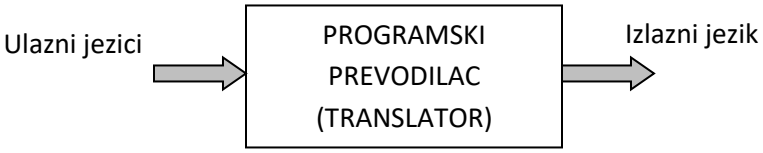
1. Objasniti namenu programskih prevodioca?

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



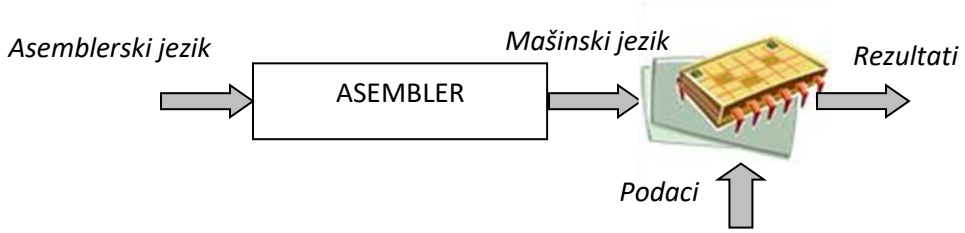
1. **Slika 1.1 Programski prevodilac**

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).

2. Sta je to assembler?

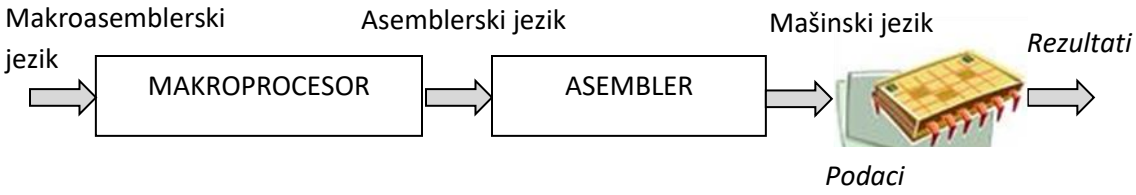
Mašinski jezik je programski jezik najnižeg nivoa i podrazumeva da se program sastoji od naredbi i podataka napisanih u binarnom obliku (preko nizova jedinica i nula) uz korišćenje apsolutnih adresa podataka. Na ovom nivou programski prevodilac nije potreban, naredbe se upisuju direktno u operativnu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



3. 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.



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.

4. 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.

5. 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 odrđenih 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.

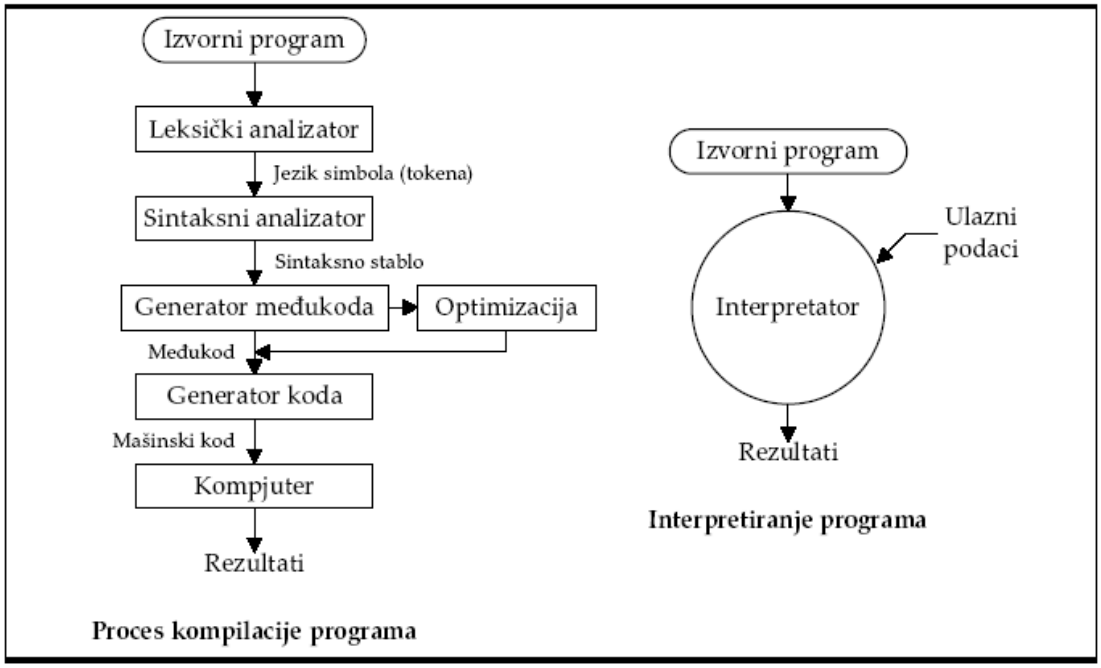
Kod koji kompilator genriše u prvom prolazu obično sadrži puno redundatnosti pa se zato mora vršiti optimizacija koda. Ova optimizacija može da bude mašinski nazavisna, kada se vrši na nivou međukoda ali i mašinski zavisna, kada se vrši na nivou generisanog koda.

Savremeni kompilatori pored sintakne 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.

6. 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.

7. Nacrtati potpunu semu strukture kompilatora?



Slika 1.4 Kompilator i interpretator

8. 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.

9. 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 ε.

1. ε reč nad azbukom V

2. Ako je x reč azbuke V i ako je a element azbuke V tada je i xa reč azbuke V.

3. y je reč nad azbukom V ako i samo ako je dobijen pomoću pravila 1. i 2.

Primer 2.2 Reči

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

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

10. 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 ε su njeni prefiksi, sufiksi i podnizovi.

Sekvenca reči **x** je svaki niz koji se dobija izbacivanjem nula ili više sukcesivnih simbola iy reči x.

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 su pravi 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*.

baa je podsekvenca reči *banana.*

11. Navesti osnovne operacije koje se mogu izvršavati nad rečima.

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

Eksponent reči

Višestruki proizvod iste reči se definiše kao eksponent reči.

Primer 2.5 Eksponent reči

xx = $\mathbf{x}^2$

xxx = $\mathbf{x}^3$.

$\mathbf{x}^1$=**x**

$\mathbf{x}^0$= ε .

Transponovana reč

Transponovana reč reči **x** u oznaci $\mathbf{x}^T$ definiše se na sledeći način:

1. $\varepsilon^T = \varepsilon$
2. $(xa)^T = ax^T$

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 $\mathbf{x}^T$=*abab*

Ako je **x**=*radar* tada je $\mathbf{x}^T$=*radar*. Ovakve reči nazivamo palindromima.

12. 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***={ ε , *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.

13. Navesti osnovne operacije nad jezicima.

Ako je **L** formalni jezik onda se može definisati i njegov reverzni jezik u oznaci $\mathbf{L^R}$, kao skup svih reverznih reči jezika **L**.

$$\mathbf{L^R}=\{\mathbf{x}^T \mid \mathbf{x}\in\mathbf{L}\}$$

Ako su **L** i **M** formalni jezici, onda je $\mathbf{L}\cup\mathbf{M}$, unija ova dva jezika i obuhvata sve reči koje pripadaju ili jeziku **L** ili jeziku **M**.

$$\mathbf{L}\cup\mathbf{M}=\{\mathbf{x} \mid \mathbf{x}\in\mathbf{L} \vee \mathbf{x}\in\mathbf{M}\}$$

Ako su **L** i **M** formalni jezici, onda je $\mathbf{L}\circ\mathbf{M}$, proizvod ova dva jezika i obuhvata sve reči nastaju operacijom nadovezivanja reči jezika **M** na reči jezika **L**.

$$\mathbf{L}\circ\mathbf{M}=\{\mathbf{xy} \mid \mathbf{x}\in\mathbf{L} \vee \mathbf{y}\in\mathbf{M}\}$$

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

$$\mathbf{L}^*=\bigcup_{i=0}^{\infty}\mathbf{L}^i$$

Pozitivno zatvaranje jezika L, za koje se koristi oznaka $\mathbf{L}^+$ se definiše kao**:**

$$\mathbf{L}^+=\bigcup_{i=1}^{\infty}\mathbf{L}^i$$

14. 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**=(**V**$_n$, **V**$_t$, S, **P**) gde je **V**$_t$ skup terminalnih sibola**,** **V**$_n$ skup neterminalnih simbola**,** **S** startni simbol i **P** skup smena definisan na sledeći način:

$$\mathbf{x} \to \mathbf{y}, \quad \text{gde je} \quad \mathbf{x} \in \mathbf{V}^* \mathbf{V}_n \mathbf{V}^* \wedge \mathbf{y} \in \mathbf{V}^*$$

pri čemu je **V** = **V**$_t \cup$ **V**$_n$ i važi **V**$_t \cap$ **V**$_n$ = Φ

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:

$$\mathbf{L(G)} = \{\mathbf{w} \mid S \xrightarrow{*} \mathbf{w}, \mathbf{w} \in \mathbf{V}_t^*\}$$

15. 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. $W \to aW$

2. $W \to Z$

3. $W \to X$

4. $Z \to aZ$

5. $X \to a$

6. $Y \to aa$

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 $W\alpha \to Y\beta$. Nikada se ne generiše neterminal *Y*.

16. 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**

17. Navesti osnovne tipove gramatika po Čomskom.

Još u svojim ranim radovima Chomsky je formalne 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 $\mathbf{x} \rightarrow \mathbf{y}$ koja zadovoljavaju sledeći uslov:

$$|\mathbf{x}| \leq |\mathbf{y}|$$

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:

$$\mathbf{x} \rightarrow \varepsilon$$

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:

$$A \rightarrow \mathbf{y}, \text{ gde je } A \in \mathbf{V_n} \text{ i } \mathbf{y} \in \mathbf{V^*}$$

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:

$A \to a\,B$ ili , $A \to a$ gde je $A \in \mathbf{V_n}$ i $\mathbf{a} \in \mathbf{V_t}$

Naziv regularne gramatike dolazi odatle što se gramatikama tipa tri generišu regularni izrazi koji se efikasno mogu prepoznavati konačnim automatima. Regularne gramatike imaju široku primenu u mnogim oblastima u kojima treba vriti 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.

18. 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.

19. Šta su to normalne forme gramatika?

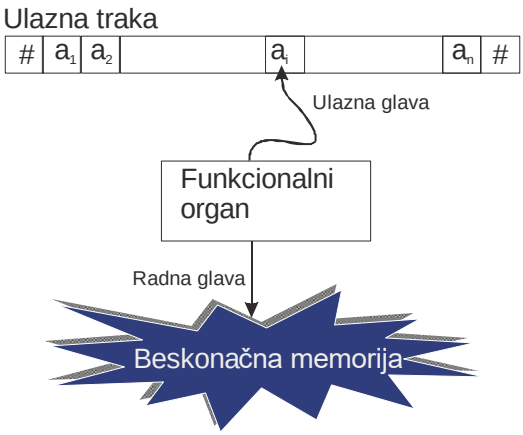
Pod normalnim formama gramatika podrazumevamo standardni način zadavanja gramatika određenog tipa. To znači da se smene jedne gramatike mogu prikazati na neki drugi način a da se pri tome ne promeni jezik koji definišu. Tako dolazimo i do pojma ekvivalentnih gramatika.. Dve gramatike su ekvivalentne 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 azličite normalne forme za određene tipove gramatika. Pomenućemo samo neke.

20. 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.

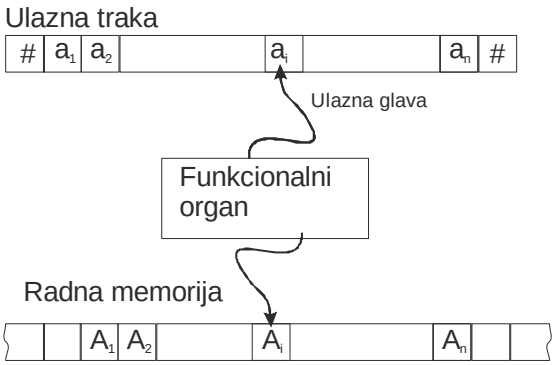
U opštem slučaju automati se mogu podeliti na četri klase:

- 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 definisana onda je to deterministički automat. Ukoliko je ovo preslikavanje višeznačno onda je to nedeterministički automat.

21. 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**, q_0, *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)

q_0 – Početno stanje operativnog organa

b – Oznaka za prazno slovo

P – Preslikavanje definisano sa:

$$\mathbf{Q}\times(\mathbf{V}\cup\#)\times\mathbf{S}\rightarrow\{\mathbf{Q}\times(\mathbf{S}\setminus\{b\})\times\{-1,0,+1\}\times\{-1,0,+1\}\}$$

odnosno ako je ulazna glava pozicionirana na simbolu *a*, funkcionalni organ se nalazi u stanju q_i, a radna glava iznad simbola *A*, automat će preći u stanje q_j, na radu traku će upisti simbol **B** i iyvršiće 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**).

22. 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 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 prpoznavanje jezika.

23. 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.

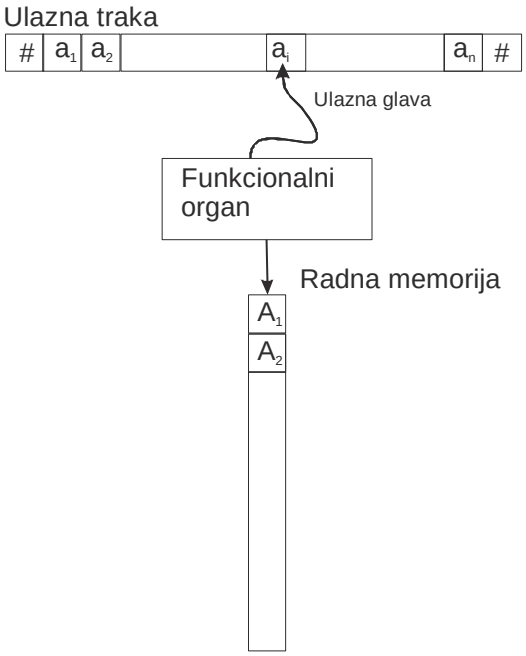
Odnosno, ako se prepoznaje reč $w \in \mathbf{V^*}$, gde je $|\mathbf{w}|=n$, na radnu traku se upisuje reč dužine *r,* pri čemu je $r \leq n*k$

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.
3. Ekvivalentnost 2N i 2D LOA do sada nije dokazana.
4. Svaka klasa LOA ne prepoznaje sve jezike tipa jedan.

24. 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**, q_0, 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**)

q_0 – Početno stanje operativnog organa

Z_0 – Početni simbol na vrhu magacina

P – Preslikavanje definisano sa:

$$\mathbf{Q}\times(\mathbf{V}\cup\#)\times\mathbf{S}\rightarrow\{\mathbf{Q}\times\mathbf{S}^{*}\times\{-1,0,+1\}\}$$

Primer 3.1 Magacinski automat

-1N magacinski automat prepoznaje jezik L ako i samo ako je L beskonteksni jezik tipa 2.

• 2N i 2D magacinskim automatima mogu se prepoznavati svi beskontesni i neki, ali ne svi, konteksni jezici.

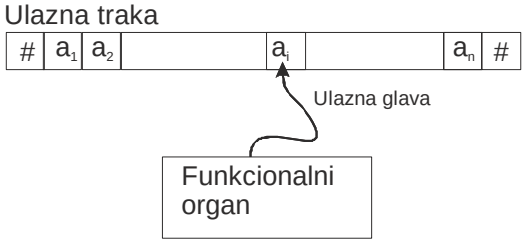
• Postoje beskonteksni jezici koji se ne mogu prepoznati 1D magacinskim automatima.

• Magacinski automati se koriste u sintaksnoj analizi programskih jezika.

25. Dati definiciju konačnih automata.

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

Na osnovu stanja operastivnog 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**, q_0, **F**), gde je:

Q – Skup stanja operativnog organa

V – Skup simbola na ulaznoj traci

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

q_0 – Početno stanje operativnog organa

P – Preslikavanje pravilima oblika: $\mathbf{Q} \times \{\mathbf{V} \cup \#\} \rightarrow \mathbf{Q}$

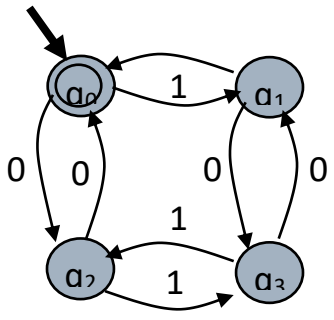
Za ulazni niz $\mathbf{w} \in \mathbf{V}^*$, kažemo da je prepoznat automatom ako automat prevodi iz početnog u neko od njegovih krajnjih stanja, odnosno ako je $\mathbf{P}(q_0, w) \in \mathbf{F}$.

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 $P(q_i,a)=q_j$ može tumačiti da se stanje q_i automata za ulazno slovo a preslikava u stanje q_j, 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.



Slika 25.2 Graf automata iz Primera 3.2.

26. Objasniti vezu između konačnih automata i jezika tipa tri.

Za svaki jezik definisan gramatikom tipa tri može se odrediti konačni automat koji ga prepoznaje.

Ulazna azbuka odgovara skupu terminalnih simbola gramatike.

Skup stanja odgovara skupu neterminalnih simbola gramatike.

Početno stanje je startni simbol gramatike.

Preslikavanje oblika g(qi,a)=qj odgovara smeni gramatike qi→aqj.

Završna stanja su oni neterminalni simboli za koje postoji smena oblika qi→ε.

Primer 3.5 **Gramatika u konački automat**

1. Definisati konačni automat za prepoznavanje označenih celih brojeva. Označeni ceo broj definisan je sledećom gramatikom:

<OznaceniCeoBroj> → + <NeoznaceniCeoBroj>

<OznaceniCeoBroj> → - <NeoznaceniCeoBroj>

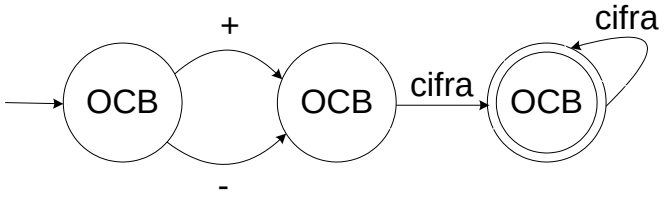
<NeoznaceniCeoBroj> → :**cifra** <NizCifara>

<NizCifara> → **cifra** <NizCifara>

<NizCifara> → ε

Rešenje:

Graf automata koji prepoznaje jezik definisan gramatikom dat je na slici 3.7.



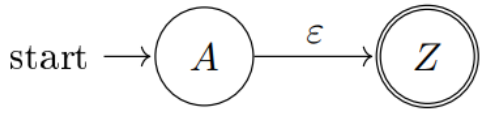
27. Dati definiciju regularnih izraza. Navesti nekoliko primera regularnih izraza.

Jezici tipa tri koji se prepoznaju konačnim automatima se nazivaju i regularnim jezicima zato što se mogu opisati regularnim izrazima. 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.

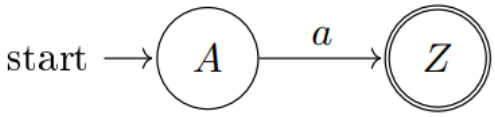
Regularni izraz nad azbukom **V** definise se sledećim skupom pravila:
1. Φ (prazan skup) je regularni izraz.

2. $\boldsymbol{\varepsilon}$ je regularni izraz.
3. Ako je $a \in \mathbf{V}$, onda je a regularni izraz.
4. Ako su **r**1 i **r**2 regularni izrazi, onda su i (**r**1| **r**2) i (**r**1 ◦ **r**2) regularni izrazi.
5. Ako je **r** regularni izraz onda je i **r**$*$ takodje regularni izraz.
6. Nema drugih regularnih izraza nad azbukom **V**

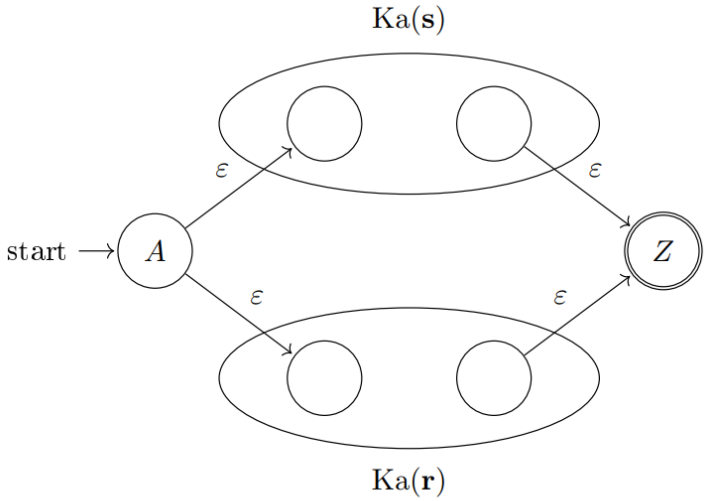
28. Pokazati kako se osnovni regularni izrazi preslikavaju u automate.



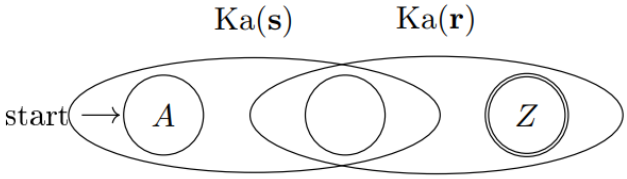
Slika 3.5: Automat za regularni izraz $\mathbf{r} = \varepsilon$



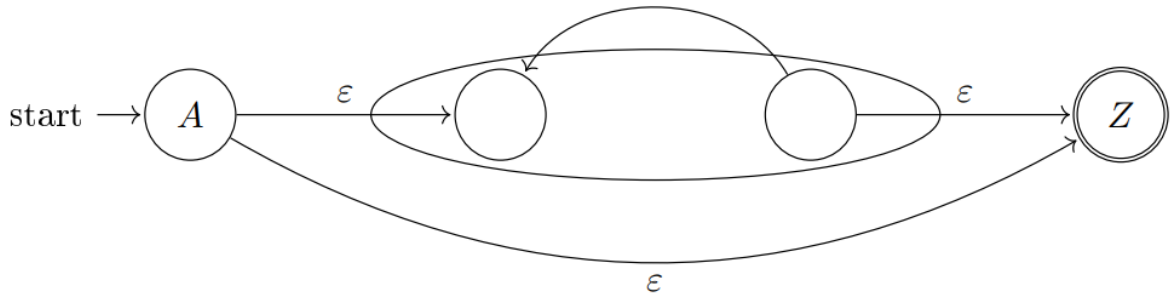
Slika 3.6: Automat za regularni izraz $\mathbf{r} = a$



Slika 3.7: Automat za regularni izraz $\mathbf{r} \mid \mathbf{s}$



Slika 3.8: Automat za regularni izraz $\mathbf{r} \circ \mathbf{s}$



Slika 3.9: Automat za regularni izraz $(\mathbf{s})^*$

29. 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 grešaka sa ulaznim kodom. Npr. vodi računa o broju *newline* simbola, što se koristi kod referenciranja na greške.

30. 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.

31. 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: anewline 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.

32. 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 u 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 taj slog.

33. Koje greške može da otkriva leksički analizator?

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

34. Šta je i čemu služi LEX?

Lex je generator leksickih analizatora.

35. Objasniti kako radi LEX.

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

36. 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.

37. 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.

38. 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.

39. 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.

40. 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 preimenjuje 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.

41. Opisati osnovni algoritam za Top-down analizu.

1. U izvedenu sekvencu upisati startni simbol gramatike, proglasiti ga za tekuci u izvedenoj sekvenci i pročitati prvi simbol iz ulaznog koda.

2. Ukoliko je tekuci simbol u izvedenoj sekvenci neterminalni simbol, zameniti ga desnom stranom prve smene na čijoj je levoj strani taj neterminalni simbol.

3. Ukoliko je tekuci simbol u izvedenoj sekvenci terminalni simbol jednak tekućem ulaznom simbolu, prihvati ga (preći na analizu sledećeg simbola).

4. Ukoliko je tekuci simbol u izvedenoj sekvenci terminalni simbol različit od tekućeg ulaznog simbola, poništiti dejstvo poslednje primenjene smene. Ukoliko postoji još koja smena za preslikavanje istog neterminalnog simbola, pokušati sa primenom sledeće smene, u suprotnom vratiti se još jedan korak nazad.

5. Ukoliko se vraćanjem došlo do startnog simbola i ne postoji više smena za njegovo preslikavanje, ulazni kod sadrži sintaksnu grešku.

6. Ukoliko nakon prihvatanja poslednjeg ulaznog simbola, ni u izvedenoj sekvenci nema neobradjenih simbola, kod je sintaksno korektan.

42. 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 analize ne može da se primeni kod tzv. levo rekurzivnih gramatika. Levo rekurzivne gramatike su gramtike kod kojih postoje levo rekurzivna pravila, odnosno pravila oblika:

$$A \rightarrow A\mathbf{r}, \text{ gde je } A \in \mathbf{V_n}, \text{ a } \mathbf{r} \in \mathbf{V^*}$$

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 analizu nije moguće završiti.

43. 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:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$

Ovaj skup pravila moguće je zameniti sledećim skupom nerekurzivnih pravila:

$A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m \mid \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$

$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \mid \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A'$

Moguća je i sledeća transformacija:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$

44. Šta je to indirektna rekurzija i kako se eliminiše?

Eliminisanje indirektne rekurzije moguće je sledećom trenaformacijom:

Sve skupove pravila oblika:

$X \rightarrow Y\alpha$

$Y \rightarrow \beta 1 \mid \beta 2 \ldots \mid \beta 3$

treba zameniti pravilima:

$X \rightarrow \beta 1\ \alpha \mid \beta 2\ \alpha \ldots \mid \beta 3\ \alpha$

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

45. Objasniti osnovni algoritam za Bottom-up analizu.

1. Pročitati prvi simbol iz ulaznog koda i upisati ga u radni magacin.

2. Ponavljati sledeće korake dok se ne dođe do kraja ulaznog koda:

a) Ukoliko se na vrhu radnog magacina nalazi desna strana neke smene,

redukovati frazu sa vrha radnog magacina (tj. frazu sa vrha radnog magacina

zameniti simbolom sa leve strane odgovarajuće smene);

b) u suprotnom pročitati novi simbol iz ulaznog koda i smestiti ga u radni magacin.

3. Kada se dođe do kraja ulaznog koda, ukoliko je u radnom

magacinu samo startni simbol gramatike, analiza je uspešno

završena (kod je prihvaćen).

4. Kada se dođe do kraja ulaznog koda, ukoliko kod nije redukovan

na startni simbol gramatike, a usput su izvršene neke redukcije,

vratiti se na korak kada je izvršena poslednja redukcija, poništiti

njeno dejstvo i preći na korak 2.

5. Ako se došlo do kraja ulaznog koda i pri tom nije izvršena ni jedna

redukcija, analiza je završena neuspešno, ulazni kod je sintaksno

neispravan.

46. Dati definiciju proste LL-1 gramatike.

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

$$A \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n \text{ gde je } a_i \in \mathbf{V}_t \text{ , } \alpha_i \in \mathbf{V}^*, \wedge\, a_i \neq a_j \text{ za } i \neq j$$

Primer 6.1 Prosta LL-1 gramatika

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

1. $S \Rightarrow aS$
2. $S \Rightarrow bA$
3. $A \Rightarrow d$
4. $A \Rightarrow ccA$

47. 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:

$$Ts(A,a) = \begin{Bmatrix} pop & \text{ako je } A = a, \quad a \in \mathbf{V}_t \\ acc & \text{ako je } A = \# \wedge a \in \# \\ (a\alpha, i) & \text{ako je } A \Rightarrow a\alpha \quad i - \text{to pravilo} \\ err & \text{u svim ostalim slucajevima} \end{Bmatrix}$$

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

1. U postupku se koriste 1 radni magacin koji čuva trenutno stanje u razvoju i niz koji pamti primenjene smene. Na početku analize se u radni magacin upisuje granični simbol i startni simbol gramatike i pročita se prvi simbol iz ulaznog niza.

2. Na osnovu simbola na vrhu radnog magacina i izdvojenog ulaznog simbola, određuje se akcija koju treba izvršiti. Sve moguće akcije zapamćene su u LL(1) sintaksnoj tabeli.

3. Korak 2 se ponavlja dok pročitana akcija iz sintaksne tabele ne bude "PRIHVATI" (acc) ili "GREŠKA" (err).

49. Objasniti pojam leve faktorizacije.

Naime, ukoliko u gramatici postoji veći broj pravila za isti neterminalni simbol koja na desnoj strani imaju reči sa istim prefiksom, odnosno smene oblika:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

gde su $\alpha, \beta_1, \beta_2, \cdots, \beta_n, \gamma \in \mathbf{V}*$ i $A \in \mathbf{Vn}$,

ovo se naziva levom faktorizacijom i takva gramatika sigurno nije LL-1 gramatika. Međutim ona se transformiše u LL-1 gramatiku sledećom smenom:

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

50. Dati definiciju funkije FIRST.

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

Ako je $\alpha \in \mathbf{V}^+$

$$FIRST(\alpha) = \left\{ a \mid \alpha \xrightarrow{*} a\beta, \quad a \in \mathbf{V_t}, \beta \in \mathbf{V}* \right\}$$

51. 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:

$$FOLLOW(A) = \left\{ s \in \boldsymbol{V_t} \mid S' \xrightarrow{*} \alpha A s \gamma, \ A, S' \in \boldsymbol{V_n}, \ \gamma \in \boldsymbol{V}^* \right\}$$

gde je S' startni simbol gramatike.

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* na desnoj strani smene može da se nađe u jednom od sledećih konteksta:

$Z \to \alpha X x \beta \wedge x \in \mathbf{V_t} \Rightarrow x \in \text{FOLLOW}(X)$

$Z \to \alpha XY \beta \wedge Y \in \mathbf{V_n} \Rightarrow \text{FIRST}(Y) \subset \text{FOLLOW}(X)$

3. $Z \text{ -> } \alpha X \text{ ->>} \Rightarrow \text{Follow}(Z) \subset \text{Follow}(X)$

52. Dati definiciju LL-1 gramatika bez ε pravila.

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

$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$ slupovi $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \ldots, \text{FIRST}(\alpha_n)$ disjunktni po parovima, odnosno:

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \Phi, \quad i \neq j.$$

53. Kako se popunjava Sintaksna tabela kod LL-1 gramatika bez ε pravila?

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

$$Ts(A,a) = \left\{ \begin{array}{l} pop \quad \text{ako je } A = a, \quad a \in \mathbf{V_t} \\ acc \quad \text{ako je } A = \# \wedge a \in \#_t \\ (\beta, i) \quad \text{ako je } a \in FIRST(\beta) \text{ i } A \Rightarrow \beta \text{ je } i - \text{to pravilo} \\ err \quad \text{u svim ostalim slucajevima} \end{array} \right\}$$

54. Kako se popunjava Sintaksna tabela kod LL-1 gramatika sa ε pravilima?

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

$$Ts(A,a) = \left\{ \begin{array}{l} pop \quad \text{ako je } A = a, \quad a \in \mathbf{V_t} \\ acc \quad \text{ako je } A = \# \wedge a \in \#_t \\ (\alpha, i) \quad \text{ako je } a \in FIRST(\alpha) \text{ i } A \Rightarrow \alpha \text{ je } i - \text{to pravilo} \\ \text{ili } a \in FOLLOW(A) \text{ i } A \Rightarrow \alpha \text{ je } i - \text{to pravilo i } \varepsilon \in FIRST(\alpha), \text{za } A \in V_n \\ err \quad \text{u svim ostalim slucajevima} \end{array} \right\}$$

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

55. 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:

$C \to \mathbf{p}AB\mathbf{q}$, gde je: $A,B,C \in \mathbf{V_n}$ i $\mathbf{p,q} \in \mathbf{V^+}$

56. Definisati osnovne relacije prvenstva između terminalnih simbola.

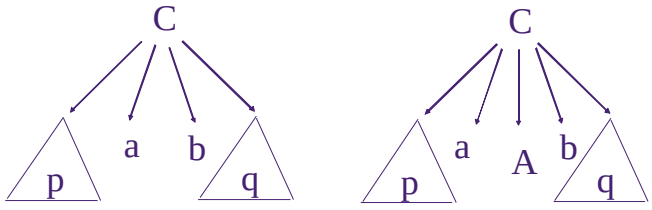
Postoje tri osnovne relacije i one su definisane na sledeći način:

Relacija istog prioriteta $a \doteq b$

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

$$C \Rightarrow pabq \text{ ili } C \Rightarrow paAbq, \text{ gde je } C,A \in V_n,\ a,b \in V_t,\ p,q \in V^*$$

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 $a < \cdot b$

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

$$C \Rightarrow paAq \text{ i } A \xrightarrow{*} br \text{ ili } A \xrightarrow{*} Dbr, \text{ gde je } C,A,D \in V_n,\ a,b \in V_t,\ p,q,r \in V^*$$

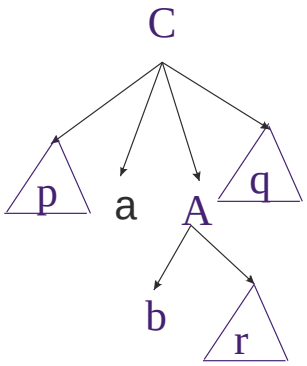
Predstavljeno na sintaksnom stablu to znači da se terminalni simbol *a* pojavljuje levo i na višem nivou u odnosu na terminalni simbol *b*, Slika 7.2.

Relacija višeg prioriteta $a \cdot > b$

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

$$C \Rightarrow pAbq \text{ i } A \xrightarrow{*} ra \text{ ili } A \xrightarrow{*} raD, \text{ gde je } C,A,D \in V_n,\ a,b \in V_t,\ p,q,r \in V^*$$

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.



Slika 7.2 Relacija nižeg prioriteta

57. 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.

58. Šta je to Operatorska tablica prvenstva? Dati primer.

U postupaku 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.

59. 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 mo\e 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:

$a \lessdot b \Leftrightarrow f(a) < g(b)$

$a \doteq b \Leftrightarrow f(a) = g(b)$

$a \gtrdot b \Leftrightarrow f(a) > g(b)$

Memorijski proctor potreban za pamćenje funkcija prvenstva je $2\cdot(n+1)$.

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

Ako važi relacija $a \gtrdot b$, tada postoji poteg od čvora $f(a)$ prema čvoru $g(b)$.

Ako važi relacija $a \lessdot b$, tada postoji poteg od čvora $g(b)$ prema čvoru $f(a)$.

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

Vrednost funkcije prvenstva se odrdjuje 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.

60. 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 $\lessdot$ ili relacija $\doteq$, 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 $\gtrdot$ treba ići dublje u stek i tražiti prvi terminal u steku koji je $\lessdot\bullet$ od svog sledbenika u steku i proveriti da li fraza sa vrha steka (od uočenog simbola do vrha, fraza između simbola $\lessdot\bullet$ i $\bullet\gtrdot$) 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.

61. Dati definiciju Gramatika prvenstva.

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

62. 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.

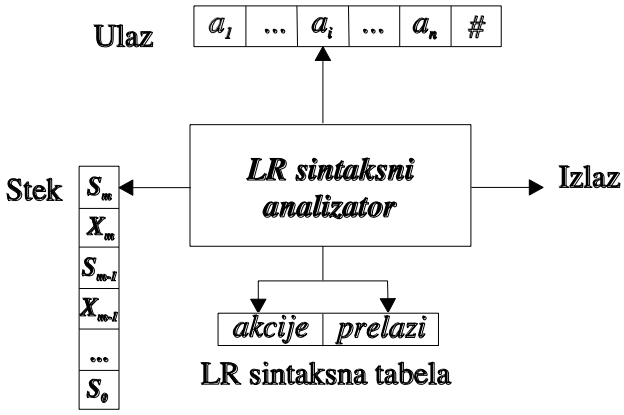
63. Opisati algoritam za sintaksnu analizu Gramatika prvenstva.

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 $<\cdot$ ili relacija $\doteq$, tekući element iz ulaznog niza smestiti u stek α i preći na analizu sledećeg simbola.
2. Ako važi relacija $\cdot>$ treba ići dublje u stek i tražiti prvi simbol u steku koji je $<\cdot$ 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.

66. 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.

67. 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.

68. 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.

69. Šta su to LR(0) članovi?

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

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

70. Šta je to kanonička kolekcija LR(0) članova?

Kanonički skup LR članova obuhvata sve vidljive prefikse reči jezika koji je opisan gramatikom. Određivanje kanoničkog skupa kreće od LR(0) člana $S' \to \bullet S$ koji se formira na osnovu fiktivne smene $S' \to S$, gde je *S* startni simbol gramatike. Nakon formiranja ovog početnog LR(0) člana primenjuju se funkcije *closure* i *goto* sve dok je to moguće. Rezultat ovog koraka je kanonički skup zatvaranja LR(0) članova K={**I_0, I_1, ..., I_k}.**

71. Kreiranje SLR sintaksne tabele

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 $S' \to \bullet S$ koji se formira na osnovu fiktivne smene $S' \to S$, gde je *S* startni simbol gramatike. Nakon formiranja ovog početnog LR(0) člana primenjuju se funkcije *closure* i *goto* sve dok je to moguće. Rezultat ovog koraka je kanonički skup zatvaranja LR(0) članova K={**I_0, I_1, ..., I_k}.**

2. Crtanje grafa prelaza konačnog automata za prepoznavanje vidljivih prefiksa pri čemu važe sledeća pravila:
    a. Svakom zatvaranju iz kanonickog skupa pravila K={**I_0, I_1, ..., I_k}.**dodeljuje se jedno stanje u grafu automata.
    b. Potezi u grafu određeni su *goto* funkcijama iz kanoničkog skupa LR(0) pri čemu ako je *goto*(**I_i**, *X*) = **I_j** onda postoji odlazni poteg iz stanja Ii do stanja Ij za slovo *X,* gde je $X \in \mathbf{V}$.
    c. Sva stanja automata koja odgovaraju zatvaranjima LR(0) članova kojima pripadaju članovi oblika $A \to \alpha \bullet$ su završna stanja (stanja redukcije za pravila $A \to \alpha$.
    d. Stanje kome pripada LR(0) član $S' \to \bullet S$, nastao na osnovu fiktivne smene je početno stanje automata.
3. Popunjavanje LR sintaksne tabele koje se vrši primenom sledećih pravila:

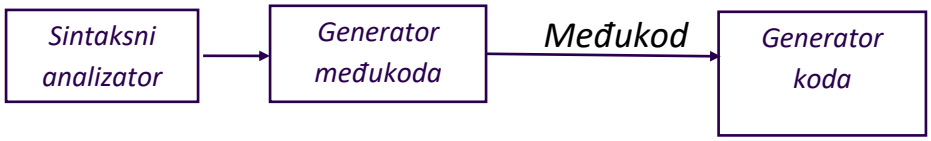
    a. if$[A \to \alpha \bullet a\beta] \in \mathbf{I}_i$ i $\text{goto}(\mathbf{I}_i, a) = \boldsymbol{I}_j$

tada je $action(i, a) = shift\ j, \quad a \in V_t$.

$b.$ if$[A \Rightarrow \alpha \bullet] \in \mathbf{I}_i$ uzeti
$action(i, a) = reduce\ A \Rightarrow \alpha$ za a ∈ FOLLOW(A), A $\neq$ S'.

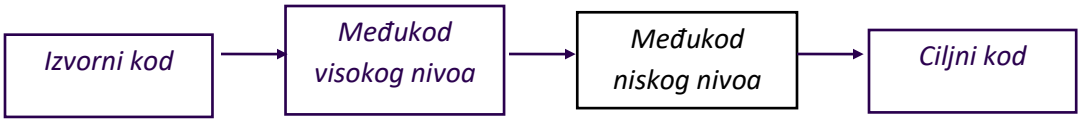
$c.$ $if\ [S' \Rightarrow S \bullet] \in \boldsymbol{I}_i$ tada je $action(i, \#) = accept$.

72. Objasniti mesto i namenu međukoda u strukturi kompilatora.



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.



Slika 10.2 Međukod u više nivoa

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

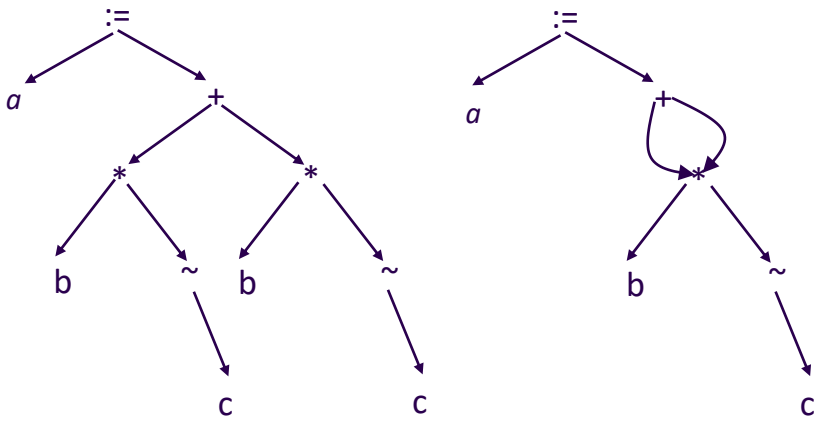
1. Prenosivost koda. Na osnovu međukoda može se generisati kod za različite ciljne mašine
2. Na nivou međukoda može da se vrši mašinski nezavisna optimizacija

73. Navesti osnovne vrste međukoda.

Međukod može da bude generisan u različitim oblicima. Ovde ćemo razmotriti apstraktno 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.

74. Dati primer apstraktnog sintaksnog stabla.

Apstraktno sintaksno stablo je struktura koja se generiše na osnovu sintaksnog stabla 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 apstraktno 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.



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

75. 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.

76. 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

77. 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)

78. 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
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

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

79. 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.

80. 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 atribut *code* 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*.

S ⇒ *id* = E;	S.code = E.code || *gen(top.get(***id***.lexeme) '=' E.addr)*
E ⇒ E1 + 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*
E ⇒ ***id***	*E.addr* = top.get(id.lexeme) E.code = ''

81. 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).* irina matrice a je *w*=24, i pretpostavi'emo 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 semanti;kih pravila biće generisan sledeći troadresni međukod:

t1 = i * 12

t2 = j * 4

t3 = t1 + t2

t4 = a [t3]

t5 = c + t4

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

PRAVILA GRAMATIKE	**SEMANTIČKA PRAVILA**
P ⇒ *S*	*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*

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

PRAVILA GRAMATIKE	**SEMANTIČKA PRAVILA**
B ⇒ *B1 || 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*
B ⇒ B1 && 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*
B ⇒ E1 **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)*

84. 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.

Razmotrićemo primer jednog jednostavnog programskog jezika koji je definisan sledećom gramatikom:

85. 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)*

86. 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*

87. Dati primer semantičkih pravila kojima se vrši provera naredbi.

88. 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*

89. Šta je zadatak modula za proveru tipova podataka.

Jedan od važnih problema koji se rešava u okviru kompilatora je i analiza tipova podataka što je u skladu sa konceptom jakih tipova podataka koji je standardno postavljen kod savremenih

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

90. Zadaci komponente za upravljanje memorijom u toku izvršenja programa.

Zadaci:

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

91. Nabrojati strategije za upravljanje memorijom u toku izvršenja programa i objasniti statičku alokaciju memorije.

Staička alokacija memorije

Dinamička alokacija memorije

Polu-statička ili stek alokacija memorije

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

92. 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

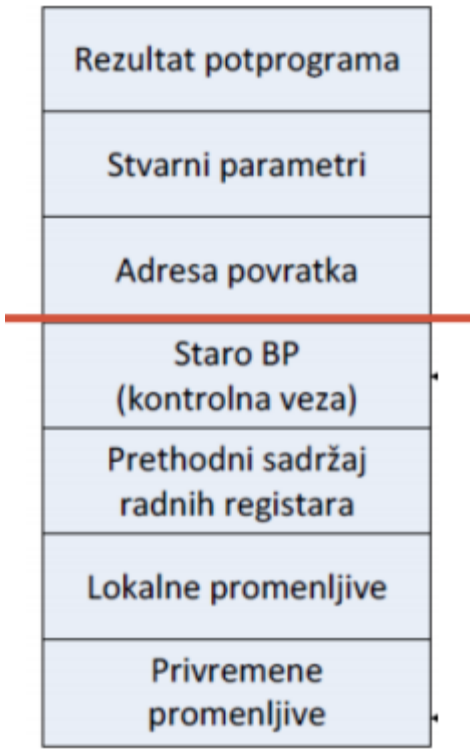
93. Polustatička alokacija memorije.

• Operativna memorija se deli na 2 zone:
  • Statičku – čuva kod programa i globalne podatke
  • 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

94. Pojam i sadržaj aktivacionog sloga.

Prilikom poziva potprograma kreira se tzv. aktivacioni slog (aktivacioni zapis) potprograma i smešta se u stek.

U aktivacionom slogu se pamte sledece stvari :



95. Pojam aktivacionog stabla.

Aktivaciono stablo mozemo kreirati da bismo vizuelno reprezentovali nacin na koji kontrola ulazi I izlazi iz aktivacije, odnosno to je stablo gde:

svaki cvor predstavlja aktivaciju procedure,

koren stabla je main program,

cvor a je roditelj cvoru b ako I samo ako se kontrola predaje od cvora a do cvora b,

cvor a je levo od cvora b ako je zivotniciklus a zavrsio pre b,

96. Prevođenje poziva potprograma.

- 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

97. Prevođenje povratka u pozivajući modul.

- 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 steka

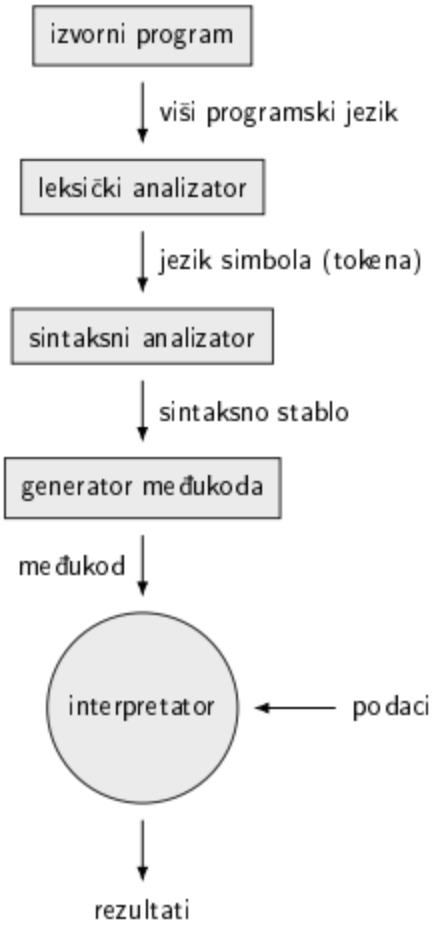
    ADD SP, size

  - Preuzivanje rezultata

    POP reg

98.Sta podrazumevamo pod hibridnim prevodiocima?

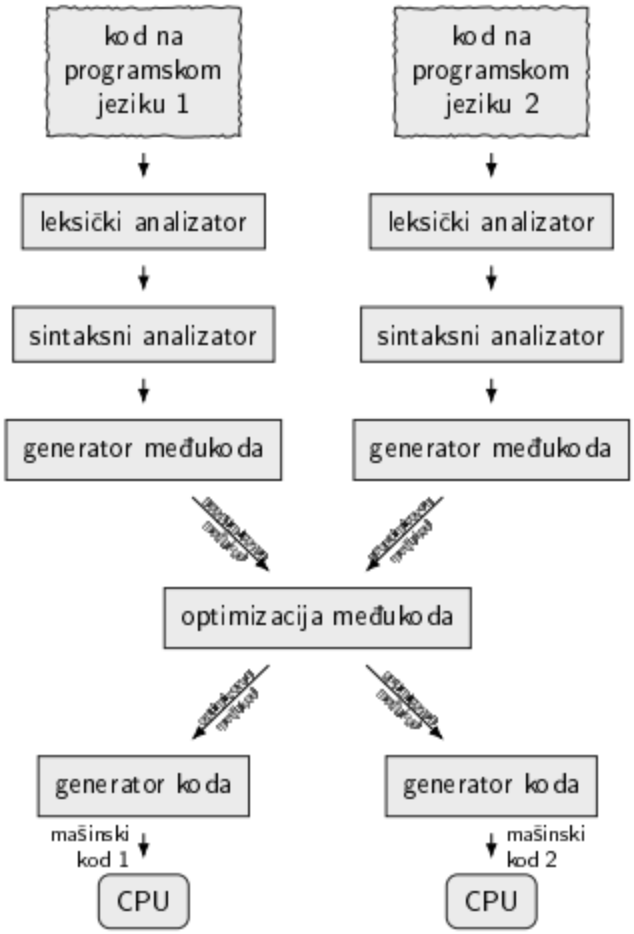
Pored cistih kompilatora i interpretatora danas se koriste i hibridni prevodioci (slika 1.6) koji predstavljaju neku vrstu mesavine kompilatora i interpretatora, vrse analizu koda kao kompilatori, generisu medjukod programa i nakon toga interpretiraju taj medjukod.



Slika 1.6: Hibridni prevodilac

99.Visejezicni prevodioci

Koris¢enje medjukoda, takodje, omogucava realizaciju visejezi£nih kompilatora (slika 1.8) . To su kompilatori koji imaju posebne delove za analizu koda za vise razlizitih programskih jezika, pri cemu se kod svih ovih analizatora generise isti medjukod programa. Posle toga, taj medjukod moze da se prevodi u masinski jezik odredjenog racunara, jednog ili vise razlicitih procesora.



Slika 1.8: Višejezički kompilatori

100.Nabrojati vrste programskih prevodioca

Asembleri, makroprocesori, prevodioci jezika viseg nivoa (kompilatori, interpretatori), hibridni prevodioci, multijezicki prevodioci.

101. Razlika izmedju lekseme, tokena I sablona

* Leksema - ulazni niz koji se prepoznaje na osnovu formalnog opisa pomocu sablona i za koji se generise odredjeni token.
* Token - izlazni simbol koji se generise kada je prepoznat odreženi ulazni niz znakova.

* Sablon (pattern) - regularni izraz, formalni opis ulaznih nizova za koje se generise odedjeni token.

Primer :

LEKSEMA	ŠABLON	TOKEN
const	Const	**const**
if	If	**if**
$<, \leq, =, \neq, >, \geq$	$<\|\leq\|=\|\neq\|>\|\geq$	**rel**
pi, C, P2	*slovo* (*slovo* | *cifra*)*	**id**
3.1416, 0, 6.02E23	(+ | −)? *cifra*+ (.*cifra*+)? (E(+ | −)? *cifra*+)?	**num**
"Ovo je podatak"	"(*slovo* | *blanko* | *cifra*)*"	**literal**

102. Lex, flex, jflex

Lex generiše kod koji prepoznaje instance zadatih regularnih izraza u ulaznom tekstu. Osnovna namena Lex-a jeste generisanje leksičkih analizatora.

Lex generiše kod na osnovu ulazne Lex specifkacija koja sadrži tri osnovna dela koja se zapisuju na sledeć¡i na£in:
<definicije>
%%
<pravila>
%%
<korisnički_potprogrami>

Jlex je alat, koji generiše izvorni kod leksičkih analizatora u programskom jeziku Java. Ovo je besplatan i open source alat. JFlex specifkacija, takođe, sadrži tri osnovna dela, ali su druga£ije raspoređeni nego u Lex specifkaciji. Dakle, format JFlex specikacije je:
<korisnički_kod>
%%
<opcije_i_direktive>
%%
<pravila>

103. Osnovna ideja LL(k) algoritma za sintaksnu analizu

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 ¢e da se primeni se donosi i na osnovu slova u ulaznom nizu koje treba da se prepozna, na kome je trenutno ulazni pokaziva£. Primenjuje se pravilo koje ¢e sigurno da generi^{2}e bar to slovo. Ovakvi analizatori su poznati kao LL-1 analizatori (look-left 1), gde cifra 1 ukazuje na to da se predikcija vr^{2}i na osnovu jednog slova u ulaznom nizu. Generalno gledano, mogu da se deni^{2}u i LL-k analizatori kod kojih se predikcija vr^{2}i na osnovu re£i du^{o}ine *k*, ali su LL-1 mnogo upotrebljivij

104.lr algoritam za sintaksnu analizu

Postaviti ip na početak niza w#;

while true 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 stek 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->β

then begin

Izbaciti 2x|β| simbola iz steka, ubaciti A u stek, a zatim i

oznaka stanja koja se dobija na osnovu goto(s',A), gde je s'

oznaka stanja u koje smo se vratili posle redukcije , generiše se

izlaz A->β .

end

else if action(s,#) = accept then

return

else error()

end

105. Kreiranje LR sintaksne tabele na osnovu grafa prelaza konačnog automata za prepoznavanje vidljivih prefiksa.

• Vrste u LR sintaksnoj tabeli su stanja LR sintaksne analize, tj. stanja konačnog automata za prepoznavanje vidljivih prefiksa.

• Ako u grafu postoji poteg između čvorova i i k pod dejstvom terminalnog simbola a, tada je akcija(i,a)=sk.

• Ako u grafu postoji poteg između čvorova i i k pod dejstvom neterminalnog simbola A, tada je prelaz(i,A)=k.

• Ako zatvaranju li pripada član oblika A $\rightarrow$ a. (a smena pod rednim brojem k ima oblik: A $\rightarrow$ a), tada će u grafu stanje i biti označeno kao završno stanje što će značiti da je to redukciono stanje za smenu k. U tom slučaju akcija(i,a)=rk za svako a koje pripada skupu FOLLOW(A).

106. Pojam atributnih gramatika

• 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.

107. Vrste atributa simbola u atributnim gramatikama

• Vrednost atributa simbola određuju se semantičkom rutinom koja je pridružena pravilu (smeni gramatike) koje se koristi u odgovarajućem čvoru sintaksnog stabla.

• Na osnovu toga kako se propagiraju vrednosti atributa, atributi se dele na:

• Generisane

• Nasleđene

Vrednosti generisanih atributa se izračunavaju kao funkcije atributa potomaka odgovarajućeg čvora dok se vrednosti nasleđenih atributa izračunaveju kao funkcije atributa roditeljskih čvorova i atributa ostalih čvorova na istom nivou.

108. Pojam semantičke rutine

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

109.Pojam sintaksno-upravljanog prevodjenja

• 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 stabla.

110.Metode za realizaciju sintaksno-upravljanog prevodjenja

• 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

111.Šta se podrazumeva pod pojmom obeleženo (notirano) sintaksno stablo?

• 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.

112. Type checker

• 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.

113.Provera tipova u aritmetičkim izrazima korišćenjem atributnih gramatika.

Vec ima gore ona tabela glupa

114. Razlike izmedju sintaksnog stabla I apstraktnog sintaksnog stabla

Postupkom sintaksne analize određuje se skup i redosled pravila koja treba primeniti da bi se generisala zadata reč. Redosled primene pravila se može predstaviti sintaksnim stablom. Na osnovu sintaksnog stabla generiše se apstraktno sintaksno stablo (AST - abstract syntax tree) u
kome se u čvorovima nalaze operatori a grane vode do operanada.Ovo stablo predstavlja osnovu za generisanje ciljnog koda.

115.Načini memorisanja apstraktnog sintaksnog stabla

Apstraktno sintaksno stablo se može memorisati kao statička ili dinamička struktura.

Prilikom generisanja strukture mogu da se izvršavaju i procedure za redukciju stabla tako da se ne generišu novi čvorovi ako u stablu već postoje odgovarajući. U tom slučaju dobija
se struktura dijagrama u okviru koje do nekih čvorova vodi više pokazivača

U slučaju statičke implementacije umesto pokazivača u poljima slogova pamte se adrese odgovarajućih slogova sa kojima su povezani.

116. Struktura jflex specifikacije

Format JFlex specifkacije je:
<korisnički_kod>
%%
<opcije_i_direktive>
%%
<pravila>

Korisnički kod sadrži java naredbe koje se prepisuju na početak generisanog fajla. Tu se uglavnom pišu import i package naredbe. Opcije i deklaracije sadrže:
- opcije za podešavanje generisanog leksera:
    JFlex direktive i
    Java kod koji se uključuje u različite delove leksičkog analizatora (Java klase koja se generiše);
- deklaracije posebnih početnih stanja
- defnicije makroa.

117. Nabroj I objasni osnovne jflex direktive

JFlex direktive koje se najčešće koriste su sledeće.
- Direktive kojima se podešavaju parametri generisane klase leksičkog analizatora:
%class <ImeKlase > - definišee 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 nasleđuje navedenu klasu.

- Direktive kojima se podešavaju parametri funkcije koja vrši izdvajanje sledeće reči iz ulaznog teksta:
%function <ImeFunkcije > - definišee 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.
%char - uključuje brojač karaktera u izvornom fajlu.
%column - uključuje brojač karaktera u tekućoj liniji.

- Direktive kojima se nalaže generisanje samostalne aplikacije (tj. generisanje main metoda):
%debug - generiše se main metod koji poziva leksički analizator sve dok se ne dođe 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 dođe do kraja ulaznog fajla. Ispravno prepoznati tokeni se ignorišu, a na standardni izlaz se ispisuju neprepoznati delovi ulaznog fajla.

118. Definicija specijalnih početnih stanja u jflex specifikaciji i njihova uloga.

Prilikom prepoznavanja svake nove reči, polazi se od unapred denisanog početnog stanja konačnog automata. Po podrazuemvanim podešavanjima, početno
stanje generisanog konačnog automata je stanje YYINITIAL. 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 denisana specijalna početna stanja - za njih treba definisati
prelaze za svaki simbol ulazne azbuke. Simbolička imena takvih stanja se u jflex
specikaciji navode u liniji koja počinje sa %x.

-Parcijalno denisana 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 YYINITIAL. Za prepoznavanje onih reči koje imaju isto značenje i u jednom i
u drugom slučaju koristiće se uzorci navedeni za podrazumevano početno
stanje. Simbolička imena takvih stanja se u jflex specikaciji navode u liniji
koja počinje sa %s.

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

119. Jflex pravila.

Pravila u JFlex specikaciji se denišu na isti način kao i u Lex-u. Jedina razlika u tome je što akcija mora da bude navedena kao blok naredbi (između zagrada {} i kada ona sadrži samo jednu Java naredbu).

Minimalni sadržaj jflex specikacije je:
%%
<pravila>

Jedno pravilo jflex specikacije sadrži dva dela:

- uzorak - regularni izraz koji deniše sekvencu simbola koja se traži u ulaznom tekstu;
- akciju - kod koji će se izvršiti kada se navedeni uzorak prepozna. Akcija se od uzorka odvaja pomoću {}.

120. Metakarakteri koji se koriste u regularnim izrazima u jflex-u.

`+`	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}`	prethodni 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 izraza. 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 počinje od navedenog stanja

121. Struktura parsera generisanog pomoću cup-a.

Sintaksni analizator koji se generise pomo¢u Cup-a ima dva dela:

-fiksni deo koji je defnisan u paketu **java_cup.runtime** koji je nezavistan od gramatike jezika za koji se prevodilac piše i

-promenljivi deo koji je zavistan od gramatike jezika koji se generiše na osnovu Cup specifkacije

Klasa **java_cup.runtime** sadrži sledeće :

-Klasu **Symbol** koja služi za predstavljanje podataka o svim simbolima gramatike.

-Interfejs **Scanner** - klasa koja služi za leksičku analizu mora da impelmentira ovaj interfejs

- Klasu **lr_parser** čije su najbitnije funkcije:

public Symbol parse()

public Symbol debug_parse()

Generisani deo aplikacije (zavistan od jezika) sadrži:

- klasu **sym** koja sadrži definicije tipova terminalnih simbola (celobrojne identifkatore tokena);
- klasu **parser** (čije ime može biti i promenjeno) koja je izvedena iz klase **lr_parser** i koja sadrži funkcije za inicjalizaciju potrebnih tabela.

Privatnu klasa CUP$action koja sadrži definicije akcija (akcije koje korisnik definiše u Cup specifkaciji). Najbitinija funkcija ove klase je CUP$do_action()

122. Struktura cup specifikacije.

Cup specifkacija po svojoj strukturi malo odstupa od strukture koju imaju Lex, Yacc i JFlex specifkacija. Delovi cup specifikacije se međusobno ne odvajaju simbolom %%, i ključne reči ne počinju simbolom %.

Delovi Cup specifkacije su:

- import sekcija - Import sekcija sadrži import i package naredbe koje se prepisuju na početak fajla sa kodom analizatora i njihova sintaksan ispravnost se ne proverava.,
- korisnički kod,
- liste terminalnih i neterminalnih simbola - Liste terminalnih i netermianlnih simbola sadrže definicije svih simbola koje gramatika sadrži ((terminal ili non terminal) [<ImeKlase >] <ImeSimbola1 >, <ImeSimbola2 >, ... ;),
- prioriteti i asocijativnosti operatora – asocijativnost->(precedence (left ili right ili nonassoc) <ImeSimbola1 >, <ImeSimbola2 >, ... ;), prioritet-> zavisi od redosleda navođenja

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

Ovi delovi u specifkaciji mora da budu napisani u redosledu kojim su ovde navedeni.

123. Definicija gramatike u cup specifikaciji.

Opis gramatike sadrži (opciono) definiciju startnog simbola gramatike i (obavezno) definicje njenih smena. Format definicija startnog simbola gramatike je:
start with <ImeSimbola >;
Ukoliko je defnicija 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 defnišu 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 specifkaciji (koristi se **|** simbol).
Za oporavak od grešaka se takođe koristi ista notacija kao u Yacc specifikaciji (uvode se pomoćne smene sa error simbolom)

124. Definisanje prioriteta i asocijativnosti operatora u cup specifikaciji.

Asocijativnost operatora:

Asocijativnosti operatora se definišu deklaracijama

precedence left <ImeSimbola1 >, <ImeSimbola2 >, ... ;

precedence right <ImeSimbola1 >, <ImeSimbola2 >, ... ;

precedence nonassoc <ImeSimbola1 >, <ImeSimbola2 >, ... ;

Prioritet operatora:

Prioritet operatora je, kao i u Yacc specifikaciji, određen redosledom navođenja

njihovih asocijativnosti.

125. Detekcija i prijava sintaksnih grešaka u cup specifikaciji.

Za oporavak od grešaka se koristi ista notacija kao u Yacc specifikaciji
(uvode se pomoćne smene sa error simbolom)

Ukoliko se u kodu pojavi bilo kakva sintaksna greska, poziva se funkcija
void **syntax_error**(Symbol currentToken),
a zatim se pokušava da se izvrši oporavak od grške. Ukoliko oporavak ne uspe, poziva se funkcija:

void **unrecovered_syntax_error**(Symbol currentToken).

Funkcije **syntax_error** i **unrecovered_syntax_error** su defnaisane u klasi **lr_parser** i one samo pozivaju odgovarajuće metode za prikaz poruke o grešci.

Metode za prikaz poruka o greškama su sledeće :

-public void **report_error**(string message, object info) – za prikaz poruke o sintaksnoj grešci.

-public void **report_fatal_error**(string message, object info) – za prikaz poruke o grešci kada oporavak nije moguć

126. Korišćenje atributa simbola u cup specifikaciji.

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.

Npr ako definisemo neterminalni simbol Izraz na sledeći način:

non terminal Integer Izraz;

u sledećoj smeni, atribute simbola koristimo na sledeći način :

```
Izraz ::= Izraz:i1 PLUS Izraz:i2
{:
    RESULT = new Integer(i1.intValue() + i2.intValue());
:}
```

127. Tabela simbola.

Ovo ima negde gore valjda al evo malo drugačije.

Tabele simbola su strukture podataka koje u programskim prevodiocima čuvaju
informacije o simboličkim imenima koja se koriste u programu. Kako se simbolička imena mogu koristiti za imenovanje različitih tipove entiteta u programu
(promenljivih, konstanti, funkcija, korisni£kih tipova podataka i sl), jasno je da
su informacije koje se pamte o simboličkim imenima veoma raznorodne.
Na primer,

- za promenljive je potrebno zapamtiti
  - -ime,
  - -tip (zbog provere slaganja tipova u izrazima u kojima ućestvuje),
  - -tačku u kodu u kojoj je defnisana (zbog prijave greške ukoliko se koristi van oblasti njenog važenja),
  - -tačku u kodu u kojoj je prvi put inicijalizovana (zbog prijave greške ukoliko se vrednost promenljive koristi pre njene inicijalizacije),
  - -tačku u kodu u kojoj je poslednji put korišćena (zbog prijave upozorenja ukoliko u programu postoje promenljive koje se nigde ne koriste),
  - -itd;
- za funkcije je potrebno znati
  - -ime (pri pozivu funkcije treba proveriti da li takva funkcija uopšte postoji),
  - -tip (zbog provere slaganja tipova u izrazima u kojima se koristi),
  - -broj i tipove atrumenata (zbog provere da li se lista stvarnih argumenata slaºe sa listom ktivnih argumenata),
  - -itd.

Već iz ovog kratkog primera može se zaključiti da se tabele simbola koriste u
semantičkoj analizi u dve svrhe:

- za proveru tipova podataka koji učestvuju u izrazima (type checking) i
- za proveru opsega važenja imena (scope checking).

128. Metode za predstavljanje simbola u tabeli simbola.

Za sve navedene tipove entiteta koji se imenuju, zajednička karakteristike su
ime i tip kojem pripadaju. Kako i informacije o tipu mogu biti vrlo složene
(postoje elementarni tipovi podataka, polja, strukture, klase, itd) simboli se
predstavljaju složenim tipovima podataka koje imaju strukturu grafa. U tim
grafovima koriste se dva osnovna tipa čvorova :

- čvorovi kojima se predstavljaju tipovi (type nodes ili structure nodes) i
- čvorovi kojima se predstavljaju imenovani entiteti u programu (object nodes).

S obzirom na raznorodnost podataka koji se o entitetima i tipovima pamte u tabeli simbola, postoje dva osnovna pristupa za implementaciju ova dva tipa čvorova:
-flat pristup kod koga jedinstvena struktura podataka sadrži sve podatke koji se mogu pamtiti o svim mogućim vrstama entiteta (odnosno tipova) i
- objektno-orijentisani pristup kod koga se u apstraktnoj klasi pamti ono što je zajedničko za sve vrste entiteta (tipova), a u konkretizovanim izvedenim klasama ono što je karakteristično za konkretne vrste entiteta/tipova.

129. Strukture podataka koje se koriste za implementiranje tabele simbola (prednosti i nedostaci svake od njih).

Najčešće korišćene su lančane liste, uređena binarna stabla i rasute (hash) tabele.

Lančana lista:

- traženje simbola u ovakvoj strukturi je sporo (vrši se linearno traženje pa je prosečno vreme traženja $n/2$),
- dodavanje novih simbola je jako brzo (dodavanje se uvek vrši na početak liste),
- brisanje simbola je, takođe, brzo (uvek se brišu najpre simboli sa početka liste)

-najčešće se koriste u situacijama kada očekivan broj simbola u tabeli nije veliki

Uređeno binarnon stablo :

- traženje simbola je brže nego u slučaju lančane liste (prosečno vreme traženja u uređenom binarnom stablu je $\log_2 n$, ali, s obzirom da je ovako kreirano binarno stablo prilično neizbalansirano, to vreme može biti i mnogo duže),
- dodavanje simbola je sporije (jer se prvo vrši traženje mesta pa je i kompleksnost dodavanja simbola takođe $\log_2 n$)
- brisanje simbola jako sporo (vrši se najpre traženje simbola koji treba da budu obrisani po celom stablu, a onda i samo brisanje)

- utrošak memorijskog prostora je veće nego kod lančane liste jer se uz svaki element pamte po dva pokazivača na potomke

Rasuta tablica :

- traženje simbola je izuzetno brzo (≈ 1),
- dodavanje simbola takođe brzo (≈ 1),
- brisanje simbola sporo (vrši se najpre traženje simbola koji treba da budu obrisani u celoj tabeli, dok je samo uklanjanje simbola brzo) i
- utrošak memorijskog prostora je veći nego kod lančane liste jer u hash tabeli uvek ostaje dosta nepopunjenih polja.

130. Pristupi u implementaciji tabele simbola za jezike sa ugnježdenim opsezima važenja imena.

Ako se uzme u obzir postojanje opsega, traženje imena u tabeli simbola bi trebalo modifkovati tako da se prvo vrši traženje u poslednjem otvorenom opsegu, pa tek ako se tu ime ne nađe, traži se u spoljašnjem, pa u prethodnom, itd. To znači da se u tabeli simbola mora da čuva i informacija o tome u kom opsegu je simbol definisan. Ovaj problem se rešava na dva načina:

1. svi simboli se smeštaju u jedinstvenu tabelu simbola pri čemu se uz svaki simbol pamti i nivo opsega na kojem je definisan i po zatvaranju bloka iz tabele se brišu svi simboli defnisani u njemu, ili
2. za svaki opseg važenja imena se pravi posebna tabela simbola i one se smeštaju u stek i po zatvaranju bloka za koji je opseg vezan, iz steka se izbacuje tabela koja odgovara tom opsegu