Erlang - funkcionalno rešenje za konkurentni svet

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Tijana Jevtić, Jelena Mrdak, David Dimić, Zorana Gajić
tijanatijanajevtic@gmail.com, mrdakj@gmail.com,
daviddimic@hotmail.com, zokaaa_gajich@bk.ru

6. april 2019.

Sažetak

Nakon čitanja rada, čitalac će imati globalnu sliku o jeziku i detaljniji pogled na neke važne koncepte, kao i uvid u korišćenu literaturu koju može konsultovati radi daljeg informisanja o temi.

Sadržaj

1 Uvod

U ovom radu je prikazan programski jezik Erlang iz različitih uglova. Kroz niz poglavlja i primera, ispričana je njegova istorija - kad, kako, gde i zašto je nastao, po čemu je karakterističan, šta ga to izdvaja od drugih programskih jezika, koji su to koncepti koji su svojevrsni Erlangu.

Akcenat će biti na funkcionalnoj paradigmi i uobičajenim mehanizmima koji funkcionalni jezici implementiraju, kao što su: sakupljači otpadaka, poklapanje obrazaca, načini izračunavanja izraza itd.

Kroz niz poglavlja i primera, ispričana je njegova istorija - kad, kako, gde i zašto je nastao 2, po čemu je karakterističan i koja mu je osnovna namena 3 4. Na samom početku rada u poglavlju 2 ćemo se upoznati sa istorijskim razvojem jezika. Nešto više reči o nameni, mogućnostima i osnovnim osobinama Erlang-a će biti u poglavljima 3 4. U narednom poglavlju 5 posvetićemo posebnu pažnju konkuretnosti. Onda ćemo videti uticaj Erlang-a u veb programiranju, upoznaćemo se sa veb okruženjima ChicagoBoss, Nitrogen i Zotonic u poglavlju 6. Sekcija 7 je posvećena instalaciji i pokretanju Erlang-a, nakon čega sledi sekcija 8 sa primerima kodova.

2 Nastanak i istorijski razvoj

1981. godine je oformljena nova laboratorija, Erikson CSLab (eng. *The Ericsson CSLab*) u okviru firme Erikson sa ciljem da predlaže i stvara nove arhitekture, koncepte i strukture za buduće softverske sisteme [6]. Eksperimentisanje sa dodavanjem konkurentnih procesa u programski jezik Prolog je bio jedan od projekata Erikson CSLab-a i predstavlja začetak novog programskog jezika. Taj programski jezik je 1987. godine nazvan Erlang [1]. Sve do 1990., Erlang se mogao posmatrati kao dijalekt Prologa. Od tada, Erlang ima svoju sintaksu i postoji kao potpuno samostalan programski jezik. Godine rada su rezultirale u sve bržim, boljim i stabilnijim verzijama jezika, kao i u nastanku standardne biblioteke OTP (eng. *The Open Telecom Platform*) [6]. Od decembra 1998. godine, Erlang i OTP su postali deo slobodnog softvera (eng. *open source software*) i mogu se slobodno preuzeti sa Erlangovog zvaničnog sajta [9]. Danas, veliki broj kompanija koristi Erlang u razvoju svojih softverskih rešenja. Neke od njih su: Erikson, Motorola, Votsap (eng. *Whatsapp*), Jahu (eng. *Yahoo!*), Fejsbuk (eng. *Facebook*) [9].

2.1 Uticaji

Erlang je funkcionalan, deklarativan i konkurentan programski jezik. Na njega, kao na funkcionalan jezik, uticao je Lisp funkcionalnom paradigmom koju je prvi predstavio. Na planu konkurentnosti Erlang je svojevrstan primer (detaljnije u poglavlju 4).

Na početku, Erlang je stvaran kao neki dodatak na Prolog, vremenom prerastao u dijalekt Prologa, a kasnije je zbog svoje kompleksnosti i sveobuhvatnosti evoluirao u potpuno novi programski jezik. Stoga je uticaj Prologa na Erlang bio neminovan. Sintaksa Erlanga u velikoj meri

[1] Erlang je jedinica saobraćaja u oblasti telekomunikacija i predstavlja kontinuirano korišćenje jednog kanala (npr. ako jedna osoba obavi jedan poziv telefonom u trajanju od sat vremena, tada se kaže da sistem ima 1 Erlang saobraćaja na tom kanalu).

podseća na Prologovu (npr. promenljive moraju počinjati velikim slovom, svaka funkcionalna celina se završava tačkom), oba jezika u velikoj meri koriste poklapanje obrazaca (eng. *pattern matching*).

Sa druge strane, Erlang je uticao na nastanak programskog jezika Eliksir (eng. *Elixir*). Eliksir, uz izmenjenu Erlangovu sintaksu, dopunjenu Erlangovu standardnu biblioteku, uživa široku popularnost (lista kompanija koje ga svakodnevno koriste).

3 Osnovna namena, svrha i mogućnosti

Sa početkom od 1981. godine, jedan od zadataka Eriksonove laboratorije za računarstvo je bio pronalaženje načina za bolje programiranje aplikacija za telekomunikacije [6]. Takve aplikacije su ogromni programi i od velike važnosti je da rade sve vreme (koliko je to moguće). Naravno, poznato je da će tolika količina koda zasigurno imati greške, ali u toj vrsti industrije, greške mogu biti fatalne. Na primer, šta se dešava ako je došlo do kvara na nekoj telefonskoj liniji, a telefon nam je hitno potreban (recimo, neko ima srčani udar). Jednostavno nije moguće zaustaviti takvu aplikaciju, popraviti je i nanovo pustiti u rad. Kako se izboriti sa greškama u softverskim sistemima kada su one neminovne je osnovna motivacija za razvoj Erlanga [6].

Tako, jedna od njegovih namena jeste pisanje što sigurnijih programa koje je moguće popraviti bez potrebe za isključivanjem čitavog sistema [5]. Vrlo brzi konkurentni i distribuirani programi su još jedna od Erlangovih specijalnosti. Poseban koncept konkurentnosti koji je implementiran u Erlangu (više u poglavlju 5), kao i funkcionalna paradigma omogućavaju lako skaliranje programa i pravljenje velikih konkurentnih i distribuiranih sistema. Velika zajednica koja se godinama razvijala je doprinela stvaranju velikog broja biblioteka i okruženja za Erlang, te proširila njegov inicijalni skup mogućnosti i namena [5].

4 Osnovne osobine

Sve paradigme koje podržava Erlang su tu da bi se dobio jednostavan, kvalitetan i siguran konkurentan jezik. Kao svaki funkcionalni jezik poseduje sakupljač otpadaka [8] u realnom vremenu, tako da se ne mogu pojaviti greške programera pri rukovanju memorijom. Sa konkurentne strane, sistem ima ugrađenu kontrolu vremena, u smislu da se može odrediti koliko će neki proces čekati na poruku pre nego što se aktivira, pa omogućava pisanje aplikacija koje rade u mekom realnom vremenu (eng. *soft real-time systems*) [8] sa odzivom od nekoliko milisekudni. U ovom poglavlju videćemo koji su tipovi podžani u Erlangu da bi se njegove osibine i namene opisane u poglavlju 3 ostvarile, kao i neka osnovna svojstva i koncepte.

4.1 Tipovi i promenjive

Erlang je dinamički i jako tipiziran jezik. Na raspolaganju nam je 8 primitivnih tipova [6]. Osim uobičajnih celobrojnih, realnih vrednosti i referenci, Erlang uvodi i neke specifične tipove:

- Atomi koji se pišu malim slovima i predstavljaju konstante i enumerisane tipove. Samo ime je njihova vrednost. Pandan su makroima i enumerisanim tipovima u C-u.
- Binarne vrednosti omogućavaju lako i čitljivo prelamanje broja na segmente u binarni zapis na zadatoj širini. U oznaci «vrednost:širina»
- Identifikatori procesa predstavljaju reference na procese. Kreiraju se funkcijom spawn
- Portovi služe za komunikaciju sa spoljašnjim svetom. Ako su u skladu sa protokolom portova preko njih se mogu slati i primati poruke

Tu su i dve osnovne strukture koje mogu da sadrže bilo koje tipove: torke $\{elem_1, eleme_2, ...elemt_n\}$ za fiksirani broj elemenata u njima, i liste $[elem_1, elem_2, ...]$ za čuvanje promenjivog broja elemenata. Osnovni operator konstrukcije liste je $[Glava|Rep]$. U okviru listi se prikazuju i niske, za koje ne postoji ugrađeni poseban tip, već su one liste vrednost koje odgovaraju vrednostima karaktera. Ako svi elementi liste mogu da se prikažu kao karakteri onda će lista biti ispisana kao niska, što ilustruje naredni primer.

```
1> [16#5A, 97+3, 2*50+14, 97, 8#166, 2#1101111].
"Zdravo"
2> [65,97,2].
[65,97,2]
```

U drugom primeru 2 se ne može prikazati kao karakter pa lista nije prikazana kao niska. Ovde vidimo i neka elementarna izračunavanja i kako sa # možemo elegantno koristiti bilo koju brojevnu osnovu. Da bismo sačuvali izračunavanja potrebne su nam promenjive.

Promenjive mogu biti vezane (eng. *bound*), one kojima je "dodeljena" neka vrednost, i slobodne. Vezivanje se vrši najviše jednom i vrednost vezanih promenjihiv više se ne može menjati (eng. *single assignment variables*) osim ako se u interpreteru ne pozove funkicija f() koja sve promenjive načini slobodnim [7]. Ovo je u skladu sa idejom funkcionalnih jezika da nema sporednih efekata što za posledicu ima jednostavno izvođenje konkurentnosti, iako Erlang nije čisto funkcionalan jezik. Zapravo, operator = ne predstavlja nikakvu dodelu već poklapanje obrazaca.

4.2 Poklapanje obrazaca

Većinu funkcija u Erlangu, kao i svako vezivanje promenjivih pišemo putem poklapanja obrazaca (eng. *pattern matching*). Neformalno[2], to je postupak poređenja terma sa obrascem. Ako obrazac i term imaju isti oblik, poklapanje uspeva, pri čemu će svaka promenjiva biti vezana sa podatkom na njemu odgovarajućoj poziciji. Ovaj proces poznat je kao *unifikacija*. Pri unifikaciji na raspolaganju je i posebna anonimna promenjiva _ , a koju koristimo kada nas neka vrednost ne zanima i ne želimo ni jednu promenjivu da vežemo za tu vrednost. U sledećem primeru prve tri linije pokazuju uspešnu unifikaciju, dok je u poslednjoj pokušana unifikacija X sa 51, što nije uspelo kako je X već vezano za {137, 42}.

```
1> Z = 2.
2> {X, macka} = {{137, 42}, macka}.
```

[2] Formalna definicija u dodatku A.1

```
3> [Glava|_] = [1,2,3,4,5,6].
4> {X, Y} = {51, kuce}.
```

Jedno proširenje mogućnosti poklapanja obrazaca, korisno za izvođenje jednostavnih testova i poređenja u šablonu daju nam čuvari (eng. *guards*) sa ključnom rečju *when*. Čuvari su izrazi odvojeni sa ',' koji sadrže samo predikate poređenja[3] kao u sledećem primeru.

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

Funkcija *max* je definisana preko pokpalanja obrazaca uz korišćenjem čuvara jednog uslova koji određuje povratnu vrednost. U narednom delu formalizovaćemo šta su funkcije u Erlangu.

4.3 Funkcije

Svaka funkcija može imati više slučajeva odvojenih sa ';' do kojih će doći poklapanjem obrazaca ili preko argumenata ili preko čuvara u *when* delu koji je opcioni. Telo funkcije od niza izraza razdvojenih ','. Tačkom se završava definicija i odvaja od ostalih funkcija. Njihovi primeri i korišćenja biće detaljnije opisani u delu 8.

```
ime_funckije(a11, a12, ... a1N) [when g11, g12, ... g1N] ->
        telo1;
...
ime_funckije(aM1, aM2, ... aMN) [when gM1, gM2, ... gMN] ->
        teloM.
```

Kao u svakom funkcionalnom jeziku funkcije su građni prvog reda, tako da ih možemo prosleđivati kao argumente, vraćati kao povratnu vrednost itd. Na raspolaganju su nam i anonimne lamda funkcije koje imaju sledeći oblik:

```
fun(a1, a2, ... aN) -> telo end.
```

Česte su rekurzivne definicije funkcija [5], ali moramo imati na umu da su najefikasnije repne rekurzivne za koje nije potreban stek. Mnoge funkcije u Erlangu dizajnirane su da se vrte u beskonačnim petljama, posebno u klijent-server modelu u ulozi servera opisanog u narednom poglavlju 5.

5 Konkurentnost kao glavna specifičnost

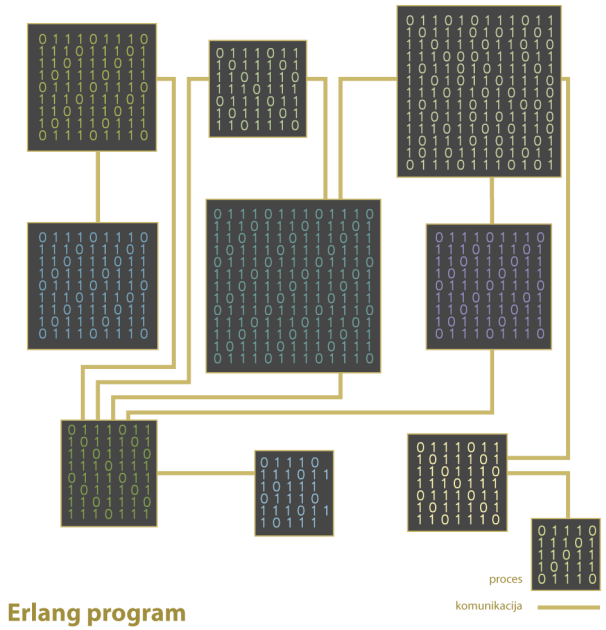
Jedna od osnovnih osobina Erlanga i specifičnost po kojom se izdvaja od drugih jezika je konkurentnost i koncept na kom je zasnovana. Posmatrajući svet oko sebe, uviđamo da je on u suštini konkurentan - u istom trenutku se dešava veliki broj procesa [6]. U tom istom trenutku, mi smo sposobni da takav svet pojmimo i odreagujemo na sve što se u njemu dešava. Dakle, mi prirodno razumemo konkurentnost. Tako se i prirodno nameće potreba za programskim jezikom koji bi omogućavao jednostavno modelovanje sveta kakav on stvarno jeste.

5.1 Koncept

Koncept konkurentnosti implementiran u Erlangu se zove konkurentnost slanjem poruka (eng. *message passing concurrency*), šematski prikazan na slici 1. Ovo podrazumeva postojanje velikog broja procesa koji

[3] Operatori poređenja su: $<$, $=<$, $>$, $=>$, $==$, $/=$, $=:=$, $=/=$

nikad ne dele memoriju, već komuniciraju isključivo asinhronim slanjem poruka [5]. Sva izračunavanja se obavljaju u okviru procesa i trebalo bi da sistem bude dizajniran tako da jedan proces radi jedan mali posao. Važno je napomenuti da procesi u Erlangu nisu procesi operativnog sistema, već Erlanga. To je moguće zbog toga što se njegovi programi izvršavaju na BEAM virtuelnoj mašini. Procesi se prave i uništavaju jako brzo, zauzimaju samo onoliko memorije koliko je neophodno (u većini slučajeva vrlo malo) i ponašaju se isto na svim operativnim sistemima [5].



Slika 1: Konkurentnost u Erlangu

5.2 Slanje i primanje poruka

Erlang omogućava jednostavano kreiranje novog procesa pozivom funkcije *spawn* koja vraća pid (eng. *process identifier*) na osnovu kojeg svaki proces može da razlikuje ostale procese.

```
Pid = spawn([Module], FunctionName, [ArgumentList])
```

Jedini način za ostvarivanje komunikacije između dva procesa je putem slanja poruka korišćenjem operatora '!'. U `Pid ! Message` procesu sa identifikatorom *Pid* šalje se poruka sadržana u promenljivoj *Message*, pri čemu poruka može biti bilo koji validni term. Operator slanja evoluira svoje argumente – prvo levi argument da bi se dobio pid procesa koji prima poruku, a potom desni da bi se dobila poruka koja se šalje. Povratna vrednost je baš ta poruka. Slanje poruka je asinhrono, pošiljalac neće čekati da poslata poruka stigne na odredište niti da bude primljena [5, 8].

Za primanje poruka koristi se operator *receive*. Svaki proces ima svoje sanduče gde se nalaze poruke u redosledu pristizanja. Poruke se porede

poklapanjem obrazaca. Kada poklapanje uspe poruka se vadi iz sandučeta i izvršava se navedena akcija. *Receive* vraća vrednost poslednjg izraza iz izvršene akcije. Ako poslata poruka nikad ne stigne na odredište, što se može desiti npr. kao posledica pada procesa koji je poslao poruku, izbegavamo beskonačno čekanje tako što se uvodi maksimalno vreme čekanja (eng. *timeout*).

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
        ...
        [after Time ->
                Expressions]
end
```

Ukoliko želimo da primimo bilo koju poruku to može da se uradi korišćenjem *AnyMessage*. Ali, češće želimo da primamo samo one poruke koje su namenjene nama. Da bismo to ostvarili moramo poslati svoj pid (ako šaljemo pismo moramo napisati svoju adresu da bismo dobili odgovor) što se postiže sa funkcijom self().

```
Pid ! {self(), Message}
```

5.3 Klijent-server model

Kada šaljemo poruku procesu moramo znati njegov pid. Ovo često nije praktično zbog mogućeg velikog broja procesa, niti poželjno iz bezbedonosnih razloga (neki procesi bi trebalo da sakriju svoj identitet). Da bi bilo omogućeno slanje poruka bez poznavanja identifikatora uvodi se pojam *registrovanja procesa*, tj. davanje imena koje mora biti atom [5, 6]. Funkcijom `register(name, Pid)` atom *name* se vezuje za *Pid* i na dalje ga možemo indentifikovati preko dodeljenog atoma.

Glavni razlog registrovanja procesa je da se omogući *klijent-server model* koji je ključni za komunikaciju između procesa u Erlangu [8, 5, 6]. U ovom modelu obe strane mogu biti procesi na istoj ili različitim mašinama. Klijent uvek započinje neko izračunavanje obraćajući se serveru, koji obrađuje zahtev i vraća odgovor klijentu. Jedan jednostavan primer ovog modela biće objašnjen u delu 8.

5.4 Greške u konkurentnim programima

Za pravljenje sistema otpornog na greške neophodna su nam dva računara, jedan *radnik* koji će izvršavati posao i drugi *supervizor* koji će posmatrati i biti spreman da preduzme posao u trenutku kada se dogodi pad prvog (eng. *worker-supervisor relationship*) [5]. U Erlangu je ovo omogućeno korišćenjem povezivanja procesa funkcijom *link*. Pretpostavićemo da su neki procesi A i B linkovani. Ako proces A iznenada prestane sa radom onda se proces B obaveštava takozvanim *završnim signalom*. Ukoliko je obrnuto, obaveštava se proces A. Šta se dešava kada proces dobije završni signal? Ako proces nije bio aktivan u tom trenutku onda se on uništava, a inače proces može da traži da se uhvate signali. Proces u ovakvom stanju se naziva *sistemskim procesom* [5].

Sledeći primer prikazuje kako uhvatiti završni signal. Funkcija *on_ exit(Pid, Fun)* pravi link ka procesu *Pid*. Pozivom funkcije *process_ flag* kreirani novi proces se transformiše u sistemski proces i onda se povezuje se sa procesom *Pid* uz pomoć funkcije *link(Pid)*. Kada proces umre onda je primljen i obrađen blokom *receive*.

```
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  receive
                      {'EXIT', Pid, Why} ->
                             Fun(Why)
                  end
          end).
```

Testiranje se vrši kreiranjem funkcije *F* koja čeka na poruku *X* od koje onda kreira atom i nekog procesa *Pid*. Onda se za nadgledanje postavlja funkcija *on_ exit*.

```
1> F = fun() ->
     receive
         X -> list_to_atom(X)
     end
   end.

2> Pid = spawn(F).

3> on_exit(Pid,
           fun(Why) ->
               io:format(" ~p died with:~p~n",[Pid, Why])
           end).
```

Ako se procesu *Pid* pošalje atom, on će umreti jer će pokušati da izračuna funkciju *list_ to_ atom* sa argumentom koji nije lista i onda će funkcija *on_ exit* prijaviti grešku.

```
4> Pid ! Zdravo.
Zdravo
<0.61.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}
```

Još jedan način za razrešavanje greški je takozvanim živo-održivim procesima (eng. *a keep-alive process*). Ideja je kreirati registrovane procese, koje smo spomenuli u 5.3, koji će uvek biti živi i ako prestanu sa radom iz bilo kog razloga, odmah će se restartovati. U narednom primeru se kreira registrovani proces pod nazivom *Name* koji izvršava *spawn(Fun)*.

```
keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
```

6 Okruženja i njihove karakteristike

Erlang je poznat za podržavanje skalabilnih sistema otpornih na greške (eng. *scalable fault-tolerant systems*), ali takođe nudi mnoštvo mogućnosti koje ga čine dobrim jezikom za veb programiranje. Na primer, mogućnost reagovanja na više korisničkih zahteva istovremeno, ne razmišljajući o problemima konkurentnosti. U tabeli 1 je prikazano poređenje 3 glavna

veb okruženja: *ChicagoBoss*, *Nitrogen* i *Zotonic* po nekim interesantnim osobinama.

Tabela 1: Poređenje Erlang veb okruženja

	ChicagoBoss	Nitrogen	Zotonic
Razvoj zasnovan na događajima	✓	✓	✓
Okruženje za testove	✓	✓	✓
Generisanje koda	✓	-	-
Django šabloni	✓	-	✓
Integrisani mejl server	✓	-	✓
UTF-8 u Erlang kodu	✓	-	✓
Višejezični podaci	-	-	✓
Generisanje *JavaScript* koda	-	✓	✓
Generisanje *JSON* formata	✓	✓	✓
Integrisani WebSocket	✓	✓	✓

Okruženje *ChicagoBoss* sadrži sloj apstrakcije baze podataka (eng. *database abstraction layer*) pod nazivom *BossDB* [1] koji je zaslužan za postavljanje upita nad bazom podataka i njeno ažuriranje. Podržani su *MySQL, Mnesia, Tokyo Tyrant i PostgreSQL*. Za razliku od *ChicagoBoss-a*, *Nitrogen* okruženje ne podržava model podataka uopšte, dok *Zotonic* [4] podržava isključivo *PostgreSQL*.

Takođe, interesantno je primetiti da neka okruženja imaju integrisani mejl server koji nudi funkcije za primanje i slanje e-pošte i ostale mogućnosti, čime olakšava rad korisnicima. Na primer, slanje e-pošte u okruženju *ChicahoBoss* izgleda ovako:

```
boss_mail:send(FromAddress, ToAddress, Subject, Body)
```

U tabeli 1 vidimo da sva tri okruženja podržavaju i okruženje za testove, gde su testovi struktuirani kao stabla nadovezivanja (eng. *trees of continuations*) [3]. Postoje gotove funkcije koje olakšavaju testiranje nekih opšte poznatih akcija kao što je provera da li je e-pošta ispravno primljena/poslata, da li je stranica na vebu modifikovana itd.

Django šabloni (eng. *Django templates*) [2] služe za jednostavnije i brže generisanje dinamičkih HTML stranica pomoću gotovih šablona. *Nitrogen* ima svoje *Nitrogen HTML* šablone ali je u procesu prelazak na *Django* šablone.

Svaki od opisanih okruženja ima svoje prednosti i mane, te zato nije jednostavno presuditi koji od ovih okruženja treba koristiti zasigurno, a koji ne treba. U zavisnosti od onoga šta je prioritet bira se odgovarajuće okruženje. U dodatku A.2 možete pogledati primere u nekim od spomenutih okruženja.

7 Instalacija i pokretanje

Postoji više načina da se instalira Erlang sa neophodnim paketima. Ovde će biti predstavljena instalacija korišćenjem prekompajliranih binar-

nih fajlova za neke operativne sisteme zasnovane na Linuksovom kernelu i pokretanje na jednom od njih, kao i instalacija za Windows.

7.1 Linux

Na operativnim sistemima zasnovanim na *Ubuntu*, Erlang se može instalirati sa: *sudo apt-get install erlang*. Nakon uspešne instalacije, Erlang kod je moguće kompajlovati ili interpretirati i pokretati u interpretatoru. Interpretator se pokreće kucanjem komande *erl* u terminalu, a iz istog se izlazi sa *Ctrl+G* iza kog sledi *q* [5]. Erlang interpretator ima u sebi ugrađen editor teksta koji je baziran na *emacs-u* [7]. Kôd iz datoteke se kompajluje komandom *erlc* i navođenjem imena fajla sa ekstenzijom *erl*. Nakon toga se dobija izvršna datoteka sa ekstenzijom *beam* koja se može pokrenuti uz navođenje adekvatnih flegova.

7.2 Windows

Na operativnom sistemu *Windows*, Erlang se može instalirati preuzimanjem binarne datoteke sa oficijalnog sajta [9] programskog jezika. Posle duplog klika na *.exe* fajl samo je potrebno ispratiti uputstva. Pokretanje interpretatora se vrši na isti način kao i na Linuks sistemima.

8 Primeri kodova sa objašnjenjima

Počećemo od jednostavnog primera "Hello World" i videti osnovnu sintaksu jezika. Jednolinijski komentari počinju znakom %. Prvo navodimo naziv našeg modula u kome se nalaze funkcije koje pišemo, a da bi one mogle da se koriste izvan modula potrebno je da ih navedemo u export naredbi. /0 označava da funkcija *start* prima 0 argumenata. Da bismo željeni tekst prikazali u konzoli, koristimo **io** modul koji sadrži potrebne IO funkcije u Erlangu.

```
% hello world program
-module(helloworld).
-export([start/0]).

start() ->
   io:fwrite("Hello, world!\n").
```

```
> Hello, world!
```

Kako su rekurzija i liste jedna od prvih asocijacija na funkcionalni jezik, počećemo primere od faktorijela, pa ćemo nastaviti sa listama.

U narednom primeru je dat modul koji definiše rekurzivnu funkciju za računanje faktorijela i pritom ilustruje poklapanje šablona i čuvare (engl. guards).

```
-module(math).
-export([factorial/1]).

factorial(0) ->
  1;
factorial(X) when X > 0 ->
  X * factorial(X-1).
```

Liste predstavljaju osnovu svih funkcionalnih jezika, pa i Erlang-a. Liste mogu sadržati brojeve, torke, druge liste itd. Stringovi su takođe liste, što možemo videti u primeru.

```
> [97, 98, 99].
"abc"
```

Posmatrajmo sledeći primer:

```
> [97,98,99,4,5,6].
[97,98,99,4,5,6]
> [233].
"é"
```

Dakle, Erlang će listu brojeva posmatrati kao listu brojeva ako ne postoji broj u listi koji je ujedno i karakter.

Za konkatenaciju dve liste koristi se operator ++, dok je njemu suprotan operator −−.

```
> [1,2,3] ++ [4,5].
[1,2,3,4,5]
> [1,2,3,4,5] -- [1,2,3].
[4,5]
> [2,4,2] -- [2,4].
[2]
> [2,4,2] -- [2,4,2].
[]
```

Oba ova operatora su desno asocijativna.

```
> [1,2,3] -- [1,2] -- [3].
[3]
> [1,2,3] -- [1,2] -- [2].
[2,3]
```

Glavu liste dobijamo pomoću funkcije **hd**, dok za rep koristimo funkciju **tl**.

```
> hd([1,2,3,4]).
1
> tl([1,2,3,4]).
[2,3,4]
```

Kao i većina funkcionalnih jezika, i Erlang podržava shvatanje listi (eng. list comprehensions), što ilustrujemo narednim primerima.

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].
[a,4,b,5,6]
```

Notacija `X <- [1, 2, a, ...]` je generator, dok je izraz `X>3` filter.

Možemo primeniti više filtera.

```
> [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
[4,5,6]
```

Takođe, moguće je kombinovati i generatore. Na primer, Dekartov proizvod dve liste možemo napisati kao

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

Algoritam QuickSort u Erlangu se može implementirati na sledeći način:

```
sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

Izraz `[X || X <- T, X < Pivot]` e lista svih elemenata iz T koji su manji od pivota. Slično, `[X || X <- T, X >= Pivot]` je lista svih elemenata iz T koji su veći ili jednaki od pivota.

Neizostavna funkcija svih funkcionalnih programskih jezika jeste map. map(F, List) je funkcija koja prima funkciju F i listu L i vraća novu listu dobijenu primernom funkcije F na svaki element liste L.

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].

double(L)  -> map(fun(X) -> 2*X end, L).

> double([1,2,3,4]).
[2,4,6,8]
```

Naredni primer pokazuje kako komuniciraju procesi u klijent-server modelu opisanog u delu 5. Klijent šalje zahtev serveru za računanje hipotenuze ili površine pravouglog trougla, a server vrši izračunavanje u funkciji *loop* i vraća odgovoru klijentu. Kada modul *trougao* bude implementiran, izvršavanje će izgledati ovako: U liniji 1 kreiramo novi serverski proces i dobijemo njegov pid, potom ga registrujemo i pozovemo funkciju *klijent* koja enkapsulira slanje zahteva i primanje odgovora.

```
1> Pid = spawn(fun trougao:loop/0).
2> register(server, Pid).
3> trougao:klijent(server, {hipotenuza,3,4}).
5.0
```

U klijentskoj funkciji pošaljilac mora da uključi svoju adresu sa *self()* i pošalje zahtev na serverski proces. Potom čeka odgovor koji je namenjen njemu, odnosno prihvata odgovor kada se poklopi obrazac torke {Pid, Response}. Tačnije, *Pid* je vezana, a *Response* slobodan promenjiva koja će biti vezana kada stigne odgovor.

```
klijent(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

Sa klijetske strane takođe se vrši poklapanje obrazaca sa atomom koji je poslat u zahtevu (šta klijent želi da računa), izračunavanje i slanje odgovora na adresu klijenta. Kompletan kod dostupan je u datoteci trougao.erl

```
loop() ->
    receive
        {From, {hipotenuza, A, B}} ->
            From ! {self(), sqrt(A*A + B*B)}, loop();
        {From, {povrsina, A, B}} ->
            From !  {self(), A * B / 2}, loop();
        {From, Other} ->
```

```
            From ! {self(), {error, Other}}, loop()
    end.
```

9 Zaključak

U ovom radu su ukratko, ali jezgrovito objašnjene najvažnije osobine i paradigme programskog jezika Erlang: od upoznavanja sa sintaksom, preko elementarnih primera, do njegovog specifičnog koncepta konkurentnosti. Elegantan i jednostavan rad sa procesima u potpunosti je opravdao ideju i namenu za koju je jezik nastao. Ipak, nijedna tehnologija, niti programski jezik nisu univerzalno rešenje, pa tako nije ni Erlang. Njegova ekspertiza su sigurni konkurentni i distribuirani sistemi, aplikacije za telekomunikaciju, Internet serveri, dok nije najbolji izbor za obradu slika, signala i velike količine podataka [9]. Kroz izložene koncepte ovaj rad pruža odličan uvod za upoznavanje sa Erlangom i podstrek čitaocu na dalje i dublje istraživanje koje bi se moglo ticati distribuiranosti, radu sa portovima, obradom grešaka i slično.

Literatura

[1] ChicagoBoss framework documentation. on-line at: http://chicagoboss.org/doc/api-db.html.

[2] Django Templates Documentation. on-line at: https://docs.djangoproject.com/fr/2.1/topics/templates/.

[3] Functional Tests As A Tree Of Continuations. on-line at: https://www.evanmiller.org/functional-tests-as-a-tree-of-continuations.html.

[4] Zotonic framework documentation. on-line at: http://docs.zotonic.com/en/latest/index.html.

[5] J. Armstrong. *Programming Erlang (2nd edition)*. Pragmatic Bookshelf, 2013.

[6] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[7] F. Hebert. *Learn You Some Erlang for Great Good!* No Starch Press, 2013.

[8] C. Wikström M. Williams J. Armstrong, R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[9] OTP team. Erlang. on-line at: http://www.erlang.org/.

A Dodatak

A.1 Pokpalanje obrazaca

Da bismo objasnili ovaj ključni koncept potrebno je prvo da definišemo pojmove terma i obrasca [6].

Definicija 1. *Osnonvni term (eng.* ground term*) se definiše kao primitivni tip, uređeni par ili lista osnovnog terma.*

Definicija 2. *Obrazac ili šablon (eng.* pattern*) može biti primitivni tip, promenjiva, uređeni par ili lista šablona. Ako su u obrascu sve promenjive različite onda se on naziva primitivnim.*

Definicija 3. *Ako je P primitivni obrazac i T term, onda kažemo da se obrazac P poklapa sa T ako i samo ako:*

- *Ako je P lista sa glavom Pg i repom Pr, a T lista sa glavom Tg i repom Tr, onda Pg mora da se unifikuje sa Tg i Pr sa Tr*
- *Ako je P torka P1,P2,...Pn i T torka T1,T2,...Tn, onda svi elementi redom mora da se unifikuju – P1 sa T1, P2 sa T2, ... Pn sa Tn.*
- *Ako je P konstanta, onda T mora da bude ista konstanta*
- *Ako je P slobodna promenjiva V onda će V biti vezana za T*

A.2 Primeri koda u veb okruženjima

U poglavlju 6 smo se upoznali sa glavnim predstavnicima Erlang veb okruženja i videli njihove osnovne karakteristike. Prikazaćemo nekoliko primera u veb okruženjima.

Prvo ćemo pokazati jedan napredniji primer programa "Hello World"u *ChicagoBoss* okruženju. Erlang fajl bi izgledao ovako:

```
-module(appname_my_controller, [Req]).
-compile(export_all).
hello('GET', [Name]) ->
        {ok, [{name, Name}]}.
```

Dok je u HTML fajlu neophodno napisati:

```
Hello, {{ name }}.
```

Primer u okruženju *Nitrogen* sa ispisom poruke kao reagovanje na klik. Prvo se učitava gotov šablon HTML strane i postavlja se naslov strane. Zatim se dodaje tekst ispod kojeg se kreira dugme sa njegovim identifikatorom i obezbeđuje se reagovanje na klik. Postoji hvatač događaja *event(click)* koji će ispisati poruku kada se klikne na dugme i zameniti sve ono što se nalazi u telu (eng. *html body*) sa novom porukom.

```
-module(module_name).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").

main() -> #template{file=template_name}.
title() -> "My Hello World".

body() ->
    #panel{body=[
        "Click on button below!!"
```

```
            #br{},
            #button{
                id=button_id_name,
                postback=click,
                text="Click Me"
            }
        ]}.

    event(click) ->
        NewBody = #panel{
            id=id_name_of_the_replacement,
            text="You clicked!!"
        },
        wf:replace(button_id_name, NewBody).
```