UDC: 004:004.432.2 Info M str. 47-54

INFRASTRUKTURA LINVAST ZA JEZIČKI-INVARIJANTNA APSTRAKTNA SINTAKSIČKA STABLA PROGRAMA
LINVAST INFRASTRUCTURE FOR LANGUAGE-INVARIANT ABSTRACT SYNTAX TREES

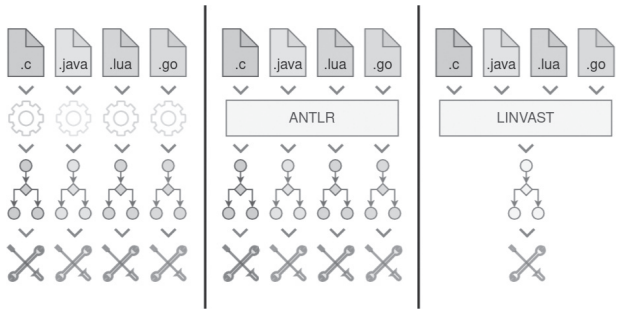
Ivan Ristović

ABSTRACT: Alati za statičku analizu programa, drugim rečima analizu programa bez njegovog izvršavanja, koriste razne apstraktne reprezentacije programa. Takve reprezentacije nastaju parsiranjem izvornog koda programa, apstrahovanjem i transformacijom rezultata parsera. Zbog toga, takve međureprezentacije su usko vezane za konkretan programski jezik i stoga su nekompatibilne između različitih programskih jezika. Uska povezanost alata i programskog jezika otežava izradu alata za analizu izvornog koda nezavisno od programskog jezika, i otežava nadogradnju postojećih alata podrškom za nove programske jezike.
Ovaj rad predstavlja infrastrukturu LINVAST — opštu jezički-invarijantnu apstrakciju izvornog koda zasnovanu na apstraktnim sintaksičkim stablima. LINVAST modeluje sintaksičke konstrukte različitih programskih jezika na uniforman način, što omogućava izradu alata koji rade nad izvornim kodom programa bez obzira na programski jezik u kojem je program napisan. Infrastruktura LINVAST u programskom jeziku C# je javno dostupna i ima skoro deset hiljada preuzimanja. Trenutno implementira podršku apstrahovanja programskih jezika C, Java, Lua i Go.
KEY WORDS: Apstraktna sintaksička stabla (AST), Jezički-invarijantni AST, Statička analiza, Parser ANTLR
REZIME: Tools for static program analysis, i.e., analysis without executing the program, rely on various abstract representations of the program. These representations are created through parsing the program's source code, abstracting, and transforming the results of the parser. As a result, such intermediate representations are tightly coupled with specific programming languages and are therefore incompatible across different languages. This tight coupling of tools with particular languages makes it difficult to develop tools for source-code analysis that are language-independent, and makes it difficult to extend existing tools with support for other programming languages.
This paper presents the LINVAST infrastructure — a common language-invariant source-code abstraction based on abstract syntax trees. LINVAST models syntactic constructs of different programming languages in a uniform way, enabling the creation of tools that operate on source code regardless of the programming language in which the program is written. The LINVAST infrastructure, implemented in the C# programming language, is publicly available and has nearly ten thousand downloads. It currently implements common abstractions for the C, Java, Lua, and Go programming languages.
KLJUČNE REČI: Abstract syntax trees (AST), Language-invariant AST, Static analysis, ANTLR parser

UVOD

Apstraktno sintaksičko stablo programa (engl. *abstract syntax tree*, skraćeno *AST*) se koristi u procesu prevođenja izvornog koda pisanog u višem programskom jeziku kao što je Java ili C, u mašinski kôd koji će se izvršavati na specifičnoj mašini. AST apstrahuje sintaksičke detalje, održavajući strukturne karakteristike programa. Stoga se koristi u *prednjem* delu (engl. *front end*) prevodioca [1] kao rezultat sintaksičke analize izvornog koda, i predstavlja osnovu za međureprezentacije programa u kasnijim fazama prevođenja [2]. Osim primene u prevodiocima, koriste se i za analizu ponašanja programa [3, 4, 5], alate za evoluciju [6] i transformaciju [7] programa, detekciju promena [8, 9], poređenje programa [10] i sl.

Model apstraktnog sintaksičkog stabla je usko povezan sa prevodiocem ili alatom za koji je namenjen, kao i za programski jezik čiji kôd je namenjen da predstavi. Stoga, isti konstrukt u različitim programskim jezicima može imati različitu apstraktnu reprezentaciju. Razlike dolaze od samih pravila u gramatikama jezika, kao i načina njihovog pisanja, ali i od razlika u svojstvima različitih programskih jezika. Na primer, AST model programskog jezika C [11] se znatno razlikuje u odnosu na AST model programskog jezika Java [12], iako, na primer, oba jezika podržavaju isti podskup sintaksičkih konstrukata i ti konstrukti se mogu predstaviti na isti način u oba modela.



Slika 1: *Pristupi izgradnji alata za statičku analizu zasnovanu na apstraktnim sintaksičkim stablima programa.*

Nepostojanje opšte apstraktne reprezentacije znatno otežava pisanje alata koji rade nad izvornim kodovima programa pisanih u različitim programskim jezicima. Takvi alati obično implementiraju zasebne apstrakcije za različite programske jezike, što prikazuje Slika 1 (levo). Generatori parsera kao što je ANTLR [13] mogu da pruže jedinstveni radni okvir za obilazak stabala parsiranja izvornih kodova različitih programskih jezika. Međutim, struktura stabla parsiranja zavisi direktno od gramatike jezika. Stoga, alat i dalje mora da implementira svojstva zasebno za svaki od programskih jezika koje podržava (Slika 1, centar). Štaviše, nije moguće jednostavno proširiti isti alat da prepozna izvorne kodove pisane u nepodržanom programskom jeziku.

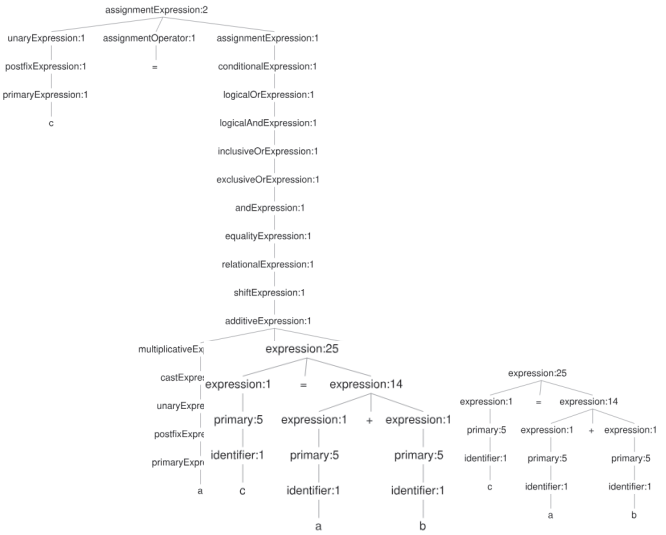
Ovaj rad predstavlja radni okvir LINVAST koji pruža opšti AST model — jedinstvenu, univerzalnu, i jezički-invarijantnu apstraktnu reprezentaciju izvornog koda, kao i podršku za građenje i obradu opštih AST. Zbog univerzalne apstraktne reprezentacije sintaksičkih konstrukata, LINVAST omogućava kreiranje univerzalnih alata koji rade nad izvornim kodovima programa nezavisno od programskih jezika u kojim su programi pisani. Pošto graditelji opštih AST u okviru infrastrukture LINVAST obavljaju posao apstrakcije konkretnog programskog jezika, jednom napisan alat koji kao ulaz prima opšti AST nije neophodno menjati da bi se proširila podrška za novi programski jezik (Slika 1, desno).

Cilj projekta LINVAST je implementacija opšte apstrakcije za najpopularnije programske jezike imperativne paradigme. Projekat je otvorenog koda [14], napisan u programskom jeziku C#, sa bibliotekom dostupnom na NuGet platformi, sa skoro deset hiljada preuzimanja [15]. Trenutno je implementirana podrška apstrahovanja izvornih kodova programa pisanih u programskim jezicima C, Java, Lua i Go. LINVAST je lako proširiv na proizvoljni programski jezik kroz intuitivan interfejs graditelja, i pruža radni okvir za česte AST operacije (npr. obilazak ili transformacije), olakšavajući proces izrade alata koji rade nad opštim AST. Evaluacija je vršena jediničnim testovima za svaki programski jezik pojedinačno, ali i nad popularnim projektima otvorenog koda. Rezultati pokazuju da LINVAST daje praktičan i proširiv radni okvir koji omogućava lako pisanje, održavanje, i ponovnu upotrebljivost alata koji rade nad sintaksičkim stablima programa.

APSTRAKTNA SINTAKSIČKA STABLA PROGRAMA

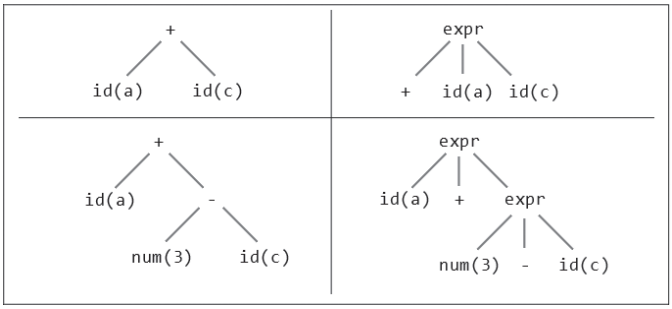
Uloga apstraktnog sintaksičkog stabla je da predstavi semantiku strukture izvornog koda u obliku stabla. AST nastaje apstrahovanjem rezultata rada *parsera*, komponente prednjeg dela prevodioca koja vrši parsiranje izvornog koda. Rezultat rada parsera, tzv. *stablo parsiranja*[1], sadrži sve relevantne informacije o izvornom kodu ali je usko vezano za gramatiku programskog jezika. Na primer, rekurzivni spust primenjen u pravilima izraza u C gramatici daje veoma duboko stablo za jednostavne izraze. Slično, sintaksičke razlike samih programskih jezika i njihovih svojstava dodatno doprinose izgledu stabla parsiranja. Slika 2 prikazuje dva stabla parsiranja za isti izraz ($c = a + b$) u različitim programskim jezicima — C (levo) i Java (desno).

[1] Parser zapravo nikada ekspicitno ne kreira stablo parsiranja, već stablo parsiranja služi kao formalizam za razumevanje rada parsera. Generatori parsera kao što je ANTLR [13] mogu da, opciono, generišu stablo parsiranja kao i interfejs za obilazak i obradu stabla parsiranja.



Slika 2: Stabla parsiranja izraza $c = a + b$ u programskim jezicima C (standard C99, levo) i Java (Java 11, desno). Slike dobijene uz pomoć alata ANTLR Lab [23].

AST se dobija apstrahovanjem stabla parsiranja, uklanjanjem informacija koje su neophodne prilikom parsiranja, ali nisu informativne u drugom kontekstu. Za razliku od stabla parsiranja, čvorovi ne odgovaraju nužno pravilima u gramatici jezika, već se model skupa tipova čvorova dizajnira tako da se održi semantika a ukloni sintaksička redundantnost. Postoji više varijanti AST u zavisnosti od toga kako se modelira skup tipova čvorova AST. Slika 3 prikazuje neke varijante AST za jednostavne izraze. Na primer, varijanta AST bez regularnosti uvodi zaseban čvor koji označava sabiranje (Slika 3, levo), dok varijanta sa regularnošću tu informaciju čuva unutar apstraknijeg tipa koji predstavlja binarni izraz (Slika 3, desno). Ukoliko je cilj minimizacija kompleksnosti stabla, veličina skupa tipova čvorova se povećava. Slično, ukoliko je cilj minimizacija skupa tipova čvorova u modelu AST, informacije koje bi se prosledile kroz sam tip čvora zahtevaju eksplicitno enkodiranje.



Slika 3: Varijante apstraktnog sintaksičkog stabla bez regularnosti (levo) i sa regularnošću (desno) za izraze a+c (gore) i a+(3-c) (dole).

Dobijanje AST za proizvoljni programski jezik je moguće korišćenjem generatora parsera koji na ulazu dobijaju gramatiku jezika i kao svoj rezultat daju implementaciju leksera i parsera u proizvoljnom ciljanom programskom jeziku. U te svrhe se mogu koristiti generatori parsera kao što su ANTLR [13] ili GNU Bison [16], u zavisnosti od toga koji format gramatike je poželjan i koji ciljani programski jezici su podržani za implementaciju leksera i parsera.

ANTLR

ANTLR (*ANother Tool for Language Recognition*) [13] je generator leksera i parsera zasnovan na LL(*) algoritmu parsiranja [17]. Koristi se u izgradnji programskih jezika kao radni okvir za izradu i obilazak prvenstveno stabala parsiranja ali i apstraktnih sintaksičkih stabala. ANTLR verzije 4 je gradivni element popularnih jezika i alata među kojima su programski jezik Groovy [18], Jython [19] implementacija programskog jezika Python koja se izvršava na JVM, Apache Cassandra [20], jezik za naprednu pretragu na platformi X (Twitter) [21], programski jezik Apex kompanije Salesforce [22], i mnogi drugi.

Da bi se generisao parser za proizvoljni programski jezik pomoću alata ANTLR, neophodno je definisati kontekstno-slobodnu gramatiku programskog jezika. ANTLR zatim, na osnovu ulazne gramatike, generiše implementaciju leksera i parsera za datu gramatiku u nekom od podržanih programskih jezika, među kojima su Java, C#, Python, C++ i Go. Osim parsera, moguće je generisati i interfejs za obilazak stabla parsiranja koji prati obrazce *posetilac* (engl. *visitor*) i *posmatrač* (engl. *listener*).

ANTLR interfejs za obilazak stabla po obrascu *posetilac* se može iskoristiti za izgradnju AST. Prilikom posećivanja čvorova stabla parsiranja, moguće je sintetisati odgovarajući čvor apstraktnog sintaksičkog stabla, gradeći rezultujući AST kroz obilazak. Pošto parseri koje ANTLR generiše kreiraju stablo parsiranja čiji su čvorovi usko povezani sa gramatikom jezika, neophodno je implementirati posebnu logiku *graditelja*, koji apstrahuju stabla parsiranja u zavisnosti od gramatike jezika. U opštem slučaju, graditelji ne moraju koristiti isti model tipova čvorova AST.

SAVREMENA REŠENJA ZA UNIFIKACIJU AST MODELA

Alati i radni okviri za definisanje i generisanje apstraktnih sintaksičkih stabala se obično ograničavaju na mali podskup programskih jezika ili se oslanjaju na generatore parsera kao što su ANTLR [13] ili GNU Bison [16].

Projekat *Unifikovani AST*[2] (skr. *UAST*) [24] predstavlja uopšteni model sintaksičkog stabla programa konstruisan od sintaksičkih stabala jezika Java, JavaScript i Python. UAST koristi ANTLR za dobijanje stabala parsiranja programa koja se dalje apstrahuju u unifikovani AST, uz automatizovanu transformaciju apstraktnih sintaksičkih stabala (Astranaut) [25] kroz domenski-specifičan jezik (DSL) za opis i transformacije sintaksičkih stabala. Ovaj pristup omogućava automatsku konverziju poznatih sintaksičkih konstrukata u unifikovani AST, ali zahteva modifikovanje domenski-specifičnog jezika za podršku nepoznatih sintaksičkih konstrukata, što ne samo da može zahtevati izmene alata koji koriste UAST, nego i potencijalne konflikte u DSL pravilima. Pritom, DSL za opis transformacija ne garantuje da će se isti konstrukti u različitim programskim jezicima mapirati u isti UAST konstrukt.

JetBrains Intellij zajednica koristi modul UAST[3] [26] koji pruža unifikovani programski interfejs za rad sa konstruktima programskih jezika koji se izvršavaju na Java Virtualnoj Mašini (JVM). Trenutno pruža podršku za programske jezike Java, Kotlin, Scala i Groovy. UAST se koristi za izradu dodataka za razvojno okruženje *IntelliJ* (IDE).

Projekat *UnifiedJS* [27] predstavlja skup alata otvorenog koda koji rade nad struktuiranim podacima u formi univerzalnog sintaksičkog stabla [28]. *UnifiedJS* podržava transformacije velikog broja jezika koji se koriste za opis struktuiranog sadržaja kao npr. *HTML* ili *Markdown*. Trenutno se koristi u preko milion projekata otvorenog koda na platformi *GitHub*. Specifikacija sintaksičkih stabala projekta *UnifiedJS* nema za cilj opisivanje konstrukata programskih jezika, tj. opis apstraktnih sintaksičkih stabala, već se fokusira na opis struktuiranih podataka.

Komandni interfejs *Coala* [29] za ispravljanje koda definiše bazu činjenica koje opisuju programski jezik [30], koja se onda koristi za implementaciju podrške za razne programske jezike. Baza činjenica sadrži opis sintakse programskog jezika (ključne reči, skup tipova podataka), ali ne i sintaksička pravila. Stoga se *Coala* koristi za implementaciju alata za ispravljanje jednostavnih sintaksnih grešaka ili kao unifikovani interfejs za pozivanje eksternih alata za formatiranje ili statičku analizu.

JEZIČKI-INVARIJANTNA SINTAKSIČKA STABLA PROGRAMA

Cilj jezički-invarijantnih (skr. opštih) sintaksičkih stabla programa je da uklone sintaksičke specifičnosti programskog jezika prisutne u apstraktnim sintaksičkim stablima programa. Opšta apstraktna sintaksička stabla se dobijaju direktnim apstrahovanjem stabala parsiranja, uklanjanjem jezički-specifičnih sintaksičkih konstrukata ili njihovom zamenom odgovarajućim ekvivalentnim i univerzalnim sintaksičkim konstruktima. Model skupa tipova čvorova opštih AST je konstruisan detaljnom analizom svojstava imperativnih programskih jezika i identifikovanjem ponovno upotrebljivih sintaskičkih konstrukata.

Većina gramatika programskih jezika u okviru imperativne programske paradigme dele sllične sintaksičke konstrukte koji omogućavaju definisanje minimalnog i potpunog skupa čvorova apstraktnog sintaskičkog stabla. To uključuje literale, izraze, naredbe, procedure ili funkcije, i sl. Dodatno, moguće je definisati i hijerarhiju između ovih koncepata, ali je to neophodno učiniti tako da se ne uvede kontradikcija. Na primer, povratna vrednost funkcije može biti deo većeg izraza, ali može biti i samostalna naredba čija svrha nije njena povratna vrednost, već njena nuspojava (sporedni efekat). Model skupa tipova čvorova opšteg AST je stoga neophodno konstruisati tako da je moguće predstaviti oba slučaja.

[2] Skraćenica UAST se koristi u istom kontekstu od strane različitih projekata koji za cilj imaju unifikaciju modela AST.

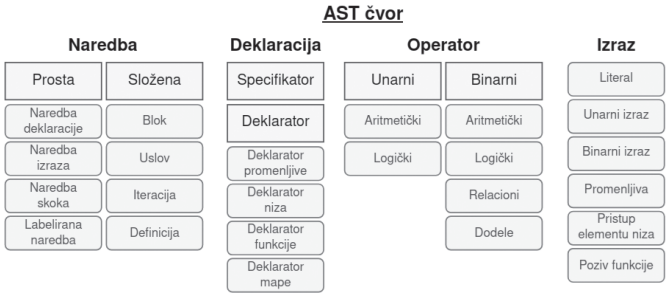
[3] Modul UAST JetBrains Intellij zajednice nije u asocijaciji sa Unifikovanim AST projektom [24].

Odlike različitih programskih paradigmi otežavaju definisanje opšte apstrakcije. Na primer, deklaracije se ne pojavljuju u skript jezicima zbog dinamičke tipiziranosti. Ali, sa druge strane, anonimne (lambda) funkcije koje potiču iz funkcionalne programske paradigme su sveprisutne u modernim programskim jezicima koji nisu pripadnici funkcionalne paradigme. Ovaj rad se zasniva prvenstveno na imperativnoj paradigmi, uz osvrt na popularne odlike drugih paradigmi koje imperativni programski jezici preuzimaju kao što su dinamička tipiziranost ili anonimne funkcije. Na primer, moguće je posmatrati i promenljive u skript jezicima kao promenljive deklarisane neposredno pre trenutka njihove upotrebe iili na početku nivoa dosega. Što se tiče njihovog tipa, može biti dozvoljena njegova promena tokom izvršavanja programa, ili, kako je izabrano u ovom radu, korišćenje specijalnog apstraktnog tipa (objekat) od kog potiču svi ostali tipovi.

Neophodno je napomenuti da se apstrahovanjem mogu izgubiti značajne informacije koje mogu promeniti semantiku programa. Ukoliko uzmemo za primer operator "+" u programskim jezicima C i Java, njegovim apstrahovanjem gubimo informaciju o redosledu izvršavanja — u C standardu nije propisano kojim redosledom će se izračunavati operandi, dok je u jeziku Java redosled izračunavanja operanada zagarantovan. Takve informacije je moguće zadržati u čvorovima opšteg AST.

MODEL SKUPA TIPOVA ČVOROVA OPŠTEG AST

Hijerarhija tipova čvorova opšteg AST (prikazana na Slici 4) počinje zajedničkim nadtipom za sve tipove čvorova (*AST čvor*). Svaki sintaksički koncept se zatim predstavlja zasebnim tipom čvora koji nasleđuje drugi tip opšteg AST čvora. To su, na primeru sa Slike 2: naredba, deklaracija, operator i izraz. Svaki od njih se dalje deli na podtipove i shodno raščlanjava na gradivne elemente. Redosled opisa tipova čvorova opšteg AST u ovom poglavlju odgovara redosledu sa Slike 2.

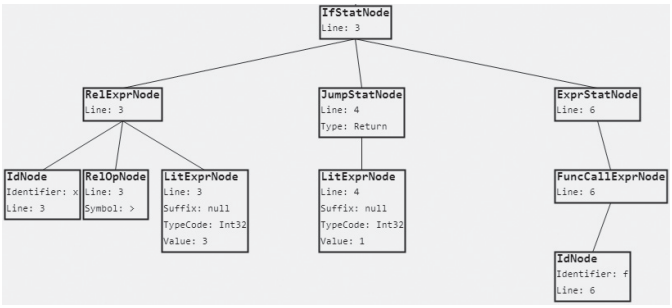


Slika 2: Hijerarhija nekih tipova čvorova opšteg AST.

ČVOROVI NAREDBI

Naredbe se u opštem AST definišu kao sintaksičke jedinice koje predstavljaju akciju. Takva definicija direktno odgovara definiciji naredbe u modernim programskim jezicima. Naredbe se dele na proste, koje se ne mogu dalje raščlaniti na jednostavnije naredbe, i složene, koje se sastoje od više drugih naredbi. Primer proste naredbe može biti dodela konstantne vrednosti promenljivoj, dok primer složene naredbe može biti definicija funkcije, čije telo čini niz naredbi, odnosno *blok* naredbi. Složene naredbe, osim definicija i blokova, uključuju i naredbe uslovnog grananja i iteracije.

Naredbe uslovnog grananja se sastoje od izraza koji predstavlja uslov grananja, naredbe koja se vrši ukoliko je uslov ispunjen (*then* grana), i opciono naredbe koja se izvršava ako uslov nije ispunjen (*else* grana). Neki programski jezici dozvoljavaju automatsku konverziju brojevnih tipova u logički kad je u pitanju izraz u uslovu grananja (npr. programski jezik C) a, štaviše, nekada je moguća i implicitna konverzija određenih tipova u logički tip definisanjem implicitnih operatora konverzije (npr. u programskom jeziku C#). Stoga, u opštem, AST, uslov grananja može biti izraz bez ograničenja na tip rezultata. Grane uslovne naredbe su modelirane kao blokovi naredbi. Slika 3 prikazuje opšti AST naredbe grananja.



Slika 3: Opšti AST naredbe uslovnog grananja *if x > 3 then return 1 else f()*.

Naredbe iteracije imaju raznovrsni oblik u programskim jezicima, pri čemu su najčešće podržane *for* i *while* petlje. U opštem slučaju, dovoljno je koristiti samo jedan tip petlji, ali zarad fleksibilnosti i *for* i *while* petlje su podržane u opštem AST. Sintaksičke varijacije *while* petlji, kao što su *do-while* ili *repeat-until* petlje, se u opštem AST svode na osnovnu while petju (Slika 4).

```
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()
```

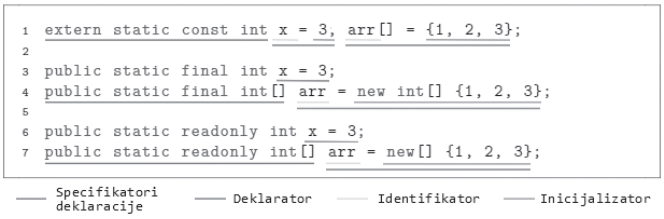
Slika 4: Svođenje *do-while* i *repeat-until* petlji na *while* petlju u opštem AST.

ČVOROVI DEKLARACIJA

Deklaracije obuhvataju tipove čvorova koji učestvuju u procesu deklaracija ili definicija promenljivih, funkcija, tipova, i sl. Deklaracije počinju kvalifikatorima (npr. *const*) i modifikatorima pristupa (npr. *public*), jednim imenom nazvanim *specifikatori deklaracije* (engl. *declaration specifiers*). Nakon specifikatora dolazi proizvoljan broj *deklaratora*[4] koji imaju specifičan oblik u zavisnosti od vrste deklaracije.

[4] Imena *specifikator deklaracije* i *deklarator* su birana po uzoru na imena odgovarajućih pravila gramatike programskog jezika C.

Na Slici 5 su prikazane deklaracije promenljive u nekoliko različitih programskih jezika i oblik deklaratora u deklaracijama promenljivih (primitivnih i nizovnih). Deklaratori se sastoje od identifikatora, koji predstavlja ime simbola koji se deklariše, kao i opcionog inicijalizatora, koji, u slučaju deklaratora promenljive, predstavlja izraz čijom evaluacijom se dobija inicijalna vrednost promenljive. U slučaju deklaratora niza, inicijalizator je lista izraza, gde svaki izraz računa vrednost narednog elementa niza. Slično važi i za druge strukture podataka često prisutne u programskim jezicima, kao što su liste ili mape. Deklaracije funkcija i složenih tipova, kao što su strukture, klase, interfejsi i sl., je takođe podržano kroz odgovarajuće pod-tipove čvorova deklaracija.



Slika 5: Delovi deklaracije promenljive i niza prikazani na isečcima koda pisanog u programskim jezicima C (linija 1), Java (linije 3 i 4) i C# (linije 6 i 7).

ČVOROVI OPERATORA

Svrha operatora je da vezuju izraze i da tako 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 iste arnosti. Binarni operatori zahtevaju dva operanda i pišu se obično infiksno, dok unarni zahtevaju jedan operand i pišu se prefiksno ili postfiksno. Ternarni uslovni operatori koji postoje u nekim programskim jezicima se mogu zameniti naredbom uslovnog grananja i stoga nemaju odgovarajuću reprezentaciju u opštem AST.

Unarni aritmetički operatori figurišu u aritmetičkim izrazima i uključuju, između ostalih i operatore promene znaka, bitovske negacije, konverzije tipa ili inkrementiranja odnosno dekrementiranja. Unarni logički operatori figurišu u logičkim izrazima, i uključuju npr. operator negacije. Možemo sve unarne operatore posmatrati apstraktno ukoliko definišemo unarni operator kao strukturu koja definiše unarnu funkciju koja transformiše svoj argument na osnovu logike konkretnog unarnog operatora. Tip argumenta i povratne vrednosti pomenute funkcije zavisi od tipa unarnog operatora — aritmetički unarni operatori mogu primiti vrednost bilo kog tipa i vraćaju vrednost proizvoljnog, ne nužno istog tipa; dok unarni logički operatori primaju i vraćaju istinitosne vrednosti.

Pozicija unarnog operatora u nekim programskim jezicima povlači drugačije izračunavanje izraza, npr., u programskom jeziku C rezultat izraza se može promeniti ako se umesto *a*++ koristi ++*a*, zbog semantike postfiksnog inkrementiranja u odnosu na prefiksno. Sa druge strane, neki programski jezici ne prave tu distinkciju, ili se njihova semantika operatora ne mora slagati sa drugim programskim jezicima. Stoga, iako je informaciju o poziciji unarnog operatora moguće sačuvati u opštem AST, to u trenutnoj verziji nije urađeno radi simplifikacije modela opšteg AST.

Binarni aritmetički operatori figurišu u aritmetičkim izrazima, i mogu biti operatori koji odgovaraju matematičkim operacijama ali i bitovski binarni operatori. Binarni relacioni operatori figurišu u relacionim izrazima, i mogu biti operatori poretka i poređenja po jednakosti ili različitosti. Binarni logički operatori figurišu u izrazima sa istinitosnim vrednostima.

Slično kao i za unarne operatore, moguće je apstraktno posmatrati sve binarne operatore tako što ih definišemo kao strukturu koja uključuje binarnu funkciju koja transformiše 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 istinitosnog tipa; dok logički binarni operatori zahtevaju da argumenti i povratna vrednost budu istinitosnog tipa.

Za operatore dodele se može primeniti isti princip kao i za aritmetičke binarne izraze, uzimajući u obzir da je dodela zapravo sporedni efekat i da se posmatra kao izraz čija je vrednost jednaka vrednosti izraza sa desne strane operatora dodele. Neki programski jezici dozvoljavaju i složene operatore dodele, koji se mogu razložiti na više jednostavnijih izraza.

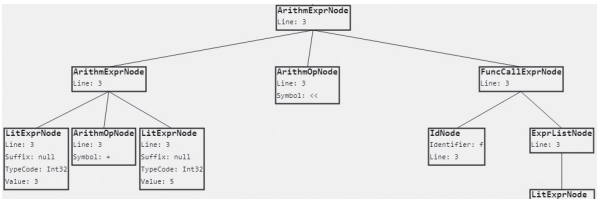
ČVOROVI IZRAZA

Najjednostavnije vrste izraza predstavljaju *literali*. Literali mogu biti brojevne konstante, karakterske konstante ili konstantne niske karaktera. Literale može pratiti i sufiks (najčešće za brojevne literale), koji određuje tip literala u slučajevima gde postoji dvosmislenost, npr., dodavanjem sufiksa L na celobrojni literal u programskom jeziku C, naglašavamo da je literal tipa *long*. Pošto nezanemarljiv broj 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 ovakve vrednosti, ali i potencijalno druge vrednosti koje označavaju praznu vrednost, postoji poseban tip literala, nazvan *NULL* literal.

Osim literala, validan izraz predstavljaju samostalne promenljive, u kom slučaju je vrednost izraza trenutna vrednost te promenljive. Slično važi za indeksni pristup strukture podataka, pri čemu je potrebno navesti izraz čija vrednost određuje indeks (to ne mora biti 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, bez semantičkih provera, pretpostavljamo da funkcija ima povratnu vrednost koja će se iskoristiti nakon poziva funkcije. Iako je u ovom radu akcenat na imperativnoj paradigmi, neki funkcionalni koncepti su implicitno podržani zbog načina na koji je implementiran opšti AST — npr. operatori kompozicije funkcija i anonimne funkcije.

Slika 6 prikazuje izgleda opšteg AST jednog izraza. 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 ostati ista.



Slika 6: Opšti AST za izraz (3 + 5) << f(4).

OSTALI ČVOROVI

Izvorni kôd se obično sastoji iz više izvornih fajlova, koji se u opštem AST modeluju kao *komponente izvornog koda* uz odgovarajući tip čvora. Podržano je i reprezentovanje čestih dopunskih elemenata kao što su definicija ili uključivanje drugih komponenti izvornog koda, biblioteka, paketa, ili prostora imena.

Programski jezici kao što su Java ili C# podržavaju *anotacije* odnosno *atribute* — metapodatke koji daju dodatni kontekst i informacije prevodiocu ili okruženju za izvršavanje programa. U opštem AST, anotacije i atributi se predstavljaju čvorom *tag*, apstraktnim metapodatkom koji, kao i anotacije i atributi, može kvalifikovati druge opšte AST čvorove.

Skup tipova čvorova opšteg AST takođe uključuje čvorove za predstavljanje koncepata prisutnih u drugim programskim paradigmama. Tako se koncepti iz funkcionalne paradigme, kao što su npr. *anonimne* (*lambda)* funkcije, mogu predstaviti u opštem AST. Dinamička tipiziranost prisutna u skript paradigmi se može predstaviti kroz umetanje eksplicitnih deklaracija za odgovarajuće promenljive, uz korišćenje nad-tipa *object* kao markera za dinamički tip.

IMPLEMENTACIJA, EVALUACIJA I UPOTREBA

Implementacija opšte AST apstrakcije za imperativne jezike je dostupna u okviru infrastrukture LINVAST [14], koja predstavlja nadogradnju inicijalnog prototipa pod imenom LICC [31, 32]. Infrastruktura LINVAST se sastoji od skupa biblioteka za rad sa opštim AST, kao i skupa alata za generisanje, serijalizaciju i vizualizaciju opštih AST. Projekat LINVAST je u potpunosti implementiran u programskom jeziku C#.

Evaluacija projekta LINVAST je vršena na dva nivoa — jediničnom (mikro) nivou i sistemskom (makro) nivou. Evaluacija na jediničnom nivou predstavlja jedinično testiranje LINVAST graditelja zasebno za svaki programski jezik, zarad garancije korektnosti procesa apstrahovanja. Jedinični testovi se izvode neprekidno tokom razvoja projekta radi sprečavanja regresija. Evaluacija na sistemskom nivou predstavlja korišćenje projekta LINVAST nad popularnim projektima otvorenog koda.

Infrastruktura LINVAST pruža jednostavni i nadogradivi interfejs za kreiranje *graditelja* koji apstrahuju ANTLR stabla parsiranja proizvoljnog programskog jezika. Posao graditelja je da obiđe stablo parsiranja i kao rezultat generiše opšti AST. LINVAST pruža bibliotečke funkcije koje implementiraju česte operacije i time olakšavaju posao graditelja (npr. parsiranje i evaluacija izraza).

POKRIVENOST GRAMATIKA PROGRAMSKIH JEZIKA

Tabela 1 prikazuje rezultate jediničnog testiranja LINVAST graditelja za programske jezike C, Go, Java i Lua. Kompletan spisak pravila koja sačinjavaju gramatike se može pronaći u zvaničnoj ANTLR kolekciji gramatika [33]. Broj pravila u gramatici odgovarajućeg programskog jezika je prikazan u drugoj koloni, a broj implementiranih pravila u trećoj koloni. Četvrta kolona prikazuje procenat implementiranih pravila u odnosu na broj ukupnih pravila, dok je broj testiranih pravila je prikazan u poslednjoj koloni.

Pravilo se smatra implementiranim ukoliko su sve grane pravila implementirane. Treba napomenuti da se neka neimplementirana C99 pravila odnose na GCC ekstenzije i atribute. Broj testiranih pravila se odnosi na eksplicitno testirana pravila za koja postoje zasebni ciljani testovi. Neka pravila koja nisu pokrivena zasebnim testovima, kao na primer pravila za aritmetičke ili logičke izraze, su dodatno testirana evaluatorom izraza.

Gramatika	Broj pravila	Implementiranih pravila	Implementirano (%)	Testirano pravila
C99	125	108	86.4	95
Go	101	61	60.4	50
Java 11	107	80	74.77	72
Lua	50	41	82	25

Tabela 1: Jedinično testiranje LINVAST graditelja. Metrika uspeha je procenat pokrivenosti gramatičkih pravila. Gramatike su preuzete iz zvanične ANTLR kolekcije gramatika [33].

EVALUACIJA NAD PROJEKTIMA OTVORENOG KODA

Tabela 2 predstavlja rezultate sistemskog testiranja LINVAST Java graditelja. Testirani su popularni projekti otvorenog koda, sa desetinama hiljada fajlova i milionima linija izvornog koda. Svaka jedinica izvornog koda se zasebno prevodi u odgovarajući opšti AST. Tokom procesa izgradnje opšteg AST, meri se broj linija koda koje su parsirane i prevedene u opšti AST. Pritom, ukoliko ne uspe prevođenje kompilacione jedinice, sve linije izvornog koda koje sačinjavaju tu kompilacionu jedinicu se tretiraju kao neuspeh.

Projekat	Uspešno parsirano		Neuspešno parsirano		Procenat uspeha
	Fajlova	Linija koda	Fajlova	Linija koda	
antlr4	513	73577	8	1917	**98.5**
graal	11155	2012335	1097	398405	**91.2**
jdk11	41892	8212099	6978	1121008	**85.7**

Tabela 2: Sistemsko testiranje LINVAST Java graditelja nad projektima otvorenog koda (dostupno na https://github.com/LINVAST/eval). Metrika uspeha je odnos uspešno parsiranih Java fajlova u odnosu na ukupan broj Java fajlova u projektu. U slučaju neuspeha, čitav fajl (sve njegove linije) se tretira kao neuspešno parsiran.

UPOTREBA INFRASTRUKTURE LINVAST

Da bi LINVAST podržao novi programski jezik, potrebno je napisati ANTLR gramatiku jezika, ukoliko ona nije već dostupna [33]. Zatim, neophodno je generisati ANTLR parser za datu gramatiku, i implementirati posetioca stabla parsiranja. Primer ovog procesa je dostupan u okviru infrastrukture LINVAST kroz podršku za *pseudojezik* — programski jezik sintaksno sličan pseudokodu. Gramatika pseudojezika, iako jednostavna, sadrži koncepte modernih proceduralnih jezika — uslovne skokove, grananja, petlje, procedure, funkcije i slično. Graditelj opšteg AST za pseudojezik pokriva sva gramatička pravila i implementiran je u 307 linija koda, što pokazuje lakoću dodavanja podrške za nove programske jezike u okviru infrastrukture LINVAST. LINVAST trenutno implementira graditelje za programske jezike C (standard 99), Java (verzija 11), Go, i Lua.

Izlaz LINVAST alata za generisanje opštih AST je serijalizovan opšti AST u JSON format. Rezultujući JSON može biti generisan u proširenoj ili kompaktnoj formi. Slika 7 prikazuje primer JSON izlaza alata LINVAST. Opšti AST tako može biti ulaz za interne ali i eksterne alate koji rade nad opštim sintaksičkim stablima. Kao primer implementacije takvog alata, u okviru infrastrukture LINVAST je dostupan alat *LINVisualizer* za grafičku vizualizaciju i bolje razumevanje strukture opštih AST. Primer izlaza alata *LINVisualizer* se može videti na slikama 3 i 6.

```
…

{
  “NodeType”: “Func-
DefNode”,
  “Line”: 8,
  “Children”: [
    {
        “Modifiers”: {
          “AccessModi-
fiers”: “Unspecified”,
          “Qualifi-
erFlags”: “None”
      },
        “TypeName”:
“int”,
        “NodeType”:
“DeclSpecsNode”,
        “Line”: 8,
        “Children”: []
    },
    {
        “Pointer”:
false,
        “NodeType”:
“FuncDeclNode”,
        “Line”: 8,
        “Children”: [
          {
          “Identifier”:
“main”,
          “NodeType”:
“IdNode”,
          “Line”: 8,
          “Children”:
[]
          }
        ]
    },
    {
        “NodeType”:
“BlockStatNode”,
        “Line”: 10,
        “Children”: [
 …
```

```
{“Name”:null,”NodeType”:”Sour
ceNode”,”Line”:1,”Children”:[
{“NodeType”:”FuncDefNode”,”Li
ne”:1,”Children”:[{“Modifiers
”:{“AccessModifiers”:”Unspecif
ied”,”QualifierFlags”:”None”},
”TypeName”:”object”,”NodeType
”:”DeclSpecsNode”,”Line”:1,”C
hildren”:[]},{“Pointer”:false
,”NodeType”:”FuncDeclNode”,”L
ine”:1,”Children”:[{“Identifie
r”:”fact”,”NodeType”:”IdNode”
,”Line”:1,”Children”:[]},{“Is
Variadic”:false,”NodeType”:”F
uncParamsNode”,”Line”:1,”Chil
dren”:[{“NodeType”:”FuncParam
Node”,”Line”:1,”Children”:[{“
Modifiers”:{“AccessModifiers”:”
Unspecified”,”QualifierFlags”:”
None”},”TypeName”:”object”,”N
odeType”:”DeclSpecsNode”,”Lin
e”:1,”Children”:[]},{“Pointer
”:false,”NodeType”:”VarDeclNo
de”,”Line”:1,”Children”:[{“Id
entifier”:”n”,”NodeType”:”IdNo
de”,”Line”:1,”Children”:[]}]}
]}]}]},{“NodeType”:”BlockStat
Node”,”Line”:2,”Children”:[{“
NodeType”:”IfStatNode”,”Line”
:2,”Children”:[{“NodeType”:”R
elExprNode”,”Line”:2,”Childre
n”:[{“Identifier”:”n”,”NodeTyp
e”:”IdNode”,”Line”:2,”Childre
n”:[]},{“Symbol”:”==”,”NodeTy
pe”:”RelOpNode”,”Line”:2,”Chi
ldren”:[]},{“Value”:0,”Suffix”
:null,”TypeCode”:”Int32”,”Nod
eType”:”LitExprNode”,”Line”:2
,”Children”:[]}]},{“NodeType”
:”BlockStatNode”,”Line”:3,”Ch
ildren”:[{“Type”:”Return”,”N
odeType”:”JumpStatNode”,”Line
”:3,”Children”:[{“NodeType
…
```

Slika 7: Deo serijalizovanog opšteg AST u JSON format proširene forme (levo) i kompaktne forme (desno), dobijen kao izlaz alata LINVAST.

DISKUSIJA

Prilikom dizajna modela skupa čvorova opšteg AST su donete informisane odluke koje prioritizuju minimizaciju modela. Informacije koje se gube apstrahovanjem je moguće zadržati po cenu komplikacije modela. Razvoj modela skupa čvorova opšteg AST je vođen činjenicom da je minimalni model lako proširiti bez izmena programa koji ga koriste, dok obratno nije jednostavno i obično zahteva period stabilizacije dok korisnici adaptiraju izmene modela.

Evaluacija LINVAST graditelja kroz jedinične i sistemske testove pokazuje da graditelji:

- pokrivaju veliki podskup pravila podržanih programskih jezika,
- garantuju stabilnost infrastrukture tokom razvoja i sprečavaju pojavu regresija
- mogu sa velikom dozom uspešnosti parsirati izvorni kod modernih projekata sa više miliona linija koda kroz desetine hiljada izvornih fajlova.

U ovom radu su pomenuti i drugi projekti koji imaju sličan cilj kao i LINVAST — unifikaciju AST modela. Projekat *UnifikovaniAST* koristi automatizovanu transformaciju apstraktnih sintaksičkih stabala (*Astranaut*) [25] kroz domenski-specifičan jezik (DSL) za opis i transformacije sintaksičkih stabala. Za razliku od LINVAST čvorova, DSL za opis transformacija ne garantuje da će se isti konstrukti u različitim programskim jezicima mapirati u isti unifikovani konstrukt. Druga rešenja kao npr. modul *UAST* zajednice JetBrains IntelliJ su implementirana samo za jezike koji se izvršavaju na JVM.

ZAKLJUČAK

U ovom radu je predstavljen model opšte AST apstrakcije za imperativne programske jezike i njegova implementacija u okviru infrastrukture LINVAST za apstrahovanje izvornog koda pisanog u programskim jezicima C, Lua, Java i Go. Prikazana je evaluacija na jediničnom nivou kroz ekstenzivne testove, i sistemskom nivou nad popularnim projektima otvorenog koda. Pokazano je da infrastruktura LINVAST može da sa velikom dozom uspešnosti parsira izvorni kôd programa, što podržava činjenica da LINVAST biblioteka ima skoro deset hiljada preuzimanja.

Budući rad obuhvata povećanje pokrivenosti za trenutno podržane programske jezike, i implementaciju LINVAST graditelja za druge popularne programske jezike kao što su Python, C++, i drugi. Planirana je implementacija alata za statičku analizu koji koriste LINVAST za implementaciju jezički-invarijantnih analizatora.

LITERATURA

[1] Aho, Sethi, Ravi Sethi, and D. Jeffrey. “Ullman, Compilers: Principles, Techniques, and Tools.” (1986): 63-64.

[2] Cooper, Keith D., and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022.

[3] Meqdadi, Omar, and Shadi Aljawarneh. “A study of code change patterns for adaptive maintenance with AST analysis.” *International Journal of Electrical and Computer Engineering* 10.3 (2020): 2719.

[4] Martinez, Matias, Laurence Duchien, and Martin Monperrus. "Automatically extracting instances of code change patterns with AST analysis." *2013 IEEE international conference on software maintenance*. IEEE, 2013.
[5] Crew, Roger F. "ASTLOG: A Language for Examining Abstract Syntax Trees." *DSL*. Vol. 97. No. 18. 1997.
[6] Neamtiu, Iulian, Jeffrey S. Foster, and Michael Hicks. "Understanding source code evolution using abstract syntax tree matching." *Proceedings of the 2005 international workshop on Mining software repositories*. 2005.
[7] Kolachina, Prasanth, and Aarne Ranta. "From abstract syntax to universal dependencies." *Linguistic Issues in Language Technology* 13 (2016).
[8] Baxter, Ira D., et al. "Clone detection using abstract syntax trees." *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998.
[9] Koschke, Rainer, Raimar Falke, and Pierre Frenzel. "Clone detection using abstract syntax suffix trees." *2006 13th Working conference on reverse engineering*. IEEE, 2006.
[10] Fluri, Beat, et al. "Change distilling: Tree differencing for fine-grained source code change extraction." *IEEE Transactions on Software Engineering* 33.11 (2007): 725-743.
[11] C Abstract Syntax Tree (CAST) Representation. 2025. URL: https://www-old.cs.utah.edu/flux/flick/current/doc/guts/gutsch6.html
[12] Eclipse JDT AST. URL: https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/
[13] Parr, Terence J., and Russell W. Quong. "ANTLR: A predicated-LL (k) parser generator." *Software: Practice and Experience* 25.7 (1995): 789-810.
[14] LINVAST. 2019, accessed: 2025-10-08. URL: https://github.com/LINVAST
[15] LINVAST — NuGet package library, accessed: 2025-10-08. URL: https://www.nuget.org/packages/LINVAST
[16] GNU Bison, accessed: 2025-10-08. URL: https://www.gnu.org/software/bison/
[17] Parr, Terence, and Kathleen Fisher. "LL (*) the foundation of the ANTLR parser generator." *ACM Sigplan Notices* 46.6 (2011): 425-436.
[18] Koenig, Dierk, et al. "Groovy in action". Manning Publications Co., 2007.
[19] Juneau, Josh, et al. "The definitive guide to Jython: Python for the Java platform". Apress, 2010.
[20] Apache Cassandra, accessed: 2025-10-08. URL: https://cassandra.apache.org
[21] X (Twitter). "Advanced search query language", accessed: 2025-07-05. URL: https://x.com/search-advanced
[22] Salesforce Apex, accessed: 2025-10-08. URL: https://www.apexhours.com/introduction-to-apex
[23] ANTLR Lab, accessed: 2025-10-08. URL: http://lab.antlr.org/
[24] Code Quality Foundation. "Unified Abstract Syntax Tree (UAST)", accessed: 2025-10-08. URL: https://github.com/cqfn/uast
[25] Code Quality Foundation., "Astranaut", accessed: 2025-10-08. URL: https://github.com/cqfn/astranaut
[26] JetBrains. "UAST – Unified Abstract Syntax Tree", accessed: 2025-10-08. URL: https://plugins.jetbrains.com/docs/intellij/uast.html
[27] Unified Collective. "Unified JS", accessed: 2025-10-08. URL: https://unifiedjs.com/
[28] Unified Collective. "Universal Syntax Tree", accessed: 2025-10-08. URL: https://github.com/syntax-tree
[29] Coala development group. "Coala", accessed: 2025-10-08. URL: https://github.com/coala
[30] Coala development group. "CoAST", accessed: 2025-10-08. URL: https://github.com/coala/coAST
[31] Ristović, Ivan. "Jezički invarijantna provera semantičke ekvivalentnosti strukturno sličnih segmenata imperativnog koda." (master rad, 2020)
[32] "LICC." 2020, accessed: 2025-10-08 URL: https://github.com/ivan-ristovic/LICC/
[33] ANTRL4 Grammars, accessed: 2025-10-08. URL: https://github.com/antlr/grammars-v4

mast. inf. Ivan Ristović, student doktorskih akademskih studija smera Informatika na Matematičkom fakultetu, Univerziteta u Beogradu
Kontakt: ivan.ristovic@matf.bg.ac.rs
Oblasti interesovanja: verifikacija softvera, softversko inženjerstvo, implementacije programskih jezika