

Inferencia de Tipos

Lógica de Primer Orden

- Conjunto de Símbolos de Función $\mathcal{F} = \{f, g, h, \dots\}$. Cada símbolo de función tiene una aridad asignada.
 - Aquellas que tienen aridad 0 son constantes.
- Conjunto de Símbolos de Predicado $\mathcal{P} = \{P, Q, R, \geq, \leq, \dots\}$. Cada símbolo de predicado tiene una aridad asignada.
- Variables $\mathcal{X} = \{X, Y, Z, \dots\}$

Programación Lógica

Tips

- Casos base al comienzo. El caso base permite unificar cuando está vacío.
- Recordar siempre los llamados recursivos.
- Recordar para matchear cosas que sean iguales decirlo directamente en la definición.
 - `notasEstudiante(E, [(E, M, N) — T], [(E, M, N) — Res]) :- notasEstudiante(E, T, Res)`. Almacena las notas del estudiante si las E coinciden.
 - `notasEstudiante(E, [(E2, M, N) — T], Res) :- E \= E2 → importante el E \= E2 porque sino, aunque sea diferente va a armar diferentes ramas donde seguro aparezca el estudiante de vuelta. Nosotros solo queremos un resultado.`
- Recordar siempre si un predicado es reversible o no para saber cómo usarlo.

Funciones Útiles

```
member(?elemento, ?lista): Devuelve verdadero si el elemento está en la lista.
Eso sí, solo devuelve verdadero si el elemento está completo.
Ej.: member((tomas, plp, N), [(tomas, plp, 10), (angel, plp, 3)]) devuelve N = 10 porque es lo
que le falta para unificar.
Ej.: member((tomas, plp, 10), [(tomas, plp, 10)]) devuelve true.
Obs: usar member con cuidado, uno de los parámetros enviarlo seguro.

length(?lista, ?longitud): Devuelve la longitud de la lista si no se proporciona la
longitud, caso contrario, devuelve true o false.
Ej.: length([1, 2], A): devuelve A = 2
Ej.: length([1, 2], 2): devuelve true
Ej.: length(A, 2): devuelve [_ , _]
Ej.: length(A, B): devuelve todas las posibles listas, es decir: A = [_], B = 0, A=[_ , _], B = 1...

sum_list(+Lista, ?Res): Devuelve la suma de los elementos de una lista si no se especifica Res.
Si se especifica, devuelve true o false.
Ej.: sum_list([1, 2], 3): true
Ej.: sum_list([1, 2], A): A = 3

desdeReversible(): Devuelve tantos elementos haya de distancia de X a Y. Uno por uno.
Abre n ramas siendo n la distancia entre X e Y.
desdeReversible(X, Y) :- var(Y), Y = X.
desdeReversible(X, Y) :- nonvar(Y), X =< Y.
desdeReversible(X, Y) :- var(Y), X1 is X + 1, desdeReversible(X1, Y).

between(+Low, +High, ?N): Devuelve como respuesta
los elementos que están entre Low y High inclusive. Abre n ramas donde n es la distancia de 1 a 2.
Ej.: between(1, 2, N): N = 1 y N = 2.
```

Paradigma Orientado a Objetos

Básico

- Los programas están conformados por objetos que interactúan entre sí con mensajes.
- Un mensaje es simplemente una solicitud al objeto receptor donde este mensaje tiene respuesta sí y solo sí el objeto receptor entiende ese mensaje. Esta respuesta es ofrecida por un método del mismo objeto.

- Colaboradores Internos: Son los atributos o variables de instancia de un objeto.
- Colaboradores Externos: Son los parámetros o argumentos que tiene un mensaje particular.
 - 1@1 insideTriangle with: 0@0 with: 0@0 with: 0@0. Se realiza una instancia de Point 1@1 y se envía el mensaje insideTriangle con tres keywords o también llamados parámetros.

Encapsulamiento

Una clase debería estar abierta a extensión pero cerrada a modificaciones.

Solo es posible interactuar con un objeto a través de sus métodos los cuales ofrece pues, el estado interno de un objeto es inaccesible desde el exterior.

Objetos, objetos y objetos

Todo objeto es instancia de alguna clase, y a su vez, estas son objetos.

- Una clase es un objeto que abstrae el comportamiento de todas sus instancias.
- Todas las instancias de una clase tienen los mismos atributos
 - Cada instancia puede modificar a gusto sus valores sin afectar a las demás. Cada instancia es única.
- Todas las instancias de una clase usan el mismo método para responder un mismo mensaje.
 - Todas las instancias responden de **la misma manera**. Esto es porque el receptor no conoce al emisor, salvo que el emisor se envíe como colaborador.

Palabras Reservadas

nil, true, false, self, super, thisContext

Literales

- Caracteres: \$
- Strings: 'hola'
- Símbolos (inmutables): #hola
- Constantes numéricas: 29, -1, 5

Herencia

- Cada clase es subclase de alguna otra clase. Si no se especifica, las clases heredan de Object por defecto.
- Una clase hereda todos los métodos de su superclase.
- Una clase puede hacer un override a un método definido en la superclase por otro más específico.
- self: Hace referencia al objeto de instancia.
- super: Hace referencia a la super-clase de la instancia.

Nota: self==super porque refieren al mismo objeto pero difieren en que, si se usa super la búsqueda del método que implementa el mensaje m debe comenzar desde la superclase.

super vs self

- super busca siempre hacia arriba, es decir, las super-clases de la clase a la que mandamos el mensaje. Si el mensaje no está en el super, arroja un error NotUnderstand.
- self busca siempre hacia abajo. Si el mensaje no está, arroja un error NotUnderstand.

Clase Abstracta

Llamamos clase abstracta a una clase que está destinada a abstraer el comportamiento de sus subclases pero no tienen instancias.

Tipos de Mensajes

- Mensajes Unarios: Reciben un solo parámetro.
 - 1 class
 - Mensaje: class — Receptor: 1
- Mensajes Binarios: Reciben dos parámetros.
 - 1 + 2
 - Mensaje: + — Receptor: 1 — Colaborador: 2

- Mensajes Keyword: Reciben parámetros que se pueden distinguir con nombre. No importa el orden en cual se envían porque están dados por una key.
 - a at: 1 put: 'hola'
 - Mensaje: at:put — Receptor: a — Colaborador/es: 1, 'hola'

La prioridad de los mensajes es la siguiente: *unario* > *binario* > *keyword*

Nota: Los paréntesis **()** definen la prioridad máxima.

Bloques / Closures

- Permiten reutilizar código. Recuerdan el estado cuando fueron definidos y qué variables estaban presente. Es una secuencia de envíos de mensajes.
- No usar return dentro de bloques. Corta todo tipo de ejecución.
- Los argumentos obligatorios tienen prioridad sobre los locales.
- Los parámetros se envían como value: param
- Cuando se almacenan en una variable **no se ejecutan**.
- Para llamar a un bloque hay que enviar todos sus parámetros.
- Devuelven como resultado la última expresión.

```
hacerAlgo
|bloque val|
bloque := [:x :y | |z| z:=10. x+y+z].
val := bloque value: 1 value: 2. // retorna 13
```

```
hacerAlgo
|bloque val val2|
bloque := [:x | [:y | |z| z:=10. x+y+z]].
val := bloque value: 1. // retorna el bloque [:y | |z| z:=10. x+y+z] que
recuerda el valor de x cuando se definió, es decir, 1.
val2 := val value: 2. //retorna el resultado del bloque más interno, es decir,
1+2+10 = 13.
```

Return

- Se indica con \wedge .
- Corta todo tipo de ejecución, es decir: $[[x]|x := 0. \wedge 0]$ value. devuelve 0.
- No usar return en bloques. Porque como el bloque vive en un universo aparte, el return es algo peligroso.
- Si el return no se indica dentro del método, devuelve self. Es decir, la instancia del objeto que recibió el mensaje.

Colecciones

Existen varias: Bag (Multiconjunto), Set (Conjunto), Array (Arreglo), OrderedCollection (Lista), SortedCollection (Lista Ordenada) y Dictionary Hash (Hash).

- Bag with: 1 with: 2 with: 4
- #(1 2 4) = (Array with: 1 with: 2 with: 4)
- Bag withAll: #(1 2 4)

Mensajes más comunes

- add: agrega un elemento
- at: devuelve el elemento en una posición (indexa desde 1).
- at:put: agrega un elemento a una posición.
- includes: responde si un elemento pertenece o no.
- includesKey: responde si una clave pertenece o no.
- do: evalúa un bloque con cada elemento de la colección. No muta ni devuelve un resultado, solo sirve para efectos secundarios.
- select: Devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
- reject: la negación del select
- collect: Es el map de funcional.
- detect: devuelve el primer elemento que cumple un predicado.
- detect:ifNone: permite ejecutar un bloque si no se encuentra ningún elemento
- reduce: toma un bloque de dos o mas parámetros de entrada y hace fold de los elementos de izquierda a derecha.

Booleanos

- `ifFalse:`, `ifTrue:ifFalse`, `&`, `—`, `and:`, `or:`, `not`, `=`, `≤`, `≥`
- `\\`: te da el resto

Machete

Sintáxis

```
| var1 var2... |  
[:arg1 :arg2 | | var1 var 2 | expresion1. expresion2...]  
expresion1. expresion2. expresion3  
objeto mensaje  
objeto msg1; msg2.  
var := expresion  
^expresion
```

Palabras Reservadas

```
self, super, thisContext, false, true, nil
```

Literales

```
123, 123.4, $c, 'texto', #simbolo, #(123 123.3 $a) array
```

Ejercicios Útiles

Jerarquía

```
20 + 3 * 5
```

```
Mensaje: + | Obj Receptor: 20 | Colaboradores: 3 | Res = 23
```

```
Mensaje: * | Obj Receptor: 23 | Colaboradores 5 | Res = 115
```

```
20 + (3*5)
```

```
Mensaje: * | Obj Receptor: 3 | Colaboradores: 5 | Res = 15
```

```
Mensaje: + | Obj Receptor: 20 | Colaboradores 15 | Res = 35
```

```
1 = 2 ifTrue: ['what!?!'].
```

```
Mensaje: = | Obj Receptor: 1 | Colaboradores: 2 | Res = instanciaFalse
```

```
Mensaje: ifTrue | Obj Receptor: instanciaFalse | Colaboradores: ['what!?!'].
```

```
| Res = False.
```

```
1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.
```

```
Mensaje: insideTriangle:with:with: | Obj. Receptor: 1@1
```

```
(instancia de point) | Colaboradores: 0@0 with: 2@0 with: 0@2
```

```
Object subclass: #SnakesAndLadders
```

```
instanceVariableNames: 'players squares turn die over'
```

```
classVariableNames: ''
```

```
poolDictionaries: ''
```

```
category: 'SnakesAndLadders'
```

Hay varios mensajes acá.

```
Mensaje: subclass | Receptor: Object Class |Colaborador: #SnakesAndLadders
```

```
Mensaje: instanceVariableNames | Receptor: SnakesAndLadders |
```

```
Colaboradores: 'players squares turn die over' cada uno por separado
```

```
Mensaje: classVariableNames | Receptor: SnakesAndLadders |Colaborador: ''
```

```
Mensaje: poolDictionaries | Receptor: SnakesAndLadders | Colaborador: ''
```

```
Mensaje: category | Receptor: SnakesAndLadders |Colaborador: 'SnakesAndLadders'
```

Bloques

```
[ :x :y | |z| z:=x+y ] value: 1 value: 2. Bloque bien definido, dos parámetros  
y una variable local.
```

```
[ :x :y | x+1 ] value: 1. Arroja error, falta un parámetro.
```

```
[ :x | [:y | x+1]] value: 2. Bloque bien definido, devuelve un nuevo bloque [:y | x+1]
```

```
[ :x :y :z | x + y + z] valueWithArguments: #(1 2 3). Bloque bien definido,
envía tres argumentos en orden.
[ |x y z| x + 1] Arroja error. x es UndefinedObject.
[ :x :y :z | x + 1] Bloque bien definido, espera 3 argumentos
obligatorios pero termina usando uno.
```

```
Class: BlockClosure
```

```
curry
```

```
  ^[:x | [:y | self value: x value: y]].
```

```
flip
```

```
  ^[:x :y | self value: y value: x].
```

```
Class: Integer
```

```
Iterativa
```

```
  timesDo: aBlock
```

```
  | count |
```

```
  count := 1.
```

```
  [count <= self]
```

```
    whileTrue:
```

```
      [aBlock value. count := count + 1]
```

```
Recursiva
```

```
  timesDo2: aBlock
```

```
  self > 0 iffFalse: ^self.
```

```
  aBlock value.
```

```
  self - 1 timesDo: aBlock.
```

```
Class: Collection
```

```
map: aBlock
```

```
  |col2|
```

```
  col2:=(self class) new.
```

```
  self do: [ :elem | col2 add: (aBlock value: elem)].
```

```
  ^col2.
```

```
minimo: aBlock
```

```
"Una implementación poco elegante de la obtención del valor original que genera un mínimo
luego de aplicar un bloque."
```

```
  | minElement minValue |
```

```
  self do: [:each | | val |
```

```
    minValue ifNotNil: [
```

```
      (val := aBlock value: each) < minValue iffTrue: [
```

```
        minElement := each.
```

```
        minValue := val]]
```

```
  ifNil: ["Caso del primer elemento que se lee"
```

```
    minElement := each.
```

```
    minValue := aBlock value: each].
```

```
  ].
```

```
  ^minElement
```

Listas

```
#collect: es el map.
```

```
| res |
```

```
res := #(1 2 4) collect: [:numero | numero * 2].
```

```
El resultado sería multiplicar por dos todos los
elementos de la lista, es decir, [2, 4, 8].
```

```
#select: es el filter
```

```
| res |
res := #(1 2 3) select: [:numero | numero = 1 ].
```

El resultado sería [1]

```
sabeResponder: L
|res|
res := #(1 2 3) select:[:each | each respondsTo: #ptff]. -> Los true no hace falta colocarle =.
^res
```

```
sabeResponder (solo closure)
^[L | L select: [:each | each respondsTo: #ptff]].
```

#inject: into: El primer argumento es el resultado de la llamada anterior y el segundo el elemento actual.

```
listaNumeros := OrderedCollection with: 1 with: 2 with: 3.
listaNumerosSuma := listaNumeros
inject: 0
into: [ :result :elem | result + elem ].
```

El resultado seria 6.

#reduce: (o #fold): Es el foldl que conocemos, hace algo de izquierda a derecha.
 #(10 20 5 30 15) reduce: [:max :each | max max: each].
 El resultado en este ejemplo sería el 30.

#(1 2 3 4 5) reduce: [:product :each | product * each].
 El resultado en este ejemplo sería 120.

#reduceRight: es un foldr convencional. Resuelve de derecha a izquierda.
 #(1 2 3 6) reduceRight: [:acc :each | each-acc].
 El resultado en este ejemplo sería 0.

#do

```
listaNumeros := OrderedCollection with: 1 with: 2 with: 3.
listaNumeros2 := OrderedCollection new.
listaNumeros do: [ :each | listaNumeros2 add: each + 1 ].
```

En este caso, listaNumeros2 termina teniendo los valores de [2, 3, 4].

Almacenar en una lista todos los divisores de un numero

```
Op1:
SmallInteger << divisores
| lista |
lista := OrderedCollection new.
1 to: self do: [:each | (self \\ each) = 0 ifTrue: [lista add:each]].
^lista.
```

```
Op2:
SmallInteger << divisores
|count lista|
lista := OrderedCollection new.
[count <= self]
whileTrue: [
  (self \\ count) = 0 ifTrue: [lista add: count].
  count := count+1.
].
^lista.
```