Alumnos:

Cotarelo, Rodrigo 98577

Longo, Nicolás 98271

Reffle, Lucía 97788

Materia: Teoría del lenguaje

Fecha: 1° Cuatrimestre 2018

Origen e historia:

El lenguaje Julia comienza a gestarse en 2009 como un proyecto liderado por Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman con el objetivo de crear un lenguaje que tuviera una gran capacidad computacionalmente y fuera veloz.

En Febrero de 2012 se lanzó y en 2015 se creó Julia Computing para proveer soporte, entrenamiento y servicios de consultoría a clientes (aquellos que ya utilizaban el lenguaje y potenciales interesados). Desde entonces ha resultado un lenguaje sumamente atractivo para resolver problemáticas de alto contenido matemático y análisis numérico en disciplinas como, sólo para dar los ejemplos más importantes, programación concurrente y distribuida y *data science*.

Para enero del 2018, la comunidad de Julia ascendía a 1.800.000 descargas.

-

Carecterísticas básicas

Manejo de variables y constantes matemáticas:

Julia es un lenguaje que busca cooptar la atención de la comunidad científica y especializarse en la facilidad y el poder de manejo matemático. Es por esto que cuenta con características muy interesantes. Para empezar cualquier variable puede adoptar cualquier valor (si en principio no se declara su tipo, más sobre esto más adelante).

```
In [1]: miVariable = 1
        miVariable = miVariable + 5
Out[1]: 6

In [2]: δ = 0.00001
        δ = "Hola a todos y todas"
Out[2]: "Hola a todos y todas"
```

Julia cuenta con una serie de constantes matemáticas predefinidas.

```
In [3]: π
Out[3]: π = 3.1415926535897...

In [4]: miVariable = 2*π + 0.12
Out[4]: 6.403185307179586

In [5]: miVariable = miVariable + Inf
Out[5]: Inf
```

También permite el manejo de números irracionales.

```
In [6]: res = (1 + 4im)*(2 - 3im)
Out[6]: 14 + 5im

In [7]: cos(res)
Out[7]: 10.147261924626015 - 73.50624621453295im
```

Y números racionales (minimizando la pérdida de dígitos por redondeo).

```
In [8]: 1//3
Out[8]: 1//3

In [9]: float(1//3)
Out[9]: 0.3333333333333333
```

Paradigmas:

Julia es un lenguaje multiparadigma y como tal permite abordar programación tanto imperativa como funcional y orientada a objetos. Para ello se pueden definir funciones convencionales, funciones lambda, clases e interfaces, solo para mencionar algunas características de estos paradigmas, y combinarlos.

Tipado

En Julia el tipado es dinámico y los tipos son, por defecto, omitidos de manera tal que las variables admitan valores de cualquier tipo. Y esto se da así: son los valores los que tienen tipo mientras que las variables son simples notaciones que hacen referencia a entidades. De todas formas, el compilador también permite la especificación del tipo esperado para una cierta variable. Esto suele mejorar no solamente la robustez de los programas, sino también su performance.

```
In [19]: x = 50.74
Out[19]: 50.74

In [20]: x = "Soy un string"
Out[20]: "Soy un string"

In [22]: x
         y::Int64
         y = x
         TypeError: typeassert: expected Int64, got ASCIIString
```

Una característica interesante de Julia es que la declaración de nuevos tipos puede llevar parámetros y esto introduce una versatilidad importante.

```
julia> struct Point{T}
           x::T
           y::T
       end
```

Así, a partir de Point {T} podemos tener una variable que sea de tipo Point {Float64} o Point {Abstract String}, con lo cual tenemos toda una familia de tipos a partir de la definición de una con un parámetro T. Esto es extensible a varios parámetros.

En Julia, todos los tipos son objetos, pero sus métodos no "pertenecen" a la clase, sino que son parte de un listado que el compilador utiliza a la hora de interpretar el código. Ésta es una consideración del diseño del lenguaje que favorece el Multiple Dispatch.

Compilacion e interpretación

Julia posee un compilador denominado JIT (“Just in Time”) que aunqe se utiliza en tiempo de ejecución, el compilador infiere los tipos a través de una primera compilación o “precompilación”. Mediante este mecanismo el lenguaje aparenta ser interpretado en términos de velocidad y así es comparable a lenguajes como C o Fortran, pero con un código muchísimo más simple.

Manejo de errores:

Ante un evento inesperado, Julia tiene un manejo de errores comparable al de la mayoría de los lenguajes. Si la función no puede devolver una salida razonable, entonces levanta una excepción y termina el programa indicando un código de error. En algunos casos, el programador puede especificar hasta el mensaje de error a mostrar.

In [13]:
```
tup = (1, 2, 3)
tup[1] # => 1
```

Out[13]:
```
1
```

In [14]:
```
tup = (1, 2, 3)
try:
    tup[1] = 3 # MethodError
catch e
    println(e)
end
```

```
MethodError(setindex!,(:tup,3,1))
```

In [15]:
```
ampliacion_sqrt(x) =

x >= 0 ? sqrt(x) : error("Valor de x negativo es inválido")

ampliacion_sqrt(100)
```

Out[15]:
```
10.0
```

In [16]:
```
ampliacion_sqrt(-6)
```

```
Valor de x negativo es inválido

 in ampliacion_sqrt at ./In[15]:1
```

Funciones y argumentos

Ejemplo de función en Julia que devuelve dos parámetros.

In [17]:
```
function foo(a,b)
            a+b, a*b
end
x, y = foo(2,3)
```

Out[17]:
```
(5,6)
```

En Julia se pueden utilizar expresiones lambda

In [18]:
```
(x -> x^2 + 2x - 1)(2)
```

Out[18]:
```
7
```

In [19]:
```
map(x -> x^2 + 2x - 1, [1,3,-1])
```

Out[19]:
```
3-element Array{Int64,1}:
  2
 14
 -2
```

Los argumentos en Julia se pasan siguiendo una convención que suele llamarse "pass-by-sharing". Esto significa que no se pasan por copia, sino que los argumentos funcionan en sí mismos como una nueva variable que tiene idéntico valor al que ha sido pasado.

También se puede definir valores de argumentos por default

In [20]:
```
function default(a,b,x=5,y=6)
    return "$a $b and $x $y"
end

default('h','g')
```

Out[20]:
```
"h g and 5 6"
```

In [21]:
```
default('h', 'g', 'i')
```

Out[21]:
```
"h g and i 6"
```

Duck Typing

Duck typing es el estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica.

En duck typing, el programador solo se ocupa de los aspectos del objeto que van a usarse, y no del tipo de objeto que se trata. Por ejemplo en un lenguaje sin duck-typing uno puede crear una función que toma un objeto de tipo Pato y llama los métodos "caminar" y "parpar" de ese objeto. En un lenguaje con duck-typing, la función equivalente tomaría un objeto de cualquier tipo e invocaría los métodos caminar y parpar. Si el objeto tratado no tiene los métodos pedidos, la función enviará una señal de error en tiempo de ejecución.

En los siguientes ejemplos se puede ver cómo Julia intentará aplicarle a mis datos la función pedida, sean del tipo que sean.

```
In [1]: f2(x) = x^2
Out[1]: f2 (generic function with 1 method)

In [3]: f2(2)
Out[3]: 4

In [4]: Ad = rand(3, 3)
        f2(Ad)
Out[4]: 3×3 Array{Float64,2}:
         1.23116  0.915372  0.746678
         1.50719  1.21557   0.948403
         1.52776  1.18283   1.00988

In [6]: f2("hola") #La multiplicacion de este parametro implica una concatenacion
Out[6]: "holahola"

In [7]: v = rand(3)
        f2(v)
        DimensionMismatch("Cannot multiply two vectors")

        Stacktrace:
         [1] power_by_squaring(::Array{Float64,1}, ::Int64) at ./intfuncs.jl:169
         [2] f2(::Array{Float64,1}) at ./In[1]:1
```

Como vemos, en el último ejemplo el compilador levantó una excepción porque la operacion definida en f2 es ambigua ya que hay diferentes manera de multiplicar un vector.

Funciones mutantes y no mutantes:

Por convención las funciones seguidas por un “!” alteran, o bien mutan, el contenido de sus argumentos y las que carecen de un “!”, no lo hacen.

```
In [27]: v = [4, 7, 1]
Out[27]: 3-element Array{Int64,1}:
          4
          7
          1

In [28]: sort(v)
Out[28]: 3-element Array{Int64,1}:
          1
          4
          7

In [29]: v
Out[29]: 3-element Array{Int64,1}:
          4
          7
          1
```

```
In [30]: sort!(v)
Out[30]: 3-element Array{Int64,1}:
          1
          4
          7

In [31]: v
Out[31]: 3-element Array{Int64,1}:
          1
          4
          7
```

Corutinas (Task):

Las Task son elementos de control de flujo que permiten una manera sencilla de suspenderse y reactivarse. También son llamados lightweight threads dado que su ejecución no se realiza en distintos CPU.

```
In [33]: # Se genera la lista de tasks
         t = @task Any[ for x in [1,2,4] println(x) end ]

         # No hay tasks?
         println(istaskdone(t))

         # Cual es la siguiente task?
         println(current_task())

         # Ejecutar task
         println(consume(t))

         false
         Task (runnable) @0x00007fb443677490
         1
         2
         4
         Any[nothing]
```

Scheduler:

El scheduler se encarga de suspender tasks cuando se realiza una operación de comunicación como fetch() y wait() y selecciona otra task para realizar, reiniciando la suspendida cuando finaliza el evento que está esperando.

Canales:

Los canales permiten pasar datos entre task en ejecución donde el tiempo total de ejecución se puede mejorar si se pueden ejecutar otras tareas mientras, por ejemplo, se lee un archivo.

Múltiples escritores en distintos tasks pueden escribir en el mismo canal concurrentemente haciendo un llamado put!() y múltiples lectores en distintos tasks pueden leer datos concurrentemente con un llamado take!().

Si un canal está vacío, una llamada a take!() bloqueará al lector hasta que los datos estén disponibles, al igual que si un canal está lleno, una llamada put! () bloqueará al escritor

hasta que haya espacio disponible.

In [33]:
```
# Se genera la lista de tasks
t = @task Any[ for x in [1,2,4] println(x) end ]

# No hay tasks?
println(istaskdone(t))

# Cual es la siguiente task?
println(current_task())

# Ejecutar task
println(consume(t))
```

```
false
Task (runnable) @0x00007fb443677490
1
2
4
Any[nothing]
```

La plataforma de programación paralela de Julia usa Tasks para cambiar entre múltiples cálculos.

Paralelismo:

Como Julia está pensado para ser útil para la computación científica busca favorecerla permitiendo utilizar todos los recursos disponibles en una máquina, como los procesadores múltiples. Por esto Julia permite correr operaciones en paralelo, este se basa en el paso de mensajes entre programas corriendo en varios procesadores con memoria compartida, donde cada proceso tiene un id (siendo el 1 el principal y el resto “workers”).

Este paso de mensajes es unilateral y está construido sobre dos recursos: remote references y remote calls. Al hacerse un remote call se obtiene una remote reference *Future* y el valor de esta se puede recuperar usando la función fetch ().

In [1]:
```
addprocs(2)
for w in workers()
    rref=remotecall(myid, w)
    sleep(1)
    println(fetch(rref))
end
```

```
2
3
```

<u>Ciclos</u>

In [30]:
```
nheads = @parallel (+) for i=1:200000000
  Int(rand(Bool))
end
```

Out[30]: 99997168

Este es un ciclo for que se realiza de manera paralela entre los workers disponibles y la última operación suele ser conocida como reducción, la cual es parte de un patrón de computación paralela usual, en este caso se resuelve de manera paralela las iteraciones y los resultados se reducen a través de la adición.

Al usar un for paralelo hay que tener en cuenta que las instrucciones no se ejecutan en un

orden específico y que no se está manejando al mismo worker, por lo que las escrituras a variables y arrays no serán visibles globalmente ya que las iteraciones corren en distintos procesos y cada uno tendrá su propia copia de la variable o array. Por esto se necesitan utilizar variables como Shared Arrays.

In [32]:
```
a = SharedArray(Float64, 100000)
@parallel for i=1:100000
  a[i] = i
end
a
```

Out[32]:
```
100000-element SharedArray{Float64,1}:
     1.0
     2.0
     3.0
     4.0
     5.0
     6.0
     7.0
     8.0
     9.0
    10.0
    11.0
    12.0
    13.0
     ⋮
 99989.0
 99990.0
 99991.0
 99992.0
 99993.0
 99994.0
 99995.0
 99996.0
 99997.0
 99998.0
 99999.0
100000.0
```

Se puede reemplazar el @Parallel utilizando la función pmap para que ejecute una función en paralelo dividendo el trabajo entre los workers cuando cada llamada a función realiza una gran cantidad de trabajo.

Variables Globales

Las llamadas remotas con referencias globales integradas (solo en el módulo principal) administran las variables globales generando bindings de estas variables en los procesos a los que se llama, volviendolo a enviar a un trabajador de destino sólo si su valor ha cambiado.

Ejemplo de Machine Learning:

Este punto es muy importante a la hora de estudiar el lenguaje, ya que éste ha sido creado para encarar problemáticas computacionales como las que el programador se topa clásicamente en Machine Learning. Esto es, una gran cantidad de datos a los cuales hay que aplicarles una serie de funciones o transformaciones matemáticas con el objetivo de manipular los mismos y obtener algún tipo de conclusión sobre ellos: ya sea graficarlos para encontrar su distribución o bien predecir un valor asociado al dato.

En el ejemplo que adjuntamos a continuación aplicamos el algoritmo de KMeans++ a un set de

datos que sacamos de R (utilizando un package de Julia llamado Rdatasets). Éste algoritmo resuelve el problema de Clustering, recibiendo como hiper-parámetro la cantidad de clusters que hay en los datos. Para ver eso, primero graficamos la distribución de los mismos en R^2 (utilizamos PyPlot para realizar el gráfico).

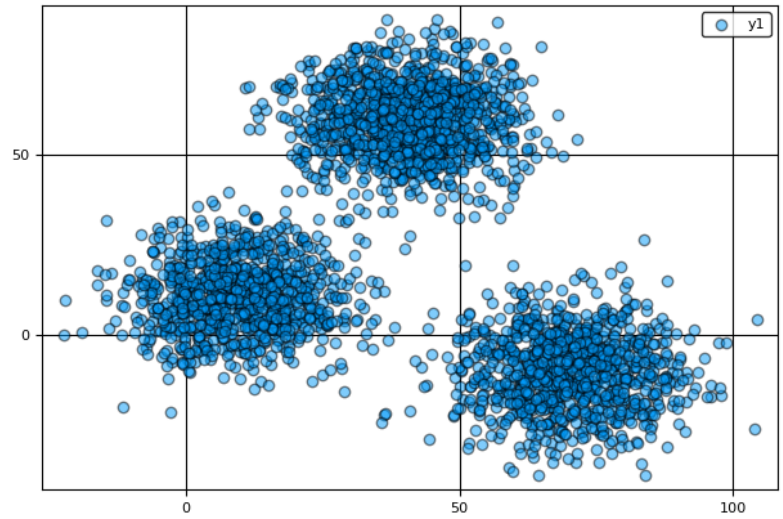
In [1]:
```
using RDatasets
xclara = dataset("cluster", "xclara")
size(xclara)
```

Out[1]:
```
(3000,2)
```

In [2]:
```
x = xclara[:V1]
y = xclara[:V2]
using Plots
scatter(x, y ,alpha=0.5)
```

```
[Plots.jl] Initializing backend: pyplot
```

Out[2]:



In [3]:
```
using Clustering

xclara = convert(Array, xclara)
xclara = xclara'
xclara_kmeans = kmeans(xclara, 3)
```

```
sys:1: MatplotlibDeprecationWarning: The set_axis_bgcolor function was deprecated in version 2.0. Use set_facecolor
 instead.
```

Out[3]:
```
Clustering.KmeansResult{Float64}(2x3 Array{Float64,2}:
 40.6836   69.9242   9.47805
 59.7159  -10.1196  10.6861 ,[3,3,3,3,3,3,3,3,3,3  …  2,2,2,2,2,2,2,2,2,2],[248.826,97.5463,81.803,16.542,268.662,6
71.193,182.102,25.6328,234.163,256.382  …  466.729,9.36828,133.564,226.749,262.743,260.775,225.043,29.9449,437.147,
243.235],[1149,952,899],[1149.0,952.0,899.0],611605.8806933898,4,true)
```

Por último, incluiremos un ejemplo similar. Ésta vez resolveremos el problema de clustering a través del algoritmo DBSCAN el cual es un poco más avanzado que KMeans++, en general tiene un orden de tiempo mayor, y posee la ventaja por encima del primero que no requiere la cantidad de clusters como hiper-parámetro, sino que este número lo encuentra por sí mismo.

In [4]:
```
using Distances
dclara = pairwise(SqEuclidean(), xclara);

xclara_dbscan = dbscan(dclara, 10, 40);
xclara_dbscan.counts
```

Out[4]:
```
3-element Array{Int64,1}:
 274
 603
 356
```

Estadísticas

Index TIOBE Mayo 2018

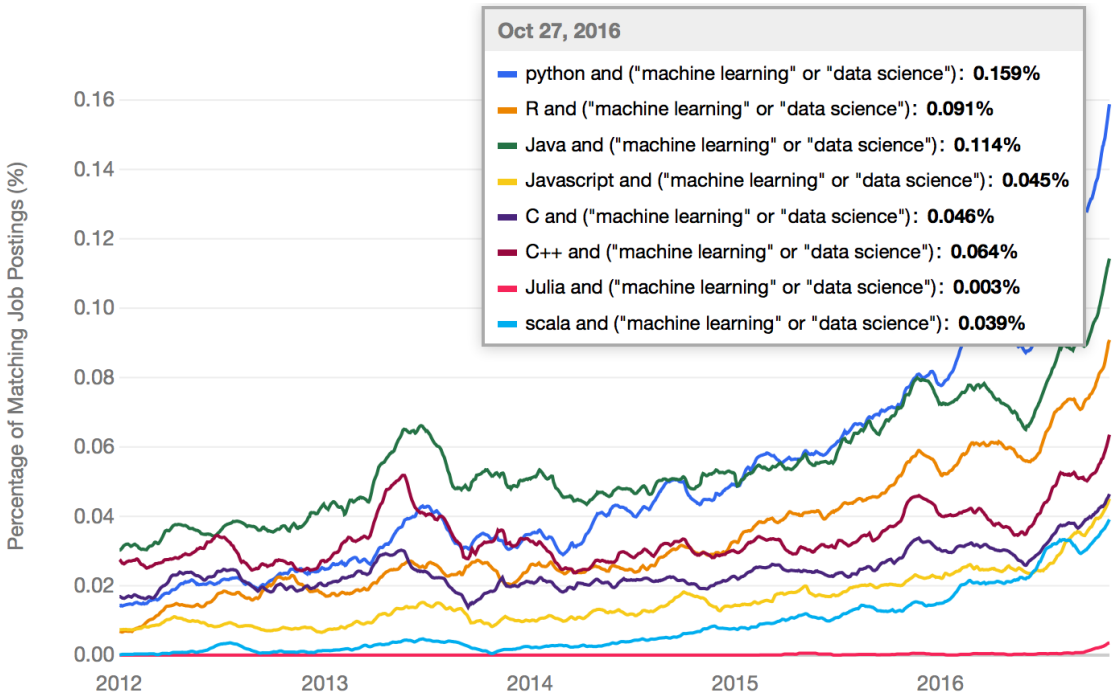
41	OpenCL	0.415%
42	ABAP	0.409%
43	REXX	0.382%
44	Scheme	0.381%
45	ML	0.377%
46	Julia	0.342%
47	ActionScript	0.321%
48	Haskell	0.320%
49	Kotlin	0.292%
50	RPG	0.281%

Populridad en Githut en 2018

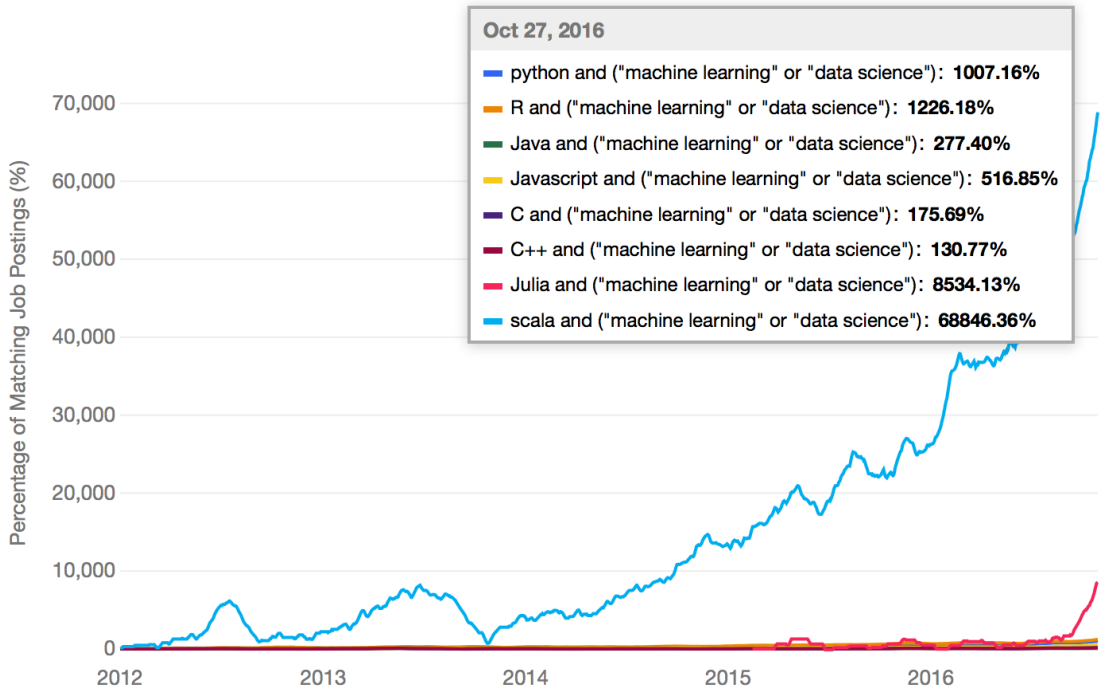
31	Vim script	0.139%	
32	HCL	0.138%	⌃⌃
33	F#	0.127%	⌃
34	Puppet	0.117%	⌄
35	Julia	0.112%	⌄⌄
36	Smalltalk	0.107%	⌃⌃
37	Matlab	0.096%	⌄
38	Visual Basic	0.074%	⌄⌄
39	Crystal	0.071%	⌃⌃
40	FreeMarker	0.053%	⌃⌃

A fines del 2017 Julia subió al puesto 27 de popularidad en Githut, en la atualidad decayó al puesto 35, lo que muestra que en Github se está usando menos el lenguaje para proyectos.

Popularidad de lenguajes en ofertas laborales en cuanto a machine learning o data science en 2016



Julia parece no ser muy considerado para machine learning o data science pero cuando cambiamos a la popularidad relativa vemos que últimamente su uso está aumentando mucho.



Conclusiones

1. Julia es un lenguaje muy nuevo, con una comunidad de desarrolladores realmente muy reducida, pero esto se debe en gran parte a su condición de lenguaje reciente. Las últimas proyecciones muestran un crecimiento en este punto y es algo esperable dado el potencial que posee.
2. Estadísticamente quedó reflejado que tiene el poder computacional que se buscó desde su concepción. Hoy se encuentra al nivel de los lenguajes más veloces y planteándose como una real alternativa para resolver problemáticas que buscan mejorar órdenes temporales algorítimcos.
3. Volviendo a hacer hincapié en la reducida cantidad de desarrolladores que posee la comunidad de Julia, éste punto es lo que lo posiciona en la práctica por detrás de otros lenguajes que no muestran el rendimiento que muestra Julia, pero sí poseen numerosísimos recursos disponibles. Es por esto que Julia a su vez trata de permitir al programador importar código de otros lenguajes como por ejemplo Python, C o Fortran.

Referencias

https://julialang.org/

https://juliacomputing.com

https://github.com/DataWookie/MonthOfJulia

https://docs.julialang.org

https://juliabox.com/#