Povezivanje logičkog programiranja s drugim programskim paradigmama

Šestak, Petar

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Petar Šestak

POVEZIVANJE LOGIČKOG PROGRAMIRANJA S DRUGIM PROGRAMSKIM PARADIGMAMA

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Petar Šestak

Matični broj: 45310/16–R

Studij: Informacijsko i programsko inženjerstvo

POVEZIVANJE LOGIČKOG PROGRAMIRANJA S DRUGIM PROGRAMSKIM PARADIGMAMA

DIPLOMSKI RAD

Mentorica:

Izv. prof. dr. sc. Sandra Lovrenčić

Varaždin, rujan 2018.

Petar Šestak

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu se opisuje koncept programskih paradigmi pri čemu se obrađuju neke vrste zajedno s pripadajućim karakteristikama i predstavnicima. U središtu samog rada je logička paradigma te načini povezivanja nje s drugim programskim paradigmama. Od tehnologija je u radu fokus na programskom jeziku Prolog koji je najznačajniji predstavnik logičke paradigme, a zbog relativno velikog broja modula i same zajednice je konkretno odabrana implementacija SWI-Prolog. U razradi teme se tako navode moduli i proširenja koji omogućuju da se spomenutim programskim jezikom iskoriste koncepti drugih programskih paradigmi, ali se ujedno obrađuju načini povezivanja programa napisanih u Prologu s onima izrađenih korištenjem drugih programskih jezika pa tako i paradigmi. Prilikom obrade pojedinih načina povezivanja različitih paradigmi, priloženi su i pripadajući programski primjeri (kako isječci programskog koda, tako i opisi izrađenih aplikacija priloženih uz rad) kojima su dočarane motivacija i primjenjivost za povezivanje logičke paradigme s nekom programskom paradigmom na određeni način.

Ključne riječi: programske paradigme, logička paradigma, logičko programiranje, Prolog, JPL, cjevovodi, SWI-Prolog, 3-slojna arhitektura softvera

Sadržaj

1. Uvod……1
2. Programska paradigma..... 2
2.1. Proceduralna paradigma..... 3
2.2. Objektno-orijentirana paradigma..... 5
2.2.1. Nasljeđivanje i podtipovi..... 6
2.2.2. Enkapsulacija i apstrakcija..... 7
2.2.3. Dinamički odabir metoda..... 8
2.2.4. Polimorfizam..... 8
2.3. Logička paradigma..... 9
2.3.1. Logičko programiranje s ograničenjima.....10
2.4. Funkcionalna paradigma.....11
2.5. Programiranje pogonjeno događajima.....12
3. Načini povezivanja logičke paradigme s drugim paradigmama.....14
3.1. Ugrađeni elementi drugih paradigmi.....14
3.1.1. Proceduralna paradigma.....15
3.1.2. Objektno-orijentirana paradigma.....16
3.1.3. Logičko programiranje s ograničenjima.....21
3.1.4. Funkcionalna paradigma.....22
3.1.5. Programiranje pogonjeno događajima.....26
3.2. Povezivanje s drugim programskim jezicima.....28
3.2.1. Komunikacija s procesom (izvršavatelja) programa.....28
3.2.2. Korištenje gotovog sučelja za rad s interpreterom jezika.....33
4. Programski primjeri povezivanja.....36
4.1. Aplikacija za prikaz rodbinskih odnosa određene osobe.....36
4.1.1. Inačica potpuno kreirana u Prologu.....38
4.1.2. Inačica s prezentacijskim slojem kreiranim u Javi.....40
4.1.3. Inačica s prezentacijskim i obradbenim slojem kreiranim u Javi.....42

4.2. Aplikacija za prikaz svih podređenih zaposlenika 42
4.2.1. Inačica napisana korištenjem Prologa s XPCE modulom 43
4.2.2. Inačica napisana korištenjem Flora-2 programskog jezika 45
4.3. Aplikacija za rješavanje problema raspoređivanja nastave 46
4.3.1. Inačica s naknadnim prikazom svih rezultata 50
4.3.2. Inačica s prikazom svakog rezultata na zahtjev 51
4.3.3. Inačica s kontinuiranim prikazom pronađenih rezultata 52
5. Zaključak 53
6. Popis literature 54
7. Popis slika 58
8. Popis tablica 59

1. Uvod

Izumom računalā prije više od 50 godina, prepoznat je njihov potencijal kojim bi ona rješavala vremenski-zahtjevne proračune te rutinske i repetitivne poslove. No za to je potrebno u njih usaditi znanje na jeziku koja ona poznaju, a to je isključivo strojni jezik koji (za sada) poznaje samo 2 stanja, a to je stanje kada ne teče struja (ili je izrazito malog intenziteta) te stanje kada ona teče. Međutim, kako bi se programerima olakšalo pisanje programa za računala, osmišljeni su programski jezici viših razina kao i sami prevoditelji koji napisani programski kôd prevode u strojni jezik. Tako su u ranim počecima programi bili pisani u zbirnom jeziku (engl. *assembly*), nešto kasnije u programskim jezicima poput C-a, COBOL-a, Pascala i FORTRAN-a zatim su se počeli pojavljivati programski jezici različitih paradigmi, odnosno prikladni za rješavanje specifičnih vrsta problema.

Kako između različitih programskih jezika postoje sličnosti (npr. u načinu razmišljanja, dekompoziciji problema i zadavanja rješenja, rukovanju resursima), javila se potreba da se oni (i njihove sposobnosti) grupiraju. Tako je nastao pojam programskih paradigmi kojih danas ima nekoliko desetina. Najpopularnije među njima su proceduralna, objektno-orijentirana, logička, funkcionalna, kao i paradigma programiranja pogonjenog događajima.

U ovom radu ću opisati koncept programskih paradigmi, navesti neke vrste, njihove karakteristike i predstavnike, dok će u nastavku biti osvrt načina njihovog povezivanja s logičkog paradigmom koja je u središtu razmatranja ovog rada. U radu ću se fokusirati na programski jezik Prolog koji je najznačajniji predstavnik razmatrane paradigme i to ponajprije onaj implementacije SWI-Prolog. Tako će se razrada teme bazirati na navođenju modula i proširenja koji omogućuju da se spomenutim programskim jezikom dobije pristup elementima drugih programskih paradigmi, ali ujedno i na načinu povezivanja programa napisanih u Prologu s onima izrađenih korištenjem drugih programskih jezika pa tako i paradigmi. Pokušat ću pritom u što većoj mjeri potkrijepiti obrađivane načine povezivanja s izradom i opisom programskih primjerima na kojima će biti vidljivo što je iskorišteno iz koje paradigme te što je time postignuto.

2. Programska paradigma

U ovom poglavlju objasnit ću što je programska paradigma i definirati podjelu programskih paradigmi. Mogu reći kako svaki programski jezik ostvaruje jednu ili više programskih paradigmi. Mnogi programski jezici podržavaju dvije programske paradigme. Prva paradigma izabrana je za vrstu problema koju rješava programski jezik, dok je druga paradigma izabrana zbog apstrakcije i modularnosti te se ona koristi prilikom programiranja i projektiranja većih projekata. Programski jezik Prolog u svojoj standardnoj varijanti tako podržava dvije paradigme, a to su paradigma logičkog programiranja koja je sastavljena od unifikacije i dubinskog pretraživanja po bazi (engl. *Depth-first search*), dok je druga imperativna paradigma. Za dodavanje činjenica u bazu znanja, Prolog koristi predikate poput *retract* i *assert* koji programu omogućuju dodavanje i brisanje programskih klauzula. (Trevisan, 2015)

Programsku paradigmu možemo definirati kao stil ili način pisanja programa. Za programsku paradigmu možemo reći kako je to način djelovanja (poput programiranja), no ne i konkretna stvar kao što su to programski jezici koji nam olakšavaju pisanje programa podržavanjem paradigmi. (Ray, *s.a.*)

Prisutna su nastojanja da se programske paradigme grupira po kategorijama po nekim sličnostima i karakteristikama. Tako je najpoznatija klasifikacija paradigmi na imperativnu i deklarativnu paradigmu pri čemu imperativnu karakterizira mijenjanje stanja memorije prilikom rješavanja algoritamskih problema, dok se kod deklarativne nastoji problem riješiti na višoj razini apstrakcije čime se odbacuje potreba za promjenom stanja memorije. Tako se na imperativnoj paradigmi baziraju paradigme poput proceduralne i objektno-orijentirane, dok se deklarativnima smatraju logička, funkcionalna i paradigma programiranja pogonjenog događajima. Ova podjela je prilično gruba jer kako bi deklarativan programski jezik bio proširiv i opće namjene, javlja se potreba za definiranjem stanja memorije (barem jednokratnim korištenjem inicijalizacije) čime takvi jezici poprimaju elemente imperativne paradigme. (Gabbrielli i Martini, 2010; Trois i sur., 2016)

Jedna paradigma može biti sasvim dovoljna za rješavanje nekog općenitog problema iz neke specifične domene, no najčešće su problemi složeniji te jedna paradigma nije prikladna za rješavanje baš svih dijelova problema. U Tablici 1 su navedene prednosti i mane određenih programskih paradigmi te na temelju nekih od njih razvojni inženjer može procijeniti koja paradigma je prikladna za rješavanje kakvih zadataka. Od prethodno spomenutih paradigmi više razine, ovdje je uvršteno i logičko programiranje s ograničenjima, koje je zapravo proširenje logičke paradigme, s obzirom da će i ona biti obrađivana u daljnjim poglavljima.

Tablica 1: Usporedba nekih programskih paradigmi

Paradigma	Prednosti	Mane
Proceduralna	Mogućnost dobro organiziranog i strukturiranog kôda kao i laka pretraga po njemu	Teška za razumijevanje, otežano naknadno održavanje, sekvencijalna, prisutne prateće pojave
Objektno-orijentirana	Dobro strukturiranje i modularnost, nasljeđivanje, komunikacija između procesa	Neprikladna za rješavanje kompleksnih algoritamskih problema
Logička	Dostupnost pravila zaključivanja, jednostavna notacija i sintaksa, upoznavanje s formalnom logikom	Neprikladna za prikaz proceduralnog znanja, relativno loša organizacija kôda
Logičko programiranje s ograničenjima	Prikladna za rad s aritmetičkim izrazima, prikladna za rješavanje problema raspoređivanja	Nemogućnost brzog rješavanja naknadno promijenjenog problema raspoređivanja
Funkcionalna	Dobar pregled toka podataka, modularnost od vrha prema nižoj razini, nema prisutnih pratećih pojava	Mnogo linija kôda, mnogo funkcija kao i njihovih parametara
Pogonjena događajima	Prikladna kada tijek izvođenja programa nije sekvencijalan, obrada korisničkih događaja	Relativno loše performanse programa, nepotrebno se otežava rješavanje problema sa sekvencijalnim tijekom

(Prema: Böck, 1991 i Barták, 2009)

Baš zbog toga je ponekad prikladno kombiniranje elemenata i pogodnosti više različitih paradigmi kako bi se razvilo traženo programsko rješenje sa što boljim odnosom vremena potrebnog za razvoj, lakoće održavanja i proširivanja, ali i brzine razumijevanja kôda od strane novog programera.

U nastavku slijedi razrada svake od navedenih programskih paradigmi kako bi se u sljedećem poglavlju mogli prepoznati potreba i elementi prilikom povezivanja s logičkom paradigmom koja je predmet razmatranja u ovome radu. Samim time, detaljnije u odnosu na ostale ću zbog toga obraditi logičku paradigmu, kao i objektno-orijentiranu paradigmu budući da je ona trenutno najpopularnija paradigma te su njeni elementi ugrađeni u mnoge programske jezike. (TIOBE Software BV, 2018; Carbonnelle, 2018)

2.1. **Proceduralna paradigma**

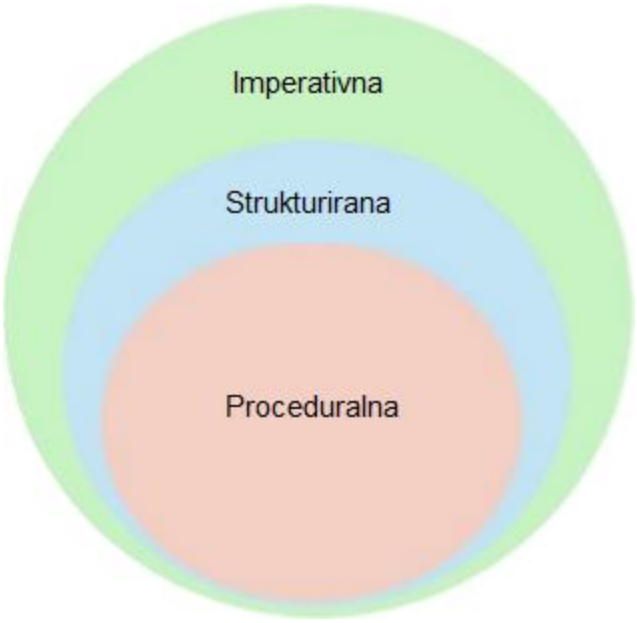
Kao što je već spomenuto, proceduralna paradigma zapravo pripada imperativnoj paradigmi što znači da se proces rješavanja problema programskim jezicima ove paradigme bazira na navođenju instrukcija koje se izvode potpuno sekvencijalno te su pri tome vezane uz promjenu stanja memorije. Razlog tome jest što se imperativna programska paradigma temelji

se na Von Neumannovoj arhitekturi računala koja je prevladavajuća arhitektura računalnog sklopovlja (engl. *hardware*) gdje se podaci prenose između procesora i memorije. Svi programi (neovisno o paradigmi koja je korištena prilikom njihovog pisanja) kad se izvršavaju prosljeđuju procesoru instrukcije na interpretaciju upravo na način kao da je program zapravo pisan jezikom imperativne paradigme. Imperativne programe karakterizira promjena stanja što predstavlja da je za imenovani dio memorije (tzv. varijablu) u jednom trenutku u programu vezana jedna vrijednost, dok kasnije za njega može biti vezana neka sasvim druga vrijednost. (Vujošević-Janičić i Tošić, 2008, str. 69)

Proceduralna paradigma ujedno pripada strukturiranoj paradigmi programiranja koja pak predstavlja kontrolne strukture za provedbu grananja (engl. *selections*) i ponavljanja (engl. *iterations*). Konceptima te paradigme se tako omogućava da izvorni kôd programa bude programeru čitljiviji i jednostavniji za razumijevanje. Tako se odnos skupova pogodnosti koje nudi proceduralna, strukturirana i imperativna paradigma može vizualno prikazati kao na Slici 1. (Šribar i Motik, 2010)

Ideja proceduralne paradigme je da se dijelovi programskog koda programa (odnosno niz njegovih naredbi) mogu imenovati kako bi se kasnije mogli iskoristiti navođenjem/pozivom definiranog naziva prilikom sljedeće potrebe za njihovim izvršavanjem. Slijed instrukcija/naredbi u imperativnim programskim jezicima se naziva potprogram, procedura ili funkcija. Njihovom postojanošću u programskim jezicima je ujedno moguće rješavati problem funkcionalnom dekompozicijom, odnosno pristupati rješavanju problema s najviše i najapstraktnije razine prema nižim i konkretnijim, što je ostvareno na način da procedure viših razina pozivaju niz procedura nižih razina koje pak izvršavaju procedure još nižih razina sve dok procedure najnižih razina ne vrše isključivo konkretne operacije poput aritmetičko-logičkih ili dohvata/pohrane podataka iz/u memoriju. Ovakav analitički pristup se zove pristup rješavanju problema odozgo prema dolje (engl. *top-down approach*) kojim razvojni inženjer za početak definira korake za rješavanje problema nakon čega za svaki definirani korak definira korake niže razine i tako sve dok mu koraci koje želi primijeniti ne postanu dostupni operatorima i ugrađenim procedurama korištenog programskog jezika. (Šribar i Motik, 2010)

Procedure također mogu biti iskoristive za grupiranje programskog koda koji je veoma sličan, ali ne i identičan. Naime, promjenjive vrijednosti koje se uspoređuju ili koje sudjeluju u aritmetičko-logičkim operacijama se mogu zamijeniti referencama na memorijsku lokaciju na koju se pohranjuje određena vrijednost prilikom svakog poziva te procedure. Navedeno se vrši navođenjem vrijednostī zajedno prilikom navođenja naziva procedure koju se želi poslati, odnosno vrijednosti se prosljeđuju pozivajućoj proceduri putem tzv. parametara ili argumenata.



Slika 1. Odnos proceduralne, strukturirane i imperativne paradigme prikazan Vennovim dijagramom (Prema: Pecos Martínez, 2014)

Proceduralnu paradigma je podržana od brojnih programskih jezika, od FORTRAN-a koji još od 1958. omogućava izradu korisnički definiranih procedura, Pascala, jezika C pa i kod aktualnijih poput Pythona i C++. Primjerice, Pascal je razvio Niklaus Wirth kao jednostavan i učinkovit programski jezik koji je namijenjen poticanju dobre programske prakse pomoću strukturiranog programiranja. Navedeni jezik je osmišljen kako bi se studenti lakše prilagodili i jednostavnije razumjeli tehnike dobrog strukturiranja. S druge strane, C programski jezik osmislio je Dennis Ritchie, a primarna upotreba jezika C jest i dan danas implementacija operativnih sustava, ali isto tako je namijenjen za razvoj različitih vrsta aplikacija, posebice onih kod kojih su performanse te ušteda memorije i vremena izvršavanja od kritične važnosti. Python i C++ zapravo podržavaju i objektno-orijentiranu paradigmu te njihovi brojni moduli/biblioteke bazirani na njenim konceptima, no programer ipak može, ako to uistinu želi, koristiti isključivo elemente proceduralne paradigme prilikom razvoja programa. (Vujošević-Janičić i Tošić, 2008, str. 69)

2.2. **Objektno-orijentirana paradigma**

U ovom poglavlju objasnit ću važnije aspekte objektno-orijentirane paradigme. Ova paradigma svoje korijene vuče još iz ranih 60-ih i 70-ih godina prošlog stoljeća iz Simule i Smalltalka. (Gabbrielli i Martini, 2010, str. 289)

Objektno-orijentirana paradigma je također dio strukturirane paradigme pa tako i imperativne, a ujedno sadrži i sve elemente proceduralne paradigme koja je proširena

konceptima klasa i njihovih objekata zbog čega glavnu razliku čini filozofija razvoja programa. Naime, proceduralnim pristupom se kreiraju i koriste procedure kako bi se radilo s podacima, dok su kod objektno-orijentirane paradigme strukture podataka i njihove pripadajuće procedure međusobno vezane te je ideja da se kod rješavanja problema prepoznaju objekti kao i njihovi odnosi sa drugima te definiraju ponašanja za njihovo međudjelovanje. (Li i Henry, 1993, str. 53)

Uz Simulu i Smalltalk vežu se sljedeće činjenice. Smalltalk je prvi jezik u koji je bio ugrađen koncept objekata i pripadajućih klasa nalik na one iz suvremenim OOP jezicima te se je dalje nastavio proširivati u tom smjeru. Simula je programski jezik koji je izvorno dizajniran samo za potrebe simulacije. Simulu je osmislio O. J. Dahl i K. Nygraard u norveškom računalnom centru u Oslu. Simula je bila prvi programski jezik sa klasama, objektima, dinamičkom pretragom i nasljeđivanjem. Pojava Simule je tako potaknula Xerox Palo Alto istraživački centar (PARC) na razvoj Smalltalka, a kasnije i Bjarnea Stroustrupa na razvoj programskog jezika C++. Simula je prema tome pridonijela velikom razvitku objektno-orijentirane paradigme. (Mitchell, 2002, str. 300)

Neki od temeljnih koncepata objektno-orijentirane paradigme su:

- Enkapsulacija i apstrakcija
- Podtipovi – odnos kompatibilnosti koji se temelji na funkcionalnosti nekog objekta
- Nasljeđivanje – mogućnost pomnog korištenja neke metode koja se prethodno može definirati za neki drugi objekt ili za neku drugu klasu
- Dinamički odabir metoda
- Polimorfizam

U nastavku ću ukratko objasniti svaki pojedinačno. (Gabbrielli i Martini, 2010, str. 289)

2.2.1. Nasljeđivanje i podtipovi

Nasljeđivanje možemo definirati kao način izrade novih klasa i objekta koristeći već unaprijed definirane klase. Klasu koja nasljeđuje neku drugu klasu nazivamo izvedenom klasom, dok klasa koja je naslijeđena naziva se baznom klasom. Jedna od važnijih prednost nasljeđivanja je upravno u tome što paradigma objektno-orijentiranog programiranja služi za već unaprijed napisan kôd, a isto tako nam se smanjuje kompleksnost cijele aplikacije. (Vuk, 2016a)

Nasljeđivanje dopušta pomnu upotrebu kôdova u kontekstu koji se može proširiti. Specijalizacijom neke klase (tj. kreiranjem izvedene klase iz nje) se mogu iskoristiti metode i atributi natklase kako bi se proširila funkcionalnost natklase ili pristupilo nekim skrivenim podacima. (Gabbrielli i Martini, 2010, str. 290)

Specijalnu vrstu klasa čine sučelja (engl. *interfaces*) koja se u zadnjih desetak godina izrazito koriste radi postizanja modularnosti programskih rješenja i njihove daljnje lakše proširivosti. Sučelje je zapravo vrsta sažetka klase koja uključuje samo nazive i tipove metoda pri čemu nije uključena njihova implementacija. Kada se želi kreirati klasa nekog sučelja, tada se implementira željeno sučelje na način da se definiraju implementacije svih deklariranih metoda, odnosno da se specificira tijelo metoda koje sučelje propisuje.

Većina programskih jezika objektno-orijentirane paradigme omogućava da jedna klasa implementira više sučelja, no rijetko koji nudi sposobnost višestrukog nasljeđivanja, odnosno da neka klasa bude potklasa dvaju ili više natklasa. Programski jezici poput Pythona i C++-a nude navedenu sposobnost, dok ju s druge strane Java i jezici .NET platforme ne nude jer zajednice koje stoje iza njih smatraju da višestruko nasljeđivanje čini programski kôd teže razumljivim. (Gabbrielli i Martini, 2010, str. 290)

2.2.2. Enkapsulacija i apstrakcija

Uz objektno-orijentirano programiranje usko se veže i enkapsulacija. Enkapsulaciju možemo definirati kao proces vezanja srodnih pojmova zaštite atributa i funkcija u klasu. Enkapsulacija nam omogućava da podaci (atributi) budu nedostupni direktno preko samog objekta. Pristup atributima se postiže pozivom metoda koje onda sa njima rade, a često programski jezici nude specijalne metode za tu svrhu, tj. *set* metode za postavljanje vrijednosti atributa te *get* za njihovo dohvaćanje. Da bismo primijenili pravila enkapsulacije potrebno nam je sljedeće:

- Da atributi u klasi budu privatni
- Da postoje definirane javno definirane metode koje rade s njima

Enkapsulacija nam pruža prvi stupanj sigurnosti u aplikacijama, ali isto tako na svim višim razinama gdje mi tu sigurnost možemo unaprijediti. (Vuk, 2016b)

Apstrakcija se s druge strane pak usko veže uz spomenuto nasljeđivanje i definiranje podtipova. Naime, nerijetko postoji potreba da se onemogući stvaranje objekata klase od koje postoje potklase, tj. njene specijalizacije. Primjerice, želi se spriječiti instanciranje klase *Životinja* ako je ta klasa nasljediva, odnosno ako se iz nje mogu stvoriti neapstrakne (tzv. konkretne) klase poput klasa *Mačka*, *Pas*, *Medvjed* i sličnih iz kojih se onda mogu stvoriti objekti.

Apstraktnima se često mogu učiniti i metode apstraktne klase kako bi se od implementatora potklasa zahtijevalo da redefinira takve metode tako da proširi bazično ponašanje. (Gabbrielli i Martini, 2010, str. 294-297)

2.2.3. Dinamički odabir metoda

Dinamički odabir metoda je najvažniji temeljni koncept objektno-orijentirane paradigme. On nam omogućuje da konkretne klase i apstrakcije budu u međuodnosu. Mehanizam koji se koristi je vrlo jednostavan. Dinamičkim odabirom se omogućava da se nad objektom poznate apstraktne natklase, ali nepoznate konkretne potklase, vrši poziv neke od metoda koja je definirana (ne nužno i implementirana) u natklasi. To je moguće zato što je objekt zasigurno iz neke potklase apstraktne natklase čime on sadrži metodu koja se nad njim poziva. Prevoditelj (*engl. compiler*) neće odlučiti kojeg će točnog podtipa biti objekt ili pak koja će implementacija metode biti pozvana. (Gabbrielli i Martini, 2010, str. 289)

2.2.4. Polimorfizam

Polimorfizam u objektno-orijentiranoj paradigmi možemo definirati kao preobrazbu jednog objekta u više različitih oblika. Sama riječ polimorfizam znači sposobnost poprimanja više različitih oblika ili značenja. Osnovna ideja polimorfizma je da osnovnu ključnu klasu, koju nasljeđuje više njezinih klasa, mogu izvršiti iste funkcije i to svaka sa svojim određenim zahtjevima. (Vuk, 2016c)

U nastavku ću ukratko objasniti polimorfizam karakterističan za programski jezik Java. Postoje dva različita tipa polimorfizma, a to su statički i dinamički polimorfizam.

2.2.4.1. Statički polimorfizam

Java je, kao i ostali objektno-orijentirani jezici, fokusirana na objekte i omogućuje da implementiramo više metoda unutar jedne klase koje koriste jednake nazive, ali različiti skup parametara. To možemo nazvati kao preopterećenjem metoda (engl. *method-overloading*) ili statički oblik polimorfizma. Oblici se mogu razlikovati po sljedeća tri kriterija:

- Moraju imati različiti broj parametara
- Vrsta parametara mora biti različita (primjerice jedna metoda može kao argument prihvaćati znakovni niz, a druga broj)
- Parametri se mogu očekivati u nekom drugom redoslijedu

Prema tome, svaka metoda ima različito djelovanje i to dopušta prevoditelju da identificira koja metoda mora biti pozvana te da ga pritom veže na metodu poziva. Takav pristup naziva se statički polimorfizam. (Janssen, 2017)

2.2.4.2. Dinamički polimorfizam

Ovaj oblik polimorfizma ne dopušta prevoditelju da utvrdi koja bi to izvršna metoda bila. Takav odnos ima veze sa nasljeđivanjem i unutar hijerarhije nasljeđivanja potklasa može

nadjačati natklasu. Primjena takvog načina pomaže razvojnim inženjerima da potklasom prilagode ili potpuno zamjene ponašanje metode. Obje metode stvorene su od potklase i natklase. One dijele isto ime i parametre, ali daju različite funkcionalnosti. Polimorfizam opisuje koncept kako različita klasa može biti korištena za isto sučelje. (Janssen, 2017)

2.3. **Logička paradigma**

Logička paradigma temelji se na predikatnom računu prvog reda. Ovaj programski stil naglašava deklarativni opis problema. Programska logika sastoji se od:

- Aksioma - definiranje činjenica o objektima
- Pravila - definiranje načina zaključivanja novih činjenica
- Izjava - definiranje teorema

Logičko programiranje karakterizira programiranje s relacijama i zaključivanjem. Programer je zaslužan za određivanje osnovnih logičkih odnosa i ne navodi način na koji se primjenjuju pravila i zaključivanja. Logički jezici su obično zahtjevniji u računalnim resursima od proceduralnih i objektno-orijentiranih jezika. Logička paradigma sadržana je u jezicima kao što su Prolog (1970.) i Gödel (1994.) dok su pak Mercury (1995.) i Curry (1997.) multiparadigmatski programski jezici koji spajaju elemente funkcionalne paradigme poput programiranja višeg reda (engl. *higher-order programming*) i logičke programske paradigme (nedeterminističko programiranje i unifikacija). (Vujošević-Janičić i Tošić, 2008, str. 75)

Danas postoje mnoge implementirane verzije Prologa pri čemu mnoge od njih proširuju jezik s karakteristikama drugih programskih paradigmi poput objektno-orijentiranog (metode klasa, nasljeđivanje i enkapsulacija), pogonjenog-događajima (obrada događaja, posebice korisnikovih akcija) i paradigme programiranja s ograničenjima. Paradigma logičkog programiranja je posebice srasla uz paradigmu programiranja uz ograničenja zbog čega je nastala sasvim nova paradigma naziva logičko programiranje s ograničenjima ili ograničeno logičko programiranje (*engl. constraint logic programming*). (Ray, *s.a*)

Poznata jednadžba Roberta Kowalskog je sljedeća: *Algoritam = Logika + Kontrola*. Jednadžbu prema tome možemo podijeliti na tri dijela. S jedne strane imamo logiku koja nam označava ono što mora biti učinjeno, dok na drugoj strani imamo aspekte koji su vezani uz kontrolu. (Coelho i Cotta, 1988, str. 5-6)

Kako se logičko programiranje bazira na računu predikata ili logici prvog reda, valjda spomenute njegove elemente, a to su:

- Abeceda
- Uvjeti definirani preko abecede

- Dobro formirana formula definirana preko abecede

Abeceda se sastoji od simbola i dijeli se u dva podskupa: skup logičkih simbola i skup nelogičkih simbola koji su specifični za određenu interesnu domenu. Skup logičkih simbola tako sadrži sljedeće elemente:

- Logičke veze: ¬ negacija, ∧ konjunkcija, ∨ disjunkcija, ⇒ implikacija, ⇔ ekvivalencija
- Predložene konstante koje su istinite ili lažne

(Gabbrielli, Martini, 2010, str. 374)

Sintaksa logike prvog reda je opisana njenom abecedom koju Čubrilo definira na sljedeći način:

Abeceda $\mathcal{A}$ jezika $\mathcal{L}$(RP) računa predikata je unija sljedećih skupova simbola:

- $A_1 = \{c_i : i \in I \subseteq \mathbb{N}\}$ – najviše prebrojiv skup konstanti, gdje je $\mathbb{N}$ skup svih prirodnih brojeva
- $A_2 = \{f_I : I \in L \subseteq \mathbb{N}\}$ – najviše prebrojiv skup funkcija svih konačnih kratnosti
- $A_3 = \{P_k : k \in K \subseteq \mathbb{N}\}$ – najviše prebrojiv skup predikata svih konačnih kratnosti
- $A_4 = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ – skup logičkih veznika
- $A_5 = \{\forall, \exists\}$ – skup kvantifikatora „za svaki“ i „postoji neki“
- $A_6 = \{(,)\}$ – skup lijeva i desna zagrada

(Čubrilo, 1989, str. 72)

2.3.1. Logičko programiranje s ograničenjima

Logičko programiranje s ograničenjima jest paradigma koja je nastala kombiniranjem elemenata logičke paradigme i paradigme programiranja s ograničenjima (engl. *constraint programming*). Ova programska paradigma je, kao što naziv govori, izvedena iz logičke paradigme programiranja, a korisna je za rješavanje problema zadovoljenja ograničenja koji su predstavljeni kao skup varijabli i ograničenja između njih. Korištenjem ove paradigme je cilj pronaći način dodjele vrijednosti varijablama tako da nijedno ograničenje ne bude prekršeno. (SWI-Prolog Manual: Constraint Logic Programming, *s.a.*)

Ova programska paradigma je posebice prikladna za rješavanje problemā iz discipline operacijskih istraživanja, kao što su optimizacijski problemi poput analize kritičnog puta i problema raspodjele resursa u poslovnom planiranju, problema minimizacije prostora i maksimizacije performansi kod projektiranja mikročipova u elektrotehnici, problema trgovačkog putnika i transportnog problema u logistici, itd. (Jaffar i Lassez, 1987, str. 111-119)

Neke od domena na kojima se ograničeno logičko programiranje može iskoristiti za zaključivanje su skupovi poput cijelih, racionalnih i realnih brojeva, logičkim varijablama (tzv. *boolean*) i konačnim domenama (engl. *finite domains*). (SWI-Prolog Manual: Constraint Logic Programming, *s.a.*)

2.4. **Funkcionalna paradigma**

Temeljna karakteristika jezika u ovoj paradigmi je upravo to da ono ne posjeduje koncept pamćenja. Glavni konstruktor mijenja vrijednost sadržanu u varijabli, ali ne mijenja povezanost između naziva varijabli i lokacija. Sve je počelo od von Neumannovoga modela nazvanog po njemačko-američkom matematičaru koji je još davne 1940. godine koristio Turingov stroj za svoj tzv. „elektronički računalni uređaj/instrument“. Programski jezici koji predstavljaju ovaj model zovu se funkcionalni jezici, a paradigma koja proizlazi iz ovoga modela naziva se funkcionalna paradigma. (Gabbrielli i Martini, 2010, str. 334)

Funkcionalna programska paradigma temelji se na teoriji matematičkih funkcija, konkretno na lambda računu. Ona na neki način pomaže programeru da razmišlja na jedan zaseban način, tj. na višoj razini apstrakcije ili na tzv. matematičkoj prirodi problema. Funkcionalni programski jezici sastoje se od tri glavne skupine komponenti, a to su:

- Podatkovni objekti (engl. *Data-objects*) – poput lista/nizova
- Ugrađene funkcije (engl. *Built-in functions*) – koriste se za manipulaciju podatkovnim objektima
- Funkcionalni obrasci (engl. *Functional forms*) – nazivaju se još i funkcije višeg ranga koje se koriste za izgradnju novih funkcija

Samo izvođenje funkcionalnih programa sastoji se od dva temeljna mehanizma: povezivanja (*engl. Binding*) i primjene *(engl. Application*). *Binding* mehanizam se koristi za povezivanje vrijednosti s imenima, dok se *Application* rabi za dohvaćanje/pozivanje prethodno povezanih vrijednosti. Podaci i funkcije mogu se koristiti kao vrijednosti. Funkcionalni programski jezici nazivaju se još i aplikacijski iz razloga jer se funkcije pozivaju na njihove argumente. Kod obrade ove paradigme valja spomenute i programske jezike koji podržavaju ovu paradigmu, a neki od njih koji u potpunosti implementiraju njene elemente su ML (1973.), Scheme (1975.), Miranda (1982.) i Haskell (1987.). Posljednjih nekoliko godina su neki elementi ove paradigme (poput lambda izraza i monada) podržani u popularnim programskim jezicima poput C++, Pythona, JavaScripta i Jave. Mnogi profesori na fakultetima vjeruju kako je upravo funkcionalnu paradigmu vrijedno učiti iz razloga jer se smatra kako se time studente i učenike može bolje pripremiti za projekte i analizu algoritama. Funkcionalna paradigma daje nam da se pobliže fokusiramo na skup određenih problema sa kojim smo suočeni. Vjeruje se

kako dobre osnove programiranja leže na dobro naučenom znanju o prije spomenutom lambda računu. Jedini problem koji bi djeci moglo predstavljati učenje o ovoj paradigmi bio bi teško ispravno razumijevanje mehanizama algoritama zamjene koji su pak osnovni koncept za ovu paradigmu. (Vujošević-Janičić i Tošić, 2008, str. 73-74)

2.5. **Programiranje pogonjeno događajima**

Ova paradigma je prikladna za problema kod kojih tijeka izvođenja programa rješenja nije sekvencijalan i strogo određen, već postoje događaji i akcije koje trebaju nastupiti prilikom njihovog ostvarenja. Posebice je ova paradigma od koristi kada su događaji koje je potrebno obraditi klik na tipku, otvaranje padajućeg izbornika, obrada poruka (međuprocesnih ili mrežnih), stvaranje i uništavanje objekta i slično. S obzirom na spomenute primjene ove paradigme, da se zaključiti da je usko vezana uz izradu aplikacija s grafičkim korisničkim sučeljem, no može poslužiti i kod izrade programa s naredbenim sučeljem kod izrade mrežnog-poslužitelja (primjerice, za obradu pristiglih zahtjeva i/ili odaslanih odgovora) ili pak programa koji obrađuju podatke s neke sabirnice (npr. sa serijskog porta na koji je spojen mikrokontroler sa senzorima i aktuatorima). (Hinze, 2010, str. 287)

Ova programska paradigma je izrazito srasla uz objektno-orijentiranu paradigmu te većinom programski jezici koji podržavaju objektno-orijentiranu, nude elemente programiranja pogonjenog događajima – posebice ako jezik omogućava izradu aplikacija s grafičkim sučeljem ili se bazira na obradi poruka. Razlog zbog koje je prisutna ta njihova povezanost jest što se uz objekte veže pojam njihovog životnog vijeka, odnosno da se olakša praćenje događaja nad objektima (koji najčešće predstavljaju relevantne instance modelā iz stvarnog svijeta) poput njihovog stvaranja, izmjene, uništavanja ili općenito rada s njima. Temelji implementacije ove paradigme (konkretno mislim na osluškivanje događaja i izvršavanje pripadajuće akcije) se baziraju na *observer* ili *publish-subscribe* uzorku dizajna kod kojeg se definira postoji li interes za osluškivanjem nekog događaja te u tom slučaju se prosljeđuje pokazivač na funkciju obrađivaču događaja (engl. *event-handler*). Kod programskih jezika objektno-orijentirane paradigme koji ne podržavaju rad s pokazivačima (poput Jave i JavaScripta) se pak obrađivaču događaja prosljeđuje implementacija traženog sučelja pri čemu se definira tijelo funkcije (ili više njih) koje će biti izvršeno kada nastupi pripadajući događaj. (Yeager, 2014, str. 130)

Funkcije koje se izvršavaju kad nastupi osluškivani događaj se često nazivaju i funkcijama povratnog poziva (engl. *callback functions*), nisu povratnog tipa, a često mogu imati parametre iz kojih se može izvući informacija o pritisnutoj tipki s tipkovnice koja je uzrokovala nastanak događaja, o prozoru koji je zatvoren ili promijenjen (primjerice, povećan, smanjen,

minimiziran i slično), adresa odredišta s kojeg su stigli podaci preko mreže kao i sami pristigli podaci.

Najznačajniji programski jezici koji omogućavaju razvoj programa baziran na ovoj paradigmi su C++, C#, Java, JavaScript i Python, pri čemu su većinom su elementi te paradigme iskoristivi prilikom razvoja aplikacija s grafičkim sučeljem. Još kod Smalltalka su se počeli primjenjivati elementi ove paradigme za obradu pristiglih poruka kod komunikacije objekata, a spomenuti jezici su tako od njega naslijedili i te elemente uz samu objektno-orijentiranu paradigmu. Postoje specijalizirani radni okviri (engl. *frameworks*) i moduli koji omogućavaju osluškivanje i obradu korisničkih događaja nad korisničkim sučeljem aplikacije pa tako Qt okvir nudi podršku za sve spomenute jezike (osim Smalltalka) za izradu više-platformskih aplikacija, a navedeno ujedno nudi i Swing okvir za Javu. S druge strane, Microsoftovim .NET okvirom se korištenjem C++, C# i Pythona mogu izrađivati aplikacije koje nude mogućnost pretplate na daleko veći broj događaja vezanih uz različite grafičke kontrole, ali i na ostale elemente poput mrežnih utičnica (engl. *network sockets*) i serijskih portova. Međutim, mana tog radnog okvira je što takve aplikacije mogu samo raditi na distribucijama Windows operacijskog sustava s .NET platformom. Obrada događaja nastalih primitkom poruka na osluškivanu mrežnu utičnicu je moguća i kod Jave pri razvoju za Java platformu namijenjenu za izvršavanje poslužiteljskih aplikacija (engl. *Java Platform, Enterprise Edition – Java EE*). (Yeager, 2014, str. 200)

3. Načini povezivanja logičke paradigme s drugim paradigmama

Kao što je već prethodno spomenuto, logičko programiranje možemo definirati kao računalnu programsku paradigmu u kojoj probleme rješavamo programskim naredbama koje su zapravo činjenice i pravila iz sustava formalne logike. Pravila su definirana kao logičke klauzule sastavljene od tijela i glave. Najpoznatiji programski jezik logičke paradigme je Prolog koji je ujedno jedan od najpopularnijih tradicionalnih jezika u razvoju umjetne inteligencije. Iako je većina jezika logičke paradigme izrazito vezana uz deklarativnu paradigmu, Prolog osim nje podržava i imperativnu te proceduralnu paradigmu. (Computer Hope, 2017)

Prolog kao logički programski jezik uključuje skup osnovnih mehanizama kao što su strukturiranje podataka temeljenih na stablu i njegovo automatsko praćenje. Prikladan je za probleme koji uključuju poslovno strukturirane objekte i odnose između njih. Možemo reći kako je Prolog kratica za logičko programiranje, odnosno ideju koja se pojavila još u ranim sedamdesetim godinama prošlog stoljeća. Kako i sama riječ govori, u ovoj paradigmi koristi se logika kao temelj programiranja. Ovom idejom najprije su se počeli koristiti Robert Kowalski iz Edinburgha koji se bavio teoretskim dijelom, te njegov kolega Maarten Van Ewden na eksperimentalnom dijelu i Alian Colmerauer koji je bio zadužen za samu implementaciju. Tek je 1996. godine objavljen službeni ISO standard za Prolog. (Bratko, 2001, str. 30)

U ovom poglavlju se razrađuje tema ovog rada, a kao što sam u uvodu već napomenuo, od programskih jezika logičke paradigme će fokus biti na jeziku Prolog i to ponajprije implementacije SWI. Razlog odabira programskog jezika Prolog jest njegova popularnost u odnosu na ostale jezike logičke paradigme, dok je razlog odabira njegove implementacije SWI to što je navedena bogata brojnim modulima, pri čemu su neki od njih vezani uz povezivanje s drugim programskih paradigmama. (Wielemaker i Costa, 2011, str. 70)

Unatoč tome što rezultati analize mjerenja performansi različitih implementacija Prologa pokazuju da su performanse implementacije SWI relativno loše, za razradu teme ovog rada one nisu relevantne. (Demoen i Nguyen, 2001)

3.1. Ugrađeni elementi drugih paradigmi

SWI implementacija Prologa je izrazito bogata modulima koji proširuju bazu samog jezika s dodatnim funkcionalnostima čime se programeru omogućuje lakši i brži razvoj pouzdanih aplikacija. Tako neki moduli nude ugrađene predikate (poput predikata za rad s listama, kodiranje i dekodiranje podataka u i iz JSON, XML, CSV formata), dok drugi proširuju

sâm jezik na višu razinu apstrakcije, čime programer može rješavati određeni problem korištenjem prikladne paradigme bez potrebe da mijenja jezik.

Kao što je prethodno spomenuto, prema ISO standardu Prolog osim logičke paradigme (koja je podvrsta podržane deklarativne paradigme) podržava i elemente proceduralne paradigme (koja je pak podvrsta dijelom podržane imperativne paradigme), dok se elementi ostalih paradigmi mogu dobiti na korištenje uključivanjem dodatnih pripadajućih modula.

3.1.1. Proceduralna paradigma

Elementi imperativne paradigme su dostupni programeru na korištenje kada bi realizacija cijelog rješenja logičkom paradigmom bila nezgrapna te uvelike otežala razumijevanje i održavanje rješenja. Posebice do izražaja dolazi mogućnost dekomponiranja složenog cilja na podciljeve čime se taj složeni cilj može iskazati predikatom čije je tijelo konjunkcija podciljeva koji trebaju biti zadovoljeni da bi cilj bio istinit. Tom sposobnošću jezika se omogućava izrada modularnih/iskoristivih dijelova programskih rješenja, a to je ujedno primjer kako je podržana proceduralna paradigma koja se u programskim jezicima dominantno-deklarativne paradigme manifestira kroz postojanost funkcija i procedura.

Mala, ali značajna razlika jest da se kod ispitivanja cilja u slučaju nezadovoljstva nekog njegovog podcilja ne provjeravaju daljnji podciljevi s obzirom da oni više ne mogu utjecati da ispitivani cilj bude zadovoljen. Ovakvo ponašanje je prisutno kod većine programskih jezika, ali samo u evaluaciji logičkih izraza, pri čemu se koristi brza evaluacija ili tzv. evaluacija kratkog spoja (engl. *short circuit evaluation*). Kod Prologa se tako brza evaluacija primjenjuje i kod izvođenja predikata s obzirom da je on jezik logičke paradigme, zasniva se na formalnoj logici te tijela predikata ciljeva predstavljaju podciljeve koji svi zapravo trebaju biti zadovoljeni. Predikat se tako u Prologu definira na sljedeći način:

$$\texttt{Cilj :- Podcilj}_1\texttt{, …, Podcilj}_n\texttt{.}$$

Korištenjem formalne logike bi se taj predikat mogao zapisati

$$\{\texttt{Podcilj}_1\texttt{, …, Podcilj}_n\} \vDash \texttt{Cilj}$$

gdje znak „⊨“ predstavlja logičku posljedičnost, dok podciljevi unutar vitičastih zagrada označavaju skup podciljeva koji trebaju vrijediti. Navedeno bi se pak u računu sudova moglo izraziti

$$\texttt{Podcilj}_1 \wedge \ldots \wedge \texttt{Podcilj}_n \Rightarrow \texttt{Cilj}$$

što je ekvivalentno na standardnom jeziku iskazu:

$$\texttt{Podcilj}_1\texttt{ i … i Podcilj}_n\texttt{ impliciraju Cilj.}$$

Kako se proceduralna paradigma bazira na strukturiranoj i imperativnoj paradigmi, navodim još kratki primjer usporedbe realizacije grananja korištenjem elemenata imperativne paradigme i čiste deklarativne paradigme. Neke od ostalih značajki imperativne paradigme osim predikata u programskom jeziku Prolog su postojanost operatora ->/2 (lokalni rez) koji se u kombinaciji s operatorom *;/2* (logičko *ili*) može iskoristiti za ostvarivanje selekcije/grananja, kao i postojanost varijabli za pohranu podataka, iako su specifične u odnosu na iste iz većine drugih programskih jezika po tome što ih nije moguće redefinirati - moguća je samo inicijalizacija zbog toga što ne postoji klasičan operator dodjele vrijednosti, već operator unifikacije koji je specifičan za logičku paradigmu. Tako bi primjer implementacije predikata za usporedbu dva broja i dohvat atoma koji predstavlja njihov međusobni odnos napisanog korištenjem logičke paradigme bio sljedeći:

```
odnos(A, A, '=') :- !.
odnos(A, B, '<') :- A > B, !.
odnos(_, _, '>').
```

Ako bi se pak koristili elementi imperativne paradigme, tada bi se taj isti predikat mogao definirati na sljedeći način:

```
odnos(A, B, Rezultat) :-
    A > B ->
        Rezultat = '>'
    ; A < B ->
        Rezultat = '<'
    ;
        Rezultat = '='
.
```

Čitatelji koji su upoznati s imperativnim programskim jezicima poput Pythona, Jave, C-a i njegovih izvedenica laku prepoznaju kako djeluje drugi predikat s obzirom da bi u takvim jezicima rješenje bilo gotovo identično, osim što bi se zbog drugačijih sintaksnih pravila koristili drugačiji konstrukti za ostvarivanje selekcije, definiranje funkcije, terminatori za odvajanje naredbi i slično.

3.1.2. Objektno-orijentirana paradigma

Kako bi se Prolog i logičko programiranje što bolje približili zajednici razvojnih inženjera, potrebno joj je ponuditi elemente i pomagala na koje su već do sada navikli. S obzirom da po većini rang ljestvica popularnosti programskih jezika prevladavaju jezici koji podržavaju objektno-orijentiranu paradigmu programiranja, podržavanje te paradigme je od izrazite važnosti kako bi Prolog bio prihvaćen od strane šire publike. (TIOBE Index, 2018; Carbonnelle, 2018)

Mogućnost definiranja strukture objekata, postizanje modularnosti i ponovne iskoristivosti, izvođenje specijalizacija klasa korištenjem nasljeđivanja te lakše definiranje

međuovisnosti i ponašanja između različitih objekata su također neki od primjera kako objektno-orijentirana paradigma može biti programeru od koristi. (Böck, 1991, str. 629)

Primjeri nekih od modula koji omogućavaju da se u Prologu implementacije SWI koristi objektno-orijentirana paradigma su Logtalk i XPCE, pri čemu je Logtalk dostupan kao proširenje i za mnoge druge dijalekte Prologa te se nekad spominje kao zaseban programski jezik, dok je XPCE značajan za razvoj aplikacija s grafičkim sučeljem koje su portabilne na operacijske sustave s grafičkim korisničkim sučeljem i za koje je ovaj dijalekt Prologa podržan. XPCE pri tome podržava paradigmu pogonjenu događajima (*event-driven programming*), čime se omogućava da izrađene grafičke aplikacije mogu obrađivati korisničke događaje poput klika i pokreta mišem, otvaranje, zatvaranje i promjene nad grafičkim obrascem i njegovim elementima te ostale.

Vrijedi još spomenuti Flora-2 programski jezik koji se oslanja na interpreter Prologa implementacije XSB, a ovaj jezik je specifičan po tome što se bazira na F-logici, odnosno logici s okvirima (engl. *frame logic*) koja je uz logiku prvog reda i deskriptivnu logiku jedna od formalizama za prikaz ontologija te vršenje logičkog zaključivanja.

3.1.2.1. **Logtalk**

Kao što je već spomenuto, Logtalk je proširenje Prologa te mu je najveća prednost što je službeno dostupan za čak desetak implementacija poput SWI, XSB, YAP, B-Prolog i brojne druge. Željena implementacija Prologa se proširuje Logtalkom i njegovim funkcionalnostima provedbom instalacije potrebnih komponenti kroz službeni instalacijski program u obliku binarnog paketa, dok je eksluzivno za Prolog implementacije SWI moguće izvršiti proširenje kroz poziv ugrađenog predikata *pack_install/1* iz modula *prolog_pack* u interpreteru jezika, a koji je uključen u njegovim suvremenim inačicama. Prema savjetu autora ovog modula/jezika, instalacija kroz upravljač paketima SWI Prologa je prikladna za isporuku programskih rješenja baziranih na Logtalkovim pogodnostima, dok je pak nezgrapnija za sâm razvoj aplikacija s obzirom da su datoteke modula smještene iza nekolicine poddirektorija. (Moura, 2018)

Važna distinkcija kod Logtalka u odnosu, kako na ostale ugrađene elemente povezivanja Prologa s objektno-orijentiranom paradigmom, tako i na većinu programskih jezika te paradigme (poput Jave, C++ i Pythona), jest da ne postoje klase, već kategorije (engl. *categories*) koje su još prisutne kod manjeg broja jezika poput Objective-C. Kategorije naspram klasa služe za povezivanje usko-povezanih atributa i procedura zbog čega im instanciranje ne mora biti svrha postojanja s obzirom da objekti koji bi se iz njih stvarali ne bi baš bili smisleni ni uporabljivi. Naime, njihova ideja jest da se prilikom stvaranja objekta specificira kojim sve kategorijama on pripada, odnosno koje karakteristike i dijelovi programskog koda trebaju opisivati taj objekt. (Moura, 2003, str. 103-110)

Logtalk tako od objektno-orijentirane paradigme, osim što nudi mogućnost definiranja kategorija i stvaranja objekata iz njih, podržava enkapsulaciju što znači da je ugniježđenim predikatima moguće definirati vidljivost direktivom koja poziva jedan od predikata *public/1*, *protected/1* i *private/1* kojim se proslijeđenom predikatu definira istoimeni kvalifikator pristupa, baš kao što omogućuju programski jezici poput C++ i PHP-a. Ujedno postoji podrška za definiranje konstruktora kategorije (engl. *constructor*) čime se definira niz naredbi koje je potrebno izvršiti prilikom stvaranja objekta te kategorije, a definiraju se direktivom koja poziva predikat *initialization/1* kojem se kao argument prosljeđuju pozivi predikata omeđeni konjunkcijama. Također je ugrađena podrška za nasljeđivanje, pri čemu je podržano i višestruko nasljeđivanje (engl. *multiple inheritance*), dok ujedno postoji mogućnost definiranja protokola (koji su zapravo ekvivalent sučeljima) kao i njihovog implementiranja. (Moura, 2018, str. 103-110)

3.1.2.2. **XPCE**

Ovim modulom se s Prologa skida predrasuda da on ne može služiti kao programski jezik opće namjene (engl. *general-purpose programming language*), za što se kao glavni argument navodi da mu nedostaje izravna podrška za razvoj aplikacija s grafičkim sučeljem, mogućnost rada s grafičkim obrascima i kontrolama kroz proširive klase te obrada korisničkih događaja. Razvoj grafičkih aplikacija je ujedno olakšan s dostupnim uređivačem (engl. *dialog editor*) za uređivanje i definiranje dijaloških okvira, odnosno obrazaca (engl. *forms*). Ujedno aplikacije s grafičkim korisničkim sučeljem napravljene korištenjem ovog modula rade besprijekorno na različitim operacijskim sustavima (podržane su različite distribucije operacijskih sustava Windows, Macintosh, Linux i ostalih s grafičkim korisničkim sučeljem) bez potrebe za ikakvim preinakama ili prilagodbama, iako zbog toga izgled aplikacija nije nativan, no takva jest situacija i kod grafičkih aplikacija razvijenih u programskom jeziku Java.

Ovaj modul je za sada dostupan za SWI-Prolog, XSB Prolog te B Prolog, a ovdje ću se fokusirati na SWI-Prolog. Ukoliko modul nije predinstaliran sa samim Prologom implementacije SWI, potrebno ga je preuzeti i instalirati korištenjem prethodno-spomenutog predikata *packinstall/1* , pri čemu valja napomenuti da naziv modula, odnosno atom kojeg je potrebno specificirati kao argument predikatu, jest *pce*. XPCE zapravo nije Prolog, već sustav koji omogućava jezicima poput Prologa i Lispa izgradnju aplikacija korištenjem elemenata objektno-orijentirane paradigme, kao i izgradnju aplikacija s grafičkim korisničkim sučeljem. Objekti klasa nastalih korištenjem ovog modula koriste globalni prostor (engl. *global state*), omogućeno je redefiniranje varijabli klasa, uz predikate nudi i metode te ne postoji pojam nedeterminiranosti metoda. Korištenjem elemenata ovog modula se ujedno koristi proširena sintaksa jezika Prolog. (Wielemaker i Anjewierden, 2002, str. 45-46)

Elementi objektno-orijentirane paradigme se korištenjem ovog modula manifestiraju kroz združivanje pripadajućih grafičkih kontrola (ali i predikata) u zajedničke cjeline koje su dalje lako iskoristive kod različitih obrazaca koji ih rabe u istom ili sličnom formatu, a prisutna je i mogućnost nasljeđivanja, čime je prethodno definirano udruženje elemenata moguće proširiti s dodatnima bez da se zahtijeva ponovno navođenje opisa elemenata i njihovih odnosa čime bi se otežalo naknadno održavanje.

Od koncepata objektno-orijentirane paradigme, XPCE također nudi mogućnost nasljeđivanja klasa, doduše ne i višestruko nasljeđivanje. Enkapsulacija je pak podržana na način da se varijablama/atributima objekata klase definira vidljivost (točnije, pravo pristupa) ovisno o tome želi li se da im se vrijednost izvan same klase može samo mijenjati, samo čitati, oboje ili pak da im se uopće ne može pristupati – navedeno se postiže postavljanjem činjenice terma *variable/3* tako da se kao treći argument definira atom *send*, *get*, *both* ili *none* respektivno. Interesantno svojstvo prisutno kod ovog modula jest da se kod definiranja prethodno spomenute činjenice mora kao drugi parametar proslijediti tip podataka koji će u atributu klase biti pohranjen, što je u suprotnosti sa standardnom verzijom Prologa koji je netipiziran jezik (jedan od rijetkih izuzetaka je Turbo Prolog implementacija). Ugrađeni tipovi podataka kojeg atributi mogu biti su *int* (cijeli brojevi), *float* (decimalni brojevi), *name* (atomi), *chain* (specijalne liste), a atributi mogu ujedno biti i objekti, kako ugrađenih klasa poput *string* (znakovni nizovi), tako i korisnički definiranih klasa. (Wielemaker i Anjewierden, 2002, str. 49-66)

Kako bi se atributi objekta klase mogli inicijalizirati prilikom njegovog samog stvaranja, podržano je definiranje konstruktora klase koji se specificira stvaranjem specijalne metode naziva *initialise* koja je arnosti veće od 1, ovisno o tome koliko parametra se želi primiti i obraditi kod same instantacije klase. Isto tako, postojana je podrška i za definiranje destruktora stvaranjem specijalne metode *unlink/1* koja se automatski poziva prilikom uništenja svakog objekta iz pripadajuće klase. (Wielemaker i Anjewierden, 2002, str. 22)

S obzirom da se kod ove paradigme objekti kroz vrijeme mijenjaju te se na njih djeluje, valja spomenuti 4 operacije za rad s objektima – naime, riječ je o tzv. CRUD operacijama koje predstavljaju stvaranje (engl. *create*), čitanje (engl. *read*), ažuriranje (engl. *update*) i brisanje (engl. *delete*) objekata, a vrše se redom pozivom predikata *new/2*, *get/3*, *send/2* i *free/1*. (Wielemaker i Anjewierden, 2002, str. 8-10)

3.1.2.3. **Flora-2**

Flora-2 je sofisticirani sustav za rad sa znanjem baziranim nad objektima kao i za zaključivanjem nad njima. To je zapravo sustav koji u kôd Prologa s tabliranjem (konkretno kôd Prologa implementacije XSB) uz pomoć kompilatora pretvara svoj specijalni programski

jezik koji podržava logiku s okvirima, HiLog (formalnu logiku sa sintaksom više razine), transakcijsku logiku, poništavajuće zaključivanje (engl. *defeasible reasoning*) i metaprogramiranje. Flora-2 je zapravo besplatna inačica naprednijeg i komercijalnog sustava ErgoAI zbog kojeg se često naziva i Ergo-Lite, a on proširuje sustav s tehnologijama za rad sa semantičkim Webom (RDF-om, OWL-om i SPARQL-om), vjerojatnosnom logikom i rasuđivanjem (engl. *probabilistic logic and reasoning*), bogatijim sučeljem za povezivanje s programskim jezikom Java te integriranim razvojnim okruženjem (engl. *integrated development environment*) s mogućnošću pronalaženja pogrešaka (engl. *debugging*) i vršenja upita. Unatoč spomenutim funkcionalnostima komercijalne inačice, Flora-2 je ipak sama po sebi dovoljna za primjenu u razvoju inteligentnih sustava, upravljanju ontologijama, integraciji informacija i rad sa semantičkim Webom. (Kifer i sur., 2017, str. 1; Coherent Knowledge)

Flora-2, naspram XSB Prologa na kojem se bazira, kao i naspram većine drugih implementacija Prologa, podržava tipove podataka prilikom definiranja atributa klasa objekata, kao i povratnih tipova metoda i tipova njihovih parametara. Osim što mogu biti iz skupa korisnički-definiranih tipova podataka, na raspolaganju su i ugrađeni/primitivni tipovi podataka poput *_int* i *_long* za pohranu cijelih brojeva, *_double* i *_decimal* za pohranu decimalnih brojeva, *_dateTime*, *_date*, *_time* i *_duration* za rad s datumskim i vremenskim podacima, *_boolean* za rad s logičkim vrijednostima, *_string* za znakovne nizove, dok je *_list* za nizove objekata bilo kojeg tipa. (Kifer i sur., 2017, str. 101-112)

Kao i kod XPCE-a, Flora-2 podržava niz operacija za rad s objektima. Sustav razlikuje 2 vrste operacija: transakcijske i netransakcijske. Transakcijske operacije su prikladne kad se vrši rad nad objektima iz baze znanja u rekurzivnim predikatima, primjerice korištenjem pretraživanja s vraćanjem (engl. *backtracking*) kod traženja zadovoljavajuće kombinacije ili permutacije. Tako se u slučaju vraćanja na prethodni čvor s neuspješne grane unutar stabla pretraživanja vrši poništavanje svih provedenih operacija nad objektima u tom koraku. Primjeri takvih operacija su *t_insert*, *t_insertall*, *t_delete*, *t_deleteall* te *t_erase* i *t_eraseall*. Netransakcijske operacije pak obilježava da nisu tako automatski reverzibilne ili poništavajuće, a nazivi operacija takvog tipa su nalik na prethodno navedene, osim što su bez prefiksa „t_“. (Kifer i sur., 2017, str. 113-137)

Iako u ovome jeziku nije podržana enkapsulacija u smislu mogućnosti sakrivanja određenih atributa i metoda klasa, prisutna je enkapsulacija modula čime je moguće izvedene module učiniti nepromjenjivima. Nasljeđivanje je ipak u podržano čime je moguće vršiti proširivanje ili specijalizaciju klasa. (Kifer i sur., 2017, str. 92)

3.1.3. Logičko programiranje s ograničenjima

Kao što je već ranije spomenuto, ova paradigma je proširenje same logičke paradigme, čime se omogućuje pretraga rješenja unutar zadanog prostora rješenja. Za specificiranje prostora rješenja prilikom rješavanja određenog problema, za početak je potrebno odlučiti nad kojom domenom će biti definirana ograničenja i/ili u kojoj domeni se nalaze tražena rješenja. SWI Prolog kroz modul *clpr* nudi korištenje ove paradigme kada su vrijednosti racionalni brojevi, *clpr* kada su realni, *clpb* kada se radi logičkim varijablama, dok se kroz *clpfd* postiže podrška za rad nad ograničenim domenama te djelomična podrška nad cjelobrojnim brojevima. (SWI-Prolog Manual: Constraint Logic Programming)

Jedna od glavnih motivacija za nastanak ove programske paradigme je, uz reduciranje korištenja operatora *!/0* (tzv. rez), proizašla iz potrebe da se izračuna rezultat jednadžbe čak i ako se nevezana varijabla (engl. *unbounded variable*) ne nalazi s lijeve strane operatora *is/2* koji služi za izračun aritmetičkog izraza s desne strane. (Kiselyov i sur., 2008, str. 1-2)

Kao primjer navodim implementaciju predikata za računanje sljedbenika proslijeđenog broja:

```
sljedbenik(Trenutni, Sljedeci) :-
    Trenutni is Sljedeci + 1
.
```

Izvršavanjem upita *sljedbenik(15, X)* dobili bismo kao rezultat da je broj 16 sljedbenik proslijeđenog broja 15. Međutim, ako bismo postavili upit *sljedbenik(X, 16)* kojim bi se zatražio broj kojem je 16 sljedbenik, tada ovaj upit ne bi prošao. Točnije, dogodila bi se pogreška s opisom da su operatoru *is/2* pogrešno dodijeljeni argumenti zbog toga što taj operator zahtijeva da s desne strane bude aritmetički izraz bez ikakvih nepoznanica, odnosno nevezanih varijabli. Ako bi se korištenjem isključivo elemenata logičke paradigme programskog jezika Prolog pokušao učiniti spomenuti predikat prikladan za provedbu inverznog upita, tada bi se to eventualno moglo učiniti na sljedeći način:

```
sljedbenik(Trenutni, Sljedeci) :-
    var(Sljedeci),
    Sljedeci is Trenutni + 1,
    !
.
sljedbenik(Trenutni, Sljedeci) :-
    Trenutni is Sljedeci - 1
.
```

Korištenjem elemenata logičkog programiranja s ograničenjima, ovaj problem bi se riješio na domeni cijelih brojeva. Za početak bi bilo potrebno uključiti ugrađeni modul *clpfd*, a nakon toga umjesto operatora *is/2* koristiti specijalni operator *#=/2* iz prethodno uključenog modula koji pokušava izvršiti izračun nepoznanica (odnosno nevezanim varijablama definirati

zadovoljavajuću vrijednost) ili samo vrši logičku usporedbu kao operator *=:=/2* ukoliko se niti s jedne od strana ne nalaze nevezane varijable.

```
:- use_module(library(clpfd)).

sljedbenik(Trenutni, Sljedeci) :-
    Sljedeci #= Trenutni + 1
.
```

Prednost ovakvog rješenja jest što je uz jedno tijelo predikata i to s jednim ciljem postignuto vršenje upita nad predikatom koji vrši aritmetičku operaciju izračun sljedbenika, a pritom može poslužiti i za izračun prethodnika. Glavni nedostatak je pak što operator *#=/2* u SWI Prologu radi isključivo s cijelim brojevima, odnosno dosadašnja implementacija posljednjeg predikata se ne bi mogla iskoristiti za izračun broja koji je za 1,5 veći od proslijeđenoga, a ni manji. No vrijedi istaknuti da je ovo mana promatrane implementacije Prologa jer primjerice kod ECLiPSe Prologa, koji je implementacija Prologa specijalizirana za logičko programiranje s ograničenjima, spomenuti nedostatak nije prisutan, jer kod njega ovaj operator radi i s realnim brojevima. (The ECLiPSe Constraint Programming System, 2018)

3.1.4. Funkcionalna paradigma

Iako su funkcionalna paradigma i logička paradigma veoma bliske (obje pripadaju roditeljskoj deklarativnoj paradigmi), pa čak čine zasebnu paradigmu zvanu funkcionalno logičko programiranje (čiji su predstavnici spomenuti jezici Curry i Mercury), samo λProlog i NUE-Prolog u većoj mjeri nude korištenje elemenata specifičnih za funkcionalnu paradigmu. (Hanus, 2007, str. 18-19)

Vrijedi spomenuti da po samome ISO standardu Prolog ima ugrađene predikate kojima se omogućava rješavanje problema korištenjem pristupa funkcionalne paradigme programiranja. Naime, riječ je o meta-predikatima poput *call* (koji je pozitivne varijabilne arnosti), *foldl* (arnosti 2-7), *apply/2* i *maplist* (arnosti 2-5). Spomenutim predikatima se postiže mogućnost primjene programiranja višeg reda, a ono u suštini predstavlja prosljeđivanje funkcija kao parametara kod poziva funkcija (u kontekstu Prologa je tu riječ o predikatima umjesto funkcija). (Naish, 1996)

S obzirom da se u ovom radu koncentriram na SWI-Prolog, za njega vrijedi spomenuti da za neke elemente funkcionalne paradigme nudi podršku kroz nekoliko modula. U nastavku donosim sažet opis modulā *func* i *yall*.

3.1.4.1. func

Svrha ovog modula jest mogućnost poziva predikata u formatu sličnome pozivu funkcija kod većine drugih programskih jezika poput C, Python i Java. Naime, manom Prologovih predikata se smatra to što je potrebno za dohvat rezultata predikata kao i za njegovu upotrebu

u nekoj operaciji (bilo aritmetičko-logičkoj ili općenito prosljeđivanje vrijednosti kao argument nekom drugom predikatu prilikom poziva) potrebno uvoditi dodatnu varijablu u koju će se vrijednost pohraniti te onda još dodatno proslijediti tu varijablu kao operand željenoj operaciji. (Hendricks, 2016)

Kao primjer prikladnog problema navodim slučaj u kojem je potrebno izvršiti serijsku obradu podataka (engl. *batch processing*), odnosno obradu koja se sastoji od primjene *N* operacija/predikata nad podacima te se rezultat jednog predikata prosljeđuje kao ulaz drugom predikatu, a rješenje bi se sastojalo od *N* naredbi. Da bih predočio nedostatke prilikom rješavanja takvih problema, slijedi primjer niza naredbi koje vrše proizvodnju i sklapanje automobila (pretpostavimo da su korišteni predikati definirani).

```
…
izradiKaroseriju(Metal, KaroserijaAuta),
ubaciMotor(KaroserijaAuta, AutoSMotorom),
montirajKotace(AutoSMotorom, AutoSMotoromIKotacima),
obojajAuto(AutoSMotoromIKotacima, KonacanAuto),
…
```

Korištenjem pogodnosti modula *func* moguće je ovakvu obradu izvršiti navođenjem svih predikata omeđenih *of/2* operatorom čiji je rezultat objekt-funkcije koja se onda može pozvati s *call/2* meta-predikatom.

```
…
IzradaAuta = izradiKaroseriju of ubaciMotor of montirajKotace of obojajAuto,
call(IzradaAuta, Metal, KonacanAuto),
…
```

ili još kraće direktnim pozivanjem funkcije prosljeđivanjem argumenta koristeći operator *$/2* te direktnom inicijalizacijom rezultne varijable:

```
…
KonacanAuto =
    izradiKaroseriju of ubaciMotor of montirajKotace of obojajAuto $ Metal,
…
```

U ovom primjeru je ujedno prikazan i način korištenja lambda ili anonimnih funkcija s obzirom da je naveden niz predikata koje je potrebno pozvati, a u konačnici je i pozvan specificiranjem ulaznog argumenta operatorom *$/2*.

Ograničenje ovog modula moguće jest što se može iskoristiti samo za refaktoriranje predikata arnosti veće od 1 pri čemu predzadnji argument mora biti ulazni, a zadnji izlazni. Ako je pozivajući predikat arnosti veće od 2, tada je potrebno sve argumente predikata specificirati unutar složenog terma koji predstavlja objekt-funkciju, izuzev posljednja dva pri čemu se predzadnji postavlja prije samog poziva te funkcije. Međutim, ono što ide na korist ovome modulu unatoč spomenutoj limitaciji je što je zapravo uistinu većina predikata u programskom jeziku Prolog ovakvog formata, iako se gubi sposobnost provedbe inverznog upita nad

predikatom, što se ipak može riješiti definiranjem dodatnog predikata koji će vraćati rezultat inverznog upita nad tim predikatom. (Hendricks, 2016)

Još jedan nedostatak programskog jezika Prolog (kao i ostalih jezika logičke, ali i funkcionalne paradigme) jest da je predviđen za rekurzivno rješavanje problema naspram iterativnog. Prednost rekurzivnog pristupa je često mogućnost rješavanja problema u manjem broju koraka i s manjim brojem pomoćnih podataka naspram iterativnog. S druge strane najveća mana su razlike u performansama, kako sporost prilikom rješavanja problema zbog promjene konteksta prilikom ulaska i izlaska iz rekurzije, tako i nemogućnost završavanja deterministčkog algoritma zbog nastupa prenatrpavanja/prepisivanja memorijskog stoga (engl. *stack overflow*). Ipak, prevoditelji i interpreteri programskog koda mogu ponekad izbjeći spomenuti nedostatak i to u slučaju kada je korištena repna rekurzija, odnosno kada je rekurzivni poziv zadnje što se vrši u predikatu. (Meier, 1991)

Kao primjer navodim slučaj u kojem želimo napisati predikat *faktorijel/2* koji računa faktorijel nekog broja, odnosno umnožak svih prirodnih brojeva manjih ili njemu jednakih. Jedno od rješenja bi moglo biti sljedeće:

```
faktorijel(0, 1) :- !.
faktorijel(N, Rezultat) :-
    N > 0,
    N1 is N - 1,
    faktorijel(N1, Podrezultat),
    Rezultat is Podrezultat * N.
```

Ovo rješenje bi se moglo skratiti korištenjem pogodnosti *func* modula tako da bude u sljedećem obliku:

```
:- use_module(library(func)).
faktorijel(0, 1) :- !.
faktorijel(N, Rezultat) :-
    N > 0,
    Rezultat is N * (faktorijel of _ -1 $ N).
```

Nijedna od navedenih implementacije predikata za izračun faktorijela ne sadrži repnu rekurziju, iako se kod drugog primjera čini da se rekurzivni poziv vrši na samome kraju predikata što zapravo nije slučaj jer se ta naredba izvršava s desna ulijevo čime se prvo vrši rekurzivni poziv, a tek nakon toga se vrši izračun umnoška vraćene vrijednosti koji je (*N-1)!* i *N*, tj. rekurzivni poziv se nalazi u zadnjoj naredbi predikata, ali ne i u zadnjoj instrukciji!

Alternativni način rješavanja ovog problema uz ostvarivanje repne rekurzije bi bio sljedeći:

```
faktorijel(N, Rezultat) :- faktorijel(N, 1, Rezultat).
faktorijel(0, Rezultat, Rezultat) :- !.
faktorijel(N, DosadasnjiUmnozak, Rezultat) :-
    N > 0,
    N1 is N - 1,
    NoviUmnozak is DosadasnjiUmnozak * N,
    faktorijel(N1, NoviUmnozak, Rezultat).
```

Za potrebe realizacije je uveden novi predikat *faktorijel/3* kod kojeg je rekurzivni poziv na samome kraju predikata te bi se u slučaju pokretanja ovog predikata s interpreterom koji vrši optimizaciju (tj. pretvorbu rekurzivnog rješenja u iterativno u slučaju korištenja repne rekurzije) na razini strojnog jezika izvršile naredbe nalik onima koje bi nastale prevođenjem sljedećeg programa pisanog u programskom jeziku C:

```
void faktorijela(signed int N, signed int* Rezultat) {
    *Rezultat = 1;
    for (unsigned int i=N; i>=1; i--) {
        *Rezultat *= i;
    }
}
```

3.1.4.2. yall

Ovim modulom su podržani lambda izrazi čime je promijenjen pristup prilikom prosljeđivanja cilja meta-predikatima poput *bagof/3*, *findall/3* i *setof/3* koji služe za dohvat svih rješenja u obliku liste koji ispunjavaju definirani cilj. Naime, kod poziva spomenutih predikata je inače uobičajeno kao drugi argument proslijediti operaciju s egzistencijalnim operatorom *^/2* pri čemu se s njegove desne strane nalazi cilj čija se zadovoljavajuća rješenja traže, dok se s lijeve strane navode egzistencijalne varijable iz terma cilja. Drugim riječima, spomenutim operatorom se specificira koje varijable iz terma cilja se ne smiju vezati za rezultate cilja, čime se omogućava vršenje grupiranja nad rezultatima proslijeđenog cilja. Korištenjem pogodnosti ovog modula se mijenja pristup jer se, umjesto navođenja niza varijabli za koje se ne smije vezati vrijednost, zapravo navodi suprotno – niz varijabli za koje vrijednost treba biti vezana prilikom ispitivanja svih rješenja zadanog cilja. (SWI-Prolog Manual: library(yall): Lambda expressions)

U nastavku navodim primjer implementacije predikata *lista_svih_roditelja/1* koji vraća listu atoma koji predstavljaju nazive roditelja, a koji podatke izvlači iz činjenica tipa *roditelj/2* (pri kojima prvi argument predstavlja naziv roditelja, dok drugi pak naziv djeteta kojemu je roditelj) koje neću eksplicitno navoditi. Prvi primjer je realiziran korištenjem egzistencijalnog operatora, odnosno kod kojeg se navodi varijabla koje se neće prikupljati, a po kojima se pritom ne treba vršiti grupiranje.

```
lista_svih_roditelja(Lista) :-
    setof(Roditelj, Dijete ^ roditelj(Roditelj, Dijete), Lista).
```

U sljedećem se pak primjeru koriste operatori iz modula *yall*. Naime, sadržani operator *//2* kao desni argument prima cilj čija rješenja se traže, dok se s lijeve strane unutar vitičastih zagrada (koje označavaju da je riječ o skupu, odnosno da je redoslijed navođenja varijabli, ako ih je više, irelevantan) nalaze varijable iz terma cilja za koje nije potrebno vezati vrijednosti pa tako ni vršiti grupiranje.

```
:- use_module(library(yall)).
lista_svih_roditelja(Lista) :-
    setof(Roditelj, {Roditelj} / roditelj(Roditelj, _), Lista).
```

Može se zaključiti da je prvi pristup prikladan u slučaju kad je potrebno većinu ili sve argumente terma cilja staviti kao element tipa složeni term (engl. *compound term*) u rezultnu listu, dok je drugi pristup pak prikladan ukoliko ispitivani cilj ima mnogo izlaznih varijabli čije vrijednosti ne želimo prikupljati u listi ni ne želimo da se po njima vrši grupiranje.

3.1.5. Programiranje pogonjeno događajima

Obrada događaja se u programskim jezicima koji nemaju direktnu podršku za elemente ove paradigme najčešće vrši na način da postoji zasebna dretva ili proces koji za vrijeme cijelog trajanja rada aplikacije (ili prisutnosti kontrole nad kojom se osluškuje potencijalni događaj) provjerava je li se dogodio osluškivani događaj i u potvrdnom slučaju izvršava pripadajuću akciju, najčešće poziv prethodno definirane rutine/potprograma. SWI-Prolog (kao i neki drugi poput Prologa implementacije XSB, YAP, ECLiPSe i sl.) nudi podršku za dretve, procese i ostvarivanje paralelizma te bi se obrada događajima mogla implementirati na spomenuti način. Ipak, za taj Prolog kroz module postoji mogućnost obrade događaja na vrlo visokoj, apstraktnoj i pojednostavljenoj razini, bez potrebe da se programer aplikacija direktno koristi elementima paralelne obrade (engl. *parallel computing*).

Moduli koje ću opisati su zapravo bili prethodno spomenuti kod objektno-orijentirane paradigme – riječ je o XPCE i Logtalk modulima. Iako su programiranje pogonjeno događajima i objektno orijentirana paradigma vrlo usko povezani i najčešće međusobno integrirani jer su subjekti osluškivanja radi poduzimanja određenih akcija najčešće objekti (odnosno instance klasa) i njihova stanja, te dvije paradigme se tretiraju kao zasebne paradigme jer općenito ne zahtijevaju jedna drugu.

3.1.5.1. XPCE

Kao što je već spomenuto, ovaj modul pruža niz ugrađenih predikata kojima se omogućuje izgradnja stolnih aplikacija (engl. *desktop applications*) s grafičkim korisničkim sučeljem. Ono što čini ovaj modul naprednijim od ostalih iz ove paradigme jest mogućnost obrade događaja uzrokovanih korisničkom interakcijom nad aplikacijom poput klika mišem, pritiska određene tipke na tipkovnici i slično. U nastavku je prikazan programski kôd kojim se

kod poziva predikata *pocetna/0* pojavljuje grafički prozor s nazivom *Pozdravljajuca aplikacija* koji sadrži tipku s natpisom *Pozdravi me*, a na čiji se pritisak otvara novi prozor naziva *Pozdrav* i s tekstom *Bok* koji se zatvara pritiskom na bilo koju poziciju na tom prozoru.

```
:- use_module(library(pce)).

pocetna :-
    new(D, dialog('Pozdravljajuca aplikacija')),
    send_list(D, [
        width(300),
        height(200),
        append(button('Pozdravi me', message(@prolog, pozdrav))),
        open
    ]).

pozdrav :-
    new(D, dialog('Pozdrav')),
    send(D, width, 100),
    send(D, height, 50),
    send(D, append, text('Bok')),
    send(D, recogniser,
        click_gesture(left, '', single, message(@prolog, send, D, destroy))
    ),
    send(D, open).
```

Element programiranja pogonjenog događajima u navedenom primjeru jest prosljeđivanje objekta *message* arnosti 2 ili više pri čemu se kao prvi parametar prosljeđuje primatelj/obrađivač događaja (najčešće *@prolog* ili *@pce* , ovisno o tome koji sustav treba izvršiti obradu), drugi naziv predikata koji je potrebno pozvati, dok su daljnji parametri izborni, a prosljeđuju se predikatu kod poziva. Kod definicije predikata *pocetna/0* se taj objekt prosljeđuje objektu klase *button* prilikom njegove instantacije/stvaranja (tj. kao parametar konstruktoru ili inicijalizatoru) s obzirom da se za tipke (objekte iz klase *button*) pretpostavlja da vrlo vjerojatno treba postojati neka akcija kad se dogodi njihov pritisak. S druge strane, kada se želi definirati akcija klika mišem (ili neki drugi korisnički događaj) na grafičku kontrolu kojoj se ne može prilikom kreiranja proslijediti način obrade događaja (primjerice jer standardno nije predviđena da se nad njome treba obraditi takav događaj), tada se objektu nad kojim se želi osluškivati događaj treba promijeniti vrijednost atributa *recogniser* koji je zapravo objekt konkretizacije istoimene klase. Tako je kod definicije predikata *pozdrav/0* definirano uništavanje stvorenog dijaloškog okvira prilikom klika lijevom tipkom miša na njega. (Wielemaker i Anjewierden, 2002, str. 82-85)

3.1.5.2. Logtalk

Obrada događaja je kod ovog programskog jezika veoma specifična zbog toga što je za svako slanje poruka između objekata omogućeno izvršavanje određene akcije neposredno prije nastupa događaja, kao i izvršavanje pripadajuće akcije nakon što događaj završi. Može se tako prepoznati velika sličnost s okidačima (engl. *triggers*) iz baza podataka kod kojih je isto

tako moguće izvršiti određeni niz naredbi neposredno prije izvršavanja operacije koja je uzrokovala pokretanje okidača, kao i drugi niz naredbi odmah nakon što ta operacija završi. (Moura, 2018)

Obrada događaja se tako u Logtalku vrši kreiranjem zasebnog objekta implementiranjem protokola *monitoring*, odnosno definiranjem njegovih metoda/predikata *before/3* i *after/3*. Spomenute metode kao prvi argument imaju varijablu koja predstavlja sâm objekt koji prima poruku, drugi predstavlja samu poruku koja je primljena, dok zadnji predstavlja pošiljatelja, odnosno objekt koji je tu poruku poslao. Ujedno je potrebno u samom izvornom kodu programa pozvati direktivu *set_logtalk_flag*/2 te joj proslijediti kao argumente atome *events* i *allow* čime se Logtalk prevoditelju daje do znanja da svi objekti trebaju imati podršku za rad s događajima. (Moura, 2003, str. 150-141)

Moguća je ujedno kombinacija XPCE-a i Logtalka prilikom razvoja aplikacija. Tako predikati unutar XPCE klasa mogu slati poruke Logtalk objektima, dok predikati iz Logtalk objekata i kategorija mogu pozivati XPCE metode. Također je ugrađena podrška za korištenjem Logtalk ciljeva za slanje poruka kao XPCE ciljeva povratnog poziva. (Moura, 2011)

3.2. Povezivanje s drugim programskim jezicima

Ponekad se može dogoditi da programski jezik logičke paradigme ne sadržava nikakve module za rješavanje određenog problema (što posebice dolazi do izražaja kod ostalih implementacija Prologa koji nisu kompatibilni sa SWI-Prologom), neprikladan je za rješavanje određenog problema unatoč pomoćnim modulima, no dostupan je određeni dio tražene funkcionalnosti napisan u nekome drugome programskom jeziku ili pak razvojni inženjer preferira programski jezik neke druge programske paradigme za rješavanje suočenog potproblema prilikom rješavanja određenog problema. Za sve navedene situacije se kao rješenje nudi ovaj način povezivanja logičke paradigme s drugima.

U nastavku slijede metode kojima se može vršiti izgradnja kompleksnih programskih rješenja koja su izgrađena od više različitih programskih jezika, a koji pritom mogu biti različitih programskih paradigmi.

3.2.1. Komunikacija s procesom (izvršavatelja) programa

Kod ove metode jedan program pisan u jednom jeziku poziva drugi program pisan u drugom. Drugim riječima, proces koji izvodi jedan program stvara novi proces koji izvršava drugi program pri čemu se može pratiti rezultat njegovog izvođenja, ali se može i upravljati njegovim tijekom izvođenja. Razlog zbog kojeg je u naslovu ovog odlomka spomenut izvršavatelj programa jest da su neki programski jezici skriptni te se ne pokreću direktno (poput

programa napisanih u programskim jezicima koji se prevode ili kompiliraju), već se prvo pokreće interpreter koji onda poziva skriptu ili program koji korisnik želi izvršiti. Preduvjet za povezivanje programā na ovakav način jest da pozivajući program može stvarati procese te komunicirati sa njima, odnosno da programski jezik u kojem je on napisan podržava neku od metoda za međuprocesnu komunikaciju (engl. *inter-process communication*). Neke od metoda za ostvarivanje međuprocesne komunikacije su sljedeće:

- Korištenje zajedničke dijeljene memorije između procesa (engl. *shared memory*) i nekog sinkronizacijskog mehanizma za ostvarivanje međusobnog isključivanja (engl. *mutual exclusion*) poput semafora (engl. *semaphore*) ili monitora (engl. *monitor*)
- Korištenje mrežne utičnice
- Korištenje mehanizma redova poruka (engl. *message queues*)
- Korištenje cjevovoda (engl. *pipeline*)

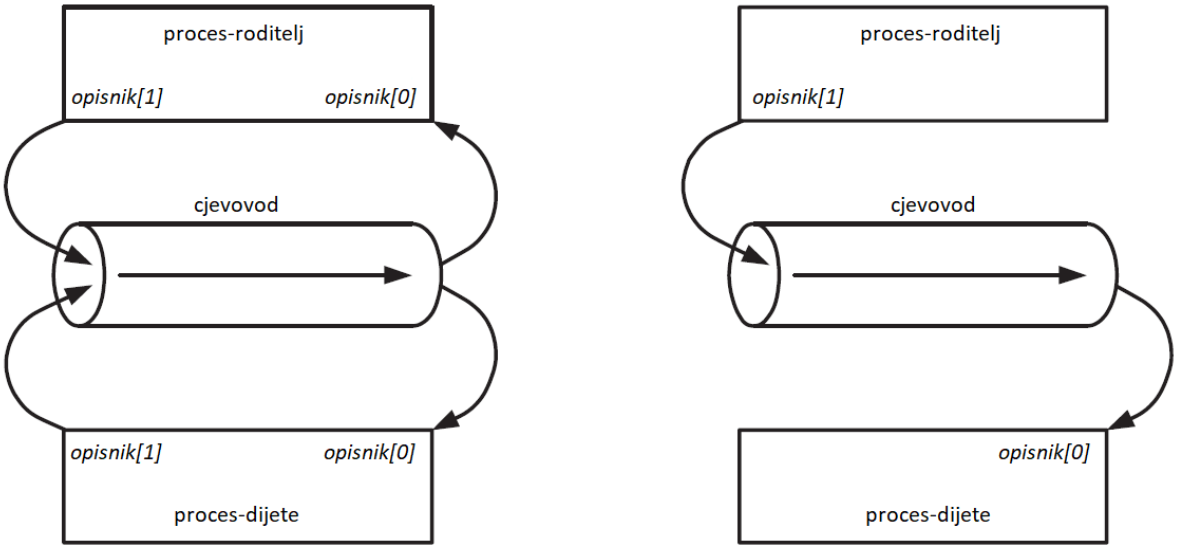
Najsofisticiranija metoda od navedenih je korištenje cjevovoda zbog toga što su cjevovodi kompleksan mehanizam koji je najčešće implementiran u suvremenim operacijskim sustavima korištenjem primitivnijeg mehanizma redova poruka. (Stevens, 1998, str 6-8)

Cjevovod je zapravo mehanizam ugrađen u većinu modernih operacijskih sustava koji služi za ostvarivanje komunikacije između više procesa. S jedne strane cjevovoda se tako nalazi proces koji objavljuje podatke (engl. *publish*), dok s druge strane jedan ili više procesa koji su pretplaćeni (engl. *subscribe*) na taj cjevovod ih čekaju, čitaju i obrađuju. (Budin i sur., 2013, str. 62-63)

Korištenjem ovog načina povezivanja dvaju procesa programā je moguće ostvariti komunikaciju ako osluškivana aplikacija šalje podatke na cjevovod poput standardnog izlaznog toka (engl. *standard output stream*) ili ima mogućnost obrade podataka zaprimljenih na cjevovod poput standardnog ulaznog toka podataka (engl. *standard input stream*). Ako su zadovoljena oba uvjeta, tada je moguća i dvosmjerna komunikacija (engl. *duplex communication*), odnosno moguće je upravljati i tijekom izvođenja osluškivanog i pokrenutog programa, a ujedno se zaprima rezultat koji se šalje tijekom izvođenja osluškivanog programa. (Tanenbaum i van Steen, 2002, str. 172-174)

Na Slici 2 je prikazan tok informacija između dva procesa kako bi se ostvarila međuprocesna komunikacija. Tako je lijevo na slici prikazan način ostvarenja dvosmjerne komunikacije između procesa, dok je na desnoj strani prikazana komunikacija u samo jednom smjeru. Roditeljski proces u primjeru stvara novi proces (tzv. proces-dijete) te koristi cjevovod za uspostavljanje komunikacije s njim. Valja napomenuti da se natpis *opisnik[0]* sa slike odnosi na opisnik datoteke (engl. *file descriptor*) koji pokazuje na otvoreni standardni ulazni tok

podataka (tzv. *stdin*), dok se pak *opisnik[1]* odnosi na opisnik datoteke koja pokazuje na otvoreni standardni izlazni tok podataka (tzv. *stdout*). Također je kod međuprocesne komunikacije moguće koristiti i treći dostupni opisnik datoteke koji se nalazi treću po redu (odnosno po dosadašnjoj konvenciji imenovanja bi se zvao *opisnik[2]*), a pokazivao bi na otvoreni standardni izlazni tok za prijenos poruka o pogreškama (tzv. *stderr*).



Slika 2: Prikaz tokova informacija kod dvosmjerne (lijevo) i jednosmjerne međuprocesne komunikacije (desno) (Prema: Kerrisk, 2010)

Ukoliko se želi ostvariti komunikacija između dva programa pri čemu jedan program osluškuje izlazni rezultat koji generira drugi program, a podatke mu šalje jednokratno prilikom njegovog pokretanja, tada je moguće navedeno učiniti korištenjem cjevovoda za komunikaciju u jednom smjeru, tj. od drugog programa prema prvome, dok je inicijalne podatke moguće proslijediti iz prvoga drugome kroz parametre prilikom pokretanja programa (engl. *command-line arguments*). Ako se eventualno ujedno želi poslati drugome programu zahtjev o prijevremenom završetku, navedeno se može realizirati slanjem sistemskog signala/prekida kojeg onda ovaj drugi može pravovaljano obraditi, primjerice završetkom daljnje obrade i vraćanjem kakvih završnih podataka.

Cjevovodi su ujedno jedina od spomenutih metoda za međuprocesnu komunikaciju koja ne zahtijeva da programski jezik, u kojem je napisan osluškivani proces programa, podržava posebne metode za rad sa njima, već je samo dovoljno da se u programu podaci mogu slati na standardni izlazni tok i/ili čitati s standardnog ulaznog toka. Tako je korištenjem ove metode moguće ostvariti izgradnju složenih programskih rješenja sadržanih od više programa napisanih u više programskih jezika sve dok se prosljeđivani podaci razumljivi svim potprogramima.

Vrijedi spomenuti da se ovom metodom može vršiti stvaranje procesa koji će izvršavati drugi program, neovisno o tome pokreće li se program direktno (u slučaju da je zapravo izvršna datoteka, npr. programi Visual Prolog dijalekta se kompiliraju) ili se pak pokreće putem interpretera pri čemu je potrebno da je interpreter tog programa instaliran na korištenoj radnoj okolini (većina ostalih implementacija Prologa zahtijeva interpreter za izvođenje programā). (Boer, 2008, str. 134-135)

U nastavku ću ukratko opisati pozivanje procesa programa, ovisno o tome u kakvoj je interakciji sa svojim roditeljskim procesom.

3.2.1.1. Jednosmjerna međuprocesna komunikacija

Za ostvarenje jednosmjerne komunikacije (bilo potpune ili djelomične prosljeđivanjem inicijalnih podataka prilikom pozivanja drugog programa) dovoljno je samo da proces-roditelj posjeduje mogućnost stvaranja procesa iz željenog programa kao i dohvaćanja tokova podataka za stvoreni proces. Kod ovog pristupa bilo koja implementacija Prologa može biti na osluškivanoj strani, dok god se rezultat obrade koji bi se trebao podijeliti sa roditeljskim procesom ispisuje na standardni izlazni tok podataka. Ako se pak iz Prologa želi stvarati proces nekog programa, tada on mora posjedovati predikate za kreiranje procesa koji će vršiti određenu obradu te ujedno iščitavati podatke s izlaznog toka stvorenog procesa.

SWI-Prolog posjeduje predikat *process_create/3* koji kao prvi argument prima naziv programa kojeg je potrebno pokrenuti s putanjom u kojoj se on nalazi, kao drugi listu parametara koji će biti proslijeđeni programu prilikom njegovog pokretanja, dok se za zadnji argument veže lista s opcionalnim parametrima vezanima uz samo stvaranje procesa. Upravo se kroz spomenutu listu opcionalnih parametara mogu dohvatiti tokovi podataka od stvaranog procesa. Upravljanje tokovima *stdin*, *stdout* i *stderr* se provodi postavljanjem istoimenih termova arnosti 1 kao elemenata te liste, a sâm dohvat objekta, koji predstavlja tok podataka s kojim je moguće dalje raditi, se vrši prosljeđivanjem nevezane varijable unutar terma *pipe/1* kao argumenta jednom od spomenutih termova. Ujedno je u listu opcionalnih parametara moguće kao element uvrstiti term *process/1* koji bi u njemu uvrštenu varijablu vratio identifikator procesa čime bi se moglo vršiti čekanje stvorenog procesa ili bi mu se mogli slati signali za prekid. (SWI-Prolog Manual: library(process): Create processes and redirect I/O)

U nastavku kao primjer navodim implementaciju predikata za SWI-Prolog koji stvara proces iz programa čiji je naziv proslijeđen u formatu atoma kao prvi argument, dok je kao drugi argument moguće specificirati parametre koji će mu biti proslijeđeni prilikom stvaranja te se kao rezultat predikata vraća prvi redak rezultata i to tek kada završi cjelokupno izvođenje stvorenog procesa programa.

```
:- use_module(library(process)).
dohvatiRezultatProcesa(NazivPrograma, Rezultat) :-
    dohvatiRezultatProcesa(NazivPrograma, [], Rezultat).
dohvatiRezultatProcesa(NazivPrograma, Parametri, Rezultat) :-
    process_create(path(NazivPrograma), Parametri, [
                                                    stdout(pipe(IzlazniTok)),
                                                    process(IdProcesa)
                                                     ]),
    read_line_to_codes(IzlazniTok, KodoviRezultata),
    atom_codes(Rezultat, KodoviRezultata),
    process_wait(IdProcesa, _)
.
```

3.2.1.2. Dvosmjerna međuprocesna komunikacija

Iako je u prethodnom odlomku spomenut (a ujedno i kroz praktični primjer potkrijepljen) način jednosmjerne komunikacije kod koje osluškivani proces može primati podatke prilikom samog pozivanja programa, pri čemu mogu biti poslani i prekidni signali, taj način se ne može nazvati dvosmjernim u punom smislu riječi jer nakon početka izvođenja pozvanog programa više nije moguće primati podatke od strane roditeljskog procesa.

Za realizaciju prave dvosmjerne komunikacije između Prologa i nekog drugog programskog jezika, osim što je potrebno da glavni proces posjeduje mogućnost stvaranja procesa iz željenog programa te dohvaćanja njegovih tokova podataka, oba procesa bi trebala posjedovati mogućnost razmjene opisnika datoteke korištenjem utičnica za međuprocesnu komunikaciju (engl. *inter-process communication sockets* ili skraćeno *IPC sockets*) kako bi se tokovi roditeljskog procesa za ulaz i izlaz podataka mogli podijeliti sa stvorenim procesom te bi on mogao osluškivati poslane podatke od roditeljskog procesa. Kod ovog pristupa Prolog može biti na biloj kojoj strani, tj. kako stvaratelj procesa pozivajućeg programa, tako i stvoren od procesa drugog programa. (Kerrisk, 2010, str. 1284)

S obzirom da je podrška za rad s mrežnim utičnicama slabo podržana od strane većine implementacija Prologa te je za većinu problema zadovoljavajuće rješenje s jednosmjernom komunikacijom iz smjera stvorenog procesa prema roditeljskom uz mogućnost prosljeđivanja podataka kao parametara prilikom samog pokretanja pozivajućeg programa, daljnja razrada ovog načina ostvarivanja komunikacije neće biti nastavljena.

3.2.1.3. Bez komunikacije između procesa

Mogao bi se kao zaseban način povezivanja različitih programskih jezika navesti onaj kod kojeg procesi dvaju (ili više) programa uopće ne komuniciraju, već jedan proces samo stvori drugi te eventualno čeka da on završi. Za ostvarenje navedenoga je samo potrebno da program roditeljskog procesa posjeduje mogućnost stvaranja procesa.

Kod ovog pristupa se mogu eventualno poslati podaci iz roditeljskog procesa kreiranome prilikom njegovog stvaranja. Implementiranje ovakvog načina povezivanja se

može ostvariti na sličan način kao i kod jednosmjerne komunikacije, osim što u listi opcionalnih parametra nije potrebno postaviti termove za dohvat tokova podataka stvaranog procesa. Baš zbog toga je ujedno moguće koristiti i ugrađeni predikat *shell/1* koji kao argument prima naziv programa kojeg je potrebno izvršiti, a pri tome je ispred naziva samog programa moguće specificirati putanju ako je potrebno, dok je iza naziva programa moguće staviti i parametre s kojima će se stvarani program pokrenuti. Taj predikat je ujedno dostupan i kod nekih drugih implementacija Prologa poput SICStus, XSB (unutar *shell* modula) i ostalih. (SWI-Prolog Manual: library(process): Create processes and redirect I/O; SICStus Prolog – Predicate Index; Swift i sur., 2007, str. 183)

3.2.2. Korištenje gotovog sučelja za rad s interpreterom jezika

Kako bi se razvojnog inženjera poštedjelo razmišljanja na niskoj razini apstrakcije prilikom razvoja, kao i vođenja računa o pojedinostima pozadinskog operacijskog sustava za koji se razvija programsko rješenje, razvijena su gotova sučelja za povezivanje različitih programskih jezika. Nažalost, takva sučelja nisu općenita, već su specijalizirana za povezivanje određena dva programska jezika. Pri tome sučelja mogu biti za povezivanje u samo jednom smjeru, ali i u oba smjera.

Za programski jezik Prolog su dostupna sljedeća sučelja:

- InterProlog (jednosmjerna, od SWI, YAP ili XSB Prologa prema Javi)
- JPL (dvosmjerna između SWI Prologa i Jave)
- C++ sučelje za SWI (dvosmjerna između SWI Prologa i C++)
- C sučelje za SWI (dvosmjerna između SWI Prologa i jezika C)

U slučaju da se program nekog programskog jezika želi integrirati s Prolog programom, a za njihovu integraciju ne postoji gotovo sučelje, potrebno je koristiti prethodno obrađivanu tehniku povezivanja programā. Kroz sljedeća poglavlja niže razine slijedi kratki osvrt na svako od navedenih sučelja.

3.2.2.1. InterProlog sučelje

Svrha ovog sučelja je omogućiti iz različitih implementacija Prologa rad s komponentama iz Java razvojne okoline. Tako se primjerice ovo sučelje može iskoristiti za instanciranje, prilagodbu i sâm prikaz grafičkih obrazaca, kao i kontrola korisničkog sučelja kojima se obrasci mogu popunjavati. Motivacija za korištenje ovog sučelja može biti i dohvat programa pisanih u Javi koji nude uslugu koja je nedostupna u korištenoj implementaciji Prologa, ali i rješavanje problema koji nisu predviđeni za rješavanje logičkom paradigmom. Kod ovog sučelja je integracija Prologa i Jave zapravo realizirana korištenjem mrežnih utičnica

i Java sučelja za povezivanje s nativnim programima (engl. *Java Native Interface - JNI*). (Calejo, 2004)

Implementacije Prologa koje mogu koristiti elemente Jave su SWI, YAP i XSB Prolog, dok je već neko vrijeme u najavi podrška za GNU Prolog. Trenutno je jedino sučelje prema Javi koje je podržano za više implementacija jezika Prolog. S obzirom da su programska rješenja razvijena Javom višeplatformska, a isto vrijedi i za programe napisane podržanim implementacijama Prologa, ovom paradigmom mogu razviti cjelokupni programski sustavi koji su pritom lako portabilni. (Calejo, 2004, str. 80; Declarativa)

3.2.2.2. JPL sučelje

Ovo sučelje je standardno za povezivanje Prologa implementacije SWI s programskim jezikom Java pa tako čak dolazi predinstalirano s tim Prologom. Nasuprot prethodno spomenutog InterProlog sučelja, ovo sučelje nudi mogućnost povezivanja u oba smjera. Tako je moguće iz SWI-Prologa pozvati programski kôd napisan u Javi, ali je moguće i obrnuto. Za rad s logičkim programima napisanima u Prologu iz Java programā, potrebno je za početak u projekt izrađivane aplikacije dodati referencu na *jpl.jar* datoteku/arhivu koja zapravo sadrži skup metoda za vršenje upita i upravljanje pozadinskom sustavom i bazom znanja Prologa, ali i struktura podataka kojima se ostvaruje djelomična kompatibilnost između tipova podataka ta dva jezika. (A SWI-Prolog to Java interface, 2004)

JNI se i kod JPL-a koristi u pozadini za izvedbu komunikacije između Prologa i Jave, a vrijedi spomenuti način suradnje sakupljačā smeća (engl. *garbage collectors*) tih dvaju jezika. Naime, kako se oba jezika nastoje učiniti lakšima za korištenje, tako u svojim sustavima imaju ugrađen svaki svoj sakupljač smeća koji s vremenom oslobađa nekorištene resurse kako se programer ne bi trebao zamarati njihovim otpuštanjem. No, s obzirom da je moguće, kao što je već spomenuto, iz Prologa putem ovog sučelja kreirati i upravljati objektima iz Java okoline, čime se oni nalaze u Java okolini, dok im se referenca čuva od strane sakupljača smeća iz sustava Prologa, uspostavljena je komunikacija između sakupljačā smeća tih sustava kako bi se unaprijedilo upravljanje memorijom na razini više aplikacija. (Wielemaker, 2009, str. 220)

3.2.2.3. C++ sučelje za SWI

Korištenjem programa *swipl-ld* koji dolazi uz sâm SWI-Prolog je moguće umrežene programe pisane u jeziku Prolog i C++ integrirati u jednu izvršnu datoteku. Na taj način se ujedno uz samu izvršnu datoteku vežu i ostale potrebne biblioteke kako bi se program učinio što lakše prenosivim. Ipak, s obzirom se C++ programi prevode u izvršni/strojni jezik, nije moguće kao kod prethodnih sučelja prenijeti izvršni program s operacijskog sustava jedne arhitekture na drugi, već je potrebno prenijeti datoteke s izvornim kodom programā oba

programska jezika te ih opet korištenjem *swipl-ld* povezivača (engl. *linker*) pretvoriti u izvršnu datoteku. (SWI-Prolog Manual: Static linking and embedding)

3.2.2.4. C sučelje za SWI

S obzirom da je programski jezik C većinski kompatibilan s jezikom C++, prije pojave prethodnog sučelja je ovo bio glavni način povezivanja s Prologom. Kako C programski jezik ne podržava elemente objektno-orijentirane paradigme, tako programski elementi koji dolaze s ovim sučeljem čine skup funkcija i izvedenih pripadajućih tipova podataka. Kako je prethodno obrađeno sučelje zapravo proširenje i preinaka ovoga, moguće je radi performansi ili bilo kojeg drugog razloga kombinirati korištenje elemenata obiju sučelja bez ikakvih negativnih posljedica. Kao i kod povezivanja s prethodnim sučeljem, *swipl-ld* povezivač se isto tako može iskoristiti i kod programā povezanih ovim sučeljem za pretvorbu cjelokupnog programskog rješenja u jedinstvenu izvršnu datoteku. (SWI-Prolog Manual: The C++ versus the C interface)

4. Programski primjeri povezivanja

U ovom poglavlju se nalaze opisi programskih rješenja koji su izrađeni kao primjeri uz obrađivanu temu, odnosno programski primjeri demonstriraju način povezivanja programskog jezika Prolog, kao predstavnika logičke paradigme, s drugim programskim paradigmama. Ponajviše će biti obrađeno povezivanje s paradigmama za koje u prethodnom poglavlju nije priložen nikakav isječak programskog koda koji bi potkrijepio povezivanje na obrađivani način s opisivanom paradigmom. Gotovo sva rješenja se baziraju na Prologu implementacije SWI, osim primjera s Flora-2 jezikom koji se, kao što je već spomenuto, bazira na XSB Prologu.

4.1. Aplikacija za prikaz rodbinskih odnosa određene osobe

Kako je u većini drugih programskih jezika početni primjer prilikom učenja programiranja izrada programa koji ispisuje na zaslon tekst *Pozdrav svijetu!* (engl. *Hello world!*), tako je u Prologu početnički primjer izrada rodoslovnog stabla. On se bazira na navođenju činjenica koje predstavljaju roditeljske odnose između osoba, kao i spolove svake osobe, te se vrši implementacija pravila/predikata koji predstavljaju uvjete/ciljeve koji moraju biti ispunjeni da bi se odnos jedne osobe prema drugoj mogao nazvati određenim rodbinskim nazivom. (Bratko, 2001, str. 2-3)

Funkcionalnosti razvijene aplikacije su sljedeće mogućnosti: dohvat podataka iz ulaznih datoteka (podržani su XML, JSON i CSV načini strukturiranja podataka), prikaz svih iščitanih osoba uz pripadajuće osnovne podatke (puni naziv, spol, datum rođenja i eventualne smrti), vršenje pretrage osoba po zadanom dijelu imena, te prikaz uloga označene osobe prema drugima s kojima je u rodu. Kako je sustav programskog jezika Prolog prikladan za rad s bazama znanja, definiranje pravila i vršenje upita nad njima, upravo će on biti korišten za navedene svrhe, kao i za pohranu činjenica izvedenih iz iščitanih podataka.

Kako bi se programska rješenja učinila što lakšim za održavanje, preporuča se prilikom razvoja koristiti prikladne, kako uzorke dizajna, tako i arhitekturalne uzorke. Naime, ideja jest podijeliti programsko rješenje na više slojeva, odnosno grupirati međusobno slične programske segmente. Tako je danas jedna od najpopularnijih višeslojnih arhitektura troslojna arhitektura kod koje postoje 3 sloja: prezentacijski, obradbeni (tzv. sloj poslovne logike) i podatkovni. (Fowler, 2002, str. 19-22)

Tako sam prilikom izrade ove aplikacije koristio upravo troslojnu arhitekturu, a radi mogućnosti kasnijeg kvalitetnijeg analiziranja načina povezivanja logičke paradigme s drugima, odlučio sam se izraditi ovu aplikaciju na 3 različita načina. Tako je jedna u potpunosti

bazirana na programskom jeziku Prolog pri čemu je za izradu grafičkog sučelja i obradu događaja korišten XPCE sustav. Kod druge je pak za izradu i rad s grafičkim sučeljem korišten programski jezik Java sa Swing razvojnim okvirom, dok je sve ostalo napisano u Prologu. Kod posljednje varijante ove aplikacije se Java ujedno koristi i za čitanje, obradu ulaznih podataka iz datoteka, pretraživanje osoba te Prologu prosljeđuje važeće činjenice, dok Prolog samo pretražuje koje sve uloge ima označena osoba u svojoj obitelji. Komunikacija između Java i Prolog programa će biti ostvarena korištenjem spomenutog JPL sučelja. Osim zbog demonstracije tog sučelja, jedan od razloga odabira Jave kao programskog jezika koji će pozivati Prolog program jest što je s Javom moguće razvijati višeplatformske (engl. *cross-platform*) aplikacije, odnosno aplikacije koje bez ikakvih preinaka mogu raditi na različitim operacijskim sustavima, sve dok je izvršna okolina za Javu (engl. *Java Runtime Environment*) dostupna i instalirana na operacijskom sustavu.

Sve inačice aplikacija sadrže jednake funkcionalnosti koje su prethodno spomenute. Što se tiče vizualnih razlika samih programskih rješenja, razlika je ponajprije vidljiva u samome izgledu grafičkih obrazaca, kao i načinu na koji sustavi reagiraju na događaje uzrokovane korisnikovom interakcijom sa sustavom. Što se tiče same implementacije i izvornog koda programa (odnosno više njih), razlike između prve i treće obrađene inačice su ogromne u usporedbi svake od njih sa drugom. Kao primjer se mogla razviti i varijanta aplikacije napravljena u potpunosti u Javi, odnosno kod koje su svi slojevi arhitekture aplikacije napravljeni u tome jeziku, no s obzirom da je tema ovog rada povezivanje logičke paradigme s ostalima, a kako Java uopće ne podržava logičku paradigmu, takav programski primjer ne bi baš bio previše relevantan.

Kreirane varijante aplikacije sam ujedno pokušao i usporediti prema nekim kriterijima koji smatram relevantnima. Tako sam se odlučio za odabir sljedećih kriterija: broj linija izvornog koda, mogućnost prilagodbe grafičkih kontrola te prosječno vrijeme potrebno za pokretanje svake inačice. Kod izračuna broja linija koda u obzir nisam uzimao izvorni kôd (pa čak i ako je bio dostupan) iskorištenih ugrađenih i vanjskih modula, dok kod razmatranih datoteka nisam brojao linije s programskim komentarima te eventualni veći broj praznih linija između predikata (Prolog) i metoda (Java) sam tretirao kao jednu. Ujedno sam proučavanjem dostupnih tuđih Prolog programa ustanovio da je praksa navoditi točku kod definicija tijela predikata u istom redu gdje i posljednji njegov cilj, a ne ispod (čime se točka nalazi sama u redu) zbog čega sam odlučio ne brojati takve redove. Kod mogućnosti prilagodbe grafičkih kontrola sam definirao skalu od 3 razine pri čemu sam kao moguće vrijednosti definirao „Slaba“, „Osrednja“ i „Jaka“ ovisno o količini svojstava koje je moguće definirati, kao i događaja koje je moguće obraditi. Kao zadnji kriterij je odabrano prosječno vrijeme pokretanja koje predstavlja aritmetičku sredinu 5 izvršenih mjerenja vremena. Vrijeme potrebno za pokretanje aplikacije sam definirao

kao vrijeme proteklo od pritiska tipke *Enter* na prečicu za pokretanje aplikacije do pojave grafičkog obrasca aplikacije. Testiranje je izvršeno na Windows 10 operacijskom sustavu pod jednakim okolnostima za sve inačice aplikacije, a rezultate provedene usporedbe izlažem u Tablici 2.

Tablica 2: Višekriterijska usporedba inačica aplikacije za prikaz rodbinskih odnosa osobe

Inačica aplikacije	**Broj linija izvornog koda**	**Mogućnost prilagodbe grafičkih kontrola**	**Prosječno vrijeme pokretanja**
Potpuno u Prologu	380 (Prolog)	Slaba	0.75 s
Prezentacija u Javi	255 (Java) + 245 (Prolog)	Jaka	1.23 s
Prezentacija i obrada u Javi	455 (Java) + 125 (Prolog)	Jaka	0.97 s

Razlog zbog kojeg je prva inačica aplikacije kraća od ostalih je zbog korištenja samih mehanizama povezivanja, ali i zbog toga što je Prolog jezik deklarativne paradigme kod koje se smatra da programer ne treba definirati prilikom rješavanja problema kako nešto postići, već samo specificirati što želi postići. Varijanta aplikacije koja je u potpunosti napisana u Prologu je loše ocijenjena po drugom kriteriju zbog nemogućnosti pretplate na događaje poput pritiska tipke, ili pak odabira retka ili ćelije tablice, dok kod Swinga tih problema nije bilo. Broj atributa kojima se može definirati izgled grafičkih kontrola je također poprilično malen u odnosu na one koje nudi Swing okvir kod Jave. Razlog dominacije prve inačice programa kod vremena potrebnog za njeno okretanje leži ponajprije u veličini i složenosti Java izvršne okoline koja se mora pokrenuti prilikom pokretanja, dok je s druge strane XPCE okolina memorijski mnogo lakša. Ujedno napominjem da su moguća mala odstupanja od stvarnih vrijednosti kod broja linija koda zbog navedenog većeg broja uvjeta brojanja linija, kao i kod izračuna prosječnog vremena pokretanja programā zbog korištenja ručne štoperice i nezanemarivog vremena reagiranja na pojavu grafičkog obrasca aplikacija.

U nastavku slijedi kratki opis svake varijante te ujedno i popis korištenih paradigmi zajedno s ostvarenim načinom povezivanja.

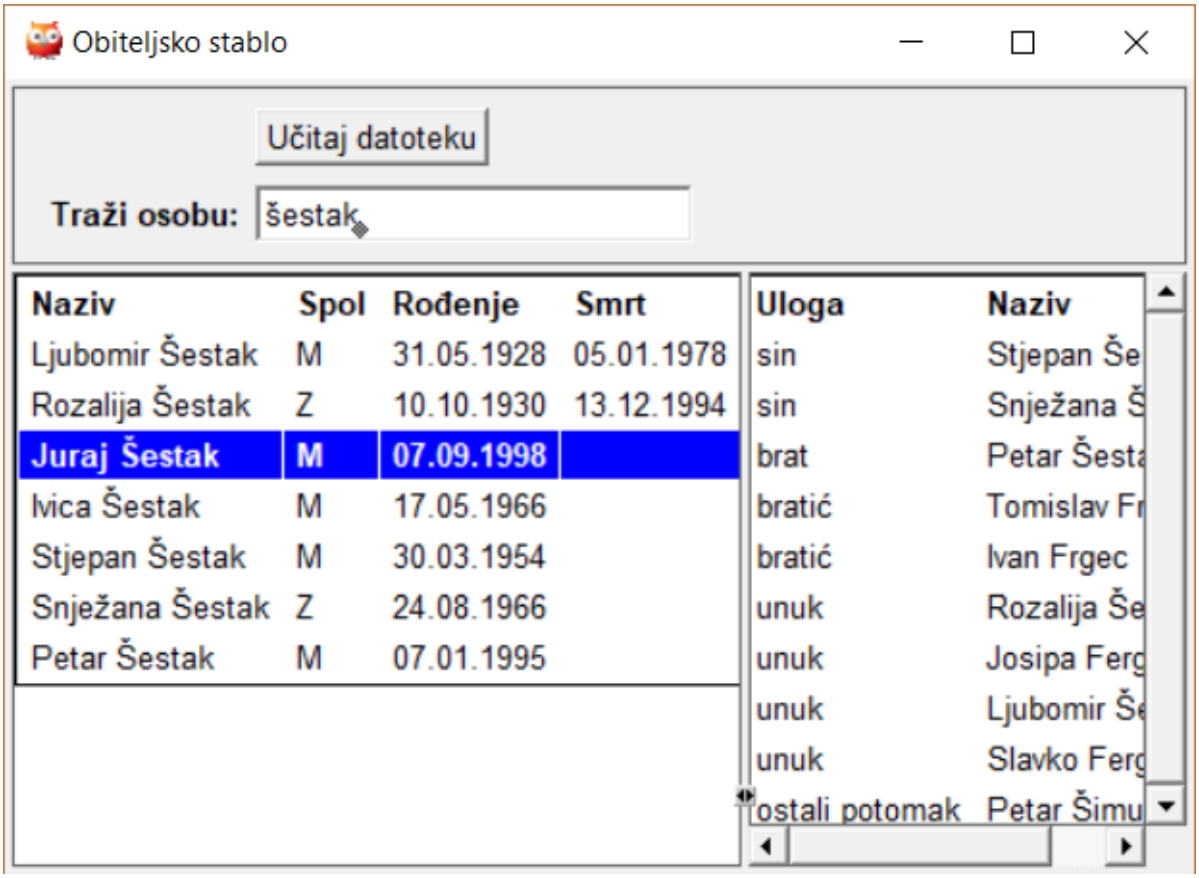
4.1.1. Inačica potpuno kreirana u Prologu

Ova inačica aplikacije je u potpunosti izrađena korištenjem programskog jezika Prolog implementacije SWI. Za izradu grafičkog sučelja i obradu korisničkih događaja je korišten obrađeni modul XPCE. Od elemenata objektno-orijentirane paradigme, kod ove aplikacije nisu iskorištene njegove pogodnosti vezane uz definiranje korisničkih klasa, ali se koriste ugrađene klase iz XPCE sustava na način da se kreiranjem njihovih objekata stvaraju grafički obrasci i njima dodaju korisničke kontrole, a brisanjem zatvaraju i briše se sadržaj sa njih. Operacije čitanja i mijenjanja objekata su uvelike prisutne prilikom obrade događaja kada se treba iščitati

promjena koju je korisnik učinio nad kontrolom koja je uzrokovala nastup događaja, kao i kada se treba pravovaljano ažurirati sadržaj zaslona aplikacije.

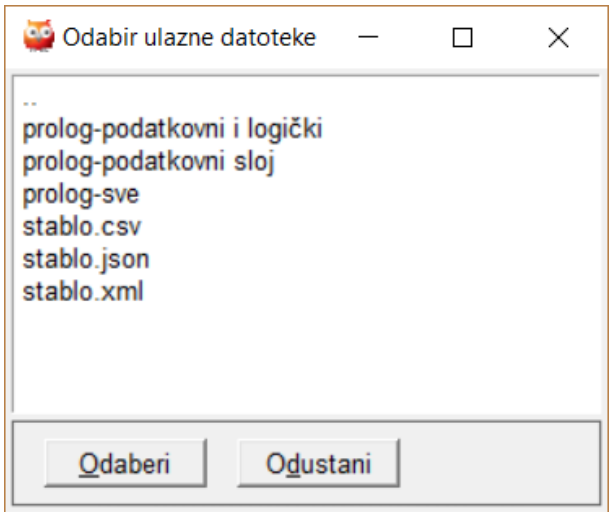
XPCE je također korišten za povezivanje akcija sa samim događajima prilikom nastupa kojih se one trebaju izvršiti. Primjeri događaja koji su u aplikaciji obrađeni su pritisak tipke za učitavanje datoteke s ulaznim podacima, pritisci tipki u prozoru za odabir datoteke s ulaznim podacima kao i odabir selektirane datoteke ili direktorija, selektiranje retka tablice s osobom čije nas rodbinske uloge zanimaju te unos teksta po kojem se želi vršiti pretraga iščitanih osoba.

Kako za korisničke kontrolne elemente XPCE sustava ne postoji velik broj ugrađenih događaja na koje se je moguće pretplatiti, tako je zbog nepostojanosti događaja promjene sadržaja kontrole za tekstualni unos za provedbu filtriranja osoba potrebno pritisnuti tipku *Enter* kako bi se ažurirale tablice shodno postavljenom unosu. Isto tako zbog nemogućnosti osluškivanja događaja klika mišem na redak ili ćeliju tablice, kao zaobilazno rješenje se označavanje osobe vrši klikom isključivo na tekstualni sadržaj ćelije unutar željenog retka. Na Slici 3 je prikazan izgled ove inačice aplikacije nakon što se iščitaju podaci, izvrši filtriranje i odabere osoba čije se rodbinske uloge žele pregledati.



Slika 3: Izgled glavnog prozora inačice aplikacije za pregled rodbinskih odnosa osobe potpuno izrađene u Prologu

Prilikom izrade ove aplikacije XPCE/Prolog tehnologijom, najveću zamjerku ipak pripisujem manjku ugrađenih korisničkih kontrola. Naime, ne postoji klasa interaktivne tablice za rad s podacima koja bi bila prikladna za prikaz podataka o osobama i za njihovo označavanje, već je navedeno postignuto korištenjem statičke tablice za prikaz podataka zbog čega je prisutan spomenuti problem kod označavanja retka željene osobe. Isto tako ne postoji ni klasa koja predstavlja gotovi prozor za odabir datoteke s datotečnog sustava zbog čega je bilo potrebno ni iz čega sastaviti prozor koji će omogućavati navigaciju kroz direktorije datotečnog sustava, kao i odabir željene datoteke. Na Slici 4 prilažem njen izgled koji je veoma primitivan, ali služi svrsi.



Slika 4: Prikaz prozora za odabir datoteke s ulaznim podacima kod inačice aplikacije za pregled rodbinskih odnosa osobe izrađene u potpunosti s programskim jezikom Prolog

Datoteke koje nemaju sufiks „.csv“, „.json“ ili „.xml“ se ni ne prikazuju s obzirom da ne mogu biti odabrane, budući da se njihov sadržaj vrlo vjerojatno ne bi mogao obraditi. Prikazane stavke koje nemaju nijedan od spomenutih sufiksa u nazivu zapravo predstavljaju direktorije čijim odabirom se prikazuje njihov sadržaj. Kao prva stavka je uvijek ponuđena opcija za pozicioniranje u roditeljski direktorij trenutnoga, osim u slučaju kada je trenutni direktorij ujedno i korijenski. Samim odabirom podržane datoteke se u glavnome prozoru prikazuju iščitani podaci datoteke, baš kao što je prikazano na Slici 3.

4.1.2. Inačica s prezentacijskim slojem kreiranim u Javi

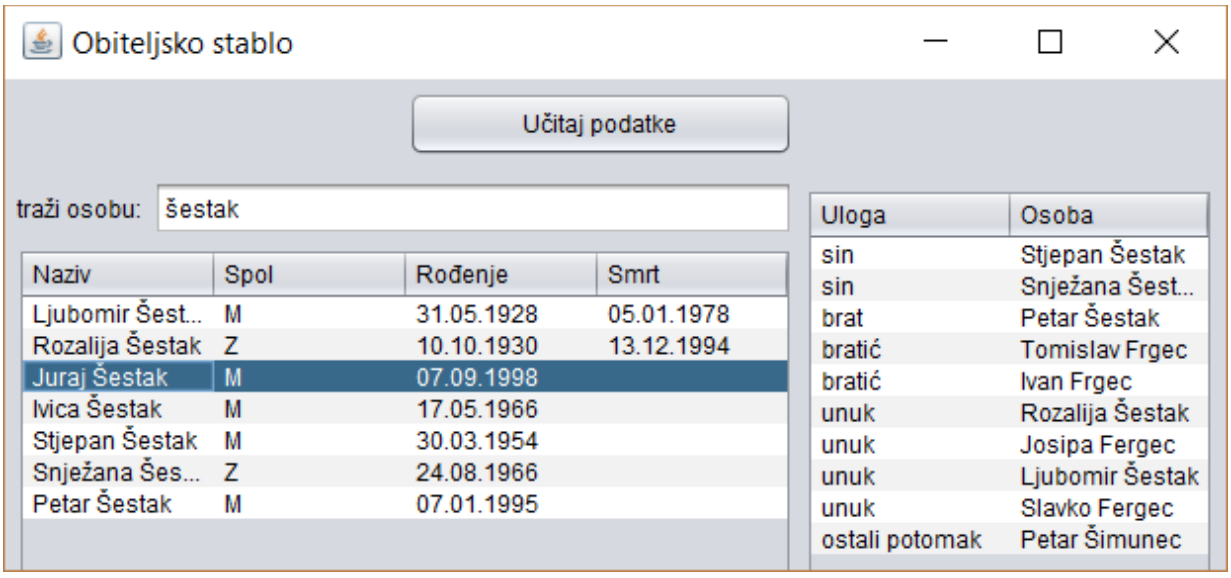
Kod ove inačice je za realizaciju gotovo cijelog programskog rješenja korišten programski jezik Prolog implementacije SWI, osim za izradu samog grafičkog sučelja aplikacije za što je korišten jezik Java sa Swing razvojnim okvirom. Dakle, ovo rješenje se tako zapravo sastoji od 2 programa (jedan s Java kodom, a drugi s Prolog kodom) pri čemu Java program

korištenjem JPL sučelja poziva Prolog program kojem prosljeđuje traženi upit, Prolog vrši ispitivanje upita te vraća potencijalne rezultate natrag pozivajućem Java programu.

Neki od upita koje Java program u ovoj inačici aplikacije šalje izvršavanom Prolog programu su slanje zahtjeva za čitanje sadržaja datoteke s proslijeđene apsolutne adrese, slanje unesenog teksta prema kojem se treba vršiti filtriranje te povrat popisa svih osoba valjanih naziva, kao i slanje zahtjeva s nazivom označene osobe za pronalazak svih njegovih rodbinskih uloga.

Kod programskog rješenja pisanog u Prologu se koriste isključivo elementi logičke paradigme, dok pozivajući program pisan u Javi koristi elemente objektno-orijentirane paradigme te paradigme programiranja pogonjenog događajima. S obzirom da je Java u potpunosti bazirana na objektno-orijentiranoj paradigmi te u jako velikoj mjeri podržava sve njezine koncepte, prilikom izrade njenog dijela rješenja su korišteni elementi poput izrade korisničkog obrasca nasljeđivanjem iskoristivih svojstava i metoda iz postojećih kontrola, dinamičkog odabira metoda povezivanjem metoda implementiranih od strane natklase, kao i dinamičkog polimorfizma nadjačavanjem metoda od natklase.

Swing razvojni okvir naspram XPCE-a nudi kontrolu za odabir datoteke s datotečnog sustava kojoj se ujedno lako može proslijediti način filtriranja datoteka koje treba prikazati i koje je moguće odabrati, a nudi i kontrole za rad s podacima u tablici za koje je pritom podržan široki spektar događaja na koje se je moguće pretplatiti. Kod ove inačice aplikacije se tablica s osobama ažurira odmah kako korisnik unosi svako slovo pojma pretraživanja u tražilicu. Prikaz rješenja je vidljiv na Slici 5.



Slika 5: Izgled glavnog prozora inačice aplikacije za pregled rodbinskih odnosa osobe s Java grafičkim sučeljem

4.1.3. Inačica s prezentacijskim i obradbenim slojem kreiranim u Javi

Posljednja inačica je vizualno identična prethodnoj, stoga je njen prikaz također vidljiv na Slici 5. Ono u čemu se razlikuje su pozadinski procesi, odnosno drugačija je raspodjela poslova između dva povezana programa. Naime, s Prolog programa je u ovom primjeru skinut dio zadaća te je premješten na izvršavanje na strani programa pisanog u Javi kojeg izvršava istoimena izvršna okolina.

Zadaci koji su prebačeni na stranu Java programa su čitanje i obrada odabrane ulazne datoteke, kao i filtriranje popisa osoba iščitanih iz spomenute datoteke prema postavljenom pojmu pretraživanja. Kako se na strani Java programa dohvaćaju osnovni podaci o osobama iz specificirane ulazne datoteke, iz Java programa se ujedno uvrštavaju činjenice u bazu znanja Prologa pozivanjem upita nad *assert/1* predikatom putem JPL sučelja pri čemu se umetane činjenice formiraju korištenjem funkcija za rad sa znakovnim nizovima.

Paradigme koje su korištene u ovoj varijanti rješenja su također iste kao i u prethodnima, a valja spomenuti da se ovdje od koncepata objektno-orijentirane paradigme koristi i enkapsulacija. Navedena je iskorištena kod definiranja klasa putem kojih se vrši deserijalizacija podataka iz ulazne datoteke, odnosno za kreiranje objekata klase *Osoba* , pri čemu su *get* i *set* metode njenih atributa javne, dok su sami atributi u koje se pohranjuju konkretne vrijednosti privatni.

4.2. Aplikacija za prikaz svih podređenih zaposlenika

Problem koji ova aplikacija rješava je veoma sličan onome koji je rješavala prethodna aplikacija, odnosno jedan od potproblema je kod prošle aplikacije bio prikazati sve osobe kojima je označena osoba predak ili potomak, a rješavanje tog problema se kod programskih jezika logičke paradigme bazira na rekurziji. Svrha ove aplikacije je demonstrirati sposobnosti i načine korištenja modulā Prologa za podršku elemenata objektno-orijentirane paradigme čiji moduli iz prijašnjeg poglavlja su bili jedini za koje nije priložen nikakav isječak programskog koda. Razlog zbog kojeg nisu bili priloženi u prethodnom poglavlju jest zato što nisu baš malog obujma te bi stoga previše prostora zauzimali da su navedeni kao isječci koda. Tako će se i ova aplikacija sastojati od više varijanti, tj. isti problem će biti riješen više puta pri čemu svaki put s jednim od modula.

Pretpostavimo da su poslovna pravila koja vrijede i koja će ujedno biti predefinirana u izvornom kodu samih programa ta da u nekom poduzeću, čija se organizacijska struktura modelira, rade različiti zaposlenici: programeri, građevinari te voditelji različitih razina. Prilikom evidentiranja podataka o zaposlenicima treba za sve zabilježiti njihovo ime, dok dodatno za

programere treba evidentirati korištene tehnologije kojima se služe, za građevinare domenu u kojoj djeluju, dok za voditelje naziv odjela kojeg vode te listu zaposlenika koji su im izravni podređeni. Za kraj još pretpostavimo da je potrebno omogućiti izračun ukupnog broja podređenih svakoga voditelja, kao i prikaz svih takvih zaposlenika.

Varijante ove aplikacije su također podvrgnute analizi. S obzirom da prilikom ispitivanja vremena potrebnog za izvršavanje svakog od upita sam kao rezultat dobivao da je upit izvršen za manje od 1 milisekunde, odlučio sam kod ovog primjera ne isprobavati taj kriterij. I ovaj puta se koristi indikator koji predstavlja broj linija izvornog koda te se ovdje također koristi isti način brojanja linija programskog koda, tj. primjenjuje se kod oba jezika način brojanja koji je kod prethodne analize korišten za Prolog. Drugi kriterij po kome se ovdje vrši usporedba je broj znakova svih upita i to nakon što se maksimalno skrate nazivi varijabli te uklone nepotrebne praznine. Razlog tog odabira je što za upite nema smisla provjeravati broj linija/redaka s obzirom da se upiti najčešće navode kao samo jedan redak pa se širina upita čini kao prikladnija opcija. Rezultati mjerenja tih karakteristika su iskazani u Tablici 3.

Tablica 3: Višekriterijska usporedba inačica aplikacije za prikaz svih podređenih zaposlenika

Inačica aplikacije	**Broj linija izvornog koda**	**Broj znakova svih upita nakon minimizacije imena varijabli**
Prolog s XPCE	87	36 + 56 + 70
Flora-2	47	33 + 24 + 39

Iz navedenih rezultata se može uočiti da je Flora-2 iz svih promatranih aspekata nedvojbeno prikladnija za rješavanje ovog problema. Naime, problem je bio rješiv s manje linija programskog koda zbog specijalizirane sintakse programskog jezika Flora-2 koja je prilagođena za izradu modela podataka i rad s objektima. Kod Prologa s XPCE modulom se s druge strane zahtijevala izrada konstruktora kako bi se instanciranjem klasa mogli inicijalizirati njegovi atributi, a najveći utjecaj je imala nepodržanost rekurzivnih metoda zbog koje tražena metoda mora pozvati predikat koji onda dalje rekurzivno traži rješenje. Što se pak tiče širine upita, kod Prologa su se za dohvat podataka trebali koristiti specijalni predikati, kako za sâm dohvat korištenjem *get/3*, tako i za konverziju podataka iz specijalnog tipa *chain* (namijenjenog za pohranu kolekcija kod objekata klasa) u standardnu listu.

4.2.1. Inačica napisana korištenjem Prologa s XPCE modulom

Kao što je već ranije spomenuto, korištenjem pogodnosti XPCE-a se mogu definirati konstruktori i destruktori klasa koji nisu zapravo za rješavanje zadanog problema ni toliko potrebni: desktruktori se ni ne rabe, dok su vrijednosti kreiranim objektima mogle biti postavljene korištenjem *send/2* predikata umjesto konstruktorom. Jedan od razloga zbog kojeg

se u izvornom kodu ove varijante rješenja nalazi nekolicina predikata izvan samih klasa jest što, kako je već spomenuto, metode koje je potrebno kreirati trebaju rješavati problem koji se rješava rekurzivno, a s obzirom da metode kod XPCE-a ne mogu biti rekurzivne, one pozivaju uobičajene predikate Prologa koji onda rekurzivno vrše traženu pretragu ili kalkulaciju.

U nastavku prilažem kratki isječak dijela rješenja s kojeg je vidljivo kako se vrši specijalizacija klase korištenjem nasljeđivanja, kako se navode varijable/atributi klasa, poziva implementacija metode naslijeđene natklase, definira konstruktor i inicijaliziraju atributi te kako se vrši instantacija/kreiranje objekata klase.

```
:- use_module(library(pce)).
% definicija klase 'zaposlenik'
:- pce_begin_class(zaposlenik(naziv), object).
    % atributi klase
    variable(naziv,         name,    both,    "Naziv osobe").
    variable(podredjeni,    chain,   both,    "Lista podređenih").

    initialise(Subjekt, Naziv:name) :->        % definicija konstruktora
        send(Subjekt, naziv, Naziv),
        chain_list(Podredjeni, []),
        send(Subjekt, podredjeni, Podredjeni).

:- pce_end_class.

% definicija klase 'gradjevinar' koja nasljeđuje atribute i konstruktor
:- pce_begin_class(gradjevinar(domena), zaposlenik).

    variable(domena,        string, both,    "Područje specijalizacije").

    initialise(Subjekt, Naziv:name, Domena:string) :->
        send_super(Subjekt, initialise, Naziv),   % poziv roditeljske metode
        send(Subjekt, domena, Domena).

:- pce_end_class.

% kreiranje globalne instance/objekta klase 'gradjevinar'
:- pce_global(@ivan, new(gradjevinar('Ivan Ivanovic', "niskogradnja"))).
```

Sivom bojom su označeni komentari koji označavaju što predstavlja koji dio programskog koda. Komentare ujedno predstavlja 4. argument terma *variable/4* koja ukratko opisuje koja je namjena pojedinog atributa klase. U slučaju da se kod klase *gradjevinar* nije ponovno definirala specijalna metoda s nazivom *initialise*, bilo bi moguće stvarati objekte te klase korištenjem baznog/roditeljskog konstruktora, čime bi bilo moguće izbjegnuti definiranje njenog atributa *domena*. U navedenom primjeru je konstruktor roditeljske klase ipak iskorišten, jer ga se poziva iz definicije novog konstruktora kako bi se izbjeglo ponovno navođenje načina na koji treba inicijalizirati atribute *naziv* i *podredjeni*.

Iako je prema početnim poslovnim pravilima po kojima se razvila aplikacija bilo navedeno da podređene mogu imati samo voditelji, radi lakšeg ostvarenja traženih metoda za pronalazak svih podređenih kao i njihovog broja je atribut *podredjeni* stavljen u natklasu koju

naslјeđuju sve ostale konkretne klase, dok je konstruktorom klasama poput *programer* i *gradjevinar* definirano da nijedan stvoreni zaposlenik iz tih klasa nema nijednog podređenog.

Za kraj još navodim načine vršenja traženih upita nad ovim programom. Ako pretpostavimo da se objekt globalnog dosega iz klase *voditelj* nalazi iza varijable s nazivom *@ana*, tada se ukupni broj podređenih osoba tog voditelja dohvaća vršenjem sljedećeg upita:

```
get(@ana, broj_ukupno_podredjenih, UkupanBrojPodredjenih).
```

Za dohvat identifikatora globalnih varijabli koje se referenciraju na objekte koji predstavljaju sve podređene određenog voditelja, potrebno je izvršiti sljedeći upit:

```
get(@ana, svi_podredjeni, _SviPodredjeniChain),
chain_list(_SviPodredjeniChain, _SviPodredjeniLista),
member(PodredjeniZaposlenik, _SviPodredjeniLista).
```

Razlog zbog kojeg je prikaz ovakvog oblika je zbog toga što XPCE za pohranu kolekcija koristi tip *chain* umjesto klasične liste iz Prologa pa je potrebno vršiti konverziju. Ako se pak žele dohvatiti puni nazivi podređenih zaposlenika, tada upit treba malo modificirati da bude sljedećeg oblika:

```
get(@ana, svi_podredjeni, _SviPodredjeniChain),
chain_list(_SviPodredjeniChain, _SviPodredjeniLista),
member(_PodredjeniZaposlenik, _SviPodredjeniLista),
get(_PodredjeniZaposlenik, naziv, NazivPodredjenogZaposlenika).
```

4.2.2. Inačica napisana korištenjem Flora-2 programskog jezika

Iako je često nezgodno vršiti usporedbu opsegā izvornog koda programskih rješenja (posebice ako su pisani različitim programskim jezicima) zbog mogućnosti pisanja jedne instrukcije kroz više redaka kao i više instrukcija u jednom retku, valja spomenuti da je broj linija izvornog programskog koda ove varijante rješenja gotovo dvostruko manji od izvornog koda prethodne. Ako se pritom pogleda i sâm programski kod, može se uočiti kako kod ove varijante izgleda čišće, u smislu da se koristi manji broj operatora te nije potrebno vršiti nekoliko poziva predikata kako bi se izvršila konverzija iz jednog tipa u drugi. Neki od razloga tome su to što Flora-2 podržava logiku s okvirima i tehniku metaprogramiranja te je ujedno zasebni programski jezik koji se naknadno prevodi u Prolog kôd pa programer prilikom samog razvoja ne mora voditi brigu o značajkama Prologa koje nisu prikladne za objektno-orijentiranu paradigmu, već logičku.

Korištenjem jezika Flora-2 se isječak koda koji je bio priložen kod XPCE-a može navesti sa sljedećim nizom naredbi za koje smatram da ih nije potrebno dodatno objašnjavati.

```
gradjevinar::zaposlenik.

zaposlenik[
    naziv => _string,
    podredjeni => _list
].

gradjevinar[
    domena => _string
].

ivan:gradjevinar[naziv->'Ivan Ivanovic', domena->'niskogradnja'].
```

Iz primjera se može uočiti da za potrebe rješavanja ovakvih problema, odnosno za vršenje modeliranja odnosa između klasa namijenjenih za pohranu podataka i zaključivanje nad njima, Flora-2 posjeduje specijaliziranu i prikladnu sintaksu. Identifikator objekta je ovom primjeru znakovni niz, no može biti bilo koji podatak bilo kojeg od podržanih tipova koji su prethodno navedeni u radu.

Vršenje upita se vrši jednako elegantno kao što se vrši izrada podatkovnog modela iz prethodnog primjera. Tako ako pretpostavimo da je identifikator objekta koji predstavlja voditelja *ana* kao što je bio i kod vršenja upita s XPCE/Prologom, primjer upita za dohvat ukupnog broja podređenih zaposlenika nad priloženim cjelokupnim programom bi bio sljedeći:

```
ana[ukupan_broj_podredjenih->?ukupan_broj_pronadjenih].
```

Kao i prije, valja napomenuti da je ovo samo poziv prethodno definiranog atributa, dok se njegova implementacija nalazi u samom izvornom kodu programa koji nije priložen u ovom radu, već dolazi uz njega. Dohvat identifikatora svih zaposlenika koji su podređeni ovom istom voditelju bi se izvršio sljedećim upitom:

```
ana[svi_podredjeni->?svaki_podredjeni].
```

Ako bi se pak htjeli dohvatiti puni nazivi svih podređenih zaposlenika subjekta nad kojim se vrši upit, tada bi se pak ovaj upit proširio na sljedeći način:

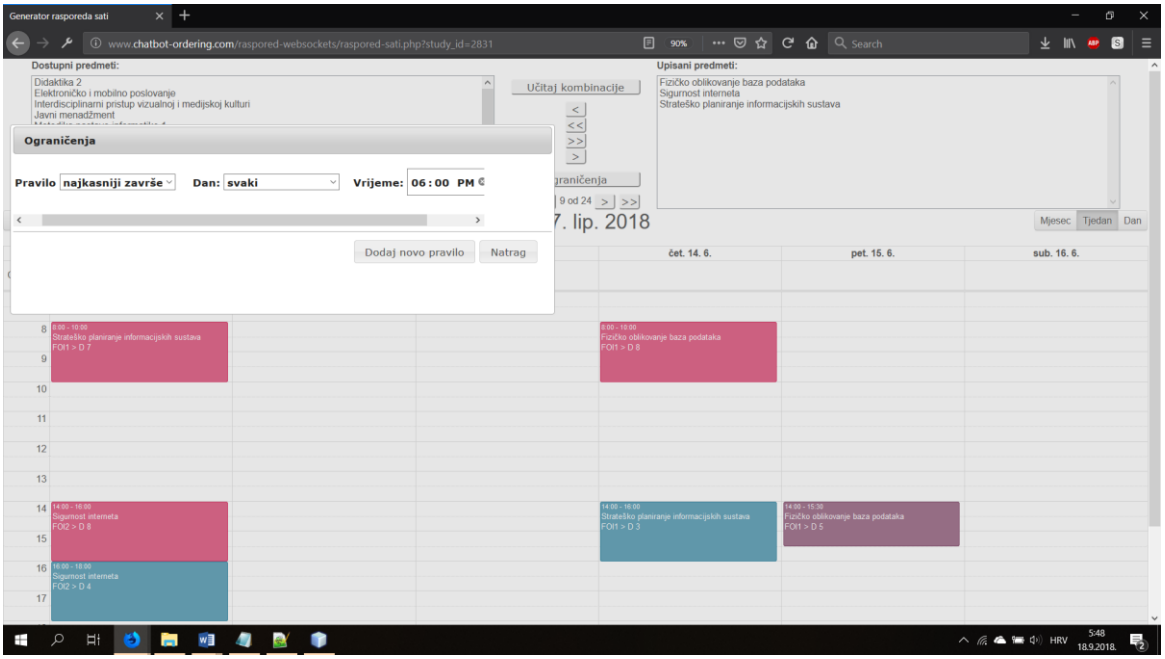
```
ana[svi_podredjeni->?_svaki_podredjeni], ?_svaki_podredjeni[naziv->?naziv].
```

4.3. Aplikacija za rješavanje problema raspoređivanja nastave

Ideja za izradu ove aplikacije je proizašla iz problema s kojim se često susreću studenti visokoškolskih obrazovnih institucija, odnosno problemom raspoređivanja nastave iz skupa često velikog broja raspoloživih termina nastave pri čemu za svaki pojedini predmet ili oblik nastave vrijede različita pravila vezana uz dolaznost na nastavu. Aplikacija je namijenjena za korištenje studentima na mom matičnom fakultetu, odnosno na Fakultetu organizacije i

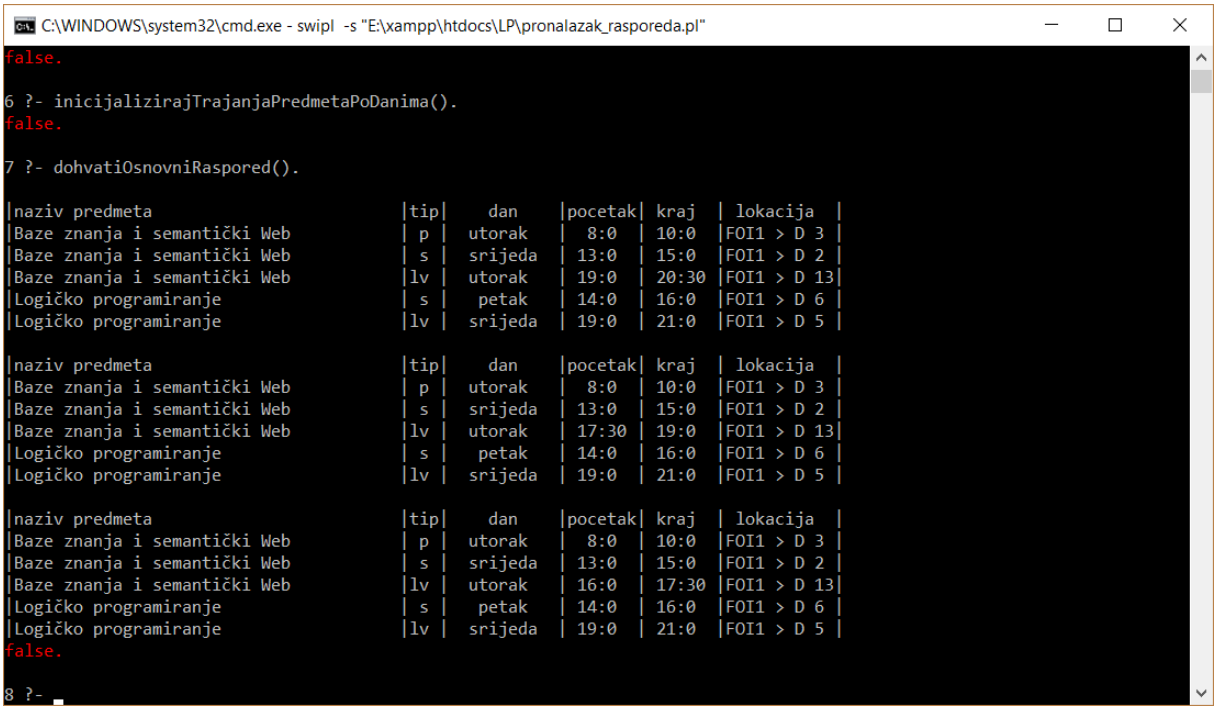
informatike, Sveučilišta u Zagrebu. Izvorni programski kôd, osim što je priložen uz rad, je ujedno dostupan na GitHub web-servisu za verzioniranje programskih rješenja na adresi https://github.com/zeko868/schedule-generator, a repozitorij rješenja se sastoji od 3 grane (engl. *branches*) pri čemu svaka predstavlja jednu od varijanti rješenja.

Ova aplikacija se ponajprije razlikuje od prethodno obrađenih po tome što je web-aplikacija, odnosno njenom korisničkom sučelju se pristupa korištenjem web-preglednika. Razlog odabira web-preglednika kao platforme na kojoj će se izvršavati aplikacija jest mogućnost pružanja određene funkcionalnosti bez da se pritom od krajnjeg korisnika aplikacije zahtijeva išta osim suvremenog web-preglednika za njeno korištenje. Druga posebitost ove aplikacije je što se kao način povezivanja Prologa i ostatka web-aplikacije koristi preostali obrađeni način povezivanja logičke paradigme s ostalima. Naime, kada je putem web-aplikacije poslužitelju upućen zahtjev za generiranjem skupa zadovoljavajućih kombinacija rasporeda nastave za proslijeđena ograničenja, PHP skripta koja se nalazi na pozadinskoj strani sustava poziva interpreter SWI-Prologa s inicijalnim podacima poput korisničkih zahtjeva nad pripadajućim programom te se osluškuju podaci koje on tijekom izvođenja šalje na svoj standardni izlazni tok. Može se zaključiti da je riječ o povezivanju PHP-a s Prologom korištenjem jednosmjerne komunikacije jer se podaci (generirane kombinacije rasporeda) šalju samo u smjeru prema PHP skripti, dok podaci jedino do Prologa stižu prilikom njegovog pokretanja kada mu se kao argumenti šalju činjenice o predmetima i pravilima te pozivi predikata koje je potrebno izvršiti. Na Slici 6 je prikazan primjer jedne od implementacija, a identičnog izgleda su i ostale varijante.

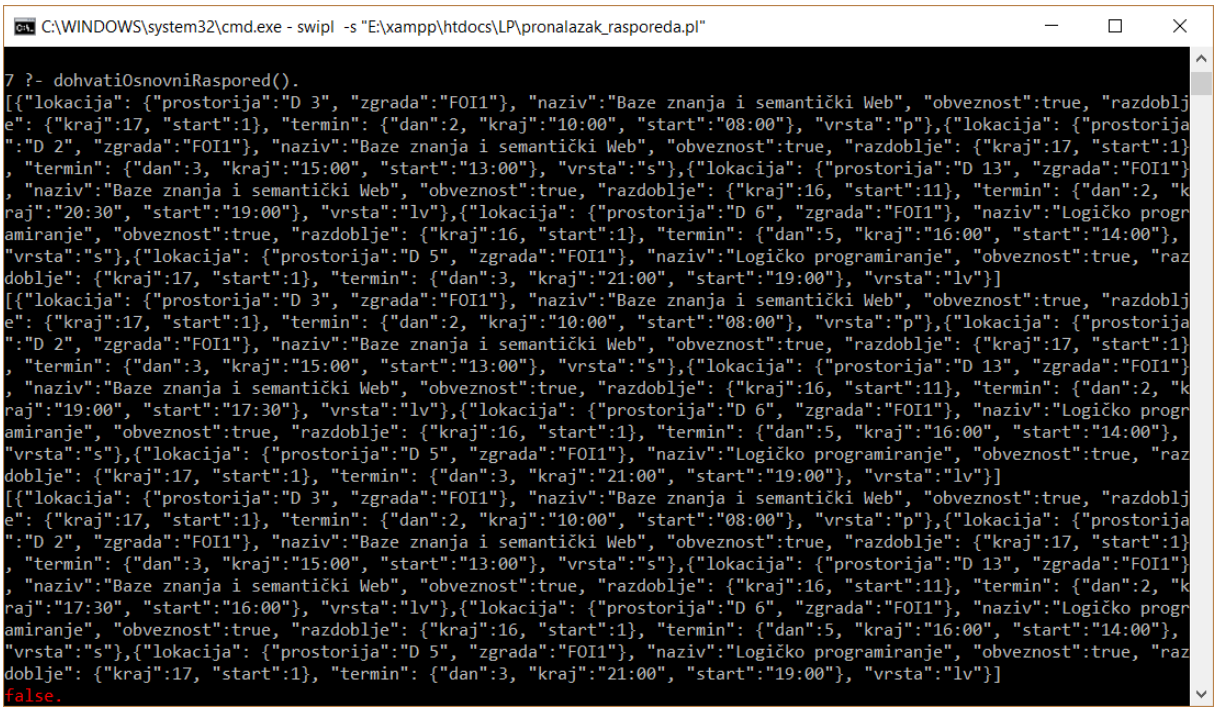


Slika 6: Prikaz web-aplikacija s valjanim kombinacijama rasporeda nastave za zadane parametre

Prolog aplikacija iz cjelokupnog rješenja može zapravo raditi zasebno, odnosno uz pripremljenu ulaznu datoteku s raspoloživim terminima nastave predmetā, ručno definiranje činjenica vezanih uz pohađanje predmeta i željena pravila te poziv predikata za vršenje pretrage zadovoljavajućih kombinacija rasporeda kao i njihov ispis na standardni izlazni tok podataka, tj. na zaslon korisnikovog ekrana. Sâm rezultat je kod te aplikacije inicijalno u tekstualnom obliku pri čemu su podaci strukturirani JSON-formatom, a zamjenom postojećeg pravila baze predikata *nadjiRaspored/1* s pripadajućim priloženim koji je trenutno neaktivan jer je zakomentiran, može se izvršiti prikaz podataka u tabličnom obliku, iako i dalje bez pravog grafičkog korisničkog sučelja. Na Slici 7 tako prilažem prikaz rezultata izvođenja samog Prolog programa na definirane ulazne podatke koji su korisniku u čitljivom obliku, dok je na Slici 8 prikaz rezultata na iste ulazne podatke koji su razumljivi i lako obradivi ostatku cjelokupne web-aplikacije koja koristi te podatke za obradu i generiranje web-stranice s rezultatima pretrage. (Šestak, 2018)



Slika 7: Prikaz rezultnih valjanih kombinacija rasporeda nastave za zadane parametre u tabularnom obliku iz Prolog programa (izvor: Šestak, 2018)



Slika 8: Prikaz rezultnih valjanih kombinacija rasporeda nastave za zadane parametre u JSON-strukturiranom obliku iz Prolog programa (izvor: Šestak, 2018)

Spomenuta ulazna datoteka, koju Prolog program iziskuje kako bi definirao činjenice o raspoloživim terminima nastave predmeta kao i njihovim obveznostima pohađanja, se zapravo automatski generira prilikom korištenja cjelokupne web-aplikacije ako njena najaktualnija verzija već nije prisutna, tj. ukoliko se u direktoriju aplikacije ne nalazi datoteka čiji naziv označava da ona sadrži podatke o nastavi aktualnog semestra tekuće akademske godine. Samo generiranje te ulazne datoteke se vrši primjenom *web-scraping* tehnike nad web-aplikacijom *Nastava* matičnog fakulteta, odnosno skeniranjem i izvlačenjem podataka s web-stranica vezanih uz rasporede nastave i predmete studijskog programa korisnika ove aplikacije. (Šestak, 2018)

Kao što sam već naveo, s obzirom da je ovo programsko rješenje web-aplikacija, jedino što krajnji korisnik treba imati kako bi se koristio aplikacijom jest web-preglednik. Ako čitatelj ovog rada ipak želi da se svi dijelovi cjelokupne aplikacije izvršavaju na njegovom računalu, tada on mora osigurati da se na računalu nalazi odgovarajuća programska potpora. Konfiguracija izvršne okoline na kojoj bi radila cjelokupna aplikacija je navedena u nastavku:

Izvršna okolina na kojoj bi s pouzdanošću trebao cijeli sustav aplikacije funkcionirati je jedna od sljedećih:

- Ubuntu 16.06 LTS s Apache 2.4.18 web-poslužiteljem, PHP interpreterom verzije 7.2 (s uključenim *libxml* modulom verzije 2.9.5 i *cURL* modulom verzije

7.47.0), SWI implementacijom Prologa verzije 7.2.3 i Node.js izvršnim okruženjem verzije 4.2.6

- Windows 10 s Apache 2.4.29 web-poslužiteljem, PHP interpreterom verzije 7.2 (s uključenim *libxml* modulom verzije 2.9.4 i *cURL* modulom verzije 7.56.0), SWI implementacijom Prologa verzije 7.6.0-rc2 i Node.js izvršnim okruženjem verzije 8.9.4

Nakon postavljanja neke od navedenih platformi s pripadajućim programskim paketima, potrebno je osigurati da su putanje do interpretera Prologa i Node.js-a definirane u varijabli okoline s putanjama (tj. `PATH`).

(Šestak, 2018, str. 6)

Nakon izvršenih navedenih koraka, potrebno je još samo osigurati da je web-poslužitelj aktivan te da je izvorni kôd određene inačice ove aplikacije smješten u korijenskom web-direktoriju. Pristupanjem skripti *raspored-sati.php* na domeni web-poslužitelja putem web-preglednika bi se trebala učitati ova web-aplikacija.

U Tablici 4 su prikazane prednosti i nedostaci po inačicama ove web-aplikacije.

Tablica 4: Usporedba inačica aplikacija za rješavanje problema raspoređivanja nastave

Inačica	**Prednosti**	**Nedostaci**
Prikaz svih rezultata (tradicionalni web)	Jednostavna implementacija	Neresponzivnost aplikacije do pronalaska rješenja
Prikaz svakog rezultata na zahtjev (asinkrona komunikacija)	Neopterećenost sustava	Naporno dohvaćanje svih kombinacija, neresponzivnost u slučaju nepostojanja rješenja
Kontinuirani prikaz svih rezultata (komunikacija putem web-utičnica)	Potpuna responzivnost aplikacije	Nepodržanost starijih web-preglednika, nepotrebno opterećivanje web-poslužitelja

U nastavku slijede kratki opisi svake od inačica aplikacija pri čemu će za svaku od njih biti priložena adresa na kojoj će aplikacija biti dostupna narednih nekoliko mjeseci za korištenje, bez ikakve potrebe za konfiguriranjem izvršne okoline na korisnikovom računalu.

4.3.1. Inačica s naknadnim prikazom svih rezultata

Ova implementacija aplikacije je dostupna na poveznici www.chatbot-ordering.com/raspored/raspored-sati.php te predstavlja njezinu najprimitivniju inačicu. Naime, ona nakon zadanog korisnikovog zahtjeva za pretragom valjanih kombinacija rasporeda

nastave osluškuje proces interpretera SWI-Prologa sve dok on ne završi s radom i tek tada generira web-stranicu za korisnika te mu prikazuje pronađene rezultate. Ako nije pronađena nijedna kombinacija koja odgovara na zadane kriterije korisnika, tada se sukladno tome prikazuje odgovarajuća poruka.

Nedostatak ove varijante rješenja je da je aplikacija na korisničkoj strani neresponzivna u periodu nakon slanja zahtjeva poslužitelju pa sve do trenutka kad pristignu rezultati. U slučaju da korisnik želi generirati kombinacije rasporedā iz velikog skupa termina nastave, broj mogućih kombinacija je tada izrazito velik, te ako ograničenjima nisu definirana pravila kojima će se rezati grane prostora pretraživanja, korisniku će aplikacija postati neuporabljiva, dok ne osvježi web-stranicu ili dok se ne ispitaju sve moguće kombinacije i pošalju rezultati.

Kod ovog rješenja, sav posao obrade podataka, uz Prolog program, odrađuje PHP skripta. Na klijentskoj strani se izvršava JavaScript kôd koji omogućava interaktivan rad s korisničkim sučeljem poput premještanja raspoloživih predmeta u upisane i obrnuto, definiranja ograničenja te prikaza i pregledavanja kombinacija rasporeda. Od paradigmi se u rješenju uz logičku paradigmu koristi paradigma programiranja pogonjenog događajima na strani spomenute klijentske JavaScript aplikacije, dok se ujedno nudi podrška objektno-orijentiranoj paradigmi koju podržava PHP, no kod ovog primjera njeni elementi baš i nisu korišteni.

4.3.2. Inačica s prikazom svakog rezultata na zahtjev

Na poveznici www.chatbot-ordering.com/raspored-ajax/raspored-sati.php se može koristiti druga inačice ove aplikacije. Njome se slanjem prvog zahtjeva za dohvaćanje valjane kombinacije nad odabranim skupom predmeta vrši privremena neresponzivnost klijentske strane web-stranice, dok se ne pronađe prva kombinacija. U većini slučajeva postoje valjane kombinacije pa Prolog program vrlo brzo obavijesti PHP skriptu o pronađenoj valjanoj kombinaciji koja se onda prikazuje na strani korisnikovog web-preglednika. Ako je korisnik zainteresiran za ostale kombinacije iz tog istog skupa predmeta, svakim sljedećim pritiskom tipke s oznakom „>“ se pokušava pronaći sljedeća kombinacija rasporeda, bez da se dalje uzrokuje neresponzivnost aplikacije. Razlog tome jest što se dalje zahtjevi i odgovori šalju korištenjem AJAX tehnologije, a njome se postiže asinkrona komunikacija ili tzv. parcijalno osvježavanje sadržaja web-stranice. Dok se pritom čeka odgovor od poslužitelja o sljedećoj kombinaciji rasporeda (ili o nepostojanosti daljnjih valjanih kombinacija), na korisničkom sučelju web-aplikacije se prikazuje poruka o trajanju pretrage. Kada pristigne nova kombinacija, ona se odmah prikazuje preko zaslona te se ažurira ukupan broj kombinacija pronađenih do sada.

S obzirom da se šalje samo jedna kombinacija po zahtjevu, sâm Prolog program je kod ovog rješenja malo promijenjen i prestaje s radom čim pronađe prvu zadovoljavajuću kombinaciju. Kako bi se kod slanja zahtjeva za sljedeću kombinaciju spriječilo da se vrši iznova provjera kombinacija koje su do sada već ispitane, prilikom njegovog pokretanja mu se šalje informacija o posljednje pronađenoj kombinaciji iz čijeg skupa sadržanih termina predmeta se vrši rezanje stabla pretraživanja i brz pronalazak sljedeće kombinacije.

Nedostatak ovog rješenja je što nije moguće dobiti uvid u ukupan broj zadovoljavajućih rasporeda nastave bez da se manualno zatraži svaka postojeća kombinacija. Još jedna mana jest što u slučaju nepostojanja nijedne valjane kombinacije rasporeda za zadani ulazni skup predmeta i ograničenja aplikacija postaje neresponzivna kao i njena prijašnja varijanta. Od programskih paradigmi su korištene sve kao i do sada.

4.3.3. Inačica s kontinuiranim prikazom pronađenih rezultata

Ova varijanta rješenja je dostupna na www.chatbot-ordering.com/raspored-websockets/raspored-sati.php te je najsofisticiranija verzija ove aplikacije. Kod nje se slanjem zahtjeva za generiranje zadovoljavajućih kombinacija rasporeda na strani poslužitelja kreira web-utičnica (engl. *web socket*) te se njezin identifikator šalje kao odgovor natrag klijentskoj aplikaciji. Primitkom te informacije, JavaScript skripta uspostavlja dvosmjernu komunikaciju s iščekujućim zasebnim poslužiteljem koji je pisan u PHP-u te baziran na objektno-orijentiranoj i paradigmi programiranja pogonjenog događajima. Uspostavljanjem komunikacije se pokreće veza između klijentske strane aplikacije i tog poslužitelja koji obavještava korisnika svakih nekoliko sekundi (za sada je definirano 5) o novo-pronađenim kombinacijama rasporeda. Za to vrijeme pretrage se na klijentskoj strani, uz poruku o trajanju pretrage, konstantno ažurira popis ukupno pronađenih zadovoljavajućih kombinacija rasporeda koji se odmah mogu započeti pregledavati.

Prednost ovog pristupa je da u slučaju da postoji mnogo kombinacija koje treba ispitati, korisniku se prije pronalaska svih kombinacija već prikazuju do tada pronađene, a ujedno ni ne treba ručno slati zahtjev za svaku sljedeću kombinaciju, kao što je bio slučaj u prijašnjem primjeru. Također valja napomenuti da je Prolog program kod ovog rješenja identičan kao i u prvom primjeru. Ovoj inačici rješenja se može pripisati da je jedini nedostatak što web-utičnice nisu dostupne na starijim web-preglednicima s obzirom da su relativno nova tehnologija.

5. Zaključak

Prilikom rješavanja problema se odlučujemo na način pristupanja njegovom rješavanju, odnosno odabiremo na što ćemo se fokusirati kako bismo ga što lakše i kvalitetnije riješili. Kod izrade računalnih programa se tako ovisno o vrsti problema odabire prikladna programska paradigma. Problemi koji se rješavaju su često složeni i sastoje se od niza podproblema koji mogu biti međusobno različiti te se rješavanju takvih problema može pristupiti korištenjem više programskih paradigmi.

Logička paradigma programiranja je prikladna za rješavanje specijalnih problema iz domene umjetne inteligencije, a kako bi jezici bazirani na njoj mogli biti primjenjiviji, uz logičku se nudi podrška i za elemente drugih paradigmi. Od programskih jezika primarno logičke paradigme je najznačajniji Prolog, a u radu je posebni naglasak stavljen na implementaciju SWI-Prolog budući da za njega postoji velik broj proširenja. S obzirom da prevladava predrasuda prema ovoj paradigmi po kojoj su jezici bazirani na njoj neuporabljivi za općenitu upotrebu, cilj rada je bio istražiti s kojim sve ostalim programskim paradigmama, te na koje načine, se programski jezik Prolog može povezati.

Kroz rad su tako opisani brojni operatori i predikati iz raznih modula SWI-Prologa kojima se za razvoj programskih rješenja mogu koristiti koncepti drugih programskih paradigmi. Neke od paradigmi koje su tako opisane su proceduralna kojom se omogućava dekompozicija složenog problema kao i naknadna iskoristivost dijelova programa, objektno-orijentirana kojom se mogu izraziti odnosi i interakcije između objekata, logička paradigma s ograničenjima kojom se zadavanjem ograničenja definira prostor pretrage rješenja problema, funkcionalna s kojom je omogućeno definiranje anonimnih funkcija/predikata te paradigma programiranja pogonjenog događajima kod kojeg je fokus na obradi pretplaćenih događaja.

Također je u radu obrađen način povezivanja više programa u cjelinu korištenjem mehanizama poput cjevovoda, kao i korištenjem gotovih sučelja za povezivanje programa određenih jezika, čime se omogućuje da se za izradu programskog rješenja koristi više paradigmi.

Ujedno su predstavljeni izrađeni programski primjeri koji se svi temelje na povezivanju logičke paradigme programiranja s drugima. Jednom od usporednih analiza je zaključeno da se programskim jezikom Prolog pod određenim okolnostima mogu izrađivati aplikacije s grafičkim sučeljem koje su boljih performansi od onih izrađenih u popularnoj Javi, a pritom su napisane i s manje linija programskog koda. Kao završnu misao iznosim da ne treba oklijevati kod odabira logičke paradigme prilikom rješavanja problema čiji dijelovi su prikladni za rješavanje njome s obzirom da se s lakoćom može kombinirati i s drugim paradigmama.

6. Popis literature

[1] Barták, R. (2009) *A Generalized Framework for Constraint Planning*. Prag: Charles University, Department of Theoretical Computer Science

[2] Boer, T.W. (2008) *A Beginners Guide to Visual Prolog 7.1*. Groningen.

[3] Bratko, I. (2001) *Prolog Programming for artificial intelligence* (3. izdanje). Boston: Addison-Wesley Longman Publishing.

[4] Budin, L., Golub, M., Jakobović, D., Jelenković, L. (2013) *Operacijski sustavi*. Zagreb: ELEMENT d.o.o.

[5] Coelho, H., Cotta, J. C. (1988) *Prolog by Example: How to Learn, Teach and Use It*. Heidelberg: Springer-Verlag Berlin.

[6] Calejo, M. (2004) InterProlog: Towards a Declarative Embedding of Logic Programming in Java. U Alferes, J. J., Leite, J. (ur.) *Logics in Artificial Intelligence, 9th European Conference*, pp. 714-717. Heidelberg: Springer-Verlag Berlin.

[7] Čubrilo, M. (1989) *Matematička logika za ekspertne sustave*. Zagreb: Informator.

[8] Demoen, B., Nguyen, P. L. (2001) *Odd Prolog benchmarking*. Leuven: Katholieke Universiteit Leuven, Department of Computer Science.

[9] Gabbrielli, M., Martini, S. (2010) *Programming Languages: Principles and Paradigms*. London: Springer Verlag-London Ltd.

[10] Hanus, M. (2007) Multi-paradigm Declarative Languages. U Dahl, V., Niemelä, I. (ur.) *Proceedings of the 23rd international conference on Logic programming, September 8-13, 2007, Proceedings*. Heidelberg: Springer-Verlag Berlin.

[11] Hinze, A. M. (2010) *Principles and Applications of Distributed Event-Based Systems*. Hershey (Pennsylvania): Information Science Reference.

[12] Jaffar, J., Lassez, J. L. (1987) Constraint Logic Programming. U *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 111-119. München.

[13] Kerrisk, M. (2010) *The Linux programming interface: a Linux and UNIX system programming handbook*. San Francisco: No Starch Press.

[14] Kifer, M., Yang, G., Wan, H., Zhao, C. (2017) *Ergo-Lite (a.k.a. Flora-2): User's Manual*. New York: Stony Brook University, Department of Computer Science.

[15] Kiselyov, O., Byrd, W. E., Friedman, D. P., Chan, C. C. (2008) Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl), U Garrigue, J., Hermenegildo, M. (ur.) *Proceedings of the 9th international conference on Functional and logic programming*. Heidelberg: Springer-Verlag Berlin.

[16] Li, W., Henry, S. (1993) Maintenance Metrics for the Object-Oriented Paradigm. U *Proceedings First International Software Metrics Symposium*, pp. 52-60. Baltimore (Maryland).

[17] Meier, M. (1991) Recursion vs. Iteration in Prolog, U Furukawa, K. (ur.) *Proceedings of the Eighth International Conference on Logic Programming.* Cambridge: MIT Press.

[18] Mitchell, J. C. (2002) *Concepts in Programming Languages*. Cambridge: Cambridge University Press.

[19] Moura, P. J. L. (2015) *Logtalk - Design of an Object-Oriented Logic Programming Language* (Doktorska disertacija). Universidade da Beira Interior, Covilhã.

[20] Naish, L. (1996) *Higher-order logic programming in Prolog*. Melbourne: University of Computer Science, Department of Computer Science.

[21] Stevens, W. R. (1998) *UNIX Network Programming: Interprocess Communications*. Upper Saddle River (New Yersey): Prentice Hall.

[22] Swift, T., Warren, D. S., Sagonas, K., Rao, P., Cui, B., Johnson, E., ... Kifer, M. (2017) *The XSB System: Programmer's Manual*. Preuzeto dana 13.09.2018. s http://xsb.sourceforge.net/manual1/manual1.pdf.

[23] Šestak, P. (2018) *Problem raspoređivanja nastave*, Neobjavljeni seminarski rad. Varaždin: Sveučilište u Zagrebu, Fakultet organizacije i informatike.

[24] Šribar, J., Motik, B. (2010) *Demistificirani C++* (3. izdanje). Zagreb: ELEMENT d.o.o.

[25] Tanenbaum, A. S., van Steen, M. (2002) *Distributed Systems Principles and Paradigms*. Upper Saddle River (New Yersey): Prentice Hall.

[26] Trois, C., Del Fabro, de Bona, L. C. E., Martinello, M. (2016) Programming Languages: Principles and Paradigms, U Ying-Dar, L. (ur.) *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 2687-2712.

[27] Vujošević-Janičić, M., Tošić, D. (2008) The role of programming paradigms in the first programming courses, U Marjanović, M., Kadelburg, Z. (ur.) *The Teaching of Mathematics*, pp. 63-83. Beograd: Društvo matematičara Srbije.

[28] Wielemaker J., Costa V. S. (2011) On the Portability of Prolog Applications. U Rocha, R., Launchbury, J. (ur.) *Practical Aspects of Declarative Languages: 13th International Symposium*. Heidelberg: Springer Science & Business Media.

[29] Wielemaker, J., Anjewierden, A. (2002) *Programming in XPCE/Prolog*. Amsterdam: Universiteit van Amsterdam, Department of Social Science Informatics.

[30] Wielemaker, J. (2009) *Logic programming for knowledge-intensive interactive applications* (Doktorska disertacija). Universiteit van Amsterdam, Amsterdam.

[31] Yeager, D. P. (2014) *Object-Oriented Programming Languages and Event-Driven Programming*. Herndon (Virginia): Mercury Learning & Information.

[32] A SWI-Prolog to Java interface (2004). Preuzeto na datum 18.09.2018. s http://www.swi-prolog.org/packages/jpl/prolog_api/overview.html.

[33] Carbonnelle, P. (2018) *PYPL PopularitY of Programming Language*. Preuzeto dana 26.08.2018. s https://pypl.github.io/PYPL.html.

[34] Computer Hope (15.09.2017.) *Logic Programming.* Preuzeto dana 23.06.2018. s https://www.computerhope.com/jargon/l/logic-programming.htm.

[35] Coherent Knowledge (s.a.) *Comparison of ErgoAI to Flora-2*. Preuzeto dana 30.08.2018. s http://coherentknowledge.com/comparison-of-ergoai-to-flora-2/.

[36] Declarativa (2008) *Installation of InterProlog, a Java + Prolog Interface*. Preuzeto dana 18.09.2018. s http://www.declarativa.com/InterProlog/INSTALL.htm.

[37] The ECLiPSe Constraint Programming System (04.08.2018.) *library(ic)*. Preuzeto dana 31.08.2018. s http://eclipseclp.org/doc/bips/lib/ic/index.html.

[38] Fowler, M. (2002) *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley Professional.

[39] Hendricks, M. (2016) *Apply and compose functions in Prolog*, GitHub repozitorij. Preuzeto dana 12.09.2018. s https://github.com/mndrix/func/.

[40] Janssen, T. (22.12.2017.) *OOP Concepts for beginners: What is polymorphism* Stackify. Preuzeto dana 26.06.2018. s https://stackify.com/oop-concept-polymorphism/.

[41] Moura, P. J. L. (2003) *Logtalk and XPCE*, Stack Overflow – Where Developers Learn, Share & Build Careers. Preuzeto dana 11.09.2018. s https://stackoverflow.com/questions/6095147/logtalk-and-xpce.

[42] Moura, P. J. L. (2018) *Logtalk.* Preuzeto dana 27.08.2018. s https://logtalk.org/.

[43] Pecos Martínez, D. (2014) *Object Oriented Programming vs Functional Programming*. Preuzeto dana 05.09.2018. s https://www.slideshare.net/dpecos/20140401-oopfp-presentacion-33034962.

[44] Ray, T. (s.a.) *Programming paradigms*. Preuzeto dana 24.06.2018. s http://cs.lmu.edu/~ray/notes/paradigms/.

[45] SICStus Prolog (s.a.) *SICStus Prolog – Predicate Index*. Preuzeto dana 13.09.2018. s https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_48.html.

[46] SWI-Prolog Manual (s.a.) *Constraint Logic Programming*. Preuzeto dana 30.08.2018. s http://www.swi-prolog.org/pldoc/man?section=clp.

[47] SWI-Prolog Manual (s.a.) *The C++ versus the C interface*. Preuzeto na datum 18.09.2018. s http://www.swi-prolog.org/pldoc/man?section=cpp-vs-c.

[48] SWI-Prolog Manual (s.a.) *library(yall): Lambda expressions*. Preuzeto dana 13.09.2018. s http://www.swi-prolog.org/pldoc/man?section=yall.

[49] SWI-Prolog Manual (s.a.) *library(process): Create processes and redirect I/O*. Preuzeto dana 13.09.2018. s http://www.swi-prolog.org/pldoc/man?section=process.

[50] SWI-Prolog Manual (s.a.) *Static linking and embedding.* Preuzeto na datum 18.09.2018. s http://www.swi-prolog.org/pldoc/man?section=cpp-linking.

[51] TIOBE Software BV (2018) *TIOBE Programming Community Index.* Preuzeto dana 26.08.2018. s https://www.tiobe.com/tiobe-index//.

[52] Trevisan, Z. (2015) *Programming Paradigms*. Preuzeto dana 29.08.2018. s http://atelier.inf.unisi.ch/~dalsat/sai/projects/2015/html/sw/programming_paradigms.html.

[53] Vuk, A. (2016a) *OOP: Nasljeđivanje [Objašnjeno] [Blog post]*. Preuzeto dana 25.06.2018. s https://almirvuk.blogspot.com/2016/04/oop-nasljeivanje-objasnjeno.html.

[54] Vuk, A. (2016b) *OOP: Enkapsulacija [Objašnjeno] [Blog post]*. Preuzeto dana 24.06.2018. s https://almirvuk.blogspot.com/2016/03/oop-enkapsulacija-objasnjena.html.

[55] Vuk, A. (2016c) *OOP: Polimorfizam [Objašnjeno] [Blog post]*. Preuzeto dana 26.06.2018. s https://almirvuk.blogspot.com/2016/04/oop-polimorfizam-objasnjen.html.

7. Popis slika

Slika 1. Odnos proceduralne, strukturirane i imperativne paradigme prikazan Vennovim dijagramom (Prema: Pecos Martínez, 2014) 5
Slika 2: Prikaz tokova informacija kod dvosmjerne (lijevo) i jednosmjerne međuprocesne komunikacije (desno) (Prema: Kerrisk, 2010) 30
Slika 3: Izgled glavnog prozora inačice aplikacije za pregled rodbinskih odnosa osobe potpuno izrađene u Prologu 39
Slika 4: Prikaz prozora za odabir datoteke s ulaznim podacima kod inačice aplikacije za pregled rodbinskih odnosa osobe izrađene u potpunosti s programskim jezikom Prolog 40
Slika 5: Izgled glavnog prozora inačice aplikacije za pregled rodbinskih odnosa osobe s Java grafičkim sučeljem 41
Slika 6: Prikaz web-aplikacija s valjanim kombinacijama rasporeda nastave za zadane parametre 47
Slika 7: Prikaz rezultnih valjanih kombinacija rasporeda nastave za zadane parametre u tabularnom obliku iz Prolog programa (izvor: Šestak, 2018) 48
Slika 8: Prikaz rezultnih valjanih kombinacija rasporeda nastave za zadane parametre u JSON-strukturiranom obliku iz Prolog programa (izvor: Šestak, 2018) 49

8. Popis tablica

Tablica 1: Usporedba nekih programskih paradigmi 3
Tablica 2: Višekriterijska usporedba inačica aplikacije za prikaz rodbinskih odnosa osobe ..38
Tablica 3: Višekriterijska usporedba inačica aplikacije za prikaz svih podređenih zaposlenika 43
Tablica 4: Usporedba inačica aplikacija za rješavanje problema raspoređivanja nastave 50