Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Sección de Ingeniería Informática

Construcción de un Compilador y un Intérprete de Scheme Usando Haskell

Building a Compiler and Interpreter for Scheme Using Haskell

Francisco Nebrera Perdomo

La Laguna, 28 de agosto de 2015

D. **Casiano Rodríguez León**, con N.I.F. 42.020.072-S profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*Construcción de un Compilador y un Intérprete de Scheme Usando Haskell.*"

ha sido realizada bajo su dirección por D. **Francisco Nebrera Perdomo**, con N.I.F. 79.064.507-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 28 de agosto de 2015

Agradecimientos

Casiano Rodríguez León
Jorge Riera Ledesma
Luz Marina Moreno de Antonio
Francisco Carmelo Almeida Rodríguez
Blas C. Ruiz
F. Gutiérrez
P. Guerrero
E. Gallardo
Daniel Díaz Casanueva

Licencia

© Esta obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.

Resumen

El objetivo de este trabajo ha sido crear un compilador e intérprete del lenguaje Scheme, usando Haskell. Para ello se ha hecho uso de un parser monádico como es Parsec.

La competencia [E6], que figura en la guía docente, indica que en la memoria del trabajo se ha de incluir: antecedentes, problemática o estado del arte, objetivos, fases y desarrollo del proyecto, conclusiones, y líneas futuras.

Se ha incluido el apartado de 'Licencia' con todas las posibles licencias abiertas (Creative Commons). En el caso en que se decida hacer público el contenido de la memoria, habrá que elegir una de ellas (y borrar las demás). La decisión de hacer pública o no la memoria se indica en el momento de subir la memoria a la Sede Electrónica de la ULL, paso necesario en el proceso de presentación del TFG.

El documento de memoria debe tener un máximo de 50 páginas.

No se deben dejar páginas en blanco al comenzar un capítulo, ya que el documento no está pensado para se impreso sino visionado con un lector de PDFs.

También es recomendable márgenes pequeños ya que, al firmar digitalmente por la Sede, se coloca un marco alrededor del texto original.

El tipo de letra base ha de ser de 14ptos.

Palabras clave: Palabra reservada1, Palabra reservada2, ...

Abstract

Here should be the abstract in a foreing language...

Keywords: *Keyword1, Keyword2, Keyword2, ...*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Todo empezó con un hilo en el foro de internet forocoches.com (`https://www.forocoches.com`), en él hablaban de que la programación funcional iba a tener cada día más relevancia porque cuenta con ventajas de las cuales la imperativa carece.

A raíz de ello, me interesé por este paradigma y empecé a (e incluso terminé de) leer numerosos libros sobre el lenguaje y la programación funcional en general, y a crear pequeños programas en Haskell. Haskell es muy interesante debido a que es el lenguaje con mayor nivel de abstracción en el que he programado hasta hoy.

Haskell es idóneo para crear lenguajes de dominio específico. En otras palabras, antes de escribir un compilador se captura el lenguaje a compilar (el lenguaje fuente) en un tipo. Las expresiones de ese tipo representarán términos en el lenguaje fuente y normalmente son bastante similares al mismo, a pesar de ser, realmente, tipos de Haskell.

Luego se representa el lenguaje objetivo como otro tipo más. Finalmente, el compilador es realmente una función del tipo fuente al tipo objetivo y las traducciones son fáciles de escribir y leer. Las optimizaciones también son funciones como cualquier otra (ya que realmente en Haskell todo es una función, y además, currificada) que mapean del dominio del lenguaje fuente al codominio del lenguaje objetivo.

Por ello los lenguajes funcionales con sintaxis ligera y un fuerte sistema de tipos se consideran muy adecuados para crear compiladores y muchas otras cosas cuya finalidad es la traducción.

Además, Haskell cuenta con mecanismos de abstracción muy fuertes que permiten escribir códigos escuetos que se comportan muy bien, como por ejemplo:

- Reconocimiento de patrones
- Tipos de datos algebraicos (generalizados o no)
- Lambdas (y por ello, mónadas)
- Plegados de listas

A continuación se describen los mencionados constructos en su sección correspondiente.

1.1. Reconocimiento de patrones y tipos de datos algebraicos

Para entender qué es el reconocimiento de patrones primero debemos saber qué es **casar**. Para ello usaremos las acepciones pertinentes del diccionario de la Real Academia:

- Dicho de dos o más cosas: Corresponder, conformarse, cuadrar.
- Unir, juntar o hacer coincidir algo con otra cosa. Casar la oferta con la demanda.
- Disponer y ordenar algo de suerte que haga juego con otra cosa o tengan correspondencia entre sí.

Es un término que se usa bastante en las expresiones regulares, para ver si una expresión casa con un texto dado, y en qué lugar. Veamos un ejemplo de reconocimiento de patrones:

```
dime :: Int -> String
dime 1 = "¡Uno!"
dime 2 = "¡Dos!"
dime 3 = "¡Tres!"
dime 4 = "¡Cuatro!"
dime 5 = "¡Cinco!"
dime x = "No está entre 1 y 5"
```

La función **dime** hace reconocimiento de patrones con su primer argumento, de tipo **Int**, y va de arriba a abajo intentando encontrar una coincidencia. Cuando recibe un número entre 1 y 5, lo canta con ahínco, si no lo encuentra, nos devolverá un mensaje diciéndonoslo. Notar además que si hubiéramos puesto la línea **dime x = "No está entre 1 y 5"** al principio, nuestra función siempre devolvería "No está entre 1 y 5", aun siendo cierto. Por tanto, debemos ordenar los patrones por probabilidad; de los menos probables a los más probables.

Cuando hablamos de reconocimiento de patrones hablamos, en realidad, de reconocimiento de constructores. En concreto en Haskell existen dos tipos de constructores, los constructores de tipos (los tipos que aparecen en las declaraciones de las funciones) y los constructores de valor (aquellos que se suelen poner entre paréntesis, y son funciones que recibiendo un valor crean un tipo que encapsula dicho valor).

Importante: los dos guiones al principio de cada línea representan comentarios en Haskell, y su presencia aquí es de carácter didáctico.

```
data Persona = CrearPersona String Int
--                          |      |
--                          |      |
--                          |      La edad de la persona
--                          El nombre de la persona
```

A la izquierda del igual está el constructor de tipo. A la derecha del igual están los constructores de datos. El constructor de tipos es el nombre del tipo y usado en las declaraciones de tipos. Los constructores de datos son funciones que producen valores del tipo dado. Si solo hay un constructor de datos, podemos llamarlo igual que el de tipo, ya que es imposible sustituirlos sintácticamente (recuerda, los constructores de tipos van en las declaraciones, los constructores de valor en las ecuaciones).

```
data Persona = Persona String Int
--   |         |
--   |         |
--   |         Constructor de valor (o datos)
--   Constructor de tipo
```

El tipo del último ejemplo se conoce como **tipo de dato algebraico**; tipos de datos construidos mediante la combinación de otros tipos. El reconocimiento de patrones es una manera de desestructurar un tipo de dato algebraico, seleccionar una ecuación basada en su constructor y luego enlazar los componentes a variables. Cualquier constructor puede aparecer en un patrón; ese patrón casa con un valor si la etiqueta del patrón es la misma que la etiqueta del valor y todos los subpatrones casan con sus correspondientes componentes.

Importante: el reconocimiento de patrones es en realidad reconocimiento de constructores.

1.2. Variables de tipo

Las variables de tipo son aquellas que se declaran en **data** después del nombre del tipo que vamos a crear. Su finalidad principal es hacer saber qué puede formar parte del tipo, y además permitir a cualquier tipo formar parte de nuestro tipo personalizado. Veámoslo con un ejemplo:

```
data Persona a = PersonaConCosa String a | PersonaSinCosa String
--             |                         |
--             |                         |
--             |                         Podemos usarla aquí
--             Introduciendo una variable de tipo aquí
```

En los siguientes ejemplos se ilustra el deber de informar al compilador qué tipo queremos que nuestra función devuelva, y así producir un tipo **Persona Int**, **Persona String**,...,etc.

```
franConEdad :: Persona Int
franConEdad = PersonaConCosa "fran" 25

franSinEdad :: Persona Int
franSinEdad = PersonaSinCosa "fran"
```

Ahora llega el reconocimiento de patrones propiamente dicho; según se encuentre el constructor **PersonaConCosa String a** ó **PersonaSinCosa String**, nuestra función debe ser programada para actuar en consecuencia:

```
getNombre :: Persona Int -> String
getNombre (PersonaConCosa nombre _) = nombre
getNombre (PersonaSinCosa nombre)   = nombre

getEdad :: Persona Int -> Maybe Int
getEdad (PersonaConCosa _ edad) = Just edad
getEdad (PersonaSinCosa _)      = Nothing
```

Como vemos, a las variables **nombre** y **edad** respectivamente se le han enlazado sus valores reales, que son los que nuestra función devuelve. Como el constructor **PersonaSinCosa** sólo contiene el nombre y no la edad, utilizamos el tipo **Maybe** para devolver **Nothing** en caso de que ese patrón (constructor) sea reconocido. En el otro caso, devolvemos **Just edad** ya que en este caso la tenemos.

1.3. Lambdas

En un lenguaje funcional, poner nombre a todas las funciones que usemos podría resultar tedioso. Además, es cómodo definir funciones al vuelo, es decir, rápidamente y en el punto del programa en el que realmente sea necesario. Las lambdas sirven para este propósito.

Las lambdas se suelen declarar entre paréntesis para que el compilador sepa que se tratan de un "todo". No obstante, en el caso de las mónadas hay veces en que no son necesarios y se resuelve todo mediante la indentación.

La sintaxis de las lambdas es la siguiente:

```
\arg1 arg2 ... argn -> cuerpo_función
```

Es decir, para que Haskell sepa que estamos trabajando con una lambda, se usa la backslash \ y a continuación se encuentran dos partes bien diferenciadas, separadas por una flecha →:

- A la izquierda de la flecha → la lista de nombres de argumentos, separados por espacios.
- A la derecha de la flecha → el cuerpo de la función. El tipo de esa expresión será el tipo retorno de la lambda.

Definamos la lambda más sencilla que existe, lo único que hace es devolver su argumento:

```
\x -> x
```

Definamos una lambda que eleve al cubo un número:

```
\x -> x*x*x
```

Veamos ahora una lambda que sume sus dos argumentos:

```
\x y -> x + y
```

Y por último, veamos una que ignora su primer argumento y devuelva el segundo:

```
\_ x -> x
```

Como vemos, se puede usar el patrón subrayado (barra baja) para expresar que no nos importa el valor del primer parámetro, ya que sólo usamos el segundo. Las lambdas tienen mucha importancia en Haskell, y son muy útiles.

1.4. Plegados de listas

Como ejemplo pondré un DFA hecho mediante un plegado de listas por la izquierda.

```
probarDFA :: DFA -> [Char] -> Bool
probarDFA (DFA i a t) = a . foldl' t i
```

Un DFA se podría implementar en programación imperativa con un bucle for que fuera sobreescribiendo el estado en cada iteración, haciendo un lookup en su tabla de estados dependiendo de su estado actual y el símbolo leído.

Esto, en Haskell, se puede hacer usando la función **foldl** (aunque aquí por razones de rendimiento y uso de memoria se ha optado por **foldl'**).

Se trata de empezar con un acumulador (en este caso, el estado inicial), y nos vamos moviendo por la lista (cadena de entrada) de izquierda a derecha, haciendo un lookup con el acumulador y el carácter leído en ese instante, y luego el resultado (estado siguiente) se convertirá en el nuevo acumulador y se repetirá el proceso.

La función **scanl** nos permite ver la lista de todos los valores que ha

ido tomando el acumulador durante la ejecución del programa, y se comporta como **foldl** pero devolviendo la lista completa:

```
*DFA> leerDFA "dfa1.txt"
Cadena:AAABABABA
["Q1","Q2","Q1","Q2","Q2","Q2","Q1"]
```

Luego de la aplicación de **foldl'** vemos un punto, que significa composición, es decir, aplicará la función **a** al resultado de **foldl'**, donde **a** es una función que comprueba si ese estado pertenece a la lista de estados finales, devolviendo un booleano que indicará la aceptación o rechazo de la cadena por el autómata.

Como vemos, en Haskell con una sóla línea se pueden hacer virguerías, el programa completo que simula un DFA, leyendo desde fichero y pidiendo continuamente entrada tras computar la anterior, ocupa 35 líneas y puede ser consultado en `https://github.com/freinn/libroshaskell/blob/master/write48/DFA_propio/DFA.hs`.

Capítulo 2

Librería Parsec

2.1. Introducción a Parsec

En el capítulo anterior se ha realizado una introducción a la programación funcional con Haskell, y ahora se describirá un módulo muy útil en la creación del intérprete, Parsec.

Parsec es un módulo de Haskell, un conjunto de funciones exportables que suelen tener una finalidad común y se pueden importar en otros programas. En nuestro intérprete hemos importado Parsec, pues es el módulo utilizado en el tutorial que seguimos, Write Yourself a Scheme in 48 hours.

Parsec se diseñó desde cero como una librería de parsers con capacidades industriales. Es simple, segura, está bien documentada, provee de buenos mensajes de error y es rápida. Se define como un transformador de mónadas que puede ser apilado sobre mónadas arbitrarias, y también es paramétrico en el tipo de flujo de entrada. La documentación de la versión usada en el presente Trabajo Fin de Grado se puede consultar online en `https://hackage.haskell.org/package/parsec-3.1.9`

Parsec se puede leer en "inglés plano"(siempre que nuestros parsers

tengan los nombres adecuados). Se pueden hacer analogías entre las funciones de Parsec y las expresiones regulares, como veremos en el ejemplo de código de este capítulo.

La mejor manera de describir el módulo es mediante un ejemplo, un parser del conocido formato JSON.

2.2. Librerías necesarias

```
import Text.ParserCombinators.Parsec hiding ((<|>), many)
import Text.Parsec.Numbers (parseFloat)
import Control.Applicative
import Control.Monad
import Prelude hiding (Null, null)
```

2.3. Formato a parsear

El formato JSON (JavaScript Object Notation) es de los más comunes hoy en día para el intercambio de información a través de la red. Es un formato sencillo y fácil de parsear, y por ello está ganando terreno frente a su competidor principal, XML. Sus principales elementos son:

- Number
- String
- Boolean
- Array
- Object
- Null

Empezaremos definiendo los parsers más sencillos, cuyo fin es devolver argumentos que entrarán en constructores de valor para tipos de JSON que siempre valgan lo mismo. Estos 3 tipos son: **true**, **false** y **null**.

```
alwaysTrue :: Parser Bool
alwaysTrue = pure True

alwaysFalse :: Parser Bool
alwaysFalse = pure False

alwaysNull :: Parser String
alwaysNull = pure "null"
```

La misión de *pure* :: $a\ f\ a$ (donde en este caso **f** es la mónada **Parser**) no es otra que envolver los dos **Bool** y la **String** en un valor monádico, devolviendo de este modo un **Parser Bool** o un **Parser String**. Por tanto, estas funciones devuelven un **Parser**, que cuando se ejecuta (mediante la función **parse**) devuelve un **Bool** o una **String**.

Ahora lo que debemos hacer es usar el parser **string**, que intenta casar con una cadena dada, devolviéndola en caso de que consiga casar:

```
matchTrue :: Parser String
matchTrue = string "true"

matchFalse :: Parser String
matchFalse = string "false"

matchNull :: Parser String
matchNull = string "null"
```

Por último, no devolveremos la cadena propiamente dicha, sino un valor puro (por ello antes definimos funciones que usan **pure**):

```
boolTrue :: Parser Bool
boolTrue = matchTrue *> alwaysTrue
```

```
boolFalse :: Parser Bool
boolFalse = matchFalse *> alwaysFalse

null :: Parser String
null = matchNull *> alwaysNull
```

Aquí usamos un operador de la clase de tipos **Applicative**, que en inglés se suele llamar "star arrow". Este operador ejecuta primero el parser de la izquierda, luego el de la derecha, y devuelve sólo lo que parsee el de la derecha (el sitio al que apunta la flecha).

Ahora veamos qué pasa si un token puede pertenecer a un tipo aún más general:

```
bool :: Parser Bool
bool = boolTrue <|> boolFalse
```

$< | >$ es el operador de elección, y se parece mucho a la barra vertical | de las expresiones regulares. Pueden ser encadenados tantos parsers como queramos. Este operador lo que hace es:

- 1. intenta el parser de la izquierda, que no debería consumir entrada...(ver **try**)
- 2. intenta el parser de la derecha.

Si el parser de la izquierda consume entrada, podríamos usar **try**, el cual intenta ejecutar ese Parser, y, si falla, vuelve al estado anterior, es decir, deja la entrada sin consumir. Sólo funciona a la izquierda de $< | >$, es decir, si queremos encadenar varios **try**, deben estar a la izquierda de la cadena de $< | >$.

try es como un lookahead, y se puede ver como algo para procesar cosas de manera atómica, **try** es realmente backtracking, y por ello no es demasiado eficiente.

Como en este caso las string que vamos a parsear no tienen prefijos coincidentes, no hace falta usar **try** por si hay que volver a empezar.

Ahora empezaremos a ver algo que se parece aún más a las expresiones regulares:

```
stringLiteral :: Parser String
stringLiteral = char '"' >> (many (noneOf ['"'])) << char '"'
```

Aquí vemos que Parsec puede leerse casi en "inglés plano", ya que esta línea casi se autodescribe. Primero, debe encontrarse un carácter **"**, luego vemos la función **many**, que equivale a la estrella ***** de las expresiones regulares, es decir, podría haber muchos, uno o ninguna ocurrencia del parser que reciba **many**.

Luego vemos una función noneOf, que es un parser que acepta todo menos los carácteres que pertenezcan a una lista determinada, en este caso acepta todo menos las comillas dobles, en caso de toparse con comillas dobles (la cadena ha acabado), deja de consumir entrada.

Para terminar, se vuelve a parsear un carácter **"**, que debe estar obligatoriamente. Ahora vemos que nuestra combinación aplicativa sigue una estructura $\mathbf{a} > * \mathbf{b} < * \mathbf{c}$, esto indica que los parsers **a**, **b** y **c** deben tener éxito, pero como sólo se devuelve lo que está apuntado por las flechas, sólo devolveremos lo que haya parseado b, que en este caso corresponde a **(many (noneOf ['"']))**.

La siguiente línea da error de tipos:

```
value = bool <|> stringLiteral
```

Solución: crear un tipo algebraico que contenga Bool y String (entre otros). Los tipos de datos algebraicos son una herramienta muy útil para los parsers, ya que permiten saber exactamente a qué tipo pertenece el token que hemos parseado.

```
data JSONValue = B Bool
               | S String
               | N Double --número de JSON
               | A [JSONValue] --array de JSON
               | O [(String, JSONValue)] --objeto de JSON
               | Null String
               deriving Show
```

Como vemos, tenemos un sólo constructor de tipo, **JSONValue**, es decir, nuestros parsers tendrán tipo **Parser JSONValue**. Sin embargo, tenemos 6 constructores de valor, que por simplicidad son simplemente las letras Iniciales de cada tipo de valor a parsear, salvo **Null**, en el cual se usó el nombre completo ya que **N** se usó para el tipo Number de JavaScript.

Veamos ahora el parser principal, es decir, un parser genérico capaz de parsear cualquier valor de JSON:

```
jsonValue :: Parser JSONValue
jsonValue = spaces >> (jsonNull
                   <|> jsonBool
                   <|> jsonStringLiteral
                   <|> jsonArray
                   <|> jsonObject
                   <|> jsonNumber
                   <?> "JSON value")
```

No te preocupes demasiado por el parser **spaces**, lo explicaré más adelante en conjunto con **lexeme**. Pero, ¿qué es esa interrogación? <? > es un combinador que permite dar mensajes de error en caso de parseo fallido. En este caso, se le pasa una String con el mensaje de error que queremos que aparezca. En caso de error, saldrá algo como "expected JSON value", pues ese es el argumento de <? > para cuando falle el parser **jsonValue**.

Lo malo de esto es que seguimos teniendo error de tipos porque:

```
bool :: Parser Bool
stringLiteral :: Parser String
```

Lo bueno es que con $(<\$>) :: Functor\, f => (a -> b) -> f\, a -> f\, b$, que en este caso tendría el tipo: $(<\$>) :: (a -> b) -> Parser\, a -> Parser\, b$, podemos solucionarlo.

Recordemos que los constructores de valor son en realidad funciones como otra cualquiera (salvo que empiezan por mayúscula). Por ejemplo, el tipo de **B** es $B :: Bool -> JSONValue$. Por tanto, si hacemos **B <$> bool** tendremos como resultado un **Parser JSONValue**, y eso haremos en todos nuestros parsers anteriormente nombrados.

```
jsonBool'' :: Parser JSONValue
jsonBool'' = B <$> bool

matchNull'' = lexeme matchNull'

jsonStringLiteral :: Parser JSONValue
jsonStringLiteral = lexeme (S <$> stringLiteral)
```

Aquí lo único que nos llama la atención es el parser **lexeme**. **lexeme** está definido en Parsec por defecto, pero nosotros lo programaremos más que nada por razones didácticas.

lexeme es un parser que, recibiendo otro parser, devuelve un parser del mismo tipo, pero que consume todos los espacios (incluyendo tabuladores y newlines) que haya detrás (a la derecha) del token parseado.

```
ws :: Parser String -- whitespace
ws = many (oneOf " \t\n")

lexeme :: Parser a -> Parser a
lexeme p = p <* ws
```

De este modo, con aplicar lexeme a cada uno de los parsers que vayamos a usar, tenemos resuelto el problema de los espacios entre tokens.

Bueno, ahora que el problema de los espacios está resuelto...¡sorpresa! no lo está del todo...Como hemos dicho, el combinador **lexeme** se çome"todos los espacios, tabuladores o newlines que encuentre des-

pués del token parseado. Pero, ¿y si esos espacios estuvieran antes del primer token que llegamos a parsear? Probablemente se produciría un error.

Solución: añadir el parser spaces a nuestro parser principal **jsonValue**. Esto se hizo mediante el operador monádico >>, que en la mónada de Parsec tiene el efecto de ejecutar ese parser, y si tiene éxito no guardar el resultado del parsing, sino pasar al siguiente. Se ha usado >> para ilustrar el uso de esta función, ya que se había introducido antes: $*>$, también llamado "star arrow".

A continuación creemos un parser que permita parsear números. Para ello usaremos la función **parseFloat**, que permite parsear cualquier tipo de número, incluso con signo, exponente, parte decimal...es decir, el formato de coma flotante.

```
jsonNumber :: Parser JSONValue
jsonNumber = N <$> parseFloat
```

¡Listo! ya tenemos un parser más. Ahora veamos algo un poco más complejo, los arrays de JSON. Un array de JSON tiene el siguiente formato:

```
[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName":"Jones"}
]
```

Como vemos, tenemos:

- 1. Un carácter abrir corchetes [
- 2. Un conjunto de tokens de JSON, separados por comas.
- 3. Un carácter cerrar corchetes]

Sabido esto, lo único nuevo que tenemos que introducir aquí es el parser **sepBy**. **sepBy** recibe dos argumentos, el primero es el parser que se usará para cada token, y el segundo es el parser que se usará para el separador o separadores. Veamos el parser completo.

```
array :: Parser [JSONValue]
array =
  (lexeme $ char '[')
  *>
  ( jsonValue `sepBy` (lexeme $ char ',') )
  <*
  (lexeme $ char ']')
```

Como los arrays contienen tokens de JSON, lo que hacemos es una llamada recursiva a **jsonValue**. De este modo, vemos que dentro de un array de JSON puede haber "lo que sea"(siempre que esté correctamente escrito y formateado) pero el array debe empezar por el carácter [y terminar con] para garantizar que dicho formato sea correcto. Como vemos, este parser nos devuelve una lista de **JSONValue**, y eso no es un tipo **JSONValue**. Por tanto, debemos aplicar $fmap(< \$ >)$, en este caso de manera infija:

```
jsonArray :: Parser JSONValue
jsonArray = A <$> array
```

Ahora parsearemos algo parecido pero no del todo igual, los objetos de JSON. El formato de los objetos es:

- 1. Un carácter abrir llaves **{**
- 2. Una lista de pares separador por el carácter dos puntos ':'
- 3. Un carácter cerrar llaves **}**

```
jsonObject :: Parser JSONValue
jsonObject = O <$> ((lexeme $ char '{') *>
                    (objectEntry `sepBy` (lexeme $ char ','))
                    <* (lexeme $ char '}'))

objectEntry :: Parser (String, JSONValue)
objectEntry = do
  key <- lexeme stringLiteral
  char ':'
  value <- lexeme jsonValue
  return (key, value)
```

Ahora consigamos que el parser **jsonBool'** sea capaz de lidiar con espacios, tabuladores y nuevas líneas después del token que parsea:

```
jsonBool' = lexeme jsonBool''
```

Ya casi hemos terminado, pero aún falta un pequeño detalle. ¿Y si alguien se equivoca y escribe por ejemplo "falseee", o "nullpointer", o cualquier otra cosa siguiendo a las palabras reservadas **true**, **false** o **null**? Nuestro parser lo aceptaría, cuando eso no debería ser así. Queremos exactamente esas palabras, ni un carácter más ni uno menos, para que nuestro parser sea correcto. Para ello, Parsec nos provee con un parser que falla en caso de que otro esté seguido de ciertos carácteres, es **notFollowedBy**. **notFollowedBy** recibe un parser, y si éste tiene éxito, falla. Un ingenioso truco que nos saca del atolladero de manera muy sencilla y casi autoexplicativa.

```
jsonBool :: Parser JSONValue
jsonBool = jsonBool' <* notFollowedBy alphaNum

jsonNull :: Parser JSONValue
jsonNull = matchNull'' <* notFollowedBy alphaNum
```

Por último, creemos la función **main** que nos permitirá compilar el programa. Para ello seguiremos los siguientes pasos:

- 1. Mostrar por pantalla qué queremos.
- 2. Obtener el nombre del fichero por entrada estándar (teclado)

y ligarlo al nombre **filename**.

- 3. Aplicar nuestro parser principal (**jsonValue**) a nuestro fichero **filename** mediante la función **parseFromFile**.

```
main = do
  putStr "Nombre_fichero: "
  filename <- getLine
  parseFromFile jsonValue filename
```

Eso es todo, ya tenemos nuestro parser de JSON funcionando.

Capítulo 3

El lenguaje Scheme

3.1. Introducción a Scheme

El lenguaje Scheme es un dialecto de Lisp. Es un lenguaje que, como Lisp, es funcional. Es un lenguaje relativamente sencillo de aprender, con un poco de estudio se puede llegar a tener un conocimiento medio en poco tiempo.

Scheme fue elegido para escribir uno de los mejores libros sobre Ciencias de la Computación que se han escrito: Structure and Interpretation of Computer Programs, de la editorial MIT Press. Asimismo, para mis pruebas he usado el intérprete de Scheme del MIT, llamado mit-scheme.

El lenguaje usa la notación polaca para las llamadas a funciones y para cualquier tipo de evaluación. Además, cualquier definición o llamada a una función deberá hacerse entre paréntesis. Asimismo, las listas, contengan lo que contengan, se representan como una lista de ciertos valores de cierto tipo, entre paréntesis y separados por espacios.

Veamos una sesión con el intérprete oficial de Scheme para familiarizarnos un poco con el lenguaje:

```
(+ 3 5)
8
```

```
(sqrt 144)
12
```

Las funciones se definen mediante la palabra reservada **define**, una lista que contiene el nombre de la función y la lista de argumentos (todos separados por espacios), y, por último, el cuerpo de la función. Todo esto debe estar entre paréntesis.

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

Las funciones **car**, **cdr** y **cons** actúan sobre una lista. La función **car** devuelve el primer elemento de la lista (si éste existe), mientras que **cdr** devuelve una lista compuesta por el segundo elemento de la lista y todos los que le siguen. La función **cons** añade un elemento al principio de una lista (por su izquierda). Se usa un wrapper llamado **rev** que llama a la función **rev-rec** con la lista que se le pasa y la lista vacía **()**.

```
(define (rev lst) (rev-rec lst ()))

(define (rev-rec lst acc)
  (cond ((null? lst) acc)
        (else (rev-rec (cdr lst)
                       (cons (car lst) acc)))
        ))
```

3.2. Segundo apartado de este capítulo

3.3. Tercer apartado de este capítulo

Capítulo 4

Funcionamiento del Compilador e Intérprete

En el capítulo 3 se describió el lenguaje Scheme, dando una idea general sobre el mismo. En el presente capítulo se tratará de explicar el funcionamiento del programa principal del presente Trabajo Fin de Grado.

4.1. Estructura general del programa

En un programa escrito en Haskell, la mayoría de funciones son puras, es decir, sólo conocen el conjunto de argumentos que reciben y sólo pueden actuar sobre ellos, devolviendo al final de su ejecución un valor de su tipo de retorno.

El concepto descrito de función pura no se corresponde con el de las funciones de lenguajes imperativos como C, C++ o Java. En los mencionados lenguajes, las funciones no tienen demasiado que ver con las funciones matemáticas formales. Las funciones de C, C++ o Java pueden imprimir por pantalla, interactuar con el usuario y, en definitiva, recibir información que está más allá del ámbito de sus parámetros.

Para superar esta dificultad se creó el concepto de mónada. Las móna-

das tienen fama de algo más que abstracto, muy difícil de entender, y estoy de acuerdo en que no son sencillas, pero las intentaré explicar al menos hasta donde llega mi entendimiento. Las mónadas, en esencia, son constructos que se realizan con lambdas estructuradas de un modo especial y con sus argumentos, y de este modo se consigue encapsular una computación o acción. Como un argumento podría depender de argumentos de lambdas anteriores, se consigue, en el caso de la mónada IO, que las acciones se ejecuten en el orden deseado. Recordemos que Haskell, al ser un lenguaje funcional, ejecuta "todo a la vez", y mediante las mónadas, conseguimos una ejecución ordenada (entre otras cosas).

En el programa que se pasará a describir se han usado varias mónadas, cada cual cumple una función distinta, que, sin más dilación, pasamos a explicar.

El tipo **IO** es instancia de la clase de tipos **Monad**, mónada es un concepto, decir que un valor pertenece a la clase de tipos **Monad** es decir:

- 1) Hay (un cierto tipo de) información oculta adjunta a este valor.
- 2) La mayoría de funciones no se tienen que preocupar de esa información.

En este caso:

La información extra son acciones **IO** que se ejecutarán usando los valores que se van pasando de una a otra; mientras que el valor básico (el cual tiene información adjunta) es void, la tupla vacía o unidad, **()**.

IO [**String**] e **IO ()** pertenecen al mismo tipo, el de la mónada **IO**, pero tienen distintos tipos base. Actúan sobre (y se pasan unos a otros) valores de distintos tipos, [**String**] y **()**.

Los valores con información oculta adjunta son llamados valores monádicos.

Los valores monádicos se suelen denominar acciones, porque la manera más fácil de pensar en el uso de la mónada **IO** es pensar en una secuencia de acciones afectando al mundo exterior. Cada acción de la mencionada secuencia de acciones podría actuar sobre valores básicos (no monádicos). Por tanto:

- **m a** es una acción
- **(a -> m ())** es una función que devuelve una acción que contiene la tupla vacía o unidad **()**.

En los bloques do no se pueden mezclar acciones de mónadas diferentes.

Hay dos maneras de crear una acción IO:

- Elevar un valor ordinario en la mónada IO, usando la función **return**.
- Combinar dos acciones existentes.

Para combinar estas acciones, usamos un bloque **do**. Un bloque **do** consiste en una serie de líneas (las cuales tienen que tener la misma indentación). Cada línea puede tener una de estas dos formas:

- nombre $<-$ acción1
- acción2

La primera forma liga el resultado de **acción1** a nombre, para que esté disponible en las siguientes acciones. Por ejemplo, si el tipo de **acción1** es **IO [String]**, entonces el nombre estará ligado en todas

las acciones y lo podremos usar en acciones posteriores, y esto se consigue mediante el operador bind (>>=).

En la segunda opción, simplemente ejecutamos la acción (por ejemplo, imprimir algo por pantalla) pero no ligamos nada a ningún nombre, ya que consideramos que no es necesario. Esto se consigue mediante el operador (>>).

Parsec (en realidad, **genParser**) es otro ejemplo de mónada: en este caso, la información extra que se encuentra oculta es toda aquella relativa a la posición en la cadena de entrada, registro de backtracking, conjuntos first y follow...etcétera.

La función **parse** devuelve un **Either**, que tendremos que manejar según construya un **Left** (ParseError) o **Right** (valor correcto).

readExpr recibe una **String** (la cadena de entrada) y devuelve otra **String** con información de lo que haya parseado.

readExpr utiliza la función parse, que devuelve un **Either**, que **readExpr** maneja según construya un valor de tipo **Left** (error) o un valor de tipo **Right** (valor correcto).

Luego se trata de ir parseando los diferentes tokens de Scheme y luego construir, mediante un constructor de valor para el tipo **LispVal**, un valor determinado.

Para ello se aplican parsers de Parsec y se extrae la información oculta de aquello que han parseado mediante el constructo **<-**, o se usa (**liftM** función valor_monádico). Esto quizás requiera una explicación adicional:

- **<-** permite ligar a un nombre la información oculta en un valor monádico.
- (**liftM** función valor_monádico) "eleva" (de ahí que su nombre

empiece por lift) una función que actúa sobre tipos no monádicos a otra que actúa sobre los valores que la monáda tiene dentro.

Parsers recursivos:

En un lenguaje funcional la recursividad es uno de los métodos más interesantes. En un lenguaje como Scheme y muchos otros, encontramos estructuras de datos que pueden contener a otras, por ejemplo, una lista puede contener:

- otras listas (sean normales o de tipo dotted)
- cualquier otra expresión.

Por tanto, en el intérprete se llama recursivamente al parser principal, **parseExpr :: Parser LispVal**, con el objetivo de parsear lo que haya dentro de cada expresión.

Por ejemplo, en **parseList** y **parseDottedList** se usan, respectivamente:

- sepBy parseExpr spaces
- endBy parseExpr spaces

Los cuales van a devolver una [**LispVal**], justo el argumento que necesita el constructor de tipo **List** y el primero que necesita **DottedList**.

Por tanto, vemos que mediante el uso de **sepBy** y **endBy** estamos haciendo llamadas recursivas a **parseExpr** y por ello nuestro parser es capaz de lidiar con estructuras anidadas de manera casi gratuita.

Lo primero de todo es hacer **LispVal** instancia de **Show** para poder imprimir por pantalla los valores de tipo **LispVal**.

Para ello se crea la función **showVal :: LispVal -> String** y se iguala a **show** (**instance Show LispVal where show = showVal**). Para los dos tipos de listas, **List** y **DottedList**, se usa la función **unwordsList**:

```
unwordsList :: [LispVal] -> String
unwordsList = unwords . map showVal
```

Ahora empezamos con el evaluador propiamente dicho:

```
eval :: LispVal -> LispVal
eval val@(String _) = val
eval val@(Number _) = val
eval val@(Bool _) = val
eval (List [Atom "quote", val]) = val
```

LispVal se puede ver como una expresión, y cambiaremos **LispVal** para que devuelva una expresión en vez de su valor de cadena de carácteres.

```
readExpr :: String -> LispVal
readExpr input = case parse parseExpr "lisp" input of
    Left err -> String $ "No match: " ++ show err
    Right val -> val
```

Cambiamos el código de la función **main** para que evalúe las expresiones en vez de sólo imprimirlas por pantalla:

```
main :: IO ()
main = getArgs >>= print . eval . readExpr . head
```

Añadimos a la función eval una ecuación que nos permitirá aplicar funciones a sus argumentos (para aplicar funciones se debe poner una lista cuyo primer elemento es el nombre de la función y luego los demás elementos serán los argumentos de dicha función):

```
eval (List (Atom func : args)) = apply func $ map eval args
```

Como vemos, tenemos hecha una ecuación que es recursiva, mapea eval sobre los argumentos, con lo cual ya tenemos resuelto el problema de evaluar listas anidadas de manera "gratuita".

```
apply :: String -> [LispVal] -> LispVal
apply func args = maybe (Bool False) ($ args) $ lookup func primitives
```

La función **maybe** recibe un valor por defecto, una función, y un valor de tipo **Maybe**. Si el valor de tipo **Maybe** es **Nothing**, la función devuelve el valor por defecto. Si no, aplica la función al valor dentro del **Just** y devuelve el resultado.

```
primitives :: [(String, [LispVal] -> LispVal)]
primitives = [("+", numericBinop (+)),
              ("-", numericBinop (-)),
              ("*", numericBinop (*)),
              ("/", numericBinop div),
              ("mod", numericBinop mod),
              ("quotient", numericBinop quot),
              ("remainder", numericBinop rem)]

numericBinop :: (Integer -> Integer -> Integer) -> [LispVal] -> LispVal
numericBinop op params = Number $ foldl1 op $ map unpackNum params

unpackNum :: LispVal -> Integer
unpackNum (Number n) = n
unpackNum (String n) = let parsed = reads n :: [(Integer, String)] in
                           if null parsed
                              then 0
                              else fst $ parsed !! 0
unpackNum (List [n]) = unpackNum n
unpackNum _ = 0
```

Aquí vemos la función **primitives**, que devuelve una lista de pares (string, función), estableciendo una correspondencia entre funciones de Scheme y Haskell. En **numericBinop** tenemos una función que recibe:

- una función binaria $\mathbb{Z}^2 \to \mathbb{Z}$
- una lista de **LispVal**

Lo que hace es primeramente desempaquetar los números, es decir, pasarlos al tipo **Integer**. Después les aplica un plegado a la izquierda, es decir, va haciendo la operación binaria con los enteros de la lista, dos a dos (asonciando a izquierdas), y va acumulando el resultado

para usarlo con el siguiente entero de la lista. Esto nos permite que operaciones no conmutativas como la resta den el resultado correcto de manera casi gratuita.

Manejo de errores:

En primera instancia crearemos un tipo de dato algebraico que generalice los errores mediante un constructor de tipos, y los concretice mediante constructores de valor:

```
data LispError = NumArgs Integer [LispVal]
               | TypeMismatch String LispVal
               | Parser ParseError
               | BadSpecialForm String LispVal
               | NotFunction String String
               | UnboundVar String String
               | Default String
```

Luego, crearemos una mónada llamada **ThrowsError**, que en realidad se comporta como la mónada **Either**:

```
type ThrowsError = Either LispError
```

la línea de arriba está currificada, se podría escribir así también:

```
type ThrowsError b = Either LispError b
```

ThrowsError es, por tanto, una mónada que puede contener **LispError** (en el caso de **Left**) o un tipo b, que en nuestro programa es **LispVal** (en el caso de **Right**). Por tanto, cuando accedemos a su interior, encontraremos un **LispError** o un **LispVal**. Es decir, estamos definiendo un tipo que puede ser del tipo **b**, o bien dar error. Ahora ya nuestro intérprete no trabajarÃ¡ directamente con **LispVal**, sino con **ThrowsError LispVal**. El sentido de todo esto es poder crear valores que describan el error que ha ocurrido, para luego imprimir por pantalla una explicación detallada del mismo. Ahora todas las funciones que puedan dar errores devolverán un **ThrowsError LispVal**, y tendremos que crear ecuaciones que capturen esos errores y construu-

yan un **LispError** apropiado para posteriormente informar al usuario.

```
readExpr :: String -> ThrowsError LispVal
eval :: LispVal -> ThrowsError LispVal
showVal :: LispVal -> String
```

Either es una mónada en la cual **bind (>>=)** para su ejecución cuando encuentra un **Left**, devolviendo ese **Left** y ahorrando mucho tiempo de computación.

La mónada **Either** también provee otras dos funciones a parte de las monádicas estándar:

throwsError recibe un valor de tipo **Error** y lo eleva al constructor **Left** (error) de un **Either**.

Se usa **throwsError** porque en realidad, no existe el constructor de valor **LispError**, sino que es un constructor de tipo. Por ello, mediante **throwsError**, creamos un **Left** (el **LispError** concreto que sea), lo cual es un resultado de tipo **ThrowsError LispVal**, el tipo retorno de **readExpr**.

catchError: recibe un valor **Either** (una acción) y si es **Right**, lo devuelve, si es **Left**, le aplica la función que recibe (en este caso está hardcoded, y lo que hace es pasar del **Left** a un valor normal de **LispVal**). El sentido de todo esto es que el **Either** resultado siempre tenga un valor **Right**:

```
trapError action = catchError action (return . show)
```

De este modo lo que hacemos es transformar los errores (**Left**) en su representación como valor de tripo **String** en el contexto de la mónada **Either**.

Ahora que tenemos asegurado que todos los valores van a ser **Right**, hagamos un accessor efectivo:

```
extractValue :: ThrowsError a -> a
extractValue (Right val) = val
```

La función **parse** devuelve un **Either**, que tendremos que manejar según construya un **Left** (ParseError) o **Right** (valor correcto).

Ahora **eval** va a devolver un valor monádico, con lo cual, en vez de **map** debemos usar **mapM**, y usar **return** para encapsular en valores monádicos los resultados de **eval**.

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]:
```

mapM mf xs recibe una función monádica (con tipo **Monad m => (a -> m b)**) y la aplica a cada elemento en la lista **xs**; el resultado es una lista (con elementos del tipo **b**, en este caso) dentro de una mónada. Por tanto **mapM eval args** da como resultado un valor de tipo **ThrowsError [Integer]**.

```
eval :: LispVal -> ThrowsError LispVal
eval val@(String _) = return val
eval val@(Number _) = return val
eval val@(Bool _) = return val
eval (List [Atom "quote", val]) = return val
eval (List (Atom func : args)) = mapM eval args >>= apply func
eval badForm = throwError $ BadSpecialForm "Unrecognized special form" badForm

apply :: String -> [LispVal] -> ThrowsError LispVal
apply func args = maybe (throwError $ NotFunction "Unrecognized primitive function args" func)
                        ($ args)
                        (lookup func primitives)

primitives :: [(String, [LispVal] -> ThrowsError LispVal)]

numericBinop :: (Integer -> Integer -> Integer) -> [LispVal] -> ThrowsError LispVal
numericBinop op           []  = throwError $ NumArgs 2 []
numericBinop op singleVal@[_] = throwError $ NumArgs 2 singleVal
numericBinop op params        = mapM unpackNum params >>= return . Number . foldl1 op

unpackNum :: LispVal -> ThrowsError Integer
unpackNum (Number n) = return n
unpackNum (String n) = let parsed = reads n in
                           if null parsed
                             then throwError $ TypeMismatch "number" $ String n
                             else return $ fst $ parsed !! 0
unpackNum (List [n]) = unpackNum n
unpackNum notNum     = throwError $ TypeMismatch "number" notNum
```

```
main :: IO ()
main = do
  args <- getArgs
  evaled <- return $ liftM show $ readExpr (args !! 0) >>= eval
  putStrLn $ extractValue $ trapError evaled
```

Aquí lo más complicado es saber el tipo de **evaled**:

- 1) $readExpr(args!!0) >>= eval$: readExpr da un **ThrowsError LispVal**, luego bind lo que hace es pasar el **LispVal** a eval, y acaba dando otro **ThrowsError LispVal**.
- 2) **liftM show** sobre la mónada **ThrowsError LispVal** da una **ThrowsError String**.
- 3) hacer **return** sobre una **ThrowsError String** nos devuelve un (**IO (Either ThrowsError String)**, y esto se hace para que al operar con el constructo sigamos teniendo el **ThrowsError String**, que es lo que recibe (**trapError**).

  Recuerda:, si estamos trabajando en un do de una mónada tipo **IO**, el **return** va a envolver el dato en una mónada **IO**, y esto es válido para cualquier mónada.
- 4) **trapError** nos devuelve un **Either** del tipo **Either String**, porque recordemos, **Left** era **LispError**, **Right** era **String**, y **catchError** siempre devuelve **Right**.
- 5) **extractValue** nos devuelve un valor de tipo **String**.

Todas las funciones cuyo primer argumento es **LispVal** deben usar pattern matching para saber qué constructor de valor se ha usado para crear el mencionado **LispVal**.

cons es el operador **(:)** de Haskell, es decir, el que sirve para concatenar un elemento a una lista del tipo de ese elemento.

Ahora definimos un cuantificador existencial (sí, aunque se llame **forall**, no es universal). Esto lo que hace es crear un constructor de valor **AnyUnpacker** que recibe funciones de **Lispval** a **ThrowsError a**, para todo tipo a que sea instancia de la clase de tipos **Eq**:

```
data Unpacker = forall a. Eq a => AnyUnpacker (LispVal -> ThrowsError a)

unpackEquals :: LispVal} -> LispVal -> Unpacker -> ThrowsError Bool
unpackEquals arg1 arg2 (AnyUnpacker unpacker) =
             do unpacked1 <- unpacker arg1
                unpacked2 <- unpacker arg2
                return $ unpacked1 == unpacked2
        catchError (const $ return False)
```

Aquí lo que hacemos es crear un bloque **do** en el cual desempaquetamos los dos argumentos **arg1** y **arg2**, ligándolos a las variables **unpacked1** y **unpacked2**. Luego comprobamos su igualdad (serán casi seguro **LispVal**s) y Los volvemos a meter en la mónada **ThrowsError**.

Ahora entra en juego **catchError**, que, recordemos, lo que hacía es recibir un **Either** y si es **Right**, devolver ese mismo **Either**, si es **Left**, aplicarle la función de la derecha.

Veamos en qué consiste **cons** para saber por qué se ha usado.

```
const           :: a -> b -> a
const x _       =  x
```

Veamos los tipos de cada una de las partes para entenderlo mejor:

```
Prelude Control.Monad.Except> :t (const $ return False)
(const $ return False) :: Monad m => b -> m Bool
Prelude Control.Monad.Except> :t (return False)
(return False) :: Monad m => m Bool
```

Luego lo que está haciendo **const** es permitir una currificación que, da igual lo que reciba esa función (en este caso recibe un **Left** conteniendo el error), devolverá siempre lo primero que recibió, en este caso el resultado de **return False**, que no es otra cosa que un **ThrowsError Bool**.

```
equal :: [LispVal] -> ThrowsError LispVal
equal [arg1, arg2] = do
      primitiveEquals <- liftM or $ mapM (unpackEquals arg1 arg2)
                         [AnyUnpacker unpackNum, AnyUnpacker unpackStr, A
      eqvEquals <- eqv [arg1, arg2]
      return $ Bool $ (primitiveEquals || let (Bool x) = eqvEquals in x)
equal badArgList = throwError $ NumArgs 2 badArgList
```

El tipo de **unpackEquals arg1 arg2** es **Unpacker -¿ThrowsError Bool**, por tanto, **mapM (unpackEquals arg1 arg2)** sobre la lista [**AnyUnpacker unpackNum, AnyUnpacker unpackStr, AnyUnpacker unpackBool**] dará una mónada **ThrowsError** conteniendo una lista de Bool, es decir, **ThrowsError** [**Bool**]. A dicha lista le aplicaremos la función or mediante **liftM**.

Para lidiar con las cláusulas **else** se ha realizado un "hack", cada vez que **eval** se encuentra un **else**, devuelve un **Bool True** para que siempre se ejecute esa expresión.

Capítulo 5

Conclusiones y trabajos futuros

Este capítulo es obligatorio. Toda memoria de Trabajo de Fin de Grado debe incluir unas conclusiones y unas líneas de trabajo futuro

Capítulo 6

Summary and Conclusions

This chapter is compulsory. The memory should include an extended summary and conclusions in english.

6.1. First Section

Capítulo 7

Presupuesto

Este capítulo es obligatorio. Toda memoria de Trabajo de Fin de Grado debe incluir un presupuesto.

7.1. Sección Uno

Tipos	**Descripcion**
AAAA	BBBB
CCCC	DDDD
EEEE	FFFF
GGGG	HHHH

Tabla 7.1: Tabla resumen de los Tipos

Apéndice A

Título del Apéndice 1

A.1. Algoritmo XXX

```
*****************************************************************************
*
* Fichero .h
*
*****************************************************************************
*
* AUTORES
*
*
* FECHA
*
*
* DESCRIPCION
*
*
*****************************************************************************/
```

A.2. Algoritmo YYY

```
/*****************************************************************************
 *
 * Fichero .h
 *
```

```
*******************************************************************************
*
* AUTORES
*
* FECHA
*
* DESCRIPCION
*
*
*******************************************************************************/
```

Apéndice B

Título del Apéndice 2

B.1. Otro apendice: Seccion 1

Texto

B.2. Otro apendice: Seccion 2

Texto

Bibliografía

[1] D. H. Bailey and P. Swarztrauber. The fractional Fourier transform and applications. *SIAM Rev.*, 33(3):389–404, 1991.

[2] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[3] Blas C. Ruiz. *Razonando con Haskell. Un curso sobre programación funcional*. Thomson, 2004.

[4] David D. Spivak. *Category Theory for the Sciences*. MIT Press, 2014.