

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

KOPILICA 5, SPLIT

CASE STUDY:

PROGRAMSKI JEZIK JULIA

Student: Marin Kljaković-Gašpić

Smjer: Računarstvo

Split, prosinac, 2024.

SADRŽAJ:

Sadržaj

1. UVOD	3
2. OSNOVNO O JULIJU	4
3. POVIJEST PROGRAMSKOG JEZIKA JULIA	6
4. SVOJSTVA PROGRAMSKOG JEZIKA JULIA	8
4.1 Definiranje funkcija	8
4.2 Kontrolne strukture	9
4.3 Strukture podataka	12
4.4 Nasljeđivanje i „Multiple dispatch“	13
4.5 Rad s paketima	14
5. NAPREDNI KONCEPTI PROGRAMSKOG JEZIKA JULIA	15
5.1 Parametarski polimorfizam.....	15
5.2 Makroi i metaprogramiranje	15
5.3 Paralelno i distribuirano računanje	17
5.4 Rad sa tipovima i hijerarhija tipova	20
5.5 Exceptions	23
6. ZAKLJUČAK	24
7. LITERATURA	25

1. UVOD

Kada radimo na znanstvenim projektima kao što su obradu podataka, strojno učenje i umjetnu inteligenciju, imamo veliki izbor programskih jezika. Na primjer, među poznatijim programskim jezicima imamo Python, C/C++ i donekle MATLAB. Ovi jezici su u uporabi od njihovog nastanka pri kraju 20. stoljeća. Oni su prošli kroz nekoliko modifikacija i pokazali su svoju kvalitetu i fleksibilnost, ali ne koriste se svi za iste stvari jer se razlikuju prema svojim načinima funkcioniranja. Bitno je, kada izabiremo jezik za gotovi program ili aplikaciju, da izaberemo onaj koji daje najpoželjniji omjer brzine i jednostavnosti programiranja. Ali, prije toga je potrebno prvo napraviti neki prototip programa kako bi znali što ćemo napraviti na koji način. Ovdje najčešće koristimo neki drugi programski jezik koji je lako razumjeti i koji dopušta najveću kreativnost te upravo tu nastaje problem. Kada završimo sa prototipom, trebamo taj već napravljeni kod u jednom jeziku prebaciti u neki brži i efikasniji jezik koji je možda teži za razumjeti. Ovaj problem nazivamo „two language problem“ ili problem dvaju jezika.

Programski jezik Julia, je nastao 2012. godine kao jezik koji bi riješio taj problem. Cilj Julije je imati istu brzinu kao neki programski jezici bliži samom kodu procesora, dok zadržava jednostavnu sintaksu i fleksibilnost nekih lakših, ali sporijih jezika. Julia se pogotovo ističe u prije navedenim zadacima kao što znanstveno računanje sa velikim brojem podataka i strojno učenje.

Cilj ovog case studyja je istražiti kako radi programski jezik Julia, karakteristike, sposobnost rješavanja nekih svakodnevnih zadataka i usporedba sa nekim poznatijim jezicima kao što su C/C++ i Python.

2. OSNOVNO O JULIJU

Programski jezik Julia je relativno nov programski jezik visoke razine, dizajniran da brzo i efikasno rješava probleme u područjima kao što su znanstveno računanje i obrada podataka, strojno učenje, numeričke analize i simulacije i rada sa velikom količinom podataka.

Julia također ima karakteristike koje razlikuju ovaj jezik od većine drugih jezika kao što su parametarski polimorfizam, „multiple dispatch“, „just-in-time“ (JIT) kompajler, paralelno računanje i to što je dinamički tipizirani jezik.

Parametarski polimorfizam dopušta da nekom komadu koda daje generički tip varijable umjesto nekog specifičnog tipa, te kasnije danom tipu po potrebi dodjeljuje specifični tip. Ovakve funkcije i tipovi podataka se ponekad nazivaju generičke funkcije i generički tip varijable.

„Multiple dispatch“ omogućava da se, na primjer, u kodu definira više funkcija istog imena, ali različitih vrsta ulaznih argumenata. Julia dopušta definiranje više verzija iste funkcije. U mnogim objektno orijentiranim jezicima kao što je C++, funkcije se temelje na tipu prvog argumenta kao što su metode neke klase. MD omogućuje pisanje generičkog koda koji je fleksibilniji i povećava performanse jer se bira najefikasnija verzija funkcije.

Julija je dinamički tipiziran jezik, ovo znači da nije potrebno unaprijed deklarirati tipove varijabli. Ovo pomaže programerima da brže i lakše pišu kod, sličnu prednost ima Python, naravno sa potrebom deklariranja varijabli. Julia sama prepoznaje tipove varijabli tokom izvršavanja.

„Just in time“ kompajler optimizira izvršavanje koda u stvarnom vremenu i jedan je od najvažnijih razloga zašto je Julia toliko brza u usporedbi sa nekim drugim programskim jezicima. JIT kompajler koristi „Low Level Virtual Machine“ kako bi optimizirao kod i pretvorio ga u strojni jezik. Većina drugih dinamičkih jezika interpretira kod liniju po liniju, što je sporije. Julia ih kompajlira prije izvršavanja i tako dobiva velike brzine usporedive sa jezikom kao što je C. JIT kompajler radi tako da nakon što programer upiše kod koristeći dinamičku sintaksu, taj se kod dinamički kompajlira pomoću LLVM u strojni kod i izvrši se direktno na procesoru. Kada se taj isti kod pozove drugi put, on je već spremljen i ne kompajlira se ponovno, što znatno ubrzava izvođenje. Uz sve te prednosti postoje i nedostaci, kao što je usporeni prvi poziv neke funkcije. Ovo se događa jer, kao i svi programski jezici, Julia treba prvo kompajlirati funkciju prije nego što je izvrši. Sama interpretacija koda zahtijeva manje memorije nego prvo kompajliranje.

Ugrađena podrška za paralelno i distribuirano računanje je jedna ključnih prednosti koje ima Julia. Paralelno računanje je sposobnost korištenja više procesorskih jezgri za više zadataka u isto vrijeme. Na ovaj način se ubrzava obrada velikih količina podataka.

Distribuirano računanje dopušta izvršavanje zadataka na više računala u isto vrijeme koji rade zajedno u svrhu rješavanja istog problema. Paralelno i distribuirano računanje u Juliji se nalazi u standardnoj biblioteci, u paketu „Distributed“, kasnije još o paketima.



Slika 1.) Logo programskog jezika Julia

3. POVIJEST PROGRAMSKOG JEZIKA JULIA

Rad na programskom jeziku Julia je počeo 2009. godine kada je grupa ljudi odlučila napraviti jezik koji je besplatan, brz i više razine. Ti ljudi su Jeff Bezanson, Stefan Karpinski, Viral B. Shah i Alan Edelman. 14. veljače 2012. godine, osvanula je web stranica na kojoj su objasnili svrhu izrade jezika. Julia je stvorena jer, kako tim kaže na svome blogu: „Ukratko, zato što smo pohlepni.“

Postojeći jezici su bili dobri u onome što su radili, ali svaki od njih je imao svoj neki problem:

- C/Fortran su bili brzi, ali relativno složeni jezici za korištenje i nisu dopuštali dinamičku tipizaciju.
- R je bio vrlo dobar za statističku analizu, ali nije bio baš koristan za masovne količine podataka jer je spremao sve te podatke u sistemsku memoriju.
- Matlab je bio napravljen za znanstvene proračune, ali je bio spor i zatvorenog koda.
- Python je bio relativno jednostavan i fleksibilan, ali je imao problema sa brzinom obrade velikog broja podataka bez dodatnih biblioteka kao što je „Numpy“.

S ciljem da se stvore jezik koji je otvorenog koda, brzine C jezika, dinamičnosti kao Ruby, moćan za matematičko računanje kao Matlab, jednostavan za korištenje kao Python i sposoban za rukovanje podacima kao R, tim je krenuo u posao.

Isti dan, 14. veljače, prva verzija Julije je bila objavljena. Tim je javno predstavio svoj jezik na mailing listi i GitHub repozitoriju i od tada je Julia postala dostupna svima otvorenog koda pod MIT licencom. Već u prvoj verziji Julija je imala neke od svojih ključnih značajki kao što su JIT kompajler, „Multiple dispatch“, dinamička tipizacija i podrška za linearnu algebru i matrične operacije. Kako je bila napravljena za znanstveno i tehničko računanje, mnogi znanstvenici i inženjeri su pokazali interes prema novonastalom programskom jeziku.

Ubrzo nakon inicijalne objave, Julia je postala poznata po jednostavnosti korištenja, vrlo dobrim performansama i „Multiple dispatch-u“. Tada su razvijeni i prvi paketi za Juliju kao što su „DataFrames.jl“ za obradu podataka, „Plots.jl“ za vizualizaciju i „DifferentialEquations.jl“ za rješavanje diferencijalnih jednadžbi.

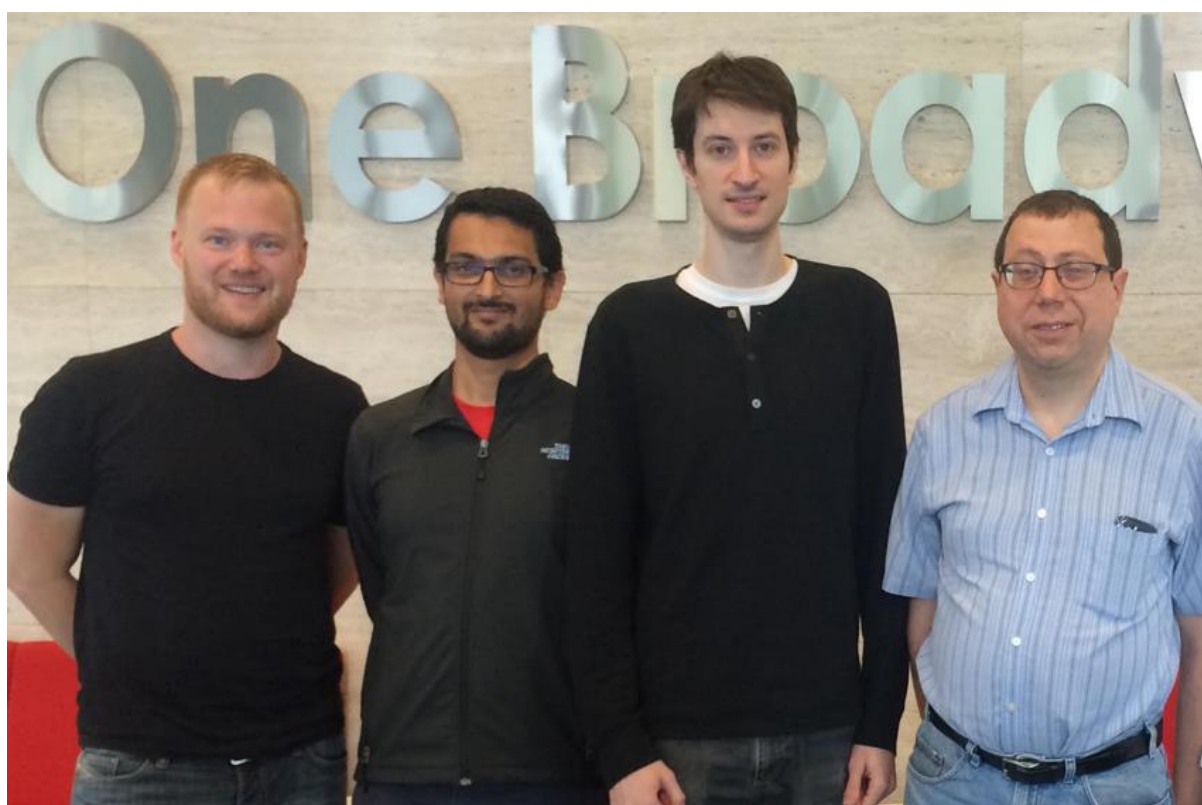
U razdoblju od 2016. i 2018. godine, tim je radio na poboljšanju i optimizaciji JIT kompajlera i „Multiple dispatch-a“. Dodana je također i podrška za multithreading, poboljšana je interoperabilnost sa C i Pythonom i poboljšana je standardna biblioteka.

2018. godine, tijekom JuliaCon-a, objavljena je Julia 1.0 koja je trebala biti stabilna verzija jezika. To je značilo da je nebi bilo više značajnijih promjena koje bi stvarale probleme sa kompatibilnosti koda i da bi jezik bio konačno spreman za industrijsku uporabu.

Nakon te objave, Julia je počela vrlo brzo rasti u popularnosti u znanstvenoj zajednici i postala je najbrže rastući jezik na GitHub-u u nekoliko uzastopnih godina. Počela se sve više koristiti za strojno učenje, numeričke simulacije i analizu velikog broja podataka.

Od verzije 1.5 pa nadalje, najveći fokus Julije je bio na poboljšanju performansi i stabilnosti. Tada Julija postaje biti korištena u financijskoj industriji, farmaceutskoj industriji i strojnom učenju.

Julija je danas jedan od najbržih dinamičkih jezika i najviše se koristi u industriji i znanstvenom sektoru. NASA koristi Juliju za simulacije u svemirskoj tehnologiji, američki Federal Reserve Bank za financijske simulacije, Microsoft i Google za strojno učenje, kao neki od primjera.



Slika 2.) Kokreatori programskog jezika Julia, s lijeva na desno: Stefan Karpinski, Viral B. Shah, Jeff Bezanson i Alan Edelman

4. SVOJSTVA PROGRAMSKOG JEZIKA JULIA

4.1 Definiranje funkcija

U programskom jeziku Julia, funkcije se mogu definirati na više načina, ovisno o složenosti i sintaktičkoj kratkoći. Funkcije su ključan dio svakog programskog jezika pa tako i Julije jer omogućavaju bolju organizaciju koda. Julia podržava:

- Standardno definiranje funkcija (ključna riječ „function“)

```
julia> function pozdrav(ime)
    println("Pozdrav, $ime !")
end
pozdrav (generic function with 1 method)

julia> pozdrav("Marin")
Pozdrav, Marin !

julia>
```

Slika 3.) Standardna funkcija bez return naredbe

```
julia> function kvadriraj(broj)
    return broj*broj
end
kvadriraj (generic function with 1 method)

julia> kvadriraj(10)
100
```

Slika 4.) Standardna funkcija sa return naredbom

- Jednolinijske funkcije koristeći znak „=“ za kraću sintaksu

```
julia> stupnjevi_u_radijane(stupanj) = stupanj * π / 180
stupnjevi_u_radijane (generic function with 1 method)

julia> stupnjevi_u_radijane(180)
3.141592653589793
```

Slika 5.) Jednolinijska funkcija koja pretvara stupnjeve u radijane

- Anonimne funkcije (lambda funkcije), služe za stvaranje funkcija bez imena

```
julia> kvadriraj2 = broj -> broj^2
#1 (generic function with 1 method)

julia> kvadriraj2(100)
10000
```

Slika 6.) Anonimna (lambda) funkcija

4.2 Kontrolne strukture

Još jedan osnovni dio svakog programskog jezika su kontrolne strukture. Julia ih podržava i one omogućuju donošenje odluka (if petlja) te ponavljanje koda (for i while petlje). Osim navedenih petlji, Julia omogućuje i iteraciju kroz druge iterativne strukture kao što su nizovi, rječnici i skupovi. U usporedbi s drugim dinamičnim jezicima, petlje u Juliji su vrlo brze.

```

julia> broj1=-12
-12

julia> broj2=0
0

julia> broj3=14
14

julia>
    if broj > 0
        println("Broj $broj je pozitivan!")
    elseif broj < 0
        println("Broj $broj je negativan!")
    else
        println("Broj je nula!")
    end
Broj -12 je negativan!

```

Slika 6.) If funkcija

```

julia> niz = [1, 2, 3, 4, 5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> niz = [1, 2, 3, 4, 5];

julia> for i in niz
        print("$i ")
    end
1 2 3 4 5

```

Slika 7.) Standardna for petlja kroz niz

```

julia> for i in 1:5
    print("$i ")
end
1 2 3 4 5
julia> for i in 1:5, j in 1:3
    println("korak u i: $i , korak u j: $j")
end
korak u i: 1 , korak u j: 1
korak u i: 1 , korak u j: 2
korak u i: 1 , korak u j: 3
korak u i: 2 , korak u j: 1
korak u i: 2 , korak u j: 2
korak u i: 2 , korak u j: 3
korak u i: 3 , korak u j: 1
korak u i: 3 , korak u j: 2
korak u i: 3 , korak u j: 3
korak u i: 4 , korak u j: 1
korak u i: 4 , korak u j: 2
korak u i: 4 , korak u j: 3
korak u i: 5 , korak u j: 1
korak u i: 5 , korak u j: 2
korak u i: 5 , korak u j: 3

```

Slika 8.) For petlja sa dva koraka

```

julia> rezultat=1;

julia> while broj > 0
    rezultat=rezultat*broj
    broj=broj-1
end

julia> rezultat
120

```

Slika 9.) While petlja, početna vrijednost varijable broj je 5

4.3 Strukture podataka

Julia koristi ključnu riječ „struct“ za definiranje svojih struktura. Strukture dijelimo na nepromjenjive („immutable“) i promjenjive („mutable“) strukture. Kako im ime kaže, nepromjenjive strukture se nakon inicijalizacije ne mogu mijenjati, zato su i brže i koriste manje memorije. Promjenjive strukture su sporije, ali se mogu mijenjati nakon inicijalizacije.

```
julia> struct Klavir <: Glazbalo
    zvuk::String
end

julia> instrument1=Klavir("zvuk1")
Klavir("zvuk1")

julia> instrument1.zvuk="zvuk2"
ERROR: setfield!: immutable struct of type Klavir cannot be changed
Stacktrace:
 [1] setproperty!(x::Klavir, f::Symbol, v::String)
      @ Base .\Base.jl:53
 [2] top-level scope
      @ REPL[244]:1
```

Slika 10.) Primjer immutable vrste strukture i što se dogodi ako se proba mijenjati

```
julia> mutable struct Harfa <: Glazbalo
    zvuk::String
end

julia> instrument2=Harfa("zvuk2")
Harfa("zvuk2")

julia> instrument2.zvuk="zvuk3"
"zvuk3"
```

Slika 11.) Primjer mutable vrste struktura

4.4 Nasljeđivanje i „Multiple dispatch“

Po pitanju nasljeđivanja, ono ne postoji u tradicionalnom smislu, nego Julia koristi hijerarhiju tipova pomoću posebnih vrsta tipova koje nazivamo apstraktni tipovi („abstract types“), kasnije o njima. „Multiple dispatch“ dopušta Juliji definiranje više verzija iste funkcije koje rade ovisno o svojim ulaznim argumentima. Ovo čini Juliju vrlo fleksibilnim jezikom za znanstvene primjene jer olakšava rad sa podacima koji nisu istog tipa.

```
julia> function ispisi(x::Int)
    println("Broj: ", x)
end
ispisi (generic function with 1 method)

julia> function ispisi(x::String)
    println("Tekst: ", x)
end
ispisi (generic function with 2 methods)

julia> ispisi(13)
Broj: 13

julia> ispisi("Objektno orijentirano programiranje")
Tekst: Objektno orijentirano programiranje
```

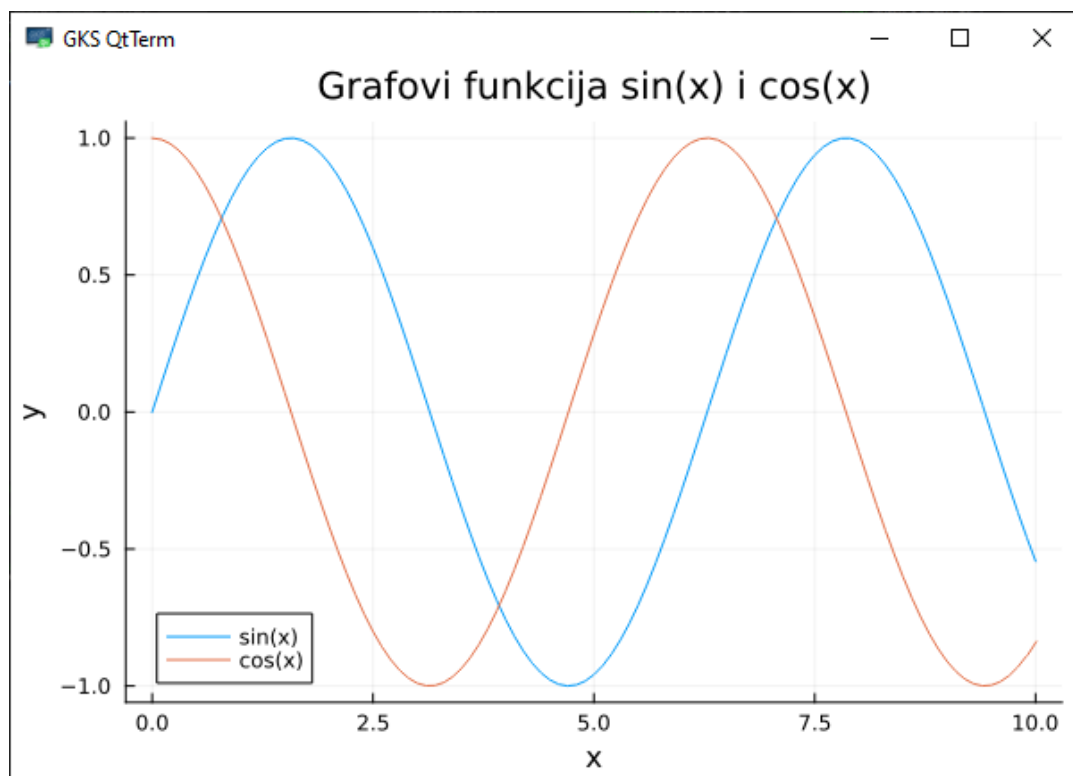
Slika 12.) Primjer „Multiple Dispatch“

4.5 Rad s paketima

Paketi su važan dio Julia ekosustava jer služe za proširenje osnovnih sposobnosti jezika. Oni omogućuju korisnicima da dodaju funkcionalnosti koje im trebaju za specifične zadatke. Neke od ovih funkcionalnosti su vizualizacija, optimizacija i strojno učenje. Paket „Plots“ je vrlo dobar primjer nečeg novog što Julija donosi.

```
julia> using Pkg  
  
julia> Pkg.add("Iml") #npr ("Plots"), ("Benchmark")
```

Slika 13.) Naredbe za dodavanje paketa

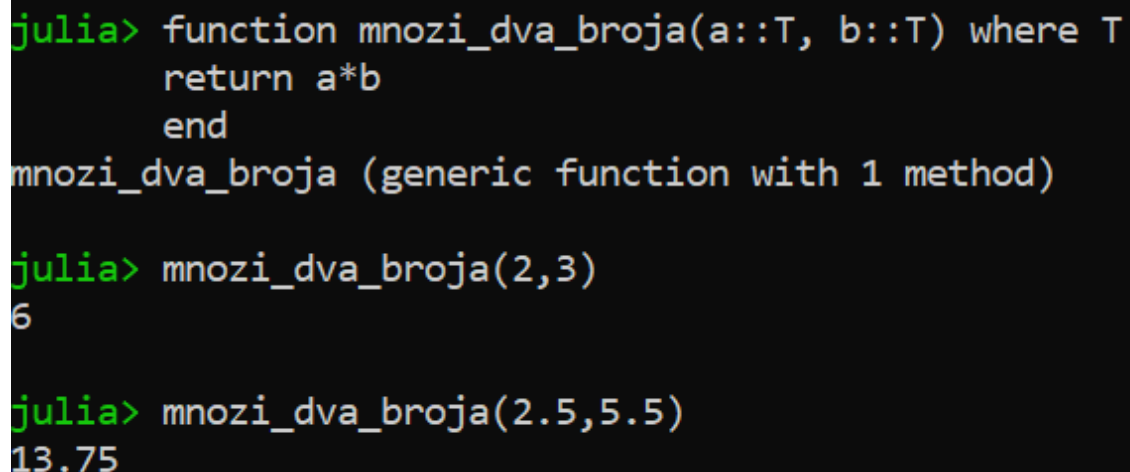


Slika 14.) Crtanje grafova sinusa i kosinusa pomoću paketa „Plots“

5. NAPREDNI KONCEPTI PROGRAMSKOG JEZIKA JULIA

5.1 Parametarski polimorfizam

Prvi napredni koncept koji ću spomenuti je parametarski polimorfizam, on omogućuje definiciju generičkih struktura i funkcija koje rade sa različitim tipovima podataka. Također omogućuje kreiranje generičkih tipova koji rade s podacima bez gubitka efikasnosti, što znači da kod postaje fleksibilniji.



```
julia> function mnozi_dva_broja(a::T, b::T) where T
    return a*b
end
mnozi_dva_broja (generic function with 1 method)

julia> mnozi_dva_broja(2,3)
6

julia> mnozi_dva_broja(2.5,5.5)
13.75
```

Slika 15.) Primjer parametarskog polimorfizma

5.2 Makroi i metaprogramiranje

Metaprogramiranje daje sposobnost pisanja koda koji može analizirati ili stvarati druge dijelove koda za vrijeme izvođenja programa. Ono omogućava da se kod generira i modificira prije nego što se izvrši. Glavni primjer metaprogramiranja koji ću spomenuti je makro. Makro omogućuje stvaranje koda koji generira drugi kod tijekom samog izvođenja. Oni koriste simbol @, te se često koriste za optimizaciju koda tako što smanjuju ponavljanje u kodu.

```
julia> macro pozdrav(ime)
    return :(println("Pozdrav, ", $ime, "!"))
end
@pozdrav (macro with 1 method)

julia> @pozdrav "Marin"
Pozdrav, Marin!
```

Slika 16.) Primjer makroa

Naredba „eval()“ dopušta izvršavanje koda koji je spremljen u neki izraz. Na slici 17. vidimo da je u kod od println(„Zdravo zdravo“) spremljen u izraz „izraz“, te nakon izvršen pomoću naredbe eval.

```
julia> izraz = :(println("Zdravo zdravo"))
:(println("Zdravo zdravo"))

julia> eval(izraz)
Zdravo zdravo
```

Slika 17.) Primjer funkcije „eval()“

Dinamičko generiranje funkcija dopušta kreiranje funkcija u stvarnom vremenu. Ovdje funkcija „napravi_funkciju“ koja prima samo ime, uz pomoć naredbe Meta.parse koja pretvara string u kod, stvara jednostavnu funkciju imena „kvadriraj“ koja će primiti broj i vratiti njegov kvadrat.

```
julia> function napravi_funkciju(ime)
    eval(Meta.parse("function $ime(x) return x^2 end"))
end
napravi_funkciju (generic function with 1 method)

julia> napravi_funkciju("kvadriraj")
kvadriraj (generic function with 1 method)

julia> kvadriraj(12)
144
```

Slika 18.) Dinamičko kreiranje funkcija

5.3 Paralelno i distribuirano računanje

Julia ima alate za paralelno i distribuirano računanje, koji omogućuju korištenje više jezgri procesora i više sustava za brže izvršavanje koda. Neki od modela paralelnog računanja su „Multithreading“ i „Distributed computing“. „Multithreading“ je izvođenje više thread-ova (niti) unutar jednog procesa. To zapravo znači da će se neki program izvršiti na više thread-ova, umjesto samo na jednom. Na Juliji se može upravljati koliko thread-ova dajemo programu Julia, npr. naredbom: „--threads 4“ dajemo 4 thread-a. Koliko thread-ova nam je dostupno možemo provjeriti naredbom: „Threads.nthreads“. Distribuirano računanje je sposobnost izvođenja koda na više procesora ili računala u isto vrijeme. Ovo čini Juliu idealnom za rad sa velikom količinom podataka. Kod distribuiranog računanja dodajemo procese pomoću naredbe „addprocs(broj_procesa)“ i već gotovog makroa `@distributed`. Sve se ove naredbe nalaze u paketu „Distributed“. Na slikama 19. i 20. ćemo vidjeti kako 4 različita „radnika“ obavljaju jednostavan zadatak ispisivanja brojeva od 1 do 100. Naravno, u ovom slučaju nema potrebe koristiti 4 procesa jer je zadatak lagan, ali na ovome se najbolje vidi kako oni funkcioniraju. Može i više procesa, ali u ovom primjeru je samo 4.

```

julia> using Distributed

julia> addprocs(4)
4-element Vector{Int64}:
 2
 3
 4
 5

julia> @distributed for i in 1:100
    println(i)
end
Task (runnable, started) @0x0000027c0b244da0

julia>
      From worker 3:      26
      From worker 4:      51
      From worker 4:      52
      From worker 4:      53
      From worker 4:      54
      From worker 4:      55
      From worker 5:      76
      From worker 5:      77
      From worker 5:      78
      From worker 3:      27
      From worker 3:      28
      From worker 3:      29
      From worker 3:      30
      From worker 3:      31
      From worker 3:      32
      From worker 3:      33
      From worker 3:      34
      From worker 3:      35
      From worker 3:      36
      From worker 3:      37
      From worker 3:      38
      From worker 3:      39
      From worker 3:      40
      From worker 3:      41
      From worker 3:      42
      From worker 3:      43
      From worker 3:      44
      From worker 3:      45
      From worker 3:      46
      From worker 3:      47
      From worker 3:      48
      From worker 3:      49
      From worker 3:      50
      From worker 2:       1
      From worker 4:      56
      From worker 4:      57
      From worker 4:      58

```

Slika 19. Primjer distribuiranog rada

From worker 4:	59
From worker 4:	60
From worker 4:	61
From worker 4:	62
From worker 4:	63
From worker 4:	64
From worker 4:	65
From worker 4:	66
From worker 4:	67
From worker 4:	68
From worker 4:	69
From worker 4:	70
From worker 4:	71
From worker 4:	72
From worker 4:	73
From worker 4:	74
From worker 4:	75
From worker 5:	79
From worker 5:	80
From worker 5:	81
From worker 5:	82
From worker 5:	83
From worker 5:	84
From worker 5:	85
From worker 5:	86
From worker 5:	87
From worker 5:	88
From worker 5:	89
From worker 5:	90
From worker 5:	91
From worker 5:	92
From worker 5:	93
From worker 5:	94
From worker 5:	95
From worker 5:	96
From worker 5:	97
From worker 5:	98
From worker 5:	99
From worker 5:	100
From worker 2:	2
From worker 2:	3
From worker 2:	4
From worker 2:	5
From worker 2:	6
From worker 2:	7
From worker 2:	8
From worker 2:	9
From worker 2:	10
From worker 2:	11
From worker 2:	12
From worker 2:	13
From worker 2:	14
From worker 2:	15
From worker 2:	16
From worker 2:	17
From worker 2:	18

Slika 20. Nastavak ispisa distribuiranog rada

5.4 Rad sa tipovima i hijerarhija tipova

Julija podržava hijerarhiju tipova, ona omogućuje definiranje apstraktnih i konkretnih tipova. Ti apstraktni tipovi služe kao sučelje za konkretne tipove podataka, što znači da konkretni tipovi nasljeđuju apstraktne. Konkretni tipovi su zapravo stvarne strukture podataka u Juliji. Ovo je najbliže što ćemo doći nasljeđivanju kao što postoji u C++ kako Julija nema klasično nasljeđivanje, nego ima „Multiple dispatch“, gdje funkcije definirane za apstraktni tip rade i za njegove podtipove. Također imamo generičke strukture, čiji članovi mogu biti bilo kojeg tipa.

```
julia> abstract type Glazbalo end

julia> struct Klavir <: Glazbalo
        zvuk::String
    end

julia> struct Bujanj <: Glazbalo
        zvuk::String
    end

julia> struct Gitara <: Glazbalo
        zvuk::String
    end

julia> glazbalo1 = Klavir("zvuk1")
Klavir("zvuk1")

julia> glazbalo2 = Bujanj("zvuk2")
Bujanj("zvuk2")

julia> glazbalo3 = Gitara("zvuk3")
Gitara("zvuk3")

julia> glazbalo3.zvuk
"zvuk3"
```

Slika 21. Primjer hijerarhijskih tipova u Juliji

```
julia> struct Objekt{T}
        vrijednost::T
    end

julia> objekt1 = Objekt("abc")
Objekt{String}("abc")

julia> objekt2 = Objekt(-45)
Objekt{Int64}(-45)
```

Slika 22.) Primjer generičke strukture

Naravno, u Juliji također postoje i matrice i nizovi.

```
julia> matrica1=zeros(Int8, 2, 3)
2×3 Matrix{Int8}:
 0  0  0
 0  0  0
```

Slika 23.) Primjer izrade matrice dimenzije 2*3 napunjene nulama

```
julia> niz1=[1,2,3,4,5,6,7]
7-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
 7

julia> niz2=[[1,2],[2,3],[3,4],[4,5]]
4-element Vector{Vector{Int64}}:
 [1, 2]
 [2, 3]
 [3, 4]
 [4, 5]

julia> niz2[1][2]
2

julia> niz1[1]
1
```

Slika 24.) Primjer deklaracije niza u Juliji

5.5 Exceptions

Julia ima vrlo sličan sustav za iznimke kao i C++/Python. Koriste se naredbe „throw“, „try“ i „catch“ za iznimke. Već postoje gotove iznimke, ali mi samo možemo napraviti svoju iznimku koja je struktura.

```
julia> struct Dijeljenje_nulom <: Exception
    poruka::String
end

julia> function dijeli_dva_broja(a::Int,b::Int)
    if b == 0
        throw(Dijeljenje_nulom("S nulom se ne dijeli!"))
    else
        return a/b
    end
end

dijeli_dva_broja (generic function with 1 method)

julia> try
    rez=dijeli_dva_broja(12,0)
    println("Rezultat: ", rez)
catch e
    println("Uhvacena greska: ", e.poruka)
end
Uhvacena greska: S nulom se ne dijeli!
```

Slika 25.) Primjer Exception-a u Juliji

6. ZAKLJUČAK

Programski jezik Julia predstavlja jedan veliki iskorak u razvoju jezika za znanstvene potrebe. On svojom fleksibilnošću, brzinom, dinamičností i jednostavnošću privlači mnoge programere, pa čak i one koji se možda ne bave znanstvenim poslom. Ovaj jezik je napravljen s ciljem da riješi ključne probleme koji se pojavljuju u drugim jezicima, te da se ne žrtvuje produktivnost programera u zamjenu za performanse programa.

„Multiple dispatch“ je bitna značajka Julije, jer dopušta pisanje vrlo učinkovitog, ali još uvijek generičkog koda. „Parametarski polimorfizam“, uz makroe, paralelno i distribuirano računanje dodatno povećava fleksibilnost ovog programskog jezika. Pomoću ovih svih stavki, Julia uzima svoje mjesto kao jedan od najefikasniji i najbržih jezika za rad sa velikom količinom podataka, složenim matematičkim operacijama, vizualizacijom grafova i modeliranjem.

Iako je Julia relativno mlad jezik, zajednica vrlo brzo raste uz sve bolji ekosustav rada, što rezultira da jezik postaje sve više i više konkurentniji u području znanstvenog rada.

Zaključak je da Julia nije samo neki maleni projekt koji je nekoliko ljudi napravilo u svoje slobodno doba iz dosade, nego novi način programiranja i izrade jednostavnijeg, ali još uvijek efikasnog koda. On dokazuje da više nije potrebno raditi kompromise između fleksibilnosti i brzine.

7. LITERATURA

Slike:

[1]https://upload.wikimedia.org/wikipedia/commons/1/1f/Julia_Programming_Language_Logo.svg

[2] <https://news.mit.edu/2018/julia-language-co-creators-win-james-wilkinson-prize-numerical-software-1226>

Ostale slike su uzete sa mog osobnog stolnog računala.

Tekst:

Velika većina:

<https://julialang.org/>

<https://news.mit.edu/2018/julia-language-co-creators-win-james-wilkinson-prize-numerical-software-1226>

<https://www.datacamp.com/blog/what-is-julia-used-for>

<https://www.coursera.org/articles/julia-programming-language>

<https://leftronic.com/blog/julia-programming-language>

<https://www.sciencedirect.com/science/article/pii/S2090123223003521>