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, 17 de abril 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.

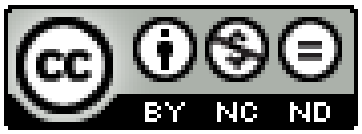
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 17 de abril de 2015

Agradecimientos

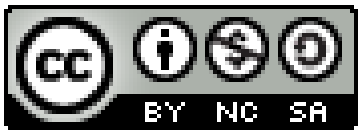
Casiano Rodríguez León
Jorge Riera Ledesma
Luz Marina Moreno de Antonio
Francisco de Sande González
Francisco Carmelo Almeida Rodríguez
Blas C. Ruiz
F. Gutiérrez
P. Guerrero
E. Gallardo
Daniel Díaz Casanueva

Licencia

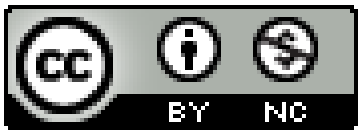
* Si NO quiere permitir que se compartan las adaptaciones de tu obra y NO quieres permitir usos comerciales de tu obra indica:



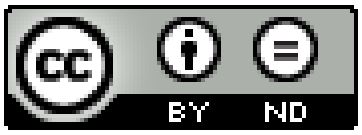
* Si quiere permitir que se compartan las adaptaciones de tu obra mientras se comparta de la misma manera y NO quieres permitir usos comerciales de tu obra indica:



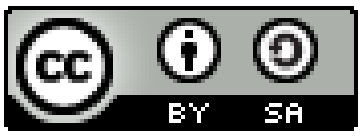
* Si quiere permitir que se compartan las adaptaciones de tu obra y NO quieres permitir usos comerciales de tu obra indica:



*Si NO quiere permitir que se compartan las adaptaciones de tu obra y quieres permitir usos comerciales de tu obra indica:



* Si quiere permitir que se compartan las adaptaciones de tu obra mientras se comparta de la misma manera y quieres permitir usos comerciales de tu obra (licencia de Cultura Libre) indica:



* Si quiere permitir que se compartan las adaptaciones de tu obra y quieres permitir usos comerciales de tu obra (licencia de Cultura Libre) indica:

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", 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 tipos. 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 datos
--  Constructor de tipos
```

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
--               Añadiendo 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

Nota: aunque para que la presentación en LaTeX sea más vistosa se han puesto las flechas como →, en realidad se escriben con un símbolo menos y un símbolo "mayor que" (−>).

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 demasiado necesarios y se va resolviendo 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

Pondremos un ejemplo real codificado por mí, 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 temas 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.

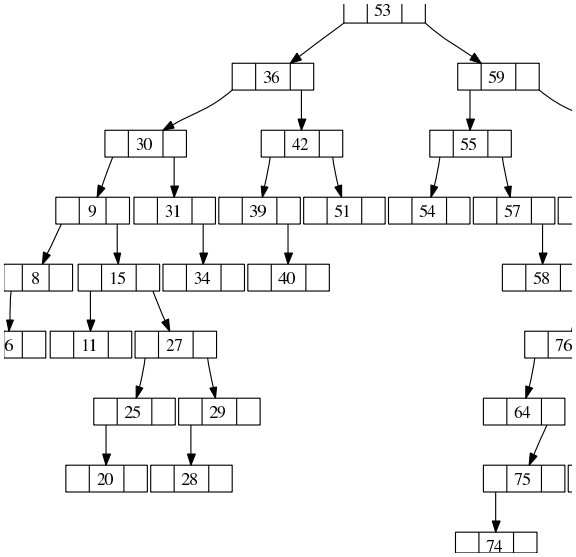


Figura 1.1: Ejemplo

Capítulo 2

Librería 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

2.1. Primer apartado de otro capitulo

Capítulo 3

Título del Capítulo Tres

Bla, Bla, Bla, ...

3.1. Primer apartado de este capitulo

3.2. Segundo apartado de este capitulo

3.3. Tercer apartado de este capitulo

Capítulo 4

Título del Capítulo Cuatro

En el capitulo 1 se describio bla, bla, bla.....

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.