Relaciones

Ernesto Rodriguez

Universidad del Itsmo

erodriguez@unis.edu.gt

Introducción a Elm

- Es un lenguaje **funcional** y **puro**
- Sintaxis inspirada en Haskell y Standard ML
- Probablemente nuevo para todos (Fair play)
- Lenguaje limpio y claro similar a la matematica estudiada hasta el momento
- Utiliza funciones como modelo de computación (ya estudiamos funciones)
- Puede ser ejecutado por un navegador web o por Node.JS

Introducción a Elm

- Instrucciones de instalación: `https://guide.elm-lang.org/install.html`
- Se puede obtener una sesión interactiva ejecutando `elm repl` en la terminal
- Si tiene instalado Node.JS, puede instalarse ejecutando `npm install elm`

Elm: Lenguaje basico

- Incluye tipos comunes: `number`, `Bool`, `String`, etc.
- Permite construir tipos nuevos mediante **constructores de tipos**
- Operadores basicos: $+, -, *,$ `not` etc
- Estructuras de control: `if/else` y **pattern matching**
- Comentarios: `-- comentario`
- **Inferencia de tipos**

Elm: Declaraciones

- Existen tres tipos de declaraciones:
  - Declaraciones de valores: `x = 42`
  - Declaraciones de funciones: `duplicar` $x = x + x$
  - Declaraciones de tipos: `type Vector = Dim2 Int Int | Dim3 Int Int`
- Media vez declarada una función, la podemos **aplicar**:

  `> duplicar x`

  84
- Podemos imprimir el tipo de una función:

  `> duplicar`

  `⟨function⟩ : number → number`
- Los **constructores de tipos** también son funciones:

  `> Dim2`

  `⟨function⟩ : number → number → number`

Parejas ordenadas

- Es posible declarar parejas ordenadas de cualquier tamaño en elm:
  $> \texttt{v2} = (3, 4)$
  $(3, 4) \ : \ (\texttt{number}, \texttt{number1})$
  $> \texttt{v3} = (3, 4, 5)$
  $(3, 4, 5) \ : \ (\texttt{number}, \texttt{number1}, \texttt{number2})$
- También es possible **deconstruir** las parejas ordenadas:
  $> \texttt{suma2}\ (x, y) = x + y$
  $> \texttt{suma2}\ (3, 4)$
  $7 \ : \ \texttt{number}$
- Podemos ignorar los valores que no nos interesan nombrandolas "_":
  $> \texttt{segundo}\ (\_, y, \_) = y$
  $> \texttt{segundo}\ (1, 2, 3)$
  $2 \ : \ \texttt{number}$

Listas

- Tienen dos **constructores de tipos**: [] (llamado *nil*) y (::) (llamado *cons*):
  $> 1 :: 2 :: 3 :: []$
  $[1, 2, 3] \ : \ \texttt{List Number}$
- Se puede utilizar la **comodidad sintactica** (syntactic sugar) para definirlas:
  $> [1, 2, 3]$
  $[1, 2, 3] \ : \ \texttt{List Number}$
- Facilita la definicion de funciones:
  $> \texttt{rango}\ (a, b) = \texttt{if}\ b < a\ \texttt{then}\ []\ \texttt{else}\ a :: \texttt{rango}\ (a + 1, b)$
  $[1, 2, 3, 4] \ : \ \texttt{List Number}$

¿Una función definida en terminos de si misma?

Recursión

- Una función es definida en terminos de si misma
- En cada ocasión, la función se acerca más a la solución
- Elm primero evalua los parametros, luego evalua la función
- ¿Que sucede cuando la función no se acerca una respuesta?

```
> divergente n = 1 + divergente (n + 1)
> divergente 1
```

Recursión

- Una función es definida en terminos de si misma
- En cada ocasión, la función se acerca más a la solución
- Elm primero evalua los parametros, luego evalua la función
- ¿Que sucede cuando la función no se acerca una respuesta?
  $> \texttt{divergente}\ n\ =\ 1 + \texttt{divergente}\ (n+1)$
  $> \texttt{divergente}\ 1$
  RangeError: Maximum call stack size exceeded
- Se dice que una función **diverge** o es **divergente** si nunca llega a una respuesta

Transparencia referencial e imutabilidad

- Elm, a diferencia de la mayoria de lenguajes es **referencialmente transparente**:
  `rango` $(1, 2)$ $=$ (`if` $1 > 2$ `then` [] `else` 1 :: (`if` $2 >$ 2 `then` [] `else` 2 :: (`if` $2 > 3$ `then` [] `else` $(\ldots)$)))
- Elm no permite modificar el estado de una variable:
  $> x = 42$
  $>$ `siguiente` $() = x = x + 1; x$
  – PARSE ERROR –
- ¿Podemos escribir programas "reales" sin estado? Si:
  `https://unsoundscapes.itch.io/flatris`
- No somos comunistas por lo tanto no queremos que el estado dicte todo lo que pasa en nuestros programas...

Funciones por casos o Pattern Matching

- Una lista puede ser una lista vacia ([]) o una lista compuesta ($x :: xs$) donde x es un elemento y xs una lista
- Se puede utilizar la sintaxis `case ... of` para definir una función por casos
- **Ejemplo:**

```
aplanar lista = case lista of
  x :: xs -> x ++ (aplanar xs)
  []      -> []
> aplanar [["Hola","como"],["estas?"]]
["Hola","como","estas?"]
```

Cadenas y Listas

- Una **cadena** es una secuencia de **caracteres**
- En elm, un **caracter** se represente utilizando una letra entre comillas simples: $'a', 'e', 'i'$
- En elm, una **cadena** es un tipo atomico, por lo tanto:
  - La función `String.toList` convierte una **cadena** a una **lista** de **caracteres**
  - La función `String.fromList` convierte una **lista** de **caracteres** a una **cadena**

```
> String.toList "hola"
['h','o','l','a']
> String.fromList ['h','o','l','a']
"hola"
```

- Ejercicio:
  - Escribir una función que detecta palindromes, como “otto” o “anna”
  - Probar con la frase: “Marge lets Norah see Sharon’s telegram”