Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 4.92
DOI: https://doi.org/10.24867/33BE13Ruzicic

RAZVOJ INTERPETERA U PROGRAMSKOM JEZIKU GO

DEVELOPMENT OF AN INTERPRETER IN THE GO PROGRAMMING LANGUAGE

Ratko Ružičić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – RAČUNARSTVO I AUTOMATIKA

Kratak sadržaj – *Ovaj rad bavi se izradom interpretera hipotetičkog programskog jezika, on predstavlja nastavak prethodnog rada autora na temu kompajlera izrađenog u programskom jeziku C, koji je koristio alate "flex" i "bison" za faze leksičke analize i parsiranja. Za razliku od tog rada, interpretator predstavljen u ovoj tezi razvijen je isključivo korišćenjem standardne biblioteke programskog jezika Go, bez oslanjanja na dodatne alate ili postojeći kod.*

Ključne reči: *Programski jezici, Interpreteri, Dinamički tipovi*

Abstract – *This paper explores the implementation of a hypothetical programming language, it is an extension of a previous paper by the same author of a compiler implemented in C programming language using tools "flex" and "bison" for lexical analysis and parsing. In contrast to that paper, an interpreter presented here was developed exclusively using a standard library of Go programming language without relying on any other tools or existing code.*

Keywords: *Programming languages, Interpreters, Dynamic types*

1. UVOD

1.1. Definicija problema

Za početak neophodno je definisati programski jezik koji bi imao sledeće funkcionalne osobine:

- dinamički sistem tipova
- podrška za kondicionale(if...else)
- podrška za funkcije
- osnovni aritmetički i logički operatori
- nizovi
- petlje
- osnovni tipovi podataka: integer, double, boolean, string
- interaktivni(REPL) mod i učitavanje programa iz fajla

Pored funkcionalnih zahteva interpreter bi trebao da zadovolji i određene nefunkcionalne zahteve. Najbitniji među njima jeste lako distribuiranje interpretera i omogućavanje krajnjim korisnicima da preuzmu interpreter i da ga pokrenu bez potrebe za mukotrpnim podešavanjem radnog okruženja i instaliranja dodatnih programa. Kao dodatni zahtev može se izdvojiti i pokretanje interpretera uz pomoć samo jedne komande.

1.1. Opseg i ograničenja

Polje kompajlera i interpretera je jako složeno i vrlo dobro proučavano, takođe postoji vrlo dobra povezanost sa industrijom pa je mnogo novca i vremena uloženo u proučavanje ove oblasti od strane vrlo uspešnih kompanija. Uzevši to u obzir može se zaključiti da su današnji interpeteri često rezultat višedecenijskog napora što od strane samih kompanija, što od strane nezavisnih kontributora što samo potvrđuje činjenicu da je dizajniranje novog programskog jezika i implementiranje interpretera za njega ogroman poduhvat i da je put od rane implementacije do prihvatanja od strane industrije i entuzijasta vrlo dug i jako nepredvidiv [1].

2. TEORIJA

2.1. Leksička analiza

Leksička analiza je prvi korak prilikom interpretacije (ili kompilacije) programskog jezika. Ulaz u leksičku analizu predstavlja "sirovi" tekst sačinjen od niza karaktera a izlaz predstavlja niz tokena. Tako na primer ako postoji ulaz sadržine "var a = 3;". Kao izlaz se može očekivati niz tokena: "VAR, IDENTIFIER, ASSIGN, NUMBER, SEMICOLON". Pored toga očekuje se da leksička analiza javi grešku u slučaju da za dati ulaz nije moguće generisati listu tokena ili da pak dostavi token koji bi označio da za dati unos postoji greška. Na primer, za ulaz "var @ = 3;", očekivani izlaz bi bio: "VAR, ERROR". Dakle lekser (nekada u literaturi nazvan i skener) je javio da je prvi token VAR a da njega sledi ERROR token pošto @ karakter nije podržan u gramatici jezika. U ovom primeru može se uočiti da je lekser efektivno prestao sa radom u momentu kada je naišao na karakter koji je uzrokovao grešku, u praksi ovo često nije slučaj. Skoro sve implementacije modernih jezika omogućavaju dalji rad leksera kako bi pronašao što više grešaka i o tome obavestio krajnjeg korisnika koji bi onda dobio listing grešaka koje treba da reši.

Današnji lekseri takođe uz sam token generišu i značajnu količinu metapodataka o samom tokenu, osnovni metapodaci mogu biti broj linije i broj karaktera unutar fajla nad kojim je vršeno leksiranje, dok neki napredniji metapodaci mogu biti dužina same lekseme, sadržaj lekseme i ime fajla u kom je leksema pronađena.

NAPOMENA:

Ovaj rad proistekao je iz master rada čiji mentor je bio dr Dunja Vrbaški, docent

2.2. AST

Kako bi opisali značenje i značaj AST-a moguće je osvrnuti se na sam akronim(u obrnutom redosledu): stabla (*eng. Tree*) označavaju da se radi o običnoj strukturi podataka stabla koje su jako česte u kompjuterskoj nauci, sintaksna (*eng. Syntax*) označava da se radi o sintaksi samog jezika tj. da se sintaksa samog jezika može opisati jednim takvim stablom i apstraktna(*eng. Abstract*) u ovom slučaju znači da ova stabla predstavljaju strukturu a ne da mapiraju svaki karakter na svaki čvor u datom stablu.

2.3. Parsiranje

Sledeća faza u interpretaciji programa jeste parsiranje, zadatak parsera programskog jezika jeste da definiše sintaksu jezika tj. da preuzme tokene dobijene iz prethodnog koraka (leksičke analize) i da utvrdi da li je redosled tih tokena ispravan i da dodatno vrati neku strukturu podataka koja bi bila reprezent samog programa.

Parseri se uglavnom mogu ručno implementirati i većina implementacija programskih jezika ima ručno napisane parsere ali takođe postoje alati za generisanje koda parsera poput “ANTLR”, “bison” ili “yacc” alata koji za definisanu formalnu gramatiku generišu skelet koda parsera a korisnik može definisati svoju logiku provere semantike jezika, pa čak i generisanje koda.

Parseri se ugrubo mogu podeliti u 2 različite kategorije, “top-down” i “bottom-up”. Glavna razlika između ova dva tipa jeste da “top-down” parseri kreću od korena abstraktnog stabla i pokušavaju da dođu do listova samog stabla pri tom proveravajući da li su zadovoljena pravila gramatike jezika, dok “bottom-up” parseri kreću od samih tokena (tj. listova stabla) i pokušavaju da "izgrade" stablo.

Generalno govoreći “top-down” parseri su lakši za implementaciju ako su ručno implementirani, dok su alati za generisanje parsera najčešće napravljeni da generišu neki derivat “bottom-up” parsera.

Klasičan primer “top-down” parsera je Pratov parser (takođe poznat kao "recursive descent parser") koji je osmislio Von Prat 1970-ih godina. Pratov parser se u suštini bavi parsiranjem izraza i rešavanjem problema redosleda aritmetičkih operacija nad izrazima, on nudi elegantan način da se predstave odnosi između različitih operacija. Sve što je potrebno jeste da se definišu operatori i njihov prioritet i Pratov parser će znati kako tačno da grupiše operatore. Ako se na primer uzme sledeći izraz za razmatranje “1 + 2 * 3 - 10 / 5”, onda izlaz Pratovog parsera može biti sama vrednost izraza, u ovom slučaju “5”, ili isti izraz ali u formatu grupisanom uz pomoć zagrada tj. “(1 + ((2 * 3) - (10 / 5)))” ili na kraju krajeva kao apstraktno sintaksno stablo koje će prikazati koje podizraze treba evaluirati prvo.

2.4. Evaluacija

Evaluacija se odnosi na korak u interpretaciji u kom se u apstraktnom sintaksnom stablu posećuju svi čvorovi i na osnovu tipa čvora (i njegovih atributa) vrši evaluacija izraza. Radi o jednostavnoj implementaciji "šetanja" kroz stablo i implementiranje logike evaluacije.
Sve do koraka evaluacije razlike između interpretera i kompajlera nije bilo ali u procesu evaluacije je došlo do bitnog razdvajanja. U ovom koraku su zaista bile evaluirane vrednosti izraza ali ako bi cilj bio da se implementira kompajler onda to ne bi bilo moguće učiniti. Kod implementacije kompajlera isti bi generisao nekakav kod, bilo da je u pitanju mašinski kod, asemblerski kod ili nekakav bajtkod za neku hipotetičku (ili pravu) virtuelnu mašinu. Razlika deluje suptilno, i u primeru jednog "hobi" programskog jezika ona suštinski to i jeste ali ako se sagleda šira slika primetno je da interpreter zapravo izvršava sve što mu je dato što znači da ako bi interpreter želeo da pristupi fajlovima ili da otvori nekakvu datoteku on bi morao da to uradi preko jezika u kome je implementiran, dok bi kompajler za taj korak samo generisao mašinski kod koji bi uputio par sistemskih poziva i otvorio fajl a taj mašinski kod bi kasnije bio i izvršen. Svaki moderan programski jezik današnjice podržava pristup fajlovima sa fajl sistema tako da to nije realan problem ali i dalje treba biti svestan činjenice da izbor jezika u kome se implementira interpreter može mnogo da doprinese performansama i funkcionalnostima rezultujućeg interpretera.

3. DIZAJN

Prilikom faze dizajniranja jezika pristupilo se istraživanju koje su popularne osobine jezika današnjice, u tome je najviše pomogao “StackOverflow Developer Survey” tj. anketa koju na kraju svake godine popunjavaju programeri. Između svih programskih jezika najviše su se izdvajali “Python” i “JavaScript” kako među profesionalcima tako i među početnicima pa je prilikom dizajniranja sintakse i odabira osobina jezika bilo logično pokušati imitirati određene osobine tih jezika [2].

3.1. Interpretacija

Prva i glavna osobina ovog programskog jezika jeste to da je isti interpretiran a ne kompajliran. Postoji više razloga zašto je interpretirani jezik ponekad dobar izbor: veća fleksibilnost, prenosivost, manja kriva učenja za početnike itd. No, za krajnjeg korisnika najbitniji razlog jeste prenosivost samog jezika, prilikom konstruisanja kompajlera bitna je odluka koju će arhitekturu pogađati kompajler: ARM, x86 ili PowerPC itd., mada je u današnje vreme donošenje takve odluke olakšano postojanjem radnih okvira za kompajlere poput LLVM. Bilo kako bilo, definitivno je lakše napisati interpreter u nekom dobro podržanom jeziku koji se može izvršavati na skoro svakoj arhitekturi (a takvih jezika danas ne manjka pogotovu zbog prethodno pomenutog LLVM) i time obezbediti da se i ovaj interpreter može pokrenuti na istim tim arhitekturama.

3.2. Dinamički tipovi

Inicijalna zamisao bila je da interpreter podržava statičke tipove, i prva implementacija interpretera zaista jeste imala statičke tipove, ali tokom korišćenja interpretera došlo se do zaključka da je ipak bolje slediti primere čisto interpretiranih jezika koji podržavaju dinamičke tipove. Većina interpretera zaista jeste dinamički tipizirana i iako možda jeste moguće naći primere statički tipiziranog jezika koji je interpreteran, npr. “TypeScript” (doduše pitanje je koliko je ovo dobar primer pošto se “TypeScript” zapravo transpajlira u “JavaScript” koji je interpretiran i dinamički tipiziran), glavni razlog jeste lakoća upotrebe jezika i to je definitivno tačno kada se radi o malim programima i

kratkim skriptama, a budući da će većina programa pisanim u jeziku koji je implementirao interpreter biti mali programi ili kratke skripte onda je logično da se sa korisnika skine "teret" tipiziranja promenljivih.

3.3. Automatsko upravljanje memorijom

Tokom uobičajenog rada programa isti zauzima i oslobađa određenu količinu radne memorije. U računarstvu, memorija je ograničen a često i veoma vredan resurs o kome se mora pažljivo voditi računa, pa je jako bitno minimizovati količinu upotrebljene memorije. Postoje dva pristupa u rukovođenju radnom memorijom. Prvi pristup jeste da se rukovođenje memorijom prepusti programeru koji implementira program. On će određivati trenutke kada i koliko memorije će biti zauzeto pozivanjem funkcija iz programskog jezika koje će tokom izvršavanja vršiti sistemske pozive ka operativnom sistemu na kom se program izvršava. U onom trenutku kada određeni komad memorije bude nepotreban, ta memorija se potom može osloboditi. Postoje dva ključna aspekta na koje programer u ovom slučaju uvek mora da misli: prvi, da ne zauzme više memorije nego što mu je neophodno i drugi, da ne zaboravi da oslobodi memoriju koju je zauzeo. Drugi pristup u rukovođenju memorije jeste da se programeru oduzme sloboda zauzimanja i oslobađanja memorije i da se rukovođenje prepusti kompajleru ili interpreteru. Ovaj deo posla vrši zasebna komponenta interpretera koja se zove modul za automatsko upravljanje memorijom, a češće se koristi pojam na engleskom tj. "garbage collector".

Kada bi se jezik interpretera implementirao u jeziku koji ne podržava automatsko upravljanje memorijom (npr. C) onda bi bilo nepohodno da se isti i implementira (zapravo postoji i druga opcija a to je da prepustimo zauzimanje/oslobađanje memorije krajnjem korisniku što nije uobičajeno ili treća opcija da nikad ne oslobađamo memoriju što je suludo). Budući da je odabran jezik koji ima ugrađeno automatsko upravljanje memorijom ovaj deo posla je već odrađen pošto svako zauzimanje memorije za promenljive u jeziku koji implementira automatski znači da će se ista operacija prevesti u zauzimanje memorije od strane jezika koji implementira interpreter.

4. ZAKLJUČAK

Interpreteri možda nisu najefikasnije rešenje za izvršavanje koda ali oni nude određene karakteristike koje ih čine vrlo privlačnim u današnjim razvojnim okruženjima poput brzine razvoja aplikacija i lakšeg otkrivanja grešaka prilikom razvoja ali to dolazi po cenu nižih performansi u odnosu na kompajlere.

Današnji intereteri postigli su mnogo u pogledu performansi, pa je ta razlika je dosta manja u odnosu na period pre par decenija što ih čini odličnim kandidatima za započinjanje novih projekata [3].

Svrha ovog rada bila je da se proširi znanje stečeno tokom master i osnovnih studija kao i upoznavanje sa novim konceptima prilikom dizajniranja i implementacije jezika, radi se o kompleksnoj temi koja je predmet istraživanja i industrije i akademije i spada u red onih tehnologija koji predstavljaju temelj modernih informacionih tehnologija.

5. LITERATURA

[1] L. A. Meyerovich and A. S. Rabkin, ”Empirical analysis of programming language adoption”, Association for Computing Machinery, vol. 48, pp. 1-18, Oktobar 2013.

[2] StackOverflow, https://survey.stackoverflow.co/2024/, (pristupljeno u junu 2025.)

[3] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J. Baer, B. N. Bershad, H. M. Levy, ”The structure and performance of interpreters”, Association for Computing Machinery, vol. 31, pp. 150-159, Septembar 1996.

Kratka biografija:

Ratko Ružičić rođen je 2000. godine u Čačku. Završio Tehničku školu u istom gradu 2019. godine. Diplomirao je na Fakultetu Tehničkih Nauka Univerziteta u Novom Sadu 2023. godine.
kontakt: rruzicic@gmail.com