Parser Combinators

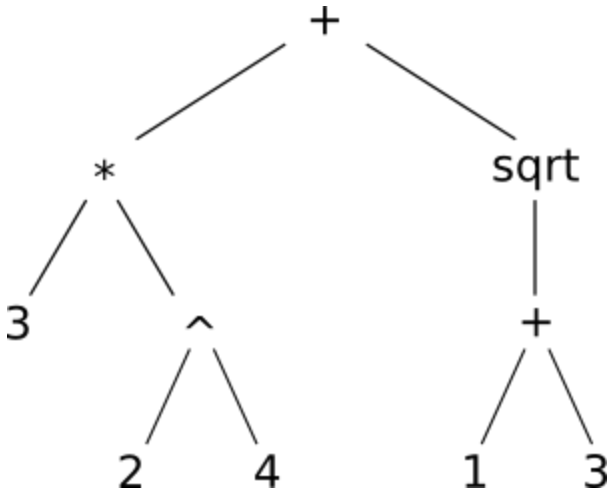
TAdeP - 1C 2023 - Trabajo Práctico Grupal: Objeto/Funcional

https://en.wikipedia.org/wiki/Parser_combinator

Introducción

Un parser es algo que toma como entrada un texto, y que como salida produce algo más fácil de interpretar o manipular en el lenguaje de programación que se esté usando, es una forma de traducir entre un lenguaje y otro.

Por ejemplo, un parser podría agarrar el texto “3 * 2 ^ 4 + sqrt(1 + 3)” y convertirlo en un árbol que tenga la siguiente forma:



Un **parser combinator** es una función de orden superior que recibe parsers como parámetro y retorna nuevos parsers. Esto va en línea con la filosofía de funcional que se basa en hacer componentes chiquitos que sirven como bloques de construcción, y a partir de combinar esos construir abstracciones más grandes.

IMPORTANTE: Este trabajo práctico debe implementarse de manera que se apliquen los principios del paradigma híbrido objeto-funcional enseñados en clase. No alcanza con hacer que el código funcione en objetos, hay que aprovechar las herramientas funcionales, poder justificar las decisiones de diseño y elegir el modo y lugar para usar conceptos de un paradigma u otro.
Se tendrán en cuenta para la corrección los siguientes aspectos:

- Uso de Inmutabilidad vs. Mutabilidad.
- Uso de Polimorfismo paramétrico (Pattern Matching) vs. Polimorfismo Ad-Hoc.
- Aprovechamiento del polimorfismo entre objetos y funciones y Orden Superior.
- Uso adecuado del sistema de tipos.
- Manejo de herramientas funcionales.
- Cualidades de Software.
- Diseño de interfaces y elección de tipos.

Descripción General del Dominio

Primeros parsers

Para empezar, se pide implementar algunos parsers que podríamos usar para realizar parseos simples, como chequear por una cadena de caracteres en particular.
Cada uno de estos parsers tiene que recibir un string sobre el cual trabajar y retornan un resultado que puede ser un **error de parseo** o un **parseo exitoso** (que contiene el elemento parseado junto con la sección del string que no fue consumida).
NOTA: Este TP se debe implementar como una implementación independiente de la entrega anterior y no debe implementarse basándose en ella.

anyChar: Lo primero que vamos a implementar va a ser un parser que lee cualquier caracter de un texto, pero solo uno, entonces, si a este parser le paso el texto “hola”, debería devolver un parseo exitoso con el carácter h como resultado. Sin embargo, si le paso un texto vacío (“”), debería fallar.

char: Ahora, queremos poder tener parsers que nos permitan decir qué carácter esperamos encontrar. Por ejemplo, queremos poder tener un parser del char ‘c’, al cuál si le paso como parámetro el texto “chau”, da un resultado exitoso con el valor ‘c’, pero si le paso el texto “hola”, debería fallar porque “hola” no empieza con c.

void: void es un parser que lee un carácter pero lo descarta. Lo que esperamos obtener como resultado de void es el valor Unit. Entonces, si lo usasemos para parsear “hola”, devolvería un resultado exitoso con Unit como valor. Pero ojo, si le pasasemos un string vacio “”, fallaría como cualquiera de los parsers anteriores.

letter: debería retornar un resultado exitoso para cualquier caracter que sea una letra (minúscula o mayúscula). Por ejemplo, si se usa para parsear “hola”, debería parsear ‘h’, pero si se usa para parsear “1234”, debería fallar.

digit: debería devolver un resultado exitoso con el caracter parseado, si el carácter consumido es 0, 1, 2, 3, 4, 5, 6, 7, 8 o 9.

alphaNum: parsea exitosamente cualquier cosa que **letter** o **digit** parseen exitosamente, devolviendo el mismo resultado que esos parsers. En cualquier otro caso falla.

string: este caso es parecido a **char**, porque necesita saber qué string es el que esperamos parsear, pero se diferencia a los parsers que aparecieron hasta ahora porque consume tantos caracteres como tenga el string esperado, y en el caso de éxito no debería tener un carácter como resultado, si no un string.
Por ejemplo, queremos poder obtener un parser con el string esperado “hola”, tal que si lo usamos para parsear “hola mundo!”, de un resultado exitoso con el valor “hola”, pero si lo usasemos para intentar parsear “holgado” falle.

Combinators

Ahora que ya tenemos algunos parsers, se requiere implementar los combinadores de los mismos, con los cuales podemos crear parsers más complejos.

OR Combinator

<|>: este es el primer combinador propiamente dicho, a partir de dos parsers crea uno que trabaja de la siguiente manera: si el primer parser retorna un resultado, ese es el resultado, si no, intenta con el segundo parser y retorna lo que retornaría este último. Un ejemplo de creación de un parser usando este combinador sería:

```
val aob = char('a') <|> char('b')
```

Si le pasamos “arbol” a aob, debería poder parsear ‘a’, y si le pasasemos “bort” a aob debería poder parsear ‘b’.

Concat Combinator

<>: un parser combinator que secuencia dos parsers. Es decir, crea uno nuevo que primero parsea lo que parsearía el primero, y usando el resto del texto aplica el segundo parser.
Esperamos que el valor que devuelva en caso exitoso tenga una tupla con los dos valores parseados.
Por ejemplo:

```
val holaMundo = string("hola") <> string("mundo")
```

Si a holaMundo le pasasemos “holamundo”, debería producir un resultado exitoso con los valores “hola” y “mundo” en una tupla. Si le pasásemos “holachau”, debería fallar, porque o

parsea todo o no parsea nada.

Rightmost Combinator

~>: (primerParser **~>** segundoParser) debería requerir que ambos parsers se hagan de manera secuencial (primero primerParser y luego segundoParser) pero me da sólo el resultado de segundoParser.

Leftmost Combinator

<~: (primerParser **<~** segundoParser) parsea en el mismo orden que el anterior, pero me da el resultado de primerParser.

Parsers parte II

Ahora queremos definir operaciones sobre los parsers para obtener parsers más complejos.

satisfies: A partir de un parser y una condición, nos permite obtener un parser que funciona sólo si el parser base funciona y además el elemento parseado cumple esa condición.

opt: convierte en opcional a un parser. Es decir, el nuevo parser siempre debería dar un resultado exitoso, pero si el parser original hubiese fallado, el resultado no contendrá ningún valor y no consumirá ningún caracter del input.
Ejemplo:

```
val talVezIn = string("in").opt
// precedencia parsea exitosamente las palabras "infija" y "fija"
val precedencia = talVezIn <> string("fija")
```

Si a precedencia le pasasemos “fija”, deberia devolver una tupla con un valor vacío y con el valor “fija”, porque talVezIn no habría consumido ningún carácter del texto original.

*: la clausura de Kleene se aplica a un parser, convirtiéndolo en otro que se puede aplicar todas las veces que sea posible o 0 veces. El resultado debería ser una lista que contiene todos los valores que hayan sido parseados (podría no haber ninguno).

+: es como la clausura de Kleene pero requiere que el parser se aplique al menos UNA vez.

sepBy: toma dos parsers: un parser de contenido y un parser separador, parsea 1 o más veces el parser de contenido (similar a la cláusula de kleene+) pero entre cada una aplica el parser separador.
Ejemplo:

```
val numeroDeTelefono = integer.sepBy(char('-'))
```

Integer sería un parser aplica a 1 o más dígitos, y devuelve un Int con el valor del número

representado por los dígitos. El parser resultado del sepBy debería funcionar si le paso “4356-1234” pero no si le paso “4356 1234”. Al parsear debería devolver una lista cuyos elementos serán el resultado de aplicar el parser de contenido a cada “grupo”.

const: recibe un valor y convierte a un parser en otro diferente que parsea de la misma manera (falla cuando fallaría el original, funciona cuando funcionaría el original) pero que como elemento parseado devuelve el valor constante que se le pasó a **const**, en vez de lo que devolvería el parser base.
Ejemplo:

```
// aplicado sobre el string "true" retorna el boolean
// true como valor parseado, en cualquier otro caso falla
val trueParser = string("true").const(true)
```

map: Dada una función de transformación y un parser, retorna un nuevo parser que parsea lo mismo que el original y convierte el valor parseado utilizando la función recibida.

```
case class Persona(nombre: String, apellido: String)
val personaParser = (alphaNum.* <> (char(' ') ~> alphaNum.*))
           .map { case (nombre, apellido) => Persona(nombre, apellido) }
```

Caso práctico: Musiquita

Tenemos modeladas notas músicales que podemos usar para reproducir melodías, pero como crear todos los objetos que componen una melodía a mano era muy tedioso decidimos inventar un lenguaje para escribir melodías y ahora necesitamos una forma de convertir cosas escritas en ese lenguaje a nuestro modelo.

A continuacion se pide desarrollar un parser para el siguiente lenguaje:

Una melodía válida se compone de una lista de "**Tocable**"s separados por espacios.
Un parser de melodía debe retornar una instancia de tipo "**Melodia**". Para más información sobre los tipos a parsear, ver el código definido en el repo.

Los "**`Tocable`**"s que conocemos son:

- **Silencio:** los silencios se diferencian entre sí por el tiempo que tardan, que siempre es igual al de alguna figura. Los silencios que tenemos son:
  - _ : silencio de blanca
  - - : silencio de negra.
  - ~ : silencio de corchea
- **Sonido:** es un tono seguido de una figura.
  - **Tono:** Un tono es una nota precedida por un número entero, que representa la octava del tono.

    - **Nota:** Una nota es el nombre de la nota (A, B, C, D, E, F o G). Puede estar seguida de un modificador como # o b (sostenido y bemol respectivamente).
    - **Figura:** Una figura se escribe como una fracción del estilo numerador/denominador y las figuras que tenemos modeladas son:
        - Redonda (1/1), Blanca (1/2), Negra (1/4), Corchea (1/8), SemiCorchea (1/16).
    - Ejemplo: si tenemos “6A#1/4”, eso debería ser un sonido para el cual la figura es Negra y el tono está compuesto por la octava 6 y la nota A sostenido.
- **Acorde:** conocemos dos formas posibles de definir acordes:
    - Explícitos: se escriben como varios tonos separadas por ‘+’, seguidos de una figura. Ejemplo: “6A+6C#+6G1/8” es un acorde con los tonos 6A, 6C#, 6G y con la duración de una Corchea.
    - Menor o mayor: se escriben como un tono, seguido de la letra m para indicar que es un acorde menor y la letra M para indicar que es un acorde mayor. Esto debe generar una figura igual que en la forma explícita.
        - La forma de calcular los acordes pueden encontrarla en las notas, usando los mensajes menor y mayor.

      Ejemplo: “6AM1/2” es un el acorde 6 A mayor, que dura como una Blanca.

A continuación hay ejemplos de algunas melodías posibles que deberían poder ser parseadas:

Feliz cumpleaños:
4C1/4 4C1/4 4D1/2 4C1/4 4F1/2 4E1/2 4C1/8 4C1/4 4D1/2 4C1/2 4G1/2 4F1/2 4C1/8 4C1/4 5C1/2 4A1/2 4F1/8 4F1/4 4E1/2 4D1/2

Canción Bonus:
4AM1/8 5C1/8 5C#1/8 5C#1/8 5D#1/8 5C1/8 4A#1/8 4G#1/2 - 4A#1/8 4A#1/8 5C1/4 5C#1/8 4A#1/4 4G#1/2 5G#1/4 5G#1/4 5D#1/2