3. Tercera sesión: *Tipos de datos abstractos*

3.1. Objetivos

- ⋆ Recordar el concepto de gramática y su representación mediante la forma de Backus-Naur
- ⋆ Recordar el concepto de tipo de dato abstracto, ya que éstos serán la base de la construcción y manipulación de gramáticas durante el curso.
- ⋆ Construir tipos de datos basados en un modelo abstracto como una gramática.
- ⋆ Operar recursivamente sobre tipos de datos usando la técnica de caza de patrones.

3.2. Introducción

3.2.1. Gramáticas y la forma de Backus-Naur

Una gramática, es en pocas palabras, una forma de generar expresiones que pertenecen a un lenguaje. Un ejemplo de gramática puede ser:

```
S → 0S | 0A
A → 1A | 1
```

Cada renglón de la gramática es llamado producción y nos indica cómo ir sustituyendo los símbolos no terminales (`S` y `A`) para llegar a una expresión compuesta por únicamente símbolos terminales (`0` y `1`). Por ejemplo, para obtener una expresión, podemos seguir los siguientes pasos:

```
S -> 0S -> 00S -> 000S -> 0000A -> 00001A -> 000011A -> 0000111
```

A pesar que este tipo de gramáticas nos permiten generar expresiones que pertenecen a un determinado lenguaje, pueden no ser intuitivas en la práctica. Para representar las gramáticas o especificaciones de un lenguaje de programación es común usar la llamada forma de Backus Naur (BNF), ésta es llamada así en honor a su inventor John Backus y a Peter Naur, quien la modificó para utilizarla en las especificaciones del lenguaje de programación ALGOL.

Las producciones de este tipo de gramáticas tienen un solo símbolo no terminal en su lado izquierdo. En lugar de utilizar el símbolo → en una producción utilizaremos el símbolo `:=`. Colocamos los símbolos no terminales entre los símbolos, `< >`, y listaremos todos los lados derechos de las producciones en una misma línea o mediante saltos de línea, separados por barras.

Por ejemplo, la gramática anterior, podría representarse como sigue en BNF:

```
<S> := 0<S>
     | 0<A>

<A> := 1<A>
     | 1
```

Otro ejemplo sería la representación de los números naturales:

```
<Nat> := cero
       | s <Nat>
```

3.2.2. Tipos de datos abstractos en Racket

El dialecto plai de Racket cuenta con un conjunto de bibliotecas especiales para el desarrollo de intérpretes. Dentro de estas herramientas tenemos una herramienta para definir tipos de datos y otra para manipularlos.

La sintaxis para crear un tipo de dato es la siguiente:

```
(define-type Nombre
    [constructor (p Tipo?)*]+)
```

Por ejemplo, para representar a los números naturales escribimos:

```
(define-type Nat
    [cero]
    [sucesor (n Nat?)])
```

Una característica importante es que al definir estos tipos de datos, el lenguaje nos regala por defecto varias cosas adicionales:

1. Un predicado para revisar que una expresión es efectivamente de ese tipo, en este caso el predicado es `Nat?` y lo podemos usar recursivamente sobre la misma definición.
2. También nos regala un predicado por cada constructor definido, en este caso obtenemos los predicados `cero?` y `sucesor?`.
3. Para cada parámetro de un constructor, nos regala una función que obtiene dicho valor, en este caso obtenemos la función `sucesor-n`.
4. Por cada parámetro de un constructor, nos regala una función para modificar el valor de dicho parámetro en este caso obtenemos la función `set-sucesor-n!`.

3.2.3. Caza de patrones sobre tipos de datos abstractos

Al igual que en matemáticas, la definición recursiva de tipos de datos permite definir funciones sobre los mismos utilizando la técnica de casamiento de patrones.

En Racket, tenemos dos formas de procesar estructuras recursivas: mediante la primitiva `type-case` que provee el dialecto `plai` o mediante `match`, previamente revisado.

Veamos algunos ejemplos:

```
;; Suma usando type-case
(define (suma1 nat1 nat2)
    (type-case Nat nat1
        [cero () nat2]
        [sucesor (n) (sucesor (suma n nat2))]))

;; Suma usando match
(define (suma2 nat1 nat2)
    (match nat1
        [(cero) nat2]
        [(sucesor n) (sucesor (suma n nat2))]))

;; Producto usando type-case
(define (producto1 nat1 nat2)
    (type-case Nat nat1
        [cero () (cero)]
        [sucesor (m) (suma2 nat2 (producto1 m nat2))]))

;; Producto usando match
(define (producto2 nat1 nat2)
    (match nat1
        [(cero) (cero)]
        [(sucesor m) (suma2 nat2 (producto2 m nat2))]))

;; Igualdad usando type-case
(define (iguales1 nat1 nat2)
    (type-case Nat nat1
        [cero ()
            (type-case Nat nat2
                [cero () #t]
                [else #f])]
        [sucesor (m)
            (type-case Nat nat2
                [cero () #f]
                [sucesor (n) (iguales1 m n)])]))

;; Igualdad usando match
(define (iguales2 nat1 nat2)
    (match nat1
        [(cero)
            (match nat2
                [(cero) #t]
                [else #f])]
        [(sucesor n)
            (match nat2
                [(cero) #f]
                [(sucesor m) (iguales2 n m)])]))
```

3.3. Ejercicios

1. Define el tipo de dato abstracto `Lista` usando BNF. Usa `vacia` y `nodo` en lugar del `empty` y `cons` de Racket. Tu lista debe ser de números enteros.

2. Define el tipo de dato abstracto `Lista` en Racket. Usa los constructores definidos en el ejercicio anterior.

3. Define las siguientes funciones sobre el tipo de dato `Lista` que definiste en el ejercicio anterior. Debes escribir una versión usando `type-case` y otra usando `match`:

   a) Una función `(longitud l)` para encontrar la longitud de una lista.

   b) Una función `(contiene? n l)` para determinar si el número `n` está contenido en la lista.

   c) Una función `(suma l)` que suma los elementos de la lista.

   d) Una función `(multiplica l)` que suma los elementos de la lista.

   e) Una función `(mapea f l)` que aplica la función `f` a cada elemento de la lista.

   f) Una función `(filtra p l)` que regresa una lista con los elementos que cumplen el predicado `p`.

3.4. Referencias

[1] Kenneth H. Rosen, *Matemática Discreta*, Mc Graw Hill, Quinta edición.

[2] Miranda F., Viso E., *Matemáticas Discreta*s, Las Prensas de Ciencias, Primera edición.

[3] Flatt, Matthew., *The Racket Reference*.
Disponible en: https://docs.racket-lang.org/reference/index.html