¡Hola, Haskell!

Fernanda Andrade

Stack Builders

28 y 29 de abril de 2016

¡Hola, mundo!

```
1 holaHaskell :: IO ()
2 holaHaskell = putStrLn "Hola, Haskell"
```

Haskell

`https://www.haskell.org/`

- Funcional
- Puro
  - Inmutabilidad
  - Sin efectos secundarios
  - Transparencia referencial
- Evaluación perezosa
- Tipificación estática
- Funciones de orden superior

Haskell es funcional

No es imperativo

Factorial en C:

```
1 int  factorial (int n) {
2     int  result = 1;
3     for (int i = 1; i <= n; ++i)
4         result *= i;
5     return result ;
6 }
```

Haskell es funcional

No es imperativo

Factorial en Haskell:

```
1
2 factorial 0 = 1
3 factorial n = n * factorial (n-1)
```

Haskell es funcional

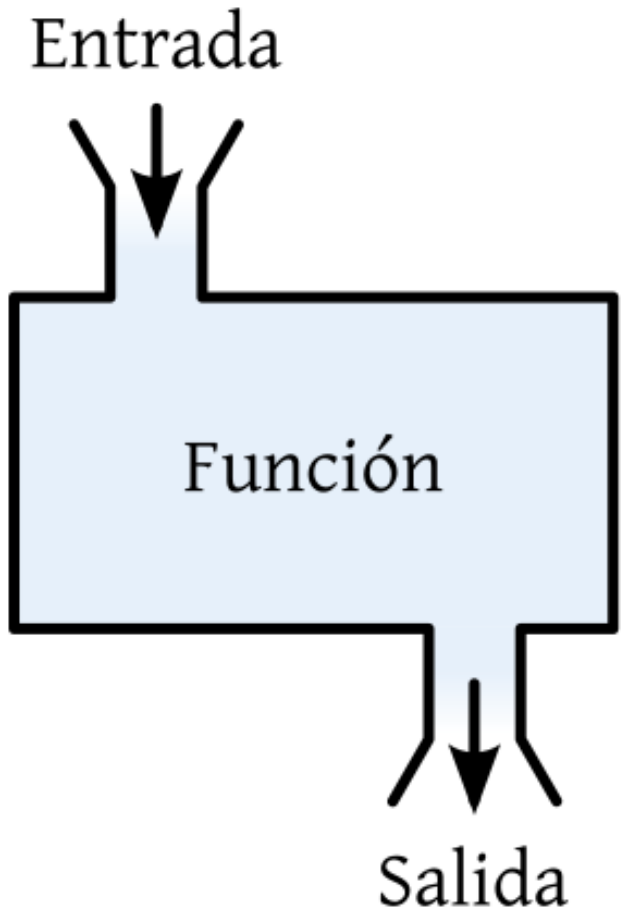
No es imperativo

Factorial en Haskell:

```
1 factorial :: Int -> Int
2 factorial 0 = 1
3 factorial n = n * factorial (n-1)
```

Haskell es funcional

Evaluación de expresiones



Haskell es funcional

Evaluación de expresiones

```
f(x) = x + x
```

Haskell es funcional

Evaluación de expresiones

```
f(x) = x + x
```

```
f(2) = 2 + 2
f(2) = 4
```

Haskell es funcional

Evaluación de expresiones

```
f(x) = x + x

f(2) = 2 + 2
f(2) = 4
f(5) = 5 + 5
f(5) = 10
```

Haskell es funcional

Evaluación de expresiones

```
1  doble :: Int -> Int
2  doble n = n + n
```

Haskell es puro

Inmutabilidad

Sumar números de 1 a n en C:

```
1 int sum(int n){
2   int result = 0;
3   for(int i=1; i<=n; i++)
4     result += i
5   return result
6 }
```

Haskell es puro

Inmutabilidad

En caso de n = 3

```
i = 1
total = 1
```

Haskell es puro

Inmutabilidad

En caso de n = 3

```
i = 1
total = 1
i = 2
total = 3
```

Haskell es puro

Inmutabilidad

En caso de n = 3

```
i = 1
total = 1
i = 2
total = 3
i = 3
total = 6
```

Haskell es puro

Inmutabilidad

Sumar números de 1 a n en Haskell:

```
1 sum [1.. n]
```

Haskell es puro

Inmutabilidad

Sumar números de 1 a n en Haskell:

```
1 sum :: Num a => [a] -> a
2 sum [] = 0
3 sum (x : xs) = x + sum xs
```

- Variables y estructuras de datos son inmutables.

Haskell no tiene efectos secundarios

En C:

```c
int main() {
    printf("Hola, mundo.");
    return (0);
}
```

Haskell no tiene efectos secundarios

En Haskell:

```
1 count :: [a] -> Int
```

- Funciones sólo pueden calcular y retornar valores.
- Funciones garantizan integridad.

Haskell es puro

Transparencia referencial

En c:

```c
int global = 5;

int suma (int n){
  return (n + global);
}

int main(){
  int resultado;

  // resultado = 6
  resultado = suma(1);

  global = 0;

  // resultado = 1
  resultado = suma(1);

}
```

Haskell es puro

Transparencia referencial

En Haskell:

```
1 x :: Int
2 x = 5
3
4 suma :: Int -> Int
5 suma n = n + x
```

Haskell es puro

Transparencia referencial

```
> suma 1
6
> suma 1
6
```

- Si una función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado.

Haskell es puro

Transparencia referencial

```
1 x :: Int
2 x = 5
3
4 suma :: Int -> Int
5 suma n = n + x
6
7 x = 0
```

```
    Multiple declarations of ‘x’
    Declared at: suma.hs:2:1
                 suma.hs:7:1
Failed, modules loaded: none.
```

Haskell tiene evaluación perezosa

```haskell
square :: Int -> Int
square x = x * x
```

```
> square (1 + 2)
=> (1 + 2) * (1 + 2)
=> 3 * (1+2)
=> 3 * 3
=> 9
```

- Haskell no calculará resultados hasta que se vea realmente forzado a hacerlo.

Haskell tiene evaluación perezosa

Primeros 5 números de una lista infinita

```
> take 5  [1..]
 [1,2,3,4,5]
```

- Es posible trabajar con estructura de datos infinitos.

Haskell es un lenguaje tipificado estáticamente

```
1 printString :: String -> IO ()
2 printString word = putStrLn word
```

Haskell es un lenguaje tipificado estáticamente

```
> printString 5
```

```
<interactive>:18:13:
 No instance for (Num String) arising from the
 literal ‘5’
 In the first argument of ‘printString’, namely
 ‘5’
 In the expression: printString 5
 In an equation for ‘it’: it = printString 5
```

- Haskell verifica que el tipo de dato declarado coincide con el tipo inferido (en tiempo de compilación).

Funciones de orden superior

map

```
1 map :: (a -> b) -> [a] -> [b]
2 map _ [] = []
3 map f (x:xs) = f x : map f xs
```

- Funciones pueden tomar funciones como parámetros y devolver funciones como resultado.

Funciones de orden superior

map

```
> map (+3) [0,1,2,3,4,5]
 [3,4,5,6,7,8]
```

Funciones de orden superior

filter

```
1  filter  :: (a -> Bool) -> [a] -> [a]
2  filter  _ []  = []
3  filter  p (x:xs)
4      | p x         = x :  filter  p xs
5      | otherwise =  filter  p xs
```

Funciones de orden superior

filter

```
> filter even [1,2,3,4,5,6]
[2,4,6]
```

```
> filter (>3) [1,2,3,4,5,6]
[4,5,6]
```