Análisis de Gramáticas a partir de Parser Combinators

Materia: Programación Funcional

Integrantes: Biagini, Martín - 56343

Filipic, Joaquín - 56266

Fecha: 24/05/2020

Índice

1. Introducción

El presente trabajo surge a partir de la intención de poder analizar diversos lenguajes de manera similar. Esto llevó a la alternativa de obtener parsers particulares para cada uno de ellos pero que compartan su estructura de forma genérica.

Por lo tanto, lo primero que se ofrece es un set de funciones a modo de librería para facilitar la construcción de dichos parsers para gramáticas propias. La técnica empleada para ello es la de *Parser Combinator*, que ofrece ciertas ventajas para llevar a cabo dicha tarea.

Se presenta, junto a la librería, una serie de ejemplos de parsers sencillos construidos para evidenciar la facilidad de uso de la misma. Sin embargo, el hincapié se hace en los parsers construidos para 2 gramáticas particulares. Por la funcionalidad para la cual fueron creadas, dichas gramáticas serán llamadas “Gramática de Red” y “Gramática de Base de Datos”. Ambas gramáticas están detalladas.

En adición, se implementa un programa sencillo con interfaz para el usuario, con el cual este puede ejecutar comandos de ambas gramáticas de forma interactiva y evaluar su comportamiento.

Más adelante se indican los pasos necesarios para ejecutar el programa, así como ejemplos de uso detallados de las gramáticas.

2. Parser Combinator

Un combinador de parsers es una función de alto orden que, a través de una estrategia recursiva, acepta múltiples parsers como entrada y devuelve un nuevo parser como salida. Individualmente, cada parser es una función que acepta una cadena de caracteres como entrada y devuelve cierta estructura como salida.

Empleando esta técnica diferente a la de parsers convencionales, puede hacerse una librería genérica que sirva para crear parsers para gramáticas particulares de manera sencilla y legible. Lo interesante de esta técnica es que aprovecha las facilidades del paradigma funcional (en particular del lenguaje Haskell, utilizado en este trabajo), tales como:

- Uso de funciones de alto orden
- Polimorfismo
- *Lazy Evaluation*
- Sistema de Tipos

Concretamente, la librería creada consta de las funciones básicas de combinadores y una extensión, con funciones derivadas diseñadas para cubrir patrones recurrentes en el diseño de parsers (simplificado por el hecho de que, al tratarse siempre de funciones, pueden tanto recibirse como devolverse parsers).

El tipo fundamental del parser se observa en el *Código1*, y a partir de él se van armando las funciones para combinarse. Todo parser consume una lista de símbolos s (en nuestro caso serán siempre *Char*), y devuelve una lista de pares conteniendo la estructura de salida y el resto de la lista sin consumir, en caso de que no se pudiera seguir procesando la entrada (lista de pares porque, si la gramática es ambigua, puede arrojar más de una forma de parseo, pero esto no viene al caso).

```
type Parser s t =([s] -> [(t, [s])])
```

Código1: estructura básica de un parser.

Algunas de las observaciones que pueden hacerse sobre las ventajas de esta aproximación son las siguientes:

- En primer lugar, es esencial el empleo del polimorfismo. Se trabaja de tal manera que los tipos del parser y los tipos del resultado quedan completamente separados y tratados de forma general (tratan tipos genéricos, *a -> b*).

- El sistema de clases de tipos permite definir las interfaces entre el parseo y el resto del programa de manera precisa. Esto facilita la identificación de los tokens válidos para el parseo, las estructuras correctas de salida, etc.
- En lo referente a la sintaxis, al poder definirse operadores de manera infija, las definiciones para cada parser se asemeja a las gramáticas para las cuales están hechas.
- Por otro lado, hay funciones definidas específicamente para evitar guardar de forma innecesaria los resultados parciales de una combinación de parsers, y con ello se ahorra memoria.
- Como con esta técnica se van analizando las posibles alternativas con un parser en combinación con las de otros parsers, puede importar el orden en que se efectúen esos análisis. Hay operadores definidos para evaluar antes un operando que el otro, y así poder evitar cómputo.

Las definiciones para cada parser pueden ser casi un isomorfismo con las definiciones de los tipos de datos usados en la gramática (ejemplo en *Código2*, utilizando la estructura de un árbol binario sencillo). Sin entrar en el detalle de los operadores utilizados en el parser "*pTree*" (*<|>*, *<$$>* y *<**>*), se puede observar la similitud entre ambas definiciones: por un lado, se hace una división para parsear una hoja (*Leaf*) o parsear una rama binaria (*Bin*). En el caso de una hoja, se usa el parser "*pChar*" para parsear un *Char*, porque así está indicado en el primer constructor, y en caso de tratarse de una rama, se indica que deben parsearse a su vez 2 árboles más, como lo indica el segundo constructor de su definición.

De esta forma, puede seguir extendiéndose la librería de forma ilimitada. También puede optarse por ir evaluando el parseo a medida que se lleva a cabo, si fuera sencilla esa evaluación, y devolver cierto valor final como estructura de salida.

```
data Tree = Leaf Char
          | Bin Tree Tree
          deriving Show

pTree :: Parser Char Tree
pTree =       Leaf <$$> pChar
      <|>  pParens (Bin <$$> pTree <**> pTree)
```

Código2: se muestra la construcción de un parser que deja a su salida una estructura del tipo "Tree", definida de manera análoga.

Como mención, quedan implementaciones que pueden llevarse a cabo para ofrecer más riqueza a la librería. La más destacable, tal vez, es la detección y corrección de errores en el parseo. Sin embargo, como de cualquier forma se depende del *backtracking* para obtener

distintas alternativas de parseo, tomar ese camino implica tipos de datos más complejos, almacenamiento de información contextual del parseo y un considerable *overhead* de cómputo como precio a pagar.

Por lo tanto, los parsers creados con esta librería devuelven una estructura vacía en caso de que no haya podido encontrarse un parseo válido para la cadena de entrada.

3. Gramáticas

La idea principal para hacer uso de la librería es crear gramáticas libres de contexto (las reglas de producción se efectúan reemplazando símbolos no terminales por una sucesión de símbolos terminales, sin importar su contexto) no ambiguas y que no tengan el problema de recursión izquierda, que es un requisito. Dicho eso, los casos presentados están orientados a tratar a temas particulares.

3.1 Gramática de Red

La primera gramática definida tiene como objetivo ofrecer una introducción al pensamiento imperativo: de manera interactiva, el usuario puede hacer uso de las instrucciones permitidas por el lenguaje para poder llegar a un objetivo secuencialmente.

El marco semántico en el cual se presenta gráficamente el uso del lenguaje es un mapa (que representa una suerte de *network*) en el cual un *host* debe enviar un paquete con un mensaje a otro *host* pasando sí o sí por todos los *routers* intermedios, para lo cual el usuario puede pensar en cómo usar loops y procedimientos para simplificar la resolución. Las operaciones se realizan sobre el paquete a enviar, y las posiciones por las cuales puede pasar dentro del mapa están basadas en coordenadas (se debe mover el paquete posición por posición dentro del mapa de 2 dimensiones).

La gramática está definida de la siguiente forma:

```
<instruction>      ::= <instruction> <separator> <instruction>
                     | <decimal_number>? <movement>
                     | Translate
                     | <decimal_number> '*' (<instruction>)
                     | <word> = (<instruction>)
                     | <word>

<separator>        ::= '+' | \n
<decimal_number>   ::= <non_zero_digit> <digit>*
<digit>            ::= 0 | <non_zero_digit>
<non_zero_digit>   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<movement>         ::= Left | Right | Front
<word>             ::= <letter>+
<letter>           ::= [a-zA-Z]
```

Toda instrucción puede tratarse como una concatenación de instrucciones: esto simplemente para poder agrupar comandos y tener que ejecutar uno por uno. Además, el desafío se presenta al tratar de resolver cada mapa pensando de antemano los pasos a seguir y mandar a ejecutarlos todos a la vez.

Pasando a las operaciones propiamente dichas, para efectuar un movimiento, se cuenta con las instrucciones de rotación a izquierda y derecha (ya que existen 4 direcciones, además de la posición actual) y de avance al frente. Estas pueden estar precedidas por un número natural que indica cuántas veces efectuar el movimiento. La última a mencionar, es una función de operación genérica ("*Translate*"), que debe usarse cuando el paquete pasa por un router, ya que sin ello no puede completarse el mapa.

También se pueden indicar *loops*, es decir, agrupar un set de instrucciones y repetirlas la cantidad de veces que se desee. Finalmente, para aportar mayor versatilidad, se pueden definir procedimientos y ejecutarlos las veces requeridas: se asocia un set de instrucciones cualesquiera a un nombre para definirlo, y luego, cuando se explicite ese nombre, se reemplazará por esas instrucciones.

3.2 Gramática de Base de Datos

La finalidad de esta segunda gramática es popular una especie de diccionario con entradas clave-valor, en el cual se asocian ideas de cualquier tipo (representadas por palabras, strings). La idea es una sugerencia a la base de funcionamiento de los lenguajes lógicos, en donde se van formalizando declaraciones y se permite hacer consultas sobre ellas.

La definición es la siguiente:

```
<instruction>       ::=  define <word> as <list_of_words>
                         | undefine <word> as <list_of_words>
                         | delete <word>
                         | list
                         | <query>

<list_of_words>     ::= <word>(,<word>)*
<word>              ::= <letter>+
<letter>            ::= [a-zA-Z]
<query>             ::=  <constant>
                         | type of <word>
                         | exists <word>
                         | is <word> of type <word>
                         | not <query>
```

| (<query> <operator> <query>)

<constant> ::= *True* | *False*
<operator> ::= *or* | *and* | *then* | *xor* | *iff*

Para hacer una diferenciación, se tienen comandos de declaración y comandos de consulta. En cuanto a las declaraciones, se puede definir la asociación de una palabra a una serie de ideas, eliminar esas asociaciones individualmente, o directamente eliminar todo el registro de esa idea. Esto para popular la base.

Luego se cuenta con todas las operaciones de consulta, que permiten preguntar qué asociaciones tienen las palabras (o si tiene una en particular), si tiene asociaciones o no, y la oportunidad de emplear los conectores lógicos *o, y, o exclusivo, entonces, si y sólo si*. Así esto se puede experimentar con un amplio abanico de consultas para poder decidir cómo seguir modificando la base de datos.

NOTA: El programa a ejecutar por el usuario, dentro del cual es posible elegir con qué lenguaje interactuar, los relaciona de la siguiente manera: como la Gramática de Red está presentada gráficamente dentro de un ambiente en el que el objetivo es enviar un mensaje de un *host* a otro, cuando se cumple ese objetivo, la Gramática de Base de Datos puede leer ese mensaje para popular su base de datos. El usuario puede definir instrucciones de la segunda gramática como mensaje a entregar por la primera. Esto es simplemente anecdótico, sólo una representación de la semántica.

4. Instrucciones de Uso

4.1 Instalaciones Previas

Dado que todo el proyecto está implementado en Haskell, debe tenerse instalado el compilador GHC. Luego, la única librería a descargar es “ansi-terminal”, para hacer más amena la interfaz gráfica para el usuario. Para ello, puede emplearse el manejador de librerías de Haskell, Cabal, ejecutando el siguiente comando:

$> *cabal install ansi-terminal*

4.2 Estructura de Carpetas y Archivos

El proyecto está contenido en 2 carpetas: */src*, donde están los archivos de código:

- *CombinatorParserLibrary.hs:* librería con todas las funciones para crear parsers propios.
- *ExampleGrammars.hs:* importa internamente a la librería, y define una serie de parsers sencillos de ejemplo.
- *NetLayerGrammar.hs:* Define los tipos de datos, los parsers y las funciones de evaluación de la gramática de Red. Importa internamente a la librería.
- *ApplicationLayerGrammar.hs:* Define los tipos de datos, los parsers y las funciones de evaluación de la gramática de Base de Datos. Importa internamente a la librería.
- *GUI.hs:* Archivo principal donde se encuentra el *main* a ejecutar por el usuario. Al correrlo, se muestra en consola un menú para elegir con qué lenguaje tratar interactivamente. Importa internamente a la librería y a ambos archivos de gramáticas.

Todas las funciones relevantes del código están comentadas identificando su funcionamiento. La otra carpeta del proyecto, */files*, contiene los siguientes archivos:

- *netLayerMessage.txt:* Este archivo representa el mensaje a entregar por la Gramática de Red. Puede completarse con instrucciones para la Gramática de Base de Datos. Por defecto, contiene comandos para popular la base con asociaciones entre los productos que fabrica una empresa y las fábricas donde se producen.
- *appLayerInstructions.txt:* Inicialmente vacío, al “completar” un mapa usando la Gramática de Red, y por consiguiente “entregándose” el mensaje al destino, se completa este archivo con el contenido del archivo anteriormente descrito, y es este del cual se leen las instrucciones, opcionalmente, por la Gramática de Base de Datos (nada impide, de todas formas, que se edite directamente este archivo, pero este trabajo no está orientado a estas cuestiones).

- */routersDistributions/*:* Estos archivos se utilizan para ofrecer distintos “mapas” para la Gramática de Red, es decir, distintas distribuciones de puntos por los cuales pasar indicándose los comandos. Se puede extender la cantidad de mapas y presentar más alternativas, pero no es el objetivo del trabajo, por eso no se incluye una descripción detallada de su contenido.

4.3 Ejecución y Ejemplos de Uso

Ya sea desde el modo interactivo GHCi:

$> *ghci*
$ghci> *:load GUI.hs*
$ghci> *main*

o compilando el archivo GUI.hs (que ya incluye los vínculos a los demás archivos) y corriendo el ejecutable:

$> *ghc -o main GUI.hs*
$> *./main*

el menú principal del programa indica que se debe elegir uno de los lenguajes con el cual interactuar.

Dentro de la Gramática de Red, el mapa con la distribución de componentes se ve como lo muestra la *Imagen1*. En esta distribución, se parte de la posición indicada por la “S” (*source*), y debe llegarse a la “D” (*destination*), pero antes pasar por las 4 “R” (*routers*), efectuando la operación “Translate” al posicionarse en cada uno.

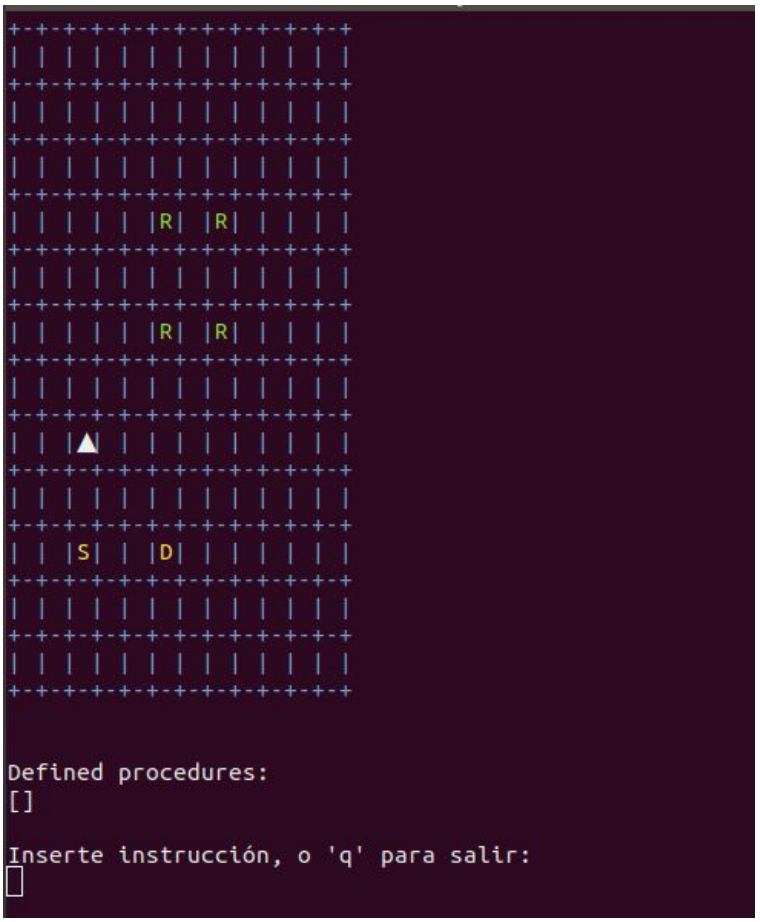

Imagen1: se muestra un mapa para la Gramática de Red, indicando los puntos por donde debe pasarse y la posición final.

Ejemplos de uso:

- "2Front+Right" implicaría moverse 2 posiciones al frente y una rotación a derecha.
- "3*(2Front+Right)" implicaría efectuar 3 veces la instrucción anterior.
- "reachAndTranslate=(Front+Translate)" implicaría definir un procedimiento con las instrucciones para avanzar una posición y efectuar la operación en el *router*.
- "reachAndTranslate" implicaría ejecutar las instrucciones indicadas en el previamente definido procedimiento.

Dentro de la Gramática de Base de Datos, se puede popular la base, por ejemplo, con las instrucciones mostradas en la *Imagen2*.

```
define PERIFERICO as TECNOLOGIA
define COMPUTACION as TECNOLOGIA
define ELECTRODOMESTICO as TECNOLOGIA
define TV as TECNOLOGIA
define ALEMANIA as EUROPA
define FRANCIA as EUROPA
define ARGENTINA as AMERICA
define BRASIL as AMERICA
define PC as COMPUTACION,ARGENTINA,ALEMANIA,FRANCIA
define NOTEBOOK as COMPUTACION,ARGENTINA
define HELADERA as ELECTRODOMESTICO,BRASIL
define COCINA as ELECTRODOMESTICO,BRASIL
define LAVARROPA  as ELECTRODOMESTICO,BRASIL,ARGENTINA
define MOUSE as PERIFERICO,ARGENTINA
define TECLADO as PERIFERICO,ARGENTINA
```

Imagen2: se muestran instrucciones de declaración de la Gramática de Base de Datos.

Este es un caso de ejemplo en el que una empresa fabrica productos de distintas clases en distintos países, y para mantener la organización se asocian a cada producto esos datos (tanto su clase como el país). En base a esto, se pueden hacer consultas de interés por si es necesaria alguna modificación de la producción de la empresa.

Ejemplos de uso para hacer declaraciones y consultas:

- “define SMARTTV as TV,FRANCIA,ESPAÑA” asociaría el término SMARTTV con las ideas TV y FRANCIA, con el significado que es de la categoría TV y se fabrica en Francia y España.
- “undefine SMARTTV as ESPAÑA” eliminaría sólo esa asociación.
- “delete SMARTTV” elimina toda asociación con el término SMARTTV.
- “list” mostraría todas las asociaciones registradas en la base.
- “type of TV” devolvería - TECNOLOGIA -.
- “is NOTEBOOK of type AMERICA” devolvería - *True* -, ya que es del tipo ARGENTINA, y este a su vez del tipo AMERICA.
- “exists CELULAR” devolvería - *False* -.
- “(is MOUSE of type AMERICA and TECLADO of type AMERICA)” devolvería - *True* -.

5. Conclusiones

Por un lado, la implementación de una librería genérica para construir los parsers es una herramienta muy potente, que puede aprovecharse mucho, siempre que se tengan en cuenta los lineamientos bajo los cuales está hecha.

La técnica de *Parser Combinator* es muy interesante por la sencillez que ofrece para usarla, sumado a que aprovecha algunos puntos fuertes del lenguaje funcional. Además, como se mencionó, se optó por no implementar mejoras de detección y manejo de errores, pero podrían tenerse en cuenta esta y otras extensiones para dar mayor robustez a la librería.

Sin embargo, no debe olvidarse el uso que se le da: gramáticas que presenten ambigüedad, es decir, la opción de parsearse de forma diferente pero que todas sean correctas, no serán el punto fuerte, ya que se gastan demasiados recursos haciendo *backtracking*. Si se esperan cadenas de entrada con una alta frecuencia de errores, pasa algo similar.

En cuanto a las gramáticas creadas, si bien se exponen algunos comandos interesantes (como la definición y utilización de procedimientos), queda claro que pueden extenderse a mucha mayor escala, dada la necesidad, ya que con la librería propuesta pueden inventarse diversas combinaciones y analizarse sintaxis más complejas.

6. Bibliografía

Papers:

[1] Jeroen Fokker. "Functional Parsers". August 1997.
[2] S. Doaitse Swierstra and Pablo R. Azero Alcocer. "Fast, Error Correcting Parser Combinators: A Short Tutorial". February 2000.
[3] S. Doaitse Swierstra. "Combinator Parsing: A Short Tutorial". January 2009.

Sintaxis:

https://www.haskell.org/onlinereport/haskell2010/haskellch10.html
https://en.wikipedia.org/wiki/Prolog_syntax_and_semantics

Actividades online:

http://pilasbloques.program.ar/online/#/libros/2
https://gobstones.github.io/gobstones-jr/?course=gobstones/curso-InPr-UNQ

Librerías:

https://hackage.haskell.org/package/ansi-terminal