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, 7 de septiembre 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 7 de septiembre 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. Se utilizan los conceptos clave de la programación funcional como pueden ser la currificación, los plegados de listas, tipos de datos algebraicos, funtores, mónadas y transformadores monádicos.

El rendimiento del intérprete es bastante bueno, y el código si bien tiene algunas partes un poco oscuras para que sea más paramétrico y resumido, en general se entiende bastante bien.

Hay una versión web del proyecto en `francisconebrera.com/interpreter`*, y también una versión de escritorio del presente Trabajo Fin de Grado que puede descargarse en* `https://github.com/freinn/libroshaskell/tree/master/tfg`*.*

Asimismo, como parte de este Trabajo Fin de Grado, se ha elaborado un tutorial de Haskell en el que explico el lenguaje y la programación funcional desde lo más básico y se puede encontrar en `https://github.com/freinn/libroshaskell/blob/master/tutorial_de_haskell/Tutorial_de_Haskell.pdf`*.*

Abstract

The main purpose of this Final Degree Project is to create a compiler and interpreter for the Scheme language, using Haskell. The parsing phase of the compiler was implemented using Parsec. The program uses key concepts in functional programming like currying, list folding, algebraic data types, functors, monads and monad transformers.

The interpreter performance is good, and the code is well-design and easy to read. However there are a few parts which are more obscure due to its parametric design which makes it shorter.

There is a web version at `http://francisconebrera.com/interpreter` and a desktop version of this Final Degree Project that you can download at `https://github.com/freinn/libroshaskell/tree/master/tfg`.

You can also find my tutorial of the Haskell programming language, which covers a lot of content, at `https://github.com/freinn/libroshaskell/blob/master/tutorial_de_haskell/Tutorial_de_Haskell.pdf`.

Índice general

1. Introducción **1**
1.1. Idoneidad de Haskell para la creación de compiladores . 1
1.2. Reconocimiento de patrones y tipos de datos algebraicos 2
1.3. Variables de tipo 5
1.4. Lambdas 6
1.5. Plegados de listas 7
1.6. El lenguaje Scheme 8

2. Estado del arte **10**

3. Una herramienta de Haskell para la construcción de compiladores: el módulo Parsec **12**
3.1. La librería Parsec: descripción y exposición de un parser ilustrativo: 12
3.2. Un ejemplo completo: uso de Parsec para parsear JSON 13

4. Funcionamiento del Compilador e Intérprete **23**
4.1. Estructura general del programa 23
4.2. Evaluación de expresiones: 28
4.3. Manejo de errores: 30
4.4. Evaluación: segunda parte: 33
4.5. Construyendo un REPL: 37
4.6. Añadiendo variables y asignación: 38
4.7. Definiendo funciones de Scheme: 42

5. Conclusiones y trabajos futuros **45**

6. Summary and Conclusions **47**

Bibliografía **49**

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.

1.1. Idoneidad de Haskell para la creación de compiladores

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. Final-

mente, 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. A lo largo de la presente memoria se explica el procedimiento de implementación de un compilador e intérprete del lenguaje Scheme.

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.

1.2. 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.

1.3. 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.

1.4. 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.

1.5. 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`.

1.6. El lenguaje Scheme

El lenguaje Scheme es un dialecto de Lisp (LISt Processing Language). Es el lenguaje usado en el libro que para mucha gente es el mejor libro sobre ciencias de la computación jamás escrito: "Structure and Interpretation of Computer Programs". Es un lenguaje minoritario pero muy aclamado por sus usuarios.

El lenguaje es prefijo, y por ello es fácil de parsear, y las funciones son ciudadanos de primera clase. Además, las listas se tratan de manera idéntica a la declaración de funciones (de hecho, todo programa

Scheme es una función, en concreto una lista). Si bien no es puramente funcional, ya que cuenta con constructos imperativos, sí permite muchas técnicas de programación funcional. Además, los lenguajes derivados de Lisp, como el más actualmente usado, Clojure, tienen un gran reconocimiento por parte de sus usuarios y son lenguajes bajo mi punto de vista de gran calidad.

Capítulo 2

Estado del arte

La programación funcional es un tema candente a día de hoy por sus aplicaciones en la programación paralela, la facilidad para crear lenguajes de dominio específico, la gran modularidad, reusabilidad de las funciones ya definidas y la menor propensión a errores de los programas funcionales.

Haskell es muy buen lenguaje de programación, y es puramente funcional. Cuando empecé a programar en Haskell su ecosistema era bastante peor que el actual. La instalación de cabal (su gestor de paquetes) era algo más tediosa y tendente al temido "cabal hell", el momento en el cual toca borrar su carpeta y empezar a instalar las librerías de cero y en otro orden. A eso se le añade el hecho de que en ghc 7.10 se decidió hacer Applicative superclase (de tipos) de Monad, y esto provocó una situación de code breaking en todo el código antiguo que no tuviera instancia de Applicative para sus tipos instancia de Monad.

Haskell ha sido elegido por ser idóneo para la construcción de compiladores e intérpretes, y ha facilitado mucho la labor de diseño. Su principal fortaleza radica en su parecido con el lenguaje Scheme, con el cual tiene diferencias importantes pero en esencia ambos se basan en el cálculo lambda y es sencillo crear un DSL en Haskell que represente a los tipos de Scheme. Además, definir funciones en función de la entrada para que produzcan una salida, cosa que se suele hacer en la

cabecera de la misma función, es casi una guía automática para hacer un intérprete de Scheme.

Scheme es un dialecto de Lisp muy aclamado por aquellos que lo usan, es un lenguaje que usa la notación prefija, lo cual también facilita las labores de parseo e interpretación. Asimismo, en Lisp (y por ello también en Scheme), no hay diferencia en cómo se representan las funciones y las estructuras de datos, puesto que, recordemos, las funciones en un lenguaje funcional son ciudadanos de primera clase, y por ello se tratan como cualquier otro dato. Esto hace la reutilización de código bastante más viable y recurrente.

El lenguaje es cada vez más cómodo, existen gran cantidad de librerías de parsers (en las que destaca Parsec), librerías matemáticas y frameworks para crear páginas web, entre otras muchas utilidades.

El paradigma era relativamente nuevo para mí, había hecho alguna práctica en Prolog, pero no tiene demasiado que ver con Haskell, aunque lo parezca a priori. Cierto es que queda patente desde el primer momento que es un lenguaje idóneo para hacer intérpretes y compiladores, por las características ya mencionadas: reconocimiento de patrones, tipos de datos algebraicos, funtores, mónadas, plegados de listas, currificación y lambdas.

Considero que la programación funcional está en auge, y es un buen momento para aprenderla y profundizar en ella para así poder gozar de sus numerosas ventajas en las aplicaciones en las cuales más brilla, como los compiladores.

Capítulo 3

Una herramienta de Haskell para la construcción de compiladores: el módulo Parsec

3.1. La librería Parsec: descripción y exposición de un parser ilustrativo:

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 utlizado el módulo de Haskell Parsec, que permite crear parsers complejos a partir de otros más sencillos, combinándolos.

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, y que si lo desea puede descargar en `https://github.com/freinn/libroshaskell/tree/master/tfg/JSON_parser`.

3.2. Un ejemplo completo: uso de Parsec para parsear JSON

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 "come" todos los espacios, tabuladores o newlines que encuentre después 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 4

Funcionamiento del Compilador e Intérprete

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 mónadas tienen fama de algo más que abstracto, muy difícil de entender, y estoy de acuerdo en que no son sencillas, pero las intentaré expli-

car 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
```

4.2. Evaluación de expresiones:

```
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.

4.3. 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 trabajará 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 construyan 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 una función. Si el valor **Either** 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
```

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**.

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

4.4. Evaluación: segunda parte:

boolBinop es una función definida para ser currificada, y por ello es muy versátil:

```
boolBinop :: (LispVal -> ThrowsError a) -> (a -> a -> Bool) -> [LispVal] -> ThrowsError LispVal
boolBinop unpacker op args = if length args /= 2
                             then throwError $ NumArgs 2 args
                             else do left <- unpacker $ args !! 0
                                     right <- unpacker $ args !! 1
                                     return $ Bool $ left `op` right
```

Gracias a ella podemos usar una función arbitraria que pase de **LispVal** a **ThrowsError a** y con ello crear una plantilla para una operación binaria entre los tipos que queramos:

```
numBoolBinop  = boolBinop unpackNum
strBoolBinop  = boolBinop unpackStr
boolBoolBinop = boolBinop unpackBool
```

Esto nos permite ampliar nuestra lista de primitivas más cómodamente:

```
("=", numBoolBinop (==)),
("<", numBoolBinop (<)),
(">", numBoolBinop (>)),
("/=", numBoolBinop (/=)),
(">=", numBoolBinop (>=)),
("<=", numBoolBinop (<=)),
("&&", boolBoolBinop (&&)),
("||", boolBoolBinop (||)),
("string=?", strBoolBinop (==)),
("string<?", strBoolBinop (<)),
("string>?", strBoolBinop (>)),
("string<=?", strBoolBinop (<=)),
("string>=?", strBoolBinop (>=)),
```

Ahora empieza lo interesante, implementemos condicionales sencillos, aún sin else:

```
eval (List [Atom "if", pred, conseq, alt]) =
     do result <- eval pred
        case result of
             Bool False -> eval alt
             otherwise  -> eval conseq
```

car de Scheme es como **head** de Haskell, y su implementación se basa en el reconocimiento de patrones:

```
car :: [LispVal] -> ThrowsError LispVal
car [List (x : xs)]         = return x
car [DottedList (x : xs) _] = return x
car [badArg]                = throwError $ TypeMismatch "pair" badArg
car badArgList              = throwError $ NumArgs 1 badArgList
```

cdr de Scheme es como **tail** de Haskell:

```
cdr :: [LispVal] -> ThrowsError LispVal
cdr [List (x : xs)]         = return $ List xs
cdr [DottedList [_] x]      = return x
cdr [DottedList (_ : xs) x] = return $ DottedList xs x
cdr [badArg]                = throwError $ TypeMismatch "pair" badArg
cdr badArgList              = throwError $ NumArgs 1 badArgList
```

cons es el operador **(:)** de Haskell, es decir, el que sirve para concatenar un elemento a una lista del tipo de ese elemento. Si aplicamos **cons** a una lista que sólo contenga elementos, obtendremos una **DottedList**:

```
cons :: [LispVal] -> ThrowsError LispVal
cons [x1, List []] = return $ List [x1]
cons [x, List xs] = return $ List $ x : xs
cons [x, DottedList xs xlast] = return $ DottedList (x : xs) xlast
cons [x1, x2] = return $ DottedList [x1] x2
cons badArgList = throwError $ NumArgs 2 badArgList
```

Lo siguiente es definir una función que establezca un criterio de igualdad entre valores. Esto se hace recibiendo una lista con dos elementos, los valores a comparar (los cuales también pueden ser listas). Lo primero es comprobar que los dos valores son del mismo tipo, ya que estamos implementando una comparación fuerte. Si esto no ocurre, se devuelve falso. Si los dos valores son del mismo tipo, se les aplica la función **(==)** de Haskell:

```
eqv :: [LispVal] -> ThrowsError LispVal
eqv [(Bool arg1), (Bool arg2)]             = return $ Bool $ arg1 == arg2
eqv [(Number arg1), (Number arg2)]         = return $ Bool $ arg1 == arg2
eqv [(String arg1), (String arg2)]         = return $ Bool $ arg1 == arg2
eqv [(Atom arg1), (Atom arg2)]             = return $ Bool $ arg1 == arg2
eqv [(DottedList xs x), (DottedList ys y)] = eqv [List $ xs ++ [x], List $ ys ++ [y]]
eqv [(List arg1), (List arg2)]             = return $ Bool $ (length arg1 == length arg2) &&
                                                             (all eqvPair $ zip arg1 arg2)
     where eqvPair (x1, x2) = case eqv [x1, x2] of
                                Left err -> False
                                Right (Bool val) -> val
eqv [_, _]                                 = return $ Bool False
eqv badArgList                             = throwError $ NumArgs 2 badArgList
```

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**. Para usarlo debemos añadir a la cabecera de nuestro programa el pragma **{-# LANGUAGE ExistentialQuantification #-}**:

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

Aunque parezca muy extraño y novedoso, indagando un poco en por qué se le llama existencial nos daremos cuenta de que todo cobra sentido; esto es un constructor de tipos como otro cuaquiera, pero estamos obligando a que el tipo **a** sea instancia de la clase de tipos **Eq**. Luego, si y sólo si existe la instancia para **Eq** del tipo que queramos instanciar como tipo **Unpacker**, el compilador nos permitirá instanciarlo. Estamos ordenando nuestro código y evitando errores en tiempo de ejecución a costa de generar potenciales errores en tiempo de compi-

lación, puro Haskell.

Nuestra intención es tener también un modo "débil"de comparar, y para ello lo que haremos será probar uno a uno todos nuestros unpackers, y desde el momento en que uno de ellos devuelva **True**, eso mismo devolveremos, y si en cambio todos devuelven **False**, entonces devolveremos **False**.

Lo primero que implementaremos será un helper que determinará si dos **LispVal** son iguales, usando un **Unpacker**, es decir, un desempaquetador arbitrario. 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**.

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

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 la definición de la función **const**:

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

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

```
(const $ return False) :: Monad m => b -> m Bool
(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, AnyUnpacker unpackBool]
      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**.

4.5. Construyendo un REPL:

Un REPL (read-eval-print loop) es un bucle de lectura-evaluación-impresión, es decir, nos permite usar nuestro programa como si de una consola se tratara, y es así como funcionan la mayoría de intérpretes pasados y actuales. Lo primero que haremos será crear un helper que imprima por pantalla una cadena (la salida que da el intérprete al evaluar algo) y limpie después el flujo:

```
flushStr :: String -> IO ()
flushStr str = putStr str >> hFlush stdout
```

A continuación usaremos la función recientemente definida para imprimir una cadena por pantalla, limpiar el flujo y posteriormente pedir una línea al usuario.

```
readPrompt :: String -> IO String
readPrompt prompt = flushStr prompt >> getLine
```

Sacamos de **main** el código para parsear y evaluar una cadena y capturar los errores y lo ponemos en su propia función:

```
evalString :: Env -> String -> IO String
evalString env expr = runIOThrows $ liftM show $ (liftThrows $ readExpr expr) >>= eval env
```

Ahora creamos una función que evalúe una cadena e imprima el resultado:

```
evalAndPrint :: Env -> String -> IO ()
evalAndPrint env expr =  evalString env expr >>= putStrLn
```

Necesitamos un bucle con una condición de parada (una función, Â¡qué sorpresa!), pero como en la programación funcional no existen los bucles, lo que hacemos para simular uno es llamar recursivamente a la función que definimos, **until_**, a menos que la condición de parada se cumpla, en cuyo caso devolveremos un valor de tipo tupla vacía **()**.

```
until_ :: Monad m => (a -> Bool) -> m a -> (a -> m ()) -> m ()
until_ pred prompt action = do
   result <- prompt
   if pred result
      then return ()
      else action result >> until_ pred prompt action
```

Ahora que tenemos todas las piezas, sólo nos queda unirlas:

```
runRepl :: IO ()
runRepl = primitiveBindings >>= until_ (== "quit") (readPrompt "Lisp>>> ") . evalAndPrint
```

Ahora adaptamos nuestra función **main** para que si el programa no recibe argumentos cuando se ejecuta se ponga en marcha el intérprete, y si recibe un argumento (la expresión a ejecutar) la evalúe y salga:

```
main :: IO ()
main = do args <- getArgs
          if null args then runRepl else runOne $ args
```

4.6. Añadiendo variables y asignación:

Usaremos el modo de hilos de estado de Haskell, llamado **IORef**. Este módulo se usa en medio de la mónada IO y sirve para usar variables mutables sin esfuerzo en Haskell. Necesitamos una mónada que contenga una lista que asocie cadenas con mónadas que contengan valores

de tipo **LispVal**. Esto es así porque hay dos maneras de modificar un entorno:

- la función **set!** cambia el valor de una variable de la lista
- la función **define** añade una nueva variable a la lista

```
import Data.IORef

type Env = IORef [(String, IORef LispVal)]
```

Definimos una función que cree un entorno vacío:

```
nullEnv :: IO Env
nullEnv = newIORef []
```

Ahora definiremos un transformador monádico, es decir, algo que combina varias mónadas. Es una especie de **Either** currificado que puede contener dos mónadas, **LispError** e **IO**, es decir, encapsulamos una computación IO la posibilidad de que produzca error:

```
type IOThrowsError = ErrorT LispError IO
```

Como los bloques do no nos permiten usar mónadas de distintos tipos y tenemos muchas funciones que hacen uso de **ThrowsError**, debemos definir un lifter que pase de una mónada a la otra:

```
liftThrows :: ThrowsError a -> IOThrowsError a
liftThrows (Left err) = throwError err
liftThrows (Right val) = return val
```

También queremos un helper que ejecute la acción **IOThrowsError**, ejecute la computación del error y capture los errores, devolviendo una acción **IO**. Esto usa **trapError** para convertir los valores de error en su representación de cadena, y luego ejecuta la computación mediante **runErrorT**. Luego se extrae el valor y se envuelve en la mónada **IO**:

```
runIOThrows :: IOThrowsError String -> IO String
runIOThrows action = runErrorT (trapError action) >>= return . extractValue
```

Se muestra el tipo de **readIORef** por claridad. Definimos ahora una función que devuelva un booleano que indique si una variable está li-

gada o no:

```
readIORef :: IORef a -> IO a

isBound :: Env -> String -> IO Bool
isBound envRef var = readIORef envRef >>= return . maybe False (const True) . lookup var
```

Ahora repetimos la estrategia pero para obtener un valor de tipo **LispVal** dentro de la mónada **IOThrowsError**:

```
liftIO :: IO a -> m a

getVar :: Env -> String -> IOThrowsError LispVal
getVar envRef var  =  do env <- liftIO $ readIORef envRef
                         maybe (throwError $ UnboundVar "Getting an unbound variable" var)
                               (liftIO . readIORef)
                               (lookup var env)
```

Ahora definiremos una función para cambiar el valor de una variable. Es muy parecida a la anterior pero hace uso de dos funciones nuevas:

- **flip**: invierte el orden de los argumentos de una función.
- **writeIORef**: escribe un valor en el **IORef**.

```
setVar :: Env -> String -> LispVal -> IOThrowsError LispVal
setVar envRef var value = do env <- liftIO $ readIORef envRef
                             maybe (throwError $ UnboundVar "Setting an unbound variable" var)
                                   (liftIO . (flip writeIORef value))
                                   (lookup var env)
                             return value
```

Una función algo más interesante es aquella capaz de definir una nueva variable. Para ello se ha de crear un nuevo **IORef** con el valor que tendrá en principio, luego crear un par (nombreVariable, valorIORef) para añadirlo al principio de la lista (esto se hace por eficiencia):

```
defineVar :: Env -> String -> LispVal -> IOThrowsError LispVal
defineVar envRef var value = do
     alreadyDefined <- liftIO $ isBound envRef var
     if alreadyDefined
        then setVar envRef var value >> return value
        else liftIO $ do
             valueRef <- newIORef value
             env <- readIORef envRef
             writeIORef envRef ((var, valueRef) : env)
             return value
```

La siguiente función enriquece un entorno con una lista de variables y valores, y para ello usa una tubería monádica y dos helpers:

- **extendEnv** añade al principio de un entorno una lista de pares
- **addBinding** recibe un par (nombre, valor) y devuelve un par adaptado para ser añadido a un entorno.

```
bindVars :: Env -> [(String, LispVal)] -> IO Env
bindVars envRef bindings = readIORef envRef >>= extendEnv bindings >>= newIORef
     where extendEnv bindings env = liftM (++ env) (mapM addBinding bindings)
           addBinding (var, value) = do ref <- newIORef value
                                        return (var, ref)
```

Añadimos después de la ecuación para if las siguientes dos ecuaciones:

```
eval env (List [Atom "set!", Atom var, form]) =
     eval env form >>= setVar env var
eval env (List [Atom "define", Atom var, form]) =
     eval env form >>= defineVar env var
```

Actualizamos el resto de funciones de evaluación:

```
evalAndPrint :: Env -> String -> IO ()
evalAndPrint env expr =  evalString env expr >>= putStrLn
```

```
evalString :: Env -> String -> IO String
evalString env expr = runIOThrows $ liftM show $ (liftThrows $ readExpr expr) >>= eval env
```

Adaptamos ahora nuestras funciones de ejecución para que empiecen con el entorno vacío:

```
runOne :: String -> IO ()
runOne expr = nullEnv >>= flip evalAndPrint expr

runRepl :: IO ()
runRepl = nullEnv >>= until_ (== "quit") (readPrompt "Lisp>>> ") . evalAndPrint
```

Finalmente cambiamos nuestra función **main** para que sólo funcione con un argumento o ninguno.

```
main :: IO ()
main = do args <- getArgs
          case length args of
               0 -> runRepl
               1 -> runOne $ args !! 0
               otherwise -> putStrLn "Program takes only 0 or 1 argument"
```

4.7. Definiendo funciones de Scheme:

Lo primero que debemos hacer es añadir nuevos constructores de valor al tipo **LispVal**. El constructor **PrimitiveFunc** guarda una función que recibe una lista de argumentos y devuelve un **ThrowsError LispVal**, teniendo el mismo tipo que nuestra lista de primitivas. El constructor **Func**, sin embargo, usa lo que se conoce en Haskell como **record syntax**, es decir, que definimos los tipos que compondrán nuestro tipo y se crearán automáticamente las funciones para acceder a ellos (como vemos, la declaración de los tipos se parece mucho a una cabecera de función). **Func** sirve para guardar funciones definidas por el usuario y se construye con cuatro elementos:

- los nombres de los argumentos, como si estuvieran ligados al cuerpo de la funcion.
- si la función acepta o no una lista de argumentos de tamaño variable, y si lo hace, el nombre de la variable que la guarda.
- el cuerpo de la función, como una lista de expresiones.
- el entorno en el que la función fue creada.

```
| PrimitiveFunc ([LispVal] -> ThrowsError LispVal)
| Func { params :: [String], vararg :: (Maybe String),
        body :: [LispVal], closure :: Env }
```

Lo primero es comprobar el largo de la lista de parámetros contra el número esperado de argumentos. Luego se usa una tubería monádica que enlaza los argumentos a un nuevo entorno y ejecuta las instrucciones del cuerpo. La primera cosa que hacemos es usar **zip**, es decir, unir dos listas en una lista de pares del tipo (1er argumento de la primera lista, 1er argumento de la segunda lista).Con esta lista y la clausura creamos el entorno en el cual se ejecutará la función. Luego se enlazan los argumentos restantes a la variable **varArgs**, mediante la función **bindVarArgs**. Si esta variable vale **Nothing**, entonces devolvemos el entorno exitente. Si no, creamos una lista que tenga el nombre de la variable como el valor, y se la pasamos a **bindVars**. Definimos la

variable local **remainingArgs** por limpieza, usando la función **drop** para ignorar los argumentos que ya han sido ligados a variables. Lo último es evaluar el cuerpo en este nuevo entorno. Usamos para ello la función **evalBody**, que mapea la función monádica **eval env** sobre cada expresión del cuerpo, y luego devuelve el valor de la última expresión:

```
apply :: LispVal -> [LispVal] -> IOThrowsError LispVal
apply (PrimitiveFunc func) args = liftThrows $ func args
apply (Func params varargs body closure) args =
      if num params /= num args && varargs == Nothing
         then throwError $ NumArgs (num params) args
         else (liftIO $ bindVars closure $ zip params args) >>= bindVarArgs varargs >>= evalBody
      where remainingArgs = drop (length params) args
            num = toInteger . length
            evalBody env = liftM last $ mapM (eval env) body
            bindVarArgs arg env = case arg of
                Just argName -> liftIO $ bindVars env [(argName, List $ remainingArgs)]
                Nothing -> return env
apply (IOFunc func) args = func args
```

Para que nuestro intérprete sea más cómodo y parecido a uno real, ligamos las primitivas que tenemos hardcodeadas como funciones de Haskell pero con el nombre de las de Scheme al iniciar el programa. Además queremos que esto se haga siempre, por ello llamamos a esta función en la ejecución de **runOne** y **runRepl**:

```
primitiveBindings :: IO Env
primitiveBindings = nullEnv >>= (flip bindVars $ map makePrimitiveFunc primitives)
     where makePrimitiveFunc (var, func) = (var, PrimitiveFunc func)

runOne :: String -> IO ()
runOne expr = primitiveBindings >>= flip evalAndPrint expr

runRepl :: IO ()
runRepl = primitiveBindings >>= until_ (== "quit") (readPrompt "Lisp>>> ") . evalAndPrint
```

Definimos un helper que nos permita crear funciones en general, y luego creamos dos funciones que son una currificación de **makeFunc** (la función más general) para crear funciones con un número fijo o variable de argumentos:

```
makeFunc varargs env params body = return $ Func (map showVal params) varargs body env
makeNormalFunc = makeFunc Nothing
makeVarArgs = makeFunc . Just . showVal
```

Finalmente añadimos a **eval** nuevas ecuaciones que nos permitan definir funciones y lambdas, y cambiamos el modo en el que se aplican las funciones (última ecuación de **eval**):

```
eval env (List (Atom "define" : List (Atom var : params) : body)) =
     makeNormalFunc env params body >>= defineVar env var
eval env (List (Atom "define" : DottedList (Atom var : params) varargs : body)) =
     makeVarArgs varargs env params body >>= defineVar env var
eval env (List (Atom "lambda" : List params : body)) =
     makeNormalFunc env params body
eval env (List (Atom "lambda" : DottedList params varargs : body)) =
     makeVarArgs varargs env params body
eval env (List (Atom "lambda" : varargs@(Atom _) : body)) =
     makeVarArgs varargs env [] body
eval env (List (function : args)) = do
     func <- eval env function
     argVals <- mapM (eval env) args
     apply func argVals
```

Ya tenemos listo un compilador e intérprete del lenguaje Scheme.

Capítulo 5

Conclusiones y trabajos futuros

La programación funcional es un paradigma en auge ahora mismo, sobre todo por sus aplicaciones en paralelismo y matemáticas, así como por su capacidad para que los programas sean casi inmunes a ataques de fuzzing, y por dar lugar a programas menos propensos a errores por parte del programador.

Sin embargo, el aprendizaje de la programación funcional es, sobre todo, lento, y muy relacionado con el álgebra abstracta, el lambda cálculo y la teoría de las categorías. Es el paradigma más abstracto que conozco y requiere una cabeza amueblada para tal fin, por ello lo veo más orientado (al menos a priori) a matemáticos que a ingenieros informáticos.

Los programas funcionales son muy, muy modulares, en el sentido de que hay gran cantidad de funciones definidas en ellos (de hecho en la programación funcional no se puede hacer otra cosa), y esto permite que sean poco propensos a errores, pues las funciones no comparten estado, se limitan a recibir entradas y producir salidas (esto se conoce como transparencia referencial o "pureza"). El programador puede entonces limitarse a pensar cómo a partir de una entrada producir una

salida, y esto lleva en muchos casos a generalizaciones y abstracciones bastante más potentes que las que se consiguen en otros paradigmas.

En el caso de los compiladores e intérpretes, un lenguaje como Haskell es muy potente. Los tipos de datos algebraicos nos permiten construir valores para luego aplicar reconocimiento de patrones, creando lenguajes de dominio específico muy fácilmente. Esto se hace especialmente idóneo para interpretar lenguajes funcionales como Scheme. En este caso, contamos con múltiples funciones y estructuras de datos que se comportan como las de Scheme, y nos aprovechamos de ello.

En cuanto al rendimiento, es muy bueno, y puede trabajar con enteros inmensamente grandes para calcular, por ejemplo, factoriales enormes. El código es fácilmente adaptable y se le pueden añadir nuevas funciones o ecuaciones a funciones ya existentes con facilidad.

Creo que la programación funcional es muy válida para según qué tareas, y no tanto para otras, por ello, los lenguajes con mayor proyección hoy en día son, al menos para mí, Javascript y Rust. Estos dos lenguajes son mixtos (tienen parte imperativa y funcional) y es el camino que creo que tomarán los lenguajes en el futuro. La programación funcional sirve para enfocar los problemas de otra manera, y abre la mente a los programadores imperativos.

Como líneas de trabajo futuro, se podrían usar las ideas del código ya hecho para hacer un compilador de Scheme a C, usando también Haskell u otro lenguaje funcional. Asimismo, se podría implementar el estado del programa mediante el combinador punto fijo. También se podrían añadir las características que se proponen en el tutorial en el cual me he basado, como por ejemplo macros (preprocesar el código y hacerle modificaciones antes de que se ejecute), o continuaciones.

Capítulo 6

Summary and Conclusions

Functional programming is a paradigm booming right now, especially for its applications in parallel computing and mathematics, as well as its capacity to ensure that programs are almost inmune to fuzzing attacks, and less likely to lead errors.

However, learning functional programming is, above all, slow, and closely related to abstract algebra, the lambda calculus and category theory. It is the most abstract paradigm I know and requires a head fitted for this purpose, therefore I see more targeted (at least a priori) to mathematicians to computer engineers.

Functional programs are very, very modular, in the sense that there are a lot of functions defined in them (in fact in functional programming you can't do anything else), and this allows them to be little prone to errors, because functions don't share state, cause functions are limited to receive inputs and produce outputs (this is known as referential transparency or purity). The programmer can then merely thinking how from an input he will produce an output, and in many cases this leads to generalizations and abstractions far more powerful than those that are available in other paradigms abstractions.

For compilers and interpreters, a language like Haskell is very powerful. Algebraic data types allow us to build values and then apply pattern matching, creating domain specific languages easily. This is specially suited to interpret functional languages like Scheme. In this case, we have multiple definitions and data structures that behave like Scheme, and we took advantage of it.

Performance-wise, it's very good, and can work with inmensely large integers to calculate, for example, huge factorials. The code is easily customizable and you can add new functions or equations to existing functions easily.

I think functional programming is valid for depending on what task, rather than to others, therefore, the languages I think have a better future today are, at least for me, Javascript and Rust. These two languages are mixes (they have imperative and functional features), and is the way I think will take the languages in the future. Functional programming servers to approach problems differently, and opens the mind of imperative programmers.

As future lines of work, they could use the ideas of the code already done to make a Scheme to C compiler, also using Haskell or another functional language. Also, the state could be implemented via the fixed-point combinator. They could also add the features proposed in the tutorial in which I have drawn, such as macros (pre-process the code and make modifications before running), or continuations.

Bibliografía

[1] *tutorial Write Yourself a Scheme in 48 Hours.*

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

[3] Greg Michaelson. *An Introduction to Functional Programming Through Lambda Calculus.* Dover, 2011.

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

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