la página Web y el contenido del HTML.

En esta sección se describirá la función *downloadImages*.

Para descargar las imágenes, además de la librería LibCurl, también se está utilizando otras librerías de Haskell:

- *GD*(Bringert, s.f.): Se utiliza para convertir el contenido de una imagen (*String*) descargada de la Web en un archivo de imagen con formato específico.
- *TagSoup*(Mitchell, s.f.): Se utiliza para buscar el *URL* de las imágenes a descargar en el contenido de una página Web.
- *URL*(Diatchki, s.f.): Se utiliza para hacer operaciones sobre una dirección *URL* (por ejemplo, encontrar el *URL* base).

Lo primero que se realiza para descargar las imágenes es obtener las *URLs* de las imágenes a descargar. Luego se elimina todas las *URL* que son repetidas (para no tener que descargar más de dos veces una imagen). Finalmente, se asigna las funciones correspondientes para convertir las imágenes descargadas a un determinado tipo de imagen.

El Código Haskell 91 y 91 muestran la implementación de la función *downloadImages*, la cual se encarga de descargar todas las imágenes de una página Web.

```
downloadImages base stringHTML
  = do let imgTags  = [ fromAttrib "src" tag
                      | tag ← parseTags stringHTML
                      , tagOpen (≡ "img") (anyAttrName (≡ "src")) tag
                      ]
           imgSRCs = nub imgTags   -- elimina los repetidos
           imgfuns  = map getImageFunctionNameType imgSRCs
       mapM_ downloadprocess imgfuns
  where downloadprocess (url, name, fload, fsave)
          = do img    ← download base url
               gdimg ← fload img
               let path = tmpPath ++ name
               fsave path gdimg
               putStrLn $ "image saved at " ++ path
download base url
  = do let url' =   if URL.isAbsolute url
                    then url
                    else if URL.isHostRelative url
                         then base ++ url
                         else base ++ "/" ++ url
       (cod, obj) ← curlGetString_ url' []
       putStrLn $ show cod ++ " at " ++ url'
       return obj
```

Código Haskell 91: Funciones para descargar imágenes, parte 1

```
getImageFunctionNameType url
  = let name = takeFileName url
    in case takeExtension url of
      ".jpg"   → (url, name, loadJpegByteString, saveJpegFile (−1))
      ".png"   → (url, name, loadPngByteString, savePngFile)
      ".gif"   → (url, name, loadGifByteString, saveGifFile)
      otherwise → error $ "[DownloadProcess] error with ..."
```

Código Haskell 92: Funciones para descargar imágenes, parte 2

Se ha definido un directorio común (./*tmp*/ del ejecutable) para guardar todas las imágenes descargadas.

La función *download* del Código Haskell 91 se encarga de descargar un recurso desde la Web. Pero antes de proceder a descargar hace ciertas operaciones para tener una dirección *URL* absoluta (dirección completa o entera del recurso a descargar).

La función *getImageFunctionNameType* del Código Haskell 92 asigna las funciones correspondientes para convertir la imagen descargada (*String*) en un archivo de imagen. Note que sólo se reconocen 3 tipos de formato de imagen: JPG, PNG y GIF.

10.2.3. Descargar Hojas de Estilo

El proceso de descargar hojas de estilo es básicamente el mismo que para descargar imágenes. La única diferencia es que ya no se necesita funciones de conversión entre un *String* y el archivo de imagen, porque se descarga un archivo de texto plano. Otra de las diferencias es que la dirección *URL* para descargar se encuentra en los elementos *link* de la página Web.

10.3. El proceso de Renderización

En esta sección se describirá el proceso de renderización de una página Web. También se define las acciones para los botones *get* y *update* de la interfaz gráfica de usuario.

El proceso de renderización comprende los siguiente pasos:

1. Obtener la dirección *URL* de la página Web que se quiere renderizar.
2. Descargar el contendido del *URL*.
3. Parsear el contenido del *URL*.
4. Procesar el *NTree* y generar el *FSTree*.
5. Procesar la estructura de formato fase 1.
6. Procesar la estructura de formato fase 2.
7. Limpiar la ventana por si existe alguna otra página Web renderizada.
8. Renderizar la página Web en la ventana.

Otra de las acciones importantes de un Navegador Web, aparte de renderizar una página Web, es el de actualizar una página Web ya existente. La actualización no sólo puede ser generada por el botón *update*, sino también por el redimensionamiento de la ventana principal o cuando el Navegador Web lo requiera.

La actualización de una página Web no necesita realizar todo el proceso de renderización descrita en la anterior lista, sino que solamente lo correspondiente a redimensionar, es decir desde el paso 5.

Entonces, para hacer posible la acción de actualizar se ha definido una variable de *WxHaskell* denominada: *varfstree*. La variable *varfstree* guarda el resultado de procesar el *NTree*, de manera que, no se tenga que volver a procesarlo cuando se tenga que actualizar la página Web.

A continuación se describen las funciones de renderización y actualización de páginas Web.

10.3.1. Renderizar una página Web

Así como se describió en la sección anterior, el primer paso para renderizar, es obtener la dirección *URL* de la página Web. Para lo cual, se ha definido una función que obtenga el texto del *widget entry* y llame a otra función para renderizar la dirección *URL*:

```
renderPage pnl inp varfstree varzipper varDefaultCSS4Html varbaseurl
  = do url ← get inp text
       goToURL pnl inp varfstree varzipper varDefaultCSS4Html varbaseurl url
```

La función *goToURL* se encarga de renderizar el *URL* que recibe como parámetro:

```
goToURL pnl inp varfstree varzipper varDefaultCSS4Html varbaseurl url
  = do ...
```

Lo primero que se realiza, es obtener el contenido del URL:

```
(eurl, content) ← getContenidoURL url
```

Luego, se obtiene el *URL* base y se guarda en la variable *varbaseurl*:

```
let baseurl = getBaseUrl eurl
set varbaseurl [value := baseurl]
```

Seguidamente, se actualiza el *widget entry* con el *URL* devuelto por la función *getContenidoURL*. También se actualiza la lista de navegación de páginas con el nuevo *URL* que se está renderizando:

```
set inp [text := eurl]
  -- inserting into the historial
zipper ← get varzipper value
let newzipper = insert url zipper
set varzipper [value := newzipper]
```

Lo siguiente es "parsear" el contenido del *URL*:

```
ast ← parseHTML content
```

Luego se debe procesar el *NTree* y generar la estructura de formato. El resultado de esta parte, es guardado en la variable *varfstree*:

```
defaultcss4html ← get varDefaultCSS4Html value
let fstree = genFormattingEstructure ast defaultcss4html
set varfstree [value := fstree]
```

Finalmente, se llama a la función *updateInitialContainer* y se repinta el panel de renderización:

```
updateInitialContainer pnl inp varfstree varzipper varDefaultCSS4Html varbaseurl
repaint pnl
```

La función *updateInitialContainer* se encarga de procesar la estructura de formato que se encuentra en la variable *varfstree*:

```
updateInitialContainer icb inp varfstree varzipper varDefaultCSS4Html varbaseurl
  = do ...
```

Lo primero es limpiar la ventana de renderización (por si existe una página ya renderizada) y se posiciona la barra de desplazamiento en la parte inicial:

```
windowDestroyChildren icb
scrolledWindowScroll icb (pt 0 0)
```

Luego se recolecta la información necesaria para procesar la estructura de formato:

```
(Size w h) ← windowGetClientSize icb
baseurl ← get varbaseurl value
```

Y se obtiene la estructura de formato de la variable *varfstree*:

```
result ← get varfstree value
```

Si el contenido de la variable es *Nothing* no se realiza nada, caso contrario se procesa el *fstree*:

```
case result of
  (Just fstree) → do let boxtree
                           = processFSTree1 fstree icb (w, h)
                         (_, fresult, (wc, hc))
                           = processFSTree2 boxtree
                                            baseurl
                                            icb
                                            (goToURL icb
                                                     inp
                                                     varfstree
                                                     varzipper
                                                     varDefaultCSS4Html
                                                     varbaseurl)
                     fresult icb
                     sw ← get icb size
                     let ns@(Size nw nh) = sizeMax sw (sz wc hc)
                     set icb [virtualSize := ns, scrollRate := sz (nw `div` 100) (nh `div` 100)]
  Nothing       → return ()
```

Después de procesar la estructura de formato se hace ciertas operaciones para configurar la barra de desplazamiento del panel de renderización.

10.3.2. Acciones para los botones de la interfaz gráfica

A continuación se asignará las acciones para los botones *Get* y *Update* de la interfaz gráfica de usuario.

Cuando se hace clic en el botón *Get*, simplemente se llama a la función para renderizar la página Web:

```
set get [on command := renderPage pnl
                                  inp
                                  varfstree
                                  varzipper
                                  varDefaultCSS4Html
                                  varbaseurl]
```

Si se hace clic en el botón *Update* se llama a la función *updateInitialContainer*:

```
set upd [on command := updateInitialContainer pnl
                                              inp
                                              varfstree
                                              varzipper
                                              varDefaultCSS4Html
                                              varbaseurl]
```

Si la ventana principal cambia sus dimensiones, también se llamar a la función *updateInitialContainer* para redimensionar la página Web:

```
set pnl [on resize := updateInitialContainer pnl
                                             inp
                                             varfstree
                                             varzipper
                                             varDefaultCSS4Html
                                             varbaseurl]
```

Finalmente, si se presiona la tecla *enter* en el *widget entry*, se debe renderizar la página sin necesidad de hacer clic en el botón *Get*:

```
set inp [on enterKey := renderPage pnl
                                   inp
                                   varfstree
                                   varzipper
                                   varDefaultCSS4Html
                                   varbaseurl]
```

10.4. Acciones para los botones *goForward* y *goBackward*

En esta sección se definirá la funcionalidad para la navegación de páginas visitadas.

10.4.1. El módulo *ZipperList*

Para navegar entre las páginas Web visitadas se ha desarrollado el módulo *ZipperList* el cual define funciones para navegar sobre una lista de páginas Web.

Las funciones que tiene el módulo *ZipperList* son:

- *forward*: Obtiene un elemento que se encuentra a la derecha del elemento actual de la lista.
- *backward*: Obtiene un elemento que se encuentra a la izquierda del elemento actual de la lista.
- *insert*: Inserta un *url* a la izquierda del elemento actual de la lista.
- *getElement*: Obtiene el *url* donde apunta el elemento actual de la lista.
- *initZipperList*: Inicializa el *ZipperList*.

10.4.2. Configurar las acciones

Para configurar las acciones *goForward* y *goBackward* se ha definido la función *onButtonHistorial*, que recibe una acción (*fmove*) de *ZipperList*, la variable *varzipper* (que guarda el *ZipperList*) y otros argumentos necesarios para la renderización:

```
onButtonHistorial fmove varzipper inp pnl varfstree varDefaultCSS4Html varbaseurl
  = do zipper ← get varzipper value
       let newzipper = fmove zipper
       set varzipper [value := newzipper]
       set inp [text := getElement newzipper]
       renderPage pnl
                  inp
                  varfstree
                  varzipper
                  varDefaultCSS4Html
                  varbaseurl
```

La función *onButtonHistorial* primeramente obtiene el valor de la variable *varzipper*, aplica la función de movimiento sobre el *ZipperList*, actualiza la variable *varzipper* y el *widget entry* con el nuevo valor y finalmente llama a la función *renderPage* para renderizar la página Web.

Con la función *onButtonHistorial* se puede configurar las acciones *backward* y *forward* tanto del menú y botones de la interfaz:

En los menus:

```
set mgoFord [on command := onButtonHistorial forward
                                             varzipper
                                             inp
                                             pnl
                                             varfstree
                                             varDefaultCSS4Html
                                             varbaseurl]
set mgoBack [on command := onButtonHistorial backward
                                             varzipper
                                             inp
                                             pnl
                                             varfstree
                                             varDefaultCSS4Html
                                             varbaseurl]
```

En los botones:

```
set goForward [on command := onButtonHistorial  forward
                                                varzipper
                                                inp
                                                pnl
                                                varfstree
                                                varDefaultCSS4Html
                                                varbaseurl]
set goBackward [on command := onButtonHistorial backward
                                                varzipper
                                                inp
                                                pnl
                                                varfstree
                                                varDefaultCSS4Html
                                                varbaseurl]
```

10.5. Archivos de hojas de estilo

En esta sección se describe la funcionalidad para manipular las hojas de estilo para el usuario *User* y *UserAgent*.

10.5.1. Archivos de Configuración

Se está utilizando un archivo de configuración que contiene las direcciones de los archivos que almacenan las hojas de estilo. Esto es realizado, con el objetivo de permitir que el Navegador Web pueda recordar el archivo de hoja de estilo que estuvo utilizando antes que se cierre el programa.

El archivo de configuración es bastante simple, su formato es:

```
nombre_tipo_hoja_estilo = "/path/hoja/estilo.css"
```

En la interfaz gráfica se ha definido la variable *sfiles* para almacenar la lista de archivos de hojas de estilo. El valor de *sfiles* es de tipo *Map String FilePath*, donde la clave es el nombre que representa el archivo y su valor es el *path* de la hoja de estilo a la que hace referencia el nombre.

Entonces, cuando se inicia el programa, se lee el archivo de configuración para recordar la hoja de estilo que se estaba utilizando:

$$lfiles \leftarrow readConfigFile$$
$$sfiles \leftarrow variable\ [value := lfiles]$$

La función *readConfigFile* se encarga de leer el archivo de configuración y de convertirlo al tipo deseado para la variable.

10.5.2. Variable para las Hojas de Estilo

También se está utilizando variables para guardar las hojas de estilo tanto de *UserAgent* y *User*. Por ejemplo, la variable *varDefaultCSS4Html* guarda la estructura *HojaEstilo* de *UserAgent*.

Entonces, cuando se hace clic en el ítem *User Agent Stylesheet* del menú *Settings* se lanza una ventana para seleccionar un archivo de hoja de estilo. Una vez que se selecciona el archivo, se modifica las variables y archivos correspondientes.

Por ejemplo, para el menú de *UserAgent*, se tiene:

$$set\ magentS\ [on\ command := getUserAgentStylesheet]$$

La función *getUserAgentStylesheet* modifica la variable *varDefaultCSS4Html*:

```
getUserAgentStylesheet
  = do mf ← selectFile "User Agent Stylesheet"
       newStylesheet ← case mf of
                         Just path → parseFileUserAgent path
                         Nothing   → return $ Map.empty
       set varDefaultCSS4Html [value := newStylesheet]
```

La función *selectFile* lanza la ventana para seleccionar el archivo de hoja de estilo, si se selecciona un archivo, se debe modificar el archivo de configuración de hojas de estilo.

```
selectFile nm = do
  mf ← fileOpenDialog f True True ("Select " ++ nm) [("Stylesheet",["*.css"])] "" ""
  case mf of
    Just fn  → do lf1 ← get sfiles value
                  let nmc = concat $ List.intersperse "_" $ words nm
                      lf2 = Map.insert nmc fn lf1
                  writeConfigFile lf2
    Nothing → return ()
  return mf
```

Capítulo 11

Conclusiones y Recomendaciones

El objetivo general que se ha propuesto para este proyecto fue: *Desarrollar un Navegador Web con el lenguaje de programación funcional Haskell.*

Los objetivos específicos que se persiguieron durante el desarrollo de este proyecto fueron:

- *Desarrollar el módulo de comunicación entre el Navegador Web y los Protocolos HTTP y modelo TCP/IP.*
- *Desarrollar un intérprete de la información HTML.*
- *Desarrollar los algoritmos que nos permitirán mostrar la información en la pantalla del Navegador Web.*
- *Desarrollar la Interfaz Gráfica de Usuario (GUI).*
- *Desarrollar un módulo que de soporte a CSS (Style Sheet Cascade).*

Indirectamente, también se pretendió experimentar las capacidades del lenguaje funcional Haskell y sus herramientas en el desarrollo de un Navegador Web.

Al final de todo este trabajo y como una conclusión general, se encontró que el lenguaje de programación funcional Haskell fue apropiado y maduro para el desarrollo de un Navegador Web. Fue apropiado porque varias partes de la implementación fueron expresadas de mejor manera utilizando mecanismos de la programación funcional, y fue maduro porque en muchos casos no se ha implementado nuevas librerías, sino simplemente se ha utilizado las ya existentes en el lenguaje.

Además, las herramientas y librerías utilizadas han jugado un rol importante en la simplificación de la complejidad en el desarrollo del proyecto.

A pesar de que normalmente se utilizan, para el desarrollo de este tipo de programas, lenguajes convencionales e imperativos, Haskell ha sido de bastante utilidad, beneficiando al proyecto con varias de sus características, entre las más importantes: código modular, funciones de alto-orden, evaluación no estricta y emparejamiento de patrones.

Sin embargo, también se ha tenido varias dificultades en la implementación de algunos algoritmos, y algunas limitaciones de algunas librerías.

En las siguientes secciones se describirá todos estos puntos en más detalle.

11.1. Presentación del proyecto

Se ha desarrollado un Navegador Web con Haskell, el cual lleva por nombre: **Simple San Simon Functional Web Browser (3S-WebBrowser)** en honor al nombre de la Universidad donde se inició el desarrollo y al paradigma utilizado en el proyecto.

Figura 11.1: Logotipo de 3S-WebBrowser

En la Figura 11.1 se muestra el logotipo del proyecto. También se tiene una página Web que contiene información actualizada del proyecto (`http://hsbrowser.wordpress.com`).

11.1.1. Soporte de HTML/XHTML/XML

El proyecto desarrollado tiene soporte para un subconjunto de la gramática del lenguaje HTML, XHTML y XML. El parser genérico del proyecto le permite reconocer cualquier etiqueta, con la única restricción de que el nombre de la etiqueta de inicio sea el mismo que la etiqueta final.

Gracias a CSS, el proyecto tiene soporte para:

- Modificar las características de un elemento (box)
  - Dimensiones
  - Colores
  - Estilos
  - Tipo de *box* (*block*, *inline*)
  - Posiciones
- Modificar el estilo del texto
  - Fuente
  - Color
  - Tamaño
  - Estilo
  - Transformaciones
- Listas
- Generación de contenidos

11.1.2. Soporte de estilos de CSS

El proyecto desarrollado también tiene soporte para un subconjunto de la gramática de CSS.

Se tiene soporte para 48 propiedades de CSS (sin incluir las propiedades *shorthand*): *display, margin-top, margin-bottom, margin-right, margin-left, padding-top, padding-right, padding-bottom, padding-left, border-top-width, border-right-width, border-bottom-width, border-left-width, border-top-color, border-right-color, border-bottom-color, border-left-color, border-top-style, border-right-style, border-bottom-style, border-left-style, font-size, font-weight, font-style, font-family, position, top, right, bottom, left, float, color, width, height, line-height, vertical-align, content, counter-increment, counter-reset, quotes, list-style-position, list-style-type, background-color, text-indent, text-align, text-decoration, text-transform, white-space.*

El GUI del proyecto permite modificar las hojas de estilo para los usuarios *User* y *UserAgent*.

11.1.3. Otras características

- El proyecto desarrollado tiene soporte para trabajar con los protocolos HTTP y File.
- El GUI permite la navegación entre páginas Web visitadas.

11.2. El lenguaje de programación utilizado

En el desarrollo de los actuales Navegadores Web (Firefox, Chrome, Internet Explorer) se utilizó lenguajes imperativos (C, C++, Java), sin embargo, en este proyecto se utilizó el lenguaje de programación funcional Haskell.

Haskell es un lenguaje de programación poderoso que puede beneficiar al programador y proyecto con sus características. A continuación se mostrará las características más importantes de Haskell que beneficiaron al proyecto.

11.2.1. Datatypes de Haskell

La forma rica de definir los *datatypes* de Haskell ha permitido que algunos de los tipos de datos del proyecto sean expresados de mejor manera. Por ejemplo, el tipo de dato *Property* (Sección 6.3.1, página 59), que se describe a continuación:

```
data Property
  = Property { nombre           :: String
             , inherited        :: Bool
             , initial          :: Valor
             , valor            :: Parser Valor
             , propertyValue    :: PropertyValue
             , fnComputedValue  :: FunctionComputed
             , fnUsedValue      :: FunctionUsed
             }
```

El tipo de dato *Property* refleja lo que la especificación de CSS define. Por ejemplo, la especificación indica que una propiedad debe tener un nombre, valor inicial, parser para sus valores, etc. Los mecanismos de Haskell permitieron expresar la especificación de la propiedad casi de forma plana y directa.

En la definición de *Property* se utiliza funciones (*fsComputedValue* y *fnUsedValue*), las cuales actúan como cualquier otro tipo de dato. Esto permite que la definición de una instancia para *Property* sea bastante expresivo, de manera que el programador debe especificar una función que no es evaluada al momento de la definición, sino cuando el proyecto lo requiera.

11.2.2. Biblioteca de funciones de Haskell

Haskell dispone de una amplia biblioteca de funciones que pueden ser usadas para implementar distintos tipos de comportamientos.

Por ejemplo, se ha utilizado varias de las funciones de listas de Haskell para implementar la propiedad *white-space* (Sección 8.2.16, página 109).

También se ha utilizado las funciones de la estructura *Map* para representar la lista de propiedades de CSS de un elemento.

11.2.3. Definición de funciones de Haskell

Haskell provee varias formas de definir una misma función. Muchas veces, la definición en cierta forma, es más simple y expresiva que las otras.

Este es el caso de la definición de la función *doComputedValue* (Sección 6.3.4, página 67), donde se utilizó la definición por emparejamiento de patrones (*pattern matching*), que resulto ser más expresiva y simple.

11.2.4. Aplicación parcial de funciones

En Haskell, cualquier función que tiene 2 o más argumentos, puede ser parcialmente aplicado a uno o más argumentos, lo cual es una forma poderosa de construir funciones como resultados (Thompson, 1999).

Esta característica es utilizada en dos partes del proyecto: Generación de ventanas renderizables desde la estructura de formato de fase 2 (Sección 7.4.4, página 93) y en la función *goToURL* (Sección 10.3.1, página 127) que es utilizada para implementar la función *onClick* (Código Haskell 72).

11.2.5. Modularidad

Haskell permite definir módulos en los que se agrupa tipos de datos y funciones. Cada módulo permite definir funciones y tipos que estén disponibles para su uso en otros módulos.

En el proyecto, se ha definido varios módulos con su respectiva funcionalidad, lo cual ha permitido que el código del proyecto sea más organizado.

11.3. Las herramientas y librerías utilizadas

Las herramientas y librerías utilizadas han jugado un rol importante en la simplificación de la complejidad en el desarrollo del proyecto.

A continuación de describe las principales herramientas y librerías utilizadas.

11.3.1. Librería *uu-parsinglib*

La librería *uu-parsinglib* fue utilizada para implementar los analizadores sintácticos del lenguaje de marcado genérico y lenguaje de estilos de CSS.

Uno de los principales beneficios que la librería ha provisto al proyecto fue el hacer que la gramática implementada sea robusta, es decir, que se puedan hacer correcciones en la entrada si esta es incorrecta.

Otro de los beneficios de la librería, es que no se dependió de otra herramienta en la que se tenga que utilizar otra sintaxis, sino más bien, se utilizó el mismo lenguaje Haskell para implementar el analizador sintáctico.

11.3.2. Herramienta UUAGC

También se ha utilizado la herramienta UUAGC para la mayor parte de la descripción de la semántica del proyecto.

Esta herramienta ha sido beneficiosa para el proyecto, porqué en primer lugar, permite expresar las computaciones haciendo un movimiento de información a través del árbol de la gramática. Esto permite enfocarse en la resolución del problema, es decir en la computación, la cual es expresada con código Haskell.

Entre los otros beneficios están: generar sólo las partes que se necesita para el proyecto. Por ejemplo si sólo se necesita las funciones semánticas, es posible generar sólo las funciones semánticas y no los tipos de datos (En la Sección B.5, página 168 se muestra la forma de generar código Haskell desde UUAGC).

La herramienta también brinda la posibilidad de evitar código repetitivo a través de las reglas de copiado de la herramienta. Esto ha permitido que el código del proyecto sea compacto.

Finalmente, la herramienta también permite dividir el código en varios archivos, de manera que cada archivo represente una parte específica.

11.3.3. Librería WxHaskell

Otra de las librerías principales que se utilizó en el proyecto fue *WxHaskell*, que permitió implementar toda la parte gráfica del proyecto y modelar el proceso de renderización (Sección 10.3, página 126) utilizando variables de *WxHaskell* para almacenar el resultado de partes que no requieren reprocesamiento.

Esta librería presentó algunas limitaciones en la implementación, las cuales se muestran a continuación.

- *WxHaskell* no tiene soporte para el color transparente en la plataforma Gnu/Linux. Esto afectó en la implementación de la propiedad *background-color*.

- *WxHaskell* es una librería para Haskell que utiliza la librería *WxWidgets*, una librería gráfica para C++. Sin embargo, *WxHaskell* no provee toda la funcionalidad de *WxWidgets* y en algunos casos, el mapeo de funciones se hizo de manera incorrecta. Por ejemplo, en *WxWidgets* la función *wxFont*::*SetFaceName* retorna un valor *Bool* como resultado de ejecutar la función, pero en *WxHaskell*, la misma función (*fontSetFaceName*) no retorna nada. Esto ha puesto limites en la implementación, porque no se pudo saber la existencia de una fuente de texto.

11.4. Limitaciones del proyecto

El proyecto presentado en este documento corresponde simplemente a una pequeña parte de la amplia y compleja área de los Navegadores Web.
Los actuales Navegadores Web, son eficientes en el trabajo que realizan, dando soporte a un amplio conjunto de documentos e implementando una gran cantidad de funcionalidad.

En este proyecto sólo se ha considerado un subconjunto de las versiones estándar de HTML/XHTML y CSS. Por ejemplo, no se dio soporte a formularios, tablas, ni *frames* de HTML. Tampoco se implementó todo el comportamiento definido por la especificación de CSS, sólo se implementó el posicionamiento estático (*normal flow*) y relativo (No se implementó el posicionamiento flotante, ni absoluto). Finalmente, sólo se dio soporte a 48 propiedades de CSS.

Los algoritmos para acomodar los elementos en la pantalla, fueron desarrollados manualmente, es decir sin utilizar mecanismos de la librería gráfica, lo cual afecta el tiempo de renderización y redimensionamiento de páginas Web.

A pesar de las limitaciones del proyecto, es apreciable y valorable como las características, librerías y herramientas de Haskell colaboraron a reducir los costos de desarrollo en el proyecto, permitiendo que el código sea modular, compacto y fácil de entender.

11.5. Recomendaciones para trabajos futuros

El presente trabajo presentado en este documento, puede ser extendido de varias maneras. A continuación se presenta algunas recomendaciones para trabajos futuros.

- **Dar soporte completo al parser de HTML/XHTML, considerando las descripciones de un DTD (*Document Type Defintion*).** Básicamente, se puede implementar de 2 formas: (a) escribir un parser para DTD que genere otro parser para HTML/XHTML, (b) implementar el parser de HTML/XHTML respetando las reglas de un DTD.
- **Extender el soporte para la especificación de CSS.** Esto implica dar soporte a tablas y posicionamientos (flotante y absoluto). Además, se debe dar soporte para más propiedades de CSS (Existe como 80 propiedades de CSS para la renderización en la pantalla, de las cuales sólo se implementó 48).
- **Delegar la tarea de dimensionamiento y posicionamiento a la librería gráfica.** Actualmente el dimensionamiento y posicionamiento es realizado de forma manual (se

calcula las dimensiones y posiciones manualmente para cada box), lo cual causa el consumo de mucho tiempo en el redimensionamiento. Si la librería gráfica haría este trabajo, el redimensionamiento sería eficiente.

- **Mejorar la implementación del algoritmo para la asignación de valores a las propiedades.** Para esta parte, se puede considerar el árbol lexicográfico para las reglas de CSS del Navegador Web Firefox (Sección 2.5.1, página 15).
- **Implementar Javascript.** La mayoría de los actuales Navegadores Web tienen soporte para Javascript. La implementación implicaría desarrollar un parser para Javascript, y un motor de ejecución de código Javascript.

Apéndice A

Tutorial para la librería uu-parsinglib

En este apéndice encontraras un tutorial para el manejo de la librería *uu-parsinglib*, el cual está en base al Curso de Compiladores de la Universidad de Utrecht(Doaitse Swierstra, 2004, cap. 2), como también en el reporte técnico de (Swierstra, 2008).

Además, como una aplicación del tutorial, se desarrollará un módulo de combinadores elementales o básicos.

A.1. Librería *uu-parsinglib*

La librería *uu-parsinglib* es una herramienta **EDSL**[1] para *Haskell* que permite procesar una entrada a través de una descripción similar a la *Sintaxis Concreta* del lenguaje que se quiere procesar.

Entre los beneficios que *uu-parsinglib* ofrece, se tiene:

- Usar el mismo mecanismo de abstracción, tipado y nombrado de *Haskell*.
- Crear *parsers* al vuelo o en tiempo de ejecución del programa.
- No depender de otros programas separados para generar el parser. Hacer todo en *Haskell*.
- Usar el mismo formalismo para describir scanners y parsers.
- Usar el mismo formalismo para describir funciones semánticas y parsers.
- Trabajar con versiones limitadas de gramáticas infinitas.

Otro de los beneficios que tiene la librería es el de corregir los errores en la entrada, así el resultado que devuelva el *Parser* estará de acuerdo a la gramática del lenguaje.

[1] EDSL: Embeded Domain Specific Language

A.2. Módulo *Parser* e Interfaces

En esta sección se define el módulo para los combinadores básicos, junto con las funciones básicas para que se puedan probar los ejemplos de las siguientes secciones.

Junto con la definición el módulo se debe importar las librerías que se va a utilizar, por ejemplo: la librería *uu-parsinglib* y la librería *Data.Char*:

```
module CombinadoresBasicos where
import Text.ParserCombinators.UU
import Data.Char
```

Código Haskell 93: Módulo CombinadoresBasicos

Luego se continúa con la definición de la función *parseIO*, que llama a la función principal *parse* de *uu-parsinglib*.
La función *parse* se encarga de parsear la entrada con el parser que recibe como argumento:

```
parseIO :: Parser a → [Char] → IO a
parseIO p input
   = do let (res, err) = parse ((,) <$> p <* > pEnd) (listToStr input (0, 0))
        show_errors err
        return res
```

Código Haskell 94: Función *parseIO*

También se define 2 funciones que faciliten el procesamiento de *String* y Archivos:

```
parseString :: Parser a → [Char] → IO a
parseString = parseIO
parseFile :: Parser a → FilePath → Paser a
parseFile p file = do input ← readFile file
                      parseString p input
```

Código Haskell 95: Función parseString y parseFile

A.3. Combinadores de Parsers básicos

Antes de describir los combinadores básicos se debe conocer algunas ideas básicas sobre los combinadores de parsers:

- Cada 'No Terminal' de la gramática corresponde a un *Parser*.
- Cada *Parser* es representado por una función *Haskell*.
- Las funciones especiales (también llamadas combinadores) combinan *parsers* en nuevos *parsers*.
- Los *parsers* son ciudadanos de primera clase, de manera que pueden ser pasados como argumentos, y ser devueltos como resultados. Como consecuencia, el lenguaje de la

Gramática Libre de Contexto (ejemplo BNF) es extendido con los mecanismos convencionales de abstracción de *Haskell*.

- Como los *parsers* están escritos en *Haskell*, se benefician gratuitamente de la revisión de tipos de *Haskell* para las funciones semánticas.

A.3.1. pSym

pSym es una de las funciones más básicas de *uu-parsinglib*. Esta función permite construir un *Parser* que reconozca el parámetro que tiene como argumento.

pSym tiene 3 formas de utilización: reconocer un caracter simple, reconocer un rango de caracteres, y reconocer un caracter a través de una función.

Reconocer un caracter

Si se envía un caracter a la función *pSym*, este reconocerá el argumento que recibe.
Por ejemplo, en una sesión de `ghc-interactivo` (*ghci CombinadoresBasicos.hs*) se puede hacer:

```
*ParsersBasicos> parseString (pSym 'a') "a"
'a'
```

Descripción 31 Ejemplo sencillo con *pSym*

Si se envía una entrada incorrecta a *pSym*, la librería corregirá la entrada:

```
*ParsersBasicos> parseString (pSym 'a') "b"
-- >    Deleted  'b' at position (0,0) expecting 'a'
-- >    Inserted 'a' at position (0,1) expecting 'a'
'a'
```

Descripción 32 Ejemplo de corrección de errores con *pSym*

Reconocer un rango de caracteres

La segunda forma de *pSym* es reconocer una rango de caracteres. La forma es *pSym* (x, y), donde el rango está dado por: [x,y].

Por ejemplo, si se quiere reconocer un dígito:

```
*ParsersBasicos> parseString (pSym('0','9')) "2"
'2'
*ParsersBasicos> parseString (pSym('0','9')) "9"
'9'
*ParsersBasicos> parseString (pSym('0','9')) "a"
-- >    Deleted  'a' at position (0,0) expecting '0'..'9'
-- >    Inserted '0' at position (0,1) expecting '0'..'9'
'0'
```

Descripción 33 Ejemplo para reconocer un rango de caracteres con *pSym*

Con esta forma es posible reconocer algunos símbolos básicos de una gramática.
Por ejemplo:

```
pNumero          :: Parser Char
pNumero          = pSym ('0', '9')
pMinuscula       :: Parser Char
pMinuscula       = pSym ('a', 'z')
pMayuscula       :: Parser Char
pMayuscula       = pSym ('A', 'Z')
pHexadecimalChar :: Parser Char
pHexadecimalChar = pSym ('a', 'f')
```

Código Haskell 96: Ejemplos de combinadores simples, versión 1

Reconocer un caracter a través de una función

Esta última forma generaliza las dos anteriores formas. *pSym* recibe 3 parámetros, una función de tipo $Char \rightarrow Bool$ que se encarga de reconocer un caracter, una cadena de descripción de la función y un caracter por defecto, que es usado en caso de encontrar errores.

Como ejemplo se reescribirá las funciones que se definió en Código Haskell 96:

```
pNumero2     :: Parser Char
pNumero2     = pSym (isDigit, "digito", '0')
pMinuscula2  :: Parser Char
pMinuscula2  = pSym (isLower, "minuscula", 'a')
pMayuscula2  :: Parser Char
pMayuscula2  = pSym (isUpper, "mayuscula", 'A')
pHexadecimal :: Parser Char
pHexadecimal = pSym (isHexDigit, "hexadecimal", 'a')
```

Código Haskell 97: Ejemplos de combinadores simples, versión 2

A.3.2. pReturn

pReturn es un *Parser* especial que reconoce la cadena vacía y retorna el símbolo que recibe como argumento.

Ejemplo:

```
*ParsersBasicos> parseString (pReturn 'a') ""
'a'
*ParsersBasicos> parseString (pReturn 'a') "b"
-- >    The token 'b' was not consumed by the parsing process.
'a'
*ParsersBasicos> parseString (pReturn 'a') "a"
-- >    The token 'a' was not consumed by the parsing process.
'a'
```

Descripción 34 Ejemplos con *pReturn*

Note que el único caso en que no devuelve errores es cuando se le envía una cadena vacía.

A.3.3. <|>

El combinador <|>, llamado "combinador alternativo", tiene la función de combinar 2 o más producciones en un nuevo parser que reconozca todas las alternativas.

Este combinador es similar a la función **case of** de *Haskell*, donde todas las alternativas devuelven un determinado tipo de resultado. De la misma manera, todas las alternativas de este combinador deben tener un mismo tipo.

Una aplicación sencilla del combinador <|> es cuando se quiere reconocer letras mayúsculas y minúsculas:

$$pLetra :: Parser\ Char$$
$$pLetra = pMinuscula < | > pMayuscula$$

Código Haskell 98: Ejemplos sencillos

A.3.4. pFail

El combinador *pFail* es un combinador especial que siempre falla sin importar la entrada que tenga. Por ejemplo:

```
*ParsersBasicos> parseString pFail "a"
*** Exception: no correcting alternative found
```

Descripción 35 Ejemplo de error con *pFail*

Una aplicación importante es cuando se tiene al combinador *pFail* como una de las alternativas de <|>, en ese caso siempre se prefiere revisar las otras alternativas diferentes a *pFail*. Por ejemplo:

```
*ParsersBasicos> parseString (pFail <|> pSym 'a') "a"
'a'
```

Descripción 36 Ejemplo de aplicación de *pFail*

A.3.5. <*>

El combinador <*>, llamado "combinador de composición secuencial", combina 2 parsers en uno nuevo. La forma de combinar los 2 parsers es aplicando el resultado del primer parser al resultado del segundo.

Por ejemplo, se quiere reconocer un número y añadirle un cero después de reconocerlo:

```
*ParsersBasicos> parseString (pReturn (:'0':[]) <*> pNumero ) "1"
"10"
*ParsersBasicos> parseString (pReturn (:'0':[]) <*> pNumero ) "2"
"20"
*ParsersBasicos> parseString (pReturn (:'0':[]) <*> pNumero ) "3"
"30"
```

Descripción 37 Ejemplo con el combinador secuencial

En el ejemplo se tiene un primer parser *pNumero* que retorna un caracter número, y un segundo parser *pReturn* que devuelve una función que está esperando un caracter para añadirlo junto con un cero a una lista. Así, el resultado del primer parser se aplica al resultado del segundo parser.

Vea que el primer parser es el que está más a la derecha, porque <*> tiene una asociación hacia la derecha, lo que permite usar más de dos parsers sin tener que agruparlos entre paréntesis.

Otro ejemplo: se quiere reconocer una letra mayúscula, un número y una letra minúscula. Y agruparlos en una tri-tupla:

```
*ParsersBasicos> parseString (pReturn (,,)  <*> pMayuscula
                                            <*> pNumero
                                            <*> pMinuscula) "A2a"
('A','2','a')
*ParsersBasicos> parseString (pReturn (,,)  <*> pMayuscula
                                            <*> pNumero
                                            <*> pMinuscula) "B2b"
('B','2','b')
*ParsersBasicos> parseString (pReturn (,,)  <*> pMayuscula
                                            <*> pNumero
                                            <*> pMinuscula) "C3a"
('C','3','a')
```

Descripción 38 Ejemplo con el combinador secuencial

A.3.6. <<|>

El combinador <<|> es un combinador especial alternativo, este combinador siempre que puede, da preferencia al parser que se encuentra en el lado izquierdo, y no hace nada con la alternativa que se encuentra en el lado derecho. Siempre que puede significa encontrar un resultado válido.

En caso de no encontrar un resultado en el lado izquierdo, revisará el lado derecho.

Se puede reescribir este combinador para la función opcional 'opt':

```
p `opt` value = p <<|> pReturn value
```

Descripción 39 Ejemplo con el combinador especial alternativo

A.4. Combinadores Derivados

En esta sección se describirá algunos de los combinadores derivados. Para una descripción detallada, se puede revisar la documentación de librería que corresponde a 'Derived'.

A.4.1. Combinadores derivados simples

En base a los combinadores de la anterior sección, la librería define nuevos combinadores para facilitar su manejo y tener una mejor expresividad:

```
f <$> p = pReturn f   <*> p
f <$  p = const   f   <$> p
p <*  q = (\x _ -> x) <$> p <*> q
p *>  q = (\_ x -> x) <$> p <*> q
```

Descripción 40 Definición de combinadores derivados

A.4.2. Combinadores Secuenciales

pList, pList1, pListSep, pList1Sep

Estos combinadores permiten reconocer una lista de símbolos especificados por un parser.

Por ejemplo, se puede usar *pList* para reconocer una lista de *cero o más* espacios, y devolver como resultado el número de espacios que se ha reconocido:

```
*ParsersBasicos> parseString (length <$> pList (pSym ' ')) "   "
3
*ParsersBasicos> parseString (length <$> pList (pSym ' ')) "      "
6
*ParsersBasicos> parseString (length <$> pList (pSym ' ')) ""
0
*ParsersBasicos> parseString (length <$> pList (pSym ' ')) " "
1
```

Descripción 41 Ejemplos con *pList*

La otra variante de *pList* es *pList1*, este combinador reconoce una lista de *uno o más* símbolos (de ahí su nombre *pList1*), mientras que *pList* reconoce una lista de *cero o más* símbolos.

Con estos nuevos combinadores se puede reconocer palabras, números y otras secuencias de símbolos:

```
palabra  :: Parser String
palabra  = pList1 pLetra
natural  :: Parser String
natural  = pList1 pNumero
espacios :: Parser String
espacios = pList1 (pSym ' ')
```

Código Haskell 99: Combinadores para lista de símbolos

A continuación se muestra algunos ejemplos para los combinadores del Código Haskell 99:

```
*ParsersBasicos> parseString palabra "carlos"
"carlos"
*ParsersBasicos> parseString natural "1231"
"1231"
*ParsersBasicos> parseString espacios " "
" "
```

Descripción 42 Ejemplos para los combinadores definidos en Código Haskell 99

Si se quiere reconocer una lista de palabras separadas por espacios, o una lista de números separados por comas, entonces se puede utilizar los combinadores *pListSep* para reconocer una lista de *cero o más* símbolos o *pList1Sep* para reconocer una lista de *uno o más* símbolos.

Ambos combinadores reciben un parser para el separador y otro para el símbolo a reconocer. Al construir el resultado estos desechan el separador y sólo consideran el símbolo.

Por ejemplo:

```
*ParsersBasicos> parseString (pListSep espacios palabra) "carlos"
["carlos"]
*ParsersBasicos> parseString (pListSep espacios palabra) "carlos  gomez"
["carlos","gomez"]
*ParsersBasicos> parseString (pList1Sep (pSym ',') natural) "1"
["1"]
*ParsersBasicos> parseString (pList1Sep (pSym ',') natural) "1,2,3,4,5"
["1","2","3","4","5"]
```

Descripción 43 Ejemplos de combinadores con *pListSep* y *pList1Sep*

A.5. Módulo de Combinadores Elementales

En la sección A.2 se ha definido el módulo e interfaces para comunicarse con la librería, en esta sección se definirá los combinadores básicos.

A.5.1. pInutil

Se comienza con la definición de un combinador que es muy utilizado. Es común encontrar en las gramáticas símbolos que no son necesarios (es decir inútiles), por ejemplo: espacios, saltos de línea, retornos de carro, tabs.

En algunos casos puede no haber alguno de estos, para ello se define *pInutil*, pero en otros casos debe al menos existir uno de ellos, para esos casos se define *pInutil1*.

```
pInutil  :: Parser String
pInutil  = pList (pAnySym " \n\r\t")
pInutil1 :: Parser String
pInutil1 = pList1 (pAnySym " \n\r\t")
```

Código Haskell 100: Combinadores elementales

A.5.2. pSimbolo y variaciones

Otra de las tareas comunes es el de reconocer un símbolo compuesto de uno o más caracteres, así como en el caso de *pSimbolo*.

También puede darse el caso de que se quiera reconocer un símbolo que por el lado derecho, izquierdo o ambos tiene caracteres inútiles. En esos casos, se desecha los caracteres inútiles y sólo se devuelve el símbolo reconocido.

```
pSimbolo :: String → Parser String
pSimbolo = pToken
pSimboloIzq :: String → Parser String
pSimboloIzq str = pInutil *> pToken str
pSimboloDer :: String → Parser String
pSimboloDer str = pToken str <* pInutil
pSimboloAmb :: String → Parser String
pSimboloAmb str = pInutil *> pToken str <* pInutil
```

Código Haskell 101: Combinadores elementales, símbolos

A.5.3. Dígitos, Hexadecimales y Números

Entre los otros combinadores básicos están los de dígitos y hexadecimales. Estos están definidos con *pDigitChar* y *pHex*.

En algunos casos se necesita reconocer el signo de un número positivo o negativo, en otros casos sólo se necesita reconocer el signo de un número positivo.

Pero en ambos casos reconocer el signo es opcional, es por eso que se utiliza el combinador *pMaybe* para reconocer el signo de un número.

```
pDigitoChar :: Parser Char
pDigitoChar = pSym (isDigit, "digito", '0')
pHex :: Parser Char
pHex = pSym (isHexDigit, "digito hexadecimal", 'a')
pSigno :: Parser (Maybe Char)
pSigno = pMaybe (pSym '+' <|> pSym '-')
pSignoMas :: Parser (Maybe Char)
pSignoMas = pMaybe (pSym '+')
```

Código Haskell 102: Combinadores elementales, básicos

Reconocer un número implica convertir una cadena en el tipo correcto que se necesita (*Int* o *Float*). La conversión es realizada utilizando la función polimórfica *read*. Se envía la cadena que se quiere convertir y la función *read* devuelve el número en el tipo deseado.

El número puede tener un signo, si el signo es negativo, se multiplica el número por -1, y si es positivo o si no tiene signo se multiplica por 1.

```
toFloat :: Maybe Char → String → Float
toFloat sg str = signo sg * numero
  where numero = read str
        signo = maybe 1 valorSigno
        valorSigno '+' = 1
        valorSigno '-' = −1
toInt :: Maybe Char → String → Int
toInt sg str = signo sg * numero
  where numero = read str
        signo = maybe 1 valorSigno
        valorSigno '+' = 1
        valorSigno '-' = −1
```

Código Haskell 103: Combinadores elementales, funciones

Entonces, para reconocer un número entero, se reconoce un signo opcional seguido de una lista de uno o más caracteres dígitos. En el caso de querer reconocer un número positivo, se reconoce un signo positivo opcional seguido de la lista de caracteres dígitos:

```
pEntero :: Parser Int
pEntero     = toInt <$> pSigno <*> pList1 pDigitoChar
pEnteroPos :: Parser Int
pEnteroPos = toInt <$> pSignoMas <*> pList1 pDigitoChar
```

Código Haskell 104: Combinadores elementales, números

Para reconocer un número *float* se debe distinguir 3 formas en que un número *float* se puede presentar:

- *Digitos*: Un número float puede ser simplemente una lista de dígitos.
- *Con punto en medio*: Un número float puede tener un punto en medio de los dígitos. Ejemplo: *123.45*

- *Con punto al inicio*: Un número float puede comenzar con un punto y luego los dígitos. Ejemplo: *.125* que se considera como si fuera *0.125*

En las 3 formas, el signo que viene al principio es opcional.

```
pNumeroFloat :: Parser Float
pNumeroFloat
  =     toFloat <$> pSigno <*> pList1 pDigitoChar
  <|> (λsg n1 d n2 → toFloat sg (n1 ++ [d] ++ n2))
                <$> pSigno <*> pList1 pDigitoChar <*> nums
  <|> (λsg d n2 → toFloat sg ("0" ++ [d] ++ n2))
                <$> pSigno                        <*> nums
  where nums = pSym '.' <*> pList1 pDigitoChar
pNumeroFloatPos :: Parser Float
pNumeroFloatPos
  =     toFloat <$> pSignoMas <*> pList1 pDigitoChar
  <|> (λsg n1 d n2 → toFloat sg (n1 ++ [d] ++ n2))
                <$> pSignoMas <*> pList1 pDigitoChar <*> nums
  <|> (λsg d n2 → toFloat sg ("0" ++ [d] ++ n2))
                <$> pSignoMas                        <*> nums
  where nums = pSym '.' <*> pList1 pDigitoChar
```

Código Haskell 105: Combinadores elementales, números

A.5.4. Combinadores para texto

El combinador básico para un texto es reconocer un caracter alfanumérico, así como *pAlphaNum*.

Luego se puede reconocer una palabra con *pPalabra*.

```
pAlphaNum :: Parser Char
pAlphaNum = pSym (isAlphaNum, "alpha num", 'a')
pPalabra :: Parser String
pPalabra = pList1 pAlphaNum
```

Código Haskell 106: Combinadores elementales, palabras

En muchos casos se necesita reconocer un texto donde se quiere restringir caracteres no deseados. Así, se ha definido el combinador *pTextoRestringido*, que restringe la lista de caracteres que recibe como argumento.

```
pTextoRestringido :: String → Parser String
pTextoRestringido deny = pList1 (pSym (fcmp, text, ' '))
  where fcmp = ¬ ∘ (∈ deny)
        text = "diferente a " ++ show deny
pHTMLTexto :: Parser String
pHTMLTexto = pTextoRestringido "</>"
```

Código Haskell 107: Combinadores elementales, textos

Como ejemplo se ha definido el combinador *pHTMLTexto* del Código Haskell 107 que restringe los caracteres '`</>`'.

A.5.5. Combinadores para Strings

Por último, se define los combinadores para reconocer *Strings*. Un *String* puede presentarse de dos formas distintas: encerradas entre comillas simples, o encerradas entre comillas dobles.

Además se tiene 2 tipos de Strings: una simple que contiene sólo texto, y otra compleja que puede tener cualquier caracter excepto el de salto de línea y el caracter que se utiliza para limitarlo.

Para esto, se ha definido un combinador *pDeLimitarCon* que recibe el parser para el delimitador y el parser para el contenido.

En la definición del combinador *pDeLimitarCon* se está utilizando el combinador *pPacked*, que tiene 3 parámetros, los primeros 2 son los delimitadores, y el 3ro es el parser para el contenido.

```
pDeLimitarCon :: Parser a → Parser b → Parser b
pDeLimitarCon d c = pPacked d d c
pSimpleString   =   pDeLimitarCon (pSym '\"') pPalabra
                <|> pDeLimitarCon (pSym '\'') pPalabra
pComplexString  =   pDeLimitarCon (pSym '\"') (pTextoRestringido "\"\n")
                <|> pDeLimitarCon (pSym '\'') (pTextoRestringido "\'\n")
```

Código Haskell 108: Combinadores elementales, delimitadores

Apéndice B

Tutorial para la librería UUAGC

En este apéndice hallarás un tutorial simple para la librería *UUAGC*[1](Swierstra, s.f.-a). El lector interesado en una descripción completa puede revisar la página Web de la librería y encontrar el manual de *UUAGC*.

Además de mostrar un tutorial simple, este apéndice se enfoca en mostrar el flujo de información que ocurre cuando se utiliza la librería *UUAGC* para definir la semántica de la gramática.

Como una aplicación del apéndice, se desarrollará un módulo que genere las posiciones, dimensiones y líneas para renderizar una estructura de árbol.

B.1. Introducción

Si se tiene el siguiente código HTML:

```
<html>
    <head> <style>
                estilo
           </style>
    </head>
    <body>
        <p> texto1 </p>
        <p> texto2 </p>
    </body>
</html>
```

Descripción 44 Ejemplo de HTML

El cual puede ser representado en Haskell, sin considerar el texto, de la siguiente manera:

[1]UUAGC: Utrecht University Atribute Grammar Compiler

```
FSBox "html" [FSBox "head" [FSBox "style" [FSBox "texto" []]]
             , FSBox "body" [FSBox "p" [FSBox "texto" []]
                            , FSbox "p" [FSBox "texto" []]
                            ]
             ]
```

Código Haskell 109: Representación Haskell de la Descripción 44

y lo que se quiere, es dibujar una estructura de árbol que represente el código HTML de la Descripción 44:

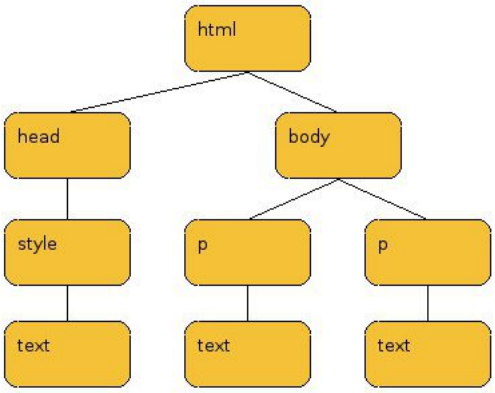


Figura B.1: Renderización del ejemplo de la Descripción 44

Entonces, para renderizar la Figura B.1 que representa la Descripción 44, se debe generar una posición y dimensión para cada *FSBox* del Código Haskell 109. También se debe generar las líneas entre cada *FSBox*.

En este apéndice se utilizará la librería *UUAGC* para generar toda la información que se va a renderizar: posición, dimensión y líneas.

La librería *UUAGC* es una herramienta que permite describir la semántica de una gramática a través de atributos. Lo interesante es que la librería permite utilizar código Haskell para la descripción de la semántica.
La librería *UUAGC* es un lenguaje que tiene su propia sintaxis, pero que genera código Haskell, que puede ser utilizado en cualquier compilador de Haskell.

Se ha utilizado la librería *UUAGC* porque permite describir la semántica de la gramática de manera sencilla, comprensible y en Haskell, además porque permite ahorrar la cantidad de código que se tiene que escribir.

En las siguientes secciones, primero se describirá algunos elementos de la librería *UUAGC* (los que se ha considerado importantes) y luego se describirá la forma de generar las posiciones, dimensiones y líneas para el *FSBox*.

B.2. Declaraciones DATA

La librería *UUAGC* permite definir una gramática como una colección de declaraciones **DATA**.

Un **DATA** declara un *No-Terminal* y sus producciones, donde cada producción contiene el nombre del constructor y campos que contiene la producción. Cada campo debe tener un nombre único y tipo.

Por ejemplo, la declaración de *DATAs* para el *FSBox* que se usará es:

```
DATA FSBox
  | FSBox name : String
         boxes : FSBoxes
TYPE FSBoxes = [FSBox]
```

Código UUAGC 33: Declaración DATA para FSBox

B.3. Descripción del comportamiento con UUAGC

La librería *UUAGC* permite especificar atributos y semánticas para describir el comportamiento que se desea implementar.

En otras palabras, la información que se desea procesar es almacenada en atributos, pero la *forma* de procesar la información es descrita a través de la semántica.

B.3.1. Atributos de UUAGC

Un atributo abstrae la información que se va a procesar y la forma de flujo de información que se va a realizar.

Se tiene, básicamente, dos formas de flujo: de *abajo-arriba* y de *arriba-abajo*. Por ejemplo, se puede sumar una lista de enteros de dos formas:

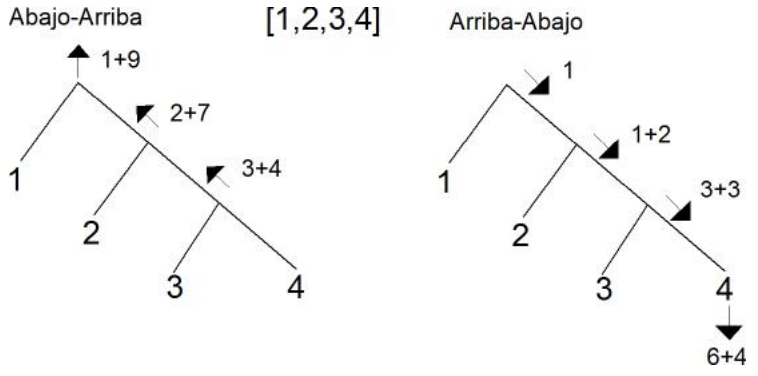


Figura B.2: Flujos de información para sumar una lista de enteros

Entonces, de acuerdo al flujo de información, se tiene 3 formas de atributos:

- **Atributos Heredados**. Son atributos que tienen un flujo de información de *arriba-abajo*.
- **Atributos Sintetizados**. Son atributos que tienen un flujo de información de *abajo-arriba*.
- **Atributos Encadenados**. Son atributos que son heredados y sintetizados al mismo tiempo.

Para declarar un atributo se debe seguir la siguiente estructura:

```
ATTR No_Terminales  [ atributos_Heredados
                    | atributos_Encadenados
                    | atributos_Sintetizados
                    ]
```

Descripción 45 Estructura para declarar un atributo

Donde `No_Terminales` es una lista de No-Terminales separados con espacios, `atributos_Heredados`, `atributos_Encadenados`, `atributos_Sintetizados` son una lista de declaraciones separados por espacios que tienen la siguiente forma:

```
nombreAttributo : tipoAttributo
```

El *tipoAtributo* puede ser simple (*Int*, *Float*, *String*, ...) o complejo ([*Int*], *Map String String*, *Maybe Int*), cuando es complejo debe estar encerrado entre llaves ({[*Bool*]}).

Algunos ejemplos de declaraciones de atributos son:

ATTR *Arbol* [| | *valmin* : *Int*]
ATTR *Arbol* [*ming* : *Int* | |]
ATTR *Arbol* [*profundidad* : *Int* | *minimo* : *Int* | *salida* : {[*Bool*]}]
ATTR *Decl* [*contador1* : *Int* *contador2* : *Int* | | *contador* : *Int*]

Código UUAGC 34: Ejemplos de declaraciones de Atributos con *UUAGC*

B.3.2. Especificación de la semántica con *UUAGC*

La semántica define el como se va procesar la información de un atributo. La librería *UUAGC* provee la estructura **SEM** para la especificación de la semántica:

SEM *Noterminal*
| $Constructor_1$ $referencia_1.nombreAtributo_1 = expresionHaskell_1$
$referencia_2.nombreAtributo_2 = expresionHaskell_2$
...
| $Constructor_n$ $referencia_n.nombreAtributo_n = expresionHaskell_n$

Descripción 46 Estructura para declarar la semántica

La estructura **SEM** se utiliza para definir la semántica para un No-Terminal. Este es definido a través de una expresión de Haskell para una producción y atributo determinado.

Para referirse a un atributo se utiliza una referencia (**lhs**,**loc**,*nombreProduccion*) y el nombre de un atributo. Las referencias a los atributos también pueden aparecer en la expresión Haskell, pero deben estar prefijadas con el símbolo '@'. Los atributos **loc** son variables locales a nivel de la producción que se definen directamente en la estructura **SEM**.

Dependiendo el lugar donde se encuentren las referencias a los atributos, lado derecho (dentro expresión Haskell) o lado izquierdo (ver Descripción 46), tienen diferentes significados:

- **lhs**: Si se encuentra en el lado izquierdo, hace referencia a un atributo sintetizado, pero si se encuentra en el lado derecho (dentro de la expresión Haskell) hace referencia al atributo heredado del No-Terminal padre del actual No-Terminal.
- **nombreProduccion**: Si se encuentra en el lado izquierdo, hace referencia al atributo heredado, pero si se encuentra en el lado derecho (dentro de la expresión Haskell) hace referencia al atributo sintetizado del *nombreProduccion*.
- **loc**: En ambos lados hacen referencia al mismo atributo.

B.3.3. Declaraciones TYPE

La estructura *TYPE* permite definir No-Terminales comunes utilizando una sintaxis especial.

Por ejemplo, se puede definir listas de la siguiente manera:

```
TYPE Enteros = [Numero]
```

Código UUAGC 35: Sintaxis especial para la definición de listas

La definición del Código UUAGC 35 genera la siguiente declaración **DATA**:

```
DATA Enteros
  | Cons hd : Numero
       tl  : Enteros
  | Nil
```

Código UUAGC 36: Definición DATA para listas

Además de listas, la librería también provee una sintaxis especial para declarar tuplas, estructuras *Map*, *Maybe*, *Either*, *IntMap*.

B.4. Generar la información

En esta sección se procederá a generar la información necesaria para renderizar: posiciones, dimensiones y líneas.

Se inicia definiendo un **DATA** *Root* el cual se encarga simplemente de marcar el nodo Root de la estructura de árbol:

DATA *FSRoot*
| *FSRoot fsbox* : *FSBox*

Código UUAGC 37: Definición de Root

B.4.1. Generando la posición 'y'

Se comenzará con la generación de la posición 'y' para cada *FSTree* del Código UUAGC 33.

Inicialmente, se tiene una posición 'y' inicial donde se comenzará a dibujar: $yInit = 10$.

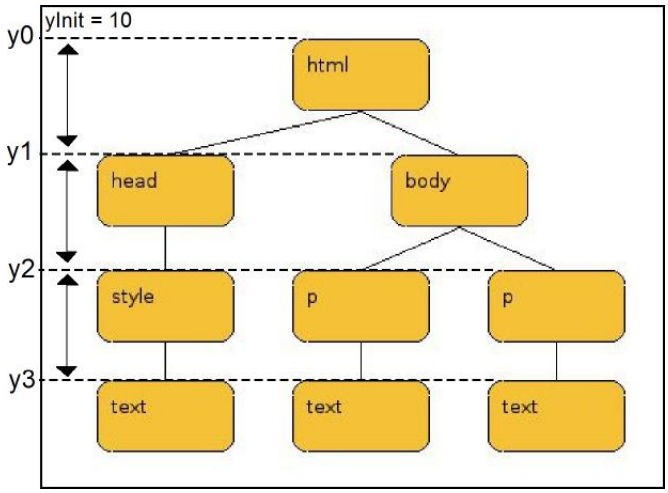


Figura B.3: Posición 'y' para cada *FSBox*

A partir de la posición inicial se genera la posición 'y' para cada *FSTree*. Para calcular la posición 'y' para los nodos hijos de un *FSTree* se incrementa una distancia, así como se ve en la Figura B.3.

Entonces, la forma de asignar una posición 'y' a cada *FSTree* es utilizando un atributo heredado (*arriba-abajo*) para *FSBox* y *FSBoxes*:

ATTR *FSBox FSBoxes* [*yPos* : *Int* | |]

Código UUAGC 38: Atributo heredado *yPos*

Se especifica la posición inicial en el nodo *Root*, el lugar donde comienza todo el *FSTree*:

```
SEM FSRoot
  | FSRoot fsbox.yPos = 10
```

Código UUAGC 39: Posición inicial 'y'

Se crea una variable local (sólo por motivos didácticos) *yPos* para cada *FSBox*, el cual guarda la posición 'y' del *FSBox*.

```
SEM FSBox
  | FSBox loc.yPos   = @lhs.yPos
          boxes.yPos = @loc.yPos + ySep
```

Código UUAGC 40: Posición 'y' para *FSBox*

En el Código UUAGC 40 se especifica que la posición 'y' (*yPos*) para los hijos (*boxes*) es la posición del *FSBox* (**@loc**.*yPos*) más una cantidad de separación (*ySep*). La cantidad de separación es una constante que equivale a: 80.

En el No-Terminal *FSBoxes*, el atributo *yPos* se copia a cada elemento:

```
SEM FSBoxes
  | Cons hd.yPos = @lhs.yPos
         tl.yPos = @lhs.yPos
```

Código UUAGC 41: Posición 'y' para *FSBoxes*

En el Código UUAGC 41, sólo se declara la semántica para la producción *Cons* porque la producción *Nil* no tiene ningún campo que herede el atributo *yPos*.

Nota.- No es necesario especificar la semántica para *FSBoxes* del Código UUAGC 41, porque la librería *UUAGC* puede derivar la semántica aplicando una regla de copiado. Pero no existe ningún problema si este es especificado.

Gráficamente, las siguientes figuras: Figura B.4, Figura B.5, Figura B.6 muestran el movimiento del atributo *yPos* para el *FSRoot*, *FSBox* y *FSBoxes* respectivamente:

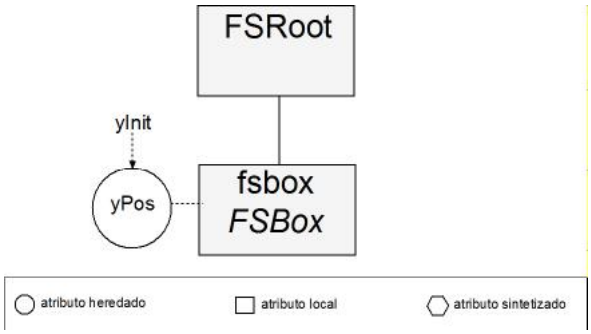


Figura B.4: Atributo *yPos* para *FSRoot*

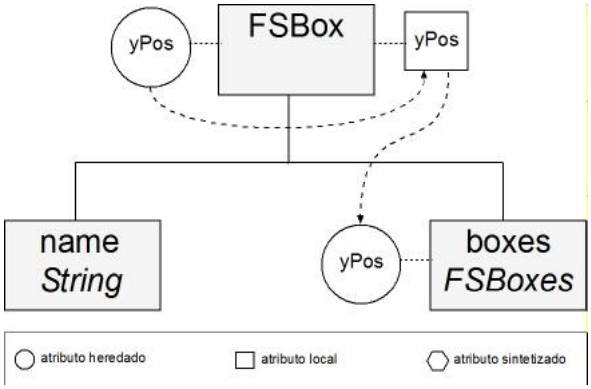


Figura B.5: Atributo *yPos* para *FSBox*

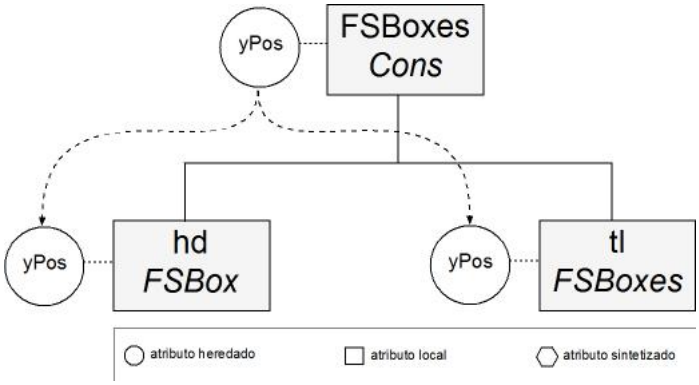


Figura B.6: Atributo *yPos* para *FSBoxex*

La Figura B.7 muestra el movimiento de información para el ejemplo de la Descripción 44 con el atributo *yPos*:

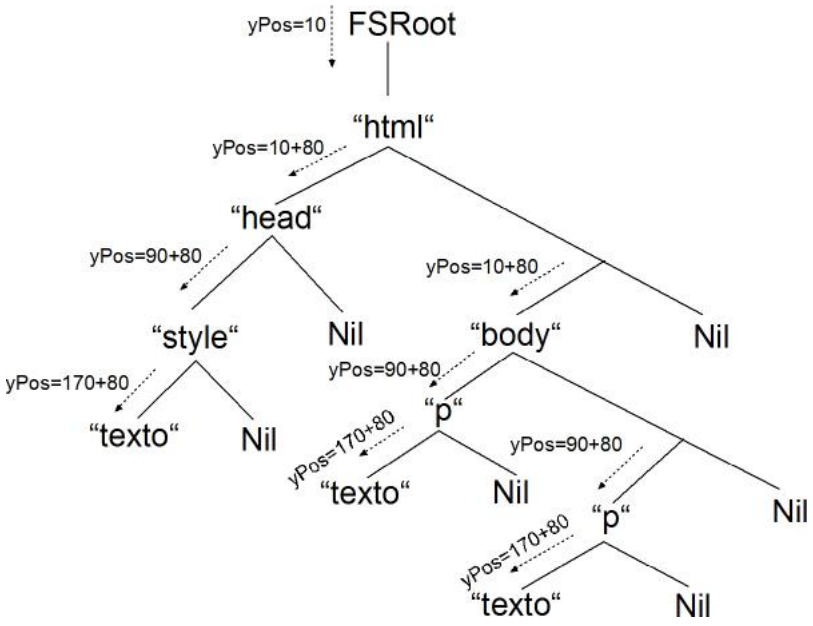


Figura B.7: Movimiento de información para el atributo *yPox*

B.4.2. Calculando el ancho que ocupa un *FSBox*

En la Figura B.8 se muestra que cada *FSBox* tiene un ancho.

El ancho de cada *FSBox* es la sumatoria del ancho de todos *FSBox* hijos. Si un *FSBox* no tiene hijos, entonces el ancho debe ser un valor por defecto: la longitud del *FSBox* más la distancia de separación, esto es:

```
anchoPorDefecto = widthBox + xSep

widthBox = 95
xSep     = 40
```

El Código UUAGC 42 muestra la especificación para calcular el ancho para el *FSBox* y *FSBoxes*.

Se utiliza un atributo sintetizado (*abajo-arriba*) *len* para *FSBox* y *FSBoxes*. También se utiliza una variable local *len* en *FSBox* para guardar el ancho que ocupa el *FSBox*.

```
ATTR FSBox FSBoxes [ | | len : Int]
SEM FSBox
  | FSBox loc.len = if@boxes.len ≡ 0
                    then widthBox + xSep
                    else@boxes.len
          lhs.len = @loc.len
SEM FSBoxes
  | Cons lhs.len = @hd.len + @tl.len
  | Nil  lhs.len = 0
```

Código UUAGC 42: Calcular el ancho de un *FSBox* y *FSBoxes*

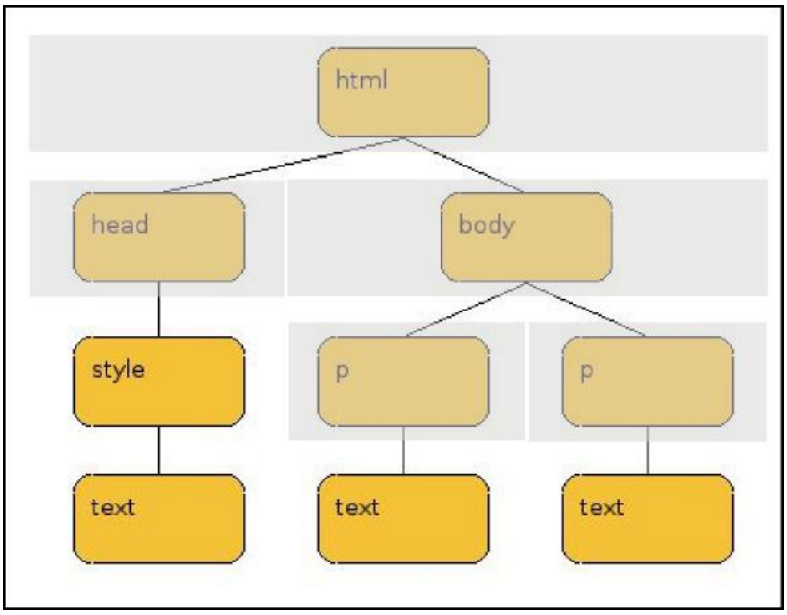


Figura B.8: Ancho de cada *FSBox*

La Figura B.9 y Figura B.10 representa el movimiento del atributo *len* para *FSBox* y *FSBoxes* respectivamente. Y la Figura B.11 muestra el movimiento de información para el ejemplo de la Descripción 44 con el atributo *len*:

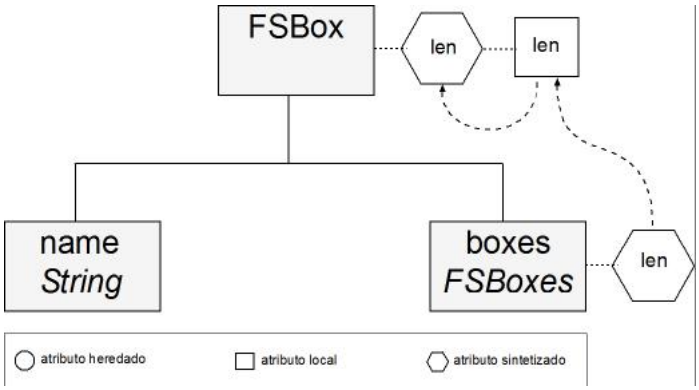


Figura B.9: Atributo *len* para *FSBox*

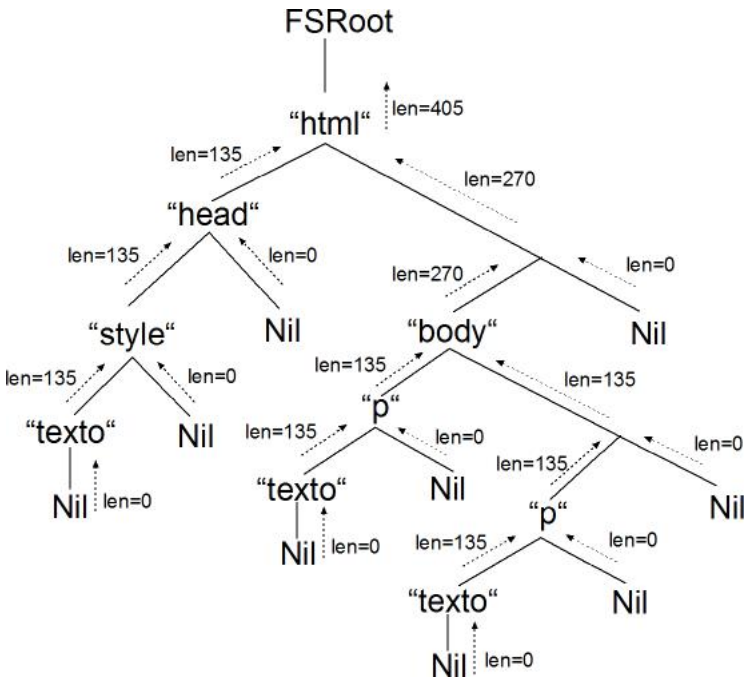


Figura B.11: Movimiento de información para el atributo *len*

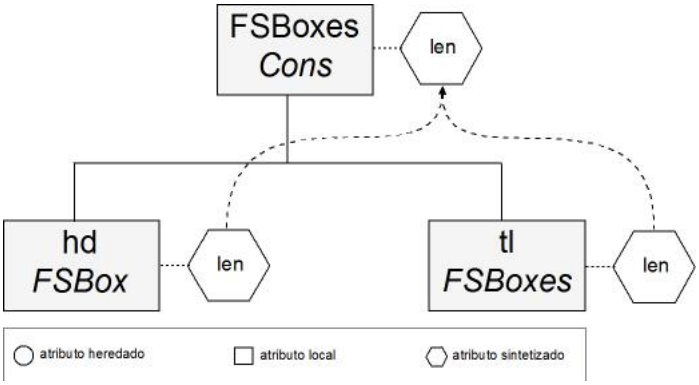


Figura B.10: Atributo *len FSBoxes*

Nota: Es posible reducir la cantidad de líneas para el Código UUAGC 42 utilizando la cláusula **USE** que la librería *UUAGC* provee. La cláusula **USE**, que se utiliza sólo en atributos sintetizados, calcula su atributo sintetizado aplicando una función a todas las producciones que tienen más de dos campos. Si la producción tiene menos de 2 campos, entonces el atributo sintetizado es un valor por defecto.

Por ejemplo, para el caso del *FSBoxes* se puede aplicar la cláusula **USE** sobre la producción *Cons* con la función +, y para la producción *Nil* se utiliza el valor por defecto 0. El Código UUAGC 43 muestra la nueva versión.

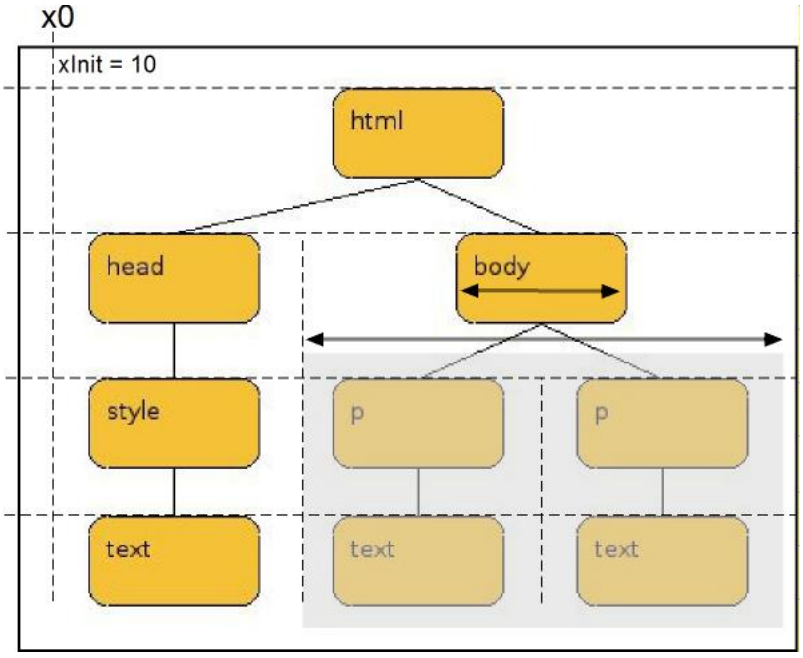


Figura B.12: Posición 'x' para *FSBox*

```
ATTR FSBox FSBoxes [ | | len USE {+} {0} : Int]
SEM FSBox
  | FSBox loc.len = if@boxes.len ≡ 0
                    then widthBox + xSep
                    else@boxes.len
          lhs.len = @loc.len
```

Código UUAGC 43: Calcular el ancho de un *FSBox* y *FSBoxes*, versión 2

B.4.3. Generando la posición 'x'

En la Figura B.12 se muestra que la asignación de la posición 'x', al igual que para 'y', tiene una posición de inicio. También se puede ver que cada *FSBox* debe estar ubicado en el centro del ancho que ocupa el mismo *FSBox*.

Para su implementación se utiliza un atributo heredado *xPos* sobre *FSBox* y *FSBoxes*. El Código UUAGC 44 muestra la especificación para calcular la posición 'x'. El atributo *xPos* para los nodos hijos (*boxes*) de un *FSBox* se inicia en el mismo valor de *xPos* heredado. Se utiliza una variable local *xPos* para guardar la posición 'x' del *FSBox*.

Para el caso del *FSBoxes*, *xPos* se incrementa con el ancho (*len*) de cada *FSBox*.

```
widthBox  = 95
heightBox = 50
```

Código Haskell 110: Dimensión para *FSBox*

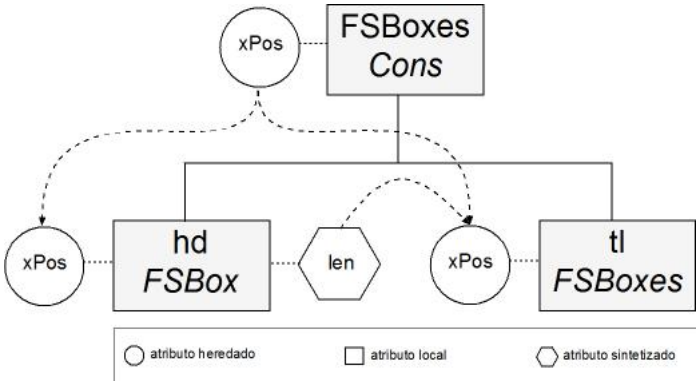


Figura B.13: Movimiento del atributo *xPos* para *FSBoxes*

ATTR *FSBox FSBoxes* [*xPos* : *Int* | |]
SEM *FSRoot*
| *FSRoot fsbox.xPos* = 10
SEM *FSBox*
| *FSBox boxes.xPos* = @**lhs**.*xPos*
loc.*xPos* = @**lhs**.*xPos* + (@**loc**.*len* ‘*div*‘ 2) + (*xSep* ‘*div*‘ 2) − (*widthBox* ‘*div*‘ 2)
SEM *FSBoxes*
| *Cons hd.xPos* = @**lhs**.*xPos*
tl.xPos = @**lhs**.*xPos* + @*hd.len*

Código UUAGC 44: Especificación para calcular la posición ‘x’

El movimiento de atributos es similar a los anteriores, pero conviene ver la Figura B.13 la cual muestra el movimiento de atributos para *FSBoxes*. Lo nuevo es que se utiliza un atributo sintetizado para calcular el valor del atributo heredado.

B.4.4. Generando puntos para las líneas

En la Figura B.14 se puede ver que se necesita que cada *FSBox* genere dos puntos para conectar las líneas. Cada punto debe estar en el centro de la parte superior e inferior de cada *FSBox*.

En el Código UUAGC 45 se muestra la especificación que genera los dos puntos para un *FSBox*. Se ha definido un atributo sintetizado *pt1* en *FSBox* de tipo (*Int*, *Int*), también se ha definido una variable local *pt2* para guardar el segundo punto.

Así mismo se ha definido un atributo sintetizado *pt1* en *FSBoxes* de tipo [(*Int*, *Int*)] que almacena la lista de puntos de todos los nodos hijos.

Para calcular el centro se utiliza las dimensiones definidas en Código Haskell 110.

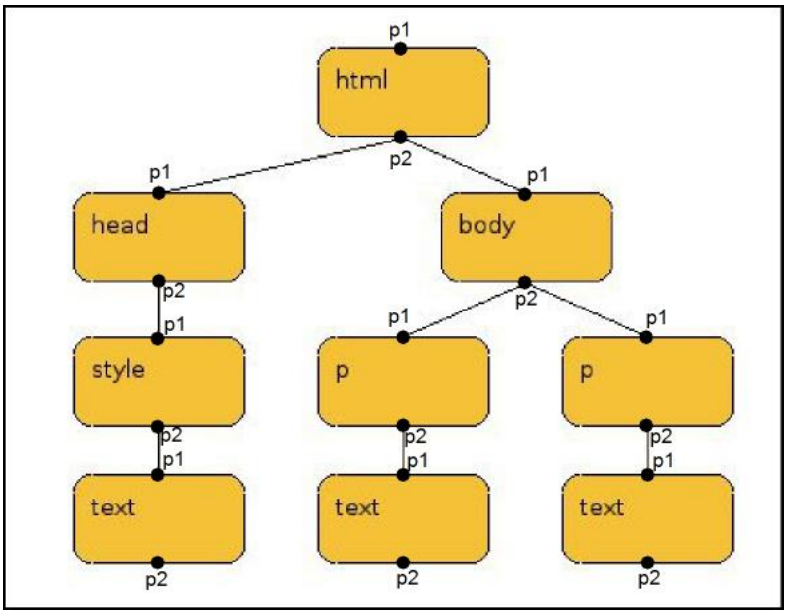


Figura B.14: Puntos para las líneas de un *FSBox*

ATTR *FSBox* [| | *pt1* : {(*Int*, *Int*)}]
SEM *FSBox*
| *FSBox* **lhs**.*pt1* = (@**loc**.*xPos* + (*widthBox* ‘*div*‘ 2), @**loc**.*yPos*)
loc.*pt2* = (@**loc**.*xPos* + (*widthBox* ‘*div*‘ 2), @**loc**.*yPos* + *heightBox*)
ATTR *FSBoxes* [| | *pt1* **USE** {:} {[]} : {[(*Int*, *Int*)]}]

Código UUAGC 45: Especificación para calcular los puntos extremos de cada línea

B.4.5. Generando información para renderizar

Ahora que se ha calculado toda la información para renderizar, se procederá a generar el resultado final.

El resultado final será una lista de objetos renderizables. Se ha definido el tipo de dato resultado *Object* en el Código UUAGC 46 el cual esta compuesto de un constructor *OBox* que tiene nombre, posición y dimensión, y también se tiene un constructor *OLine* que tiene *pt1* y *pt2*.

```
DATA Object
  | OBox name      : String
         position  : {(Int, Int)}
         dimension : {(Int, Int)}
  | OLine pt1 : {(Int, Int)}
          pt2 : {(Int, Int)}
```

Código UUAGC 46: Definición del tipo de dato para el resultado final

En el Código UUAGC 47 se muestra la especificación para obtener el resultado final. Se ha utilizado un atributo sintetizado *out* de tipo [*Object*] sobre *FSRoot*, *FSBox* y *FSBoxes* para recolectar todo el resultado.

Se ha utilizado la cláusula **USE** sobre *FSBoxes* con la función ++ (concatenar) y una lista vacía como valor por defecto.

No se ha especificado nada para *FSRoot* porque la librería *UUAGC* deriva el valor para su atributo sintetizado utilizando una copia directa del atributo sintetizado de *FSBox*. Sin embargo, sólo se ha especificado la semántica para *FSBox*. Lo primero que se hace es generar la lista de líneas *cmdVec*. Luego se genera el *OBox* con la información necesaria: nombre, posición y dimensión (son valores constantes definidos en Código Haskell 110). Finalmente se concatena todos los objetos generados.

```
ATTR FSRoot FSBox FSBoxes [ | | out USE {++} {[]} : {[Object]}]
SEM FSBox
  | FSBox lhs.out = let cmdVec = map (OLine@loc.pt2) @boxes.pt1
                        box    = OBox @name
                                      (@loc.xPos, @loc.yPos)
                                      (widthBox, heightBox)
                    in (box : @boxes.out) ++ cmdVec
```

Código UUAGC 47: Especificación de la semántica para el resultado final

Finalmente, en la Figura B.15 se muestra el movimiento del atributo *out* para *FSBox*.

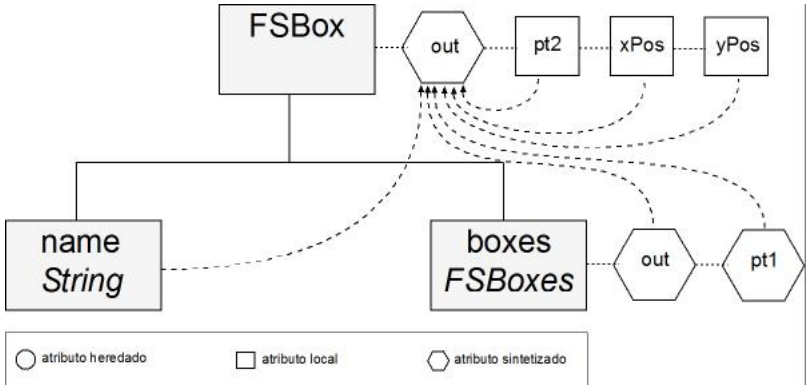


Figura B.15: Movimiento del atributo *out* para *FSBox*

B.5. Generación de código Haskell desde UUAGC

Una vez que los atributos y semánticas han sido descritos en un archivo con extensión 'ag', se utiliza la herramienta UUAGC para generar código Haskell.

UUAGC permite, entre otras cosas, generar los tipos de datos (opción `data`), las funciones semánticas (opción `semfuns`), los tipos de las funciones semánticas (opción `catas`), y la cabecera del módulo para el código generado (opción `module`).

Cada una de estas opciones puede generarse de manera independiente y en archivos diferentes o también todas las opciones en una sola y en un sólo archivo.

Por ejemplo, para generar los atributos y semánticas de este apéndice, se puede escribir lo siguiente (suponga que la descripción se encuentra en el archivo *fsbox.ag*):

```
uuagc --data --semfuns --catas --module fsbox.ag
```

Apéndice C

Documentación de la librería Map

En este apéndice se mostrará la documentación de la librería *Map* de Haskell. Sólo se mostrará las funciones que son relevantes al proyecto. La versión original, que se encuentra en el idioma Ingles, está en la siguiente dirección URL:
`http://haskell.org/ghc/docs/7.0-latest/html/libraries/containers-0.4.0.0/Data-Map.html`

Este apéndice sólo es una transcripción de la versión en Ingles, de las partes más relevantes de la documentación de la librería.

C.1. Descripción

Una implementación eficiente de maps de claves a valores (diccionarios).

Como que los nombres de las funciones (pero no del tipo) pueden entrar en conflicto con los nombres del Preludio de Haskell, este modulo es normalmente importado de manera renombrada (qualified), por ejemplo

```
import Data.Map (Map)
import qualified Data.Map as Map
```

La implementación de *Map* está basado en un árbol binario de tamaño balanceado (o árboles de balance limitado), que está descrito en:

Stephen Adams, "Efficient sets: a balancing act", Journal of Functional Programming 3(4):553-562, October 1993, http://www.swiss.ai.mit.edu/~adams/BB/.

J. Nievergelt and E.M. Reingold, "Binary search trees of bounded balance", SIAM journal of computing 2(1), March 1973.

Vea que la implementación es de preferencia-por-izquierda, es decir que los elementos de un primer argumento son preferidos ante el segundo, por ejemplo en *union* o *insert*. Los comentarios contienen el tiempo de complejidad en notación Big-O (`http://en.wikipedia.org/wiki/Big_O_notation`).

C.2. El tipo Map

data *Map k a*

Un *Map* de claves 'k' a valores 'a'.

C.3. Operadores

C.3.1. $(!) :: Ord\ k \Rightarrow Map\ k\ a \to k \to a$

$O\ (log\ n)$. Encontrar el valor de una clave. Se lanza un error cuando el elemento se encuentra.

$fromList\ [(5, \texttt{'a'}), (3, \texttt{'b'})]\ !\ 1\ Error : el\ elemento\ no\ esta\ en\ el\ map$
$fromList\ [(5, \texttt{'a'}), (3, \texttt{'b'})]\ !\ 5 \equiv \texttt{'a'}$

C.3.2. $(\backslash\backslash) :: Ord\ k \Rightarrow Map\ k\ a \to Map\ k\ b \to Map\ k\ a$

Lo mismo que la operación de diferencia.

C.4. Consulta

C.4.1. $null :: Map\ k\ a \to Bool$

$O\ (1)$. ¿Esta el map vacío?

$Data.Map.null\ (empty) \equiv True$
$Data.Map.null\ (singleton\ 1\ \texttt{'a'}) \equiv False$

C.4.2. $size :: Map\ k\ a \to Int$

$O\ (1)$. El número de elementos en el map.

$size\ empty \equiv 0$
$size\ (singleton\ 1\ \texttt{'a'}) \equiv 1$
$size\ (fromList\ ([(1, \texttt{'a'}), (2, \texttt{'c'}), (3, \texttt{'b'})])) \equiv 3$

C.4.3. *member* :: *Ord k* ⇒ *k* → *Map k a* → *Bool*

O (*log n*). ¿Es la clave miembro del map?, también vea *notMember*.

```
member 5 (fromList [(5, 'a'), (3, 'b')]) ≡ True
member 1 (fromList [(5, 'a'), (3, 'b')]) ≡ False
```

C.4.4. *notMember* :: *Ord k* ⇒ *k* → *Map k a* → *Bool*

O (*log n*). ¿No es clave miembro del map?, también vea *member*.

```
notMember 5 (fromList [(5, 'a'), (3, 'b')]) ≡ False
notMember 1 (fromList [(5, 'a'), (3, 'b')]) ≡ True
```

C.4.5. *lookup* :: *Ord k* ⇒ *k* → *Map k a* → *Maybe a*

O (*log n*). Buscar el valor de una clave en el map. La función retornará el correspondiente valor como (*Just value*), o *Nothing* si la clave no se encuentra en el map.

Un ejemplo del uso de *lookup*:

```
import Prelude hiding (lookup)
import Data.Map
employeeDept    = fromList ([("John", "Sales"), ("Bob", "IT")])
deptCountry     = fromList ([("IT", "USA"), ("Sales", "France")])
countryCurrency = fromList ([("USA", "Dollar"), ("France", "Euro")])
employeeCurrency :: String → Maybe String
employeeCurrency name
  = do dept ← lookup name employeeDept
       country ← lookup dept deptCountry
       lookup country countryCurrency
main = do putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
          putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

A continuación se muestra el resultado:

```
John's currency : Just "Euro"
Pete's currency : Nothing
```

C.4.6. *findWithDefault* :: *Ord k* ⇒ *a* → *k* → *Map k a* → *a*

O (*log n*). La expresión (*findWithDefault def k map*) retorna el valor de la clave 'k' o retorna el valor por defecto 'def' cuando la clave no está en el map.

findWithDefault `'x'` 1 (*fromList* [(5, `'a'`), (3, `'b'`)]) ≡ `'x'`
findWithDefault `'x'` 5 (*fromList* [(5, `'a'`), (3, `'b'`)]) ≡ `'a'`

C.5. Construcción

C.5.1. *empty* :: *Map k a*

O (1). El map vacío.

empty ≡ *fromList* [ ]
size empty ≡ 0

C.5.2. *singleton* :: *k* → *a* → *Map k a*

O (1). Un map con sólo un elemento.

singleton 1 `'a'` ≡ *fromList* [(1, `'a'`)]
size (*singleton* 1 `'a'`) ≡ 1

C.5.3. Insertar

insert :: *Ord k* ⇒ *k* → *a* → *Map k a* → *Map k a*

O (*log n*). Insertar una nueva clave y valor en el map. Si la clave está presente en el map, el valor asociado es reemplazado con el valor recibido. *insert* es equivalente a *insertWith const*.

insert 5 `'x'` (*fromList* [(5, `'a'`), (3, `'b'`)]) ≡ *fromList* [(3, `'b'`), (5, `'x'`)]
insert 7 `'x'` (*fromList* [(5, `'a'`), (3, `'b'`)]) ≡ *fromList* [(3, `'b'`), (5, `'a'`), (7, `'x'`)]
insert 5 `'x'` *empty* ≡ *singleton* 5 `'x'`

insertWith :: *Ord k* ⇒ (*a* → *a* → *a*) → *k* → *a* → *Map k a* → *Map k a*

O (*log n*). Insertar con una función, que combina el nuevo y antiguo valor.
insertWith f key value mp insertará la tupla (*key*, *value*) en 'mp' si la clave 'key' no existe en el map. Si la clave existe, la función insertará la tupla (*key*, *f new_value old_value*).

$insertWith\ (+\!\!+)\ 5\ \texttt{"xxx"}\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"xxxa"})]$
$insertWith\ (+\!\!+)\ 7\ \texttt{"xxx"}\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"}), (7, \texttt{"xxx"})]$
$insertWith\ (+\!\!+)\ 5\ \texttt{"xxx"}\ empty \equiv singleton\ 5\ \texttt{"xxx"}$

$insertWithKey :: Ord\ k \Rightarrow (k \to a \to a \to a) \to k \to a \to Map\ k\ a \to Map\ k\ a$

$O\ (log\ n)$. Insertar con una función, se combina la clave, nuevo valor y antiguo valor.

$insertWithKey\ f\ key\ value\ mp$ insertará la tupla $(key, value)$ en el 'mp' if la clave no se encuentra en el map. Si la clave existe, la función insertará la tupla $(key, f\ key\ new\_value\ old\_value)$. Vea que la clave pasada a 'f' es la misma clave pasada a $insertWithKey$.

let $f\ key\ new\_value\ old\_value = (show\ key) +\!\!+ \texttt{":"} +\!\!+ new\_value +\!\!+ \texttt{"|"} +\!\!+ old\_value$
$insertWithKey\ f\ 5\ \texttt{"xxx"}\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})])$
$\quad \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"5:xxx|a"})]$
$insertWithKey\ f\ 7\ \texttt{"xxx"}\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})])$
$\quad \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"}), (7, \texttt{"xxx"})]$
$insertWithKey\ f\ 5\ \texttt{"xxx"}\ empty$
$\quad \equiv singleton\ 5\ \texttt{"xxx"}$

C.5.4. Eliminar/Actualizar

$delete :: Ord\ k \Rightarrow k \to Map\ k\ a \to Map\ k\ a$

$O\ (log\ n)$. Elimina una clave y su valor del map. Cuando la clave no es un miembro del map, el map original es retornado.

$delete\ 5\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv singleton\ 3\ \texttt{"b"}$
$delete\ 7\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"})]$
$delete\ 5\ empty \equiv empty$

$adjust :: Ord\ k \Rightarrow (a \to a) \to k \to Map\ k\ a \to Map\ k\ a$

$O\ (log\ n)$. Actualizar el valor de una clave específica con el resultado de la función proveída. Cuando la clave no es miembro del map, el map original es devuelto.

$adjust\ (\texttt{"new "} +\!\!+)\ 5\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"new a"})]$
$adjust\ (\texttt{"new "} +\!\!+)\ 7\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"})]$
$adjust\ (\texttt{"new "} +\!\!+)\ 7\ empty \equiv empty$

adjustWithKey :: *Ord k* ⇒ (*k* → *a* → *a*) → *k* → *Map k a* → *Map k a*

O (*log n*). Ajustar el valor de una clave específica. Cuando la clave no es un miembro del map, el map original es devuelto.

```
let f key x = (show key) ++ ":new " ++ x
adjustWithKey f 5 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "5:new a")]
adjustWithKey f 7 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "a")]
adjustWithKey f 7 empty ≡ empty
```

update :: *Ord k* ⇒ (*a* → *Maybe a*) → *k* → *Map k a* → *Map k a*

O (*log n*). La expresíon (*update f k map*) actualiza el valor de 'x' en 'k' (si esta en el map). If (*f x*) es *Nothing*, el elemento es eliminado. Si es (*Just y*), la clave 'k' es cambiada al nuevo valor 'y'.

```
let f x = if x ≡ "a" then Just "new a" else Nothing
update f 5 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "new a")]
update f 7 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "a")]
update f 3 (fromList [(5, "a"), (3, "b")]) ≡ singleton 5 "a"
```

updateWithKey :: *Ord k* ⇒ (*k* → *a* → *Maybe a*) → *k* → *Map k a* → *Map k a*

O (*log n*). La expresión (*updateWithKey f k map*) actualiza el valor 'x' en 'k' (si esta en el map). Si (*f k x*) es *Nothing*, el elemento es eliminado. Si es (*Just y*), la clave 'k' es cambiada al nuevo valor 'y'.

```
let f k x = if x ≡ "a" then Just ((show k) ++ ":new a") else Nothing
updateWithKey f 5 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "5:new a")]
updateWithKey f 7 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "a")]
updateWithKey f 3 (fromList [(5, "a"), (3, "b")]) ≡ singleton 5 "a"
```

alter :: *Ord k* ⇒ (*Maybe a* → *Maybe a*) → *k* → *Map k a* → *Map k a*

O (*log n*). La expresión (*alter f k map*) altera el valor de 'x' en 'k', o su ausencia. *alter* puede ser usado para insertar, eliminar, o actualizar un valor en el *Map*. En palabras simples: *lookup k* (*alter f k m*) = *f* (*lookup k m*)

let $f\ \_ = Nothing$
$alter\ f\ 7\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"})]$
$alter\ f\ 5\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv singleton\ 3\ \texttt{"b"}$
let $f\ \_ = Just\ \texttt{"c"}$
$alter\ f\ 7\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"}), (7, \texttt{"c"})]$
$alter\ f\ 5\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"c"})]$

C.6. Combine

C.6.1. Unión

$union :: Ord\ k \Rightarrow Map\ k\ a \to Map\ k\ a \to Map\ k\ a$

$O\ (n + m)$. La expresión ($union\ t1\ t2$) toma la unión de preferencia-por-izquierda de t1 y t2. Prefiere t1 cuando se encuentran duplicados, por ejemplo, ($union \equiv unoinWith\ const$). La implementación usa el algoritmo *hedge-union*. *hedge-union* es más eficiente en (*bigset* ‘*union*‘ *smallset*).

$union\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})])\ (fromList\ [(5, \texttt{"A"}), (7, \texttt{"C"})])$
$\quad \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"a"}), (7, \texttt{"C"})]$

$unionWith :: Ord\ k \Rightarrow (a \to a \to a) \to Map\ k\ a \to Map\ k\ a \to Map\ k\ a$

$O\ (n+m)$. Unión con una función de combinación. La implementación utiliza el algoritmo *hedge-union*.

$unionWith\ (+\!\!+)\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})])\ (fromList\ [(5, \texttt{"A"}), (7, \texttt{"C"})])$
$\quad \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"aA"}), (7, \texttt{"C"})]$

$unionWithKey :: Ord\ k \Rightarrow (k \to a \to a \to a) \to Map\ k\ a \to Map\ k\ a \to Map\ k\ a$

$O\ (n + m)$. Unión con una función de combinación.

let $f\ key\ left\_value\ right\_value$
$\quad = (show\ key) +\!\!+ \texttt{":"} +\!\!+ left\_value +\!\!+ \texttt{"|"} +\!\!+ right\_value$
$unionWithKey\ f\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})])\ (fromList\ [(5, \texttt{"A"}), (7, \texttt{"C"})])$
$\quad \equiv fromList\ [(3, \texttt{"b"}), (5, \texttt{"5:a|A"}), (7, \texttt{"C"})]$

C.7. Recorrido

C.7.1. Map

$map :: (a \to b) \to Map\ k\ a \to Map\ k\ b$

$O\ (n)$. Mapear una función sobre todos los valores de map.

$map\ (+\!\!+ \texttt{"x"})\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"bx"}), (5, \texttt{"ax"})]$

$mapWithKey :: (k \to a \to b) \to Map\ k\ a \to Map\ k\ b$

$O\ (n)$. Mapear una función sobre todos los valores en el map.

let $f\ key\ x = (show\ key) +\!\!+ \texttt{":"} +\!\!+ x$
$mapWithKey\ f\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv fromList\ [(3, \texttt{"3:b"}), (5, \texttt{"5:a"})]$

C.7.2. Fold

$fold :: (a \to b \to b) \to b \to Map\ k\ a \to b$

$O\ (n)$. *Fold* los valores en el map, de manera que $fold\ f\ z \equiv foldr\ f\ z.elems$. Por ejemplo,

$elems\ map = fold\ (:)\ [\,]\ map$
let $f\ a\ len = len + (length\ a)$
$fold\ f\ 0\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"bbb"})]) \equiv 4$

$foldWithKey :: (k \to a \to b \to b) \to b \to Map\ k\ a \to b$

$O\ (n)$. *Fold* las claves y valores en el map, de manera que
$foldWithKey\ f\ z \equiv foldr\ (uncurry\ f)\ z.toAscList$.
Por ejemplo,

$keys\ map = foldWithKey\ (\lambda k\ x\ ks \to k : ks)\ [\,]\ map$
let $f\ k\ a\ result = result +\!\!+ \texttt{"("} +\!\!+ (show\ k) +\!\!+ \texttt{":"} +\!\!+ a +\!\!+ \texttt{")"}$
$foldWithKey\ f\ \texttt{"Map: "}\ (fromList\ [(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv \texttt{"Map: (5:a)(3:b)"}$

Esto es idéntico a *foldrWithKey*, y usted debería usar aquella en vez de esta. Su nombre es guardado sólo por compatibilidad.

$foldrWithKey :: (k \to a \to b \to b) \to b \to Map\ k\ a \to b$

$O\ (n)$. Post-order fold. La función es aplicada desde el valor más pequeño al más grande.

foldlWithKey :: $(b \to k \to a \to b) \to b \to Map\ k\ a \to b$

O(n). Pre-order fold. La función será aplicada desde el valor más pequeño al más grande.

C.8. Conversión

C.8.1. *elems* :: $Map\ k\ a \to [a]$

$O\ (n)$. Retornar todos los elementos del map en orden ascendente de sus claves.

elems (*fromList* $[(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv [\texttt{"b"}, \texttt{"a"}]$
elems empty $\equiv [\,]$

C.8.2. *keys* :: $Map\ k\ a \to [k]$

$O\ (n)$. Retornar todas las claves del map en orden ascendente.

keys (*fromList* $[(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv [3, 5]$
keys empty $\equiv [\,]$

C.8.3. Listas

toList :: $Map\ k\ a \to [(k, a)]$

$O\ (n)$. Convertir a una lista de tupla de clave/valor.

toList (*fromList* $[(5, \texttt{"a"}), (3, \texttt{"b"})]) \equiv [(3, \texttt{"b"}), (5, \texttt{"a"})]$
toList empty $\equiv [\,]$

fromList :: $Ord\ k \Rightarrow [(k, a)] \to Map\ k\ a$

$O\ (n * log\ \ n)$. Construir un map desde una lista de tuplas clave/valor. Vea también *fromAscList*. Si la lista contiene más de un valor para la misma clave, el último valor para la clave es retenido.

fromList $[\,] \equiv$ *empty*
fromList $[(5, \texttt{"a"}), (3, \texttt{"b"}), (5, \texttt{"c"})] \equiv$ *fromList* $[(5, \texttt{"c"}), (3, \texttt{"b"})]$
fromList $[(5, \texttt{"c"}), (3, \texttt{"b"}), (5, \texttt{"a"})] \equiv$ *fromList* $[(5, \texttt{"a"}), (3, \texttt{"b"})]$

C.9. Filtro

C.9.1. *filter* :: *Ord k* ⇒ (*a* → *Bool*) → *Map k a* → *Map k a*

O (*n*). Filtrar todos los valores que satisfagan el predicado.

filter (>"a") (*fromList* [(5, "a"), (3, "b")]) ≡ *singleton* 3 "b"
filter (>"x") (*fromList* [(5, "a"), (3, "b")]) ≡ *empty*
filter (<"a") (*fromList* [(5, "a"), (3, "b")]) ≡ *empty*

C.9.2. *filterWithKey* :: *Ord k* ⇒ (*k* → *a* → *Bool*) → *Map k a* → *Map k a*

O (*n*). Filtrar todos los valores que satisfagan el predicado.

filterWithKey (λ*k* _ → *k* > 4) (*fromList* [(5, "a"), (3, "b")]) ≡ *singleton* 5 "a"

C.9.3. *mapMaybe* :: *Ord k* ⇒ (*a* → *Maybe b*) → *Map k a* → *Map k b*

O (*n*). Mapear los valores y coleccionar los resultado *Just*.

let *f x* = **if** *x* ≡ "a" **then** *Just* "new a" **else** *Nothing*
mapMaybe f (*fromList* [(5, "a"), (3, "b")]) ≡ *singleton* 5 "new a"

C.9.4. *mapMaybeWithKey* :: *Ord k* ⇒ (*k* → *a* → *Maybe b*) → *Map k a* → *Map k b*

O (*n*). Mapear clave/valor y coleccionar los resultado *Just*.

let *f k* _ = **if** *k* < 5 **then** *Just* ("key : " ++ (*show k*)) **else** *Nothing*
mapMaybeWithKey f (*fromList* [(5, "a"), (3, "b")]) ≡ *singleton* 3 "key : 3"

C.10. Índice

C.10.1. *elemAt* :: *Int* → *Map k a* → (*k*, *a*)

O (*log n*). Recuperar un elemento por el índice. Se lanza un error cuando el índice no es válido.

elemAt 0 (*fromList* [(5, "a"), (3, "b")]) ≡ (3, "b")
elemAt 1 (*fromList* [(5, "a"), (3, "b")]) ≡ (5, "a")
elemAt 2 (*fromList* [(5, "a"), (3, "b")]) *Error* : *index out* **of** *range*

C.10.2. $updateAt :: (k \to a \to Maybe\ a) \to Int \to Map\ k\ a \to Map\ k\ a$

$O\ (log\ n)$. Actualizar el elemento del índice. Se lanza una error cuando el índice no es válido.

```
updateAt (λ_ _ → Just "x") 0 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "x"), (5, "a")]
updateAt (λ_ _ → Just "x") 1 (fromList [(5, "a"), (3, "b")]) ≡ fromList [(3, "b"), (5, "x")]
updateAt (λ_ _ → Just "x") 2 (fromList [(5, "a"), (3, "b")])      Error : index out of range
updateAt (λ_ _ → Just "x") (−1) (fromList [(5, "a"), (3, "b")]) Error : index out of range
updateAt (\__ → Nothing) 0    (fromList [(5, "a"), (3, "b")]) ≡ singleton 5 "a"
updateAt (\__ → Nothing) 1    (fromList [(5, "a"), (3, "b")]) ≡ singleton 3 "b"
updateAt (\__ → Nothing) 2    (fromList [(5, "a"), (3, "b")])      Error : index out of range
updateAt (\__ → Nothing) (−1) (fromList [(5, "a"), (3, "b")])    Error : index out of range
```

C.10.3. $deleteAt :: Int \to Map\ k\ a \to Map\ k\ a$

$O\ (log\ n)$. Eliminar el elemento del índice. Definido como $(deleteAt\ i\ map = updateAt\ (k\ x \to Nothing)\ i\ map)$.

```
deleteAt 0 (fromList [(5, "a"), (3, "b")]) ≡ singleton 5 "a"
deleteAt 1 (fromList [(5, "a"), (3, "b")]) ≡ singleton 3 "b"
deleteAt 2 (fromList [(5, "a"), (3, "b")])      Error : index out of range
deleteAt (−1) (fromList [(5, "a"), (3, "b")]) Error : index out of range
```

Apéndice D

Hoja de Estilo para UserAgent

En este apéndice encontraras la hoja de estilo para el usuario *UserAgent*, que es utilizada por el Navegador Web como la hoja de estilo por defecto para renderizar un documento.

D.1. Hoja de Estilo

```
html, address, blockquote, body, dd, div,
dl, dt, fieldset, form, frame, frameset,
h1, h2, h3, h4, h5, h6, noframes,
ol, p, ul, center, dir, hr, menu, pre { display: block}

li { display: list-item }

head { display: none }

body { margin: 8px }

h1 { font-size: 2em
   ; margin: .67em 0px
   }

h2 { font-size: 1.5em
   ; margin: .75em 0px
   }

h3 { font-size: 1.17em
   ; margin: .83em 0px
   }

h4, p, blockquote, ul, fieldset, form,ol, dl, dir,menu { margin: 1.12em 0px }

h5 { font-size: .83em
   ; margin: 1.5em 0px
   }

h6 { font-size: .75em
   ; margin: 1.67em 0px
   }
```

```
h1, h2, h3, h4, h5, h6, b, strong {font-weight: bold }

blockquote { margin-left: 40px
           ; margin-right: 40px
           }

i, cite, em, var, address {font-style: italic }

pre, tt, code, kbd, samp {font-family: monospace }

pre {white-space: pre }

big {font-size: 1.17em }

small, sub, sup {font-size: .83em }

sub {vertical-align: sub }
sup {vertical-align: super }

s, strike, del {text-decoration: line-through }

ol, ul, dir,menu, dd {margin-left: 40px }

ol {list-style-type: decimal }

ol ul, ul ol,ul ul, ol ol { margin-top: 0px
                          ; margin-bottom: 0px
                          }

u, ins {text-decoration: underline }

br:before { content: "\A"
          ; white-space: pre-line
          }

center {text-align: center }

a {text-decoration: underline }
```

Referencias

Base de datos de aplicaciones y librerias de haskell. (s.f.). Disponible en `http://www.haskell.org/hackage`

Bird, R. (2000). *Introduccion a la Programacion Funcional con Haskell* (2da ed.; P. Hall, Ed.).

Brian Totty, D. (s.f.). *Http, The definitive Guide* (Orelly, Ed.).

Bringert, B. (s.f.). *Gd library.* Disponible en `http://hackage.haskell.org/packages/gd/` (Version 3000.6)

Diatchki, I. S. (s.f.). *Url library.* Disponible en `http://www.haskell.org/haskellwiki/Url` (Version 2.1.2)

Doaitse Swierstra, A. A. J. B. A. P. (2004, November). *Implementation of programming languages.* Lecture Notes.

Fairley, R. E. (1987). *Ingenieria de software* (McGRAW-HILL, Ed.).

Finne, S. (s.f.). *Libcurl.* Disponible en `http://hackage.haskell.org/packages/curl/` (Version 1.3.6)

Haskell community and activities report. (s.f.). Disponible en `http://haskell.org/hcar`

Jeuring, J., y Swierstra, S. D. (2000). *Gramaticas y analisis sintactico.* (Texto base del curso de Automatas en Lic. Informatica, UMSS)

Leijen, D. (2004). *wxhaskell, a portable and concise gui library for haskell.* Disponible en `http://hackage.haskell.org/package/wx/` (Version 0.12.1.6)

Meyer, E. (2004). *Cascading style sheets: The definitive guide* (2da ed.; O'Relly, Ed.).

Mitchell, N. (s.f.). *Tagsoup library.* Disponible en `http://community.haskell.org/~ndm/tagsoup/` (Version 0.12)

Ohlendorf, M. (2007). *A Cookbook for the Haskell XML Toolbox with Examples for Processing RDF Documents.*

Peyton Jones, S. (s.f.). *Introduction to Haskell.* Disponible en `http://www.haskell.org/haskellwiki/Introduction`

Peyton Jones, S. (2002, September). *Haskell 98, Language and Libraries* (Inf. Téc.). Haskell Community. Disponible en `http://haskell.org`

Sebesta, R. W. (2006). *Programming the World Wide Web* (3th ed.; A. Wesley, Ed.).

Swierstra, S. D. (s.f.-a). *Utrecht University Attribute Grammar Compiler, Software informático uuagc.* Disponible en `http://hackage.haskell.org/package/uuagc/` (Version 0.9.29)

Swierstra, S. D. (s.f.-b). *Utrecht university parser combinator library.* Disponible en `http://www.cs.uu.nl/wiki/bin/view/HUT/ParserCombinators` (Version 2.5.5)

Swierstra, S. D. (2008, December). *Combinator parsing: A short tutorial* (Inf. Téc. n.º UU-CS-2008-044). Institute of Information and Computing Sciences, Utrecht University. Disponible en `www.cs.uu.nl`

Thompson, S. (1999). *Haskell, The Craft of Funcional Programming* (2da ed.; Addison-Wesley, Ed.).

W3C. (1999). *HTML 4.01, Specification.* Disponible en `http://www.w3.org/TR/1999/REC-html401-19991224`

W3C. (2009, September). *Cascading style sheets level 2 revision 1 (css 2.1) specification.* Disponible en `http://www.w3.org/TR/2010/WD-CSS2-20101207`