Primjena programskog jezika F# u analizi i prikazu podataka

Bilonić, Barbara

Undergraduate thesis / Završni rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:372569

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-20**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKUFAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

PRIMJENA PROGRAMSKOG JEZIKA F# U ANALIZI I PRIKAZU PODATAKA

Završni rad

Barbara Bilonić

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 17.08.2020.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Barbara Bilonić
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4037, 23.09.2019.
OIB studenta:	28445766298
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Primjena programskog jezika F# u analizi i prikazu podataka
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	17.08.2020.
Datum potvrde ocjene Odbora:	09.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 14.09.2020.

Ime i prezime studenta:	Barbara Bilonić
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4037, 23.09.2019.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Primjena programskog jezika F# u analizi i prikazu podataka**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1
1.1 Zadatak završnog rada 1
2. PREGLED TRENUTNOG STANJA U FUNKCIONALNOM PROGRAMIRANJU ZA ANALIZU PODATAKA 3
2.1 Funkcionalno programiranje 3
2.2 Korištenje programskog jezika F# 3
2.3 Funkcionalno programiranje u analizi podataka 4
3. NAČELA FUNKCIONALNOG PROGRAMIRANJA 6
3.1 Funkcionalno programiranje 6
3.1.1 Glavne značajke 6
3.1.2 Lambda račun 7
3.1.3 Povijest razvoja funkcionalnog programiranja 7
3.2 Programski jezici za funkcionalno programiranje 8
3.3 Programski jezik F# - svojstva i ograničenja, mogućnosti 8
4. PROGRAMSKI JEZIK F# 10
4.1 O jeziku F# 10
4.2 Povijest jezika F# 12
4.3 Osnove sintakse i građe programskog rješenja 14
4.3.1 Tipovi podataka 15
4.3.2 Funkcije 20
4.3.3 Logika grananja 22
4.4 Razvojna okolina za funkcionalno programiranje s programskim jezikom F# 24
5. PRIJEDLOG MODELA OKOLINE ZA ANALIZU I PRIKAZ PODATAKA 26
5.1 Model analize međusobnog utjecaja klimatskih i ekonomskih pokazatelja 26
5.2 Vremenski i prostorni rasponi i obilježja podataka 26
5.3 Dohvaćanje i priprema podataka 26
5.4 Statistički postupci korišteni za analizu podataka 27
5.5 Prikaz i interpretiranje dobivenih rezultata 28
5.6 Prikaz funkcionalnosti programskog rješenja pomoću dijagrama 28
6. PROGRAMSKO RJEŠENJE ZA ANALIZU MEĐUSOBNOG UTJECAJA KLIMATSKIH I EKONOMSKIH POKAZATELJA 30
6.1 Postavljanje razvojne okoline za funkcijsko programiranje s programskim jezikom F# 30

6.2 Opis raspoloživih biblioteka 30
6.3 Programsko ostvarenje za dohvaćanje i pripremu podataka 30
6.4 Programsko ostvarenje analize podataka 32
6.5 Programsko ostvarenje za prikaz i interpretiranje rezultata analize 34
6.6 Ispitivanje programskog rješenja na primjerima s analizom 35
6.6.1. Rezultati za regiju Istočna Europa i Pacifik 36
6.6.2. Rezultati za regiju Europa i Središnja Azija 37
6.6.3. Rezultati za regiju Istočna Azija 39
7. ZAKLJUČAK 42
LITERATURA 43
SAŽETAK 45
ABSTRACT 45
ŽIVOTOPIS 46
PRILOZI (na CD-u) 47

1. UVOD

Funkcionalno programiranje je programska paradigma u kojoj su funkcije glavni elementi za izgradnju programa. Postoji mnogo funkcionalnih programskih jezika, a jedan od njih je F#. To je prvenstveno funkcionalni jezik opće namjene koji osim funkcionalnog podržava i objektno-orijentirani način programiranja. Funkcionalno programiranje i jezik F# imaju mnoge primjene, a jedna od njih je analiza podataka. Analiza podataka je danas često korištena u raznim područjima primjene za dobivanje dodatnih informacija iz podataka koje će pridonijeti donošenju zaključaka, odluka ili stvaranju nekih spoznaja. Trenutno aktualna tema u svijetu je ispuštanje ugljičnog dioksida i njegov utjecaj na Zemlju. U ovom radu analiza podataka koristit će se kako bi se vidjelo jesu li BDP (bruto društveni proizvod) i korištenje energije dobivene iz obnovljivih izvora u korelaciji s ispuštanjem ugljikovog dioksida. To će se postići koristeći dva izraza za izračun korelacije. To su Spearmanov i Pearsonov izraz za izračun koeficijenta korelacije. Korištenje navedena dva izraza dat će uvid u to ima li razlika između koeficijenata dobivenih tim izrazima ili će biti konzistentni.

U drugom poglavlju rada riječ je o trenutnom stanju područja funkcionalnog programiranja i F# jezika te o njihovoj primjeni u analizi podatka. U trećem poglavlju dana su osnovna načela funkcionalnog programiranja, kao i njegova povijest te neki jezici za funkcionalno programiranje. U četvrtom poglavlju načinjen je pregled programskog jezika F#, njegova povijest, svojstva, sintaksa i razvojna okolina. U petom poglavlju opisan je model okoline koji se koristi za ovaj primjer analize podataka, dok je šestom poglavlju prikazano i analizirano programsko rješenje.

1.1 Zadatak završnog rada

U završnom radu treba proučiti i opisati načela funkcionalnog programiranja, kao i funkcionalne, imperativne i objektno-orijentirane mogućnosti funkcionalnog programskog jezika F#. Nadalje, treba opisati svojstva i ograničenja jezika u odnosu na druge jezike, te razvojne okoline i alate. Navedena načela i svojstva treba pokazati koristeći prikladne primjere. Također, u radu je potrebno predložiti okolinu za analizu i prikaz podataka koja omogućuje dohvaćanje, pripremu, analizu podataka prikladnim postupcima strojnog učenja, te njihov prikaz i opis, a podržava načela funkcionalnog, odnosno programski jezik F#. U praktičnom dijelu rada treba osmisliti, modelirati i programski ostvariti opisanu okolinu prikladnu za analizu međusobnog utjecaja klimatskih i ekonomskih

pokazatelja za određeni vremenski i prostorni raspon podataka, kao i prikaz i sažeti opis dobivenih rezultata.

2. PREGLED TRENUTNOG STANJA U FUNKCIONALNOM PROGRAMIRANJU ZA ANALIZU PODATAKA

2.1 Funkcionalno programiranje

Funkcionalno programiranje može biti korišteno u raznim područjima primjene budući da postoji mnogo funkcionalnih jezika koji imaju različite namjene, a i mnogi jezici koji se baziraju na drugačijem stilu programiranja implementiraju neke značajke funkcionalnog programiranja. Funkcionalno programiranje može se koristiti za financije, umjetnu inteligenciju [1], izradu web stranica i web aplikacija [2], telekomunikacijske sustave, poslužitelje za internet aplikacije, telekomunikacijske operacije [3], kreiranje sustava za upravljanje podatcima, spajanje *backend* sustava i baza podataka [4] i drugo.

Iako se funkcionalno programiranje može koristiti u mnogim područjima, prema izvoru [5], ono nije često korišteno zbog složenosti i cijene programskog dizajna rješenja zasnovanih na takvom načinu programiranja. Prilikom korištenja čistog funkcionalnog programiranja, mala promjena može izazvati potrebu za velikim restrukturiranjem programa i time izazvati troškove. Prema nekim istraživanjima, pisanje koda potpuno u funkcionalnom stilu može povećati sigurnost budući da takav način iziskuje pisanje koda u kojem nema promjene stanja što znači da se vrijednosti varijabli ili svojstava ne mijenjaju tijekom izvođenja programa. Također, zbog značajke nepromjenjivosti u funkcionalnom programiranju ono je pogodno za raspodijeljene sustave (*eng. distributed systems*), a funkcije čistog reda čine ga pogodnim za paralelno programiranje. Iako možda nije toliko korišteno u praktičnoj primjeni, postaje sve popularnije u akademskim krugovima i sve više sveučilišta ga dodaje u svoje programe [5].

2.2 Korištenje programskog jezika F#

F# je jezik opće namjene što znači da je korišten u mnogim područjima kao što su znanost o podatcima (*eng. data science*), web programiranje, mobilne aplikacije i drugo.

Pimjer korištenja jezika F# za razvoj web aplikacija je projekt WebSharper razvijen od strane tvrtke Intellifactory koji omogućuje razvoj web aplikacija pisanjem koda u F#-u i njegovim prevođenjem u JavaScript [6]. Također se za razvoj web aplikacija može koristiti SAFE Stack. U sklopu toga dolazi biblioteka Saturn koja zajedno sa ASP.NET Core pruža pouzdan web poslužitelj, Microsoftova

platforma za poslužiteljske i platformske usluge u oblaku Azure, biblioteka Feable koja omogućuje interoperabilnost F#-a i JavaScripta i biblioteka Elmish koja omogućuje pisanje korisničkih sučelja koja su kompatibilna s funkcionalnim programiranjem [7].

Za razvoj mobilnih aplikacija korištenjem jezika F# može se koristiti Fabulous, razvojni okvir koji omogućuje razvoj aplikacija na više platformi (*eng. cross-platform*) s Xamarin.Forms [8].

Budući da postoje bilioteke za pristup podatcima, F# je pogodan i za tu svrhu. Korištenjem biblioteka Fsharp.Data i Json.NET olakšan je rad s formatima kao što su CSV, JSON, HTML i XML. Također postoje biblioteke za pristup SQL podatcima kao što su FSharp.Data.SqlClient, FSharp.Data.SQLProvider, SqlDataConnection Type Provider i druge [9].

2.3 Funkcionalno programiranje u analizi podataka

Funkcionalno programiranje ima uporabu i u analizi podataka. R je jezik i okruženje za analizu i prikaz podataka. Okruženje pruža alate za rukovanje s podatcima i njihovo skladištenje, skup operatora za izračune na poljima, alate za analizu podataka i programski jezik [10]. Korištenje R-a za analizu podataka može se vidjeti u [11] gdje je zajedno s drugim tehnologijama korišten za morsko geološko istraživanje Marijanske brazde. Pomoću R-a proučavane su nejednakosti raznih čimbenika geomorfološke strukture. Korištene su razne R biblioteke za analizu regresije, kompozicijske ljestvice, grafove za usporedbu podataka po tektonskim pločama, rangiranje točkastih grafova za analizu korelacije, mozaične grafove (*eng. mosaic plots*) i drugo.

Još jedan primjer okruženja i funkcionalnog jezika za analizu podataka je Luna. Luna pruža okruženje za obradu i prikaz podataka. To je prvi programski jezik koja sadrži dvije ravnopravne reprezentacije sintakse, tekstualnu i vizualnu, gdje promjena jednog odmah utječe na drugo [12].

F# također može biti korišten za analizu podataka. U primjeru iz izvora [13], F# je korišten za razvoj web aplikacije za usporedbu upisa na fakultet u raznim državama i regijama u svijetu. Korištena je biblioteka FunScript koja prevodi citate (*eng. quotations*) programa u JavaScript. Korištenjem biblioteke Fsharp.Data i pružatelja tipa (*eng. type provider*) za Svjetsku banku (*eng. World Bank*) preuzeti su podatci o upisu na fakultet, te su na temelju tih podataka prikazani grafovi za države i regije izabrane na korisničkom sučelju.

Još jedan primjer korištenja F#-a za analizu podataka je biblioteka Bristlecone koja omogućuje definiciju modela podataka, metodu Ugradnje modela i odabira modela (*eng. model-fitting and model-selection, MFMS*) i smanjenje grešaka u definiciji modela i podataka prije započinjanja MFMS analize. U ovoj biblioteci dani su alati koji donose zaključke o strukturi i veličini procesa iz dugoročnih ekoloških podataka [14]. Keggle također koristi F# za jezgrenu analizu podataka [15].

3. NAČELA FUNKCIONALNOG PROGRAMIRANJA

3.1 Funkcionalno programiranje

Funkcionalno progamiranje je programska paradigma čija je osnova lambda račun. Ono se može definirati na mnoge načine, a jedan od njih je da je to stil programiranja koji stavlja naglasak na evaluaciju izraza, a ne na izvedbu naredbi. Također, može se reći da je to stil programiranja u kojem su osnovni elementi za izgradnju programa funkcije. Program napisan funkcionalnim stilom programiranja definira što je rezultat, a ne naredbe čije izvođenje dovodi do tog rezultata [16].

3.1.1 Glavne značajke

Što su točno značajke funkcionalnog programiranja, teško je definirati. Značajke mogu varirati od jezika do jezika. Ovdje će biti nabrojane neke njih:

- Nepromjenjivost - mogućnost kreiranja varijabli čije se vrijednosti ne mogu mijenjati. Ono odvaja podatke od funkcionalnosti.
- Izrazi - pojam prema kojem svaka operacija ima opipljiv rezultat o kojem se može donositi zaključke [17].
- Prvorazredne (*eng. first-class*) funkcije i funkcije višeg reda - omogućuju spremanje funkcija u strukture podataka, njihovu predaju kao ulaznih argumenata drugih funkcija i vraćanje funkcije kao rezultata. Jednostavnije se može reći da se funkciju može tretirati kao podatak.
- Čiste funkcije (*eng. pure functions*) - funkcije koje nemaju nuspojave. Rad s ovakvim funkcijama znači da je moguće ukloniti funkciju ako ni jedna druga funkcija nije ovisna o njezinom rezultatu, dobiti iste rezultate svaki put kada je se pozove za iste ulazne podatke, promijeniti raspored pozivanja različitih funkcija bez promjene funkcionalnosti i paralelno obrađivati pozive funkcija.
- Rekurzivne funkcije - funkcije koje pozivaju same sebe i spremaju vrijednost svakog poziva na stog. Mora postojati uvjet kada će se pozivanje prekinuti.
- Lijena evaluacija (*eng. lazy evaluation*) – odgađa evaluaciju izraza dok god njegova vrijednost nije potrebna [18].

Još neke značajke funkcionalnog programiranja bit će objašnjene u sljedećem poglavlju na primjeru jezika F#.

3.1.2 Lambda račun

Lambda račun još se naziva i najmanjim univerzalnim programskim jezikom na svijetu. Predstavio ga je Alonzo Church u 1930-tima kao formalizirani način na koji bi se mogle izražavati funkcije. Bilo koja izračunljiva funkcija može se izraziti pomoću lambda računa. To je pristup koji simbolično prikazuje pravila transformacije, ali ne prikazuje samu unutarnju implementaciju funkcije [19].

Ovdje će biti prikazan osnovni princip korištenja lambda izračuna za prikaz funkcije. Za primjer će biti uzeta funkcija koja prima jedan argument a i vraća istu tu vrijednost. Grčko slovo λ koristi se za označavanje argumenata funkcije. Nakon argumenata piše se „tijelo“ funkcije koje govori kako će se ulazni podatak ili podatci obrađivati i koje se od argumenata odvaja točkom. Dakle, prethodno navedena funkcija zapisala bi se kao: λa.a. Kada bi htjeli primjeniti tu funkciju na varijablu x, to bi zapisali kao: (λa.a)x. Zapis [x/a] koristi se kako bi se pokazalo da je a zamijenjen s varijablom x unutar funkcije. Tako se prema [19] može zapisati: (λa.a)x→[x/a]a→x.

3.1.3 Povijest razvoja funkcionalnog programiranja

Veliki utjecaj na razvoj funkcionalnog programiranja imao je rad Alonza Churcha na lambda računu u 1930-ima. Lambda račun nekada se opisuje i kao prvi funkcionalni jezik [20]. U kasnim 1950-ima je došao programski jezik LISP koji je napravio John McCarthy. LISP je sadržavao neke elemente funkcionalnog programiranja. Zbog toga ga se smatralo funkcionalnim programskim jezikom, no LISP procedura može vratiti različite vrijednosti za svaki poziv s istim ulaznim argumentima što krši načelo čiste funkcije [21]. 1960-ih došao je jezik Iswim kreatora Petera Landina. Ovaj jezik donio je inovacije u tada poznato funkcionalno programiranje. U njemu su prvi put predstavljeni `let` i `where` izrazi, korištenje uvlačenja umjesto separatora kao što su zarez, točka-zarez ili vitičaste zagrade i drugo. Jezik APL koji je također izašao 1960-ih kreatora Iversona nije bio u potpunosti funkcionalan jezik, ali se i dalje može spominjati u kontekstu razvoja funkcionalnog programiranja. APL je nastao iz želje za stvaranjem programskog jezika za rad sa poljima i originalno je sadržavao funkcionalnu notaciju. APL je koristio posebnu abecedu u kojoj je svako slovo odgovaralo jednom operatoru. Iz rada na APL-u proizašao je jezik FAC, funkcionalni programski jezik koji je sadržavao većinu sintakse i programskih načela APL-a [20]. Godine 1977. napravljen je jezik FP od strane Johna Backusa koji ga je stvorio kao alternativu tadašnjim programima kod većine kojih se stanje programa mijenjalo tijekom njegovog izvođenja. John Backus smatrao je da je takav princip stvarao složene sustave koji imaju veću cijenu održavanja [21]. U 1970-ima je razvijen i ML koji također nije bio u

potpunosti funkcionalan jezik, ali je poticao funkcionalni stil programiranja. Godine 1985. David Turnes napravio je funkcijski jezik Miranda koji je korišten na operacijskim sustavima UNIX [21]. Miranda je omogućavala definiranje korisničkih tipova podataka. U 1990-ima došao je Haskell, potpuno funkcionalni jezik opće namjene. Haskell sadrži nmnoga načela funkcionalnog programiranja kao što su lijena evaluacija (*eng. lazy evaluation*), korisnički tipovi podataka i podudaranje uzoraka (*eng. pattern matching*) [20].

3.2 Programski jezici za funkcionalno programiranje

Osim funkcionalnih programskih jezika, postoje i jezici koji uz neke druge stilove programiranja podržavaju i funkcionalno. Oni podržavaju neke značajke funkcionalnog programiranja, no ne sve. Neki od takvih jezika su Python, Scala, C++ i Kotlin. Neki funkcionalni programski jezici su Lisp, OCaml, Haskell, Elm, Elrang, Clojure i F# . Ti jezici, iako su funkcionalni, ne podržavaju svi iste principe funkcionalnog programiranja. Također, neki jezici mogu biti prvenstveno funkcionalni, ali podržavati i druge stilove programiranja.

Haskell je u potpunosti funkcionalan programski jezik koji se koristi u mnogim područjima kao što su financije, umjetna inteligencija i drugo [2]. On se često uzima za dobar primjer funkcionalnog jezika, jer implementira mnoga funkcionalna načela [18]. Elm je funkcionalni programski jezik koji se koristi za izradu web stranica i web aplikacija [3]. Erlang je jezik napravljen od strane tvrtke Ericsson koji se koristi za telekomunikacijske sustave, poslužitelje za internet aplikacije, telekomunikacijske operacije i aplikacije baza podataka [4]. Clojure je također funkcijski programski jezik. Koriste ga mnoge tvrtke u razne svrhe kao što su kreiranje sustava za upravljanje podatcima, spajanje *backend* sustava i baza podataka, kreiranje platformi za financijske usluge i mnoge druge [5]. OCaml koriste velike korporacije kao što su Dassault Systèmes, Microsoft i IBM za razvoj industrijskih projekata [22].

3.3 Programski jezik F# - svojstva i ograničenja, mogućnosti

F# je prvenstveno funkcionalni programski jezik opće namjene koji je dio .NET platforme. Kaže se da je prvenstveno funkcionalan, jer potiče na korištenje funkcionalnog načina programiranja no podržava i druge programske paradigme kao što je objektno-orijentirano programiranje. Implementira mnoga načela funkcionalnog programiranja, ali nudi i načine da se izbjegne njihovo korištenje. Radi na Linuxu, Mac OS X-u, Androidu, iOS-u, Windowsima i internetskim preglednicima. Neki od

elemenata funkcionalnog programiranja koje implementira su nepromjenjivost (*eng. immutability*), prvorazredne (*eng. first-class*) funkcije, funkcije višeg reda, rekurzija i drugi.

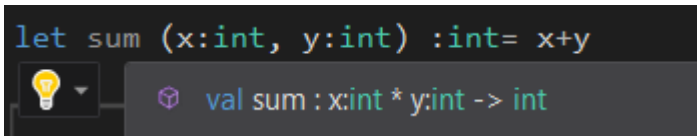
4. PROGRAMSKI JEZIK F#

4.1 O jeziku F#

F# je prvotno funkcionalni jezik opće namjene. Osim funkcionalne, podržava i objektno-orijentiranu paradigmu. Prema opisu F#-a sa F# Software Foundation stranice, on je zreli, prvenstveno funkcionalni, višeplatformski jezik otvorenog koda [9]. Možemo reći da je zreli jezik budući da ima razvijenu zajednicu koja omogućuje lako pronalaženje resursa i pomoći. Višeplatformski je jezik budući da radi na više operacijskih sutava kao što su Linux, iOS, Windows i drugi [9]. Prvenstveno funkcionalni znači da potiče na funkcionalno programiranje no dopušta korištenje i drugih programskih paradigmi. Razlog tome je potreba da bude kompatibilan s ostalim jezicima na .NET platfomi.

Neka svojsta programskog jezika F# su:

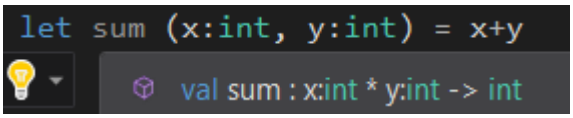
- Nepromjenjivost (*eng. immutability*) - u F#-u vrijednosti svih varijabli su originalno nepromjenjive. To nekada može otežati pisanje koda, no može se zaobići korištenjem ključne riječi `mutable`.
- Lagana sintaksa (*eng. lightweight syntax*) - F# ima kraću sintaksu koja koristi uvlačenja za odvajanje dijelova programa umjesto ključnih riječi.
- Zaključivanje o tipovima (*eng. type inference*) - u F# nije nužno pisati tipove varijabli već ih prevoditelj sam odredi prilikom prevođenja. Kako bi se prikazalo zaključivanje o tipovima, koristit će se, prema slici 4.1 primjer jednostavne funkcije `sum` koja prima dva argumenta `x` i `y` te vraća njihov zbroj.



Sl. 4.1. *Funkcija `sum` sa eksplicitno određenim tipovima podataka.*

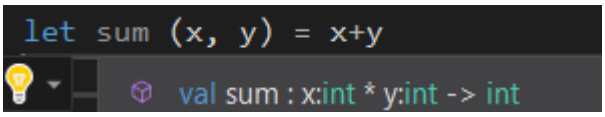
Na slici 4.1 vidi se primjer funkcije `sum` s eksplicitno određenim tipovima podataka. Oba argumenta su tipa `int` pa je i povratna vrijednost tog tipa.

Na slici 4.2 dan je primjer funkcije `sum` u kojem tip povratne vrijednosti nije eksplicitno zadan. Usprkos tome, tip povratne vrijednosti je i dalje `int`, a razlog tome je što F# ne dopušta implicitne konverzije tipova podataka tako da, budući da su `x` i `y` tipa `int` i rezultat će biti istog tog tipa. *Type inference* ne radi dobro s tipovima iz .NET BCL („Base Case Library“), te sa klasama i metodama.



Sl. 4.2. *Funkcija `sum` bez eksplicitno određenog tipa povratne vrijednosti.*

Na kraju se na slici 4.3 može vidjeti primjer funkcije `sum` u kojem niti jednoj vrijednosti nije eksplicitno definiran tip, no oba argumenta i povratna vrijednost su i dalje tipa `int`. Razlog tome je što se operator + automatski veže uz tip `int`. Kada bi htjeli da tip povratne vrijednosti u ovakvoj funkciji bude na primjer `float` morali bi jednom od argumenata ili povratnoj vrijednosti eksplicitno zadati taj tip.



Sl. 4.3. *Funkcija `sum` bez eksplicitno određenih tipova.*

- *First-class* funkcije - u F# se s funkcijama može raditi sve što i s ugrađenim tipovima podataka. Može ih se vezati uz identifikator pomoću ključne riječi `let` koja bilo koju vrijednost veže uz simbol, tj. identifikator. Funkcije se također može spremati u strukture podataka, davati drugim funkcijama kao argumente i vraćati kao povratne vrijednosti funkcija. Prethodno dvije nabrojane stavke svojstva su funkcija višeg reda. Te funkcije primaju druge funkcije kao argumente i njihov poziv vraća funkciju kao rezultat.
- Usklađivanje uzoraka (*eng. pattern matching*) - logika grananja koja omogućuje usporedbu podataka s logičkim strukturama, rastavljanje podataka na dijelove i izvlačenje informacija iz podataka. Za najsličniji primjer ovome mogao bi se uzeti izraz `switch/case`.

- Asinkroni način rada - F# podržava asinkroni način rada. On omogućuje obavljanje zadataka u pozadini i primanja obavijesti kada taj rad u pozadini završi. To omogućuje obavljanje više zadataka u isto vrijeme.
- Svaka funkcija u F#-u mora imati povratnu vrijednost - funkcija ne može vratiti vrijednost `void` kao u nekim drugim jezicima. Usprkos tome F# ima nešto što može zamijeniti `void` a to je `unit`. Tip `unit` se nalazi na bilo kojem mjestu na kojem bi funkcija u nekom drugom jeziku vratila `void`. `Unit` se u F#-u gleda kao normalan objekt i može biti vraćen iz bilo kojeg dijela koda. Tako svaka funkcija nešto vraća iako povratna vrijednost nije eksplicitno zadana, a to nešto je `unit` objekt. Tako se može reći i da svaka funkcija uvijek prima barem jednu ulaznu vrijednost, jer ako ulazne vrijednosti nisu eksplicitno zadane prima tip `unit` [17].
- Djelomična primjena funkcija - pozivanje *curried* funkcije i dobivanje nove funkcije nazad. *Carried* funkcije su funkcije koje omogućuju davanje samo nekih ulaznih elemenata prilikom poziva i kao rezultat vraćaju novu funkciju koja očekuje preostale ulazne argumente [17].

Jedno od ograničenja F#-a je da trenutno ne postoji službena potpora za WPF ili *Windows Forms* kao što je slučaj kod recimo C#-a. Za njihovo korištenje moraju se koristiti biblioteke. F# također ne podržava implicitne konverzije tipova podataka. Prilikom pisanja koda mora se paziti kako ne bi došlo do grešaka korištenjem različitih tipova. Jedna od pozitivnih stvari kod F# je što je kompatibilan sa C#-om. Tipovi podataka i metode iz C#-a mogu se koristiti u F# kodu i obratno.

Iako je prvenstveno funkcionalan jezik, F# se razlikuje od čistih funkcionalnih jezika kao što je na primjer Haskell. Prvenstveni je razlog što F# podržava objektno-orijentirano programiranje. Također, iako su vrijednosti varijabli izvorno nepromjenjive, F# dopušta da se to zaobiđe koristeći ključnu riječ `mutable`.

4.2 Povijest jezika F#

F# potječe iz Microsoftove podružnice za istraživanje Microsoft Research. Nastao je kao ideja Dona Symea. On je htio napraviti funkcionalni programski jezik koji bi radio na platformi .NET. Prvo je pokušao napraviti implementaciju Haskella za .NET platformu, no tu ideju je kasnije odbacio, jer bi ona zahtijevala velike promjene Haskella. Kasnije je odlučio napraviti inačicu jezika OCaml za .NET izvorno nazvanu „Caml.Net“ koja je kasnije preimenovana u F# [23]. U 2001. započela je izrada F#-

a s implementacijom osnovne sintakse OCamla kao i prevoditelja koji je inicijalno pisan u OCamlu. F# ne implementira sve značajke OCamla, neke su izbačene, kao na primjer funktori. Budući da je F# napravljen za .NET, sve u F# je moralo biti kompatibilno s .NET-om i obrnuto. Tako su tipovi u F#-u morali biti tipovi na .NET-u i funkcije i tipovi definirani u F#-u su morali moći biti korišteni od strane drugih .NET jezika.

F# 1.0 izašao je 2005. Neke od značajki koje je sadržavao su:

- Protočne strukture (*eng. pipelines*).
- Podržavanje objektno-orijentiranih značajki.
- Aktivni uzorci (eng. active patterns) i druge.

Godine 2007. F# je počeo dobivati na popularnosti i neke veće tvrtke su ga počele koristiti u svojim sustavima [23]. Iste godine izašao je F# 2.0 u kojem su dodane mjerne jedinice i pružatelji tipa (*eng. type providers*). Također je 2010. izašao Visual F# 2.0 kao dio Visual Studia 10.

Pružatelji tipa bili su veliki dio F# 3.0, te su nakon njegovog izlaska postali veliki dio F# ekosistema. Važnost pružatelja tipa može se vidjeti i u tome da je kasnije kreirana biblioteka FSharp.Data koja je sadržavala često korištene pružatelje tipova za HTML,CSV i XML [23].

F# nije od početka bio *open source*, a to je postao tek 2010. godine, dok je 2014. godine Microsoft počeo prihvaćati vanjske doprinose F#-u.

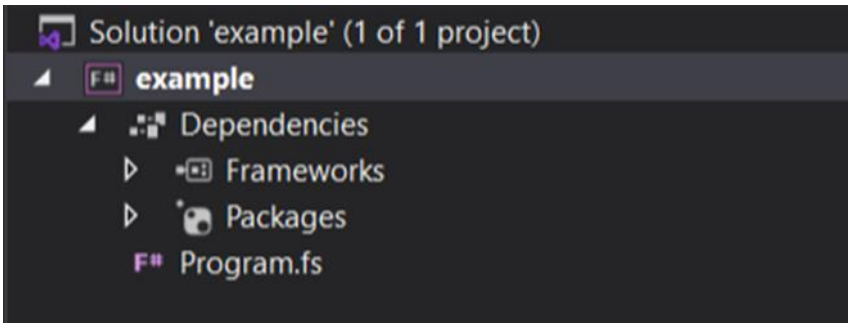
Već se F# 1.0 mogao izvoditi na Linuxu koristeći Mono. Godine 2016. Microsoft je izdao višeplatformsku podršku za .NET pod nazivom .NET Core. Od 2017. podrška za F# je uključena u .NET Core SDK i standardne Linux pakete [23]. Danas pružatelji usluga oblaka kao što su Google i Amazon podržavaju F# preko .NET Core.

Tijekom godina su nastale mnoge F# zajednice, ali važnu ulogu je imalo osnovanje zajednice pod imenom F# Software Fundation (FSSF) poznatije kao fsharp.org. Osnovanje FSSF-a bilo je značajno, jer je do tada Microsoft bio smatran odgovornim za rješavanje problema, pružanje resursa i komunikaciju o tehnologijama vezanim za F#. Kasnije je izašlo još inačica F#-a od kojih je svaka uvela nešto novo. Trenutno najnovija inačica je F#.4.7.

4.3 Osnove sintakse i građe programskog rješenja

U ovom poglavlju bit će dani primjeri za neke dijelove sintakse F#-a i prikazan primjer programskog rješenja.

Slika 4.4 prikazuje građu programskog rješenja za najjednostavniji primjer programa u F#-u.



Sl. 4.4. *Građa programskog rješenja.*

Rješenje sadrži ovisnosti (*eng. dependencies*) pod kojima se nalaze instalirani paketi i razvojni okviri (*eng. frameworks*), a u ovom slučaju to je Microsoft.NET.Core.App razvojni okvir. Programsko rješenje također sadrži *Program.fs* koji je ovdje početna točka programa. U slučaju kada rješenje ima više datoteka bitan je red kojim su one navedene u rješenju. Datoteka može pristupiti samo funkcijama i tipovima iz datoteka navedenih iznad nje.

Za uvod u sintaksu F#-a bit će dan primjer najjednostavnijeg „Hello World!“ programa koji prikazuje slika 4.5.

```
open System

[<EntryPoint>]
let main argv =

    printfn "Hello World!"


    0
```

Sl 4.5. *Primjer „Hello World!“ progarma u F#-u.*

U ovom primjeru vide se neke osnove sintakse F#-a. Atribut `[<EntryPoint>]` označava da je `main` funkcija koju treba pozvati prilikom pokretanja aplikacije. U ovom primjeru se također može vidjeti da nema ključne riječi `return`. Razlog tome je što ona u F#-u nije potrebna već se smatra da je zadnji izraz u području djelovanja *(eng. scope*) funkcije rezultat te funkcije. Funkcija `main` prima jedan argument `argv` koji je polje stringova. Funkcija `printfn` koristi se za ispis na konzolu. Ključna riječ `let` koristi se kako bi se vrijednosti vezale za njihove identifikatore. Budući da se u F#-u funkcije gledaju kao vrijednosti, korištena je i kod funkcije `main`. U ovom primjeru također je vidljivo da nema točka-zareza (;) ili vitičastih zagrada ({}). Oni u F#-u nisu potrebni. Točka-zarez se može koristiti, no odvajanje u novi red je dovoljno da se označi da je nešto novi izraz. Umjesto vitičastih zagrada, za označavanje područja djelovanja u F#-u se ono označava uvlačenjem koda koji mu pripada.

Slika 4.6 prikazuje primjer označavanja komentara u F#-u. Ako se želi označiti blok komentara tada se komentar piše između oznaka `(*  *)`, a za označavanje samo jednog reda koristi se `//`.

```
(*This is
 a comment.*)

//This is a comment.
```

Sl. 4.6. *Označavanje komentara u F#-u.*

4.3.1 Tipovi podataka

Kao što je prikazano u tablici 4.1, F# ima mnogo tipova podataka, a ovdje će biti nabrojani samo neki od njih. Osim primitivnih tipova podataka F# sadrži zbirke kao što su lista, polje (*eng. array*) i niz (*eng. sequence*).

U F# je lista poredani niz elemenata istog tipa čije su vrijednosti nepromjenjive. Slika 4.7 prikazuje neke načine kreiranja liste. Listu se može definirati tako da se eksplicitno navedu njeni elementi kao što je slučaj kod `list1` i `list2`, u slučaju liste `list1` elementi su odvojeni koristeći točku-zarez, a u slučaju liste `list2` odvojeni su prelaskom u novi red. Ove dvije liste su iste.

Tab. 4.1. *Neki od osnovnih tipova podataka u F#*

TIP	OPIS
`int`	Cjeli broj,vrijednosti od -2,147,483,648 do 2,147,483,647
`bool`	Vrijednosti `true` ili `false`
`char`	Znak, vrijednost od 0 do 255
`float,double`	64-bitna vrijednost sa pomičnim zarezom
`byte`	vrijednost od 0 do 255

```
let list1=[1;2;3;4;5]
let list2=
    [   1
        2
        3
        4
        5
    ]
let list3=[1..5]
let list4=[for x in 1..5-> x*x]
```

Sl. 4.7. *Kreiranje liste u F#-u.*

Lista `list3` definirana je korištenjem operatora raspona (`..`), elementi te liste su brojevi od 1 do 5. Lista `list4` definirana je koristeći obuhvaćanje (*eng. comprehensions*) koje se zasniva na sintaksi `for` petlje, elementi te liste su kvadrati brojeva od 1 do 5. Neka od svojstava liste su `Item` koje omogućuje pristup elementu liste, `IsEmpty` koje vraća bool vrijednost `true` ako u listi nema elemenata, `Head` koje daje pristup prvom elementu liste i `Lenght` koje vraća broj elemenata liste. Početni indeks u listi je nula.

Neke od funkcija za rad s listama su `List.sort`, `List.sortBy` i `List.sortWith` za sortiranje listi, `List.find`, `List.tryFind` i `List.tryFindIndex` za pronalaženje elementa ili indeksa elementa koji odgovara uvjetu pretrage. `List.zip` uzima dvije liste i kao rezultat vraća jednu listu čiji su elementi parovi podataka u kojima su podatci odvojeni zarezom (*eng. tuple*), a `List.unzip`

takvu listu rastavlja na izvorne dvije liste. `List.iter` omogućuje pozivanje neke funkcije na svakom elementu liste. `List.filter` kao rezultat daje novu listu koja sadrži samo elemente koji odgovaraju zadanom uvjetu. Neke od funkcija koje omogućuju obavljanje aritmetičkih operacija na listama su `List.sum`, `List.average` i `List.sumBy`. `List.map` funkcija kreira novu listu čiji su elementi rezultat primjenjivanja neke funkcije na svaki element liste.

Polja su zbirke elemenata istog tipa čije su vrijednosti promjenjive. Slika 4.8 prikazuje neke načine kreiranja polja u F#-u.

```
let array1=[|1;2;3;4;5|]
let array2=
    [|  1
        2
        3
        4
        5
    |]
let array3=[|1..5|]
let array4=[|for x in 1..5-> x*x|]
```

Sl. 4.8. *Kreiranje polja u F#-u.*

Razlika između kreiranja polja i liste u F#-u je u tome što se kod polja elementi stavljaju unutar `[| |]`, a kod liste unutar `[ ]`. Početni index u polju je nula baš kao i kod liste. Pristup elementu polja moguć je preko operatora točka (`.`) i uglatih zagrada `[ ],` na pimjer `array.[0]`. Za polja postoje sve funkcije koje su prethodno nabrojane i za liste. Također, postoji funkcija `Array.set` koja postavlja element na zadanu vrijednost. U F#-u su podržana i 2D, 3D i 4D polja.

Nizovi su zbirke elemenata istog tipa. Dobri su za velike zbirke podataka kod kojih se neće nužno koristiti svi elementi. Razlog tome je što se svaki element niza obrađuje onda kada je potreban i zbog toga mogu dati bolje performanse nego korištenje liste [24]. Slika 4.9 prikazuje neke načine kreiranja nizova.

```
let seq1=seq{1;2;3;4;5}
let seq2=seq{0..2..10}
let seq3=seq{for x in 1..5->x*x}
let seq4=seq{for x in 1..5 do yield x*x}
let seq5=seq{for x in 0..10 do if x%2=0 then x }
```

Sl. 4.9. *Kreiranje nizova u F#-u.*

Nizovi se kreiraju pomoću slijednih izraza koji kao rezultat daju niz. U primjeru je `seq1` kreiran tako da su eksplicitno zadani svi elementi niza. Niz `seq2` kreiran je tako da sadrži elemente koji su između nula i deset, ali tako da je inkrement između elemenata dva umjesto jedan. Nizovi `seq3` i `seq4` sadrže iste elemente i oba su kreirana koristeći `for` izraz, razlika je u tome što `seq3` koristi `->` operator koji označava koja će vrijednost postati dio niza, a `seq4` ključnu riječ `do` zajedno sa ključnom riječi `yield`. Ključna riječ `yield` nije obavezna, ona ima sličnu funkcionalnost kao `return`, tj. vraća neku vrijednost. Za rad s nizovima postoje iste funkcije kao i za liste i polja.

Još jedan tip podatka u F#-u je *tuple*. Ovaj tip podatka omogućuje grupiranje podataka različitih tipova. *Tuple* je dobar način vraćanja više vrijednosti iz funkcije. Slika 4.10 prikazuje primjere *tuple* podataka.

```
let tuple1=("a","b","c")
let tuple2=(1,2)
let tuple3=("a",1,"b")
```

Sl. 4.10. *Primjer tuple tipa podataka.*

Tuple `tuple1` sadrži tri podatka tipa `string` i njegov tip je `string*string*string`, `tuple2` sadrži dva podatka tipa `int` i njegov tip je `int*int` i `tuple3` sadrži dva podatka tipa `string` i jedan tipa `int` i njegov tip je `string*int*string`.

Još jedan tip podatka u F#-u je slog (*eng. record*). Slogovi omogućuju spremanje različitih tipova podataka koji imaju dodijeljena imena. Slika 4.11 prikazuje primjer deklaracije sloga u F#-u.

```
type Country={
    Name:string
    CapitalCity:string
    Population:int

}
```

Sl. 4.11. *Primjer deklaracije sloga u F#-u.*

Slog `Country` sadrži tri člana: ime (`Name`) tipa `string`, glavni grad (`CapitalCity`) tipa `string` i populaciju (`Population`) tipa `int`. Slika 4.12 prikazuje primjer kreiranja instance sloga `Country`.

```
let country={
    Name="United States"
    CapitalCity="Washington, D.C."
    Population=382200000
}
```

Sl. 4.12. *Primjer kreiranja instance sloga* `Country`.

Prilikom kreiranja instance sloga svakom članu se mora dodijeliti vrijednost. Također, ne može se koristiti vrijednost jednog člana kako bi se postavila vrijednost drugog. Pojedinom članu sloga može se pristupiti pomoću točka operatora (`.`). Ako bismo htjeli pristupiti imenu države to možemo učiniti kao `country.Name`.

Budući da F# podržava objektno-orijentirano programiranje, jedan od tipova podataka u F#-u su klase. Slika 4.13 prikazuje primjer deklaracije klase u F#-u.

```
type Country=class
    val Name:string
    val CapitalCity:string
    val Population:int

    new(name,capitalCity,population)={Name=name;CapitalCity=capitalCity;Population=population}

    member x.PrintCountryData=
        printfn "%s has population of %d and it's capital city is %s." x.Name x.Population x.CapitalCity

    end
```

Sl. 4.13. *Primjer deklaracije klase u F#-u.*

Prilikom deklaracije klase, njeni članovi, koji bi se u jeziku kao što je C# nazivali atributi, ispred imaju ključnu riječ `val`. Konstruktor se kreira pomoću ključne riječi `new` i zatim se u zagradi navode argumenti. U tijelu funkcije se ulazni argumenti daju kao vrijednosti članovima klase. Metode se kod F# klase definiraju pomoću ključne riječi `member`. Prije imena metode mora se koristiti samoidentifikator za tu klasu. To može biti `this.` ili u ovom slučaju `x.`.

4.3.2 Funkcije

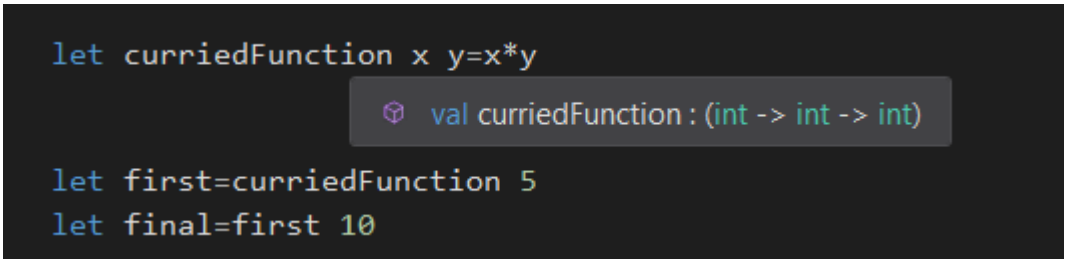
U F#-u razlikuje se *tupled* funkcije i *curried* funkcije. *Tupled* funkcije su funkcije kojima se svi argumenti moraju predati odjednom. Potpis takvih funkcija je: `(tip1*tip2...*tipN)->rezultat`. Iz ovoga je vidljivo da ulazni argumenti imaju isti potpis kao i *tuple* tip podatka. O *curried* funkcijama je već bilo riječi ranije, no to su funkcije koje dozvoljavaju predaju samo nekih argumenata i kao rezultat vraćaju funkciju koja kao argumente prima preostale koji prethodno nisu predani. Potpis takvih funkcija je: `argument1->...->argumentN->rezultat`.

Slika 4.14 prikazuje funkciju *tupled* i njezin poziv.

```
let tupledFunction (x,y)=x*y
                    val tupledFunction : (int * int -> int)
let result=tupledFunction(5,10)
```

Sl. 4.14. *Tupled funkcija i njezin poziv.*

Slika 4.15 prikazuje funkciju *curried* i njezin poziv.



Sl. 4.15. *Curried funkcija i njezin poziv.*

Način na koji je ova funkcija pozvana naziva se djelomična primjena funkcija i to je zapravo samo pozivanje funkcije uz predaju samo dijela argumenata. Ova funkcija je također mogla biti pozvana tako da joj se odmah predaju oba argumenta na način: `curriedFunction 5 10`.

Protočne strukture (*eng. pipelines*) koriste se kada je potrebno pozvati više funkcija gdje je izlaz jedne funkcije ulaz u drugu. Operator *foward pipe* omogućuje da se više funkcija slijedno izvede. Njegova sintaksa je: `argument |> funkcija`. On uzima vrijednost s lijeve strane i prenese je na desnu stranu kao zadnji argument funkcije. Slika 4.16 prikazuje primjer korištenja *pipeline* operatora.

```
let zipped=List.zip list1 list2
           |>List.map(fun (x,y)->x+y)
```

Sl. 4.16. *Primjer korištenja pipeline operatora.*

U ovom primjeru prikazan je kod u kojem se spajaju dvije liste tako da se njihovi elementi zbroje. Prvo, kako bi se liste spojile potrebno je koristiti funkciju `List.zip`, a zatim korištenjem `List.map` zbrojiti elemente listi. Tu operator *pipeline* uvelike olakšava, jer se može slijedno pozvati te funkcije. Prvo će se kreirati nova lista korištenjem funkcije `List.zip`, a zatim će se ta lista dati kao argument funkciji `List.map`. Lista nastala pozivom funkcije `List.map` pridružit će se identifikatoru `zipped`.

U F#-u funkcije mogu biti višeg reda. To su funkcije koje primaju druge funkcije kao argumente ili vraćaju funkciju kao rezultat. Slika 4.17 prikazuje primjer funkcije višeg reda.

```
let zip operation =List.zip list1 list2
                        |>List.map(fun (x,y)->operation(x,y))

let add(x,y)=x+y
let result=zip add
```

Sl 4.17. *Primjer funkcije višeg reda u F#-u.*

U ovom primjeru iskorištena je funkcija iz prethodnog primjera, no umjesto da funkcija zbraja elemente listi ona prima funkciju `operation` koja izvršava željenu operaciju nad elementima. U ovom primjeru to je zbrajanje, jer joj se predaje funkcija `add`, no isto tako joj se mogla predati funkcija koja množi, oduzima ili dijeli elemente ili pak funkcija koja implementira neku drugu operaciju.

U F#-u se rekurzivne funkcije definiraju koristeći ključnu riječ `rec`. Primjer rekurzivne funkcije u F#-u dan je slikom 4.18.

```
let rec fact x=
    if x<1 then
        1
    else x*fact(x-1)
```

Sl. 4.18. *Primjer rekurzivne funkcije u F#-u.*

Ova funkcija računa faktorijel zadanog broja. `If x<1` je uvjet pod kojim se prekida daljnji poziv funkcije. U izrazu `else x*fact(x-1)` funkcija poziva samu sebe i to se odvija dok god je `x` veći od jedan.

4.3.3 Logika grananja

Izraz `If/else` u F#-u sličan je kao u mnogim drugim jezicima. Slika 4.19 prikazuje primjer `if/else` izraza. U ovom primjeru se na temelju vrijednosti varijable `day` određuje naziv dana u tjednu.

```
let dayOfTheWeek=
    if day=1 then "Monday"
    elif day=2 then "Tuesday"
    elif day=3 then "Wednesday"
    elif day=4 then "Thursday"
    elif day=5 then "Friday"
    elif day=6 then "Saturday"
    elif day=7 then "Sunday"
    else "Number of the day has to be between 1 and 7"
```

Sl. 4.19. *Primjer if/else izraza u F#-u.*

Kao alternativu `if/else` izrazu F# nudi *pattern matching* koji je već prethodno spomenut. Kod ovog mehanizma usporedbe se uvijek rade s istim izvorom. Osim same logike grananja, omogućuje rastavljanje slogova ili tipova `tuple`. Slika 4.20 prikazuje primjer korištenja ovog mehanizma u F#-u.

```
let dayOfTheWeek=
    match day with
    |1-> "Monday"
    |2-> "Tuesday"
    |3-> "Wednesday"
    |4-> "Thursday"
    |5-> "Friday"
    |6-> "Saturday"
    |7-> "Sunday"
    |_-> "Number of the day has to be between 1 and 7"
```

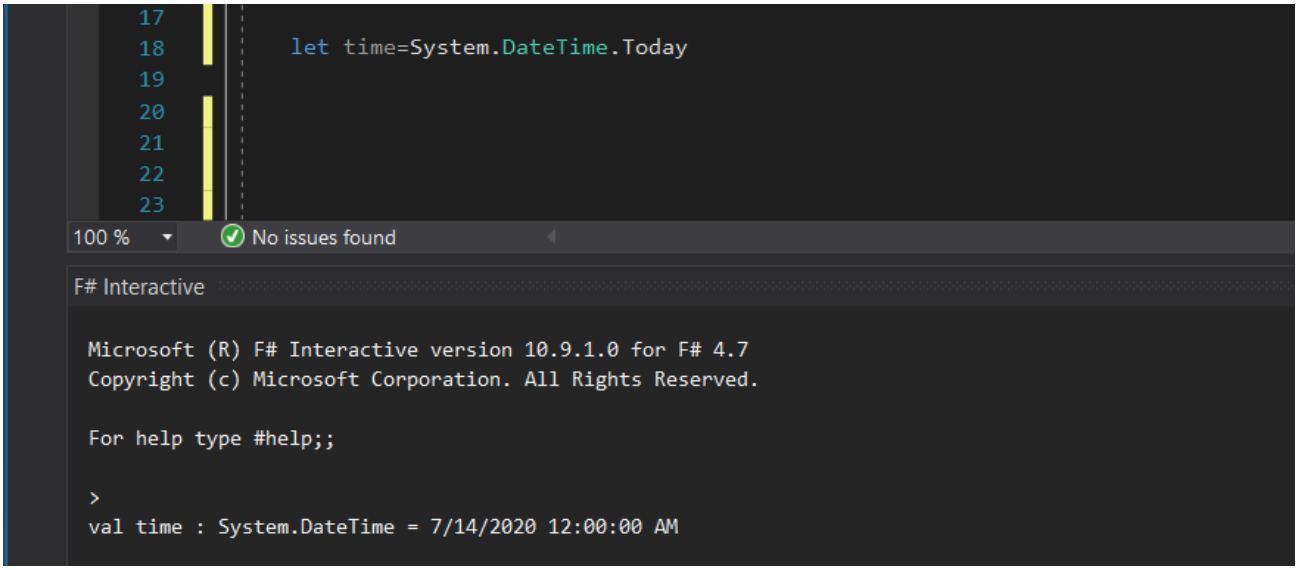
Sl. 4.20. *Pattern matching u F#-u.*

Ovaj primjer ima istu funkcionalnost kao i prethodni. Ovdje se vrijednost varijable `day` „spaja“ s jednom od navedenih vrijednosti i određuje se naziv dana u tjednu. `|_` pokriva sve ostale vrijednosti.

4.4 Razvojna okolina za funkcionalno programiranje s programskim jezikom F#

Postoji više razvojnih okolina za programiranje F#-om. Neke okoline su Visual Studio, Visual Studio for Mac i JetBrains Rider. Ovdje će biti riječ o razvojnoj okolini Visual Studio.

Kako bi se Visual Studio mogao koristiti za programiranje F#-om, potrebno je instalirati „ASP.NET and web development“. Visual Studio nudi alate za razvoj, *debugiranje*, testiranje i analizu aplikacija. Ima uređivač koda koji različito boja dijelove sintakse koda, nudi prijedloge mogućih funkcija i varijabli prilikom pisanja koda, te nudi rješenja za jednostavnije greške napravljene u kodu. *Debuger* omogućuje praćenje vrijednosti varijabli tijekom izvođenja programa, dodavanje točaka prekida (*eng. breakpoint*) koje omogućuju zaustavljanje programa na točno određenom mjestu tijekom izvođenja, te kada se dobije željena informacija nastavak izvođenja programa i mnoge druge mogućnosti. U sklopu Visual Studia za F# također dolazi REPL („ *Read Evaluate Print Loop* ”) pod nazivom F# Interactive. To je mehanizam koji omogućuje slanje proizvoljnog dijela koda na evaluaciju i dobivanje povratne informacije o njemu. Može se koristiti kao način testiranja koda. U F# interactive moguće je direktno upisati dio koda ili označiti željeni dio koda i pritiskom tipki `Alt+Enter` „poslati“ kod u F# interactive. Slika 4.21 prikazuje korištenje F# ineractive mogućnosti.



Sl. 4.21. *Korištenje F# interactive mogućnosti.*

Kada se izraz `let time=System.DateTime.Today` označi i pritisne `Alt+Enter` „pošalje“ ga se u F# interactive i može se vidjeti koja je njegova vrijednost.

Za F# dostupne su mnoge biblioteke kao što su Fsharp.Data i Deedle za rad s podatcima, Fsharp.Charting i Xplot za vizualizaciju podataka, Math.NET Numerics koja sadrži brojne algoritme iz linearne algebre, statistike i slično, FsXaml za izradu korisničkih sučelja i mnoge druge.

5.PRIJEDLOG MODELA OKOLINE ZA ANALIZU I PRIKAZ PODATAKA

5.1 Model analize međusobnog utjecaja klimatskih i ekonomskih pokazatelja

Analiza podataka radit će se na tri podatka: emisiji ugljikovog dioksida, BDP-u (bruto domaći proizvod) i postotku energije koja dolazi iz obnovljivih izvora u ukupnoj potrošenoj energiji. U analizi će se izračunati koeficijent korelacije koristeći dvije formule, onu za izračun Pearsonovog koeficijenta korelacije i onu za izračun Spearmanovog koeficijenta korelacije. Korelacija će se računati između emisije ugljikovog dioksida i BDP-a i emisije ugljikovog dioksida i korištenja energije dobivene iz obnovljivih izvora.

5.2 Vremenski i prostorni rasponi i obilježja podataka

Podatci korišteni za analizu nalaze se u vremenskom rasponu od 2000. do 2014. godine. Prikazuju vrijednosti za sljedeće pojedine regije u svijetu: Istočna Azija i Pacifik, Europa i Središnja Azija, Latinska Amerika i Karibi, Srednji Istok i Sjeverna Afrika, te Sjeverna Amerika.

U emisije ugljikovog dioksida svrstane su one emisije nastale gorenjem fosilnih goriva i proizvodnjom cementa. Uključuju ugljikov dioksid nastao korištenjem čvrstih, tekućih i plinovitih goriva. To su podatci za godišnju emisiju ugljikovog dioksida u kilotonama [25].

BDP je vrijednost dobara i usluga proizvedenih u nekoj zemlji u razdoblju od jedne godine izražena u novčanim jedinicama. Podatci prikazuju vrijednost BDP-a za pojedine regije izraženu u američkim dolarima [25].

Podatci o korištenju energije iz obnovljivih izvora prestavljaju koliki postotak u ukupnoj potrošnji energije zauzima energija dobivena iz obnovljivih izvora [25].

5.3 Dohvaćanje i priprema podataka

Podatci su preuzeti sa World Bank stranice [25] u obliku CSV datoteke. World Bank Open Data omogućuje izabiranje željenih prostornih i vremenskih raspona željenih pokazatelja i njihovo preuzimanje u željenom obliku. U preuzetoj CSV datoteci nalaze se podatci o nazivu regije, kodu regije, nazivu indikatora, kodu indikatora te same vrijednosti za indikator po godinama.

U programskom rješenju će biti potrebno izvući podatke iz CSV datoteke, te ih filtrirati tako da ostanu samo vrijednosti pokazatelja.

5.4 Statistički postupci korišteni za analizu podataka

Za analizu podataka korištena su dva izraza za izračun koeficijenta korelacije, Pearsonov i Spearmanov. Koeficijent korelacije predstavlja mjeru asocijacije između dvije varijable, no iz njega se ne može odrediti kauzalnost.

Pearsonov koeficijent korelacije je najčešće korišten koeficijent. Može imati vrijednost u intervalu [-1,1]. Ako je njegova vrijednost nula, tada varijable nisu ni u kakvoj korelaciji [26]. Izraz za izračunavanje Pearsonovog koeficijenta korelacije glasi:

$$r = \frac{n*\sum xy - \sum x \sum y}{\sqrt{\left[n*\sum x^2 - (\sum x)^2\right]\left[n*\sum y^2 - (\sum y)^2\right]}} \quad (5\text{-}1).$$

gdje su:

r - Pearsonov koeficijent korelacije

n - broj podataka u nizu

x i *y* - varijable.

Spearmanov koeficijent korelacije još se naziva i Spearmanov rang. Zasniva se na rangu podataka, a ne na samoj vrijednosti podatka. Računa se tako da se svakom podatku u nizu odredi rang u odnosu na njegovu vrijednost, pa će podatak s najvećom vrijednošću imati rang jedan, a najmanji podatak rang n, gdje je n broj podataka. Rangovi se dodijele obojima skupovima podataka. Prema [26], izraz za izračun Spearmanovog koeficijenta korelacije glasi:

$$r = 1 - \frac{6*\sum d^2}{n*(n^2-1)} \quad (5\text{-}2).$$

gdje su:

r - Spearmanov koeficijent korelacije

d - razlika rangova

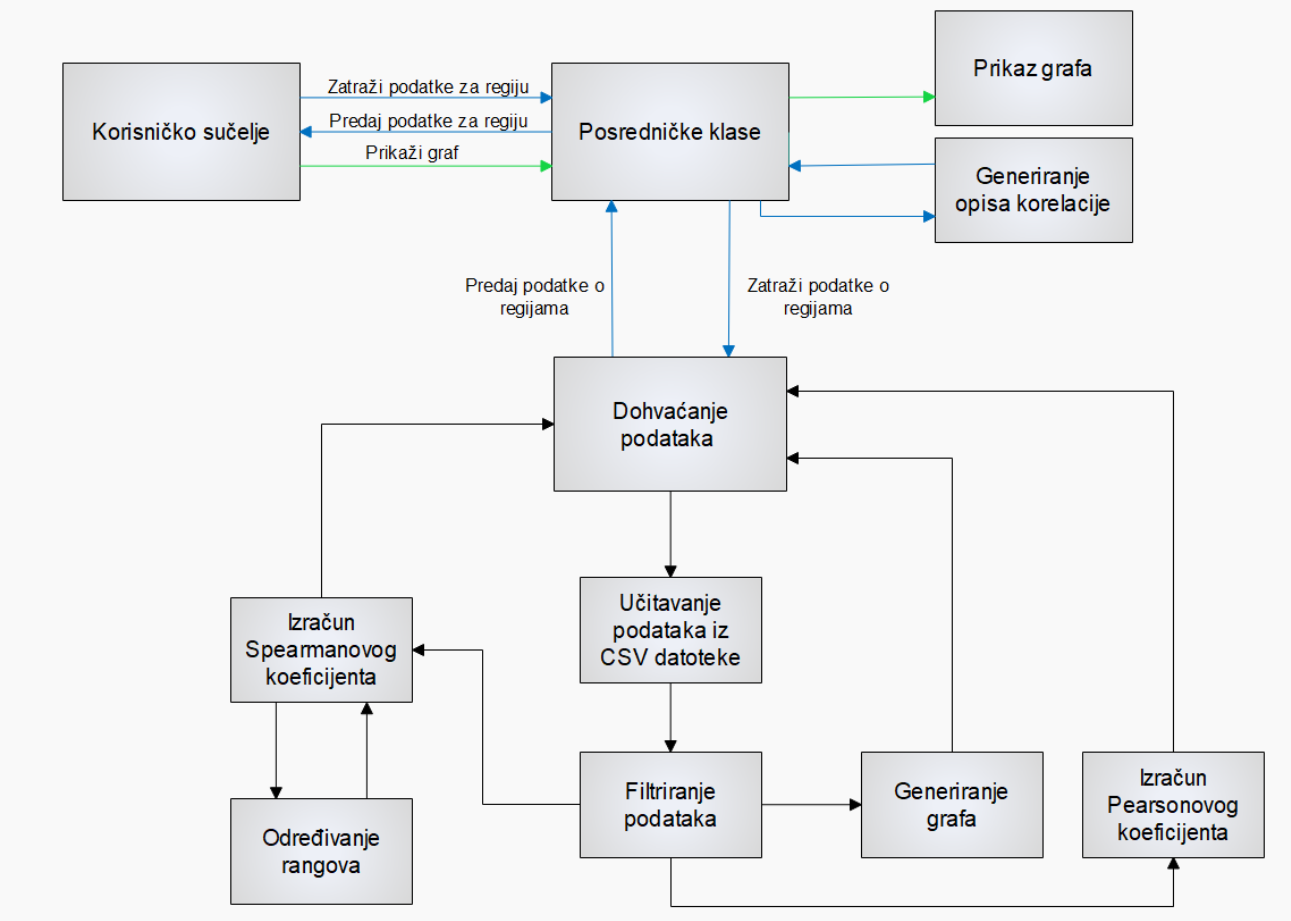
n - broj podataka u nizu.

5.5 Prikaz i interpretiranje dobivenih rezultata

Koeficijenti korelacije bit će prikazani na korisničkom sučelju koje će biti ostvareno pomoću jezika C# u obliku WPF aplikacije. Korisnik će imati mogućnost odabira za koju regiju želi vidjeti koeficijente korelacije. Uz njih će dobiti generirani odgovor jesu li pokazatelji u korelaciji i je li korelacija pozitivna ili negativna. Također će biti omogućen prikaz podataka na grafu. Graf se može koristiti kao pomoć za određivanje jesu li koeficijenti korelacije vjerodostojni budući da se može vidjeti imaju li grafovi sličan oblik.

5.6 Prikaz funkcionalnosti programskog rješenja pomoću dijagrama

Slika 5.1 prikazuje dijagram funkcionalnosti programa.



Sl. 5.1. *Dijagram funkcionalnosti programa.*

Korisničko sučelje pisano u jeziku C# omogućuje prikaz podataka za traženu regiju preko posredničkih klasa također pisanih u jeziku C#. Posredničke klase moraju dohvatiti podatke o

regijama prije nego što ih proslijede korisničkom sučelju. To čine pozivanjem funkcije implementirane u jeziku F#.

Prilikom dohvaćanja podataka prvo je potrebno te podatke dohvatiti iz CSV datoteke te ih zatim filtrirati. Podatci se moraju filtrirati jer CSV datoteka osim samih brojčanih vrijednosti sadrži i ime regije i pokazatelja kao i njihove kodove. Prilikom filtriranja potrebno je te vrijednosti ukloniti tako da ostanu samo brojčane vrijednosti. To je najlakše ostvariti uklanjanjem prve četiri vrijednosti iz retka budući da se nazivi i kodovi nalaze na tim mjestima.

Nakon što su podatci filtrirani, oni se predaju funkcijama za izračun Spearmanovog i Pearsonovog koeficijenta korelacije kao i funkciji za generiranje grafa implementiranih u jeziku F#. Funkcije za izračun Spearmanovog i Pearsonovog koeficijenta implementiraju izraze za njihove izračune. Funkcija za izračun Spearmanovog koeficijenta također poziva funkciju koja određuje rangove elemenata, jer se za izračun tog koeficijenta ne koriste vrijednosti varijabli već njihovi rangovi. Rangovi se određuju tako da se niz vrijednosti prvo sortira, a zatim se svakom elementu dodjeljuje rang n-i gdje je n broj elemenata u nizu, a i indeks elementa. Tako će najmanji element imati rang n, a najveći rang jedan. Funkcija za generiranje grafa prima niz podataka i na temelju tog niza generira graf.

Korisničkom sučelju je također potrebno preko posredničkih klasa predati opis korelacije za svaki koeficijent koji se dobije pozivom funkcije implementirane u F# jeziku. Opis govori jesu li pokazatelji u korelaciji i je li ta korelacija pozitivna ili negativna.

Na korisničkom sučelju može odabrati opcija prikaza grafa. Tada se preko posredničkih klasa poziva funkcija napisana u jeziku F# koja predana željena dva grafa prikazuje u internetskom pregledniku.

6. PROGRAMSKO RJEŠENJE ZA ANALIZU MEĐUSOBNOG UTJECAJA KLIMATSKIH I EKONOMSKIH POKAZATELJA

6.1 Postavljanje razvojne okoline za funkcijsko programiranje s programskim jezikom F#

Za ostvarenje programskog rješenja korišten je Visual Studio 2019. Kako bi se moglo raditi s F#-om u Visual Studiu, potrebno je dodati „ASP.NET and web development“ radno opterećenje (*eng. workload*). Kreirana su dva rješenja, jedno je tipa C# Windows Application pod nazivom UI, a drugo F# Console Application pod nazivom Operations. Također je potrebno u UI dodati zavisnost (*eng. dependency*) za projekt Operations kako bi se povezali. Zadnje što je potrebno napraviti je dodati potrebne biblioteke. Dodane su biblioteke Fsharp.Data i Xplot.Plotly.

6.2 Opis raspoloživih biblioteka

U programu su korištene dvije biblioteke, Fsharp.Data i Xplot.Plotly. Fsharp.Data je biblioteka koja služi za pristup različitim podatcima. Sadrži pružatelje tipa (eng. type providers) za rad s formatima kao što su CSV, HTML, XML i JSON. Također, pruža alate za parsiranje CSV, HTML i JSON datoteka i slanje HTTP zahtjeva [27]. Ova biblioteka je dostupna na NuGet-u i njen kod je dostupan na GitHub-u. Xplot je paket za vizualizaciju podataka koji sadrži JavaScript biblioteke GoogleCharts i Plotly. Xplot.Plotly omogućuje crtanje raznih tipova grafova i njihovo uređivanje [28].

6.3 Programsko ostvarenje za dohvaćanje i pripremu podataka

Podatci se nalaze u CSV datoteci te ih je u programu potrebno dohvatiti. Funkcija `GetData()` bavi se dohvaćanjem podataka te vraća niz čiji su elementi tipa sloga `Region`. Slika 6.1 prikazuje dio funkcije koji učitava podatke iz datoteke.

```
let rows=CsvFile.Load("../../../../data.csv").Rows
            |>Seq.toArray
```

Sl. 6.1. *Dohvaćanje podataka iz CSV datoteke.*

Tip `CsvFile` dio je biblioteke Fsharp.Data. Pruža dvije metode za učitavanje podataka i to `Parse` koja se koristi kada treba učitati podatke u obliku stringa i `load`. Ovdje je korištena metoda `load` budući da su podatci koji su potrebni u ovom programu tipa `double`. Svojstvo `Rows` korišteno je kako bi se dobili podatci u obliku niza redova. Kako bi pristup pojedinim elementima bio olakšan niz je prebačen u polje pomoću funkcije `Seq.toArray`.

Podatci su spremljeni u tip `Region` koji sadržava članove: `coefficents` tipa `Coefficents` i `charts` tipa `Charts`. Tip `Coefficents` sadrži četiri člana koji su tipa `double` i u koje se spremaju izračunati koeficijenti korelacije za tu regiju. Tip `Charts` sadrži tri člana tipa `PlotlyChart` u koje se spremaju grafovi za svaki indikator.

Slika 6.2. Prikazuje dio funkcije `GetData()` u kojem se kreira niz tipa `seq<Region>`.

```
let regions=seq{for i in 0..3..((Array.length rows)-3) ->
                    {


                                coefficents=SetCoefficents(rows.[i],rows.[i+1],rows.[i+2])
                                charts=SetCharts(rows.[i],rows.[i+1],rows.[i+2])

                    }
                    }
```

Sl. 6.2. *Kreiranje niza tipa `seq<Region>`.*

Ovdje je korišten izraz za kreiranje niza. Koristi se varijabla `i` koja poprima vrijednosti 0,3,6...N gdje je N duljina niza minus tri. Razlog tome je što su podatci u CSV datoteci složeni tako da podatci za istu regiju idu slijedno jedan za drugim i to redom: emisije ugljikovog dioksida, BDP i korištenje energije dobivene iz obnovljivih izvora. Budući da je tip `Region` slog, svi se podatci moraju predati odjednom tako da odmah moramo pristupiti svim redovima za jednu regiju. Tako se emisijama ugljikovog dioksida za regiju pristupa kao `rows.[i]`, BDP-u kao `rows.[i+1]` i korištenju energije dobivene iz obnovljivih izvora kao `rows.[i+2]`.

Za postavljanje vrijednosti članova korištene su dodatne funkcije. Slika 6.3 prikazuje te funkcije.

```
let getValuesOnly(x:CsvRow)=
    x.Columns |>Seq.skip 4 |>Seq.map double



let SetCoefficents(co2:CsvRow,gdp:CsvRow,renEn:CsvRow):Cofficents=
    {
        pearsonCoefficentCo2GDP=pearsonCoefficent(getValuesOnly(co2),getValuesOnly(gdp))
        pearsonCoefficentCo2RenEn=pearsonCoefficent(getValuesOnly(co2),getValuesOnly(renEn))
        spearmanCoefficentCo2GDP=spearmanCoefficent(getValuesOnly(co2),getValuesOnly(gdp))
        spearmanCoefficentCo2RenEn=spearmanCoefficent(getValuesOnly(co2),getValuesOnly(renEn))
    }

let SetCharts(co2:CsvRow,gdp:CsvRow,renEn:CsvRow) : Charts=
    {       Co2Chart=getChart(getValuesOnly(co2),co2.Columns.[2])
            GDPChart=getChart(getValuesOnly(gdp),gdp.Columns.[2])
            RenEnChart=getChart(getValuesOnly(renEn),renEn.Columns.[2])
    }
```

Sl. 6.3. *Dodatne funkcije za postavljane članova tipa* `Region`.

Za postavljanje člana `coefficents` korištena je funkcija `SetCoefficents` koja prima tri ulazna parametra tipa `CsvRow`. Ova funkcija postavlja članove u tipu `coefficents` pozivanjem funkcija za izračun Spearmanovog i Pearsonovog koeficijenta korelacije. Prilikom predaje argumenata tim funkcijama poziva funkciju `getValuesOnly`. Ta funkcija prima ulazni argument tipa `CsvRow` i vraća niz tipa `seq<double>`. Razlog korištenja ove funkcije je to što jedan red ne sadrži samo brojčane podatke već i naziv i kod regije i pokazatelja. Ova funkcija te vrijednosti uklanja korištenjem naredbe `Seq.skip 4` koja vraća novi niz bez prva četiri člana predanog niza.

Za postavljanje člana `charts` poziva se funkcija `SetCharts` koja prima tri ulazna parametra tipa `CsvRow`. Ova funkcija postavlja članove tipa `Charts` pozivajući funkciju `getChart`. Ta funkcija prima niz tipa `seq<double>` koji predstavlja niz brojčanih vrijednosti koje trebaju biti prikazane na grafu i string koji predstavlja naziv pokazatelja koji će se postaviti kao naslov grafa. Pod naziv pokazatelja predaje joj se treći stupac u redu budući da se u tom stupcu nalazi ime pokazatelja. Ova funkcija kao rezultat vraća tip `PlotlyChart`.

6.4 Programsko ostvarenje analize podataka

Za izračun koeficijenata korelacije implementirane su dvije funkcije, `pearsonCoefficent` i `spearmanCoefficent`. Slika 6.4 prikazuje funkciju `pearsonCoefficent`.

```
let pearsonCoefficent (x : seq<double>, y : seq<double>)=
    let n=double ( Seq.length x)
    let nominator=n * getSumxy(x,y)-Seq.sum x * Seq.sum y
    let denominator=(n*getSumOfSqares x-getSqareOfSum x)*(n*getSumOfSqares y-getSqareOfSum y)
                    |>sqrt

    nominator/denominator
```

Sl. 6.4. *Funkcija* `pearsonCoefficent`.

Ova funkcija prima dva ulazna parametra tipa `seq<double>` koji predstavljaju skupove podataka koji će biti korišteni za izračun Pearsonovog koeficijenta korelacije. Za izračun brojnika i nazivnika poziva pomoćne funkcije za izračun zbroja umnožaka `x` i `y` elemenata, zbrojeve elemenata pojedinog niza i zbrojeve kvadrata elemenata pojedinog niza. Duljina niza `n` mora se eksplicitno pretvoriti u tip `double` budući da F# ne podržava implicitne konverzije tipova.

Slika 6.5 prikazuje implementaciju funkcije `spearmanCoefficent`.

```
let spearmanCoefficent(x:seq<double>, y:seq<double>)=
    Seq.zip (getRanks x) (getRanks y)
            |>Seq.map (fun (x,y)->x-y)
            |>spearmanCalculateFormula (double (Seq.length x))
```

Sl. 6.5. *Funkcija* `spearmanCoefficent`.

Ova funkcija prima dva ulazna podatka tipa `seq<double>` koji predstavljaju skupove podataka koji će biti korišteni za izračun Spearmanovog koeficijenta korelacije. Ova funkcija prvo stvara novi niz koji se dobije spajanjem nizova čije su vrijednosti rangovi elemenata. Zatim se pomoću `Seq.map` dobije niz koji sadrži razlike između elemenata nizova s rangovima. Taj niz se zatim predaje funkciji `spearmanCalculateFormula` u kojoj je implementirana formula za izračun koeficijenta korelacije.

Slika 6.6 prikazuje implementaciju funkcije `getRanks` koja se koristi za dobivanje niza koji sadrži rangove elemenata.

```
let getRanks(x:seq<double>)=

    let xSorted=Seq.sort x
    seq{ for xel in x -> double((Seq.length x)-(Seq.findIndex (fun el->el=xel) xSorted))}
```

Sl 6.6. *Funkcija* `getRanks`.

Ova funkcija prvo sortira primljeni niz i zatim generira novi čiji se elementi dobiju tako da se za svaki element niza `x` nađe njegov ekvivalent u sortiranom nizu te se od broja elemenata oduzme indeks tog elementa. Tako će, na primjer, najmanji element biti na prvom mjestu u sortiranom nizu i njegov indeks će biti nula. Niz `x` sadrži petnaest podataka budući da je vremenski raspon podataka od 2000. do 2014. godine. Kada se od 15 oduzme nula dobit će se da je rang tog elementa 15.

Slika 6.7 prikazuje implementaciju funkcije `spearmanCalculateFormula`.

```
let spearmanCalculateFormula n d=
            1.0-(6.0*(getSumOfSqares d))/(n*(n*n-1.0))
```

Sl 6.7. *Funkcija* `spearmanCalculateFormula`.

Ova funkcija implementira izraz za izračun Spearmanovog koeficijenta korelacije. Konstante 1.0 i 6.0 su napisane u decimalnom obliku, jer F# ne podržava implicitnu pretvorbu tipova.

6.5 Programsko ostvarenje za prikaz i interpretiranje rezultata analize

Za prikaz podataka napravljena je WPF aplikacija u jeziku C#. Izgled korisničkog sučelja napisan je u XAML-u i napravljene su dvije klase napisane u C#-u koje služe za komunikaciju sa sučeljem i F# kodom. C# može lako komunicirati s F#-om. Može pozvati bilo koju funkciju napisanu u F#-u i raditi s tipovima podataka iz F#-a.

Sučelje omogućuje odabir jedne od ponuđenih regija te ispis koeficijenata korelacije za emisije ugljikovog dioksida i BDP-a i emisije ugljikovog dioksida i korištenje energije dobivene iz obnovljivih izvora. Također, za svaki koeficijent daje odgovor jesu li varijable u korelaciji i je li korelacija negativna ili pozitivna. Slika 6.8 prikazuje funkciju `generateResponse` koja generira opis koeficijenta korelacije.

```
let generateResponse(coefficent:double,secondIndicator:string)=
    let response=
        match coefficent with
        |0.0->" are not in correlation."
        |_->if coefficent<0.0 then " have negative correlation."
            else " have positive correlation."
    "Co2 emissions and " + secondIndicator + response
```

Sl. 6.8. *Funkcija* `generateResponse`.

Ova funkcija prima koeficijent korelacije i naziv drugog indikatora budući da je prvi uvijek emisija ugljikovog dioksida. Prvo pomoću mehanizma *pattern matching* provjerava je li koeficijent nula. Ako je, generira dio odgovora. U slučaju da je bilo koja vrijednost osim nula, pomoću izraza `if/else` provjerava je li veći ili manji od nule te generira odgovor na temelju toga.

Sučelje također omogućuje prikaz podataka u obliku grafa u internetskom pregledniku klikom na gumb „See Graph Here..“. Slika 6.9 prikazuje funkciju `ShowChart` koja prikazuje graf u internetskom pregledniku pomoću funkcije `Chart.ShowAll`.

```
let ShowChart (x:PlotlyChart, y:PlotlyChart)=
    Chart.ShowAll(seq{x;y})
```

Sl. 6.9. *Funkcija* `ShowChart`.

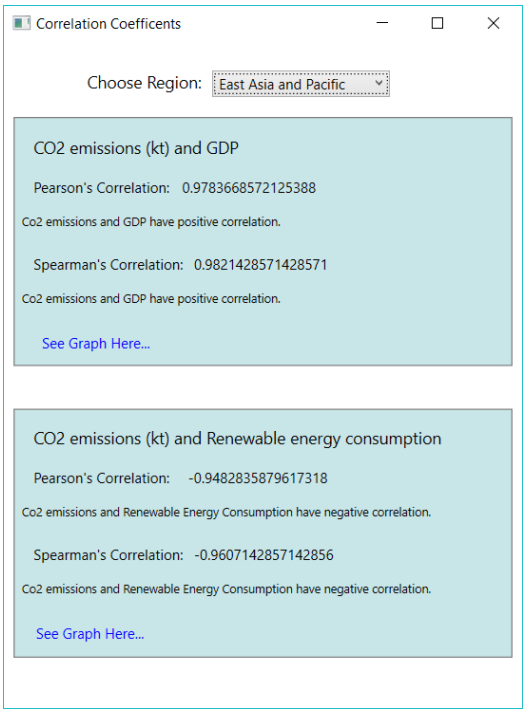
Korištena je funkcija `ShowAll` koja prima niz grafova i zatim njih prikazuje u istom prozoru preglednika.

6.6 Ispitivanje programskog rješenja na primjerima s analizom

U ovom potpoglavlju bit će prikazani rezultati korelacije između emisija ugljikovog dioksida i BDP-a i emisija ugljikovog dioksida i postotka korištenja energije dobivene iz obnovljivih izvora za neke od ponuđenih regija kao što su: Istočna Europa i Pacifik, Europa i Središnja Azija i Istočna Azija.

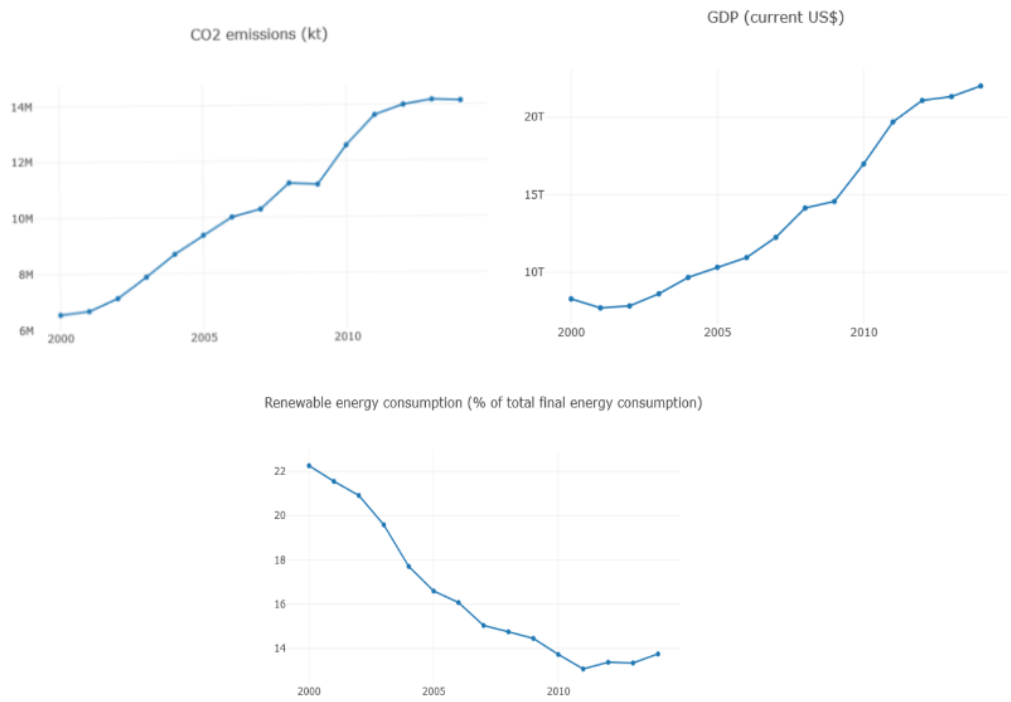
6.6.1. Rezultati za regiju Istočna Europa i Pacifik

Slika 6.10 prikazuje odabir regije Istočna Europa i Pacifik.



Sl. 6.10. *Podatci za regiju Istočna Europa i Pacifik.*

Iz ovoga je vidljivo da emisije ugljikovog dioksida i BDP imaju pozitivnu korelaciju što znači da se oboje povećavaju ili smanjuju. Pearsonov koeficijent korelacije je **0.9784** što je poprilično velik koeficijent budući da je najveća moguća apsolutna vrijednost jedan. Vrijednost Spearmanovog koeficijenta korelacije je također visoka i iznosi **0.9821**. Vidljivo je da između ova dva koeficijenta postoji razlika u vrijednosti, no ona nije jako velika. Emisije ugljikovog dioksida i postotak korištenja energije dobivene iz obnovljivih izvora imaju negativnu korelaciju što znači da se jedna varijabla smanjuje, a druga povećava. Vrijednosti oba koeficijenta su visoke kao i kod prethodnih. Međutim, ovdje je razlika između Pearsonovog i Spearmanovog koeficijenta veća. Slika 6.11 prikazuje grafove vrijednosti pokazatelja za ovu regiju.

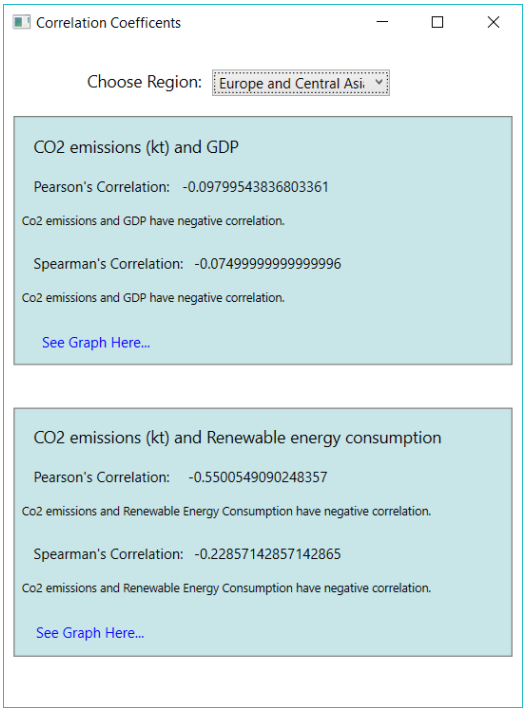


Sl. 6.11. *Grafovi vrijednosti za regiju Istočna Europa i Pacifik.* **a)** *Emisije ugljikovog dioksida.* **b)** *BDP.* **c)** *Korištenje energije dobivene iz obnovljivih izvora.*

Vidljivo je da grafovima pod 6.11.a) i 6.11.b) vrijednosti rastu, a zato je korelacija između tih varijabli pozitivna. Vrijednosti na grafu 6.11.c) opadaju zato je on u negativnoj korelaciji s emisijama ugljikovog dioksida.

6.6.2. Rezultati za regiju Europa i Središnja Azija

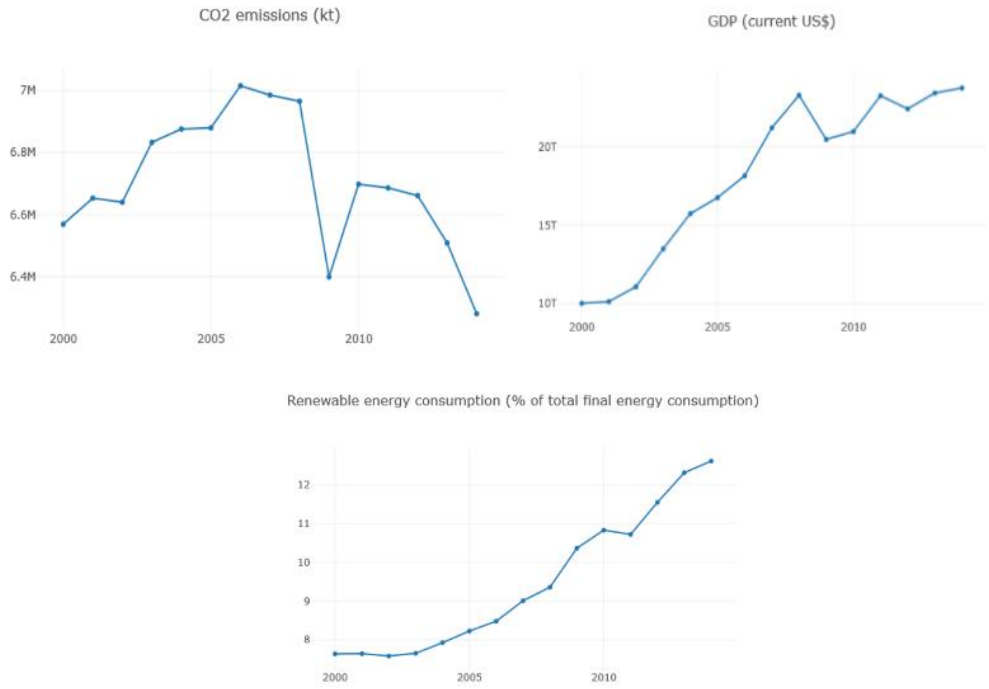
Slika 6.12 prikazuje odabir regije Europa i Središnja Azija. Iz nje je vidljivo da emisije ugljikovog dioksida i BDP imaju negativnu korelaciju, no vrijednosti koeficijenata su izrazito male tako da se može reći da je korelacija praktički nepostojeća. Također, postoji veća razlika u vrijednosti Spearmanovog i Pearsonovog koeficijenta.



Sl. 6.12. *Podatci za regiju Europa i Središnja Azija.*

Emisije ugljikovog dioksida i postotak korištenja energije dobivene iz obnovljivih izvora imaju negativnu korelaciju. Koeficijent korelacije je veći u odnosu na onaj sa BDP-om, no i dalje nije visok. Razlika između vrijednosti koeficijenata je također velika. Razlog tome je što su vrijednosti u ovom primjeru u odnosu na prethodni manje linearno raspoređene. Što su vrijednosti više linearno raspoređene, koeficijent korelacije bit će precizniji i vjerodostojniji.

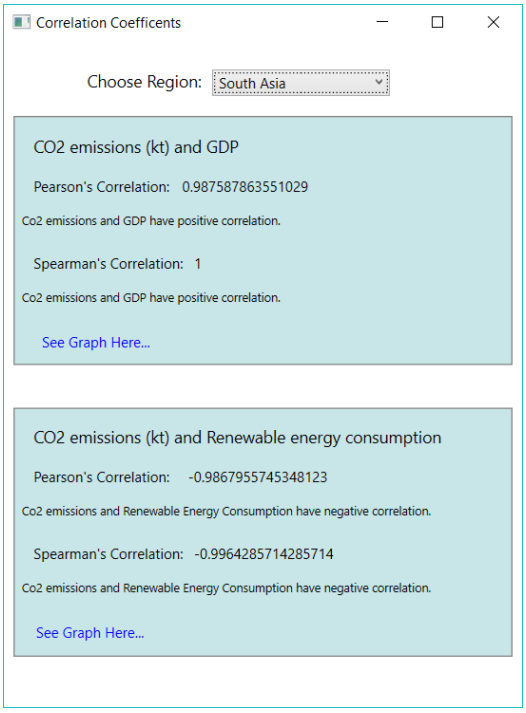
Slika 6.13 prikazuje grafove vrijednosti pokazatelja za ovu regiju i iz nje je vidljivo da grafovi 6.13.b) i 6.13.c) imaju porast koji sliči linearnom, ali graf 6.13.a) nema. Zbog toga koeficijenti korelacije imaju izrazito male vrijednosti i postoje velike razlike u vrijednosti Spearmanovog i Pearsonovog koeficijenta.



Sl. 6.13. *Grafovi vrijednosti za regiju Europa i Središnja Azija.* **a)** *Emisije ugljikovog diosida.* **c)** *BDP.* **c)** *Korištenje energije dobivene iz obnovljivih izvora.*

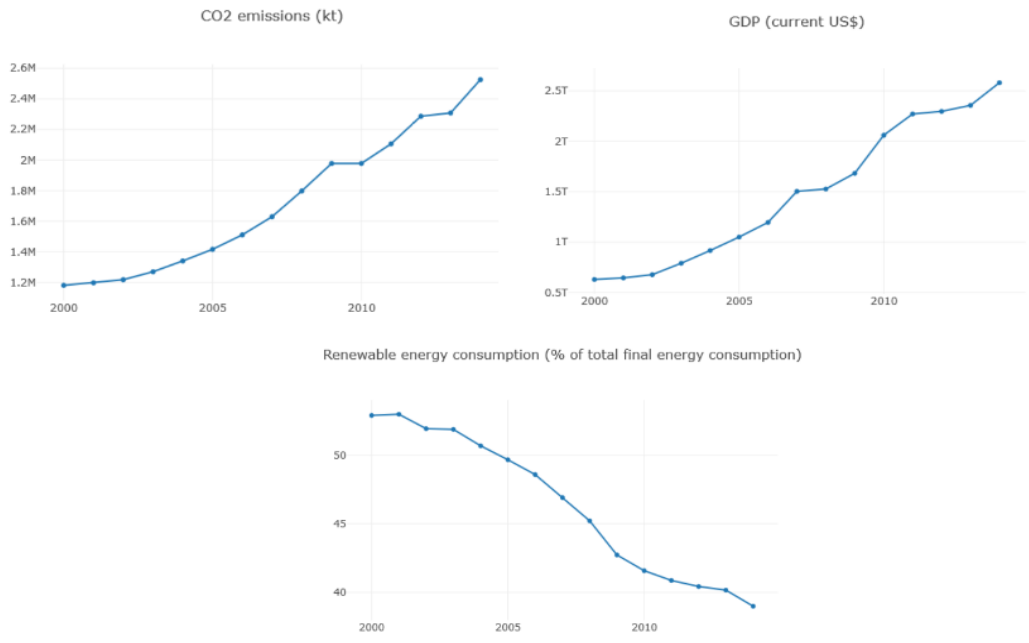
6.6.3. Rezultati za regiju Istočna Azija

Slika 6.14 prikazuje vrijednosti dobivene odabirom regije Istočna Azija. Emisije ugljikovog dioksida i BDP imaju pozitivnu korelaciju. Vrijednosti koeficijenata su poprilično visoke. Pearsonov koeficijent ima vrijednost **0.988**, a Spearmanov **1**. Vrijednost **1** znači da su varijable u potpunosti korelirane, ali budući da postoji razlika u odnosu na Pearsonov koeficijent, to ovdje nije slučaj. Razlika među koeficijentima postoji, no nije velika kao u prethodnom primjeru. Emisije ugljikovog dioksida i postotak korištenja energije dobivene iz obnovljivih izvora imaju negativnu korelaciju. Apsolutne vrijednosti koeficijenata su visoke i nema velike razlike između vrijednosti Pearsonovog i Spearmanovog koeficijenta.



Sl. 6.14. *Podatci za regiju Istočna Azija.*

Slika 6.15 prikazuje grafove vrijednosti pokazatelja za ovu regiju.



Sl. 6.15. *Grafovi vrijednosti za regiju Istočna Azija.* **a)** *Emisije ugljikovog dioksida.* **c)** *BDP.* **c)** *Korištenje energije dobivene iz obnovljivih izvora.*

Iako grafovi 6.15.a) i 6.15.b) imaju izrazito sličan oblik, vidljivo je da nisu u potpunosti isti te da emisije ugljikovog dioksida i BDP nisu u potpunosti korelirani.

7. ZAKLJUČAK

Koristeći funkcionalni programski jezik F# i načela funkcionalnog programiranja u radu su uspješno implementirani izrazi i postupci izračuna Pearsonovog i Spearmanovog koeficijenta korelacije. Također, implementirane su funkcije za dohvaćanje i grafički prikaz podataka. Korištenjem programskog jezika C#, implementirano je korisničko sučelje i klase za komunikaciju s tim sučeljem i programskim kodom u F#-u. Rezultati dobiveni ovim programom pokazali su da kod nekih regija postoji jaka korelacija između emisija ugljikovog dioksida i BDP-a i emisija ugljikovog dioksida i postotka korištenja energije dobivene iz obnovljivih izvora. No, u nekim slučajevima korelacija je bila zanemarivo mala. Također je u nekim slučajevima razlika između vrijednosti Pearsonovog i Spearmanovog koeficijenta korelacije bila velika što pokazuje da oni nisu uvijek pouzdani.

Programski jezik F# omogućuje pisanje sažetog i čitljivog koda. Iako se u prvom redu radi o funkcionalnom jeziku, on dopušta i korištenje drugih programskih paradigmi što može uvelike olakšati programiranje nekome tko se prvi put susreće s funkcionalnim programiranjem. Iako postoje biblioteke koje omogućuju izradu korisničkog sučelja korištenjem F#-a i funkcionalnog programiranja, mnogi i dalje uzimaju korištenje C#-a za bolji izbor. Programski jezici C# i F# su u potpunosti kompatibilni i to olakšava komunikaciju između C# i F# koda. Prednosti korištenja F#-a za rješavanje ovog problema su kratki i jasni kod. Većina implementiranih funkcija su kratke, lako čitljive i razumljive. Zbirke i pružatelji podataka u F#-u omogućili su lako pristupanje, filtriranje i rad s podatcima. Kompatibilnost s programskim jezikom C# omogućila je laku izradu korisničkog sučelja i pristupanje svim funkcijama i tipovima podataka implementiranim u F#-u.

Iako implementirani program nije presložen, ima veliki potencijal za proširenja implementiranjem drugih složenijih metoda određivanja korelacije ili drugih analitičkih metoda. Također ga je moguće proširiti tako da radi s puno većim i složenijim skupovima podataka.

LITERATURA

[1] Haskell in Industry, HaskellWiki, dostupno na: http://wiki.haskell.org/Haskell_in_industry , [7.7.2020.]

[2] An Introduction to Elm, Elm, dostupno na: https://guide.elm-lang.org/, [7.7.2020.]

[3] What is Erlang, Erlang, dostupno na: http://erlang.org/faq/introduction.html, [7.7.2020.]

[4] Succes Stories, Clojure, dostupno na: https://clojure.org/community/success_stories, [7.7.2020.]

[5] A. Khanfor, Y. Yang, An Overview of Practical Impacts of Functional Programming, 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), Nanjing, China, 2017, str. 50-54.

[6] D. Soshnikov, Using Functional Programming for Development of Distributed, Cloud and Web Applications in F#, ArXiv, 2015.

[7] I. Abraham, SAFE Stack: Functional Web Programming for .NET Core, Vol. 34, No. 10, listopad, 2019.

[8] Fabulous for Xamarin.Forms, fsprojects, dostupno na: https://fsprojects.github.io/Fabulous/Fabulous.XamarinForms/ [8.7.2020.]

[9] fsharp.org, dostupno na: https://fsharp.org/ [10.7.2020.]

[10] What is R?, R-project.org, dostupno na: https://www.r-project.org/about.html [8.7.2020.]

[11] P. Lemenkova, An Empirical Study od R Applications for Data Analysis in Marine Geology, Marine Science and Technology Bulletin, str. 1-9, 2019.

[12] Luna, dostupno na: https://luna-lang.org/ [8.7.2020.]

[13] T. Petricek, D. Syme, Doing Web-Based Data Analytics with F# (Case study)

[14] A.C. Martin, Bristlecone: An F# Library for Model-Fitting and Model Selection of Ecological Time Series Models, OSF Preprints, 2019.

[15] Testimonials and Quotes, Fsharp.org, dostupno na: https://fsharp.org/testimonials/ [8.7.2020.]

[16] I. Čukić, Functional Programming in C++, Manning Publications Co., Shelter Island, NY 11964, 2019.

[17] I. Abraham, Get Programming with F#: A Guide for .NET developers, Manning Publications Co., United States, 2018.

[18] J.P. Mueller, Functional Programming For Dummies, John Wiley & Sons, Inc., Hobokenn New Jersey, 2019.

[19] R.Rojas, A Tutorial Introduction to the Lambda Calculus, ArXiv, 2015.

[20] P. Hudak, Conception, Evolution, and Application of Functional Programming Languages, ACM Computing Surveys, Vol. 21, No. 3, str. 360-411, 1989.

[21] J. Kunasaikaran, A. Iqbal, A Brief Overview of Functional Programming Languages, electronic Journal of Computer Science and Information Technology (eJCSIT), Vol. 6, No. 1, str. 32-36, 2016.

[22] What is Ocaml?, OCaml, dostupno na: https://ocaml.org/learn/description.html [10.7.2020]

[23] D., Syme, The Early History of F#, Proceedings of the ACM on Promgramming Languages, Vol. 4, No. HOPL, članak 75, lipanj, 2020.

[24] F# Documentation, Microsoft, dostupno na: https://docs.microsoft.com/en-us/dotnet/fsharp/ [12.7.2020]

[25] World Bank Open Data, World Bank, dostupno na: https://data.worldbank.org/ [18.7.2020.]

[26] S. Boslaugh,Statistics in a Nutshell, Second Edition, O'Reilly Media, Inc., SAD, 2013.

[27] F# Data: Library for Data Access, dostupno na: https://fsharp.github.io/FSharp.Data/ [19.7.2020]

[28] Xplot.Plotly, nuget, dostupno na: https://www.nuget.org/packages/XPlot.Plotly/ [19.7.2020]

SAŽETAK

Funkcionalno programiranje jedna je od programskih paradigmi koja ima mnoge primjene. Jedna od primjena funkcionalnog programiranja je analiza podataka. U ovom radu je u programskom jeziku F# ostvaren program za analizu podataka. Taj program analizira podatke o emisijama ugljikovog dioksida, BDP-u i postotku korištenja energije dobivene iz obnovljivih izvora za pojedine regije u svijetu računajući Pearsonov i Spearmanov koeficijent korelacije. Prije izračunavanja koeficijenata korelacije potrebno je dohvatiti podatke iz CSV datoteke te filtrirati vrijednosti potrebne za izračun koeficijenata. Pomoću grafičkog sučelja implementiranog u jeziku C#, program omogućuje prikaz vrijednosti koeficijenata za izabranu regiju kao i prikaz grafova vrijednosti indikatora te regije. Iz izračunatih koeficijenata korelacije može se vidjeti jesu li indikatori korelirani i podudaraju li se vrijednosti Spearmanovog i Pearsonovog koeficijenta korelacije ili među njihovim vrijednostima postoji veća razlika.

Ključne riječi: analiza podataka, F#, funkcionalno programiranje, klimatski i ekonomski pokazatelji, koeficijent korelacije.

ABSTRACT

Functional programming is one of the programming paradigms that has many applications. One of which is data analysis. In this paper F# was used to realize a program for data analysis. That program analyses carbon dioxide emissions, GDP and percentage of renewable energy use data for individual regions in the world by calculating Pearson’s and Spearman’s correlation coefficients. Before calculating correlation coefficients, it is necessary to extract data from CSV file and filter it so that only the needed values remain. Through graphical user interface implemented in C# the program enables display of values of the coefficients for chosen region as well as charts which display values of the indicators. From calculated correlation coefficients it is possible to see if the indicators are in correlation and whether the values of Spearman’s and Pearson’s coefficients match or is there a difference between them.

Keywords: data analysis, F#, functional programming, , climate and economic indicators, correlation coefficient

ŽIVOTOPIS

Barbara Bilonić rođena je 10. kolovoza 1998. u Osijeku. Godine 2013. završila je osnovnu školu „OŠ Tin Ujević“ u Osijeku. Godine 2017. završila je III. gimnaziju Osijek i iste godine upisala Fakultet elektrotehnike, računarstva i informacijskih tehnologija na kojem trenutno pohađa treću godinu preddiplomskog sveučilišnog studija računarstva.

PRILOZI (na CD-u)

Prilog 1. UI implementacija korisničkog sučelja u C#-u.

Prilog 2. Operations implementacija funkcija u F#-u.

Prilog 3. data.csv csv datoteka s podatcima o regijama.