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:

prof. dr Filip Marić
Univerzitet u Beogradu, Matematički fakultet

doc. dr Milan Banković
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Zahvaljujem se mentoru doc. Mileni Vujošević Janičić na inspiraciji, pomoći i savetima.

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 sintaksičke analize (skr. rasčlanjavanja, odnosno parsiranja) ulaznog programa i predstavlja osnovu za semantičku analizu koda i druge kasnije faze kompilacije. 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 sintaksičko 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. Ovaj rad opisuje opštu AST reprezentaciju za imperativne programske jezike sa ciljem njene primene u analizi semantičke ekvivalentnosti implementacija algoritama u različitim programskim jezicima. Koncept opšteg AST i semantičkih upoređivača je pružen kroz aplikaciju LICC, implementiranu u programskom jeziku C# koristeći ANTLR4 generator parsera. LICC dozvoljava kreiranje apstrakcije na osnovu zadate gramatike imperativnog programskog jezika, pri čemu je koncept prikazan na primeru jednostavnog pseudokoda, kao i na programskim jezicima C i Lua.

Ključne reči: Apstraktno sintaksičko stablo, AST, ANTLR, opšta AST apstrakcija, semantička ekvivalentnost imperativnih segmenata koda

Sadržaj

Glava 1

Uvod

Apstraktno sintaksičko stablo (engl. *abstract syntax tree*, skr. *AST*) programa ima značajnu ulogu u procesu kreiranja izvršivog programa od izvornog koda. AST nastaje u fazi sintaksičke analize (ili *fazi rasčlanjavanja*) kao rezultat apstrahovanja stabla sintaksičke analize dobijenog od strane sintaksičkog analizatora (skr. *parsera*). Parser čita izvorni kôd i pokušava da u njemu pronađe primene određenih pravila jezika čiji je kôd dizajniran da rasčlani. Svaki programski jezik ima specifična sintaksna pravila pa su stoga i skupovi pravila (tzv. *gramatike*) programskih jezika raznorodni, što se zatim prenosi i na generisana stabla sintaksičke analize. Stablo sintaksičke analize se apstrahuje tako što se iz njega izvuku samo bitne sintaksičke a uklone neke tehničke informacije.

Ovakva apstrakcija se najpre koristi u semantičkoj analizi programa koju vrši prevodilac nakon faze sintaksičke analize i provere sintaksičke ispravnosti koda. Ukoliko program prođe semantičke provere, prelazi se na prevođenje u međureprezentaciju i fazu optimizacije. Nakon faze optimizacije sledi generisanje asemblerskog koda koji se zatim prevodi u mašinski kôd.

AST, zbog svoje uloge u semantičkoj analizi, može poslužiti i za analizu programa pre samog prevođenja, kroz proces poznat pod nazivom *statička analiza*. Posmatranje programa kroz AST pruža mogućnost za poređenje dva programa na apstraktnom nivou. Jedna primena ove ideje u okviru statičke analize može biti provera semantičke ekvivalentnosti. Provera semantičke ekvivalentnosti dva programa je neodlučiv problem u opštem slučaju, međutim pod određenim pretpostavkama koje pojednostavljuju problem moguće je dizajnirati algoritme koji daju smislene rezultate u praksi. Jedna od često korišćenih pretpostavki je pretpostavka sličnosti strukture dva programa. Interesantno, provera da li dva programa

zadovoljavaju ovu premisu se može proveriti posmatranjem izgleda i sličnosti u strukturi na apstraktnom nivou — problem koji se može rešiti primenom algoritama za rad sa stablima jer je u pitanju AST (ali i grafovima uopšte, jer je stablo specijalizacija grafa).

Iako je AST reprezentacija neophodna za kompilaciju i primenjiva za neke druge vrste problema, ipak je specifična za konkretni programski jezik s obzirom da nastaje od stabla sintaksičke analize koje je usko vezano za gramatiku konkretnog programskog jezika. Motivacija za ovaj rad dolazi od nepostojanja opštih apstrakcija sintaksičkih stabala koje bi se mogle koristiti za analizu programa napisanih u različitim programskim jezicima. Iako je broj programskih jezika danas veoma veliki, u okviru iste programske paradigme jezici moraju implementirati koncepte koji su potrebni da bi se programiralo u toj paradigmi i ta zajednička svojstva se mogu iskoristiti za formiranje zajedničke apstrakcije.

U ovom radu će biti predstavljena opšta AST apstrakcija za imperativne programske jezike, sa ciljem da se omogući zajednička apstraktna reprezentacija velikog broja imperativnih jezika, pa čak i onih koji pripadaju skript paradigmi. Njena upotreba će biti demonstrirana na problemu semantičke ekvivalentnosti dobijenih apstrakcija kroz naivni algoritam poređenja simboličkih promenljivih. Štaviše, na apstraktnom nivou nije važno od kog se programskog jezika dobio AST, što može imati primenu u procesu migracije na nove tehnologije. Na slici 1.1 se mogu videti primeri dve funkcije pisane u različitim programskim jezicima koje se mogu apstrahovati tako da imaju skoro identičan AST. Razlike koje moraju postojati u ovom slučaju su tipovi argumenata — u drugoj funkciji nije zagarantovano da argumenti (ali i povratna vrednost) moraju biti tipa `int`. Treba napomenuti da ovakvo apstrahovanje često dovodi do gubitka informacija — u dobijenim apstrakcijama funkcija sa slike 1.1 nisu poznate konkretne vrste petlji od kojih se dobila apstraktna petlja (u opštem slučaju je moguće sve vrste petlji svesti na jednu).

Naravno, semantička ekvivalentnost se ne mora zasnivati na apstrahovanju programa, već se takođe često rešava spuštanjem na nivo međukoda između višeg programskog jezika i asemblera. U nekim slučajevima se može ići i do asemblera pa i mašinskog jezika. Ukoliko bi se posmatrali asemblerski ili mašinski kôd, vršilo bi se poređenje kodova prilagođenih određenoj arhitekturi procesora. Neki moderni radni okviri kao što je *Microsoft .NET* radni okvir, imaju kao svoju komponentu i virtualnu mašinu na kojoj se izvršavaju programi koji koriste taj radni okvir, bez obzira na programski jezik u kojem su ti programi napisani. Virtualna mašina

```c
void array_sum(int[] arr, int n) {
    int sum = 0, i = 0;
    while (i < n) {
        int v = arr[i];
        sum += arr[i];
        i++;
    }
    return sum;
}
```

```lua
function array_sum(arr, n)
    local sum = 0
    for i,v in ipairs(arr) do
        sum = sum + v
    end
    return sum
end
```

Slika 1.1: Segmenti koda pisani u različitim programskim jezicima (C gore, i Lua dole) koji se mogu apstrahovati tako da imaju skoro identični AST.

prevodi međureprezentacije programa dobijene od prevodioca u mašinski jezik i izvršava ih. Međutim, iako je međukod isti, AST programa pisanih u različitim jezicima i dalje nije. U ovom radu je odabran pristup zasnovan na AST, s obzirom na važnosti i značaj apstraktnih sintaksičkih stabala, ali i zbog nedostatka opštih apstrakcija.

U poglavlju 2 će biti opisani relevantni pojmovi potrebni za razumevanje rada uz akcenat na apstraktnim sintaksičkim stablima i procesu njihovog dobijanja. 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 algoritma 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

Generisanje i korišćenje apstraktnih sintaksičkih stabala

U ovom poglavlju će biti opisani koncepti i alatke čije je razumevanje potrebno kako bi se razumeo proces kreiranja i poređenja opštih apstrakcija, kao i implementacije programa za njihovo kreiranje i poređenje. Umesto analize samog sadržaja izvornog koda analizira se *apstraktno sintaksičko stablo*, opisano u odeljku 2.1. Kako bi se od izvornog koda došlo do stabla sintaksičke analize (u daljem tekstu *stablo parsiranja*) a potom i do apstraktnog sintaksičkog stabla, koriste se leksički analizator (skr. *skaner* ili, u daljem tekstu *lekser*) i *sintaksički analizator* (u daljem tekstu *parser*). S obzirom da je cilj kreirati univerzalnu reprezentaciju, neophodno je kreirati leksere i parsere za razne programske jezike. Više reči o samom procesu dobijanja stabla parsiranja od izvornog koda i alatkama koje mogu da generišu leksere i parsere biće u odeljku 2.2, sa akcentom na alat *Another Tool For Language Recognition* [2], u daljem tekstu *ANTLR*. Da bi se dobijena stabla koristila, neophodno je poznavati *projektne obrasce* opisane u odeljku 2.3 koji se u ovom radu koriste za pružanje interfejsa obilaska stabala i izračunavanja vrednosti nad njima.

2.1 Apstraktna sintaksička stabla

Kako bi se kôd pisan u nekom programskom jeziku (*izvorna datoteka*) preveo u kôd koji će se izvršavati na nekoj mašini (*izvršiva datoteka*), prevodilac izvršava određene korake. Prvi korak je prepoznavanje gradivnih elemenata programskog jezika koji se nazivaju *lekseme*, nalik na prepoznavanje reči u rečenicama

govornog jezika, i raspodela leksema u grupe po funkciji (nalik na funkcije reči u govornom jeziku — subjekti, predikati, objekti itd.). Nakon prepozavanja leksema, proverava se da li niz prepoznatih leksema ispunjava pravila odnosno *sintaksu* programskog jezika, nalik na proveru da li se govorne rečenice sastoje od subjekta i predikata u određenom redosledu. Na kraju se proverava značenje odnosno *semantika* koda, nalik na proveru da li govorne rečenice, iako sintaksno pravilne, imaju smisla. Deo prevodioca koji određuje da li je izvorni kôd ispravno formiran u terminima leksike, sintakse i semantike se naziva *prednji deo* (engl. *front end*). Ukoliko je izvorni kôd ispravan, prednji deo kreira *međureprezentaciju* koda (engl. *intermediate representation*, u daljem tekstu *IR*) nad kojom se vrše optimizacije koda i koja se koristi za dalji proces prevođenja. Ukoliko to nije slučaj, prevođenje ne uspeva i programeru se daje poruka sa obrazloženjem zašto prevođenje nije uspelo [8].

Za potrebe ovog rada, što se procesa prevođenja tiče, dovoljno je poznavanje pomenutih procesa koje izvodi prednji deo, stoga neće biti reči o ostalim koracima u fazi prevođenja (generisanje IR, optimizovanje). Zainteresovani čitalac može više detalja pronaći u [1], [8] i [25].

Pretpostavimo da želimo da prevedemo izvorni kôd pisan u programskom jeziku C sa slike 2.1. Primetimo da postoji greška u datom kodu — simbol `c` koji se koristi u dodeli u liniji 7 će biti prepoznat kao identifikator koji ne odgovara nijednoj deklarisanoj promenljivoj — stoga ne možemo prevesti ovaj kôd. Ovo nije greška u sintaksi — izraz `a+c` je validan u programskom jeziku C. Problem će postati vidljiv 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 *sintaksičke greške*.

Pre nego što prednji kraj prevodioca dobije kôd koji treba da prevede u međureprezentaciju, vrši se *pretprocesiranje* od strane programa koji se naziva *pretprocesor*. U fazi pretprocesiranja se izvode operacije kao što su brisanje komentara ili zamena makroa u okviru konteksta[1] u jezicima kao što je C. Rezultat rada pretprocesora za izvorni kôd sa slike 2.1 se može videti na slici 2.2[2].

U toku faze leksičke analize, kako prevodilac ne bi radio nad sirovim karakte-

[1]Makroi se, na primer, neće zameniti unutar niski.

[2]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.

```c
#include<stdio.h>
#define T int

int main()
{
    T a, b;
    a = a + c;         // c nije deklarisano
    printf("%d", a);
    return 0;
}
```

Slika 2.1: Primer izvornog koda sa semantičkom greškom (C).

```c
# 1 "<stdin>"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4

...

extern int fprintf (FILE *__restrict __stream,
                    const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);

...

# 868 "/usr/include/stdio.h" 3 4
# 2 "<stdin>" 2
# 2 "<stdin>"

int main()
{
    int a, b;
    a = a + c;
    printf("%d", a);
    return 0;
}
```

Slika 2.2: Prikaz rezultata rada pretprocesora za izvorni kôd sa slike 2.1 — kôd iznad `main` funkcije je uključen iz `stdio.h` zaglavlja.

rima izvornog koda, potrebno je izvršiti transformaciju karaktera izvornog koda. Prevodilac ima u vidu elemente programskog jezika i šablone za njihovo prepoznavanje. Lekseme su one sekvence karaktera izvornog koda koje zadovoljavaju ove

šablone. Lekseme se dalje razvrstavaju u kategorije, tzv. *tokene* — ključne reči, operatore, promenljive itd. Proces dobijanja tokena od izvornog koda se naziva *tokenizacija*. Komponenta prednjeg dela koja vrši tokenizaciju se naziva *skener* ili *lekser*. Pojednostavljen primer šablona koje lekser pokušava da prepozna se mogu videti na slici 2.3. Primer izlaza leksera za izlaz pretprocesora sa slike 2.2 se može videti na slici 2.4. Svako pravilo se sastoji od imena pravila nakon čega posle simbola `:` dolaze definicije pravila razdovene simbolom `|` ukoliko ih ima više od jedne[3]. Simbol `;` označava kraj pravila. Definicija simbola u sebi može referisati na druga pravila ili koristiti *regularne izraze*[4] ili njihove specijalne simbole zadržavajući njihovu semantiku[5]. Ovaj format definisanja pravila koristi alat ANTLR4 za definisanje gramatika koje su čitljive i lake za pisanje.

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

Slika 2.3: Primer delimične definicije tokena za ime promenljive po C11 standardu.

Nakon faze skeniranja potrebno je proveriti da li niz pročitanih tokena zadovoljava sintaksu programskog jezika. Prevodilac stoga mora da uporedi strukturu koda sa unapred definisanom strukturom za određeni programski jezik što zahteva formalnu definiciju sintakse jezika. Programski jezik možemo posmatrati kao

[3]Ukoliko se simbol `|` nalazi unutar definicije pravila kao specijalan simbol, mora se pojaviti unutar zagrada i u tom slučaju ne predstavlja kraj trenutne definicije.

[4]Regularni izraz je sekvenca karaktera koja se često koristi da definiše šablon pretrage ili poklapanja. Formalno, regularni izraz definiše čitav regularni jezik.

[5]U regularnom izrazu `a?b+c*|d`, simbol `?` označava opciono pojavljivanje simbola `a`, simbol `+` označava jedno ili više pojavljivanja simbola `b`, a simbol `*` označava proizvoljan broj pojavljivanja simbola `c` — kombinacija simbola `?` i `+`, dok simbol `|` označava izbor — ili će se poklopiti izraz sa leve strane ili izraz sa desne strane. Dodatno, unutar para srednjih zagrada (`[ ]`) možemo navesti spisak karaktera koji ulaze u obzir prilikom poklapanja — skraćena verzija ulančavanja simbola `|`.

```
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 2.4: Proces tokenizacije koda sa slike 2.2 (generisano pomoću kompajlera `clang` [7]).

skup *pravila* pod imenom *gramatika* [17]. Na slici 2.5 se može videti deo gramatike programskog jezika C. Proces provere zadovoljenosti sintaksnih pravila se naziva *parsiranje*, a komponenta prednjeg dela koja vrši parsiranje se naziva *parser*.[6]

Parser, imajući u vidu gramatiku jezika, kreira *stablo parsiranja* (eng. *parse tree* ili *derivation tree*) [17]. Takvo stablo i dalje sadrži sve relevantne informacije o izvornom kodu. Vizuelni prikaz rada parsera za C11 gramatiku i izvorni kod sa slike 2.1 je dat na slici 2.6. Stablo parsiranja se koristi u narednim fazama prevođenja. Važno je napomenuti da, iako je rečeno da je stablo parsiranja rezultat

[6]Moderni kompilatori u toku svog izvršavanja nemaju odvojene faze skeniranja i parsiranja, već se skeniranje odvija paralelno sa fazom parsiranja iako su skener i parser zasebne komponente prednjeg dela kompilatora. Međutim, to nas ne sprečava da ispišemo tokene onda kada se oni prepoznaju, što to je demonstrirano na slici 2.4.

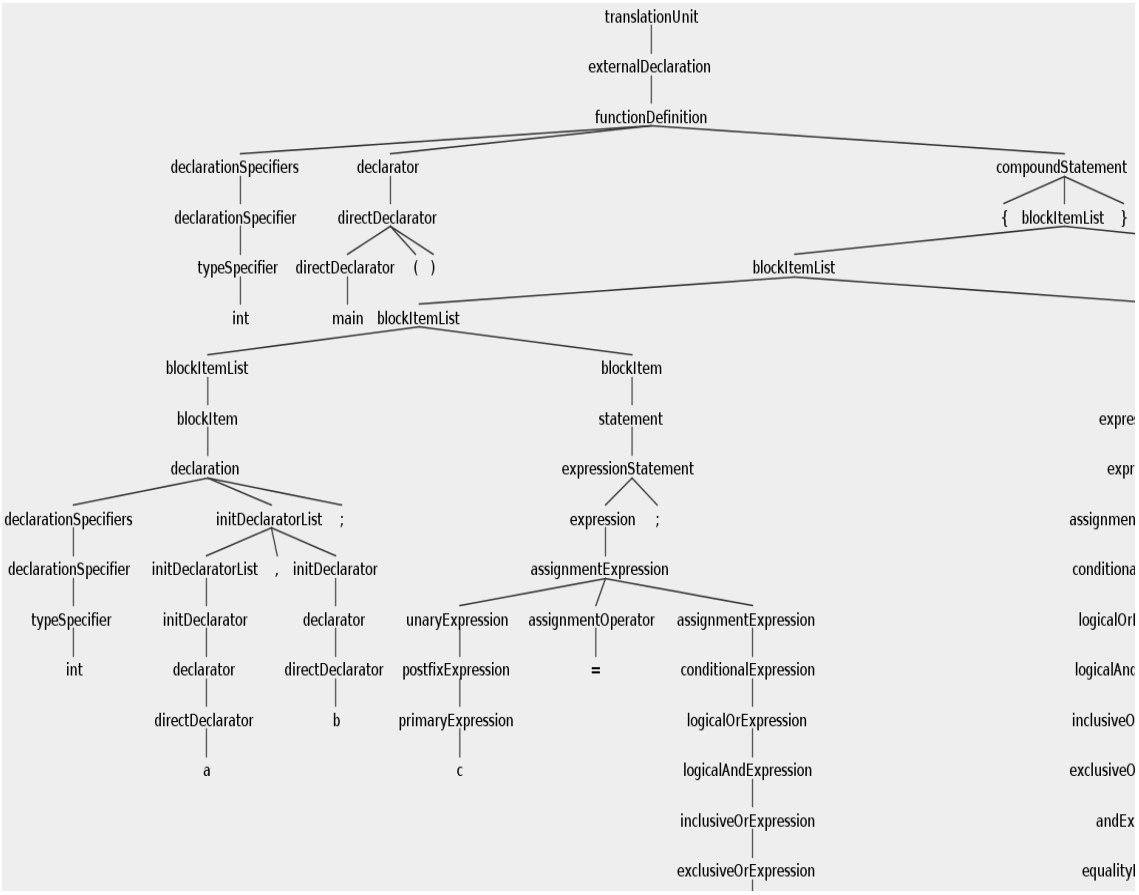
```
1  declarationList
2      :   declaration
3      |   declarationList declaration
4      ;
5  declaration
6      :   declarationSpecifiers initDeclaratorList ';'
7      |   declarationSpecifiers ';'
8      |   staticAssertDeclaration
9      ;
```

Slika 2.5: Prikaz para pravila iz gramatike programskog jezika C po standardu C11.

rada parsera, takvo stablo u memoriji tokom rada parsera nikada neće biti kreirano (osim ukoliko modifikujemo svrhu parsera da eksplicitno kreira takvo stablo, kao što je slučaj sa parserima koji će biti generisani kasnije u implementaciji), već služi samo kao formalizam za razumevanje rada parsera.

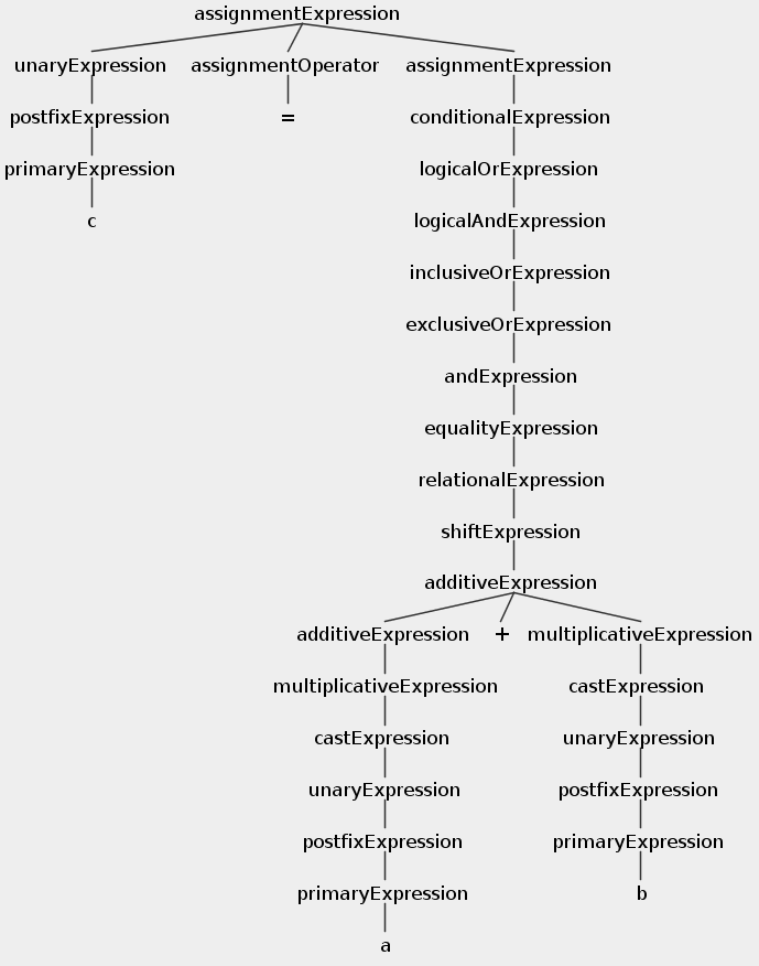
Stablo parsiranja sadrži sve informacije potrebne u fazi parsiranja uključujući detalje korisne samo za parser prilikom provere ispunjenosti gramatičkih pravila. Sa druge strane, *apstraktno sintaksičko stablo* sadrži samo sintaksičku strukturu u jednostavnijoj formi. Na slici 2.7 se može videti koliko stablo parsiranja može biti kompleksno čak i za jednostavne aritmetičke izraze. Razlog kompleksnosti u ovom slučaju dolazi iz rekurzivnih pravila iz C11 gramatike. Parseru su sve ove informacije neophodne ali za naredne analize i proces prevođenja one nisu potrebne i zato se stablo parsiranja apstrahuje. Na primer, važna semantička odlika izraza `a+c` je da je to zbir vrednosti nekih promenljivih za razliku od pozicija delova ovog izraza u izvonom kodu. Na slici 2.8 se mogu videti različita apstraktna sintaksička stabla za pomenuti izraz, ali takođe i za malo složenije izraze. Podrazumeva se, naravno, da je ulaz već tokenizovan.

Uloga apstraktnog sintaksičkog stabla [24] je da pokaže semantiku strukture koda preko stabla. Kao što se vidi na slici 2.8, postoji određeni nivo slobode prilikom njegovog kreiranja. 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 za jednostavniju implementaciju čvorova izraza — ukoliko binarni izraz posmatramo kao apstrakciju, za implementaciju je lakše koristiti ovakav tip



Slika 2.6: Prikaz dela stabla parsiranja koje generiše parser kreiran od strane alata ANTLR4 [2] za izvorni kôd sa slike 2.2.

apstraktnih sintaksičkih stabala i stoga će biti korišćen kasnije u implementaciji opšte apstrakcije. Primetimo takođe da se u stablima 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 je apstraktno sintaksičko stablo apstrakcija stabla parsiranja, više semantički ekvivalentnih izraza može imati isto apstraktno sintaksičko stablo ali različito stablo parsiranja; na primer, ako razmatramo izraz `(a+5)-x/2` i izraz `a+5-(x/2)`.



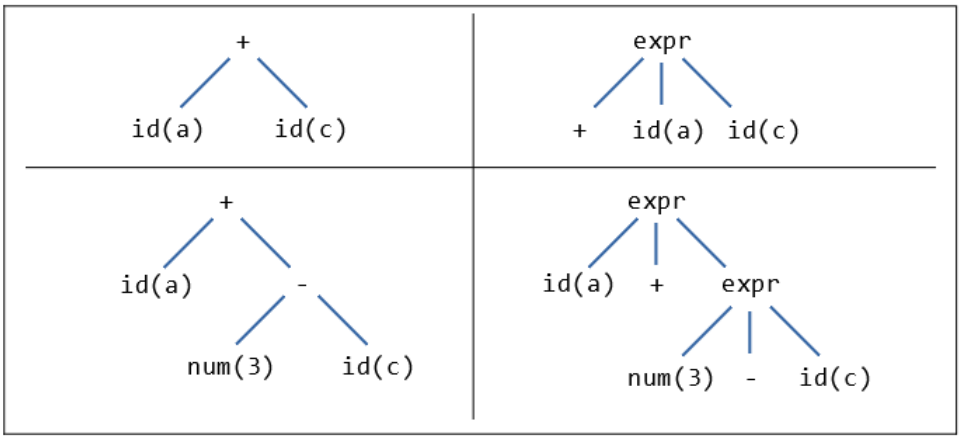
Slika 2.7: Prikaz kompleksnosti stabla parsiranja za izraz `a+c` u C11 gramatici.

2.2 Parsiranje gramatika programskih jezika

Ukoliko 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 nekom specifičnom programskom jeziku?

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 2.8: AST varijante bez regularnosti (levo) i sa regularnošću (desno) za izraze `a+c` (gore) i `a+(3-c)` (dole).

Lex i Flex

Lex [19] je program koji generiše leksere. Danas se više koristi *flex* [10], kreiran kao alternativa *lex*-u, s obzirom da je i do dva puta brži od *lex*-a, koristi manje memorije nego *lex*, i vreme kompilacije leksera koje *flex* generiše je i do tri puta kraće nego kompilacija leksera koje generiše *lex*. Pošto *flex*, isto kao i *lex*, generiše samo leksere, najčešće se koristi u kombinaciji sa drugim alatkama koje mogu da generišu parsere, kao što su npr. *GNU Bison*, *YACC* ili *BYACC*.

YACC i BYACC

YACC [19] je program koji generiše *LALR* [1] parser na osnovu gramatike date na ulazu zajedno sa akcijama koje će se izvršiti kada se određeno pravilo prepozna u izvornom kodu. *YACC* ne vrši leksičku analizu, stoga se obično koristi zajedno sa popularnim leksičkim analizatorima kao što su *lex* i *flex*.

Berkeley YACC, skraćeno *BYACC* [5], je generator parsera pisan po ANSI C standardu i otvorenog je koda. Posmatra se od strane mnogih kao *najbolja varijanta YACC-a* [19]. *BYACC* dozvoljava tzv. *reentrant* kôd — omogućava bezbedno konkurentno izvršavanje koda na način kompatibilan sa Bison-om i to je delom razlog njegove popularnosti.

GNU Bison

GNU Bison [15] je generator parsera i deo GNU projekta [16], često referisan samo kao *Bison*. *Bison* generiše parser na osnovu korisnički definisane kontekstno slobodne gramatike [17], upozoravajući pritom na dvosmislenosti prilikom parsiranja ili nemogućnost primene gramatičkih pravila. Generisani parser je najčešće C a ređe C++ program, mada se u vreme pisanja ovog rada eksperimentiše sa Java podrškom. Generisani kôd je u potpunosti prenosiv i ne zahteva specifične kompajlere. Bison može da, osim podrazumevanih *LALR(1)* [1] parsera, generiše i kanoničke *LR* [23], *IELR(1)* [9] i *GLR* [14] parsere.

ANTLR

Another Tool for Language Recognition, ili kraće *ANTLR* [2], je generator *LL(*)* [21] 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 daljem tekstu ANTLR4, u vreme pisanja ovog rada generiše parsere u narednim programskim jezicima: Java, C#, C++, JavaScript, Python, Swift i Go.[7]

Parseri generisani koristeći ANTLR4 koriste novu tehnologiju koja se naziva *Prilagodljiv LL(*)* (engl. *Adaptive LL(*)*) ili *ALL(*)* [22], 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. Ovaj pristup je takođe efikasniji zbog značajno manjeg prostora ulaznih sekvenci u parser.

Najbolji aspekt ANTLR-a je lakoća definisanja gramatičkih pravila koji opisuju sintaksičke konstrukte. Primer jednostavnog pravila za definisanje aritmetičkog izraza je dat na slici 2.9. Pošto izraz možemo definisati na više načina, pišemo alternative u definiciji pravila. Pravilo `exp` je levo rekurzivno jer barem jedna od njegovih alternativnih definicija referiše baš na pravilo `exp`. ANTLR4 automatski zamenjuje levo rekurzivna pravila u nerekurzivne ekvivalente. Jedini zahtev koji mora biti ispunjen je da leva rekurzija mora biti *neposredna* — pravila odmah moraju referisati na svoje ime u definiciji. *Posredna* leva rekurzija nije dozvoljena

[7] ANTLR verzije 4 je izabran u ovom radu zbog svoje popularnosti, jednostavnosti, intuitivnosti i podrške za mnoge moderne programske jezike. Verzija 4 je izabrana po preporuci autora ANTLR-a, na osnovu eksperimentalne analize brzine i pouzdanosti te verzije u odnosu na prethodnu.

— pravila ne smeju referisati drugo pravilo 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 2.9: Definicija uprošćenog aritmetičkog izraza po ANTLR4 gramatici koristeći neposrednu levu rekurziju.

2.3 Korišćenje generisanih stabala

Kako bi se stabla parsiranja i apstraktna sintaksička stabla mogla koristiti, potrebno je pružiti i uniformni interfejs za njihov obilazak. Postoje situacije kada se stablo obilazi sa ciljem izvršavanja operacija prilikom ulaska ili izlaska iz čvorova određenog tipa, ili pak sa ciljem izračunavanja neke konkretne vrednosti. Prilikom razvoja softvera se često nailazi na ovakve probleme i stoga su kreirana ponovno upotrebljiva rešenja za te probleme.

Projektni obrasci (engl. *design patterns* [13], drugačije nazvani i *projektni šabloni, uzorci*) predstavljaju opšte i ponovno upotrebljivo rešenje čestog problema, obično implementirani kroz koncepte objektno-orijentisanog programiranja. Svaki projektni obrazac ima četiri osnovna elementa:

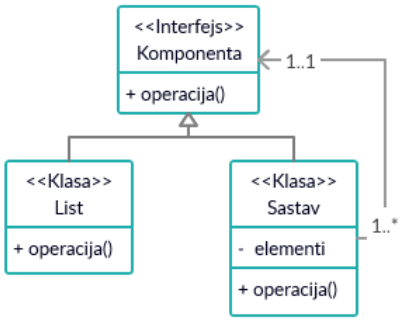
- ime — ukratko opisuje problem, rešenje i posledice,
- problem — opisuje slučaj u kome se obrazac koristi,
- rešenje — opisuje elemente dizajna i odnos tih elemenata,
- posledice — obuhvataju rezultate i ocene primena obrasca.

Projektne obrasce je moguće grupisati po situaciji u kojoj se mogu iskoristiti ili načinu na koji rešavaju zadati problem. Stoga je opšte prihvaćena podela na *gradivne obrasce* (engl. *creational patterns*), *strukturne obrasce* (engl. *structural patterns*) i *obrasce ponašanja* (engl. *behavioral patterns*).

Za potrebe ovog rada, projektni obrasci će se koristiti kao opšte prihvaćeno i programerski intuitivno rešenje određenih problema. Apstraktno sintaksičko stablo je primer upotrebe projektnog obrasca *sastav*. Takođe, u kontekstu stabala parsiranja i apstraktnih sintaksičkih stabala, obrasci *posmatrač* i *posetilac* su od velikog značaja jer pružaju interfejs za obilazak takvih stabala. Ovi obrasci, opisani u narednim odeljcima, se koriste od strane ANTLR alata. Takođe, s obzirom da su ovi obrasci opšte-prihvaćeno rešenje za pružanje interfejsa obilaska stabala, biće korišćeni i u implementaciji opšte apstrakcije. U nastavku će zbog opisanih razloga biti opisani samo obrasci sastav, posmatrač i posetilac, dok zainteresovani čitalac može pročitati više u [13].

Projektni obrazac „Sastav”

Projektni obrazac *Sastav* je strukturni obrazac koji opisuje grupu objekata koji se tretiraju na isti način kao i individualna instanca istog tipa. Uloga sastava je da objekte organizuje u stablolike strukture kako bi se reprezentovale hijerarhije u kojima su neki elementi delovi a neki su celine. Sastav omogućava uniformno posmatranje individualnih objekata i čitavog sastava. Na slici 2.10 se može videti UML dijagram [11] ovog obrasca. Za potrebe ovog rada, sastav će biti korišćen u implementaciji opšteg AST, s obzirom da je AST primer sastava.

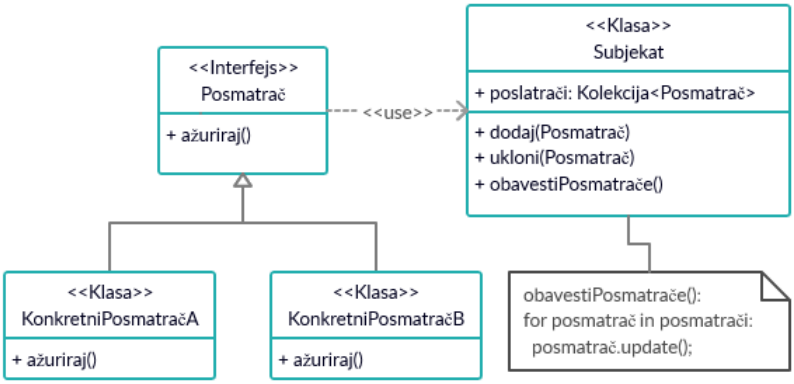


Slika 2.10: UML dijagram projektnog obrasca „Sastav”.

Projektni obrazac „Posmatrač”

Projektni obrazac *Posmatrač* je obrazac ponašanja 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

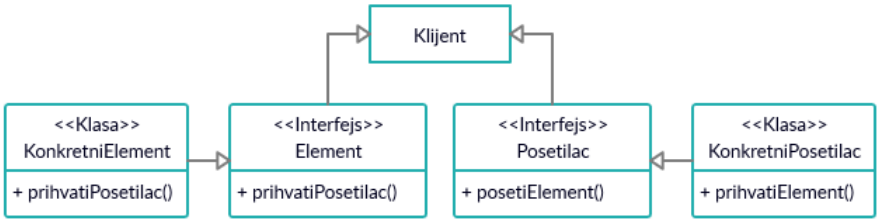
ažurirani. Posmatrač predstavlja *pogled* (engl. *View*) u MVC (engl. *Model-View-Controller*) arhitekturi. Na slici 2.11 se može videti UML dijagram [11] ovog obrasca. Za potrebe ovog rada, primer upotrebe ovog obrasca može biti obilazak stablolike kolekcije (recimo stabla parsiranja) i obaveštavanje o nailasku na čvorove određenih tipova. Te informacije se dalje mogu iskoristiti za izračunavanja nad pomenutom strukturom ili generisanje novih struktura (recimo AST).



Slika 2.11: UML dijagram projektnog obrasca „Posmatrač".

Projektni obrazac „Posetilac"

Projektni obrazac *Posetilac* je obrazac ponašanja 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 [11] ovog obrasca. Za potrebe ovog rada, primer upotrebe ovog obrasca može biti prikupljanje informacija o kolekciji stablolike strukture (recimo stablo parsiranja) i korišćenje istih za neko izračunavanje ili generisanje novih struktura (recimo AST).



Slika 2.12: UML dijagram projektnog obrasca „Posetilac".

Glava 3

Opis opšte AST apstrakcije za imperativne jezike

Broj različitih stilova programiranja je veliki i moderni programski jezici često ne pripadaju striktno jednom stilu. Zbog ove raznovrsnosti, apstrahovati svaki programski jezik je težak podvig, ako je uopšte i moguće dovesti fundamentalno različite koncepte na isti nivo apstrakcije. Stoga će u ovom radu biti apstrahovan samo imperativni stil programiranja, međutim zbog načina na koji je imperativno programiranje evoluiralo, moderni imperativni programski jezici često pružaju i proceduralne ali i funkcionalne koncepte. Stoga će u odeljku 3.1 biti reči o popularnim stilovima programiranja, dok će u odeljku 3.2 biti reči o apstrakciji imperativnog stila programiranja ali i njegovih derivata: strukturnog, proceduralnog, skript i OO stila.

3.1 Programske paradigme

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 efikasnosti izvršavanja. Način, ili stil programiranja se naziva *programska paradigma* [12]. Može se pokazati da sve što je rešivo putem jedne, može 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. Izračunavanja se vrše putem niza 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 u imperativnoj paradigmi se može videti na slici 3.1.

```
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 3.1: Primer koda pisanog po striktno imperativnoj paradigmi, bez naredbi kontrole toka i procedura odnosno funkcija.

Stariji programski jezici najčešće prate ovu paradigmu iz nekoliko razloga. Prvi je taj što imperativna paradigma najbliže oslikava samu mašinu na kojoj se program izvršava, pa je programer mnogo bliži mašini. Ova paradigma je bila veoma popularna zbog ranih ograničenja u hardveru i potrebe za efikasnim programima. Danas, zbog mnogo bržeg razvoja računara mnogo boljih performansi, efikasnost dobijena pisanjem koda u jezicima veoma bliskim mašini se sve manje uzima u obzir.

Imperativna paradigma svoje nedostatke. Naime, najveći problem je razumevanje i verifikovanje ispravnosti programa zbog postojanja propratnih efekata[1]. Stoga je zahtevno i pronalaženje grešaka u programima pisanim u imperativnoj

[1]Propratni efekti (promene stanja mašine, drugačije nazvani i *bočni efekti*, engl. *side effects*) 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.*

paradigmi. Pošto je kôd veoma niskog nivoa, obično je dozvoljen i direktan pristup memorijskim adresama putem *pokazivača*, što takođe otežava verifikaciju koda.

Danas se veliki broj drugih paradigmi svrstava pod imperativnu paradigmu i treba napomenuti da se u ovom odeljku pod terminom imperativna paradigma smatra striktno imperativni pristup bez naredbi kontrole toka i procedura odnosno funkcija. Van ovog odeljka, u opisu opšte apstrakcije ali i u ostatku rada, pod terminom imperativna paradigma se podrazumeva tradicionalna imperativna paradigma ali i njeni derivati kao što su strukturna ili skript paradigma.

Strukturna i proceduralna paradigma

Strukturna paradigma je vrsta imperativne paradigme gde se kontrola toka vrši putem niza naredbi, uslovnih grananja i petlji. Promenljive su obično lokalne za blok u kome su definisane, što određuje i njihov životni vek i vidljivost. Primer koda pisanog u strukturnoj paradigmi se može videti na slici 3.2. Danas je najpopularnija kombinacija strukturne paradigme sa *proceduralnom paradigmom*, baziranom 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 3.2: Primer koda pisanog u strukturnoj paradigmi.

Skript paradigma i njen odnos sa proceduralnom paradigmom

Tradicionalni programski jezici namenjeni za razvoj samostalnih aplikacija imaju za cilj da prime neku vrstu ulaza i generišu neku vrstu izlaza. Međutim, često je korisno napraviti kompoziciju već postojećih programa što je moguće uraditi i u tradicionalnim proceduralnim jezicima ali taj proces je spor i sklon greškama.

Skript jezici, iako se mogu koristiti za razvoj samostalnih aplikacija, se najčešće koriste za pisanje i izvršavanje *skriptova* — liste komandi koje mogu biti izvršene bez interakcije sa korisnikom. Ovo je pristup koji prate jezici kao što su `Python`, `Lua`, `Perl`, `bash` itd. Iako proceduralni, oni se razlikuju od klasičnih predstavnika proceduralne paradigme i njihove razlike su vremenom postale tolike da se skript jezici obično svrstavaju u zasebnu, *skript paradigmu*. Stoga će se u nastavku pod terminom *proceduralni jezik* smatrati tradicionalni proceduralni jezik, ukoliko nije naznačeno drugačije.

Kako hardver postaje moćniji, više se ceni vreme koje programer provede u procesu pisanja koda nego koliko je taj kôd 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. Skript jezici često preuzimaju koncepte iz OO i funkcionalne paradigme, kao i više programske koncepte kako bi proces kodiranja učinili što bržim. Na slici 3.3 se mogu uočiti navedene razlike u brzini izvršavanja.

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 najčešće zahtevaju eksplicitno definisanje tipa promenljive u kodu jer su većinom *statički tipizirani*, što znači da se tipovi promenljivih određuju u fazi prevođenja — posledica toga je da promenljive ne mogu menjati svoj tip tokom izvršavanja programa. Proces uvođenja imena za memorijsku lokaciju koja predstavlja mesto skladištenja vrednosti promenljive određenog tipa se 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 su, za razliku od proceduralnih, najčešće *dinamički tipizirani*, što znači da tipovi promenljivih zavise od trenutnih vrednosti promenljviih u fazi izvršavanja. Stoga je proces pisanja koda u dinamički tipiziranim jezicima brži jer se ne moraju navesti tipovi promenljivih[2]. Slično, parametri i povratne vrednosti funkcija takođe ne moraju biti fiksnog tipa.

[2]U nekim programskim jezicima koji su statički tipizirani (npr. Haskell) prevodilac može da zaključi tip na osnovu konteksta, stoga programer ne mora da eksplicitno navede tipove funkcija. Takođe, većina skript jezika dozvoljava eksplicitno definisanje tipa, ali to nije neophodno da bi se kôd preveo.

```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
```

```lua
k = 0
for i = 0, 1000000 do
    k += 1
end
```

```
$ time: 0.17s user 0.03s system 92% cpu 0.203 total
```

Slika 3.3: Primer koda pisanog u tradicionalnoj proceduralnoj paradigmi (gore, `C`) i u modernoj skript paradigmi (dole, `Lua`) kao i odgovarajuća vremena izvršavanja dobijena komandom `time`.

Kod statički tipiziranih proceduralnih jezika, mogu se koristiti strukture podataka koje omogućavaju brz pristup svojim elementima. To su na primer 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 (npr. `C`, `C++`, `Go`, `Rust`). Većina modernih proceduralnih jezika (npr. `Java`, `JavaScript`. `Python`, `Lua`) ne dozvoljava slobodan pristup memoriji, dok neki to dozvoljavaju ali 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. Za razliku od proceduralnih jezika gde su osnovne strukture podataka često kontinualni blokovi memorije sa proizvoljnim pristupom po indeksu, primarna struktura podataka kod skript jezika je najčešće *lista* — heterogena kolekcija

elemenata. Skript jezici uglavnom omogućavaju indeksni pristup elementima liste, pa programeru izgleda kao da radi nad običnim nizom (u nekim implementacijama liste su zaista implementirane preko niza). 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`[3], i za njih važe iste slobode kao i za liste. Kompleksnije strukture podataka uključuju skupove i rečnike ili *mape* (engl. *dictionaries, maps*) koji predstavljau kolekciju ključ-vrednost parova gde je dozvoljen indeksni pristup vrednosti para koristeći ključ. Razne implementacije mapa postoje i u proceduralnim jezicima, ali ključna razlika je ta što tipovi u skript jezicima nisu striktni — ključevi međusobno, 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. U programskom jeziku Lua, asocijativni nizovi (tzv. *tabele*) implementiraju sve ostale strukture podatake, pa i klase, što direktno odgovara ideji poređenja objekata sa rečnicima odnosno asocijativnim nizovima.

Skript programski jezici su skoro uvek interpretirani, iako se neki jezici mogu kompilirati po potrebi za efikasnije ponovno izvršavanje. S obzirom da efikasnost nije u glavnom planu, u skript jezicima nije dozvoljen slobodan pristup memoriji putem pokazivača ili referenci. Memorija za promenljive koje se više ne mogu referisati se obično automatski oslobađa kroz sakupljače otpada. Programer stoga ne mora da razmišlja o organizaciji niti da eksplicitno oslobađa memoriju.

OO paradigma i njen odnos sa proceduralnom paradigmom

Objektno-orijentisana paradigma (kraće *OOP* ili *OO paradigma*) 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*. Pošto objekti operišu nezavisno jedni od drugih, moguće je enkapsulirati ih u module koji sadrže lokalnu sredinu i metode dok se komunikacija sa objektom vrši prosleđivanjem poruka. Objekti su organizovani u klase, od kojih nasleđuju atribute i metode. OO paradigma omogućava ponovno korišćenje

[3]Torka (engl. *tuple*) je nepromenljiva, uređena, heterogena struktura podataka koja predstavlja sekvencu elemenata.

i jednostavnu proširivost koda. Primer koda pisanog u OO paradigmi se može videti na slici 3.4.

```
1  class Person
2  {
3      private string name;
4      private int wage;
5      private int income = 0;
6
7      Person(string name, int wage) {
8          this.name = name;
9          this.wage = wage;
10     }
11
12     public void Work(int hours) {
13         this.income += hours * this.wage;
14     }
15 }
16
17 Person p1 = new Person("John Doe", 30);
18 Person p2 = new Person("Dave Doe", 35);
19 p1.Work(7);
20 p2.Work(8);
```

Slika 3.4: Primer koda pisanog u OO paradigmi.

Iako se OOP posmatra kao zasebna paradigma, dosta modernih programskih jezika se često svrstava u OO paradigmu iako nisu nužno primarni predstavnici OO paradigme, već samo koriste neke koncepte OO paradigme kao što su klase ili nasleđivanje. Jedan od primera je i programski jezik `Python` koji, iako svrstan u skript paradigmu, pruža i OO koncepte kao što su klase, metode i nasleđivanje. Razlog za ovo je najviše prednost koje OO paradigma pruža ukoliko se radi na velikim programima, ali i taj što implementacija metoda OO klasa podseća na proceduralni kôd. Neki programski jezici kao što je `Lua` nemaju koncept klase, ali imaju koncept nasleđivanja. U ovom radu neće biti implementirane apstrakcije OO koncepata kao što su klase ili interfejsi.

Funkcionalna paradigma i njen odnos sa drugim paradigmama

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. Izračunavanja se vrše primenom i kompozicijom funkcija. Strukture podataka su nepromenljive i mogu biti beskonačne jer se izračunavanje elemenata kolekcija može vršiti po potrebi (npr. u programskom jeziku Haskell). Primer koda pisanog po funkcionalnoj paradigmi se može videti na slici 3.5.

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

Slika 3.5: 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[4] ili vratiti kao rezultat izračunavanja drugih funkcija. Pošto se funkcije često prosleđuju drugim funkcijama, obično su podržane i *anonimne funkcije* (drugačije nazvane i *lambda funkcije*, po uzoru na *lambda račun* na kojem je zasnovano funkcionalno programiranje). 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. Takođe, programi pisani u funkcionalnoj paradigmi komponovanjem funkcija višeg reda su često veoma čitljivi i kratki. Zbog svojih prednosti, funkcionalni koncepti se često uključuju u moderne predstavnike proceduralne paradigme (npr. `C++`, `Java`, `C#`, `Python`, `Lua`) kroz funkcije višeg reda i anonimane funkcije. Mane su potencijalno veće vreme izvršavanja, što danas obično ne predstavlja problem, kao i teškoća implementacije specifične sekvencijalne aktivnosti ili potreba za stanjem, što bi se lako implementiralo imperativno ili preko OO paradigme.

[4]Funkcije koje primaju druge funkcije kao argumente ili kao povratnu vrednost vraćaju funkcije se nazivaju *funkcije višeg reda*.

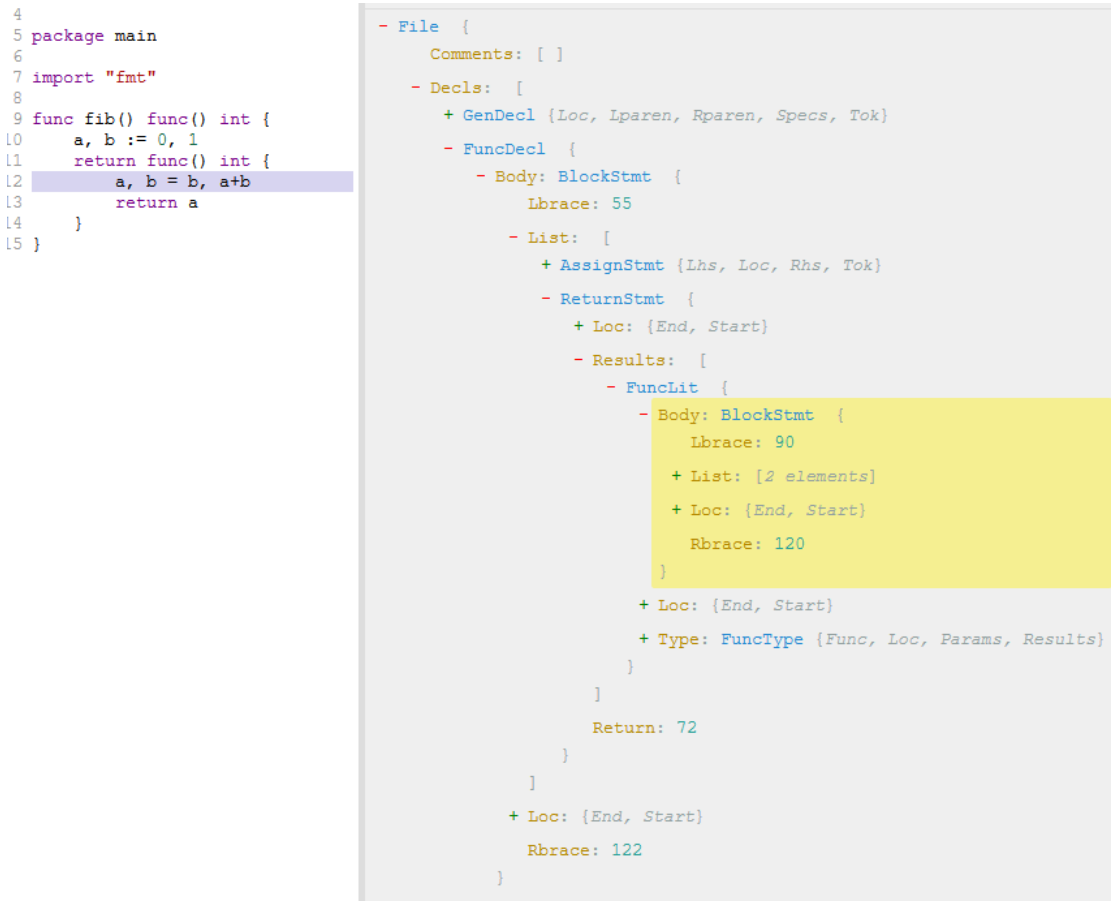
3.2 Opšte apstraktno sintaksičko stablo

Svaki programski jezik ima svoju gramatiku i na osnovu toga ima svoja gramatička pravila koja se oslikavaju u apstraktnim sintaksičkim stablima tih jezika. Na slikama 3.6 i 3.7 se mogu videti razlike jezika `Lua` i `Go`, kao primere skript odnosno proceduralne paradigme, kad se posmatra njihov AST.

Slika 3.6: AST isečka koda pisanog u programskom jeziku Lua.[5]

Kako bi se kreirala smislena apstrakcija stabla parsiranja, potrebno je identifikovati bitne informacije u stablu parsiranja ali i koncepte same gramatike koji su ponovno upotrebljivi. Najjednostavnije rešenje je oponašati 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 kreira-

[5]Prikazano putem `https://astexplorer.net/`.



Slika 3.7: AST isečka koda pisanog u programskom jeziku Go.[6]

ti 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 neke tipove rekurzije — ta pravila ne bi trebalo da imaju odgovarajuće tipove u opštoj apstrakciji.

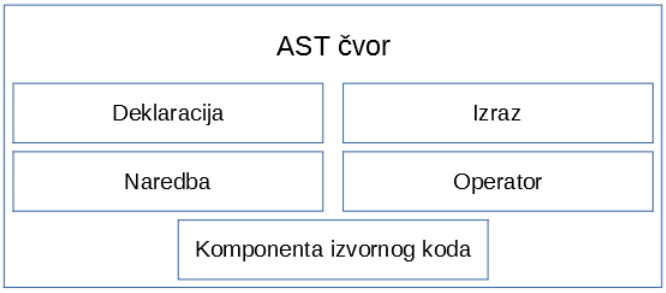
Pošto su u pitanju gramatike programskih jezika, onda je jasno da dosta različitih gramatika dele slične koncepte i da je moguće definisati tipove čvorova koji odgovaraju tim konceptima. Neki od njih mogu biti: naredba, izraz, deklaracija, poziv funkcije, dodela itd. Može se uočiti i hijerarhija između navedenih koncepata, međutim poziv funkcije se može smatrati kao samostalna naredba u

[6]Prikazano putem `https://astexplorer.net/`.

nekim programskim jezicima kao npr. u programskom jeziku Lua, ali može biti i deo izraza. Dakle, prilikom definisanja hijerarhije ne treba dozvoliti nešto što nema smisla (npr. ako je dozvoljeno višestruko nasleđivanje u okviru hijerarhije koncepata i poziv funkcije je u isto vreme i naredba i izraz, onda se izrazi u kojima figurišu pozivi funkcija sastoje od više naredbi).

Osim naredbi i izraza (koje vezuju operatori), kao osnovnih koncepata imperativnih jezika, deklaracije se ne pojavljuju u skript jezicima zbog dinamičke tipiziranosti. Moguće je, međutim, posmatrati i promenljive u kodovima skript jezika kao promenljive deklarisane neposredno pre trenutka njihove upotrebe. 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.

Neophodno je napomenuti da se apstrahovanjem mogu izgubiti značajne informacije koje mogu promeniti semantiku koda koji se apstrahuje. Ukoliko uzmemo za primer operator sabiranja u programskim jezicima C i Java, apstahovanjem gubimo informaciju o redosledu izvršavanja — u C standardu nije propisano kojim redosledom će se izračunavati operandi, dok u jeziku Java redosled izračunavanja je zagarantovan. U ovom radu, ukoliko je reč o programskom jeziku C, nije vođeno računa o pažljivom apstrahovanju informacija koje nisu propisane standardom jezika C (npr. u radu je pretpostavljeno da je celobrojni tip veličine 4 bajta).



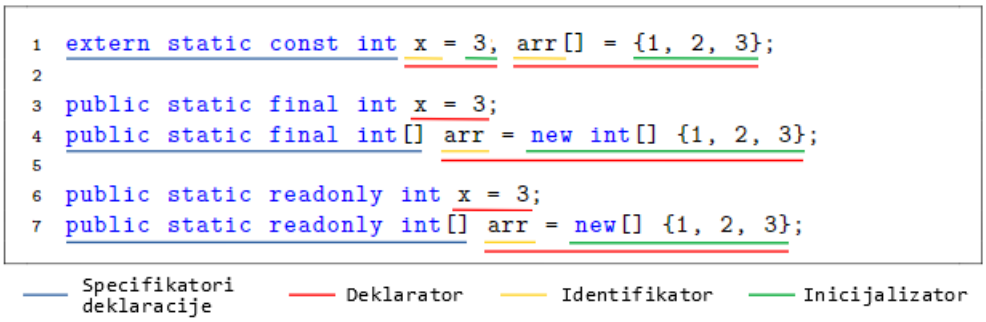
Slika 3.8: Prikaz osnovnih vrsta AST čvorova.

Na slici 3.8 se mogu videti osnovni tipovi AST čvorova zasnovani na konceptima opisanim iznad. U nastavku će po odeljcima biti detaljnije opisan svaki od prikazanih tipova. Na ovom dijagramu (ali i na ostalim dijagramima koji opisuju tipove čvorova opšte apstrakcije u ovom poglavlju) predstavljene su hijerarhije — ukoliko je jedan pravougaonik unutar drugog to odgovara specijalizaciji, drugim rečima tip naveden u unutrašnjem pravougaoniku je specijalizacija tipa čije je ime navedenog u pravougaoniku koji ga sadrži.

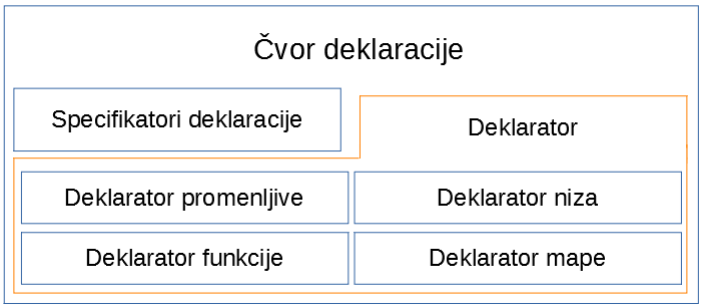
Čvorovi deklaracija

Kao što je to već rečeno, u statički 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. Oba imena su uzeta po uzoru na imena pravila gramatike programskog jezika C.

Veliki broj proceduralnih jezika dozvoljava deklarisanje više promenljivih odjednom koje dele iste specifikatore deklaracije. Stoga specifikatore neće pratiti jedan deklarator, nego *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 3.9 se može videti dekompozicija deklaracije promenljive i niza u različitim proceduralnim programskim jezicima a na slici 3.10 uočena hijerarhija sa podvrstama deklaratora.



Slika 3.9: 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).



Slika 3.10: Prikaz vrsti AST čvorova deklaracije.

Kao što je prikazano na slici 3.9, 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 tipova `float` i `double`, postoji i tip `decimal`[7] prisutan u programskom jeziku C#. Sva tri ova tipa mogu da se posmatraju na istom nivou apstrakcije kao tip realnih brojeva. Za korisnički definisane tipove isto ne može da se primeni.

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 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 (pošto funkcije nisu građani prvog reda u imperativnoj paradigmi[8]).

Deklaratori promenljive i niza mogu dodatno sadržati i *inicijalizator*. 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. Štaviše, serijalizacija se može iskoristiti i nad konstruktorima i metodima klasa, pritom označavajući statička i privatna polja. Ovim pristupom je moguće porediti mapu ili instancu klase definisane u skript jeziku sa objektom klase definisane u OO jeziku ili strukturom

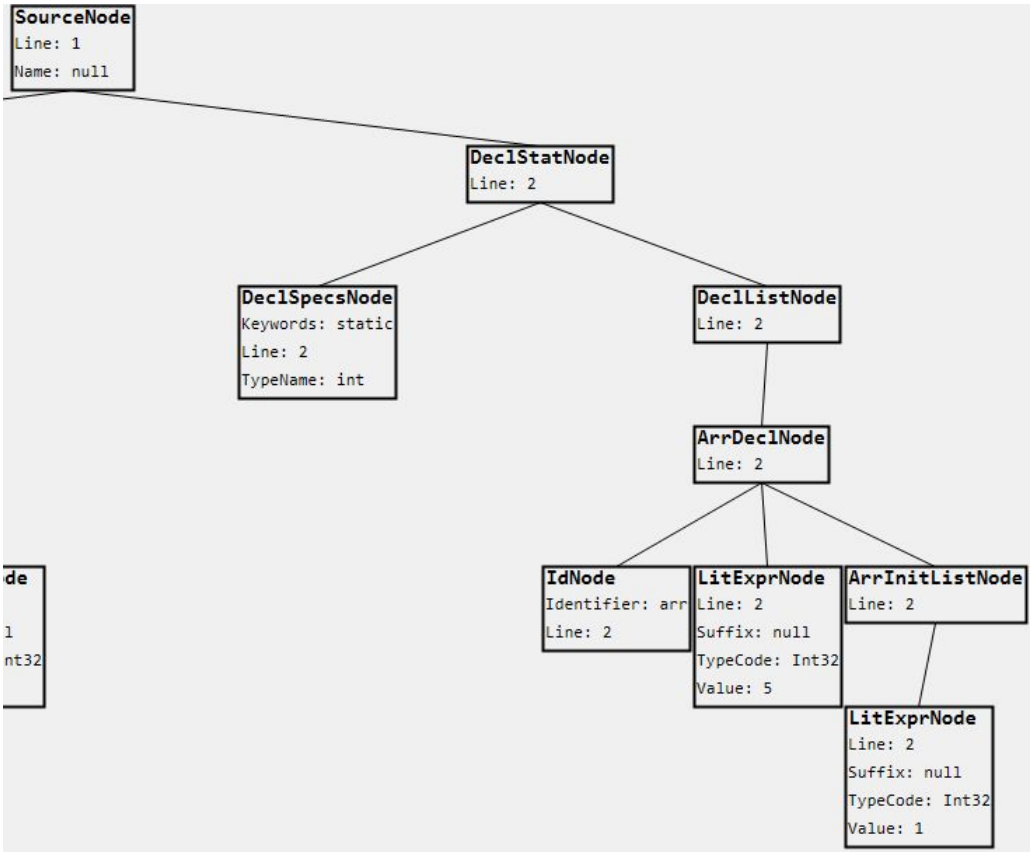
[7]Tip `decimal` predstavlja 128-bitni realan broj sa povećanom veličinom mantise a smanjenom veličinom eksponenta u odnosu na tip `double`. Koristi se pri numeričkm izračunavanjima gde preciznost primitivnih tipova realnih brojeva nije dovoljna.

[8]Moguće je prosleđivati pokazivače na funkcije drugim funkcijama, ali to u ovom radu nije razmatrano.

definisanom u proceduralnom jeziku. Pošto je klasa primarno OO koncept, njeno apstrahovanje nije razmatrano u ovom radu. Skupovi i torke, za sada nije moguće predstaviti u apstrakciji. Takođe, informacija o vrsti mape se takođe ne čuva u okviru apstakcije mape.

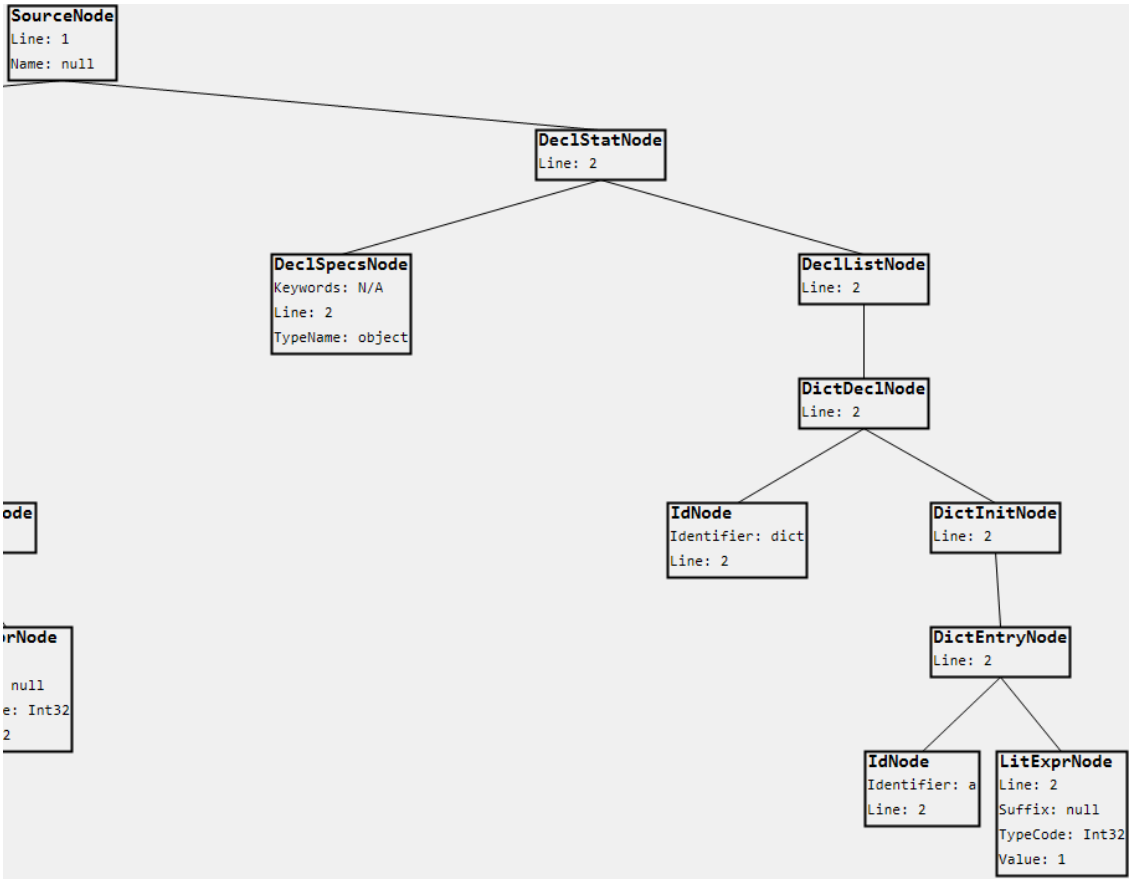
Na slici 3.11 se može videti kreirani AST za nekoliko deklaracija pisanih u programskom jeziku C, a na slici 3.12 se može isto videti demonstracija *automatske deklaracije* promenljivih za skript programski jezik Lua. Naime, pre prvog pojavljivanja identifikatora biće dodata deklaracija tog identifikatora, kako bi razlika između apstrakcija dobijenih iz proceduralnih i skript jezika bila što manja. U ovom slučaju će se deklaracija i dodela spojiti u deklaraciju sa inicijalizatorom.

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



Slika 3.11: Primer deklaracije promenljive i niza u programskom jeziku C i deo odgovarajućeg opšteg AST.

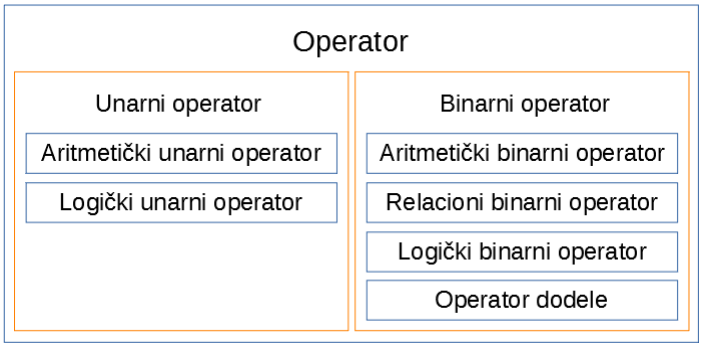
```
1  arr = { 1, 2 }
2  dict = { a = 1 }
```



Slika 3.12: Primer deklaracije niza i mape u programskom jeziku Lua i deo odgovarajućeg opšteg AST.

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 sa istom arnošću. Na slici 3.13 se može videti hijerarhija operatora korišćena dalje u apstrakciji. Binarni operatori zahtevaju dva operanda i pišu se infiksno, dok unarni zahtevaju jedan operand i pišu se prefiksno. Ternarni uslovni operatori koji postoje u nekim programskim jezicima se mogu zameniti naredbom uslovnog grananja.



Slika 3.13: Podela operatora na osnovu njihove arnosti.

Unarni aritmetički operatori su unarni operatori koji figurišu u aritmetičkim izrazima, npr. operator promene znaka, operator bitovske negacije[9], 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 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[10] i vraćaju vrednost proizvoljnog, ne nužno istog tipa; dok unarni logički operatori primaju i vraćaju bulovsku vrednost[11]. 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 ali i bitovski binarni operatori. 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 operato-

[9]Bitovski izrazi se mogu posmatrati kao vrsta aritmetičkih izraza.

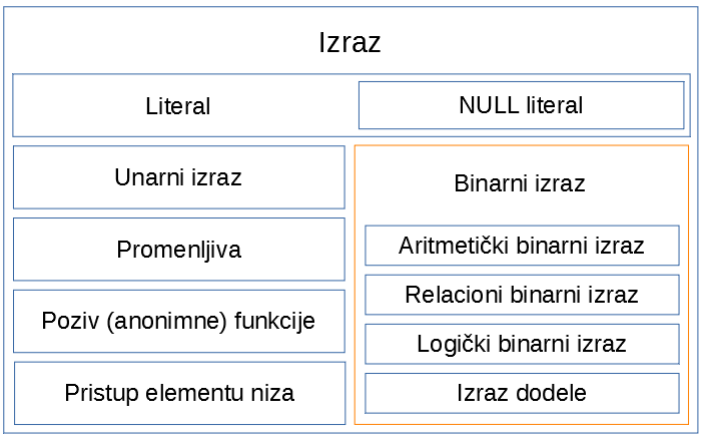
[10]Ne postoji ograničenje na brojevne tipove jer se u nekim jezicima operatori mogu predefinisati tako da rade i za korisnički definisane tipove (engl. *operator overloading*).

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

re, moguće je apstraktno posmatrati sve binarne operatore tako što ih definišemo kao strukturu koja definiše 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 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 više jednostavnijih izraza.

Čvorovi izraza

Izraz, kao što se može videti na primeru gramatike sa slike 2.9, se definiše rekurzivno i izraze mogu proširiti razni operatori. Na slici 3.14 se mogu videti tipovi apstraktnih konstrukcija koje će se koristiti da bi se predstavili izrazi. Dodatno, za vezivanje izraza će se koristiti apstrakcije operatora definisane u prethodnom odeljku.



Slika 3.14: Vrste čvorova izraza.

Najjednostavniji izraz predstavljaju konstante ili *literali*. Literali mogu biti 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 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, može se kreirati poseban tip literala, na slici 3.14 nazvan `NULL literal` (ime je pozajmljeno od praznih pokazivača u programskim jezicima kao što je npr. C, nije podržan pokazivački tip čvora u okviru apstrakcije).

Osim literala, samostalne promenljive mogu predstavljati validan izraz, u kom slučaju je vrednost izraza trenutna vrednost te promenljive. Slično važi i za indeksni pristup nizu[12]. U slučaju indeksnog pristupa, potrebno je navesti 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.

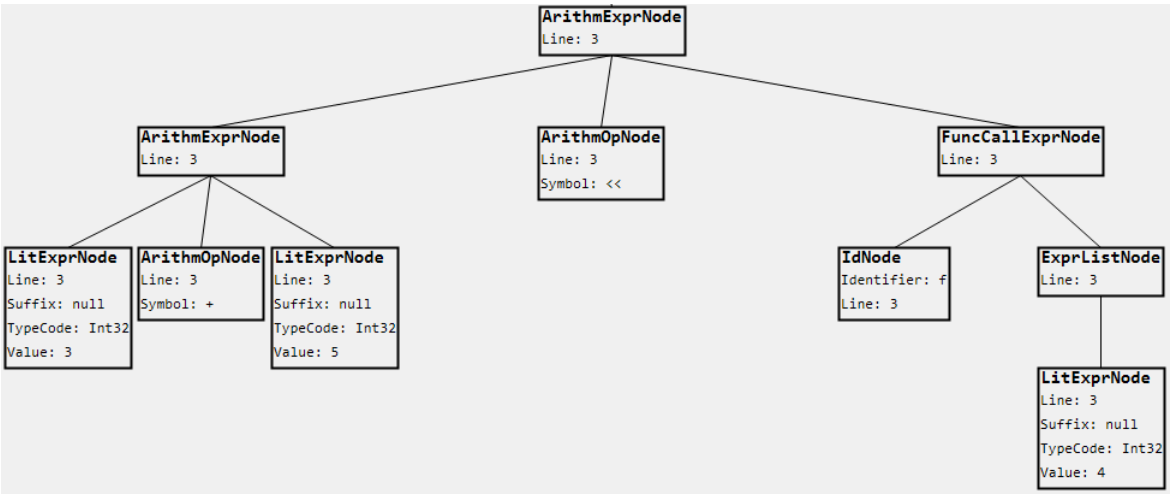
Unutar izraza se mogu naći i pozivi funkcija. Naravno, pretpostavljamo da funkcija ima povratnu vrednost, koja će se iskoristiti nakon poziva funkcije u kontekstu iz kojeg je ona pozvana. Iako je u ovom radu akcenat na imperativnoj paradigmi, neki funkcionalni koncepti su implicitno podržani zbog načina na koji je implementiran opšti AST — operatori kompozicije funkcija (na način opisan u 3.2) i anonimne funkcije (koje se mogu smatrati validnim izrazima). Sa druge strane, poređenje funkcionalnog koda sa imperativnim kodom nije razmatrano.

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 3.13), binarne izraze delimo na sličan način. Naravno, svaki od tipova binarnog izraza zahteva odgovarajući tip binarnog ope-

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

ratora. Slično se može uraditi i za unarne izraze, ali takođe i napraviti podela na prefiksne i postfiksne unarne izraze. S obzirom da je cilj napraviti opšti AST, činjenica da li je unarni operator prefiksni ili postfiksni nije od suštinskog značaja, pogotovo ukoliko se uzme u obzir da dva programska jezika mogu imati unarne operatore sa istom semantikom ali različitom pozicijom u odnosu na operand — u jednom jeziku taj operator može biti prefiksni a u drugom postfiksni. Kako bi poređenje ovakvih operatora funkcionisalo bez obzira na njihovu poziciju u odnosu na operand, u ovom radu nije pravljena podela na prefiksne i postfiksne unarne operatore.

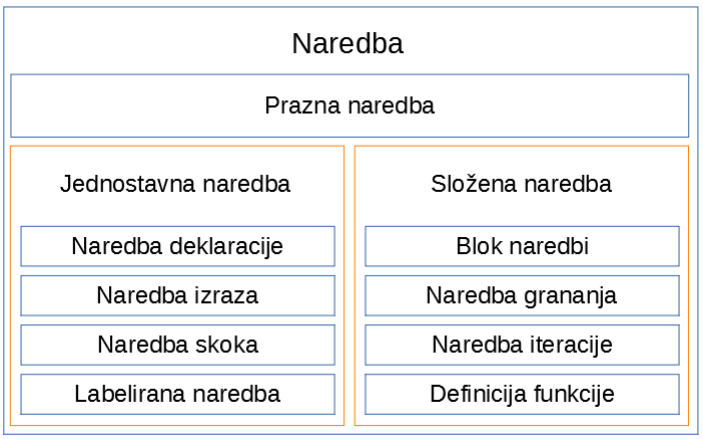
Na slici 3.15 se mogu videti kreirani AST za izraz `(3 + 5) << f(4)`. Ovaj izraz poprima isti oblik bez obzira na to koji je programski jezik u pitanju, ali iako se sintaksa bude razlikovala ili operatori budu imali drugi simbol, logika operatora opisana putem funkcije će ostati ista.



Slika 3.15: AST generisan od izraza `(3 + 5) << f(4)`.

Čvorovi naredbi

Naredbe su najkomplikovanije za apstrahovanje zbog njihove raznovrsnosti. Programski jezici često uvode nove sintaksičke 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 3.16 se mogu videti tipovi apstraktnih konstrukcija koje će se koristiti da bi se predstavile naredbe.



Slika 3.16: Vrste čvorova naredbi.

Veliki broj programskih jezika podržava praznu naredbu, sa semantikom ne izvršavanja nikakvih operacija. U programskim jezicima koji su zasnovani na sintaksi jezika C, praznu naredbu navodimo samo korišćenjem simbola za kraj naredbe (`;`), dok u programskom jeziku Python koristimo ključnu reč `pass`.

Naredbe su podeljene na *jednostavne* i *složene*, koje 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 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 okviru iste naredbe, 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.

Naredbe se mogu označiti, po uzoru na koncept *labele* u imperativnim jezicima — identifikatorom 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.

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 — recimo za programski jezik C pomoću velikih zagrada (`{}`). Za skript jezike često nije potrebna nikakva eksplicitna oznaka već se blok naredba prepoznaje implicitno ili se navodi korišćenjem različitih nivoa indentacije (Python). Neki skript jezici, na primer Lua, zahtevaju eksplicitno navođenje ključnih reči pre početka i nakon kraja blok naredbe ukoliko je ona deo složenije naredbe.

Naredbe uslovnog grananja se sastoje od *uslova*, koji može biti relacioni ili logički izraz, naredbe koja se vrši ukoliko je uslov ispunjen (*then* grana), i opciono naredbe koja se izvršava ako uslov nije ispunjen (*else* grana). 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. Na slici 3.18 se može videti AST za naredbu grananja.

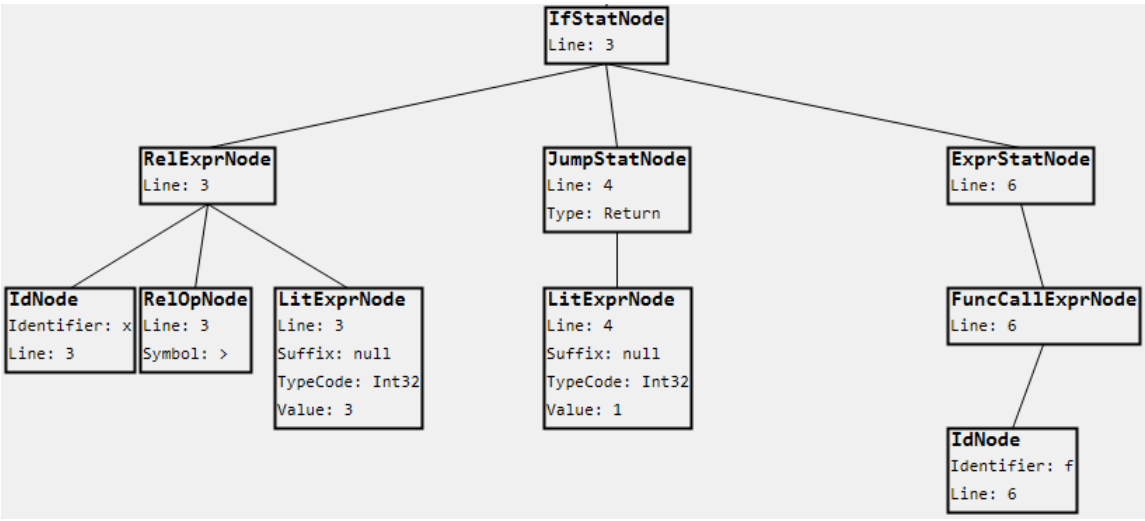
Naredbe iteracije imaju raznovrsni oblik u programskim jezicima. Najčešće podržane naredbe iteracije su *for* i `while` petlje. U opštem slučaju, dovoljno je koristiti samo jedan tip petlji, ali zarad jednostavnosti i prisutnosti ovih tipova u velikoj većini programskih jezika oba će biti podržana. Ostali tipovi petlji, kao što su *do-while* ili *repeat-until* petlje, će se svoditi na njih. *do-while* petlja se

```
do                                  something()
    something()                     while (condition) do
while (condition)                       something()
```

```
repeat                              something()
    something()                     while (not condition) do
until (condition)                       something()
```

Slika 3.17: Procedura svođenja ređih tipova petlji (levo) na *while* petlju (desno) prikazana u pseudo-jeziku.

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[13]. Ovaj proces je ilustrovan na slici 3.17.



Slika 3.18: AST naredbe grananja.

[13]Proces svođenja je znatno teži u prisustvu naredbi bezuslovnog skoka, kao što su prelazak na sledeću iteraciju petlje ili prekid petlje. Stoga, za potrebe ovog rada, prilikom svođenja se ne vodi računa o ovakvim komplikacijama.

Glava 4

Semantičko poređenje opštih AST

Jedna od motivacija svođenja imperativnih jezika na isti nivo apstrakcije može biti poređenje kodova pisanih u različitim programskim jezicima. Pritom, s obzirom da su u pitanju stabla, moguće je koristiti razne algoritme za poređenje stabala (ali i grafova uopšte) nad ovakvim apstrakcijama. Dodatno, potrebno je i definisati kriterijum poređenja — moguće je porediti kodove *strukturno*, *semantički* 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 u praksi, nalik na one dobijene u ovom radu. Precizna definicija strukturne sličnosti se obično iskazuje u terminima sličnosti strukture njihovih apstrakcija. Naravno, postoje kodovi koji nisu strukturno ekvivalentni ali su semantički ekvivalentni — takvi slučajevi se onda neće razmatrati zbog neispunjenosti pretpostavke o strukturnoj sličnosti.

Pretpostavka strukturne sličnosti je ograničavajuća i znatno smanjuje broj slučajeva upotrebe takvog upoređivača. Međutim, danas je od velikog značaja provera ispravnosti programa dobijenih sitnim refaktorisanjem već postojećih programa, nekada ne menjajući strukturu uopšte (a ako se struktura koda menja onda se to obično radi u manjim koracima između kojih i dalje može da važi pretpostavka strukturne sličnosti kodova u susednim koracima). Slično, prepisivanje programa sa jednog programskog jezika na drugi se javlja najčešće prilikom migracija na nove tehnologije. U takvim situacijama implicitno je prisutna strukturna slučnost, što zavisi od konkretnih programskih jezika ali se u praksi često smanjuje napor tako što se održava struktura koda, barem u inicijalnim verzijama.

Definicija strukturne sličnosti za potrebe ovog rada će se odnositi na sličnosti u rasporedu blokova naredbi dok će raspored naredbi u blokovima biti nebitan.

Na primer, ukoliko se prvi program sastoji od bloka naredbi u kome se nalaze dva druga bloka, očekuje se da i drugi program ima istu organizaciju blokova, pri čemu se pod terminom blok podrazumevaju i složene naredbe koje se sastoje od bloka naredbi u sebi, kao što su definicije funkcija, uslovna grananja, petlje i ostali tipovi složenih naredbi opisanih u 3.2. Pod ovim uslovima, moguće je porediti blokove prvog programa sa odgovarajućim blokovima drugog programa.

Neophodno je napomenuti da se apstahovanjem gube informacije koje su od važnosti za statičku analizu, stoga sve pozitivne rezultate analize sprovedene u ovom radu je potrebno pažljivo razmotriti. Konkretno, tipovi podataka u jezicima koji ne propisuju striktno veličinu tipova (npr. celobrojni tip u programskom jeziku C) možda neće biti pravilno predstavljeni u apstrakciji. Prilikom rada sa ovim vrednostima u nekim slučajevima može doći do prekoračenja, ili nedostatka prekoračenja, što je različito ponašanje od originalnog. Takođe, ukoliko se apstrahuje specifikacija pisana u programskom jeziku koji ne propisuje striktno redosled izračunavanja operanada izraza, prilikom analize neće nužno biti isti redosled izvršavanja operanada izraza u odnosu na specifikaciju.

4.1 Simboličko izvršavanje

Današnji softver je veoma kompleksan i često funkcioniše na različitim nivoima arhitekture velikih projekata. Stoga je proces verifikacije softvera veoma značajan i delikatan. U procesu verifikacije se najčešće koriste ručno pisani testovi i pregledi koda od strane drugih programera. 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 ulaze zbog njihovog ogromnog broja (ukoliko posmatramo samo funkciju jedne promenljive koja prima 32-bitni ceo broj, broj mogućih ulaza je 2^{32}) potrebno je 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 deli dva broja, te klase mogu biti celi brojevi, realni brojevi, neke specijalne vrednosti specifične za operaciju deljenja (recimo 0), kao i granice za tip podataka iz čijeg domena argumenti funkcije 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 ako ispoljavanje istih zavisi

od stanja drugih komponenti ili pak nekih nedeterminističkih ponašanja samog sistema. Poželjno je čitav izvorni kôd pokriti testovima (engl. *code coverage*) — iako dostignuta pokrivenost koda od 100% i dalje ne znači da taj kôd ispravno radi.

Statička analiza koda predstavlja analizu izvornog koda bez pokretanja istog sa ciljem ispitivanja stanja u kojima se može naći program i proveru rada jedinice koja se testira za mnogobrojne ulaze. Iako ispravna u teoriji, u praksi nailazi na puno problema — osnovni je razlika u apstrakcijama koju prave statički analizator i programer.

Simboličko izvršavanje [3] predstavlja sredinu između klasične verifikacije putem pisanja testova i statičke analize koda. Prilikom simboličkog izvršavanja, umesto stvarnih vrednosti ulaza 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 koda sa slike 4.1. Pretpostavimo da imamo deklarisanje promenljive `a`, `b` i `c` i da se neke operacije izvršavaju nad njima, reprezentovano komentarom u liniji 3. 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.

```
1  int a, b, c;
2
3  // ...
4
5  int x = 0, y = 0, 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 4.1: Isečak C koda dat kao primer nad kojim će se prikazati simboličko izvršavanje.

Ukoliko put izvršavanja programa zavisi od simboličke promenljive, kao što je to slučaj za izvorni kôd sa slike 4.1, simbolička promenljiva se konceptualno „grana" i analiza se nastavlja za oba slučaja posebno. Tako se dobija drvo izvršavanja, gde svaka putanja odgovara mnogim individualnim testovima koji bi uzrokovali prolazak izvršavanja tom putanjom. Vrednosti promenljivih u tim testovima moraju zadovoljiti uslove na kraju svake putanje — tzv. *uslove putanje* (engl. *path conditions*). Odgovarajuće stablo izvršavanja za izvorni kôd sa slike 4.1 sa definisanim simboličkim vrednostima α, β i γ se može videti na slici 4.2. Svaka naredba dodele je uokvirena pravougaonikom dok je uslov uokviren elipsom. Boje grana odgovaraju istinitosnoj vrednosti uslova iz poslednje linije u tom trenutku. Na kraju svake grane se nalazi uslov putanje za tu granu koje u nekim slučajevima može jedinstveno odrediti vrednost simboličke promenljive koja dovodi do prolaska tom putanjom ili u opštem slučaju generisati test primer koji dovodi do prolaska tom putanjom. Dakle, ukoliko je stablo simboličkog izvršavanja poznato, moguće je generisati kontra-primere koji pokazuju da program ne radi kao što je očekivano.

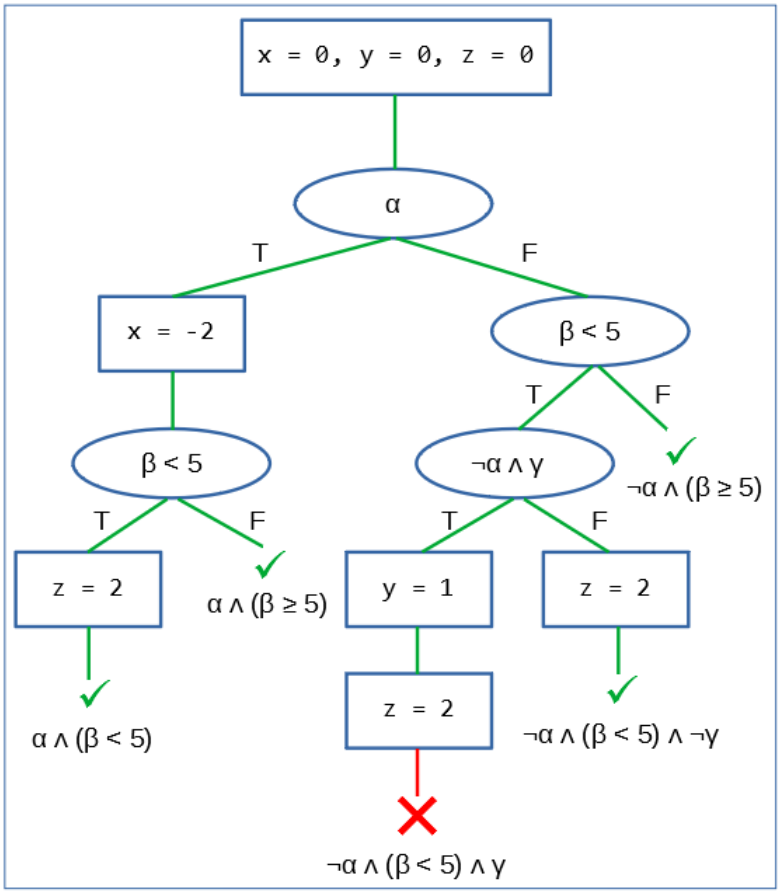
Simboličko izvršavanje, iako konceptualno moćno, ima par problema koji su uzrokovani pre svega kompleksnošću problema koji se rešava:

- *Eksplozija putanja* — Broj putanja izvršavanja eksponencijalno zavisi od broja uslovnih grananja u kodu. Ukoliko imamo tri naredbe grananja, broj putanja izvršavanja je $2^3 = 8$. Štaviše, dovode do značajnog uvećanja broja stanja, jer ukoliko u petlji postoji uslov koji zavisi od simboličke vrednosti koja uzima vrednosti iz opsega 32-bitnog celog broja, broj putanja kroz petlju je u tom slučaju 2^{31}, a u nekim slučajevima i beskonačan. Slično važi i za rekurziju.

- *Ograničenja rešavača* — Kako broj putanja raste, povećava se i broj uslova koji se moraju zadovoljiti prilikom nalaženja kontraprimera. U nekim slučajevima je moguće osloniti se na *SMT rešavače* [4] za nalaženje kontra-primera kako bi se ublažio ovaj problem.

- *Modelovanje podataka* (engl. *heap modelling*) — Kreiranje simboličkih struktura podataka i pokazivača nije jednostavan proces.

- *Modelovanje okruženja* (engl. *environment modelling*) — Nije uvek jednostavno adaptirati mehanizam za česte potrebe prilikom dizajna softvera kao



Slika 4.2: Drvo simboličkog izvršavanja na kom su prikazane sve putanje koje se razmatraju. Na kraju svake grane je napisan uslov koji mora da važi da bi se došlo do lista te grane.

što je korišćenje eksternih biblioteka i sistemskih poziva ali i specifičnosti sistema.

Postoji dosta alata i biblioteka koje pružaju interfejs za simboličko izvršavanje u raznim programskim jezicima — jedan od najpoznatijih alata za simboličko izračunavanje je *KLEE* [6], izgrađen nad *LLVM* infrastruktorom [20] [18] i dizajniran za analizu koda pisanog u programskom jeziku C. U ovom radu je simboličko izvršavanje korišćeno za detekciju razlika u vrednostima promenljivih uz pomoć biblioteka za programski jezik C#.

4.2 Algoritam za semantičko poređenje

U ovom radu je poređenje vršeno pomoću algoritma pisanog specifično za rad sa opštim apstrakcijama opisanim u 3.2. Grubi opis algoritma za poređenje, u daljem tekstu *upoređivač*, je prikazan na slici 4.3. 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.

```
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 4.3: Osnovni AST upoređivač.

Podrazumevana implementacija poređenja može biti takva da se uporede atributi svih čvorova a zatim se svako dete prvog čvora rekurzivno uporedi sa odgovarajućim detetom drugog čvora (ukoliko imaju isti broj dece). Ako neki par dece nije ekvivalentan, onda to ne važi ni za njihove roditelje. Za većinu tipova čvorova ovakvo poređenje je dovoljno.

Naredbe koje se sastoje od više drugih naredbi, kao što su npr. definicije funkcija ili petlje, se moraju porediti drugačije jer složena naredba može sadržati lokalne

promenljive. Niz naredbi koji predstavlja jednu složenu naredbu će u nastavku biti referisan pod terminom *blok* ili *blok naredbi*. Za poređenje blokova naredbi je stoga definisana posebna procedura poređenja opisana u nastavku.

4.3 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, a možda ih čak ima i različit broj.

Upoređivač se zasniva na poređenju vrednosti promenljivih na kraju svakog bloka naredbi. Apstrakcije dva programa se porede paralelno — *blok-po-blok*. Naredbe svakog bloka se izvršavaju i prate se izmene vrednosti promenljivih deklarisanih do tog trenutka (bilo u bloku koji se trenutno razmatra, ili u roditeljskim blokovima). Na kraju svakog bloka se vrši provera jednakosti simboličkih vrednosti promenljivih iz oba programa deklarisanih do tog trenutka i svaka razlika se prijavljuje kao potencijalna greška. Pošto broj promenljivih u programima ne mora biti isti iako su oni semantički ekvivalentni, za one promenljive koje nemaju parnjaka prilikom poređenja se prijavljuju upozorenja ali ne i greške ukoliko nije bilo konflikata prilikom poređenja vrednosti ostalih promenljivih. Upoređivač blokova naredbi je prikazan na slici 4.4.

1: **procedure** UporediBlokove(b_1, b_2)
2: $gds_1 \leftarrow$ *simboli iz svih predaka bloka* b_1
3: $gds_2 \leftarrow$ *simboli iz svih predaka bloka* b_2
4: $lds_1 \leftarrow$ *lokalni simboli za blok* b_1
5: $lds_2 \leftarrow$ *lokalni simboli za blok* 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 4.4: 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 dok su globalni simboli deklarisani van trenutnog bloka a mogu se referisati iz njega. Pronalaženje deklarisanih simbola u bloku podrazumeva prolaz kroz naredbe bloka

i registrovanje svih naredbi deklaracije, izvlačenje deklaratora iz njih i, uzimajući u obzir opcione inicijalizatore, kreiranje simboličke vrednosti za upravo deklarisani identifikator. Identifikator i opcioni simbolički inicijalizator čine *simbol*. Isto se ponavlja za sve naredbe deklaracije u bloku i rezultat je skup deklarisanih simbola.

Nakon registrovanja svih lokalnih simbola proverava se njihova ekvivalentnost u funkciji `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 4.5. 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 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 program). Kad se blokovi izvuku, rekurzivno se poziva upoređivač blokova za pronađene parnjake. 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 razlike jer neki blokovi neće imati svog parnjaka ili njihovo uparivanje nije jednoznačno.

Takođe je važno napomenuti da procedura `IzvrsiNaredbe` vraća povratnu vrednost koja se u algoritmu sa slike 4.4 ignoriše. Razlog za to je što, u nekim slučajevima, iako su unutrašnji blokovi naredbi različiti, želimo ipak da uporedimo konaćne vrednosti promenljivih pre nego što zaključimo da postoji problem. Ukoliko samo poredimo par čvorova složenih naredbi, u tom slučaju ćemo iskoristiti povratnu vrednost ove procedure jer ne postoji širi kontekst iz kog je ona pozvana.

```
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 4.5: Upoređivač blokova naredbi.

Glava 5

Implementacija i evaluacija

U ovom poglavlju će biti opisana implementacija pratećeg projekta nazvanog *Language Invariant Code Comparer* (skr. *LICC*), pisanog u programskom jeziku C# 8.0, koristeći *.NET Core 3.1* radni okvir. Lekseri i parseri za ulazne gramatike su takođe generisani u programskom jeziku C#. C# je izabran zbog lakoće implementacije velikih projekata i sveobuhvatne podrške u vidu 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 kôd, ali i da poredi takav AST sa drugim. Čitav projekat je dostupan u potpunosti na servisu GitHub na adresi `https://github.com/ivan-ristovic/LICC`.

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 njen adapter, što dozvoljava poređenje koda sa specifikacijom datom u obliku pseudo-koda. Čitav projekat se sastoji od više komponenti, organizovanih po prostorima imena, od kojih su značajnije:

`LICC` — Glavni program (korisnički interfejs) koji omogućava generisanje, prikaz, serijalizaciju i poređenje AST.

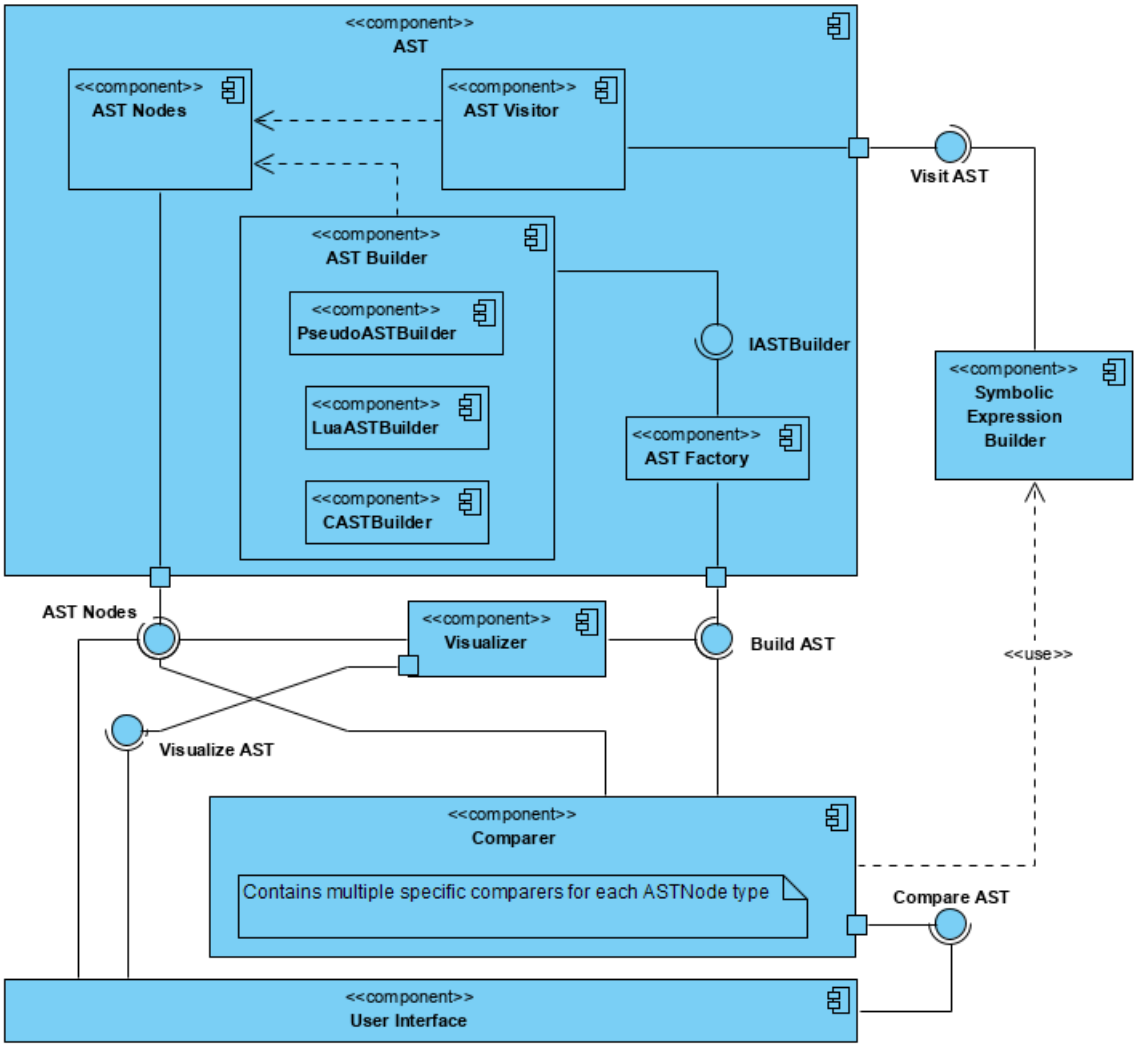
`LICC.AST` — Biblioteka klasa za rad sa opštom AST apstrakcijom.

`LICC.Core` — Upoređivač opštih AST (konzolni izlaz).

`LICC.Visualizer` — Komponenta za vizualizaciju (grafički prikaz AST).

`LICC.Tests` — Prateći testovi jedinica koda i integracioni testovi.

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 Actions* podrškom za neprekidnu integraciju (engl. *continuous integration*, skraćeno *CI*). CI omogućava prevođenje izvornog koda nakon svake izmene kao i izvršavanje akcija nakon prevođenja kao što su testiranje ili generisanje predmeta za upotrebu (engl. *artifacts*) koji predstavljaju rezultat procesa prevođenja i mogu se direktno isporučiti.



Slika 5.1: UML dijagram komponenti implementacije.

5.1 Generisanje parsera uz pomoć ANTLR4

U ovom odeljku će biti opisan proces generisanja leksera i parsera za izvorne kodove pisane u proizvoljnom programskom jeziku korišćenjem alata ANTLR4. Poznati programski jezici kao što su C i Lua već imaju definisane ANTLR4 gramatike, tako da će se krenuti od procesa kreiranja gramatike za proizvoljni programski jezik kako bi se pokazalo da je moguće dobiti AST polazeći i od proizvoljne gramatike, a zatim će se koristiti ANTLR4 za generisanje leksera i parsera za tu gramatiku. Nakon toga, biće opisan i interfejs za obilazak stabla parsiranja koje generiše parser, i taj interfejs će se koristiti u procesu kreiranja opšteg AST ali i kao inspiracija za kreiranje interfejsa za obilazak opšteg AST.

Preduslovi za pokretanje ANTLR4

Kako bi se ANTLR4 koristio, potrebno je instalirati ANTLR4 i imati *Java Runtime Environment* (skr. *JRE*) instaliran na sistemu i dostupan globalno pokretanjem putem komande `java`. Instalacija se sastoji od preuzimanja najnovije *.jar* datoteke[1], sa zvanične stranice [2] ili recimo korišćenjem *curl* alata[2]:

```
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* datoteku na sledeći način (pretpostavljajući da se *.jar* datoteka 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* datoteka nalazi u direktorijumu `C:\lib`):

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

[1]Takođe je moguće prevesti izvorni kôd dostupan na servisu GitHub `https://github.com/antlr/antlr4`

[2]`https://curl.haxx.se/`

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

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

Dodatno, za Unix sisteme[3], 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 ANTLR4 `.jar` datoteku i moguće je napraviti prečicu za brzo pokretanje (nalik na ANTLR4 alias):

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

Generisanje parsera koristeći ANTLR4

U ovom odeljku će biti opisan proces kreiranja interfejsa za parsiranje programa pisanih u imperativnom, strogo tipiziranom pseudo-programskom jeziku (u nastavku *pseudo-jezik*), sličnom pseudokodu. Dobijeni interfejs za obilazak stabla parsiranja može da se koristi u opšte svrhe, a za potrebe ovog rada će se koristiti za generisanje apstraktnog sintaksičkog stabla za izvorni kôd pisan u pseudojeziku. Najpre će biti definisana gramatika pseudo-jezika prateći ANTLR4 pravila za definisanje gramatika. Tek nakon kompletnog opisa gramatike biće iskorišćen ANTLR4 kako bi se generisao parser za pseudo-jezik. Kao i za svaki drugi imperativni jezik, treba podržati neke osnovne koncepte: *identifikatore*, *izraze*, *naredbe* i slično. Primer izvornog koda pisanog u pseudojeziku je prikazan na slici 5.2.

Identifikatori su niske karaktera koje predstavljaju oznaku koja odgovara određenoj memorijskoj adresi. Identifikatori se koriste umesto sirovih vrednosti adresa kako bi kôd bio čitljiviji i lakši za pisanje — na nivou asemblera se većinom koriste adrese ili automatski generisane oznake. Na slici 5.3 se može videti definicija

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

```
algorithm Sample
begin
    declare real pi = 3.14
    declare real x = (1 + x) * (1 - pi / 4)
    declare integer array arr
    declare point set points
    declare boolean x = False

    function fib(n : integer) returning integer
    begin
        if n < 0 then error "Requiring positive integer"
        else if n <= 1 then return n
        else return call fib(n-1) + call fib(n-2)
    end
end
```

Slika 5.2: Primer koda pisanog u pseudojeziku.

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 ANTLR4 praviti lekser i parser, možemo i tokene definisati na isti način kao i gramatička pravila dajući regularni izraz za njihovo poklapanje. Listovi stabla parsiranja su uvek tokeni, drugim rečima se nazivaju i *terminalni simboli*. Tokeni se, osim u listovima, mogu naći bilo gde u stablu parsiranja. ANTLR4 dozvoljava jednostavne definicije pravila u kojima figuriše promenljiv broj drugih pravila, pri čemu se koriste simboli kao u regularnim izrazima, što je iskorišćeno za definiciju pravila za definisanje identifikatora.

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

Slika 5.3: Definicija identifikatora za pseudo-jezik.

Pseudo-jezik će biti strogo tipiziran. Stoga je potreban koncept tipa podataka (videti definiciju deklaracije), čija je definicija data na slici 5.4. Tip može biti *primitivan* (drugim rečima *prost*) ili *složen*. Primitivni tipovi su podržani u samoj

sintaksi jezika — u našem slučaju brojevni tipovi i niske. Brojevi mogu biti celi (`integer`) ili realni (`real`). U složene tipove spadaju korisnički definisani tipovi (sa imenom `NAME`, u četvrtoj alternativi pravila `typename` sa slike 5.4) 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. Specijalne reči kao što su `integer` ili `array` će biti rezervisane reči našeg pseudo-jezika, tzv. *ključne reči*. Ključne reči se u pravilima navode između apostrofa.

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

Slika 5.4: Definicija tipa podataka za pseudo-jezik.

Definicija *literala* je prikazana na slici 5.5. Literali, u okviru pseudojezika, predstavljaju istinitosne konstante `True` i `False`, brojevne konstante ili niske karaktera. Brojevne konstante mogu bili celobrojni ili realni dekadni brojevi. Realne konstante 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.

Izrazi, iako definisani rekurzivno, se mogu posmatrati kao kombinacija promenljivih, operatora i poziva funkcija sa odlikom da se mogu *evaluirati*, tj. moguće je izračunati njihovu vrednost. Iz definicije pravila `exp` sa slike 5.6, mogu se uočiti tipovi izraza. Literal predstavlja validan izraz. Promenljive, definisane pravilom `var` su takođe izrazi, jer se trenutna vrednost promenljive posmatra kao vrednost

```
literal : 'True' | 'False' | INT | FLOAT | STRING ;
STRING : '"' ( EscapeSequence | ~('\\'|'"') )* '"'  ;
INT : Digit+ ;
FLOAT
    : Digit+ '.' Digit* ExponentPart?
    | '.' Digit+ ExponentPart?
    | Digit+ ExponentPart
    ;

fragment
ExponentPart : [eE] [+-]? Digit+ ;
fragment
Digit : [0-9] ;
fragment
EscapeSequence : '\\' [abfnrtvz"'\\] | '\\' '\r'? '\n' ;
```

Slika 5.5: Definicija literala za pseudo-jezik.

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[4], 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 vezani binarnim operatorima, što znači da oni zahtevaju dva argumenta. Postoje i unarni operatori, od kojih su podržani operatori promene znake i logičke negacije, što se vidi iz pravila `uop`. Poziv funkcije je takođe validan izraz jer funkcije imaju povratne vrednosti i on je označen imenom `cexp` (skraćeno od *function call expression*)[5].

Sledeći korak je definisanje naredbi pseudo-jezika — samostalnih izvršivih jedinica koda. Slično kao i u drugim programskim jezicima, potrebno je podržati koncept deklaracije promenljive ili procedure odnosno funkcije, dodele vrednosti

[4]Prioritet i asocijativnost aritmetičkih operacija se može postaviti uvođenjem zagrada ili se određuje na osnovu redosleda pravila.

[5]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.

```
exp
    : literal
    | var
    | '(' exp ')'
    | exp aop exp
    | exp rop exp
    | exp lop exp
    | uop exp
    | cexp
    ;
var
    : NAME ('[' iexp ']')?
    ;
iexp
    : literal
    | var
    | aexp
    ;
cexp
    : 'call' NAME '(' explist? ')'
    ;
aexp
    : exp aop exp
    ;
explist
    : exp (',' exp)*
    ;
aop : '*' | '/' | '+' | '-' | 'div' | 'mod' ;
rop : '>' | '>=' | '<' | '<=' | '==' | '=/=' ;
lop : 'and' | 'or' ;
uop : '-' | 'not' ;
```

Slika 5.6: Definicija izraza za pseudo-jezik.

izraza promenljivoj, naredbe kontrole toka (grananje i petlje) i slično. U nekim slučajevima će biti potrebno definisanje kompleksnih naredbi koje se sastoje od više drugih naredbi — blokovi naredbi. Kako bismo označili da su naredbe deo bloka naredbi, koristićemo ključne reči `begin` i `end`, osim ukoliko je reč o samo jednoj naredbi. Na slici 5.7 je definisano šta se sve smatra jednom naredbom (prateći redosled alternativa pravila): prazna naredbe (označena ključnom rečju `pass`), deklaracija, dodela, poziv funkcije, vraćanje vrednosti izraza (ključna vrednost `return`) iz funkcije, prekidanje izvršavanja davanjem poruke o grešci, naredba grananja, *while* petlja, *repeat-until* petlja i inkrementiranje/dekrementiranje vrednosti promenljive.

```
statement
    : 'pass'
    | declaration
    | assignment
    | cexp
    | 'return' exp
    | 'error' STRING
    | 'if' exp 'then' block ('else' block)?
    | 'while' exp 'do' block
    | 'repeat' block 'until' exp
    | ('increment' | 'decrement') var
    ;
```

Slika 5.7: Definicija naredbe za pseudo-jezik.

Svaka promenljiva mora biti određenog tipa, što se postiže kroz pravilo `type`. Promenljivoj se, opciono, može pridružiti početna vrednost, drugim rečima ona se može *inicijalizovati* tako da joj se pridruži vrednost nekog izraza. Procedure i funkcije imaju opcione parametre, vrednosti izraza koje im se prosleđuju kasnije u pozivu kao argumenti. Lista parametara, takođe prikazana na slici 5.8, se navodi kao lista proizvoljno mnogo parova `NAME : type`, što se vidi iz definicije pravila `parlist`.

```
declaration
    : 'declare' type NAME ('=' exp)?
    | 'procedure' NAME '(' parlist? ')' block
    | 'function' NAME '(' parlist? ')' 'returning' type block
    ;
parlist
    : NAME ':' type (',' NAME ':' type)*
    ;
```

Slika 5.8: Definicija deklaracije za pseudo-jezik.

Program možemo posmatrati kao niz naredbi kome je pridružen identifikator koji označava ime programa. Na slici 5.9 se može videti pravilo koje definiše program[6] i blok naredbi pseudo-jezika. Pritom, `NAME` je identifikator koji predstavlja ime programa (algoritma).

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

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

Slika 5.9: Definicija jedinice prevođenja i bloka naredbi za pseudo-jezik.

Na kraju, treba definisati sve ono što lekser treba da preskoči tokom prolaska kroz izvorni kôd. To su beline (nevidljivi karakteri kao što su razmaci, tabulatori i novi redovi) i komentari. Definicije ovih pravila se mogu videti na slici 5.10. 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 5.10: Definicija komentara i belina za pseudo-jezik.

Ovako definisanu gramatiku možemo sačuvati u datoteku sa imenom `Pseudo.g4`, potrebno je navesti ime gramatike na početku datoteke, kao na slici 5.11. Naredni korak je kreiranje leksera i parsera koristeći ANTLR4, pretpostavljajući da je instaliran na način opisan u 5.1. Pokretanjem ANTLR-a generišemo lekser i parser za gramatiku pseudo-jezika:

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

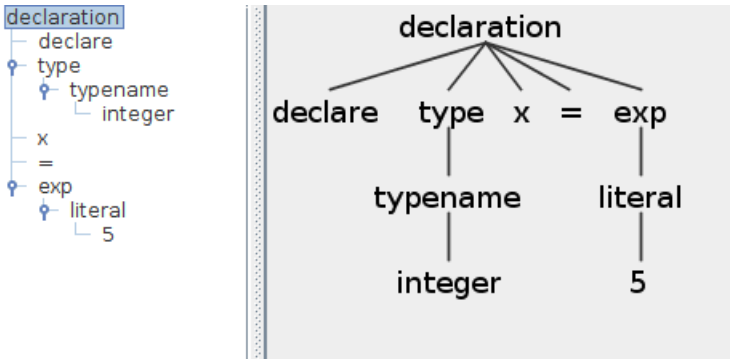
```
1 grammar Pseudo;
```

Slika 5.11: Definicija imena gramatike za pseudo-jezik.

Za veliki broj već postojećih programskih jezika, uključujući jezike C i Lua, dovoljno je preuzeti već standardizovane gramatike i generisati leksere i parsere za njih koristeći ANTLR4. ANTLR4 će generisati lekser i parser podrazumevano napisane u programskom jeziku Java u odvojenim izvornim datotekama kao zasebne klase. Ukoliko želimo to da promenimo, možemo koristiti opciju `-Dlanguage`, što će biti korisno jer lekseri i parseri treba da budu generisani u programskom jeziku C#.

Kako bismo testirali generisani lekser i parser, možemo koristiti ANTLR4 `TestRig` da vizualno prikažemo stablo parsiranja, s tim što moramo prvo kompilirati generisane Java klase. `TestRig` pozivamo navođenjem imena gramatike (koje se poklapa sa imenom leksera i parsera) i imenom pravila od koga će parser krenuti. Opcija `-gui` pokreće vizualni prikaz stabla parsiranja prikazan na slici 5.12 (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 korišćenjem mehanizma cevi (engl. *pipeline*) na sistemima zasnovanim na UNIX-u (na slici 5.12 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 5.12: Grafički prikaz stabla parsiranja koje generiše parser kreiran od strane `TestRig` biblioteke za naredbu deklaracije celobrojne promenljive u pseudo-jeziku.

Obilazak stabla parsiranja

ANTLR4, osim leksera i parsera za datu gramatiku, može da kreira interfejse i bazne klase koji prate projektne obrasce *posetilac* (engl. *visitor*) i osluškivač (engl. *listener*)[7] opisane u 2.3. Tako kreirani interfejsi i klase imaju metode za obilazak stabla parsiranja. ANTLR4 podrazumevano generiše interfejs osluškivača (slika 5.13) kao i baznu klasu koja implementira generisani interfejs tako što su sve implementirane metode prazne. Stoga, ukoliko korisnik želi da definiše operaciju samo u slučaju da se prilikom obilaska stabla parsiranja naiđe na određeni tip čvora, nije potrebno implementirati ceo interfejs osluškivača već je moguće naslediti baznu klasu i predefinisati samo jedan metod. ANTLR4 može, pored osluškivača, da generiše i interfejs posetilac (slika 5.14) ukoliko se navede odgovarajuća opcija `-visitor` prilikom pokretanja. Slično, ukoliko nije potrebno generisati osluškivač, može se koristiti opcija `-no-listener`.

Sa slike 5.13 se vidi da je moguće definisati metode koje će se pozivati prilikom ulaska ali i prilikom izlaska iz čvora određenog tipa prilikom obilaska stabla parsiranja. Pritom je važno kako se stablo obilazi. U slučaju ANTLR4, 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 strukture[9], može

[7] Osluškivač je varijanta obrasca *posmatrač* (engl. *observer*)

[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 (po-

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

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

Slika 5.14: 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 5.14) 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

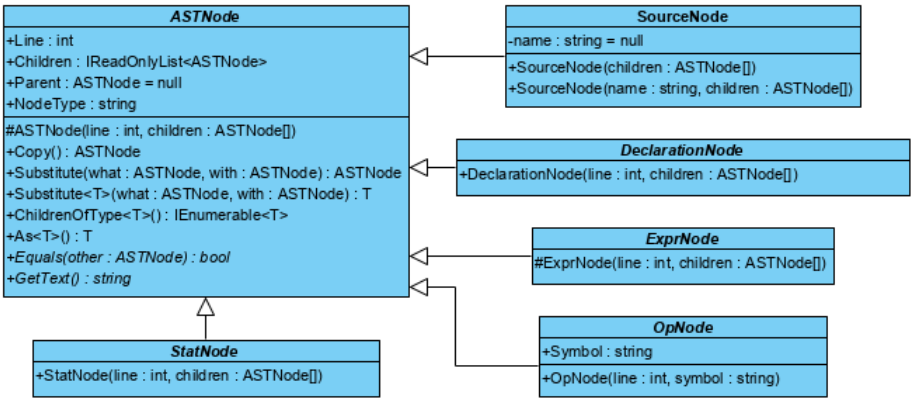
slednji ubačen) a da se ne ukloni isti. U slučaju opisanom iznad, implementacija LIFO strukture se naziva stek (engl. *stack*).

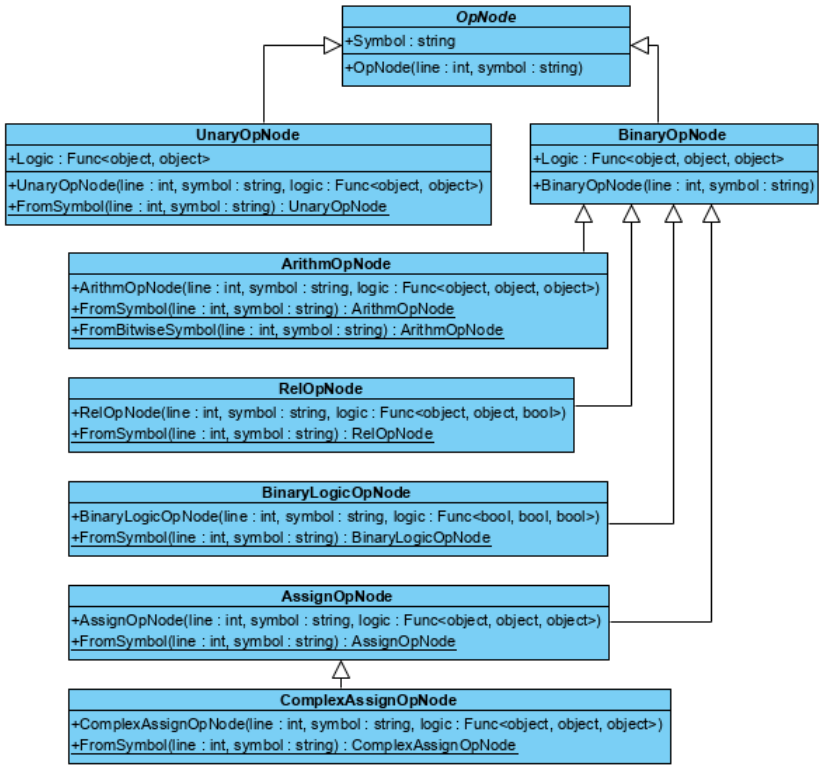
posetioca ne pozivaju automatski. Stoga je na programeru da nastavi obilazak i da odluči u koje čvorove želi da se spusti. Jasno je da i osluškivač i posetilac imaju svoje primene — ukoliko je potrebno obići stablo parsiranja i dovući neke informacije može se iskoristiti osluškivač jer onda ne moramo brinuti o obilasku. S 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 metode čija imena odgovaraju pravilima koja se javljaju u definiciji samog pravila koje odgovara kontekstu. Stoga za `BlockContext`, imajući u vidu definiciju sa slike 5.9 gde se koristi i pravilo `statement`, u okviru `BlockContext` klase biće implementiran i metod `statement()` koji vraća kontekst pravila tipa `StatementContext[]`. Metod `statement()` vraća niz 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 neće biti `null`.

5.2 Implementacija opšteg AST

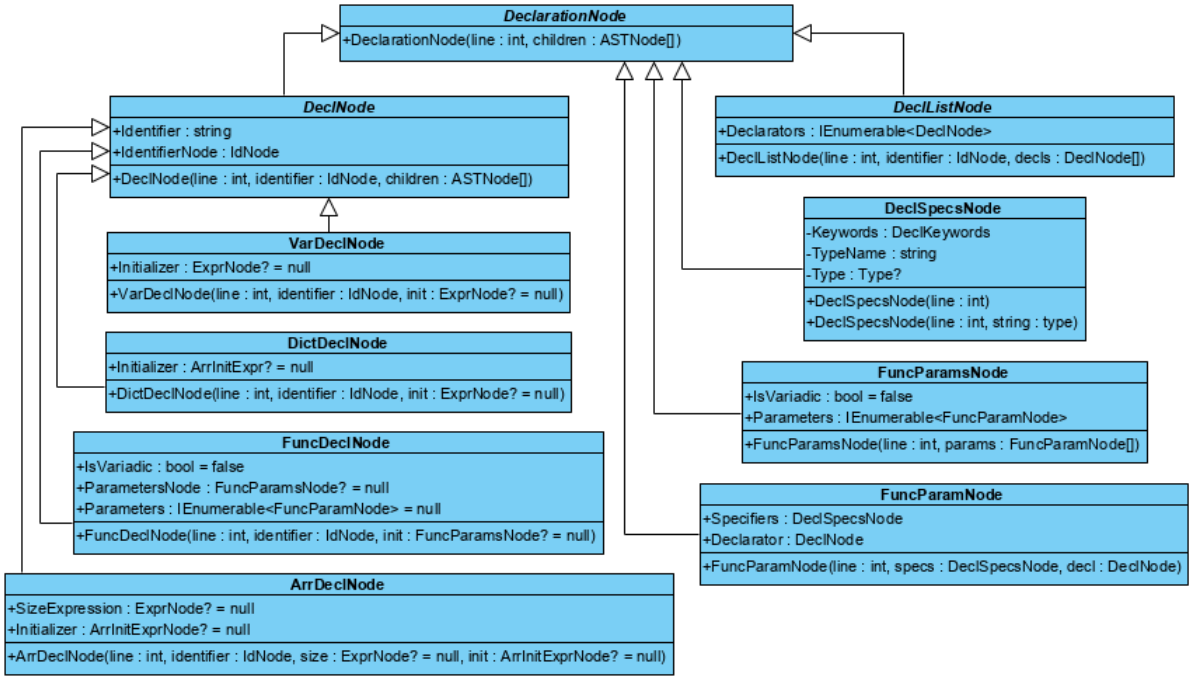
Implementacija prati hijerarhije opisane u poglavlju 3 kroz mehanizam nasleđivanja. Svaki tip čvora je implementiran kao zasebna klasa koja direktno ili tranzitivno nasleđuje apstraktnu klasu `ASTNode`. Klase koje čine apstrakciju zajedno sa njihovom hijerarhijom se može videti na slikama 5.15, 5.16 i 5.17.





Slika 5.15: UML klasni dijagram opšte apstrakcije (deo 1).

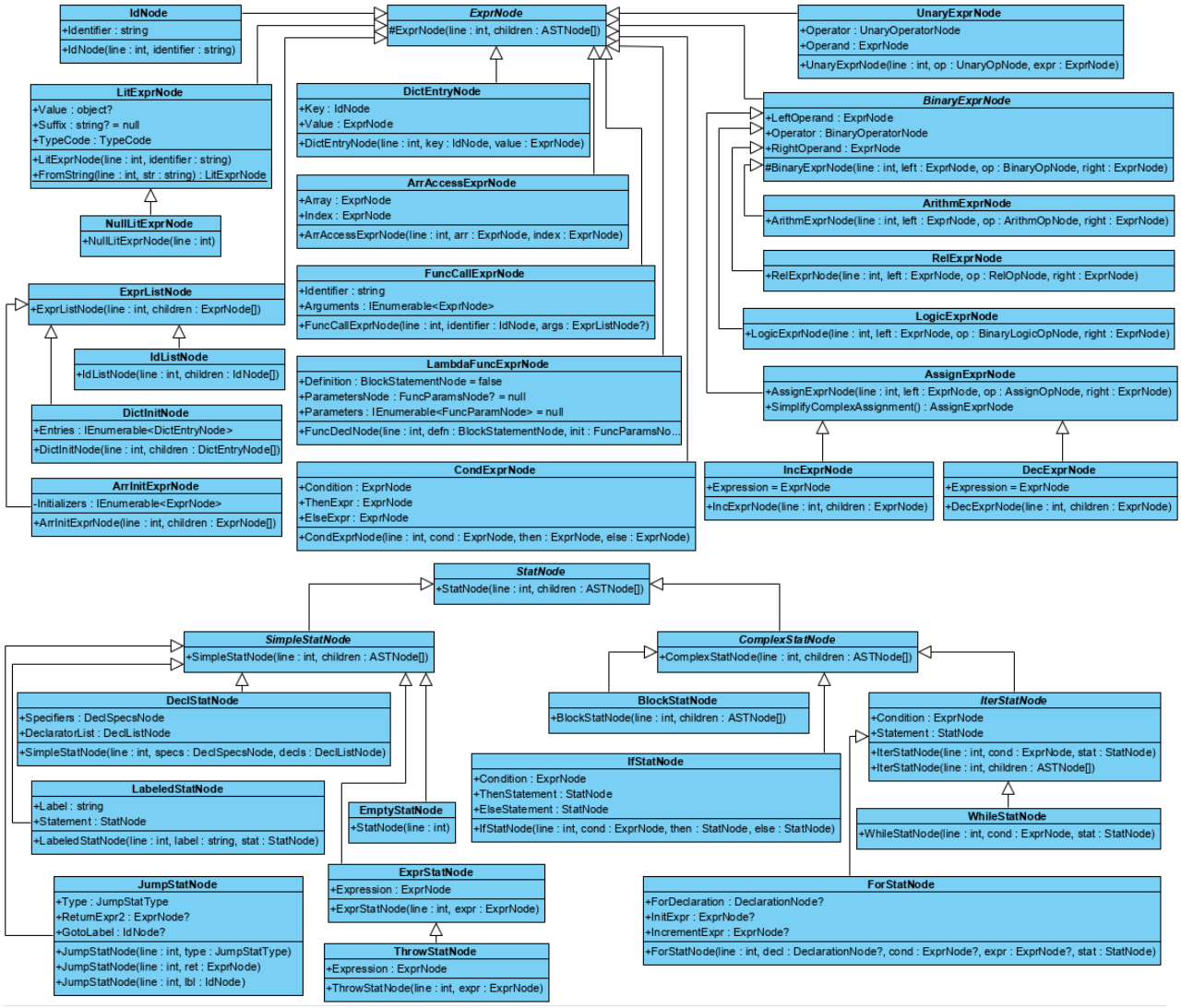
Jednom kreirani AST čvor je *nepromenljiv*, što znači da je i ceo AST nepromenljiv — ne mogu se dodavati ili uklanjati deca čvorovima u stablu. Međutim, moguće je klonirati AST čvorove ili vršiti zamenu određenog podstabla drugim podstablom ne menjajući original tako što se vraća izmenjena kopija originala. Svaki AST čvor se može porediti po jednakosti sa drugim AST čvorom po intuitivnoj logici poređenja pruženom kroz predefinisane operatore poređenja. Dostupne su i ekstenzije koje omogućavaju serijalizaciju stabla u JSON format.



Slika 5.16: UML klasni dijagram opšte apstrakcije (deo 2).

Pored implementacije klasa koje predstavljaju AST čvorove, kreiran je i javno dostupni način za obilazak AST putem obrazca posetilac kroz apstraktnu klasu `BaseASTVisitor<TResult>`. Ova klasa ima implementirane javne virtualno predefinisani metod `TResult Visit(ASTNode node)` jednog argumenta za svaki mogući tip koji nasleđuje tip `ASTNode`. Svi ti metodi su definisani tako da se pozove metod `VisitChildren(ASTNode)` koji će posetiti svu decu trenutnog čvora. Svaki naredni rezultat poziva metoda `Visit()` za svako dete se agregira pozivanjem metoda `TResult AggregateResult(TResult curr, TResult next)` koji podrazumevano samo vraća `next`. Dodatno, moguće je predefinisati metod `bool ShouldVisitNextChild(ASTNode node, TResult curr)` koji određuje da li je potrebno prekinuti posećivanje pre nego što se posete sva deca. Ovaj metod se poziva svaki put pre nego što se poseti dete i podrazumevano vraća `true`. `BaseASTVisitor` klasa je u implementaciji korišćena za kreiranja graditelja simboličkog izraza od sintaksičkog stabla izraza kao i evaluatora takvog stabla izraza.

Adapteri ili graditelji se koriste kao spona između stabla parsiranja za konkretni programski jezik i opšte AST apstrakcije. Uloga adaptera je da od stabla parsiranja kreiraju opšti AST tako što implementiraju posetilac stabla parsiranja



Slika 5.17: UML klasni dijagram opšte apstrakcije (deo 3).

konkretnog programskog jezika. Kako bi se pružio ujednačen način za kreiranje adaptera, pružena su dva interfejsa prikazana na slici 5.18.

Postupak kreiranja AST za dati izvornu datoteku se sastoji iz više koraka. Prvo se izvlači ekstenzija izvorne datoteke. Zatim se putem refleksije pronalazi klasa koja implementira interfejs `IAbstractASTBuilder` i ima u sebi prisutan atribut `ASTBuilderAttribute(string)`[10] čija se niska poklapa sa ekstenzijom izvorne

[10]Atributi u C#-u su deklarativni tagovi (nalik na anotacije u Java programskom jeziku) koji se koriste da se raznim elementima pridruže informacije dostupne u toku prevođenja. Ti elementi mogu biti klase, metode, osobine itd. Atributi se definišu kao klase i mogu imati polja i konstruktore

```
1  public interface IAbstractASTBuilder
2  {
3      ASTNode BuildFromSource(string code);
4  }
5
6  public interface IASTBuilder<TParser> : IAbstractASTBuilder where
       TParser : Parser
7  {
8      TParser CreateParser(string code);
9      ASTNode BuildFromSource(string code, Func<TParser,
           ParserRuleContext> entryProvider);
10 }
```

Slika 5.18: Interfejs graditelja opšteg AST od stabla parsiranja.

datoteke. Naposletku se kreira instanca pronađene klase i, na osnovu toga što implementira interfejs `IAbstractASTBuilder`, poziva se metod `BuildFromSource()` za sadržaj izvorne datoteke.

Dedukovani adapter implementira uvek `IASTBuilder` interfejs na osnovu tipa parsera koji je generisao ANTLR alat za gramatiku konkretnog programskog jezika. Metod `CreateParser()` instancira parser odgovarajućeg tipa. Razlog zašto ovaj proces nije uniforman je taj što se pre kreiranja parsera uvek kreira i lekser koji prvi dobija tok podataka pa se tek onda kreira parser. Dodatno, prilikom kreiranja parsera se definiše i pravilo gramatike od kojeg parser kreće (tzv. *ulazno pravilo*, engl. *entry rule*) koje je specifično za programski jezik. Stoga je nemoguće kreirati jedinstveni niz operacija koji će kreirati svaki parser pa se to radi prilikom implementacije adaptera.

Predefinisani metod `BuildFromSource()` interfejsa `IASTBuilder` treba da obiđe stablo parsiranja počev od pravila koje se dobija pozivom funkcije prosleđene kao drugi argument za dati tip parsera. Ovo je generalizacija kreiranja parsera i na prvi pogled izgleda suvišno jer, pošto se parsira validan izvorni kôd, mora se uvek krenuti od podrazumevanog ulaznog pravila za konkretni programski jezik. Međutim, ovaj način kreiranja AST je pogodan prilikom testiranja jedinica koda, jer ukoliko se npr. testira generisanje AST od izraza, nije potrebno pisati čitave funkcije sa naredbama samo da bi se testiralo generisanje apstrakcije izraza, već je moguće samo navesti izraz i instrukovati adapter da kreira parser tako da on počne od pravila koje definiše izraz.

s tim što vrednosti atributa moraju biti poznate za vreme prevođenja — konstante.

Trenutno implementirani adapteri u prostoru imena `LICC.AST.Builders` za programske jezike C i Lua nisu potpuni, dok je adapter za pseudojezik kompletan. Za programski jezik C, nisu podržane strukture, `gcc` ekstenzije, statičke tvrdnje (engl. *static assertions*), višestruki pokazivači, bitska polja, `switch naredba`, kvalifikovani i višedimenzioni nizovi. Za programski jezik Lua, nisu podržani složeni konstruktori tabela, složeni pozivi funkcija (oni pozivi gde je funkcija koja se poziva zapravo rezultat složenog izraza ili pristupa polju tabele) i kolekcijske petlje.

5.3 Implementacija semantičkog upoređivača

Implementacija algoritma upoređivača opisana u poglavlju 4.2 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.

Upoređivač kao rezultat svog rada vraća kolekciju potencijalnih problema (upozorenja ili grešaka) koje je detektovao prilikom analize. Ovakav pristup je odabran zbog lakoće testiranja upoređivača, s obzirom da se može očekivati određena kolekcija problema za određeni izvorni kôd. Problem se modeluje kao apstraktna klasa `BaseIssue` dok se upozorenja ili greške modeluju kroz njene konkretizacije `BaseWarning` i `BaseError`. Konkretne greške, kao što su npr. nedostatak deklaracija se mogu onda modelovati kao konkretizacije ovih klasa u zavisnosti od ozbiljnosti problema.

5.4 Vizualni prikaz stabla

Osim serijalizacije AST u JSON format, kreiran je potprogram za vizualni prikaz AST-a (u daljem tekstu *vizualizator*). Primeri izlaza vizualizatora se mogu videti na slikama iz poglavlja 3 — 3.11, 3.12, 3.15 i 3.18. Prikaz je izvršen koristeći

nativni `Graphics` paket i, s obzirom da je u pitanju *.NET Core 3.1* radni okvir, moguće je dobiti vizualni prikaz nezavisno od sistema.

Vizualizacija počiva na rekurzivnom algoritmu prikaza u dubinu — za svaki čvor se prikažu potomci, rasporede jednako po širini, a onda se roditelj centrira u odnosu na ukupnu širinu koju zauzimaju deca. Ovaj pristup nije prostorno optimalan, zbog varijacija u broju dece za čvorove različitih tipova. Što se informacija za svaki čvor tiče, prikazuju se vrednosti svih atributa čvora zajedno sa njegovim tipom u zaglavlju kao i grane do njegovih potomaka.

5.5 Korisnički interfejs

S obzirom da je aplikacija konzolna (osim dela komponente za vizualizaciju), korisnički interfejs se sastoji od argumenata komandne linije. Pokretanje programa bez argumenata pruža prikaz za pomoć u kome su nabrojane sve moguće opcije. Pomoć koja se pruža korisniku se može videti na slici 5.19.

Program zahteva da se kao prvi argument prosledi *glagol* (engl. *verb*) koji će odrediti operaciju koju program treba da izvrši. Od glagola zavisi broj i tip ostalih argumenata u nastavku. Mogući glagoli, sa svojim dodatnim opcijama, su:

- `ast [-v -c -t] source-path [-o output-path]`
  Generiše opšti AST za izvorni kôd na putanji `source-path` u JSON formatu i ispisuje isti na standardni izlaz, ili u datoteku na putanji `output-path` ako je prisutna opcija `-o`. Moguće je generisati kompaktan JSON (bez poravnanja) navođenjem opcije `-c`. Vizualizacija u obliku stabla se prikazuje u novom prozoru ukoliko je prisutna opcija `-t`.

- `cmp [-v] specification-path test-path`
  Generiše opšti AST za izvorne kodove na putanjama `specification-path` i `test-path` i poredi generisana stabla.

Dodatno, svi glagoli podržavaju opciono detaljno logovanje operacija koje program izvršava navođenjem opcije `-v`. Takođe, navođenjem opcije `--help` nakon glagola se ispisuje uputstvo specifično za taj glagol. Verzija programa se može proveriti opcijom `--version`. Opcije `-o`, `-c`, `-t` i `-v` imaju svoje duže sinonime — `--output`, `--compact`, `--tree` i `--verbose`, redom.

```
$ ./licc
ERROR(S):
No verb selected.

ast         AST generation commands
cmp         Compare source against the specification source
help        Display more information on a specific command.
version     Display version information.

$ ./licc ast
ERROR(S):
A required value not bound to option name is missing.

-v, --verbose    (Default: false) Verbose output
-t, --tree       (Default: false) Visualize AST tree
-o, --output     Output path
-c, --compact    (Default: false) Compact AST output
--help           Display this help screen.
--version        Display version information.

value pos. 0     Required. Source path

$ ./licc cmp
ERROR(S):
A required value not bound to option name is missing.

-v, --verbose    Set output to verbose messages.
--help           Display this help screen.
--version        Display version information.

value pos. 0     Required. Specification path.
value pos. 1     Required. Test source path.
```

Slika 5.19: Korisniči interfejs programa pružen kroz argumente komandne linije.

5.6 Testovi

Komponentu za kreiranje AST i komponentu za poređenje AST prate testovi jedinica koda. Testovi su organizovani u zasebnom projektu na sledeći način:

- `LICC.Tests.AST` — Testovi za adaptere i posetioce, kao i testovi funkcionalnosti metoda klase `ASTNode`.
- `LICC.Tests.Core` — Testovi upoređivača.

Radni okvir koji se koristi za testiranje je `NUnit`[11] koji pruža tzv. *model ograničenja* (engl. *constraint model*) i time omogućava pisanje čitljivog koda. Pisanje testova po modelu ograničenja se sastoji od korišćenja jednog metoda za pisanje svih testova koji kao argumente prima objekat koji se testira i složeni objekat koji predstavlja ograničenje koje objekat koji se testira treba da zadovoljava. Primer testa pisanog u ovom radnom okviru uz model ograničenja u kontekstu implementacije ovog rada se može videti na slici 5.20.

Osim testova jedinica koda, prisutni su i testovi integracije svih komponenti. Kao što je opisano u prethodnim odeljcima, rezultat rada adaptera je AST, dok je rezultat upoređivača za data dva stabla kolekcija problema. Ta dva odvojena procesa se onda mogu spojiti kako bi se testirala integracija te dve komponente — dakle, od dva programa očekivati određenu kolekciju problema. Primer za *swap* algoritam se može videti na slici 5.21.

5.7 Evaluacija

U ovom odeljku će biti prikazano nekoliko slučajeva upotrebe implementirane aplikacije. Prvo će biti pokazan primer generisanja opšteg AST u JSON formatu a zatim i primer poređenje dve implementacije istog algoritma u dva različita programska jezika. Algoritam koji će biti korišćen u nastavku kao primer je algoritam razmene vrednosti promenljivih implementiran kroz funkciju `swap` koja menja vrednosti dveju globalnih promenljivih. Na slici 5.22 se mogu videti implementacije ovog algoritma koje će biti polazne tačke za kreiranje opšteg AST i poređenja istih.

Generisanje opšteg AST

AST je moguće generisati od izvornog koda navođenjem glagola `ast`. Ukoliko su dostupni izvorni kodovi sa sadržajima kao na slici 5.22 u odgovarajućim datotekama, moguće je generisati opšti AST u JSON formatu zadavanjem glagola `ast` kao na slici 5.23. U gornjem delu slike je prikazan samo deo izlaza zbog veličine generisanog JSON sadržaja, u srednjem delu slike je prikazan kompaktni JSON ispis zadat opcijom `-c`, dok je u donjem delu slike prikazan ispis zadat opcijom `-v`

[11] `https://nunit.org/`

```
protected FuncDefNode AssertFunctionSignature(
  string src, int line, string fname,
  string returnType = "void", bool isVariadic = false,
  AccessModifiers access = AccessModifiers.Unspecified,
  QualifierFlags qualifiers = QualifierFlags.None,
  params (string Type, string Identifier)[] @params)
{
  FuncDefNode f = this.GenerateAST(src).As<FuncDefNode>();
  this.AssertChildrenParentProperties(f);
  this.AssertChildrenParentProperties(f.Definition);
  Assert.That(f, Is.Not.Null);
  Assert.That(f.Line, Is.EqualTo(line));
  Assert.That(f.Declarator, Is.Not.Null);
  Assert.That(f.Declarator.Parent, Is.EqualTo(f));
  Assert.That(f.Keywords.AccessModifiers, Is.EqualTo(access));
  Assert.That(f.Keywords.QualifierFlags, Is.EqualTo(qualifiers));
  Assert.That(f.Identifier, Is.EqualTo(fname));
  Assert.That(f.ReturnTypeName, Is.EqualTo(returnType));
  Assert.That(f.IsVariadic, Is.EqualTo(isVariadic));
  if (@params?.Any() ?? false) {
    Assert.That(f.Parameters, Is.Not.Null);
    Assert.That(f.Parameters, Has.Exactly(@params.Length).Items);
    Assert.That(f.ParametersNode, Is.Not.Null);
    Assert.That(
      f.Parameters.Select(
        p => (p.Specifiers.TypeName, p.Declarator.Identifier)),
          Is.EqualTo(@params)
      );
    }
  return f;
}

[Test]
public void ComplexDefinitionTest()
{
  FuncDefNode f = this.AssertFunctionSignature(@"
    float f(const unsigned int x, ...) { return 3.0; }",
    2, "f", "float", isVariadic: true,
    @params: ("unsigned int", "x")
  );
  Assert.That(f.Definition.Children, Has.Exactly(1).Items);
}
```

Slika 5.20: Primer jediničnog testa za proveru generisanog AST čvora za datu funkciju.

pri čemu je prikazan samo dao izlaza koji se generiše prilikom posećivanja stabla parsiranja i generisanja AST čvorova.

```
protected void Compare(ASTNode src, ASTNode dst,
  MatchIssues? expectedIssues = null)
{
  expectedIssues ??= new MatchIssues();
  var issues = new ASTNodeComparer(src, dst).AttemptMatch();
  Assert.That(issues, Is.EquivallentTo(expectedIssues));
}

[Test]
public override void DifferenceTests()
{
  this.Compare(
    this.FromPseudoSource(@"
      algorithm Swap
      begin
        declare integer x = vx
        declare integer y = vy
        procedure swap()
        begin
          declare integer tmp = x
          x = y
          y = tmp
        end
      end
    "),
    this.FromCSource(@"
      int x = vx, y = vy;
      void swap() { int tmp = x; y = tmp; x = y; }
    "),
    new MatchIssues()
      .AddError(new BlockEndValueMismatchError("x", 1, "vy", "vx"))
      .AddError(new BlockEndValueMismatchError("x", 3, "vy", "vx"))
  );
}
```

Slika 5.21: Primer kompletnog testa za algoritam *swap*.

U okviru repozitorijuma projekta je moguće pronaći još primera upotrebe. Na stranici `https://github.com/ivan-ristovic/LICC/LICC.AST/Samples` se mogu naći izvorne datoteke u kojima su iskorišćene sve podržane sintaksičke strukture od kojih se može kreirati AST. Izvorne datoteke su pisane u programskim jezicima C, Lua i pseudojeziku (po jedna za svaki programski jezik). Ove datoteke u okviru projekta služe kao integracioni testovi ali čitaocu mogu dati jasan uvid u koncepte koje LICC podržava u zavisnosti od programskog jezika. U okviru repozitorijuma se nalaze i uputstva za kreiranje novih graditelja i upoređivača.

```
1  int x = vx, y = vy;
2  void swap() { int tmp = y; y = x; x = tmp; }
```

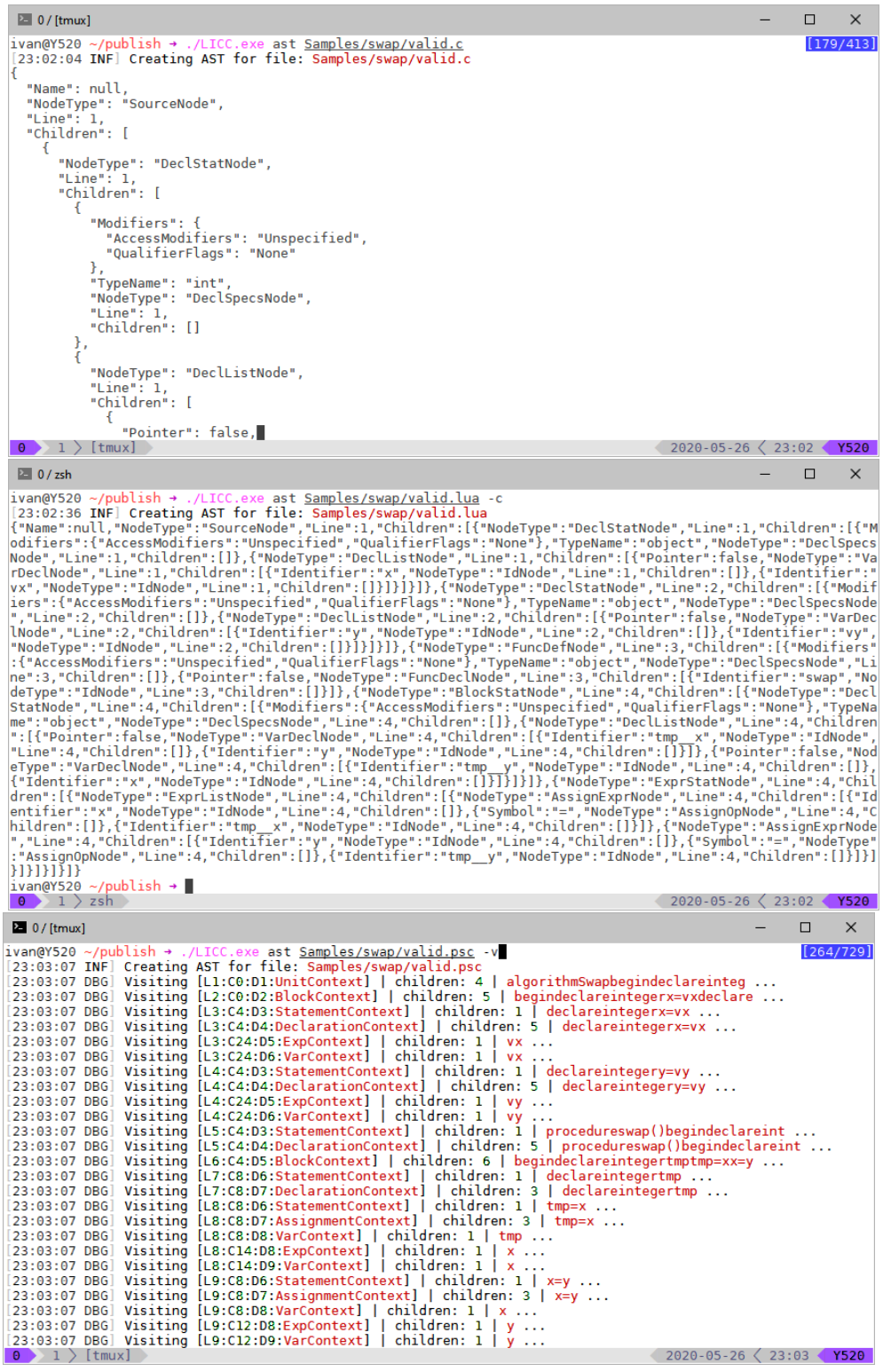
```
1  x = vx
2  y = vy
3  function swap()
4      x, y = y, x
5  end
```

```
1  algorithm Swap
2  begin
3      declare integer x = vx
4      declare integer y = vy
5      procedure swap()
6      begin
7          declare integer tmp
8          tmp = x
9          x = y
10         y = tmp
11     end
12 end
```

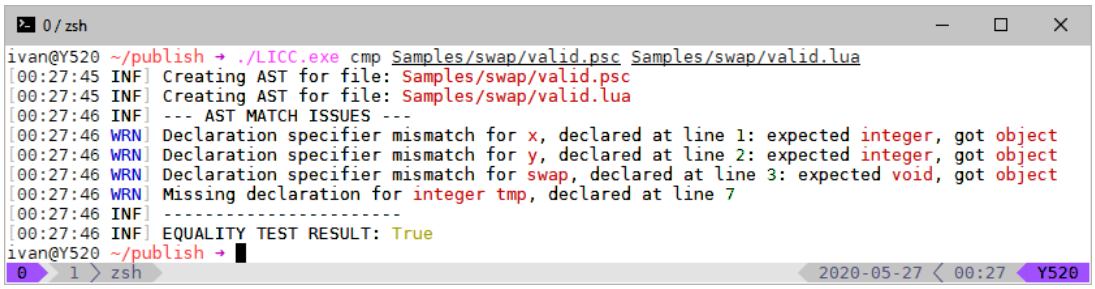
Slika 5.22: Izvorni kodovi algoritma `swap` u programskim jezicima C (gore), Lua (sredina) i u pseudojeziku (dole).

Poređenje opštih AST

Jedan od osnovnih slučajeva upotrebe alata LICC može biti testiranje validnosti implementacije na osnovu date specifikacije, koja ne mora biti pisana u istom programskom jeziku kao i implementacija. Ukoliko kao specifikaciju za algoritam `swap` uzmemo izvorni kod u pseudo-jeziku, možemo testirati da li su implementacije u programskim jezicima C ili Lua semantički ekvivalentne specifikaciji. Izlaz rada LICC za verifikaciju implementacije algoritma `swap` u programskom jeziku Lua u odnosu na specifikaciju u pseudo-jeziku se može videti na slici 5.24. Primetimo da su prisutna upozorenja o odudaranju tipova promenljivih i funkcija — Lua nije striktno tipiziran jezik, a specifikacija nalaže da su globalne promenljive celi brojevi, dok su u implementaciji one tipa `object`, dok funkcija `swap` može imati i povratnu vrednost. S obzirom na prirodu skript jezika ovakve razlike nisu prijavljene kao greške, već samo kao upozorenja.



Slika 5.23: Vizualni prikaz generisanja AST od izvornih kodova sa slike 5.22 redom.



Slika 5.24: Semantičko poređenje implementacija sa slike 5.22 (Lua u odnosu na pseudo-jezik).

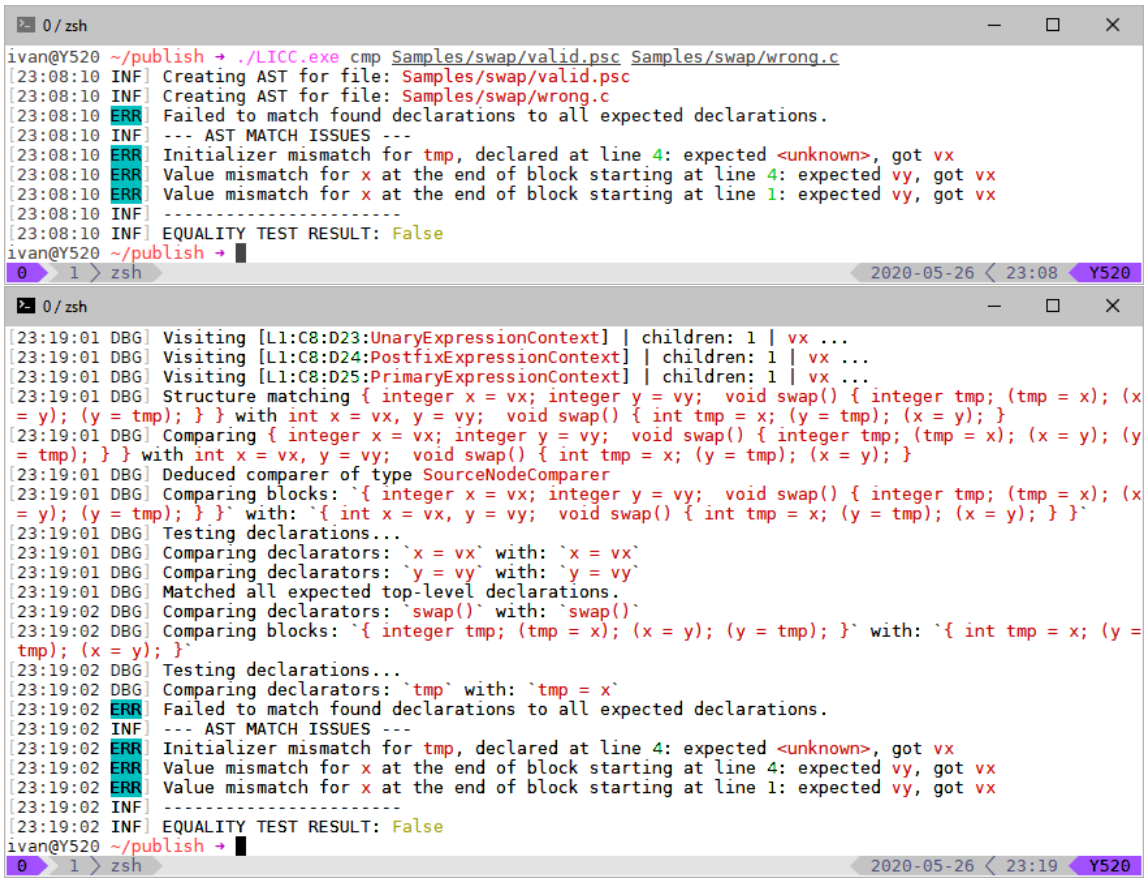
Ukoliko pak izvorni kôd ne odgovara specifikaciji, LICC će dati detaljan spisak razlika, koje su često tražene greške. U nekim slučajevima je moguće da je sematnička ekvivalentnost održana iako stabla imaju značajne razlike — LICC će prijaviti sve te razlike kao greške iako one to možda nisu. Izlaz za poređenje nevalidne implementacije algoritma `swap` sa slike 5.25 u odnosu na specifikaciju se može videti na slici 5.26. Vidimo da jedna od globalnih promenljivih nije pravilno zamenila vrednost, što se detektuje dvaput — po jednom za svaki od blokova u izvornom kodu. Dodatno, prijavljena je i greška o odudaranju izraza inicijalizatora za promenljivu `tmp`.

```
1  int x = vx, y = vy;
2  void swap() { int tmp = x; y = tmp; x = y; }
```

Slika 5.25: Nevalidna implementacija algoritma `swap` (C).

Ukoliko imamo već verifikovanu implementaciju algoritma u jednom programskom jeziku, može se desiti potreba za prelaskom na novije tehnologije što uključuje i prepisivanje algoritma sa jednog programskog jezika na drugi. LICC se može iskoristiti za poređenje tih implementacija, konkretno za algoritam `swap` na slici 5.27 se može videti rezultat poređenja implementacija u programskim jezicima C i Lua, pri čemu je takođe prikazan izlaz koji se dobija ukoliko se navede opcija `-v`.

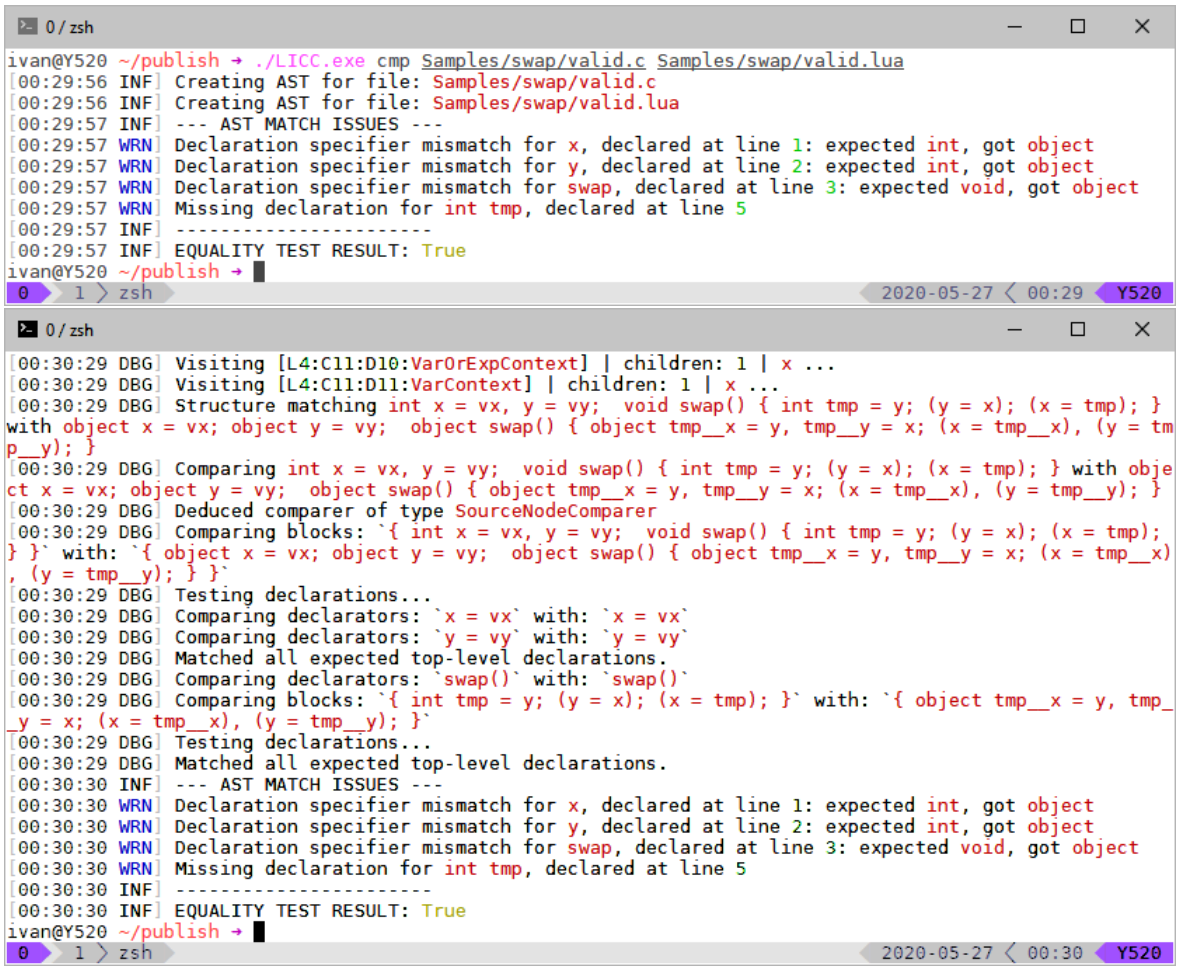
Još jedan slučaj upotrebe LICC može biti verifikacija međuverzija koda u procesu refaktorisanja. LICC pretpostavlja strukturnu sličnost kodova, što u procesu refaktorisanja često implicitno važi, ili barem važi u malim koracima između polazne i finalne verzije nakon refaktorisanja.



Slika 5.26: Semantičko poređenje nevalidne implementacije sa slike 5.25 u odnosu na specifikacije sa slike 5.22 – C (gore) i pseudojezik (dole).

Kompletan prikaz rada alata LICC za algoritam razmene vrednosti se može naći u okviru repozitorijuma na adresi `https://github.com/ivan-ristovic/LICC/LICC.Core/Samples/swap`. U okviru ovog direktorijuma su, uz ispravne i izmenjene izvorne datoteke algoritma, pružene i slike izlaza alata LICC za svaki par izvornih datoteka.

Kao primer prikaza poređenja deklaratora (promenljivih, nizovnih i funkcijskih), dat je primer koji se može pogledati na stranici `https://github.com/ivan-ristovic/LICC/LICC.Core/Samples/declarations`. Ovaj primer ilustruje poređenje više naredbi deklaracija čiji redosled nije nužno isti u izvornim datotekama, kao i detektovanje razlika u deklaratorima. U okviru ovog primera takođe se pokazuje i evaluacija izraza, kroz inicijalizatore celobrojnih i nizovnih promenljivih. Pošto je reč o deklaracijama, pružene su samo izvorne datoteke u



Slika 5.27: Semantičko poređenje implementacija sa slike 5.22 — Lua u odnosu na C (gore), Lua u odnosu na pseudojezik (dole, sa detaljnim ispisom).

programskom jeziku C i to validni primer, primer sa greškom kao i refaktorisani primer uz rezultate međusobnog poređenja parova izvornih datoteka.

Kao primer prikaza poređenja definicija funkcija sa parametrima, dat je primer koji se može pogledati na stranici `https://github.com/ivan-ristovic/LICC/LICC.Core/Samples/fparams`. Ovaj primer ilustruje poređenje funkcija čiji parametri utiču na rezultate analize unutar definicije funkcije. Kao primer prikaza poređenja kompleksnih naredbi, dat je primer koji se može pogledati na stranici `https://github.com/ivan-ristovic/LICC/LICC.Core/Samples/conditions`. Ovaj primer ilustruje poređenje naredbi grananja, pri čemu se detektuju izmene u uslovima grananja ali i odvojenim granama. Izvorne datoteke u okviru ova dva primera su pisane u programskim jezicima C, Lua i pseudojeziku i sastoje se od

validnog primera, primera sa greškom i refaktorisanog primera uz rezultate međusobnog poređenja parova izvornih datoteka. Treba napomenuti da je provera bazirana na uparivanju grana uslova a zatim njihovom poređenju, što znači da se naredbe grananja koje imaju zamenjene grane i negiran uslov neće biti detektovane kao ekvivalentne.

Glava 6

Zaključak

U tezi je opisan način posmatranja apstrakne sintakse programa kroz AST, opisan je proces kreiranja AST od proizvoljne gramatike programskog jezika i opšte-prihvaćen interfejs za obilazak istog. Opisan je model opšte AST apstrakcije sa ciljem dovođenja imperativnih i skript jezika na isti nivo apstrakcije. Ova apstrakcija je korišćena za određivanje semantičke ekvivalentnosti strukturno sličnih segmenata koda kroz naivni algoritam poređenja vrednosti na krajevima blokova.

Kao glavni doprinos teze, implementiran je računarski program za kreiranje opšteg AST od izvorne datoteke, serijalizaciju i prikaz istog. Mehanizam dobijanja opšteg AST omogućava jednostavno proširenje za već postojeće ali i za proizvoljne gramatike, kroz implementaciju adaptera za tu gramatiku koji služi kao posrednik između stabla parsiranja i opšteg AST. Dodatno, kao jedna primena opšte AST apstrakcije, implementiran je opisani algoritam za semantičko poređenje kroz proširiv model upoređivača tipova opštih AST čvorova. Priloženo je par primera upotrebe na segmentima koda programskih jezika C, Lua i pseudojezika (kao primera proizvoljnog programskog jezika, definisanog specifično za ovaj rad).

Naredni koraci u dizajniranju modela opšte apstrakcije bi bili usmereni na podršku za korisnički definisane tipove kroz čvorove za opis klasa, struktura i enumeracija. Takođe, klase se u skript jezicima često izbegavaju tako što se podaci smeste u mapu gde ključevi imitiraju atribute klase. Stoga bi bilo poželjno imati i interfejs za kreiranje mape objekta od datog klasnog čvora i obrnuto. Postoji i potreba za apstrahovanjem čestih struktura podataka kao što su skupovi, torke i redovi sa prioritetom. Na taj način, ako se u jednom programu koristi niz, a u drugom lista sa definisanim indeksnim pristupom, moguće je vršiti analizu uz potencijalno upozorenje o gubitku na efikasnosti.

Literatura

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 2002.

[2] ANTLR4. `https://www.antlr.org/`.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques, 2018.

[4] A Biere, M Heule, and H Van Maaren. *Handbook of Satisfiability*. IOS Press, 2009.

[5] BYACC. `https://invisible-island.net/byacc/byacc.html`.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.

[7] Clang. `https://clang.llvm.org/`.

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

[9] 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.

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

[11] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley, 2015.

[12] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer-Verlag London Ltd., 2010.

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

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

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

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

[17] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson, 2007.

[18] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[19] LexYacc. `http://dinosaur.compilertools.net/`.

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

[21] Terence Parr and Kathleen Fisher. Ll(*). *ACM SIGPLAN Notices*, 46(6):425, 2011.

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

[23] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*. Springer Berlin Heidelberg, 1990.

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

[25] William McCastline Waite and Gerhard Goos. *Compiler construction*. Springer, 1985.

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.