Julia (v1.0)

Julia to język programowania wysokiego poziomu przeznaczony do prowadzenia wydajnych obliczeń numerycznych stoworzony na MIT. Zalety języka Julia w porównaniu do innych podobnych systemów (Octave, Scilab, Matlab):

- jest znacznie nowszy i szybciej się rozwija (choć być może nie stworzono jeszcze takiego ogromu kodu w tym języku)
- kompilacja kodu do kodu maszynowego w czasie działania programu (dzięki LLVM) – dzięki temu np. pętle działają znacznie szybciej
- Łatwość tworzenia ładnych wykresów
- możliwość pisania własnych funkcji, tym samym na rozszerzanie Octave

Przykładowe pola zastosowań to:

- analiza danych
- przetwarzanie obrazów
- analizy statystyczne
- obliczenia w ekonometrii

Istnieją pakiety pozwalające prowadzić w języku Julia obliczenia symboliczne, ale nie jest to główny obszar zainteresowań.

Instalacja pod Linuksem jest z reguły bardzo prosta (z reguły wystarczy zainstalować pakiet julia), pod Windows można ściągnąć gotowy plik wykonywalny: `https://julialang.org/downloads/`. Można też, po uprzednim zalogowaniu korzystać z interfejsu webowego (Jupyter) pod adresem `https://www.juliabox.com/`

1 Konsola

Julia przechowuje w ktalogu na dysku skompilowane wersje programów i pakietów. Na komputerach wydziałwych katalog ten domyślnie położony jest w katalogu wędrującego profilu i szybko go zapełnia. Aby wskzać lepszą lokalizację dla tego katalogu, można wykonać (w `cmd.exe`):

```
setx JULIA_DEPOT_PATH J:\.julia\
```

Po uruchomieniu Julia czeka na polecenia. Po naciśnięciu klawisza Enter polecenie zostanie wykonane a jego wynik wyświetlony na ekranie. Aby nie wyświetlać wyniku polecenie należy zakończyć średnikiem. Dostępną mamy specjalną zmienną `ans`, która przechowuje wynik ostatniego polecenia.

```
julia> 1
1

julia> 1;

julia> 2^30
1073741824

julia> 2^63
-9223372036854775808

julia> 2^64
0

julia> 7^1000
-5479429532463080511

julia> BigInt(2)^64
18446744073709551616
```

W trybie interaktywnym dostępne są specjalne tryby po naciśnięciu klawiszy przy pustym wierszu:

- `?` – pomoc i dokumentacja – można wpisać nazwę funkcji lub wyszukać z użyciem `apropos`
- `;` – powłoka systemowa
- `]` – menadżer pakietów – można instalować, auktualniać i testować pakiety języka Julia

2 Zmienne

Zmienne skalarne tworzymy i nadajemy im wartości używając operatora `=`:

```
a = 1
```

Wielkośc liter ma znaczenie: `A` to nie to samo co `a`. Poza tym nazwy zmiennych mogą być dowolne (z wszystkimi znakami Unikodu włącznie), choć dobrze trzymać się konwencji: liczby i wektory zaczynają się małymi literami, macierze wielkimi.

3 Typy

Każda zmienna w języku Julia ma typ. Najważniejsze wbudowane typy to: `Float64`, `BigFloat`, `Complex`.

Jeżeli nie zadeklarujemy typu, to Julia spróbuje zmieścić zmienną w taki typ, który najbardziej jej pasuje. Zmiennej można przypisać wartości różnych typów (mówimy o języku dynamicznym), ale jednocześnie Julia raczej nie zmienia typów za programistę (mówimy o silnej typizacji).

Jeżeli chcemy zaznaczyć jakiego typu jest argument funkcji (bądź pole struktury), to używamy podwójnego dwukropka po nazwie parametru:

```
polyval(p::Poly, x)
```

3.1 Liczby stałoprzecinkowe

Najważniejsze typy stałoprzecinkowe to: `Int64`, `UInt64`, `BigInt`. Jak łatwo się domyślić, dwa pierwsze to liczby 64-bitowe (ze znakiem lub bez), ostatnie to liczby dowolnej długości (ale czas wykonywania działa zależy od wielkości liczb i jest sporo dłuższy niż dla pozostałych typów).

```
julia> dump(1000000000)
Int64 1000000000

julia> dump(100000000000000000000000000000000)
Int128 100000000000000000000000000000000

julia> dump(10000000000000000000000000000000000000000000000000000000000000)
BigInt
  alloc: Int32 5
  size: Int32 4
  d: Ptr{UInt64} @0x0000563d773dcee0

julia> dump(1000000000^1000)
Int64 0

julia> dump(100000000000000000000000000000000^1000)
Int128 0

julia> dump(10000000000000000000000000000000000000000000000000000000000000^1000)
BigInt
  alloc: Int32 3020
```

```
  size: Int32 3011
  d: Ptr{UInt64} @0x0000563d77527420
```

3.2 Liczby wymierne

Julia wspiera obliczenia na liczbach wymiernych (do pewnego stopnia). Służy do tego typ `Rational` (licznik i mianownik mogą być dowolnego typu stałoprzecinkowego, w szczególności `BigInt`). Skrótowo liczby takie można zapisywać używając operatora `//`:

```
julia> 145//234+11//87
5063//6786

(v1.1) pkg> add ContinuedFractions#master
   Cloning git-repo `https://github.com/johnmyleswhite/ContinuedFractions.jl.git`
  Updating git-repo `https://github.com/johnmyleswhite/ContinuedFractions.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.1/Project.toml`
  [b28563ba] + ContinuedFractions v0.1.0+ #master (https://github.com/johnmyleswhite/Continue
  Updating `~/.julia/environments/v1.1/Manifest.toml`
  [b28563ba] + ContinuedFractions v0.1.0+ #master (https://github.com/johnmyleswhite/Continue

julia> using ContinuedFractions

julia> collect(convergents(ContinuedFraction(Float64(MathConstants.pi))))
14-element Array{Rational{Int64},1}:
         3//1
        22//7
       333//106
       355//113
    103993//33102
    104348//33215
    208341//66317
    312689//99532
    833719//265381
   1146408//364913
   4272943//1360120
   5419351//1725033
  80143857//25510582
 165707065//52746197
```

3.3 Liczby zmiennoprzecinkowe

Julia wspiera obliczenia numeryczne, więc implementuje typy zmiennoprzecinkowe: `Float64`, `BigFloat`.

Pierwszy to znane z innych języków liczby zmiennoprzecinkowe podwójnej precyzji (implementowane sprzętowo). Drugi to liczby dowolnej, ale z góry ustalonej precyzji (precyzja nie rośnie w trakcie obliczeń). Precyzję (w bitach) można zmieniać funkcją `setprecision`.

```
julia> bitstring(0.1)
"0011111110111001100110011001100110011001100110011001100110011010"

julia> sqrt(2)
1.4142135623730951
```

```
julia> sqrt(big(2))
1.414213562373095048801688724209698078569671875376948073176679737990732478462102

julia> setprecision(500)
500

julia> sqrt(BigFloat(2))
1.414213562373095048801688724209698078569671875376948073176679737990732478462107
038850387534327641572735013846230912297024924836055850737212644121497099 7
```

3.4 Tablice

Jak większośc języków programowania Julia wspiera tworzenie Tablic (typ `Array`). Tablice mogą zawierać wartości różych typów, choć my z reguły będziemy w nich przechowywać liczby zmiennoprzecinkowe i będziemy traktować je jako wektory i macierze.

Tablice jednowymiarowe tworzymy używając przecinka:

```
v = [1, 2, 3] // (tablica n-elementowa)
```

Tablice dwywymiarowe tworzymy używając spacji i średników:

```
row = [1 2 3] // wektor rzędowy (macierz 1 x n)
column = [1; 2; 3] // wektor kolumnowy (tablica n-elementowa)
A = [1 2 3;
     4 5 6;
     7 8 9] // macierz
```

Aby zmieniać wartości pojedynczych elementów wektorów i macierzy można używać indeksów (indeksy zaczynają się od 1, nie od 0 jak np. w Pythonie). Macierze (tablice dwuwymiarowe) można traktować jako tablice jednowymiarowe, przy czym elementy są wypisywane kolumnami. Przy powyższych defnicjach możemy więc napisać:

```
row[3] = 4
column[3] = 4
A[3,3] = 10
A[8] = 15
```

Można też pobrać cały rząd czy kolumnę macierzy jako wektor używając dwukropka:

```
A[2,:]
A[:,3]
```

Równie dobrze można rząd czy kolumnę macierzy zastąpić innym wektorem, o ile wymiar (liczba elementów) jest odpowiednia:

```
A[2,:] = row
A[:,3] = row
```

Tablice można sklejać ze sobą. Służą do tego funkcje `vcat` i `hcat`. Pierwsza skleja wzdłuż pierwszego wymiaru, druga wzdłuż drugiego wymiaru. Obie funkcje mają równoważną wersję z nawiasami klamrowymi:

```
[ A; B; C] // vcat(A, B, C)
[ A B C] // hcat(A, B, C)
```

Przydatne konstrukcje:

- `rand(n), rand(n, m)` – tworzy wektory i macierze o niezależnych wejściach z rozkładu jednostajnego z przedziału $[0, 1)$

- `randn(n), randn(n, m)` – tworzy wektory i macierze o niezależnych wejściach ze standardowego rozkładu normalnego
- `ones(T, n), ones(T, n, m)` – tworzy wektory i macierze wypełnione jedynkami dla zadanego typu `T`
- `zeros(T, n), zeros(T, n, m)` – tworzy wektory i macierze wypełnione zerami dla zadanego typu `T`
- `Matrix{T}(I, n, m)` – tworzy macierz jednostkową wymiaru $n \times m$ zadanego typu `T` (wymaga zaimportowania modułu `LinearAlgebra`)
- `range(a, stop=b, length=n, step=x)` – tworzy obiekt (generator) zwracający kolejne wartości: `a`, `a+x`, `a+2x`, ...; trzeba podać dokładnie jeden z argumentów `stop` lub `length` (`step` to domyślnie 1)

4 Przydatne działania i funkcje

Na zmiennych zarówno typów skalarnych jak i macierzowych możemy wykonywać różne działania/wywoływać funckje. Poniżej zestawienie najważniejszych z nich:

- `2A` – mnożenie zmiennych skalarnych i macierzy przez liczby
- `A+B` – dodawanie liczb i macierzy tego samego wymiaru
- `2I+A` – dodawanie macierzy jednostkowej (odpowiedniego wymiaru)
- `A*B` – normalny iloczyn macierzy
- `dot(A, B)` – iloczyn skalarny wektorów (macierzy) tego samego wymiaru
- `sum(A)` – suma elementów macierzy
- `prod(A)` – iloczyn elementów macierzy
- `min(A)` – element najmniejszy
- `max(A)` – element największy
- `a*A` – iloczyn macierzy przez liczbę
- `A^a` – potęga macierzy kwadratowej
- `A.^a` – podniesienie wszystkich elementów macierzy (niekoniecznie kwadratowej) do potęgi
- `A.*B` – iloczyn Hadamarda macierzy (tego samego wymiaru)
- `exp(A)` – funkcja wykładnicza dla macierzy kwadratowej (suma szeregu)
- `exp.(A)` – zastosowanie funkcji do każdego elemntu macierzy
- `log(A)` – logarytm naturalny macierzy
- `log.(A)` – macierz logarytmów naturalnych
- `sqrt(A)` – pierwiastek macierzy
- `sqrt.(A)` – macierz pierwiastków
- `det(A)` – wyznacznik macierzy kwadratowej

- `inv(A)` – macierz odwrotna
- `cond(A)` – wskaźnik uwarunkowania macierzy
- `eigen(A)` – wartości i wektory własne macierzy
- `rank(A)` – rząd macierzy
- `tr(A)` – ślad macierzy
- `A'` – macierz hermitowsko sprzężona do `A`
- `transpose(A)` – macierz transponowana do `A`
- `A\B` – rozwiązuje równanie macierzowe $Ax = B$
- `B/A` – rozwiązuje równanie macierzowe $xA = B$
- `Polynomials.polyval(p::Polynomials.Poly, x)` – znajduje wartość wielomianu o współczynnikach z tablicy `p` w punkcie `x`
- `Polynomials.polyint(p::Polynomials.Poly)` – znajduje całkę wielomianu zadanego tablicą `p`
- `Polynomials.polyder(p::Polynomials.Poly)` – znajduje pochodną wielomianu zadanego tablicą `p`
- `Polynomials.roots(p::Polynomials.Poly)` – znajduje pierwiastki wielomianu zadanego tablicą `p`
- `Polynomials.polyfit(x, y, n=length(x)-1)` – znajduje wielomian stopnia `n` interpolujący punkty (`x` `y`)

5 Zarządzanie pakietami

Kod w języku Julia zorganizowany jest w moduły i pakiety. Moduł to jednostka organizacji kodu (aby dodać funkcjonalność do projektu importujemy moduły). Kilka modułów tworzy pakiet, jednostkę instalacji (wiele pakietów zawiera tylko jeden moduł). Część modułów wbudowana jest w język (np. `LinearAlebra`), ale wiele przydatnych funkcjonalności nie jest częścią języka Julia. Są one częścią dodatkowych, instalowanych osobno pakietów. W szczególności twórcy tych pakietów nie muszą być powiązani z twórcami języka Julia. To rozwiązanie ma zalety i wady: poprawki mogą trafiać do użytkowników pomiędzy kolejnymi wersjami języka, z drugiej strony wystepują czasem promy ze zgodnością.

Julia zawiera wbudowany menadżer pakietów, który instaluje i aktualizuje pakiety. Najprostszym sposobem uzywania menadżera jest wciśnięcie klawisza `]`. Najważniejsze polecenia menadżera to:

- `add` – dodaje pakiet do listy pakietów żądanych przez projekt, instalując odpowiednie zależności jeśli trzeba
- `rm` – usuwa pakiet
- `up` – uaktualnia zainstalowane pakiety do najnowszych wersji

W teorii można mieć kilka projektów z różnymi pakietami i różnymi wersjami pakietów, my nie będziemy tego robić. Pakiety są instalowane w domyślnym miejscu, które niestety zajmuje miejsce w katalogu domowym. Aby zmienić miejsce zapisywania pakietów używamy zmiennej środowiskowej `JULIA_DEPOT_PATH`. Może być to przydatne na uczlnianych instalacjach Windows (gdzie paiety są domyślnie instalowane na wędrującym profilu – który jest niewielki).

Aby używać *zainstalowanego* modułu używamy w języku Julia poleceń `import` i `using`. Co do zasady robią prawie to samo, z tym, że po `import` importowane typy i funkcje musimy poprzedzać nazwą pakietu, po `using` nazwę pakietu możemy pominąć. Dla przykładu, poniżej mamy przkład z modułem (i pakietem) `Polynomials`:

```
(v1.0) pkg> add Polynomials
   Cloning default registries into /dev/shm/registries
   Cloning registry General from "https://github.com/JuliaRegistries/General.git"
  Updating registry at `/dev/shm/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
 Installed Polynomials — v0.5.1
  Updating `/dev/shm/environments/v1.0/Project.toml`
  [f27b6e38] + Polynomials v0.5.1
  Updating `/dev/shm/environments/v1.0/Manifest.toml`
  [f27b6e38] + Polynomials v0.5.1
  [2a0f44e3] + Base64
  [8ba89e20] + Distributed
  [b77e0a4c] + InteractiveUtils
  [8f399da3] + Libdl
  [37e2e46d] + LinearAlgebra
  [56ddb016] + Logging
  [d6f4376e] + Markdown
  [9a3f8284] + Random
  [9e88b42a] + Serialization
  [6462fe0b] + Sockets
  [2f01184e] + SparseArrays
  [8dfed614] + Test

julia> import Polynomials
[ Info: Precompiling Polynomials [f27b6e38-b328-58d1-80ce-0feddd5e7a45]

julia> polyfit([0, 1, 2], [0, 1, 4])
ERROR: UndefVarError: polyfit not defined
Stacktrace:
 [1] top-level scope at none:0

julia> Polynomials.polyfit([0, 1, 2], [0, 1, 4])
Poly(1.0*x^2)

julia> using Polynomials

julia> polyfit([0, 1, 2], [0, 1, 4])
Poly(1.0*x^2)
```

6 Wykresy

Tworzenie wykresów nie jest funkcjonalnością wbudowane w język. Może być to zaskakujące dla języka, którego głównym zastosowaniem mają być obliczenia, ale proszę pamiętać, że nie wszystkie komputery używane do obliczeń mają w ogóle wyświetlacze graficzne. Poza tym samo wyświetlenie wykresu bardzo mocno zależy od używanego systemu operacyjnego czy spodziewanej funkcjonalności (np. czy użytkownik może taki wykres powiększać).

Podstawowym narzędziem służącym do tworzenia wykresów jest pakiet `Plots`. Pakiet ten nie tworzy właściwie sam wykresów, ale unifikuje pod jednym interfejsem programistycznym obsługę kilku już istniejących mechanizmów tworzenia wykresów (*backendów*).

- matplotlib – pakiet do tworzenia wykresów w języku Python
- GR – bibliotek do tworzenia i wyświetlania grafiki z naciskiem na wykresy
- plotly – biblioteka do tworzenia wizualizacji danych w języku JavaScript (wyświetlanych w przeglądarce)

Backendy mają niestety różne możliwości, pakiet `Plots` to coś w rodzaju najmniejszego wspólnego mianownika. Z wyżej wymienionych mechanizmów najmniej problematyczny jest `plotly`, który tworzy interaktywny plik z wykreseme i wyświetla go w przeglądarce. Unimożliwia to pewne formy interakcji.

Poiżej mamy przykład jak tworzyć proste wykresy. Bardzije skompplikowane przykłady zobaczymy później:

```
julia> using Plots

julia> plotly()
Plots.PlotlyBackend()

julia> x = range(-π, stop=π, length=200)
-3.141592653589793:0.03157379551346526:3.141592653589793

julia> s = sin.(x);

julia> c = cos.(x);

julia> plot(x,[s, c])

julia> plot(x,[s, c], label=["sin" "cos"])

julia> plot(x,[s, c], label=["sin" "cos"], aspect_ratio=:equal)

julia> scatter(s, c)
```

7 Makra

Kod w języku Julia jest parsowany do postaci pośredniej, jak w każdym innym kompilatorze. Nie każdy język daje jednak dostęp do tak powstałej reprezentacji na poziomie języka. Julia pozwala na modyfikację i przeglądanie kodu powstałego przez parsowanie wyrażeń języka. Pozwala to na tzw. *metaprogramowanie*. Cały temat zdecydowanie wykracza poza ramy tego wprowadzenia, ale jest kilka tzw. makr, które mogą być przydatne czy chociaż ciekawe. Makra generują kod, który jest potem kompilowany i wykonywany prze Julię.

- `@which` – makro to pozwala stwierdzić, która funkcja zostanie użyta w wywołaniu
- `@less` – makro to wyświetla kod funkcji (bibliotecznej), ktora zostanie wywołana
- `@code_lowered`, `@code_typed`, `@code_llvm`, `@code_native` – makra wyświetlają kod, który zostanie wykonany, dwa pierwsze wyświetlają reprezentacje w języku Julia, trzecie – kod pośredni LLVM, ostatnie makro wyświetla skompilowany kod maszynowy w asemblerze (składnia AT&T)
- `@evalpoly` – generuje kod wykonujący algorytm Hornera
- `@fastmath` – generuje kod, który może naruszać standard IEE754, ale prawdopodobnie wykona się szybciej
- `@time` – wyświetla czas potrzeny na obliczenia

```
julia> @which 1+1
+(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt

julia> @which 1+big(1)
+(c::Union{Int16, Int32, Int64, Int8}, x::BigInt) in Base.GMP at gmp.jl:447

julia> @which big(1)+big(1)
+(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:403
```

```
julia> @less big(1)+big(1)

julia> f(x) = x+1
f (generic function with 1 method)

julia> @code_lowered f(7)
CodeInfo(
1 ─ %1 = x + 1
└──      return %1
)

julia> @code_typed f(7)
CodeInfo(
1 ─ %1 = (Base.add_int)(x, 1)::Int64
└──      return %1
) => Int64

julia> @code_llvm f(7)

;  @ REPL[0]:1 within `f'
define i64 @julia_f_12288(i64) {
top:
; ┌ @ int.jl:53 within `+'
   %1 = add i64 %0, 1
; └
  ret i64 %1
}

julia> @code_native f(7)
        .text
; ┌ @ REPL[0]:1 within `f'
; │┌ @ REPL[0]:1 within `+'
        leaq    1(%rdi), %rax
; │└
        retq
        nopw    %cs:(%rax,%rax)
; └

julia> f(x) = @evalpoly(x,2,3,4)
f (generic function with 1 method)


julia> @code_typed f(7)
CodeInfo(
1 ─      goto #3 if not false
2 ─      nothing::Nothing
3 ┄ %3 = (Base.mul_int)(x, 4)::Int64
│   %4 = (Base.add_int)(%3, 3)::Int64
│   %5 = (Base.mul_int)(x, %4)::Int64
│   %6 = (Base.add_int)(%5, 2)::Int64
└──      return %6
) => Int64
```

```
julia> fib(n) = n <= 1 ? big(1) : fib(n-1)+fib(n-2)
fib (generic function with 1 method)

julia> @time fib(10)
  0.000081 seconds (446 allocations: 7.758 KiB)
89

julia> @time fib(20)
  0.002918 seconds (54.73 k allocations: 940.781 KiB)
10946

julia> @time fib(30)
  0.327900 seconds (6.73 M allocations: 112.983 MiB, 29.49% gc time)
1346269

julia> @time fib(40)
 40.852016 seconds (827.90 M allocations: 13.570 GiB, 31.18% gc time)
165580141

julia> fib(n) = n<= 1 ? 1 : fib(n-1)+fib(n-2)
fib (generic function with 1 method)

julia> @time fib(40)
  0.533209 seconds (4.07 k allocations: 255.487 KiB)
165580141
```