



Universidad Tecnológica Nacional
FACULTAD REGIONAL CORDOBA

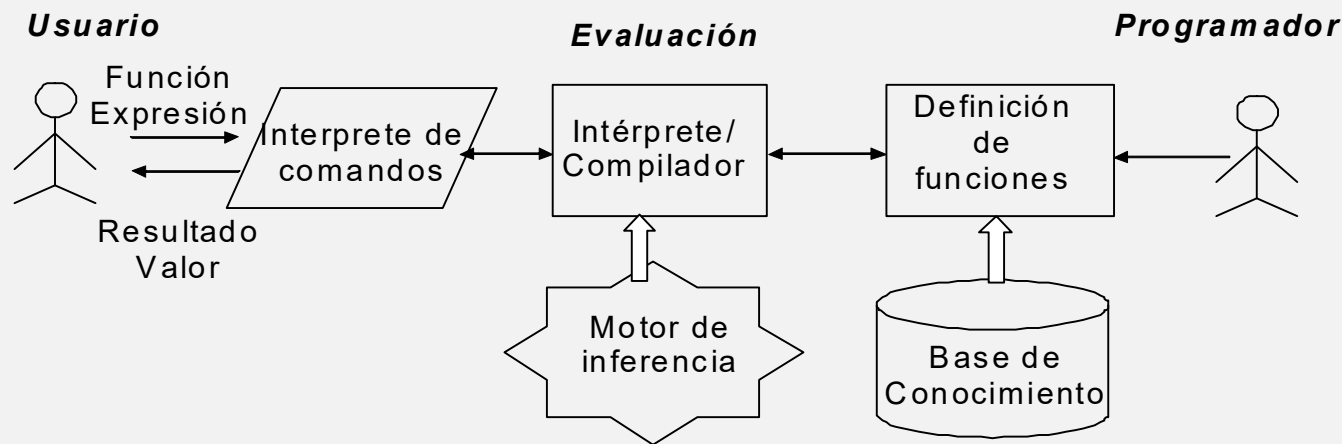
PARADIGMAS DE PROGRAMACION

Unidad IV Paradigma Funcional Parte I

Paradigma funcional

Es un paradigma declarativo que se basa en un modelo matemático de composición de funciones.

Funciones + Control = programa



Características

- Basado en la **matemática** y en la **teoría de funciones**.
- La manera de construir declaraciones es a través de **funciones**.
- **Funciones de orden superior**: pueden recibir funciones como parámetros y/o retornar funciones como resultados.
- **Transparencia referencial**: Los valores resultantes son inmutables. No existe el concepto de cambio de estado.
- **Tipos de datos genéricos**: polimorfismo.
- **Recursividad**: funciones que se llaman a si mismas.
- **Evaluación ansiosa** (*eager evaluation*) / Evaluación perezosa (*lazy evaluation*): evalúa los argumentos de una función antes de conocer si son necesarios / evalúa los argumentos cuando los necesita.

Ventajas y Limitaciones

Ventajas

- Fácil de formular matemáticamente.
- Administración automática de la memoria.
- Simplicidad en el código.
- Rapidez en la codificación de los programas.

Limitaciones

- No es recomendable para modelar lógica de negocios o para realizar tareas de índole transaccionales.
- Difícil de integrar con otras aplicaciones.

Áreas de aplicación

- Sistemas distribuidos de control (por ejemplo, control de tráfico aéreo, mensajería instantánea, servicios basados en Web).
- Demostraciones de teoremas.
- Creación de compiladores, analizadores sintácticos.
- Resolución de problemas que requieran demostraciones por inducciones.
- Se utiliza en centros de investigaciones y en universidades.
- En la industria para resolver problemas matemáticos complejos.

Cálculo Lambda

- El cálculo lambda fue desarrollado por Alonso Church en la década del 30 con el objeto de dar una teoría general de las funciones.
- Aporta:
 - Una sintaxis básica, para formalizar el modo de escribir funciones.
 - Una semántica para el concepto de función como proceso de transformación de argumentos en resultados.
 - Un medio para definir primitivas de Programación.

Cálculo Lambda

- Es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión.
- Consiste en un esquema simple para definir funciones (abstracción funcional) y de una regla de transformación simple por substitución de variables (aplicación de función).
- Por ejemplo:

si una función f es definida por la ecuación: $f(x) = t$,
donde t es algún término que contiene a x ,
entonces la aplicación $f(v)$ devuelve el valor $t[v/x]$,
donde $t[v/x]$ es el término que resulta de sustituir v en cada
aparición de x en t .

si $f(x)=x*x$, entonces $f(3)=3*3=9$.
abstracción funcional *Aplicación de función*

Funciones en el Paradigma Funcional: Sintaxis

Las funciones se denominan expresiones y se pueden representar con la siguiente sintaxis:

<expresión> ::=

<variable> |

<constante> |

(<variable>, ... ,<variable>) <expresión> |

<expresión> (<expresión>, ... ,<expresión>)

Funciones: Abstracción Funcional

- Una abstracción funcional es una expresión por la cual se define una función:

$$3*5$$

en general:

$$(x) 3 * x$$

generalizando más:

$$(x, y) x * y$$

construcción general:

(<variable>, ,<variable>) <expresión>

ó

(x) (y) <expresión>

Funciones: Sintaxis

- Todas las expresiones denotan un valor, pueden ser expresiones que no admitan ser aplicadas a ningún argumento o expresiones que pueden ser aplicadas a un cierto número de argumentos para ser totalmente evaluadas. Formas :
- **expresiones atómicas**, que pueden ser <variable> y <constante>
- **expresiones compuestas**
 - abstracciones funcionales: (<variable>, . . . ,<variable>) <expresión>
 - aplicaciones funcionales: <expresión> (<expresión>, ,<expresión>)

Funciones de orden mayor

- Es una función tal que alguno de sus argumentos es una función o que devuelve una función como resultado.
- Por ejemplo, definimos una función que invoque a otra función dos veces, por ejemplo el incremento en 1. Expresión:

$$(f, x) \rightarrow f(f(x)) \quad \text{ó} \quad (f)(x) \rightarrow f(f(x))$$

- Ejemplo:

Si consideramos la función: **inc (x) = x+1**

y definimos la función **dos_veces (inc) (x) = inc(inc(x))**, cuyo argumento de entrada es la función inc y la variable x

Al evaluar la segunda función con el valor de x = 3:

$$\begin{aligned} \text{dos_veces (inc) 3} & \implies \text{inc(inc(3))} \\ & \implies 5 \end{aligned}$$

Lenguaje Haskell

- Es un lenguaje funcional puro.
- Es fuertemente tipado.
 - El sistema de tipos es utilizado para detectar errores en expresiones y definiciones de función.
 - El sistema de inferencia de tipos ofrece mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución. Posee un algoritmo que infiere el tipo de las expresiones, el programador no está obligado a declarar el tipo de las expresiones y si lo hace, el sistema chequea que el tipo declarado coincida con el tipo inferido.
- Posee como características: evaluación perezosa, funciones de alto orden, polimorfismo de tipos, tipos definidos por el usuario, entre otras.

Tipos de Datos predefinidos

Clasificación:

- **tipos *básicos***, cuyos valores se toman como primitivos, por ejemplo, Enteros, Flotantes, Caracteres y Booleanos.
- **tipos *compuestos***, cuyos valores se construyen utilizando otros tipos, por ejemplo, listas, funciones y tuplas.

Tipos Básicos

Nombre	Tipo	Valores	Operaciones
Booleanos	Bool	True/False	(&&), () y not.
Enteros	Int	enteros positivos y negativos.	+, -, *, /, ^
	Integer	enteros sin límites superior ni inferiores	
Flotantes	Float/Double	reales, fraccionarios, notación científica (cantidades muy grandes para ser representadas por un entero)	+, -, *, /, ^ Truncate, round, sqrt
Caracter	Char	valor entre comillas simples, por ejemplo 'a', '0', '.' y 'Z'.	chr, digitToInt, isAlpha, isDigit, isLower, isUpper, isSpace, toLower, toUpper
Cadenas	String	Lista de char, [Char]	Operaciones de listas

Funciones en Haskell

- Las funciones en Haskell son objetos de primera clase. Pueden ser argumentos o resultados de otras funciones o ser componentes de estructuras de datos.
- Sintaxis:
 - *Definición de signatura*: La definición de tipo de una función, donde se especifican los tipos de los datos de entrada y salida que posee la función;
 - *Implementación del cuerpo de la función*: Se especifica la descripción de la función para transformar los datos de entrada en los datos de salida.

```
nombre_funcion :: tipo_argumento -> tipo_resultado  
nombre_funcion nombre_argumento = <implementacion>
```

Funciones en Haskell

Ejemplo:

- *Función parcializada (curried):*

`suma :: Integer -> Integer -> Integer`

`suma x y = x + y`

La invocación : `suma 2 5 ==> 7`

- *Función no parcializada (uncurried):*

`suma :: (Integer, Integer) -> Integer`

`suma (x, y) = x + y`

La invocación : `suma (2,5) ==> 7`

Funciones : nominación

- Existen dos formas de nombrar una función:
- Identificador :

Ejemplos: sum, product y fact

- Símbolo de operador :

Ejemplos: * y +

Los operadores se utilizan usando notación infija:

$4 + 3$ ó $x + y$

También es posible usar la notación prefija:

$(+) 4 3$ ó $(+) x y$

Funciones : Expresiones lambda

- Las expresiones *lambda* tienen la forma:
 $\backslash \langle \text{parámetros} \rangle \rightarrow \langle \text{expr} \rangle$
- Esta expresión denota una función que toma un número de parámetros, produciendo el resultado especificado por la expresión $\langle \text{expr} \rangle$. También llamada función anónima.
- El cálculo lambda permite utilizar una función dentro de otra sin darle nombre.

```
inc x = x + 1
```

```
funcion x = inc x
```

es lo mismo que decir:

```
funcion x = (\x-> (x+1)) x
```

Funciones : Expresiones lambda

- Ejemplos:

cuadrado :: Integer -> Integer

*cuadrado x = x*x*

o en forma anónima: $(\lambda x \rightarrow x*x)$

suma :: Integer -> Integer -> Integer

suma x y = x + y

Su forma anónima: $\lambda x y \rightarrow x+y$

Si evaluamos esta última:

$(\lambda x y \rightarrow x+y) 2 3 ==> 5$

Sintaxis

Case sensitive

- La sintaxis de Haskell diferencia entre mayúsculas y minúsculas.

Comentarios

- En una línea: utiliza el símbolo “--” que inicia un comentario y ocupa la parte de la línea hacia la derecha del símbolo. Por ejemplo:
-- Este es un comentario, cuando empieza con "--“
- MultiLínea: utiliza “{-” para iniciar el comentario y -} para finalizarlo. Por ejemplo:
{- Este es un comentario,
en dos líneas-}

Sintaxis : Valores y expresiones

- Los valores son expresiones irreducibles, se consideran como la respuesta a un cálculo, Cada valor tiene asociado un tipo ($\langle \text{valor} \rangle :: \langle \text{tipo} \rangle$):

$2 :: \text{Int}$

- Las expresiones son términos contruidos a partir de valores:

$(2*3)+(4-5)$

- La reducción o evaluación de una expresión consiste en aplicar una serie de reglas que transforman la expresión en un valor

$(2*3)+(4-5) \rightarrow 6+(4-5) \rightarrow 6+(-1) \rightarrow 6-1 \rightarrow 5$

- Toda expresión tiene un tipo: el tipo del valor que resulta de reducir la expresión

$(2*3)+(4-5) :: \text{Int}$

Sintaxis : Identificadores

- Un identificador consta de una letra seguida por cero o más letras, dígitos, subrayados y comillas simples. Ejemplos:

sum f f" fintSum nombre_complejo

- Los siguientes son palabras reservadas y no pueden utilizarse como nombres de funciones o variables:

case of where let in if then else
data type infix infixl infixr class instance primitive

- La letra inicial distingue familias de identificadores:
 - *Mayúscula* los tipos y constructores de datos
 - *Minúscula* los nombres de función y variables
- Caso especial: Variables anónimas representadas por `_` (guión bajo), son variables sin nombre que se utilizan en los argumentos de la funciones.

```
funcion1 valor1 valor2 = valor1 + valor2
```

```
funcion1 valor1 _ = valor1
```

Sintaxis : Operadores

- Son funciones que pueden escribirse entre sus (dos) argumentos en lugar de precederlos.

- Por ejemplo,

3 <= 4 en lugar de: menorIgual 3 4

5 + 7 en lugar de: + 5 7 ó suma 5 7

- Un símbolo de operador es escrito utilizando uno o más de los siguientes caracteres:

:	!	#	\$	%	&	*	+	.	/
<	=	>	?	@	\	^		-	==

Sintaxis : Operadores

Operadores	
Aritméticos	+, −, *, /, ^ (**), 'mod' (modulo).
Relacionales	==, / =, >, >=, <, <=
Lógicos	not, , &&

Sintaxis : Operadores

- **Precedencia**

Cada operador tiene asignado un valor de precedencia (un entero entre 0 y 9).

- **Asociatividad**

Por ejemplo, el símbolo (-) se puede decir que es:

- Asociativo a la izquierda: si la expresión "x-y-z" se toma como "(x-y)-z"
- Asociativo a la derecha: si la expresión "x-y-z" se toma como "x-(y-z)"
- No asociativo: Si la expresión "x-y-z" se rechaza como un error sintáctica, es decir, un operador no asociativo, no puede utilizarse más de una vez sin aclarar el significado con paréntesis.

Análisis de expresiones

- El análisis de las expresiones en Haskell, posee dos fases:
 - Análisis sintáctico, para chequear la corrección sintáctica de las expresiones.
 - Análisis de tipo, para chequear que todas las expresiones tienen un tipo correcto.

Entorno de Haskell - WinHugs

- Entorno libre, disponible para distintas plataformas (MS-Windows, Mac OS X, Unix)
- Funciona siguiendo el modelo de una calculadora en el que se establece una sesión interactiva entre el ordenador y el usuario.
- Una vez arrancado, el sistema muestra un *prompt Hugs>* y espera a que el usuario introduzca una expresión (denominada expresión inicial y presione la tecla <RETURN>).
- Cuando la entrada se ha completado, el sistema evalúa la expresión e imprime su valor y después vuelve a mostrar el *prompt*.

Entorno de Haskell - Hugs

Evaluación de expresiones:

```
Hugs> (2+3)*8
```

```
40 :: Integer
```

El usuario introduce la expresión "(2+3)*8" que es evaluada por el sistema imprimiendo como resultado el valor "40".

```
Hugs> (2+3)+(3*5)
```

```
20 :: Integer
```

Un programa en Haskell

- Para escribir un programa en Haskell hay que cumplir los siguientes pasos:
 - Escribir un script con la definición de todas las funciones que necesitemos para resolver el problema, en un archivo de texto que se graba con la extensión `.hs`.
 - Cargar el script en Hugs. Para ello utilizamos el comando `:load` seguido del nombre del archivo.
 - Ingresar en el prompt la expresión a evaluar y presionar la tecla `<RETURN>`, para que el sistema evalúe la expresión e imprima su valor antes de volver a mostrar el *prompt*.

Entorno de Haskell - Hugs

Definición de funciones

- Las funciones se almacenan en un archivo con extensión `hs`, para utilizarlas en el proceso de evaluación.
- Ejemplo, se crea el archivo `:edit ejemplo1.hs`:

```
cuadrado :: Int -> Int
cuadrado x = x * x
```

- Para utilizar las definición de este archivo hay que cargarlo en el sistema con la orden `:load`.

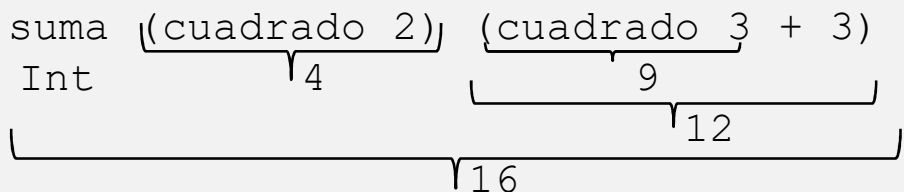
```
Hugs>:load ejemplo1.hs
```

- Si el fichero se cargó con éxito, el usuario ya podría utilizar la definición:

```
Main> cuadrado 2
4 :: Int
```

Entorno de Haskell - Hugs

continuación:

- `Main> cuadrado (2+3)`
`25 :: Int`
- Por ejemplo, definimos una función suma: editamos *ejemplo1.hs*:
`suma :: Int -> Int -> Int`
`suma x y = x + y`
- En la invocación:
 - `Main> suma 2 3`
`5 :: Int`
 - `Main> suma (cuadrado 2) 3`
`7 :: Int`
 - `Main> suma (cuadrado 2) (cuadrado 3 + 3)`
`16 :: Int`


Entorno de Haskell - Hugs

continuación:

- Otro ejemplo, definimos una función cubo: editamos *ejemplo1.hs*:

```
--obtiene el cubo de un número  
cubo::Integer -> Integer  
cubo x = x*x*x
```

- En la invocación:

- Main> suma (cuadrado 2) (cubo 3)

31 :: Int

- Main> cuadrado (suma 2 4)

36 :: Int

- Main> cubo (suma (cuadrado 2) (cuadrado 4))
8000 :: Int
-
- ```
 (cuadrado 2) (cuadrado 4)
 | |
 4 16
 | |
 +-----+-----+
 | |
 20
 |
 +-----+
 |
 8000
```