Trabajo Práctico Final de ALP

Mauricio Salichs

1 Descripción

Se describe un lenguaje markup creado para renderizar documentos en PDF con algunas características simples. La aplicación primero convierte este lenguaje a otro lenguaje markup (HTML5, sin complementar con CSS) y se usa una librería de Haskell para convertir esta representación de HTML a un documento PDF. Adicionalmente, al llamar al programa, se pueden definir algunos parámetros del documento para que sean aplicados a la hora de renderizar.

El repositorio del código se puede encontrar en:
https://github.com/mauriciosalichs/proyecto-alp

2 El Lenguaje

El lenguaje creado consta de reglas muy simples para mostrar en el PDF final. Nuevas reglas son fácilmente agregables al lenguaje. A la hora de describir el documento con el lenguaje markup, tenemos tres partes diferenciables, cada una iniciando con un comando particular:

- Una parte donde se definen los estilos que vamos a usar. Utilizamos el comando /estilos para definir todos los diferentes estilos que luego queramos usar en el documento.
- Una parte donde se define el título del documento. Se utiliza el comando /titulo para definirlo.
- La parte final es la descripción de las secciones del documento. Se utiliza el comando /secciones para definir esta parte.

2.1 Estilos

Para definir cada nuevo estilo, escribimos su nombre ente <> y luego le damos los parámetros. Hay cuatro parámetros que podemos definir: tamaño, color, fuente y alineación. La manera es escribiendo los parámetros deseados uno abajo del otro y su definición al lado. Una opción adicional es heredar los parámetros de alguna otra fuente y luego cambiar algun parámetro adicional. Este es un pequeño ejemplo de tres estilos:

```
<Arial>
tamaño 14
color negro
fuente arial
alineacion izquierda

<ArialGrande>
hereda Arial
tamaño 20

<Centrado>
alineacion centrado
```

Nótese que no hace falta agregar todos los parámetros para poder definir un estilo.

2.2 Título

El título se escribe directamente acompañado al lado de su comando. Si se le quiere dar un estilo especial, debe precederse el nombre del estilo entre <>. Dos ejemplos de títulos son:

```
\titulo Resúmen del año
\titulo <Centrado> Resumen del año
```

Si se quiere que el título ocupe solo la página inicial, se setea con la llamada al programa.

2.3 Secciones

Esta es la parte principal del documento, donde se describen las secciones o capítulos del mismo. Para definir secciones y subsecciones se utliza MOÑITO.

Para una sección simple, se usa uno solo acompañado de su título. Para agregar niveles de subsección se escribe uno por cada nivel, seguido del título. Si se le quiere dar un estilo especial, debe precederse el nombre del estilo entre <>.
Ejemplo:

```
~ Primer capítulo.
    ...
~ ~ Primera subsección del cap.
    ...
~ ~ <Arial> Segunda subsección del cap.
    ...
~ Segundo capítulo.
    ...
```

2.4 Cuerpo de las secciones

Hay cuatro tipo de bloques que podemos tener dentro del cuerpo de una sección:

- Párrafos simples. Se escriben directamente sin ningún agregado.
- Items. El comando para crearlo es 'items' y y en cada linea siguiente se escriben los items antecedidos por : (dos puntos).
- Imágenes. El comando para insertar una imágen es 'imagen' y luego se escribe la ruta de la imágen, tomando como referencia la ubicación donde esta el archivo fuente.
- Tablas. Su comando es 'tabla' seguido de una linea para cada fila, antecedida cada una por : (dos puntos). En cada linea se escriben las celdas de cada fila separados por ; (punto y coma).

Todos los diferentes bloques se pueden estilizar, escribiendo despues del comando correspondiente el nombre del estilo entre <>.

En todo texto que haya en cualquier parte del documento, tenemos tres atributos para darles:

- Texto en negrita. Se consigue encerrando el texto entre astériscos ($*$).
- Texto en cursiva. Se consigue encerrando el texto entre guiones bajos (_).
- Links. Para que un bloque de texto lleve a una URL, encerramos este texo entre corchetes. La dirección de la URL es lo primero que se escribe entre los corchetes.

¿Mostrar ejemplos de todo esto?

3 Estructura de los módulos del código

Tenemos cinco módulos fuente y uno principal. Los módulos son:

- Lang. En este módulo definimos la estructura interna de los tipos que vamos a usar. Por un lado tenemos los estilos (Style) junto con sus dos versiones de diccionarios donde se mapean los nombres de los estilos con sus definiciones. Por el otro lado tenemos el AST del documento en sí.
- Parse. En este módulo se define el parser que toma como entrada el archivo fuente y devuelve un diccionario de estilos preprocesado y un elemento de tipo Document, donde se guarda toda la estructura interna del documento.
- Styles. En este módulo tenemos las funciones necesarias para convertir el diccionario preprocesado en uno ya procesado, con los estilos bien definidos y completos.

- BuildIntHTML. La librería que convierte nuestra representación interna del lenguaje markup en una representación interna de HTML5, de la librería Text.Blaze.
- Lib. Una librería auxiliar con algunas funciones útiles para el módulo Main.
- Main. Nuestro módulo principal, que toma los argumentos de archivo fuente y (opcionalmente) archivo salida, y con la posibilidad de setear ciertas características de la renderización vía línea de comandos, y va haciendo un procesamiento por las distintas etapas hasta generar el archivo PDF final.

4 Versiones del proyecto

Hago un breve repaso de las diferentes versiones de la aplicación y que cambios se fueron decidiendo hacer para optimizar o hacer un poco mas elegante la estructura del código. Todas estas versiones estan etiquetadas en el Log del repositorio en Github.

4.1 Conversor básico a HTML

En la primer versión funcional de esta aplicación, tenemos un conversor básico que parsea el archivo fuente (con algunos errores al momento de parsear subsecciones), y un conversorde nuestro AST interno a String, creando las cadenas de texto del html manualmente. Esta versión ignora los estilos, solo se completan en una tabla preprocesada. El aspecto mas importante de esta versión y las siguientes es que aún no se renderiza a PDF, el producto final es un archivo con código HTML.

4.2 Agregado de estilos al documento

El primer gran arreglo para esta versión fue modificar un poco el AST interno del documento y las secciones, para no abusar de las tuplas, y emprolijar un poco mas usando records. Después, se fue modificando la manera de estructurar los estilos y su diccionario (en principio había solo una versión de este).

Luego se decidió hacer dos versiones del diccionario. Una que parseara tal cual lo que aparecía en el archivo fuente, con estilos incompletos en algunos parámetros y la noción de herencia. En la versión procesada del diccionario, se elimina la noción de herencia y se le completan todos los parámetros a cada estilo, dejándolos normalizados.

Otro arreglo fue en el parser, el cual ahora retorna el error correspondiente a un archivo fuente mal escrito o con algun simbolo faltante para el correcto parseado. También se eliminó la información del estilo por defecto de este archivo, ya que

es un concepto interno de Estilos y este no era el módulo correspondiente.

Finalmente se agregaron los estilos al archivo HTML generado.

4.3 Utilizando una representación abstracta de HTML

Para esta versión, el primer cambio fue utilizar la mónada Reader para llevar en un entorno común, el diccionario de estilos, en vez de arrastrarlo en todas las funciones como un argumento más. No hace falta usar la mónada State para esto, ya que el diccionario es estático. Es decir, no muta durante su uso en la fabricación del código HTML.

Como segundo cambio, se agregó la librería Options.Applicative, basada en funtores aplicativos, que además de parsear distintos tipos de opciones al correr el programa en la línea de comandos, analiza este parser para generar automáticamente una ayuda que se muestra al correr el programa con la opción –help (o al escribir mal una opción).

Y el gran cambio que se hizo para esta versión fue dejar de construir el código HTML a mano, y pasar a usar una representación interna de HTML definido en alguna librería de Haskell, para asi no tener que lidiar con la representación interna y poder abstraerse a la representación que nos ofrece la librería. Esto permite, entre otras cosas, poder agregar nuevas features de nuesto lenguaje markup con mas facilidad. La librería usada para esta representación es Text.Html.

4.4 Renderización a PDF

El problema principal de usar esta librería para la representación de HTML es que no es fácil de renderizar al formato PDF, ya que no es una librería muy común. Por eso el gran cambio acá es reemplazar esta librería por Text.Blaze.Html. Este nuevo tipo es fácilmente renderizable a PDF mediante otra librería comunmente usada (Yesod.Content.PDF).

Este enfoque tuvo la dificultad de necesitar cambiar la mónada Reader por el transformador de mónada ReaderT, ya que internamente el tipo Html de Blaze es una mónada aplicada a Unit y las estructuras del lenguaje se definen con notación do. Llevando esto a usar una mónada sobre otra.

Con esta mónada finalmente podemos completar la aplicación, transformando la representación abstracta de Html a Pdf y luego escribiendo esto en un archivo.

4.5 Ajustes finales y pasado de parámetros

Para concluir la aplicación, el último gran cambio es pasar ciertas características de la renderización, como por ejemplo el tamaño de página, a través de la llamada al programa. Esto también llevo a mover el parseado de estos parámetros

a un módulo externo (Lib) donde también haya manejos de errores.

También se incluyeron advertencias para casos de error en las definiciones de estilos, un gran faltante de la aplicación. En principio se consideró usar alguna librería de manejo de excepciones, como por ejemplo Control.Monad.Except. Pero dado que la mayoría solo necesitan ser mensajes de advertencia que no modifiquen el flujo del programa, se optó por usar simplemente IO para notificar los malos usos. El único caso donde se necesita quizás el uso de excepciones es en la función *getStyle*, la cual es usada por BuildIntHTML y aún no cuenta con un mensaje de notificación si el usuario intenta usar un estilo inexistente.

5 Manejo del proyecto y uso de la aplicación

Se usó el programa Stack para desarrollar el proyecto, ya que resultó muy cómodo para crear su estructura automáticamente y su facilidad para agregar librerías. Para correr la aplicación, se hace la siguiente llamada:

```
stack run -- [OPCIONES] archivo_fuente [archivo salida]
```

Las opciones disponibles son:

```
-p,--paisaje              Orienta la página en formato paisaje.
-t,--titulo               Para que la primer página sea solo el título.
-f,--formato ARG          Formato de la página. Opciones posibles: "a4",
                            "carta" o configurar manualmente con el formato
                            'largo ancho' (en cm).
--margenv ARG             Margenes verticales (en cm).
--margenh ARG             Margenes horizontales (en cm).
```

Las librerías usadas para poder desarrollar la aplicación fueron las siguientes:

- parsec: Librería para escribir el parser del archivo fuente.
- blaze-html: Para la representación HTML de Blaze.
- blaze-markup: Acceso a la representación interna de Blaze, para poder combinarla con la mónada Reader.
- yesod-content-pdf: Para poder generar un pdf a partir de la representación HTML de Blaze.
- optparse-applicative: Usada para parsear opciones desde la línea de comando.
- bytestring: Usamos su función *writeFile* para poder escribir a un archivo a partir del pdf generado.
- mtl: Necesaria para el uso de la monada Reader.
- directory: Para obtener la ruta de la aplicación y poder generar la correcta ruta de las imágenes.

6 Conclusión y posibles extensiones

En mi experiencia, la parte más difícil del trabajo fue la elección del tema. Realmente me costó encontrar uno que se sintiera realmente útil, distinto del resto y que se adecuara a las necesidades de la materia. Finalmente se optó por hacer algo común, no tan innovador pero que aprovechará claramente conceptos de la materia que se habían visto, como por ejemplo un Parser.

Después de haber realizado el trabajo hasta este punto, noté que era un trabajo díficil para sacar provecho a los conceptos de la materia. Actualmente lo considero en una étapa bastante simple, y que resulta difícil encontrarle nuevas features como para complejizarlo lo suficiente y que no se haga demasiado díficil de continuar.

El trabajo se considera terminado con las características presentadas, pero dejo a continuación algunas mejoras que se le podrían agregar:

- Agregado de carácteres especiales del idioma español. Tanto para poder escribir correctamente el parámetro "tamaño" de los estilos (actualmente sólo se puede escribir "tamano"), como para el uso de cualquier caracter latino, hubo problemas tanto en el parser como en la renderización.
- Bucles en la sintaxis del lenguaje. En un principio estaba previsto desarrollar el uso de bucles para algunas características en los documentos, para escribir de manera acortada, por ejemplo, tablas con datos similares o estructuras que siguen algun patrón. Esta tarea fue descartada porque en su versión simple no aportaba mucho a la expresividad del lenguaje, y en su manera mas compleja se volvía una tarea demasiado extensa de realizar.
- Agregado de más tipos de elementos HTML y agrandar la sintaxis del lenguaje. Extender los comandos del lenguaje con más características de HTML que fueran renderizables a PDF, como por ejemplo listas numeradas. Esto no significaba muchos cambios complicados, mas que nada extender el AST interno, el parser y el conversor. Pero no aportaba algo académicamente, era mas de lo mismo.
- Mayor manejo de excepciones a la hora de encontrar errores en el archivo fuente, con mensajes más específicos para el usuario. Este cambio si hubiera aportado algo más a la aplicación y académicamente, y podría ser una buena mejora si este proyecto se continuara.