

# 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

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. 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

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Nastanak i istorijski razvoj</b>	<b>2</b>
<b>3</b>	<b>Osnovna namena, svrha i mogućnosti</b>	<b>3</b>
<b>4</b>	<b>Osnovne osobine</b>	<b>3</b>
<b>5</b>	<b>Konkurentnost kao glavna specifičnost</b>	<b>6</b>
<b>6</b>	<b>Okruženja i njihove karakteristike</b>	<b>8</b>
<b>7</b>	<b>Instalacija i pokretanje</b>	<b>9</b>
<b>8</b>	<b>Primeri kodova sa objašnjenjima</b>	<b>9</b>
<b>9</b>	<b>Zaključak</b>	<b>11</b>
	<b>Literatura</b>	<b>11</b>
<b>A</b>	<b>Dodatak</b>	<b>12</b>

# 1 Uvod

Ovaj rad ćemo posvetiti programskom jeziku Erlang. Najpre ćemo se osvrtnuti na istoriju, preciznije na nastanak i razvoj jezika. Videćemo po čemu se Erlang izdvaja u odnosu na druge programske jezike, kao i koje prednosti njegovo korišćenje donosi.

Nešto više reči će biti o nameni, mogućnostima i osnovnim osobinama Erlang-a. Akcenat će biti na funkcionalnoj paradigmi i uobičajenim mehanizmima koji funkcionalni jezici implementiraju, kao što su: sakupljači otpadaka, poklapanje obrazaca, načini izračunavanja izraza itd.

Konkurentnost predstavlja jednu od važnih osobina jezika i njoj ćemo posvetiti posebnu pažnju u radu.

U sekciji 6 ćemo videti uticaj Erlang-a u veb programiranju, gde njegova konkurentnost dolazi do izražaja. Takođe, upoznaćemo se sa veb okruženjima ChicagoBoss, Nitrogen i Zotonic.

Sekcija 7 je posvećena instalaciji i pokretanju Erlang-a, nakon čega sledi sekcija sa primerima kodova.

Na kraju ćemo izdvojiti specifičnosti jezika.

## 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. 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<sup>1</sup>. 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*) [7]. 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!*), Amazon, Fejsbuk (eng. *Facebook*).

### 2.1 Uticaji

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

---

<sup>1</sup>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).

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 podseća na Prologovu (npr. promenljive moraju počinjati velikim slovom u oba jezika, 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 Elik-sir (eng. *Elixir*). Elik-sir, uz izmenjenu Erlangovu sintaksu, dopunjenu Erlangovu standardnu biblioteku, uživa široku popularnost.

### 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 [7]. 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 [7].

Tako, jedna od njegovih namena jeste pisanje što sigurnijih programa koje je moguće popraviti bez potrebe za isključivanjem čitavog sistema [6]. Vrlo brzi konkurentni i distribuirani programi su još jedna od Erlangovih specijalnosti. Poseban koncept konkurentnosti koji je implementiran u Erlangu (više u poglavlju 4), 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 [6].

### 4 Osnovne osobine

Erlang je zasnovan na deklarativnoj i funkcionalnoj paradigmi sa akcentom na konkurentnosti. Kao pripadnik funkcionalne paradigme poseduje sakupljač otpadaka koji upravlja memorijom u realnom vremenu tako da se ne mogu pojaviti greške programera pri rukovanju memorijom. Takođe, 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*) sa odzivom od nekoliko milisekundi. U ovom poglavlju videćemo koji su tipovi podžani u Erlangu da bi se njegove osobine i namene opisane u poglavlju 3 ostvarile, kao i neka osnovna svojstva i koncepte.

## 4.1 Tipovi i promenjive

Na raspolaganju nam je 8 primitivnih tipova. Osim uobičajnih celobrojnih, realnih vrednosti i referenci, Erlang uvodi i neke specifične tipove kao što su:

- Atomi koji se pišu malim slovima i predstavljaju konstante i enumerisane tipove. Samo ime je njihova vrednost
- 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 primiti poruke

Tu su i dve osnovne strukture koje mogu da sadrže bilo koje tipove: torke  $\{elem_1, elem_2, \dots, elem_n\}$  za fiksirani broj elemenata u njima, i liste  $[elem_1, elem_2, \dots]$  za čuvanje promenljivog 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 vrednosti 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 brojevnju 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 promenljivih više se ne može menjati (eng. *single assignment variables*) osim ako se u interpreteru ne pozove funkcija `f()` koja sve promenjive načini slobodnim. 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 promenljivih pišemo putem poklapanja obrazaca. Da bismo objasnili ovaj ključni koncept potrebno je prvo da definišemo pojmove terma, obrasca i čuvara.

**Definicija 1.** *Osnovni 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.*

Poklapanje obrazaca (eng. *pattern matching*) je postupak poređenja terma sa obrascem. Neformalno<sup>2</sup>, 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 koja se označava sa `_`, 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 uspeo kako je X već vezano za {137, 42}.

```
1> Z = 2.
2> {X, macka} = {{137, 42}, macka}.
3> [Glava|_] = [1,2,3,4,5,6].
4> {X, Y} = {51, kuce}.
```

**Definicija 3.** Čuvari (eng. guards) su izrazi koji sadrže samo predikate oblika *A op B* odvojeni zarezom pri čemu su *op* validni binarni operatori poređenja<sup>3</sup>. Izraz sa čuvarima može da se evaluiira samo u *true* ili *false*.

Čuvari sa ključnom rečju *when* predstavljaju dodatno proširenje mogućnosti poklapanja obrazaca, za izvođenje jednostavnih testova i poređenja u šablonu, kao u primeru funkcije za maksimum dve vrednosti:

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

Sama funkcija je definisana preko poklapanja obrazaca sa korišćenjem čuvara. U sledećem delu formalizovaćemo šta je funkcija u Erlangu i kako se definiše.

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

Kao u svakom funkcionalnom jeziku funkcije su građni prvog reda, tako da ih možemo prosledivati 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, 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.

<sup>2</sup>Formalna definicija u dodatku

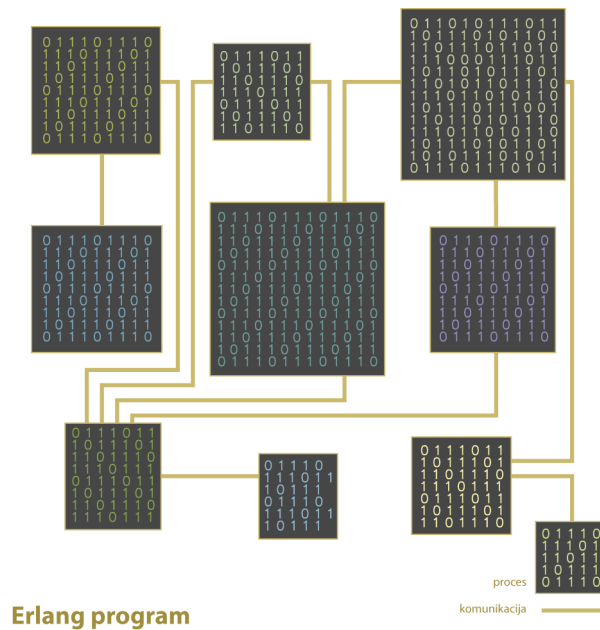
<sup>3</sup>Operatori poređenja su: `<`, `=<`, `>`, `=>`, `==`, `/=`, `:=`, `:=/`

## 5 Konkurentnost kao glavna specifičnost

Jedna od osnovnih osobina Erlanga i specifičnost po којom 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. [7] 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 Implemetacija

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 nikad ne dele memoriju, već komuniciraju isključivo asinhronim slanjem poruka [6]. 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 programi napisani u Erlangu izvršavaju na BEAM virtuelnoj mašini. Tako, procesi u Erlangu se prave i uništavaju jako brzo, zauzimaju samo onoliko memorije koliko je neophodno, u većini slučajeva jako malo i ponašaju se isto na svim operativnim sistemima [6].



Slika 1: Konkurentnost u Erlangu

Poređenja radi, koncept konkurentnosti korišćen od strane većine programskih jezika je takozvani koncept konkurentnosti deljenih stanja (eng. *shared state concurrency*) gde procesi menjaju memoriju (nasuprot tome, u

Erlangu vrednost promenljivoj može biti dodeljena samo jednom) [6]. U slučaju da više procesa žele da menjaju istu memoriju, tzv. kritična sekcija, potrebno je nekako zaštititi taj deo memorije (muteksi, katanci i dr.). U slučaju da do greške dođe baš u kritičnoj sekciji, ostali procesi ne znaju kako da se nose sa datom situacijom i u najboljem slučaju sistem prekida sa radom.

## 5.2 Slanje i primanje poruka

Erlang omogućava jednostavno 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 '!'. Procesu sa identifikatorom *Pid* šalje se poruka sadržana u promenljivoj *Message*, pri čemu poruka može biti bilo koji validni term. Operator '!' 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.

```
Pid ! Message
```

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 *self()* funkcijom koja vraća *pid* procesa koji šalje zahtev.

```
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. Funkcijom *register* atom se vezuje za pid i na dalje ga možemo indentifikovati preko dodeljenog atoma.

```
register(name, Pid)
```

Glavni razlog registrovanja procesa je da se omogući *klijent-server model* koji je ključni za komunikaciju između procesa u Erlangu. 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.

## 6 Okruženja i njihove karakteristike

Programski jezik 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	✓	-	✓
Integrirani mejl server	✓	-	✓
UTF-8 u Erlang kodu	✓	-	✓
Višejezični podaci	-	-	✓
Generisanje <i>JavaScript</i> koda	-	✓	✓
Generisanje <i>JSON</i> formata	✓	✓	✓
Integrirani 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* [5] 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 *ChicagoBoss* izgleda ovako:

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

U tabeli 1 videli smo da sva tri okruženja podržavaju i okruženje za testove, gde su testovi struktuirani kao stabla nastavaka (eng. *trees of*



*continuations*). Detaljnije o ovome možete pogledati u radu autora *ChicagoBoss* okruženja [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 webu 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. Za *Nitrogen* [4] je karakteran *web DSL* [10] sa korišćenjem *Nitrogen* elemenata [4] što u suštini omogućava pisanje HTML/Javascript korišćenjem Erlang uslova pre nego HTML uslova.

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.

## 7 Instalacija i pokretanje

Postoji više načina da se instalira Erlang sa neophodnim paketima. U ovom poglavlju će biti predstavljena instalacija korišćenjem prekompajliranih binarnih fajlova za neke operativne sisteme zasnovane na Linuksom 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* [6]. Erlang interpretator ima u sebi ugrađen editor teksta koji je baziran na *emacs-u* [8]. 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 *Linux* sistemima. Ukoliko naidete na neki problem sa instalacijom i kompilacijom možete pronaći rešenje u dokumentaciji programskog jezika [9].

## 8 Primeri kodova sa objašnjenjima

Poćećemo od jednostavnog primera "Hello World" i videti osnovnu sintaksu. 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!
```

Pošto smo videli kako string možemo ispisati na standardni izlaz, sada ćemo videti kako broj možemo ispisati. Sledeći program prikazuje zbir dva intiger-a.

```
-module(helloworld).
-export([start/0]).

start() ->
    io:fwrite("~w", [1+1]).

> 2
```

Na sličan način možemo ispisivati i razne druge tipove.

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 primenom 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:

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

Prvo kreiramo novi serverski proces i dobijemo njegov pid, potom ga registrujemo i pozovemo funkciju *klijent* koja enkapsulira slanje zahteva i primanje odgovora. U njoj klijent mora da uključi svoju adresu sa *self()*. Potom čeka odgovor koji je namenjen njemu, odnosno prihvata odgovor kada se poklopi obrazac torke {Pid, Response}.

```
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 na:

```
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

## 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] Nitrogen framework documentation. on-line at: <http://nitrogenproject.com/doc/index.html>.
- [5] Zotonic framework documentation. on-line at: <http://docs.zotonic.com/en/latest/index.html>.
- [6] J. Armstrong. *Programming Erlang (2nd edition)*. Pragmatic Bookshelf, 2013.
- [7] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.
- [8] F. Hebert. *Learn You Some Erlang for Great Good!* No Starch Press, 2013.
- [9] OTP team. Erlang. on-line at: <http://www.erlang.org/>.
- [10] Zef Hemel, Danny M. Groenewegen, Lennart C.L. Kats, Eelco Visser. Webdsl: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, USA, October 2008.

## A Dodatak