# Upute za izradu laboratorijskih vježbi iz predmeta *Prevođenje programskih jezika*

*Autori:*
Ivan Budiselić, Ivan Žužak

*Datum posljednje izmjene:*
11. listopada 2012.

Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroelektroniku, inteligentne i računalne sustave

# Sadržaj

# 1. Uvod

Ovaj dokument namijenjen je studentima predmeta *Prevođenje programskih jezika* na Fakultetu elektrotehnike i računarstva. U dokumentu je opisan zadatak laboratorijskih vježbi predmeta te organizacija, provedba i ocjenjivanje.

## 1.1 Cilj laboratorijskih vježbi

Studenti tijekom vježbi izrađuju vlastiti jezični procesor (preciznije, kompilator) s ciljem boljeg razumijevanja kako jezični procesor radi. Kao rezultat laboratorijskih vježbi studenti predaju pojedine dijelove jezičnog procesora, a konačno i cijeli jezični procesor.

Kako bi se projekt izrade jezičnog procesora najviše moguće naslonio na prethodno znanje stečeno u dosadašnjem tijeku studija, jezični procesor koji će se izrađivati na laboratorijskim vježbama treba prevoditi zadani podskup jezika *C* u mnemonički jezik procesora FRISC. Prevedeni programi moći će se izvoditi na simulatoru koji će biti dostupan na stranicama predmeta prije početka rada na generiranju FRISC koda.

Jezični procesor neće imati grafičko korisničko sučelje, nego će s korisnikom komunicirati putem standardnog ulaza i izlaza kao što je i uobičajeno.

## 1.2 Grupe za laboratorijske vježbe

Studenti laboratorijske vježbe rade u grupama. Svaki student u grupi praktično izrađuje neke dijelove jezičnog procesora prema dogovoru unutar grupe, ali mora biti upoznat sa svim dijelovima jezičnog procesora. Grupu vodi jedan od studenata, a rad voditelja nadziru asistenti. Voditelj grupe u pravilu se bira na temelju rezultata iz predmeta *Uvod u teoriju računarstva* prethodne akademske godine.

## 1.3 Odabir jezika izgradnje jezičnog procesora

Svaka grupa za jezik izgradnje jezičnog procesora, tj. programski jezik u kojem će jezični procesor grupe biti ostvaren, bira između jezika *C*, *C++*, *Java*, *C#* i *Python* (2.x ili 3.x). *C#* rješenja prevodit će se kompilatorom iz Mono platforme i izvodit će se na Mono

platformi[1]. Ako niste sigurni koji programski jezik odabrati, vjerojatno je najsigurniji izbor *Java*.

Pretpostavka je da iz položenih predmeta na nižim godinama studija svi studenti koji pristupaju laboratorijskim vježbama znaju *C*. Ipak, *C* u praksi predmeta nije čest odabir za jezik izgradnje. Prvenstveni razlog za to je što je čak i jednostavan jezični procesor relativno složen program koji nužno koristi strukture podataka koje dinamički rastu. U jeziku *C* to zahtijeva intenzivno korištenje dinamičke alokacije memorije i pokazivača što može izazvati dosta poteškoća. Ako vam se čini da bi najbolji izbor za vašu grupu bio *C*, razmislite o korištenju jezika *C++*. *C++* podržava više programskih paradigmi te se kao takav može koristiti i u proceduralnom stilu kao *C*, bez primjene objektno orijentirane paradigme. Međutim, *C++* u standardnoj biblioteci sadrži većinu struktura podataka koje će vam trebati u izgradnji jezičnog procesora te omogućiti u velikoj mjeri ili potpuno izbjegavanje dinamičke alokacije memorije i rada s pokazivačima.

## 1.4 Provedba laboratorijskih vježbi

Provedba laboratorijskih vježbi organizirana je u četiri laboratorijske cjeline te završnu predaju ostvarenog jezičnog procesora. U svakoj laboratorijskoj cjelini grupa izrađuje zadani dio jezičnog procesora te ostvareno predaje na računalno ocjenjivanje ili na predajli laboratorijske vježbe objašnjava izgradnju i prikazuje rad ostvarenog dijela, ovisno o cjelini. Zadaci za pojedine laboratorijske cjeline detaljno su opisani u kasnijim poglavljima, a dijele se na:

**I laboratorijska cjelina:** izgradnja leksičkog analizatora

**II laboratorijska cjelina:** izgradnja sintaksnog analizatora

**III laboratorijska cjelina:** izgradnja semantičkog analizatora

**IV laboratorijska cjelina:** izgradnja generatora ciljnog programa

Završna predaja ostvarenog jezičnog procesora uključuje predaju izvornog koda i izvodive verzije ostvarenog jezičnog procesora.

Studenti tijekom izgradnje jezičnog procesora mogu konzultirati asistente koji vode laboratorijske vježbe, e-mailom ili na konzultacijama.

## 1.5 Kako pripremiti rješenje laboratorijskih vježbi za računalno ocjenjivanje

Za potrebe računalnog ocjenjivanja, vaše rješenje će s ispitnim sustavom komunicirati putem *standardnog ulaza* (*stdin*) i *standardnog izlaza* (*stdout*). Ispitni sustav će pri pokretanju programa na standardni ulaz preusmjeriti odgovarajuću ulaznu datoteku i standardni

[1] Neće biti moguće predati *Visual Studio* projekt, nego će se organizacija rješenja morati prilagoditi u skladu s uputama koje su dane za svaku pojedinu vježbu.

izlaz preusmjeriti u datoteku koja će se onda uspoređivati s očekivanim izlazom. Iz tog razloga od presudne je važnosti da se vaš program strogo drži formata ispisa zadanog za pojedinu vježbu i da na standardni izlaz ne ispisuje ništa osim onog što se traži. Za sav ostali ispis (npr. ispis grešaka korisniku ili ispis za praćenje rada programa) koristite standardni izlaz za greške (*stderr*).

Zadnji znak u svakoj ulaznoj datoteci bit će znak novog retka. To znači da ako čitate ulaz redak po redak nekom funkcijom koja čuva znakove novog retka (na primjer `fgets` u *C*-u ili `readline` u *Pythonu*), možete računati na to da će svaki redak, uključujući i zadnji, završavati znakom novog retka.

Ulazne datoteke će uvijek biti zadane u točno onakvom formatu kao što je opisano u ovom dokumentu, tj. nije potrebno provjeravati je li ulaz ispravno formatiran i ispisivati greške u suprotnom.

Nadalje, **testirajte svoja rješenja** prije predaje. To što program radi za primjere koje ćete naći u ovoj uputi ne mora značiti da je program točan. Iako se ova tvrdnja čini očita, čest je slučaj da studenti predaju rješenja koja uopće nisu testirana i sadrže elementarne greške koje su se lako mogle ukloniti već i uz malo testiranja nad vlastitim primjerima.

# 2. Prva laboratorijska vježba

Tema prve laboratorijske vježbe je leksička analiza. Grupe izrađuju generator leksičkog analizatora sličan programu Lex. U nastavku je opisano kako generator leksičkog analizatora i generirani leksički analizator trebaju raditi i dani su savjeti za implementaciju. Ova uputa pretpostavlja da je čitatelj upoznat sa ispredavanim gradivom o leksičkoj analizi. S obzirom na opsežnost uputa, preporuča se jednom ih pročitati od početka do kraja, a nakon toga se upute mogu koristiti kao referenca prilikom implementacije.

Ukupni rezultat laboratorijskih vježbi bit će jezični procesor koji prevodi podskup jezika *C* u mnemonički jezik procesora FRISC. U prvoj laboratorijskoj vježbi izrađuje se relativno općeniti generator leksičkog analizatora koji će se onda iskoristiti i za generiranje leksičkog analizatora za zadani podskup jezika *C*. Nakon završetka sve četiri laboratorijske vježbe, grupe će generirani leksički analizator povezati s ostalim dijelovima u cjeloviti jezični procesor.

Na prvoj laboratorijskoj vježbi izrađuje se *generator* leksičkog analizatora umjesto specijaliziranog leksičkog analizatora za podskup jezika *C*. Za leksički složene jezike, a tu se prvenstveno misli na dozvoljeni slobodan način zapisa ulaznog programa, izgradnja specijaliziranog leksičkog analizatora nije bitno lakša od izgradnje donekle općenitog generatora leksičkog analizatora. S druge strane, izgradnja generatora daje bolji uvid u način rada leksičke analize bilo kojeg programskog jezika. Također, izgradnja generatora omogućuje bolje razumijevanje i uspješnije korištenje postojećih profesionalnih generatora leksičkih analizatora.

## 2.1 Generator leksičkog analizatora


Slika 2.1: Način rada generatora leksičkog analizatora.

Način rada generatora leksičkog analizatora prikazan je slikom 2.1. Generator leksičkog analizatora na standardnom ulazu dobiva opis procesa leksičkog analizatora. Opis procesa

leksičkog analizatora zadan je tekstualnom datotekom (dalje Ulazna Datoteka)[1]. Izlaz iz generatora leksičkog analizatora treba biti izvorni kod leksičkog analizatora napisan u jeziku izgradnje. Ulazna Datoteka bit će zadana u sljedećem formatu:

*regularne definicije*

%X *stanja leksičkog analizatora*

%L *imena leksičkih jedinki*

*pravila leksičkog analizatora*

U nastavku je objašnjeno kako pojedini dio datoteke izgleda.

### 2.1.1 Regularne definicije

Regularne definicije čine nadgradnju nad regularnim izrazima sa ciljem jednostavnijeg i preglednijeg opisa nekog regularnog jezika. Regularne definicije opisane su u udžbeniku predmeta "Uvod u teoriju računarstva" [2, poglavlje 2.3.2]. U nastavku su opisani regularni izrazi i oblik regularnih definicija koje će se koristiti u Ulaznoj Datoteci.

#### Oblik regularnih izraza

Kako bi se pojednostavio postupak pretvorbe regularnih izraza koji se koriste za definiciju leksičkih jedinki u konačni automat, svi regularni izrazi u Ulaznoj Datoteci koristit će isključivo:

- izbor podizraza koristeći operator `|`
- nadovezivanje (slijedno napisani znakovi ili grupe znakova, bez posebnog operatora)
- Kleeneov operator ponavljanja `*` (ponavljanje nula ili više puta)
- grupiranje koristeći oble zagrade `(` i `)`

Kao što je uobičajeno, prednost operatora ponavljanja `*` veća je od prednosti nadovezivanja, a prednost nadovezivanja veća je od prednosti operatora izbora `|`. Odgovarajuća interpretacija regularnog izraza koja nije u skladu s ovim prednostima, može se postići grupiranjem koristeći zagrade.

Dodatno, zbog jednostavnosti, za prazan niz $\varepsilon$ će se koristiti znak `$` (dolar). Zbog toga nigdje u regularnom izrazu neće biti praznih podizraza (npr. `a||b` će biti zapisano kao `a|$|b`). U izrazima se neće pojaviti uzastopni operatori ponavljanja (na primjer `a**`).

[1] U skladu s napomenama u uvodu, ostvareni generator leksičkog analizatora treba čitati opis procesa leksičkog analizatora sa standardnog ulaza, a Ulazna Datoteka se onda preusmjerava na standardni ulaz prilikom pokretanja generatora.

#### Regularne definicije u Ulaznoj Datoteci

U Ulaznoj Datoteci bit će jedna regularna definicija po retku, sljedećeg oblika:
`{`*imeRegularneDefinicije*`}␣`*regularniIzraz*

Pritom, *imeRegularneDefinicije* može sadržavati mala i velika slova engleske abecede, a *regularniIzraz* je regularni izraz oblika u skladu s prije opisanim pravilima pri čemu osim pojedinačnih znakova smije sadržavati i proizvoljan broj referenci na prethodno definirane regularne definicije (vidi primjer ispod). Početni znak `{` će se nalaziti u prvom stupcu određenog retka, a između znaka `}` i početka regularnog izraza nalazit će se točno jedan razmak. U području regularnih definicija u Ulaznoj Datoteci neće se nalaziti niti jedan prazan redak.

Na primjer, dio regularnih definicija u Ulaznoj Datoteci može izgledati ovako:

```
1 {oktalnaZnamenka} 0|1|2|3|4|5|6|7
2 {dekadskaZnamenka} {oktalnaZnamenka}|8|9
3 {hexZnamenka} a|b|c|d|e|f|{dekadskaZnamenka}|A|B|C|D|E|F
```

Ispis 2.1: Primjer regularnih definicija u Ulaznoj Datoteci.

Kao što se vidi u zadnjem retku primjera u ispisu 2.1, referenca na prethodno definiranu regularnu definiciju može se nalaziti bilo gdje u regularnom izrazu.

### 2.1.2 Stanja leksičkog analizatora

Za pojedine klase leksičkih jedinki nužno je promatrati lijevi kontekst, tj. dio ulaznog niza koji je pročitan prije podniza koji čini leksičku jedinku. Jedan od načina praćenja lijevog konteksta koji se često koristi u leksičkim analizatorima su stanja leksičkog analizatora.

Za potrebe generatora leksičkog analizatora koji se gradi u sklopu prve laboratorijske vježbe, u Ulaznoj Datoteci će nakon zadnje regularne definicije biti redak u kojem su definirana stanja koja će koristiti leksički analizator. Redak će započeti znakovima `%X` i točno jednim razmakom, nakon kojeg slijedi niz jedne ili više oznaka stanja odvojenih točno jednim razmakom (vidi primjer ispod). Oznake stanja će se sastojati od malih i velikih slova engleske abecede i započinjat će nizom `S_` (to će ujedno biti jedino mjesto u imenu stanja na kojem će biti znak `_` (engl. *underscore*)). Prvo stanje navedeno nakon `%X` je početno stanje leksičkog analizatora. Na primjer, definicija stanja leksičkog analizatora može izgledati kao u ispisu 2.2.

```
1 %X S_pocetno S_komentar S_unarniMinus
```

Ispis 2.2: Primjer imena stanja u Ulaznoj Datoteci.

### 2.1.3 Imena leksičkih jedinki (uniformni znakovi)

Odmah u sljedećem retku nakon definicije stanja leksičkog analizatora, u Ulaznoj Datoteci nalazit će se redak koji definira imena leksičkih jedinki koje generirani leksički analizator

treba koristiti. U nastavku upute ponekad se umjesto sintagme *ime leksičke jedinke* koristi *uniformni znak* i važno je uočiti da se obje sintagme odnose na istu stvar. Redak će započeti znakovima `%L` i točno jednim razmakom, nakon kojeg slijedi niz jednog ili više imena leksičkih jedinki odvojenih točno jednim razmakom (vidi primjer ispod). Imena leksičkih jedinki će se sastojati od malih i velikih slova engleske abecede i znakova `_` (engl. *underscore*). Na primjer, definicija imena leksičkih jedinki može izgledati kao u ispisu 2.3.

```
1 %L IDENTIFIKATOR brojcanaKonstanta znakovnaKonstanta OP_PLUS
```

Ispis 2.3: Primjer imena stanja u Ulaznoj Datoteci.

### 2.1.4 Pravila leksičkog analizatora

Pravila leksičkog analizatora definiraju sve leksičke jedinke ulaznog jezika i definiraju na koji način će generirani leksički analizator pohraniti sve potrebne podatke vezane uz leksičku jedinku. U Ulaznoj Datoteci će pravila leksičkog analizatora biti navedena nakon retka koji definira imena leksičkih jedinki u sljedećem obliku:
<*imeStanja*>*regularniIzraz*
{
*argumentiAkcije*
}

U datoteci će biti proizvoljan broj pravila, jedno iza drugog bez praznih redaka između pravila. *imeStanja* odgovarat će nekom od stanja definiranih ranije i bit će okruženo znakovima < i >. *regularniIzraz* slijedit će neposredno nakon znaka > (bez razmaka) i bit će u prethodno opisanom formatu. U regularnom izrazu mogu se pojaviti imena definirana u regularnim definicijama okružena znakovima { i } (vidi primjer ispod).

*Napomena: Sva pravila* leksičkog analizatora imat će zadano stanje na početku pravila. Određeno pravilo je aktivno ako i samo ako se leksički analizator nalazi u odgovarajućem stanju (vidi savjete za implementaciju ispod). Prvenstveni razlog za ovakvu definiciju pravila je uniformnost što omogućuje jednostavniju implementaciju.

U sljedećem retku definicije pravila nalazit će se znak { koji započinje dio za definiciju argumenata akcije. Svi retci argumenata akcije počet će u prvom stupcu i niti jedan neće započinjati znakom }. Završetak dijela argumenata akcije bit će jednoznačno određen retkom koji sadrži samo znak }. Argumenti akcije određuju što generirani leksički analizator treba učiniti kada prepozna odgovarajuću leksičku jedinku. Svaka akcija imat će jedan do četiri argumenata, svaki u svom retku. U prvom retku argumenata akcije uvijek će se nalaziti ime pripadne leksičke jedinke (jedno od imena definiranih ranije u Ulaznoj Datoteci) ili znak − (minus). U slučaju da je u akciji navedeno ime leksičke jedinke, generirani leksički analizator treba zadanu leksičku jedinku dodati u niz uniformnih znakova, kao što je detaljnije objašnjeno u poglavlju 2.2 ove upute. S druge strane, znak minus označava da pročitani dio ulaznog niza treba odbaciti i da ne predstavlja leksičku jedinku. Odbacivanje će se koristiti za izbacivanje komentara, bjelina i slično. Pojedine akcije imat će jedan do tri posebna argumenta koji redom omogućuju brojanje redaka, upravljanje stanjem leksičkog analizatora i razrješavanje nejednoznačnosti.

Prvi posebni argument je `NOVI_REDAK` koji označava leksičkom analizatoru da je u

izvornoj datoteci programa koji se prevodi došlo do promjene retka. Retci se broje kako bi jezični procesor mogao ispisivati korisne podatke o mjestu pogreške, i tijekom leksičke analize i tijekom ostalih faza rada. Povećani broj retka odnosi se tek na sljedeću leksičku jedinku, tj. ako akcija prepoznaje neku leksičku jedinku (u prvom retku argumenata akcije nalazi se neko ime leksičke jedinke, a ne znak $-$) i ima argument `NOVI_REDAK`, za broj retka te leksičke jedinke zapisuje se stari broj retka, prije uvećanja.

Drugi posebni argument je `UDJI_U_STANJE` *imeStanja* kojim leksički analizator prelazi iz trenutnog stanja u stanje *imeStanja*. Niz znakova *imeStanja* bit će odvojen od niza `UDJI_U_STANJE` točno jednim razmakom. Treći posebni argument je `VRATI_SE` *naZnak* koji određuje da se od pročitanih znakova u leksičku jedinku treba grupirati prvih *naZnak* znakova, a ostali znakovi vraćaju se u ulazni niz, kao da nisu ni pročitani (vidi primjer). Ova naredba je ekvivalentna naredbi `yyless(naZnak)` u Lexu. *naZnak* će biti dekadski cijeli broj bez vodećih nula i bit će odvojen od niza `VRATI_SE` točno jednim razmakom. Svi posebni argumenti prikazani su u primjeru, uz dodatne komentare u savjetima za implementaciju.

### 2.1.5 Specijalni znakovi

Pojedini znakovi se u opisu regularnih definicija i regularnih izraza u pravilima leksičkog analizatora koriste kao posebne oznake. Kako bi generirani leksički analizator mogao te iste znakove prepoznati u izvornom kodu programa koji se prevodi, nužno je uvesti sustav prefiksiranja (engl. *escaping*). Kao što je uobičajeno, specijalni znakovi prefiksiraju se znakom `\` (engl. *backslash*) da bi dobili svoje izvorno značenje. Dodatno, kako bi se u regularnom izrazu mogao prepoznati i sam znak `\`, on se također mora prefiksirati dodatnim znakom `\` (vidi primjer ispod). Specijalni znakovi koji se koriste u generatoru leksičkog analizatora su:

```
( ) { } | * $ \
```

Osim ovih znakova, generator leksičkog analizatora mora u Ulaznoj Datoteci prepoznati i niz znakova `\n` kao znak za novi redak i niz znakova `\t` kao znak tab. Dodatno, razmak ćemo označavati nizom `\_` (*backslash underscore*) kako bi Ulazna Datoteka bila čitljivija i kako ne bi bilo problema s uređivačima teksta koji brišu razmake na krajevima redaka.

### 2.1.6 Primjer

U primjeru je prikazana Ulazna Datoteka jednostavanog jezika za računanje matematičkih izraza. Jezik podržava samo binarno oduzimanje i unarni minus pa ćemo ga zvati *minusLang*. Unarni minus može se pojaviti više puta uzastopce. Podržani su samo cijeli brojevi u dekadskom, oktalnom ili heksadekadskom zapisu. Grupiranje se ostvaruje korištenjem oblih zagrada. Dozvoljen je slobodan način zapisa matematičkog izraza kroz više redaka i s proizvoljnim brojem praznina. Također su dozvoljeni i komentari koji prolaze kroz proizvoljan broj redaka, a ograđeni su nizovima `#|` i `|#`. Zbog jednostavnosti Ulazne Datoteke, u jeziku nije dozvoljeno da se komentar pojavi ispred unarnog minusa.

Primjer programa pisanog u zadanom jeziku prikazan je u ispisu 2.4. Ulazna Datoteka za leksičku analizu ovog jezika mogla bi izgledati ovako.

```
#| ovo je primjer |#
3 -  -0x12 - ( #| ovdje ce doci grupirane
    operacije |#
3- -
--076) #| 3 - ---076 = 3 - -076 = 3 + 076 |#

```

Ispis 2.4: Primjer programa pisanog u jeziku *minusLang*.

U području regularnih definicija, definirana je jedinstvena definicija broja koja prihvaća zapis broja u sve tri baze (oktalne brojeve ne trebamo posebno definirati jer izgledaju kao dekadski s vodećom nulom). To znači da leksička analiza neće razlikovati brojevne baze, odnosno neće provjeravati sadrži li oktalni broj znamenke `8` ili `9`, nego će te provjere odraditi neka kasnija faza rada jezičnog procesora, tipično semantička analiza[2].

Na početku regularnog izraza definicije `{sviZnakovi}` nalaze se svi specijalni znakovi, prefiksirani znakom \, kao što je ranije opisano. Izbacivanje komentara odrađuje se s četiri pravila prikazana u ispisu 2.5.

```
<S_pocetno>#\|
{
-
UDJI_U_STANJE S_komentar
}
<S_komentar>\|#
{
-
UDJI_U_STANJE S_pocetno
}
<S_komentar>\n
{
-
NOVI_REDAK
}
<S_komentar>{sviZnakovi}
{
-
}
```

Ispis 2.5: Pravila za izbacivanje komentara iz ulaznog programa.

Prva akcija prepoznaje početni niz znakova koji označava komentar. Za prepoznavanje specijalnog znaka `|` koristi se prefiks \. Slično, drugo pravilo pronalazi kraj komentara i vraća leksički analizator u početno stanje. S obzirom na to da se komentari mogu provlačiti kroz proizvoljan broj redaka, nužno je i unutar stanja za prepoznavanje komentara nastaviti brojati retke. Brojanje redaka ostvaruje se trećim pravilom. Konačno, u komentaru se može pojaviti mnogo znakova koji inače nisu dozvoljeni van komentara.

[2] O ovom i sličnim pitanjima bit će govora na predavanjima predmeta tijekom semestra.

Važno je primijetiti da četvrto pravilo prepoznaje jedan po jedan znak unutar komentara koristeći regularnu definiciju `{sviZnakovi}`. Na taj način će biti moguće prepoznati niz znakova koji predstavlja kraj komentara (prioritet po duljini, a i po redoslijedu pravila) i znak kraja retka (prioritet po redoslijedu pravila). Naime, i znak novog retka i oba znaka koji predstavljaju kraj komentara nalaze se u regularnom izrazu za definiciju `{sviZnakovi}`. Kada bi se za četvrto pravilo koristio složeniji regularni izraz, na primjer `{sviZnakovi}{sviZnakovi}*` s ciljem prepoznavanja cijelog komentara, onda bi cijeli program nakon prvog početka komentara bio preskočen zbog pririteta po duljini (osim ako komentar sadrži nula znakova).

Prepoznavanje unarnog minusa riješeno je većim dijelom kroz pravila prikazana u ispisu 2.6.

```
<S_pocetno>-{bjelina}*-
{
OP_MINUS
UDJI_U_STANJE S_unarni
VRATI_SE 1
}
<S_pocetno>\({bjelina}*-
{
LIJEVA_ZAGRADA
UDJI_U_STANJE S_unarni
VRATI_SE 1
}
<S_unarni>-
{
UMINUS
UDJI_U_STANJE S_pocetno
}
```

Ispis 2.6: Pravila za detekciju unarnog minusa.

U prva dva pravila definirana su dva mjesta gdje se unarni minus može pojaviti — iza binarnog operatora oduzimanja ili iza otvorene zagrade. Između lijevog i desnog znaka može se pojaviti proizvoljan broj bjelina. U oba pravila koristi se posebni argument `VRATI_SE` *naZnak*. Na primjer, ako leksički analizator u stanju `S_pocetno` pročita niz
`-\t\t \n -`
tada akcija prvog pravila treba prepoznati *samo prvi minus* kao jedinku `OP_MINUS` (zbog `VRATI_SE 1`, što znači da se "glava za čitanje" treba vratiti na znak s indeksom `1` u pročitanom nizu, pri čemu indeksi počinju od `0`). Dodatno, leksički analizator treba ući u stanje `S_unarni` i vratiti sve znakove osim prvog minusa u ulazni niz (vidi savjete za implementaciju). Bitno je primijetiti da je vraćanje znakova u ulazni niz ključan element ove akcije. Na taj način će pravilo za brojanje redaka u stanju `S_unarni` dobiti pročitani znak kraja retka i brojanje redaka će raditi ispravno.

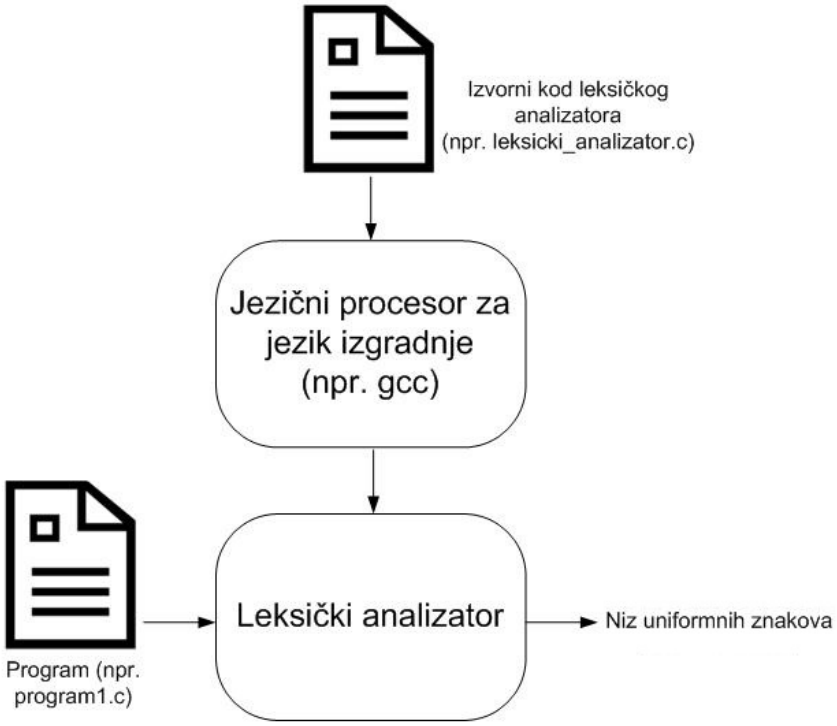
S obzirom na to da je u definiciji jezika navedeno da se unarni minus može pojavljivati više puta uzastopce, nužno je uvesti i dodatno pravilo (zadnje pravilo u datoteci) koje je prikazano u ispisu 2.7.

```
<S_unarni>-{bjelina}*-
{
UMINUS
VRATI_SE 1
}
```

Ispis 2.7: Pravila za detekciju ponavljanja unarnog minusa.

## 2.2 Leksički analizator

Izlaz iz generatora leksičkog analizatora treba biti izvorni kod leksičkog analizatora pisan u jeziku izgradnje jezičnog procesora. Dio koda leksičkog analizatora, primjerice deklaracije tipova podataka, uključivanje biblioteka te algoritam leksičkog analizatora, potpuno je neovisno o Ulaznoj Datoteci. Na osnovi Ulazne Datoteke potrebno je generirati tablicu konačnog automata koju će leksički analizator koristiti u algoritmu grupiranja znakova u leksičke jedinke. Način konstruiranja konačnog automata ($\varepsilon$-NKA) iz regularnog izraza opisan je u [2, poglavlje 2.2.2], a pomoć za implementaciju tog algoritma navedena je u savjetima za implementaciju u ovom dokumentu.




Slika 2.2: Način rada leksičkog analizatora.

Slika 2.2 prikazuje dinamiku rada leksičkog analizatora. Izvorni kod leksičkog analizatora dobiven primjenom izgrađenog generatora leksičkog analizatora treba prevesti u izvodivi oblik primjenom odgovarajućeg jezičnog procesora (ako je jezik izgradnje interpretirani jezik onda se ovaj korak preskače). Izvodivi leksički analizator čita izvorni kod nekog programa pisanog u zadanom izvornom jeziku (na primjer jeziku $C$) *sa standardnog*

*ulaza*. Rezultat izvođenja leksičkog analizatora u jezičnom procesoru je *niz uniformnih znakova*[3].

### 2.2.1 Ispis leksičkog analizatora

Za potrebe prve laboratorijske vježbe, leksički analizator *na standardni izlaz* treba za svaki element niza uniformnih znakova ispisati redak sljedećeg oblika:

*uniformniZnak*␣*redakPrograma*␣*leksičkaJedinka*

Pritom je *uniformniZnak* odgovarajući uniformni znak iz Ulazne Datoteke, *redakPrograma* broj retka u ulaznom programu u kojem se leksička jedinka nalazi ako se **retci broje od 1** i *leksičkaJedinka* je niz znakova ulaznog programa koji su grupirani u leksičku jedinku. Ta tri dijela retka međusobno su odvojeni sa po jednim razmakom. Primjer izlaza prikazan je u poglavlju 2.2.6.

### 2.2.2 Strukture podataka leksičkog analizatora

Leksički analizator na ulazu prima programski kod u jeziku opisanom Ulaznom Datotekom. Kao rezultat, leksički analizator treba stvoriti niz uniformnih znakova.

Kako leksički analizator neće graditi tablicu znakova, u nizu uniformnih znakova za svaku prepoznatu leksičku jedinku uz ime leksičke jedinke i redak u kojem se ta jedinka nalazi u ulaznom programu treba se nalaziti i niz znakova ulaznog programa koji su grupirani u leksičku jedinku.

### 2.2.3 Algoritam leksičkog analizatora

U udžbeniku su opisana dva algoritma leksičkog analizatora. U poglavlju 2.9.6, opisan je algoritam zasnovan na DKA, a u poglavlju 2.9.7 algoritam zasnovan na $\varepsilon$-NKA. Implementacijom algoritma zasnovanog na $\varepsilon$-NKA neće biti potrebno ostvarivati pretvorbu $\varepsilon$-NKA u DKA (koja je obrađena na predmetu "Uvod u teoriju računarstva") te je stoga taj algoritam predložen za prvu laboratorijsku vježbu (vidi savjete za implementaciju).

### 2.2.4 Razrješavanje nejednoznačnosti

Generirani leksički analizator nejednoznačnost treba rješavati koristeći pravila P2 i P3 iz poglavlja 2.9.3 i 2.9.4 udžbenika (prioritet po duljini prepoznatog niza i prioritet po poretku pravila).

[3]Na predavanjima i u udžbeniku ste vidjeli da leksički analizator može inicijalizirati tablicu znakova, ali to u ovim laboratorijskim vježbama neće biti potrebno. Tablica znakova će se ostvariti u kasnijim fazama rada kada postaje potrebna.

### 2.2.5 Postupak oporavka od pogreške

Za postupak oporavka od pogreše treba koristiti jednostavan postupak odbacivanja prvog znaka iz ulaznog programa kao što je opisano u poglavlju 2.9.5 udžbenika. Podatke o pogreški treba ispisivati isključivo na izlaz za greške (*stderr*). Dodatno, iako se u ispisu leksičkog analizatora eksplicitno ne navode greške niti nije vidljivo kako se analizator oporavlja od pogreške, ispravnost implementacije ovog postupka utjecat će na ispis za sve ulazne programe koji sadrže barem jednu leksičku pogrešku.

### 2.2.6 Primjer

Kao nastavak prije obrađenog primjera, prikazat ćemo rezultat rada leksičkog analizatora generiranog za primjer u poglavlju 2.1.6 ovih uputa, nad ulaznim programom koji je zbog preglednosti ponovljen u ispisu 2.8.

```
1 #| ovo je primjer |#
2 3 -  -0x12 - ( #| ovdje ce doci grupirane
3     operacije |#
4 3- -
5 --076) #| 3 - ---076 = 3 - -076 = 3 + 076 |#
6
```

Ispis 2.8: Primjer programa pisanog u jeziku *minusLang*.

Datoteka s ovim primjerom za potrebe testiranja može se pronaći ovdje. Leksički analizator u skladu s ranije opisanom Ulaznom Datotekom generira sljedeći niz uniformnih znakova:

| uniformni znak | redak | grupirani znakovi |
|---|---|---|
| OPERAND | 2 | 3 |
| OP_MINUS | 2 | - |
| UMINUS | 2 | - |
| OPERAND | 2 | 0x12 |
| OP_MINUS | 2 | - |
| LIJEVA_ZAGRADA | 2 | ( |
| OPERAND | 4 | 3 |
| OP_MINUS | 4 | - |
| UMINUS | 4 | - |
| UMINUS | 5 | - |
| UMINUS | 5 | - |
| OPERAND | 5 | 076 |
| DESNA_ZAGRADA | 5 | ) |

U drugom stupcu niza uniformnih znakova naveden je redak u kojem se uniformni znak nalazi u ulaznoj datoteci. U trećem stupcu navedeni su znakovi iz ulaznog programa koji su grupirani u leksičku jedinku.

Očekivani ispis na standardni izlaz prikazan je ovdje.

## 2.3 Način računalnog ocjenjivanja rješenja

Za potrebe računalnog ocjenjivanja, rješenje prve laboratorijske vježbe treba predati u jednoj *zip* datoteci. U korijenskom direktoriju trebaju se nalaziti **sve** datoteke s izvornim kodom **generatora** leksičkog analizatora. Korijenski direktorij *zip* datoteke mora sadržavati poddirektorij `analizator`. Poddirektorij `analizator` može inicijalno sadržavati dijelove implementacije leksičkog analizatora koji ne ovise o Ulaznoj Datoteci, ali i ne mora sadržavati ništa. Na primjer, za rješenje u jeziku *C*, sadržaj *zip* datoteke mogao bi izgledati kao u ispisu 2.9.

```
analizator/bar.c
analizator/bar.h
analizator/lekser.h
analizator/predlozak.txt
foo.c
foo.h
generator.c
```

Ispis 2.9: Primjer sadržaja *zip* datoteke. Imena datoteka *nisu važna*.

Za prevođene jezike izgradnje, sustav za računalno ocjenjivanje će prevesti predane datoteke u izvodivi oblik generatora leksičkog analizatora u korijenskom direktoriju. Za prethodni primjer, rezultat bi mogao izgledati kao u ispisu 2.10.

```
analizator/bar.c
analizator/bar.h
analizator/lekser.h
analizator/predlozak.txt
foo.c
foo.h
generator.c
generator.exe
```

Ispis 2.10: Primjer sadržaja korijenskog direktorija nakon prevođenja generatora.

Generatoru leksičkog analizatora na standardni ulaz se predaje opis procesa leksičkog analizatora, a generator priprema izvorni kod leksičkog analizatora u poddirektoriju `analizator`, kao u ispisu 2.11. Važno je uočiti da se *izlaz generatora* ne provjerava izravno — generator ne mora ispisati ništa, ali mora pripremiti tj. generirati analizator.

```
analizator/bar.c
analizator/bar.h
analizator/lekser.h
analizator/lekser.c
analizator/predlozak.txt
analizator/tablica.txt
foo.c
foo.h
generator.c
```

```
generator.exe
```

Ispis 2.11: Primjer sadržaja korijenskog direktorija nakon izvođenja generatora.

Nakon izvođenja generatora leksičkog analizatora, u poddirektoriju `analizator` mora se nalaziti sav potrebni izvorni kod za prevođenje leksičkog **analizatora**. Sustav će prevesti datoteke u poddirektoriju u izvodivi oblik leksičkog analizatora u istom poddirektoriju, kao što je prikazano u ispisu 2.12.

```
analizator/analizator.exe
analizator/bar.c
analizator/bar.h
analizator/lekser.h
analizator/lekser.c
analizator/predlozak.txt
analizator/tablica.txt
foo.c
foo.h
generator.c
generator.exe
```

Ispis 2.12: Primjer sadržaja korijenskog direktorija nakon izvođenja generatora.

Konačno, izvodivi leksički analizator na standardni ulaz dobiva izvorni kod programa i treba ispisati rezultat izvođenja kao što je ranije opisano.

Uočite da su **generator** i **analizator** dva *odvojena programa*. Nije dozvoljeno kopirati Ulaznu Datoteku u "generatoru" i onda je ponovno parsirati u "analizatoru", stvarati automate i tek onda analizirati ulazni program. Generator je taj program koji mora završiti definiciju analizatora. To se, na primjer, može ostvariti tako da generator generira programski kod koji u analizatoru inicijalizira tablice automata ili da generira datoteku s tablicama automata koju onda analizator čita[4].

Za rješenja u *Javi*, ulazna točka u generator **mora biti** u razredu `GLA`, a ulazna točka u analizator u razredu `LA`. Za rješenja u *Pythonu*, za izvođenje generatora će se pozvati datoteka `GLA.py`, a za izvođenje analizatora datoteka `LA.py`. Za rješenja u ostalim jezicima ulazna točka (main funkcija/metoda) može biti bilo gdje.

Detalji oko predaje, kao što su inačice alata koji će se koristiti za prevođenje i izvođenje rješenja, bit će objavljeni naknadno na FERWebu.

## 2.4 Savjeti za implementaciju

Savjeti za implementaciju navedeni u ovom poglavlju zamišljeni su kao pomoć u izradi generatora leksičkog analizatora. Kao takvi, nisu obvezujući i grupe po vlastitom nahođenju mogu dio ili sve savjete u ovom poglavlju zanemariti.

[4]Ova druga opcija ekvivalentna je serijalizaciji/deserijalizaciji što je isto dozvoljeno.

### 2.4.1 Generiranje koda leksičkog analizatora

Veliki dio izvornog koda generiranog leksičkog analizatora neovisan je od Ulazne Datoteke i treba biti napisan unaprijed. Preporuča se da se taj dio koda zasebno razvije i testira, na primjer koristeći izmišljenu tablicu konačnog automata. Na taj način, generator leksičkog analizatora može razvijeni kod iskopirati u datoteku s izvornim kodom leksičkog analizatora i dopuniti datoteku definicijom tablice konačnog automata i slično.

### 2.4.2 Priprema regularnih izraza za generiranje konačnog automata

Prije generiranja $\varepsilon$-NKA iz regularnog izraza, moguće je sve regularne definicije koji se pojavljuju u regularnom izrazu zamijeniti odgovarajućim regularnim izrazima. S obzirom na ograničenja regularnih definicija, svaka regularna definicija će prije njenog referenciranja biti definirana. Prema tome, zamjena regularnih definicija se tijekom rada generatora leksičkog analizatora može obaviti algoritmom prikazanim u pseudokodu u ispisu 2.13.

```
1 za svaku regularnu definiciju {regDef} u Ulaznoj Datoteci
2   neka je regEx regularni izraz koji opisuje definicjiu {regDef}
3   za svaku referencu {refRegDef} u regularnom izrazu regEx
4     zamijeni {refRegDef} u izrazu regEx s
5           "(" + regularniIzraz[{refRegDef}] + ")"
6   regularniIzraz[{regDef}] = regEx
```

Ispis 2.13: Pseudokod za pripremu regularnih izraza za generiranje konačnog automata.

Ključno je primijetiti da se koristeći ovaj algoritam svakoj regularnoj definiciji neposredno nakon što je pročitana iz Ulazne Datoteke pridruži "čisti" regularni izraz koji ne referencira niti jednu regularnu definiciju. Također, kako bi se očuvala očekivana prednost operatora, nužno je prije zamjene regularni izraz koji opisuje neku regularnu definiciju okružiti zagradama. Koristeći vrlo sličan algoritam i "čiste" regularne izraze za regularne definicije, moguće je pripremiti sve regularne izraze iz Ulazne Datoteke za generiranje konačnog automata, tj. zamijeniti sve reference na regularne definicije odgovarajućim regularnim izrazom.

Na primjer, neka su zadane sljedeće regularne definicije:
`{znamenka} 0|1|2|3|4|5|6|7|8|9`
`{hexZnamenka} {znamenka}|a|b|c|d|e|f|A|B|C|D|E|F`

Regularni izraz za regularnu definiciju `{znamenka}` ne sadrži reference na niti jednu regularnu definiciju. Regularni izraz za definiciju `{hexZnamenka}` sadrži referencu na definiciju `{znamenka}` tako da će primjenom opisanog algoritma regularni izraz za definiciju `{hexZnamenka}` biti preveden u `(0|1|2|3|4|5|6|7|8|9)|a|b|c|d|e|f|A|B|C|D|E|F`

Kada bi u dijelu pravila bilo definirano pravilo:
```
<S_nekoStanje>0x{hexZnamenka}{hexZnamenka}*
{
HEX_KONSTANTA
}
```

uvrštavanjem takvog regularnog izraza za definiciju `{hexZnamenka}` dobili bi jednostavan regularan izraz za prikazano pravilo (oba retka dio su istog izraza):

```
0x((0|1|2|3|4|5|6|7|8|9)|a|b|c|d|e|f|A|B|C|D|E|F)
((0|1|2|3|4|5|6|7|8|9)|a|b|c|d|e|f|A|B|C|D|E|F)*
```

Ovakav regularni izraz može se prevesti u $\varepsilon$-NKA koristeći ranije spomenuti algoritam opisan u udžbeniku "Uvod u teoriju računarstva" [2, poglavlje 2.2.2].

### 2.4.3 Izgradnja $\varepsilon$-NKA iz regularnih izraza za potrebe leksičkog analizatora

Ako se u obradi referenci na regularne definicije koristi algoritam koji je opisan u prethodnom poglavlju, za regularne izraze definicija nije potrebno graditi konačne automate jer se ti regularni izrazi na tekstualnoj razini umeću u regularne izraze pravila.

Kao što je navedeno u opisu pravila, svako pravilo bit će pridruženo točno jednom stanju leksičkog analizatora. Implementacija generatora mora osigurati da generirani leksički analizator koristi isključivo ona pravila koja su aktivna u trenutnom stanju leksičkog analizatora. To ograničenje moguće je ostvariti tako da se za svako stanje leksičkog analizatora generira zaseban $\varepsilon$-NKA te se ulazni znakovi predaju samo automatu koji je zadužen za obradu trenutno aktivnog stanja leksičkog analizatora.

Postupak izgradnje $\varepsilon$-NKA za potrebe leksičkog analizatora opisan je u udžbeniku [1, poglavlje 2.9.2]. Bitno je primijetiti da je prilikom dodavanja početnog stanja $p_0$ (vidi udžbenik) i spajanja svih automata $M_i$ u jedinstveni automat nužno osigurati da sva stanja u automatima $M_i$ imaju jedinstvene oznake. U suprotnom automat vrlo vjerojatno neće raditi ispravno! Kako bi se izbjegla potreba za preimenovanjem stanja, moguće je, umjesto spajanja automata $M_i$ u jedinstveni automat za neko stanje leksičkog analizatora, automate $M_i$ pohraniti odvojeno. Na taj način će svako pravilo pridruženo nekom stanju leksičkog analizatora imati svoj $\varepsilon$-NKA. Ova promjena utječe na sam algoritam rada leksičkog analizatora opisan u [1, poglavlje 2.9.7] jer će leksički analizator umjesto jednog koristiti nekoliko $\varepsilon$-NKA.

Izgradnja $\varepsilon$-NKA za regularni izraz svakog pravila načelno je opisana u udžbeniku predmeta "Uvod u teoriju računarstva". S obzirom na to da je postupak relativno složen, a nije središnja tema ove laboratorijske vježbe, u nastavku je dan pseudokod koji opisuje jedan način na koji se pretvorba može ostvariti.

Funkciji u pseudokodu predaje se regularni izraz i automat koji treba izgraditi. Vizualno, automat se gradi s lijeva na desno pa se imena varijabli naslanjaju na tu predodžbu. Funkcija kao rezultat vraća "lijevo" i "desno" stanje automata. Lijevo stanje za cijeli izraz odgovara početnom stanju automata, a desno stanje za cijeli izraz odgovara prihvatljivom stanju automata. Algoritam izgradnje $\varepsilon$-NKA iz regularnog izraza jamči da će automat imati točno jedno prihvatljivo stanje. Poziv algoritma prikazan je u ispisu 2.14.

```
1 ParStanja rezultat = pretvori(regularni_izraz, automat)
2 automat.pocetno_stanje = rezultat.lijevo_stanje
3 automat.prihvatljivo_stanje = rezultat.desno_stanje
```

Ispis 2.14: Poziv funkcije `pretvori`.

Stanja automata označavat će se cijelim brojevima od nula na više. Za dodavanje stanja automatu koristit će se funkcija `novo_stanje`. Funkcija `novo_stanje` vraća oznaku dodanog stanja. Pretpostavlja se da će početni broj stanja automata biti nula. Pseudokod funkcije `novo_stanje` prikazan je u ispisu 2.15.

```
1 int novo_stanje(automat)
2   automat.br_stanja = automat.br_stanja + 1
3   vrati automat.br_stanja - 1
```

Ispis 2.15: Pseudokod funkcije za dodavanje stanja automatu.

Prvi korak algoritma je potraga za operatorima izbora koji se nalaze izvan svih zagrada. Operator izbora ima najnižu pretpostavljenu prednost od svih operatora koje koristimo u regularnim izrazima pa je zbog toga nužno kao prvi korak niz podijeliti na podnizove između kojih se može birati. Na primjer, u regularnom izrazu
`(\)a|b)\|\(|x*|y*`
postoje dva operatora izbora izvan svih zagrada (neposredno ispred `x` i neposredno ispred `y`). Rezultat prvog koraka algoritma bila bi tri nova regularna izraza
`(\)a|b)\|\(`
`x*`
`y*`

Traženje operatora izbora izvan svih zagrada zahtijeva brojanje zagrada s lijeva na desno. Brojač zagrada kreće od `0`, povećava se za jedan za svaku otvorenu zagradu, a smanjuje za jedan za svaku zatvorenu zagradu. Ako algoritam naiđe na operator izbora kada je brojač zagrada jednak nuli, preostali dio izraza podijeli oko operatora izbora i nastavi dalje prema desno. Iz primjera je očito da pritom posebnu pažnju treba posvetiti zagradama i operatorima izbora koji su prefiksirani znakom `\` i nemaju svoje originalno značenje. Općenito, nužno je provjeriti nalazi li se ispred određenog operatora paran ili neparan broj znakova `\`. U algoritmu pretvorbe koristit će se pomoćna funkcija `je_operator` koja za dani regularni izraz i indeks operatora provjerava ima li operator svoje značenje ili je prefiksiran neparnim brojem znakova `\`. Pseudokod funkcije `je_operator` prikazan je u ispisu 2.16.

```
1 bool je_operator(izraz, i)
2   int br = 0
3   dok je i-1>=0 && izraz[i-1]=='\\' // jedan \, kao u C-u
4       br = br + 1
5       i = i - 1
6   kraj dok
7   vrati br%2 == 0
```

Ispis 2.16: Pseudokod funkcije `je_operator`.

Dio funkcije `pretvori` koji broji zagrade i dijeli izraz na podizraze odvojene operatorom izbora prikazan je u ispisu 2.17.

```
1 ParStanja pretvori(izraz, automat)
2   niz izbori
3   int br_zagrada = 0
```

```
za (i=0; i<duljina(izraz); i=i+1)
  ako je izraz[i]=='(' && je_operator(izraz, i)
    br_zagrada = br_zagrada + 1
  inace ako je izraz[i]==')' && je_operator(izraz, i)
    br_zagrada = br_zagrada - 1
  inace ako je br_zagrada==0 && izraz[i]=='|' && je_operator(izraz, i)
    grupiraj lijevi negrupirani dio niza znakova izraz u niz izbori
  kraj ako
kraj za
ako je pronadjen barem jedan operator izbora
  grupiraj preostali negrupirani dio niza znakova izraz u niz izbori
...
```

Ispis 2.17: Početak funkcije `pretvori`.

Nakon ovog odsječka, moguće je da je u nizu pronađen neki broj operatora izbora ili da se izraz na najvišoj razini (dakle, izvan svih zagrada) sastoji samo od podizraza povezanih nadovezivanjem. Nastavak algoritma koji započinje izgradnja automata prikazan je u ispisu 2.18.

```
...
int lijevo_stanje = novo_stanje(automat)
int desno_stanje = novo_stanje(automat)
ako je pronadjen barem jedan operator izbora
    za (i=0; i<br_elemenata(izbori); i=i+1)
      ParStanja privremeno = pretvori(izbori[i], automat)
      dodaj_epsilon_prijelaz(automat,
         lijevo_stanje,
         privremeno.lijevo_stanje)
      dodaj_epsilon_prijelaz(automat,
                         privremeno.desno_stanje,
                         desno_stanje)
    kraj za
inace
  ...
```

Ispis 2.18: Nastavak funkcije `pretvori`.

Kao što je prikazano odsječkom u ispisu 2.18, ako su u izrazu pronađeni podizrazi odvojeni operatorom izbora, svaki podizraz se rekurzivno obradi pozivom funkcije `pretvori` i lijeva i desna stanja dobivena od rekurzivnog poziva povežu se epsilon prijelazima s lijevim i desnim stanjem za cijeli izraz. Ovaj postupak je poopćenje točke *p4* u [2, poglavlje 2.2.2]. U ispisu 2.19 je opisana obrada drugog slučaja u kojem nisu pronađeni operatori izbora.

```
... // nastavlja se inace iz proslog odsjecka
bool prefiksirano = laz
int trenutno_stanje = lijevo_stanje
za (i=0; i<duljina(izraz); i=i+1)
```

```
5         ako je prefiksirano istina
6           *slucaj 1*
7         inace
8           *slucaj 2*
9         kraj ako
10      kraj za
11      dodaj_epsilon_prijelaz(automat, trenutno_stanje, desno_stanje)
```

Ispis 2.19: Obrada nadovezivanja u funkciji `pretvori`.

Varijabla `prefiksirano` služi za prepoznavanje je li trenutni znak u izrazu prefiksiran znakom \. Varijabla `trenutno_stanje` sadrži broj najdesnijeg stanja u postupku izgradnje automata. Kao što je prije spomenuto, postupak kreće s lijeva na desno. Na kraju obrade cijelog izraza (zadnja linija pseudokoda), najdesnije generirano stanje povezuje se epsilon prijelazom s desnim stanjem koje će se vratiti na kraju funkcije. Pri obradi svakog znaka izraza postoje dva slučaja, ovisno o tome je li znak prefiksiran znakom \ ili nije. Pseudokod za *slučaj 1* prikazan je u ispisu 2.20.

```
1  // slucaj 1
2  prefiksirano = laz
3  char prijelazni_znak
4  ako je izraz[i] == 't'
5    prijelazni_znak = '\t' // jedan znak, kao u C-u
6  inace ako je izraz[i] == 'n'
7    prijelazni_znak = '\n' // jedan znak, kao u C-u
8  inace ako je izraz[i] == '_'
9    prijelazni_znak = '␣' // obican razmak
10 inace
11   prijelazni_znak = izraz[i]
12 kraj ako
13
14 int sljedece_stanje = novo_stanje(automat)
15 dodaj_prijelaz(automat, trenutno_stanje, sljedece_stanje,
16     prijelazni_znak)
17 *provjeri ponavljanje*
18 trenutno_stanje = sljedece_stanje
```

Ispis 2.20: Slučaj 1 u funkciji `pretvori`.

Odsječak za slučaj 1 prvo postavlja varijablu `prefiksirano` u laž zato jer je prefiks iz prošlog znaka potrošen na znak koji se trenutno obrađuje. Ograničenja na Ulaznu Datoteku osiguravaju da će u slučaju 1 `izraz[i]` biti neki od specijalnih znakova opisanih u ovoj uputi u poglavlju 2.1.5. Posebnu pažnju treba posvetiti znakovima za tab, novi redak i prazninu. Prijelazni znak u automatu treba biti onaj znak kojeg će leksički analizator pročitati u ulaznom programu. Nakon što je određen prijelazni znak, automatu se dodaje novo stanje i odgovarajući prijelaz.

Provjera za operator ponavljanja odvojena je jer će se ponoviti nekoliko puta u cijelom pseudokodu. Ponavljanje se prevodi u automat slično kao u točki *p6* u [2, poglavlje 2.2.2]. Pseudokod je prikazan u ispisu 2.21.

```
// provjeri ponavljanje
ako je i+1<duljina(izraz) && izraz[i+1]=='*'
  dodaj_epsilon_prijelaz(automat, trenutno_stanje, sljedece_stanje)
  dodaj_epsilon_prijelaz(automat, sljedece_stanje, trenutno_stanje)
  i = i+1
kraj ako
```

Ispis 2.21: Obrada ponavljanja Kleeneovim operatorom.

Nakon dodavanja epsilon prijelaza koji obrađuju ponavljanje, znak ponavljanja se preskače pomicanjem indeksa za jedan unaprijed.

Obrada slučaja 2 u kojem znak nije prefiksiran znakom \ prikazana je u ispisu 2.22.

```
// slucaj 2
ako je izraz[i] == '\\' // jedan znak \, kao u C-u
  prefiksirano = istina
  nastavi za petlju // continue u C-u
kraj ako
ako je izraz[i] != '('
  *slucaj 2a*
inace
  *slucaj 2b*
kraj ako
```

Ispis 2.22: Slučaj 2 u funkciji `pretvori`.

U slučaju 2, prvo se provjerava je li trenutni znak baš znak \. Ako je, postavlja se varijabla prefiksirano i prelazi se na sljedeći znak. U suprotnom, provjerava se je li trenutni znak otvorena zagrada koja započinje neki podizraz. Time se slučaj dijeli na dva podslučaja. Ako trenutni znak nije otvorena zagrada onda se sigurno radi o nekom znaku kojeg treba nadovezati na dosad izgrađeni automat. Pseudokod za ovaj slučaj prikazan je u ispisu 2.23.

```
// slucaj 2a
int sljedece_stanje = novo_stanje(automat)
ako je izraz[i] == '$'
  dodaj_epsilon_prijelaz(automat, trenutno_stanje, sljedece_stanje)
inace
  dodaj_prijelaz(automat, trenutno_stanje, sljedece_stanje, izraz[i])
kraj ako
*provjeri ponavljanje*
trenutno_stanje = sljedece_stanje
```

Ispis 2.23: Slučaj 2a u funkciji `pretvori`.

Posebna pažnja pridaje se znaku $ koji označava prazan niz. Dodatno, nakon obrade znaka provjerava se postoji li nakon znaka operator ponavljanja.

Konačno, slučaj 2b pokriva mogućnost podizraza ograđenog zagradama. Pseudokod

za ovaj slučaj prikazan je u ispisu 2.24.

```
// slucaj 2b
int j = *pronadji odgovarajucu zatvorenu zagradu*
ParStanja privremeno = pretvori(izraz[i+1..j-1], automat)
dodaj_epsilon_prijelaz(automat, trenutno_stanje,
    privremeno.lijevo_stanje)
i = j
trenutno_stanje = privremeno.desno_stanje
ako je i+1<duljina(izraz) && izraz[i+1]=='*'
  dodaj_epsilon_prijelaz(automat, privremeno.lijevo_stanje, privremeno.
     desno_stanje)
  dodaj_epsilon_prijelaz(automat, privremeno.desno_stanje, privremeno.
     lijevo_stanje)
  i = i+1
kraj ako
```

Ispis 2.24: Slučaj 2b u funkciji `pretvori`.

Traženje odgovarajuće zatvorene zagrade može se obaviti brojeći zagrade na vrlo sličan način kao na početku funkcije pretvori pa ovdje neće biti opisano. Kada je pronađena zatvarajuća zagrada, izraz u zagradi rekurzivno se obradi i poveže s trenutno najdesnijim stanjem epsilon prijelazom. Konačno, provjeri se postoji li operator ponavljanja. Ostvarenje ponavljanja koristi različite varijable u odnosu na prethodna dva slučaja pa je ovdje posebno navedeno da se izbjegne zabuna.

Funkcija `pretvori` na kraju vraća vrijednosti varijabli `lijevo_stanje` i `desno_stanje` u strukturi `ParStanja`. Cijeli pseudokod u jednoj datoteci može se naći ovdje.

## 2.5 Leksička analiza podskupa jezika *C*

Koristeći generator leksičkog analizatora potrebno je generirati leksički analizator za zadani podskup jezika *C*. Ulazna Datoteka koja definira taj leksički analizator može se naći ovdje. Generirani leksički analizator će prepoznati ključne riječi jezika *C* navedene u tablici 2.1.

| | | |
|---|---|---|
| `break` | `else` | `return` |
| `char` | `for` | `void` |
| `const` | `if` | `while` |
| `continue` | `int` | |

Tablica 2.1: Podržane ključne riječi.

Podržani su cijeli brojevi u dekadskom, oktalnom i heksadekadskom zapisu. Znakovne konstante i nizovi znakova zapisuju se jednako kao u jeziku *C*. Dozvoljeni su komentari koji započinju znakovima `//` i traju do kraja retka i komentari koji su omeđeni nizovima `/*` i `*/`.

Generirani analizator će prepoznati specijalne znakove iz jezika *C* navedene u tablici 2.2.

```
[   ]   (   )   {   }   ;   ,   =
++  --  &   |   *   +   -   ~   !
/   %   <   >   <=  >=  ==  !=  ^   &&  ||
```

Tablica 2.2: Podržani specijalni znakovi.

Točno značenje pojedinog specijalnog znaka (npr. radi li se o binarnom ili unarnom minusu) odredit će se tijekom kasnijih faza rada jezičnog procesora. Primjer programa koji ispituje većinu leksičkih zahtjeva ovog podskupa jezika *C* može se naći ovdje. Leksički analizator bi trebao javiti tri greške u retku 27 i odbaciti sva tri apostrofa (jednog po jednog) u postupku oporavka od pogreške[5]. Očekivani izlaz za dani primjer prikazan je ovdje.

[5]Kao što je prije spomenuto, pripazite da leksički analizator ne ispisuje **ništa** na standardni izlaz osim traženog ispisa - greške ispisujte npr. na `stderr`.

# 3. Druga laboratorijska vježba

Tema druge laboratorijske vježbe je sintaksna analiza. Grupe izrađuju generator sintaksnog analizatora po uzoru na program Yacc, ali u bitno pojednostavljenom obliku. Generirani sintaksni analizator za parser će koristiti LR(1) kanonski parser opisan u udžbeniku od stranice 147. Iako se u praksi zbog velikog broja stanja LR(1) parsera često preferira korištenje LALR parsera, kanonski LR(1) parser odabran je s ciljem pojednostavljenja generatora kojeg treba izgraditi. U nastavku je opisano kako generator sintaksnog analizatora i generirani sintaksni analizator trebaju raditi i dani su neki savjeti za implementaciju. Ova uputa pretpostavlja da je čitatelj upoznat sa ispredavanim gradivom o sintaksnoj analizi.

*Napomena*: ova vježba potpuno je neovisna od prve vježbe i možete ju raditi i ako niste uspjeli završiti prvu vježbu.

## 3.1 Generator sintaksnog analizatora


Slika 3.1: Način rada generatora sintaksnog analizatora.

Način rada generatora sintaksnog analizatora analogan je načinu rada generatora leksičkog analizatora iz prve vježbe i prikazan je na slici 3.1. Generator sintaksnog analizatora kojeg je potrebno ostvariti treba prihvaćati opis procesa sintaksnog analizatora zadan tekstualnom datotekom (dalje Ulazna Datoteka). Rezultat izvođenja generatora treba biti sintaksni analizator, slično kao u prvoj vježbi. U praksi generatori sintaksnih analizatora kao izlaz daju izvorni kod sintaksnog analizatora, ali kao i u prvoj vježbi, dozvolit ćemo da generator analizatoru potrebne podatke predaje putem datoteke ili na neki sličan način. Ulazna Datoteka bit će zadana u sljedećem formatu:

%V *nezavršni znakovi gramatike*

%T *završni znakovi gramatike*

%Syn *sinkronizacijski završni znakovi*

*produkcije gramatike*

U nastavku je objašnjeno kako pojedini dio datoteke treba izgledati. Zbog jednostavnosti implementacije, generator sintaksnog analizatora može pretpostaviti da je Ulazna Datoteka ispravno zadana u skladu s pravilima opisanim u ovom dokumentu.

### 3.1.1 Nezavršni znakovi gramatike

Prvi redak Ulazne Datoteke započinje znakovima `%V` i točno jednim razmakom nakon čega slijede jedan ili više nezavršnih znakova gramatike odvojeni točno jednim razmakom. Svaki nezavršni znak gramatike pojavit će se u ovom retku točno jednom. Ime nezavršnog znaka sastoji se od jednog ili više malih i velikih slova engleske abecede i znakova _ (underscore) okruženih trokutastim zagradama. Trokutaste zagrade su sastavni dio nezavršnog znaka i služe za jednostavno razlikovanje nezavršnih i završnih znakova. Prvi navedeni nezavršni znak je **početni nezavršni znak gramatike**.

Na primjer, prvi redak Ulazne Datoteke može izgledati ovako:

```
%V <Program> <Naredba> <primarni_izraz>
```

U prikazanom primjeru, `<Program>` je početni nezavršni znak gramatike.

### 3.1.2 Završni znakovi gramatike

U drugom retku Ulazne Datoteke navedeni su završni znakovi gramatike. Redak počinje znakovima `%T` i točno jednim razmakom nakon čega slijede jedan ili više završnih znakova gramatike odvojeni točno jednim razmakom. Svaki završni znak gramatike bit će naveden točno jednom. Završni znakovi gramatike su uniformni znakovi koje generira leksički analizator te su istog oblika kao imena uniformnih znakova. Drugim riječima, imena završnih znakova će se sastojati od malih i velikih slova engleske abecede i znakova _ (underscore).

Na primjer, drugi redak Ulazne Datoteke može izgledati ovako:

```
%T IDENTIFIKATOR brojcanaKonstanta znakovnaKonstanta OP_PLUS
```

### 3.1.3 Sinkronizacijski završni znakovi

U trećem retku Ulazne Datoteke navedeni su sinkronizacijski završni znakovi. Generirani parser treba ove znakove koristiti u postupku oporavka o pogreške. Postupak oporavka od pogreške opisan je kasnije u poglavlju 3.2.4.

Redak započinje znakovima `%Syn` i točno jednim razmakom nakon čega slijede jedan ili više završnih znakova gramatike odvojeni točno jednim razmakom.

Na primjer, treći redak Ulazne Datoteke može izgledati ovako:

```
%Syn TOCKAZAREZ ZAREZ
```

### 3.1.4 Produkcije gramatike

U nastavku Ulazne Datoteke navedene su produkcije gramatike. Lijeve strane produkcije sastoje se od točno jednog nezavršnog znaka koji stoji sam u retku čiji početni znak je lijeva trokutasta zagrada nezavršnog znaka (vidi primjer). Nakon retka s lijevom stranom produkcije slijedi jedan ili više redaka koji započinju točno jednim znakom razmaka i predstavljaju desnu stranu produkcije. Desna strana svake produkcije sastoji se od jednog ili više završnih ili nezavršnih znakova odvojenih točno jednim razmakom. Desna strana epsilon produkcije prikazuje se retkom koji počinje znakom razmaka nakon kojeg slijedi znak `$` (dolar).

Na primjer, produkcije gramatike mogu izgledati ovako:

```
<A>
 <B> <C> c
<B>
 $
 b <C> <D> <E>
<A>
 e <D> <B>
<C>
 <D> a <B>
 c a
<D>
 $
 d <D>
<E>
 e <A> f
 c
```

Ispis 3.1: Primjer produkcija gramatike.

Na ovaj način prikazana je gramatika (opisana BNF notacijom):

```
<A> ::= <B> <C> c | e <D> <B>
<B> ::= "" | b <C> <D> <E>
<C> ::= <D> a <B> | c a
<D> ::= "" | d <D>
<E> ::= e <A> f | c
```

Ispis 3.2: Primjer produkcija gramatike u BNF notaciji.

Cjeloviti primjer Ulazne Datoteke koja opisuje gramatiku sa 100. stranice udžbenika nalazi se ovdje. Nad zadanom gramatikom je u udžbeniku prikazan način računanja ZAPOČINJE skupova. Sinkronizacijski znak je odabran nasumično kako bi se zadovoljila pravila za oblik Ulazne Datoteke.

### 3.1.5 Zadaci generatora sintaksnog analizatora

Generator sintaksnog analizatora treba pročitati Ulaznu Datoteku i izgraditi tablice Akcija i NovoStanje odgovarajućeg kanonskog LR(1) parsera. Generator treba izgrađene tablice ugraditi u zasebno razvijen sintaksni analizator. Tablice parsera mogu se ugraditi u sintaksni analizator na više načina isto kao što su se tablice konačnog automata ugrađivale u leksički analizator u prvoj laboratorijskoj vježbi.

Algoritam izgradnje tablica LR(1) parsera opisan je od stranice 148 udžbenika. Algoritam počinje dodavanjem novog početnog nezavršnog znaka i nove produkcije u zadanu gramatiku. Ovaj postupak opisan je u fusnoti na stranici 140 udžbenika. U teoretskom smislu, ako se originalni početni nezavršni znak gramatike ne pojavljuje s desne strane niti jedne produkcije, nije nužno dodavati novi početni nezavršni znak. Zbog uniformnosti, u ovoj laboratorijskoj vježbi generator LR(1) parsera **uvijek treba dodati novi početni nezavršni znak i odgovarajuću produkciju**.

Algoritam nastavlja izgradnjom nedeterminističkog konačnog automata s epsilon prijelazima. Stanja automata označena su LR(1) stavkama. Zbog uniformnosti oznaka stanja, nije potrebno u $\varepsilon$-NKA ugraditi stanje $q_0$, nego se za početno stanje može koristiti LR(1) stavka koja proizlazi iz dodane produkcije koja iz novog početnog nezavršnog znaka generira originalni početni nezavršni znak gramatike zadan u Ulaznoj Datoteci. U ovom slučaju, automat sa i bez stanja $q_0$ potpuno su ekvivalentni tj. prihvaćaju isti jezik.

Prilikom izgradnje automata, nužno je računati ZAPOČINJE skupove za nizove znakova. Način računanja ZAPOČINJE skupova opisan je na 102. stranici udžbenika. Za potrebe generiranja LR(1) parsera potrebno je proširiti domenu ZAPOČINJE skupova sa desnih strana produkcije na bilo koji sufiks desne strane produkcije (ili jednostavno na proizvoljne nizove završnih i nezavršnih znakova). Algoritam računanja ostaje potpuno isti. Dodatno, za generiranje LR(1) parsera **nije potrebno** određivati SLIJEDI skupove.

Dobiveni $\varepsilon$-NKA treba pretvoriti u istovjetni DKA. Postupak pretvorbe $\varepsilon$-NKA u DKA opisan je u udžbeniku "Uvod u teoriju računarstva". Stanja DKA označena su skupovima LR(1) stavki. Iz stanja i prijelaza DKA popunjavaju se tablice *Akcija* i *NovoStanje*.

### 3.1.6 Razrješavanje nejednoznačnosti

Iako je klasa LR(1) jezika široka i odgovara klasi determinističkih kontekstno neovisnih jezika (jezici koje može prepoznati deterministički potisni automat), pojedine konstrukte iz mnogih programskih jezika teško je opisati LR(1) gramatikom bez bitnog usložnjavanja gramatike ili je uopće nemoguće napisati odgovarajuću gramatiku. S obzirom da je složenost gramatike važna za semantičku analizu, pokazuje se da je dobro rješenje zadati gramatiku s pažljivo odabranim nejednoznačnostima koje se onda razriješe ili ručno ili koristeći jednostavna pravila koja su opisana u nastavku.

Nejednoznačnosti u gramatici u gradnji LR(1) parsera očituju se kroz *Pomakni/Reduciraj* i *Reduciraj/Reduciraj* proturječja. *Pomakni/Reduciraj* proturječje izgrađeni generator treba razriješiti u korist akcije Pomakni. *Reduciraj/Reduciraj* proturječje potrebno je razriješiti u korist one akcije koja reducira produkciju zadanu ranije u Ulaznoj Datoteci. Poželjno je da generator korisniku ispiše gdje je došlo do proturječja i kako je proturječje

razriješeno[1].

## 3.2 Sintaksni analizator

Uloga sintaksnog analizatora u ovoj laboratorijskoj vježbi bit će parsiranje niza uniformnih znakova. Drugim riječima, sintaksni analizator treba provjeravati zadovoljava li niz uniformnih znakova sintaksna pravila zadana gramatikom u Ulaznoj Datoteci i konstruirati generativno stablo za dani program. Sintaksni analizator treba generativno stablo ispisati na standardni izlaz prema uputi u poglavlju 3.2.3. U nastavku je opisan način rada sintaksnog analizatora kojeg je potrebno ostvariti za ovu laboratorijsku vježbu.

### 3.2.1 Ulaz u sintaksni analizator

Analizator čita niz uniformnih znakova sa standardnog ulaza. Niz uniformnih znakova bit će oblikovan identično kao izlaz iz prve laboratorijske vježbe. U ulazu će se pojavljivati isključivo oni uniformni znakovi koji su definirani kao završni znakovi gramatike u Ulaznoj Datotoeci koja je predana generatoru. Leksičke jedinke neće započinjati ili završavati znakom praznine ili tabom i neće se protezati kroz više redaka, ali mogu sadržavati praznine (vidi primjere).

Za sam postupak sintaksne analize važni su jedino uniformni znakovi, a broj retka i leksička jedinka služe za prijavljivanje grešaka i dodatno za ispis generativnog stabla kao što je opisano u nastavku.

### 3.2.2 Simulator LR parsera

Središnji algoritam generiranog sintaksnog analizatora je simulator LR parsera. Koristeći tablice *Akcija* i *NovoStanje*, simulator čita uniformne znakove iz niza uniformnih znakova i provjerava sintaksnu ispravnost ulaznog programa u jezični procesor. Dodatno, simulator gradi generativno stablo niza uniformnih znakova. Listovi generativnog stabla označeni su uniformnim znakovima ili epsilonom (znakom `$`) u slučaju da list predstavlja desnu stranu $\varepsilon$-produkcije. Kad bi se generativno stablo obišlo dubinskim pretraživanjem uz obradu djece s lijeva na desno i uz ispisivanje listova, opet bismo dobili (isti) niz uniformnih znakova (epsilon predstavlja prazan niz). Uz sam uniformni znak, list generativnog stabla treba biti označen i retkom u kojem se pripadna leksička jedinka nalazila u ulaznom programu i samom leksičkom jedinkom[2]. Unutrašnji čvorovi generativnog stabla su označeni nezavršnim znakovima zadane gramatike.

U ovoj laboratorijskoj vježbi, generativno stablo gradit će se eksplicitno. S tim ciljem, na stogu simulatora LR parsera trebaju se nalaziti čvorovi generativnog stabla (uz ostale podatke nužne za rad samog simulatora). Prilikom provođenja akcije *Pomakni*, simulator gradi novi list generativnog stabla i označava ga u skladu s ranijim opisom. Prilikom provođenja akcije *Reduciraj*, simulator gradi novi unutrašnji čvor stabla i čvorove sa stoga

[1] Ovo se očito *neće* provjeravati pri automatskoj evaluaciji rješenja na sustavu SPRUT.

[2] Alternativno, list može biti označen i nekim drugim podacima pomoću kojih je moguće rekonstruirati ova tri podatka.

koji odgovaraju uzorku za zamjenu povezuje kao djecu novog unutrašnjeg čvora. U skladu s pravilima rada simulatora LR parsera, uzorak za zamjenu uklanja se sa stoga i zamjenjuje se znakom lijeve strane produkcije uz pripadno stanje i čvor stabla. Slijedeći ovaj postupak, prilikom izvođenja akcije *Prihvati*, na stogu simulatora nalazit će se korjen generativnog stabla označen početnim nezavršnim znakom gramatike.

### 3.2.3 Ispis generativnog stabla

U ovoj laboratorijskoj vježbi, sintaksni analizator na standardni izlaz treba ispisati konstruirano generativno stablo. Ispis treba obaviti dubinskim obilaskom generativnog stabla, pri čemu se prvo ispisuje oznaka čvora roditelja, a nakon toga se ispisuju podstabla čiji korijeni su djeca čvora roditelja, s lijeva na desno. Na svakoj sljedećoj razini stabla, ispis treba prefiksirati jednim dodatnim razmakom.

Pri ispisu oznaka unutrašnjih čvorova stabla treba ispisivati nezavršni znak gramatike kojim je čvor označen. Pri ispisu oznaka listova treba ispisati ili znak `$` ako se radi o čvoru označenom epsilonom ili uniformni znak, broj retka i leksičku jedinku identično kao što je ista leksička jedinka opisana u ulazu analizatoru. Drugim riječima, nakon odgovarajućeg broja praznina (ovisno o dubini u generativnom stablu), treba ispisati uniformni znak, jedan znak razmaka, broj retka, jedan znak razmaka i leksičku jedinku.

### 3.2.4 Oporavak od pogreške

Za potrebe ove laboratorijske vježbe, u simulator LR parsera treba ugraditi jednostavan postupak oporavka od pogreške traženjem sinkronizacijskog znaka. Kada naiđe na pogrešku u nizu uniformnih znakova, simulator treba ispisati poruku o pogrešci u kojoj su navedeni[3]:

1. broj retka u kojem se pogreška dogodila
2. očekivani uniformni znakovi (oni znakovi koji ne bi izazvali pogrešku)
3. pročitani uniformni znak iz niza uniformnih znakova i odgovarajući znakovni prikaz iz izvornog koda programa (dohvaćen iz tablice znakova).

Nakon ispisa pogreške, simulator treba preskočiti sve znakove u nizu uniformnih znakova do prvog sljedećeg sinkronizacijskog znaka. Sinkronizacijski znakovi za gramatiku definirani su u Ulaznoj Datoteci. Kada je pronašao prvi sljedeći sinkronizacijski znak, simulator sa stoga odbacuje stanja (i pripadne podatke i čvorove stabla) dok ne dođe do nekog stanja s u kojem je definirana $Akcija[s, sinkronizacijski\_znak]$. Nakon toga simulator nastavlja s normalnim radom[4].

[3] Ovo se neće provjeravati u automatskoj evaluaciji putem sustava SPRUT, ali se može provjeravati na usmenom odgovaranju vježbe.

[4] Ovaj postupak oporavka od pogreške vrlo je jednostavan, ali očito daje i vrlo slabe rezultate, tj. za veliku većinu sintaksnih pogrešaka oporavak neće biti osobito uspješan.

## 3.3 Primjeri

U ispisu 3.3 je prikazana Ulazna Datoteka za gramatiku sa 148. stranice udžbenika, pri čemu je izbačen nezavrnšni znak `S` i pripadna prva produkcija, a nezavršni znak `A` je postavljen za početni nezavršni znak. Znak `S` je u toj gramatici služio kao "novi početni nezavršni znak", a njega sintaksni analizator iz ove vježbe samostalno dodaje.

```
%V <A> <B>
%T a b
%Syn b
<A>
 <B> <A>
<B>
 a <B>
 b
<A>
 $

```

Ispis 3.3: Ulazna Datoteka za gramatiku sa 148. stranice udžbenika.

Sinkronizacijski znak je ovdje arbitrarno postavljen na `b`, ali na ovom primjeru nećemo promatrati oporavak od pogreške. Završni znakovi gramatike `a` i `b` predstavljaju uniformne znakove leksičkih jedinki.

Primjer ulaza u sintaksni analizator generiran za prethodnu Ulaznu Datoteku prikazan je u ispisu 3.4. U primjeru su nizu uniformnih znakova arbitrarno pridruženi brojevi redaka i leksičke jedine - recimo da uniformni znak `a` predstavlja leksičke jedinke koje se sastoje od uzastopnih znakova `x` (opcionalno odvojenih prazninama), a uniformni znak `b` predstavlja leksičke jedinke koje se sastoje od uzastopnih znakova `y` (opcionalno odvojenih prazninama).

```
a 1 x x x
b 2 y y
a 3 xx xx
a 4 xx xx xx
b 4 y

```

Ispis 3.4: Ulaz u analizator za gramatiku sa 148. stranice udžbenika.

Ispis sintaksnog analizatora za dani primjer prikazan je u ispisu 3.5. Ispis prikazuje stvoreno generativno stablo u skladu s opisom u poglavlju 3.2.3.

```
<A>
 <B>
  a 1 x x x
  <B>
   b 2 y y
```

```
<A>
 <B>
  a 3 xx xx
  <B>
   a 4 xx xx xx
   <B>
    b 4 y
 <A>
  $

```

Ispis 3.5: Ispis analizatora za gramatiku sa 148. stranice udžbenika i ulaz is ispisa 3.4.

Primjer Ulazne Datoteke generatora za podskup jezika C nalazi se ovdje[5]. U ispisu 3.6 prikazan je "C program" sa sintaksnom pogreškom.

```
int x = 3 &;
int y = 2;

```

Ispis 3.6: Jednostavan C program sa sintaksnom greškom.

Za prikazani program, sintaksni analizator bi na standardni ulaz dobio ulaz prikazan u nastavku.

```
KR_INT 1 int
IDN 1 x
OP_PRIDRUZI 1 =
BROJ 1 3
OP_BIN_I 1 &
TOCKAZAREZ 1 ;
KR_INT 2 int
IDN 2 y
OP_PRIDRUZI 2 =
BROJ 2 2
TOCKAZAREZ 2 ;

```

Ispis 3.7: Ulaz za sintaksni analizator za program 3.6.

Do sintaksne greške ovdje dolazi kada analizator čita uniformni znak `TOCKAZAREZ` koji odgovara znaku `;` iz prvog retka programa[6].

Kako je `TOCKAZAREZ` jedan od sinkronizacijskih znakova, u postupku oporavka od pogreske se ne odbacuje niti jedan uniformni znak iz ulaza. U drugom koraku postupka

[5]Za ovaj primjer generator generira $\varepsilon$-NKA s 3115 stanja i 6343 prijelaza iz čega nastaje DKA sa 691 stanja i 5404 prijelaza uz jedno *Pomakni/Reduciraj* proturječje vezano uz `else` dio naredbe grananja.

[6]Ponovno, izvorni kod je ovdje prikazan samo kako bi bilo lakse pratiti primjer — niti jedan program u ovoj vježbi neće dobiti taj izvorni kod, već će sintaksni analizator izravno dobiti niz uniformnih znakova iz ispisa 3.7

oporavka od pogreške sa stoga se izbacuje list stabla koji predstavlja znak `OP_BIN_I` tj. znak `&` iz ulaznog programa. Nakon izbacivanja tog znaka, parser može prihvatiti znak `TOCKAZAREZ` pa ga i prihvaća i nastavlja s radom, tj. postupak oporavka od pogreške je bio uspješan.

Očekivani ispis analizatora može se vidjeti ovdje.

Svi ovi kao i nekoliko složenijih primjera nalaze se u ovoj arhivi. Za jednostavnije primjere je dana i datoteka imena `*_gen.txt` koja sadrži dio kontrolnog ispisa generatora što možda može biti korisno kod uklanjanja pogrešaka pri razvoju generatora. U tim datotekama su navedene izračunate relacije i generirani konačni automati, a za najjednostavniji primjer i tablice LR-parsera.

Ulazne Datoteke za generator u arhivi imaju nastavak `.san`, a ulazi i očekivani ispisi iz analizatora imaju nastavke `.in` i `.out`. Dodatno, uz primjere je priložen i `.gif` generativnog stabla kako biste mogli dobiti osjećaj za to kako stablo izgleda, pogotovo za primjer podskupa jezika C što će biti važno za treću laboratorijsku vježbu.

## 3.4 Savjeti za implementaciju

U ovom poglavlju navedene su neke napomene koje mogu pomoći u ostvarenju ove laboratorijske vježbe. Poglavlje će biti prošireno ako se za tim pokaže potreba u skladu sa čestim pitanjima.

### 3.4.1 Računanje refleksivnog tranzitivnog okruženja relacije

Prilikom računanja skupova ZAPOČINJE, potrebno je izračunati relaciju *ZapočinjeZnakom* koja je refleksivno tranzitivno okruženje relacije *ZapočinjeIzravnoZnakom*. Na stranici 102 udžbenika na visokoj razini je opisano kako se spomenuto okruženje može izračunati. Korisno je relaciju *ZapočinjeZnakom* zamisliti kao usmjereni graf u kojem su znakovi čvorovi, a bridovi su članovi relacije *ZapočinjeZnakom*. Drugim riječima, ako i samo ako je u tablici relacije *ZapočinjeZnakom* za neki par znakova navedena jedinica, ti znakovi su u grafu spojeni usmjerenim bridom. Neki par znakova (X, Y) će biti u tranzitivnom okruženju dane relacije ako i samo ako u usmjerenom grafu postoji put od znaka X do znaka Y. Dodatno, s obzirom da tražimo *refleksivno* tranzitivno okruženje, u relaciji će biti i svi parovi (X, X).

U skladu s ovom interpretacijom, postupak računanja opisan u knjizi trebalo bi ponavljati iterativno dok god se tablica nove relacije ne prestane mijenjati ili jednostavno ponoviti ga $n$ puta gdje je $n$ dimenzija tablice. Također, bilo koji drugi postupak za traženje puteva u grafu može se iskoristiti za računanje nove relacije.

## 3.5 Predaja druge laboratorijske vježbe

Predaja druge laboratorijske vježbe na sustav SPRUT odvijat će se isto kao predaja prve laboratorijske vježbe.

Rješenje treba biti zapakirano u `zip` arhivu pri čemu se programski kod generatora nalazi u korijenskom direktoriju arhive. Nakon prevođenja i izvođenja generatora, u poddirektoriju `analizator` treba se nalaziti sve potrebno za prevođenje i izvođenje sintaksnog analizatora.

Ulazna točka u generator za Javu treba biti u razredu `GSA`, a ulazna točka u analizator u razredu `SA`. Za Python, ulazna točka u generator će biti datoteka `GSA.py`, a ulazna točka u analizator datoteka `SA.py`. Za ostale jezike (uključujući C#), imena datoteka su proizvoljna.

*Napomena:* predano rješenje za drugu vježbu ni na koji način ne ovisi o prvoj vježbi. Ulaz u sintaksni analizator u ovoj vježbi je niz uniformnih znakova u skladu s prethodnim opisom i nigdje u ovoj vježbi ne baratamo s izvornim kodom programa. Zato nemojte spajati leksički analizator s rješenjem ove vježbe i očekivati izvorni kod na ulazu.

# 4. Treća laboratorijska vježba

Tema treće laboratorijske vježbe je semantička analiza. Za razliku od prve dvije vježbe gdje je cilj bio ostvariti generator leksičkog odnosno sintaksnog analizatora, u ovoj vježbi je cilj ostvariti semantički analizator za jedan zadani programski jezik. Važno je uočiti da je ova vježba potpuno neovisna o rješenjima prve dvije vježbe.

Kako je semantička analiza tipično složenija od leksičke i sintaksne analize, među ostalim i zbog teže primjenjive formalne podloge, programski jezik koji će se koristiti u ovoj vježbi je semantički relativno jednostavan. Dodatno, neka ograničenja koja bi tipično bila u domeni semantičke analize prebačena su u sintaksnu analizu, opet s ciljem pojednostavljivanja semantičke analize. Nadalje, iako se semantička analiza tipično provodi nad sintaksnim stablom[1], za potrebe laboratorijskih vježbi ćemo ovaj korak preskočiti, tj. semantička analiza će se provoditi nad generativnim stablom. U ovoj uputi navode se samo semantička pravila koja su važna za semantičku analizu, a pravila će biti dodatno proširena u uputi za sljedeću laboratorijsku vježbu za potrebe generiranja koda.

Uz sva pojednostavljenja, opis semantike svakog programskog jezika (pa tako i ovog) je inherentno relativno složen. Na primjer, u *C* standardu opis jezika (bez standardnih biblioteka) ima nešto manje od 200 stranica, a od toga se većina bavi semantikom. Naravno, *C* je daleko složeniji jezik od jezika kojim se bave ove laboratorijske vježbe i opis ovog jezika nije ni približno toliko potpun, ali je istovremeno znatno jednostavniji za potrebe implementacije semantičkog analizatora. Cilj upute je da većina pravila bude očita nakon prvog čitanja, a da istovremeno što veći dio jezika bude pokriven, tj. da ne bude nužno postavljati mnogo pitanja tipa "A što ako...?".

## 4.1 Ulaz i ispis semantičkog analizatora

Semantički analizator će na stadardni ulaz dobiti generativno stablo u istom obliku kao ispis sintaksnog analizatora iz druge laboratorijske vježbe. Usprkos tome, u većini primjera kojima se u uputi ilustrira neko semantičko pravilo prikazuje se program a ne generativno stablo. Program **nije** ulaz u rješenje ove laboratorijske vježbe, ali je semantička pravila lakše objasniti nad programom nego nad relativno nepreglednim generativnim stablom. Iz tog razloga se za testiranje rješenja tijekom razvoja preporuča korištenje leksičkog i sintaksnog analizatora kako biste mogli rad semantičkog analizatora provjeravati nad programima i kako ne biste morali ručno mijenjati generativno stablo.

Za ovaj način rada ne morate mijenjati rješenja prošlih laboratorijskih vježbi ni na

[1] Tipično sintaksni analizator izravno generira sintaksno stablo ili se sintaksno stablo izgradi naknadnim obilaskom generativnog stabla.

koji način. Ulazna datoteka koja opisuje leksički analizator jezika *ppjC* nalazi se ovdje[2], a ulazna datoteka koja opisuje sintaksni analizator jezika *ppjC* nalazi se ovdje[3]. Koristeći ove dvije datoteke i rješenja prethodnih laboratorijskih vježbi možete generirati leksički i sintaksni analizator za jezik *ppjC*. Nadalje, kako analizatori čitaju ulaz sa stadnardnog ulaza i ispisuju rezultat izvođenja na standardni izlaz, a ulazi i izlazi su im kompatibilni, analizatore možete povezati u "lanac" u naredbenom retku.

Ako pretpostavimo da je *LA* ime izvodivog leksičkog analizatora, *SA* ime izvodivog sintaksnog analizatora i *SemantickiAnalizator* ime izvodivog semantičkog analizatora (na kojem radite u ovoj vježbi), onda bi naredba kojom možete testirati rad semantičkog analizatora nad programom izgledala na primjer ovako:
*LA* `<program.c | SA | SemantickiAnalizator`

Semantički analizator treba obići generativno stablo i na standardni izlaz "ispisati produkciju" u kojoj je otkrivena prva semantička pogreška, kao što je precizno definirano u poglavlju 4.4.2. Semantički analizator ne treba provoditi nikakav postupak oporavka od pogreške. Drugim riječima, čim otkrije prvu semantičku pogrešku i obavi odgovarajući ispis, semantički analizator prestaje s radom.

Ako je program prikazan generativnim stablom na ulazu semantički ispravan, semantički analizator ne smije ispisati ništa na standardni izlaz[4].

## 4.2 Predaja treće laboratorijske vježbe

Za predaju treće laboratorijske vježbe, sve datoteke s programskim kodom treba zapakirati u `zip`-arhivu. Ostala pravila jednaka su kao za prethodne vježbe, osim što više nije potreban direktorij `analizator`.

Ulazna točka za Javu treba biti u razredu `SemantickiAnalizator`, a ulazna točka za Python treba biti u datoteci `SemantickiAnalizator.py`. Za ostale jezike imena datoteka i razreda su proizvoljna.

## 4.3 Opis jezika *ppjC*

U ovom poglavlju dan je pregled jezika kako bi se olakšalo praćenje semantičkih pravila u poglavlju 4.4. Neki dijelovi ovog opisa možda neće biti odmah jasni, ali bit će dodatno razjašnjeni kroz opis semantičkih pravila.

### 4.3.1 Tipovi podataka

Jezik *ppjC* ima vrlo jednostavan sustav tipova što značajno olakšava semantičku analizu. U nastavku poglavlja navedeni su tipovi podataka u jeziku i objašnjeno je na koje načine

[2]Ovo je ista datoteka kao `simplePpjLang.lan`

[3]Ovo je malo pojednostavljena inačica datoteke `simplePpjLang.san`. Generator generira $\varepsilon$-NKA s 2936 stanja i DKA s 543 stanja.

[4]Isto kao u prethodnim vježbama, sav ostali ispis, na primjer smislenije poruke o pogrešci namijenjene korisnicima cjelovitog kompilatora, mora ići na standardni izlaz za greške, tj. `stderr`.

se vrijednosti jednog tipa mogu implicitno ili eksplicitno (*cast* operatorom) pretvoriti u vrijednosti drugog tipa.

### Brojevni tipovi

Jezik *ppjC* ima dva brojevna tipa: `char` i `int`. Raspon vrijednosti tipa `char` je 0–255 (8b NBC). Tip `int` koristi 32 bita i predstavlja i nenegativne cijele brojeve (pozitivne cijele brojeve i nulu) i negativne cijele brojeve u dvojnom komplementu. Drugim riječima, raspon vrijednosti tipa `int` je $-2^{31} = -2147483648 \leq v \leq 2147483647 = 2^{31} - 1$.

U ostatku upute će se brojevni tipovi označavati sa $T$, tj. $T$ može biti `char` ili `int`.

### const-kvalifikator

Brojevni tip prefiksiran ključnom riječi `const` (uniformni znak `KR_CONST`) naziva se `const`-kvalificiran tip i u ostatku upute će biti označavan sa *const*($T$) gdje je $T$ ili `char` ili `int`. Varijablu `const`-kvalificiranog tipa obavezno se mora inicijalizirati pri definiciji i tada pridružena vrijednost ne može se promijeniti tijekom izvođenja programa.

U ostatku upute će se brojevni tip s opcionalnim `const`-kvalifikatorom označavati znakom $X$, tj. $X$ može biti `char`, `int`, *const*(`char`) ili *const*(`int`). Koristeći oznaku $T$, $X$ može biti $T$ ili *const*($T$).

### Nizovi (engl. *array*)

Jezik podržava nizove isključivo brojevnih tipova (`const`-kvalificiranih ili ne). Drugim riječima, ne postoje nizovi nizova (tj. višedimenzionalni nizovi), a ovo ograničenje je osigurano u gramatici, tj. sintaksnoj analizi. Tip *niz* se u uputi označava sa *niz*($X$) gdje je $X$ neki (možda `const`-kvalificiran) brojevni tip.

### void

Ključna riječ `void` ima vrlo sličnu ulogu kao u jeziku *C*, a kao tip može biti isključivo povratni tip funkcije.

### Tip funkcija

Tip funkcije određen je tipom povratne vrijednosti i svih formalnih parametara. Na primjer, funkcije u ispisu 4.1 redom imaju tipove *funkcija*(`[char, char]` → `int`), *funkcija*(`[int]` → `void`) i *funkcija*(`void` → `void`). Tipovi parametara organizirani su u listu, a ako funkcija nema parametara, to je označeno sa `void` (ali ne sa `[void]`, što ne bi imalo smisla).

```
1 int f1(char a, char b) {
2     return a == b;
3 }
```

```
4
5  void f2(int x) {
6      return;
7  }
8
9  void f3(void) {
10     return;
11 }
```

Ispis 4.1: Primjer jednostavne funkcije.

### Implicitne promjene tipa

Sve vrijednosti tipa $const(T)$ mogu se implicitno pretvoriti u vrijednost tipa $T$[5]. Vrijedi i obrat, tj. sve vrijednosti tipa `int` ili `char` mogu se implicitno pretvoriti u vrijednosti odgovarajućeg `const`-kvalificiranog tipa.

Nadalje, sve vrijednosti tipa `char` mogu se implicitno pretvoriti u vrijednost tipa `int`. Kako `int` sadrži raspon tipa `char` kao podskup svog raspona, pri ovoj implicitnoj promjeni sama vrijednost se ne mijenja.

Konačno, vrijednost tipa $niz(T)$ gdje $T$ **nije** `const`-kvalificiran tip može se pretvoriti u vrijednost tipa $niz(const(T))$.

U semantičkim pravilima u nastavku upute, izraz $U \sim V$ je zadovoljen ako se vrijednost tipa $U$ može implicitno pretvoriti u vrijednost tipa $V$, a inače se radi o semantičkoj pogrešci. Relacija $\sim$ je refleksivna i tranzitivna.

### Logički podtip tipa `int`

Logički tip u jeziku *ppjC* nije poseban tip nego je podtip tipa `int` s vrijednostima `0` i `1`, pri čemu `0` predstavlja logičku neistinu, a `1` logičku istinu. Sve vrijednosti različite od nule pretvaraju se u logičku vrijednost `1`, a nula se "pretvara" u logičku vrijednost `0`[6].

### Eksplicitne promjene tipa

Eksplicitne promjene tipa dozvoljene su samo nad vrijednostima brojevnih tipova, a zadaju se *cast* operatorom. Drugim riječima, jedina promjena tipa koju je moguće ostvariti samo eksplicitno je promjena iz vrijednosti tipa `int` u vrijednost tipa `char`. Ako iznos `int` vrijednosti nije u opsegu tipa `char`, **rezultat operacije nije definiran**[7], a u suprotnom se iznos ne mijenja.

[5] Važno je uočiti da se promjene tipa (i implicitne i eksplicitne) odvijaju nad **vrijednostima** a ne nad **varijablama**. Tip varijable koji je određen njenom definicijom nije moguće nikako promijeniti.

[6] Ova rečenica ni na koji način ne utječe na semantičku analizu u ovoj laboratorijskoj vježbi, ali će biti važna za generiranje koda u sljedećoj vježbi.

[7] Vidi 4.4.1.

### 4.3.2 Konstante

Brojevne konstante (uniformni znak `BROJ`) su tipa `int` i moraju imati vrijednost u dozvoljenom rasponu za tip `int`.

Znakovne konstante (uniformni znak `ZNAK`) su tipa `char` i imaju vrijednost jednaku ASCII kodu znaka. Kroz leksičku analizu proći će i neke znakovne konstante koje nisu ispravne pa takve pogreške treba otkriti u semantičkoj analizi. Konkretno, od znakovnih konstanti koje između jednostrukih navodnika imaju više od jednog znaka (tj. imaju znak prefiksiran znakom \), *ppjC* dozvoljava isključivo znakove `'\t'`, `'\n'`, `'\0'`, `'\''` (jednostruki navodnik), `'\"'` (dvostruki navodnik) i `'\\'`. Sve ostale dvoznačne konstante predstavljaju semantičku grešku[8]. Nadalje, znak dvostrukog navodnika se može pojaviti neprefiksiran (dakle `'"'`) sa istim značenjem.

Konstantni znakovni nizovi (uniformni znak `NIZ_ZNAKOVA`) su tipa *niz*(*const*(`char`)) i implicitno završavaju znakom `'\0'` (kao u *C*-u). Ispravni konstantni znakovini nizovi počinju i završavaju dvostrukim navodnikom (ovo je osigurano leksičkom analizom), i mogu sadržavati sve ispisive ASCII znakove, a znakom \ mogu biti prefiksirani samo znakovi koji su navedeni u prethodnom odlomku. Slično kao za znakovne konstante, leksička analiza će propustiti neke konstantne znakovne nizove koji nisu ispravni, npr. nizove `"\"` i `"\x"`.

### 4.3.3 Djelokrug deklaracija i životni vijek varijabli

Jezik *ppjC* po klasifikaciji iz udžbenika koristi *statičko pravilo djelokruga bez ugniježđenih procedura* [1, str. 223–224], ali podržava ugniježđene blokove, isto kao *C*. Kako je sintaksa procedura u udžbeniku bliska *Pascalu*, a taj stil danas više nije previše zastupljen i mnogi od vas se možda s njim nisu prije susreli, za potrebe laboratorijskih vježbi se oslonite više na djelokrug deklaracija *C*-a i na primjere u nastavku.

Životni vijek globalnih varijabli počinje u trenutku njihove definicije, a završava na kraju izvođenja programa. Za varijable lokalne nekom bloku (što uključuje i parametre funkcija), životni vijek počinje njihovom definicijom i završava na kraju tog bloka. Varijable koje nisu eksplicitno inicijalizirane (prilikom definicije ili kasnije) imaju *neodređenu vrijednost* (drugim riječima, mogu imati bilo koju vrijednost) i rezultat korištenja takve vrijednosti u bilo kakvom izrazu nije definiran[9].

```
1 int x = 3;
2 int main(void) {
3     int y = x + 1; // 4
4     return 0;
5 }
```

[8] Ovo pravilo se dakako moglo osigurati i u leksičkoj analizi, ali je namjerno ostavljeno za semantičku analizu kako bi se ilustriralo da leksička, sintaksna i semantička pravila nisu zadana nekim univerzalnim kanonom, nego autor jezičnog procesora mora odlučiti koje značajke će provjeravati na koji način. Ovo svakako nije previše uobičajen primjer jer je vrlo lako uključiti ovo pravilo u regularne izraze, ali na primjer provjera raspona brojevnih konstanti je značajno jednostavnija u domeni semantičke analize nego u ovakvom modelu leksičke analize.

[9] Za razliku od jezika *C*, ovo se odnosi i na varijable u globalnom djelokrugu.

Ispis 4.2: Jednostavan primjer djelokruga deklaracija.

Jednostavan primjer prikazan je u ispisu 4.2. Varijabla `x` deklarirana je u globalnom djelokrugu u kojem je deklarirana i funkcija `main`. Tijelo funkcije `main` je blok, tj. složena naredba, te kao takvo ima vlastiti djelokrug deklaracija u kojem je deklarirana varijabla `y`. U tijelu funkcije `main`, prethodno deklarirana imena iz globalnog djelokruga (u ovom slučaju `x`) također su dostupna.

```
1 int main(void) {
2     int y = x + 1; // greska
3     return 0;
4 }
5 int x = 3;
```

Ispis 4.3: Primjer pogreške u djelokrugu—varijabla `x` nije deklarirana prije korištenja.

Primjer u ispisu 4.3 sadrži semantičku pogrešku u retku `2` jer varijabla `x` nije prethodno deklarirana.

```
1 int x = 3;
2 int main(void) {
3     int x = 5;
4     int y = x + 1; // 6
5     return 0;
6 }
```

Ispis 4.4: Sakrivanje globalne deklaracije.

Lokalno ime može sakriti ime iz ugniježđujućeg djelokruga. Primjer tog slučaja prikazan je u ispisu 4.4. Deklaracija varijable `x` u retku `3` skriva globalnu varijablu `x`.

```
1  int main(void) {
2      int x = 3;
3      int z;
4      {
5        int x = 5;
6        int y = x + 1; // 6
7        z = y + 1; // 7
8      }
9      z = x + 1; // 4
10     return 0;
11 }
```

Ispis 4.5: Sakrivanje deklaracije u bloku.

Isto pravilo zajedno s pravilom o životnom vijeku varijabli određuje značenje programa u ispisu 4.5. Važno je uočiti da je moguće sakriti samo deklaraciju iz ugniježđujućeg djelokruga, ali ne i iz istog djelokruga (u tom slučaju radi se o pokušaju redeklaracije što za varijable nije dozvoljeno, a pravila za funkcije su objašnjena u poglavlju 4.3.4).

Primjer ove greške prikazan je u ispisu 4.6.

```
1 int main(void) {
2     int x = 5;
3     int x = 6; // greska
4     return 0;
5 }
```

Ispis 4.6: Nedozvoljena redeklaracija varijable.

### 4.3.4 Funkcije

Svaki program u jeziku *ppjC* mora imati funkciju s prototipom `int main(void)`. Ova funkcija je ulazna točka za izvođenje programa.

Sve funkcije koje ne primaju argumente, u deklaraciji između oblih zagrada moraju imati navedenu ključnu riječ `void`. Drugim riječima, deklaracija funkcije oblika `int f();` je neispravna, a to je osigurano u gramatici tj. u sintaksnoj analizi.

Svaka funkcija prije korištenja mora biti **deklarirana**, pri čemu je **definicija** funkcije ujedno i njena deklaracija. Deklaracija je u tom slučaju u potpunosti završena prije znaka lijeve vitičaste zagrade koji započinje tijelo funkcije. Funkcija može biti deklarirana proizvoljan broj puta, a mora biti definirana točno jednom. Pritom, sve deklaracije (uključujući i definiciju) moraju imati identične povratne tipove i tipove formalnih parametara.

U deklaracijama je (za razliku od *C*-a) obvezno navesti imena formalnih parametara, ali ta imena nisu važna tj. mogu se razlikovati među različitim deklaracijama iste funkcije (drugim riječima, treba ih ignorirati).

```
1 int main(void) {
2   return f(); // greska
3 }
4 int f(void) {
5   return 0;
6 }
```

Ispis 4.7: Funkcija `f` nije deklarirana prije korištenja.

U ispisu 4.7, semantička pogreška pojavljuje se pri pozivu funkcije `f` u retku `2`, jer funkcija `f` nije prethodno deklarirana. Grešku je moguće ispraviti na više načina, a tri načina prikazana su u nastavku.

```
1 int f(void);
2 int main(void) {
3   return f();
4 }
5 int f(void) {
6   return 0;
```

```
}
```

Ispis 4.8: Dodatak deklaracije funkcije `f` prije definicije funkcije `main`.

Prvo, kao što je prikazano u ispisu 4.8, moguće je prije definicije funkcije `main` (koja poziva funkciju `f`) dodati deklaraciju funkcije `f`.

Drugo, za razliku od definicija funkcija koje su dozvoljene isključivo u globalnom djelokrugu, funkcija se može deklarirati i unutar (bilo kojeg) bloka. U ispisu 4.9 deklaracija funkcije `f` prebačena je iz globalnog djelokruga u tijelo funkcije `main`.

```
int main(void) {
  int f(void);
  return f();
}
int f(void) {
  return 0;
}
```

Ispis 4.9: Deklaracija funkcije `f` neposredno prije poziva u tijelu funkcije `main`.

Kako je definicija funkcije ujedno i deklaracija, u ovom slučaju je moguće definiciju funkcije `f` prebaciti prije definicije funkcije `main`, kao što je prikazano u ispisu 4.10.

```
int f(void) {
  return 0;
}
int main(void) {
  return f();
}
```

Ispis 4.10: Definicija funkcije `f` je ujedno i njena deklaracija.

Nadalje, s obzirom na to da deklaracija funkcije završava prije lijeve vitičaste zagrade koja započinje tijelo funkcije, za ostvarenje rekurzije nije potrebno koristiti eksplicitnu deklaraciju funkcije. Primjer semantički ispravne rekurzivne funkcije prikazan je u ispisu 4.11.

```
int fact(int n) {
  if (n > 0) {
    return n * fact(n-1);
  } else {
    return 1;
  }
}
```

Ispis 4.11: Rekurzivne funkcije ne treba eksplicitno deklarirati prije definicije.

Ipak, u slučaju da se dvije funkcije međusobno pozivaju, nije moguće izbjeći eksplicitnu deklaraciju jedne od njih. Primjer takvog slučaja prikazan je u ispisu 4.12.

```
1  int bar(int x);
2  int foo(int a, int b) {
3    if (a > 0) {
4      return a + bar(b);
5    } else {
6      return 0;
7    }
8  }
9  int bar(int a) {
10   return foo(a, a-1);
11 }
```

Ispis 4.12: Nužna deklaracija funkcije `bar` prije definicije funkcije `foo`.

U deklaraciji funkcije `bar` u retku `1`, navedeno je drugačije ime formalnog parametra nego u definiciji funkcije u retku `9`, ali to ne predstavlja grešku jer se imena parametara u deklaraciji ignoriraju.

Povratni tip funkcije može biti `char` ili `int` **bez** `const`-kvalifikatora, ili `void` što znači da funkcija ne vraća ništa. Semantička je pogreška ako funkcija deklarirana da vraća `void` pokuša vratiti neku vrijednost. Nadalje, ako funkcija deklarirana da vraća neku vrijednost brojevnog tipa u nekom slijedu izvođenja ne vrati vrijednost, rezultat izvođenja nije definiran (ali se ne radi o semantičkoj pogrešci, tj. semantički analizator ne treba provjeravati vraća li funkcija stvarno neku vrijednost u svim sljedovima izvođenja). Ako funkcija vraća neku vrijednost, tip vrijednosti mora se moći implicitno pretvoriti u deklarirani povratni tip funkcije (ovo pravilo pokriva i `void` povratni tip s obzirom na to da se niti jedan tip vrijednosti ne može implicitno pretvoriti u `void`).

U ispisu 4.13 je prikazana semantički ispravna funkcija `foo` i semantički neispravna funkcija `bar`. Funkcija `foo` je ispravna jer vrijedi `char` $\sim$ `int`, a funkcija `bar` je neispravna jer obrat ne vrijedi pa nije moguće vratiti `int` vrijednost iz funkcije kojoj je povratni tip `char`.

```
1  int foo(void) {
2    return 'a';
3  }
4
5  char bar(void) {
6    return 97; // greska
7  }
```

Ispis 4.13: Ispravna funkcija `foo` i neispravna funkcija `bar`.

Funkciju `bar` moglo bi se ispraviti korištenjem *cast* operatora, kao što je prikazano u ispisu 4.14.

```
1  char bar(void) {
2    return (char)97;
3  }
```

Ispis 4.14: Ispravljena funkcija `bar`.

Nadalje, hoće li se `return` naredba ikada izvesti ili ne nema utjecaja na semantičku ispravnost programa. Na primjer, funkcija u ispisu 4.15 također je neispravna, iako bi tijekom izvođenja uvijek vraćala vrijednost ispravnog tipa.

```
char bar(void) {
  if (0) {
    return 97; // greska
  } else {
    return (char)97;
  }
}
```

Ispis 4.15: Ispravnost tipova povratnih vrijednosti ne ovisi o tome hoće li se određena `return` naredba ikada izvršiti ili ne.

Prilikom poziva funkcije, *vrijednost* argumenata brojevnog tipa prenosi se u odgovarajuće parametre[10]. S druge strane, u formalne parametre tipa $niz(X)$ koji se deklariraju sintaksom $X$ `a[]` prenosi se *adresa* argumenta. U oba slučaja, tip vrijednosti argumenta mora se moći implicitno pretvoriti u tip odgovarajućeg parametra.

Razlika u načinu prenošenja argumenata u funkciju između brojevnih tipova i nizova prikazana je u ispisu 4.16.

```
void f(int x, int a[]) {
  x = x + 1;
  a[0] = a[0] + 1;
}

int main(void) {
  int x = 3;
  int a[8] = {0};
  f(x, a); // x == 3, a[0] == 1
  return 0;
}
```

Ispis 4.16: Brojevni tipovi se prenose *razmjenom vrijednosti*. Nizovi se prenose *razmjenom adresa*.

### 4.3.5 Operatori

Većina operatora u jeziku *ppjC* definirana je isključivo nad tipom `int`, a i rezultat je tipa `int`. Jedna donekle iznenađujuća posljedica ovog pravila u kombinaciji s pravilima o pretvorbi tipova je da je program u ispisu 4.17 neispravan. Naime, kako operator zbrajanja

[10] Način prenošenja argumenata ne utječe na ovu laboratorijsku vježbu, ali je ovdje naveden zbog cjelovitosti pregleda funkcija u jeziku *ppjC*.

očekuje operande tipa `int` i daje rezultat tipa `int`, vrijednost varijable `c` se implicitno pretvara u tip `int` i zbraja se s konstantom `1` (koja je tipa `int`). Rezultat operacije je opet tipa `int`, a kako ne vrijedi `int` $\sim$ `char`, radi se o semantičkoj pogrešci u pridruživanju.

```
1 int main(void) {
2   char c = 'a';
3   c = c + 1; // greska
4   return 0;
5 }
```

Ispis 4.17: Neispravan program s operatorom zbrajanja.

Pogreška se može lako ispraviti korištenjem eksplicitne promjene tipa, kao što je prikazano u ispisu 4.18.

```
1 int main(void) {
2   char c = 'a';
3   c = (char)(c + 1);
4   return 0;
5 }
```

Ispis 4.18: Ispravljen program iz ispisa 4.17.

Ako pri aritmetičkim operacijama dođe do preljeva ili podljeva, rezultat operacije nije definiran.

## 4.4 Semantička pravila jezika *ppjC*

Semantička pravila jezika definirana su u nastavku prirodnim jezikom i uz pomoć formalne notacije slične notaciji atributne prijevodne gramatike, a istovremeno je objašnjena i gramatika koja opisuje sintaksu jezika *ppjC*. Sve produkcije gramatike u BNF obliku nalaze se ovdje. Važno je razumjeti gramatiku ovog jezika jer se semantička analiza provodi nad generativnim stablom čiji oblik izravno ovisi o gramatici. Preporučljivo je prije čitanja semantičkih pravila pogledati produkcije gramatike i razmisliti o ulogama pojedinih znakova i produkcija. Gramatika se sastoji od tri skupine produkcija: produkcije za izraze, produkcije za naredbenu struktura podataka i produkcije za deklaraciju i definiciju varijabli i funkcija. Opisa semantičkih pravila u nastavku prati ovu strukturu.

### 4.4.1 Razlika između semantičke pogreške i *nedefiniranog ponašanja*

U nastavku ove upute, važno je razlikovati slučaj u kojem neki jezični konstrukt nije semantički ispravan i slučaj u kojem nije definiran *rezultat izvođenja* tog jezičnog konstrukta.

Kao primjer iz jezika *C*, semantička je pogreška pokušati zbrojiti dva pokazivača. Svaki ispravni *C* kompilator mora odbiti prevođenje programa koji sadrži ovu semantičku pogrešku. S druge strane, oduzimanje dva pokazivača na kompatibilne tipove semantički

je ispravno i svaki ispravni *C* kompilator program koji sadrži takvo oduzimanje mora prevesti (naravno, ako program ne sadrži druge pogreške). Međutim, pojednostavljeno rečeno, ako pokazivači ne pokazuju u isti niz, *rezultat oduzimanja* nije definiran (točnije, ponašanje takvog programa u cijelosti postaje nedefinirano)[11].

Drugi sličan primjer je dereferenciranje *null* pokazivača. U mnogim slučajevima kompilator ne može provjeriti hoće li neki pokazivač imati vrijednost *null* pokazivača tijekom izvođenja programa i samim time tijekom prevođenja ne može upozoriti programera da će se to dogoditi. Međutim, ako tijekom rada programa dođe do referenciranja *null* pokazivača, ponašanje programa postaje nedefinirano sa stanovišta *C* standarda.

To što je ponašanje programa u nekom slučaju nedefinirano u kontekstu izrade kompilatora znači da se o tim slučajevima ne treba posebno razmišljati (barem u okviru ove vježbe) — tijekom izvođenja prevedenog programa, može se dogoditi bilo što, uključujući i prestanak rada kompilatora, tj. u kontekstu ove vježbe, semantičkog analizatora. Posljedica ove činjenice je da vaša rješenja nećemo testirati sa generativnim stablima programa koji sadrže jezične konstrukte nedefiniranog ponašanja.

### 4.4.2 Obilazak stabla i opis semantičkih pravila

Kako generativno stablo prikazuje na koji način je iz gramatike generiran neki niz završnih znakova, struktura stabla u svakom čvoru jedinstveno određuje koja produkcija je primijenjena kako bi se znak u korijenu svakog podstabla zamijenio znakovima kojima su u stablu označena djeca tog korijena. U ovoj laboratorijskoj vježbi se pretpostavlja provjera semantičkih pravila dubinskim pretraživanjem generativnog stabla s lijeva na desno, pri čemu se teži provjeri semantičkog pravila što je prije moguće. To znači da se u produkcijama koje sadrže više završnih i nezavršnih znakova s desne strane produkcije, podstabla koja odgovaraju tim znakovima provjeravaju s lijeva na desno. Nakon svake provjere nekog podstabla s desne strane produkcije, provjeravaju se sva pravila u samoj produkciji za koja su dostupna sva potrebna svojstva znakova desne strane. Kako je redoslijed provjera važan jer o njemu ovisi ispis analizatora, uz svaku produkciju gramatike naveden je redoslijed provjere pravila. Drugim riječima, opis obilaska u ovom odlomku je čisto informativan, tj. služi kako bi bilo jasno zašto je u određenoj produkciji odabran baš taj redoslijed provjera pravila.

Za opis semantičkih pravila koriste se većinom izvedena i nekoliko nasljednih svojstava. Njihova uloga je opis pravila i nije obavezno u implementaciji koristiti ista svojstva ili svojstva uopće koristiti. Nadalje, semantička pravila nisu u potpunosti opisana pravilima računanja svojstava nego su dobrim dijelom opisana i prirodnim jezikom. Rješenje se testira isključivo na principu ulaz/izlaz, a način ostvarenja je proizvoljan dok god će rezultat biti jednak ovom konceptualnom modelu koji je opisan u uputi. Dakako, jedna mogućnost je implementirati upravo ovaj model obilaska stabla.

Semantička pravila navedena su po produkcijama, a produkcije su grupirane po nezavršnom znaku s lijeve strane. Produkcije za određene grupe operatora koje imaju identičnu strukturu i semantička pravila, ali različite operatore (na primjer produkcije koje generiraju operatore zbrajanja i oduzimanja), opisane su u okviru jedne produkcije koja na mjestu operatora ima regularan izraz koji pokriva sve moguće vrijednosti (na

[11] Tehnički pojam za ovakav slučaj je *nedefinirano ponašanje* (engl. *undefined behavior*).

primjer (PLUS | MINUS)).

Nezavršni znakovi su naslovi odjeljaka, nakon čega slijedi kratak opis uloge nezavršnog znaka u gramatici i popis produkcija s pripadnim semantičkim pravilima. Neposredno ispod svake produkcije navedena su pravila računanja izvedenih svojstava nezavršnog znaka s lijeve strane produkcije, a samo ime znaka zbog čitljivosti nije navedeno. Velika većina pravila računanja bi trebala biti sasvim očita nakon prvog čitanja, a složenija pravila su objašnjena riječima ispod produkcije.

Ispod pravila računanja su uz redne brojeve nabrojana semantička pravila koja je potrebno provjeriti i opisan je način obilaska tog podstabla. Za provjeru svih semantičkih pravila u podstablu čiji korijen je označen nekim nezavršnim znakom, koristi se oznaka *provjeri*(<ime_znaka>). Nužno je pratiti redoslijed kojim su pravila navedena, a cilj je da taj redoslijed bude očit po prethodno opisanom obrazloženju. Ako bilo koje pravilo nije zadovoljeno, semantički analizator treba ispisati danu produkciju u skladu s prethodnim opisom i završiti s radom.

Pravila računanja izvedenih svojstava nezavršnog znaka lijeve strane produkcije konceptualno se izvode nakon provjere svih pravila te produkcije i koriste vrijednosti svojstava izračunate tijekom provjere semantičkih pravila s desne strane produkcije. Kada u pravilima može doći do zabune zbog ponavljanja istog znaka više puta, uz znakove je naveden subskript.

Provjera pravila nad generativnim stablom započinje od korijena koji je uvijek označen nezavršnim znakom <prijevodna_jedinica>.

### 4.4.3 Ispis semantičkog analizatora u slučaju greške

Čim naiđe na prvu semantičku pogrešku, analizator treba ispisati produkciju u kojoj je pogreška otkrivena i završiti s radom. Uz završne znakove gramatike treba u zagradi ispisati i broj retka i pripadnu leksičku jedinku.

Na primjer, za pogrešku u naredbi `int x = 1 + "abc"` u drugom retku programa, ispis bi bio kao u nastavku.

```
1 <aditivni_izraz> ::= <aditivni_izraz> PLUS(2,+) <multiplikativni_izraz>
2
```

Ispis 4.19: Primjer ispisa.

Zašto se ispisuje ta produkcija bit će jasno kada pročitate semantička pravila, ali važno je uočiti da se uz uniformni znak operatora zbrajanja (PLUS) u zagradi ispisuje redak i leksička jedinka koju uniformni znak predstavlja. Prazan drugi redak znači samo to da je na kraju prvog retka znak za novi redak, kao i u svim vježbama do sada.

Dodatno, nakon obilaska stabla, analizator treba provjeriti još neka pravila koja su zajedno s pripadnim ispisom u slučaju greške opisana u poglavlju 4.4.7.

### 4.4.4 Izrazi

U gramatici jezika *ppjC* postoji veći broj nezavršnih znakova koji prikazuju neku vrstu izraza, a služe prvenstveno za osiguravanje ispravnog prioriteta operatora u izrazima. U nastavku poglavlja prikazane su različite vrste izraza i pripadna semantička pravila koja proizlaze iz značenja operatora.

Izrazi imaju izvedeno svojstvo *tip* koje sadrži oznaku tipa vrijednosti izraza. Dodatno, većina izraza i završni znak `IDN` imaju izvedeno svojstvo *l-izraz* koje označava da je izrazu moguće pridružiti neku vrijednost, tj. da izraz (među ostalim) može stajati s lijeve strane operatora pridruživanja[12]. Ovo svojstvo je logičko, tj. može imati vrijednosti `0` (ako izraz *nije l-izraz*) ili `1` (ako izraz *je l-izraz*). Od završnih znakova gramatike, jedino `IDN` (identifikator) može biti *l-izraz* i to samo ako predstavlja varijablu brojevnog tipa (`char` ili `int`) bez `const`-kvalifikatora. Identifikator koji predstavlja funkciju ili niz nije *l-izraz*.

`<primarni_izraz>`

Nezavršni znak `<primarni_izraz>` generira najjednostavnije izraze koji se sastoje od jednog identifikatora, neke vrste konstante ili izraza u zagradi.

- `<primarni_izraz> ::= IDN`
  *tip* ← `IDN`.*tip*
  *l-izraz* ← `IDN`.*l-izraz*

  1. `IDN`.*ime* je deklarirano

  U skladu s pravilima o djelokrugu deklaracija, provjera u točki 1 provodi se u lokalnom djelokrugu, prvom ugniježđujućem djelokrugu, i tako dalje sve do (uključujući) globalnog djelokruga (ili dok se deklaracija ne pronađe, što god bude prije). Tip identifikatora i je li identifikator *l-izraz* određuje se pomoću tablice znakova.

- `<primarni_izraz> ::= BROJ`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. vrijednost je u rasponu tipa `int`

- `<primarni_izraz> ::= ZNAK`
  *tip* ← `char`
  *l-izraz* ← `0`

  1. znak je ispravan po 4.3.2

- `<primarni_izraz> ::= NIZ_ZNAKOVA`
  *tip* ← *niz*(*const*(`char`))
  *l-izraz* ← `0`

[12] Svojstvo *l-izraz* u skladu s pravilima računanja efektivno označava da je izraz cjelobrojna varijabla bez `const`-kvalifikatora, opcionalno u zagradi. Ovo bi se moglo provjeriti i bez svojstva gdje je to potrebno, ali je ovo svojstvo uključeno kako bi se ilustriralo što se sa svojstvima može raditi u semantičkoj analizi osim praćenja tipova.

1. konstantni niz znakova je ispravan po 4.3.2

- `<primarni_izraz> ::= L_ZAGRADA <izraz> D_ZAGRADA`
  *tip* ← `<izraz>`.*tip*
  *l-izraz* ← `<izraz>`.*l-izraz*

  1. *provjeri*(`<izraz>`)

`<postfiks_izraz>`

Nezavršni znak `<postfiks_izraz>` generira neki primarni izraz s opcionalnim postfiks-operatorima.

- `<postfiks_izraz> ::= <primarni_izraz>`
  *tip* ← `<primarni_izraz>`.*tip*
  *l-izraz* ← `<primarni_izraz>`.*l-izraz*

  1. *provjeri*(`<primarni_izraz>`)

- `<postfiks_izraz> ::= <postfiks_izraz> L_UGL_ZAGRADA <izraz> D_UGL_ZAGRADA`
  *tip* ← $X$
  *l-izraz* ← $X \neq const(T)$

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*tip* $= niz(X)$
  3. *provjeri*(`<izraz>`)
  4. `<izraz>`.*tip* $\sim$ `int`

  Ova produkcija omogućuje indeksiranje nizova. Izraz kao što je `a[1][2]` je sintaksno dozvoljen, što je vidljivo iz ove i prethodne produkcije, ali predstavlja semantičku pogrešku, čak i ako je `a` tipa $niz(X)$. Naime, tip izraza `a[1]` će tada po pravilima računanja svojstva *tip* biti $X$. Novo indeksiranje cjelobrojnog tipa nije dozvoljeno i u koraku 2 dolazi do greške.

  U ispisu 4.20 prikazan je program koji sadrži ovu semantičku pogrešku, a u ispisu 4.21 je prikazan očekivani ispis semantičkog analizatora. Drugo indeksiranje je namjerno napisano u novom retku kako bi se ilustriralo da do semantičke pogreške dolazi upravo tada.

```
1 int main(void) {
2   int a[10];
3   a[1]
4    [2];
5   return 0;
6 }
```

Ispis 4.20: Program sa semantičkom pogreškom višestrukog indeksiranja niza.

```
1 <postfiks_izraz> ::= <postfiks_izraz> L_UGL_ZAGRADA(4,[) <izraz>
     D_UGL_ZAGRADA(4,])
2
```

Ispis 4.21: Ispis semantičkog analizatora za generativno stablo programa 4.20.

Redak je prelomljen jer ne stane na stranicu, ali između znaka `<izraz>` i znaka `D_UGL_ZAGRADA` treba se nalaziti točno jedan razmak.

- `<postfiks_izraz> ::= <postfiks_izraz> L_ZAGRADA D_ZAGRADA`
  *tip* ← *pov*
  *l-izraz* ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*tip* = *funkcija*(`void` → *pov*)

  Ova produkcija omogućuje pozivanje funkcija bez parametara, što se upravo i provjerava u točki 2. Nadalje, nije potrebno provjeriti je li funkcija koja se poziva deklarirana, zato jer će se to sigurno provjeriti u točki 1, tj. tijekom obrade nezavršnog znaka s desne strane produkcije. Povratnoj vrijednosti, bez obzira na tip, nije moguće ništa pridružiti, pa ona nikada nije *l-izraz*.

- `<postfiks_izraz> ::= <postfiks_izraz> L_ZAGRADA <lista_argumenata> D_ZAGRADA`
  *tip* ← *pov*
  *l-izraz* ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. *provjeri*(`<lista_argumenata>`)
  3. `<postfiks_izraz>`.*tip* = *funkcija*(*params* → *pov*) i redom po elementima *arg-tip* iz `<lista_argumenata>`.*tipovi* i *param-tip* iz *params* vrijedi *arg-tip* ∼ *param-tip*

  Ova produkcija omogućuje poziv funkcije s argumentima i razlikuje se od prethodne produkcije po tome što se tipovi argumenata pokušavaju implicitno pretvoriti u tipove parametara funkcije.

- `<postfiks_izraz> ::= <postfiks_izraz> (OP_INC | OP_DEC)`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*l-izraz* = `1` i `<postfiks_izraz>`.*tip* ∼ `int`

  I prefiks i postfiks inkrement operatori u dijelu promjene vrijednosti varijable imaju značenje kao naredba `v = (v.`*tip*`)(v + 1);`. Za dekrement operatore značenje je isto uz zamjenu operatora zbrajanja s operatorom oduzimanja. Drugim riječima, moguće je inkrementirati ili dekrementirati varijable tipova `int` i `char`. Provjera

vrijednosti svojstva *l-izraz* u točki dva osigurava da se radi o varijabli bez `const`-kvalifikatora, a zajedno s drugim uvjetom osigurava se da se radi o varijabli brojevnog tipa. Važno je uočiti da rezultat primjene ovih operatora više nije *l-izraz*, nego je vrijednost tipa `int`[13].

`<lista_argumenata>`

Nezavršni znak `<lista_argumenata>` generira listu argumenata za poziv funkcije, a za razliku od nezavršnih znakova koji generiraju izraze, imat će svojsto *tipovi* koje predstavlja listu tipova argumenata, s lijeva na desno.

- `<lista_argumenata> ::= <izraz_pridruzivanja>`
  *tipovi* ← `[ <izraz_pridruzivanja>.` *tip* `]`

  1. *provjeri*(`<izraz_pridruzivanja>`)

  Ova produkcija generira krajnje lijevi (moguće i jedini) argument i postavlja njegov tip kao jedini element liste u svojstvu *tipovi*.

- `<lista_argumenata> ::= <lista_argumenata> ZAREZ <izraz_pridruzivanja>`
  *tipovi* ← `<lista_argumenata>.` *tipovi* `+ [ <izraz_pridruzivanja>.` *tip* `]`

  1. *provjeri*(`<lista_argumenata>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

  Ova produkcija omogućuje nizanje argumenata odvojenih zarezom. Tip novog argumenta koji je predstavljen nezavršnim znakom `<izraz_pridruzivanja>` dodaje se na desni kraj liste tipova koji su određeni za prethodne argumente.

`<unarni_izraz>`

Nezavršni znak `<unarni_izraz>` generira izraze s opcionalnim prefiks unarnim operatorima.

- `<unarni_izraz> ::= <postfiks_izraz>`
  *tip* ← `<postfiks_izraz>.` *tip*
  *l-izraz* ← `<postfiks_izraz>.` *l-izraz*

  1. *provjeri*(`<postfiks_izraz>`)

- `<unarni_izraz> ::= (OP_INC | OP_DEC) <unarni_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<unarni_izraz>`)
  2. `<unarni_izraz>.` *l-izraz* $=$ `1` i `<unarni_izraz>.` *tip* $\sim$ `int`

[13] Značenje operatora u širem kontekstu izraza je vrlo slično kao u *C*-u, ali nije važno za potrebe semantičke analize.

Prefiks inkrement i dekrement imaju analogna semantička pravila postfiks inačicama istih operatora.

- `<unarni_izraz> ::= <unarni_operator> <cast_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<cast_izraz>`)
  2. `<cast_izraz>`.*tip* ∼ `int`

  Unarni operatori primjenjivi su na vrijednosti tipa `int` što se provjerava u točki 2, a rezultat je opet tipa `int`. Iako je u produkciji nezavršni znak `<unarni_operator>`, nije potrebno provjeravati nikakva semantička pravila u toj grani stabla jer taj nezavršni znak jednostavno generira neki od unarnih operatora (što je prikazano u nastavku). Konačno, bez obzira na to je li `<cast_izraz>` *l-izraz* ili ne, rezultat primjene unarnog operatora je samo *vrijednost* i nije *l-izraz*. Na primjer, naredba `+a = 3;` je semantički neispravna zbog ovog pravila.

#### `<unarni_operator>`

Nezavršni znak `<unarni_operator>` generira aritmetičke (`PLUS` i `MINUS`), bitovne (`OP_TILDA`) i logičke (`OP_NEG`) prefiks unarne operatore. Kako u ovim produkcijama u semantičkoj analizi ne treba ništa provjeriti, produkcije ovdje nisu navedene.

#### `<cast_izraz>`

Nezavršni znak `<cast_izraz>` generira izraze s opcionalnim *cast* operatorom.

- `<cast_izraz> ::= <unarni_izraz>`
  *tip* ← `<unarni_izraz>`.*tip*
  *l-izraz* ← `<unarni_izraz>`.*l-izraz*

  1. *provjeri*(`<unarni_izraz>`)

- `<cast_izraz> ::= L_ZAGRADA <ime_tipa> D_ZAGRADA <cast_izraz>`
  *tip* ← `<ime_tipa>`.*tip*
  *l-izraz* ← `0`

  1. *provjeri*(`<ime_tipa>`)
  2. *provjeri*(`<cast_izraz>`)
  3. `<cast_izraz>`.*tip* se može pretvoriti u `<ime_tipa>`.*tip* po poglavlju 4.3.1

  Ova produkcija omogućuje eksplicitne promjene tipa vrijednosti. Kako se nezavršni znak `<ime_tipa>` koristi na više mjesta u gramatici, on može generirati i tip `void` iako će to u kontekstu *cast* operatora uvijek biti semantička greška. Nadalje, vrijednost kojoj se tip pokušava promijeniti može biti bilo kojeg tipa, pa se u točki 2 mora provjeriti da se radi o brojevnom tipu, u skladu s pravilima iz poglavlja 4.3.1.

  Na primjer, svi izrazi u ispisu 4.22 su semantički ispravni, pod pretpostavkom da su varijable `x` i `y` brojevnog tipa.

```
(int)'a'
(const char)x
(const int)'a'
(char)((const int)300 + (int)'a')
(int)(char)(const int)(const char)(x + y)
```

Ispis 4.22: Ispravne primjene *cast*-operatora.

`<ime_tipa>`

Nezavršni znak `<ime_tipa>` generira imena opcionalno `const`-kvalificiranih brojevnih tipova i ključnu riječ `void`. U ovim produkcijama će se izračunati izvedeno svojstvo *tip* koje se koristi u produkcijama gdje se `<ime_tipa>` pojavljuje s desne strane i dodatno će se onemogućiti tip `const void` (koji je sintaksno ispravan, ali nema smisla).

- `<ime_tipa> ::= <specifikator_tipa>`
  *tip* ← `<specifikator_tipa>`.*tip*

  1. *provjeri*(`<specifikator_tipa>`)

  Prva produkcija koristi se za tipove koji nisu `const`-kvalificirani.

- `<ime_tipa> ::= KR_CONST <specifikator_tipa>`
  *tip* ← *const*(`<specifikator_tipa>`.*tip*)

  1. *provjeri*(`<specifikator_tipa>`)
  2. `<specifikator_tipa>`.*tip* ≠ `void`

  U drugoj produkciji generiraju se `const`-kvalificirani tipovi. Kao što je prije spomenuto, u točki 2 se onemogućuje tip `const void`.

`<specifikator_tipa>`

Nezavršni znak `<specifikator_tipa>` generira jedan od tri završna znaka `KR_VOID`, `KR_CHAR` i `KR_INT`. U semantičkoj analizi ćemo iz završnog znaka odrediti vrijednost svojstva *tip* nezavršnog znaka, ali u ovim produkcijama ne može doći do semantičke pogreške.

- `<specifikator_tipa> ::= KR_VOID`
  *tip* ← `void`

- `<specifikator_tipa> ::= KR_CHAR`
  *tip* ← `char`

- `<specifikator_tipa> ::= KR_INT`
  *tip* ← `int`

`<multiplikativni_izraz>`

Nezavršni znak `<multiplikativni_izraz>` generira izraze u kojima se opcionalno koriste operatori množenja, dijeljenja i ostatka. Struktura produkcija osigurava da sva tri operatora imaju isti prioritet i to manji od unarnih (prefiks i postfiks) operatora, a veći od ostalih operatora. Nadalje, lijeva asocijativnost ovih operatora osigurana je lijevom rekurzijom u produkcijama.

Svi ostali binarni operatori u jeziku (čija pravila su prikazana kasnije) ostvareni su sličnim produkcijama i provjeravaju se slična pravila.

- `<multiplikativni_izraz> ::= <cast_izraz>`
  *tip* ← `<cast_izraz>`.*tip*
  *l-izraz* ← `<cast_izraz>`.*l-izraz*

  1. *provjeri*(`<cast_izraz>`)

- `<multiplikativni_izraz> ::= <multiplikativni_izraz> (OP_PUTA | OP_DIJELI | OP_MOD) <cast_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<multiplikativni_izraz>`)
  2. `<multiplikativni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<cast_izraz>`)
  4. `<cast_izraz>`.*tip* ∼ `int`

  Ova produkcija generira operator množenja, dijeljenja ili ostatka između dva podizraza. Kao i većina binarnih operatora u jeziku, ovi operatiri su definirani samo nad vrijednostima tipa `int` i rezultat provođenja operacije je opet tipa `int`. U točkama 2 i 4 provjerava se mogu li se lijevi i desni operand pretvoriti u vrijednosti tipa `int`. Važno je uočiti da znak `<multiplikativni_izraz>` može imati bilo koji tip (iako je rezultat primjene bilo kojeg od ova tri operatora tipa `int`) zato što u prethodnoj produkciji `<multiplikativni_izraz>` preuzima tip od znaka `<cast_izraz>` i potencijalno tako dalje sve do znaka `<primarni_izraz>`.

`<aditivni_izraz>`

Nezavršni znak `<aditivni_izraz>` generira izraze u kojima se opcionalno koriste operatori zbrajanja i oduzimanja.

- `<aditivni_izraz> ::= <multiplikativni_izraz>`
  *tip* ← `<multiplikativni_izraz>`.*tip*
  *l-izraz* ← `<multiplikativni_izraz>`.*l-izraz*

  1. *provjeri*(`<multiplikativni_izraz>`)

- `<aditivni_izraz> ::= <aditivni_izraz> (PLUS | MINUS) <multiplikativni_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<aditivni_izraz>`)
  2. `<aditivni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<multiplikativni_izraz>`)
  4. `<multiplikativni_izraz>`.*tip* ∼ `int`

`<odnosni_izraz>`

Nezavršni znak `<odnosni_izraz>` generira izraze u kojima se opcionalno koriste odnosni operatori `<` (uniformni znak `OP_LT`), `>` (uniformni znak `OP_GT`), `<=` (uniformni znak `OP_LTE`) i `>=` (uniformni znak `OP_GTE`).

- `<odnosni_izraz> ::= <aditivni_izraz>`
  *tip* ← `<aditivni_izraz>`.*tip*
  *l-izraz* ← `<aditivni_izraz>`.*l-izraz*

  1. *provjeri*(`<aditivni_izraz>`)

- `<odnosni_izraz> ::= <odnosni_izraz> (OP_LT | OP_GT | OP_LTE | OP_GTE) <aditivni_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<odnosni_izraz>`)
  2. `<odnosni_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<aditivni_izraz>`)
  4. `<aditivni_izraz>`.*tip* ∼ `int`

`<jednakosni_izraz>`

Nezavršni znak `<jednakosni_izraz>` generira izraze u kojima se opcionalno koriste jednakosni operatori `==` (uniformni znak `OP_EQ`) i `!=` (uniformni znak `OP_NEQ`).

- `<jednakosni_izraz> ::= <odnosni_izraz>`
  *tip* ← `<odnosni_izraz>`.*tip*
  *l-izraz* ← `<odnosni_izraz>`.*l-izraz*

  1. *provjeri*(`<odnosni_izraz>`)

- `<jednakosni_izraz> ::= <jednakosni_izraz> (OP_EQ | OP_NEQ) <odnosni_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<jednakosni_izraz>`)

2. `<jednakosni_izraz>`.*tip* ∼ `int`
3. *provjeri*(`<odnosni_izraz>`)
4. `<odnosni_izraz>`.*tip* ∼ `int`

`<bin_i_izraz>`

Nezavršni znak `<bin_i_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `&` (uniformni znak `OP_BIN_I`)[14]. Bitovni operatori imaju različite prioritete pa zato svaki operator ima pripadni nezavršni znak.

- `<bin_i_izraz> ::= <jednakosni_izraz>`
  *tip* ← `<jednakosni_izraz>`.*tip*
  *l-izraz* ← `<jednakosni_izraz>`.*l-izraz*
  1. *provjeri*(`<jednakosni_izraz>`)
- `<bin_i_izraz> ::= <bin_i_izraz> OP_BIN_I <jednakosni_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`
  1. *provjeri*(`<bin_i_izraz>`)
  2. `<bin_i_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<jednakosni_izraz>`)
  4. `<jednakosni_izraz>`.*tip* ∼ `int`

`<bin_xili_izraz>`

Nezavršni znak `<bin_xili_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `^` (uniformni znak `OP_BIN_XILI`).

- `<bin_xili_izraz> ::= <bin_i_izraz>`
  *tip* ← `<bin_i_izraz>`.*tip*
  *l-izraz* ← `<bin_i_izraz>`.*l-izraz*
  1. *provjeri*(`<bin_i_izraz>`)
- `<bin_xili_izraz> ::= <bin_xili_izraz> OP_BIN_XILI <bin_i_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`
  1. *provjeri*(`<bin_xili_izraz>`)
  2. `<bin_xili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_i_izraz>`)
  4. `<bin_i_izraz>`.*tip* ∼ `int`

[14] Smislenije ime nezavršnog i završnog znaka sadržavalo bi *bit* umjesto *bin*, ali ova greška u imenu postoji zbog konzistentnosti s drugim materijalima.

`<bin_ili_izraz>`

Nezavršni znak `<bin_ili_izraz>` generira izraze u kojima se opcionalno koristi bitovni operator `|` (uniformni znak `OP_BIN_ILI`).

- `<bin_ili_izraz> ::= <bin_xili_izraz>`
  *tip* ← `<bin_xili_izraz>`.*tip*
  *l-izraz* ← `<bin_xili_izraz>`.*l-izraz*

  1. *provjeri*(`<bin_xili_izraz>`)

- `<bin_ili_izraz> ::= <bin_ili_izraz> OP_BIN_ILI <bin_xili_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<bin_ili_izraz>`)
  2. `<bin_ili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_xili_izraz>`)
  4. `<bin_xili_izraz>`.*tip* ∼ `int`

`<log_i_izraz>`

Nezavršni znak `<log_i_izraz>` generira izraze u kojima se opcionalno koristi logički operator konjunkcije `&&` (uniformni znak `OP_I`). Slično kao za bitovne operatore, kako logički operatori imaju različite prioritete svakom je pridružen vlastiti nezavršni znak.

- `<log_i_izraz> ::= <bin_ili_izraz>`
  *tip* ← `<bin_ili_izraz>`.*tip*
  *l-izraz* ← `<bin_ili_izraz>`.*l-izraz*

  1. *provjeri*(`<bin_ili_izraz>`)

- `<log_i_izraz> ::= <log_i_izraz> OP_I <bin_ili_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<log_i_izraz>`)
  2. `<log_i_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<bin_ili_izraz>`)
  4. `<bin_ili_izraz>`.*tip* ∼ `int`

`<log_ili_izraz>`

Nezavršni znak `<log_ili_izraz>` generira izraze u kojima se opcionalno koristi logički operator disjunkcije `||` (uniformni znak `OP_ILI`).

- `<log_ili_izraz> ::= <log_i_izraz>`
  *tip* ← `<log_i_izraz>`.*tip*
  *l-izraz* ← `<log_i_izraz>`.*l-izraz*

  1. *provjeri*(`<log_i_izraz>`)

- `<log_ili_izraz> ::= <log_ili_izraz> OP_ILI <log_i_izraz>`
  *tip* ← `int`
  *l-izraz* ← `0`

  1. *provjeri*(`<log_ili_izraz>`)
  2. `<log_ili_izraz>`.*tip* ∼ `int`
  3. *provjeri*(`<log_i_izraz>`)
  4. `<log_i_izraz>`.*tip* ∼ `int`

`<izraz_pridruzivanja>`

Nezavršni znak `<izraz_pridruzivanja>` generira izraze u kojima se neka vrijednost opcionalno pridružuje varijabli koristeći operator pridruživanja `=` (uniformni znak `OP_PRIDRUZI`). Za razliku od prethodno prikazanih binarnih operatora, operator pridruživanja je desno asocijativan što je u gramatici osigurano primjenom desne rekurzije. Desna asocijativnost omogućuje nizanje pridruživanja, npr. `a = b = c = 42;`.

- `<izraz_pridruzivanja> ::= <log_ili_izraz>`
  *tip* ← `<log_ili_izraz>`.*tip*
  *l-izraz* ← `<log_ili_izraz>`.*l-izraz*

  1. *provjeri*(`<log_ili_izraz>`)

- `<izraz_pridruzivanja> ::= <postfiks_izraz> OP_PRIDRUZI <izraz_pridruzivanja>`
  *tip* ← `<postfiks_izraz>`.*tip*
  *l-izraz* ← `0`

  1. *provjeri*(`<postfiks_izraz>`)
  2. `<postfiks_izraz>`.*l-izraz* = `1`
  3. *provjeri*(`<izraz_pridruzivanja>`)
  4. `<izraz_pridruzivanja>`.*tip* ∼ `<postfiks_izraz>`.*tip*

  Na primjer, u skladu s pravilima računanja svojstva *l-izraz*, za varijablu `a` tipa `int` semantički su ispravna pridruživanja `a = 'a';` i `(a) = 10;`.

`<izraz>`

Nezavršni znak `<izraz>` omogućuje opcionalno nizanje izraza koristeći operator `,` (uniformni znak `ZAREZ`). Vrijednost takvog složenog izraza jednaka je vrijednosti krajnje desnog izraza u nizu.

- `<izraz> ::= <izraz_pridruzivanja>`
  *tip* ← `<izraz_pridruzivanja>`.*tip*
  *l-izraz* ← `<izraz_pridruzivanja>`.*l-izraz*

  1. *provjeri*(`<izraz_pridruzivanja>`)

- `<izraz> ::= <izraz> ZAREZ <izraz_pridruzivanja>`
  *tip* ← `<izraz_pridruzivanja>`.*tip*
  *l-izraz* ← `0`

  1. *provjeri*(`<izraz>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

### 4.4.5 Naredbena struktura programa

Programi u jeziku *ppjC* se sastoje od deklaracija i definicija varijabli i funkcija. Tijelo svake funkcije je *složena naredba* tj. blok. Blokovi se sastoje od deklaracija funkcija, definicija (ujedno i deklaracija) varijabli i niza naredbi. Ova struktura programa ostvarena je nezavršnim znakovima i produkcijama opisanim u nastavku. Skup produkcija za naredbenu strukturu programa semantički je znatno jednostavniji od produkcija za izraze.

`<slozena_naredba>`

Nezavršni znak `<slozena_naredba>` predstavlja blok naredbi koji opcionalno počinje listom deklaracija. Svaki blok je odvojeni djelokrug, a nelokalnim imenima se pristupa u ugniježđujućem bloku (i potencijalno tako dalje sve do globalnog djelokruga).

- `<slozena_naredba> ::= L_VIT_ZAGRADA <lista_naredbi> D_VIT_ZAGRADA`

  1. *provjeri*(`<lista_naredbi>`)

  Ova produkcija generira blok koji nema vlastite deklaracije (ali neka od naredbi u bloku može biti novi blok koji ima lokalne deklaracije).

- `<slozena_naredba> ::= L_VIT_ZAGRADA <lista_deklaracija>`
  `<lista_naredbi> D_VIT_ZAGRADA`

  1. *provjeri*(`<lista_deklaracija>`)
  2. *provjeri*(`<lista_naredbi>`)

  S druge strane, ova produkcija generira blok s lokalnim deklaracijama. Kao i u jeziku *C*, deklaracije su dozvoljene samo na početku bloka.

`<lista_naredbi>`

Nezavršni znak `<lista_naredbi>` omogućuje nizanje naredbi u bloku.

- `<lista_naredbi> ::= <naredba>`

1. *provjeri*(<naredba>)

- <lista_naredbi> ::= <lista_naredbi> <naredba>

  1. *provjeri*(<lista_naredbi>)
  2. *provjeri*(<naredba>)

<naredba>

Nezavršni znak <naredba> generira blokove (<slozena_naredba>) i različite vrste jednostavnih naredbi (<izraz_naredba>, <naredba_grananja>, <naredba_petlje> i <naredba_skoka>). Kako su sve produkcije jedinične (s desne strane imaju jedan nezavršni znak) i u svim produkcijama se provjeravaju semantička pravila na znaku s desne strane, produkcije ovdje nisu prikazane.

<izraz_naredba>

Nezavršni znak <izraz_naredba> generira opcionalni izraz i znak ; (uniformni znak TOCKAZAREZ) i predstavlja jednostavnu naredbu. Za potrebe uvjeta u for-petlji, znaku <izraz_naredba> se pridjeljuje izvedeno svojstvo *tip*.

- <izraz_naredba> ::= TOCKAZAREZ
  *tip* ← int

  Ova produkcija generira "praznu naredbu" koja može biti korisna kao tijelo petlje koja ne radi ništa i slično. Prazna naredba će imati tip int kako bi se mogla koristiti kao uvijek zadovoljen uvjet u for-petlji (na primjer u kakonskoj beskonačnoj petlji for(;;)).

- <izraz_naredba> ::= <izraz> TOCKAZAREZ
  *tip* ← <izraz>.*tip*

  1. *provjeri*(<izraz>)

  Za zadani izraz, svojstvo *tip* se preuzima od znaka <izraz>.

<naredba_grananja>

Nezavršni znak <naredba_grananja> generira if naredbu u jeziku.

- <naredba_grananja> ::= KR_IF L_ZAGRADA <izraz> D_ZAGRADA <naredba>

  1. *provjeri*(<izraz>)
  2. <izraz>.*tip* ∼ int
  3. *provjeri*(<naredba>)

  Ova produkcija generira if naredbu bez else dijela.

- `<naredba_grananja> ::= KR_IF L_ZAGRADA <izraz> D_ZAGRADA <naredba>`$_1$ `KR_ELSE <naredba>`$_2$

  1. *provjeri*(`<izraz>`)
  2. `<izraz>`.*tip* $\sim$ `int`
  3. *provjeri*(`<naredba>`$_1$)
  4. *provjeri*(`<naredba>`$_2$)

`<naredba_petlje>`

Nezavršni znak `<naredba_petlje>` generira `while` i `for` petlje.

- `<naredba_petlje> ::= KR_WHILE L_ZAGRADA <izraz> D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz>`)
  2. `<izraz>`.*tip* $\sim$ `int`
  3. *provjeri*(`<naredba>`)

- `<naredba_petlje> ::= KR_FOR L_ZAGRADA <izraz_naredba>`$_1$ `<izraz_naredba>`$_2$ `D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz_naredba>`$_1$)
  2. *provjeri*(`<izraz_naredba>`$_2$)
  3. `<izraz_naredba>`$_2$.*tip* $\sim$ `int`
  4. *provjeri*(`<naredba>`)

  Ova produkcija generira `for`-petlju bez opcionalnog izraza koji se tipično koristi za promjenu indeksa petlje dok sljedeća produkcija generira petlju sa tim izrazom.

- `<naredba_petlje> ::= KR_FOR L_ZAGRADA <izraz_naredba>`$_1$ `<izraz_naredba>`$_2$ `<izraz> D_ZAGRADA <naredba>`

  1. *provjeri*(`<izraz_naredba>`$_1$)
  2. *provjeri*(`<izraz_naredba>`$_2$)
  3. `<izraz_naredba>`$_2$.*tip* $\sim$ `int`
  4. *provjeri*(`<izraz>`)
  5. *provjeri*(`<naredba>`)

`<naredba_skoka>`

Nezavršni znak `<naredba_skoka>` generira `continue`, `break` i `return` naredbe.

- `<naredba_skoka> ::= (KR_CONTINUE | KR_BREAK) TOCKAZAREZ`

  1. naredba se nalazi unutar petlje ili unutar bloka koji je ugniježđen u petlji

I naredba `continue` (uniformni znak `KR_CONTINUE`) i naredba `break` (uniformni znak `KR_BREAK`) dozvoljene su isključivo unutar neke petlje, a imaju isto značenje kao u jeziku *C*.

- `<naredba_skoka> ::= KR_RETURN TOCKAZAREZ`

  1. naredba se nalazi unutar funkcije tipa *funkcija*(*params* $\rightarrow$ `void`)

  Naredba `return` bez povratne vrijednosti može se koristiti jedino u funkcijama koje ne vraćaju ništa.

- `<naredba_skoka> ::= KR_RETURN <izraz> TOCKAZAREZ`

  1. *provjeri*(`<izraz>`)
  2. naredba se nalazi unutar funkcije tipa *funkcija*(*params* $\rightarrow$ *pov*) i vrijedi `<izraz>`.*tip* $\sim$ *pov*

`<prijevodna_jedinica>`

Nezavršni znak `<prijevodna_jedinica>` je početni nezavršni znak gramatike i generira niz nezavršnih znakova `<vanjska_deklaracija>` koji generiraju definicije (i deklaracije) u globalnom djelokrugu programa.

- `<prijevodna_jedinica> ::= <vanjska_deklaracija>`

  1. *provjeri*(`<vanjska_deklaracija>`)

- `<prijevodna_jedinica> ::= <prijevodna_jedinica> <vanjska_deklaracija>`

  1. *provjeri*(`<prijevodna_jedinica>`)
  2. *provjeri*(`<vanjska_deklaracija>`)

`<vanjska_deklaracija>`

Nezavršni znak `<vanjska_deklaracija>` generira ili definiciju funkcije (znak `<definicija_funkcije>`) ili deklaraciju varijable ili funkcije (znak `<deklaracija>`). Obje produkcije su jedinične i u obje se provjeravaju pravila u podstablu kojem je znak s desne strane korijen.

### 4.4.6 Deklaracije i definicije

Preostale produkcije u gramatici odnose se na deklaracije i definicije. U semantičkoj analizi se na osnovi dijela generativnog stabla označenog ovim produkcijama gradi (ili dopunjuje) tablica znakova.

`<definicija_funkcije>`

- `<definicija_funkcije> ::= <ime_tipa> IDN L_ZAGRADA KR_VOID D_ZAGRADA <slozena_naredba>`

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>`.*tip* $\neq$ *const*($T$)
  3. ne postoji prije definirana funkcija imena `IDN`.*ime*
  4. ako postoji deklaracija imena `IDN`.*ime* u globalnom djelokrugu onda je pripadni tip te deklaracije *funkcija*(`void` $\rightarrow$ `<ime_tipa>`.*tip*)
  5. zabilježi definiciju i deklaraciju funkcije
  6. *provjeri*(`<slozena_naredba>`)

  Točka 2 u skladu s definicijom znaka `<ime_tipa>` osigurava da je povratni tip funkcije `int`, `char` ili `void`. U točki 3 osigurava se pravilo da svaka funkcija može biti definirana najviše jednom. Konačno, u točki 4 se provjerava da prethodne deklaracije te funkcije u **globalnom djelokrugu** imaju isti tip kao i sama definicija funkcije. Provjerava se globalni djelokrug jer sintaksa jezika osigurava da se sve funkcije definiraju u globalnom djelokrugu.

  Ako su sva ova pravila zadovoljena, definicija i deklaracija funkcije se zapisuju u odgovarajuće strukture podataka **prije** obrade podstabla kojem je korijen označen znakom `<slozena_naredba>`. Drugim riječima, funkcija je u svom tijelu već deklarirana i može se rekurzivno pozivati.

  Djelokrug definiran složenom naredbom, tj. tijelom funkcije, pridružuje se definiciji funkcije, npr. kako bi bilo moguće provjeriti pravila vezana uz `return` naredbu.

- `<definicija_funkcije> ::= <ime_tipa> IDN L_ZAGRADA <lista_parametara> D_ZAGRADA <slozena_naredba>`

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>`.*tip* $\neq$ *const*($T$)
  3. ne postoji prije definirana funkcija imena `IDN`.*ime*
  4. *provjeri*(`<lista_parametara>`)
  5. ako postoji deklaracija imena `IDN`.*ime* u globalnom djelokrugu onda je pripadni tip te deklaracije *funkcija*(`<lista_parametara>`.*tipovi* $\rightarrow$ `<ime_tipa>`.*tip*)
  6. zabilježi definiciju i deklaraciju funkcije
  7. *provjeri*(`<slozena_naredba>`) uz parametre funkcije koristeći `<lista_parametara>`.*tipovi* i `<lista_parametara>`.*imena*.

  Ova produkcija generira definicije funkcija s listom parametara, tj. funkcije koje primaju jedan ili više argumenata. Za točku 7 je zato važno osigurati da se prije provjere pravila u tijelu funkcije u lokalni djelokrug ugrade parametri funkcije.

`<lista_parametara>`

Nezavršnom znaku `<lista_parametara>` pridružit ćemo svojstvo *tipovi* koje sadrži listu tipova parametara i svojstvo *imena* koje sadrži imena parametara. Vrijednosti svojstva grade se analogno kao kod znaka `<lista_argumenata>`.

- `<lista_parametara> ::= <deklaracija_parametra>`
  *tipovi* ← `[ <deklaracija_parametra>.` *tip* `]`
  *imena* ← `[ <deklaracija_parametra>.` *ime* `]`

  1. *provjeri*(`<deklaracija_parametra>`)

- `<lista_parametara> ::= <lista_parametara> ZAREZ <deklaracija_parametra>`
  *tipovi* ← `<lista_parametara>.` *tipovi* `+ [ <deklaracija_parametra>.` *tip* `]`
  *imena* ← `<lista_parametara>.` *imena* `+ [ <deklaracija_parametra>.` *ime* `]`

  1. *provjeri*(`<lista_parametara>`)
  2. *provjeri*(`<deklaracija_parametra>`)
  3. `<deklaracija_parametra>.` *ime* ne postoji u `<lista_parametara>.` *imena*

  U točki 3 se provjerava jedinstvenost imena parametara u jednoj deklaraciji funkcije.

`<deklaracija_parametra>`

Nezavršni znak `<deklaracija_parametra>` služi za deklaraciju jednog parametra i ima svojstva *tip* i *ime*.

- `<deklaracija_parametra> ::= <ime_tipa> IDN`
  *tip* ← `<ime_tipa>.` *tip*
  *ime* ← `IDN.` *ime*

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>.` *tip* $\neq$ `void`

  Ova produkcija generira deklaraciju cjelobrojnog parametra.

- `<deklaracija_parametra> ::= <ime_tipa> IDN L_UGL_ZAGRADA D_UGL_ZAGRADA`
  *tip* ← *niz*(`<ime_tipa>.` *tip*)
  *ime* ← `IDN.` *ime*

  1. *provjeri*(`<ime_tipa>`)
  2. `<ime_tipa>.` *tip* $\neq$ `void`

  Ova produkcija generira parametre koji su nizovi.

`<lista_deklaracija>`

Nezavršni znak `<lista_deklaracija>` generira jednu ili više deklaracija na početku bloka.

- `<lista_deklaracija> ::= <deklaracija>`
  1. *provjeri*(`<deklaracija>`)
- `<lista_deklaracija> ::= <lista_deklaracija> <deklaracija>`
  1. *provjeri*(`<lista_deklaracija>`)
  2. *provjeri*(`<deklaracija>`)

`<deklaracija>`

Nezavršni znak `<deklaracija>` generira jednu naredbu deklaracije.

- `<deklaracija> ::= <ime_tipa> <lista_init_deklaratora> TOCKAZAREZ`
  1. *provjeri*(`<ime_tipa>`)
  2. *provjeri*(`<lista_init_deklaratora>`) uz nasljedno svojstvo `<lista_init_deklaratora>`.*ntip* ← `<ime_tipa>`.*tip*

Specifičnost točke 2 je nasljedno svojstvo *ntip* nezavršnog znaka `<lista_init_deklaratora>`. Svojstvo *ntip* služi za prijenos jednog dijela informacije o tipu u sve deklaratore. Za varijable brojevnog tipa *ntip* će biti cijeli tip, za nizove će biti tip elementa niza, a za funkcije će biti povratni tip.

`<lista_init_deklaratora>`

Nezavršni znak `<lista_init_deklaratora>` generira deklaratore odvojene zarezima. Na primjer, u naredbi `int x, y=3, z=y+1;`, znak `<lista_init_deklaratora>` generira `x, y=3, z=y+1` dio (dakako, generira odgovarajuće uniformne znakove).

- `<lista_init_deklaratora> ::= <init_deklarator>`
  1. *provjeri*(`<init_deklarator>`) uz nasljedno svojstvo `<init_deklarator>`.*ntip* ← `<lista_init_deklaratora>`.*ntip*
- `<lista_init_deklaratora>`$_1$ `::= <lista_init_deklaratora>`$_2$ `ZAREZ <init_deklarator>`
  1. *provjeri*(`<lista_init_deklaratora>`$_2$) uz nasljedno svojstvo `<lista_init_deklaratora>`$_2$.*ntip* ← `<lista_init_deklaratora>`$_1$.*ntip*
  2. *provjeri*(`<init_deklarator>`) uz nasljedno svojstvo `<init_deklarator>`.*ntip* ← `<lista_init_deklaratora>`$_1$.*ntip*

`<init_deklarator>`

Nezavršni znak `<init_deklarator>` generira deklarator s opcionalnim inicijalizatorom.

- `<init_deklarator> ::= <izravni_deklarator>`

  1. *provjeri*(`<izravni_deklarator>`) uz nasljedno svojstvo
     `<izravni_deklarator>`.*ntip* $\leftarrow$ `<init_deklarator>`.*ntip*
  2. `<izravni_deklarator>`.*tip* $\neq$ *const*($T$)
     i
     `<izravni_deklarator>`.*tip* $\neq$ *niz*(*const*($T$))

  Točka 2 osigurava da `const`-kvalificirane varijable i nizovi `const`-kvalificiranih tipova moraju biti inicijalizirani pri definiciji kako bi imali određenu vrijednost. Važno je uočiti da se provjerava izvedeno svojstvo *tip* znaka `<izravni_deklarator>`, a ne nasljedno svojstvo *ntip*.

- `<init_deklarator> ::= <izravni_deklarator> OP_PRIDRUZI <inicijalizator>`

  1. *provjeri*(`<izravni_deklarator>`) uz nasljedno svojstvo
     `<izravni_deklarator>`.*ntip* $\leftarrow$ `<init_deklarator>`.*ntip*
  2. *provjeri*(`<incijalizator>`)
  3. ako je `<izravni_deklarator>`.*tip* $T$ ili *const*($T$)
     `<inicijalizator>`.*tip* $\sim T$
     inače ako je `<izravni_deklarator>`.*tip* *niz*($T$) ili *niz*(*const*($T$))
     `<inicijalizator>`.*br-elem* $\leq$ `<izravni_deklarator>`.*br-elem*
     za svaki $U$ iz `<inicijalizator>`.*tipovi* vrijedi $U \sim T$
     inače *greška*

  Točka 3 provjerava semantičku ispravnost inicijalizacije. Za brojevne tipove je dovoljno da se tip inicijalizatora može implicitno pretvoriti u odgovarajući tip **bez** `const`-kvalifikatora (inače ne bi bilo moguće inicijalizirati varijable `const`-kvalificiranog tipa vrijednostima koje nisu `const`-kvalificiranog tipa; npr. `const int x = 3;` bi bila pogreška).

  Nadalje, za nizove treba provjeriti je li broj elemenata u inicijalizatoru manji ili jednak broju elemenata niza i mogu li se elementi inicijalizatora implicitno pretvoriti u odgovarajući tip, isto kao u slučaju brojevnih tipova.

  Konačno, u svim ostalim slučajevima (a to je jedino deklaracija funkcije), program ima semantičku pogrešku.

`<izravni_deklarator>`

Nezavršni znak `<izravni_deklarator>` generira deklarator varijable ili funkcije. Znak ima nasljedno svojstvo *ntip* i izvedeno svojstvo *tip* u koje se pohranjuje potpuni tip varijable ili funkcije. Ako se deklarira niz, znak dodatno ima i izvedeno svojstvo *br-elem* koje označava broj elemenata niza.

- `<izravni_deklarator> ::= IDN`
  *tip* ← *ntip*

  1. *ntip* ≠ `void`
  2. `IDN.`*ime* nije deklarirano u lokalnom djelokrugu
  3. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom

  Ova produkcija služi za generiranje varijabli cjelobrojnog tipa. Važno je uočiti da je varijabla deklarirana odmah nakon navedenog identifikatora, a prije opcionalnog inicijalizatora. To znači da je inicijalizacija `int x = x + 1;` semantički ispravna, ali rezultat nije definiran jer `x` na desnoj strani ima neodređenu vrijednost.

- `<izravni_deklarator> ::= IDN L_UGL_ZAGRADA BROJ D_UGL_ZAGRADA`
  *tip* ← *niz*(*ntip*)
  *br-elem* ← `BROJ.`*vrijednost*

  1. *ntip* ≠ `void`
  2. `IDN.`*ime* nije deklarirano u lokalnom djelokrugu
  3. `BROJ.`*vrijednost* je pozitivan broj ($> 0$) ne veći od `1024`
  4. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom

  Ova produkcija služi za deklariranje nizova. Obavezno mora biti naveden broj elemenata niza (to je sintaksno osigurano ovom produkcijom) i taj broj mora biti pozitivan i maksimalnog iznosa `1024`.

  Sintaksno nije moguće broj elemenata zadati nekakvim izrazom, čak ni ako se on u cijelosti sastoji od konstanti.

- `<izravni_deklarator> ::= IDN L_ZAGRADA KR_VOID D_ZAGRADA`
  *tip* ← *funkcija*(`void` → *ntip*)

  1. ako je `IDN.`*ime* deklarirano u lokalnom djelokrugu, tip prethodne deklaracije je jednak *funkcija*(`void` → *ntip*)
  2. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom ako ista funkcija već nije deklarirana u lokalnom djelokrugu

- `<izravni_deklarator> ::= IDN L_ZAGRADA <lista_parametara> D_ZAGRADA`
  *tip* ← *funkcija*(`<lista_parametara>.`*tipovi* → *ntip*)

  1. *provjeri*(`<lista_parametara>`)
  2. ako je `IDN.`*ime* deklarirano u lokalnom djelokrugu, tip prethodne deklaracije je jednak *funkcija*(`<lista_parametara>.`*tipovi* → *ntip*)
  3. zabilježi deklaraciju `IDN.`*ime* s odgovarajućim tipom ako ista funkcija već nije deklarirana u lokalnom djelokrugu

  Ova i prethodna produkcija generiraju deklaracije funkcija. Za razliku od varijabli, funkcije se mogu deklarirati u istom djelokrugu proizvoljan broj puta (zato jer je deklaracija varijable ujedno i njena definicija, što kod funkcija nije slučaj).

`<inicijalizator>`

Nezavršni znak `<inicijalizator>` generira izraz ili složeni inicijalizator za nizove. Za inicijalizaciju varijabli brojevnog tipa, `<inicijalizator>` će imati izvedeno svojstvo *tip* koje će sadržavati tip izraza s kojim se varijabla pokušava inicijalizirati. Za inicijalizaciju nizova, `<inicijalizato>` će imati izvedena svojstva *br-elem* i *tipovi*. Svojstvo *br-elem* sadržavat će broj elemenata inicijalizatora, a svojstvo *tipovi* će biti lista tipova izraza u složenom inicijalizatoru.

- `<inicijalizator> ::= <izraz_pridruzivanja>`
  ako je `<izraz_pridruzivanja>` $\overset{*}{\Rightarrow}$ `NIZ_ZNAKOVA`
  *br-elem* ← duljina niza znakova `+ 1`
  *tipovi* ← lista duljine *br-elem*, svi elementi su `char`
  inače
  *tip* ← `<izraz_pridruzivanja>.` *tip*

  1. *provjeri*(`<izraz_pridruzivanja>`)

  Kao podsjetnik iz UTR-a, simbol $\overset{*}{\Rightarrow}$ označava da se iz lijeve strane primjenom proizvoljnog broja produkcija generira desna strana. U ovoj gramatici, navedena relacija je zadovoljena samo ako se iz znaka `<izraz_pridruzivanja>` primjenom jediničnih produkcija konačno generira `NIZ_ZNAKOVA`, tj. slijed generiranja je `<izraz_pridruzivanja>` ⇒ `<log_ili_izraz>` ⇒ ... ⇒ `<primarni_izraz>` ⇒ `NIZ_ZNAKOVA`. Taj poseban slučaj u pravilima računanja svojstava u ovoj produkciji služi za poistovjećivanje inicijalizacije znakovnih nizova konstantnim nizom znakova (uniformni znak `NIZ_ZNAKOVA`) sa istovjetnim složenim inicijalizatorom.

```
1 char a[10] = "abc"; // isto kao redak 2
2 const char b[10] = { 'a', 'b', 'c', '\0' };
3 char c[10] = a; // greska
```

Ispis 4.23: Dva istovjetna načina inicijalizacije niza znakova i greška.

Nadalje, važno je uočiti da se nizovi ne mogu inicijalizirati drugim nizom koji nije konstanta (potpuno neovisno o tome jesu li elementi niza `const`-kvalificirani ili ne). Dodatno, ova pravila omogućuju i (neobičnu) inicijalizaciju niza `int` elemenata konstantnim nizom znakova (što nije dozvoljeno u *C*-u), kao što je prikazano u ispisu 4.24.

```
1 int a[10] = "abc"; // isto kao redak 2
2 int b[10] = { 'a', 'b', 'c', '\0' };
3 int c[10] = { 97, 98, 99, 0 };
4 int d[10] = a; // greska
```

Ispis 4.24: Tri istovjetna načina inicijalizacije niza brojeva i greška.

Drugi slučaj u pravilima računanja ove produkcije odnosi se na sve ostale tipove vrijednosti (uključujući i nizove znakove koji nisu konstante, tj. nisu označeni uniformnim znakom `NIZ_ZNAKOVA`). Samim tim, svojstva *br-elem* i *tipovi* tada neće

postojati pa će po pravilima za `<init_deklarator>` doći do semantičke greške u točki 3 druge produkcije.

- `<inicijalizator> ::= L_VIT_ZAGRADA <lista_izraza_pridruzivanja> D_VIT_ZAGRADA`
  *br-elem* ← `<lista_izraza_pridruzivanja>`.*br-elem*
  *tipovi* ← `<lista_izraza_pridruzivanja>`.*tipovi*

  1. *provjeri*(`<lista_izraza_pridruzivanja>`)

`<lista_izraza_pridruzivanja>`

Nezavršni znak `<lista_izraza_pridruzivanja>` generira jedan ili više izraza odvojenih zarezima. U produkcijama se računaju izvedena svojstva *br-elem* i *tipovi*, s istim značenjem kao i do sada.

- `<lista_izraza_pridruzivanja> ::= <izraz_pridruzivanja>`
  *tipovi* ← `[ <izraz_pridruzivanja>`.*tip* `]`
  *br-elem* ← `1`

  1. *provjeri*(`<izraz_pridruzivanja>`)

  `<lista_izraza_pridruzivanja> ::= <lista_izraza_pridruzivanja> ZAREZ <izraz_pridruzivanja>`
  *tipovi* ← `<lista_izraza_pridruzivanja>`.*tipovi* `+ [ <izraz_pridruzivanja>`.*tip* `]`
  *br-elem* ← `<lista_izraza_pridruzivanja>`.*br-elem*`+ 1`

  1. *provjeri*(`<lista_izraza_pridruzivanja>`)
  2. *provjeri*(`<izraz_pridruzivanja>`)

### 4.4.7 Provjere nakon obilaska stabla

Konačno, nakon obilaska stabla i provjere svih navedenih semantičkih pravila, semantički analizator treba provjeriti još dva pravila[15]

1. u programu postoji funkcija imena `main` i tipa *funkcija*(`void` → `int`)

   Ako ovo pravilo nije zadovoljeno, semantički analizator treba na standardni izlaz ispisati samo `main` u vlastiti redak i završiti s radom.

2. svaka funkcija koja je deklarirana bilo gdje u programu (u bilo kojem djelokrugu) mora biti definirana

   Ako ovo pravilo nije zadovoljeno, semantički analizator treba na standardni izlaz ispisati samo `funkcija` u vlastiti redak i završiti s radom.

[15] Ova pravila tipično ne bi bila u domeni semantičke analize, ali kako jezik *ppjC* ne podržava više od jedne datoteke s izvornim programom i samim time ne postoji potreba za povezivačem (engl. *linker*), semantički analizator treba provjeriti i ova pravila.

## 4.5 Savjeti za implementaciju

U ovom poglavlju navedene su neke napomene koje mogu pomoći u ostvarenju ove laboratorijske vježbe. Poglavlje će biti prošireno ako se za tim pokaže potreba u skladu sa čestim pitanjima.

### 4.5.1 Djelokrug deklaracija i tablica znakova

U semantičkoj analizi jezika *ppjC*, naglasak je na provjeri tipova podataka i djelokruga deklaracija. Obje provjere se ostvaruju korištenjem tablice znakova. Na primjer, pri deklaraciji varijable, tip varijable se zapisuje u tablicu znakova, a kada se varijabla koristi u istom ili ugniježđenom djelokrugu, tip varijable se određuje na osnovi zapisa u tablici znakova.

Kako je djelokrug deklaracija u jeziku *ppjC* određen hijerarhijom blokova tj. složenim naredbama (što je vrlo uobičajeno), prirodno je tablicu znakova također organizirati hijerarhijski, kao raspodijeljenu strukturu nad generativnim stablom. Pojedini dijelovi takve tablice znakova također čine stablastu strukturu koja u potpunosti odgovara blokovskoj strukturi programa koja je prikazana u generativnom stablu.

Svakom djelokrugu deklaracija u programu, dakle globalnom djelokrugu i svim blokovima (što uključuje i tijela funkcija) pridružen je jedan čvor stablaste tablice znakova. Svaki čvor logički ima dva dijela: tablicu loklanih imena i kazaljku na čvor koji predstavlja ugniježđujući blok (ako takav postoji). Korijen stablaste tablice znakova očito predstavlja globalni djelokrug deklaracija i jedini je čvor koji nema kazaljku na ugniježđujući blok.

Koristeći takvu strukturu tablice znakova, provjera mnogih semantičkih pravila jezika *ppjC* može se pojednostaviti u odnosu na korištenje jednorazinske tablice znakova. Na primjer, u jednostavnoj tablici znakova osim tipa varijable (i eventualno još nekih značajki) nužno je na neki način zapisati u kojem djelokrugu je varijabla deklarirana. Nadalje, ako u dva ili više djelokruga postoji varijabla istog imena (što je semantički ispravno), u takvoj jednostavnoj tablici znakova trebalo bi biti više zapisa za isto ime.

S druge strane, semantička pravila jezika garantiraju da su imena u svakom čvoru stablaste tablice znakova jedinstvena. Pretraživanje tablice svodi se na traženje imena u tablici lokalnih imena, a u slučaju da se ime tamo ne nalazi, pretražuje se čvor roditelj i tako sve do globalnog djelokruga.

Konačno, čvor stablaste tablice znakova može se pri provjeri semantičkih pravila tretirati kao nasljedno svojstvo većine znakova gramatike. Prije obrade bloka (tj. složene naredbe), gradi se novi čvor koji se koristi pri obradi tog bloka naredbi. Kazaljka na ugniježđujući blok novog čvora usmjeri se na nasljedno svojstvo složene naredbe.

# 5. Četvrta laboratorijska vježba

Tema četvrte laboratorijske vježbe je generiranje koda. Vježba se u velikoj mjeri naslanja na treću laboratorijsku vježbu, ali i djelomično riješena treća vježba može poslužiti kako bi se djelomično riješila četvrta vježba.

U praksi je generiranje *kvalitetnog* koda (učinkovitog po raznim parametrima) vrlo složen problem koji se i sam tipično obavlja u više faza, kao što je i objašnjeno u udžbeniku i na predavanjima. U ovoj laboratorijskoj vježbi je generiranje koda maksimalno pojednostavljeno, pa se tako ne ocjenjuje kvaliteta generiranog koda nego isključivo njegova točnost.

Jedan od problema u prijašnjim inačicama ovih laboratorijskih vježbi bila je činjenica da većina studenata ili ne zna ili slabo poznaje mnemonički jezik nekog stvarnog procesora. Kako bi se izbjegao taj problem, u ovoj vježbi će se generirati mnemonički jezik procesora FRISC koji se uči na predmetu *Arhitektura računala* FER-2 programa. Simulator procesora FRISC u web-pregledniku dostupan je ovdje[1]. Isti simulator može se pokrenuti i iz naredbenog retka, koristeći Node.js platformu. Uz instaliran Node.js, bit će potrebno u jedan direktorij dohvatiti i datoteke friscasm.js, friscjs.js i main.js. Ako je FRISC mnemonički program zapisan u datoteci `a.frisc`, program se može simulirati naredbom `node main.js a.frisc`.

Alternativno, za ispitivanje ispravnosti generiranih programa može se koristiti i ATLAS kao na predmetu *Arhitektura računala*. Ako naiđete na neke razlike u funkcionalnosti između ATLAS-a i gore opisanog simulatora, javite se na e-mail listu predmeta *ppj@zemris.fer.hr*.

## 5.1 Ulaz i izlaz generatora koda

Ulaz u četvrtu laboratorijsku vježbu bit će isti kao ulaz u treću vježbu, tj. generativno stablo *ppjC* programa. Važno je uočiti da je ulaz "obično" generativno stablo i nad njim treba provesti semantičku analizu. Svi zadani programi bit će semantički ispravni, ali je semantičku analizu ipak potrebno provesti kako bi bilo moguće generirati kod. Iz tog razloga je predviđeni način rješavanja ove laboratorijske vježbe proširenje rješenja treće laboratorijske vježbe u kojoj se provodi semantička analiza.

Rješenje četvrte laboratorijske vježbe treba zapisati generirani mnemonički program za

[1] Preciznije rečeno, uneseni program pisan mnemoničkim jezikom procesora FRISC prevodi se mnemoničkim asemblerom u strojne naredbe procesora FRISC. Generirane strojne naredbe pune se u memoriju od adrese `0` (ako nije drugačije zadano asemblerskom naredbom `` `ORG ``) i simulira se rad procesora FRISC. U nastavku upute ćemo zbog jednostavnosti o cijelom ovom postupku govoriti kao o simulatoru.

procesor FRISC u datoteku `a.frisc` u istom direktoriju u kojem se nalazi sam generator koda. Kako zbog jednostavnosti jezik *ppjC* ne podržava unos i ispis podataka, kao rezultat izvođenja programa promatrat će se povratna vrijednost iz funkcije `main` koju generirani mnemonički program prije izvođenja instrukcije `HALT` mora zapisati u registar `R6`.

**Važna napomena:** Iz prethodno navedenog razloga, moguće je "optimizirati" cijeli ulazni program u dvije instrukcije—jednu koja puni registar `R6` rezultatom izvođenja i jednu `HALT` instrukcij—tako da generator jednostavno odsimulira ulazni program. Iako bi se moglo reći da je to najbolji mogući generirani kod za zadani program, to očito nije smisao ove laboratorijske vježbe i takva rješenja bit će bodovana s 0 bodova. S druge strane, dozvoljeno je (iako ne i nužno) tijekom prevođenja izračunati konstante izraze, tj. izraze koji sadrže samo konstante i eventualno `const`-kvalificirane varijable. Naprednije postupke kojima bi se moglo izračunati i izraze koji sadrže varijable koje nisu `const`-kvalificirane **nemojte** provoditi.

Prethodno navedeni simulator za naredbeni redak na standardni izlaz ispisuje vrijednost u registru `R6` nakon izvođenja instrukcije `HALT`, a isti simulator koristit će se pri automatskoj evaluaciji rješenja. Drugim riječima, u ovoj vježbi će se automatski provjeravati samo rezultat izvođenja FRISC programa kojeg generira rješenje ove laboratorijske vježbe, a ne i sam generirani FRISC program.

Konačno, u jednoj rečenici bi se moglo reći da je zadatak četvrte laboratorijske vježbe napisati program koji će, na osnovi zadanog generativnog stabla na standardnom ulazu, u datoteku `a.frisc` generirati program u mnemoničkom jeziku procesora FRISC koji povratnu vrijednost funkcije `main` pohranjuje u registar `R6`. Sve ostalo odradit će sustav SPRUT.

## 5.2 Značajke simulatora

Simulator koji će se koristiti pri evaluaciji rješenja ove laboratorijske vježbe bi po svim funkcionalnim značajkama trebao biti jednak onom koji se koristi na predmetu *Arhitektura računala*. Drugim riječima, simulator podržava sve mnemoničke naredbe procesora FRISC i sve *pseudonaredbe* kao što su `` `ORG `` i `` `DW ``.

Na procesor je spojena memorija od `256KB` na adresama `0`–`3FFFF`. Iz tog razloga je preporučljivo pokazivač stoga (registar `R7`) na početku programa inicijalizirati na adresu $\texttt{40000}_{16} = \texttt{262144}_{10}$ (to je prva memorijska adresa koja ne postoji).

## 5.3 Karakteristike ispitnih primjera za četvrtu laboratorijsku vježbu

S ciljem omogućavanja djelomičnog bodovanja rješenja ove vježbe, veliki dio ispitnih primjera bit će vrlo kratki i fokusirani na neku specifičnu značajku jezika. Na taj način će i s rješenjem koje ne podržava dio jezika (na primjer, neke operatore, rekurzivno pozivanje funkcija i slično) biti moguće ostvariti dio bodova. Ipak, važno je imati na umu da svaki *ppjC* program ima barem jednu funkciju (funkciju `main`) i provjerava se povratna vrijednost funkcije pa je za bilo kakve bodove nužno generirati kod barem za jednostavne

funkcije bez parametara i lokalnih varijabli koje vraćaju vrijednost.

## 5.4 Predaja četvrte laboratorijske vježbe

Ulazna točka za Javu treba biti u razredu `GeneratorKoda`, a ulazna točka za Python treba biti u datoteci `GeneratorKoda.py`. Ostala pravila za predaju četvrte laboratorijske vježbe jednaka su kao za treću vježbu.

## 5.5 Proširenje nekih semantičkih pravila

U ovom poglavlju su ukratko navedena još neka semantička pravila koja se odnose na generiranje koda. Drugim riječima, u semanatičkoj analizi ne treba ništa novo provjeravati, nego pravila definiraju kakav kod treba generirati.

### 5.5.1 Semantička pravila inkrement i dekrement operatora

Isto kao *C*, jezik *ppjC* ima postfiks i prefiks inačice inkrement i dekrement operatora. Precizan opis značenja tih operatora u *C*-u, točnije trenutak u kojem se događa promjena vrijednosti varijable, relativno je složen i sličan opis ovdje ne bi imao smisla jer se radi o detalju koji nije previše bitan.

Zato ćemo za jezik *ppjC* definirati da se vrijednost varijable mijenja neposredno prije čitanja vrijednosti za prefiks inkrement i dekrement operatore i neposredno poslije čitanja vrijednosti za postfiks inkrement i dekrement. Na primjer, ako se izrazi izračunavaju koristeći stog[2], za izraz `i++` (što može biti dio nekog većeg izraza) treba generirati kod koji dohvaća vrijednost varijable `i` u registar, pohranjuje tu vrijednost na stog, povećava vrijednost u registru i zapisuje vrijednost u memoriju na adresu koja odgovara varijabli `i`.

Dodatno, definirat ćemo da sva korištenja ovih operatora koja imaju nedefinirano značenje u jeziku *C* imaju nedefinirano značenje i u jeziku *ppjC*, tj. neće se pojavljivati u ispitnim primjerima[3]. Ovo drugim riječima znači da zbog ovih operatora ne treba posebno brinuti o tome kojim redoslijedom se izračunavaju lijeva i desna strana binarnih operatora, vrijednosti argumenata funkcija i slično.

### 5.5.2 Semantička pravila operatora %

Zbog jednostavnosti, operator `%` ćemo definirati samo za pozitivne operande i za slučaj kada je lijevi operand `0`, a desni operand pozitivan broj. Rezultat je ostatak pri dijeljenju lijevog operanda s desnim.

[2] Vidi poglavlje 5.6.3.

[3] Primjeri takvih naredbi su: `a[i] = i++; x = --i + i; i = i--; x = f(i, i++);` i slično.

### 5.5.3 Evaluacija operanada logičkih operatora

Logički operatori u jeziku *ppjC*, isto kao u *C*-u, **ne evaluiraju** desni operand ako se rezultat može jednoznačno odrediti na osnovi vrijednosti lijevog operanda[4]. Konkretno, za operator `&&`, ako lijevi operand nije istinit, rezultat primjene operatora će sigurno biti neistina, bez obzira na vrijednost desnog operanda. U tom slučaju, desni operand se ne evaluira.

Ispravnost mnogih programa u jezicima koji definiraju ovakvo značenje logičkih operatora ovisi upravo o tome da se desni operand konjunkcije ne evaluira kada je lijevi operand neistinit. Pretpostavimo da je definiran neki niz znakova `a` duljine `n`. Ovaj način evaluacije operanada bio bi važan, na primjer, za izraz `i<n && a[i]='x'`. Ako je indeks `i` prevelik za indeksiranje u niz `a`, tj. lijevi operand operatora `&&` je lažan, važno je da se desni operand ne evaulira[5].

Slično, za operator `||`, desni operand se ne evaluira ako je lijevi operand istinit jer je tada rezultat sigurno istina, bez obzira na vrijednost desnog operanda.

## 5.6 Savjeti za implementaciju

U ovom poglavlju su navedeni neki savjeti za implementaciju generatora koda. Preporučamo da savjete pročitate, a po želji možete neke ili sve u potpunosti zanemariti. Svi primjeri FRISC koda u ovom poglavlju su ilustrativni, tj. ni u kom slučaju nije nužno da generator generira *točno takav* kod. Štoviše, većina generatora generirat će za ove primjere kod sa više instrukcija i to nije nikakav problem ako je funkcionalnost izvornog *ppjC* programa očuvana.

### 5.6.1 Organizacija generiranog programa

Prije početka rada na generiranju koda, treba razmisliti kako će generirani program biti organiziran, tj. gdje će se nalaziti potprogrami, gdje će biti globalne varijable, na kojoj adresi počinje stog i slično. U nastavku poglavlja je načelno i na kratkim primjerima opisana jedna moguća organizacija generiranog programa.

Kako izvođenje programa počinje od adrese `0`, na početku generiranog programa moraju biti instrukcije (a ne podaci). Svaki program mora inicijalizirati stog, pozvati proceduru koja je generirana na osnovi funkcije `main`, povratnu vrijednost pohraniti u registar `R6` i izvesti instrukciju `HALT`. Kao što je spomenuto u poglavlju 5.2, preporučljivo je pokazivač stoga inicijalizirati na adresu $\texttt{40000}_{16}$ jer stog procesora FRISC raste prema nižim adresama. Nadalje, pretpostavit ćemo da svi potprogrami vraćaju vrijednost putem registra `R6`[6]. Uz ove dvije odluke, početak i kraj izvođenja programa ostvaruju se sa tri instrukcije na adresama `0`, `4` i `8`, prikazane u ispisu 5.1.

[4] Ovakav način evaluacije operanada se u literaturi na engleskom jeziku obično naziva *short-circuit evaluation*.

[5] U primjeru je namjerno kao desni operand korišten izraz pridruživanja kako bi bila jasna razlika između rezultata izvođenja u slučaju da se taj desni operand evaluira i u slučaju da se ne evaluira, iako je jednako neispravno samo i pročitati vrijednost izvan granica niza.

[6] Vidi poglavlje 5.6.2.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT
```

Ispis 5.1: Instrukcije za inicijalizaciju i završetak izvođenja programa.

Labela `F_MAIN` u ispisu predstavlja adresu potprograma koji je generiran iz funkcije `main` ulaznog programa u jeziku *ppjC*. Na primjer, za jednostavan ulazni program u ispisu 5.2 mogli bismo generirati program sličan onom u ispisu 5.3.

```
int main(void) {
  return 42;
}
```

Ispis 5.2: Jednostavan *ppjC* program.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  MOVE %D 42, R6
        RET
```

Ispis 5.3: Mogući generirani FRISC mnemonički program za ispis 5.2.

Nakon generiranih instrukcija za inicijalizaciju, mogu se generirati potprogrami za svaku funkciju u programu, slično kao što je prikazano u prethodnom primjeru za funkciju `main`. Svakom potprogramu se pridijeli jedinstvena labela koja omogućuje pozivanje tog potprograma instrukcijom `CALL`.

Nakon koda potprograma, mogu se rezervirati memorijske lokacije za globalne varijable. Na primjer, za program u ispisu 5.4 mogao bi se generirati program sličan onom u ispisu 5.5.

```
int x = 12;
int main(void) {
  return x;
}
```

Ispis 5.4: Jednostavan *ppjC* program s globalnom varijablom.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  LOAD R6, (G_X)
        RET

G_X     DW %D 12
```

Ispis 5.5: Mogući generirani FRISC mnemonički program za ispis 5.4.

Konačno, prostor nakon globalnih varijabli može biti koristan za neke konstante u programu, a s kraja memorije ga puni i rastući stog.

### 5.6.2 Preslikavanje funkcije jezika *ppjC* u FRISC potprogram

Kako svaki *ppjC* program sadrži barem jednu funkciju, važno je razmisliti o različitim pitanjima koja se pojavljuju pri preslikavanju funkcija u potprograme u mnemoničkom jeziku.

#### Prenošenje argumenata i povratne vrijednosti

U ostvarenju preslikavanja funkcija u potprograme, ključno je odlučiti na koji način će potprogram komunicirati sa svojim pozivateljem, tj. na koji način pozivatelj potprogramu prenosi argumente i na koji način potprogram vraća povratnu vrijednost pozivatelju. I za prenošenje argumenata i za povratnu vrijednost postoje barem tri mogućnosti: korištenje registara, korištenje stoga i korištenje poznatih memorijskih lokacija tj. adresa. Štoviše, različiti potprogrami mogu na različite načine komunicirati s pozivateljem.

Zbog jednostavnosti, preporuča se za sve potprograme koristiti stog za argumente i registar `R6` za povratnu vrijednost. Uz takvu organizaciju, kako bi pozvao potprogram, pozivatelj na stog mora postaviti argumente i izvesti instrukciju `CALL` s odgovarajućom labelom, a povratna vrijednost se nakon povratka iz potprograma nalazi u registru `R6`. Potprogram pristupa predanim argumentima koristeći pokazivač stoga (registar `R7`) i registarsko indirektno adresiranje s odmakom. Naime, kako generator koda generira sve ove instrukcije, lako je *tijekom prevođenja* odrediti potrebne odmake za sve parametre.

Preostaje odlučiti što učiniti s argumentima na stogu nakon završetka izvođenja potprograma. Jedna mogućnost je da pozivatelj nakon instrukcije `CALL` uveća pokazivač stoga za odgovarajuću vrijednost i na taj način "obriše" argumente sa stoga. Ova mogućnost prikazana je u ispisu 5.7.

```
1 int f(int y) {
2   return y;
3 }
4 int x = 12;
5 int main(void) {
6   return f(x);
7 }
```

Ispis 5.6: Jednostavan *ppjC* program s pozivom funkcije.

```
1         MOVE 40000, R7
2         CALL F_MAIN
3         HALT
```

```
4
5  F_MAIN LOAD R0, (G_X)
6         PUSH R0
7         CALL F_F
8         ADD R7, 4, R7
9         RET
10
11 F_F    LOAD R6, (R7+4)
12        RET
13
14 G_X    DW %D 12
```

Ispis 5.7: Mogući generirani FRISC mnemonički program za ispis 5.6.

Naredba koja pomiče pokazivač stoga nalazi se u retku 8. U ovom slučaju, pokazivač stoga se pomiče za `4` jer je postojao samo jedan četverobajtni argument. Uočite da bi se bez ove naredbe vrijednost argumenta (u ovom slučaju `12`) interpretirala kao povratna adresa potprograma `F_MAIN` u instrukciji `RET`.

Druga mogućnost je da sam potprogram prije završetka rada ukloni argumente sa stoga *pazeći na svoju povratnu adresu*. I ovu mogućnost je relativno jednostavno ostvariti, ali nije ovdje prikazana. Preporuča se konzistentno korištenje jednog od ova dva načina obnavljanja stoga za sve potprograme.

### Lokalne varijable

Lokalne varijable se slično kao i parametri mogu pohranjivati na stog, u registre ili na određene memorijske lokacije. Kako bi se omogućilo rekurzivno pozivanje potprograma, najjednostavnije je lokalne varijable pohraniti na stog, na početku potprograma. Lokalne varijable se onda u potprogramu koriste isto kao i parametri, registarskim indirektnim adresiranjem s odmakom.

Eventualne inicijalizacije lokalnih varijabli mogu se obaviti i nakon što su za sve parametre osigurana mjesta na stogu kako bi odmaci od pokazivača stoga bili konstantni. Mjesto se na stogu može osigurati jednostavnim smanjenjem pokazivača stoga, pri čemu je po pravilima jezika dozvoljeno da neinicijalizirane lokalne varijable imaju neodređene vrijednosti (vrijednosti će ovisiti o sadržaju stoga u nekom prethodnom izvođenju nekog potprograma). Prije izvođenja instrukcije `RET`, potprogram mora vratiti pokazivač stoga na originalno mjesto, tj. ukloniti lokalne varijable sa stoga.

```
1 int f(int y) {
2   int x = y;
3   int z;
4   return x;
5 }
6 int x = 12;
7 int main(void) {
8   return f(x);
9 }
```

Ispis 5.8: Funkcija s lokalnom varijablom.

```
        MOVE 40000, R7
        CALL F_MAIN
        HALT

F_MAIN  LOAD R0, (G_X)
        PUSH R0
        CALL F_F
        ADD R7, 4, R7
        RET

F_F     SUB R7, 8, R7 ; mjesto za x i z
        LOAD R0, (R7+0C)
        STORE R0, (R7+4) ; inicijalizacija lokalne varijable x
        LOAD R6, (R7+4) ; povratna vrijednost
        ADD R7, 8, R7 ; skidanje lokalnih varijabli sa stoga
        RET

G_X     DW %D 12
```

Ispis 5.9: Mogući generirani FRISC mnemonički program za ispis 5.8.

#### Čuvanje konteksta

Ako se u generiranju koda koristi neki algoritam za dodjelu registara (što se u ovoj vježbi **ne preporuča**; vidi poglavlje 5.6.3), onda može biti važno da potprogrami tijekom svog rada ne promijene sadržaj određenih registara. Kontekst se tipično čuva tako da se na početku potprograma vrijednosti nekih ili svih registara zapišu na stog te se prije završetka rada potprograma te iste vrijednosti pohrane nazad u registre.

Dodatno, treba razmisliti o tome hoće li potprogram čuvati i vrijednost statusnog registra ili ne. Čuvanje vrijednosti statusnog registra može pozitivno djelovati na neke postupke optimiranja, ali za potrebe ove vježbe se ne preporuča.

Nedostatak čuvanja konkteksta je potrošnja procesorskog vremena, ali takva organizacija potprograma bitno olakšava neke postupke dodjele registara i općenito omogućuje učinkovitije korištenje registara jer pozivi potprograma ne utječu na vrijednosti registara (osim eventualno na vrijednost registra koji se koristi za prijenos povratne vrijednosti).

### 5.6.3 Jednostavno izračunavanje izraza korištenjem stoga

Kako je generiranje koda koji učinkovito koristi registare čak i u jednostavnim inačicama relativno složen problem, za potrebe ove laboratorijske vježbe se preporuča izračunavanje izraza korištenjem stoga. Pritom se registri koriste samo privremeno, kako bi se dohvatile

odgovarajuće vrijednosti sa stoga i kako bi se obavila odgovarajuća operacija, nakon čega se izračunata vrijednost opet pohranjuje na stog.

Pokazivač stoga će se tijekom ovog postupka mijenjati što otežava pristupanje parametrima i lokalnim varijablama potprograma. Ispravni odmaci se svakako daju izračunati, a alternativa je u svakom potprogramu rezervirati jedan registar (na primjer `R6` u koji se na kraju potprograma pohranjuje povratna vrijednost) za adresu *okvira funkcije* koja se može inicijalizirati, na primjer, nakon zauzimanja prostora za lokalne varijable i nakon toga se ne mijenja do kraja izvođenja potprograma. U tom slučaju, vrijednost tog registra treba pohraniti na stog prije svake `CALL` instrukcije u potprogramu i dohvatiti ju sa stoga nakon te instrukcije.

Na primjer, za računanje vrijednosti izraza `x + y` gdje su `x` i `y` globalne varijable, mogao bi se generirati FRISC kod prikazan u ispisu 5.10.

```
1        LOAD R0, (G_X)
2        PUSH R0
3
4        LOAD R0, (G_Y)
5        PUSH R0
6
7        POP R1
8        POP R0
9        ADD R0, R1, R0
10       PUSH R0
```

Ispis 5.10: Računanje izraza `x + y` primjenom stoga.

Pri obradi primarnih izraza `x` i `y` generirale bi se po dvije instrukcije u retcima 1–2 i 4–5, a pri obradi operatora zbrajanja (u produkciji znaka `<aditivni_izraz>`) generirale bi se četiri instrukcije u retcima 7–10.

### 5.6.4 Množenje, dijeljenje i operator ostatka

Kako FRISC nema instrukcije za obavljanje ovih operacija, njihovo provođenje mora se riješiti ručno što čini implementaciju ovih operatora znatno složenijom od implementacije ostalih operatora. Najjednostavniji način za ostvariti ove operatore je sa dva potprograma koji se mogu ugraditi u svaki program koji koristi operatore (ili čak i u programe koji ih ne koriste). Pritom ne treba paziti na rezultate koji izlaze iz opsega tipa `int` jer su takvi rezultati u pravilima jezika ostavljeni nedefiniranima.

Implementacija ovih operatora nije previše važna (pogotovo s obzirom na relativnu složenost u odnosu na implementaciju ostalih operatora) i preporuča se implementirati ove operatore pri kraju rada na generatoru koda, ako za to bude vremena.

### 5.6.5 Kako početi rješavati laboratorijsku vježbu

Prošlih godina je jedno od čestih pitanja bilo upravo kako započeti rad na generatoru koda. Dobra strategija je početi od najjednostavnijeg primjera *ppjC* programa i za njega napisati

odgovarajući FRISC kod te ga isprobati na simulatoru. Iz jednostavnih primjera moguće je osmisliti općenite pristupe za generiranje različitih dijelova programa i to inkrementalno implementirati u generator.

Pri pisanju odgovarajućeg FRISC koda, bitno je pokušati "imitirati" trenutni mentalni model generatora koda, što smo pokušali ilustrirati u ovim savjetima za implementaciju. Kada se taj model promijeni, treba revidirati prethodno obrađene primjere i provjeriti je li sve u redu. Na primjer, ako bismo odlučili da će se izrazi izračunavati koristeći stog, čini se da bismo uniformnom primjenom tog modela za jednostavni program iz ispisa 5.2 u kojem funkcija `main` jednostavno vraća konstantu `42` mogli umjesto instrukcije `MOVE %D 42, R6` generirati tri instrukcije, kao što je prikazano u ispisu 5.11.

```
1         MOVE 40000, R7
2         CALL F_MAIN
3         HALT
4
5 F_MAIN  MOVE %D 42, R0
6         PUSH R0
7         POP R6
8         RET
```

Ispis 5.11: FRISC mnemonički program za ispis 5.2 uz konzistentno računanje izraza pomoću stoga.

Instrukcije u retcima 5 i 6 mogle bi se generirati pri obradi izraza `42`, a instrukcije u retcima 7 i 8 pri obradi `return` naredbe.

# Bibliografija

[1] Siniša Srbljić: *Prevođenje programskih jezika*, Element, 2007.

[2] Siniša Srbljić: *Uvod u teoriju računarstva*, Element, 2007.