

Upute za izradu laboratorijskih vježbi iz predmeta
Prevođenje programskih jezika

Autori:
Ivan Budiselić, Ivan Žužak

Datum posljednje izmjene:
30. listopada 2011.

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

Sadržaj

1	Uvod	4
1.1	Cilj laboratorijskih vježbi	4
1.2	Grupe za laboratorijske vježbe	4
1.3	Odabir jezika izgradnje jezičnog procesora	4
1.4	Provedba laboratorijskih vježbi	5
1.5	Kako pripremiti rješenje laboratorijskih vježbi za računalno ocjenjivanje . .	5
2	Prva laboratorijska vježba	7
2.1	Generator leksičkog analizatora	7
2.1.1	Regularne definicije	8
2.1.2	Stanja leksičkog analizatora	9
2.1.3	Imena leksičkih jedinki	9
2.1.4	Pravila leksičkog analizatora	10
2.1.5	Specijalni znakovi	11
2.1.6	Primjer	11
2.2	Leksički analizator	14
2.2.1	Ispis leksičkog analizatora	15
2.2.2	Strukture podataka leksičkog analizatora	15
2.2.3	Algoritam leksičkog analizatora	15
2.2.4	Razrješavanje nejednoznačnosti	15
2.2.5	Postupak oporavka od pogreške	16
2.2.6	Primjer	16
2.3	Način računalnog ocjenjivanja rješenja	17
2.4	Savjeti za implementaciju	19
2.4.1	Generiranje koda leksičkog analizatora	19

2.4.2	Priprema regularnih izraza za generiranje konačnog automata . . .	19
2.4.3	Izgradnja ε -NKA iz regularnih izraza za potrebe leksičkog analizatora	20
2.4.4	Ostvarenje tablice znakova	25
2.5	Leksička analiza podskupa jezika C	26
3	Druga laboratorijska vježba	27
3.1	Generator sintaksnog analizatora	27
3.1.1	Nezavršni znakovi gramatike	28
3.1.2	Završni znakovi gramatike	28
3.1.3	Sinkronizacijski završni znakovi	28
3.1.4	Produkcije gramatike	29
3.1.5	Zadaci generatora sintaksnog analizatora	30
3.1.6	Razrješavanje nejednoznačnosti	30
3.2	Sintaksni analizator	31
3.2.1	Ulaz u sintaksni analizator	31
3.2.2	Simulator LR parsera	31
3.2.3	Ispis generativnog stabla	32
3.2.4	Oporavak od pogreške	32
3.3	Primjeri	33
3.4	Savjeti za implementaciju	35
3.4.1	Računanje refleksivnog tranzitivnog okruženja relacije	35
3.5	Predaja druge laboratorijske vježbe	35
	Literatura	37

1. Uvod

Ovaj dokument namijenjen je studentima predmeta *Prevodenje 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. Grupnu 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

platformi¹. 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. To pak 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 izvode 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

¹Neće 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 i zadnji znak svakog ispisa treba biti znak novog retka. Drugim riječima, kada ispisujete očekivani izlaz, nakon sadržaja svakog retka treba staviti znak novog retka. Na primjer, pretpostavimo da program tijekom rada u niz (engl. *array*) upiše *n* cijelih brojeva tipa *int*, koje treba ispisati jednog po retku. Tada bi u jeziku *C* algoritam za ispis izgledao kao u isječku 1.1.

```
1 for (i=0; i<n; i++) {  
2     printf("%d\n", A[i]);  
3 }
```

Ispis 1.1: Primjer algoritma za ispis.

Smisao ovog primjera i prethodnog odlomka je da uočite da i nakon zadnjeg broja dolazi znak 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)
- Kleenove operator ponavljanja `*` (ponavljanje nula ili više puta)
- grupiranje koristeći obale 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 ε 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**`).

¹U 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 nalaziti 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 pročitao 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

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

treba koristiti. 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 jedinice 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 nalaziti će se znak { koji započinje dio za definiciju argumenata akcije. Svi retci argumenata akcije početi 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 jedinice (jedno od imena definiranih ranije u Ulaznoj Datoteci) ili znak – (minus). U slučaju da je u akciji navedeno ime leksičke jedinice, generirani leksički analizator treba zadanu leksičku jedinku dodati u odgovarajuće tablice i u tablice upisati početne vrijednosti za tu leksičku jedinku (ovo je detaljnije objašnjeno u odjeljku 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 što 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 jedinice, a ne znak —) i ima argument `NOVI_REDAK`, za broj retka te leksičke jedinice 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 `yyvalless(naZnak)` u Lexu. *naZnak* će biti 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 *minus-Lang*. Unarni minus može se pojaviti više puta uzastopce. Podržani su samo cijeli brojevi u dekadskom, oktalnom ili heksadekadskom zapisu. Grupiranje se ostvaruje korištenjem oblika 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**.

```

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 2.4: 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 brojevnice 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 analiza².

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.

```

1 <S_pocetno>#\|
2 {
3 -
4 UDJI_U_STANJE S_komentar
5 }
6 <S_komentar>\|#
7 {
8 -
9 UDJI_U_STANJE S_pocetno
10 }
11 <S_komentar>\n
12 {
13 -
14 NOVI_REDAK
15 }
16 <S_komentar>{sviZnakovi}
17 {
18 -
19 }

```

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 trećim pravilom. Konačno, u komentaru se može pojaviti mnogo znakova koji inače nisu dozvoljeni van komentara.

²O 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.

```

1 <S_pocetno>--{bjelina}*--
2 {
3   OP_MINUS
4   UDJI_U_STANJE S_unarni
5   VRATI_SE 1
6 }
7 <S_pocetno>\({bjelina}*--
8 {
9   LIJEVA_ZAGRADA
10  UDJI_U_STANJE S_unarni
11  VRATI_SE 1
12 }
13 <S_unarni>--
14 {
15   UMINUS
16   UDJI_U_STANJE S_pocetno
17 }
```

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 pročitanoj 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.

```

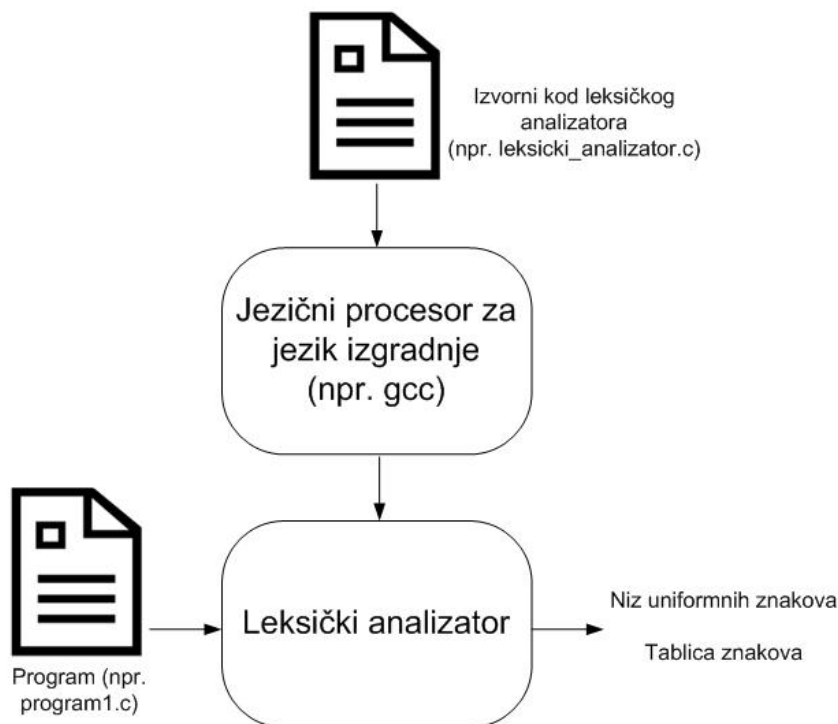
1 <S_unarni>--{bjelina}*--
2 {
3 UMINUS
4 VRATI_SE 1
5 }

```

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 jedinice. Način konstruiranja konačnog automata (ε -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 su *niz uniformnih znakova* i *tablica znakova*.

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 odjeljku 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 i tablicu znakova (engl. *symbol table*). Zbog jednostavnosti Ulazne Datoteke, tablica znakova može biti jedino homogena, odnosno jedna tablica (kako god bila implementirana) sadržavati će na kraju rada leksičkog analizatora sve leksičke jedinice koje se nalaze u ulaznom programu. Drugim riječima, Ulazna Datoteka ne omogućuje razvrstavanje simbola u tablicu ključnih riječi, tablicu konstanti i tablicu identifikatora.

Niz uniformnih znakova mora za svaku prepoznatu leksičku jedinku sadržavati ime leksičke jedinice, redak u kojem se ta jedinka nalazi u ulaznom programu i neku informaciju kako doći do odgovarajućeg zapisa u tablici znakova; to može biti pokazivač, indeks ili nešto treće.

Nakon završetka rada leksičkog analizatora, tablica znakova treba sadržavati točno jedan zapis za svaku leksičku jedinku u ulaznom programu.

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 ε -NKA. Implementacijom algoritma zasnovanog na ε -NKA neće biti potrebno ostvarivati pretvorbu ε -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).

2.2.5 Postupak oporavka od pogreške

Za postupak oporavka od pogreške treba koristiti jednostavan postupak odbacivanja prvog znaka iz ulaznog programa kao što je opisano u poglavlju 2.9.5 udžbenika. Prijavu pogrešaka obavljati 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 odjeljku 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 i tablicu znakova:

uniformni znak	redak	indeks
OPERAND	2	0
OP_MINUS	2	1
UMINUS	2	2
OPERAND	2	3
OP_MINUS	2	1
LIJEVA_ZAGRADA	2	4
OPERAND	4	0
OP_MINUS	4	1
UMINUS	4	2
UMINUS	5	2
UMINUS	5	2
OPERAND	5	5
DESNA_ZAGRADA	5	6

0	OPERAND	3
1	OP_MINUS	-
2	UMINUS	-
3	OPERAND	0x12
4	LIJEVA_ZAGRADA	(
5	OPERAND	076
6	DESNA_ZAGRADA)

U drugom stupcu niza uniformnih znakova naveden je redak u kojem se uniformni znak nalazi u ulaznoj datoteci. U trećem stupcu naveden je indeks u tablicu znakova.

U ovom primjeru važno je uočiti dvije stvari. Prvo, iako se operand 3 u tekstu programa pojavljuje na dva mjesta, u tablici postoji samo jedan (zajednički) zapis jer se radi o dva pojavljivanja iste leksičke jedinice. Drugo, iako se zapisi za binarni i unarni minus u tekstu programa ne razlikuju, u tablici znakova postoje zasebni zapisi za binarni i unarni minus zato jer se radi o različitim leksičkim jedinkama.

Očekivani ispis na standardni izlaz prikazan je **ovdje**. Treba uočiti da ispis završava praznim retkom tj. da svi retci (osim tog praznog) završavaju znakom kraja retka.

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.

```
1 analizator/bar.c
2 analizator/bar.h
3 analizator/lekser.h
4 analizator/predlozak.txt
5 foo.c
6 foo.h
7 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.

```
1 analizator/bar.c
2 analizator/bar.h
3 analizator/lekser.h
4 analizator/predlozak.txt
5 foo.c
6 foo.h
7 generator.c
8 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.

```
1 analizator/bar.c
2 analizator/bar.h
3 analizator/lekser.h
4 analizator/lekser.c
5 analizator/predlozak.txt
6 analizator/tablica.txt
7 foo.c
8 foo.h
9 generator.c
10 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.

```
1 analizator/analizator.exe
2 analizator/bar.c
3 analizator/bar.h
4 analizator/lekser.h
5 analizator/lekser.c
6 analizator/predlozak.txt
7 analizator/tablica.txt
8 foo.c
9 foo.h
10 generator.c
11 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 ju 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.

Za rješenja u *Javi* i *C#*-u, 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**.

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 nahodjenju mogu dio ili sve savjete u ovom poglavlju zanemariti.

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 ϵ -NKA iz regularnog izraza, potrebno 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 definiciju {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 de-

iniciju $\{\text{znamenka}\}$ tako da će primjenom opisanog algoritma regularni izraz za definiciju $\{\text{hexZnamenka}\}$ 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
}`

uvršćavanjem takvog regularnog izraza za definiciju $\{\text{hexZnamenka}\}$ 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 ε -NKA koristeći ranije spomenuti algoritam opisan u udžbeniku "Uvod u teoriju računarstva" [2, poglavlje 2.2.2].

2.4.3 Izgradnja ε -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 ε -NKA te se ulazni znakovi predaju samo automatu koji je zadužen za obradu trenutno aktivnog stanja leksičkog analizatora.

Postupak izgradnje ε -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 ε -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 ε -NKA.

Izgradnja ε -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 ε -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 `ново_stanje`. Funkcija `ново_stanje` vraća oznaku dodanog stanja. Pretpostavlja se da će početni broj stanja automata biti nula. Pseudokod funkcije `ново_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 zgradama 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
4   za (i=0; i<duljina(izraz); i=i+1)
5     ako je izraz[i]=='(' && je_operator(izraz, i)
6       br_zagrada = br_zagrada + 1
7     inace ako je izraz[i]==')' && je_operator(izraz, i)
8       br_zagrada = br_zagrada - 1
9     inace ako je br_zagrada==0 && izraz[i]=='|' && je_operator(izraz, i)
10      grupiraj lijevi negrupirani dio niza znakova izraz u niz izbori
11    kraj ako
12  kraj za
13  ako je pronadjen barem jedan operator izbora
14    grupiraj preostali negrupirani dio niza znakova izraz u niz izbori
15  ...
```

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.

```
1   ...
2   int lijevo_stanje = novo_stanje(automat)
3   int desno_stanje = novo_stanje(automat)
4   ako je pronadjen barem jedan operator izbora
5     za (i=0; i<br_elemenata(izbori); i=i+1)
6       ParStanja privremeno = pretvori(izbori[i], automat)
7       dodaj_epsilon_prijelaz(automat,
8         lijevo_stanje,
9         privremeno.lijevo_stanje)
10      dodaj_epsilon_prijelaz(automat,
11        privremeno.desno_stanje,
12        desno_stanje)
13    kraj za
14  inace
15  ...
```

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.

```

1      ... // nastavlja se inace iz proslog odsjecka
2      bool prefiksirano = laz
3      int trenutno_stanje = lijevo_stanje
4      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 odjeljku 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.

```

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

```

Ispis 2.21: Obrada ponavljanja Kleenovim 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.

```

1 // slucaj 2
2 ako je izraz[i] == '\\\' // jedan znak \, kao u C-u
3   prefiksirano = istina
4   nastavi za petlju // continue u C-u
5 kraj ako
6 ako je izraz[i] != '('
7   *slucaj 2a*
8 inace
9   *slucaj 2b*
10 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.

```

1 // slucaj 2a
2 int sljedece_stanje = novo_stanje(automat)
3 ako je izraz[i] == '$'
4   dodaj_epsilon_prijelaz(automat, trenutno_stanje, sljedece_stanje)
5 inace

```



```

6  dodaj_prijelaz(automat, trenutno_stanje, sljedece_stanje, izraz[i])
7  kraj ako
8  *provjeri ponavljanje*
9  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 zagrada. Pseudokod za ovaj slučaj prikazan je u ispisu 2.24.

```

1  // slucaj 2b
2  int j = *pronadji odgovarajucu zatvorenu zagradu*
3  ParStanja privremeno = pretvori(izraz[i+1..j-1], automat)
4  dodaj_epsilon_prijelaz(automat, trenutno_stanje,
5      privremeno.lijevo_stanje)
6  i = j
7  trenutno_stanje = privremeno.desno_stanje
8  ako je i+1<duljina(izraz) && izraz[i+1]=='*'
9      dodaj_epsilon_prijelaz(automat, privremeno.lijevo_stanje, privremeno.
10         desno_stanje)
10     dodaj_epsilon_prijelaz(automat, privremeno.desno_stanje, privremeno.
11         lijevo_stanje)
11     i = i+1
12 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](#).

2.4.4 Ostvarenje tablice znakova

Kroz treću laboratorijsku vježbu, zapisi u tablici znakova morat će se proširivati dodatnim podacima. Za prvu laboratorijsku vježbu to znači da je poželjno sve pristupe tablici znakova ostvariti kroz pomoćne funkcije ili metode tako da kasnije bude moguće potrebne promjene obaviti na što manjem broju mjesta. Dodatno, tablica znakova tijekom kasnijih faza rada jezičnog procesora može i rasti. Na primjer, leksička analiza stvara jedan zapis u tablici znakova za svaku pojavu nekog identifikatora "x" u izvornom programu. Međutim, ako se spomenuti identifikator pojavljuje u različitim kontekstima (na primjer, negdje se

radi o `int`, a negdje o `char` varijabli), bit će nužno u tablicu znakova dodati zapise koji opisuju te različite instance identifikatora jednakog imena.

Alternativno, moguće je u kasnijim fazama rada koristiti više tablica znakova, ali o tome će više riječi biti u trećoj laboratorijskoj vježbi.

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](#).

<code>break</code>	<code>else</code>	<code>return</code>
<code>char</code>	<code>for</code>	<code>void</code>
<code>const</code>	<code>if</code>	<code>while</code>
<code>continue</code>	<code>int</code>	

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](#).

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

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 pogreške³. Očekivani izlaz za dani primjer prikazan je [ovdje](#).

³Kao što je prije spomenuto, pripazite da leksički analizator ne ispisuje **ništa** na standardni izlaz osim traženog ispisa - greške ispisuje npr. na `stderr`.

3. Druga laboratorijska vježba

Tema druge laboratorijske vježbe je sintakсна analiza. Grupe izrađuju generator sintaksnog analizatora po uzoru na program Yacc, ali u bitno pojednostavljenom obliku. Generirani sintakсни 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 sintakсни analizator trebaju raditi i dani su neki savjeti za implementaciju. Ova uputa pretpostavlja da je čitatelj upoznat sa ispredavanim gradivom o sintakсноj 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 sintakсни 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 pojaviti 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 odjeljku [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:

```
1 <A>
2 <B> <C> c
3 <B>
4 $
5 b <C> <D> <E>
6 <A>
7 e <D> <B>
8 <C>
9 <D> a <B>
10 c a
11 <D>
12 $
13 d <D>
14 <E>
15 e <A> f
16 c
```

Ispis 3.1: Primjer produkcija gramatike.

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

```
1 <A> ::= <B> <C> c | e <D> <B>
2 <B> ::= "" | b <C> <D> <E>
3 <C> ::= <D> a <B> | c a
4 <D> ::= "" | d <D>
5 <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 ε -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 ε -NKA treba pretvoriti u istovjetni DKA. Postupak pretvorbe ε -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

razriješeno¹.

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 odjeljku 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 Datoteci koja je predana generatoru. Leksičke jedinice 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 jedinica 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 ϵ) u slučaju da list predstavlja desnu stranu ϵ -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 pripadni uniformni znak nalazio u ulaznoj datoteci i indeksom odgovarajućeg zapisa u tablici znakova. 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 koji odgovaraju uzorku za zamjenu povezuje kao djecu novog unutrašnjeg čvora. U skladu

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

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. Sljedeći 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 navedeni²:

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 radom³.

²Ovo se neće provjeravati u automatskoj evaluaciji putem sustava SPRUT, ali se može provjeravati na usmenom odgovaranju vježbe.

³Ovaj 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 nezavršni 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.

```
1 %V <A> <B>
2 %T a b
3 %Syn b
4 <A>
5 <B> <A>
6 <B>
7 a <B>
8 b
9 <A>
10 $
11
```

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 jedinice koje se sastoje od uzastopnih znakova **x** (opcionalno odvojenih prazninama), a uniformni znak **b** predstavlja leksičke jedinice koje se sastoje od uzastopnih znakova **y** (opcionalno odvojenih prazninama).

```
1 a 1 x x x
2 b 2 y y
3 a 3 xx xx
4 a 4 xx xx xx
5 b 4 y
6
```

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 odjeljku 3.2.3.

```
1 <A>
2 <B>
3 a 1 x x x
4 <B>
5 b 2 y y
```

```

6 <A>
7 <B>
8   a 3 xx xx
9 <B>
10  a 4 xx xx xx
11 <B>
12  b 4 y
13 <A>
14 $
15

```

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 ovdje⁴. U ispisu 3.6 prikazan je “C program” sa sintaksnom pogreškom.

```

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

```

Ispis 3.6: Jednostavan C program sa sintaksnom greškom.

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

```

1 KR_INT 1 int
2 IDN 1 x
3 OP_PRIDRUZI 1 =
4 BROJ 1 3
5 OP_BIN_I 1 &
6 TOCKAZAREZ 1 ;
7 KR_INT 2 int
8 IDN 2 y
9 OP_PRIDRUZI 2 =
10 BROJ 2 2
11 TOCKAZAREZ 2 ;
12

```

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 programa⁵.

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

⁴Za ovaj primjer generator generira ε -NKA s 3115 stanja i DKA sa 691 stanja uz jedno *Pomakni/Reduciraj* proturječje vezano uz **else** dio naredbe grananja.

⁵Ponovno, 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 pod-direktoriju `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 sintaksnu 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.

Bibliografija

- [1] Siniša Srbljić: *Prevodenje programskih jezika*, Element, 2007.
- [2] Siniša Srbljić: *Uvod u teoriju računarstva*, Element, 2007.