UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ivan Ristović

JEZIČKI INVARIJANTNA PROVERA SEMANTIČKE EKVIVALENTNOSTI STRUKTURNO SLIČNIH SEGMENATA IMPERATIVNOG KODA

master rad

Beograd, 2020.

Mentor:

doc. dr Milena Vujošević-Janičić
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić
University of Disneyland, Nedođija

dr Laza Lazić
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

TODO zahvalnica

Naslov master rada: Jezički invarijantna provera semantičke ekvivalentnosti strukturno sličnih segmenata imperativnog koda

Rezime: Apstraktno sintaksičko stablo (engl. abstract syntax tree, skraćeno AST) nastaje kao rezultat parsiranja ulaznog programa i predstavlja osnovu za semantičku analizu koda. U okviru semantičke analize koda veoma su važne analize koje omogućavaju obezbeđivanje visokog kvaliteta softvera, bilo kroz analize koje uključuju statičko otkrivanje grešaka u kodu ili kroz analize koje imaju za cilj automatsku transformaciju koda u semantički ekvivalentan oblik sa nekom željenom karakteristikom (na primer, u okviru procesa refaktorisanja koda). Međutim, apstraktno sintaksno stablo je specifično za konkretan viši programski jezik i na osnovu njega se ne mogu porediti karakteristike programa napisanih u različitim programskim jezicima, što je posebno važno u okviru procesa migracije na nove tehnologije. Osnovni cilj rada je kreiranje i implementacija opšte AST reprezentacije za imperativne programske jezike sa ciljem njene primene u analizi semantičke ekvivalentnosti implementacija algoritama u različitim programskim jezicima. Implementacija će dozvoljavati kreiranje apstrakcije na osnovu zadate gramatike imperativnog programskog jezika, pri čemu će koncept biti prikazan na primeru jednostavnog pseudokoda, kao i na programskim jezicima C i Lua. Implementacija će biti urađena u programskom jeziku C# koristeći ANTLR4 generator parsera.

Ključne reči: TODO

Sadržaj

Glava 1

Uvod

Apstraktna sintaksna stabla (engl. *abstract syntax tree*, skr. *AST*) imaju veliku ulogu u procesu analize programa. Na osnovu toga da li se analiza vrši pre ili u toku izvršavanja programa, te analize mogu biti *statičke* i *dinamičke*, pri čemu se AST najviše koristi za statičku analizu. Statičke analiza omogućava otkivanje raznih grešaka veoma rano, u nekim slučajevima čak i tokom pisanja problematičnog dela programa.

AST nastaje parsiranjem izvornog koda, kao rezultat apstrahovanja stabla parsiranja koje generiše parser - program koji parsira izvorni kod i identifikuje sintaksne konstrukte jezika. Parser pokušava da u izvornom kodu pronađe određena gramatička pravila jezika koji je dizajniran da prepozna. Svaki programski jezik ima specifična sintaksna pravila pa stoga su i gramatike programskih jezika raznorodne, što se stoga prenosi i na stabla parsiranja. Za statičku analizu često nije neophodno posmatrati kod na nivou parsera (kroz stablo parsiranja), pošto informacije koje su neophodne parseru često nisu potrebne za neku konkretnu analizu. Stoga se stablo parsiranja često apstrahuje tako da što iz njega izvuku bitne sintaksne informacije - što za rezultat daje AST.

Posmatranje programa na apstraktnom nivou kroz AST pruža mogućnost za poređenje dva programa na apstraktnom nivou. Jedna od primena može biti semantička ekvivalentnost, u okviru statičke analize. Semantička ekvivalentnost je neodlučiv problem u opštem slučaju, međutim pod određenim pretpostavkama je moguće kreirati algoritme koji daju smislene rezultate. Te pretpostavke mogu uključiti i sličnost u strukturi programa, što se takođe može odraziti i na izgled i sličnosti u strukturi njihovih AST-ova, problem koji se može rešiti primenom algoritama za rad sa stablima (ali i grafovima uopšte, jer su stablo specijalizacija

grafa).

Iako veoma konceptualno moćan alat, AST je previše specifičan za konkretni programski jezik, s obzirom da nastaje od stabla parsiranja koje je usko vezano za gramatiku konkretnog programskog jezika. Motivacija za ovaj rad počiva u nepostojanju opštih apstrakcija za više programskih jezika. Danas postoji dosta programskih jezika ali u okviru iste programske paradigme ne postoji puno mesta za inovacije, s obzirom da su koncepti koje programski jezici te paradigme pružaju univerzalni. U ovom radu će biti predstavljena opšta AST apstrakcija za proceduralne programske jezike, sa ciljem da je moguće što više proceduralnih jezika dovesti na istu apstraktnu reprezentaciju, čak i programske jezike koji pripadaju skript paradigmi. Takođe, na apstraktnom nivou nije važno od kog se programskog jezika dobio AST, što ima veliku važnost u procesu migracije na nove tehnologije, s obzirom da je prepisivanje programa sa jednog jezika na drugi danas veoma česta operacija.

Naravno, semantička ekvivalentnost se ne mora zasnivati na apstrahovanju programa, već se takođe često zasniva na spuštanju na nivo međukoda između visokog programskog jezika i asemblera, u nekim slučajevima i do mašinskog jezika. U ovom radu je odabran apstraktni pristup s obzirom na pomenutu važnost AST-ova i nedostatkom opštih apstrakcija.

U poglavlju 2 će biti opisani relevantni pojmovi potrebni za razumevanje rada uz akcenat na apstraktnim sintaksnim stablima i procesu dobijanja istih. Opšta AST apstrakcija za imperativne jezike biće opisana u poglavlju 3, a njena upotreba u problemu odlučivanja semantičke ekvivalentnosti kao i sam algoritam za poređenje opštih apstrakcija biće opisani u poglavlju 4. Implementacija apstrakcije i algoritama semantičkog poređenja će biti opisana u poglavlju 5. Na kraju, biće dati glavni zaključci ovog rada kao i moguća unapređenja i budući koraci.

Glava 2

Pregled relevantnih pojmova

U ovom poglavlju će biti opisani koncepti i alati čije je razumevanje potrebno kako bi se razumeo opis dalje apstrakcije i implementacije samog programa. Umesto analize samog sadržaja izvornog koda analizira se *apstraktno sintaksno stablo* (eng. *Abstract Syntax Tree*, u daljem tekstu *AST*), opisano u odeljku 2.1. Više reči o alatima koji mogu da od proizvoljne gramatike kreiraju leksere i parsere biće u 2.4, sa akcentom na alatu *Another Tool For Language Recognition* [1], u daljem tekstu *ANTLR*, opisan u odeljku 2.4. Parser generiše AST specifičan za datu gramatiku i nema sličnosti u dobijenim apstrakcijama za različite jezike. Kako bismo poredili stabla različitih jezika, kreiramo reprezentaciju na višem nivou i specifični AST podižemo na taj nivo. Ta reprezentacija će biti opisana u narednim poglavljima, kao i načini kako se ona može analizirati. Takođe, pojmovi specifični za implementaciju će takođe biti opisani u ovom poglavlju.

2.1 Apstraktna sintaksna stabla - AST

Kako bi se kod pisan u nekom programskom jeziku (*izvorni fajl*) preveo u kod koji će se izvršavati na nekoj mašini (*izvršivi fajl*), prevodilac prolazi kroz određene korake. Da bi se izvorni kod preveo, prevodilac mora da zna njegovu formu, ili *sintaksu*, i njegovo značenje, ili *semantiku*. Deo prevodioca koji određuje da li je izvorni kod ispravno formiran u terminima sintakse i semantike se naziva *prednji deo* (engl. *front end*). Ukoliko je izvorni kod ispravan, prednji deo kreira *međureprezentaciju* koda (engl. *intermediate representation*, u daljem tekstu *IR*). Ukoliko to nije slučaj, prevođenje ne uspeva i programeru se daje poruka o detaljima zašto prevođenje nije uspelo [5].

Postupak rada prednjeg dela će biti opisan kroz konkretan primer. Pretpostavimo da želimo da prevedemo kod pisan u programskom jeziku C prikazan na slici 2.1. Primetimo da postoji greška u datom kodu - simbol `c` koji se koristi u dodeli u liniji 8 će biti prepoznat kao identifikator koji ne odgovara nijednoj deklarisanoj promenljivoj - stoga ne možemo prevesti ovaj kod. Ovo, doduše, nije sintaksna greška - izraz `a+c` je sasvim validan u programskom jeziku C bez analize konteksta u kom se javlja. Problem će postati očigledan tek nakon parsiranja izvornog koda i provere ispunjenosti sintaksih pravila, tačnije u fazi semantičke provere. Stoga se ovakve greške nazivaju *semantičke greške*, dok se greške u sintaksi nazivaju *sintaksne greške*.

```
1   #include<stdio.h>
2
3   #define T int
4
5   int main()
6   {
7       T a, b;
8       a = a + c;          // c nije deklarisano
9       printf("%d", a);
10      return 0;
11  }
```

Slika 2.1: Primer izvornog koda pisanog u programskom jeziku C.

Pre nego što prednji kraj prevodioca uopšte dobije kod koji treba prevesti, vrši se *pretprocesiranje* od strane programa koji se naziva *pretprocesor*. U fazi pretprocesiranja se izvode samo tekstualne operacije kao što su brisanje komentara ili zamena makroa u jezicima kao što je C. Rezultat rada pretprocesora za kod sa slike 2.1 bi izgledao kao na slici 2.2 [1].

Da bi proverio sintaksu izvornog koda, prevodilac mora da uporedi strukturu istog sa unapred definisanom strukturom za određeni programski jezik. Ovo zahteva formalnu definiciju sintakse jezika. Programski jezik možemo posmatrati kao skup *pravila* koji se naziva *gramatika* [4], prikazana na slici 2.3. U prednjem

[1] U nekim implementacijama C standardne biblioteke, moguće je da se poziv funckije `printf` zameni pozivom funkcije `fprintf` sa ispisom na `stdout`. U standardu se propisuje da funkcije kao što je `printf` mogu biti implementirane kao makroi. Izlaz na slici 2.2 je generisan od strane `GCC 7.4.0` po C11 standardu i ovo nije slučaj u datom okruženju.

```
1       # 1 "<stdin>"
2       # 1 "<built-in>"
3       # 1 "<command-line>"
4       # 31 "<command-line>"
5       # 1 "/usr/include/stdc-predef.h" 1 3 4
6
7       ...
8
9       extern char *ctermid (char *__s) __attribute__
           ((__nothrow__ , __leaf__));
10      # 840 "/usr/include/stdio.h" 3 4
11      extern void flockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__));
12      extern int ftrylockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__)) ;
13      extern void funlockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__));
14      # 868 "/usr/include/stdio.h" 3 4
15      # 2 "<stdin>" 2
16      # 2 "<stdin>"
17
18      int main()
19      {
20          int a, b;
21          a = a + c;
22          printf("%d", a);
23          return 0;
24      }
```

Slika 2.2: Prikaz rezultata rada pretprocesora za izvorni kod sa slike 2.1. Pritom, prikazano je samo par linija sa početka i kraja izlaza pretprocesora - kod iznad `main` funkcije je uključen iz `stdio.h` zaglavlja.

delu se izvode dva procesa koji određuju da li ulaz zaista zadovoljava gramatiku određenog programskog jezika. Ova dva procesa se nazivaju *skeniranje* i *parsiranje*, a komponente prednjeg dela koje vrše te procese se nazivaju *skener* (takođe se naziva i *lekser*) i *parser*, redom.

Prilikom faze prevođenja, kako prevodilac ne bi radio nad sirovim karakterima izvornog koda, potrebno je izvršiti pripremu istog - skeniranje. Prevodilac ima u vidu moguće elemente programskog jezika, tzv. *tokene*, koje treba prepoznati u datom fajlu - ključne reči, operatore, promenljive itd. Proces prepoznavanja

```
1     functionDefinition
2         :   declarationSpecifiers? declarator
3             declarationList? compoundStatement
4         ;
5
6     declarationList
7         :   declaration
8         |   declarationList declaration
9         ;
10
11    declaration
12        :   declarationSpecifiers initDeclaratorList ';'
13        |   declarationSpecifiers ';'
14        |   staticAssertDeclaration
15        ;
```

Slika 2.3: Isečak gramatike programskog jezika C po standardu C11.

tokena u izvornom fajlu se naziva *tokenizacija*. Pojednostavljen primer tokena koje lekser pokušava da prepozna se može videti na slici 2.4. Primer izlaza leksera za izlaz pretprocesora sa slike 2.2 se može videti na slici 2.5. [2]

```
1     Identifier : IdentifierNondigit
2                  (IdentifierNondigit | Digit)*
3                ;
4
5     IdentifierNondigit : Nondigit
6                        | UniversalCharacterName
7                        ;
8
9     Nondigit : [a-zA-Z_]
10              ;
11
12    Digit : [0-9]
13          ;
```

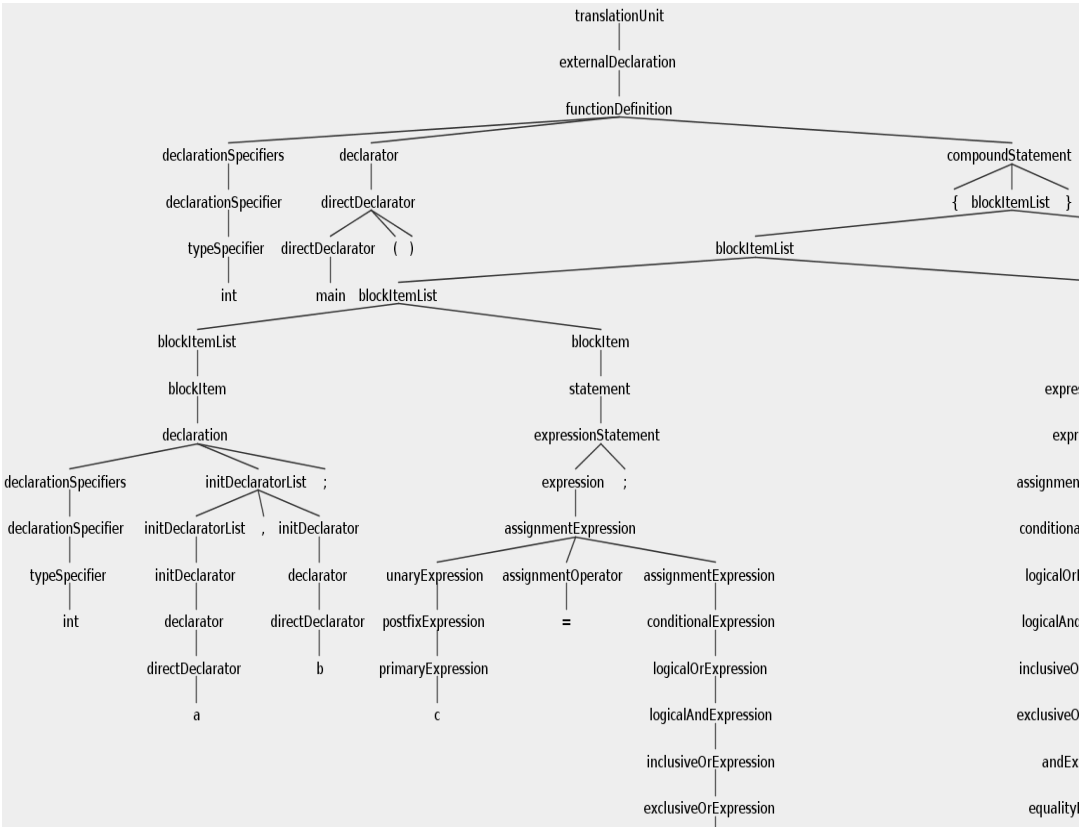
Slika 2.4: Primer delimične definicije tokena za ime promenljive po C11 standardu.

[2]Moderni kompajleri često nemaju odvojene faze u kojima se pozivaju skeniranja i parsiranja, već se skeniranje odvija paralelno sa fazom parsiranja. Međutim, to nas ne sprečava da ispišemo tokene onda kada se oni prepoznaju, i to je demonstrirano na slici 2.5.

```
1     identifier 'main'    [LeadingSpace] Loc=<sample.c:3:5>
2     l_paren '('     Loc=<sample.c:3:9>
3     r_paren ')'     Loc=<sample.c:3:10>
4     l_brace '{'  [StartOfLine]  Loc=<sample.c:4:1>
5     int 'int'    [StartOfLine] [LeadingSpace]
        Loc=<sample.c:5:5>
6     identifier 'a'   [LeadingSpace] Loc=<sample.c:5:9>
7     comma ','       Loc=<sample.c:5:10>
8     identifier 'b'   [LeadingSpace] Loc=<sample.c:5:12>
9     semi ';'        Loc=<sample.c:5:13>
10    identifier 'a'   [StartOfLine] [LeadingSpace]
        Loc=<sample.c:6:5>
11    equal '='    [LeadingSpace] Loc=<sample.c:6:7>
12    identifier 'a'   [LeadingSpace] Loc=<sample.c:6:9>
13    plus '+'     [LeadingSpace] Loc=<sample.c:6:11>
14    identifier 'c'   [LeadingSpace] Loc=<sample.c:6:13>
15    semi ';'        Loc=<sample.c:6:14>
16    identifier 'printf'  [StartOfLine] [LeadingSpace]
        Loc=<sample.c:7:5>
17    l_paren '('     Loc=<sample.c:7:11>
18    string_literal '"%d"'       Loc=<sample.c:7:12>
19    comma ','       Loc=<sample.c:7:16>
20    identifier 'a'   [LeadingSpace] Loc=<sample.c:7:18>
21    r_paren ')'     Loc=<sample.c:7:19>
22    semi ';'        Loc=<sample.c:7:20>
23    return 'return'  [StartOfLine] [LeadingSpace]
        Loc=<sample.c:8:5>
24    numeric_constant '0'     [LeadingSpace]
        Loc=<sample.c:8:12>
25    semi ';'        Loc=<sample.c:8:13>
26    r_brace '}'  [StartOfLine]  Loc=<sample.c:9:1>
27    eof ''      Loc=<sample.c:9:2>
```

Slika 2.5: Proces tokenizacije koda sa slike 2.2. Generisano uz pomoć `clang` [3] kompajlera.

Nakon faze skeniranja potrebno je parsirati dobijene tokene. Parser, imajući u vidu gramatiku jezika, pokušava da kreira *stablo parsiranja* (eng. *parse tree* ili *derivation tree*). Takvo stablo i dalje sadrži sve relevantne informacije o izvornom kodu. Vizuelni prikaz rada parsera za gramatiku sa slike C11 i izvonog koda sa slike 2.2 je dat na slici 2.6. Stablo parsiranja se koristi u narednim fazama prevođenja.

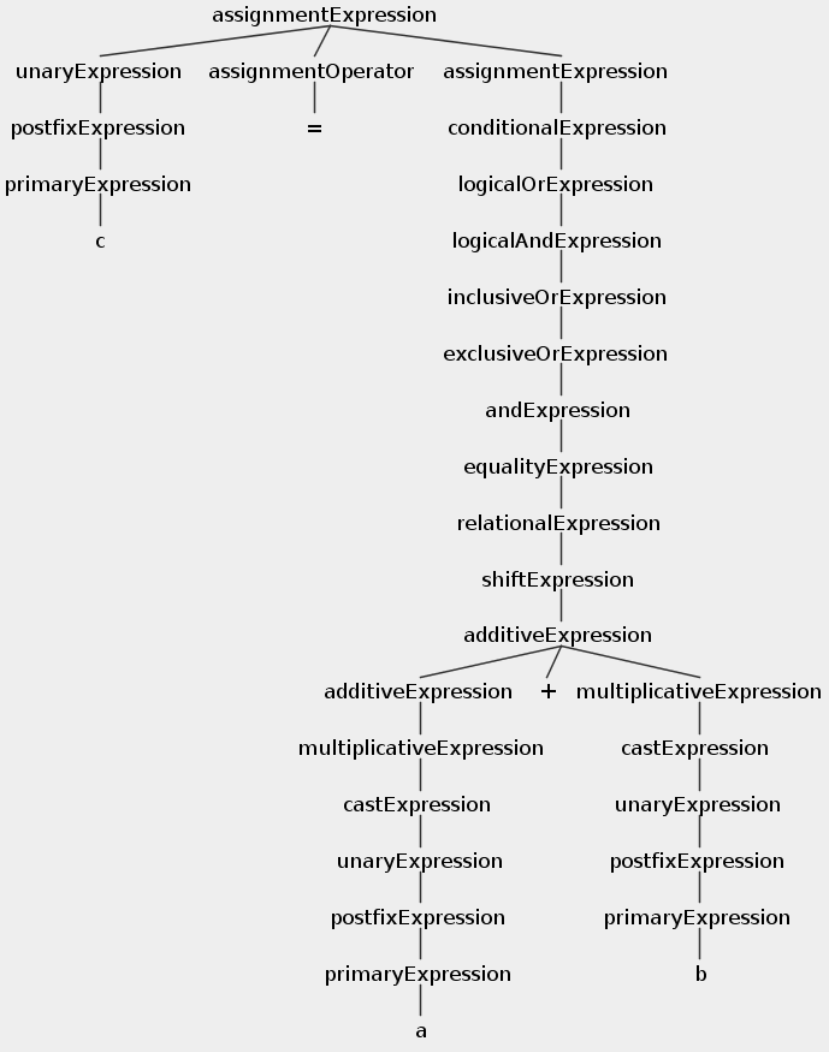


Slika 2.6: Prikaz dela stabla parsiranja koje generiše parser kreiran od strane alata ANTLR4 za kod sa slike 2.2.

Za potrebe ovog rada, što se procesa prevođenja tiče, dovoljno je pozvnavanje prednjeg dela, stoga neće biti reči o daljim koracima u fazi prevođenja (semantička provera, optimizacije, generisanje IR). Zainteresovani čitalac može više detalja pronaći u [5] i [24].

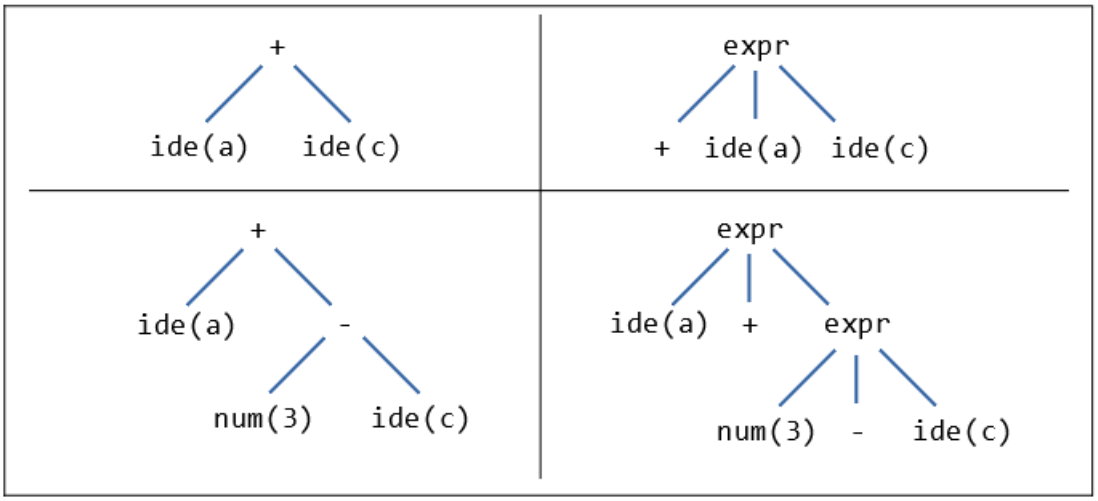
Stablo parsiranja sadrži sve informacije potrebne u fazi parsiranja uključujući detalje korisne samo za parser prilikom provere ispunjenosti gramatičkih pravila. Sa druge strane, *apstraktno sintaksno stablo* sadrži samo sintaksnu strukturu u jednostavnijoj formi. Na slici 2.7 se može videti koliko je stablo parsiranja komplikovano čak i za naizgled jednostavne aritmetičke izraze. Razlog ovolike komplikovanosti dolazi iz rekurzivnih pravila definisanih u C11 gramatici. Parseru su sve ove informacije neophodne ali na apstratknijem nivou nisu potrebne. Jedina važna semantička odlika izraza `a+c` je ta da je to zbir vrednosti nekih promenljivih - sve ostale informacije su nepotrebne. Na slici 2.8 se mogu videti različita apstraktna sintaksna stabla za pomenuti izraz, ali takođe i za malo komplikovanije izraze.

Podrazumeva se, naravno, da je ulaz već tokenizovan.



Slika 2.7: Prikaz kompleksnosti stabla parsiranja za izraz `a+c` u C11 gramatici.

Uloga apstraktnih sintaksnih stabala [20] je da pokažu semantiku strukture koda preko stabala. Kao što se vidi na slici 2.8, postoji određeni nivo slobode u dizajniranju ovih stabala. Generalno, *terminalni simboli*, simboli koji predstavljaju listove stabla parsera, koji odgovaraju operatorima i naredbama se podižu naviše i postaju koreni podstabala, dok se njihovi operandi ostavljaju kao njihovi potomci u stablu. Desna stabla sa slike ne prate u potpunosti ovaj princip, ali se takođe koriste zbog regularnosti izraza - recimo ukoliko binarni izraz posmatramo kao koncept, mnogo je lakše raditi sa ovakvom strukturom. Ovakva struktura će biti korišćena kasnije u implementaciji programa. Primetimo takođe da se u stablima



Slika 2.8: Varijante apstraktnih sintaksnih stabala bez regularnosti(levo) i sa regularnošću (desno) za izraze `a+c` (gore) i `a + (3 - c)` (dole).

za izraz `a + (3 - c)` (dole) implicitno sačuvala informacija o prioritetu operacije oduzimanja u izrazu. Jasno je, dakle, da se računanje vrednosti aritmetičkih izraza onda vrši kretanjem od listova stabla ka korenu. Takođe, pošto su apstraktna sintaksna stabla apstrakcija stabla parsiranja, više istih izraza jezika može imati isto apstraktno sintaksno stablo ali različito stablo parsiranja; na primer, ako razmatramo izraz `(a + 5) - x / 2` i izraz `a + 5 - (x / 2)`.

Apstrakna sintaksna stabla će u daljem tekstu biti referisana skraćenicom *AST*, koja dolazi od engleskog naziva *Abstract Syntax Trees*. Takođe, reči samom dizajnu i tipovima čvorova AST-a korišćenih u implementaciji će biti u poglavlju 3. O procesu generisanja leksera i parsera za datu gramatiku programskog jezika će više biti reči u poglavlju 2.4.

2.2 Simboličko izračunavanje

Tokom procesa pronalaženja grešaka u radu softvera koriste se ručno pisani testovi i pregledi koda od strane drugih programera, i ove mere provere koda mogu verifikovati da softver, ili deo softvera, funkcioniše na različitim nivoima komplikovane arhitekture velikih projekata. Uprkos svim ovim merama, greške su i dalje nezaobilazne - jedan test može proveriti ponašanje koda za samo jedan ulaz. S obzirom da je nemoguće testirati sve moguće ulaze zbog njihovog ogromnog broja (ukoliko posmatramo samo funkciju jedne promenljive koja prima 32-bitni ceo

broj, broj mogućih ulaza je 2^{32}) nadamo se da testovi dobro *generalizuju* - da pokrivaju opšte ali i neke specijalne ulaze. To se postiže uočavanjem da se vrednosti ulaza mogu razvrstati u klase po tome kakav izlaz uzrokuju. Ukoliko imamo funkciju koja treba da podeli dva broja, te klase mogu biti celi brojevi, realni brojevi, neke specijalne vrednosti specifične za to šta se testira (u ovom slučaju, 0), kao i granice za tip podataka iz čijeg domena argumenti mogu uzeti vrednost. Čak i ovakav pristup, iako drastično smanjuje broj testova i eliminiše redundantne testove, i dalje zahteva relativno veliki broj testova u slučaju većih projekata i stoga je teško pronaći sve greške, pogotovo u slučajevima koji se retko dešavaju i zavise recimo od stanja drugih komponenti ili pak nekih nedeterminističkih ponašanja. U komplikovanim projektima je teško pokriti čitav izvorni kod testovima (engl. *code coverage*) - iako pokrivenost koda od 100% i dalje ne znači da taj kod ispravno radi.

Statička analiza koda predstavlja analizu izvornog koda bez pokretanja istog. Ideja je baš ispitivanje svih mogućih stanja u kojima se može naći program i testiranje svih mogućih ulaza u jedinicu koja se testira. U praksi se nailazi na puno problema - osnovni je razlika simboličke i programerske apstrakcije.

Simboličko izvršavanje [22] predstavlja sredinu između klasične verifikacije putem pisanja testova i statičke analize koda. Prilikom simboličkog izvršavanja, umesto stvarnih vrednosti ulaza koje se koriste u testovima, koriste se *simboličke promenljive*. Simbolička promenljiva nije vezana za specifičnu vrednost i analiza se dalje vrši samo nad njom - samim tim se istovremeno mogu testirati višestruke klase sličnih ulaza.

Primer simboličkog izvršavanja će biti opisan na isečku C sa slike 2.9. Pretpostavimo da imamo deklarisanje promenljive `a`, `b` i `c` i da se neke operacije izvršavaju nad njima, reprezentovano komentarom. U nekom trenutku se vrednosti tih promenljivih koriste kao uslovi od kojih zavisi prolaznost testa u poslednjoj liniji. Dodelimo svakoj promenljivoj simboličku vrednost - `a` = α, `b` = β, `c` = γ. Možemo izgraditi stablo izvršavanja i uslove koji moraju da važe nad simboličkim vrednostima α, β i γ kako bi test u poslednjoj liniji prošao.

Ukoliko put izvršavanja programa zavisi od simboličke promenljive, kao što je to slučaj za izvorni kod sa slike 2.9, simbolička promenljiva se konceptualno „grana" i analiza se nastavlja za oba slučaja posebno. Tako se dobija drvo izvršavanja, gde svaki put odgovara mnogim individualnim testovima koji bi uzrokovali prolazak izvršavanja tim putem. Vrednosti promenljivih u tim testovima moraju

```
1       int a, b, c;
2
3       // ...
4
5       int x = 0; int y = 0; int z = 0;
6       if (a)
7           x = -2;
8       if (b < 5)  {
9           if (!a && c)
10              y = 1;
11          z = 2;
12      }
13
14      assert(x + y + z != 3);
```

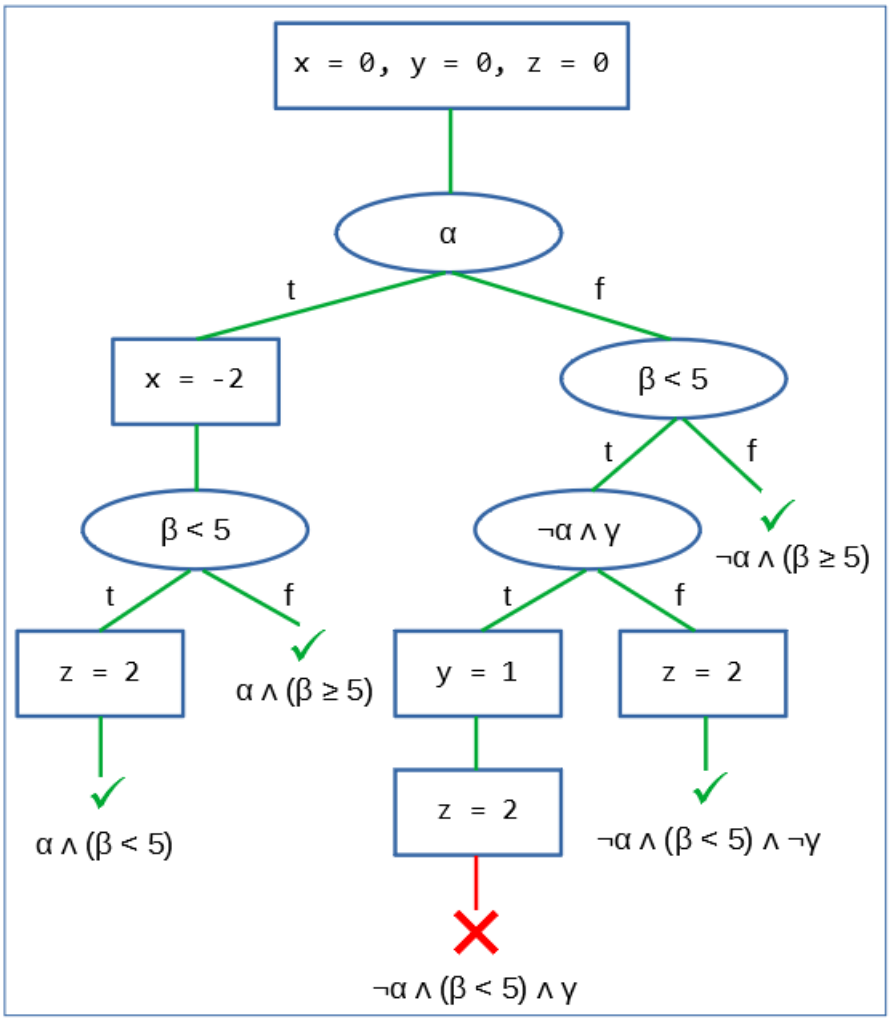
Slika 2.9: Isečak C koda dat kao primer nad kojim će se prikazati simboličko izvršavanje.

zadovoljiti uslove na kraju svakog puta - tzv. *uslove puta* (engl. *path conditions*). Odgovarajuće stablo izvršavanja za izvorni kod sa slike 2.9 sa definisanim simboličkim vrednostima α, β i γ se može videti na slici 2.10. Svaka naredba dodele je uokvirena pravougaonikom dok je uslov uokviren elipsom. Boje grana odgovaraju istinitosnoj vrednosti uslova iz poslednje linije koda u tom trenutku. Na kraju svake grane se nalazi uslov puta za tu granu koje u nekim slučajevima može jedinstveno odrediti vrednost simboličke promenljive koja dovodi do prolaska tim putem ili u opštem slučaju generisati test primer koji dovodi do prolaska tim putem. Dakle, ukoliko je stablo simboličkog izvršavanja poznato, moguće je trivijalno generisati kontra-primere koji dokazuju da program ne radi kao što je očekivano.

Simboličko izvršavanje, iako konceptualno moćno, ima par problema:

- *eksplozija putanja* - Broj puteva izvršavanja eksponencijalno zavisi od broja uslovnih grananja u kodu. Ukoliko imamo 3 naredbe grananja, broj puteva izvršavanja je $2^3 = 8$. Štaviše, petlje su još komplikovanije, jer ukoliko imamo u petlji uslov koji zavisi od simboličke vrednosti koja uzima vrednosti iz opsega 32-bitnog celog broja, broj puteva kroz petlju je u tom slučaju $2^3 1$, a u komplikovanijim slučajevima i beskonačan. Slično važi i za rekuriju.

- *ograničenja rešavača* - U nekim slučajevima je moguće osloniti se na *SMT*



Slika 2.10: Drvo simboličkog izvršavanja na kom su prikazane sve putanje koje se razmatraj. Na kraju svake grane je napisan uslov koji mora da važi da bi se došlo do tog lista u drvetu.

rešavače [21] za nalaženje kontra-primera.

- *modelovanje podataka* (engl. *heap modelling*) - Kreiranje simboličkih struktura podataka i pokazivača nije jednostavna.
- *modelovanje okruženja* (engl. *environment modelling*) - Nije uvek jednostavno adaptirati mehanizam da radi sa čestim potrebama prilikom dizajna softvera - eksterne biblioteke i sistemski pozivi, specifičnosti sistema i okruženja.

Postoji dosta alata i biblioteka koje pružaju simboličko izračunavanje - jedan

od najpoznatijih alata za simboličko izračunavanje je *KLEE* [13], izgrađen nad *LLVM* infrastruktorom [16] i dizajniran za analizu koda pisanog u programskom jeziku C. U nastavku će simboličko izvršavanje biti korišćeno za detekciju razlika u vrednostima promenljivih iz dva AST-a kroz biblioteke za rad sa simboličkim vrednostima u programskom jeziku C#.

2.3 Obrasci za projektovanje

Obrasci za projektovanje (engl. *design patterns* [7], drugačije nazvani i projektni šabloni, uzorci) predstavljaju opšte i ponovno upotrebljivo rešenje čestog problema, često implementirani koristeći koncepte objektno-orijentisanog programiranja. Svaki obrazac za projektovanje ima četiri osnovna elementa:

- ime - ukratko opisuje problem, rešenje i posledice
- problem - opisuje slučaj u kome se obrazac koristi
- rešenje - opisuje elemente dizajna kao i odnos tih elemenata
- posledice - obuhvataju rezultate i ocene primena obrasca

Obrasce za projektovanje je moguće grupisati po situaciji u kojoj se mogu iskoristiti ili načinu na koji rešavaju zadati problem. Stoga je opšte prihvaćena podela na tri grupe:

- *gradivni obrasci* (engl. *creational patterns*)
- *strukturni obrasci* (engl. *structural patterns*)
- *obrasci ponašanja* (engl. *behavioral patterns*)

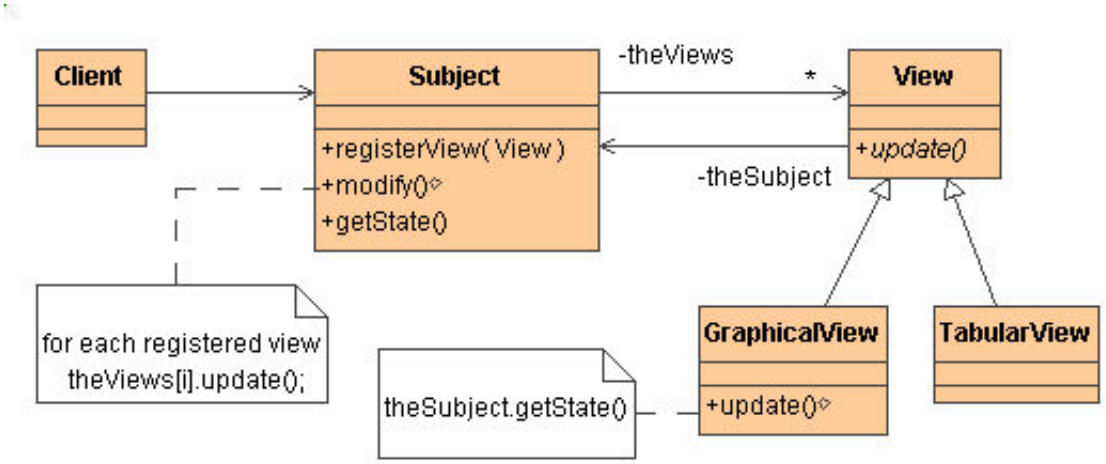
Gradivni obrasci apstrahuju proces pravljenja objekata i važni su kada sistemi više zavise od sastavljanja objekata nego od nasleđivanja. Neki od najvažnijih gradivnih obrazaca su *apstraktna fabrika* (engl. *abstract factory*), *graditelj* (engl. *builder*), *proizvodni metod* (engl. *factory method*), *prototip* (engl. *prototype*) i *unikat* (engl. *singleton*). Strukturni obrasci se bave načinom na koji se klase i objekti sastavljaju u veće strukture. Neki od najvažnijih strukturnih obrazaca su *adapter* (engl. *adapter*), *most* (engl. *bridge*), *sastav* (engl. *composite*), *dekorater* (engl. *decorator*), *fasada* (engl. *facade*), *muva* (engl. *flyweight*) i *proksi* (engl.

proxy). Obrasci ponašanja se bave načinom na koji se klase i objekti sastavljaju u veće strukture. Neki od najvažnijih strukturnih obrazaca su *lanac odgovornosti* (engl. *chain of responsibility*), *komanda* (engl. *command*), *interpretator* (engl. *interpreter*), *iterator* (engl. *iterator*), *posmatrač* (engl. *observer*), *strategija* (engl. *strategy*) i *posetilac* (engl. *visitor*).

Za potrebe ovog rada, obrasci za projektovanje će se koristiti kao opšte prihvaćeno i programerski intuitivno rešenje određenih problema. Takođe, u kontekstu stabala parsiranja i apstraktnih sintaksnih stabala, opisanih u poglavlju 2.1, obrasci *posmatrač* i *posetilac* su od velikog značaja jer pružaju interfejs za obilazak takvih stabala. Stoga se, osim u implementaciji opisanoj u narednim poglavljima, koriste i od strane drugih alata korišćenih u radu kao što je ANTLR, opisan u poglavlju **??**. Stoga će u nastavku biti opisani samo obrasci posmatrač i posetilac, dok zainteresovani čitalac može pročitati više o ostalim obrascima u [9].

Obrazac „Posmatrač"

Obrazac za projektovanje *Posmatrač* (u daljem tekstu samo *posmatrač*) je strukturni obrazac za projektovanje koji se koristi kada je potrebno definisati jedan-ka-više vezu između objekata tako da ukoliko jedan objekat promeni stanje (subjekat) svi zavisni objekti su obavešteni o izmeni i shodno ažurirani. Posmatrač predstavlja *pogled* (engl. *View*) u MVC (engl. *Model-View-Controller*) arhitekturi. Na slici 2.11 se može videti UML dijagram [23] za obrazac posmatrač.



Slika 2.11: UML dijagram obrasca za projektovanje „Posmatrač".

Primer upotrebe ovog obrasca može biti aukcija gde je aukcionar subjekat i započinje aukciju, dok učesnici aukcije (objekti) posmatraju aukcionera i reaguju na podizanje cene. Prihvatanje promene cene menja trenutnu cenu i aukcioner oglašava promenu iste, a svi učesnici aukcije dobijaju informaciju da se izmena izvršila. Za potrebe ovog rada, primer upotrebe može biti obilazak stablolike kolekcije (recimo stabla parsiranja) i obaveštavanje o nailasku na čvorove određenih tipova. Te informacije se dalje mogu iskoristiti za izračunavanja nad pomenutom strukturom ili generisanje novih struktura (recimo AST).

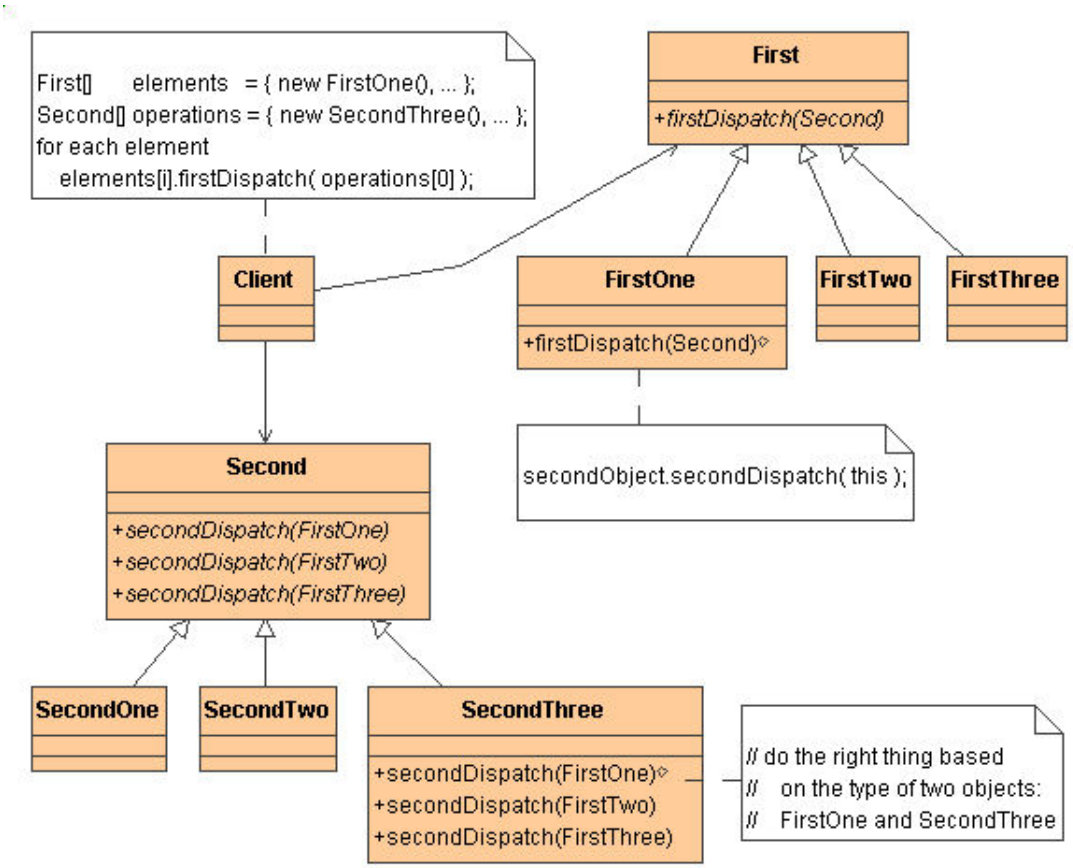
Obrazac „Posetilac"

Obrazac za projektovanje *Posetilac* (u daljem tekstu samo *posetilac*) je strukturni obrazac za projektovanje koji predstavlja operaciju koju je potrebno izvesti nad elementima objektne strukture. Posetilac omogućava definisanje nove operacije bez izmena klasa elemenata nad kojima operiše. Operacija koja će se izvesti zavisi od imena zahteva, tipa posetioca i tipa elementa kog posećuje. Na slici 2.12 se može videti UML dijagram [23] za obrazac posetilac.

Primer upotrebe ovog obrasca može biti operisanje taksi kompanija. Kada osoba pozove taksi kompaniju (prihvatanje posetioca), kompanija šalje vozilo osobi koja je pozvala kompaniju. Nakon ulaska u vozilo (posetilac), mušterija ne kontroliše svoj transport već je to u rukama taksiste (posetioca). Za potrebe ovog rada, primer upotrebe može biti prikupljanje informacija o kolekciji stablolike strukture (recimo stablo parsiranja) i korišćenje istih za neko izračunavanje ili generisanje novih struktura (recimo AST, za potrebe ovog rada).

2.4 Parsiranje gramatika programskih jezika

Pretpostavljajući da imamo gramatiku proizvoljnog programskog jezika, postavlja se pitanje: *Da li je moguće definisati postupak i zatim napraviti program koji će generisati kodove leksera i parsera napisane u određenom programskom jeziku za proizvoljnu gramatiku datu na ulazu?*. Odgovor je potvrdan i postoji veliki broj alata koji se mogu koristiti u ove svrhe, od kojih je navedeno par njih u odeljcima ispod.



Slika 2.12: UML dijagram obrasca za projektovanje „Posetilac".

GNU Bison

GNU Bison [11] je generator parsera i deo GNU projekta [12], često referisan samo kao *Bison*. Bison generiše parser na osnovu korisnički definisanih kontekstno slobodnih gramatika [4], upozoravajući pritom na dvosmislenosti prilikom parsiranja ili nemogućnost primena gramatičkih pravila. Generisani parser je najčešće C a ređe C++ kod, mada se u vreme pisanja ovog rada eksperimentiše sa Java podrškom. Generisani kodovi su u potpunosti prenosivi i ne zahtevaju specifične kompajlere. Bison može da, osim podrazumevanih *LALR(1)* [14] parsera, generiše i kanoničke *LR* [17], *IELR(1)* [6] i *GLR* [10] parsere.

Flex

Flex [8] je kreiran kao alternativa *lex*-u [15], Flex generiše samo leksere pa se stoga najčešće koristi u kombinaciji sa drugim alatima koji mogu da generišu

parsere, kao što je *BYACC*, opisan u nastavku.

BYACC

Berkeley YACC, skraćeno *BYACC* [2], pisan po ANSI C standardu i otvorenog koda, se smatra od strane mnogih kao *najbolja varijanta YACC-a* [15]. BYACC dozvoljava tzv. *reentrant* kod - memorija je deljenja između poziva pa je bezbedno konkurentno izvršavanje koda - na način kompatibilan sa Bison-om i to je delom razlog njegove popularnosti.

ANTLR

Another Tool for Language Recognition, ili kraće *ANTLR* [1], je generator *LL(*)* [**?**] leksera i parsera pisan u programskom jeziku Java sa intuitivnim interfejsom za obilazak stabla parsiranja. Verzija 3 podržava generisanje parsera u jezicima Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, i Standard ML, dok verzija 4 u vreme pisanja ovog rada samo generiše parsere u narednim jezicima: Java, C#, C++, JavaScript, Python, Swift i Go.

ANTLR verzije 4 je izabran u ovom radu zbog svoje jednostavnosti, intuitivnosti i podrške za mnoge moderne programske jezike. Verzija 4 je izabrana umesto verzije 3 po preporuci autora, na osnovu eksperimentalne analize brzine i pouzdanosti verzije 4 u odnosu na verziju 3. Lekseri i parseri za ulazne gramatike će u implementaciji biti generisani u programskom jeziku C#.

Parseri generisani koristeći ANTLR koriste novu tehnologiju koja se naziva *Prilagodljiv LL(*)* (engl. *Adaptive LL(*)*) ili *ALL(*)* [18], dizajniranu od strane Terensa Para, autora ANTLR-a, i Sema Harvela. *ALL(*)* vrši *dinamičku analizu* gramatike u fazi izvršavanja, dok su starije verzije radile analizu pre pokretanja parsera, tzv. *statičku analizu*. Ovaj pristup je takođe efikasniji zbog značajno manjeg prostora ulaznih sekvenci.

Najbolji aspekt ANTLR-a je lakoća definisanja gramatičkih pravila koji opisuju sintaksne konstrukte nalik na aritmetičkim izrazima u programskim jezicima. Primer jednostavnog pravila za definisanje aritmetičkog izraza je dat na slici 2.13. Pravilo `exp` je levo rekurzivno jer barem jedna od njegovih alternativnih definicija referiše na baš pravilo `exp`. ANTLR4 automatski zamenjuje levo rekurzivna pravila u nerekurzivne ekvivalente. Jedini zahtev koji mora biti ispunjen je da levo rekurzivna pravila moraju biti *direktna* - da pravila odmah referišu sama sebe.

Pravila ne smeju referisati drugo pravilo sa leve strane definicije takvo da se eventualno kroz rekurziju stigne nazad do pravila od kog se krenulo bez poklapanja sa nekim tokenom.

```
1      exp : (exp)
2          | exp '*' exp
3          | exp '+' exp
4          | INT
5          ;
```

Slika 2.13: Definicija uprošćenog aritmetičkog izraza u ANTLR4 gramatici.

Preduslovi za pokretanje ANTLR4

Kako bi ANTLR generisao parser u proizvoljnom programskom jeziku, potrebno je instalirati ANTLR i imati *Java Runtime Environment* (skr. *JRE*) instaliran na sistemu i dostupan globalno pokretanjem putem komande `java`. Instalacija se sastoji od preuzimanja najnovijeg *.jar* fajla [3], sa zvanične stranice [1] ili recimo korišćenjem *curl* alata:

```
1 $ curl -O http://www.antlr.org/download/antlr-4-complete.jar
```

Na UNIX sistemima moguće je kreirati alias `antlr4` ili *shell* skript unutar direktorijuma `/usr/local/bin` sa imenom `antlr4` koji će pokrenuti *.jar* fajl na sledeći način (pretpostavljajući da se *.jar* fajl nalazi u direktorijumu `/usr/local/lib`):

```
1 #!/bin/sh
2 java -cp "/usr/local/lib/antlr4-complete.jar:$CLASSPATH"
     org.antlr.v4.Tool $*
```

Na Windows sistemima moguće je kreirati *batch* skript sa imenom `antlr4.bat` koji će pokrenuti ANTLR4, na sledeći način (pretpostavljajući da se *.jar* fajl nalazi u direktorijumu `C:\lib`):

```
1 java -cp C:\lib\antlr-4-complete.jar;%CLASSPATH%
     org.antlr.v4.Tool %*
```

[3] Takođe je moguće kompajlirati izvorni kod dostupan na servisu GitHub `https://github.com/antlr/antlr4`

Ukoliko su aliasi ili skript fajlovi imenovani kao iznad, moguće je iz komandne linije pojednostavljeno pokretati ANTLR4:

```
1 $ antlr4
2 ANTLR Parser Generator Version 4.0
3 -o ___    specify output directory where all output is
     generated
4 -lib ___  specify location of .tokens files
5 ...
```

Dodatno, za Unix sisteme [4], moguće je kreirati dodatni alias `grun` (ili alternativno, kreirati `shell script`) za biblioteku `TestRig`. Biblioteka TestRig se može koristiti za brzo testiranje parsera - moguće je pokrenuti parser od bilo kog pravila i dobiti izlaz parsera u raznim formatima. TestRig dolazi uz ANTLR `.jar` fajl i moguće je napraviti prečicu za brzo pokretanje (nalik na ANTLR alias):

```
1 $ alias grun='java -cp
      "/usr/local/lib/antlr-4-complete.jar:$CLASSPATH"
      org.antlr.v4.gui.TestRig'
```

Generisanje parsera koristeći ANTLR4

Prvi korak u izradi aplikacije koja u sebi koristi parsiranje nekog jezika je definisanje gramatike jezika i kreiranje leksera i parsera za isti. U nastavku će biti opisan proces kreiranja interfejsa za parsiranje programa pisanih u pseudo-programskom jeziku (u nastavku *pseudo-jezik*), nalik na pseudokod. Ovako dobijeni interfejs će moći da se koristi u opšte svrhe, za potrebe ovog rada će se koristiti za generisanje AST stabla za program pisan u pseudo-jeziku.

Definišimo gramatiku pseudo-jezika prateći ANTLR pravila za definisanje gramatika. Kao i za svaki drugi programski jezik, treba obezbediti da postoje određeni koncepti koji se pojavljuju u programskim jezicima: *identifikatori*, *izrazi*, *naredbe*, *funkcije* i slično. Za sada se fokusirajmo na naredbe, kao samostalne izvršive jedinice koda. Stoga program možemo smatrati kao niz naredbi. U nekim slučajevima će biti potrebno definisanje kompleksnih naredbi koje se sastoje od više drugih naredbi, i ovakve složene naredbe ćemo zvati *blok* ili *blok naredbi*. Stoga, radi

[4] Za Windows operativni sistem je moguće kreirati *batch* skript po opisu na `https://github.com/antlr/antlr4/blob/master/doc/getting-started.md`.

konzistentnosti, program će biti blok naredbi. Kako bismo označili da su naredbe deo bloka naredbi, koristićemo reči `begin` i `end`, osim ukoliko je reč o samo jednoj naredbi. Ovakve situacije rešavamo definisanjem *alternativa* u definiciji pravila - više definicija razdvojenih simbolom `|`. Specijalne reči kao što su `begin` i `end` će biti rezervisane reči našeg pseudo-jezika, tzv. *ključne reči*. Na slici 2.14 se može videti definicija programa [5] i bloka naredbi pseudo-jezika, pri čemu se ključne reči u pravilima navode između apostrofa. ANTLR dozvoljava jednostavne definicije pravila u kojima figuriše promenljiv broj drugih pravila, pri čemu se koriste simboli kao u regularnim izrazima [6], što je iskorišćeno za definiciju pravila bloka naredbi. `NAME` predstavlja ime programa, što je zapravo identifikator. Identifikatore ćemo definisati kasnije, za sada možemo posmatrati identifikator kao nisku karaktera s tim što će postojati restrikcije vezane za to koji karakteri se mogu naći unutar identifikatora ali o tome će biti reči kasnije.

```
1       unit
2           : 'algorithm' NAME block EOF
3           ;
4
5       block
6           : 'begin' statement+ 'end'
7           | statement
8           ;
```

Slika 2.14: Definicija jedinice prevođenja i bloka naredbi za pseudo-jezik.

Sledeći korak je definisanje naredbe pseudo-jezika. Slično kao i u drugim programskim jezicima, potrebno je podržati koncept deklaracije promenljive, dodele vrednosti izraza promenljivoj, naredbe kontrole toka - grananje i petlje. Na slici 2.15 je definisano šta se sve smatra jednom naredbom. Naredbe mogu biti i prazne, što je označeno ključnom rečju `pass`. Iz definicije naredbe sa slike se jasno vidi šta sve može biti naredba (prateći redosled alternativa pravila):

- deklaracija

[5] Drugim rečima, jedan program u pseudo-jeziku je jedinica prevođenja, pa je zato pravilo nazvano *unit*.

[6] U regularnim izrazima, simbol `a?` označava opciono pojavljivanje simbola `a`, simbol `a+` označava jedno ili više pojavljivanja simbola `a`, a simbol `a*` označava proizvoljan broj pojavljivanja simbola `a` - kombinacija simbola `?` i `+`.

- dodela
- poziv funkcije (označen kao `cexp`, skraćeno od *function call expression*) [7]
- vraćanje vrednosti izraza (ključna vrednost `return`) iz funkcije
- prekidanje izvršavanja davanjem poruke o grešci
- naredba grananja
- *while* petlja
- *repeat-until* petlja
- inkrementiranje/dekrementiranje vrednosti promenljive

```
1     statement
2         : 'pass'
3         | declaration
4         | assignment
5         | cexp
6         | 'return' exp
7         | 'error' STRING
8         | 'if' exp 'then' block ('else' block)?
9         | 'while' exp 'do' block
10        | 'repeat' block 'until' exp
11        | ('increment' | 'decrement') var
12        ;
```

Slika 2.15: Definicija naredbe za pseudo-jezik.

Deklaracija, prikazana na slici 2.16, uvodi pojavljivanje simbola datog preko identifikatora `NAME` kao oznaku za promenljivu, funkciju ili proceduru - funkciju bez povratne vrednosti. Svaka promenljiva mora biti određenog tipa, što se postiže pravilom `type`. Promenljivoj se, opciono, može pridružiti početna vrednost, drugim rečima promenljiva se može *inicijalizovati* tako da joj se pridruži vrednost nekog izraza. Procedure i funkcije imaju opcione parametre, vrednosti izraza koje

[7] Funkcije mogu vratiti vrednosti pa se stoga njihovi pozivi mogu naći u izrazima - dakle poziv funkcije je validan izraz (stoga `expression` u imenu `function call expression`). Naravno, ta vrednost se može ignorisati ili pak sama funkcija može biti takva da nema povratnu vrednost već je samo neophodno izvršiti je zbog sporednih efekata.

im se prosleđuju kasnije u pozivu kao argumenti. Lista parametara, takođe prikazana na slici 2.16, se navodi kao lista proizvoljno mnogo parova `NAME : type`, što se vidi iz definicije pravila `parlist`.

```
1      declaration
2          : 'declare' type NAME ('=' exp)?
3          | 'procedure' NAME '(' parlist? ')' block
4          | 'function' NAME '(' parlist? ')' 'returning' type
               block
5          ;
6
7      parlist
8          : NAME ':' type (',' NAME ':' type)*
9          ;
```

Slika 2.16: Definicija deklaracije za pseudo-jezik.

Identifikatori su niske karaktera koje predstavljaju ime koje odgovara određenoj memorijskoj adresi. Identifikatori se koriste umesto sirovih vrednosti adresa kako bi kod bio čitljiviji i lakši za pisanje - na nivou asemblera se većinom koriste adrese ili automatski generisane oznake. Na slici 2.17 se može videti definicija identifikatora. Identifikator se sastoji od slova, cifara i simbola _, s tim što ne sme početi cifrom. Ovo je konvencija koju prati dosta jezika, uključujući programski jezik C. Primetimo da je identifikator nešto što bi lekser trebalo da prepozna tokom tokenizacije. Međutim, kada definišemo gramatiku od koje će ANTLR praviti lekser i parser, možemo i tokene definisati preko gramatičih pravila dajući regularni izraz za njihovo poklapanje. Listovi stabla parsiranja su uvek tokeni, drugim rečima se nazivaju i *terminalni simboli*. Tokeni se, naravno, mogu naći bilo gde u stablu parsiranja.

```
1      NAME
2          : [a-zA-Z_][a-zA-Z_0-9]*
3          ;
```

Slika 2.17: Definicija identifikatora za pseudo-jezik.

Pošto želimo da pseudo-jezik bude strogo tipiziran, potreban je koncept tipa (što smo videli u deklaracijama), čija je definicija data na slici 2.18. Tip može biti

primitivan (drugim rečima *prost*) ili *složen*. Primitivni tipovi su podržani u samoj sintaksi jezika - u našem slučaju brojevi i niske. Brojevi mogu biti celi ili realni. U složene tipove spadaju korisnički definisani tipovi (sa imenom `NAME`, u četvrtoj alternativi pravila `typename` sa slike 2.18) i kolekcije. Od kolekcija su podržani nizovi, liste i skupovi. Prilikom definicije kolekcije mora se navesti tip elemenata kolekcije i taj tip mora biti uniforman - isti za sve elemente kolekcije.

```
1       type
2           : typename 'array'?
3           | typename 'list'?
4           | typename 'set'?
5           ;
6
7       typename
8           : 'integer'
9           | 'real'
10          | 'string'
11          | NAME
12          ;
```

Slika 2.18: Definicija tipa podataka za pseudo-jezik.

Izrazi, iako se definišu rekurzivno, se mogu posmatrati kao kombinacija promenljivih, operatora i poziva funkcija sa odlikom da se mogu *evaluirati*, tj. moguće je izračunati njegovu vrednost. Iz definicije pravila `exp` na slici 2.19, mogu se uočiti tipovi izraza, pri čemu nije vođeno računa o matematičkom prioritetu operatora, radi jednostavnosti. Izraz može biti *literal*, koji predstavlja konstantu, bilo brojevnu, logičku ili nisku karaktera. Promenljive, definisane pravilom `var` su takođe izrazi, jer se trenutna vrednost promenljive posmatra kao vrednosti izraza. Primetimo da promenljiva može biti kolekcijskog tipa, u kom slučaju se navodi redni broj elementa nakon identifikatora promenljive - taj redni broj može biti rezultat evaluacije drugog izraza, ali ne bilo kakvog, stoga se u pravilu `iexp` definiše šta sve može biti korišćeno da se indeksira element kolekcije. Izrazima se može dati prioritet pomoću zagrada, što se vidi u trećoj alternativi pravila `exp`. U naredne tri alternative su opisani tipovi izraza: aritmetički, relacioni i logički. Aritmetički izrazi su vezani aritmetičkim operatorima definisanim preko pravila `aop`, slično važi i za ostala dva tipa. Svi tipovi izraza navedeni iznad su binarni, što znači da

operatori zahtevaju dva argumenta. Postoje i unarni izrazi, od kojih su podržane promena znaka i logička negacija, što se vidi iz pravila `uop`.

```
1       exp
2           : literal
3           | var
4           | '(' exp ')'
5           | exp aop exp
6           | exp rop exp
7           | exp lop exp
8           | uop exp
9           | cexp
10          ;
11
12      var
13          : NAME ('[' iexp ']')?
14          ;
15
16      iexp
17          : literal
18          | var
19          | aexp
20          ;
21
22      cexp
23          : 'call' NAME '(' explist? ')'
24          ;
25
26      explist
27          : exp (',' exp)*
28          ;
29
30      aop : '+' | '-' | '*' | '/' | 'div' | 'mod' ;
31      rop : '>' | '>=' | '<' | '<=' | '==' | '=/=' ;
32      lop : 'and' | 'or' ;
33      uop : '-' | 'not' ;
```

Slika 2.19: Definicija izraza za pseudo-jezik.

Definicija literala je prikazana na slici 2.20. Literali mogu bili istinitosne konstante `True` i `False`, brojevne konstante ili niske karaktera. Brojevne konstante mogu bili celobrojni ili realni dekadni brojevi. Realne konstatne je moguće definisati u fiksnom ili pokretnom zarezu. Niske se mogu definisati između navodnika

ili apostrofa. Pritom, kao i u modernim programskim jezicima, moguće je navesti sekvence koje predstavljaju specijalne karaktere kao što su novi red, tabulator itd. Oznaka `fragment` označava optimizaciju, naime nije potrebno da postoji na primer pravilo `Digit`, već samo dajemo simbol za regularni izraz koji će se koristiti u više drugih pravila i poklapati jednu dekadnu cifru.

```
1      literal : 'True' | 'False' | INT | FLOAT | STRING ;
2
3      STRING
4          : '"' ( EscapeSequence | ~('\\'|'"') )* '"'
5          ;
6
7      fragment
8      EscapeSequence
9          : '\\' [abfnrtvz"'\\]
10         | '\\' '\r'? '\n'
11         ;
12
13     INT
14         : Digit+
15         ;
16
17     FLOAT
18         : Digit+ '.' Digit* ExponentPart?
19         | '.' Digit+ ExponentPart?
20         | Digit+ ExponentPart
21         ;
22
23     fragment
24     ExponentPart
25         : [eE] [+-]? Digit+
26         ;
27
28     fragment
29     Digit
30         : [0-9]
31         ;
```

Slika 2.20: Definicija konstanti za pseudo-jezik.

Poslednje što treba definisati je sve ono što lekser treba da preskoči tokom prolaska kroz izvorni kod programa. To su beline (nevidljivi karakteri kao što su

razmaci, tabulatori i novi redovi) i komentari. Definicije ovih pravila se mogu videti na slici 2.21. Vidimo da se u njima koristi posebna oznaka `-> skip`, koja predstavlja instrukcije lekseru da preskoči sve ono što ovo pravilo poklopi. Komentari su u stilu kao u programskom jeziku C (ali naravno, isti stil se koristi i u mnogim jezicima) i mogu biti jednolinijski ili višelinijski. Beline koje treba preskočiti su definisane u pravilu `WS`, skraćeno od *whitespace*, što u prevodu sa engleskog znači *beli prostor, belina.*

```
1       BlockComment
2           :   '/*' .*? '*/'  -> skip
3           ;
4       LineComment
5           :   '//' ~[\r\n]*  -> skip
6           ;
7       WS
8           : [ \t\u000C\r\n]+ -> skip
9           ;
```

Slika 2.21: Definicija komentara i belina za pseudo-jezik.

```
1       grammar Pseudo;
```

Slika 2.22: Definicija imena gramatike za pseudo-jezik.

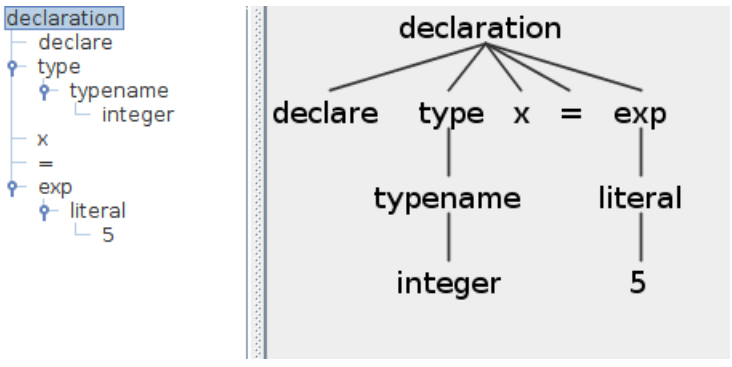
Ovako definisanu gramatiku možemo sačuvati u fajl sa imenom `Pseudo.g4`, potrebno je samo navesti ime gramatike na početku fajla, kao na slici 2.22. Naredni korak je kreiranje leksera i parsera koristeći ANTLR4, predpostavljajući da je instaliran na način opisan u **??**. Pokretanjem ANTLR-a generišemo lekser i parser za gramatiku pseudo-jezika:

```
1 $ antlr4 Pseudo.g4
```

ANTLR4 će generisati lekser i parser podrazumevano napisane u programskom jeziku Java. Ukoliko želimo to da promenimo, možemo koristiti opciju `-Dlanguage=...`. Kako bismo testirali generisani lekser i parser, možemo koristiti ANTLR TestRig da vizualno prikažemo stablo parsiranja, s tim što moramo

prvo kompajlirati generisane Java kodove. TestRig pozivamo navođenjem ime gramatike (koje se poklapa sa imenom leksera i parsera) i imenom pravila od koga će parser krenuti. Opcija `-gui` pokreće vizualni prikaz stabla parsiranja pokazan na slici 2.23 (vizualni prikaz je moguće preskočiti i samo ispisati stablo u LISP formi koristeći opciju `-tree`), mada je moguće i ispisati samo tokene koristeći opciju `-tokens`. Ulaz se prosleđuje programu dok se ne naiđe na simbol `EOF`, ili alternativno se može preneti ulaz putem UNIX pipeline-a (na slici 2.23 se može videti izlaz koji se dobija korišćenjem opcije `-gui`):

```
1  $ javac *.java
2  $ echo "declare integer x = 5" | grun Pseudo declaration
       -tokens
3  [@0,0:6='declare',<'declare'>,1:0]
4  [@1,8:14='integer',<'integer'>,1:8]
5  [@2,16:16='x',<NAME>,1:16]
6  [@3,18:18='=',<'='>,1:18]
7  [@4,20:20='5',<INT>,1:20]
8  [@5,22:21='<EOF>',<EOF>,2:0]
9  $ echo "declare integer x = 5" | grun Pseudo declaration
       -tree
10 (declaration declare (type (typename integer)) x = (exp
       (literal 5)))
11 $ echo "declare integer x = 5" | grun Pseudo declaration -gui
```



Slika 2.23: Grafički prikaz stabla parsiranja koje generiše parser kreiran od strane alata ANTLR4 TestRig za kod pisan u pseudo-jeziku.

Obilazak stabla parsiranja

ANTLR, osim leksera i parsera za datu gramatiku, može da kreira interfejse i bazne klase koji prate obrasce za projektovanje *posetilac* (engl. *visitor*) i osluškivač (engl. *listener*, varijanta obrasca *posmatrač* engl. *observer*) opisane u 2.3. Tako kreirani interfejsi i klase imaju metodi za obilazak stabla parsiranja. ANTLR podrazumevano generiše interfejs osluškivača (slika 2.24) kao i baznu klasu koja implementira generisani interfejs tako što su sve implementirane metodi prazne. Stoga, ukoliko korisnik želi da definiše operaciju samo u slučaju da se prilikom obilaska stabla parsiranja naiđe na samo jedan tip čvora, nije potrebno implementirati ceo interfejs osluškivača, već je moguće naslediti baznu klasu i predefinisati samo jedan metod. ANTLR može da generiše i posetilac (slika 2.25), ukoliko se navede odgovarajuća opcija `-visitor` prilikom pokretanja. Slično, ukoliko nije potrebno generisati osluškivač, može se koristiti opcija `-no-listener` kako se ne bi generisao osluškivač.

```
1  public interface IPseudoListener : IParseTreeListener
2  {
3      void EnterUnit([NotNull] PseudoParser.UnitContext
           context);
4      void ExitUnit([NotNull] PseudoParser.UnitContext
           context);
5      void EnterBlock([NotNull] PseudoParser.BlockContext
           context);
6      void ExitBlock([NotNull] PseudoParser.BlockContext
           context);
7      void EnterStatement([NotNull]
           PseudoParser.StatementContext context);
8      void ExitStatement([NotNull]
           PseudoParser.StatementContext context);
9
10     ...
11 }
```

Slika 2.24: Delimični prikaz interfejsa osluškivača generisanog od strane ANTLR4 za pseudo-jezik definisan u prethodnom odeljku (C#).

Sa slike 2.24 se vidi da je moguće definisati metodi koje će se pozivati prilikom ulaska ali i prilikom izlaska iz čvora određenog tipa prilikom obilaska stabla parsiranja. Pritom, važno je kako se stablo obilazi. U slučaju ANTLR, to je pretraga

u dubinu (engl. *depth-first search, DFS*) [8], stoga će se metod `Exit` za proizvoljni čvor pozvati tek kad se obiđu sva deca tog čvora - dakle nakon poziva njihovih `Enter` i `Exit` metoda. Pošto se DFS obično implementira putem LIFO [9] strukture, može se reći da se `Enter` metod poziva onog trenutka kad se čvor ubaci u strukturu, a `Exit` metod onda kada se čvor ukloni iz strukture.

```
1 public interface IPseudoVisitor<Result> :
     IParseTreeVisitor<Result>
2 {
3      Result VisitUnit([NotNull] PseudoParser.UnitContext
          context);
4      Result VisitBlock([NotNull] PseudoParser.BlockContext
          context);
5      Result VisitStatement([NotNull]
          PseudoParser.StatementContext context);
6      Result VisitDeclaration([NotNull]
          PseudoParser.DeclarationContext context);
7
8      ...
9 }
```

Slika 2.25: Delimični prikaz interfejsa posetioca generisanog od strane ANTLR4 za pseudo-jezik definisan u prethodnom odeljku (C#).

Za razliku od osluškivača, posetilac je prirodnije koristiti ukoliko je potrebno izvršiti neko izračunavanje nad strukturom koja se obilazi. Interfejs posetioca (slika 2.25) je šablonski, i metodi imaju povratnu vrednost šablonskog tipa za razliku od metoda osluškivača i, u odnosu na osluškivač, nema para metoda za svaki čvor već samo jedan metod. Dodatna razlika, ali i najveća, je ta što se metodi posetioca ne pozivaju automatski. Stoga je na programeru da nastavi obilazak i da odluči u koje čvorove želi da se spusti. Jasno je da i osluškivač i posetilac imaju svoje primene - ukoliko je potrebno obići stablo parsiranja i dovući neke informacije može se iskoristiti osluškivač jer onda ne moramo brinuti o obilasku. S

[8] DFS je obilazak stabla takav da se obilazak duž grane stabla nastavlja sve dok je moguće ići dublje, a ako to nije moguće vratiti se unazad i obići druge grane.

[9] *Last In, First Out* struktura podataka je apstraktna struktura podataka sa operacijama ubacivanja i izbacivanja elemenata, pri čemu je element koji se izbacuje onaj koji je poslednji ubačen. Primer LIFO strukture je držač za CD-ove - ne mogu se ukloniti CD-ovi ispod CD-a na vrhu (poslednji ubačen) a da se ne ukloni isti. U slučaju opisanom iznad, implementacija LIFO strukture se naziva stek *stack*.

druge strane, ukoliko je potrebno izračunati neku vrednost prirodno je iskoristiti rekurziju i iskoristiti posetilac - rekurzivni pozivi prilikom obilaska nam idu u prilog jer koristimo povratne vrednosti tih metoda da gradimo rezultat od listova ka korenu stabla parsiranja. U nastavku će se koristiti posetilac zbog kontrole obilaska ali i činjenice da se stablo parsiranja obilazi sa ciljem da se izgradi AST, koji je takođe rekurzivna struktura i gradi se inkrementalno kroz rekurziju.

Bilo da se koristi osluškivač ili posetilac, potrebno je nekako proslediti informacije o samom čvoru na koji se naišlo tokom obilaska stabla parsiranja. Te informacije se metodima osluškivača i posetioca prosleđuju putem potklasa apstrakne klase konteksta pravila `ParserRuleContext` - u primeru iznad `UnitContext`, `BlockContext` itd. Svaki kontekst pravila po imenu odgovara pravilima definisanim u gramatici i sadrži informacije bitne za trenutni čvor u stablu parsiranja koji odgovara tipu konteksta. Takođe, u svakom kontekstu su prisutne i metodi čija imena odgovaraju pravilima koja se javljaju u definiciji samog pravila koje odgovara kontekstu. Tako da, za `BlockContext`, imajući u vidu definiciju sa slike 2.14, pošto se u definiciji osim tokena koristi i pravilo `statement`, u okviru `BlockContext` klase biće implementiran i metod `statement()` koji vraća kontekst pravila u ovom slučaju tipa `StatementContext[]` jer u prvoj alternativi stoji `statement+` - dakle možemo imati više `statement` poklapanja. Sa ovim u vidu, moguće je odrediti kako će se obilazak nastaviti (u slučaju posetioca) ili dovući informacije o delovima definicije pravila. Ukoliko pravilo ima više alternativa, metodi koje vraćaju kontekst pravila koje figuriše u alternativi koja nije korišćena za poklapanje pravila će vratiti `null`. Pošto se `statement()` pravilo javlja u obe alternative pravila `block` (i nije opciono), možemo biti sigurni da povratna vrednost `statement()` metoda nikada neće biti `null`.

U poglavlju 3 će se koristiti posetilac za obilazak stabla parsiranja i kreiranje AST apstrakcije od istog. Pritom, koristiće se implementacija posetioca u programskom jeziku C#. Pre toga, potrebno je objasniti pojam *simboličke promenljive* s obzirom da će se ti koncepti koristiti u samoj analizi AST-ova.

2.5 Programske paradigme i gramatičke razlike programskih jezika

Iako se u suštini svode na mašinski jezik ili asembler, viši programski jezici mogu imati velike razlike međusobno - kako u načinu pisanja koda, tako i u efika-

snosti izvršavanja. Način, ili stil programiranja se naziva *programska paradigma* [19]. Može se pokazati da sve što je rešivo putem jedne, može i da se reši i putem ostalih; međutim neki problemi se prirodnije rešavaju koristeći specifične paradigme. Neke poznatije programske paradigme su navedene u nastavku zajedno sa njihovim odlikama i primerima upotrebe.

Imperativna paradigma

Imperativna paradigma pretpostavlja da se promene u trenutnom stanju izvršavanja mogu sačuvati kroz promenljive. Izračunvanja se vrše kroz niz koraka, u svakom koraku se te promenljive referišu ili se menjaju njihove trenutne vrednosti. Raspored koraka je bitan, jer svaki korak može imati različite posledice s obzirom na trenutne vrednosti promenljivih na početku tog koraka. Primer koda pisanog po imperativnoj paradigmi se može videti na slici 2.26.

```
1      result = []
2      i = 0
3  start:
4      numPeople = length(people)
5      if i >= numPeople goto finished
6      p = people[i]
7      nameLength = length(p.name)
8      if nameLength <= 5 goto nextOne
9      upperName = toUpper(p.name)
10     addToList(result, upperName)
11 nextOne:
12     i = i + 1
13     goto start
14 finished:
15     return sort(result)
```

Slika 2.26: Primer koda pisanog po imperativnoj paradigmi.

Imperativni programski jezici najčešće prate ovu paradigmu više nego bilo koju drugu iz par razloga. Prvi je taj što imperativna paradigma najbliže oslikava samu mašinu na kojoj se program izvršava, pa je programer mnogo „bliži" istoj. Posledica ovog pristupa, a to je i drugi razlog za popularnost imperativne paradigme, je omogućila da je ova paradigma bila najefikasnija zbog ograničenja u hardveru.

Danas, zbog mnogo bržeg razvoja i mnogo jačih računara, efikasnost se sve manje i manje uzima u obzir.

Naravno, imperativna paradigma ima i svoje nedostatke. Naime, najveći problem je razumevanje i verifikovanje semantike programa zbog postojanja sporednih efekata [10]. Stoga je i pronalaženje grešaka u kodovima koji prate imperativnu paradigmu znatno komplikovanije. Apstrakcija je takođe više ograničena u imperativnoj nego u ostalim paradigmama. Na kraju, redosled izvršavanja je vrlo bitan, što neke probleme čini težim ukoliko se pokušaju rešiti pomoću imperativne paradigme.

Strukturna paradigma

Strukturna paradigma je vrsta imperativne paradigme gde se kontrola toka vrši putem ugnježdenih petlji, uslovnih grananja i podrutina. Promenljive su obično lokalne za blok u kome su definisane, što određuje i njihov životni vek i vidljivost. Primer koda pisanog po imperativnoj paradigmi se može videti na slici 2.27. Najpopularniji derivat strukturne paradigme je *proceduralna paradigma*, bazirana na konceptu poziva *procedure* - podrutine ili funkcije koja sadrži seriju koraka koje je potrebno izvršiti redom.

```
1 result = [];
2 for i = 0; i < length(people); i++ {
3     p = people[i];
4     if length(p.name)) > 5 {
5         addToList(result, toUpper(p.name));
6     }
7 }
8 return sort(result);
```

Slika 2.27: Primer koda pisanog po strukturnoj paradigmi.

[10] Sporedni efekti (promena stanja mašine) ne poštuju *referencijalnu transparentnost* koja se definiše na sledeći način: *Ako važi* $P(x)$ *i* $x = y$ *u nekom trenutku, onda* $P(x) = P(y)$ *važi tokom čitavog vremena izvršavanja programa.*

Logička paradigma

Logička paradigma koristi deklarativni pristup rešavanju problema. Umesto zadavanja instrukcije koje treba da dovedu do rezultata, opisuje se sam rezultat kroz činjenice - skup logičkih pretpostavki. Taj opis se zatim prevodi u upit koji se dalje koristi. Uloga računara je održvanje i logička dedukcija. Logička paradigma se deli u tri sekcije:

- niz deklaracija i definicija koje opisuju problem iz nekog domena,
- relevantne činjenice i
- relevantni ciljevi u formi upita.

Bilo koji rezultat dedukcije rešenja upita predstavlja rezultat izvršavanja. Deklaracije i definicije se konstruišu iz relacija, npr. $X \in Y$ ili $X \in [a, b]$. Prednost ovog pristupa je mala količina programiranja, pošto dedukcioni sistem traži rešenje problema. Takođe, verifikovanje validnosti je stoga trivijalno. Primer koda pisanog po logičkoj paradigmi se može videti na slici 2.28.

```
1  domains
2      being = symbol
3  predicates
4      animal(being)         % sve životinje su ziva bica
5      dog(being)            % svi psi su ziva bica
6      die(being)            % svi ziva bica umiru
7  clauses
8      animal(X) :- dog(X)   % svi psi su zivotinje
9      dog(fido).            % fido je pas
10     die(X) :- animal(X)   % sve životinje umiru
```

Slika 2.28: Primer koda pisanog po logičkoj paradigmi.

Funkcionalna paradigma

Funkcionalna paradigma posmatra sve potprograme kao funkcije u matematičkom smislu - uzimaju argumente i vraćaju jedinstven rezultat. Povratna vrednost zavisi isključivo od argumenata, što znači da je nebitan trenutak u kom je funkcija pozvana. Izračunvanja se vrše primenom aplikacije funkcija, kompozicijom

funkcija i redukcijom. Primer koda pisanog po funkcionalnoj paradigmi se može videti na slici 2.29.

```
1 people
2     |> map    (extract_name . to_upper)
3     |> filter (\name -> length name > 5)
4     |> sort
5     |> take 5
6     |> join ", "
```

Slika 2.29: Primer koda pisanog po funkcionalnoj paradigmi.

Funkcionalni programski jezici se baziraju na funkcionalnoj paradigmi. Takvi jezici dozvoljavaju tretiranje funkcija kao *građana prvog reda* - mogu biti tretirane kao podaci pa se mogu proslediti drugim funkcijama ili vratiti kao rezultat izračunavanja drugih funkcija. Prednosti funkcionalnih jezika su visok nivo apstrakcije, što prevazilazi mnogo detalja programiranja i stoga eliminiše pojavu velikog broja grešaka, nezavisnost od redosleda izračunavanja, što omogućava paralelizam, i formalnu matematičku verifikaciju. Mane su potencijalna manja efikasnost, što danas predstavlja manji problem, kao i teškoća implementacije specifične sekvencijalne aktivnosti ili potreba za stanjem, što bi se lako implementiralo imperativno ili preko OO paradigme.

Objektno-orijentisana paradigma

Objektno-orijentisana paradigma (kraće *OOP*) je paradigma u kojoj se objekti stvarnog sveta posmatraju kao zasebni entiteti koji imaju sopstveno stanje koje se modifikuje samo pomoću procedura ugrađenih u same objekte - tzv. *metode*. Posledica zasebnog operisanja objekata omogućava njihovu enkapsulaciju u module koji sadrže lokalnu sredinu i metode. Komunikacija sa objektom se vrši prosleđivanjem poruka. Objekti su organizovani u klase, od kojih nasleđuju metode i ekvivalentne promenljive. OOP omogućava ponovnu iskorišćenost koda i ekstenzibilnost koda. Primer koda po strukturnoj paradigmi je dat na slici 2.30.

Nova klasa (A) može *naslediti* ili *konkretizovati* drugu klasu (B). B se zove *bazna klasa* ili *natklasa*, dok se A naziva *izvedena klasa* ili *potklasa*. Izvedenna klasa nasleđuje sve odlike bazne klase - strukturu i ponašanje - postaje specijalizacija bazne klase dok bazna klasa postaje generalizacija svoje potklase. Osim nasleđenih

```
1  abstract class Employee
2  {
3      private String name;
4
5      Employee(String name) {
6          this.name = name;
7      }
8
9      abstract void work();
10 };
11
12 class WageEmployee extends Employee
13 {
14     private double wage;
15     private double hours;
16
17     WageEmployee(String name) {
18         super(name);
19         this.name = name;
20     }
21
22     void work() {
23         wage += 200;
24         hours += 8;
25     }
26 };
27
28 var bill = new WageEmployee("Bill Gates");
29 bill.work();
```

Slika 2.30: Primer koda pisanog po OO paradigmi u programskom jeziku Java.

osobina, potklasa može imati dodatna stanja (instancne promenljive) ili dodatna ponašanja (metode). Dozvoljeno je i predefinisanje ponašanja bazne klase. Mehanizam nasleđivanja je dozvoljen i ukoliko nije dozvoljen pristup izvornom kodu bazne klase.

Idealno, stanju objekta može da pristupi i modifikuje samo pomoću metoda tog objekta. Većina OO jezika dozvoljava direktnu manipulaciju stanja ali taj pristup nije preporučen. Kako bi se enkapsulacija i skrivanje informacija kao najveće prednosti OO paradigme mogle iskoristiti, interfejs klase (kako se pristupa objektima) bi trebalo da bude odvojen od implementacije klase (izvornog koda

metoda klasa).

Skript paradigma i njen odnos sa proceduralnom paradigmom

Čak i unutar jedne paradigme kao što je proceduralna paradigma, mogu se naći veoma velike varijacije u izgledu koda pisanog u različitim programskim jezicima koji prate proceduralnu paradigmu. Kako hardver postaje moćniji, više se ceni vreme koje programer provede u procesu pisanja koda nego koliko je taj kod efikasan. Štaviše, u nekim slučajevima je dobitak u efikasnosti veoma mali u poređenju sa vremenom koje je potrebno utrošiti da bi se ta efikasnost postigla. Ukoliko se program pokreće veoma retko, možda nije ni bitno da li se on izvršava sekundu sporije od efikasnog programa, ako je za njegovo pisanje utrošeno znatno manje vremena. Ovo je pristup koji prate *skript* jezici kao što su `Python, Perl, bash` itd. Iako proceduralni, oni se razlikuju od klasičnih predstavnika proceduralne paradigme i njihove razlike su vremenom postale tolike da nije neuobičajeno da se skript jezici svrstaju u zasebnu, *skript paradigmu*. Stoga će se u nastavku, pod terminom *proceduralni jezik* smatrati tradicionalni proceduralni jezik, ukoliko nije naznačeno drugačije. Na slici 2.31 se mogu uočiti navedene razlike.

Promenljive predstavljaju jedan od osnovnih koncepata na kojem se zasnivaju i proceduralni i skript jezici. Promenljivu odlikuje, između ostalog, i njen *tip* koji određuje količinu memorije potrebnu za njeno skladištenje. Proceduralni programski jezici zahtevaju definisanje tipa promenljive i obično su i *statički*, što znači da promenljive ne mogu menjati svoj tip tokom izvršavanja programa. Proces uvođenja imena promenljive se u naziva *deklaracija promenljive*. Slično kao i za promenljive, potrebno je deklarisati i funkcije pre trenutka njihovog korišćenja kako bi prevodilac znao broj i tipove parametara funkcije kao i njihove povratne vrednosti. Skript jezici žrtvuju strogu tipiziranost kako bi proces pisanja koda bio brži. Stoga su oni obično *dinamički* - promenljive mogu menjati tip tokom izvršavanja programa. Pošto promenljive mogu menjati svoj tip, definisanje tipa prilikom uvođenja imena promenljive postaje redundantno jer prevodilac može to sam da zaključi. Stoga i sam proces uvođenja imena promenljive postaje redundantan. Slično, parametri funkcija takođe nisu fiksnog tipa. Slično važi i za povratnu vrednost funkcije.

Kod proceduralnih jezika, pošto su obično strogo tipizirani, mogu se iskoristi-

```c
int main() {
    int k = 0;
    for (int i = 0; i < 1000000; i++)
        k++;
    return 0;
}
```

```
$ time: 0.03s user 0.00s system 70% cpu 0.044 total
```

```python
k = 0
for i in range(1000000):
    k += 1
```

```
$ time: 0.16s user 0.03s system 93% cpu 0.200 total
```

Slika 2.31: Primer koda pisanog po tradicionalnoj proceduralnoj paradigmi (gore, `C`) i po modernoj skript paradigmi (gore, `Python 3`) kao i odgovarajuća vremena izvršavanja dobijena komandom `time`.

ti strukture podataka koje omogućavaju brz pristup svojim elementiram. To su obično nizovi koji predstavljaju kontinualni blok memorije u kom su elementi niza smešteni jedan do drugog. Pristup se vrši na osnovu indeksa i, pošto su svi elementi istog tipa (zauzimaju jednaku količinu memorije), može se u konstantnom vremenu izračunati memorijska lokacija na kojoj se nalazi element niza sa datim indeksom. Kompleksnije strukture podataka obično nisu podržane u samom jeziku. Neki proceduralni jezici dozvoljavaju veoma niski pristup kroz *pokazivače* ili *reference* na memorijske adrese (`C` i `C++`). Većina modernih proceduralnih jezika ne dozvoljava rad sa pokazivačima, ne brinući puno o efikasnosti, dok neki dozvoljavaju korišćenje pokazivača u specijalnim situacijama sa eksplicitnom naznakom (`C#`).

Pored dinamičnosti kad je u pitanju tip promenljivih, skript jezici često imaju neke specifične strukture podataka ugrađene u sam jezik kao olakšice prilikom programiranja. Primarna struktura podataka je *jednostruko ulančana lista* [11], za

[11] Lista je rekurzivna kolekcija podataka koja se sastoji od glave koja sadrži vrednost određenog tipa, i pokazivača na rep - drugu listu. Specijalno, praznim pokazivačima se označava kraj liste

razliku od niza kod proceduralnih jezika. Razlog zašto se koriste liste je delimično zbog toga što, kao i ostale promenljive, liste nisu strogo tipizirane. Moguće je u listu ubacivati elemente različitih tipova - što onemogućava skladištenje u kontinualnom bloku memorije (osim ukoliko je lista imutabilna, što nije obično slučaj). Skript jezici uglavnom omogućavaju indeksni pristup elementima liste pa programeru izgleda kao da radi nad običnim nizom. Neki skript jezici omogućavaju kreiranje *asocijativnih nizova*, gde indeks niza ne mora biti ceo broj već može uzimati vrednost iz domena bilo kog tipa. Osim listi, obično su podržane i torke, i za njih važe iste slobode kao i za liste. Kompleksnije strukture podataka uključuju skupove i rečnike (drugačije nazivane i *heš mape*, engl. *hash map*) koji su kolekcija ključ-vrednost parova gde je dozvoljen indeksni pristup po vrednosti ključa. Skript programski jezici su skoro uvek interpretirani, iako se neki jezici mogu kompajlirati po potrebi za efikasnije ponovno izvršavanje. S obzirom da efikasnost nije u glavnom planu, u skript jezicima nije dozvoljen direktan pristup memoriji putem pokazivača ili referenci.

(prazna lista).

Glava 3

Opis opštih AST-ova za imperativne jezike

Kao što je opisano u odeljku 2.5, dosta različitih „pod-paradigmi" potiče iz imperativne paradigme. Strukturna, proceduralna i skript paradigma, iako naizgled različite, poseduju veliki broj sličnih osobina i koncepata. Svaki programski jezik ima svoju gramatiku i na osnovu toga ima svoja gramatička pravila koja se oslikavaju u apstraktnim sintaksnim stablima tih jezika. Na slikama 3.1 i 3.2 se mogu videti razlike u strukturi AST-a za jezike `Lua` i `Go`, kao primeri skript odnosno proceduralne paradigme.

U ovoj poglavlju će biti predstavljena AST apstrakcija dizajnirana tako da je uz pomoć nje moguće predstaviti kodove proizvoljnog imperativnog programskog jezika. To uključuje i skript jezike koji, kako će biti pokazano u ovom radu, mogu da se posmatraju na istom nivou kao i svoji proceduralni „rođaci".

Kako bi se kreirala smislena apstrakcija stabla parsiranja, potrebno je identifikovati bitne informacije u stablu parsiranja ali i koncepte same gramatike koji su od značaja. Najjednostavnije rešenje je mimikovati čvorove stabla parsiranja, ukoliko su gramatička pravila kreirana tako da oslikaju koncepte jezika koji gramatika definiše. Na primer, ukoliko u gramatici imamo pravilo `deklaracija` sa alternativama `deklaracijaPromenljive` i `deklaracijaFunkcije`, možemo kreirati apstraktni koncept `Deklaracija` sa konkretizacijama `DeklaracijaPromenljive` i `DeklaracijaFunkcije`. Kako se definišu deklaracije promenljivih i funkcija zavisi dalje od definicija pravila `deklaracijaPromenljive` i `deklaracijaFunkcije`. Naravno, nije uvek moguće primeniti ovakav postupak. Takođe, nekada u gramatici definišemo pomoćna pravila kako bismo se izborili sa rekurzijom ili izbegli

Slika 3.1: Prikaz AST-a isečka koda pisanog u programskom jeziku Lua. Prikazano putem `https://astexplorer.net/`

neke tipove rekurzije - ta pravila ne bi trebalo da imaju odgovarajuće tipove u apstrakciji.

Pošto su u pitanju gramatike programskih jezika, onda je jasno da dosta različitih gramatika dele slične koncepte i da je moguće definisati tipove čvorova koji odgovaraju tim konceptima. Neki od njih mogu biti: naredba, izraz, deklaracija, poziv funkcije, dodela... Jasno, postoji i hijerarhija između navedenih koncepata - poziv funkcije se može smatrati kao samostalna naredba ali može biti i deo izraza. Dakle, treba biti jako pažljiv u definisanju hijerarhije tako da ne dozvoli nešto što u opštem slučaju ne bi trebalo da bude dozvoljeno (npr. ako je dozvoljeno višestruko nasleđivanje i poziv funkcije je i naredba ali i izraz, onda se izrazi u kojima figurišu pozivi funkcija sastoje od više naredbi, što nema smisla.).

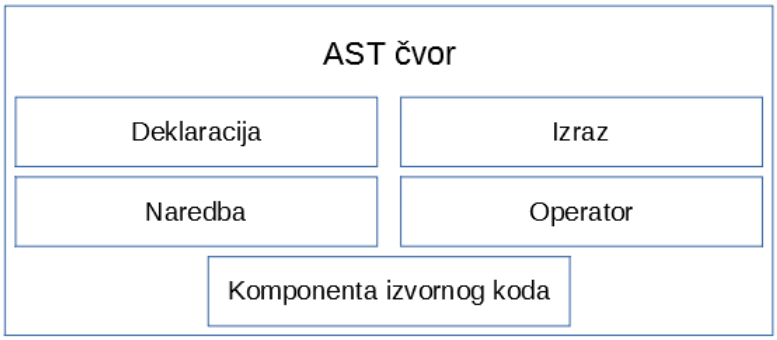
```
package main

import "fmt"

const TIPS = `abc`;

func PrintTips() {
    fmt.Println(TIPS)
}
```

```
- File  {
    Comments: [ ]
  - Decls:  [
    + GenDecl {Loc, Lparen, Rparen, Specs, Tok}
    - GenDecl  {
      + Loc: {End, Start}
        Lparen: 0
        Rparen: 0
      - Specs:  [
        - ValueSpec  {
          + Loc: {End, Start}
          + Names: [1 element]
          - Values:  [
            - BasicLit  {
                Kind: "STRING"
              + Loc: {End, Start}
                Value: "`abc`"
              }
            ]
          }
        ]
        Tok: "const"
      }
    + FuncDecl {Body, Loc, Name, Type}
    ]
  + Imports: [1 element]
  + Loc: {End, Start}
  + Name: Ident {Loc, Name}
    Package: 6
  + Unresolved: [1 element]
  }
```

Slika 3.2: Prikaz AST-a isečka koda pisanog u programskom jeziku Go. Prikazano putem `https://astexplorer.net/`

Osim naredbi i izraza (koje vezuju operatori), kao osnovnih koncepata imperativnih jezika, deklaracije se ne pojavljuju u skript jezicima. Moguće je, međutim, posmatrati i promenljive u kodovima skript jezika kao promenljive deklarisane pre trenutka njihove upotrebe - detaljnije opisano u 3.1. Što se tiče njihovog tipa, može biti dozvoljena promena istog, ili, kako je izabrano u ovom radu, biće iskorišćen specijalni tip od kog potiču svi ostali tipovi.

Na slici 3.3 se mogu videti osnovni tipovi AST čvorova zasnovani na konceptima opisanim iznad. U nastavku će po odeljcim biti detaljnije opisan svaki od tipova AST čvorova sa slike.



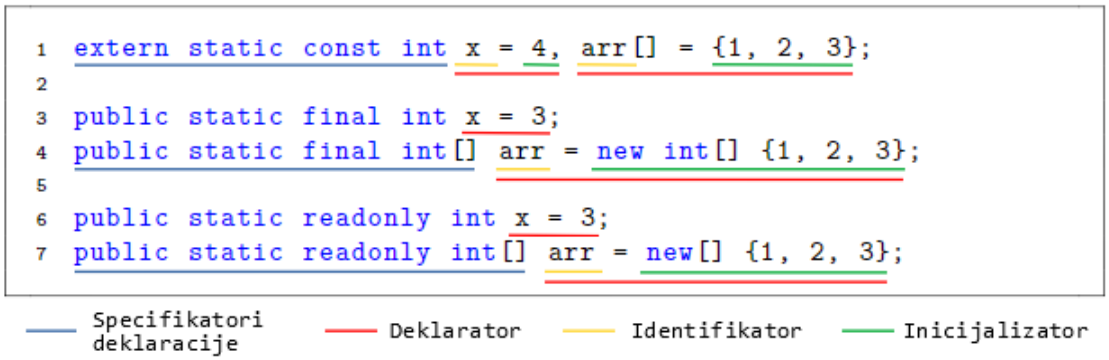
Slika 3.3: Prikaz osnovnih vrsta AST čvorova.

3.1 Čvorovi deklaracija

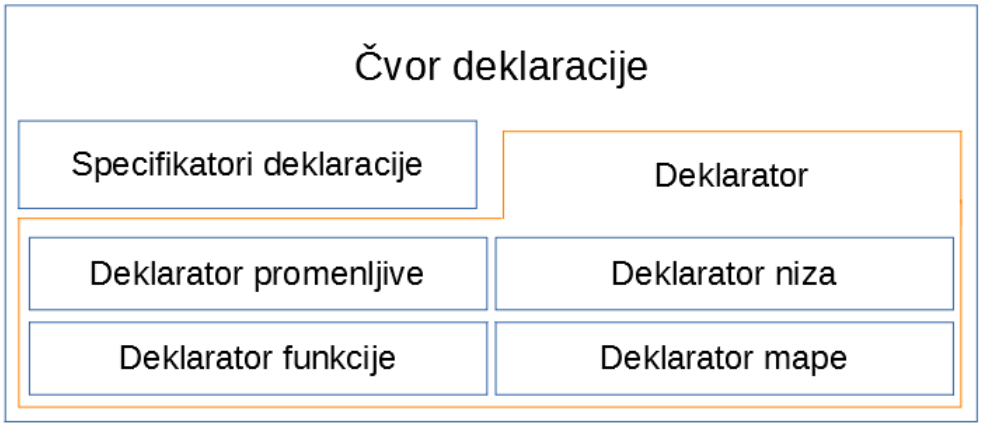
Kao što je opisano u odeljku 2.5, u striktno tipiziranim proceduralnim jezicima promenljive i funkcije koje se koriste se moraju deklarisati pre trenutka njihovog korišćenja. Prateći kvalifikatori (statičnost, konstantnost itd.) i modifikatori pristupa (javni, privatni itd.) će se u nastavku nazivati *specifikatori deklaracije* (engl. *declaration specifiers*). Nakon specifikatora deklaracije dolazi konkretan *deklarator*, koji ima specifičan oblik u zavisnosti od toga šta se deklariše - promenljiva ili funkcija. Oba ova imena su uzeta po uzoru na imena pravila gramatike programskog jezika C.

Veliki broj proceduralnih jezika dozvoljava deklarisanje više promenljivih koji dele iste specifikatore deklaracije. Stoga specifikatore neće pratiti jedan deklarator, nego potencijalno više deklaratora, u daljem tekstu *lista deklaratora*. Takođe, deklaratori u listi ne moraju biti samo deklaratori promenljivih - moguće je deklarisati i nizovnu promenljivu zajedno sa deklaracijama običnih promenljivih. Na slici 3.4 se može videti dekompozicija deklaracije promenljive i niza u različitim proceduralnim programskim jezicima a na slici 3.5 uočena hijerarhija sa podvrstama deklaratora.

Kao što je prikazano na slici 3.4, specifikatori deklaracije pokrivaju kvalifikatore, specifikatore pristupa i ime tipa. Pošto se pravi zajednička apstrakcija, potrebno je uočiti ekvivalenetne ključne reči u različitim programskim jezicima - u primeru sa slike to su `const`, `final` i `readonly`. Imena tipova u programskim jezicima Java i C# uzeta su po uzoru na programski jezik C, tako da tu ne vidimo razlike. U opštem slučaju, moguće je definisati mapiranje imena tipa u apstraktni tip. Ukoliko, na primer, posmatramo tipove koji predstavljaju realne brojeve, osim



Slika 3.4: Delovi deklaracije promenljive i niza prikazani na isečcima koda pisanog u programskim jezicima C, Java i C#.



Slika 3.5: Prikaz vrsti AST čvorova deklaracije.

tipova `float` i `double`, postoji i tip `decimal` prisutan u programskom jeziku C# [1]. Sva tri ova tipa mogu da se posmatraju na istom nivou apstrakcije kao tip realnih brojeva. Za korisnički definisane tipove, to ne može da se primeni pa se njihova imena ne mogu posmatrati apstraktnije.

Deklaratori za proceduralne jezike mogu biti deklaratori promenljive, niza ili funkcije i od toga zavisi njihov sastav. Svi deklaratori moraju sadržati informaciju o idenfitikatoru. Ukoliko je reč o deklaratoru niza, dodatno se očekuje i oznaka za niz (obično par srednjih zagrada - `[]`) i opcioni izraz koji predstavlja dimenziju niza, obično unutar oznake niza. Ukoliko je reč o deklaratoru funkcije, pored identifikatora se očekuje i lista parametara funkcije obično navedena unutar para

[1] Tip `decimal` predstavlja 128-bitni realan broj sa povećanom veličinom mantise a smanjenom veličinom eksponenta u odnosu na tip `double`. Koristi se u numeričkm izračunavanjima gde preciznost primitivnih tipova realnih brojeva nije dovoljna.

običnih zagrada. Lista parametara funkcije se može posmatrati rekurzivno - svaki parametar se može posmatrati kao varijanta deklaracije - sadrži specifikatore deklaracije (koji uključuju i tip) i deklarator, s tim što u ovom slučaju nije dozvoljeno da taj deklarator bude deklarator funkcije.

Deklaratori promenljive i niza mogu dodatno sadržati i *inicijalizatore*. Inicijalizator možemo posmatrati kao opcioni izraz u slučaju deklaratora promenljive. U slučaju deklaratora niza, inicijalizator može biti lista izraza. Deklaratori funkcije ne mogu imati inicijalizatore.

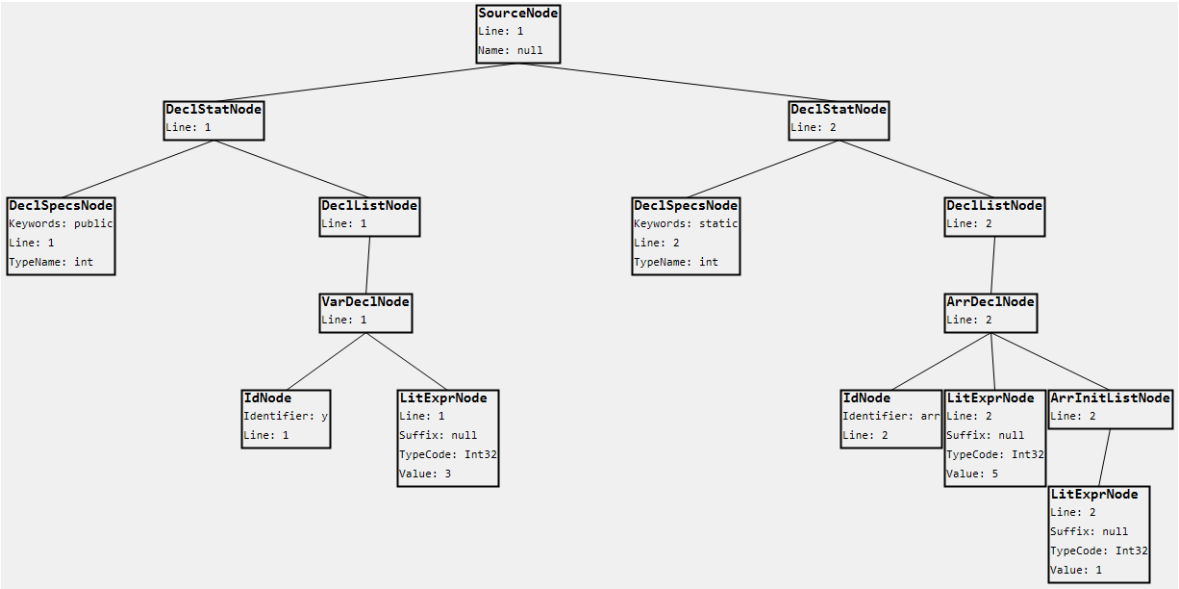
U skript jezicima su uobičajeno podržane strukture podataka kao što su skupovi i mape. Stoga, kako bi se i mape mogle predstaviti apstraktno, dodat je tip deklaratora koji predstavlja deklarator mape. Mapa se sastoji od skupa ključeva pri čemu je svakom ključu dodeljena vrednost ne nužno istog tipa kao što je tip ključa. Mape postoje i u proceduralnim jezicima, ali ključna razlika je ta što tipovi u skript jezicima nisu striktni - ključevi, ali i vrednosti mogu biti različitog tipa. Vredi naglasiti da se mape mogu porediti sa objektima određenih klasa - svaki objekat se može serijalizovati u mapu gde su ključevi imena javnih atributa klase a vrednosti su vrednosti javnih atributa objekta koji se serijalizuje. Neki jezici (kao što je Python), imaju funkcije koje od objekta vraćaju baš ovakvu mapu. Ova ideja se dalje može proširiti kako bi se serijalizovale i metode klase, označila statička i privatna polja kao i sačuvale informacije o definisanim konstruktorima. Zatim je moguće porediti mape definisane u skript jezicima sa objektima iz proceduralnih jezika.

Na slici 3.6 se mogu videti kreirani AST za nekoliko deklaracija pisanih u programskom jeziku C a na slici 3.7 se može isto videti demonstracija *automatske deklaracije* promenljivih za skript programski jezik Lua. Naime, pre prvog pojavljivanja identifikatora veštački će biti deklarisan taj identifikator, kako bi razlika između AST-ova dobijenih iz proceduralnih i skript jezika bila što manja. U ovom slučaju će se deklaracija i dodela spojiti u deklaraciju sa inicijalizatorom.

3.2 Čvorovi operatora

Svrha operatora je da vezuju izraze i da time grade nove izraze. Operator se karakteriše simbolom i *arnošću*, tj. brojem argumenata koje taj operator prima. Na osnovu arnosti, svaki operator se može apstraktno posmatrati kao članica grupe operatora sa istom arnošću. Na slici 3.8 se može videti hijerarhija operatora

```
1 extern int y = 3;
2 static int arr[5] = { 1 };
```



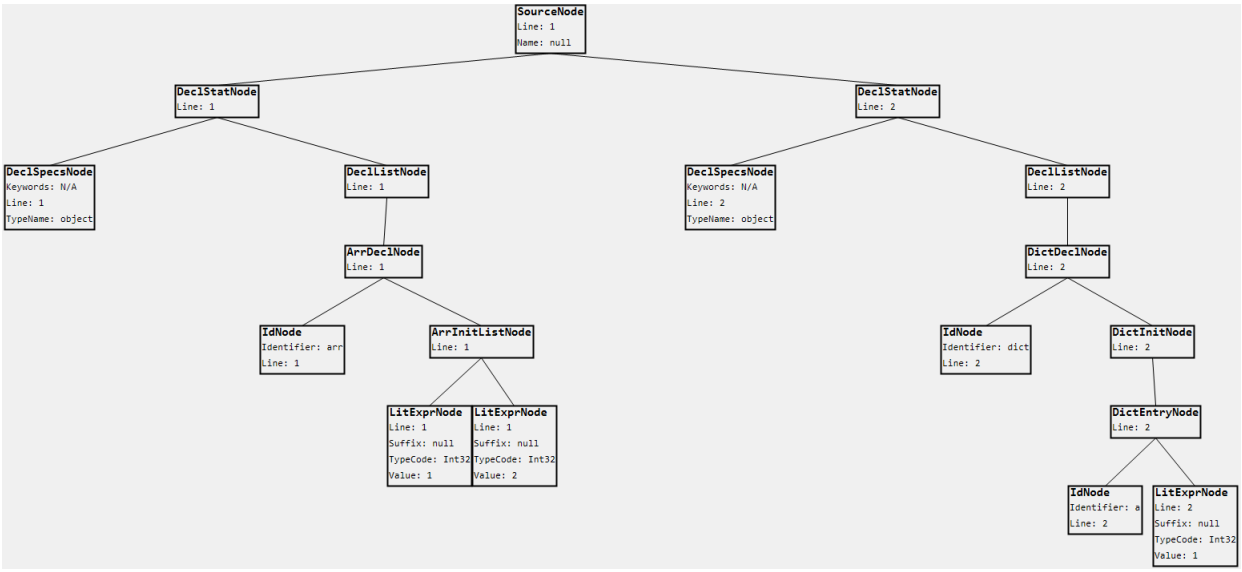
Slika 3.6: Primer deklaracije promenljive i niza u programskom jeziku C i odgovarajući AST.

korišćena dalje u apstrakciji. Binarni operatori zahtevaju dva operanda i pišu se infiksno, dok unarni zahtevaju jedan operand i pišu se prefiksno. Ternarni operatori koji postoje u nekim programskim jezicima nisu razmatrani jer se mogu posmatrati kao druge strukture [2].

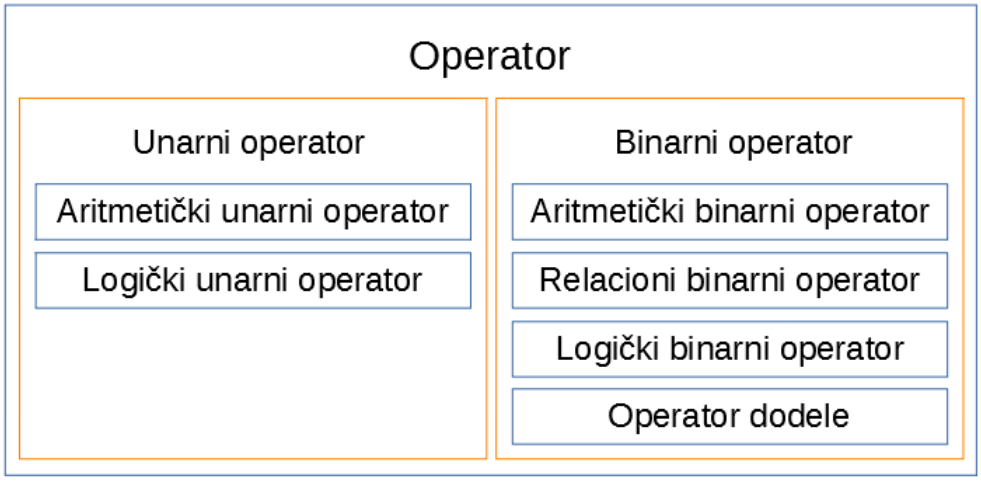
Unarni aritmetički operatori su unarni operatori koji figurišu u aritmetičkim izrazima, npr. operator promene znaka, operator bitovske negacije *Bitovski izrazi se mogu posmatrati kao vrsta aritmetičkih izraza*, operatori kastovanja ili inkrementiranja odnosno dekrementiranja. Unarni logički operatori su unarni operatori koji figurišu u logičkim izrazima, npr. operator negacije. Možemo sve ove unarne operatore posmatrati apstraktno ukoliko definišemo unarni operator kao strukturu koja definiše funckiju jednog argumenta koja transformiše dati argument na osnovu logike konkretnog unarnog operatora. Tip argumenta i povratne vrednosti pomenute funkcije zavisi od tipa unarnog operatora - aritmetički unarni operatori

[2]Na primer, ternarni operator `?:` prisutan u jezicima zasnovanim na sintaksi programskog jezika C se može zameniti naredbom uslovnog grananja.

```
1 arr = { 1, 2 }
2 dict = { a = 1 }
```



Slika 3.7: Primer deklaracije promenljive i niza u programskom jeziku Lua i odgovarajući AST.



Slika 3.8: Podela operatora na osnovu njihove arnosti.

mogu primiti vrednost bilo kog tipa [3] i vračaju vrednost proizvoljnog, ne nužno

[3]Ne postoji ograničenje na brojevne tipove jer se u nekim jezicima operatori mogu predefinisati tako da rade i za korisnički definisane tipove (engl. *operator overloading*).

istog tipa; dok unarni logički operatori primaju i vraćaju bulovsku vrednost [4]. Koristeći ovaj pristup, nije potrebno praviti novi AST čvor za svaki mogući operator, već je dovoljno da postoji samo jedan čvor koji predstavlja unarni operator - ovakav pristup odgovara varijanti AST sa regularnošću (videti sliku 2.8), omogućava opisivanje proizvoljnih operatora i nije vezan za konkretnu programsku paradigmu.

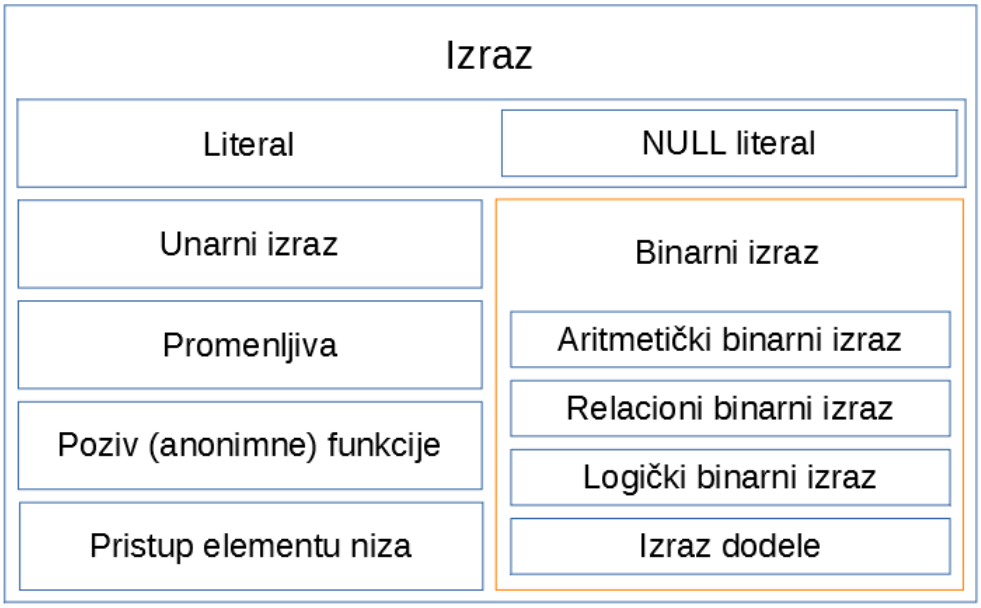
Binarni aritmetički operatori su binarni operatori koji figurišu u aritmetičkim izrazima, npr. operatori koji odgovaraju matematičkim operacijama sabiranja, oduzimanja, množenja, deljenja, stepenovanja itd. ali i bitovski binarni operatori kao što su operatori bitovskog pomeranja. Binarni relacioni operatori su binarni operatori koji figurišu u relacionim izrazima, npr. operatori poretka ($<$, $\leq$, $>$, $\geq$) i poređenja po jednakosti ili različitosti ($=$, $\neq$). Binarni logički operatori su binarni operatori koji figurišu u logičkim izrazima, npr. bulovske operacije ($\wedge$, $\vee$). Slično kao i za unarne operatore, moguće je apstraktno posmatrati sve binarne operatore tako što ih definišemo kao strukturu koja definiše binarnu funkciju dva argumenta koja transformiše date argumente na osnovu logike konkretnog binarnog operatora. Tip argumenata i povratne vrednosti te funkcije zavisi od tipa binarnog operatora, kao i u slučaju unarnih operatora - aritmetički binarni operatori primaju dva argumenta proizvoljnog tipa i vraćaju rezultat proizvoljnog, ne nužno istog tipa; relacioni binarni operatori primaju iste tipove argumenata kao i aritmetički binarni operatori, međutim povratna vrednost mora biti bulovskog tipa; dok logički binarni operatori zahtevaju da argumenti i povratna vrednost budu bulovskog tipa. Pritom, na prvi pogled nije jasno kako se operator dodele može uklopiti u ovaj šablon ali, na osnovu toga da je dodela zapravo sporedni efekat i da se posmatra kao izraz čija je vrednost jednaka vrednosti izraza sa desne strane operatora, može se primeniti isti princip kao i za aritmetičke binarne izraze. Neki programski jezici dozvoljavaju i složene operatore dodele, koji se mogu dekomponovati na kompoziciju jednostavnijih operatora.

3.3 Čvorovi izraza

Izraz, kao što se može videti na primeru gramatike sa slike 2.13, se definiše rekurzivno i izraze proširuju razni operatori. Na slici 3.9 se mogu videti tipovi

[4]U nekim programskim jezicima postoji implicitna konverzija brojevnih tipova u bulovski tip, što se jednostavno može posmatrati kao poređenje vrednosti po jednakosti sa nulom.

apstraktnih konstrukcija koje će se koristiti da bi se predstavili izrazi. Dodatno, za vezivanje izraza će se koristiti apstrakcije operatora definisane u odeljku 3.2.



Slika 3.9: Vrste čvorova izraza.

Najjednostavniji izraz predstavljaju konstante ili *literali*. Literali mogu bili bilo brojevne konstante, karakterske konstante ili konstantne niske. Literali često mogu imati i sufiks (najčešće za brojevne literale), koji određuje tip literala u slučajevima gde postoji dvosmislenost. Na primer, literal 5 možemo posmatrati kao 32-bitni ceo broj ili kao 64-bitni ceo broj (ali i kao realan broj, ako ne zahtevamo da realne brojeve moramo pisati u nepokretnom ili pokretnom zarezu). Da bi se ova dvosmislenost uklonila, možemo eksplicitno naznačiti da se govori o 64-bitnom celom broju dodavanjem sufiksa `L`, ako je u pitanju programski jezik C ili njemu slični jezici. Takođe, pošto većina programskih jezika dozvoljava rad sa pokazivačima ili neposredno koristi alokaciju memorije za kreiranje objekata, uobičajeno je korišćenje prazne adrese kao specijalne vrednosti (`null` ili `nil`). Za ovu specijalnu vrednost je moguće kreirati poseban tip literala, jer ovaj literal može biti bilo kog tipa koji nije primitivni tip.

Osim literala, samostalne promenljive mogu predstavljati validan izraz, u kom slučaju je vrednost izraza trenutna vrednost te promenljive. Slično važi i za indeksni pristup elementu niza [5]. U slučaju indeksnog pristupa, potrebno je navesti

[5]Isto važi i za bilo koju drugu kolekciju, ukoliko je nad njom definisan operator indeksnog pristupa, predefinisanje ovog operatora nije razmatrano u ovom radu.

izraz čija vrednost označava indeks (to ne mora biti jednostavni literal). Postoje smislena ograničenja šta sve sme da se nađe unutar izraza koji predstavlja indeks elementa niza tako da to semantički ima smisla, ali se na ovom nivou ne bavimo semantičkom analizom.

Unutar izraza se mogu naći i pozivi funkcija. Naravno, pretpostavljamo da funkcija ima povratnu vrednost, koja će se iskoristiti nakon poziva funkcije u kontekstu iz kojeg je ona pozvana. Pod uticajem funkcionalne paradigme, veliki broj programskih jezika dozvoljava definisanje anonimnih (lambda) funkcija, čija se definicija može smatrati izrazom koji se može dodeliti nekom simbolu. Stoga se i anonimne funkcije mogu smatrati kao izrazi.

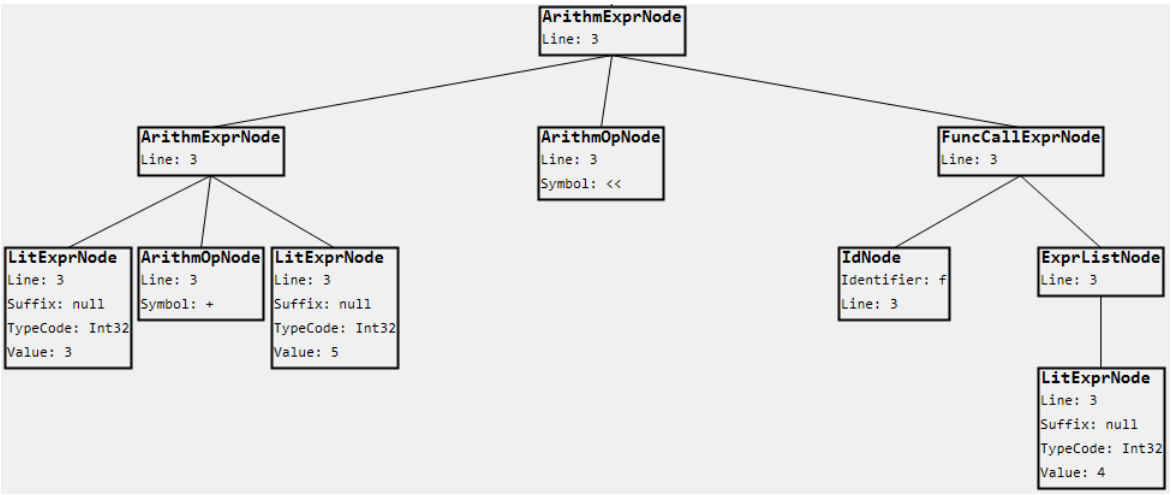
Operatori opisani u 3.2 mogu vezati sve tipove iznad i formirati složenije izraze. U zavisnosti od broja izraza koje operator vezuje, izraze možemo podeliti na unarne i binarne. Unarne izraze nadograđuju unarni operatori dok su binarni izrazi dobijeni primenom binarnog operatora na dva izraza. U zavisnosti od tipa binarnog operatora (videti sliku 3.8), binarne izraze delimo na sličan način. Naravno, svaki od tipova binarnog izraza zahteva odgovarajući tip binarnog operatora. Slično se može uraditi i za unarne izraze, ali takođe i napraviti podela na prefiksne i postfiksne, ali to nije rađeno u ovom radu zarad pojednostavljenja - u nekim jezicima su određeni operatori prefiksni dok su u nekim ti isti operatori postfiksni pa stoga nije pravljena ova podela.

Na slici 3.10 se mogu videti kreirani AST za izraz `(3 + 5) << f(4)`. Ovaj izraz poprima isti oblik bez obzira na to koji je programski jezik u pitanju, ali iako se sintaksa bude razlikovala ili operatori budu imali drugi simbol, logika operatora opisana putem funkcije će biti ista.

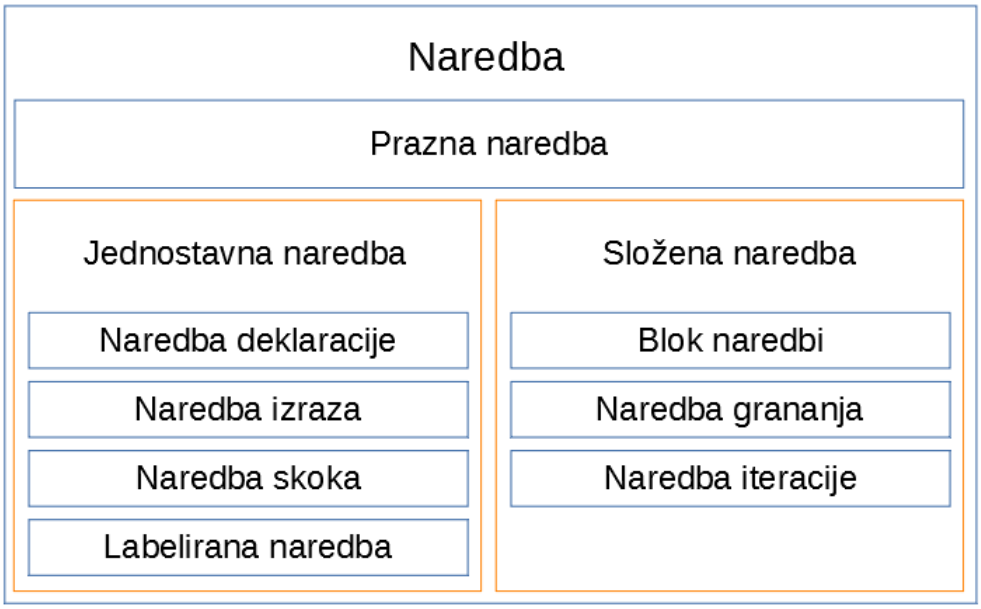
3.4 Čvorovi naredbi

Naredbe su najkomplikovanije za apstrahovanje zbog njihove raznovrsnosti. Programski jezici često uvode nove sintaksne strukture i naredbe koje nisu do tada viđene u ostalim jezicima. Uprkos svemu tome, ipak je moguće uočiti neke sličnosti sa već postojećim konceptima i svesti ih na isti nivo. Na slici 3.11 se mogu videti tipovi apstraktnih konstrukcija koje će se koristiti da bi se predstavile naredbe.

Veliki broj programskih jezika podržava praznu naredbu, sa semantikom ne izvršavanja nikakvih operacija. U programskim jezicima koji su zasnovani na



Slika 3.10: AST generisan od izraza `(3 + 5) << f(4)`.



Slika 3.11: Vrste čvorova naredbi.

sintaksi jezika C, praznu naredbu navodimo samo korišćenjem simbola za kraj naredbe (`;`), dok u programskom jeziku Python koristimo ključnu reč `pass`.

Naredbe se mogu podeliti na *jednostavne* i *složene*. Složene naredbe se sastoje od više drugih naredbi. Primer jednostavne naredbe može biti deklaracija promenljive, dok primer složene naredbe može biti definicija funkcije koja se sastoji od više jednostavnih deklaracija ali možda i drugih složenih naredbi kao što su

grananja i petlje.

Jednostavne naredbe uključuju naredbe deklaracije i izraza. Razlog zašto se deklaracije i izrazi opet pojavljuju sada je taj što izrazi sami po sebi mogu biti deo drugih naredbi. Ukoliko se naredba sastoji samo od izraza, onda nju zovemo naredbom izraza. Primer može biti izraz dodele - vrednost izraza dodele se može koristiti u drugim izrazima ali, ukoliko samo želimo da izvršimo dodelu i ništa više u istoj naredbi, onda izraz dodele umotavamo u naredbu izraza. Slično važi i za deklaracije, ukoliko razmotrimo idiomsku `for` petlju (od standarda C99) - moguće je deklarisati promenljive koje se koriste unutar ciklusa ali to nije naredba deklaracije već deklaracija koja se koristi unutar druge naredbe.

Osim naredbi deklaracija i izraza, podržane su i labelirane naredbe i naredbe. Naredbe se mogu označiti, po uzoru na koncept *labele* u imperativnim jezicima - identifikatora koji označava lokaciju u izvornom kodu. Labele se u imperativnim jezicima najviše koriste da bi se izvršili skokovi na određene lokacije u kodu ali su takođe prisutne i u proceduralnim jezicima (npr. kroz naredbu višestrukog grananja - `switch` ili u nekim jezicima `case`). Labelirana naredba se sastoji od naredbe i identifikatora koji predstavlja labelu. Moguće je debatovati da li je ona stoga složena naredba, ali pošto se ne sastoji od više od jedne naredbe stoga je svrstana u proste naredbe. Naredba skoka,

Naredbe skoka se koriste obično u paru sa labeliranim naredbama, ali to ne mora uvek biti slučaj. Iako ove čvorove koristimo da bismo predstavili naredbe skoka prisutne u imperativnim jezicima, one predstavljaju i naredbe prekida (`break` ili `continue`) ili povratka vrednosti funkcije (`return`). U slučaju da je u pitanju skok na određenu labelu, onda se sastoji i od identifikatora koji predstavlja labelu na koju se skače. Ukoliko je u pitanju naredba prekida, nisu potrebne nikakve dodatne informacije (mada se i u tom slučaju može iskoristiti činjenica da su u pitanju skokovi pa se može labelirati petlja na koju se odnosi naredba prekida). U slučaju povratka vrednosti funcije, sadrži opcioni izraz čija vrednost predstavlja povratnu vrednost funkcije.

Složene naredbe se sastoje od više drugih naredbi (ne nužno samo od jednostavnih). Često je potrebno izvršiti više naredbi u okviru jednog konteksta i za to se koristi blok naredba. Blok naredba grupiše više drugih naredbi u jednu. Blok naredba se u proceduralnim jezicima obično navodi eksplicitno, za programski jezik C pomoću velikih zagrada (`{}`), a za skript jezike često je implicitna ili se navodi korišćenjem različitih nivoa indentacije (Python).

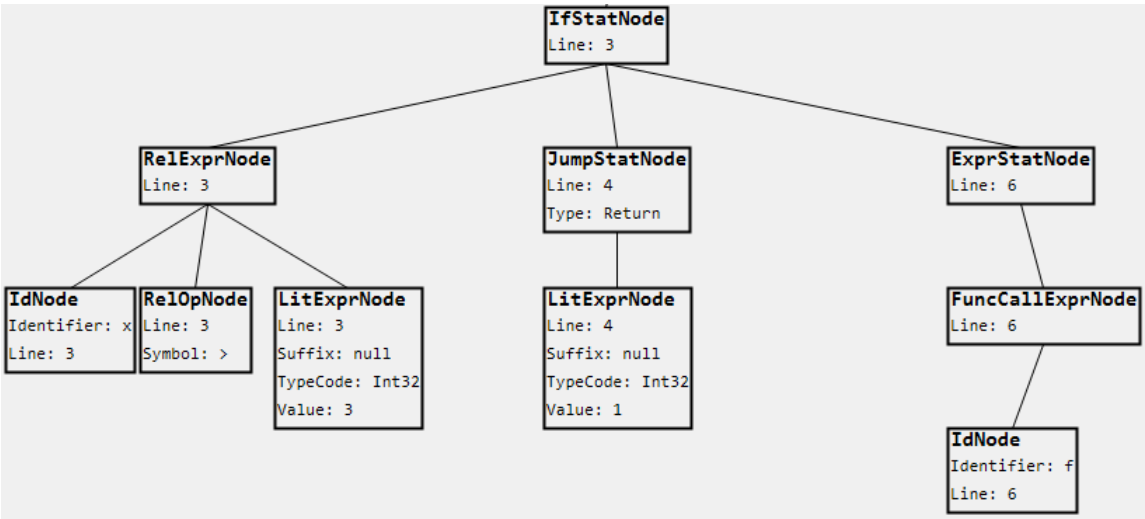
Naredbe uslovnog grananja se sastoje od *uslova*, koji može biti relacioni ili logički izraz, naredbe koja se vrši ukoliko je uslov ispunjen, i opciono naredbe koja se izvršava ako uslov nije ispunjen. Rezultat uslovnog izraza, iako mora biti istinitosna vrednost, je dozvoljeno da bude bilo kog tipa (dakle nema ograničenja samo na relacione i logičke izraze) iz razloga što određeni programski jezici dozvoljavaju automatsku konverziju brojevnih tipova u logički (C). Štaviše, nekada je moguća i implicitna konverzija određenih tipova u logički tip definisanjem implicitnih operatora konverzije (C#). Zato će u apstrakciji uslov biti bilo koji izraz. Što se *then* i *else* grana tiče, one mogu biti bilo koje naredbe, ali zarad konzistentnosti će obe biti blokovi naredbi.

```
1 do                                    something()
2     something()                       while (condition) do
3 while (condition)                         something()
```

```
1 repeat                                something()
2     something()                       while (not condition) do
3 until (condition)                         something()
```

Slika 3.12: Procedura svođenja ređih tipova petlji (levo) na *while* petlju (desno) prikazana u pseudo-jeziku.

Naredbe iteracije imaju raznovrsni oblik u programskim jezicima. Najčešće podržane naredbe iteracije su *for*, *while* i *foreach* petlje. U opštem slučaju, dovoljno je koristiti samo jedan tip petlji, ali zarad jednostavnosti i prisutnosti ovih tipova u velikoj većini programskih jezika oba će biti podržana. Ostali tipovi petlji, kao što su *do-while* ili *repeat-until* petlje, će se svoditi na njih. *do-while* petlja se može svesti na *while* petlju jednostavnim ponavljanjem tela petlje pre same petlje i kreiranjem obične *while* petlje sa istim uslovom i telom. Slično se može uraditi i za *repeat-until* petlju, s tim što je potrebno samo negirati uslov u dobijenoj *while* petlji. Ovaj proces je ilustrovan na slici 3.12.



Slika 3.13: AST naredbe grananja.

Glava 4

Poređenje opštih AST-ova

Jedna od motivacija svođenja AST-ova imperativnih jezika na isti nivo apstrakcije (opisane u poglavlju 3) može biti poređenje istih. Pritom, s obzirom da su u pitanju stabla, moguće je koristiti razne algoritme za poređenje stabala (ali i grafova uopšte) nad ovakvim apstrakcijama. Pritom, potrebno je i definisati kriterijum poređenja - moguće je porediti kodove po njihovoj *strukturnoj ekvivalentnosti*, *semantičkoj ekvivalentnosti* itd. U ovom radu je od značaja semantička ekvivalentnost, koja je u opštem slučaju neodlučiv problem. Međutim, ukoliko se ograničimo samo na strukturno slične kodove, moguće je dobiti smislene rezultate, nalik na one dobijene u ovom radu. Precizna definicija strukturne ekvivalentnosti se obično definiše u terminima sličnosti strukture njihovih AST-ova. Naravno, postoje kodovi koji nisu strukturno ekvivalentni ali su semantički ekvivalentni - takvi slučajevi se onda neće razmatrati zbog neispunjenosti pretpostavke o strukturnoj ekvivalentnosti.

Pretpostavka strukturne ekvivalentnosti je velika i značajno smanjuje prostor stabala koja se mogu porediti. Međutim, danas je od velikog značaja verifikacija programa dobijenih sitnim refaktorisanjem već postojećh i verifikovanih programa. Slično, prepisivanja programa sa jednog programskog jezika na drugi su jako uobičajena, pa je i u takvim situacijama implicitno prisutna strukturna ekvivalentnost (naravno, ovo zavisi od konkretnih programskih jezika ali se u praksi često smanjuje napor tako što se održava struktura koda, barem u inicijalnim verzijama).

U ovom radu je poređenje vršeno pomoću algoritma pisanog specifično za rad sa opštim AST-ovima. Grubi opis algoritma za poređenje, u daljem tekstu *upoređivač*, će biti opisan u nastavku. Upoređivač se sastoji od više upoređivača koje porede specifične tipove čvorova. Za početak, potreban je jedan adapter koji će dobiti

pokazivače na korene stabala koje je potrebno uporediti. S obzirom da tipovi čvorova mogu biti različiti, potrebno je proveriti da li su tipovi isti. Ukoliko to nije slučaj, prijavljuje se greška i rad se prekida. U protivnom, potrebno je odrediti tip čvorova i pozvati konkretni algoritam za poređenje. Ovaj početni postupak je opisan na slici 4.1.

```
1:  procedure UPOREDI(n1, n2)
2:      if n1 i n2 su istog tipa then
3:          t ← tip čvora n1
4:          if postoji definisan upoređivač za čvorove tipa t then
5:              U ← upoređivač čvorova tipa t
6:              return U(n1, n2)
7:          else
8:              if BrojDece(n1) ≠ BrojDece(n2) thend
9:                  return False
10:             else
11:                 if Atributi(n1) ≠ Atributi(n2) then
12:                     return False
13:                 for i ← 0 to BrojDece(n1) do
14:                     d1 ← dete i čvora n1
15:                     d2 ← dete i čvora n2
16:                     if not Uporedi(d1, d2) then
17:                         return False
18:                 return True
19:     else
20:         return False
```

Slika 4.1: Osnovni upoređivač AST-ova.

Podrazumevana implementacija poređenja može biti takva da se uporede atributi svih čvorova a zatim za svako dete prvog čvora uporedi sa odgovarajućim detetom drugog čvora (ukoliko imaju isti broj dece). Ako neki par dece nije jednak, onda ni njihovi roditelji nisu jednaki. Za većinu tipova čvorova ovakvo poređenje je dovoljno. Međutim, poređenje blokova naredbi je fundamentalno drugačije i za njega će biti definisane posebne procedure poređenja opisane u nastavku.

4.1 Upoređivač blokova naredbi

Podrazumevani način poređenja dece svakog čvora nije dobar u opštem slučaju za blokove naredbi jer je osetljiv na izmene redosleda naredbi - na primer promena

redosleda deklaracija. Stoga je upoređivač blokova potrebno napisati tako da može da uoči semantičku ekvivalentnost iako naredbe nisu nužno jednake.

Ideja se zasniva na poređenju vrednosti promenljivih na kraju svakog bloka naredbi. AST-ovi će se porediti paralelno - *blok-po-blok*. Naredbe svakog bloka će se izvršavati i pratiće se izmene vrednosti promenljivih deklarisanih do sada (bilo u trenutnom bloku, ili u roditeljskim blokovima). Na kraju svakog bloka će se izvršiti provera vrednosti promenljivih - svaka razlika će se prijaviti kao potencijalna greška a finalnu presudu o jednakosti će dati analiza jednakosti promenljivih. Ceo algoritam je prikazan na slici 4.2.

1: **procedure** UporediBlokove(b_1, b_2)
2: $gds_1 \leftarrow$ *deklarisani simboli svim roditeljskom blokovima bloka* b_1
3: $gds_2 \leftarrow$ *deklarisani simboli svim roditeljskom blokovima bloka* b_2
4: $lds_1 \leftarrow$ *lokalni deklarisani simboli u bloku* b_1
5: $lds_2 \leftarrow$ *lokalni deklarisani simboli u bloku* b_2
6: UporediSimbole(lds_1, lds_2)
7: IzvrsiNaredbe($b_1, b_2, lds_1, lds_2, gds_1, gds_2$)
8: **return** UporediSimbole(lds_1, lds_2) $\wedge$ UporediSimbole(gds_1, gds_2)

Slika 4.2: Upoređivač blokova naredbi.

U opisu algoritma se koristi termin *simbol* koji se sastoji od identifikatora i simboličke vrednosti promenljive. Lokalni simboli su deklarisani unutar bloka a globalni su svi simboli koji su deklarisani van trenutnog bloka a koji se mogu referisati iz njega. Pronalaženje deklarisanih simbola u bloku podrazumeva prolaz kroz naredbe bloka i registrovanje svih naredbi deklaracije, uzimanje deklaratora iz tih naredbi i, uzimajući u obzir opcione inicijalizatore, kreiranje simboličke vrednosti za upravo deklarisani identifikator. Identifikator i opcioni simbolički inicijalizator čine *simbol*. Ovo se radi za sve naredbe u bloku i rezultat je skup deklarisanih simbola.

Nakon uzimanja svih lokalnih simbola, proverava se njihova ekvivalentnost putem funkcije `UporediSimbole`. Ova funkcija proverava da li se svi simboli iz prvog bloka nalaze u drugom i prijavljuje ukoliko neki simboli fale ili ukoliko postoje simboli koji su „višak”. Zatim, za simbole koji se nalaze u oba skupa, proverava njihove simboličke vrednosti. Ukoliko su te vrednosti različite, prijavljuje se potencijalna greška i na osnovu toga da li je bilo konflikata vraća se istinitosna vrednost. Razlog zašto se ta vrednost ne koristi dalje nakon prvog poziva ove funkcije je ta što različiti inicijalizatori ne znače nužno da postoji problem. Problem postoji

ukoliko se nakon izvršavanja svih naredbi i dalje dešavaju konflikti u simboličkim vrednostima za neke promenljive.

Procedura `IzvrsiNaredbe` izvršava paralelno naredbe iz oba bloka i na osnovu toga koje su naredbe u pitanju može i da ažurira simboličke vrednosti unutar skupova deklarisanih simbola. Pseudokod ove procedure je dat na slici 4.3. Naredbe se za svaki blok izvršavaju dok se ne naiđe do naredbe iz koje se može izvući novi blok - to mogu biti naredbe grananja, iteracije, definicije funkcija i slično. Sve naredbe do pronađene naredbe koja sadrži blok se izvršavaju. Procedura `IzvrsiNaredbu` će proveriti tip naredbe i, u zavisnosti od toga da li je to naredba dodele, eventualno promeniti vrednosti u skupovima prosleđenih simbola. Nakon izvršavanja svih naredbi do pronađene naredbe koja sadrži blok, izvlači se blok iz nje (to isto se radi i za drugi blok). Kad se blokovi izvuku, rekurzivno se poziva upoređivač blokova za izvučene blokove. Po povratku iz rekurzivnog poziva nastavlja se isti postupak sve dok se ne izvrše sve naredbe. Pritom, algoritam se oslanja na strukturnu sličnost - ukoliko jedan AST ima više blokova na istoj dubini u odnosu na drugi, poređenje možda neće uočiti neke greške.

```
1:  procedure IzvrsiNaredbe(b_1, b_2, lds_1, lds_2, gds_1, gds_2)
2:      n_1 ← niz naredbi bloka b_1
3:      n_2 ← niz naredbi bloka b_2
4:      i ← j ← 0
5:      ni ← nj ← 0
6:      eq ← True
7:      while True do
8:          ni ← indeks prve naredbe koja sadrži blok u n_1 počev od indeksa ni
9:          nj ← indeks prve naredbe koja sadrži blok u n_2 počev od indeksa nj
10:         for naredba ∈ {n_1[x] | x ∈ [i..ni]} do
11:             IzvrsiNaredbu(naredba, lds_1, gds_1)
12:         i ← i + ni
13:         for naredba ∈ {n_2[x] | x ∈ [j..nj]} do
14:             IzvrsiNaredbu(naredba, lds_2, gds_2)
15:         j ← j + nj
16:         if i > Duzina(n_1) ∨ j > Duzina(n_2) then
17:             prekini petlju
18:         nb_1 ← izvuci blok iz naredbe n_1[i]
19:         nb_2 ← izvuci blok iz naredbe n_2[j]
20:         eq ← eq ∧ UporediBlokove(nb_1, nb_2)
21:         i ← i + 1
22:         j ← j + 1
23:     return eq
```

Slika 4.3: Upoređivač blokova naredbi.

Glava 5

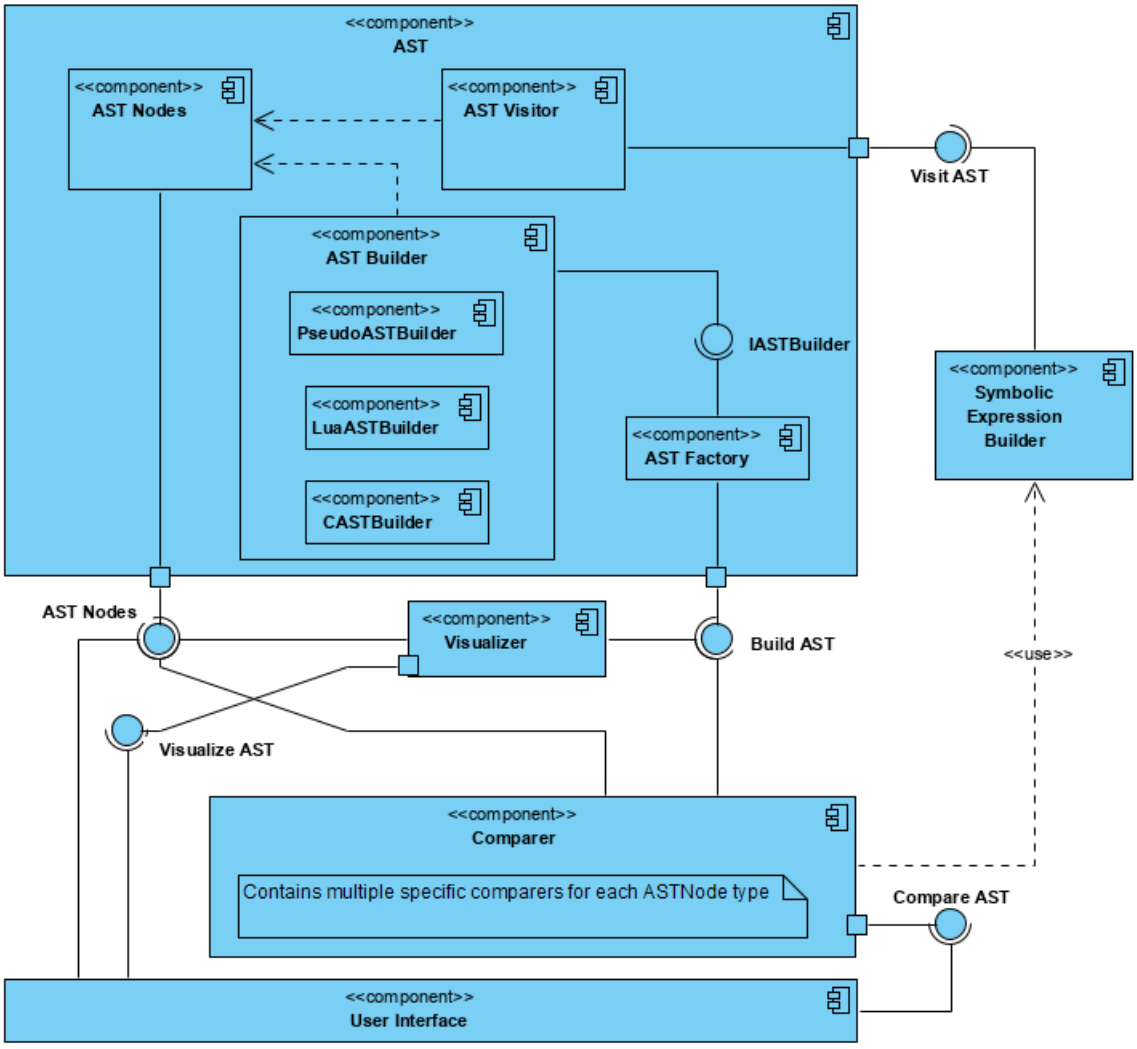
Implementacija

U ovom poglavlju će biti opisana implementacija pratećeg projekta pisanog u programskom jeziku C# 8.0, koristeći *.NET Core 3.1* radni okvir. C# je izabran zbog lakoće implementacije velikih projekata i velike podrške paketa koji se mogu preuzeti, od kojih su korišćeni *ANTLR Runtime* paket koji daje potrebne biblioteke za rad sa ANTLR generisanim parserima i *Math.NET Symbolics* paket za rad sa simboličkim vrednostima. Rezultat je konzolna aplikacija koja može da generiše, serijalizuje ili prikaže opšti AST za dati izvorni kod, ali i da ga poredi sa drugim AST-om. Čitav projekat je dostupan u potpunosti na servisu GitHub.

Jedan od glavnih ciljeva aplikacije je modularnost i jednostavna proširivost. U tom duhu se, pored implementacije klasa potrebnih za predstavljanje opšte AST apstrakcije, pruža i interfejs za kreiranje adaptera koji će od proizvoljnog stabla parsiranja kreirati opšti AST. Kao primer, adapteri su kreirani za programske jezike C i Lua, a za primer potpune slobode u izboru gramatike je kreirana gramatika za pseudo-jezik i adapter za istu, što dozvoljava poređenje kodova sa specifikacijom datom u obliku pseudo-koda. Čitav projekat se sastoji od više komponenti, od kojih su značajnije:

- Biblioteka koja pruža klase za rad sa opštom AST apstrakcijom
- Komponenta za kreiranje AST od proizvoljne gramatike putem adaptera - moguća serijalizacija u JSON
- Komponenta za semantičko poređenje AST-ova - konzolni izlaz
- Komponenta za vizualizaciju AST-ova - grafički prikaz AST-a u obliku stabla
- Korisnički interfejs - komandna linija

Čitava arhitektura data putem UML dijagrama komponenti se može videti na slici 5.1. Osim implementacije same aplikacije, svaki funkcionalni deo projekta prate i testovi jedinica koda, koji su povezani sa GitHub alatom za neprekidnu integraciju.

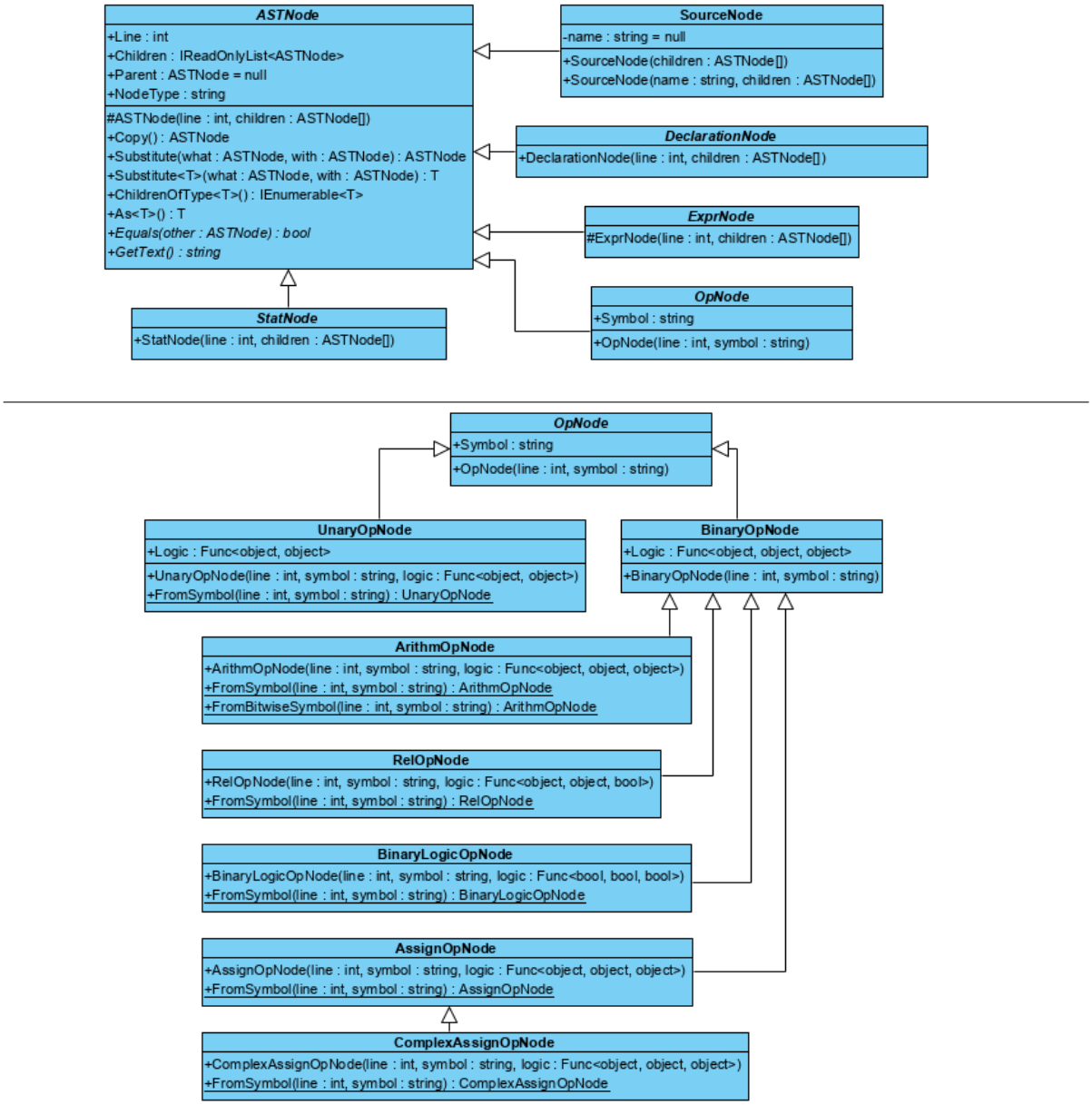


Slika 5.1: UML dijagram komponenti implementacije.

Biblioteka za kreiranje AST kao i komponenta za kreiranje AST-a od proizvoljne gramatike će biti opisane u odeljku 5.1, dok će komponenta za semantičko poređenje biti opisana u odeljku 5.2.

5.1 Implementacija apstrakcije

Implementacija prati hijerarhije opisane u poglavlju 3 kroz mehanizam nasleđivanja. Svaki tip čvora će biti zasebna klasa koja nasleđuje apstraktnu klasu `ASTNode`. Dijagram klasa koje nasleđuju klasu `ASTNode` se može videti na slikama 5.2 i 5.3. Pored implementacije klasa koje predstavljaju AST čvorove, kreiran je i interfejs za obilazak AST-a putem obrazca posetilac.



Slika 5.2: UML klasni dijagram (deo 1).

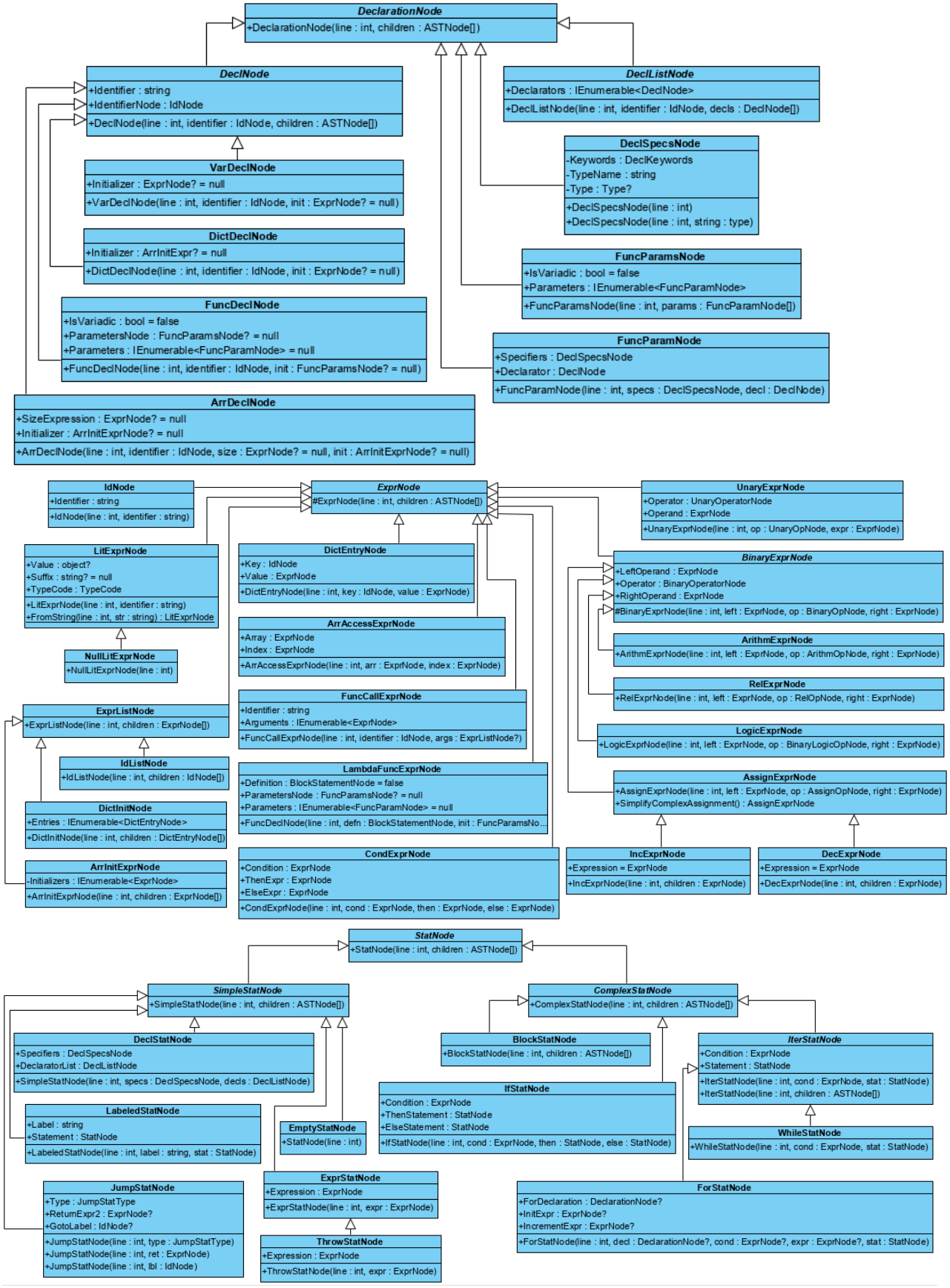
Ovako kreirana AST struktura je *imutabilna* - ne mogu se dinamički dodavati ili uklanjati deca čvorovima. Moguće je klonirati AST čvorove ili vršiti zamenu određenog podstabla drugim podstablom unutar AST-a, ne menjajući original već kopiju originala. Svaki AST čvor se može porediti po jednakosti sa drugim AST čvorom sa intuitivnom logikom poređenja pruženom kroz predefinisane operatore poređenja po jednakosti.

5.2 Implementacija upoređivača

Implementacija algoritma upoređivača opisana u poglavlju 4 se svodi na implementaciju funkcija za poređenje za svaki tip AST čvora. Te funkcije su enkapsulirane u klase koje implementiraju interfejs za upoređivač čvorova. Te klase nisu javne, tako da se poređenje vrši kroz upoređivač koji poredi instance tipa `ASTNode`, a koji putem refleksije određuje konkretni tip čvorova i, ukoliko su tipovi isti, pronalazi konkretni upoređivač i poziva operaciju interfejsa upoređivača. Upoređivači međusobno pozivaju jedni druge, kako bi se logika poređenja uprostila - pošto se naredbe deklaracije sastoje od specifikatora deklaracije i liste deklaratora, upoređivač naredbi deklaracije može pozivati upoređivač za specifikatore deklaracije i upoređivač za listu deklaratora. Rezultat rada upoređivača za algoritam *swap* se može videti na slici 5.4.

5.3 Implementacija vizualnog prikaza AST

5.4 Implementacija korisničkog interfejsa



Slika 5.3: UML klasni dijagram (deo 2).

Slika 5.4: Rezultat rada upoređivača za `swap` algoritam.

Glava 6

Zaključak

Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab:

muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab: muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Literatura

[1] ANTLR4. https://www.antlr.org/.

[2] BYACC. http://byaccj.sourceforge.net/.

[3] Clang. https://clang.llvm.org/.

[4] Context-Free Grammars. https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html.

[5] Keith Cooper and Linda Torczon. *Engineering a Compiler, 2nd Edition*. 2013.

[6] Joel Denny and Brian Malloy. IELR(1): practical LR(1) parser tables for non-LR(1) grammars with conflict resolution. pages 240–245. Association for Computing Machinery, New York, NY, United States.

[7] Design Patterns. http://www.vincehuston.org/dp/.

[8] Flex. https://www.gnu.org/software/flex/.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[10] GLR. https://www.gnu.org/software/bison/manual/html_node/GLR-Parsers.html.

[11] GNU Bison. https://www.gnu.org/software/bison/.

[12] GNU Project. https://www.gnu.org/gnu/thegnuproject.en.html.

[13] KLEE. https://klee.github.io/.

[14] LALR1. https://web.cs.dal.ca/~sjackson/lalr1.html.

[15] LexYacc. http://dinosaur.compilertools.net/.

[16] LLVM. `https://llvm.org/`.

[17] LR. `http://pages.cpsc.ucalgary.ca/~robin/class/411/LR.1.html`.

[18] Terrence Parr. *The Definitive ANTLR 4 Reference*. 2012.

[19] ProgrammingParadigms. `https://cs.lmu.edu/~ray/notes/paradigms/`.

[20] Slonneger and Kurtz. *Formal syntax and semantics of programming languages*. Addison-Wesley Pub. Co, Reading, Mass, 1995.

[21] SMT. `https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/barret2_smt.pdf`.

[22] Symbolic Execution. `https://www.cs.umd.edu/~mwh/se-tutorial/symbolic-exec.pdf`.

[23] Design Patterns. `https://www.uml.org/`.

[24] William M. Waite and Gerhard Goos. *Compiler Construction*. 1995.

Biografija autora

Ivan Ž. Ristović rođen je 17.01.1995. godine u Užicu. Osnovnu školu, kao i prirodno-matematički smer Užičke gimnazije, završio je kao nosilac Vukove diplome. Tokom navedenog perioda školovanja isticao se u oblastima matematike, informatike, fizike, hemije i engleskog jezika, što potvrđuje veći broj nagrada na Državnim takmičenjima.

Smer Informatika na Matematičkom fakultetu Univerziteta u Beogradu upisuje 2014. godine. Na navedenom smeru je diplomirao 2018. godine, posle tri godine studija sa prosečnom ocenom 9,17. Master studije upisuje na istom fakultetu odmah nakon diplomiranja.

U avgustu 2018. biva izabran u zvanje „Saradnik u nastavi“ na Matematičkom fakultetu paralelno sa master studijama. Drži vežbe iz kurseva „Računarske mreže”, „Funkcionalno programiranje”, „Programske paradigme” i „Objektno orijentisano programiranje” na kasnijim godinama osnovnih studija.

Oblasti interesovanja uključuju pre svega razvoj i verifikaciju softvera, mikroservise i računarske mreže.