Biblioteca de Parsers - Parslib

Trabajo Final - Programación Funcional

Tomás Colloca Bargallo - 54167

Indice

1. Objetivo

El objetivo de este trabajo es desarrollar una biblioteca de parseo, y luego utilizarla para armar un parser. Se busca profundizar el conocimiento de programación funcional, en particular de mónadas, y de parsers elaborados funcionalmente. Por lo tanto, este trabajo no presenta conceptos nuevos, sino que busca demostrar el conocimiento adquirido respecto a estos temas, e intentar de explicarlos de forma clara.

2. Introducción

Este trabajo comienza definiendo algunos conceptos que son útiles para entender el resto del mismo. Luego se explica cómo se fue desarrollando la biblioteca de parsers incrementalmente, desde la definición del tipo de un parser, hasta la elaboración de una biblioteca que permite obtener resultados online y otras mejoras. En particular, se implementó por cuenta propia, extendiendo la biblioteca propuesta en [4], una forma de reporte de errores, la cual se describe en la Sección 5.3.3. Reporte de Errores. Además, algunos de los combinadores mencionados también fueron elaborados por cuenta propia (Sección 4.5.). Por último, al final del trabajo en la Sección 6 se muestra parte de un parser de ejemplo desarrollado con la última biblioteca de parsers desarrollada, el cual también fue elaborado por el autor de este trabajo.

3. Conceptos

3.1. Gramática

Una de las definiciones de gramática, aquella dada por Chomsky, la define como una cuádrupla $G = <N, T, S, P>$ donde N es el conjunto de símbolos no terminales, T es el conjunto de símbolos terminales, S es el símbolo inicial ($S \varepsilon N$), y P es el conjunto de reglas de producción, donde una producción tiene la forma $\alpha \rightarrow \beta$ con $\alpha \varepsilon (N \cup T)^* \times N \times (N \cup T)^*$ y $\beta \varepsilon (N \cup T)^*$ (* denota la clausura de Kleene), la cual significa que α puede reescribirse como β. De esta forma, cualquier cadena $\gamma \varepsilon T$ que pueda obtenerse tras aplicar una serie de reglas de producción comenzando por el símbolo inicial S pertenece al lenguaje generado por la gramática G. Un ejemplo de esto es la siguiente gramática: $G_1 = <\{S, A\}, \{a\}, S, P>$ con $P = \{S \rightarrow bS, S \rightarrow Aa, Aa \rightarrow aAa, A \rightarrow \lambda\}$ donde λ denota la cadena vacía.

Se puede ver que $bbaa$ pertenece al lenguaje generado por G_1 ya que $bbaa$ se puede obtener tras aplicar una serie de reglas de producción como se ve a continuación:

1. $S \rightarrow bS \rightarrow bbS \rightarrow bbAa \rightarrow bbaAa \rightarrow bba\lambda a = bbaa$

Otro ejemplo de gramática es aquella que describe al idioma español, donde se podrían tener como símbolos no terminales $Oración$, el cual sería el símbolo inicial, $Sujeto$, $Predicado$, $Artículo$, $Sustantivo$, entre otros, como símbolos terminales las palabras como el, la, $chico, corre$, y algunas reglas de producción serían $Oración \rightarrow Sujeto\ Predicado$, $Sujeto \rightarrow Artículo\ Sustantivo$, $Artículo \rightarrow el \mid la$, $Sustantivo \rightarrow chico$.

3.2. Scanner

Un *scanner*, o analizador lexicográfico, es un programa que transforma una cadena de símbolos en componentes léxicos, también llamados *tokens*. Básicamente el mismo se conforma de una serie de expresiones regulares para cada *token* donde cada una describe el conjunto de posibles "valores" llamados *lexemas* que puede tomar el mismo. Un scanner suele conformar la primera fase de un compilador.

3.3. Parser

Un *parser*, o analizador sintáctico en español, es un programa que recibe una cadena de símbolos y genera una estructura o resultado a partir de los mismos. Un ejemplo sencillo de esto podría ser un programa que recibe una cadena de caracteres, o *string*, y devuelve como resultado "Sí, es una pregunta" o "No, no es una pregunta" en función si dicha cadena termina con el signo de preguntas "?". Un parser más elaborado podría ser aquel que recibe una oración, y a partir de la gramática que describe al idioma español, genera la estructura sintáctica asociada a la misma, donde en la estructura queda representado qué parte de la oración es el sujeto, cuál es el predicado, cuál es el verbo del predicado, entre otras cosas. En lo que refiere a la informática, un parser es una de las primeras componentes de un compilador, donde se toma una secuencia de *tokens* generados por un analizador lexicográfico, y a partir de una gramática formal se genera una estructura conocida como árbol de sintaxis abstracta (AST) que representa la estructura sintáctica del código del programa. Es importante notar que un parser recibe una cadena de símbolos y busca encontrar la secuencia de reglas de producción que llevan a dicha cadena.

Finalmente, normalmente la estructura generada por el parser la toma la última parte del compilador, la cual realiza un análisis semántico del código utilizando dicho árbol, el cual permitiría por ejemplo hacer chequeo de tipos, y traduce dicha estructura a otro lenguaje de programación, normalmente de más bajo nivel.

3.4. Compresión de listas

La comprensión de listas es una forma de escribir una lista de forma implícita a partir de otras listas. Por ejemplo, a partir de la lista infinita `[1..]` que representa los números naturales, se podría definir una lista con los números naturales menores a `n` de la siguiente forma:

```
evenN :: Int -> [Int]
evenN n = [x | x <- [1..], x < n]
```

3.5. Continuation-passing style

En programación funcional, el continuation-passing style (CPS) es un estilo de programación en el cual el control del programa se pasa explícitamente como una función. Dicho en otras palabras, una función CPS recibe un argumento extra, la continuación, y cuando calculó aquello que debía calcular, “retorna” llamando a la continuación con el resultado de dicho cálculo. Por ejemplo, la siguiente función está escrita en estilo directo, y toma dos números y suma el resultado de multiplicarlos por dos:

```
sumAndDouble :: Int -> Int -> Int
sumAndDouble x y = sum (double x) (double y)

double :: Int -> Int
double x = x * 2

sum :: Int -> Int -> Int
sum x y = x + y
```

La misma se la puede reescribir en estilo CPS de la siguiente forma:

```
sumAndDoubleCps :: Int -> Int -> (Int -> r) -> r
sumAndDoubleCps x y = \ k ->
    doubleCps x                 (\ doubledX ->
    doubleCps y                 (\ doubledY ->
    sumCps    doubledX doubledY (\ doubleXplusDoubleY ->
    k         doubleXplusDoubleY)))


doubleCps :: Int -> (Int -> r) -> r
doubleCps x = \ k -> k  (x * 2)

sumCps :: Int -> Int -> (Int -> r) -> r
sumCps x y = \ k -> k (x + y)
```

Se reescribieron `double` y `sum` en estilo CPS también para extender el ejemplo, pero no era necesario. Como se puede ver, por ejemplo `sumAndDoubleCps 2 3` devuelve una función que recibe otra función que decide qué hacer con el resultado de sumar el doble de 2 con el doble de 3.

4. Parser monádico

4.1. Tipo de un parser

Como se mencionó previamente, un parser podría tomar una secuencia de símbolos, que podrían ser *tokens*, y devolver el AST asociado al mismo. De esta forma, el parser tendría el siguiente tipo:

```
type Parser s = [s] -> Ast
```

No necesariamente el parser tiene que consumir todos los símbolos que recibe como entrada, ni necesariamente lo que genera tiene que ser un AST. De esta forma se redefine el tipo del parser para que el tipo de retorno sea genérico, y además devuelva lo que no llegó a consumir de la entrada:

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

Finalmente, podría ser el caso que existan varias formas de parsear una entrada, es decir, no es única la secuencia de reglas de producción que derivan en la cadena que se recibió como entrada, y por lo tanto el parser devolvería una lista con todas las estructuras asociadas a

cada una de estas formas de parsear. Esto lleva a redefinir el tipo del parser de la siguiente forma:

```
  import Prelude hiding ((<*>), (<$>), (<*), (*>), pure)

  newtype Parser s r = P ([s] -> [(r, [s])])

  unP :: Parser s r -> ([s] -> [(r, [s])])
  unP (P p) = p
```

Se agregó un constructor `P` que devuelve el tipo del parser y se reemplazó `type` por `newtype` para que el mismo pueda instanciar clases más adelante. Además, se agregó el `import` para poder redefinir temporalmente las funciones en cuestión.

4.2. Parsers elementales

A partir de la definición anterior, se propone desarrollar el primer parser elemental, `item`, que decide si el primer elemento de la entrada que recibe corresponde con cierto símbolo, devolviéndolo:

```
item :: Eq s => s -> Parser s s
item s = P (\ inp -> case inp of
                       (x:xs) | s == x -> [(x, xs)]
                       otherwise       -> [])
```

Otro parser básico que se puede desarrollar es `pure`, que no consume ningún elemento de la entrada, pero genera un resultado que recibe por argumento:

```
pure :: r -> Parser s r
pure r = P (\ inp -> [(r, inp)])
```

Finalmente se define también `empty`, que devuelve la lista vacía, independientemente de la entrada. Notar que este parser equivale a “fallar”, puesto que el que no haya ningún elemento equivale a pensar que no hay ninguna forma de parsear la entrada:

```
  empty :: Parser s r
  empty  = P (\ inp -> [])
```

Estos últimos dos parsers tendrán más sentido en la próxima sección.

4.3. Parser Combinators

Una de las primeras cuestiones que surgen a partir de `item` es cómo se podría parsear una secuencia de dos símbolos, o incluso más. Para solucionar este problema se busca desarrollar un operador `<*>` que reciba dos parsers `p` y `q`, y los secuencialize. Es decir, `p <*> q` recibe una entrada la cual la consume inicialmente `p` generando una lista de pares `(out, r1)` donde `out` es lo que queda por consumir de la entrada para esa opción de parseo, y luego `q` recibe cada `out` y devuelve una lista de pares para cada posible resultado de `p`. La definición de `<*>` es la siguiente:

```
(<*^>) :: Parser s a -> Parser s b -> Parser s (a,b)
p <*^> q = P (\ inp -> [((r1, r2), out') | (r1, out ) <- unP p inp
                                         , (r2, out') <- unP q out])
```

Sin embargo, esta definición trae el problema que el resultado de la secuencia de muchos parsers termina siendo una anidación de pares tan larga como la cantidad de parsers en cuestión. Por eso se decide que el primer parser genere como resultado una función que sepa qué hacer con el resultado del parser que venga a continuación:

```
(<*>) :: Parser s (b -> a) -> Parser s b -> Parser s a
p <*> q = P (\ inp -> [(r1 r2, out') | (r1, out ) <- unP p inp
                                     , (r2, out') <- unP q out])
```

Otra posible combinación de dos parsers es poder elegir alguno de los dos. Por ejemplo, cuando se parsea el lado derecho de una variable, podría haber un literal o una expresión. Es por esto que se define el operator `<|>` donde `p <|> q` devuelve la concatenación de los resultados devueltos por `p` al recibir cierta entrada con los de `q` al recibir la misma entrada:

```
(<|>) :: Parser s a -> Parser s a -> Parser s a
p <|> q = P (\ inp -> unP p inp ++ unP q inp)

opt :: Parser s a -> a -> Parser s a
p `opt` v = p <|> pure v
```

El combinador `opt` simplifica una operación que se realiza bastante frecuente, y es por eso que se lo decide agregar.

Un tercer combinador de parsers útil es aquel que permite aplicar una función al resultado de un parser. Es decir, si `p` genera un resultado `r`, `f <$> p` va a generar `f r` como resultado, sin cambiar lo que resta de la cadena tras utilizar `p`. Como pure justamente permite generar un parser sin modificar la entrada, se lo utiliza para definir este nuevo operador:

```
(<$>) :: (a -> b) -> Parser s a -> Parser s b
f <$> p = pure f <*> p
```

Para ahorrar la cantidad de paréntesis se definen los siguientes órdenes de precedencia, donde un número mayor indica mayor precedencia:

```
infixl 7 <$>
infixl 5 <*>
infixr 3 <|>
```

A partir de estos operadores se puede definir `token`, un parser que consume una secuencia de caracteres específica:

```
token' :: Eq s => [s] -> Parser s [s]
token' (x:xs) = (:) <$> item x <*> token' xs
token' []     = pure []
```

Dado que la función `(:)` se suele “elevar” a parsers, se define un nuevo combinador:

```
infixr 5 <:*>

(<:*>) :: Parser s a -> Parser s [a] -> Parser s [a]
p <:*> q = (:) <$> p <*> q

token :: Eq s => [s] -> Parser s [s]
token (x:xs) = item x <:*> token xs
token []     = pure []
```

Muchas veces en los lenguajes de programación se utilizan *tokens* que permiten separar estructuras o indicar la presencia de una estructura, pero que no aportan contenido semántico al AST. Dicho de otra forma, es necesario parsearlos, pero no aportan ningún resultado útil. Es por esto que surgen dos combinadores que realizan justamente esto:

```
infixl 5 <*, *>

(*>) :: Parser s a -> Parser s b -> Parser s b
p *> q = flip const <$> p <*> q

(<*) :: Parser s a -> Parser s b -> Parser s a
p <* q = const <$> p <*> q
```

A partir de estos se puede definir por ejemplo `parens`, que dado un parser `p` genera uno nuevo donde lo que reconoce `p` debe estar entre paréntesis:

```
parens' :: Parser Char a -> Parser Char a
parens' p = token "(" *> p <* token ")"
```

Dado que la característica que cierta secuencia de símbolos esté rodeada por otros dos símbolos es común (por ejemplo en lugar de paréntesis, corchetes) se define la función `pack` que abstrae un poco más que `parens`:

```
pack :: Parser s b -> Parser s a -> Parser s c -> Parser s a
pack open p close = open *> p <* close

parens :: Parser Char a -> Parser Char a
parens p = pack (token "(") p (token ")")
```

Otro combinador muy útil es `choice`, que permite iterar sobre una lista de parsers y aplicar el operador <|>:

```
choice :: [Parser s a] -> Parser s a
choice ps = foldr (<|>) empty ps
```

Notar que el parser `empty` hace de elemento neutro frente al operador <|>, mientras que hace de elemento nulo frente al operador <*>. Se puede notar que hay cierta relación entre { <|>, $\vee$, + }, { <*>, $\wedge$, * } y {empty, falso, 0} .

Con este nuevo combinador podemos definir dos parsers básicos, `letter` y `digit`, que permiten reconocer una letra o un dígito respectivamente:

```
digit :: Parser Char Char
digit = choice (map item ['0'..'9'])

lowercase :: Parser Char Char
lowercase = choice (map item ['a'..'z'])

uppercase :: Parser Char Char
uppercase = choice (map item ['A'..'Z'])

letter :: Parser Char Char
letter = lowercase <|> uppercase
```

```
alphaNum :: Parser Char Char
alphaNum = digit <|> letter
```

A su vez, una palabra surge de una o más letras, y por lo tanto, es necesario tener un combinador que permita repetir un parser. Para esto se define el combinador `many` y `many1` donde difieren en que el primero permite cero o más repeticiones, y el segundo exige al menos una:

```
many :: Parser s a -> Parser s [a]
many p = p <:*> many p <|> pure []

many1 :: Parser s a -> Parser s [a]
many1 p = p <:*> many p
```

Ahora sí se puede definir una palabra, `word`, un identificador, `ident`, y los números naturales, `nat`:

```
word :: Parser Char String
word = many1 letter

ident :: Parser Char String
ident = many1 alphaNum

intDigit :: Parser Char Int
intDigit = (\ x -> fromEnum x - fromEnum '0') <$> digit

nat :: Parser Char Int
nat = (foldl (\a b -> a * 10 + b) 0) <$> many1 intDigit
```

Para poder obtener un parser de números enteros, `int`, se necesita tener la opción de tener un signo “-” antes del número. Para eso se define antes un operador que permite la presencia o no de lo reconocido por un parser, y qué se debe hacer para cada caso. El mismo es `<?|>`, y recibe un parser `p` y un par con una función que transforma el resultado de `p` en caso de éxito, y un elemento nulo en caso de fracaso.

```
infixl 7 <?|>

(<?|>) :: Parser s a -> (a -> b, b) -> Parser s b
p <?|> (f, z) = (f <$> p) <|> pure z

int :: Parser Char Int
int = token "-" <?|> (const negate, id) <*> nat
```

Otra característica que muchas veces tienen los lenguajes de programación es poder tener una lista de elementos. Para eso se definen `sepBy` y `sepBy1`:

```
sepBy1 :: Parser s a -> Parser s b -> Parser s [a]
sepBy1 p sep = p <:*> many (sep *> p)

sepBy :: Parser s a -> Parser s b -> Parser s [a]
sepBy p sep = sepBy1 p sep `opt` []
```

4.4. Expresiones aritméticas

A partir de los parsers definidos anteriormente, se puede realizar un parser para expresiones aritméticas. Para eso se define inicialmente un parser que dada la entrada de dos números sumados, resuelva la suma:

```
addTwo’ :: Parser Char Int
addTwo’ = (+) <$> int <* token "+" <*> int
```

Sin embargo, esto sería poco práctico, puesto que para cada operando se debería tener una función distinta. Es por eso que se decide abstraer el operando, y agregar un parámetro que parsee el operador y provea la función del mismo:

```
opTwo' :: Parser Char (Int -> Int -> Int) -> Parser Char Int
opTwo' op  = int <*> op <*> int
```

Sin embargo, el código anterior no funciona, puesto que aquel que debería generar la función que recibe dos parámetros es el primer `int`. Se define entonces un nuevo combinador que permite tener el parser que genera la función como segundo parámetro:

```
infixl 5 <**>

(<**>) :: Parser s a -> Parser s (a -> b) -> Parser s b
p <**> q = (\ a f -> f a) <$> p <*> q

opTwo :: Parser Char (Int -> Int -> Int) -> Parser Char Int
opTwo op  = int <**> op <*> int

add' :: Parser Char (Int -> Int -> Int)
add' = const (+) <$> token "+"
```

```
addTwo :: Parser Char Int
addTwo = opTwo add'
```

De nuevo se puede abstraer un poco más la función `add`, para los distintos operadores:

```
biOp :: Eq s => (a -> b -> c, [s]) -> Parser s (a -> b -> c)
biOp (f, tok) = const f <$> token tok

add :: Parser Char (Int -> Int -> Int)
add = biOp ((+), "+")
```

El próximo paso es poder realizar una operación entre más de dos números, por ejemplo, para operar 4 números se necesitaría algo como:

```
opFour' :: Parser Char (Int -> Int -> Int) -> Parser Char Int
opFour' op = (((int <**> (op <*> int)) <**> (op <*> int)) <**> (op <*> int))
```

Los paréntesis no son necesarios, pero agrupando de esta forma, se logra obtener un parser que se repite varias veces: `(op <*> int)`. El primero se aplica a `int`, el segundo al resultado de lo anterior, el tercero al resultado mencionado anteriormente, y así se podría seguir extendiendo. Hay un pequeño problema que no ocurre con el operador `(+)` pero si con `(-)`. Si se realiza `((-) 1) 5`, se obtiene `-4`, cuando en el problema mencionado anteriormente se hubiese esperado obtener `4`. Es por eso que es necesario aplicar `flip` al operador. Se define entonces una función, `applyAll`, que realiza lo primero, y se reescribe `opFour'`:

```
applyAll :: a -> [a -> a] -> a
applyAll z (f:fs) = applyAll (f z) fs
applyAll z [] = z

opFour'' :: Parser Char (Int -> Int -> Int) -> Parser Char Int
opFour'' op = applyAll <$> int <*> many (flip <$> op <*> int)
```

Se decide abstraer este último parser de los sumandos, obteniéndose `chainl`:

```
chainl :: Parser s (a -> a -> a) -> Parser s a -> Parser s a
chainl op p = applyAll <$> p <*> many (flip <$> op <*> p)
```

Simétricamente se puede obtener `chainr`. Esta función consiste en aplicar un operador entre operandos de derecha a izquierda. Se la puede pensar de manera recursiva, donde el caso base es parsear un operando, y el paso inductivo es parsear un operando, y aplicarlo al

resultado de parsear un operador binario aplicado parcialmente al resultado de la llamada recursiva. De esta forma, la función queda definida de la siguiente manera:

```
chainr' :: Parser s (a -> a -> a) -> Parser s a -> Parser s a
chainr' op p = p <|> p <**> (flip <$> op <*> chainr' op p)
```

El problema que tiene `chainr'`, es que ambas alternativas comienzas con `p`, y en caso de fallar una, se va a volver a llevar a cabo el parseo de `p`. Para solucionar esta cuestión de ineficiencia, se propone un combinador más, y se reescribe `chainr`:

```
(<??>) :: Parser s a -> Parser s (a -> a) -> Parser s a
p1 <??> p2 = p1 <**> (p2 `opt` id)

chainr :: Parser s (a -> a -> a) -> Parser s a -> Parser s a
chainr op p = p <??> (flip <$> op <*> chainr op p)
```

4.5. Combinators diseñados

Los combinadores de parsers de esta sección fueron elaborados por cuenta propia. Uno de ellos es `<++>` que permite concatenar las listas obtenidas por dos parsers consecutivos. Otro de ellos es `<*:>` que dados los resultados de una lista y un elemento, generado por dos parser consecutivos, agrega el elemento al final de la lista. A diferencia de `<:*>`, se decidió que la asociatividad sea de izquierda a derecha. Por último se agregan también `?*>` y `<*?` que funcionan de manera similar a `*>` y `<*` respectivamente, pero permitiendo la ausencia del parser cuyo resultado es ignorado. Las definiciones de las funciones descriptas son las siguientes:

```
(<++>) :: Parser s [a] -> Parser s [a] -> Parser s [a]
p <++> q = (++) <$> p <*> q

(<*:>) :: Parser s [a] -> Parser s a -> Parser s [a]
ps <*:> p = (\a b -> a ++ [b]) <$> ps <*> p

(?*>) :: Parser s a -> Parser s b -> Parser s b
p ?*> q = p <?|> (const undefined, undefined) *> q

(<*?) :: Parser s a -> Parser s b -> Parser s a
p <*? q = p <* q <?|> (const undefined, undefined)
```

4.6. Parser monádico

El parser descrito hasta el momento es no determinístico, porque permite una lista de posibles formas de parsear una entrada. Sin embargo, se podría querer tener un parser determinístico, donde por ejemplo tenga comportamiento *greedy* cuando se pueda aplicar varias veces un mismo subparser, y que se quede con sólo una forma de parsear, y en caso de error, falle. La característica no determinística del parser que se desarrolló proviene de la lista de pares de lo que sobró de la entrada y el resultado obtenido. Si se pudiera cambiar dicha lista por, por ejemplo, `Maybe`, entonces se lograría obtener un parser determinístico, y que cuando el parseo falle, se obtenga `Nothing`.

Por suerte, no son muchas las funciones que hacen referencia a la lista, que es importante destacar que es una mónada al igual que `Maybe`. Lo que se busca entonces es poder abstraernos de la mónada a utilizar, y en su lugar, usar funciones que se sabe que la misma tiene justamente por ser una mónada. Sin embargo, se necesita agregar una restricción más, y es que la mónada sea también una instancia de `Alternative`, que tiene dos operaciones: `empty` y `<|>`. Por ejemplo, `empty` en el caso de la lista devuelve la lista vacía, y `<|>` concatena dos listas. En el caso de `Maybe`, `empty` devuelve `Nothing`, y `<|>` se define de la siguiente manera (el código original podría ser distinto):

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
a@(Just _) <|> (Just _)   = a
a@(Just _) <|> Nothing    = a
Nothing    <|> b@(Just _) = b
Nothing    <|> Nothing    = Nothing
```

A partir de lo dicho anteriormente, se crea un nuevo tipo, `GenericParser`, el cual es instancia de `Monad`, y se redefine en función de `GenericParser` al tipo `Parser`. El objetivo es poder ajustar el resto del código, para que sólo se deba cambiar la mónada en la definición de `Parser` para que el comportamiento del mismo cambie. La instancia de `GenericParser` de `Monad` y sus superclases `Functor` y `Applicative`, es la siguiente:

```
import Control.Applicative (Applicative(pure, (<*>)))

newtype GenericParser m s r = P ([s] -> m (r, [s]))

instance Monad m => Functor (GenericParser m s) where
    fmap = liftM
```

```
instance Monad m => Applicative (GenericParser m s) where
    pure r  = P (\ inp -> pure (r, inp))
    p <*> q = p >>= (\ r1 ->
              q >>= (\ r2 ->
              return (r1 r2)))

instance Monad m => Monad (GenericParser m s) where
    p >>= f  = P (\ inp -> (unP p) inp >>= (\ (v, inp') -> unP (f v) inp'))

unP :: GenericParser m s r -> ([s] -> m (r, [s]))
unP (P p) = p

type Parser s r = GenericParser [] s r
```

Se puede ver en la misma que tanto en la definición de `pure` como en la de `>>=` se delega a la función correspondiente de la mónada que se está utilizando. Es importante notar que como `Applicative` es superclase de `Monad` no es necesario definir return pues la misma función por default está definida como `return = pure`.

La instancia de `GenericParser` de `Alternative` es la siguiente:

```
instance (Alternative m, Monad m) => Alternative (GenericParser m s) where
    empty   = P (\ inp -> empty)
    p <|> q = P (\ inp -> unP p inp <|> unP q inp)
```

De nuevo se puede ver que en ambas funciones se delega a los métodos de las mónadas en cuestión.

Sólo queda redefinir la función `item`, y ya no estará fijado en ningún lado del código que mónada se debe usar:

```
item :: Eq s => s -> Parser s s
item s = P (\ inp -> case inp of
                       (x:xs) | s == x -> pure (x, xs)
                       otherwise       -> empty)
```

5. Parser online

Un parser elaborado utilizando la biblioteca desarrollada hasta el momento tiene algunas desventajas:

- Para poder generar un resultado, es necesario que se analize toda la entrada. Esto podría ser muy costoso en cuanto a consumo de memoria, ya que se debería tener toda la entrada en la misma y los distintos posibles caminos que puede tomar el parser. Además, en el caso en que la entrada sea de la forma `['a', undefined]`, por más que el parser sólo consuma el caracter `'a'`, el resultado va a ser $\bot$, puesto que como se mencionó previamente es necesario llegar hasta el final de la entrada.
- En caso de error, se va a retornar lo que retorne la mónada en la función `empty`, es decir, si se está usando listas, la lista vacía, o si se está usando `Maybe`, `Nothing`. El problema de esto es que no hay forma de saber en qué posición de la entrada falló, o por ejemplo que símbolo era el esperado.

En busca de solucionar estos problemas, se busca desarrollar un parser online, es decir, que pueda generar resultados ni bien pueda, y que facilite el reporte de errores.

5.1. Clases

Uno de los cambios que se llevan a cabo es que el parser en lugar de recibir una secuencia de símbolos, pasa a recibir un estado, el cual además de la cadena de símbolos puede contener otra información como la posición actual, los errores encontrados, etc.

Teniendo esto en cuenta, se definen las siguientes clases para poder construir el parser online:

```
  type Cost = Int

  class symbol `Describes` token where
      eqSymTok    :: symbol -> token -> Bool

  class Symbol p symbol token where
      sym         :: symbol -> p token

  class (Show position, Ord position) => Provides state symbol token position |
state symbol -> token, symbol token -> position where
      splitState  :: symbol -> state -> (position, Maybe (Progress, token, state))
```

```
class Eof state where
    eof         :: state  -> Bool
```

La clase `Describes` tiene el método `eqSymTok` que permite decidir si un token corresponde con el símbolo esperado. Es importante tener en cuenta que un token puede contener más información que un símbolo, y por eso es útil tener esta relación.

La clase `Symbol` permite con el método `sym` a partir de un símbolo construir un parser que lo reconozca, generando un token.

La clase `Provides` tiene un método `splitState` el cual a partir de un símbolo y un estado se obtiene la posición en la que se encuentra el parser, y de ser posible, una tupla que contiene cuánto se avanzó en la cadena de símbolos en este llamado, el token generado, y el nuevo estado. Las dependencias funcionales `state symbol -> token` y `symbol token -> position` establecen que una vez que se fijó el tipo del estado y el símbolo, en el caso de la primera, el tipo del token queda también definido, y es único para ese par de tipos.

Finalmente, la clase `Eof` ofrece el método `eof` que permite a partir de un estado saber si se llegó al final de la entrada, o si quedan más tokens por reconocer.

Se define además una estructura, `Steps`, que hace de registro del tipo de resultado de la secuencia de parsers, que gracias a la evaluación lazy de Haskell, va a permitir que el parser a desarrollar tenga las propiedades deseadas:

```
type Progress = Int

data Error = Error (Maybe String) String

data Steps a where
    Step    :: Progress  -> Steps a                          -> Steps a
    Fail    :: [Error]                                       -> Steps a
    Apply   :: (b -> a)  -> Steps b                          -> Steps a
    Success :: Steps a                                       -> Steps a
    Endh    :: ([a], [a] -> Steps r) -> Steps (a, r)         -> Steps (a, r)
    Endf    :: [Steps a] -> Steps a                          -> Steps a
```

El constructor `Step` representa el resultado exitoso de un paso del parser, donde se guarda el progreso realizado. El constructor `Fail` representa el fracaso del parser a partir de ese punto, y el listado de errores encontrados que llevaron al mismo. El constructor `Apply` permite postergar la aplicación de una función al resultado generado por el parser. Los otros se explican en secciones posteriores.

La función `best` es la que permite que se realice la evaluación lazy de la estructura `Steps` en el orden que es deseado. La definición de la misma es la siguiente:

```
best :: Steps a -> Steps a -> Steps a
l `best` r            = norm l `best'` norm r

best' :: Steps a -> Steps a -> Steps a
Fail sl           `best'` Fail sr        = Fail (sl ++ sr)
Fail _            `best'` r              = r
l                 `best'` Fail  _        = l
Step n  l         `best'` Step  m r
    | n == m                             = Step n (l `best'` r)
    | n <  m                             = Step n (l `best'` Step (m - n) r)
    | n >  m                             = Step m (Step (n - m) l `best'` r)
Endf as l         `best'` Endf bs r      = Endf (as ++ bs) (l `best` r)
Endf as l         `best'` r              = Endf as         (l `best` r)
l                 `best'` Endf bs r      = Endf bs         (l `best` r)
Endh (as, k_st) l `best'` Endh (bs, _) r = Endh (as ++ bs, k_st) (l `best` r)
Endh as         l `best'` r              = Endh as               (l `best` r)
l                 `best'` Endh bs      r = Endh bs               (l `best` r)
l                 `best`'  r             = l `best` r

norm :: Steps a -> Steps a
norm (Apply f (Step    n l   ) ) = Step n (Apply f l)
norm (Apply f (Fail    ss    ) ) = Fail ss
norm (Apply f (Apply   g l   ) ) = norm (Apply (f . g) l)
norm (Apply f (Success l     ) ) = Success (Apply f l)
norm (Apply f (Endf    as l  ) ) = Endf (map (Apply f) as) (Apply f l)
norm (Apply f (Endh    _  _  ) ) = error "Found Endh on the loose when calling
                                          norm!"
norm steps = steps
```

La misma se encarga de decidir dadas dos alternativas de parseo, cómo se debe seguir evaluando ambas, idealmente progresando al mismo tiempo en ambas ramas, exploración a lo ancho, en lugar de explorar en profundidad como realizaba el parser hasta el momento.

En pocas palabras, en caso de que una rama falle, se toma aquella que no falló. Si ambas tienen éxito en el próximo paso, se considera que se avanza lo que consume la que avanza menos, y que de la otra rama queda por consumir la diferencia. Las otras opciones de la función se mencionan más adelante. Por otro lado, la función `norm` mueve los Steps de tipo Apply hacia el fondo hasta encontrar uno de los otros constructores.

5.2. Definición del parser

El tipo del parser se define a partir de otros tres: Un reconocedor (recogniser), un parser "histórico" (history parser), y un parser "futuro" (future parser). El primero es capaz de reconocer símbolos, pero sin conservar el token generado. El segundo lleva registro del resultado obtenido hasta el momento a partir del lado izquierdo de la función <*> hasta que el mismo pueda ser aplicado al lado derecho. Por último, el tercero funciona al revés del segundo, donde conceptualmente se guarda una pila de los resultados futuros, lo cual permite aplicar el resultado del lado izquierdo de la función <*> ni bien se lo obtiene. La ventaja que provee esto es que, por la evaluación lazy, la función del lado izquierdo del <*> va a poder realizar todo aquello que no necesite del lado derecho, sin siquiera parsear el último. Esto es lo que permite que el parser genere resultados online.

Sin embargo, hay casos donde el parser futuro necesita del parser histórico, o alguno necesita del reconocedor, y es por eso que se crea un tipo que contiene los tres:

```
--     &&&   RECOGNISER    &&&    --

newtype Pr st a = Pr (forall r. (st -> Steps r) -> st -> Steps r)

--     &&& HISTORY  PARSER &&&    --

newtype Ph st a = Ph (forall r. (a -> st -> Steps r) -> st -> Steps r)

--     &&&  FUTURE PARSER  &&&    --

newtype Pf st a = Pf (forall r. (st -> Steps r) -> st -> Steps (a, r))

--     &&& COMBINED PARSER &&&    --

data    Pm st a = Pm (Pr st a) (Ph st a) (Pf st a)


parse :: Eof st => Pm st a -> st -> a
parse (Pm _ _ (Pf pf)) = fst . eval . pf (\ rest -> if eof rest
                                                    then undefined
                                                    else error "End exception")
```

Como se puede ver, los tres parsers mencionados reciben una continuación y un estado, y retornan un nuevo `Steps`. Se puede ver que el reconocedor no tiene ninguna interacción con el token, mientras que el parser histórico toma el último token generado como primer argumento de la continuación, y el parser futuro lo pone en el tope de la pila que debajo contiene los resultados "futuros". La función `parse` permite utilizar un parser dado un estado, utilizando el parser futuro.

5.2.1. Secuenciación

```
instance Applicative (Pr st) where
    (Pr r1) <*> (Pr r2) = Pr (\ k st -> r1 (r2 k) st)
    pure r              = Pr (\ k st -> k st)

instance Applicative (Ph st) where
    (Ph p) <*> (Ph q) = Ph (\ k -> p (\ f -> q (\ a -> k (f a))))
    pure r            = Ph (\ k -> k r)

instance Applicative (Pf st) where
    (Pf p) <*> (Pf q) = Pf (\ k st -> applyf (p (q k) st))
    pure a            = Pf (\ k st -> push a (k st))


applyf :: Steps (b -> a, (b, r)) -> Steps (a, r)
applyf = Apply (\(b2a, ~(b, r)) -> (b2a b, r))

push :: v -> Steps r -> Steps (v, r)
push v = Apply (\s -> (v, s))


instance Applicative (Pm st) where
    (Pm pr ph pf) <*> ~(Pm qr qh qf)                   = Pm (pr <*> qr)
                                                           (ph <*> qh)
                                                           (pf <*> qf)

    (Pm (Pr pr) (Ph ph) (Pf pf)) <* ~(Pm (Pr qr) _ _)  = Pm (Pr (pr . qr))
                                                           (Ph (ph . (qr .)))
                                                           (Pf (pf . qr))

    (Pm (Pr pr) _ _) *> ~(Pm (Pr qr) (Ph qh) (Pf qf))  = Pm (Pr (pr . qr))
                                                           (Ph (pr . qh))
                                                           (Pf (pr . qf))

    pure a                                             = Pm (pure a) (pure a) (pure a)
```

Cómo se puede ver, respecto a la secuenciación (<*>), el recogniser usa como continuación del primer parser el llamar a q con la continuación que recibió. El parser histórico arma una cadena de continuaciones donde una vez que se obtuvo la función del primer parser y el resultado del segundo, se llama a la continuación que se recibió con el resultado de aplicar la función del primero al resultado del segundo parser. Por el otro lado, el parser futuro logra por un lado almacenar la función que se debe aplicar y el valor al que se debe aplicar en el registro dado por `Steps`, y a su vez, gracias a la definición de `applyf`, se obliga a que no se realice el pattern match del par con el resultado y stack futuro del segundo parser, permitiendo así que se

pueda ejecutar la primera función generada por el primer parser de la función (<*>) hasta donde no necesite del resultado del segundo.

En lo que refiere a la función `pure`, el reconocedor al no importarle los tokens devuelve la función identidad. El parser histórico toma el valor que recibe por parámetro como el último token del que fue testigo, pasándoselo a la continuación. En adición, el parser futuro pone en el tope de la pila de valores futuros al valor que recibió.

Para ambas funciones el parser que combina a los tres delega en la función correspondiente de cada uno de los parsers. En lo que respecta a los operadores <* y *>, se puede ver que el parser general utiliza el reconocedor para parsear aquel lado que no interesa el token producido.

5.2.2. Alternación

```
instance Alternative (Pr st) where
    (Pr r1) <|> (Pr r2) = Pr (\ k inp -> r1 k inp `best` r2 k inp)
    empty               = Pr (\ k st -> noAlts)

noAlts :: Steps a
noAlts = Fail [Error Nothing "Probably no alternative worked."]

instance Alternative (Pm st) where
    (Pm pr ph pf) <|> (Pm qr qh qf) = Pm (pr <|> qr) (ph <|> qh) (pf <|> qf)
    empty                           = Pm empty empty empty
```

Dado que los métodos son iguales para los tres parsers, sólo se visualiza el del reconocedor. Es con el operador <|> donde se hace uso de la función `best`, la cual permite cambiar la exploración en profundidad por exploración a lo ancho.

En adición, como se mencionó anteriormente, `empty` equivale a fallar, y por lo tanto, se utiliza el constructor `Fail`.

Una vez más el parser que combina los tres delega en la función correspondiente a cada uno.

5.2.3. Mapeo de función

```
instance Functor (Pr st) where
    f `fmap` (Pr r) = (Pr r)

instance Functor (Ph st) where
    f `fmap` (Ph p) = Ph (\ k -> p (\ a -> k (f a)))

instance Functor (Pf st) where
    f `fmap` p = pure f <*> p

instance Functor (Pm st) where
    f `fmap` (Pm pr ph pf) = Pm (f `fmap` pr) (f `fmap` ph) (f `fmap` pf)
```

En primer lugar, la función `fmap` es equivalente a la función `<$>` utilizada hasta el momento. Una vez más, como el recogniser no se fija en los tokens, la función recibida no tiene ningún efecto. Por el contrario, el parser histórico introduce la aplicación de la función al token reconocido a la continuación recibida por parámetro, siendo un poco más performante que la definición que se había dado previamente de la función `<$>`, la cual sí conserva el parser futuro.

Finalmente el parser que combina los tres delega la llamada a cada parser individualmente.

5.2.4. Instancia Monádica

```
instance Monad (Pm st) where
    (Pm _ (Ph p) _) >>= a2q = Pm (Pr (\ k -> p (\ a -> unPm_r (a2q a) k)))
                                 (Ph (\ k -> p (\ a -> unPm_h (a2q a) k)))
                                 (Pf (\ k -> p (\ a -> unPm_f (a2q a) k)))
```

Como se necesita del último símbolo que se pudo atestiguar, es necesario utilizar el parser histórico para los tres casos. Este es otro de los casos donde un parser de un tipo necesita de otro.

5.2.5. Reconocimiento de un símbolo

```
instance (Show symbol, (symbol `Describes` token), (Provides state symbol token pos))
    => Symbol (Ph state) symbol token where
    sym a = Ph (\ k st -> case splitState a st of
                            (pos, Just (n, t, ss)) -> if eqSymTok a t
                                                      then Step n (k t ss)
                                                      else Fail [Error (Just
                                                          (show pos)) (show a)]
                            (pos, Nothing)         -> Fail [Error (Just (show
                                                           pos)) (show a)])

instance (Show symbol, (symbol `Describes` token), (Provides state symbol token pos))
    => Symbol (Pm state) symbol token where
    sym a = Pm (sym a) (sym a) (sym a)
```

Dado que la función `sym` es muy similar para los tres tipos de parser, sólo se muestra el de parser histórico. Lo único que cambia es cómo se guarda el progreso en la estructura `Step`. Básicamente la función consiste en obtener un nuevo estado a partir del símbolo a reconocer, y en caso de éxito, es decir, que el mismo corresponda con el token obtenido, registrar el progreso realizado. En caso contrario, guardar el error el cual contiene la posición donde el parser falló, y el símbolo que se esperaba.

El parser que combina los tres una vez más delega en cada uno de los parsers que lo componen.

5.3. Extensiones

5.3.1. Comportamiento Greedy

Una de las primeras problemáticas que surge a partir de sólo aquello que se explicó de este nuevo parser, es que en caso de ambigüedad el mismo fracasa. Es por eso que se decide agregar una primera extensión que soporta comportamiento *greedy* del mismo, lidiando así con la ambigüedad.

```
class Greedy p where
    (<<|>) :: p a -> p a -> p a

instance Greedy (Ph st) where
    Ph p <<|> Ph q = Ph (\ k st -> p k st `best_gr` q k st)
```

```
best_gr :: Steps a -> Steps a -> Steps a
l `best_gr` r = norm l `best_gr'` norm r
                    where  l@(Step _ _) `best_gr'` _            = l
                           _            `best_gr'` r@(Step _ _) = r
                           l            `best_gr'` r            = l `best` r

class Try p where
    (<<<|>) :: p a -> p a -> p a
    try :: p a -> p a

instance Try (Ph st) where
    Ph p <<<|> Ph q = Ph (\ k st -> let l = p k st
                                    in maybe (l `best` q k st) id (hasSuccess id l))
    try (Ph p)      = Ph (p . (((Success .) .)))

   hasSuccess :: (Steps a -> b) -> Steps a -> Maybe b
   hasSuccess f (Step    n l ) = hasSuccess (f . Step n) l
   hasSuccess f (Apply   g l ) = hasSuccess (f . (Apply g)) l
   hasSuccess f (Success l   ) = Just (f l)
   hasSuccess f (Fail    _   ) = Nothing
```

Dada la similitud con los otros parser, sólo se muestra la implementación para el parser histórico.

La primera clase `Greedy`, como se puede ver en la definición de `best_gr`, se decide tomar aquella rama por la cual se garantice al menos una unidad de progreso, o la rama izquierda en caso que ambas puedan progresar inicialmente. Esto implica que en caso de que falle la rama elegida, no se explorará la otra.

Por el otro lado, la clase `Try` permite realizar una exploración en profundidad de una primera parte de la rama izquierda del operador `<<<|>` para decidir si se debe seguir explorando la misma. En caso de fracasar, se retoma la exploración en paralelo de las distintas ramas. Esto se logra agregando un marcador de éxito, `Success`, al final de la parte sobre la cual se va a realizar exploración en profundidad, donde a la hora de evaluar la función `<<<|>` se lo busca en la rama izquierda.

Es importante destacar que por más que la rama izquierda no tenga el marcador `Success`, podría ser que se pueda avanzar más que por la rama derecha. Es por eso que es necesario hacer exploración a lo ancho incluso cuando no se halló el `Success`.

5.3.2. Soportar Ambigüedad

Otra posible solución al problema descrito en la sección anterior es permitir la ambigüedad, y llevar un registro de los posibles caminos al igual que se lograba con la mónada lista. Para eso se introduce una clase `Ambiguous`:

```
  data Steps a where
      ...
      Endh    :: ([a], [a] -> Steps r) -> Steps (a, r)            -> Steps (a, r)
      Endf    :: [Steps a] -> Steps a                             -> Steps a

  best' :: Steps a -> Steps a -> Steps a
  ...
  Endf as l         `best'` Endf bs r      = Endf (as ++ bs) (l `best` r)
  Endf as l         `best'` r              = Endf as         (l `best` r)
  l                 `best'` Endf bs r      = Endf bs         (l `best` r)
  Endh (as, k_st) l `best'` Endh (bs, _) r = Endh (as ++ bs, k_st) (l `best` r)
  Endh as         l `best'` r              = Endh as               (l `best` r)
  l                 `best'` Endh bs      r = Endh bs               (l `best` r)
  ...

  norm :: Steps a -> Steps a
  ...
  norm (Apply f (Endf    as l  ) ) = Endf (map (Apply f) as) (Apply f l)
  norm (Apply f (Endh    _  _  ) ) = error "Found Endh on the loose when calling
                                            norm!"
  ...

class Ambiguous p where
    amb :: p a -> p [a]

instance Ambiguous (Pr st) where
    amb (Pr pr) = Pr (\ k -> removeEndh . pr (\ st' -> Endh ([undefined], \ _ -> k st') noAlts))

instance Ambiguous (Ph st) where
    amb (Ph p) = Ph (\ k -> removeEndh . p (\ a st' -> Endh ([a], \ as -> k as st') noAlts))

instance Ambiguous (Pf st) where
    amb (Pf p) = Pf (\ k inp -> combineValues . removeEndf $ p (\ st -> Endf [k st] noAlts) inp)

instance Ambiguous (Pm st) where
    amb (Pm pr ph pf) = Pm (amb pr) (amb ph) (amb pf)




removeEndh :: Steps (a, r) -> Steps r
removeEndh (Step  n          l  ) = Step n (removeEndh l)
removeEndh (Fail  ms            ) = Fail ms
```

```
removeEndh (Apply f          l  ) = error "no apply in history"
removeEndh (Endh  (as, k_st) r  ) = k_st as `best` removeEndh r

removeEndf :: Steps r -> Steps [r]
removeEndf (Step  n      l )  = Step n (removeEndf l)
removeEndf (Fail  ms       )  = Fail ms
removeEndf (Apply f      l )  = Apply (map' f) (removeEndf l)
removeEndf (Endf  (s:ss) r )  = Apply (:(map eval ss)) s `best` removeEndf r

combineValues :: Steps[(a, r)] -> Steps ([a], r)
combineValues lar = Apply (\lar' -> (map fst lar', snd (head lar'))) lar

map' :: (a -> b) -> [a] -> [b]
map' f ~(x:xs) = f x : map f xs
```

Por un lado, para el parser histórico se introduce el constructor `Endh`, el cual guarda en su primer parámetro un par el listado de posibles alternativas y la función a aplicar sobre las mismas, que como la idea es no repetir la continuación para cada opción, esta se aplica al listado de alternativas; y en el segundo las otras ramas que tienen un marcador `Endh` más profundo. De esta forma, al parser que se lo aceptará como ambiguo, se le introduce el constructor `Endh` que por un lado envuelve en una lista al token reconocido, y por otro guarda como función aquella que recibe el listado de alternativas y lo aplica a la continuación y lo que restó por consumir. La función `removeEndh` es la que se ocupa por un lado unir los resultados de alternativas donde el marcador `Endh` se encuentra en la misma profundidad, de eliminar dicho marcador y resumir el parseo a partir de las varias alternativas, además de seguir buscando en aquellas alternativas que aún no completaron.

El parser reconocedor funciona de manera similar al histórico, con la única diferencia que como no reconoce tokens, en el par que toma el constructor `Endh` se utiliza la lista que contiene $\perp$, y la función de la segunda componente simplemente consiste en llamar a la continuación.

En el caso del parser futuro, se introduce el constructor `Endf`, que contiene el listado de alternativas posibles futuras. Este se lo agrega al resultado de evaluar la continuación, y por otro lado las funciones `removeEndf` y `combineValues` buscan quitar estos marcadores y unir los resultados. Es interesante notar que el caso de `Apply` de la función `removeEndf` logra que se preserve la capacidad de obtener resultados online, ya que fuerza la evaluación lazy del listado de alternativas, y por ende se puede obtener aquel resultado que no dependa de dicho listado.

5.3.3. Reporte de Errores

Como se pudo ver previamente en la función `sym` y en la definición del constructor `Fail`, en caso de error se guarda el símbolo y la posición donde no se pudo seguir avanzando. Estos errores se convierten en texto con las siguientes funciones:

```
showErrors :: [Error] -> String
showErrors errors = intercalate "\n" (map showError (convertPairToList (map
errorToPair errors)))

errorToPair :: Error -> (Maybe String, String)
errorToPair (Error pos sym) = (pos, sym)

showError :: (Maybe String, [String]) -> String
showError (Just pos, x:[]) = "Expected: " ++ x ++ " at position " ++ pos
showError (Just pos, xs  ) = "Expected one of: [" ++ (intercalate ", " xs) ++ "] at
position " ++ pos
showError (Nothing , xs  ) = concat xs

convertPairToList:: Ord k => [(k, v)] -> [(k, [v])]
convertPairToList ls = toList (fromListWith (++) (map (\ (x, y) -> (x, [y])) ls))
```

Como se vio previamente, se decidió que la posición sea un tipo polimórfico definido por la dependencia funcional de la clase `Provides`. De esta forma, el usuario puede elegir libremente cómo llegar registro de la misma.

Hubiese sido interesante proveer una forma de poder manejar los mensajes de error a mostrar.

5.4. Reutilización de los Combinadores

Algo importante a destacar es que se pueden reutilizar los combinadores de parsers mencionados previamente definiendo una clase auxiliar `CombinableParser`:

```
class (Applicative (p st), Functor (p st), Alternative (p st)) => CombinableParser
p st where

instance CombinableParser Pm st
```

Luego aquellos verdaderos combinadores de parsers se pueden fácilmente ajustar en su descripción de tipo. Un ejemplo es el siguiente:

```
(<:*>) :: CombinableParser p st => p st a -> p st [a] -> p st [a]
p <:*> ps = (:) <$> p <*> ps
```

Sin embargo, los parsers elementales en su mayoría necesitan hacer referencia al tipo `Pm` y además las restricciones referidas a la clase `Describes` y `Provides`. Un ejemplo de esto es la función `token`:

```
token :: (Describes Char a, Provides st Char a pos) => String -> Pm st [a]
token (xs) = pure_token xs <* spaces
```

6. BotParser

Como ejemplo de uso de la biblioteca de parsers desarrollada, se decidió implementar un parser para el archivo de configuración de un bot de IRC.

El bot de IRC inicialmente recibía como archivo de configuración un JSON. Sin embargo, como los usuarios normalmente carecían conocimientos de programación, tenían dificultades con la estructura del JSON. Es por esa razón que se decidió definir una gramática que sea leíble y entendible por el usuario, la cual pueda convertirse en una estructura para luego ser compilada en el archivo de configuración JSON correspondiente.

6.1. Gramática

Las reglas de producción de la gramática son las siguientes, donde los símbolos terminales están escritos en minúscula y entre comillas, y los no terminales en mayúscula:

```
CONTENIDO := "{" REGLAS "}"

REGLAS :=  REGLA
        |  REGLA REGLAS

REGLA := "si" CONDICION ENTONCES ACCION SINOACCION "."

ENTONCES := " entonces " | λ

CONDICION {usuario} := Y_CONDICION {usuario} " o " CONDICION {usuario}
                    | Y_CONDICION {usuario}

Y_CONDICION {usuario} := CONDICION_ATOM {usuario} " y " Y_CONDICION {usuario}
                       | CONDICION_ATOM {usuario}

CONDICION_ATOM {usuario} := DICE_IGUAL {usuario}
                          | DICE_DISTINTO {usuario}
                          | DICE_RELACIONAL {usuario}

DICE_IGUAL {usuario} := "lo que dice " usuario " es " MENSAJE
                      | usuario " dice " MENSAJE
                      | "lo que dice " USUARIO=>usuario " es " MENSAJE
                      | USUARIO=>usuario " dice " MENSAJE

DICE_DISTINTO {usuario} := "lo que dice " usuario " no es " MENSAJE
                         | usuario " no dice " MENSAJE
                         | "lo que dice " USUARIO=>usuario " no es " MENSAJE
                         | USUARIO=>usuario " no dice " MENSAJE
```

```
DICE_RELACIONAL {usuario} := "lo que dice " usuario " contiene " MENSAJE
                           | "lo que dice " usuario EMP_TERM "con " MENSAJE
                           | "lo que dice " USUARIO=>usuario " contiene " MENSAJE
                           | "lo que dice " USUARIO=>usuario EMP_TERM "con " MENSAJE

EMP_TERM := " empieza " | " termina "

ACCION := "decir " MENSAJE

SINOACCION := "sino " MENSAJE
            | "sino " ACCION

MENSAJE := "\"" anyChar "\""

USUARIO := "\"" a-zA-Z0-9 "\""
```

Algo interesante de esta gramática es la posibilidad de no especificar el usuario en una condición, y que el mismo quede definido por la última aparición.

6.2. Desarrollo

A continuación se encuentra la implementación del estado y funciones para poder probar el parser desarrollado:

```
class DescribesPosition a where
    updatePos :: Position -> a -> Position

data StrState s = StrState {
                  input :: [s],
                  prog  :: !Int,
                  pos   :: !Position}

type Position = (Int, Int)

instance DescribesPosition a => Provides (StrState a) a a Position where
    splitState s (StrState []     _    pos ) = (pos, Nothing)
    splitState s (StrState (t:ts) prog pos ) = (pos, Just (prog + 1, t, StrState ts (prog + 1)
                                                          newpos))
                                             where newpos = updatePos pos s

instance Eof (StrState a) where
    eof (StrState l _ _) = null l

instance DescribesPosition Char where
    updatePos (l, c) '\t' = (l    , c + 4)
    updatePos (l, c) '\n' = (l + 1, 0    )
    updatePos (l, c) _    = (l    , c + 1)
```

```
stringToStrState :: String -> StrState Char
stringToStrState ls = StrState ls 0 (0, 0)

test :: Parser (StrState Char) a -> String -> a
test p str = parse p (stringToStrState str)

  type BotState = StrState Char
```

El estado está compuesto por la cadena de símbolos, la posición en la que se encuentra el parser, la cual está representada por un par que representa las líneas y columnas, y el progreso realizado. La función `eof` indica el final una vez que se consumió toda la entrada.

La estructura que se construirá está definida con los siguientes tipos:

```
data Rule          = IfThenElse Condition Action Action
    deriving Show

data Action        = Say String
                   | None
    deriving Show

data Condition     = Not   Condition
                   | And   Condition      Condition
                   | Or    Condition      Condition
                   | Atom  AtomCondition
    deriving Show

data AtomCondition = SaysEq       String Username
                   | SaysContains String Username
                   | SaysStarts   String Username
                   | SaysEnds     String Username
    deriving Show

type Username = String
```

Se puede ver que la misma es muy similar a la gramática definida.

Se muestran a continuación algunas de las componentes del parser a modo de ejemplo, aquellas más interesantes, y otras para mostrar lo similar que se ve el código del mismo respecto a la gramática, siendo así legible y sin la necesidad de un parser para la gramática:

```
...

pDiceEs :: Username -> Parser BotState (Username, Condition)
pDiceEs usrAnterior = toUserPair ((Atom .) . SaysEq) <$>
                        (     optUsuario usrAnterior <* pDice
                          <|> pLoQueDice *> optUsuario usrAnterior <* pEs)
                        <*> pMensaje
...

pSiEntonces :: Parser BotState Rule
pSiEntonces = IfThenElse <$>
              (pSi *> pCumpleCondicion)
              <*> (pEntonces ?*> pHacerAlgo)
              <*> pSinoHacerOtraCosa
              <*  pFin
...

condicionBool :: Username -> Parser BotState (Username, Condition)
condicionBool = foldr propagarUsuario condAtom [oOp, yOp]

...

propagarUsuario :: (Alternative p, Monad p) => p (b -> b -> b) -> (a -> p (a, b))
-> a -> p (a, b)
propagarUsuario op pf a = do {
      (a', b')   <- pf a
    ; f          <- op
    ; (a'', b'') <- propagarUsuario op pf a'
    ; return (a'', (f b' b''))} <|> pf a

...
```

La función `condicionBool` define el orden de precedencia de los operadores booleanos en función del orden en el que se encuentran en la lista, lo cual hace muy fácil agregar operadores o cambiar el orden de precedencia.

La función `propagarUsuario` es aquella que a partir de un operador, un parser “básico” que recibe al usuario, y un valor inicial de usuario, logra propagar el mismo de izquierda a derecha. Esto se ve en la llamada recursiva en el tercer paso de la *do-notation*. En adición, el primero consiste en evaluar el parser “básico” con el usuario recibido por parámetro, mientras que el segundo en obtener la función semántica que genera el parser del operador. El último paso de la do-notation devuelve el usuario devuelto por la recursión, ya que es el último de la derecha, y el resultado de aplicar la operación a los dos resultados obtenidos. Finalmente, está como opción sólo parsear el parser “básico” con el usuario recibido, la cual hace de caso base a la recursión.

7. Conclusiones

Como se pudo ver, a partir de un lenguaje funcional es posible desarrollar una biblioteca de parsers eficiente la cual tiene varias ventajas. Una de ellas es que se logra utilizar todo el potencial del lenguaje de programación que hace de host, que en este caso es Haskell. Por otro lado, no es necesario pasar por una etapa intermedia donde se escribe la gramática, y luego un generador de parsers como yacc la parsea y genera un parser a partir de la misma; sino que se puede definir el parser directamente, y la estructura del mismo es similar a la gramática sobre la que se basa. Además, como el parser está compuesto modularmente a partir de otros parsers, se facilita el testeo individual de cada parte y también la posibilidad de agregar funcionalidades nuevas.

Otra ventaja que tiene realizar un parser de manera funcional de la manera que se desarrolló, es que se pueden obtener resultados online, es decir, sin haber terminado de consumir toda la cadena de entrada. En adición, el último desarrollo también ayuda en disminuir el consumo de memoria del parser, sin embargo, un parser elaborado con la última forma puede ser más complejo y tardar más que utilizando la librería desarrollada inicialmente.

8. Bibliografía

[1] Functional Parsers, Jeroen Fokker, LNCS 952, Springer, 1995. (http://www.staff.science.uu.nl/~fokke101/article/parsers/index.html)

[2] Monadic parser combinators, Graham Hutton, 1996. (http://www.cs.nott.ac.uk/~pszgmh/bib.html#monparsing)

[3] Monadic parsing in Haskell, Graham Hutton, JFP, 1998. (http://www.cs.nott.ac.uk/~pszgmh/bib.html#pearl)

[4] Combinator Parsing: A Short Tutorial, S. Doaitse Swierstra (http://www.cs.uu.nl/research/techreps/repo/CS-2008/2008-044.pdf)