Chapter 2: Hello, Haskell

Tópicos básicos sobre el lenguaje y buenas prácticas

Ejecución del código

- Se ejecuta a través de un archivo compilado o importando un módulo a través de su REPL, *GHCi*.
- La librería estandar de Haskell es *Prelude*.
- En *GHCi* se usan expresiones que comienzan con `:` para interactuar con la REPL. Algunos de estos comandos son:
  - `:load`(`:l`). Permite cargar módulos, es decir, archivos `.hs`.
  - `:module`(`:m`). Quita los módulos cargados y recarga `Prelude`.
  - `:info`(`:i`). **MUY IMPORTANTE** Brinda la documentación sobre una función. Ejemplo

    ```
    ghci> :i (+)
    type Num :: * -> Constraint
    class Num a where
    (+) :: a -> a -> a
    ...
    -- Defined in ‘GHC.Num’
    infixl 6 +
    ```

    De donde determinamos que el operador suma es, dado un tipo `a` que pertenece a `Num`, una función que toma dos argumentos de este tipo tipo y los envía a otro del mismo tipo. Además dice donde está definida, y que el operador tiene notación (por defecto) infija (infix), es asociativa a izquiera (l de infix *l*), su precedencia es de 6 (en un rango de 0 a 9), y la función se llama +.

Sintaxis y bases

- **Todo** en Haskell es una *expresión* o una *declaración*.
  - Las expresiones pueden ser valores, combinaciones de valores, y/o funciones aplicadas a valores.
  - Las declaraciones son ligaduras (*bindings*) de alto nivel que nos permiten asignar nombres a las expresiones para referirnos a ellas después.
- Definimos la forma normal de una expresión en Haskell de forma análoga a lo forma beta normal de una expresión en el calcula lambda. A las expresiones reducbiles se les llama *redexes*.
  - Los valores son expresiones que no pueden ser reducidas. Por lo tanto son el punto final de toda reducción.
  - Haskell no evalúa todo hasta su forma normal por defecto. En cambio solo evalúa hasta su *forma normal debil* o *weak head normal form (WHNF)*. Esto significa que sólo reemplaza las ligaduras al evaluar, pero no continúa. Ejemplo:

    ```
    (\f -> (1, 2 + f)) 2

    -- Reduces to

    (1, 2 + 2)
    ```

    La evaluación de `2 + 2` no se ejecutará hasta que otra expresión lo requiera.

- Las funciones prefijas pueden usarse en notación infija rodeando el nombre de la función con backticks: `div 1 2 -> 1 `div` 2` . Las funciones binarias infijas (operadores) pueden usarse en notación prefija rodeando al operador con paréntesis: `1 + 2 -> (+) 1 2` .
- Si el nombre de una función es alfanumérico, es una función prefijo por defecto. Si su nombre es un símbolo, es infija por defecto.
- Las declaraciones de un módulo se hacen mediante la sintaxis:

```
module MyModuleName where
...
```

Estilo y formato del código

- La regla básica de indentación en Haskell es que el código que es parte de una expresión debería estar indentada bajo el comienzo de esa expresión. Más aún, partes de la expresión que están agrupadas deberían estar indentadas al mismo nivel. Ejemplos:

```
let
  x = 3
  y = 4

-- or
let x = 3
    y = 4

-- or
foo x =
    let y = x * 2
        z = x ^ 2
    in 2 * y * z
```

- Cortar una línea debería reservarse para cuando la línea supera 100 columnas en ancho.

Aritmética en Haskell

- Más allá de los operadores convencionales, hay 4 operadores que pueden ser confusos:
  - `div` y `quot`. Ambos realizan división entera, pero `div` redondea al piso del valor, mientras que `quot` redondea hacia el cero. Ejemplo:

```
(/) 10 (-3) == -3.3333
div 10 (-3) == -4 -- rounds down. floor(-3.333)=-4
quot 10 (-3) == -3 -- rounds toward zero. -3 is closer to 0 than -4.
```

  - `mod` y `rem`. `rem` retorna el residuo exacto de la operación (que bien podría ser un número negativo), mientras que `mod x n` retorna el número entre `0` y `n-1` correspondiente, en módulo `n`, al residuo.

Números negativos y sobrecarga de operadores

- El operador unario `-` en Haskell no es más que azucar sintáctica para `negate`.
- El operador `-` tiene dos posibles interpretaciones:
  - Que es usado como un alias para `negate`, de donde las siguientes expresiones son semánticamente idénticas

```
2000 + (-1234)
-- or
2000 + (negate 1234)
```

- Que es la función sustracción, por lo que la siguiente expresión usa a `-` como resta

```
2000 - 1234
```

Control de paréntesis

- Un operador que está muy presente en el código de Haskell es `($)`. Este operador tiene la menor precedencia de todos los operadores, y se define como `f $ a = f a`. A efectos prácticos, `($)` permite que se evalúe primero todo lo que está a la derecha, pudiendo ser usado para retrasar la aplicación de la función.

```
(2^) $ (*30) $ 2+2
-- Will reduce to
-- (2^) $ (*30) 4
-- (2^) 120
-- 2^120
```

- Cuando se quiere referir a una función infija sin aplicar ningún argumento, o cuando se quiere usar como un operador prefijo en vez de infijo, en ambos casos se rodea de paréntesis al operador.
- A la sintaxis más concisa que aplica parcialmente argumentos a los operadores infijos, tal como `(2^)` o `(+1)`, se le llama *sectioning*. Cuando la función es conmutativa el resultado no cambia sin importar cómo se aplica el *sectioning*, sin embargo, si la función no conmuta el orden sí importa, ya que:

```
(1/) 2 -> 0.5
(/1) 2 -> 2.0
```

  - La resta es un caso especial, ya que, debido a la sobrecarga del operador unario,

```
2 - 1 == (-) 2 1
-- but
(-2) 1 -> Error
```

  Por lo tanto el sectioning sólo funciona con el operador a derecha `(1-) 4 -> -3`, o para hacerlo por la izquierda, llamando a la función `substract`,

```
(substract 2) 1 -> -1 -- instead of (-2) 1
```

.