# Tipos de datos algebraicos

Taller de Álgebra I

Segundo cuatrimestre de 2013

# Programación funcional

- Recordemos que un tipo de datos es un conjunto dotado con una serie de operaciones sobre los elementos del conjunto.

# Programación funcional

- Recordemos que un tipo de datos es un conjunto dotado con una serie de operaciones sobre los elementos del conjunto.
- Haskell incluye varios tipos de datos primitivos (como Bool, Int, Float, etc.).
- Vamos a ver en esta clase cómo definir nuestros propios tipos de datos, por medio de un mecanismo llamado tipos de datos algebraicos.

# Programación funcional

- Recordemos que un tipo de datos es un conjunto dotado con una serie de operaciones sobre los elementos del conjunto.
- Haskell incluye varios tipos de datos primitivos (como Bool, Int, Float, etc.).
- Vamos a ver en esta clase cómo definir nuestros propios tipos de datos, por medio de un mecanismo llamado tipos de datos algebraicos.
- ¿**Por qué** nos interesa definir nuevos tipos de datos?

# Programación funcional

- Recordemos que un tipo de datos es un conjunto dotado con una serie de operaciones sobre los elementos del conjunto.
- Haskell incluye varios tipos de datos primitivos (como Bool, Int, Float, etc.).
- Vamos a ver en esta clase cómo definir nuestros propios tipos de datos, por medio de un mecanismo llamado tipos de datos algebraicos.
- ¿**Por qué** nos interesa definir nuevos tipos de datos?
  1. Representar estructuras más complejas.
  2. Representar las entidades del dominio del problema a resolver.
  3. Evitar confusiones de interpretación por parte del programador.

# Tipos de datos algebraicos

- Para crear un tipo de datos algebraico decimos qué forma va a tener cada elemento.
- Se hace definiendo constantes que se llaman constructores.
  1. Empiezan con mayscula (como los nombres de los tipos de datos).
  2. Pueden tener argumentos, pero no son funciones!
  3. Forman expresiones atómicas.

# Tipos de datos algebraicos

- Para crear un tipo de datos algebraico decimos qué forma va a tener cada elemento.
- Se hace definiendo constantes que se llaman constructores.
  1. Empiezan con mayscula (como los nombres de los tipos de datos).
  2. Pueden tener argumentos, pero no son funciones!
  3. Forman expresiones atómicas.
- Por ejemplo, el tipo algebraico **Bool** tiene dos constructores, sin argumentos:
- `True::Bool`
  `False::Bool`

# Definición de tipos algebraicos

- **Ejemplo:** Una función que dada una fecha informa a qué estación del año pertenece.

# Definición de tipos algebraicos

- **Ejemplo:** Una función que dada una fecha informa a qué estación del año pertenece.
- Esta signatura tiene un grave defecto. ¿Cuál?

```
estacion ::  Int -> Int -> Int
```

# Definición de tipos algebraicos

- **Ejemplo:** Una función que dada una fecha informa a qué estación del año pertenece.
- Esta signatura tiene un grave defecto. ¿Cuál?

```
estacion ::  Int -> Int -> Int
```

- Es mucho más expresivo, seguro y elegante definir un tipo de datos para las estaciones, con cuatro valores posibles.
- 

```
data Estacion = Verano | Otono | Invierno | Primavera
estacion ::  Int -> Int -> Estacion
estacion d m | m < 3 = Verano
estacion d m | m == 3 and d < 21 = Verano
...
```

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).
- Ejemplos:
  1. `c1 = Circ 1`

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).
- Ejemplos:
  1. `c1 = Circ 1`
  2. `c2 = Circ (4.5 - 3.5)`

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).
- Ejemplos:
  1. `c1 = Circ 1`
  2. `c2 = Circ (4.5 - 3.5)`
  3. `circulo x = Circ (x+1)`

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).
- Ejemplos:
  1. `c1 = Circ 1`
  2. `c2 = Circ (4.5 - 3.5)`
  3. `circulo x = Circ (x+1)`
  4. `r1 = Rect 2.5 3`

# Otro ejemplo

- `data Figura = Circ Float | Rect Float Float`
- Tenemos dos constructores con parámetros. Algunas figuras son círculos y otras son rectángulos.
  1. Los círculos se diferencian por un número (su radio).
  2. Los rectángulos se especifican con dos números (su base y su altura).
- Ejemplos:
  1. `c1 = Circ 1`
  2. `c2 = Circ (4.5 - 3.5)`
  3. `circulo x = Circ (x+1)`
  4. `r1 = Rect 2.5 3`
  5. `cuadrado x = Rect x x`

# Pattern matching

- La correspondencia o coincidencia de patrones es el mecanismo por el cual podemos diferenciar entre elementos construidos con distintos constructores, en el caso de un tipo algebraico.
- **Patterns**: expresiones del lenguaje formadas solamente por constructores y variables que no se repiten.
  1. `Rect x y` es un patrón
  2. `3 + x` no es un patrón
  3. `Rect x x` tampoco porque tiene una variable repetida

# Pattern matching

- La correspondencia o coincidencia de patrones es el mecanismo por el cual podemos diferenciar entre elementos construidos con distintos constructores, en el caso de un tipo algebraico.
- **Patterns**: expresiones del lenguaje formadas solamente por constructores y variables que no se repiten.
  1. `Rect x y` es un patrón
  2. `3 + x` no es un patrón
  3. `Rect x x` tampoco porque tiene una variable repetida
- **Matching**: operación asociada a un patrón.
  1. Dada una expresión, dice si la forma de la expresión coincide con el patrón.
  2. Si la correspondencia existe, entonces liga las variables del patrón a las subexpresiones correspondientes.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```
  ```

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```
- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```
- Evaluemos la expresión `area (circulo 2)`

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```
- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```
- Evaluemos la expresión `area (circulo 2)`
  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

- Evaluemos la expresión `area (circulo 2)`

  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.
  2. Primero debe evaluar `circulo2` para saber a qué constructor corresponde.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

- Evaluemos la expresión `area (circulo 2)`

  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.
  2. Primero debe evaluar `circulo2` para saber a qué constructor corresponde.
  3. La reducción da `Circ(2+1)`.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

- Evaluemos la expresión `area (circulo 2)`

  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.
  2. Primero debe evaluar `circulo2` para saber a qué constructor corresponde.
  3. La reducción da `Circ(2+1)`.
  4. Ya se puede verificar cada ecuación de `area` para buscar el matching.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

- Evaluemos la expresión `area (circulo 2)`

  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.
  2. Primero debe evaluar `circulo2` para saber a qué constructor corresponde.
  3. La reducción da `Circ(2+1)`.
  4. Ya se puede verificar cada ecuación de `area` para buscar el matching.
  5. Se logra con la primera ecuación, y `radio` queda ligada a `(2+1)`.

# Pattern matching

- ```
  area ::  Figura -> Float
  area (Circ radio) = pi * radio * radio
  area (Rect base altura) = base * altura
  ```

- ```
  circulo ::  Float -> Figura
  circulo x = Circ (x+1)
  ```

- Evaluemos la expresión `area (circulo 2)`

  1. El intérprete debe elegir cuál de las ecuaciones de `area` utilizar.
  2. Primero debe evaluar `circulo2` para saber a qué constructor corresponde.
  3. La reducción da `Circ(2+1)`.
  4. Ya se puede verificar cada ecuación de `area` para buscar el matching.
  5. Se logra con la primera ecuación, y `radio` queda ligada a `(2+1)`.
  6. Luego de varias reducciones (aritméticas) adicionales, se llega al valor de la expresión: 28.2743.

# Tipos de datos algebraicos recursivos

- Un tipo de datos algebraico se dice recursivo si el mismo tipo de datos es argumento de alguno de los constructores.

# Tipos de datos algebraicos recursivos

- Un tipo de datos algebraico se dice recursivo si el mismo tipo de datos es argumento de alguno de los constructores.
- Ejemplo:

1. `data N = Z | S N`

# Tipos de datos algebraicos recursivos

- Un tipo de datos algebraico se dice recursivo si el mismo tipo de datos es argumento de alguno de los constructores.
- Ejemplo:
  1. `data N = Z | S N`
  2. `Z` es un constructor sin argumentos.
  3. `S` es un constructor con argumentos (de tipo `N`).

# Tipos de datos algebraicos recursivos

- Un tipo de datos algebraico se dice recursivo si el mismo tipo de datos es argumento de alguno de los constructores.
- Ejemplo:
  1. `data N = Z | S N`
  2. `Z` es un constructor sin argumentos.
  3. `S` es un constructor con argumentos (de tipo `N`).
- Elementos del tipo `N`:

```
Z , S Z , S (S Z) , S (S (S Z)) , ...
```

# Tipos de datos algebraicos recursivos

- Un tipo de datos algebraico se dice recursivo si el mismo tipo de datos es argumento de alguno de los constructores.
- Ejemplo:
  1. `data N = Z | S N`
  2. `Z` es un constructor sin argumentos.
  3. `S` es un constructor con argumentos (de tipo `N`).
- Elementos del tipo `N`:

```
Z   ,   S Z   ,   S (S Z)   ,   S (S (S Z))   ,   ...
↓       ↓         ↓             ↓
0       1         2             3
```

- Este tipo de datos puede representar a los números naturales.

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- `suma :: N -> N -> N`

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z =
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) =
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z =
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) =
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m =
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m = True
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m = True
  menorOIgual (S n) Z =
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m = True
  menorOIgual (S n) Z = False
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m = True
  menorOIgual (S n) Z = False
  menorOIgual (S n) (S m) =
  ```
  ```

# Recursión estructural

- Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.
- La recursión se hace sobre la estructura de los datos. Las invocaciones recursivas se hacen sobre expresiones "de forma más simple".
- ```
  suma ::  N -> N -> N
  suma n Z = n
  suma n (S m) = S (suma n m)
  ```
- ```
  producto ::  N -> N -> N
  producto n Z = Z
  producto n (S m) = suma n (producto n m)
  ```
- ```
  menorOIgual ::  N -> N -> Bool
  menorOIgual Z m = True
  menorOIgual (S n) Z = False
  menorOIgual (S n) (S m) = menorOIgual n m
  ```
  ```

# Otro ejemplo

- Veamos un tipo algebraico recursivo no paramétrico.
- `data P = T | F | A P P | O P P | N P`

# Otro ejemplo

- Veamos un tipo algebraico recursivo no paramétrico.
- `data P = T | F | A P P | O P P | N P`
- Tiene cinco constructores:
  1. `T` y `F` son constructores sin argumentos.
  2. `A` y `O` son constructores con dos argumentos (de tipo `P`).
  3. `N` es un constructor con un argumento (de tipo `P`).

# Otro ejemplo

- Veamos un tipo algebraico recursivo no paramétrico.
- `data P = T | F | A P P | O P P | N P`
- Tiene cinco constructores:
  1. `T` y `F` son constructores sin argumentos.
  2. `A` y `O` son constructores con dos argumentos (de tipo `P`).
  3. `N` es un constructor con un argumento (de tipo `P`).
- Elementos del tipo `P`:

```
T , F , A T F , N (A T F) , ...
```

# Otro ejemplo

- Veamos un tipo algebraico recursivo no paramétrico.
- `data P = T | F | A P P | O P P | N P`
- Tiene cinco constructores:
  1. `T` y `F` son constructores sin argumentos.
  2. `A` y `O` son constructores con dos argumentos (de tipo `P`).
  3. `N` es un constructor con un argumento (de tipo `P`).
- Elementos del tipo `P`:

| `T` | , | `F` | , | `A T F` | , | `N (A T F)` | , | ... |
|---|---|---|---|---|---|---|---|---|
| ↓ | | ↓ | | ↓ | | ↓ | | |
| true | | false | | (true $\wedge$ false) | | $\neg$(true $\wedge$ false) | | |

- Este tipo puede representar a las fórmulas proposicionales.

# Funciones sobre fórmulas proposicionales

- `contarAes :: P -> Int`

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       =
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   =
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) =
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) =
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   =
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   = not (valor x)
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   = not (valor x)
  valor (A x y) =
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   = not (valor x)
  valor (A x y) = (valor x) && (valor y)
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   = not (valor x)
  valor (A x y) = (valor x) && (valor y)
  valor (O x y) =
  ```

# Funciones sobre fórmulas proposicionales

- ```
  contarAes ::  P -> Int
  contarAes T       = 0
  contarAes F       = 0
  contarAes (N x)   = contarAes x
  contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
  contarAes (O x y) = (contarAes x) + (contarAes y)
  ```

- ```
  valor ::  P -> Bool
  valor T       = True
  valor F       = False
  valor (N x)   = not (valor x)
  valor (A x y) = (valor x) && (valor y)
  valor (O x y) = (valor x) || (valor y)
  ```

# Listas

- Se trata de un tipo de datos algebraico recursivo paramétrico.

# Listas

- Se trata de un tipo de datos algebraico recursivo paramétrico.
- `data List a = Nil | Cons a (List a)`
- Interpretamos ...
  1. `Nil` como la lista vacía.
  2. `Cons x l` como la lista que resulta de agregar `x` como primer elemento de `l`.

# Listas

- Se trata de un tipo de datos algebraico recursivo paramétrico.
- `data List a = Nil | Cons a (List a)`
- Interpretamos ...
  1. `Nil` como la lista vacía.
  2. `Cons x l` como la lista que resulta de agregar `x` como primer elemento de `l`.
- Por ejemplo, `List Int` es el tipo de las listas de enteros.
  1. `Nil` es una lista con cero elementos.
  2. `Cons 2 Nil` es una lista con un elemento, el 2.
  3. `Cons 3 (Cons 2 Nil)` es una lista con primer elemento 3 y segundo elemento 2.

# Notación de listas en Haskell

- `List a` se escribe `[a]`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- Son equivalentes:
  1. `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- Son equivalentes:
  1. `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`
  2. `(2 : (3 : (2 : (0 : []))))`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- Son equivalentes:
  1. `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`
  2. `(2 : (3 : (2 : (0 : [])))`
  3. `2 : 3 : 2 : 0 : []`

# Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- Son equivalentes:
  1. `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`
  2. `(2 : (3 : (2 : (0 : [])))`
  3. `2 : 3 : 2 : 0 : []`
  4. `[2,3,2,0]`

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     =
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) =
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     =
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
sum (x:xs) =
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

- Concatenar dos listas:

```
(++) ::  [a] -> [a] -> [a]
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

- Concatenar dos listas:

```
(++) ::  [a] -> [a] -> [a]
[]     ++ ys = ys
```

# Funciones sobre listas

- Longitud de una lista:

```
length ::  [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

- Suma de los elementos de una lista de enteros:

```
sum ::  [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

- Concatenar dos listas:

```
(++) ::  [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Ejercicios

1. Escribir una función que implemente la resta entre dos números del tipo `N`.
2. Escribir una función que cuente la cantidad de constantes (true y false) de una fórmula proposicional.
3. Escribir una función que cuente la cantidad de valores positivos de una lista de enteros.
4. Escribir una función que reciba una lista $l$ y un valor $x$, y que determine el valor $x$ está en la lista $l$.
5. Escribir una función que reciba una lista y retorne otra lista igual a la recibida, pero sin el segundo elemento.
6. Escribir una función que reciba una lista y retorne otra lista igual a la recibida, pero sin los elementos negativos.
7. Escribir una función que determine si una lista es palíndroma (capicúa).