

Simple PDF

Martín Goñi

2/2/2026

¿Que es SPF?

Simple PDF, es un DSL para generar PDFs, es decir un lenguaje que al ser interpretado produce PDFs, este informe está hecho con SPF. Decidí hacer SPF porque me interesaba hacer un proyecto que hiciese algo más que dar salida a la consola. Considerando que desde que conocí LaTeX me interesó el typesetting este proyecto era la excusa ideal para interiorizarme más en el tema.

Consideraciones de diseño

A la hora de diseñar SPF me inspire en LaTeX, pero sin mantenerme atado a sus convenciones o decisiones de diseño, lo trate como una fuente de inspiración. En particular toma otra dirección en cuanto a la forma de los comandos, permitiendo solo un argumento. Junto con esto las opciones deben tener asociada una clave indicando a qué campo corresponde el valor, hice esto porque la forma de escribir opciones en LaTeX es confusa; Se pueden combinar opciones con y sin clave, algo que en mi opinión es confuso y se presta a errores. Finalmente simplifique la forma en la que estructura el documento, en LaTeX esa complejidad permite

Los comandos implementados son un subconjunto de los disponibles en LaTeX, elegidos para permitir generar documentos (mayormente)completos, manteniendo al mismo tiempo la cantidad total de comandos baja. Cabe notar que los nombres y argumentos de los comandos no se copiaron exactamente, mi objetivo era que este subconjunto tenga una *funcionalidad* similar a la de sus contrapartes en LaTeX.

Sintaxis

Para representar la sintaxis se usan ciertas convenciones; El símbolo * indica cero o más ocurrencias de lo que encierra mientras que el + indica una o mas ocurrencias. Los comentarios no están incluidos pues no son parte de la sintaxis en si, son añadidos para la conveniencia del usuario y no resultan en tokens al parsear. A continuación se presenta la sintaxis concreta de **SPF**:

```
digit      ::= '0' | '1' | ... | '9'
integer    ::= <digit> (<digit>)*
float      ::= <integer> '.' <integer>
letter     ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
letterStr   ::= <letter> (<letter>)*
specialChars ::= '\' | '{', '}' | '[' | ']' | '"' | '/' | '|'
nonSpecialChars ::= c / c != <specialChars>
filepath    ::= (<letterStr> | '/' | '\' | '-' | '_' | '.' | '+')+

#| Opciones
optionList   ::= '[' optionListElems ']'
optionListElems ::= <optionPair> (',' <optionPair>)*
optionPair    ::= <letterStr> ':' <optionValue>
optionValue   ::= <boolValue>
              | <numberValue>
              | <literalValue>
              | <identifierValue>

boolValue    ::= 'true' | 'false'
numberValue   ::= <integer> | <float>
literalValue  ::= '"' <nonSpecialChars> '"'
identifierValue ::= <letterStr>

#| Texto
text        ::= ('\bold' | '\italic' | '\emph') '{}' <textChars> '}'
textChars   ::= (<nonSpecialChars>)+ <textChars> | '\' <specialChars> <textChars> |
<empty>

#| Configuración del documento
config       ::= '\config{`<configName> '}' <optionList>
```

```

configName ::= 'size' | 'pagenumbering' | ... | 'verbnumbering'

#| Metadata del documento
metadata ::= <title> <author> <date>
title    ::= '\title{' <text> '}' | <empty>
author   ::= '\author{' <text> '}' | <empty>
date     ::= '\date{' <text> '}' | <empty>

#| Comandos
command ::= <text>
| <paragraph>
| <section>
| <subsection>
| <figure>
| <table>
| <list>
| <verb>
| <newpage>
| <hline>

paragraph      ::= '\begin{paragraph}' <optionList> <text> '\end{paragraph}'
section        ::= '\section{' <text> '}' <optionList>
subsection     ::= '\subsection{' <text> '}' <optionList>
figure         ::= '\figure{' <filepath> '}' <optionList>
table          ::= '\begin{table}' <optionList> <tableContents> '\end{table}'
tableContents  ::= <text> ('|' <text>)* '\break'
list           ::= '\begin{list}' <optionList> <listContents> '\end{list}'
listContents   ::= '\item' <text>
verb           ::= '\begin{verb}' <optionList> (<anyChar>)+ '\end{verb}'
newpage        ::= '\newpage'
hline          ::= '\hline' <optionList>

#| Documento
document      ::= (<config>)* <metadata> (<command>)+
```

Funcionamiento

Para mantener el sistema lo más modular posible SPF está dividido en componentes, cada una independiente de las demás. Dado un archivo en formato SPF este atraviesa una serie de módulos para producir un PDF, en orden, son:

1. Parser: Se encuentra en `Parser.hs`. Se encarga de *parsear* el archivo, es decir convertir el texto del mismo en un AST. Cabe notar que en este punto no se validan los datos, el parser solo se ocupa de la correctitud *sintáctica* del archivo, no la *semántica*. Puede fallar.
2. Validador: Está compuesto por múltiples archivos, se encuentran en el directorio `Validation`. Valida los contenidos del AST generado por el parser. El trabajo principal del validador es verificar que las opciones en el archivo, tanto de configuración como de los comandos sean correctas; Esto conlleva verificar que cada par `clave:valor` sea del tipo y forma correcta para el comando o la configuración a él que corresponden. Devuelve otro AST y puede fallar.
3. Recursos: Se encuentra en `Resources.hs`. Carga todos los recursos necesarios para typesetter el documento. Por un lado están las fuentes, deben cargarse siempre, si esto falla el typesetting no puede realizarse. Por el otro lado están los recursos especificados por el usuario, también deben ser cargados antes de comenzar a type setear el documento, esta carga puede fallar.
4. Typesetter: El último eslabón de la cadena, toma el AST producido por el validador junto con los recursos cargados y los utiliza para generar un PDF.

Un poco más sobre ASTs

En un principio los ASTs usados simplemente almacenaban una lista de todos los comandos utilizados en el documento. Rápidamente me di cuenta de que esto era una mala idea. En primer lugar no había separación entre los distintos tipos lógicos de tokens, todos eran *comandos* en una lista. Lo que llevaba a un segundo problema era necesario iterar repetidas veces sobre esta lista para obtener distintos tipos de tokens. Hacer esto es ineficiente y no escala bien.

Para solucionar esto los ASTs se separaron en componentes funcionales, donde cada una corresponde a un tipo de comando. De esta forma hay una separación lógica entre los tokens; Al mismo tiempo solo se tira sobre una lista cuando es necesario recorrerla, mejorando la eficiencia y haciendo más rápido el programa.

¿Como fallar?

Cómo manejar fallos en un programa de este tipo no es una pregunta fácil de resolver, principalmente porque es posible tener *muchos* fallos. Sería entonces ideal poder acumular todos los errores que ocurren mientras el programa procesa un archivo; De esta forma el usuario podría solucionar múltiples errores al mismo tiempo, haciendo que se deba ejecutar el programa menos veces. Esto sin embargo no es posible, al menos del todo. No se pueden acumular errores de distintas secciones pues cada módulo depende de que el resultado de los anteriores sea correcto. Llegamos así a un compromiso, si bien no podemos acumular *todos* los errores a lo largo del programa si podemos acumular los errores de cada etapa.

En el parser esto se hace usando *delayed parse errors*. Estos son errores que no provocan que el parser falle inmediatamente, los errores acumulados se toman en cuenta al final del paseo, donde sí hay por lo menos uno el parser falla. Cabe notar que es posible obtener un error, que haga que el parser falle inmediatamente, esto puede ocurrir por uno de dos motivos. Por un lado los *delayed parse errors* deben implementarse manualmente; No es posible cubrir todos los casos, se cubrieron los errores más comunes y fáciles de producir. Por el otro lado no siempre es posible recuperarse de un error de parseo, generalmente debido a que el fallo deja contenido sin consumir del archivo, este contenido no encaja con ninguna de las reglas del parser, por lo que no puede seguir. A continuación se presenta un ejemplo:

```
\sectio{Digital Typography}
^  
Posición del cursor
```

El comando `\sectio` es invalido, al parsearlo el parser produce un *delayed parse error* y se recupera, continuando el parseo:

```
\sectio{Digital Typography}
^  
Posición del cursor
```

El comando `{Digital ...}` es invalido, mas aun este error no es un error del que el parser se pueda recuperar, ya que al fallar el parser de comandos no hay ningún parser que consuma `{Digital ...}`. Si bien se podría hacer que al fallar el parser de comandos consuma la string restante, esto corre el riesgo de generar mensajes de error más confusos para el usuario. En el caso de un bloque `begin/end`, se consumiría el inicio del bloque pero no el fin, generando un mensaje de error inapropiado cuando el parser encuentre `\end{...}`.

El resto de los módulos utiliza el paquete `Validation`, su descripción es bastante autoexplicativa: *A data-type like Either but with an accumulating Applicative*. Es decir cumple la misma función que `Either`, dándonos un resultado o un error, pero permitiéndonos acumular errores de forma fácil; Esta facilidad viene de la mano de que el acumulador de errores es un functor aplicativo, lo que permite concatenar validaciones fácilmente mediante `<*>`.

Generando PDFs

Una vez que se tiene un AST valido falta la parte más importante, transformarlo en un PDF. Para hacer esto utilice la librería HPDF. Elegí esta librería por un motivo simple, es la única librería para generar PDFs escrita en Haskell.

Habiendo usado esta librería puedo decir que si bien es muy poderosa *tiene problemas*, el principal y del cual derivan la mayoría es que es una librería con muy pocos usuarios. Como consecuencia de esto hay muy poca información sobre la librería. La documentación está incompleta, muchas funciones tienen explicaciones muy pobres, si es que tienen. Sumado a esto nunca se explica la estructura o flujo general de un programa hecho con HPDF, ni cómo se conectan los distintos componentes de la librería.

Lo que me permitió entender cómo funciona la librería fue el archivo de prueba en el repositorio del proyecto(github.com/hsyl120/HPDF/blob/master/Test/test.hs). Este prueba casi todas las funciones de la librería generando un documento. Me permitió ver un ejemplo en funcionamiento de la misma al mismo tiempo que me permitía experimentar y realizar cambios para ver *exactamente* como funcionaba la librería.

Un bug en HPDF

Mientras estaba escribiendo un informe descubrí un bug en HPDF, provoca que dadas ciertas condiciones el algoritmo de *line breaking* no funcione. Gracias a errores de redondeo de punto flotante puede ocurrir que una palabra no sea lo suficientemente larga como para causar un *line break* pero que si sea lo suficientemente larga como para exceder el margen de la caja donde está siendo typesetead. Esto causa que el algoritmo deje de introducir *line breaks* por completo.

En la figura de abajo se ve como el bloque de texto “autoexplicativa:” excede levemente el margen derecho, causando el bug antes descrito. Notar que esto no ocurre siempre, solo cuando un bloque de texto y la linea(kern + glue + margin) generan una combinacion *problematica*. En base a mi entendimiento de la librería para solucionar este bug es necesario modificar el código fuente de la misma, algo que excede el alcance del trabajo.

En el resto de los módulos se utiliza el paquete **Validation**, su descripción es bastante autoexplicativa: A

En el resto de los módulos se utiliza el paquete **Validation**, su descripción es bastante autoexplicativa A *data-type like Either but with an accumulating Applicative*. Es decir cumple la misma función que

Figure 1: Un ejemplo del error

¿Como hacer typesetting?

Typesetting es el proceso de organizar, formatear y disponer texto y elementos gráficos asociados en una página, se busca mejorar la legibilidad y estética. Si esto se hace bien los lectores no van a darse cuenta de que lo que están leyendo fue organizado de forma rígida y poco natural.

Donald E. Knut - Digital Typography(1999)

El algoritmo

El problema a solucionar parece engañosamente simple, dada una serie de elementos *acomodarlos* como mejor colocarlos en una(o mas) paginas de forma que el resultado sea estetico y legible. Este problema se puede abstraer, en lugar de elementos discretos pensamos en **cajas**, no importa que contienen(Texto, figuras, tablas, etc), lo importante es su tamaño.

El algoritmo implementado esta enfocado en ser lo mas simple posible, en particular no queria que implicase hacer multiples pasadas ni backtracking, algo que el algoritmo de LaTeX hace. Considerando esto y luego de algunas pruebas llegue a la conclusion de que un algoritmo basado en un cursor vertical es suficiente, especialmente al combinarlo con las funcionalidades de HPDF. Esto deriva en un algoritmo muy simple y facil de implementar. Una aproximacion en pseudocodigo es:

```

1 typeset [] = done
2 typeset (elem:xs) =
3     when checkSpace makeNewPage
4
5     case elem of
6         Paragraph p -> typesetParagraph p
7         ...
8         List l -> typesetList l
9         ...
10        Newpage -> makeNewPage

```

Antes de typesetear un elemento se verifica que haya suficiente espacio en la pagina, si no es asi se crea una pagina nueva, lo que automaticamente mueve el cursor al principio de la nueva pagina. Luego se typesetea el elemento correspondiente, ajustando el cursor segun es necesario. Esto se repite hasta que no queden mas elementos que typesetear.

Los detalles

A la hora de implementar el algoritmo quedo claro que era necesario mantener un estado entre llamadas de funcion. En un principio utilice una monada `StateT` para almacenar todos los datos. Si bien esto funcionaba presentaba un problema conceptual, esta monada tenia valores que eran solo de lectura, como la configuracion; Sin embargo al estar una monada `StateT` podian ser modificados.

Para solucionar esto se usan dos monadas concatenadas, primero una monada `ReaderT` su *environment* almacena los valores solo de lectura y su monada es una monada `StateT`. A su vez esta almacena en su *state* a todos los valores dinamicos, su monada es la monada `PDF` que almacena el documento que esta siendo typeseteado. En Haskell esto se expresa de la siguiente forma:

```
type Typesetter a = ReaderT RenderEnv (StateT RenderState PDF) a
```

En cuanto a las estructuras `RenderEnv` y `RenderState` la unico notable es que todos sus campos, menos uno, son estrictos. De lo contrario un PDF largo podria llenar la RAM o causar un `StackOverflow` debido a thunks sin evaluar.

Desafios

Bookmarks

Evitando bloat