Contents

[1. Objasniti namenu programskih prevodioca? 2](#_Toc64675492)

[2. Sta je to assembler? 2](#_Toc64675493)

[3. Cemu sluze makroprocesori? 2](#_Toc64675494)

[4. Objasniti razliku izmedju kompilatora i interpretatora? 2](#_Toc64675495)

[5. Navesti osnovne osobine kompilatora? 2](#_Toc64675496)

[6. Objasniti sta je to medjukod? 2](#_Toc64675497)

[7. Nacrtati potpunu semu strukture kompilatora? 2](#_Toc64675498)

[8. Definisati pojam azbuke. 2](#_Toc64675499)

[9. Dati formalnu definiciju pojma reči. 2](#_Toc64675500)

[10. Navesti osnovne delove reči. 2](#_Toc64675501)

[11. Navesti osnovne operacije koje se mogu izvršavati nad rečima. 2](#_Toc64675502)

[12. Dati formalnu definiciju pojam jezika. 2](#_Toc64675503)

[13. Navesti osnovne operacije nad jezicima. 2](#_Toc64675504)

[14. Šta su to formalne gramatike i kako se definišu? 2](#_Toc64675505)

[15. Šta su to neredukovane gramatike. Dati primer. 2](#_Toc64675506)

[16. Sta su to nejednoznačne gramatike. Dati primer. 2](#_Toc64675507)

[17. Navesti osnovne tipove gramatika po Čomskom. 2](#_Toc64675508)

[18. Šta su to konteksne, a šta beskonteksne gramatike? 2](#_Toc64675509)

[19. Šta su to normalne forme gramatika? 2](#_Toc64675510)

[20. Šta su to automati. Objasniti kako teče proces prepoznavanja reči automatom. 2](#_Toc64675511)

[21. Kako je definisana Tjuringova mašina? 2](#_Toc64675512)

[22. Koje jezike prepoznaju automati tipa Tjuringove mašine? 2](#_Toc64675513)

[23. Šta su to Linearno ograničeni automati i koje jezike prepoznaju? 2](#_Toc64675514)

[24. Kako je definisam magacinski automat i koje jezike prepoznaje? 2](#_Toc64675515)

[25. Dati definiciju konačnih automata. 2](#_Toc64675516)

[26. Objasniti vezu između konačnih automata i jezika tipa tri. 2](#_Toc64675517)

[27. Dati definiciju regularnih izraza. Navesti nekoliko primera regularnih izraza. 2](#_Toc64675518)

[28. Pokazati kako se osnovni regularni izrazi preslikavaju u automate. 2](#_Toc64675519)

[29. Objasniti namenu leksičkog analizatora. 2](#_Toc64675520)

[30. Objasniti zašto je bolje da se leksička analiza izdvoji kao posebna faza u procesu prevođenja jezika. \* 2](#_Toc64675521)

[31. Šta su to tokeni? 2](#_Toc64675522)

[32. Šta je to tablica simbola? 2](#_Toc64675523)

[33. Koje greške može da otkriva leksički analizator? 2](#_Toc64675524)

[34. Šta je i čemu služi LEX? 2](#_Toc64675525)

[35. Objasniti kako radi LEX. 2](#_Toc64675526)

[36. Objasniti strukturu LEX specifikacije. 2](#_Toc64675527)

[37. Objasniti šta je zadatak sintaksnog analizatora. 2](#_Toc64675528)

[38. Objasniti pojam levog i desnog izvođenja. 2](#_Toc64675529)

[39. Objasniti koji se problem javlja kod nejednoznačno definisanih gramatika. 2](#_Toc64675530)

[40. Koja su to dva osnovna načina sintaksne analize? 2](#_Toc64675531)

[41. Opisati osnovni algoritam za Top-down analizu. 2](#_Toc64675532)

[42. Šta je to problem leve rekurzije i kada se javlja? 2](#_Toc64675533)

[43. Dati transformacije kojima se eliminiše leva rekurzija. \* 2](#_Toc64675534)

[44. Šta je to indirektna rekurzija i kako se eliminiše? 2](#_Toc64675535)

[45. Objasniti osnovni algoritam za Bottom-up analizu. 2](#_Toc64675536)

[46. Dati definiciju proste LL-1 gramatike. 2](#_Toc64675537)

[47. Kako je definisana Sintaksna tabela proste LL-1 gramatike? 2](#_Toc64675538)

[48. Objasniti algoritam za sintaksnu analizu proste LL-1 gramatike. 2](#_Toc64675539)

[49. Objasniti pojam leve faktorizacije. 2](#_Toc64675540)

[50. Dati definiciju funkije FIRST. \* 2](#_Toc64675541)

[51. Dati definiciju funkcije FOLLOW. \* 2](#_Toc64675542)

[52. Dati definiciju LL-1 gramatika bez pravila. \* 2](#_Toc64675543)

[53. Kako se popunjava Sintaksna tabela kod LL-1 gramatika bez pravila? 2](#_Toc64675544)

[54. Kako se popunjava Sintaksna tabela kod LL-1 gramatika sa pravilima? \* 2](#_Toc64675545)

[55. Dati definiciju operatorskih gramatika. \* **Error! Bookmark not defined.**](#_Toc64675546)

[56. Definisati osnovne relacije prvenstva između terminalnih simbola. \* 2](#_Toc64675547)

[57. Dati definiciju Operatorske gramatike prvenstva. 2](#_Toc64675548)

[58. Šta je to Operatorska tablica prvenstva? Dati primer. 2](#_Toc64675549)

[59. Šta su to funkcije prvenstva i kako se određuju? 2](#_Toc64675550)

[60. Opisati algoritam za sintaksnu analizu Operatorskih gramatika prvenstva. \* 2](#_Toc64675551)

[61. Dati definiciju Gramatika prvenstva. 2](#_Toc64675552)

[62. Šta je to Tablica prvenstva i kako se popunjava. 2](#_Toc64675553)

[63. Opisati algoritam za sintaksnu analizu Gramatika prvenstva. \* 2](#_Toc64675554)

[66. Objasniti strukturu LR analizatora. 2](#_Toc64675555)

[67. Objasniti strukturu LR tablice sintaksne analize. 2](#_Toc64675556)

[68. Objasniti postupak LR analize. 2](#_Toc64675557)

[69. Šta su to LR(0) članovi? 2](#_Toc64675558)

[70. Šta je to kanonička kolekcija LR(0) članova? 2](#_Toc64675559)

[71. Algoritam za LR sintaksnu analizu 2](#_Toc64675560)

[72. Objasniti mesto i namenu međukoda u strukturi kompilatora. 2](#_Toc64675561)

[73. Navesti osnovne vrste međukoda. 2](#_Toc64675562)

[74. Dati primer sintaksnog stabla. 2](#_Toc64675563)

[75. Šta je to troadresni medjukod. 2](#_Toc64675564)

[76. Objasniti strukturu *quadrupils* i dati primer. 2](#_Toc64675565)

[77. Objasniti strukturu *triples* i dati primer. 2](#_Toc64675566)

[78. Objasniti strukturu *Indirect triples* i dati primer. 2](#_Toc64675567)

[79. Navesti osnovne naredbe troadresnog međukoda. 2](#_Toc64675568)

[80. Dati primer semantičkih pravila kojima se generiše troadresni međukod za izraze. 2](#_Toc64675569)

[81. Dati primer semantičkih pravila kojima se generiše troadresni međukod za promenljive sa indeksima. 2](#_Toc64675570)

[82. Dati primer semantičkih pravila kojima se generiđe troadresni kod osnovnih upravljačkih struktura. 2](#_Toc64675571)

[83. Dati primer semantičkih pravila kojima se generiše troadresni mešukod za Boolove izraze. 2](#_Toc64675572)

[84. Šta je zadatak modula za proveru tipova podataka. 2](#_Toc64675573)

[85. Dati primer semantičkih pravila kojima se generišu atributi tipa identifikatora. 2](#_Toc64675574)

[86. Dati primer semantičkih pravila kojima se vrši provera kompatibilnosti tipova u izrazima. 2](#_Toc64675575)

[87. Dati primer semantičkih pravila kojima se vrši provera naredbi. 2](#_Toc64675576)

[89. Šta je zadatak modula za proveru tipova podataka. 2](#_Toc64675577)

[90. Objasniti pojam statičke alokacije memorije i navesti sekvencu naredbi koju u tom slučaju generiše kompilator 2](#_Toc64675578)

# Objasniti namenu programskih prevodioca?

Uvek kada softversku aplikaciju razvijamo nekim višim programskim jezikom, a to je danas standardni pristup, treba nam softver koji će taj program da prevede u mašinski jezik, zato što svi računari, ma koliko složeni bili, obrađuju instrukcije predstavljene u mašinskom (binarnom) obliku. Ukoliko je programski jezik koji koristimo na višem nivou, zadatak tog softvera je složeniji. Programski prevodioci su softverske komponente koje generalno prevode programe pisane u jednom programskom jeziku u odgovarajuće programe pisane drugim programskim jezikom, Slika 1.1

Izlazni jezik

Ulazni jezici

PROGRAMSKI   
PREVODILAC  
(TRANSLATOR)

1. Slika 1.1 Programski prevodilac

Obično je jezik sa kog se prevodi višeg nivoa od jezika na koji se vrši prevođenje, mada u opštem slučaju možemo da prevodimo sa jednog jezika višeg nivoa na drugi jezik višeg nivoa, ali i da se kod niskog nivoa prevodi u kod višeg nivoa ( npr. krosasembleri).

# Sta je to assembler?

Mašinski jezik je programski jezik najnižeg nivoa i podrazumeva da se program sastoji od naredbi i podataka napisanih u biarnom obliku (preko nizova jedinica i nula) uz korišćenje apsolutnih adresa podataka. Na ovom nivou programski prevodilac nije potreban, naredbe se upisuju direktno u operativnu memoriju i izvršavaju.

Drugu generaciju jezika čine asemblerski jezici kod kojih se jedna mašinska naredba predstavlja simbolički uz korišćenje simboličkih adresa naredbi i podataka. Ovi jezici su višeg nivoa od mašinskih ali su i dalje mašinski zavisni. Naime, asemblerski jezik se projektuje zajedno sa procesorom računara i struktura procesora bitno utiče na strukturu asemblerskog jezika i obrnuto. Za prevođenje zapisa sa nivoa asemblerskog jezika na mašinski nivo potreban je programski prevodilac koji se u ovom slučaju naziva Asembler (*assembler*), **Error! Reference source not found.**, pa odatle i naziv ovih jezika.

*Asemblerski jezik*

ASEMBLER

*Rezultati*



*Podaci*

*Mašinski jezik*

# Cemu sluze makroprocesori?

Makroasemblerski jezici su nešto višeg nivoa od asemblerskih jezika i omogućavaju skraćeno zapisivanje programa tako što se grupa asemblerskih naredbi predstavi jednom makro naredbom. Kod programa napisanih na ovom nivou najpre se koristi makroprocesor čiji je zadatak da pozive makro naredbi zameni odgovarajućim sekvencama asemblerskih naredbi. Nakon toga se, pomoću odgovarajćeg asemblera, dobijeni asemblerski kod prevodi u mašinski, Slika 3.1.

*Rezultati*

Asemblerski jezik

Makroasemblerski jezik

MAKROPROCESOR

ASEMBLER



*Podaci*

Mašinski jezik

Slika 3.1 Makroprocesor

Kako je asemblerski jezik niskog nivoa, a posebno zbog činjenice da je svakoj asemblerskoj naredbi odgovara jedna mašinska naredba, razvoj asemblera nije komplikovan zadatak. Slično je i sa makroproceosorom, čiji je zadatak prvenstveno da pokupi i zapamti definicije makro naredbi i da zameni pozive makronaredbi odgovarajućim sekvencama asemblerskih naredbi.

# Objasniti razliku izmedju kompilatora i interpretatora?

Kada se za razvoj programa koriste viši programski jezici onda je proces prevođenja na mašinski nivo je nešto složeniji. U te svrhe se kao programski prevodioci koriste kompilatori i interpretatori, **Error! Reference source not found.**. Osnovna razlika između kompilatora i interpretatora je u tome što kompilator najpre vrši analizu programa na osnovu koje generiše izvršni fajl celog programa. Interpretatori koriste drugu strategiju. Oni prevode naredbu po naredbu programa i čim formiraju celinu koja može da se izvršava, izvršavaju je.

# Navesti osnovne osobine kompilatora?

Kompilator započinje analizu programa fazom leksičke analize. U ovoj fazi identifikuju se leksičke celine i nastoji se da se uprosti zapis programa kako bi se pripremio za sintaksnu analizu. Na primer, za proces sintaksne analize nije važno da li je neka promenljiva u programu nazvana A ili B već je važno da se na određenom mestu javlja poromenljiva. Zbog toga se sve leksičke celine zamenjuju odgovarajućim simbolima a sve dodatne informacije o njima pamte u posebnoj strukturi koja se naziva **Tablica simbola** (*Symbol Table*).

Nakon leksičke analize sledi faza sintaksne analize. Zadatak ove faze je da se utvrdi da li je program tačno napisan u skladu sa pravilima kojima je definisan programski jezik koji je korišćen. Danas se koriste formalni opisi programskih jezika preko određenih meta jezika. U suštini zadatak sintaksnog analizatora je da rasčlani složene naredbe jezika na elementarne. Kao rezultat sintaksne analize dobija se sintaksno stablo koje pokazuje koja pravila i kojim redosledom treba primeniti da bi se generisala određena naredba jezika, odnosno od kojih se elementarnih naredbi sastoji svaka složena naredba.

Na osnovu sintasnog stabla genriše se međukod programa. Međukod je reprezentacija programa koja je mašinski nezavisna ali se jednostavno preslikava u mašinski ili asemblerski kod za određenu mašinu.

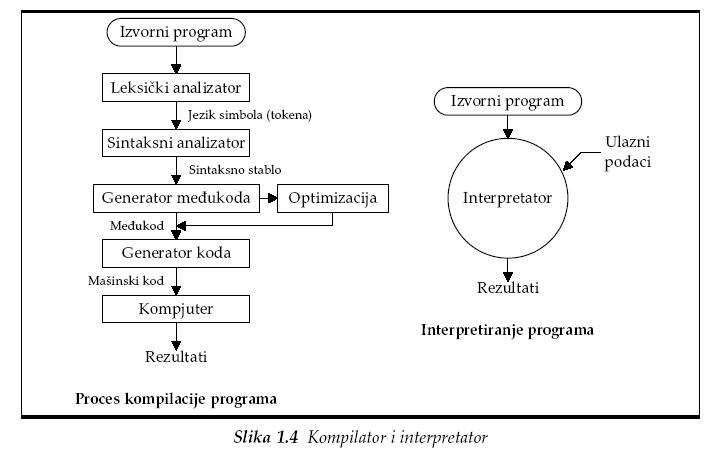
Kod koji kompilator generiše u prvom prolazu obično sadrži puno redundatnosti pa se zato mora vršiti optimizacija koda. Ova optimizacija može da bude mašinski nazavisna, kada se vrši na nivou međukoda ali i mašinski zavisna, kada se vrši na nivou generisanog koda.

Savremeni kompilatori pored sintaksne vrše i semantičku analizu koja se često integriše sa sintaksnom analizom ili se izvršava kao nezavisna faza. Zadatak semantičke analize je da utvrdi da li su u pisanju programa ispoštovana dodatna, semantička pravila koja se ne mogu formalno definisati kroz formalni opis jezika. Takva pravila su recimo pravila vezana za koncept jakih tipova podataka ili pravila o tome kojim stvarnim parametrima mogu da se zamenjuju stvarni parametri i sl. Ova dodatna pravila se obično definišu kao procedure koje se izvršavaju prilikom primene određenog sintaksnog pravila.

# Objasniti sta je to medjukod?

Na osnovu sintasnog stabla genriše se međukod programa. Međukod je reprezentacija programa koja je mašinski nezavisna ali se jednostavno preslikava u mašinski ili asemblerski kod za određenu mašinu. Koncept korišćenja međukoda kod kompilatora ima niz prednosti. Kako je međukod mašinski nezavisan, jednostavno se može preslikati u bilo koji drugi mašinski jezik. To znači da se na osnovu jednom generisanog međukoda može generisati mašinski kod za različite mašine i ostvariti prenosivost koda.

# Nacrtati potpunu semu strukture kompilatora?



# Definisati pojam azbuke.

Prilikom opisa jezika polazi se od azbuke kao osnovnog pojma. Neka je **V** konačan neprazan skup elemenata. Elemente skupa **V** nazivamo simbolima, slovima, a sam skup apstraktnom azbukom ili samo azbukom.

*Primer 8.1 Primeri azbuka*

**V**={0,1} Azbuci V pripadaju samo cifre 0 i 1;

**V**={*a,b,c*} Azbuci pripadaju samo slova *a*,*b* i *c*

Za opis programskih jezika se koriste nešto složenije azbuke.

Na primer jednu takvu azbuku mogu da čine sledeći simboli:

- Velika i mala slova abecede: *A,a,B,b,C,c* ...

- Specijalni znaci: +, -,\*, :=, ...

- Reči kao što su: **begin**, **end**, **if**, **then**, ...

Napomena: U ovom slučaju se svaka reč tretira kao jedan simbol.

# Dati formalnu definiciju pojma reči.

Reč u kontekstu formalnih jezka se definiše kao konačan niz simbola azbuke **V**. Niz koji ne sadrži nijedan simbol naziva se **prazna reč** i označava sa .

|  |
| --- |
| ***Primer 2.2 Reči***  Neka je **V**={*a,b,c*}. Sledeći nizovi su reči azbuke **V**: , *a, b, c, aa, bb, cc, ab, ac, abc, aabc* |

Reči su uređeni nizovi tako da se reč *ab* razlikuje od reči *ba*.

# Navesti osnovne delove reči.

Razlikujemo i sledeće pojmove:

**Prefiks** reči **x** je niz koji se dobija izbacivanjem nijednog ili više krajnjih simbola reči **x**.

**Sufiks** reći **x** je niz koji se dobija izbacivanjem nula ili više početnih simbola reči **x**.

**Podniz** reči **x** je reč koja se dobija kada se izbaci neki prefiks i neki sufiks reči **x**. Svaki prefiks i svaki sufiks reči **x** su podnizovi reči **x**, dok svaki podniz reči **x** ne mora da bude ni sufiks ni prefiks reči **x**. Za svaku reč **x** i **x** i  su njeni prefiksi, sufiksi i podnizovi.

***Primer 2.7 Delovi reči***

Neka je data reč *banana.*

Reči *b, ba, ban, bana, banan* i *banana* su njeni prefiksi,

*b, ba, ban, bana, banan* supravi prefiksi*.*

Reči *a, na, ana, nana, anana i banana* su njeni sufiksi

*a, na, ana, nana, anana* su pravi sufiksi.

*aba, ana, anan* su podnizovi reči *banana*.

*baaa i bnna su* podsekvenca reči *babana.*

# Navesti osnovne operacije koje se mogu izvršavati nad rečima.

**a. Proizvod reči**

Ako su **x** i **y** dve reči azbuke V, proizvod ili spajanje reči je operacija kojom se stvara nova reč tako što se na jednu reč nadovezuje druga reč.

|  |
| --- |
| ***Primer 2.4 Proizvod reči***  Neka je: **x** = *aA* i **y** = *ab.* Proizvodom ovih reči nastaje reč**z** = **xy** = *aAab* |

Takođe važi da je:**x** = **x** = **x.** Prazna reč je neutralni element za operaciju proizvoda (nadovezivanja) reči.

**b. Eksponent reči**

Višestruki proizvod iste reči se definiše kao eksponent reči.

|  |
| --- |
| ***Primer 2.5 Eksponent reči***  **xx** = **x**2  **xxx** = **x**3.  **x**1=**x**  **x**0= . |

**c. Transponovana reč**

Transponovana reč reči **x** u oznaci **x**T definiše se na sledeći način:

* + - 1. T = 
      2. (xa)T = axT

Rezultat operacije transponovanja reči je reč u kojoj su slova ispisana u obrnutom redosledu u odnosu na originalnu reč.

|  |
| --- |
| ***Primer 2.6.Transponovana reč***  Neka je data reč**x***=baba.* Tada je**x**T*=abab*  Ako je**x***=radar* tada je**x***T=radar.* Ovakve reči nazivamo palindromima. |

# Dati formalnu definiciju pojam jezika.

Neka je data azbuka **V**. Skup svih reči nad azbukom **V** se označava sa **V\***. Napomenimo da je skup **V\*** uvek beskonačan, bez obzira na to kako je definisana azbuka **V.** Čak i u slučaju kada azbuka V sadrži samo jedan simbol **V**={*a*}, skup **V\***={, *a*, *aa*, *aaa*, *aaaa*, ...}.

Formalni jezik nad azbukom **V** je bilo koji podskup skupa **V\* .** Bilo koji podskup bilo da je konačan ili beskonačan predstavlja jezik. Kako je V\* uvek beskonačan skup broj njegovih podskupova je takođe beskonačan.

# Navesti osnovne operacije nad jezicima.

Ako je **L** formalni jezik onda se može definisati i njegov reverzni jezik u oznaci **LR**, kao skup svih reverznih reči jezika **L**.

**LR** ={**x**T | **x****L**}

Ako su **L** i **M** formalni jezici, onda je **L****M**, unija ova dva jezika i obuhvata sve reči koje pripadaju ili jeziku **L** ili jeziku **M**.

**L**  **M**={**x** | **x** **L**  **x** **M**}

Ako su **L** i **M** formalni jezici, onda je **LM**, proizvod ova dva jezika i obuhvata sve reči nastaju operacijom nadovezivanja reči jezika **M** na reči jezika **L**.

**L ** **M**={**xy** | **x** **L**  **y** **M**}

Potpuno zatvaranje jezika **L**, koje se označava sa **L\*,** je skup svih reči koje nastaju kao proizvodi reči jezika **L** uključujući i .



Pozitivno zatvaranje jezika L, za koje se koristi oznaka **L+** se definiše kao**:**



# Šta su to formalne gramatike i kako se definišu? \*

Noam Chomsky je još u svojim ranim radovima dao sledeću definiciju formalnih gramatika: Svaka formalna gramatika **G** se može definisati kao skup **G**=(**Vt**, **Vn**, S, **P**) gde je **Vt** skupterminalnih sibola**, Vn** skup neterminalnih simbola**, S** startni simbol i **P** skup smena definisan na sledeći način:

pri čemu je **V** = **Vt** **Vn** i važi **Vt** **Vn  =** 

Uočimo da je uslov da reč na levoj strani pravila mora da sadrži bar jedan neterminalni simbol.

Formalni jezik **L** definisan gramatikom **G** je skup reči koje se sastoje od terminalnih simbola a dobijaju se tako što se na starni simbol primene pravila gramatike:

# Šta su to neredukovane gramatike. Dati primer.

Gramatika mora da bude redukovana. Redukovana gramatika je gramatika koja ne sadrzi beskorisna pravila. Beskorisna pravila su ona koje sadrže simbole koji ne mogu da budu izvedeni iz startnog simbola gramatike, ili smene koje sadrže “beskonačne” simbole.

***Primer 2.13 Neredukovana gramatika***

**Gramatika G**=( {*W*,*X*,*Y*,*Z*}, {*a*}, *W*, **P**} gde je:

**P**: 1. *W → aW*

*2. W → Z*

*3. W → X*

*4. Z → aZ*

*5. X→ a*

*6. Y→ aa*

je primer neredukovane gramatike iz sledećih razloga:

1. Pravilo 2. je neupotrebljivo pravilo jer je Z beskonačan simbol, nikada se ne ukida.
2. Pravilo 4. je neupotrebljivo iz istog razloga.
3. Pravilo 6. je neupotrebljivo zato što ne postoji izvođenje *W*α*Y*β. Nikada se ne generiše neterminal *Y*.

# Sta su to nejednoznačne gramatike. Dati primer.

Gramatika ne mora da bude jednoznačna. Kod nejednoznačnih gramatika za jedan ulazni niz je moguce kreirati veći broj sintaksnih stabala.

***Primer 2.14 Nejednoznačna gramatika***

**G**=( {*Izraz*}, {**const**,**+**,**\***}, *Izraz*, **P**)

**P:** *Izraz* → *Izraz* **+** *Izraz*

*Izraz* → *Izraz* **\*** *Izraz*

*Izraz* → **const**

# Navesti osnovne tipove gramatika po Čomskom.

Još u svojim ranim radovima Chomsky je identifikovao četri tipa formalnih gramatika i jezika. Gramatike koje zadovoljavaju gore date opšte uslove je nazvao gramatikama tipa nula. Praktično, svaka formalna gramatika je gramatika tipa nula. Ostali tipovi gramatika imaju strože definisane uslove koje treba da zadovolje pravila. To znači da se tim tipovaima gramtika može definisati uži skup jezika, ali se zato može jednostavnije vršiti analiza i prepoznavanje takvih jezika.

Gramatike tipa jedan ili Konteksne gramatike

Gramatike tipa jedan su formalne gramatike koje pored opštih uslova koji zadovoljavaju gramatike tipa nula imaju pravila **x y** koja zadovoljavaju sledeći uslov:

| **x |  | y |**

To znači da se reči sa leve strane smena mogu preslikavati samo u reči koje su jednake ili veće dužine. Drugim rečima, nisu dozvoljena pravila oblika:

**x **

Gramatike tipa jedan kao i gramatike tipa nula se obično nazivaju i konteksnim gramatikama. To je zbog toga što se neterminal koji je obavezan na levoj strani smene kod ovih gramatika nalazi u nekom kontekstu i čitav taj kontekst se preslikava u novu reč.

Gramatike tipa dva ili Beskontekne gramtike

Gramatike tipa dva su gramatike kod kojih su pravila definisana tako da se na levoj strani nalazi samo neterminal. Neterminali se preslikavaju u reči. Kako se neterminali prilikom izviđenja posmatraju izolovano od konteksta u kome se nalaze ove gramatike se nazivaju beskonteksnim gramatikama. Znači smene kod ovih gramatika su oblika:

*A***y,** gde je*A***Vn** i **y Vt\***

Za opis programskih jezika se obično koriste beskonteksne gramatike.

Gramatike tipa tri su formalne gramatike koje zadovoljavaju opšte uslove ali su orgraničene na smene oblika:

*A**a B*ili **,** *A**a*gde je*A***Vn** i **a Vt**

Naziv regularne gramatike dolazi odatle što su ove gramatikama tipa tri generišu regularni izrazi koji se efikasno mogu prepoznavati konačnim automatima. Regularne gramatike imaju široku primenu u mnogim oblastima u kojima treba vršiti prepoznavanje nizova. U kontekstu programskih prevodilaca koriste se za opis leksičkih elemenata jezika (identifikatora, konstanti, literala i sl.) koje prepoznaje leksički analizator, o čemu će biti reči kasnije.

# Šta su to konteksne, a šta beskonteksne gramatike?

Gramatike tipa jedan kao i gramatike tipa nula se obično nazivaju i konteksnim gramatikama. To je zbog toga što se neterminal koji je obavezan na levoj strani smene kod ovih gramatika nalazi u nekom kontekstu i čitav taj kontekst se preslikava u novu reč.

Gramatike tipa dva beskontekstne

Neterminali se preslikavaju u reči. Kako se neterminali prilikom izviđenja posmatraju izolovano od konteksta u kome se nalaze ove gramatike se nazivaju beskonteksnim gramatikama.

# Šta su to normalne forme gramatika?

Pod normalnim formama gramatika podrazumevamo standardni način zadavanja gramatika određenog tipa. To znači da se smene jedne gramatike mogu prikazati na neki drugi način a da se pri tome ne promeni jezik koji definišu. Tako dolazimo i do pojma ekvivalentnih gramatika. Dve gramatike su ekvivalentne ako definišu isti jezik.

Naime, nekada je potrebno smene gramatike predstaviti na baš određeni način da bi se mogao primeniti neki određeni postupak analize. U te svrhe su i definisane različite normalne forme za određene tipove gramatika. Pomenućemo samo neke.

# Šta su to automati. Objasniti kako teče proces prepoznavanja reči automatom.

Problem prepoznavanja jezika, odnosno utvrđivanja da li zadata reč pripada jeziku opisanim određenom gramatikom, rešava se automatima kao uređajima za prepoznavanje jezika.

Generalno gledano svi automati se mogu predstaviti šemom datom na Slika 20.1, Niz simbola (reč) koji se prepoznaje zapisan je na ulaznoj traci. Po ulaznoj traci se kreće ulazna glava koja u jednom trenutku čita jedan simbol sa trake. Ova glava se u opštem slučaju može kretati i napred i nazad.



Slika 20.1 Šema automata kao uređaja za prepoznavanje jezika

Funkcionalni deo automata je njegov najznačajniji deo. Može da se nađe u nizu različitih stanja u zavisnosti od toga kako je opisan jezik koji se prepoznaje. Funkcionalni organ beleži predistoriju prepoznavanja i za to koristi memoriju koja je u ovom opštem modelu predstavljena kao beskonačna memorija. Automat sa beskonačnom memorijom nije praktično izvodljiv tako da se realni automati razlikuju od ovog modela po tome što koriste neki određeni tip memorije.

Proces prepoznavanja reči koja je upisana na ulaznu traku kreće tako što se ulazna glava pozicionira na startni simpol (#), a funkcionalni organ se nalazi u svom početnom stanju. U svakom koraku prepoznavanja, u zavisnosti od toga gde je pozicionirana ulazna glava (koji simbol čita) i stanja funkcionalnog organa, kao i simbola u radnoj memoriji koji čita radna glava događaju se sledeće promene u automatu:

* menja se stanje funkcionalnog organa
* u radnu memoriju se upisuje jedan ili više simbola
* radna glava se pomera napred, nazad ili ostaje na poziciji na kojoj se i nalazi.

Proces prepoznavanja reči je uspešno završen u trenutku kada se ulazna glava pozicionira na krajnji granični simbol (desni #) i funkcionalni organ se nadje u jednom od svojih završnih stanja.

U opštem slučaju automati se mogu podeliti na četri klase:

* 2N Dvosmerni nedeterministički
* 1N Jednosmerni nedeterministički
* 2D Dvosmerni deterministički
* 1D Jednosmerni deterministički

Podela na jednosmerne i dvosmerne izvršena je na osnovu mogučnosti pomeranja ulazne glave. Ako se ulazna glava kreče samo napred onda su to jednosmerni automati, a u slučaju kada je dozvoljeno i vraćanje glave unatrag onda su to dvosmerni automati. Podela na determinističke i nedeterminističke izvršena je u odnosu na preslikavanje stanja funkcionalnog organa koje se vrši u toku prepoznavanja . Ukoliko je stanje u koje prelazi automat za određeni ulazni simbol i određeno zatečeno stanje, jednoznačno definisano, onda je to deterministički automat. Ukoliko je ovo preslikavanje višeznačno onda je to nedeterministički automat.

# Kako je definisana Tjuringova mašina?

**Error! Reference source not found.** prikazuje šemu Tjuringove mašine kao osnovnog automata za prepoznavanje jezika. Za razliku od opšteg modela automata, Tjuringova mašina ima radnu memoriju u vidu beskonačne trake.



2N Tjuringova mašina se može opisati sledećom entorkom: Tm=(**Q**, **V**, **S**, **P**, q0, *b,* **F**) gde je:

**Q** – Skup stanja operativnog organa

**V** – Skup simbola na ulaznoj traci

**S** – Skup simbola na radnoj traci

**F** – Skup završnih, krajnjih stanja (Podskup skupa Q)

q0 – Početno stanje operativnog organa

*b* – Oznaka za prazno slovo

**P** – Preslikavanje definisano sa:

Ako je (qj, *B*, d1, d2) P(qi, *a*, *A*), gde je qj, qi  Q, *a*V, *A*S i d1,d2 {-1, 0, 1},

odnosno ako je ulazna glava pozicionirana na simbolu *a*, funkcionalni organ se nalazi u stanju qi, a radna glava iznad simbola *A*, automat će preći u stanje qj, na radnu traku će upisti simbol **B** i izvršiće pomeranje ulazne glave i radne glave u skladu sa vrednostima d1 i d2. Pri tome važi sledeća notacija: Ako je d=0 nema pomeranja radne glave, za d=1 glava se pomera za jedno mesto u desno, a za d=-1 glava se pomera za jedno mesto u levo (vraća se nazad).

Mašina započinje prepoznavanje tako što je ulazna glava pozicionirana na početnom graničnom simbolu, funkcionalni organ se nalazi u početnom stanju i radna traka je prazna. Prepoznavanje reči se završava uspešno ako se u trenutku kada se ulazna glava pozicionira na krajni granični simbol #, automat nađe u nekom od završnih stanja (elemenata skupa **F**).

# Koje jezike prepoznaju automati tipa Tjuringove mašine?

Dokazano je da su sve četri klase Tjuringovih mašina, 2N, 1N, 2D, 1D su međusobno ekvivalentne i svaka od njih prepoznaje jezike tipa nula. To u suštini znači da je za prepoznavanje bilo kog jezika tipa nula moguće definisati bilo koji tip Tjuringove mašine, odnosno ako je moguće definisati jedan tip Tjuringove mašine onda je moguće definisati i ekvivalentne mašine ostalih tipova. Jedini problem u svemu ovome je to što model Tjuringove mašine podrazumeva beskonačnu memoriju, što je u praksi praktično nije izvodljivo. Zbog toga Tjuringova mašina ostaje samo teorijski model a u praksi se koriste automati sa konačnim memorijama, što kao posledicu ima ograničenje jezika koji se mogu prepoznavati.

Može se reći da je pojam Tjuringove mašine ekvivalent pojmovima algoritam i jezici tipa nula. To znači da se sva algoritamska preslikavanja mogu opisati jezicima tipa nula, pri čemu je moguće definisati odgovarajuću Tjuringovu mašinu za prepoznavanje jezika.

# Šta su to Linearno ograničeni automati i koje jezike prepoznaju?

Linearno ograničeni automati su u suštini Tjuringove mašine sa konačnom trakom kao radnom memorijom. Naime 2N Tjuringova mašina kod koje može da se odredi konstanta *k* takva da je dužina reči koja se upisuje na radnu traku najviše *k* puta veća od dužine reči koja se prepoznaje, naziva se Linearno ograničenim automatom. Odnosno, ako se prepoznaje reč w**V\*,** gde je **|w**|=*n*, na radnu traku se upisuje reč dužine *r,* pri čemu je *r n\*k*

Posledica ovog ograničenja je da Linerano ograničeni automati ne prepoznaju sve jezike tipa nula. Odnosno, postoje jezici tipa nula za koje nije moguće definisati linearno ograničeni automat. Naime dokazana su sledeća tvrđenja:

1. Klase 2N i 1N Linearno ograničenih automata su međusobno ekvivalentne i svaka od njih prepoznaje jezike tipa jedan. Drugim rečima, za svaki jezik tipa jedan može se definisati nedeterministički LOA koji ga prepoznaje.
2. Klase 2D i 1D Linearno ograničenih automatu su međusobno ekvivalentne. To znači da se za svaki 2D LOA može definisati ekvivalentan 1D LOA.
3. Ekvivalentnost 2N i 2D LOA do sada nije dokazana.
4. Svaka klasa LOA ne prepoznaje sve jezike tipa jedan.

# Kako je definisam magacinski automat i koje jezike prepoznaje?

Magacinski automat kao radnu memoriju koristi memoriju organizovanu u vidu magacina. Šema strukture ovog automata data je na Slika 24.1.



Slika 24.1 Magacinski automat

2N Magacinski automat je definisan entorkom Ma=(**Q**, **V**, **S**, **P**, q0, *Z*0, **F**) gde je:

**Q** – Skup stanja operativnog organa

**V** – Skup simbola na ulaznoj traci

**S** – Skup magacinskih simbola

**F** – Skup završnih, krajnjih stanja (Podskup skupa **Q**)

q0 – Početno stanje operativnog organa

*Z*0 – Početni simbol na vrhu magacina

**P** – Preslikavanje definisano sa:



***Primer 3.1* *Magacinski automat***

Neka je data gramatika **G**=({0,1,*a*}, {*S*}, *S*, **P**), sa skupom smena

**P:**

1. *S*0*S*1
2. *S**a*

Za prepoznavanje jezika definisa gramatikom **G** može da se definiše magacinski automat

Ma=(**Q**, **V**, **S**, **P**, q0, *Z*0, **F**) gde je:

******

Pokazaćemo prepoznavanje dve karakteristične reči jezika definisanog gramatikom **G.**

1. Prepoznavanje reči #000*a*111#

******

1. Prepoznavanje reči #*a*#

Magacinski automati su pogodni za prepoznavanje bezkonteksnih jezika. Dokazana su sledeća tvrđenja:

* 1N magacinski automat prepoznaje jezik L ako i samo ako je L beskonteksni jezik tipa 2.
* 2N magacinskim automatima mogu se prepoznavati svi beskontesni i neki, ali ne svi, konteksni jezici.
* Postoje beskonteksni jezici koji se ne mogu prepoznati 1D magacinskim automatima.
* 2D magacinski automati pored bezkonteksnih mogu da prepoznaju i neke konteksne jezike.
* Magacinski automati se koriste u sintaksnoj analizi programskih jezika.

# Dati definiciju konačnih automata.

Konačni automati su uređaji za prepoznavanje regularnih jezika. Za razliku od drugih tipova nemaju radnu memoriju već se predistorija preslikavanja pamti samo preko promene stanja funkcionalnog organa, Slika 25.1.

Na osnovu stanja operativnog organa i simbola koji čita ulazna glava, menja se stanje automata i ulazna glava pomera za jedno mesto ulevo ili udesno ili ostaje na svom mestu.



Slika 25.1 Šema konačnog automata

Konačan automat je opisan entorkom Ka=(**Q**, **V**, **P**, *q0*, **F**), gde je:

**Q** – Skup stanja operativnog organa

**V** – Skup simbola na ulaznoj traci

**F** – Skup završnih, krajnjih stanja (Podskup skupa **Q**)

*q0* – Početno stanje operativnog organa

**P** – Preslikavanje pravilima oblika: **Q**  {**V** #} **Q**

Za ulazni niz **w****V**\*, kažemo da je prepoznat automatom ako automat prevodi iz početnog u neko od njegovih krajnjih stanja, odnosno ako je **P**(*q0*, w) **F**.

Za predstavljanje automata se koriste grafovi, pri čemu čvorovi grafa odgovaraju stanjima automata, a potezi preslikavanjima. Na Slici 3.5 je predstavljen graf koji odgovara automatu iz primera. Početno stanje automata je označeno strelicom, a krajnje stanje sa dva koncentrična kruga.

Kako se preslikavanje *P*(*qi*,*a*)=*qj* može tumačiti da se stanje *qi* automata za ulazno slovo *a* preslikava u stanje *qj*, svako takvo preslikavanje se na grafu predstavlja odlaznim potegom od stanja qi do stanja qj na kome je oznaka a.

***Primer 3.3* *Graf automata***

Graf automata iz primera 3.2.

1

1

1

0

0

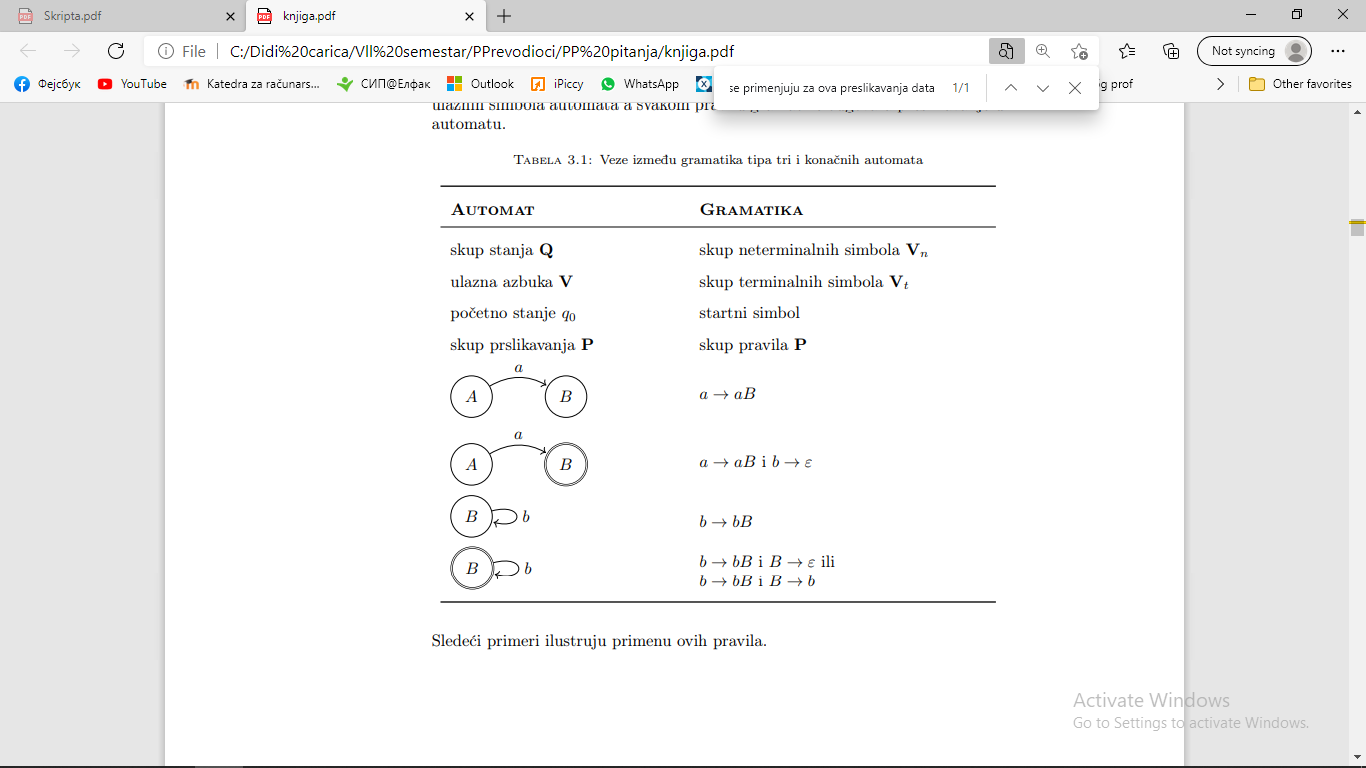
0

0

Slika 25.2 Graf automata iz Primera 3.2.

# Objasniti vezu između konačnih automata i jezika tipa tri.

Za svaki jezik definisan gramatikom tipa tri može se odrediti konačni automat koji ga prepoznaje. Pravila koja se primenjuju za ova preslikavanja data su u Tabeli



|  |
| --- |
| ***Primer 3.5* Gramatika u konačni automat**   1. Definisati konačni automat za prepoznavanje označenih celih brojeva. Označeni ceo broj definisan je sledećom gramatikom:   <OznaceniCeoBroj>  **+** <NeoznaceniCeoBroj>  <OznaceniCeoBroj>  **-** <NeoznaceniCeoBroj>  <NeoznaceniCeoBroj> :**cifra** <NizCifara>  <NizCifara>  **cifra** <NizCifara>  <NizCifara> ε  **Rešenje:**  Graf automata koji prepoznaje jezik definisan gramatikom dat je na slici 3.7. |

******

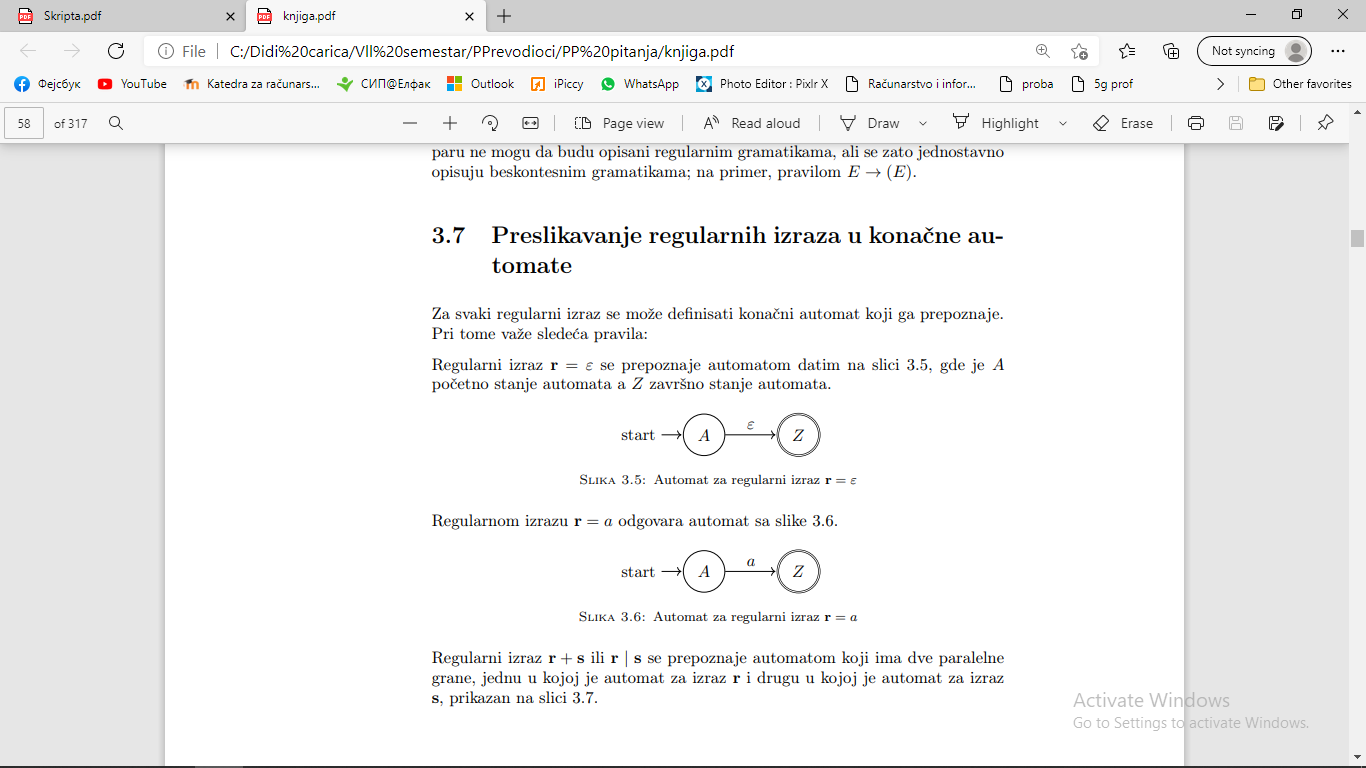
# Dati definiciju regularnih izraza. Navesti nekoliko primera regularnih izraza.

Jezici tipa tri koji se prepoznaju konačnim automatima se nazivaju i regularnim jezicima zato što se mogu opisati regularnim izrazima. Ovde ćemo posvetiti malo više pažnje pojmu regularni izraz. Oni se mogu posmatrati i kao meta jezici za predstavljanje formalnih jezika. Regularni izrazi u suštini definišu skup reči jezika. Sami regularni izrazi su nizovi formirani od slova neke azbuke **V** i skupa specijalnih simbola. Regularni izraz obično skraćeno definiše skup reči jezika definisanog nad azbukom **V**. Postoje posebna pravila kako se vrši tumačenje (denotacija) regularnog izraza i generisanje skupa reči jezika koji je opisan regularnim izrazom.

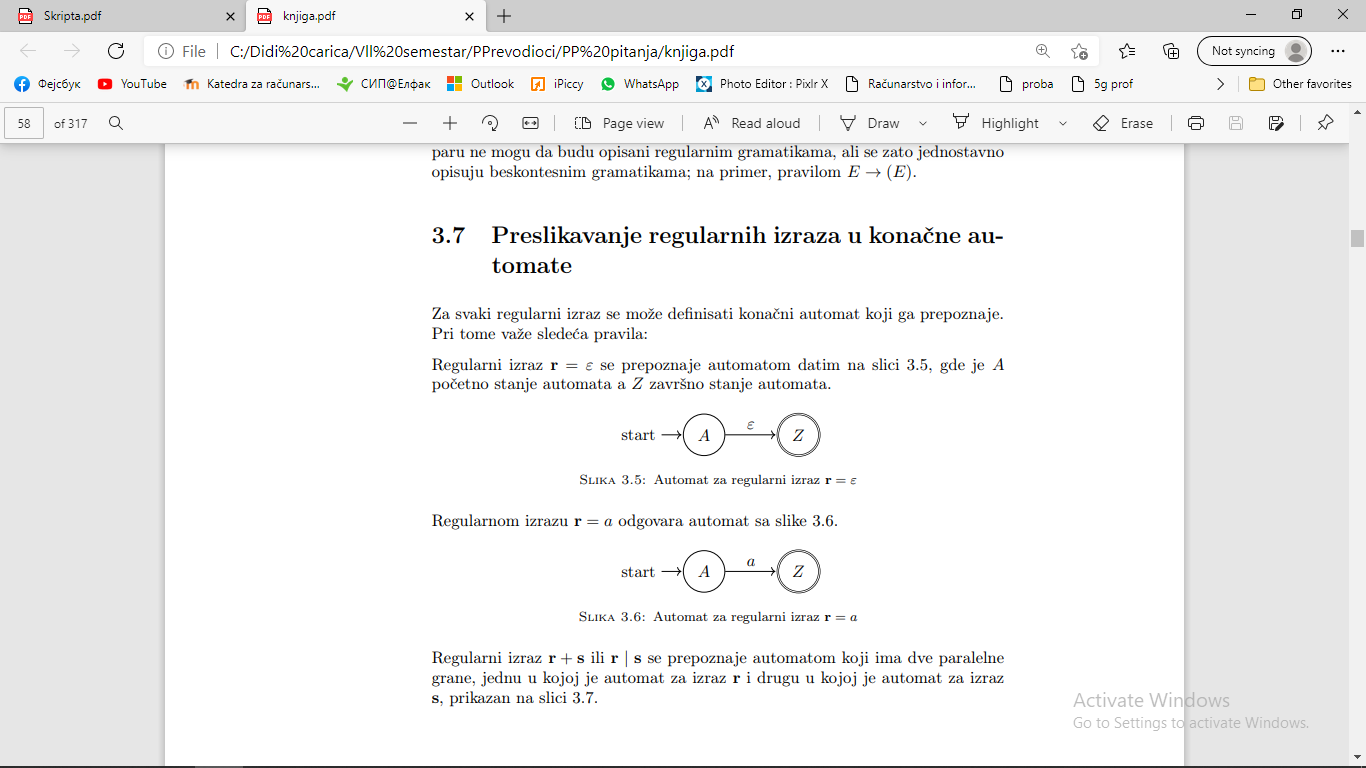
Regularni izrazi su veoma korisni za formalno i koncizno definisanje jezika. Kada je jezik konačan moguće je nabrojati sve reči koje sadrži, ali u slučaju beskonačnih jezika to postaje problem. Zbog toga se regularni izrazi i koriste u različitim disciplinama kao sredstvo za opis kompleksnih jezika. Na žalost, ne mogu se svi formalni jezici opisati regularnim izrazima. To je moguće samo u slučaju jezika tipa tri, zbog čega se oni i nazivaju regularnim.

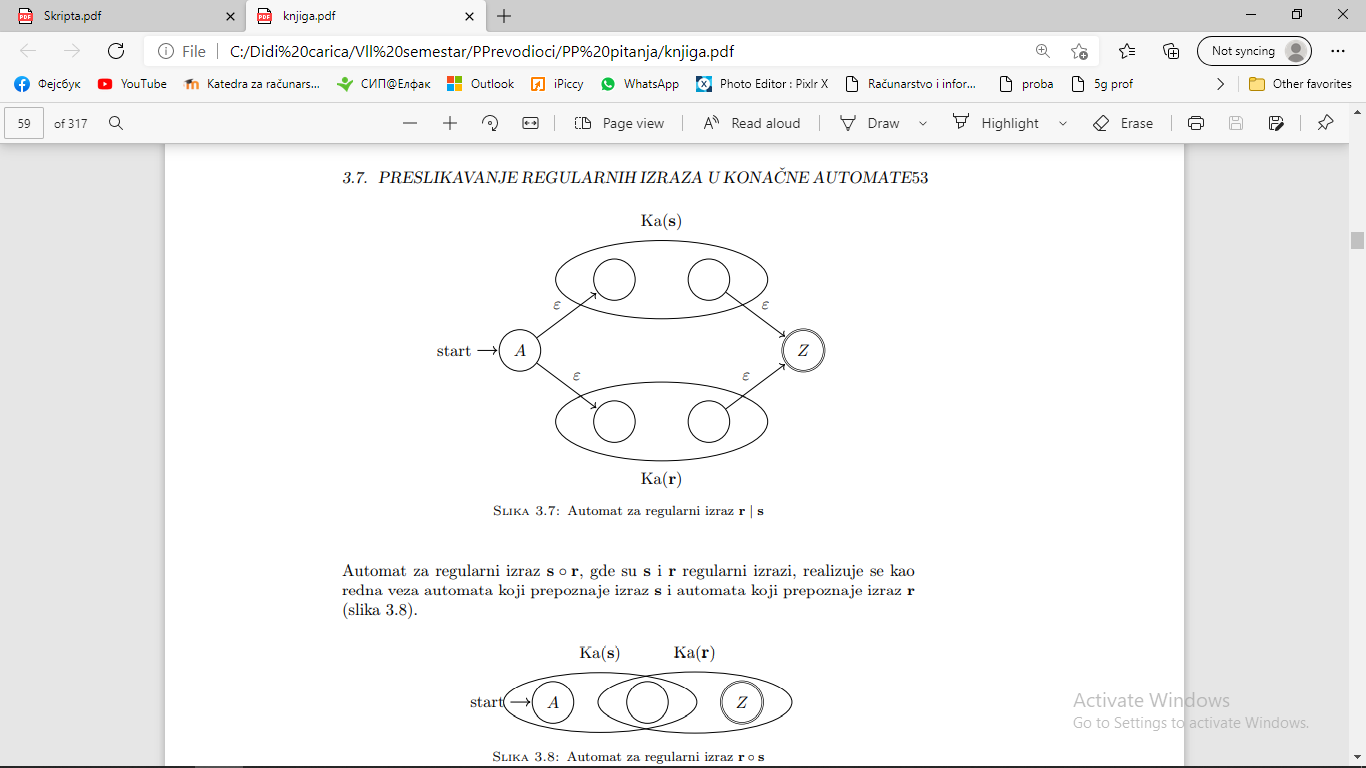
Primer : Neka je V = {a, b, c, . . . , x, y, z}. Sledeći nizovi su regularni izrazi: ∅, a, ((c ◦ a) ◦ t), ((m ◦ e) ◦ (o) ∗w), (a | (e | (i | (o | u)))), (a | (e | (i | (o | u))))∗ .

# Pokazati kako se osnovni regularni izrazi preslikavaju u automate.

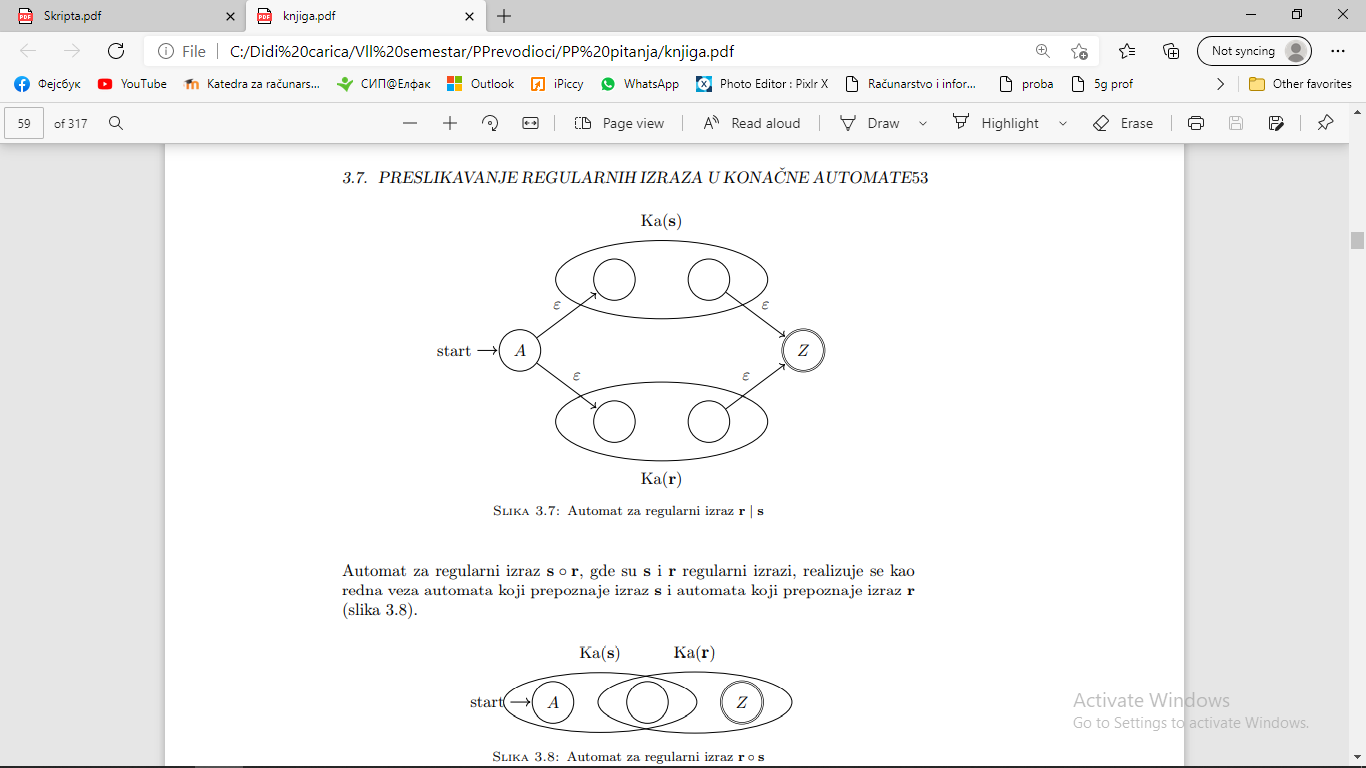
Za svaki regularni izraz se može definisati konačni automat koji ga prepoznaje. Pri tome važe sledeća pravila:

Regularni izraz r = ε se prepoznaje automatom datim na slici 3.5, gde je A početno stanje automata a Z završno stanje automata.

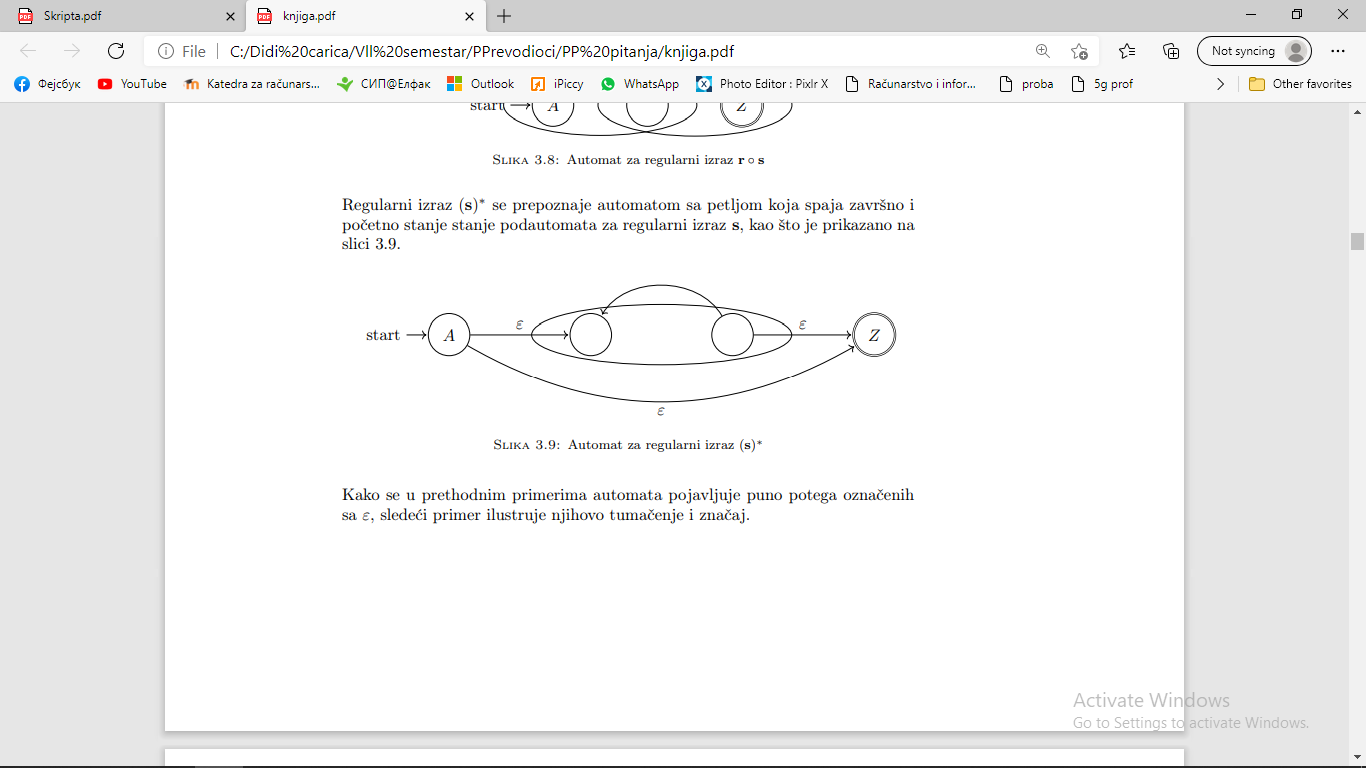
 Regularni izraz r = a



Regularni izraz r + s ili r | s se prepoznaje automatom koji ima dve paralelne grane, jednu u kojoj je automat za izraz r i drugu u kojoj je automat za izraz s.



Automat za regularni izraz s ◦ r, gde su s i r regularni izrazi, realizuje se kao redna veza automata koji prepoznaje izraz s i automata koji prepoznaje izraz r (slika 3.8).



Regularni izraz (s) ∗ se prepoznaje automatom sa petljom koja spaja završno i početno stanje stanje podautomata za regularni izraz s, kao što je prikazano na slici 3.9.



|  |
| --- |
| ***Primer 3.13 Regularni izraz u konačni automat.***  Označeni realni brojevi u eksponencijalnom zapisu definisani su sledećim regularnim izrazom:  (**+** | **-** ) **cifra** **cifra**\* **.** **cifra**\* (**E** | **e**)(**+** | **-**) **cifra** **cifra**\*  Za prepoznavanje ovih brojeva može se korititi automat predstavljen na **Error! Reference source not found.**. |

## Pravila kod regularnih izraza

* 1.  je regularni izraz.
  2.  je regularni izraz.
  3. Ako je *a*V, onda je *a* regularni izraz.
  4. Ako su r1 i r2 regularni izrazi, onda su i (r1| r2) i (r1 r2) regularni izrazi.
  5. Ako je r regularni izraz onda je i r\* takođe regularni izraz
  6. Nema drugih regularnih izraza nad azbukom V.

# Objasniti namenu leksičkog analizatora.

Leksičkim analizatorom se realizuje faza leksičke analize u procesu prevođenja jezika. Odnosno, u kodu se identifikuju leksičke celine koje imaju neki sintaksni smisao, transformišu se u simbole (tokene) i prosleđuju se sintaksnom analizatoru.

Pored ove osnovne uloge leksički analizator obavlja još neke zadatke:

* Izbacuje iz ulaznog koda delove koji nisu značajni za sintaksnu analizu: komentare, praznine (blanko znake), *tab* i *newline* simbole.
* Usklađuje listing gršaka sa ulaznim kodom. Npr. vodi računa o broju *newline* simbola, što se koristi kod referenciranja na greške.

# Objasniti zašto je bolje da se leksička analiza izdvoji kao posebna faza u procesu prevođenja jezika. \*

Postoji više razloga zbog kojih se leksički analizator izdvaja od sintaksnog analizatora. Možda najvažniji su:

* Jednostavnija realizacija.
* Tako se dobija mogućnost da se tehnike za sintaksnu analizu razvijaju nezavisno od jezika. Ulaz u sintaksni analizator je niz simbola definisan određenim formalnim jezikom, nije zavisan od programskog jezika koji se prevodi.
* Prenosivost kompilatora – U fazi leksičke analize eliminišu se svi mašinski zavisni elementi i generiše mašinski nezavisna sekvenca simbola koja se prosleđuje sintaksnom analizatoru
* Povećanje efikasnosti kompilatora. Posebnim tehnikama baferisanja koje se koriste u realizaciji leksičkog analizatora doprinosi se njegovoj efikasnosti a time i efikasnosti kompilatora.

# Šta su to tokeni?

Jedan token u suštini predstavlja skup nizova (reči). Na primer:

* Identifikator: niz koji počinje slovom a može da sadrži slova i cifre
* Integer: neprazan niz cifara, sa ili bez znaka ispred
* Ključna reč: **else, or, if, begin ....**
* Belina: Neprazan niz blanko znakova, ili znakova za: newline ili tab

Tokenima se identifikuju leksičke celine koje su od interesa za sintaksni analizator. Za sintaksnu analizu nisu bitni razmaci, znaci za nove linije, tabulatori i komentari. U nekim jezicima čak i neke reči naredbe nisu bitne. Na primer u programskom jeziku COBOL naredba može da se sastoji od reči jezika koje su bitne i koje nisu bitne za prepoznavanje naredbe već se koriste samo da bi zapis naredbe bio jasniji.

# Šta je to tablica simbola?

Pored toga što sintaksnom analizatoru predaje niz tokena leksički analizator treba da sačuva neke atribute tih tokena da bi oni bili kasnije iskorišćeni u toku generisanja koda. Na primer u fazi sintaksne analize važna je samo informacija da je na određenom mestu u nizu identifikator, a nije važno koji je to identifikator. Međutim, kasnije u fazi semantičke analize i u fazi generisanja koda, važno je koji je to identifikator, kakvog je tipa, da li je u pitanju skalar ili vektor i sl. Npr. kada se generiše listing grešaka bitno je koji je identifikator i u kojoj liniji se on nalazi.

Kako leksički analizator prvi dolazi u kontakt sa ulaznim kodom i kasnije ga transformiše on mora da sačuva te informacije za kasniju upotrebu. U te svrhe leksički analizator generiše tablicu simbola u koju smešta sve atribute relevantne za generisane tokene. Da bi se kasnije moglo pristupiti odgovarajućem slogu u tablici simbola uz token se, kao njegov atribut, prenosi i pointer na na taj slog.

# Koje greške može da otkriva leksički analizator?

Greske koje moze da otkriva leksicki analizator su greske leksickog tipa, to jest da neka rec u programu nije napisana kako treba.

# Šta je i čemu služi LEX?

Lex je generator leksickih analizatora.

# Objasniti kako radi LEX.

Lex radi tako sto formira automat ciji se graf specificira preko regularnih izraza

# Objasniti strukturu LEX specifikacije.

Lex specifikacija ima sledecu strukturu:

Ima import deo koji se ukljucuje u java kod bez provere

Deo sa definicijama makroa koji se koriste za specifikaciju pravila koja se prepoznaju

Ima deo koji sluzi da se definisu regularni izrazi koji se koriste za prepoznavanje odredjenih reci programskog jezika.

# Objasniti šta je zadatak sintaksnog analizatora.

Sintaksni analizator prima niz tokena od leksičkog analizatora i proverava da li taj niz pripada jeziku koji je opisan zadatom gramatikom. Cilj sintaksnog analizatora je da generiše sintaksno stablo za ulazni niz tokena.

# Objasniti pojam levog i desnog izvođenja.

Da bi se generisalo sintaksno stablo potrebno je odrediti koja se pravila i u kom redosledu primenjuju prilikom preslikavanja startnog simbola u analizirani niz. Generalno gledano postoje dva osnovna principa po kojima se određuje redosled pravila. Kako se za opis jezika obično koriste beskonteksne gramatike onda se u svakom koraku izvođenja jedan neterminalni simbol preslikava u neku reč pa redosled primene pravila može da bude:

1. sleva u desno – kada se zamenjuje prvi neterminalni simbol sa leve strane
2. sa desna na levo – kada se zamenjuje prvi neterminalni simbol sa desne strane.

# Objasniti koji se problem javlja kod nejednoznačno definisanih gramatika.

Sintaksno stablo koje se dobija jednim ili drugim postupkom izvođenja mora da bude jednoznačno definisano. Sintaksno stablo je osnova za generisanje objektnog koda, tako da bi dobijanje različitog stabla značilo da se za isti ulazni kod dobija različiti skup asemblerskih naredbi, odnosno kod bi bio višeznačno definisan što je nedopustivo.

# Koja su to dva osnovna načina sintaksne analize?

U odnosu na to kako obavljaju sintaksnu analizu postoje dva tipa sintaksnih analizatora:

* *Top-down* analizatori koji vrše analizu odozgo naniže i
* *Bottom-up* analizatori koji vrše analizu odozdo naviše.

U slučaju *Top-down* analize polazi se od startnog simbola i nastoji se da se odrede pravila koja treba primeniti da bi se generisala reč čija se analiza vrši. To znači da se pravila otkrivaju u redosledu u kom se i primenjuju prilikom generisanja reči, odnosno odozgo naniže ako se to posmatra na sintaksnom stablu.

U slučaju *Bottom-up* analize primenjuje se postupak redukcije. Kreće se od reči čija se analiza vrši i nastoji se da se ta reč redukuje na startni simbol. Pravila se određuju u redosledu koji je suprotan redosledu njihove primene kod generisanja reči i sintaksnog stabla, odnosno odozdo naviše ako se to gleda na sintaksnom stablu.

U principu *Bottom-up* analizatori su efikasniji i na njima se uglavnom zasnivaju komercijalna rešenja kompilatora.

# Opisati osnovni algoritam za Top-down analizu.

Kreće se od startnog simbola i bira se jedno pravilo kojim se startni simbol preslikava u neku frazu. Posle toga u svakom koraku se nastoji da se prvi neterminalni simbol sa leve strane zameni desnom stranom odgovarajućeg pravila. Praktično nastoji se da se generiše analizirani niz levim izvođenjem. Ukoliko na nekom koraku ne postoji odgovarajuća smena vraćamo se natrag do nivoa na kome je moguće primeniti neku novu alternativu. Analiza je uspešna ako se kao rezultat ovog postupka dobije niz koji se analizira. Ako na nekom nivou nema novih alternativa analiza je neuspešna.

1. U izvedenu sekvencu upisati niz *S*#, gde je *S* startni simbol, a # granični. Postaviti ***Pradni*** na prvi znak u izvedenom nizu (na statni simbol) a pokazivač ***Pulaz na***  prvi simbol ulaznog niza.
2. Ukoliko je slovo na koje pokazuje ***Pradni*** u izvedenoj sekvenci neterminalni simbol, zameniti ga desnom stranom prve smene na čijoj je levoj strani taj neterminalni simbol i postaviti ***Pradni*** na prvi simbol unete reči***.***
3. Ukoliko je tekuci simbol na koji pokazuje ***Pradni*** u izvedenoj sekvenci terminalni simbol i ukoliko je on jednak ulaznom simbolu na koji pokazuje ***Pulaz*** taj simbol se prihvata, oba pokazivača se pomeraju za jedno mesto udesno i postupak se nastavlja (prelazi se na analizu sledećeg slova).
4. Ukoliko je tekuci simbol u izvedenoj sekvenci terminalni simbol i različit od tekućeg ulaznog simbola, poništiti dejstvo poslednje primenjene smene. Vratiti se na stanje pre primene te smene i pokušati na tom nivou sa primenom nove smene. Ukoliko na tom nivou nema alternativa vratiti se nazad.. Poništiti poslednju primenjenu smenu, vratiti se na stanje pre primene te smene i pokušati sa novom alternativom na tom nivou.
5. Ukoliko se vraćanjem dodje do startnog simbola i ne postoji više smena za njegovo preslikavanje, ulazni kod sadrži sintaksnu grešku.
6. Postupak analize je uspešan kada se oba pokazivača (***Pulaz*** i ***Pradni***) nađu na graničnom simbolu.

# Šta je to problem leve rekurzije i kada se javlja?

*Top-down* sintaksnom analizom se generiše skup smena čiji redosled primene odgovara levom izvođenju. Zbog toga ovaj postupak analze ne može da se primeni kod tz. levo rekurzivnih gramatika. Levo rekurzivne gramatike su gramtike kod kojih postoje levo rekurzivna pravila, odnosno pravila oblika:

*A**A***r,** gde je*A***Vn,** a **r** **V\***

Kod ovakvih pravila preslikavanjem neterminala *A* dobija se reč koja opet počinje istim neterminalom, i to se iz koraka u korak ponavlja, odnosno ulazi se u jednu beskonačnu petlju, tako da analzu nije moguće završiti.

# Dati transformacije kojima se eliminiše leva rekurzija. \*

Svaku levu rekurzivnu gramatiku moguće je transformisati u ekvivalentnu nerekurzivnu gramatiku. navešćemo dve mogućnosti za to.

Ako gramatika sadrži skup pravila za isti neterminalni simbol, među kojima su neka levo rekurzivna, a neka ne:

Ovaj skup pravila moguće je zameniti sledećim skupom nerekurzivnih pravila:

Moguća je i sledeća transformacija:

# Šta je to indirektna rekurzija i kako se eliminiše?

Eliminisanje indirektne rekurzije moguće je sledećom transformacijom:

Sve skupove pravila oblika

*X* → *Y*α

*Y* → β1 | β2 … | β3

treba zameniti pravilima:

*X* → β1 α | β2 α … | β3 α

Posle ove transformacije treba se osloboditi svih direktnih levo-rekurzivnih smena, ukoliko postoje.

# Objasniti osnovni algoritam za Bottom-up analizu.

U slučaju *Bottom-Up* analize primenjuje se postupak redukcije reči koja se analizira. Osnovni algoritam za analizu se sastoji u tome što se ulazni niz analizira slovo po slovo od početka prema kraju i nastoji se da se neki njegov podniz prepozna kao desna stana nekog od raspoloživih pravila gramatike. Kada se takav podniz pronađe vrši se njegova redukcija na levu stranu primenjene smene i postupak nastavlja sve dok se niz ne redukuje na startni simbol. Kao i kod osnovnog algoritma za Top-down analizu i ovde ima vraćanja unatrag i poništavanja primenjenih smena u slučaju kada dođe do greške i ne može da se ide dalje sa analizom.

Pri realizaciji ovog algoritma koristi se magacin u koji se ubacuje slovo po slovo iz ulaznog niza. Pri tome se u svakom koraku ispituje da li je moguće izvršiti redukciju niza koji je u vrhu magacina. Ako redukacija nije moguća u magacin se ubacuje novo slovo, a ako je moguća prepoznati niz sa vrha magacina se redukuje na neterminal koji je na levoj strani primenjene smene, odnosno neterminal zamenjuje redukovani niz u magacinu. Analiza je uspešno izvršena kada se dođe do graničnog simbola a u magacinu ostane samo startni simbol.

# Dati definiciju proste LL-1 gramatike.

Proste LL-1 gramatike su beskonteksne gramatike u kojoj sve smene za isti neterminalni simbol započinju različitim terminalnim simbolima:

***Primer 6.1 Prosta LL-1 gramatika***

Gramatika **G**=({*S*, *A*}, {*a*, *b*, *c*, *d*}, *S* , **P**}, gde je **P** sledeći skup pravila:

# Kako je definisana Sintaksna tabela proste LL-1 gramatike?

U opštem slučaju Sintaksna tabela se sastoji od onoliko kolona koliko ima terminalnih simbola, a onoliko vrsta koliko ima neterminalnih i terminalnih simbola zajedno. Polja tablice sintaksne analize se popunjavaju tako da se u polju koje odgovara neterminalu *A*, i terminalu *a,* upisuje smena koja na levoj strani ima neterminal *A*, a na desnoj strani reč koja počinje terminalom *a*, ako takva smena postoji. U preseku vrste koja je označena terminalom i kolone koja je označena istim tip terminalom upisuje se vrednost *pop*, dok se u preseku vrste i kolone koje su označene graničnim simbolom # upisuje vrednost *accept*. Sva ostala polja su polja greške. Sintaksna tabela T(*A*, *a*), gde je A oznaka vrste, a *a* oznaka kolone se formalno može definisati na sledeći način:



# Objasniti algoritam za sintaksnu analizu proste LL-1 gramatike.

Za razliku od osnovnog algoritma za *Top-down* analizu kod kojeg se pravila biraju na osnovu prvog neterminalnog simbola sa leve strane u izvedenom nizu kod ovih analizatora odluka o pravilu koje će da se primeni se donosi na osnovu slova u ulaznom nizu koje treba da se prepozna, na kome je trenutno ulazni pokazivač. Primenjuje se pravilo koje će sigurno da generiše bar to slovo. Ovakvi analizatori su poznati kao LL-1 analizatori (*Look Left* 1), gde cifra 1 ukazuje na to da se predikcija vrši na osnovu jednog slova u ulaznom nizu. Generalno gledano mogu da se definišu i LL-k analizatori ali su LL-1 mnogo upotrebljiviji.

# Objasniti pojam leve faktorizacije.

Pokazali smo da se u slučaju LL-1 gramatika može vršiti prediktivna sintaksna analiza koja je mnogo efikasnija od osnovnog algoritma za sintaksnu analizu. Veoma značajno u svemu tome je da se gramatike koje ne zadovoljavaju uslov LL-1 gramatika mogu jednostavno transformisati u ovaj tip gramatika. Naime, ukoliko u gramatici postoji veći broj pravila za isti neterminalni simbol koja na desnoj strani imaju reči sa istim prefiksom, odnosno smene oblika:



gde su  i ,

takva gramatika sigurno nije LL-1 gramatika. Međutim ona se transformiše u LL-1 gramatiku sledećom smenom:

# Dati definiciju funkije FIRST. \*

Ukoliko postoji smena oblika *X* → α, definiše se funkcija FIRST(α) koja sadrži sve terminalne simbole koji mogu da se nađu na početku reči izvedenih iz niza α, tj:



# Dati definiciju funkcije FOLLOW. \*

Funkcija FOLLOW se definiše za neterminalne simbola, kao skup terminalnih simbola koji mogu u toku izvođenja da se nađu iza tog neterminalnog simbola, ili:



Po definiciji FOLLOW funkcija startnog simbola gramatike sadrži granični simbol #.

Za odredjivanje FOLLOW funkcije neterminalnog simbola *X* posmatraju se desne strane pravila u kojima se pojavljuje posmatrani simbol. Pri tome simbol *X* da desnoj strani smene može da se nađe u jednom od sledećih konteksta:

*Z* → α *X* *x*β ∧ *x*∈**Vt** ⇒ *x*∈FOLLOW(*X*)

*Z* → α *XY*β ∧ *Y*∈**Vn** ⇒ FIRST(*Y*)⊂FOLLOW(*X*)

# Dati definiciju LL-1 gramatika bez pravila.

Beskonteksna gramatika bez pravila je LL-1 gramatika ako su za sva pravila oblika:

 skupovi disjunktni po parovima, odnsno:

## 

## Formalna definicja LL1 sa e \*

Beskonteksna gramatika koja sadrži i pravila je LL-1 gramatika ako i samo ako za sva pravila oblika pravila  i  važi:

****

****

# Kako se popunjava Sintaksna tabela kod LL-1 gramatika bez pravila?

Sintaksna tabela za ovako definisane LL-1 gramatike definisana je sa:



# Kako se popunjava Sintaksna tabela kod LL-1 gramatika sa pravilima? \*

Sintaksna tabela kod LL-1 gramatika sa  pravilima je nešto modifikovana u odnosu na prethodne definicije. Naime sada je problem gde u tablici treba smestiti  pravila. Zbog toga je Sintaksna tabela definisana na sledeći način:

Prema ovoj definiciji sledi da se  pravila upisuju u kolone terminalnih simbola koji pripadaju funkciji FOLLOW za neterminalni simbol koji se preslikava u .

# Dati definiciju operatorskih gramatika.

Operatorske gramatike su beskonteksne gramatike kod kojih ne postoje pravila u kojima se u reči na desnoj strani javljaju dva neterminalna simbola jedan do drugog, odnosno ne postoje pravila oblika:

*C* → **p***AB***q,** gde je: *A*,*B*,*C*∈**Vn** i **p,q****V+**

# Definisati osnovne relacije prvenstva između terminalnih simbola.

Postoje tri osnovne relacije i one su definisane na sledeći način:

**Relacija istog prioriteta** *a**b*

*Terminalni simboli a i b su u relaciji istog prioriteta* ako i samo ako postoji smena oblika



Ako se to predstavi na sintaksnom stablu onda znači da se terminalni simboli a i b pojavljuju na istom nivou sintaksnog stabla, slika 7.1.



Slika 7.1 Relacija istog prioriteta

**Relacija nižeg prioriteta** 

Terminalni simbol *a* je nižeg prioriteta u odnosu na terminalni simbol *b*, ako i samo ako postoji smena oblika:



Predstavljeno na sintaksnom stablu to znači da se terminalni simbol *a* pojavljuje levo i na višem nivou u odnosu na terminalni simbol *b.*

, 

**Relacija višeg prioriteta** 

Terminalni simbol *a* je višeg prioriteta u odnosu na terminalni simbol *b*, ako i samo ako postoji smena oblika:



Posmatrano na sintaksnom stablu to znači da se terminalni simbol a pojavljuje levo od terminalnog simbola b ali na nekom od nižem nivou, Slika 7.3.

# Dati definiciju Operatorske gramatike prvenstva.

Operatorska gramatika u kojoj za svaka dva terminalna simbola važi najviše jedna relacija prvenstva naziva se *Operatorska gramatika prvenstva*.

Kod ovih gramatika moguće je primeniti algoritam za Bottom-up analizu u kome neće biti povratnih petlji.

# Šta je to Operatorska tablica prvenstva? Dati primer.

U postupku sintaksne analize operatorskih gramatika prvenstva koristi se pomoćna sintaksna tabela (operatorska tablica prvenstva). To je tablica sa onoliko vrsta i kolona koliko je terminalnih simbola azbuke (uključujući i granični simbol). Elementi operatorske tablice prvenstva su relacije prvenstva, pri čemu se simbol sa leve strane relacije uzima kao oznaka vrste, a sa desne strane relacije kao oznaka kolone.

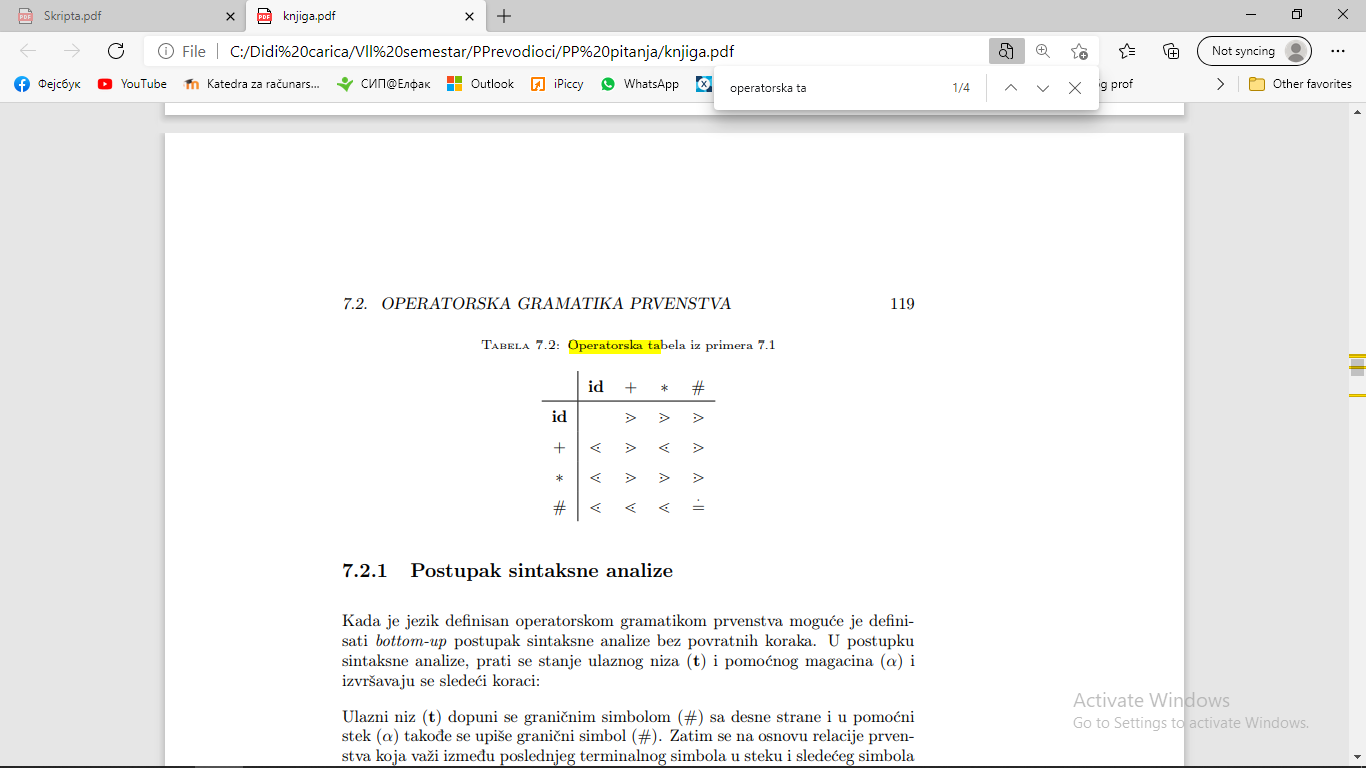
Primer: Za operatorsku gramatiku prvenstva definisanu skupom smena

E → E + T | T

T → T ∗ F | F

F → id

odrediti tablicu prvenstva.



# Šta su to funkcije prvenstva i kako se određuju?

Memorija potrebna za pamćenje tablice prvenstva je (*n*+1)⋅(*n*+1) (gde je *n* broj terminalnih simbola gramatike). Kod gramatika sa mnogo terminalnih simbola ova tablica može da bude veoma velika i da zahteva mnogo memorijskog prostora. Zato se umesto tablice prvenstva, češće u memoriji pamte funkcije prvenstva. Za svaki terminalni simbol gramatike definišu se po dve celobrojne funkcije: *f* i *g* sa sledećim osobinama:

*a*<⋅*b* ⇔ *f*(*a*) < *g*(*b*)

*a**b* ⇔ *f*(*a*) = *g*(*b*)

*a*⋅>*b* ⇔ *f*(*a*) > *g*(*b*)

Memorijski prostor potreban za pamćenje funkcija prvenstva je 2⋅(*n*+1).

Za određivanje funkcije prvenstva formira se orijentisani graf koji ima 2*n* čvorova. Potezi u grafu se određuju na sledeći način:

Ako važi relacija *a*⋅>*b*, tada postoji poteg od čvora *f*(*a*) prema čvoru *g*(*b*).

Ako važi relacija *a*<⋅*b*, tada postoji poteg od čvora *g*(*b*) prema čvoru *f*(*a*).

Ako važi relacija *a**b*, tada postoje potezi od čvora *f*(*a*) prema čvoru *g*(*b*) i od čvora *g*(*b*) prema čvoru *f*(*a*).

Vrednost funkcije prvenstva se odredjuje kao dužina najdužeg puta koji polazi iz čvora koji odgovara toj funkciji. Pod dužinom puta podrazumeva se broj čvorova kroz koje put prolazi uključujući sam taj čvor.

# Opisati algoritam za sintaksnu analizu Operatorskih gramatika prvenstva. \*

Ulazni niz (**t**) dopuni se graničnim simbolom (#) sa desne strane i u pomoćni stek (α) takođe se upiše granični simbol (#). Zatim se na osnovu relacije prvenstva koja važi između poslednjeg terminalnog simbola u steku i sledećeg simbola u ulaznom nizu određuje akcija koja će se izvršiti.

1. Ako važi relacija <⋅ ili relacija , tekući terminalni simbol iz ulaznog niza se upisuje u stek α i čita se novi simbol iz ulaznog niza. Uz terminalni simbol upisuje se i oznaka relacije prvenstva koju ima sa prethodnim simbolom.
2. Ako važi relacija ⋅> treba ići dublje u stek i tražiti prvi terminal u steku koji je <• od svog sledbenika u steku i proveriti da li fraza sa vrha steka (od uočenog simbola do vrha, fraza između simbola <• i •>) predstavlja desnu stranu neke smene. Ako takva smena postoji, treba izvršiti redukciju, tj. umesto uočene fraze u stek ubaciti uniformnu oznaku neterminalnog simbola (*P*), u suprotnom u zapisu postoji greška i dalju analizu treba prekinuti. Kada se redukcija uspešno izvede treba ispitati da li je sledeći ulazni simbol ‘#’, a sadržaj steka ‘#p’. U tom slučaju ceo ulazni niz redukovan, tj. prepoznat.

Osnovna ideja kod ovog postupka sinteze je da se identifikuju fraze koje su višeg prioriteta u odnosu na okruženje i da se one prve redukuju. Ukoliko je fraza niže u sintaksnom stablu ona je višeg prioriteta redukuje se pre simbola iz okruženja.

Ako ne postoji ni jedna od navedenih relacija znači da u zapisu postoji greška i dalju analizu treba prekinuti.

# Dati definiciju Gramatika prvenstva.

Beskonteksna gramatika kod koje se između bilo koja dva simbola može postaviti najviše jedna relacija prvenstva naziva se Gramatikom prvenstva.

# Šta je to Tablica prvenstva i kako se popunjava.

Za svaku gramatiku prvenstva može se generisati jednoznačno popunjena tablica prvenstva u koju se upisuju relacije prvenstva. Ova tablica ima onoliko vrsta i kolona koliko gramatika ima ukupno simbola i terminalnih i neterminalnih. Tablica prvenstva koristi se u postupku sintaksne analize na isti način kao i kod Operatorskih gramatika prvenstva.

# Opisati algoritam za sintaksnu analizu Gramatika prvenstva. \*

Ulazni niz (t) dopuni se graničnim simbolom (#) sa desne strane i u pomoćni stek (α) smesti se isti taj simbol. Zatim se na osnovu relacije prvenstva koja važi između poslednjeg simbola u steku i sledećeg simbola u ulaznom nizu definiše odredjuje akcija koja će da se izvrši.

1. Ako važi relacija <⋅ ili relacija , tekući element iz ulaznog niza smestiti u stek α i preći na analizu sledećeg simbola.
2. Ako važi relacija ⋅> treba ići dublje u stek i tražiti prvi simbol u steku koji je <⋅ od svog sledbenika i proveriti da li fraza sa vrha steka (sadržaj magacina od uočenog simbola do vrha) predstavlja desnu stranau neke smene. Ako jeste, umesto cele fraze u stek treba ubaciti simbol sa leve strane te smene (tj. izvršiti redukciju), u suprotnom, u zapisu postoji greška i dalju analizu treba prekinuti. Kada se redukcija uspešno izvede treba ispitati da li je sledeći ulazni simbol ‘#’, a sadržaj steka ‘#S’ (gde je S startni simbol gramatike). Ako su ti uslovi zadovoljeni, znači da je ceo ulazni niz redukovan na startni simbol gramatike, tj. da je on prepoznat.
3. Ako ne postoji nijedna od navedenih relacija znači da u zapisu postoji greška i dalju analizu treba prekinuti.

# Objasniti strukturu LR analizatora.

Šema LR analizatora prikazana je na Slici 8.1. U suštini to je automat koji kao radnu memoriju koristi magacin.



Slika 8.1 LR sintaksni analizator

Simbol na vrhu radnog magacina je trenutno stanje LR sintaksnog analizatora i on sumira informacije smeštene u stek pre njega.

# Objasniti strukturu LR tablice sintaksne analize.

Postupak sintaksne analize je upravljan LR sintaksnom tabelom. Ona se sastoji od: akcija i prelaza. Ova sintaksna tabela ima onoliko vrsta koliko je mogućih stanja sintaksnog analizatora. U delu za akcije postoji onoliko kolona koliko je terminalnih simbola gramatike (uključujući i granični simbol), a u delu za prelaze broj kolona jednak je broju neterminalnih simbola gramatike. U delu za akcije definisani su postupci koji će se izvršiti zavisno od stanja koje se nalazi na vrhu radnog magacina i tekućeg simbola u ulaznom nizu.

# Objasniti postupak LR analize.

Akcija može biti:

***- sk*** (*shift* k) - znači da u magacin treba smestiti tekući simbol iz ulaznog niza i naredno stanje (*k*) i preći na analizu sledećeg simbola iz ulaznog niza;

- ***rk*** (*reduce* *k*) - znači da treba izvršiti redukciju po smeni *k* ( *k*: A→β ) - odnosno, iz magacina treba izbaciti 2xlength(β) elemenata, u magacin smestiti neterminalni simbol sa leve strane smene k (A), a tekuće stanje odrediti kao prelaz iz prethodnog stanja u magacinu pod dejstvom tog neterminalnog simbola (A).

- ***acc*** (*accept*) - znači da je niz prepoznat i dalju analizu treba prekinuti;

- ***err*** (*error*) - znači da u ulaznom nizu postoji greška i teba prekinuti dalju analizu.

LR analizatori su u suštini i kao *Shift-Reduce* analizatori, što znači da se kroz ulazni niz prolazi sleva u desno i nastoji se da se izvrši redukcija (*reduce*), ako redukcija nije moguća prelazi se na sledeći znak u nizu (*shift*). Za razliku od osnovnog algoritma za *Bottom-up* analizu koji je praktično *brut force* algoritam, kod ovih analizatora se analizom smena gramatike generiše LR sintaksna tabela koja određuje promenu stanja u automatu u zavisnosti od tekućeg ulaznog simbola.

Algoritam:

Postaviti ulazni pokazivač *ip* na početak niza w#, koji se analizira.

repeat forever

begin

Neka je *q* znak u vrhu magacina i *a* znak na koji pokazuje ulazni pokazivač *ip*.

if *action*(*q*, *a*) = *sift* *k* then

begin

Ubaciti u magacin *a*, a zatim i *k* i pomeriti ulazni pokazivač *ip* za jedno mesto udesno.

end

else if *action*(*s*, *a*) = *reduce* *k*, pri čemu je *k*-ta smena gramatike:

****A**  then

Izbaciti 2\*|**| simbola iz steka, ubaciti *A* u stek, a zatim i oznaku stanja koja se dobija na osnovu goto(s’, *A*), gde je s’ oznaka stanja u koje smo se vratili posle redukcije . Pri tome se generiše i izlaz A**.

end

else if *action*(*s*, #) = *accept* then

return

else *error*()

end

# Šta su to LR(0) članovi?

Posmatrajmo jedan korak u desnom izvođenju nekog niza: *B***t** **t**, gde je *B* prvi neterminal sa desne strane koji se preslikava u niz b primenom pravila *B*.Оčigledno je niz **t** sastavljen od terminalnih simbola.

Gramatika **G** je LR(k) gramatika ako se za bilo koji ulazni niz, u svakom koraku izvođenja fraza (podniz)  može detektovati na osnovu prefiksa  i skaniranjem najviše *k* znakova niza **t**.

# Šta je to kanonička kolekcija LR(0) članova?

Izvođenje sa tačkom u bilo kojoj poziciji u nizu se naziva LR(0) član.

LR(0) članovi se generišu na osnovu pravila gramatike i svako pravilo gramatike daje nekoliko LR(0) članova u zavisnosti od dužine reči na desnoj strani pravila.

|  |
| --- |
| ***Primer 8.3 LR(0) članovi***  Za gramatiku iz primera 8.1 na osnovu pravila  mogu se generisati sledeći LR(0) članovi: |

LR(0) članovi će se u postupku sinteze LR analizatora koristiti da se prate vidljivi prefiksi koji su uzeti u obzir. Naime tačka praktično razdvaja vidljive prefikse od neprepoznatog dela reči.

# Algoritam za LR sintaksnu analizu

Sinteza LR analizatora se svodi na generisanje LR sintaksne tabele, pri čemo se razlikuju sledeće tri faze:

1. Određivanje kanoničkog skupa LR članova kojim će biti odredjeni svi vidljivi prefiksi reči jezika koji je opisan gramatikom. Određivanje kanoničkog skupa kreće od LR(0) člana koji se formira na osnovu fiktivne smene , gde je *S* startni simbol gramatike. Nakon formiranja ovog početnog LR(0) člana primenjuju se funkcije *closer* i *goto* sve dok je to moguće. Rezultat ovog koraka je kanonički skup zatvaranja LR(0) članova K={**I0, I1, ..., Ik}.**
2. Crtanje grafa prelaza konačnog automata za prepoznavanje vidljivih prefiksa pri čemu važe sledeća pravila:
   1. Svakom zatvaranju iz kanonickog skupa pravila K={**I0, I1, ..., Ik}.**dodeljuje se jedno stanje u grafu automata.
   2. Potezi u grafu određeni su *goto* funkcijama iz kanoničkog skupa LR(0) pri čemu ako je *goto*(**Ii**, *X*) = **I**j onda postoji odlazni poteg iz stanja Ii do stanja Ij za slovo *X,* gde je *.*
   3. Sva stanja automata koja odgovaraju zatvaranjima LR(0) članova kojima pripadaju članovi oblika su završna stanja (stanja redukcije za pravila .
   4. Stanje kome pripada LR(0) član , nastao na osnovu fiktivne smene je početno stanje automata.
3. Popunjavanje LR sintaksne tabele koje se vrši primenom sledećih pravila: \*







# Objasniti mesto i namenu međukoda u strukturi kompilatora.

*Sintaksni analizator*

*Generator međukoda*

*Generator koda*

*Međukod*

Slika 10.1 Mesto međukoda u okviru kompilatora

Kompilator može da koristi i međukod u više nivoa, kako je to predstavljeno na Slici 10.2.

*Izvorni kod*

*Međukod visokog nivoa*

*Ciljni kod*

*Međukod*

*niskog nivoa nivoa*

Slika 10.2 Međukod u više nivoa

Postoji više razloga zbog kojih se uvodi međukod, od kojih su najznačajniji:

1. Prenosivost koda. Na osnovu međukoda može se generisati kod za različite ciljne mašine
2. Na nivou međukoda može da se vrši mašinski nezavisna optimizacija

# Navesti osnovne vrste međukoda.

Međukod može da bude generisan u različitim oblicima. Ovde ćemo razmotriti sintaksno stablo (ili usmereni dijagram) i troadresni međukod kao dva najšire rasprostranjena oblika međukoda. Nekada se kao međukod koristila i Poljska inverzna notacija koja je pogodna za interpretiranje koda i generisanje mašinskog koda korišćenjem steka. Veoma često se kao međukod koristi i programski jezik C zato što se naredbe ovog jezika postavljene dosta blizu asemblerskom jeziku i lako se transformišu u asemblerski jezik.

# Dati primer sintaksnog stabla.

Sintaksno stablo je struktura koja se generiše na osnovu stabla sintaksne analize i na neki način predstavlja redukovanu reprezentaciju stabla sintaksne analize. Obično se generiše direktno u toku same sintaksne analize ili se najpre generiše Stablo sintaksne analize (Parse tree) pa ta struktura transformiše u sintaksno stablo. U Tabeli 10.1 predstavljena su pravila gramatike kojom se definišu aritmetički izrazi i semantička pravila koja se pridrućuju ovim pravilima, koja se koriste za generisanje sintaksnog stabla. U slučaju Bottom up analize, semantička pravila se primenjuju posle redukcije po odgovarajučem pravilu gramatike.

:=

+

\*

\*

~

~

b

b

c

c

*a*

:=

+

\*

~

b

c

*a*

Slika 10.3 Sintaksno stablo i usmeren dijagram za naredbu a := b \* -c + b \* -c

# Šta je to troadresni medjukod.

Troadresni međukod je neka vrsta pseudoasemblerskog jezika koji se može lako preslikati u drugi asemblerski jezik ili mašinski kod. Sastoji se od jednostavnih naredbi koje mogu da imaju najviše jedan operator tako da se složeni izrazi predstavljaju sekvencom naredbi. Na primer izraz x+y\*z može da se predstavi sledećom sekvencom troadresnih instrukcija.

t1= y \* z

t2 = t1 + x

gde su t1 i t2, privremene (temporalne) promenljive koje generiše kompilator. Takođe, troadresni kod sadrži i jednostavne upravljačke naredbe kao što su bezuslvni skokovi i uslovni skokovi koji se jednostavno preslikavaju u asemblerske naredbe.

# Objasniti strukturu *quadrupils* i dati primer.

U slučaju reprezentacije pomoću kvadrapila (quadruples) koriste se slogovi od četri polja koja su namenjena smeštaju operatora (*op)* , prvog argumenta (*arg1*), drugog argumenta (*arg2*) i rezultata (*rez*). Na primer naredba x= y + z bi imala operator +, prvi argument y, drugi argument z i rezultat x.

Postoje neka odstupanja od navedenih pravila:

Naredbe sa unarnim operatorima kao što su x= minus y ili x := y ne koriste polje *arg2*. U slucaju kopi naredbe operator je :=, dok se kod svih drugih naredbi ovaj operator implicitno podrazumeva.

Kod uslovnih i bezuslovnih naredbi skoka labela se upisuje u polje *rez*.

***Primer 10.3 Quadrupiles***

Troadresni kod iz proimera 10.2 kojim je predstavljena naredba dodeljivanja a := b \* -c + b \* -c biće memorisan preko quadrupila na sledeći način:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *op* | *arg1* | *arg2* | *rez* |
| 0 | minus | C |  | t1 |
| 1 | \* | B | t1 | t2 |
| 2 | minus | C |  | t3 |
| 3 | \* | B | t3 | t4 |
| 4 | + | t2 | t4 | t5 |
| 5 | = | t5 |  | a |
|  |  |  |  |  |

# Objasniti strukturu *triples* i dati primer.

Kod reprezentacije troadresnog međukoda pomoću Triple strukture koriste se slogovi sa tri polja. Uočimo da se kod reprezentacije pomoću quadruple slogova polje rez uglavnom koristi za smeštaj temporalnih promenljivih u kojima se čuvaju međurezultati. U slučaju triples strukture na temporalne promenljive se ukazuje preko pozicije odgovarajućeg sloga umesto preko rednog broja temporalne promenljive. To se dobro vidi iz primera 10.4 gde je predstavljena triple struktura za međukod iz primera 10.2.

Troadresni kod iz proimera 10.2 kojim je predstavljena naredba dodeljivanja a := b \* -c + b \* -c biće memorisan preko triples slogova na sledeći način:

|  |  |  |  |
| --- | --- | --- | --- |
|  | *op* | *arg1* | *arg2* |
| 0 | minus | c |  |
| 1 | \* | b | (0) |
| 2 | minus | c |  |
| 3 | \* | b | (2) |
| 4 | + | (1) | (3) |
| 5 | = | a | (4) |
|  |  |  |  |

# Objasniti strukturu *Indirect triples* i dati primer.

U slučaju strukture Indirect triples koristi se i dodatna lista pointera na naredbe koje su predstavljene preko triples slogova. Ovo može da bude pogodno kod implementacije različitih algoritama za optimizaciju koda kada se vrši preuređivanje naredbi. U tom slučaju razmeštanja se vrše samo u listi pointera asama struktura sa slogovima naredbi ostaje ista.

***Primer 10.4 Indirect triples***

Troadresni kod iz proimera 10.2 kojim je predstavljena naredba dodeljivanja a := b \* -c + b \* -c biće memorisan preko Indirect triples strukture na sledeći način:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | *Narede* |  |  | *op* | *arg1* | *arg2* |
| 35 | (0) |  | 0 | minus | c |  |
| 36 | (1) |  | 1 | \* | b | (0) |
| 37 | (2) |  | 2 | minus | c |  |
| 38 | (3) |  | 3 | \* | b | (2) |
| 39 | (4) |  | 4 | + | (1) | (3) |
| 40 | (5) |  | 5 | = | a | (4) |
|  | ... |  |  | ... |  |  |

# Navesti osnovne naredbe troadresnog međukoda.

Lista osnovnih naredbi troadresnog koda se sastoji od sledećih naredbi:

Naredba dodeljivanja x := y *op* z, gde je *op* aritmetički ili logički binarni operator.

Naredba dodeljivanja x := *op* y, gde je *op* aritmetički ili logički unarni operator (npr. minus, logička negacija, shift operator, operatori za konverziju tipova i sl. )

Naredba kopiranja x := y kojom x dobija vrednost y.

Naredba bezuslovnog skoka go to L.

Naredba uslovnog skoka if x *relop* y goto L.

param x i call p,n za poziv potprograma i *return y* za povratak iz potprograma. Primer:

param x1

param x2

....

param xn

call p,n

Indeksirana dodeljivanja u obliku: x := y [ i ] i x [ i ] : = y, gde naredba x := y [ i ] znači da će sadržaj memorijske lokacije koja y+i biti preslikan u memorijsku lokaciju x, dok naredba x [ i ] : = y znači da će sadržaj memorijske lokacije x+i biti preslikan u memorijsku lokaciju y.

Dodeljivanje adresa i pointera u obliku: x := &y , x:= \*y i \*x:=y.

# Dati primer semantičkih pravila kojima se generiše troadresni međukod za izraze.

U tabeli 10.2 data su pravila gramatike i pridružena semantička pravila kojima se generiše troadresni mešukod aritmetičkih izraza. Uočimo da se semantičkim pravilima generišu atribut *code* i atribut *addr* neterminala *S* i neterminala *E*. Atributi *S.code* i *E.code* označavaju troadresni kod za *S* i *E*, respektivno, dok *E.addr* označava adresu memorijske lokacije gde se čuva vrednost za *E*.

|  |  |
| --- | --- |
| Sid = E; | **S.code = E.code ||**  *gen(top.get(***id***.lexeme) ’=’ E.addr)* |
| EE1 + E2 | **E.addr =** *new* **Temp ()**  *E.code = E1.cod || E2.code||*  *gen(E.addr ’=’ E1.addr ’+’ E2.addr)* |
| E-E1 | **E.addr =** *new* **Temp ()**  *E.code = E1.cod ||*  **gen(E.addr ’=’ ’minus’ E1.addr)** |
| E(E1) | **E.addr = E1.addr**  *E.code = E1.code* |
| Eid | ***E.addr* = top.get(id.lexeme)**  E.code = ’’ |

# Dati primer semantičkih pravila kojima se generiše troadresni međukod za promenljive sa indeksima.

***Primer 10.5 Međukod za promenljive sa indeksima***

Razmotrićemo generisanje troadresnog međukoda za izraz c + a[i][j]. Naka je a matrica celobrojna dimenzija, i neka su c, i i j, celobrojne promenljive. Tip za a je određen sa *array(2,* *array(3,integer).* Širina matrice a je *w*=24, i pretpostavićemo da je width za celobrojne vrednosti 4. Tip za a[i] je *array(3,integer*), širine *w1*=12. Tip za a[i][j] je *integer*.

Anotirano stablo sintaksne analize za izraz c + a[i][j] koje prikazuje dodeljivanje vrednosti atributima prikazano je na Slici 10.8. Primenom semantičkih pravila biće generisan sledeći troadresni međukod:

t1 = i \* 12

t2 = j \* 4

t3 = t1 + t2

t4 = a [ t3 ]

t5 = c + t4

# Dati primer semantičkih pravila kojima se generiđe troadresni kod osnovnih upravljačkih struktura.

|  |  |
| --- | --- |
| **PRAVILA GRAMATIKE** | **SEMANTIČKA PRAVILA** |
| *PS* | *S.next = newlabel ()*  *P.code = S.code || label(S.next)* |
| *S***assign** | *S.code* ***=* assign.***code* |
| *S***if** ( *B* ) *S1* | *B.true = newlabel()*  *B.false = S1.next = S. next*  *S.code = B.code || label (B.true) || S1.code* |
| *S* **if** ( *B* ) *S1* **else** *S2* | *B.true = newlabel()*  *B.false = newlabel()*  *S1.next = S2.next = S.next*  *S.code = B.code*  *|| label(B.true) || S1.code*  *||gen(***'goto'** *S.next)*  *|| label (B.false) || S2.code* |
| *S* **while** ( *B* ) *S1* | *begin = newlabel()*  *B.true = newlabel()*  *B.folse = S.next*  *S1.next = begin*  *S.code =label(begin) || B.code*  *|| label(B.true)|| S1.code*  *|| gen(***'goto'** *begin)* |
| *S S1 S2* | *S1.next = newlabel()*  *S1.next = S.next*  *S.code = S1.code || label(S1.next) || S2.code* |

# 

# Dati primer semantičkih pravila kojima se generiše troadresni mešukod za Boolove izraze.

|  |  |
| --- | --- |
| **PRAVILA GRAMATIKE** | **SEMANTIČKA PRAVILA** |
| *BB1 || B2* | *B1.true = B.true*  *B1.false = newlabel()*  *B2.true = B.true*  *B2.false = B.false*  *B.code = B1.code || lanel(B1.false) || B2.code* |
| *BB1 && B2* | *B1.true=mewlabel*  *B1.false = B.false*  *B2.true = B.true*  *B2.false – B.false*  *B.code = B1.code || lanel(B1.true) || B2.code* |
| *B! B1* | *B1.true = B.false*  *B1.false = B.true*  *B.code = B1.code* |
| *BE1* **rel** *E2* | *B.code = E1.code || E2.code*  *|| gde( '***if***' E1.addr* **rel***.op E2.addr '***goto***' B.true)*  *||gen('***goto***' B.false)* |
| *B***true** | *B.code = gen(****'goto'****, B.true)* |
| *B***false** | *B.code = gen(****'goto'****, B.false)* |

# Šta je zadatak modula za proveru tipova podataka.

Sastavni deo savremenih kompilatora su i moduli za proveru tipova podataka koji mogu različito da budu postavljeni u zavisnosti od toga kako je postavljen koncept tipova podataka u jeziku na koji se odnosi kompilator.

Naime, cilj je da se greške na nivou tipova podataka otkrivaju u fazi kompiliranja programa, a ne u fazi izvršavanja.

# Dati primer semantičkih pravila kojima se generišu atributi tipa identifikatora.

U Tabeli 10.6 data su semantička pravila kojima se generišu atributi tipa za identifikatore.

Tabela 10.6 Semantička pravila za tipove identifikatora

|  |  |
| --- | --- |
| **PRAVILA GRAMATIKE** | **SEMANTIČKA PRAVILA** |
| *P→ D;E* |  |
| *D →D;D* |  |
| *D →id:T* | *addType (id.entry, T.type)* |
| *T →char* | *T.type:=char* |
| *T →integer* | *T.type:=integer* |
| *T → ^T1* | *T.type:=pointer(T1.type)* |
| *T →array[num] of T1* | *T.type:=array(1..num.val, T1.type)* |

# Dati primer semantičkih pravila kojima se vrši provera kompatibilnosti tipova u izrazima.

|  |  |
| --- | --- |
| **PRAVILA GRAMATIKE** | **SEMANTIČKA PRAVILA** |
| *E →literal* | *E.type:=char* |
| *E →num* | *E.type:=integer* |
| *E →***id** | *E.type:=lookup(***id***.entry)* |
| *E →E1 mod E2* | *E.type:=***if** *E1.type=integer* **and** *E2.type=ineger* **then** *integer* **else** *type\_error* |
| *E →E1[E2]* | *E.type:=if E2.type=integer and E1.type=array(s,t) then t else error}*  *t je elementarni tip dobijen iz strukturnog tipa array(s,t)* |
| *E →E1^* | *E.type:=* **if** *E1.type=pointer(t)* **then** *t* **else** *type\_error* |
| *Pozivi funkcija:*  *E →E1(E2)* | *E.type:=* **if** *E2.type=s* **and** *E1.type=s →t* **then** *t else type\_error* |

# Dati primer semantičkih pravila kojima se vrši provera naredbi.

1. Tabela 10.7 Provera naredbi

|  |  |
| --- | --- |
| **PRAVILA GRAMATIKE** | **SEMANTIČKA PRAVILA** |
| *S →id:=E* | *S.type:=* **if** *id.type=E.type* **then** *void* **else** *type\_error* |
| *S →if E then S1* | *S.type:=* ***if*** *E.type=boolean* **then** *S1.type* **else** *type\_error* |
| *S →while E do S1* | *S.type:=* **if** *E.type=boolean* **then** *S1.type* **else** *error* |
| *S →S1;S2* | *S.type:=* **if** *S1.type=void* **and** *S2.type=void* **then** *void* **else** *type\_error* |

# Objasniti pojam statičke alokacije memorije i navesti sekvencu naredbi koju u tom slučaju generiše kompilator

Kod statičkih jezika se u toku kompiliranja programa generiše po jedan aktivacioni slog za svaki potprogram tako da se isti aktivacioni slog koristi kod svakog poziva potprograma. Ovaj aktivacioni slog se smešta u statički deo memorije. U ovom slučaju nema mogućnosti za rekurzivne pozive potprograma zato što bi svaki naredni poziv potprograma prebrisao sadržaj aktivacionog sloga koji odgovara prethodnom pozivu koji još nije završen.

## Razlika izmedju lekseme, tokena i sablona

* LEKSEMA – Ulazni niz koji se prepoznaje na osnovu formalnog opisa pomoću šablona i za koji se generiše određeni token.
* TOKEN – Izlazni simbol koji se generiše kada je prepoznat određeni ulazni niz znakova. Primer : Integer ,Ključna reč: else, or, if, begin ....
* ŠABLON (pattern) – Potreban nam je način da jednoznačno definišemo koji nizovi odgovaraju kom tokenu. Regularni izraz, formalni opis ulaznih nizova za koje se generiše odeđeni token.

## Definicija prioriteta i asocijativnosti operatora (presedance, levo/desno/neasocijativni)

Operatori mogu biti levo-asocijativni, desno-asocijativni i neasocijativni. Za svaki tip asocijativnosti postoji posebna Yacc definicja (%left, %right, %nonassoc). Prioritet operatora je definisan redosledom navođenja njihovih asocijativnosti i to tako što se uvek navode najpre asocijativnosti operatori najnižeg prioritera. Npr. asocijativnost i prioriteti osnovnih aritmetičkih operatora u yacc specifikaciji mogu biti definisani na sledeći način:

%left ‘+’, ‘-’

%left ‘\*’, ‘/’

%right STEPEN

Operatori, koji su navedeni u definicjama asocijativnosti i prioriteta, ne moraju biti navedeni u listi tokena.

## JFlex pravila (Lex)

I pravila u jflex specifikaciji se definišu na isti način kao i u lex-u. Jedina razlika u tome je što akcija mora da bude navedena kao blok naredbi (izmedju zagrada {} i kada ona sdrži samo jednu java naredbu).

Centralni deo Lex specifikacije su pravila i to je jedini obavezni deo specifikacije. Minimalni sadržaj Lex specifikacije je:

%%

<pravila>

Jedno pravilo Lex specifikacije sadrži dva dela:

1. Uzorak – regularni izraz koji definiše sekvencu simbola koja se traži u ulaznom tekstu,
2. Akciju – kod na odredišnom jeziku (na programskom jeziku na kojem alat generiše izlazni kod) koji će se izvršiti kada se navedeni uzorak prepozna. Akcija se od uzorka odvaja bar jednim belim simbolom (blanko znakom, tabulatorom, ili prelazom na novi red).

Kada se Lex koristi za generisanje leksičkog analizatora, uzorcima se definišu regularni izrazi reči (leksema) programskog jezika za koji se analizator pravi, a akcija se obično završava *return* naredbom koja vraća token koji odgovara prepoznatoj reči.

U regularnim izrazima za defininisanje uzorka mogu da se koriste tekst simboli (koji se baš takvi kakvi jesu porede sa znacima iz ulaznog teksta) i metasimboli koji imaju svoje specijalno značenje.

## Metakarakteri za regularne izraze u jflex-u (Lex)

|  |  |
| --- | --- |
| Znak | Značenje |
| + | Prethodni znak ili grupa se ponavlja jednom ili više puta. |
| \* | Prethodni znak ili grupa se ponavlja nijednom ili više puta. |
| ? | Prethodni znak može ali i ne mora da se pojavi. |
| {n} | Prethodnji znak ili grupa se ponavlja tačno n puta. |
| {n,m} | Prethodni znak ili grupa se ponavlja minimalno n, a maksimalno m puta |
| {name} | Poziv makroa (ime lex promenljive) |
| [ ] | Alternativa – izbor jednog od navedenih simbola unutar zagrade |
| - | Sa prethodnim i narednim znakom definiše opseg simbola - može se koristiti samo unutar alternative |
| ^ | Koristi se na početku alternative ili na početku regularniog izrara. Ukoliko se nalazi na početku alternative, označava negaciju skupa znakova koji sledi – vrši se izbor jednog od simbola koji nisu navedeni u alternativi. Ukoliko se nalazi na početku regularnog izraza, označava da se taj regularni izraz primenjuje samo za izdvajanje podstringova (particija) sa početka linije. |
| $ | Koristi se na kraju regularnog izraza i označava da se taj regularni izraz primenjuje samo za prepoznavanje podnizova koji se nalaze na kraju linije. |
| / | Označava da se podnizovi definisani regularnim izrazom koji prethodi izdvajaju samo ukoliko ukoliko se iza njih nađe podniz definisan regularnim izrazom koji sledi. |
| ( ) | Grupisanje simbola. |
| \ | Poništava specijalno značenje metakaraktera koji sledi. |
| “ “ | Poništavaju specijalna dejstva svih metasimbola između. |
| | | Izbor alternative (bira se između simbola (ili grupe) koji prethodi i simbola (ili grupe) koji sledi. |
| ! | Negira znak koji sledi - na toj poziciji se može pojaviti svaki znak osim navedenog. |
| ~ | Prihvata sve simbole do pojave znaka ili grupe koja sledi, uključujući i taj znak/grupu. |
| . | Zamenjuje bilo koji znak. |
| <name> | Koristi se na početku regularnog izraraza i označava da se primenjuje samo ukoliko prepoznavanje podniza polčinje od navedenog stanja. |

## Definicija specijalnih pocetnih stanja u jflex

Prilikom prepoznavanja svake nove reči, polazi se od unapred definisanog početnog stanja konačnog automata. Po default-u, početno stanje generisanog konačnog automata je stanje koje ima redni broj 0. Nekada se javlja situacija da iste reči imaju različita značenja zavisno od konteksta u kojem su upotrebljene. Da bi se ta specijalna značenja pojedinih reči prepoznala uvode se nova početna stanja konačnog automata koji prepoznaje reči jezika.

Specijalna početna stanja mogu da budu:

* Potpuno definisana specijalna početna stanja – za njih treba definisati prelaze za svaki simbol ulazne azbuke. U engleskoj literaturi se ovakva stanja nazivaju *exclusive states*. Simbolička imena takvih stanja se u lex specifikaciji navode u liniji koja počinje sa %x.
* Parcijalno definisana specijalna početna stanja – za njih se mogu definisati samo prelazi za one simbole koje se nalaze na početku reči koje imaju drugačije značenje od onog koje imaju kada analiza počinje od početnog stanja 0. Za prepoznavanje onih reči koje imaju isto značenje i u jednom i u drugom slučaju koristiće se uzorci navedeni za *default*-no početno stanje. Za ovakva stanja se u engleskoj literaturi koristi termine incluseive states. Simbolička imena takvih stanja se u lex specifikaciji navode u liniji koja počinje sa %s.

Prelaz na korišćenje novog početnog stanja prilikom izdvajanja sledeće reči (podniza) koristi se funkcija begin.

## JFlex directive

Direktive kojima se podešavaju parametri generisane klase – leksičkog analizatora:

* + %class <ImeKlase>

Definiše ime generisane klase – ukoliko ova ocija nije navedena ime generisane klase je Yylex.

* + %implements <intrefejs1>[,<interfejs2>][,…]

Ukoliko je ova direktiva navedena generisana klasa implementira navedene interfejse.

* + %extends <ImeKlase>

Ukoliko je ova direktiva navedena generisana klasa nasledjuje navedenu klasu.

Direktive kojima se podešavaju parametri funkcije koja vrši izdvajanje sledeće reči iz ulaznog teksta:0

* + %function <ImeFunkcije>

Definiše ime generisane funkcije– ukoliko ova direktiva nije navedena ime generisane funkcije je yylex().

* + %int ili %integer

Postavlja povratni tip funkcije koja vrši leksičku analizu na tip int.

* + %type <ImeTipa>

Postavlja povratni tip funkcije koja vrši leksičku analizu na navedeni tip.

Ukoliko ni jedna od prethodne dve direktive nije navedena, podrazumenvani povratni tip leksičkog analizatora je Yytoken.

Direktive kojima se uključuju brojači linija i karaktera:

* + %line

Uključuje brojač linija u izvornom fajlu. Generiše se promenljiva yyline (tipa int) koja pamti redni broj tekuće linije (brojanje počinje od 0).

* + %char

Uključuje brojač karaktera u izvornom fajlu. Generiše se promenljiva yychar (tipa int) koja pamti redni broj prvog karaktera izdvojene reči (brojanje počinje od 0).

* + %column

Uključuje brojač karaktera u tekućoj liniji. Generiše se promenljiva yycolumn (tipa int) koja pamti redni broj prvog karaktera izdvojene reči u tekućoj liniji (brojanje počinje od 0).

Direktive kojima se nalaže generisanje samostalne aplikacije (tj. generisanje main metoda):

* + %debug

Generiše se main metod koji poziva leksički analizator sve dok se ne dodje do kraja ulaznog fajla. Na standardni izlaz (tj. u Java konzoli) ispisuju se svi izdvojeni tokeni kao i kod akcije koja je tom prilikom izvršena.

* + %standalone

Generiše se main metod koji poziva leksički analizator sve dok se ne dodje do kraja ulaznog fajla. Ispravno prepoznati tokeni se ignorišu, a na standardni izlaz se ispisuju neprepoznati delovi ulaznog fajla.

## Makroi u JFlexu

U jflex specifikaciji se makroi definišu na sledeći način:

<ImeMakroa> = <RegularniIzraz>

(Za razliku od lex specifikacije, ovde se nakon imena makroa navodi znak ’=’.)

## Definisanje gramatike u CUP specifikaciji \*

Sadrži (opciono) definiciju startnog simbola gramatike i (obavezno) definicje njenih smena.

Format definicija startnog simbola gramatike je:

start with <ImeSimbola>;

Ukoliko je definicija srtartnog simbola gramatike izostavljena, kao i u Yacc specifikaciji, podrazumeva se da je startni simbol gramatike simbol koji se nalazi na levoj strani prve navedene smene.

Smene gramatike se u Cup specifikaciji definišu parvilima koja se navode u formatu:

<LevaStrana> ::= <DesnaStrana> [ {: <Akcija> :} ];

Akcija je java kod koji se izvršava u trenutku kada se vrši redukcija po navedenoj smeni.

Više smena koje na levoj strani imaju isti neterminalni simbol se mogu skraćeno navoditi na isti način kao u Yacc specifikaciji.

Za oporavak od grešaka se, takodje, koristi ista notacija kao u Yacc specifikaciji (uvode se pomoćene smene sa error simbolom).

Ukoliko u akcijama treba koristiti dodatne atribute simbola koji se nalaze na desnoj strani smene, uvode se reference na te atribute na sledeći način:

<ImeSimbola> : <Ime>

<Ime> je u ovom slučaju referenca na član value objekta klase Symbol.

value atributu simbola koji se nalazi na levoj strani smene pristupa se korišćenjem reference RESULT

## Error handlovanje u sintaksnoj analizi u CUP

Sintaksni analizator generisan na osnvu specifikacije iz primera 9.11 će samo utvrditi da li je izraz korektno zapisan ili ne. U trenutku kada se u kodu pojavi prva greška, analiza će se prekinuti, biće pozvana funkcija yyerror koja će prikazati poruku “Syntax error”. Nedostatak ovog analizatora je što neće lokalizovati grešku, tj. neće nam dati informacije gde se ona nalazi, ni o kakvoj se grešci radi; a neće ni otkriti sve greške u kodu. Cilj analizatora je da prijavi sve greške i da korisniku ukaže gde se nalaze i kako da ih ispravi. Postupak koji nam omogućava da i kada se greška otkrije, analizator nastavi svoj rad se popularno naziva “Oporavak od greške”.

Pretpostavimo da je u nekom izrazu (koji se prepoznaje gramatikom koja je definisana u primeru X.11) izostavljena zatvorena zagrada. Prilikom prepoznavanja izraza sa zagradama koristilo bi se pravilo:

E : ‘(‘ E ‘)’

Oporavak od ove greške bi podrazumevao da kada zagrada nedostaje prijavimo gresku a da se analiza ipak nastavi. To znači da treba uvesti fiktivnu smenu po kojoj će se vrsiti redukcija ako zagrada nedostaje. U takvim smenama koristi se ključna reč error koja zamenjuje skup simbola proizvoljne dužine za koji akcija u odredjenom stanju nije definisana. U ovom slučaju, ako zagrada nedostaje, vršiće se redukcija po smeni:

E : ‘(‘ E error { /\* prijava greske \*/ };

Pri definisanju ovih fiktivnih smena treba voditi računa da se što je moguće više spreči višestruko prijavljivanje iste greške (što je nemoguće potpuno izbeći) i da se ne desi da se u istom stanju pri pojavi istog simbola na ulazu može vršiti redukcija po većem broju smena. U takvim slučajevima yacc će prijaviti takozvane *reduce/reduce* konflikte (koje on ne može da prevazidje) i kod neće generisati. Ako se prilikom kreiranja sintaksne tabele pojave *shift/reduce* konflikti (njihov uzrok je objašnjen u poglavlju o LR sintaknim analizatorima), prijaviće se semo upozorenje, a u sintaksnoj tabeli biće upisana *shift* akcija.

# Glava 11 – Atributne gramatike

## Atributne gramatike

Da bi se proverila semantička ispravnost (ili realizovala bilo koja naredna faza prvođenja) često je potrebno da se uz simbole koji učestvuju u smenama pamte i neke dodatne informacije o njima.

• Dodatne informacije o simbolima se nazivaju atributi simbola, a ovakve gramatike atributne gramatike.

• Sintaksno stablo obogaćeno vrednostima atributa simbola se naziva obeleženo (notirano) sintaksno stablo.

## Navesti i objasniti tipove atributa atributnih gramatika

Generisani(sintetizovani) atribut

Vrednosti atributa terminalnih simbola određuje leksički analizator.

` • Vrednosti atributa neterminalnih simbola se izračunavaju kao funkcije atributa simbola koji se nalaze u čvorovima potomcima odgovarajućeg čvora.

• Vrednosti ovih atributa se propagiraju kroz sintaksno stablo (odozdo naviše).

X→Y1 Y2 ...Yn

X.a = f(Y1 .a, Y2 .a, ..., Yk .a, ..., Yn .a)

Nasleđeni atributi

Vrednosti ovih atributa se izračunavaju kao funkcije atributa simbola koji se nalazi u roditeljskom čvoru i atributa simbola na istom nivou levo od simbola čiji se atribut određuje.

* + Ovi atributi se propagiraju kroz sintaksno stablo odozgo-naniže.

X→Y1 Y2 ...Yn

Yk .a = f(Y1 .a, Y2 .a, ..., Yk-1 .a)

## Semanticka rutina

Opis naredbi jezika preko bezkonteksnih gramatika obično nije dovoljan za uspešno prevođenje jezika. Postoji mnogo problema koji se ne mogu predstaviti formalnim opisom jezika. Zbog toga se kod praktičnih rešenja pravilima kojima se definiše jezik pridružuje skup dodatnih informacija tako što se simbolima gramatike dodaju određeni atributi. Vrednosti ovih atributa se izračunavaju preko semantičkih rutina koje se pridružuju pravilima gramatike.

Koriste se dva načina da se semantičke rutine pridruže pravilima gramatike:

* Sintaksno upravljane definicije (Syntax Directed Definitions)
* Translacione šeme (Translation Shemes)

Sintaksno upravljane definicije su jednostavno definicije rutina pridružene pravilima. Daju se na dosta visokom nivou i ne sadrže informaciju o tome kako se realizuju i kojim redosledom se primenjuju. Translacione šeme pokazuju redosled u kome se primenjuju semantička pravila što na način utiče i na njihovu implementaciju. To znači da su nešto nižeg nivoa od definicija.

Na osnovu semantičkih rutina mogu da se generišu grafovi zavisnosti (*Dependency Graphs*) koji pokazuju zavisnost između atributa i određuju kojim redosledom se izračunavaju vrednosti atributa.

## Zadaci semanticke analize (prezentacija 09)

Sintaksna analiza- utvrdjuje da li je struktura programa korektna

Semantička analiza – utvrđuje:

* 1. Da li su različiti strukturni elementi medjusobno usaglašeni
* Zato se naziva još i kontekstno-zavisna analiza (context-sensitive)
  1. Da li su sve promenljive koje se koriste u programu deklarisane,
  2. Da li je promenljivoj dodeljena vrednost pre njenog korišćenja,
  3. Da li je promenljiva „vidljiva“ u tački u kojoj se koristi (opseg važenja)
  4. Da li su operatori primenjeni nad operandima odgovarajućeg tipa,
  5. Da li je određeno ime, ime funkcije, promenljive, klase...
  6. Da li je lista stvarnih parametara u pozivu funkcije usaglašena sa listom fiktivnih parametara u definiciji funkcije,
  7. Da li klasa sartži član koji se koristi,
  8. Da li je korišćenje određenog člana klase u skladu sa njegovim pravom pristupa

## Definisati sintaksno upravljanje (prezentacija 09) \*

Rutine za proveru semantičke ispravnosti koda se pridružuju pravilima (smenama gramatike), odnosno semantičke rutine se izvršavaju u neterminalnim čvorovima sintaksnog stabla.

• Generalizacija ove ideje – sintaksno-upravljano prevođenje - sve naredne faze prevođenja se realizuju tako što se odgovarajuće rutine izvršavaju u čvorovima sintaksnog stable

## Metode za sintaksno upravljano prevodjenje (prezentacija 09)

Jedno-prolazni prevodioci

* + Svi atributi su istog tipa (nasleđeni ili generisani) prilagođeni algoritmu sintaksne analize
  + Kod top-down algoritama se koriste nasleđeni atributi
  + Kod bottom-up algoritama generisani

Više-prolazni prevodioci

* + Mogu da budu definisani i nasleđeni i generisani atributi
  + U prvom prolazu se generiše samo sintaksno stablo
  + Sintaksno stablo se obilazi više puta i prilikom svakog obilaska se izračunavaju novi atributi

## Type checking (prezentacija 09)

Najbitnija komponenta semantičkog analizatora

Gruba definicija: Type checker proverava da li su broj operanada i njihovi tipovi u izrazima odgovarajući .

Tip definiše oseg vrednosti podataka koje se mogu predstaviti i skup operacija koje se nad njima mogu izvoditi

Tip imaju:

* Promenljive
* Konstante
* Funkcije
* Izrazi

Postoje ugrađeni tipovi podataka i korinički definisani tipovi.

Zadaci : (ne mora valjda)

* 1. Da li su oba operanda u binarnim aritmetičkim izrazima numeričkog tipa,
  2. Da li je indeks polja celobrojnog tipa,
  3. Da li tip svakog stvarnog parametra u pozivu funkcije odgovara tipu formalnog parametra,
  4. Da li postoji klasa navedena kao roditeljska u definiciji izvedene klase,
  5. Da li klasa koja implementira interfejs sadrži sve funkcije definisane u interfejsu

# Glava 13 – Optimizacija koda

## Optimizacija toka programa

U toku generisanja međukoda (ili izlaynog koda) često se javljaju skokovi na skokove

* 1. Bezuslovni skok na bezuslovni
  2. Uslovni skok na bezuslovni,
  3. Bezuslovni skok na uslovn

Transformacija bezuslovnog skoka na bezuslovni:

go to L1 => go to L2

L1: go to L2 => L1: go to L2

Ako posle ove transformacije nema skokova na naredbu označenu sa L1: onda se ona može zameniti neoznačenom naredbom, a nekad i potpuno eliminisati.

Transformacija uslovnog skoka na bezuslovni:

if a < b go to L1 => if a < b go to L2

L1: go to L2 => L1: go to L2

Transformacija bezuslovnog skoka na uslovni:

go to L1 L1: if a< b go to L2

… go to L3

L1: if a< b go to L2 …

L3: L3:

* …

L2: L2:

Broj naredbi je ostao isti ali je drugi slučaj efikasniji:

• U prvom slučaju se uvek izvršava goto i if ,

• U drugom se uvek izvršava if, a go to samo kada navedeni uslov nije ispunjen

## Podela optimizacija \*

* 1. Zajednicki podizrazi
  2. Prostiranje naredbi za kopiranje
  3. Eliminisanje neaktivnog koda
  4. Optimizacija petlji
  5. Eliminacija indukcionih promenljivih
  6. Direktna redukcija

## Eliminacija reduntantnih naredbi

Eliminisanje STORE-LOAD parova:

Često se rezultat jedne operacije odmah koristi u narednoj. Kao posledica primene formalnih algoritama za prevođenje (naredba po naredba), u takvim slučajevima se generišu parovi naredbi:

Store tk

Load tk

Ovakav skup naredbi se prosto izostavlja (eliminiše).

Eliminacija se ne vrši ako je druga (Load) instrukcija obeležena (postoji skok na tu instrukciju iz drugih delova koda.

Store tk

Lab1: Load tk

Primer prezentacija 11.

## Koriscenje masinskih idioma,nabrojati ih \*

• Ovo je mašinski zavisna transformacija u kojoj se koriste prednosti i specifičnosti asemblerskog jezika ciljne mašine.

• Npr. neki asemblerski jezici imaju mogućnost izvršavanja auto-inkrementa ili auto-dekrementa nekog operanda pre ili posle izvršenja neke instrukcije.

• U 8086 asembleru loop instrukcija dekrementira sadržaj registra cx

1. samoindeksiranja,
2. samoinkrementiranja,
3. specifičnih adresiranja,
4. specifičnih registara procesora i
5. korišćenje steka.

## Nabrojati osnovne optimizacije \*

* 1. Elininisanje redundantnih naredbi.
  2. Optimizacija toka programa
  3. Algebarska uprošćenja
  4. Korišćenje mašinskih idioma

Blokovi za primere u pitanjima:

B1 B2 B3 B4

i := m -1 i:= i +1 j := j -1 if i >= j go to B6

j := n t2 := 4\*I t4 := 4 \* j

t1 := 4 \* n t3 := a [t2] t5 := a [t4]

v := a [t1] if t3 < v go to B2 if t5 > v go to B3

B5 B6

t6 := 4 \* I t11 := 4 \* I

x := a [t6] x := a [t11]

t7 := 4 \* i t12 := 4 \* i

t8 := 4 \* j t13 := 4 \* n

t9 := a [t8] t14 := a [t13]

a [t7] := t9 a [t12] := t14

t10 := 4 \* j t15 := 4 \* n

a [t10] := x t15 := 4 \* n

go to B2

## Zajednicki podizrazi

Izraz E naziva se zajedničkim ako je pre toga već bio izračunat. Npr. u bloku B5 izračunato je t6 kao 4\*i, a posle toga se ista vrednost pamti i kao t7. Isto važi i za t8 i t10. Na svim mestima gde se koristi t7 može da stoji t6, a umesto t10 može da stoji t8

Zajednički podizrazi mogu da se pronalaze i na globalnom nivou ako promenljive koje u njima učestvuju ne menjaju vrednost u međuvremenu

## Direktna redukcija u okviru petlje

Transformacija kojom se složenije (skuplje) naredbe zamenjuju jednostavnijim (jeftinijim) naredbama

• U našem primeru, direktna redukcija je primenjena u sprezi sa eliminisanjem indukcionih promenljivih.

• U okviru petlji u blokovima B2 i B3, naredbe množenja t2 := 4 \* i i t4 := 4 \* j zamenjene su jednostavnijim naredbama oduzimanja t4 := t4 - 4 i sabiranja t2 := t2 + 4.

## Indukcione promenljive

• Indukcione promenljive su promenljive čija promena izaziva promenu drugih promenljivih. Vrednost jedne promenljive direktno zavisi od druge. Ove zavisnosti se mogu iskoristiti za direktnu redukciju koda.

Npr. u bloku B3 imamo j = j-1, a zatim t4 :=4\*j. Ova naredba može da bude zamenjena naredbom: t4 := t4 – 4

B3 pre transformacije

j := j -1

t4 := 4 \* j

t5 := a [t4]

if t5 > v go to B3

B3 posle transformacije

t4 := 4 \* j

|

t4 := t4 - 4

t5 := a [t4]

if t5 > v go to B3

# Glava 14 – Generisanje koda

## Komponenta za upravljanje memorijom

1. Organizacija memorije u toku izvršenja programa:
   1. odredjivanje gde će se u memoriji smestiti kod programa,
   2. kako će se memorisati podaci različitih tipova,
   3. gde će se u memoriji smestiti podaci različitih klasa memorije
2. Pristup podacima smeštenim u različitim zonama memorije
3. Promena sadržaja memorije u trenutku poziva potprograma i povratka na pozivajući modul

3. Prevođenje naredbe poziva i naredbe povratka

## 2. Strategije za organizovanje memorije u toku izvrsenja.

Staička alokacija memorije

• Sadržaj memorije se ne menja u toku izvršenja programa – i kod programa i svi podaci su poznati u fazi prevodjenja

* + Za svaki potprogram postoji samo jedan aktivacioni slog što znači da ova strategija ne podržava rekurzivne potprograme

Dinamička alokacija memorije

• Sadržaj memorije se stalno menja u toku izvršenja programa

* + mogu se kreirati novi podaci, brisati
  + mogu se kreirati novi delovi koda, brisati
  + mogu se kreirati novi aktivacioni slogovi, brisati

• Povremeno se akriviraju posebni pomoćni programi (garbage collectori) koji vode računa o korišćenju memorijskog prostora

* + kad se uoče delovi memorije na koje ne ukazuje ni jedna referenca, oni se uklanjaju iz memorije, tj. taj deo memorije se oslobađa

Polu-statička ili stek alokacija memorije

• Operativna memorija se deli na 2 zone:

1. Statičku – čuva kod programa i globalne podatke

2.Dinamičku koja se deli na:

* + Stack – čuva aktivacione slogove potprograma
  + Heap – čuva podatke koji se po potrebi kreiraju u toku izvršenja programa i uklanjaju kada više nisu potrebni.

## Aktivacioni slogovi

|  |
| --- |
| Rezultat potprograma |
| Stvarni parametri |
| Link prema aktivacionom slogu pozivajućeg modula |
| Link prema podacima pozivajućeg modula |
| Stanje procesora u trenutku poziva |
| Lokalni podaci |
| Privrameni podaci |

Za svaki poziv potprograma kompilator obično generiše jedan aktivacioni slog u kome čuva sve informacije potrebne tom pozivu potprograma. U opštem slučaju aktivacioni slog ima strukturu. Međutim, struktura aktivacionog sloga nije ista kod svih jezika niti kod svih kompilatora za jedan jezik. Ona se obično prilagođava potrebama kompilatora

1. Na vrhu sloga, smešta se vrednost koju potprogram vraća glavnom programu. Ova vrednost se zbog efikasnosti vraća u neki od registra procesora.
2. Polje namenjeno stvarnim parametrima se koristi od strane glavnog programa da prenese parametre potprogramu koji odgovaraju konkretnom pozivu potprograma.
3. Opciono polje sa upravljačkim linkovima se koristi da se obezbedi veza sa aktivacionim slogom modula iz kojeg je pozvan potprogram.
4. Opciono polje sa linkovima za pristup podacima postoji kod jezika koji dozvoljavaju da se koriste nelokalni podaci, podaci iz aktivacionih slogova drugih potprograma. Ovo polje kod Fortrana nije potrebno zato što se svi nelokalni podaci čuvaju na jednom mestu.
5. Deo aktivacionog sloga namenjen čuvanju statusa procesora služi da se u njemu memorišu vrednosti registara pocesora koje su zatečene u trenutku poziva potprograma. Ovde se obično smešta vrednost Brojača neredbi i registara procesora opšte namene.
6. Polje namenjenu lokalnim podacima služi za smeštaj podataka koji pripadaju samo potprogram, definisani su u potprogramu.
7. Polje namenjeno privremenim podacima služi za memorisanje podataka koje generiše kompilator kao pomoćne lokacije u koje smešta međurezultate.

## Objasti pojam aktivacionog stable

Ako programski jezik dozvoljava rekurzivne pozive potprograma onda nije moguće koristiti statičku alokaciju memorije. Naime u ovom slučaju u jednom trenutku može da postoji više aktiviranih instanci potprograma i za svaki takav poziv potreban je poseban aktivacioni slog.

Jedna od tehnika koju je u ovom slučaju moguće koristiti je alokacija memorije pomoću steka, u okviru koje se prilikom svakog poziva potprograma kreira novi aktivacioni slog i smešta na stek. Kada se izvršavanje određene istance potprograma zavši odgovarajči aktivacioni slog se skida sa steka. U jednom trenutku u steku može da se nalazi više istanci aktivacionog sloga jednog potprograma, kao i više tazličitih aktivacionih slogova.

U vreme kompiliranja programa zna se samo veličina aktivacionog sloga ali ne i dubina rekurzije (koliko će aktivacionih slogova jednog potprograma u nekom trenutku biti u steku).

Ovo se resava aktivacionim stablom.

Svaki čvor u aktivacionom stablu sa prestavlja jedan poziv potprograma. Pozivi potprograma koji se izvrše iz jedne instance potprograma predstavljeni su čvorovima do kojih vode potezi iz čvora koji odgovara toj istanci. Pri tome redosled poziva odgovara redosledu čvorova poslatrano sleva u desno.

## Prevodjenje prilikom poziva \*

Prilikom poziva potprograma generise se aktivacioni slog i smesta na stek.

Koristi se bazni pokazivac koji uvek pozakuje na neku lokaciju aktivacionog sloga poslednjeg pozvanog potprograma,tj trenutni potprogram.

Kada se izvrsenje potprograma zavrsi bazni pokazivac treba da se vrati da ukazuje na aktivacioni slog pozivajuceg modula, dakle stara vrednost BP-a mora da se cuva I ta lokacija gde se cuva se zove kontrolna veza.

Na tu lokaciju BP ukazuje sve vreme.

|  |
| --- |
| Rezultat potprograma |
| Stvarni parametri |
| Adresa povratka |
| Staro BP  kontrolna veza |
| Prethodni sadrzaj radnih registara |
| Lokalne promenljive |
| Privrameni promenljive |

U trenutku poziva izvrsava se generisanje sekvence poziva a u trenutku povratka generisanje sekvence povratka.(generisanje I smestanje/skidanje aktivacionog sloga sa steka).

Sekvenca poziva:

Deo instrukcija koje pripadaju pozivajućem modulu

• Rezervacija prostora za rezultat potprograma

SUB SP, size

• Smeštanje u stek stvarnih parametara

PUSH arg\_n ... PUSH arg\_1

• Poziv potprograma CALL name

Deo instrukcija koje pripadaju pozvanom modulu

• Smeštanje u stek prethodne vrednosti baznog pokazivača

PUSH BP

• Promena sadržaja baznog pokazivača

MOV BP, SP

• Smeštanje u stek konteksta procesora

(sadržaja radnih registara)

PUSH AX PUSH BX PUSH CX PUSH DX PUSH SI PUSH DI

• Rezervacija prostora za lokalne promenljive

SUB SP, size

Sekvenca povratka:

Deo instrukcija koje pripadaju pozvanom modulu

• Upis rezultata u predviđenu lokaciju

MOV [BP+x], rez

• Izbacivanje lokalnih promenljivih ADD SP, size

• Vraćanje starog konteksta procesora

POP DI POP SI POP DX POP CX POP BX POP AX

• Preuzimanje stare vrednosti baznog pokazivača

POP BP

• Povratak u pozivajući modul

RET

Deo instrukcija koje pripadaju pozivajućem modulu

• Izbacivanje stvarnih parametara is steak

ADD SP, size

• Preuzivanje rezultata POP reg