Guía de lenguajes

Aclaraciones

- (*) Significa que sólo funciona para listas (Wollok)
- (**) Significa que está declarada en Data.List (Haskell)
- (***) Significa que el tipo presentado acá es una versión simplificada del tipo real (Haskell)
- NA significa “No Aplica”. En otras palabras, o no existe o no se recomienda su uso.

Elementos Comunes

Comentarios

Wollok	Haskell	Prolog

`// un comentario`
`/* un comentario`
`multilínea */`
	`-- un comentario`
	`{- un comentario`
	`multilínea -}`
		`% un comentario`
		`/* Un comentario`
		`multilínea */`

Valores Literales

	Wollok	Haskell	Prolog
Strings	“uNa CadEna” ‘uNa CadEna’	“uNa CadEna”	NA
Caracteres	NA	‘a’	NA
Símbolos/Átomos	NA	NA	unAtomo
Booleanos	true false	True False	NA
Colección/Lista	[1, 2, 3] (Lista) #{1, 2, 3} (Set)	[1, 2, 3] 1:2:3:[]	[1, 2, 3] [1, bart, []] [1|[2|[3|[]]]]
Tuplas	NA	(1, True, [1, 2])	NA
Data/Functores	NA	Constructor 1 True	nombreFunctor(1, unAtomo)
Bloques/Funciones Anónimas	{ param1, param2 => cuerpo }	\param1 param2 -> cuerpo	NA

Operadores lógicos y matemáticos

	Wollok	Haskell	Prolog
Equivalencia	==	==	NA
Identidad	===	NA	NA
~ Equivalencia	!=	/=	\=
Comparación de orden	> >= < <=	> >= < <=	> >= < =<
Disyunción (O lógico)	||	||	NA
Conjunción (Y lógico)	&&	&&	,
Negación	! unBool unBool.negate() not unBool	not unBool	not(Consulta)

Operadores matemáticos

	Wollok	**Haskell**	**Prolog (sólo usando is)**
Operadores aritméticos comunes	`+ - * /`	`+ - * /`	`+ - * /`
División entera	`dividendoEntero / divisor`	`div dividendo divisor`	`dividendo // divisor`
Resto	`dividendo % divisor`	`mod dividendo divisor`	`dividendo mod divisor`
Valor absoluto	`unNro.abs()`	`abs unNro`	`abs(Nro)`
Exponenciación	`base ** exponente`	`base ^ exponente`	`base ** exponente`
Raíz cuadrada	`NA`	`sqrt unNro`	`sqrt(Nro)`
Máximo ó mínimo entre dos números	`unNro.max(otroNro)` `unNro.min(otroNro)`	`max unNro otroNro` `min unNro otroNro`	`NA`

Bloques / Lambdas

	Wollok	**Haskell**
Sin parámetros	`{lo que quiero hacer}`	`NA`
De un parámetro	`{x => lo que quiero hacer con x}`	`(\ x -> lo que quiero hacer con x)`
Más de un parámetro	`{x, y => algo con x e y}`	`(\x y -> algo con x e y)`

Patrones

	Haskell	**Prolog**

Listas	`[]`
	`(cabeza:cola)`
	`(cabeza:segundo:cola)`
		`[]`
		`[Cabeza|Cola]`
		`[Cabeza,Segundo|Cola]`

Tuplas	`(componente1, componente2)`	`NA`
Data/Functores	`Constructor componente1 componente2`	`nombreFunctor(componente1,componente2)`
Variable anónima	`_`	`_`

Operaciones “simples” sin efecto (efecto colateral) sobre colecciones / listas

	Wollok (mensajes)	**Haskell (funciones)**	**Prolog (predicados)**
Longitud	`size()`	`length :: [a] -> Int` `genericLength :: Num n => [a] -> n`	`length/2`
Saber si está vacía	`isEmpty()`	`null`	`NA`
Concatenación	`NA`	`++`	`append/3`
Unión	`NA`	`union (**)`	`union/3`
Intersección	`NA`	`intersect (**)`	`intersection/3`

Acceso por índice	`get(indice)` `(base 0, Listas)`	`unaLista !! unNro` `(base 0)`	`nth0/3` `nth1/3`
Pertenencia	`contains(elem)`	`elem`	`member/2`
Máximo ó mínimo de un conjunto de números	`NA`	`maximum` `minimum`	`max_member/2` `min_member/2`
Sumatoria de un conjunto de números	`NA`	`sum`	`sumlist/2`
Aplanar colección de colecciones / lista de listas	`flatten()`	`concat`	`flatten/2`

Operaciones “avanzadas” sin efecto (efecto colateral) sobre colecciones/listas

	Wollok	Haskell
Sumatoria de un conjunto de elementos según una transformación	`sum(bloqueDe1)`	`NA`
Primeros n elementos de un conjunto	`NA`	`take :: Int -> [a] -> [a]`
Filtrar	`filter(bloqueDe1)`	`filter :: (a -> Bool) -> [a] -> [a]`
Transformar	`map(bloqueDe1)`	`map :: (a -> b) -> [a] -> [b]`
Todos cumplen (verdadero para lista vacía)	`all(bloqueDe1)`	`all :: (a -> Bool) -> [a] -> Bool`
Alguno cumple (falso para lista vacía)	`any(bloqueDe1)`	`any :: (a -> Bool) -> [a] -> Bool`
Mapear y luego aplanar	`flatMap(bloqueDe1)`	`concatMap :: (a -> [b]) -> [a] -> [b]`
Reducir/plegar a izquierda	`fold(semilla, bloqueDeSemillaYElem)`	`foldl :: (a -> b -> a) -> a -> [b] -> a` `foldl1 :: (a -> a -> a) -> [a] -> a`
Reducir/plegar a derecha	`NA`	`foldr :: (b -> a -> a) -> a -> [b] -> a` `foldr1 :: (a -> a -> a) -> [a] -> a`
Primer elemento	`head() o first()` `(sólo Lista, no Set)`	`head :: [a] -> a`
Último elemento	`NA`	`last :: [a] -> a`
Cola	`NA`	`tail :: [a] -> [a]`
Segmento inicial (la lista sin el último)	`NA`	`init :: [a] -> [a]`

Apareo de listas	`NA`	`zip :: [a] -> [b] -> [(a, b)]`
		`zipWith ::`
		`(a -> b -> c) -> [a] -> [b] -> [c]`

Buscar el primer elemento que cumpla una condición	`find(bloqueDe1)` `findOrElse(bloqueDe1, bloqueSinParametros)`	`find :: (a -> Bool) -> [a] -> a` `(**) (***)`
Cantidad de elementos que cumplen una condición	`count(bloqueDe1)`	`NA`
Lista ordenada	`sortedBy(bloqueDe2)`	`sort :: [a] -> a (**) (***)`
Buscar el máximo o mínimo según criterio.	`max(bloqueDe1)` `min(bloqueDe1)`	

	`(el bloque debe retornar un número)`	
Sin repetidos	`asSet()`	`NA`

Mensajes de colecciones con efecto (sólo Wollok)

Agrega *unObjeto* a *unaCol.* Para las listas se agregan al final.	*unaCol*.**add(***unObjeto***)**
Agrega todos los elementos de *otraCol* a *unaCol.* Para las listas se agregan al final.	*unaCol*.**addAll(***otraCol***)**
Ejecuta *unBloque* con efecto colateral para cada elemento de *unaCol.*	*unaCol*.**forEach(***unBloque***)**
Elimina *unObjeto* de *unaCol.*	*unaCol*.**remove(***unObjeto***)**
Elimina todos los elementos de *unaCol*.	*unaCol.**clear()***
Utiliza *unBloque*, que recibe dos parámetros “a” y “b” y devuelve un booleano, para reordenar la lista. Un elemento de la lista “a” será anterior a “b” si el bloque se evalúa a verdadero.	*unaLista.**sortBy(**unBloque**)***

Funciones de Haskell

Operaciones sobre funciones

`Aplica una función con un valor (con menor precedencia que la aplicación normal)`	`($) :: (a -> b) -> a -> b`
`Compone dos funciones`	`(.) :: (b -> c) -> (a -> b) -> (a -> c)`
`Invierte la aplicación de los parámetros de una función`	`flip :: (a -> b -> c) -> b -> a -> c`

Unfolds (generación de listas a partir de una semilla)

`Genera una lista que repite infinitamente al elemento dado`	`repeat  :: a -> [a]`
`Para iterate f x, genera la lista infinita [x, f x, f (f x), ...]`	`iterate :: (a -> a) -> a -> [a]`
`Genera una lista que repite una cierta cantidad de veces al elemento dado`	`replicate :: Int -> a -> [a]`
`Para cycle xs, genera la lista infinita xs ++ xs ++ xs ++ ...`	`cycle  :: [a] -> [a]`

Otros Predicados Importantes de Prolog

`Para todo`	`forall(Antecedente, Consecuente)`
`Armar una lista a partir de una consulta`	`findall(Formato, Consulta, Lista)`