

# 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 que el proyecto 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; Al mismo tiempo las opciones deben tener asociada una clave indicando a qué corresponde el valor. Finalmente simplifique la forma en la que estructura el documento, ya no es necesario separar el título y documento de forma explícita.

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.

## 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 en el parser para la conveniencia del usuario. 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> <verbContents> '\end{verb}'
verbContents   ::= (<anyChar>)+ (<notFollowedBy> '\end')>
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, siempre deben cargarse. Por el otro lado están los recursos especificados por el usuario, deben ser cargados antes de comenzar a typesetear el documento, esta carga puede fallar.

## 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 tipos de datos, 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 comandos (Configuración, metadata, etc). Hacer esto es ineficiente y no escala bien. Para solucionar esto los ASTs se separaron en componentes funcionales, donde cada uno responde a un tipo de comando. De esta forma solo se itera sobre una lista cuando es necesario recorrerla.

## ¿Como fallar?

Cómo manejar fallos en un programa de este tipo no es una pregunta facil de resolver, principalmente porque es posible tener *muchos* fallos. Seria entonces ideal poder acumular todos los errores que ocurren mientras el programa procesa un archivo. 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. Esto es más eficiente, pues le permite solucionar múltiples problemas al mismo tiempo, haciendo que se deba ejecutar el programa menos veces.

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 "normal", haciendo que el parser falle inmediatamente, esto ocurre ya que los *delayed parse errors* deben implementarse manualmente, uno a uno; Por lo tanto no es posible cubrir todos los casos posibles, se cubrieron los errores mas comunes y fáciles de producir.

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. Para que esto sea posible el tipo de datos del error debe ser un semigrupo, en este caso se usa una lista de `LocatedError`, son errores que contienen la ubicación aproximada, indicando que parte del archivo provocó el error.

## 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 esta 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 como se conectan los distintos componentes de la librería.

Lo que me permitió entender como funciona la librería fue el archivo de prueba en el repositorio del proyecto([github.com/hsyl20/HPDF/blob/master/Test/test.hs](https://github.com/hsyl20/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 error en HPDF

Mientras estaba escribiendo un informe descubrí un bug en `HPDF`, este error 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. 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 data-*

Figure 1: Un ejemplo del error

## Mas sobre typesetting

Hasta el momento he mencionado varias veces *typesetting*, pero no explique concretamente que es, formalmente se describe como:

Como: *El proceso de organizar, formatear y disponer el texto y elementos graficos en una pagina*