SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1314

JEZIČNO ORJENTIRANA RAZVOJNA OKOLINA

Željko Švedić

Zagreb, lipanj 2002.

Ovaj rad izrađen je na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave Fakulteta elektrotehnike i računarstva, Sveučilište u Zagrebu.

Mentor: Prof. dr. sc. Siniša Srbljić

Radnja ima 61 list i priloge.

SADRŽAJ

1.

UVOD

Veliki problem s kojim se suočava današnja računalna industrija jest dugotrajnost razvoja složenih programskih rješenja. Razvojni ciklusi mjere se u godinama, zahtijevaju opsežno planiranje te zahtijevaju velike ljudske resurse koje isključivo čine visoko kvalificirani uposlenici. Rezultat toga je često nekonkurentnost na tržištu, bilo zbog visoke cijene krajnjeg produkta, bilo zbog prekasnog izdavanja završnog produkta.

Taj problem je uočen od strane industrije i akademske zajednice, te ga se već duže vrijeme pokušava razriješiti ili značajno umanjiti. Navest ću par područja istraživanja koje adresiraju taj problem. *Metodologija modeliranja programske podrške* usredotočuje se na specifikacijsku stranu problema. Cijelo programsko rješenje detaljno se specificira korištenjem različitih specifikacijskih dijagrama koji prikazuju različite aspekte problema. Dobiva se jedinstvena predodžba o rješenju i načinu rješavanja nekih problema tijekom procesa razvoja. Izradi samoga rješenja pristupa se nakon specifikacije i raspodjele zaduženja između dijelova razvojnog tima. Takva metodologija bitno smanjuje vjerojatnost konceptualnih, arhitektonskih i programskih pogrešaka, te omogućava nedvosmislenu komunikaciju između dijelova razvojnog tima. *Objektno orjentiran pristup programiranju* usredotočuje su na maksimalno iskorištenje postojećeg programskog koda. Osim toga, objektno orijentiran pristup omogućava veću apstrakciju i separaciju dijelova programskog rješenja koji imaju različite namjene. Objektno orijentirani pristup programiranju je metodologija koja se veže uz objektno orijentirane jezike (Smalltalk, Java, C++, C#, itd.). *Viši programski jezici*, čiji se razvoj neprestano nastavlja, sve više adresiraju probleme brze izrade programske podrške. Dobar primjer su deklarativni jezici specifične namjene (SQL, Prolog), u kojima se određeni problemi mogu mnogo brže riješiti nego u klasičnim, široko korištenim jezicima (Pascal, C++, Java). U razvoju viših programskih jezika primjetna je tendencija prema programskom kodu koji je sve otporniji na programerske greške, iako se to često ostvaruje nauštrb brzine i veličine rezultirajućeg rješenja. Primjer su memorijski pokazivači, koji se u pravilu pokušavaju zaobići u svim novijim programskim jezicima.

Ovaj rad se bavi problematikom izrade *jezično orjentirane razvojne okoline*. Razvojne okoline omogućavaju lakši razvoj i skraćivanje razvojnog ciklusa, a adresiraju bitno drugačije probleme od gore

spomenutih metodologija. Temeljni problem na koji se usredotočuju jest problem učinkovite interakcije između korisnika (programera) i razvojnih alata koji se koriste za izgradnju određenog programskog rješenja. Razvoj programa je iterativan proces, u kojem programer neprestano dograđuje novu funkcionalnost te onda isprobava njenu ispravnost. Učinkovitost tog procesa je nužan preduvjet za brzi razvoj programske podrške.

Jezično orjentirane razvojne okoline su takve razvojne okoline koje razumiju leksičku, sintaksnu i semantičku strukturu određenog programskog jezika. Zbog toga one mogu pružiti mnoge informacije o programu tijekom samog postupka uređivanja programskog koda, te omogućavaju interakciju s korisnikom koja inače nije moguća. Neke od prednosti koje pružaju jezično orjentirane razvojne okoline su: označavanje različitih tipova leksičkih jedinki, označavanje sintaksno neispravnih dijelova programskog koda, dinamičko osvježavanje dijagrama klasa i drugih bitnih programskih struktura, automatsko dopunjavanje koda, učinkovito interpretiranje, inkrementalno prevođenje, te semantički svjesno dopunjavanje koda.

Problematika izrade učinkovite jezično orijentirane razvojne okoline je izuzetno složena. Postojeće metode analize programskih jezika, iako su i same složene i teške za implementaciju, nisu upotrebljive. Postojeće metode analiziraju datoteke s programskim kodom linearno i u cijelosti, te su zbog toga vremenski prezahtjevne. Interaktivnost se može ostvariti korištenjem inkrementalnih postupaka analize, koji analiziraju samo promijenjene dijelove ulaznih datoteka. Međutim, inkrementalni postupci analize su bitno složeniji od neinkrementalnih, a postoje i neka ograničenja njihove primjene.

Detaljno su opisani postupci izrade inkrementalnog leksičkog analizatora, te inkrementalnog sintaksnog analizatora. Priložen je primjer jezično orijentirane razvojne okoline za jedan jednostavan programski jezik koja je izrađena korištenjem opisanih inkrementalnih analizatora. Zbog svoje složenosti, problematika inkrementalne semantičke analize nije obrađena unutar ovog diplomskog rada.

2.

JEZIČNO ORJENTIRANE RAZVOJNE OKOLINE

2.1. Uvod u razvojne okoline

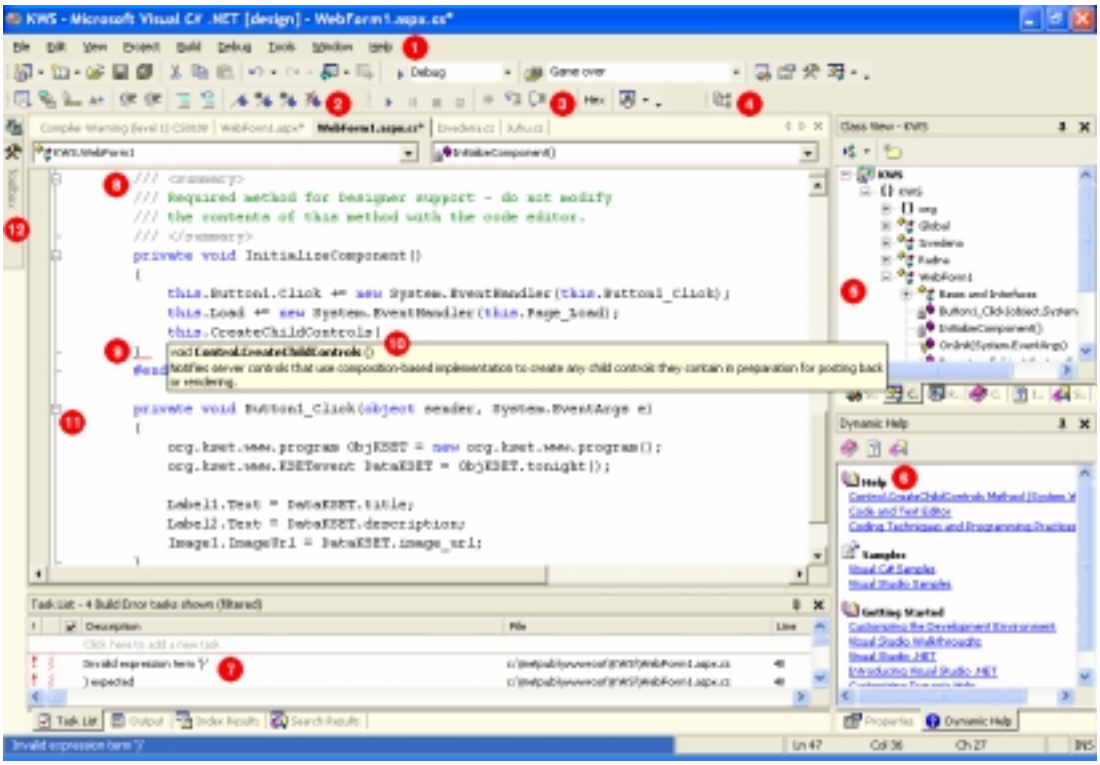
Zbog velike složenosti razvoja programske podrške, razvojne okoline uobičajen su dodatak koji integrira i olakšava uporabu osnovnih razvojnih alata. Potrebno je točno odrediti značenje pojma razvojne okoline:

Def. 1. *Razvojna okolina* jest program ili skup programa koji olakšavaju razvoj drugih programa u nekom višem programskom jeziku, a nisu nužno potrebni za stvaranje tih programa.

Iako razvojne okoline nisu nužno potrebne, one su u pravilu dio svakog komercijalnog paketa za razvoj programskih rješenja. Bez učinkovitih i integriranih razvojnih okolina, nemoguć je brzi razvoj aplikacija tj. RAD (engl. Rapid Application Development).

Razvojne okoline su u stalnom razvoju koji je uzrokovan sve većim zahtjevima korisnika i potrebama za smanjenje trajanja razvojnog ciklusa. Na slici 2.1 prikazan je primjer moderne razvojne okoline [9]. Ukratko će biti navedeni njeni bitni dijelovi i osobine. Posebno su naglašena ona svojstva koja su osobine jezično orjentiranih razvojnih okolina.

Glavni prozor razvojne okoline sa slike podijeljen je na više dijelova. Najveću površinu zauzima jezično orjentiran uređivač teksta (8, 9, 10, 11). Manje dijelove zauzimaju prozori koji omogućavaju kretanje kroz projekt (5), ispisuju svojstva pojedinih objekata i dinamičku pomoć (6), te ispisuju rezultat prevođenja (7). Budući da broj otvorenih prozora u razvojnoj okolini može biti velik, korisna je mogućnost automatskog skrivanja prozora koji se ne koriste (engl. auto hide), koji su onda smanjeni na traku s nazivom (12). Naravno, tu je i glavni izbornik (1), te trake s alatima (2, 3, 4). Slijedi detaljniji opis svakog od nabrojanih elemenata.



Slika 2.1: Primjer moderne razvojne okoline (Visual Studio .NET)

Glavni izbornik (1) jezično orjentirane razvojne okoline je kontekstno ovisan. Stavke izbornika mijenjaju se ovisno o kontekstu dokumenta koji se obrađuje unutar razvojne okoline (program, slika, forma, HTML stranica). Trake s alatima su korisnički podesive, te su također konteksno ovisne. Traka za navigaciju (2) omogućava brzo kretanje kroz bitna mjesta u aktivnom projektu. Traka za pronalaženje grešaka u programu (3) omogućava izvršavanje programa korak po korak, te ispisivanje potrebnih informacija o stanju programa. Traka za modeliranje i dokumentiranje programa (4) omogućava projektiranje i automatsku dokumentaciju korištenjem UML modela (engl. Unified Modeling Language). Prozor s opisom aktivnog projekta (5) iskorištava prednosti jezično orjentirane razvojne okoline. Sve promjene u strukturi programa koje su uzrokovane korisnikovim akcijama u uređivaču teksta istodobno se odražavaju u prozoru s opisom trenutnog projekta. Upravo utipkana programska metoda istodobno se pojavljuje u pripadnom prozoru. Klikom miša na neku od programskih struktura u prozoru, uređivač teksta postavlja se na početak te strukture. Prozor koji ispisuje dinamičku pomoć (6) također iskorištava informacije dostupne u jezično orjentiranoj razvojnoj okolini. Taj prozor ispisuje pomoć koja je kontekstno ovisna o trenutnoj poziciji pokazivača unosa teksta u uređivaču teksta. U primjeru sa slike, prozor dinamičke pomoći je ispravno ispisao pomoć za metodu koja se upravo unosi u uređivaču teksta. Ispravan prikaz dinamičke pomoći moguć je samo u semantički svjesnoj razvojnoj okolini. Prozor s rezultatima prevođenja (7) ispisuje upozorenja i

pogreške u aktivnom projektu. Odabirom pojedine linije s izvještajem o upozorenju ili pogrešci, uređivač teksta postavlja se na red ulazne datoteke koji je uzrokovao upozorenje ili pogrešku.

Najveće prednosti jezično orjentirane razvojne okoline pokazuju se u uređivaču teksta. U uređivaču teksta bojama se označavaju različite jezične strukture. Razvojna okolina prepoznaje i pamti XML (engl. Extensible Markup Language) komentare (8). Prilikom upisivanja neke metode u uređivaču teksta, kao pomoć ispisuje se deklaracija pripadne metode te njen opis (10) sadržan u odgovarajućem polju XML komentara koji je upisan neposredno prije te metode u ulaznoj datoteci. Sintaksno neispravni dijelovi ulazne datoteke označavaju se valovitom crvenom linijom (9). Jezično orjentirana razvojna okolina raspolaže informacijama o svim jezičnim strukturama unutar ulazne datoteke, tako da se unutar uređivača teksta bitne strukture mogu proširiti ili sažeti po potrebi (11). Osim onoga nabrojanog i prikazanog na slici 2.1, potrebno je spomenuti još par korisnih osobina koje posjeduje jezično orjentirana razvojna okolina. *IntelliSense*^{TM} izbornik otvara se prilikom pristupa metodi ili varijabli objekta. On omogućava brzi odabir metode ili varijable nekog objekta, te time povećava brzinu unosa programskog koda i služi kao automatska pomoć. Završavanjem unošenja jezične strukture u razvojnoj okolini sa slike 2.1, cijela struktura se na par sekundi posebno istakne. To pomaže snalaženju u programskom kodu, jer se lako uočavaju pripadni blokovi, oble, uglate i vitičaste zagrade te navodnici. Po prelasku u novi red pokazivač unosa teksta se pozicionira na mjesto koje odgovara pravilima poravnavanja programskog koda za taj jezik. Za sva navedena svojstva je odgovorna jezično orjentirana razvojna okolina unutar koje se pozadinski analiziraju novonastale promijene u ulaznim datotekama. Osvježavanje je za korisnika trenutno, pa je tako razvojna okolina prividno istodobno odgovara na akcije korisnika. Time se ostvaruje kvalitetnija komunikacija između korisnika i razvojnih alata, te se razvojni ciklus skraćuje.

Za razumijevanje problematike izgradnje jezično orjentiranih razvojnih okolina nužno je razumijevanje problematike jezičnih procesora. Jezični procesor posebne namjene je osnovni dio takve razvojne okoline. Dizajn jezičnog procesora utječe na dizajn svake razvojne okoline, a pogotovo jezično orjentirane. U slijedećem dijelu ukratko su izloženi osnovni pojmovi vezani uz jezične procesore. Također je opisana interakcija između jezičnog procesora i razvojne okoline.

2.2. Uvod u jezične procesore

Zbog širokog područja primjene, potrebno je točno odrediti što se smatra jezičnim procesorom [1]:

Def. 2. *Jezični procesor* jest program koji na temelju formalnih pravila izvornog i formalnih pravila ciljnog jezika prevodi izvorni jezik u ciljni jezik. Uvodi se slijedeća oznaka:

$$JP_{jG}^{Ij \to Cj}$$

gdje je JP jezični procesor, *Ij* jest izvorni jezik, *Cj* jest ciljni jezik, te *jG* koji je jezik izgradnje jezičnog procesora koji je izvodljiv na dostupnom računalu, bilo izravno (računalo ga samostalno izvodi), bilo da je moguća pretvorba u izvodljiv oblik.

U užem smislu se pod jezičnim procesorima podrazumijevaju prevoditelji viših programskih jezika (C, C++, PASCAL, BASIC, Java).

Svi jezični procesori proces prevođenja dijele u neke zajedničke korake. Temeljna podjela je na proces analize i proces sinteze.

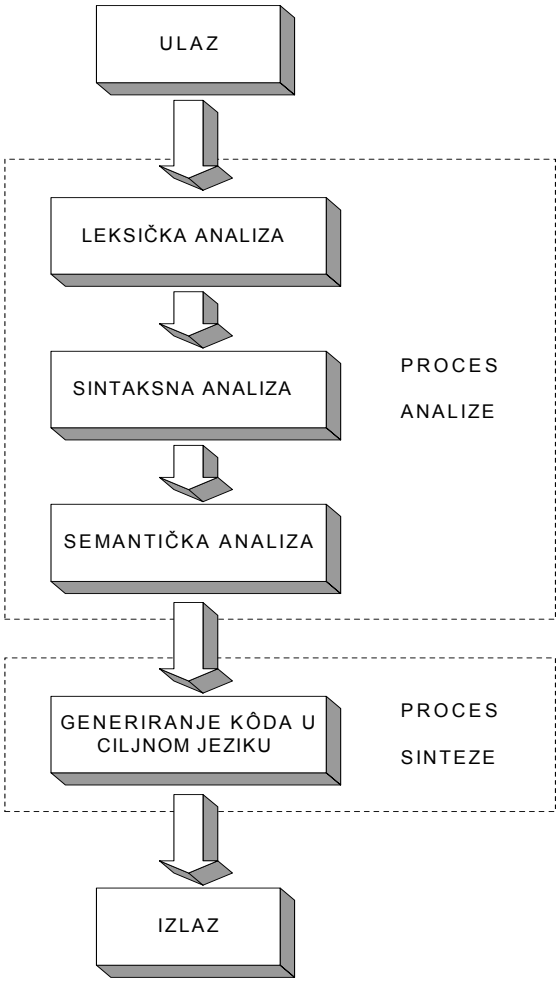
Tijekom procesa analize prikupljaju se podaci o programu napisanom u izvornom jeziku *Ij*, provjerava se ispravnost ulazne datoteke s obzirom na izvorni jezik, te se program pretvara u neki unutrašnji zapis koji je pogodan za daljnju obradu.

Tijekom procesa sinteze se na temelju svih prikupljenih podataka generiraju naredbe ciljnog jezika *Cj*.

Sinteza je u pravilu zahtjevnija za izvođenje i složenija od analize, no nije usko vezano uz temu ovoga rada. Zbog toga je detaljnije izložen proces analize. Na slici 2.2 prikazani su osnovni koraci u radu jezičnog procesora.

Leksička analiza prevodi ulaznu datoteku u niz leksičkih jedinki s pridruženim svojstvima. Leksičke jedinke su grupacije znakova ulaznog jezika koje su grupirane po nekome uvjetu. Uvjet grupiranja ovisi o leksičkim pravilima izvornog jezika. Leksičke jedinke mogu biti nazivi varijabli i funkcija, brojčane konstante, znakovne konstante, komentari u izvornom jeziku, itd. Leksička analiza nije uvijek nužno potrebna. Ako ne postoji korak leksičke analize, onda sintaksna analiza obuhvaća obradu znakova ulaznog jezika. Obično se provodi odvojen korak leksičke analize jer je u tom slučaju sintaksna analiza pojednostavljena.

Sintaksna analiza na temelju leksičkih jedinki i gramatike izvornog jezika gradi sintaksno stablo. Sintaksno stablo jest struktura koja u potpunosti predočava gramatički bitna svojstva izvornog jezika u obliku koji je pogodan za daljnju obradu u jezičnom procesoru. Sintaksna analiza je neovisna o semantičkoj analizi ili radi u sprezi sa semantičkom analizom. Neovisno izvedena sintaksna analiza gradi cijelo sintaksno stablo i prosljeđuje ga semantičkom analizatoru. Ako sintaksna analiza radi u sprezi sa semantičkom analizom, onda se izgrađuju samo dijelovi sintaksnoga stabla koji se odbacuju nakon što se na njima provede i semantička analiza.

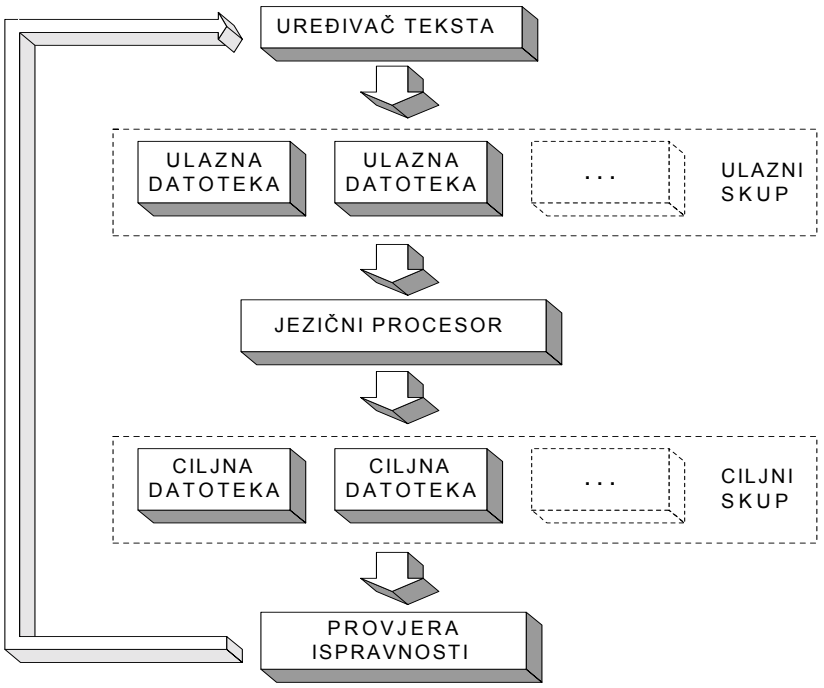


Slika 2.2: Koraci u radu jezičnih procesora

Semantička analiza čvorovima sintaksnoga stabla pridružuje semantička svojstva. Semantička svojstva ovisna su o izvornom jeziku i omogućavaju potpuno razumijevanje ulaznog programa. Čvorovi sintaksnog stabla imaju proizvoljan broj pridruženih semantičkih svojstava. Na primjer, čvor koji predstavlja brojčanu konstantu u izvornom jeziku ima pridruženo semantičko svojstvo koje ima vrijednost te brojčane konstante. Semantička svojstva prenose se po sintaksnom stablu ovisno o semantičkim pravilima izvornog jezika.

2.3. Uredi-prevedi-izvedi ciklus

Razvoj programa ostvaruje se kao niz *uredi-prevedi-izvedi* (engl. edit-compile-run) ciklusa. Uredi-prevedi-izvedi ciklus prikazan je na slici 2.3.



Slika 2.3: Uredi-prevedi-izvedi ciklus

Prvi korak je pisanje izvornog programa u jednoj ili više ulaznih datoteka. Korisnik uređuje datoteke u proizvoljnom uređivaču teksta. Po završetku uređivanja, ulazne datoteke pohranjuju se na neku računalnu jedinku za stalnu pohranu podataka.

Nakon uređivanja, ulazne datoteke prosljeđuju se jezičnom procesoru. Ako je ulazni program ispravan s obzirom na leksička, sintaksna i semantička pravila, pokreće se prevođenje. Rezultat prevođenja je jedna ili više ciljnih datoteka koje sadrže naredbe ciljnog jezika.

Jezični procesor prevodi izvorne u ciljne datoteke jedino ako su izvorne datoteke *jezično ispravne*. Za dobivene ciljne datoteke potrebno je provjeriti *logičku ispravnost* i *uporabnu ispravnost*. Program je logički ispravan ako daje ispravne rezultate za sve ulazne podatke. Ne postoje učinkoviti postupci provjere logičke ispravnosti programa, te je ona gotovo uvijek prepuštena korisniku. Program je uporabno ispravan ako je njegova učinkovitost (vremenska, memorijska) unutar zadanih granica.

Ako je program neispravan, razvoj se vraća na početak ciklusa. Korisnik u uređivaču teksta unosi one promjene u ulazne datoteke za koje smatra da će dovesti do ispravnog programa u slijedećem koraku prevođenja.

Ovaj ciklus se tijekom razvoja programa neprestano ponavlja. Programi se rijetko kada razvijaju u cijelosti. Kada bi se program razvijao u cijelosti pa tek tada ispitivao, prva ciljna inačica bi sadržavala previše grešaka za učinkovitu provjeru i otkrivanje grešaka. Uobičajen postupak je izgradnja prototipa

programa u koji se korak po korak dodaju pojedini elementi. Razvoj se sastoji od niza nadogradnji na inačice programa koje su u danom trenutku nedovršene, ali ispravno rade za neki podskup problema koji se rješava. Nakon otkrivanja greške u upravo prevedenoj inačici programa, moguće je sa velikom vjerojatnošću pretpostaviti da je uzrok greške upravo jedan od zadnje ugrađenih ili promijenjenih elemenata izvornog programa. Uredi-prevedi-izvedi ciklus ponavlja se više tisuća puta kod razvoja složenih programa.

2.4. Postojeća rješenja i njihovi nedostaci

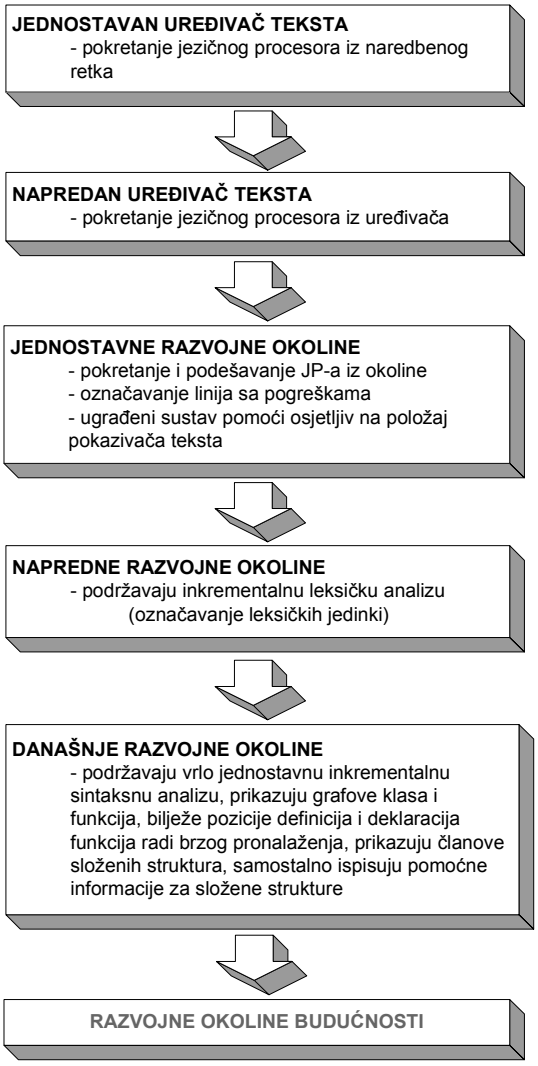
Kao što je već rečeno u uvodnom dijelu (Def. 1), za razvoj programa u nekom programskom jeziku nužno je potreban samo jezični procesor. Razvojna okolina olakšava korištenje jezičnog procesora i ubrzava razvoj programske podrške.

Razvojne okoline su u stalnom razvoju koji je uzrokovan sve većim zahtjevima korisnika i potrebama smanjenja razvojnog ciklusa. Na slici 2.4 prikazani su koraci u razvoju razvojnih okolina.

Današnje razvojne okoline omogućavaju podešavanje jezičnog procesora kroz jednostavno sučelje, označavaju redove ulaznog programa u kojima je jezični procesor prijavio grešku i brinu se za prosljeđivanje svih potrebnih ulaznih datoteka jezičnom procesoru. U sebi imaju ugrađenu inkrementalnu leksičku analizu koja različite jedinke označava različitom bojom. Imaju jednostavnu sintaksnu analizu koja pamti položaj bitnih deklaracija radi kasnijeg bržeg pronalaženja.

Radi smanjenja vremena prevođenja, koristi se poboljšani uredi-prevedi-izvedi ciklus. Osim datoteka s izvornim programom, jezičnom procesoru prosljeđuje se i datoteka u kojoj su prikazane međuovisnosti datoteka izvornog programa (engl. makefile). Jezični procesor analizira koje su se ulazne datoteke promijenile u odnosu na prethodni ciklus prevođenja. Prevode se promijenjene ulazne datoteke, kao i one ulazne datoteke koje ih posredno koriste. Time se u jednom uredi-prevedi-izvedi ciklusu prevodi manji broj ulaznih datoteka i smanjuje se vrijeme prevođenja. Za ostvarenje navedenog pristupa potreban je dodatan program *povezivač* (engl. linker). Jezični procesor prevodi svaku ulaznu datoteku u odgovarajuću ciljnu datoteku, koja je samo djelomično izvodljiva, jer joj nedostaju pozivi naredbi čiji se ciljni kôd nalazi u drugim datotekama. Program povezivač spaja te djelomično izvodljive ciljne datoteke (engl. object files) u završne, potpuno izvodljive ciljne datoteke.

Bitni dio razvojnih okolina je program za pronalaženje grešaka (engl. debugger). On omogućava izvođenje programa u koracima koji odgovaraju naredbama izvornoga jezika, te pruža uvid u stanja pojedinih elemenata programa (npr. sadržaj varijabli), kao i pojedinih elemenata računala na kojem se program izvodi (npr. sadržaj mikroprocesorskih registara). Navedeni podaci značajno olakšavaju pronalaženje logičkih grešaka u programu.



Slika 2.4: Koraci u razvoju razvojnih okolina

Uobičajeni dio razvojnih okolina su i programi za mjerenje učinkovitosti programa (engl. profiler). Oni pružaju uvid u vremenske (rjeđe memorijske) zahtjeve pojedinih dijelova programa, čime se bitno olakšava pronalaženje neučinkovitih dijelova izvornog programa.

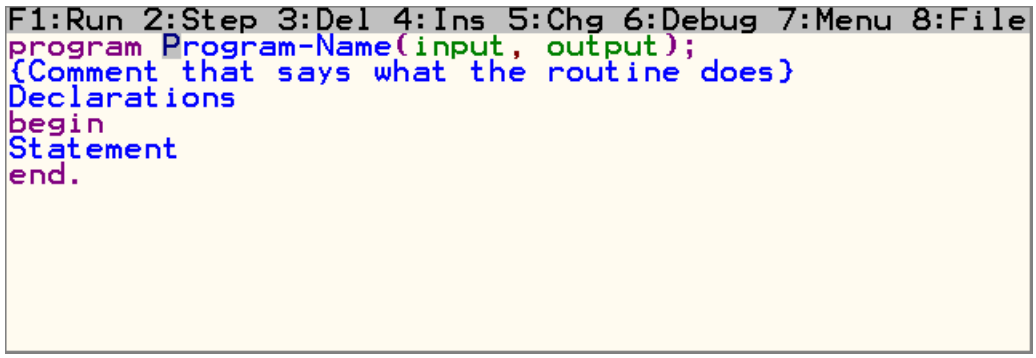
Današnje razvojne okoline nisu u potpunosti jezično svjesne. Izvorni program uređuje se kao običan tekst, što otežava i usporava neke česte operacije nad izvornim programom. Korisnik je prisiljen kopirati i izrezivati dijelove programa kao nizove znakova, a ne kao jezične strukture. Razvojna okolina ne upozorava korisnika na leksičke i sintaksne greške. Razvojna okolina mu ne pruža informacije koje bi mu bile od velike pomoći i ubrzale uređivanje (npr. označavanje pripadne zatvorene zagrade za neku otvorenu zagradu i obratno). Grafikoni bitnih programskih struktura (klase, funkcije, procedure, globalne varijable), koji imaju namjenu bržega kretanja kroz izvorni program, često su neispravni ili

prikazuju stare vrijednosti. Uređivanje izgleda programa od strane okoline (npr. odmicanje od početka reda u zavisnosti o ugniježđenosti naredbe) je često neispravno i ne odražava pravu strukturu ulaznog programa. Neke česte operacije, kao što je promjena imena lokalne varijable neke funkcije, previše su složene. Za rješavanje svih navedenih problema potrebno je izraditi okolinu koja je jezično svjesna.

Jezično orjentirane razvojne okoline bitno utječu na uredi-prevedi-izvedi ciklus. Ciklus prevođenja je u običnoj razvojnoj okolini jedini način za provjeru leksičke, sintaksne i semantičke ispravnosti. Jezično orjentirana razvojna okolina integrira u sebe jedan dio procesa prevođenja, te se time uređivanje i prevođenje velikim dijelom preklapaju. Također, dostupnost sintaksnih i semantičkih informacija tijekom uređivanja omogućava izradu učinkovitog interpretatora za neki interpretatorski jezik. Kada korisnik izda naredbu za izvođenje programa, potrebne sintaksne i semantičke strukture su dostupne, te se odmah prelazi na izvođenje programa. Time se odvojeni koraci uredi-prevedi-izvedi ciklusa stapaju u jedan, te se ispunjava zahtjev potpune integriranosti razvojnog procesa.

Postoji više načina da se razvojna okolina učini jezično svjesnom. Izložena su dva od raznih postojećih pristupa.

Sintaksom upravljani uređivači teksta (engl. syntax-directed editors) su uređivači teksta koji ne dozvoljavaju korisniku unos sintaksno neispravnih struktura, a olakšavaju mu unošenje sintaksno ispravnih struktura [2]. Bili su naročito popularni ’80 tih godina prošloga stoljeća, no nakon tog razdoblja se sve manje pronalaze u uporabi. Nastali su s prvenstvenom namjenom da olakšaju programiranje početnicima i uklone pojavljivanje sintaksnih greški. Međutim, zbog toga što u svome radu održavaju sintaksno stablo, mogu se iskoristiti i za već razmatrana jezično orjentirana poboljšanja razvojnih okolina.



Slika 2.5: Alice PASCAL, stvaranje novoga izvornog programa

Primjer takve okoline je *Alice PASCAL* [3] koji je razvijen 1985. Alice koristi isti pristup uređivanju kao i drugi sintaksom upravljani uređivači teksta. Svako uređivanje izvornog programa odvija se preko *podložaka* (engl. template). Podložak je primjerak pojedine sintaksne strukture kojemu se mijenjaju samo pojedini dijelovi. Nakon što korisnik zada naredbu za stvaranje novog izvornog programa, prvo se stvara podložak novog programa. Slika 2.5 prikazuje izgled podloška novog programa.

U podlošku novog programa su već unesene ključne riječi koje se moraju nalaziti u sintaksno ispravnom programu za PASCAL. Korisnik mijenja tekst samo u pojedinim poljima unosa. U primjeru sa slike polja unosa predstavljaju ime programa, dio programa gdje se nalaze deklaracije i dio programa gdje se nalaze naredbe glavnog tijela programa. Kako korisnik uređuje razna polja, tako se u izvorni program ubacuju potrebni podlošci. Na primjer, po ukucavanju IF ključne riječi, odmah se kreira podložak IF strukture:

IF *uvjet* THEN
BEGIN
    naredba
END;

Korisnik u podlošku mijenja polja *uvjet* i *naredba*. Moguće je i brisati cijeli podložak, te umjesto njega unijeti neki drugi. Bez obzira na korisnikove naredbe, u programu su ispunjena sintaksna pravila. Okolina (u ovome slučaju sintaksom upravljani uređivač teksta) ima u svakome trenutku na raspolaganju leksičke i sintaksne informacije o programu.

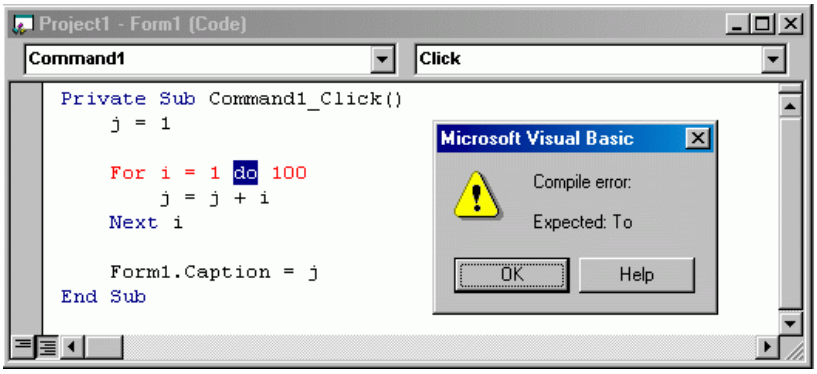
Izvedba sintaksom upravljanih uređivača teksta je olakšana, jer ne zahtjeva uobičajene postupke parsiranja. Svako polje unosa odgovara jednom čvoru sintaksnog stabla. Kada korisnik unosi tekst u polje, stvaraju se novi čvorovi-djeca u sintaksnom stablu. Brzina nije ograničavajući čimbenik, jer korisnik unosi samo nekoliko znakova u sekundi. Složeni postupci analize i parsiranja nisu nužni.

Sintaksom upravljani uređivači teksta nisu ušli u širu uporabu zbog raznih zamjerki korisnika na njihov način rada [2]. Zamjerka iskusnih programera, koji rijetko prave greške u sintaksi, jest da ih takve okoline samo usporavaju u radu. Korisniku se nameće točno određen način pisanja izvornog programa i ne dozvoljava se drugi. Operacije koje su u običnim uređivačima teksta jednostavne (npr. kopiranje teksta), u sintaksom upravljanim uređivačima postaju složene ili nemoguće. Sintaksne pogreške moraju se ispraviti odmah da bi se nastavilo uređivanje. Sintaksom upravljani uređivači teksta su obično prilagođeni i pisani za točno određeni programski jezik, te se ne mogu iskoristiti za uređivanje programa pisanih u nekom drugom izvornom jeziku.

Sintaksom upravljani uređivači teksta se i dalje koriste. Djelomičan povratak popularnosti doživjeli su pojavom raznih Internetski usmjerenih alata. Program HoTMetaL [4] je poznati predstavnik sintaksom upravljanih HTML (engl. Hypertext Markup Language) uređivača.

Retkovna jezična analiza jest postupak jezične analize tijekom uređivanja, gdje se obavlja odvojena analiza pojedinih redova. Analizira se red u kojem je završeno uređivanje. Završetak uređivanja jednoga reda događa se po prelasku pokazivača unosa teksta u neki drugi red. Retkovna jezična analiza je čest odabir u okolinama za programski jezik BASIC (Beginner's All-purpose Symbolic Instruction Code). BASIC je jednostavan programski jezik koji se gotovo uvijek prevodi interpretatorima. Naredbe BASIC-a obično se nalaze u različitim redovima tako da je pogodan za retkovnu jezičnu analizu.

Primjer učinkovite razvojne okoline i interpretatora za programski jezik BASIC je Visual BASIC [5]. U Visual BASIC-u se jezična analiza obavlja na redu iz kojega se pomaknuo pokazivač unosa teksta. Za ispravan red provodi se leksička i sintaksna analiza, te prevođenje u unutrašnji zapis interpretatora. Za neispravan red prijavljuje se pogreška u prevođenju. Slika 2.6 prikazuje primjer pogreške nakon analize neispravnog reda.



Slika 2.6: Retkovna analiza u Visual BASIC-u

Veliki uspjeh i raširenost Visual BASIC-a potvrđuju uspješnost ovoga pristupa. Zamjerke na retkovnu sintaksnu analizu ipak postoje. U slučaju leksičke ili sintaksne greške ispisuje se upozoravajuća poruka koja prekida korisnika u radu. Često korisnik želi prebaciti pokazivač unosa teksta u neki drugi red, da bi kopirao ili pogledao dio programa, te se vratio u početni red da završi sa uređivanjem. U tom slučaju je poruka o pogrešci nepotrebna i ometajuća. Slijedeća zamjerka odnosi se na to da korisnik napravljenu sintaksnu grešku uočava tek po prelasku u neki drugi red, a ne u onome trenutku kada je nastala. Zadnja zamjerka odnosi se na neprilagođenost retkovne sintaksne analize na primjenu u razvojnim okolinama ostalih viših programskih jezika, koji nisu retkovno orijentirani. Retkovna jezična analiza nije općeniti pristup i u ovome primjeru prilagođena je programskom jeziku BASIC.

Nakon pregleda postojećih rješenja izrade jezično svjesnih okolina, mogu se dati slijedeći zaključci.

Jezično svjesna razvojna okolina ne smije nametati korisniku način pisanja izvornog programa, niti ga ometati u pisanju programa. Nužno je omogućiti korisniku da piše izvorni program na proizvoljan način. Poruke o pogreškama moraju biti nenametljive korisniku razvojne okoline. Okolina treba dozvoljavati višestruke pogreške u ulaznom programu i nastavljati sa radom i nakon takvih pogrešaka.

Jezično svjesna razvojna okolina treba reagirati na sve promjene u izvornom programu u onome trenutku kada se dogode. Ako se pojavila leksička ili sintaksna pogreška, razvojna okolina treba obavijestiti korisnika o mjestu pogreške na, za korisnika, nenametljiv način. Ako je promjena ulaznog programa uzrokovala promjenu jezične strukture, ta se promjena treba odmah odraziti u unutrašnjom prikazu ulaznog programa.

Jezično svjesna razvojna okolina treba biti izgrađena korištenjem općenitih postupaka. Time se omogućava jednostavno prilagođivanje razvojne okoline uporabi proizvoljnog izvornog jezika.

2.5. Korišteni pristup

Cilj ovoga rada je opisati postupke i metodologiju koja omogućava konstrukciju jezično orjentirane razvojne okoline. Ti postupci i metodologija su maksimalno poopćeni da bi bili primjenjivi na široku klasu programskih jezika.

Temeljni dijelovi jezično orijentirane razvojne okoline su: inkrementalni leksički analizator, inkrementalni sintaksni analizator i inkrementalni semantički analizator. U ovome radu je opisan inkrementalni leksički i inkrementalni semantički analizator. Problematika izgradnje inkrementalnog semantičkog analizatora, zbog svoje složenosti i nekih još uvijek neriješenih problema, nije dotaknuta unutar ovog rada. Inkrementalni semantički analizator nije ni potreban za veliki dio korisnih funkcija jezično orijentirane razvojne okoline.

U trećem poglavlju opisan je jednostavan inkrementalni leksički analizator, koji je primjenjiv za veliki dio jezika koji zahtijevaju jednoprolaznu leksičku analizu. Jezici koji zahtijevaju višeprolaznu leksičku analizu, kao što su programski jezici C i C++, ne mogu se analizirati korištenjem opisanog inkrementalnog leksičkog analizatora.

U četvrtom poglavlju opisan je inkrementalni sintaksni analizator. On je baziran na LR(1) parseru, te je stoga primjenjiv na sve jezike koji se mogu opisati LR(1) gramatikom.

Izgrađen je pokazni program koji koristi opisane inkrementalne leksičke i sintaksne analizatore, te simulira razvojnu okolinu za jedan jednostavan programski jezik. Jezično orjentiran uređivač teksta koristi informacije dobivene od inkrementalnog leksičkog i sintaksnog analizatora za izvedbu par korisnih jezično svjesnih funkcija. Peto poglavlje opisuje pokazni program i analizira njegovu učinkovitost.

3.

INKREMENTALNA LEKSIČKA ANALIZA

U ovom poglavlju izlaže se postupak izgradnje općenitog inkrementalnog leksičkog analizatora. Prvo je izložen potreban dio teorije klasične leksičke analize. Zatim su opisana proširenja postojeće teorije koja omogućavaju inkrementalnu primjenu postupka leksičke analize.

3.1. Leksička analiza

Leksička analiza je široko područje koje zbog svoje opsežnosti nije ovdje izloženo u potpunosti. Izložen je dio nužan za razumijevanje rada inkrementalnog leksičkog analizatora. Više o navedenom području može se naći u literaturi [1,6].

3.1.1. Analiza nizova korištenjem konačnih automata

Znak jest osnovna i nedjeljiva jedinka. Primjeri znakova su brojke, slova, razni simboli itd. Skup različitih znakova naziva se *Abeceda*. Primjer abecede jest binarna abeceda `B={0,1}`. Konačan slijed znakova definiranih nad abecedom naziva se *niz znakova*. Primjeri nizova znakova nad binarnom abecedom su `"101"`, `"11100101"`, `"0"` itd. *Duljina niza* jest broj znakova u nizu. Duljina nekog niza *w* označava se sa |*w*|. *Prazan niz* jest niz znakova duljine `0`. *Neprazan niz* jest svaki niz koji nije prazan.

Osnovni zadatak leksičke analize je razdvojiti ulazni niz znakova na specifične podnizove koji se nazivaju *leksičke jedinke*. *Klasa leksičkih jedinki* je jednoznačno određen skup leksičkih jedinki, koji ne mora nužno biti ograničen (može sadržavati beskonačan broj elemenata). Svaki programski jezik se može opisati ograničenim brojem različitih *klasa leksičkih jedinki*, tj. svaka leksička jedinka ulaznog niza koji je jezično ispravan može se jednoznačno svrstati u određenu klasu leksičkih jedinki.

Deterministički konačni automat (DKA) je formalizam koji omogućava jednoznačan opis klase leksičkih jedinki.

Def. 3. *Deterministički konačni automat* (DKA) je uređena petorka

$DKA = (Q, \sum, \delta, q_0, F)$

gdje je

Q – konačan skup stanja;

$\sum$ – konačan skup ulaznih znakova;

$q_0 \in Q$ – početno stanje;

$F \subseteq Q$ – skup prihvatljivih stanja;

δ – funkcija prijelaza $Q \times \sum \rightarrow Q$.

Da bi mogli formalno opisati koje nizove znakova prihvaća određeni DKA, potrebno uvesti novu funkciju prijelaza $\hat{\delta} : Q \times \sum^* \rightarrow Q$. Oznaka $\sum^*$ označava skup svih mogućih nizova ulaznih znakova uključujući i prazan niz. Funkcija $\hat{\delta}(q, w)$ određuje stanje DKA nakon što je u stanju q učitan niz ulaznih znakova $w \in \sum^*$.

Def. 4. Funkcija prijelaza $\hat{\delta} : Q \times \sum^* \rightarrow Q$ definira se kao

(1) $\hat{\delta}(q, \varepsilon) = q$, gdje je ε prazan niz;

(2) za sve nizove ulaznih znakova w i za sve ulazne znakove α vrijedi

$$\hat{\delta}(q, w\alpha) = \delta(\hat{\delta}(q, w), \alpha), \text{ gdje je } w \in \sum^* \text{ i } \alpha \in \sum.$$

Uz proširenu funkciju prijelaza $\hat{\delta}$ pojmovi prihvaćanja i ne prihvaćanja ulaznog niza mogu se formalno definirati.

Def. 5. Deterministički konačni automat

$DKA = (Q, \sum, \delta, q_0, F)$

prihvaća podskup

$L(DKA) = \{x \mid \hat{\delta}(q_0, x) \in F\}$

skupa svih mogućih nizova ($L(DKA) \subseteq \sum^*$). Za sve ostale nizove koji nisu u skupu $L(DKA)$ kaže se da ih *DKA* ne prihvaća.

3.1.2. Postupak leksičke analize

Kao što je već prije rečeno, svaki programski jezik se može opisati ograničenim brojem različitih klasa leksičkih jedinki. Klase leksičkih jedinki opisuju osnovne strukture koje koristi taj

programski jezik: brojeve, nizovne i znakovne konstante, ključne riječi, operatore itd. Klase leksičkih jedinki se mogu opisati korištenjem *regularnih izraza* [6] ili korištenjem *konačnih automata*. U ovom radu sve su klase leksičkih jedinki opisane korištenjem pripadnih *deterministčkih konačnih automata*, pa će stoga biti opisan algoritam leksičke analize koji koristi upravo DKA. Leksički analizator linearno prolazi kroz datoteku, počevši od prvog znaka ulazne datoteke, te datoteku dijeli u podnizove. Podnizovi koji ne pripadaju nijednoj klasi leksičkih jedinki se klasificiraju kao *leksički neispravna područja*. *Leksički neispravna* ulazna datoteka jest takva datoteka u kojoj postoji barem jedno leksički neispravno područje.

Način grupiranja leksičkih jedinki ne mora biti jednoznačan. Problem nejednoznačnosti pojavljuje se kada je prefiks neke leksičke jedinke neka druga leksička jedinka. Taj problem najčešće se rješava uzimanjem najduže leksičke jedinke. Taj pristup korišten je i u izloženim algoritmima.

Leksički analizator zahtjeva poseban oblik deterministčkog konačnog automata za svoj rad. DKA se izgrađuje na slijedeći način:

(i) Za klase leksičkih jedinki k_1, k_2, … k_N se izgrade automati DKA_1, DKA_2, … DKA_N ;

(ii) Postupkom spajanja [6] automati DKA_1, DKA_2, … DKA_N se spoje u jedan jedinstveni DKA_{JED}. Dodatno, za sva stanja unutar DKA_{JED} bilježi se indeks *x* pripadne klase leksičkih jedinki k_X . Ukoliko za neko stanje DKA_{JED} postoji više pripadnih klasa leksičkih jedinki (u tom stanju pročitani ulazni niz pripada u više od jedne klase leksičkih jedinki), indeks *x* postavlja se na klasu leksičkih jedinki s najmanjim rednim brojem.

Ovako izgrađeni DKA_{JED} prepoznaje sve klase leksičkih jedinki nekog programskog jezika, a također omogućava jednoznačno određivanje pripadne klase za neki podniz ulaznog niza koji se analizira.

Leksički analizator izvodi slijedeći algoritam:

```
početak
  pocetak := 1;
  zavrsetak := 1;
  posljednji := 1;
  izraz := 0;
  ponavljaj ako nije kraj ulaznog niza
    postavi stanje automata q' u početno stanje q0;
    ponavljaj zauvijek
      slučaj TablicaIzraza[q']
        pridružen izraz i:
```

```
                izraz := i;
                posljednji := zavrsetak;
                ako je kraj datoteke tada prekini ponavljanje;
                zavrsetak := zavrsetak + 1;
                a := citaj(zavrsetak);
                q' := δ'(q', a);
        nema pridruženog izraza:
                ako je kraj datoteke tada prekini ponavljanje;
                zavrsetak := zavrsetak + 1;
                a := citaj(zavrsetak);
                q' := δ'(q', a);
           prazno stanje:
                prekini ponavljanje;
     kraj
     ako izraz = 0 tada
          ispisi_neispravno_podrucje( citaj(pocetak) );
          pocetak := pocetak + 1;
          zavrsetak := pocetak;
     inače
          napravi_leksicku_jedinku( pocetak, posljednji );
          pocetak := posljednji;
          zavrsetak := posljednji;
          izraz := 0;
     kraj
  kraj
kraj
```

U prvom dijelu se varijable koje pokazuju na određene dijelove ulaznog niza (*pocetak*, *zavrsetak*, *posljednji*) postavljaju na početne vrijednosti, tj. na prvi znak ulaznog niza. Zatim se ponavlja iterativan postupak prepoznavanja pojedinih leksičkih jedinki, sve dok se ne obrade svi znakovi ulaznog niza. Jezgru leksičkog analizatora čini simulator konačnog automata koji je proširen mogućnošću prepoznavanja izraza. Polje *TablicaIzraza* za svako stanje bilježi indeks pripadne klase leksičkih jedinki, u skladu s gornjim razmatranjem. U uvjetu na kraju algoritma obrađuju se leksički neispravni dijelovi ulaznog niza, te se ulazni niz grupira u ispravno prepoznate leksičke jedinke.

3.2. Zahtjevi inkrementalnog postupka

Klasični leksički analizator nužno analizira cijelu ulaznu datoteku. Unutar jezično orjentirane razvojne okoline zahtjeva se osvježavanje stanja nakon svake promjene ulazne datoteke (uključujući i unošenje novog znaka). Da bi se postigla potrebna brzina osvježavanja, nužno je od linearnog algoritma napraviti inkrementalni. Inkrementalni algoritam na osnovu starog stanja i analize novonastalih promjena dovodi analizator u novo, osvježeno stanje.

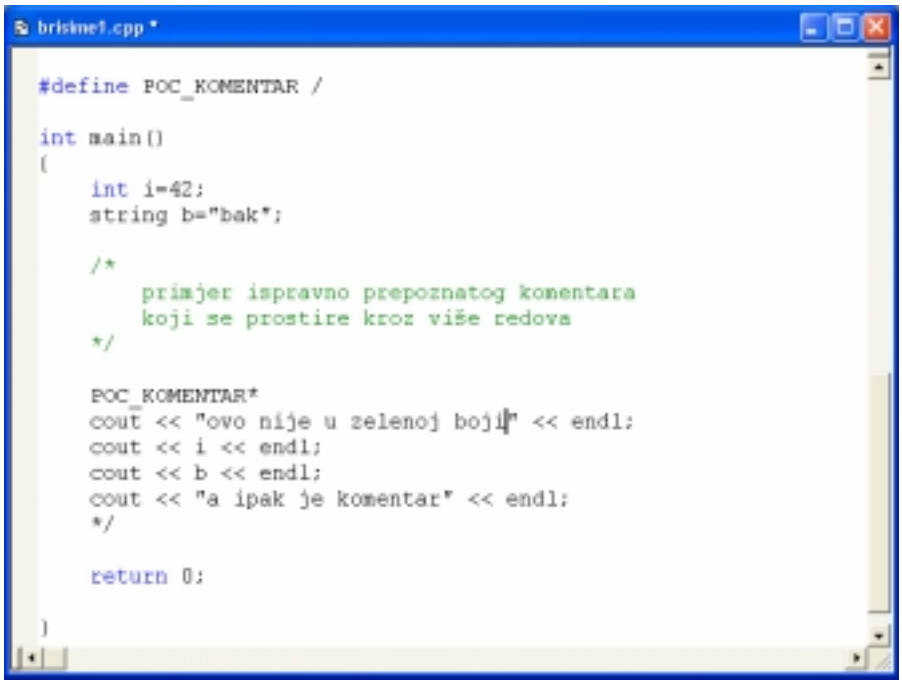
Inkrementalni leksički analizator još se više usložnjava zbog zahtjeva za vizualnim prikazom rezultata analize unutar razvojne okoline. Klasičan leksički analizator promatra ulaznu datoteku kao jednostavan niz znakova. Rezultat analize jednostavan je niz leksičkih jedinki. Ukoliko bi se koristila takva reprezentacija, vizualni prikaz ne bi bio učinkovit. Naime, unutar prozora uređivača teksta prikazuje se samo jedan djelić cjelokupne datoteke. Potrebno je prikazati samo dio datoteke u kvadratu opisanom koordinatama (prvi vidljivi red, prvi vidljivi stupac) – (zadnji vidljivi red, zadnji vidljivi stupac). Ukoliko ulazna datoteka nije podijeljena na redove, potrebno je prilikom svakog ekranskog prikaza napraviti analizu podjele redova od prvog znaka u datoteci do zadnjeg znaka vidljivog u uređivaču teksta. Stoga inkrementalni leksički analizator promatra ulaznu datoteku kao niz redova, a rezultat analize je niz leksičkih jedinki također podijeljen u redove. Takav prikaz stvara nove probleme. Neke leksičke jedinke (npr. komentari) se protežu kroz više redova. Uspješna obrada takvih slučajeva zahtjeva posebne strukturu podataka, što će biti opisano u slijedećem odjeljku.

Inkrementalna leksička analiza nekih stvarnih programskih jezika je izuzetno složena [10]. Primjer "teškog jezika" je C (odnosno C++). Glavni problem kod leksičke analize C programa predstavlja prisutnost predprocesorskih naredbi. Zbog njih je leksička analiza C programa zahtjeva dva prolaza. U prvom prolazu kroz ulaznu datoteku obrađuju se predprocesorske naredbe. Tek se u drugom prolazu ulazna datoteka u potpunosti analizira. Budući da se u razvojnoj okolini ulazna datoteka ne može nakon svake promijene analizirati u cijelosti, učinkovita inkrementalna leksička analiza postaje složeni problem.

Na slici 3.1 prikazan je primjer komercijalne razvojne okoline za C++ programski jezik [12], koja u ovom posebnom slučaju pokazuje da ugrađeni inkrementalni leksički analizator ne radi potpunu analizu.

U navedenom primjeru, razvojnu okolinu zbunjuje upravo predprocesorska naredba koja započinje komentar koji se rasprostire u više redova. Razvojna okolina nije prepoznala komentar, te je taj dio datoteke označen kao izvodljiv dio programskog koda. Ipak, navedeni primjer ispravno se prevodi u izvodljivu datoteku jer se tada za analizu koristi dvoprolazni leksički analizator.

Potpuno ispravna inkrementalna leksička analiza nije potrebna ukoliko se koristi samo za označavanje različitih leksičkih jedinki unutar uređivača teksta. Međutim, u potpuno jezično orijentiranoj razvojnoj okolini, inkrementalni leksički analizator služi kao početni korak inkrementalne sintaksne analize. Nužna je potpuno ispravna leksička analiza jer se inače ulazna datoteka ne može sintaksno analizirati.



Slika 3.1: Primjer neispravne inkrementalne leksičke analize

Inkrementalni jezični analizator koji je opisan u slijedećem dijelu ovog poglavlja radi za proizvoljan jezik koji ne zahtijeva dodatan prolaz jezičnog predpocesora. On se zbog toga ne može upotrijebiti za leksičku analizu programskog jezika C (odnosno C++).

3.3. Strukture podataka

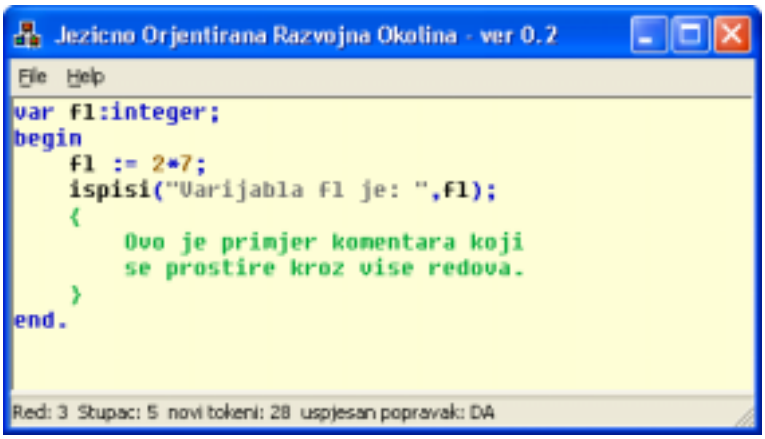
U prethodnom dijelu obrazložena je nužnost podjele ulazne datoteke na redove. Ulaz leksičkog analizatora jest niz redova, gdje je svaki red niz znakova. Izlaz leksičkog analizatora jest niz redova, gdje je svaki red niz leksičkih jedinki. Unutar svake leksičke jedinke bilježi se njena klasa i pripadni niz znakova koji čini tu leksičku jedinku. Osim klasa leksičkih jedinki koje pripadaju određenom programskom jeziku, nužne su i dodatne klase koje inkrementalni leksički analizator koristi za svoj rad. Sve klase leksičkih jedinki prikazane su u tablici 3.1.

KLASA LEKSIČKE JEDINKE	ZNAČENJE
KROS, IDN, DEK, REALNI …	Proizvoljan broj klasa leksičkih jedinki koje pripadaju određenom programskom jeziku
ODBACI	Predstavlja klasu leksičkih jedinki koja se ne prosljeđuje sintaksnom analizatoru, jer nema jezičnu važnost
GRESKA	Leksički neispravan dio ulazne datoteke
NASTAVAK	Nastavak prethodne leksičke jedinke u novom redu
NEPOZNATO	Označava dio ulazne datoteke koji još nije analiziran
ZNKRNIZA	Jedinka koja označava kraj ulaznog niza

Tablica 3.1: Klase leksičkih jedinki

ODBACI predstavlja klasu leksičkih jedinki koje nisu jezično važne: razmaci, tabulatori, komentari itd. Ta klasa se ignorira u daljnjem postupku sintaksne analize. Leksički neispravan dio ulazne datoteke se označava klasom GRESKA. Klasa NEPOZNATO označava upravo modificirani dio ulazne datoteke koji čeka da bude obrađen u procesu inkrementalne leksičke analize. Nakon analize nepoznato područje postaje niz leksičkih jedinki ili GRESKA (ukoliko se radi o leksički neispravnom dijelu ulazne datoteke). Poseban problem su leksičke jedinke koje se protežu kroz više redova (npr. komentari). Zbog toga postoji posebna leksička jedinka NASTAVAK koja označava da se radi o nastavku prethodne leksičke jedinke u novom redu.

Na slici 3.2 prikazan je primjer jedne jednostavne ulazne datoteke za programski jezik sličan PASCAL-u. Rezultat inkrementalne leksičke analize prikazan je u tablici 3.2.



Slika 3.2: Primjer jedne jednostavne ulazne datoteke

RED	KLASE LEKSIČKIH JEDINKI
0	KROS ODBACI IDN KROS KROS KROS ODBACI
1	KROS ODBACI
2	NASTAVAK IDN ODBACI KROS ODBACI DEK KROS DEK KROS ODBACI
3	NASTAVAK IDN KROS STR KROS IDN KROS KROS ODBACI
4	NASTAVAK
5	NASTAVAK
6	NASTAVAK
7	NASTAVAK
8	KROS KROS ZNKRNIZA

Tablica 3.2: Rezultat inkrementalne leksičke analize za datoteku sa slike 3.2

Pogledajmo kao primjer rezultat analize prvog reda, označenog indeksom 0. Klase KROS i IDN su klase koje pripadaju korištenom programskom jeziku. KROS predstavlja ključne riječi, operatore i specijalne znakove. IDN predstavlja identifikatore u tom programskom jeziku. Detaljnije objašnjenje može se naći u literaturi [1]. Vidimo da su leksičke jedinke "`var`", "`:`", "`integer`" i "`;`" ispravno svrstane KROS klasu leksičkih jedinki. Leksička jedinka "`fl`" je prepoznata kao IDN klasa, tj. identifikator. Razmak i znak za novi red svrstani su u klasu ODBACI, te nisu bitni za sintaksnu analizu. Poseban slučaj dešava se u redovima s indeksom 3,4,5,6 i 7. Kroz sve navedene redove proteže se jedna leksička jedinka klase ODBACI, koja se sastoji od znaka za novi red i komentara. Tu se koristi posebna klasa NASTAVAK, koja leksičku jedinku ODBACI iz trećeg reda nastavlja u slijedeća četiri reda. Stoga niz ODBACI-NASTAVAK-NASTAVAK-NASTAVAK-NASTAVAK predstavlja samo jednu leksičku jedinku.

Ovime su opisane bitne osobine struktura podataka koje su nužne za razumijevanje rada inkrementalnog postupka leksičke analize.

3.4. Inkrementalni postupak leksičke analize

Inkrementalni leksički analizator pokušava unutrašnju predodžbu ulazne datoteke kao niza leksičkih jedinki dovesti u osvježeno stanje, analizom samo promijenjenih dijelova te datoteke, te korištenjem rezultata prethodne analize. Ukoliko to nije u stanju, inkrementalni leksički analizator će odgovarajući dio ulazne datoteke označiti kao leksički neispravan dio.

Radi što veće brzine osvježavanja, nužno je analizirati što manji dio promijenjene datoteke. Postavlja se pitanje koliki dio je potrebno analizirati. Retkovna analiza, koja se spominje u uvodnom

poglavlju (koristi je Visual Basic 6.0), ponovo analizira cijeli red u kojem je došlo do promijene. To je u najvećem broju slučajeva previše, a kod drugih jezika u nekim slučajevima i premalo. Primjerice, u programskim jezicima PASCAL i C naredbe se mogu prostirati kroz mnoge redove ulazne datoteke. Minimalno područje koje je potrebno analizirati ovisi o osnovnom leksičkom analizatoru koji se proširuje inkrementalnim svojstvima [10]. Ukoliko je osnovni leksički analizator složen, te koristi stanja i razrješavanje nejednoznačnosti pretraživanjem lijevog ili desnog konteksta [1], broj leksičkih jedinki koje je potrebno analizirati je varijabilan. Stoga pojedini inkrementalni leksički analizatori za svaku leksičku jedinku bilježe broj okolnih jedinki koje je potrebno analizirati u slučaju modifikacije upravo te jedinke [10]. Takav pristup je složen, a nije nužno potreban jer se većina programskih jezika može dovoljno dobro analizirati jednostavnim leksičkim analizatorom. Stoga je u ovome radu korištena jednostavna leksička analiza, bez pamćenja stanja i razrješavanja nejednoznačnosti pretraživanjem lijevog ili desnog konteksta.

Uz takve pretpostavke, dovoljno je kod svake promijene ponovo analizirati slijedeće dijelove ulazne datoteke:

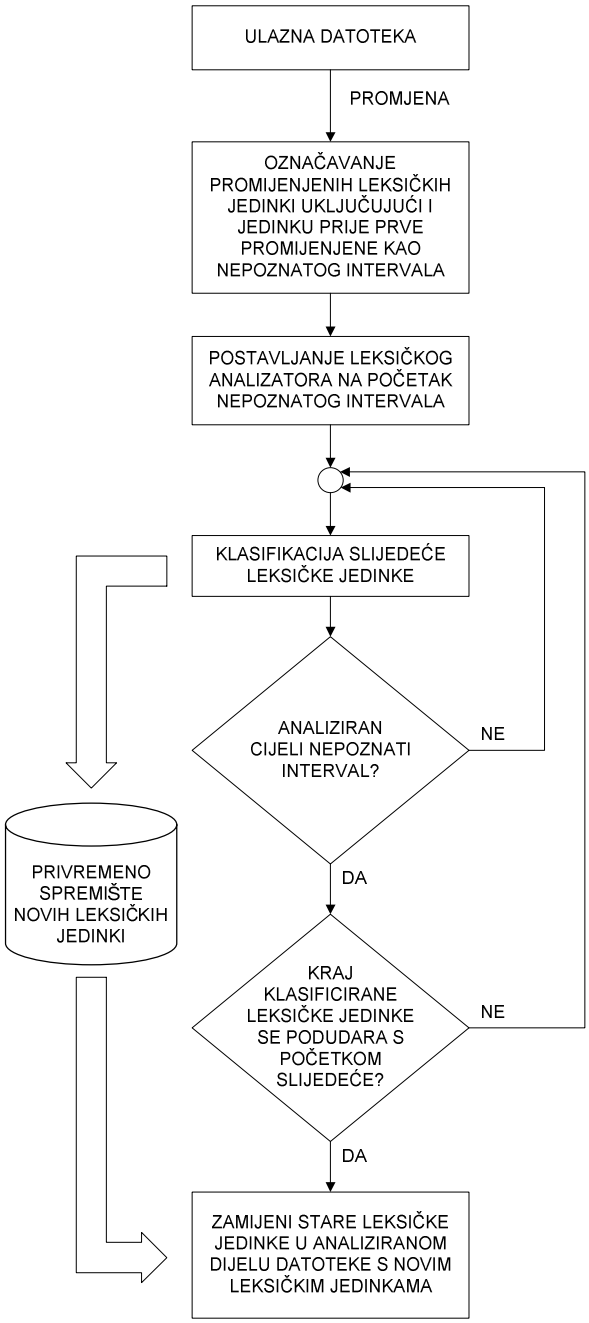
(i) Leksičke jedinke koje su promijenjene u ulaznoj datoteci;

(ii) Leksička jedinka neposredno prije prve promijenjene u ulaznoj datoteci.

Stavka (i) jasna je sama po sebi; sve promijenjene leksičke jedinke mogu zbog toga promijeniti i svoje leksičko značenje. Stavka (ii) zahtijeva dodatno objašnjenje. Prilikom procesa leksičke analize, klasa neke leksičke jedinke može biti određena na osnovu varijabilnog broja unaprijedih znakova, tj. znakova koji se nalaze iza leksičke jedinke. Zbog korištenja jednostavne leksičke analize, vrijedi da su svi unaprijedni znakovi (engl. lookahead characters) dio slijedeće leksičke jedinke. Stoga promjena unutar pojedine leksičke jedinke može utjecati na drugačiju klasifikaciju prethodne leksičke jedinke.

Posebno razmatranje zahtjeva uvjet za zaustavljanje procesa inkrementalne leksičke analize. Nakon leksičke analize potrebnog dijela ulazne datoteke, kraj zadnje analizirane leksičke jedinke se ne mora podudarati s početkom leksičke jedinke iz prethodne leksičke reprezentacije datoteke. Nepodudarnost znači da je dio znakova ulazne datoteke ostao neanaliziran. Stoga proces analize treba nastaviti sve dok se kraj upravo klasificirane leksičke jedinke ne poklopi s početkom postojeće leksičke jedinke iz prethodne leksičke reprezentacije datoteke. Tek tada je moguće obrisati dio starih leksičkih jedinki, te na njihovo mjesto ubaciti nove, koje su nastale u inkrementalnom procesu analize.

Sažeto je postupak inkrementalne leksičke analize prikazan na slici 3.3.



Slika 3.3: Postupak inkrementalne leksičke analize

4.

INKREMENTALNA SINTAKSNA ANALIZA

U ovom poglavlju izlaže se novi postupak inkrementalne sintaksne analize. Prvo je izložen potreban dio teorije formalnih jezika i analize formalnih jezika. Zatim su opisana proširenja postojeće teorije i postupci koji daju novu kakvoću procesu sintaksne analize.

4.1. Postupci analize formalnih jezika

Formalni jezici i učinkoviti postupci njihove analize izuzetno su široko područje koje zbog svoje opsežnosti nije moguće ovdje izložiti u potpunosti. Izložen je dio nužan za kasnije razumijevanje rada inkrementalnog sintaksnog analizatora. Više o navedenom području moguće je naći u literaturi [1,6].

4.1.1. Formalni jezici

Jezik jest skup nizova znakova nad abecedom. *Formalan jezik* jest jezik kod kojega je pripadnost skupu točno definirana formalnim postupkom. Postoji više formalnih postupaka za definiranje formalnih jezika. Ako je formalni jezik definiran skupom koji je konačan, onda je moguće navesti sve elemente toga skupa. Formalni jezici koji odgovaraju višim programskim jezicima nisu konačni (skupovi nizova znakova koji ih definiraju su beskonačni). Formalni postupak koji se koristi u ovome radu je definiranje jezika *konteksno-neovisnom gramatikom* [6].

Def. 6. *Konteksno-neovisna gramatika* je uređena četvorka

$$G = (V, T, P, S)$$

gdje je

- V – konačan skup nezavršnih znakova;
- T – konačan skup završnih znakova;
- P – konačan skup konteksno-neovisnih produkcija;
- S – početni nezavršni znak.

Završni znak jest jedinka niza koji pripada formalnom jeziku. Završni znakovi mogu biti elementi abecede jezika ili mogu predstavljati leksičke jedinke. Završni znakovi označavaju se malim slovima. Uvodi se poseban završni znak ε ('epsilon'). *Nezavršni znak* jest znak koji predstavlja niz završnih znakova koji su dio niza koji pripada formalnom jeziku. Nezavršni znakovi označavaju se velikim slovima abecede. Radi izbjegavanja dvoznačnosti nezavršni znakovi se često pišu unutar znakova `'<'` i `'>'` (npr. `<PRIMJER>`).

Def. 7. *Konteksno-neovisna produkcija* neke gramatike G je uređena dvojka

$$KNP = (L, D)$$

gdje je

L – nezavršni znak;

D – niz završnih i nezavršnih znakova koji su elementi skupova V i T gramatike G.

Konteksno-neovisna produkcija obično se piše u obliku

$$L \rightarrow D$$

gdje se L naziva lijeva strana produkcije, a D desna strana produkcije. Produkcija $L \rightarrow D$ je *epsilon produkcija* ako je $|D|=0$. Epsilon produkcija zapisuje se kao $L \rightarrow \varepsilon$.

Ako je $A \rightarrow \beta$ produkcija iz skupa P gramatike $G=(V,T,P,S)$ i ako su α i γ nizovi elemenata iz skupa $V \cup T$, tada vrijedi relacija:

$$\alpha A \gamma \underset{G}{\Rightarrow} \alpha\beta\gamma$$

Kaže se da se niz $\alpha\beta\gamma$ može dobiti primjenom produkcije $A \rightarrow \beta$ gramatike G na niz $\alpha A \gamma$, odnosno, da se iz niza $\alpha A \gamma$ može neposredno generirati niz $\alpha\beta\gamma$ primjenom samo jednog pravila produkcije gramatike G.

Relacija $\overset{*}{\underset{G}{\Rightarrow}}$ jest refleksivno i tranzitivno okruženje relacije $\underset{G}{\Rightarrow}$.

Svojstvo praznoće proširuje se za nizove znakova gramatike:

Def. 8. Niz znakova x gramatike G jest *prazan* ako i samo ako (akko) je ispunjen barem jedan od uvjeta:

(i) $|x| = 0$;

(ii) $x \overset{*}{\underset{G}{\Rightarrow}} w$, $|w| = 0$.

Znak gramatike jest prazan akko postoji prazan niz čiji je on element.

Definiranjem konteksno-neovisne gramatike G definira se formalan jezik L(G):

Def. 9. Niz završnih znakova w pripada formalnom jeziku L(G) kojega generira gramatika G=(V,T,P,S) akko su ispunjeni uvjeti:

(i) niz w sastoji se samo od završnih znakova koji su elementi skupa T;

(ii) niz w moguće je generirati iz početnog nezavršnog znaka S, tj. vrijedi $S \overset{*}{\underset{G}{\Rightarrow}} w$.

4.1.2. Sintaksno stablo

Stablo je složena struktura podataka [8]. Pretpostavlja se da je stablo uređeno (engl. ordered) tj. definirane su relacije lijevo i desno. Sintaksno ili generativno stablo za gramatiku G je [6]:

Def. 10. Stablo je *sintaksno stablo* ili *generativno stablo* za konteksno-neovisnu gramatiku G=(V,T,P,S) akko su ispunjeni uvjeti:

(i) čvorovi stabla su označeni znakovima iz skupa $V \cup T \cup \{\varepsilon\}$;

(ii) korijen stabla označen je početnim nezavršnim znakom S;

(iii) ako je unutrašnji čvor označen s A, onda je A nezavršni znak $A \in V$;

(iv) Neka su čvorovi n_1, n_2, ..., n_k, čvorovi sinovi od čvora n (pobrojani s lijeva na desno u sintaksnom stablu). Neka je čvor n označen s A, a čvorovi n_1, n_2, ..., n_k, s X_1, X_2, ..., X_k. Tada je

$$A \rightarrow X_1 X_2 \ldots X_k$$

konteksno-neovisna produkcija iz skupa P;

(v) Ako je čvor n označen s ε, onda je čvor n list i on je jedini neposredni sljedbenik jednog od unutarnjih čvorova.

Niz znakova koji se dobije obilaskom listova sintaksnog stabla s lijeva na desno pripada jeziku L(G). Vrijedi i obratna tvrdnja. Niz α pripada jeziku L(G) ako, i samo ako, postoji generativno stablo za gramatiku G čiji su listovi označeni s α.

Ispravnost ulaznog niza se u jezičnom procesoru provjerava izgradnjom sintaksnog stabla. Sintaksno stablo se dalje koristi u koraku semantičke analize. Stoga je za izradu učinkovitih jezičnih procesora potreban učinkovit postupak izgradnje sintaksnog stabla.

4.1.3. Postupci parsiranja

Proces izgradnje sintaksnog (generativnog) stabla za zadani niz *w* naziva se *parsiranje niza*. Program ili dio programa koji gradi sintaksno stablo naziva se *parser*.

Postupci parsiranja dijele se prema načinu izgradnje sintaksnog stabla. *Parsiranje od vrha prema dnu* jest postupak parsiranja kod kojega se sintaksno stablo gradi od korijena (početni nezavršni znak) prema listovima stabla (završni znakovi gramatike). Obratna metoda parsiranja, koja gradi stablo od listova stabla prema korijenu stabla, naziva se *parsiranje od dna prema vrhu*. Postoji mnogo postupaka parsiranja od vrha prema dnu, kao i mnogo postupaka parsiranja od dna prema vrhu.

Parseri koji koriste proces parsiranja od vrha prema dnu najčešće spadaju u klasu LL(*k*), gdje je *k* broj unaprijednih znakova koji se koriste za donošenje odluke u procesu rada parsera. Prvi 'L' označava da se ulazni niz ispituje s lijeva na desno (engl. left-to-right scanning), a drugi 'L' označava da se stablo izgrađuje od krajnje lijevog nezavršnog znaka u izgrađenom međunizu (engl. leftmost derivation). Najčešće korišten LL(*k*) parser je LL(1) parser, koji se učinkovito izgrađuje korištenjem postupka rekurzivnog spusta. LL(1) parseri zahtijevaju korištenje LL(1) gramatika izvornog jezika. To često predstavlja problem, jer se pojedine sintaksne strukture složenih programskih jezika teško mogu izraziti LL(1) gramatikom.

Parseri koji koriste proces parsiranja od dna prema vrhu najčešće spadaju u klasu LR(*k*). Razlika u odnosu na LL(*k*) parsere je u znaku 'R' koji označava da se stablo gradi obratnim postupkom zamjene krajnje desnog nezavršnog znaka (engl. rightmost derivation). Najčešće korišteni LR(*k*) parseri su LR(0) parser (naziva se i SLR parser) i LR(1) parser (naziva se i kanonski LR parser). Oni zahtijevaju pripadne LR(0) i LR(1) gramatike.

Svi LR parseri koriste isti postupak parsiranja. Različiti LR parseri razlikuju se samo u tablici parsiranja. Tablica parsiranja je dio LR parsera u kojem su, za svako stanje LR parsera, opisani slijedeći koraci rada LR parsera. Složenije gramatike zahtijevaju složenije tablice parsiranja koje je teže izgraditi.

Među gramatikama vrijedi međuodnos LL(1) $\subset$ LR(0) $\subset$ LR(1) tj. LR(0) gramatika obuhvaća LL(1) gramatiku, a LR(1) gramatika obuhvaća LR(0) i LL(1) gramatike. Postoji i LALR parser (engl. look-ahead LR) koji je po svojstvima i složenosti izgradnje između LR(0) i LR(1) parsera. Zbog veće obuhvatnosti pripadnih gramatika, LR parseri uobičajen su izbor kod izgradnje jezičnih procesora.

4.1.3.1. LR parser

LR parser ispituje ulazni niz znak po znak, s lijeva na desno. Sintaksno stablo gradi se postupkom zamjene krajnje desnog nezavršnog znaka (engl. rightmost derivation).

LR parser zahtijeva četiri strukture za svoj rad:

(i) ulazni niz *w*;

(ii) stog LR parsera;

(iii) tablica parsiranja;

(iv) konteksno-neovisna gramatika izvornog jezika.

Pretpostavlja se da je ulazni niz znakova zaključen posebnim znakom '⊥', koji se naziva *oznaka kraja niza*.

Stog je složena struktura podataka [8]. Na stog LR parsera stavljaju se stanja parsera i znakovi gramatike.

O tablici parsera ovisi klasa jezika koju prihvaća LR parser. Jezici opisani LR(1) gramatikom zahtijevaju složeniju tablicu parsiranja nego jezici opisani LR(0) gramatikom. Tablica parsiranja sastoji se od dva dijela. U prvom dijelu svakom stanju LR parsera se za svaki ulazni znak pridružuje slijedeća operacija parsera. Postoje četiri moguće operacije: *pomak*, *redukcija*, *prihvat* i *odbacivanje*. Pojedinosti izvedbe izložene su u opisu programa LR parsera. U drugom dijelu tablice parsiranja se svakom stanju LR parsera, za potrebne nezavršne znakove, pridružuje slijedeće stanje parsera nakon operacije redukcije.

Konteksno-neovisna gramatika $G=(V,T,P,S)$ izvornog jezika je potrebna radi skupa produkcija P, koji se koristi u operacijama redukcije LR parsera.

LR parser izvodi slijedeći program:

```
početak
   postavi KAZALJKU da pokazuje na prvi znak u nizu w⊥;
   postavi stog da sadrži samo početno stanje s0;
   ponavljaj zauvijek
      neka je s stanje na vrhu stoga i neka KAZALJKA pokazuje
      na znak a u nizu w⊥;
      slučaj PrviDioTabliceParsiranja[s,a]
         pomak s':
            stavi a na stog a zatim stavi s' na stog;
            pomakni KAZALJKU na slijedeći znak u nizu w⊥;
         redukcija A→β:
            skini sa stoga 2*|β| znakova;
            neka je s' stanje na stogu nakon što se skine 2*|β| znakova;
            stavi A na stog a zatim stavi stanje
            DrugiDioTabliceParsiranja[s',A] na vrh stoga;
            izgradi čvorove sintaksnog stabla koji odgovaraju
            produkciji A→β;
```

```
            prihvat:
               niz je iz jezika L(G), završi s parsiranjem;
            odbacivanje:
               obriši sve izgrađene čvorove sintaksnog stabla;
               niz nije je iz jezika L(G), završi s parsiranjem;
         kraj
      kraj
```

Program LR parsera za niz iz jezika L(*G*) gradi sintaksno stablo koje ga opisuje. Ako su povezani koraci sintaksne i semantičke analize, onda se program preuređuje tako da umjesto izgradnje sintaksnog stabla obavlja dio semantičke analize.

Zbog svoje složenosti, neće biti izložen postupak izgradnje tablice parsiranja za LR(0) i LR(1) gramatike. Pojedinosti o postupku izgradnje tablice parsiranja moguće je naći u literaturi [1].

4.2. Zahtjevi inkrementalnog postupka

Zamisao inkrementalnog sintaksnog analizatora jest da tijekom uređivanja u razvojnoj okolini sintaksno stablo cijelo vrijeme postoji u radnoj memoriji, te da se održava u ispravnom stanju sukladno sa promjenama ulazne datoteke. Pri svakoj promjeni ulazne datoteke, stablo na trenutak postane neispravno, jer ne odražava pravo stanje ulazne datoteke. Neispravan je samo dio stabla tj. jedno njegovo podstablo. Poziva se inkrementalni parser koji parsira samo dio ulazne datoteke i zamjenjuje neispravno podstablo sa ispravnim podstablom. Time sintaksno stablo ponovo postaje ispravno.

Osnovni model LR parsera nije primjenjiv za inkrementalnu sintaksnu analizu. LR parser uvijek parsira cijeli ulazni niz i izgrađuje cijelo sintaksno stablo. Proces parsiranja ne može krenuti od nekog dijela ulaznog niza, jer nije definirano stanje stoga u tom trenutku. LR parser ne može stati prije nego što dođe do oznake kraja datoteke.

Zahtjevi koji se postavljaju na inkrementalni sintaksni analizator su:

(i) potrebno je za svaku jedinku ulazne datoteke znati poziciju popravka u sintaksnom stablu;

(ii) proces parsiranja započinje i završava sa proizvoljnom jedinkom ulazne datoteke;

(iii) proces parsiranja staje kad je moguća zamjena neispravnog podstabla sa ispravnim podstablom;

(iv) nakon zamjene neispravnog podstabla sa ispravnim podstablom, sintaksno stablo treba biti ispravno, a za svaku jedinku ulazne datoteke je i dalje potrebno znati poziciju popravka u sintaksnom stablu;

(v) inkrementalni parser nastavlja s radom i nakon pojave jednog ili više neispravnih dijelova ulazne datoteke.

Postupak izgradnje inkrementalnog sintaksnog analizatora razložen je u više koraka. Prvo su iscrpno objašnjene potrebne strukture podataka. Zatim su izloženi postupci obnove stoga iz sintaksnog stabla. Nakon toga je objašnjen način obrade sintaksno neispravnih područja ulazne datoteke. Slijedi rasprava o nužnim uvjetima zaustavljanja procesa parsiranja. U zadnjem dijelu obavlja se sinteza i opisuje se rad inkrementalnog sintaksnog analizatora.

4.3. Strukture podataka

Prethodno postavljeni zahtjevi (4.2) uvjetuju posebnu strukturu ulazne datoteke. Jedinkama ulazne datoteke pridodan je pokazivač na čvor sintaksnog stabla. Pokazivač svake jedinke postavi se tako da pokazuje na onaj čvor sintaksnog stabla koji izravno postaje neispravan mijenjanjem te jedinke (tj. pokazuje upravo na dio stabla koji treba popraviti). Time se omogućava trenutno nalaženje neispravnog dijela stabla nakon svake promjene ulazne datoteke.

Zbog zamjena neispravnih dijelova stabla, jedinke ulazne datoteke često pokazuju na nepostojeće čvorove. Potrebno je brzo zamijeniti neispravne pokazivače na stare čvorove s ispravnim pokazivačima na nove čvorove. Listovi sintaksnog stabla, upravo radi brze zamjene neispravnih pokazivača, sadrže pokazivače na jedinke ulazne datoteke.

Pokazivačima u jedinkama ulazne datoteke i pokazivačima u listovima sintaksnog stabla ostvaruje se dvostruka povezanost navedenih struktura. Na slici 4.1 su, jako pojednostavljeno, prikazani sintaksno stablo i ulazna datoteka. Ulazna datoteka prikazana je kao niz jedinki ulazne datoteke. Svaki čvor sintaksnog stabla odgovara jednom završnom ili nezavršnom znaku.

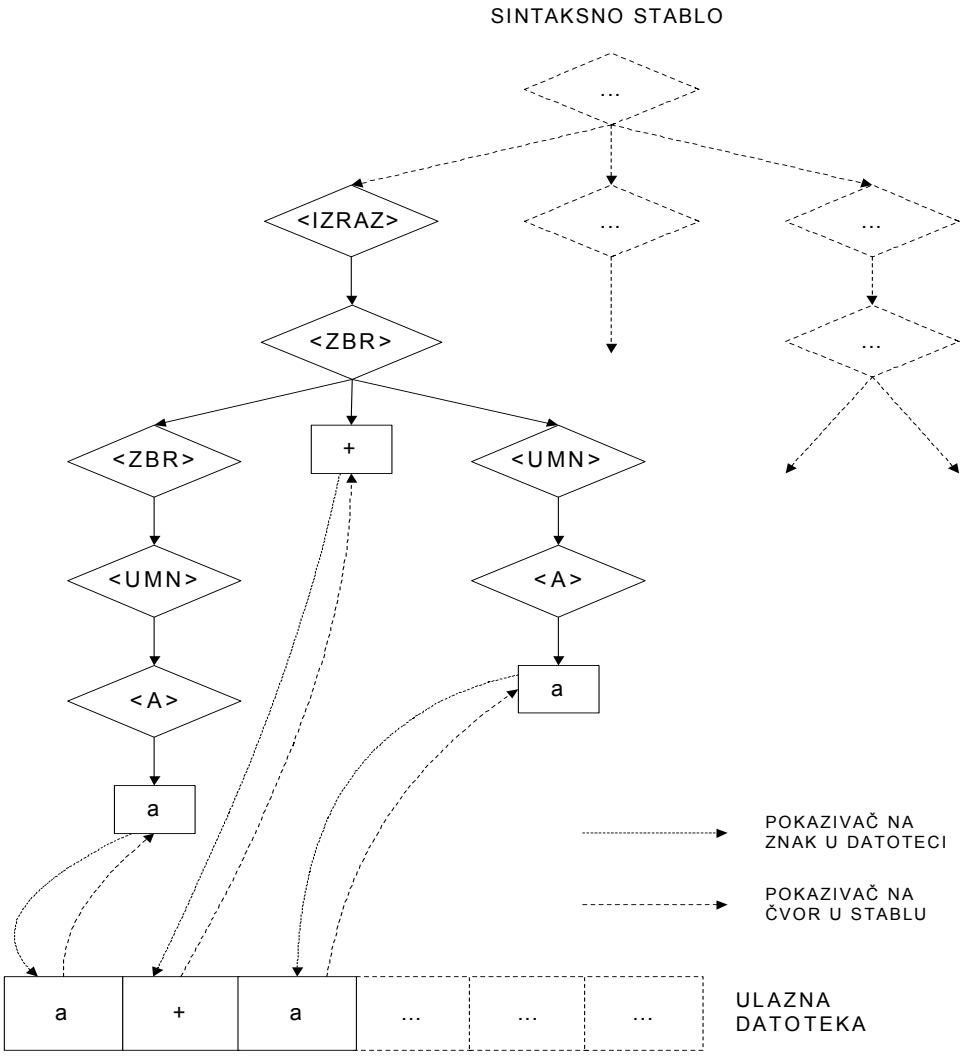
Uvode se slijedeće definicije:

Def. 11. Ulazna datoteka i sintaksno stablo su *podudarni* akko su ispunjeni slijedeći uvjeti:

(i) sintaksno stablo ispravno predstavlja cijelu ulaznu datoteku s obzirom na zadanu gramatiku *G*;

(ii) svaka jedinka ulazne datoteke pokazuje samo na *pripadni* neprazni završni znak u sintaksnom stablu;

(iii) svaki neprazni završni znak sintaksnog stabla pokazuje samo na *pripadnu* jedinku ulazne datoteke.



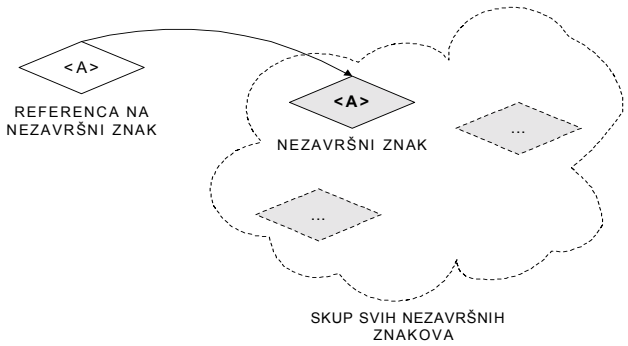
Slika 4.1: Veza između ulazne datoteke i sintaksnog stabla

Def. 12. Refleksivna relacija *pripadnosti* vrijedi između jedinke (na poziciji *X* ulazne datoteke) i nepraznog završnog znaka (na poziciji *Y* sintaksnog stabla) samo ako su im pozicije jednake ($X=Y$). Pozicija u ulaznoj datoteci određuje se kao udaljenost od početnog znaka datoteke. Prvi znak datoteke ima poziciju 0. Pozicija nepraznog završnog znaka u sintaksnom stablu određuje se kao broj nepraznih završnih znakova prije toga znaka. Prije se nalazi onaj znak koji je *lijevo* u stablu (stablo je uređeno). Krajnje lijevi neprazni završni znak u stablu ima poziciju 0.

Prazni završni znakovi (ε) ne pokazuju ni na jednu jedinku ulazne datoteke, te nijedna jedinka ne pokazuje na njih.

Sintaksno stablo prikazano na prethodnoj slici (Slika 4.1) sastoji se od početnog nezavršnog znaka kao vrha stabla, te čvorova koji su sastavljeni od nezavršnih i završnih znakova gramatike. To je pojednostavljena predodžba, jer se sami znakovi gramatike ne nalaze u stablu. Znakovi gramatike koriste se na više mjesta (u produkcijama gramatike, u sintaksnom stablu, itd.), pa ih stoga nije učinkovito svaki puta definirati (npr. ime znaka). Samo kod učitavanja gramatike izvornog jezika stvaraju se skupovi potrebnih završnih i nezavršnih znakova. Na drugim mjestima koriste se *reference na znakove gramatike*. Primjer reference na nezavršni znak prikazan je na slici 4.2.

Znak sadrži svoja statička svojstva, tj. ona koja se nikad ne mijenjaju (ime znaka, praznoća itd.). Referenca na znak gramatike sadrži pokazivač na pojedini znak, kao i dinamička svojstva znaka (npr. pokazivač na jedinku ulazne datoteke). Statička i dinamička svojstva znaka mijenjaju se ovisno o vrsti znaka (završni ili nezavršni).

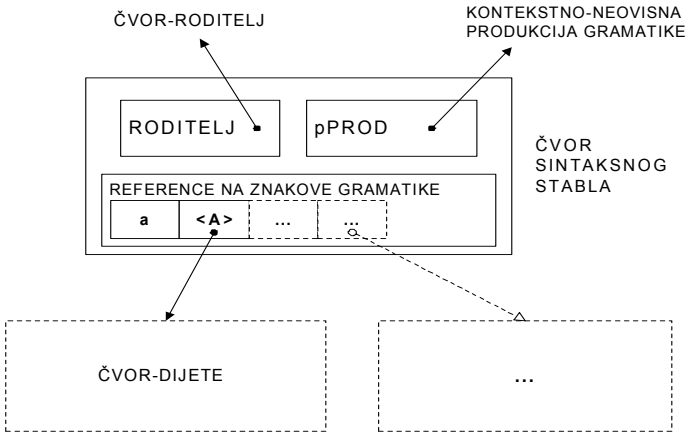


Slika 4.2: Primjer reference na znak

Obnavljanje stoga zahtjeva sintaksno stablo izgrađeno na način da je moguće kretanje od listova prema korijenu stabla. Zbog toga svaki čvor sintaksnog stabla ima pokazivač na čvor-roditelj (engl. parent node).

Svi čvorovi-djeca nekog čvora-roditelja su gramatički povezani. Čvorovi-djeca predstavljaju desnu stranu neke konteksno-neovisne produkcije, a njihov čvor-roditelj predstavlja lijevu stranu iste produkcije. Stoga je moguće uzeti *reference na produkcije* za čvorove sintaksnog stabla, umjesto referenci na znakove. To je i poželjno, radi smanjenja broja čvorova, smanjenja utroška memorije i jednostavnijeg pronalaženja pripadnih produkcija.

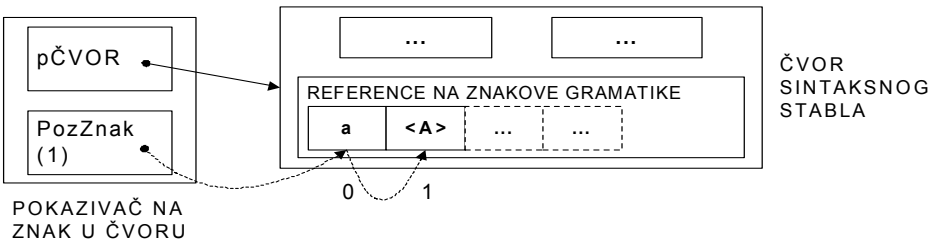
Uz navedene primjedbe, čvor sintaksnog stabla poprima slijedeću strukturu kao na slici 4.3.



Slika 4.3: Struktura čvora sintaksnog stabla

Čvor sadrži pokazivač na čvor-roditelj, pokazivač na pripadnu produkciju i niz referenci na znakove gramatike. Budući da svaka konteksno-neovisna produkcija ima jedan nezavršni znak na lijevoj strani, pokazivači na čvorove-djecu nalaze se u referencama na nezavršne znakove.

Zbog odabrane strukture čvora usložnjava se referenciranje na neki pojedini znak gramatike u sintaksnom stablu. Uvodi se struktura *pokazivača na znak u čvoru*, prikazana na slici 4.4.



Slika 4.4: Struktura pokazivača na znak u čvoru

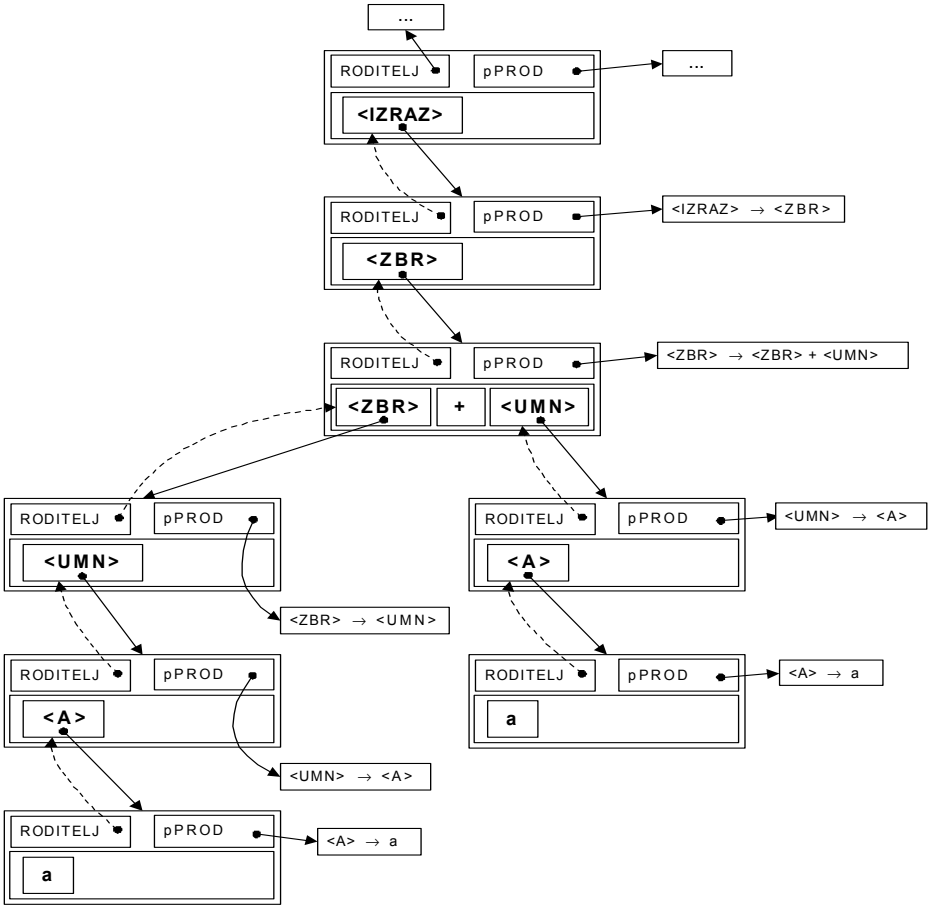
Pokazivač na znak u čvoru sastoji se od pokazivača na čvor (pČVOR) i pozicije znaka u čvoru (PozZnak), gdje krajnje lijevi znak ima poziciju 0.

Pokazivač na čvor-roditelj je upravo pokazivač na znak u čvoru. On pokazuje na referencu na nezavršni znak koji predstavlja lijevu stranu pripadne produkcije.

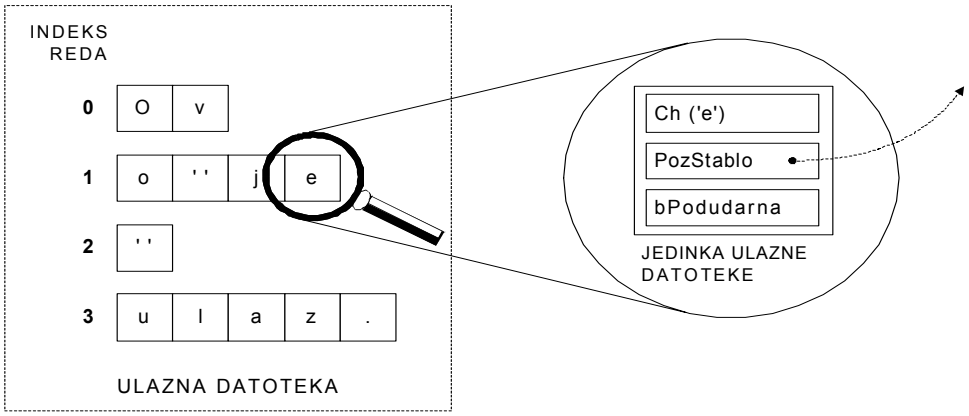
Ponekad će se reference na znakove gramatike, radi jednostavnosti, nazivati samo znakovima gramatike (na onim mjestima na kojima ne može doći do zabune).

Podgrana sintaksnog stabla, koja je pojednostavljeno prikazana na slici 4.1, prikazana je sa svim pojedinostima na slici 4.5. Sintaksno stablo će se često, radi jednostavnosti, prikazivati bez nepotrebnih detalja.

Usložnjava se i struktura ulazne datoteke. Jednostavan, linearan oblik nije podesan za uporabu jer ne dijeli ulaznu datoteku u redove (podjela u redove potrebna je za učinkoviti zaslonski prikaz). Ulazna datoteka uređena je kao *niz redova*. Svaki red je *niz jedinki*. *Jedinka* ulazne datoteke je struktura koja sadrži znak datoteke, referencu na *pripadni* (Def. 12) znak u stablu, te jednu logičku varijablu koja pamti *stanje podudarnosti* (nepodudarne jedinke uzrokuju nepodudarnost ulazne datoteke i sintaksnog stabla) za tu jedinku. Struktura ulazne datoteke prikazana je na slici 4.6. Ponekad će se ulazna datoteka, radi jednostavnosti, promatrati kao potpuno linearna struktura (kraj jednog reda nadovezuje se slijedeći red).



Slika 4.5: Pojedinosti izvedbe sintaksnog stabla



Slika 4.6: Struktura ulazne datoteke

Sve navedene strukture zadovoljavajuće opisuju stanje u nekom trenutku u kojem vrijedi podudarnost ulazne datoteke i sintaksnog stabla. U trenutku promijene ulazne datoteke prestaje vrijediti podudarnosti. Pokreće se postupak popravka, koji zahtjeva dodatne podatke.

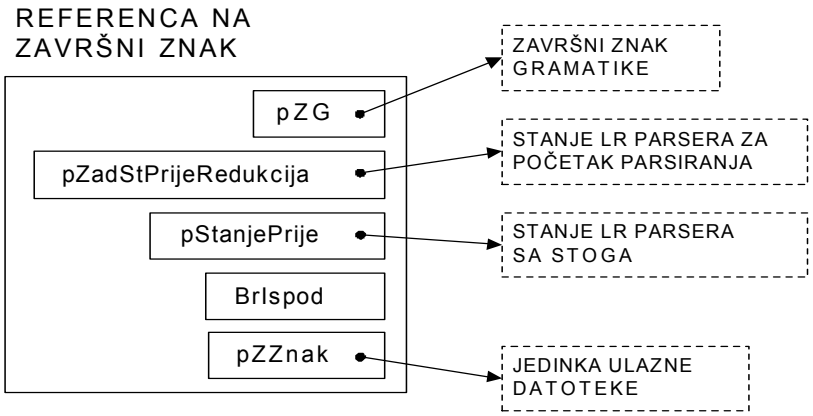
LR parser za svoj rad zahtjeva početni sadržaj stoga (4.1.3.1). Stog je moguće obnoviti iz sintaksnog stabla ako se uz svaku referencu na znak gramatike pamti i stanje LR parsera koje se nalazilo na stogu nakon premještanja reference na znak sa stoga u sintaksno stablo. Stanje LR parsera se pamti u elementu (pStanjePrije) strukture reference na znak.

U referencama na završne znakove pamti se i dodatno stanje (pZadStPrijeRedukcija) koje je potrebno za početnu pripremu inkrementalnog parsera.

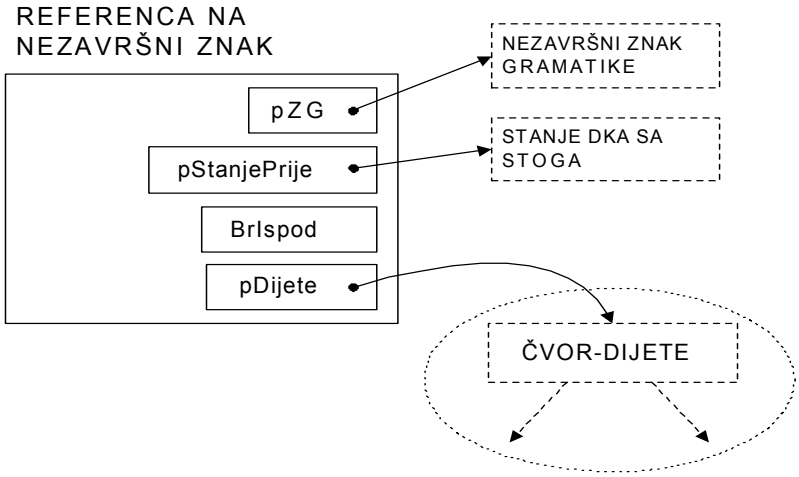
Svaka referenca na znak koja se nalazi u sintaksnom stablu predstavlja niz jedinki ulazne datoteke. To je očekivano svojstvo nezavršnih znakova, jer svaki nezavršni znak predstavlja niz završnih znakova, a time i niz jedinki ulazne datoteke. U trenutku promjene ulazne datoteke (npr. umetanje jedinke) događa se da i završni znak predstavlja više jedinki ulazne datoteke (po ponovnoj uspostavi podudarnosti završni znak ponovo predstavlja samo jednu jedinku). Broj jedinki koje predstavlja neki znak gramatike (tj. čvor sintaksnog stabla) pamti se u posebnom elementu strukture (BrIspod). Podatak o broju jedinki koje neki znak gramatike predstavlja potreban je zbog obrade neispravnih područja, te zbog zaustavljanja procesa popravka stabla.

Reference na znakove gramatike poprimaju oblik prikazan na slikama 4.7 i 4.8.

Zbog prirode procesa LR parsiranja potrebno je ulaznu datoteku zaključiti sa posebnim znakom '⊥', koji se naziva *oznaka kraja niza*. Oznaka kraja niza potrebna je za donošenje odluke o redukciji na kraju ulaznog niza, kao i za završetak parsiranja. Oznaka kraja niza dodaje se kao zadnja jedinka ulazne datoteke, a pokazuje na referencu na znak za kraj u *glupom čvoru* sintaksnog stabla.

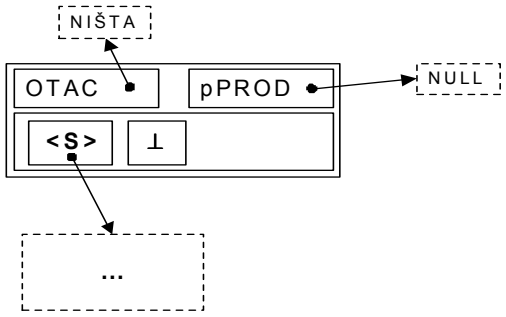


Slika 4.7: Struktura reference na završni znak



Slika 4.8: Struktura reference na nezavršni znak

Glupi čvor (engl. dummy node) jest poseban čvor koji je vrh sintaksnog stabla. Slika 4.9 prikazuje strukturu glupog čvora. Poseban je zbog više svojstava. Potrebno ga je stvoriti prije početka parsiranja ulazne datoteke. Nema čvor-roditelj (on je vrh sintaksnog stabla). Sadrži početni nezavršni znak (nije nastao nijednom produkcijom gramatike). Sadrži i referencu na oznaku kraja niza, koja dolazi iza početnog nezavršnog znaka. Takva izvedba je logički ispravna (iza ulaznog niza w nalazi se kraj datoteke $\perp$), mada nije uobičajena u sintaksnom stablu.



Slika 4.9: Početni čvor ili *glupi čvor* sintaksnog stabla

Glupi čvor izveden na ovaj način olakšava izvedbu inkrementalnog parsera. Postoji više razloga. Zadnja jedinka ulazne datoteke (koja sadrži oznaku kraja niza) pokazuje na referencu na oznaku kraja u glupom čvoru. Time su zadovoljene definicije pripadnosti (Def. 12) i podudarnosti (Def. 11) za sintaksno ispravnu ulaznu datoteku. Također, djelomično se pojednostavljuje postupak popravka sintaksnog stabla kod uređivanja kraja datoteke. U referencu na oznaku kraja niza u glupom čvoru sprema se i podatak o stanju LR parsera na kraju datoteke (element pZadStPrijeRedukcija), koji bi se u slučaju nepostojanja navedene reference morao spremiti na drugo mjesto.

4.4. Obnova stoga iz sintaksnog stabla

Da bi proces parsiranja mogao krenuti od proizvoljne jedinke ulazne datoteke, potrebno je za svaku jedinku poznavati stog LR parsera. Stog je neizravno zapisan u sintaksnom stablu, jer se u operaciji redukcije (u postupku LR parsiranja) znakovi sa vrha stoga stavljaju u sintaksno stablo. Definira se stableni stog:

Def. 13. *Stableni stog* jest prividni stog koji je definiran u sintaksnom stablu *S* za svaki element *V* tog sintaksnog stabla. Za element *V*, sadržaj stablenog stoga jednak je sadržaju običnog stoga LR parsera (tijekom izgradnje sintaksnog stabla *S*) u trenutku neposredno prije stavljanja elementa *V* na stog. *V* se naziva *vrh* stablenog stoga.

Za svaku jedinku ulazne datoteke poznata je pozicija popravka u sintaksnom stablu. Time je za svaku jedinku poznat i sadržaj stablenog stoga, te je moguć nastavak procesa parsiranja. U stableni stog se ne mogu stavljati novi elementi, jer bi onda bilo nužno mijenjati sintaksno stablo. Elementi koji se obrađuju u procesu parsiranja stavljaju se na privremeni stog:

Def. 14. *Privremeni stog* jest stog na koji se stavljaju elementi nastali u procesu LR parsiranja. Po iscrpljivanju privremenog stoga, elementi se uzimaju sa stablenog stoga. Vrijedi odnos:

STOG INKREMENTALNOG PARSERA = STABLENI STOG + PRIVREMENI STOG

Na stog se u običnoj izvedi LR parsera stavljaju dvije vrste elemenata: stanja LR parsera i reference na znakove gramatike. Zbog posebne izvedbe referenci na znakove gramatike, potrebno je na privremeni stog stavljati samo reference na znakove gramatike.

Stablenom i privremenom stogu se pristupa kroz posebne funkcije za rad sa stogom. Za izvedbu tih funkcija nužno je opisati učinkovit postupak dobivanja elemenata stablenog stoga. Potrebni su slijedeći teoremi.

Teorem 1. Za neki čvor-dijete vrijedi da se neposredno prije u stablenom stogu nalazi onaj čvor-dijete (ima isti čvor-roditelj) koji je susjedni i nalazi se neposredno lijevo, ako takav čvor postoji.

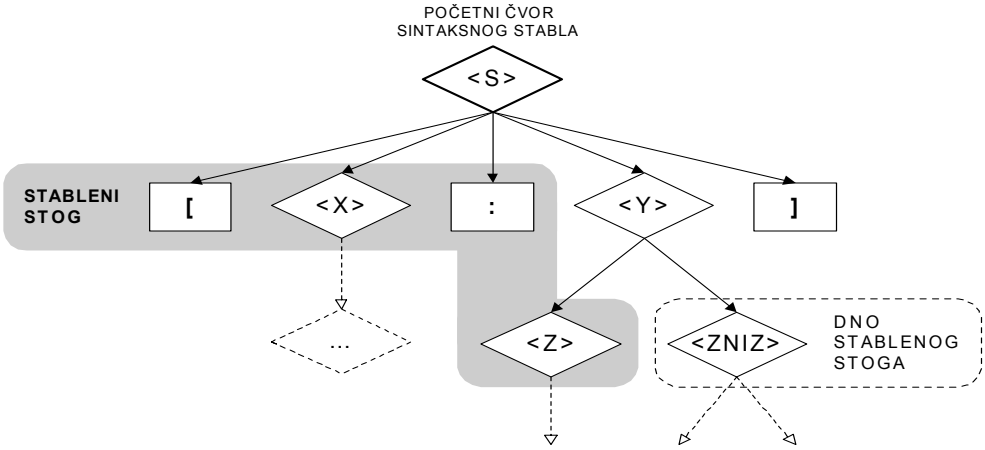
Dokaz teorema je slijedeći. Svi čvorovi-djeca nastaju kao rezultat primjene određene operacije redukcije. U postupku redukcije uzima se element po element s izvornog stoga i stavlja se u stablo u poretku s desna na lijevo. Susjedni lijevi čvor je neposredno prije na izvornom, a zbog toga i u stablenom stogu.

Teorem 2. Ako je čvor krajnji lijevi čvor-dijete, onda se neposredno prije u stablenom stogu nalazi onaj čvor koji je neposredno prije čvora-roditelja u stablenom stogu.

Dokaz teorema je slijedeći. U postupku redukcije se niz skinutih elemenata stoga $D_1, D_2, \ldots, D_n$, a koji čine desnu stranu produkcije P, zamjenjuje sa elementom L koji čini lijevu stranu produkcije P. Onaj element X koji je na stogu neposredno prije krajnje lijevog elementa D_1 desne strane produkcije je također i prije elementa L koji zamjenjuje niz $D_1, D_2, \ldots, D_n$.

Teorem 3. Vrh sintaksnog stabla nema neposrednog prethodnika u stablenom stogu.

Dokaz teorema je slijedeći. U trenutku prihvaćanja ulaznog niza se na stogu LR parsera nalazi samo element V koji predstavlja početni nezavršni znak. Element V predstavlja vrh sintaksnog stabla. Budući da je element V sam na stogu, on nema neposrednog prethodnika.



Slika 4.10: Primjer stablenog stoga

Na slici 4.10 prikazan je primjer stablenog stoga. Sivo su osjenčani znakovi koji pripadaju stablenom stogu, dok je znak koji predstavlja vrh stablenog stoga uokviren isprekidanom linijom. Vrh stablenog stoga ne pripada stogu, tj. vrh stoga pokazuje na prazno mjesto. Za prikazani vrh stoga (tj. čvor **<ZNIZ>**) sadržaj stablenog stoga je: **'[' <X> ':' <Z>** . Naredbom uzimanja elementa sa stoga dobiva se znak **<Z>**, zatim **':'** itd.

Prethodno izneseni teoremi omogućavaju učinkovito izdvajanje stablenog stoga za neki element sintaksnog stabla. Problem nastaje ako se koristi LR(1) parser za inkrementalnu izgradnju sintaksnog stabla. Tada početni vrh stoga za neku jedinku ulazne datoteke nije uvijek jednak pripadnom znaku sintaksnog stabla:

Teorem 4. Promjena jedinke ulazne datoteke uzrokuje da su u stablenom stogu pripadnog znaka sintaksnog stabla neispravni oni elementi koji su nastali primjenom neposredno prethodećeg niza redukcija.

Dokaz teorema je slijedeći. LR(1) parser odlučuje o slijedećoj operaciji na temelju jednog unaprijednog znaka ulazne datoteke. Ako je slijedeća operacija redukcija, onda se pokazivač na znak u ulaznoj datoteci neće promijeniti. Kod promjene znaka ulazne datoteke postaju nevažeće sve redukcije koje su koristile promijenjeni znak za odluku o redukciji, a to je upravo prethodeći niz redukcija.

Zbog prethodnog teorema se u funkcijama za rad sa stogom posebno obrađuje prvi pomak vrha stoga. Ako je slijedeći znak stoga vrh nekog nepraznog podstabla, onda je potrebno pronaći krajnje desni neprazni završni znak i postaviti ga za novi vrh stoga. Upravo zbog prethodnog teorema postoji poseban element strukture reference na završni znak u kojem je zabilježeno zadnje stanje prije primjene neposredno prethodećeg niza redukcija.

Osnovna procedura za rad sa stablenim stogom je ona koja pomiče vrh stablenog stoga na prethodni element:

```
procedura PomakniVrhNaPrethodni()
  ponavljaj zauvijek
    ako Vrh pokazuje na krajnje lijevo dijete tada
      pomakni Vrh na roditelja;
    inače
      pomakni Vrh na susjedno lijevo dijete;
      vrati se iz procedure;
    kraj
  kraj
kraj procedure
```

U slučaju da se prethodna procedura pozove za prazan stableni stog (npr. vrh stoga postavljen je na krajnje lijevi nezavršni znak u stablu), dolazi do pojave greške.

Prvi pomak vrha stablenog stoga potrebno je posebno obraditi (Teorem 4):

```
procedura SrediPrviPomakVrha()
  ponavljaj zauvijek
    ako Vrh ne pokazuje na referencu na nezavršni znak tada
      vrati se iz procedure;
    inače
      ako Vrh pokazuje na praznu referencu tada
        PomakniVrhNaPrethodni();
      inače
        Vrh := krajnje desno dijete od Vrh;
      kraj
    kraj
  kraj
kraj procedure
```

Funkcija stablenog stoga koja vraća element sa vrha stablenog stoga koristi procedure PomakniVrhNaPrethodni() i SrediPrviPomakVrha(), te ima slijedeći oblik:

```
funkcija DohvatiSaStablenogStoga()
  PomakniVrhNaPrethodni();
  ako prethodni pomak bio prvi pomak vrha stoga tada
    SrediPrviPomakVrha();
  vrati element na koji pokazuje Vrh;
kraj funkcije
```

Funkcija koja dohvaća element sa stoga koristi stableni stog i privremeni stog:

```
funkcija DohvatiSaStoga()
  ako je privremeni stog prazan tada
    vrati DohvatiSaStablenogStoga();
  inače
    vrati DohvatiSaPrivremenogStoga();
kraj funkcije
```

Procedura StaviNaStog() jednostavno stavlja element na privremeni stog. Ovime su potpuno definirane funkcije za rad sa stogom.

4.5. Obrada sintaksno neispravnih područja

Sintaksna greška u ulaznoj datoteci uzrokuje nepodudarnost (Def. 11) između ulazne datoteke i sintaksnog stabla. Sintaksne greške nastaju kao stvarne pogreške u uređivanju, ali i kao posljedica ispravnog uređivanja. Na primjer, svaki put kada se otvori lijeva zagrada, ostatak datoteke neće biti ispravan. Datoteka postaje sintaksno ispravna tek kada se zagrada zatvori u nekom daljnjem dijelu datoteke. Zbog postavljenog zahtjeva o nenametljivosti razvojne okoline prilikom pojave sintaksne pogreške, potrebno je sintaksno neispravne dijelove ulazne datoteke prikladno obrađivati.

Koristi se *označavanje nepodudarnih jedinki* ulazne datoteke i popravljanje *nepodudarnih područja*.

Elementarne promijene su *mijenjanje*, *umetanje* i *brisanje* samo jedne jedinke ulazne datoteke. Prilikom svake elementarne promijene odgovarajuća jedinka postavlja se u nepodudarno stanje. Prilikom mijenjanja postavlja se u nepodudarno stanje promijenjena jedinka, kod umetanja postavlja se umetnuta jedinka, a brisanjem neke jedinke postavlja se u nepodudarno stanje jedinka koja je došla na mjesto obrisane tj. ona jedinka koja je neposredno slijedila obrisanu. U prvom dijelu tablice 4.1 prikazane su jedinke koje se postavljaju u nepodudarno stanje pri svakoj elementarnoj promjeni.

Jedinke ulazne datoteke vraćaju se u podudarno stanje u trenutku uspješnog popravka neispravnog podstabla. Obilazi se uspješno popravljeno podstablo, te se za svaki neprazni završni znak pripadna jedinka ulazne datoteke postavlja u podudarno stanje.

Svaka jedinka ulazne datoteke ima pokazivač na znak u čvoru. Jedinke u podudarnom stanju pokazuju na pripadni znak u sintaksnom stablu. Jedinke u nepodudarnom stanju pokazuju na onaj znak u sintaksnom stablu od kojega treba početi proces popravka stabla. Pozicija popravka u sintaksnom stablu ovisna je o vrsti elementarne promjene. Kod mijenjanja jedinke pokazivač na pripadni znak ostaje isti. Prilikom umetanja jedinke pokazivač na pripadni znak preslikava se od jedinke koja je bila

na toj poziciji tj. neposredno slijedi umetnutu jedinku. Kod brisanja jedinka koja dolazi na mjesto obrisane zadržava svoj pokazivač na pripadni znak. U drugom dijelu tablice 4.1 prikazane su pozicije popravka za svaku elementarnu promjenu.

	JEDINKA KOJA SE POSTAVLJA U NEPODUDARNO STANJE	**NEPODUDARNA JEDINKA DOBIVA POKAZIVAČ OD**
MIJENJANJE	promijenjena jedinka	zadržava stari
BRISANJE	jedinka koja slijedi obrisanu	zadržava stari
UMETANJE	umetnuta jedinka	jedinke koja je bila na njegovom mjestu, tj. jedinke koja poslije umetanja slijedi umetnutu

Tablica 4.1. Obrada elementarnih promjena

Nakon što neki dio ulazne datoteke postane sintaksno ispravan, potrebno je popraviti cijeli taj dio. Taj uvjet nije moguće ispuniti ako se u nepodudarno stanje postavljaju samo jedinke koje su sudjelovale u elementarnim promjenama. Tvrdnja je pokazana na jednostavnom primjeru.

Pretpostavimo da ulazna datoteka sadržava podniz `"((1));"` koji je sintaksno ispravan s obzirom na danu gramatiku (Prilog A). Nakon brisanja prvoga znaka niza dobiva se novi niz `"(1));"` gdje je označen znak koji se nalazi u nepodudarnom stanju. Inkrementalni parser kreće sa parsiranjem od nepodudarne jedinke na mjestu promjene, tj. od znaka `"("`. Proces popravka ne uspijeva, jer parser prekida sa radom kod nailaska na krajnje desnu zatvorenu zagradu. Prekid je uzrokovan neispravnom sintaksnom strukturom (zatvorena zagrada bez pripadne otvorene). Brisanjem krajnje desne zatvorene zagrade dobiva se niz `"(1);"`. Označena su dva nepodudarna znaka. Inkrementalni parser kreće sa parsiranjem od znaka `";"`. Proces popravka neće uspjeti, iako je sada sintaksna struktura ispravna. Sintaksno stablo se nije promijenilo i stableni stog za znak `";"` i dalje odražava stanje koje je odgovaralo izvornom nizu `"((1));"`. Parsiranje se prekida na znaku `";"` zbog prividne sintaksne greške (nedostaje zatvorena zagrada). U ovom slučaju je za ispravan popravak potrebno krenuti sa parsiranjem od znaka `"("`, no on se nalazi na proizvoljnom mjestu u datoteci (koje nije unaprijed poznato).

Korištenje *nepodudarnih područja* omogućava obradu višestrukih sintaksnih grešaka. Prilikom pokušaja popravka, parser u ulaznoj datoteci postavlja u nepodudarno stanje sve one znakove koje stavlja na svoj stog. Nakon uspješnog parsiranja se svi znakovi koji su se našli u zamjenskom podstablu (znakovi koji su se stavljali na stog) vraćaju u podudarno stanje. Nakon neuspješnog parsiranja u

nepodudarnom stanju je cijeli dio ulazne datoteke koji je analiziran tijekom parsiranja. Nepodudarni dio datoteke predstavlja jedno nepodudarno područje. Promjena na bilo kojem mjestu unutar nepodudarnog područja ponekad uzrokuje ispravan popravak čitavog nepodudarnog područja.

Parser kreće sa parsiranjem od početka onog nepodudarnog područja koje obuhvaća promijenjeni dio ulazne datoteke. Za uspješan rad inkrementalnog sintaksnog analizatora nužno je da uspješan proces parsiranja ne stane na onoj jedinki ulazne datoteke iza koje slijedi nepodudarna jedinka. Budući da ponekad više uzastopnih nepodudarnih jedinki pokazuje na isti čvor sintaksnog stabla (pokazivač na mjesto popravka umnožava se pri umetanju neke jedinke), neispunjavanje prethodnog uvjeta uzrokuje pojavu pokazivača na nepostojeće čvorove stabla (oni koji su obrisani tijekom zamjene podstabla).

Uvodi se definicija:

Def. 15. *Pripadno nepodudarno područje* za neku nepodudarnu jedinku ulazne datoteke jest najveći neprekinuti niz nepodudarnih jedinki čiji je element i ta jedinka.

Pri svakoj promjeni ulazne datoteke popravlja se cijelo pripadno nepodudarno područje. Popravak pojedinih dijelova nepodudarnog područja nije dozvoljen. Korištenje nepodudarnih područja je pokazano na već upotrijebljenom primjeru.

Brisanjem prvoga znaka iz niza `"((1));"`, dobiva se niz `"(1));"`. Početak nepodudarnog područja je jedinka sa znakom `'('`. Parser postavlja u nepodudarno stanje sve jedinke do one u kojoj se zaustavlja parsiranje. Parsiranje se zaustavlja na krajnje desnom znaku `')'`. Novi niz je `"(1));"`. Nepodudarno područje prošireno je i obuhvaća tri jedinke ulazne datoteke. Brisanjem krajnje desne zagrade dobiva se niz `"(1);"`. Početak nepodudarnog područja je jedinka sa znakom `'('`. Popravlja se cijelo nepodudarno područje koje se ispravno prepoznaje kao izraz u naredbi pridruživanja. U sintaksnom stablu zamjenjuje se neispravno podstablo. Obilaskom novog podstabla jedinke se postavljaju u podudarno stanje i dobiva se niz `"(1);"`.

4.6. Uvjeti zaustavljanja parsiranja

Proces parsiranja staje u trenutku kad je neispravno podstablo moguće zamijeniti sa ispravnim podstablom ili kada nije moguć daljnji nastavak parsiranja. Nastavak parsiranja nije moguć kada ulazna datoteka sintaksno neispravna. Parser po nailasku na sintaksno neispravan znak izvodi operaciju odbaci (4.1.3.1).

Obični LR parser parsira ulazni niz sve do oznake kraja niza. To rješenje nije učinkovito, jer parsira i nepotrebni dio ulaznog niza.

Uvjeti uspješnog završetka parsiranja kod inkrementalnog LR parsera su:

(i) na privremenom stogu nalazi se samo jedan element koji je nužno referenca na nezavršni znak, a koji predstavlja vrh do sada izgrađenog podstabla *X*;

(ii) vrh stablenog stoga *V* je krajnje lijevo dijete čvora-roditelja *R* ili svi čvorovi-djeca čvora *R* koji se nalaze lijevo od *V* prazni;

(iii) ako je jedinka na koju pokazuje kazaljka ulazne datoteke različita od jedinke kraja datoteke, onda je ona u podudarnom stanju;

(iv) vrh podstabla *X* i vrh traženoga podstabla *T* su reference na isti nezavršni znak;

(v) podstablo *X* i traženo podstablo *T* imaju jednak broj nepraznih završnih znakova.

Posebno je obrazložen svaki od uvjeta.

Uvjet (i) je posljedica procesa parsiranja. Vrh zamjenskog podstabla je uvijek referenca na nezavršni znak, koja se nužno nalazi na privremenom stogu, jer je izgrađena u procesu parsiranja koji se upravo odvija (u stablenom stogu, tj. u stablu, nalaze se podstabla izgrađena u prethodnim procesima parsiranja). Kad bi na privremenom stogu bilo više od jednog elementa, postojalo bi više nevezanih podstabala ili bi postojali završni znakovi koji nisu uključeni u zamjensko podstablo.

Uvjet (ii) proizlazi iz slijedećeg razmatranja. Stableni stog pokazuje na prazno mjesto, te je zbog toga element na poziciji vrha stoga uključen u zamjensko stablo. Neposredno lijevi susjedi od vrha stoga nisu uključeni u zamjensko stablo (još se nalaze na stablenom stogu). U slučaju kada je traženi znak upravo čvor-roditelj (to je najbolji mogući slučaj), vrh stoga i svi desni čvorovi-djeca ostaju i nakon zamjene u novom stablu. No oni lijevi čvorovi-djeca neće biti u novom stablu (nisu dohvaćeni sa stablenog stoga). Ako je bilo koji od njih neprazan, onda postoji dio ulazne datoteke koji nije obuhvaćen sintaksnim stablom.

Uvjet (iii) rezultat je razmatranja o obradi nepodudarnih područja (Def. 15). Izuzetak je oznaka kraja ulaznog niza. Oznaka kraja niza je dodatno umetnut prazni završni znak, te se nikada neće naći na stogu.

Uvjet (iv) proizlazi iz zahtjeva da čvor-roditelj od čvora koji je vrh zamjenskog podstabla ostane lijeva strana iste produkcije. Nejednakost referenci vrha izgrađenog i vrha traženog podstabla bi uvjetovala primjenu druge produkcije gramatike, te bi sintaksno stablo na tom mjestu postalo neispravno.

Uvjet (v) proizlazi iz zahtjeva da zamjensko podstablo treba predstavljati jednaki dio ulazne datoteke kao i podstablo koje se zamjenjuje. Da bi ovaj uvjet bio zadovoljen, potrebno je kod umetanja

ili brisanja neke jedinke ulazne datoteke promijeniti dio čvorova sintaksnog stabla. Svakom čvoru na putu od vrha sintaksnog stabla do pripadnog završnog znaka treba promijeniti element strukture koji sadrži broj obuhvaćenih nepraznih završnih znakova. Na primjer, kod umetanja znaka se u svakom potrebnom čvoru broj obuhvaćenih nepraznih završnih znakova povećava za jedan. Nužnost promjene svakog čvora na putu od vrha stabla do pripadnog završnog znaka će kasnije biti dodatno pojašnjena.

Pokazivač na vrh traženog podstabla *T* se na početku procesa parsiranja postavlja na čvor-roditelj pripadnog završnog znaka. Vrijednost pokazivača na vrh traženog podstabla mijenja se tijekom parsiranja ulazne datoteke:

Teorem 5. Bilo koji nezavršni znak na putu od početnog nezavršnog znaka na vrhu stabla pa do prvotno traženog nezavršnog znaka obuhvaća dio ulazne datoteke u kojoj je jedinka sa kojom kreće proces parsiranja. Bilo koji takav nezavršni znak jest mogući vrh zamjenskog podstabla.

Dokaz teorema je slijedeći. Na neku jedinku ulazne datoteke ne pokazuje više od jednog završnog znaka (Def. 11). Dva različita podstabla u sintaksnom stablu mogu sadržavati pokazivač na istu jedinku ulazne datoteke jedino ako je jedno od ta dva podstabla upravo podstablo od onoga drugog. Jedino izravan čvor-roditelj i posredni čvorovi-roditelji (pra-roditelj, pra-pra-roditelj) obuhvaćaju istu jedinku ulazne datoteke kao i neki neprazni završni čvor.

Pokazivač na vrh traženog podstabla *T* mijenja se tijekom procesa parsiranja:

Teorem 6. Vrh stablenog stoga se nalazi u traženom podstablu *T*.

Teorem se dokazuje opovrgavanjem suprotne tvrdnje. Ako vrh stablenog stoga nije u traženom podstablu, onda nakon uspješnog završetka parsiranja i zamjene podstabala više dijelova sintaksnog stabla sadržava pokazivače na iste jedinke ulazne datoteke. Kopija znaka na mjestu vrha stablenog stoga uključena je u upravo izgrađeno podstablo (vrh pokazuje na element koji je uzet sa stablenog stoga). Nakon zamjene je taj znak u podstablu kojem je vrh traženi znak (traženo podstablo je tijekom zamjene obrisano). Budući da se vrh u trenutku zamjene nije nalazilo u traženom podstablu *T*, taj znak se također nalazi i na drugom mjestu u stablu, gdje pokazuje na istu jedinku ulazne datoteke. Nastalo sintaksno stablo je neispravno (Def. 11).

Zbog zadovoljavanja prethodnog teorema, pokazivač na traženo podstablo mijenja se sukladno sa pomicanjem vrha stablenog stoga. Pomicanje pokazivača na traženo podstablo je lako ugraditi u prije opisanu proceduru za rad sa stogom PomakniVrhNaPrethodni(). Ako vrh stablenog stoga izađe iz traženoga podstabla, onda se vrh traženog podstabla pomiče prema vrhu sintaksnog stabla (tako da ponovo obuhvati vrh stablenog stoga). Da bi bio ispunjen uvjet (v), potrebno je prije započinjanja LR

parsiranja promijeniti broj obuhvaćenih nepraznih završnih znakova za svaki mogući vrh traženog podstabla.

Traženo podstablo nije jednoznačno određeno (Teorem 5). Ako su ispunjeni uvjet (i), uvjet (ii) i uvjet (iii), onda je potrebno uvjet (iv) i uvjet (v) provjeriti za moguće vrhove traženog podstabla. Mogući vrhovi traženog podstabla su neizravni roditelji od čvora na koji pokazuje pokazivač vrha traženog podstabla.

4.7. Zamjena neispravnog podstabla

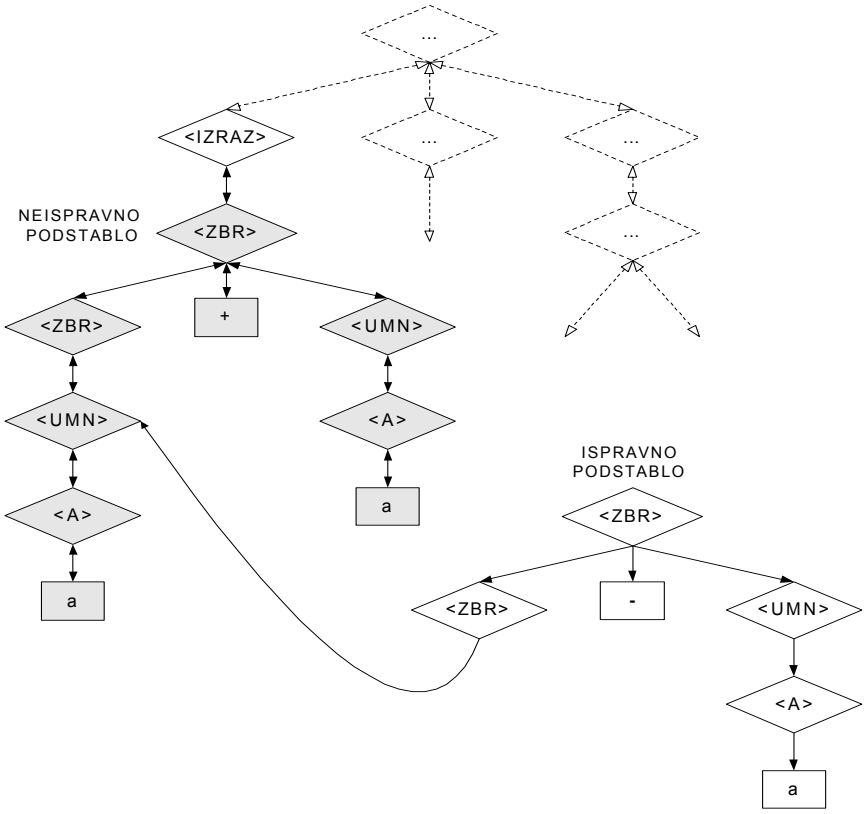
Ispunjavanjem nužnih uvjeta uspješnog završetka parsiranja, pokreće se postupak zamjene neispravnog podstabla s ispravnim podstablom. Postupak zamjene je složen, jer novo podstablo i staro podstablo dijele zajedničke čvorove.

Novo podstablo sadrži čvorove koji su dohvaćeni sa stablenog stoga. Stableni stog je prividni stog koji se nalazi u sintaksnom stablu. Pozivom funkcije koja dohvaća element sa stablenog stoga dobiva se preslik izvornog čvora u sintaksnom stablu. Preslik čvora pokazuje na čvorove-djecu koji se nalaze u izvornom sintaksnom stablu.

Na slici 4.11 prikazan je primjer zamjene neispravnoga podstabla. Tijekom izgradnje ispravnog podstabla dohvaćena je referenca na nezavršni znak <ZBR> sa stablenog stoga. U ispravnom stablu nalazi se preslik te reference na nezavršni znak. Čvor-dijete preslikane reference se i dalje nalazi u neispravnom podstablu. Izravno umetanje ispravnog podstabala i brisanje neispravnog stabla rezultira neispravnim sintaksnim stablom (postoje pokazivači na nepostojeće čvorove).

Veze između čvorova u sintaksnom stablu su dvosmjerne. Dvosmjerne veze su nužne, jer postoji potreba obilaska stabla od korijena prema listovima (npr. semantička analiza), te od listova prema korijenu (npr. stableni stog). Dvosmjerne veze se ne uspostavljaju tijekom izgradnje podstabla. Prilikom izgradnje novoga čvora postavljaju se pokazivači čvora-roditelja na čvorove-djecu. Pokazivač čvora-djeteta na čvor-roditelj ne smije se postaviti tijekom izgradnje podstabla, jer nije poznat ishod popravka. Uspješan popravak zahtjeva da čvor-dijete pokazuje na čvor-roditelj u novom podstablu. Neuspješan popravak (pojava sintaksne greške) zahtjeva da čvor-dijete zadrži pokazivač na čvor-roditelj u starom podstablu.

Uspješan popravak postavlja cijelo nepodudarno područje ulazne datoteke u podudarno stanje. Za one čvorove ispravnog podstabla koji predstavljaju nove neprazne reference na završne znakove potrebno je postaviti pripadne jedinke u podudarno stanje.



Slika 4.11: Primjer zamjene neispravnog podstabla

Zbog potreba učinkovite zamjene, čvorovi sintaksnog stabla dijele se u dva skupa: skup starih čvorova i skup novih čvorova. Podjelu je moguće ostvariti na više načina. Vremenski učinkovita podjela svakom čvoru pridjeljuje novi element koji bilježi pripadnost određenom skupu. Memorijski učinkovita podjela u privremenu strukturu (niz, lista) stavlja nove čvorove. Čvor je novi ako se nalazi u toj privremenoj strukturi (pretražuje se privremena struktura). Nakon završetka zamjene, privremena struktura se briše i oslobađa se zauzeta radna memorija.

Neuspješan popravak uzrokuje brisanje svih novih čvorova. Sintaksno stablo ostaje nepromijenjeno.

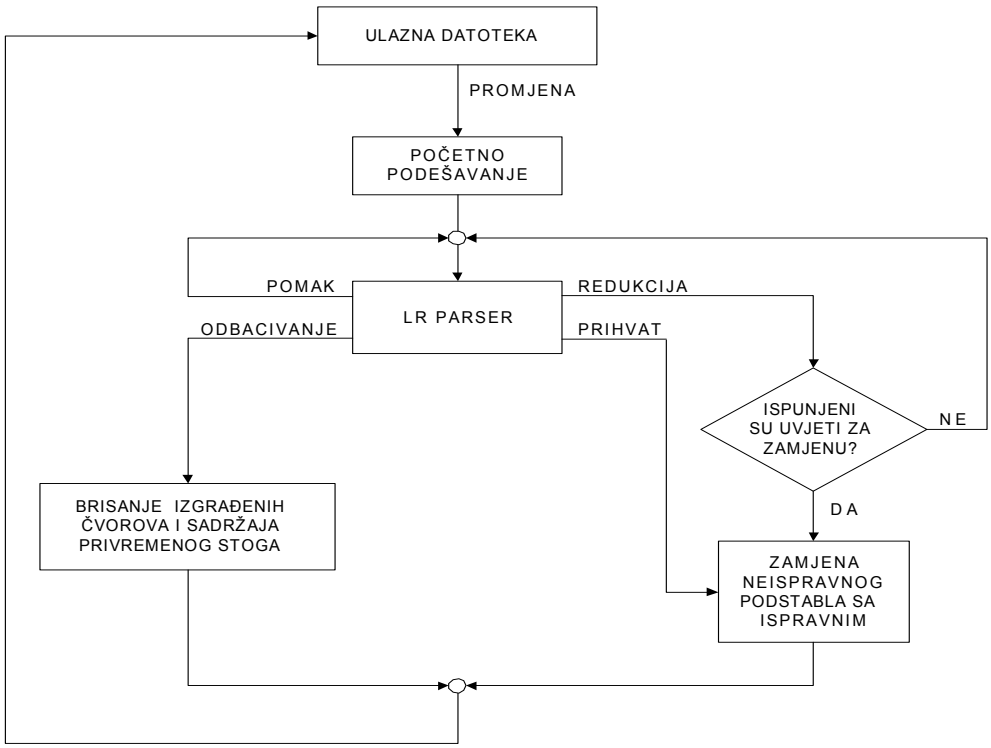
Uspješan popravak (ispunjeni su nužni uvjeti za zamjenu) uzrokuje pokretanje postupka zamjene. Ispravno podstablo obilazi se od korijena prema listovima. Pokazivač svakog čvora-djeteta postavlja se na pripadni čvor-roditelj. Za sve nove neprazne završne čvorove pripadne se jedinke postavljaju u podudarno stanje. Svaki čvor ispravnog podstabla koji nije u skupu novih čvorova stavlja se u skup novih čvorova. Dio neispravnog podstabla postaje dio ispravnog podstabla. Nakon završetka obilaska ispravnog podstabla pokreće se postupak brisanja neispravnog podstabla. Obilazi se

neispravno podstablo i brišu se svi čvorovi koji su u skupu starih čvorova. Dio neispravnog stabla koji je uključen u novo stablo se ne briše, jer se nalazi u skupu novih čvorova. Nakon brisanja se ispravno podstablo umeće na mjesto neispravnog u sintaksnom stablu i postupak zamjene je gotov.

4.8. Postupak popravka sintaksnog stabla

Inkrementalni sintaksni analizator je u ovom slučaju LR parser proširen za potrebe inkrementalne izgradnje sintaksnog stabla. Obični LR parser proširen je posebnim strukturama podataka (4.3), funkcijama za rad sa stablenim stogom (4.4), postupcima obrade sintaksno neispravnih stanja (4.5), te postupkom zamjene neispravnog podstabla sa ispravnim podstablom (4.7).

Ulazne datoteke koje se otvaraju u sintaksno svjesnoj razvojnoj okolini pretvaraju se u jedno neprekinuto nepodudarno područje. Popravak cijelog područja odgovara izgradnji cijelog sintaksnog stabla kod običnog LR parsera. Na slici 4.12 prikazan je pojednostavljeni postupak popravka sintaksnog stabla.



Slika 4.12: Postupak popravka sintaksnog stabla

Promjena u ulaznoj datoteci prikladno se obrađuje (4.5). Postupak popravka započinje početnim podešavanjem. Pronalazi se početak pripadnog nepodudarnog područja (Def. 15), postavlja se vrh stablenog stoga, te se postavlja pokazivač na početni vrh traženog podstabla. LR parser ima sve strukture potrebne za svoj rad (4.1.3.1). Izvodi se jedan korak rada LR parsera, a time se izvodi jedna od četiri moguće operacije. Daljnji postupak popravka ovisi o izvedenoj operaciji LR parsera.

Pomak uzrokuje nastavak rada LR parsera.

Odbacivanje uzrokuje brisanje svih čvorova sintaksnog stabla koji su izgrađeni tijekom postupka popravka koji se upravo odvija. Briše se sadržaj privremenog stoga. Jedinke ulazne datoteke koje su postavljene u nepodudarno stanje tijekom postupka popravka, ostaju u nepodudarnom stanju. Omogućena je ispravna obrada višestrukih sintaksnih grešaka u ulaznoj datoteci (4.5).

Prihvat se događa u slučaju parsiranja cijele ulazne datoteke. Provjera uvjeta zamjene je suvišna, jer je cijela ulazna datoteka parsirana i izgrađeno je cijelo sintaksno stablo. Referenca na početni nezavršni znak u glupom čvoru postaje korijen sintaksnog stabla.

Redukcija zahtjeva provjeru uvjeta uspješnog završetka popravka (4.6). Ako neki od uvjeta nije ispunjen, onda se izvodi slijedeći korak LR parsera. Uvjeti se provjeravaju za sve čvorove sintaksnog stabla koji su vrhovi mogućeg podstabla za zamjenu. Zadovoljavanje svih nužnih uvjeta uzrokuje pokretanje postupka zamjene (4.7). Postupak zamjene uspostavlja sve dvosmjerne veze u sintaksnom stablu, postavlja potrebne jedinke ulazne datoteke podudarno stanje, te briše nepotrebne čvorove iz neispravnog podstabla. Nakon završetka postupka zamjene, završen je popravak sintaksnog stabla. Broj nepodudarnih područja ulazne datoteke smanjio se barem za jedan.

Pojedinosti izvedbe mogu se naći u izvornom kôdu pokaznog programa (Prilog B). Izloženi postupci otvoreni su za daljnje promjene i poboljšanja.

5.

POKAZNI PROGRAM

Korištenjem izloženih postupaka inkrementalne leksičke i inkrementalne sintaksne analize izgrađen je pokazni program. Izvorni programski kôd pokaznog programa priložen je u prilogu B (ispis programa) i u prilogu C (Visual C++ 6.0 projekt). Sam pokazni program priložen je kao prilog D. Prilozi C i D zapisani su na CD-ROM jedinku pohrane podataka.

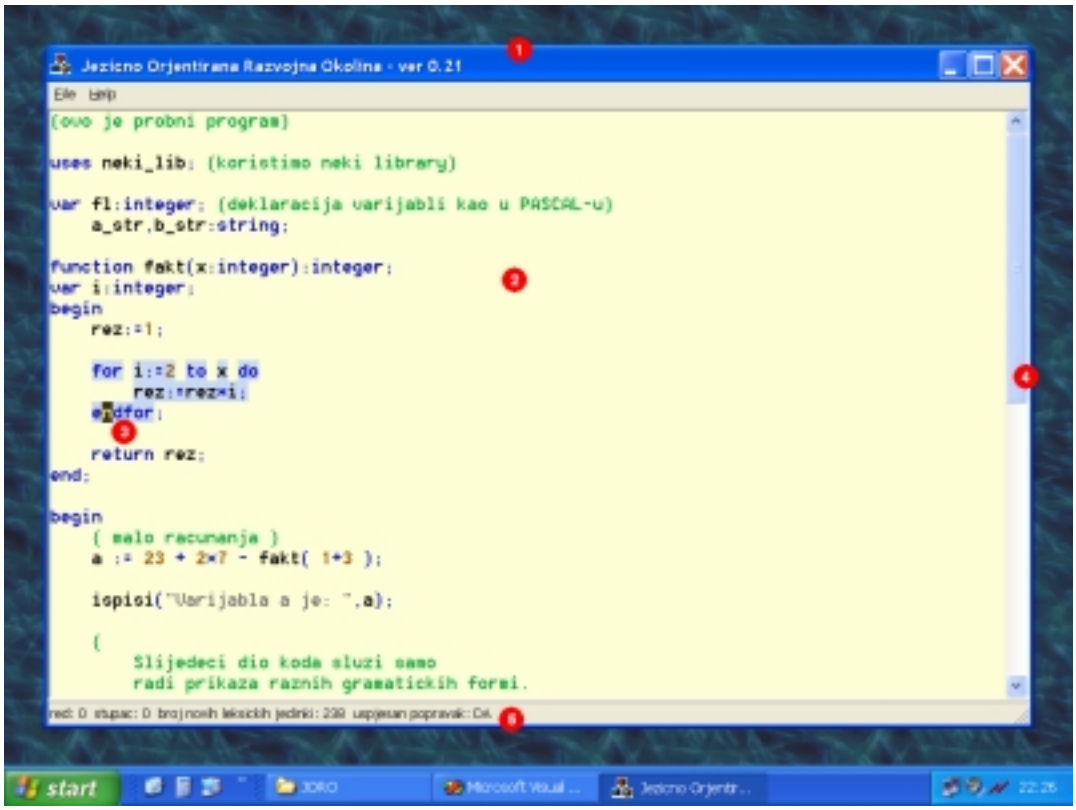
5.1. Pokretanje pokaznog programa

Za pokretanje pokaznog programa potrebno je IBM PC podudarno računalo sa CD-ROM čitačem i prisutnim Windows 95, 98, NT 4.0, 2000 ili XP operacijskim sustavom. Preporučuje se računalo sa radnim taktom većim od 200 MHz i minimalno 32 MB radne memorije. Za pregledan zaslonski prikaz preporučuje se zaslonska razlučivost od 800x600 ili 1024x768 elemenata, uz korištenje 16 ili 24 bitovne boje.

Potrebno je priloženi CD-ROM sa pokaznim programom staviti u CD-ROM čitač odabranog računala. Korištenjem bilo kojeg alata priloženog u Windows operacijskom sustavu (npr. Windows Explorer) treba se postaviti u korijensku mapu CD-ROM uređaja. Unutar korijenske mape nalazi se mapa "`Prilog D`" u kojoj se nalazi izvršna datoteka `JORO.exe` (nastavak `exe` nije uvijek vidljiv). Pokretanjem datoteke `JORO.exe`, nakon kraće pauze (zbog kreiranja tabela potrebnih za leksičke i sintaksne analizatore) na zaslonu računala prikazuje se glavni prozor programa. U slučaju pojave greške, potrebno je provjeriti ispunjava li računalo nužno potrebne zahtjeve.

5.2. Sučelje pokaznog programa

Sučelje programa prikazano je na slici 5.1. Glavni elementi sučelja pobrojani su rednim brojevima 1, 2, 3, 4 i 5.



Slika 5.1: Sučelje pokaznog programa s pobrojanim glavnim elementima sučelja

Glavni prozor (1) pokazne jezično orjentirane razvojne okoline sastoji se od više dijelova. Središnji dio je prostor uređivača teksta (2), gdje se prikazuje kôd programa koji se uređuje. Ukoliko se ulazna datoteka ne može u cijelosti prikazati na zaslonu, pojavljuje se klizna traka (4), koja omogućava kretanje kroz datoteku. Statusna traka (5) prikazuje informacije o trenutnom statusu uređivanja. Prikazane su trenutne koordinate pokazivača unosa teksta (tj. pripadni red i stupac), te informacije koje pružaju leksički i sintaksni analizator. Inkrementalni leksički analizator u statusnoj traci ispisuje broj novih leksičkih jedinki koje su analizirane u posljednjem procesu osvježavanja. Inkrementalni sintaksni analizator ispisuje uspješnost procesa popravka sintaksnog stabla. Ukoliko je ispis "NE", popravak nije bio uspješan i datoteka nije sintaksno ispravna. Kroz ulaznu datoteku kreće se korištenjem pokazivača unosa teksta (3). Njegova pozicija može se mijenjati korištenjem uobičajenih tipki pomaka, kao i pritiskom na lijevo dugme miša na određeno mjesto u područje uređivanja. Kao što je vidljivo na slici, različite leksičke jedinke ulaznog programa označavaju se različitim bojama. Također, sintaksna struktura na poziciji pokazivača unosa teksta se označava isticanjem boje pozadine.

Pokazni program učitava automat s definicijom leksičke strukture jezika, tekstualnu datoteku s LR(1) gramatikom izvornog jezika te pokaznu ulaznu datoteku s programom u definiranom jeziku. Jednostavni programski jezik koji se koristi u ovome pokaznom programu je vrlo sličan programskom

jeziku PASCAL, te sadrži sve bitne elemente nekog programskog jezika: operatorske izraze, blokove naredbi, funkcije, deklaracije varijabli, petlje, komentare, dekadske, oktalne, binarne, heksadecimalne i realne brojeve, nizovne i znakovne konstante. Budući da je taj jezik opisan samo deterministčkim konačnim automatom (leksička analiza) i LR(1) gramatikom (sintaksna analiza), svaki jezik koji je opisiv DKA i LR(1) formalizmima također se može analizirati u priloženoj pokaznoj razvojnoj okolini.

Ulazna datoteka koja se automatski učitava u pokaznu razvojnu okolinu je jednostavan program koji demonstrira bitne elemente korištenog programskog jezika. Odmah po učitavanju ulazna datoteka se analizira, a rezultat analize se, osim u prostoru uređivača teksta, može vidjeti i u statusnoj traci. Inkrementalni leksički analizator prijavljuje 238 novih uspješno analiziranih leksičkih jedinki. Budući da je to prva analiza, datoteka se mora analizirati u cijelosti, te je to sveukupni broj leksičkih jedinki u toj datoteci. Inkrementalni sintaksni analizator gradi sintaksno stablo koje odgovara sintaksnoj strukturi ulazne datoteke. Ispisuje se da je parsiranje bilo uspješno, te je ulazna datoteka sintaksno ispravna.

Po pokretanju pokaznog programa žarište unosa je u uređivaču teksta. To se očituje u treperenju pokazivača mjesta unosa teksta. Svaka pritisnuta tipka obrađuje se u uređivaču teksta. Ako se žarište unosa prenese na neki drugi dio prozora, kao što je glavni izbornik, onda je za nastavak uređivanja žarište potrebno vratiti u uređivaču teksta. Prebacivanje žarišta obavlja se klikom miša u prostor uređivača teksta.

5.3. Provjera ispravnosti

Na osnovu informacija koje pružaju izgrađeni leksički i sintaksni analizator, razvojnoj okolini su dodane tri jezično orjentirane funkcije. Pobliže će biti opisana svaka od njih.

Označavanje različitih klasa leksičkih jedinki je funkcija koja koristi informacije inkrementalnog leksičkog analizatora. Različite klase leksičkih jedinki iscrtavaju se u različitim bojama, kao što se vidi na slici 5.1. Vidljivo je da se ispravno iscrtavaju i one leksičke jedinke koje se prostiru u više redova, kao što su komentari. Ukoliko je dio ulazne datoteke leksički neispravan, iscrtava se crvenom bojom. Leksička analiza je korisniku prividno istodobna s promjenama ulazne datoteke.

Označavanje sintaksno neispravnih područja je funkcija koja koristi informacije inkrementalnog sintaksnog analizatora. Sintaksno neispravno područje se podcrtava crvenom linijom. Primjerice, ukoliko se obriše prvi znak u ulaznoj datoteci (znak "{"), tekst koji se nalazio unutar komentara će prvo biti leksički, a zatim i sintaksno analiziran. Takav početak datoteke je sintaksno neispravan, te se leksičke jedinke u prva tri reda podcrtavaju crvenom linijom. Brisanjem cijelog prvog reda, datoteka ponovo postaje sintaksno ispravna. Ovu korisnu funkciju razvojne okoline nemoguće je ostvariti bez izgradnje inkrementalnog sintaksnog analizatora.

Označavanje pripadne sintaksne strukture je funkcija koja koristi informacije inkrementalnog sintaksnog analizatora. Sintaksna struktura na poziciji pokazivača unosa teksta označava se isticanjem boje pozadine. Time je olakšano snalaženje u programskom kodu, jer lako uočavaju pripadni blokovi naredbi, petlje, strukturirane naredbe. Također se uočavaju i pripadne oble i uglate zagrade. Primjerice, pozicioniranjem pokazivača unosa teksta na ključnu riječ "`function`" u ulaznoj datoteci, označava se cijela pripadna funkcija. Pozicioniranjem na ključnu riječ "`endfor`", označava se pripadna petlja. Pozicioniranjem na lijevu oblu zagradu u složenom izrazu, označava se cijeli izraz do pripadne desne oble zagrade.

Izgrađena jezično orjentirana razvojna okolina dozvoljava višestruka leksički i sintaksno neispravna područja. Ukoliko se u datoteci koja ima više neispravnih područja napravi promjena koja jedno od tih područja čini ispravnim, razvojna okolina će analizirati samo to područje te ga označiti kao ispravno. Time je pokazana lokalnost analize i učinkovitost priloženih inkrementalnih postupaka.

Opsežnije ispitivanje napravljeno je neprekidnim umetanjem i brisanjem znakova ulazne datoteke. Nakon 20 minuta neprekidnog uređivanja nije došlo do pojave greške. Datoteka je ispravno analizirana, iako je bila bitno različita od početne izvorne datoteke. Pozadinski rad inkrementalnog leksičkog i sintaksnog analizatora nije uzrokovao primjetno zastajkivanje u odgovoru pokaznog programa.

Daljnje ispitivanje rada moguće je provesti mijenjanjem leksičke ili gramatičke specifikacije programskog jezika. U mapi `podaci`, koja se nalazi u korijenskoj mapi "`Prilog D`", nalazi se više datoteka. U tablici 5.1. ukratko je opisana namjena svake datoteke.

NAZIV DATOTEKE	**OPIS**
`JednostavanJezik.tr`	Pravila transliteracije za programski jezik. Transliteracija je predkorak leksičke analize koji selektira skupove znakova koji su leksički bitni za taj jezik.
`JednostavanJezik.dka`	Definicija determinističkog konačnog automata koji opisuje leksičku strukturu programskog jezika.
`JednostavanJezik.pal`	Paleta boja koje se koriste za označavanje pojedinih leksičkih jedinki u razvojnoj okolini.
`gramatika.dat`	LR(1) gramatika koja opisuje sintaksu programskog jezika.
`prog.txt`	Ulazna datoteka s programom koja se učitava po pokretanju pokaznog programa.

Tablica 5.1: Ulazne datoteke pokaznog programa

Mijenjanjem navedenih datoteka, moguće uvelike promijeniti ponašanje pokaznog programa i prilagoditi ga nekom drugom programskom jeziku.

5.4. Prijedlozi za poboljšanja

Izrađeni pokazni program ispravno radi, te potvrđuje ispravnost opisanih postupaka inkrementalne analize. Neizravno je pokazano da se mogu izvesti i složenije operacije uređivanja (umetanje i brisanje nizova znakova), jer se sve složene operacije mogu razložiti na elementarne (umetanje i brisanje jednog znaka). Obrada grešaka izvedena je na način nenametljiv korisniku. Brzina parsiranja u pokaznom primjeru bila je i više nego zadovoljavajuća.

Prije uporabe u pravoj razvojnoj okolini, koja postavlja mnogo veće zahtjeve na rad, potrebno je analizirati učinkovitost korištenih postupaka i predložiti moguća poboljšanja. Poboljšanja učinkovitosti mogu se razložiti na poboljšanja *vremenske* (uklanjanje zastoja u odazivu korisniku) i poboljšanja *memorijske* (manje zauzeće radnje memorije) učinkovitosti.

Vremenska neučinkovitost, tj. zastoj u odazivu, nastaje kada je ulazna datoteka velika, a nova promjena dogodila se na mjestu koje uzrokuje ponovnu izgradnju velikog dijela sintaksnog stabla. Pozicija takvih mjesta ovisi o gramatici izvornog jezika, a obično su to mjesta gdje počinju ili završavaju velike sintaksne strukture. U slučaju stvarno velikih sintaksnih struktura moguć je primjetni zastoj u odazivu prema korisniku. Problem je moguće razriješiti korištenjem više dretvi. Osnovna zamisao takvog pristupa je da inkrementalni sintaksni analizator radi popravak sintaksnog stabla samo u slobodnom procesorskom vremenu. Dretva sučelja bi u tom slučaju bila dretva višeg prioriteta koja bi samostalno obavljala uređivanje datoteke. Poruke o promjenama ulazne datoteke bi se prosljeđivale dretvi inkrementalnog sintaksnog analizatora koja bi provodila popravke sintaksnog stabla ne ometajući rad razvojne okoline. Inkrementalni sintaksni analizator bi tada bio pozadinska ili radna dretva (engl. working thread) [7]. Problematiku usklađivanja rada dretve sučelja i dretve inkrementalnog sintaksnog analizatora potrebno je u tom slučaju detaljnije riješiti.

Memorijska neučinkovitost jest posljedica velike memorijske zahtjevnosti strukture sintaksnog stabla. Zauzeće memorije ovisi o gramatici izvornog jezika (gramatika utječe na visinu sintaksnog stabla). U pretpostavljenom slučaju gdje produkcije konteksno-neovisne gramatike na svojoj desnoj strani imaju točno dva završna ili točno dva nezavršna znaka, sintaksno stablo postaje binarno stablo. Veličina ulazne datoteke je `N`, gdje je `N` broj leksičkih jedinki u datoteci. Sintaksno stablo tada ima `2*N-1` čvorova [8]. Zauzeće radne memorije je `(2*N-1)*V*U`, gdje je `V` veličina pojedinog čvora sintaksnog stabla, a `U` konstanta ovisna o učinkovitosti funkcija operacijskog sustava za rukovanje memorijom [7]. Veličina `M`, koja predstavlja omjer potrebne radne memorije i veličine ulazne datoteke, je tada:

$$M = \frac{(2 \times N - 1) \times V \times U}{N} \qquad (1)$$

Kod velikih datoteka se zanemaruje konstantan pribrojnik u (1), čime se dobiva:

$$M = 2 \times V \times U \qquad (2)$$

Uz pretpostavljene vrijednosti `V=36` i `U=2`, za omjer `M` dobiva se vrijednost `M=144`. Uza sve navedene pretpostavke, sintaksno stablo ulazne datoteke veličine 100 000 leksičkih jedinki zauzima 14 MB radne memorije.

Iako rezultat izgleda zadovoljavajući (radne memorije današnjih računala kreću se u rasponu od 64-512 MB), moguć je i znatno lošiji ishod, zbog segmentacije radne memorije i oblika sintaksnog stabla.

Ovaj problem je ispravno bio adresiran u mom seminarskom radu koji se bavio isključivo tematikom sintaksne analize [11]. Tada sam sugerirao da uvođenje dodatnog koraka leksičke analize značajno smanjuje memorijsko zauzeće, jer se više ulaznih znakova grupira u jednu leksičku jedinku. Pretpostavka se pokazala ispravnom, jer pokazna jezično orjentirana razvojna okolina izgrađena u ovom radu znatno učinkovitije koristi radnu memoriju.

Dodatno smanjenje memorijskog zauzeća moguće je postići optimiranjem struktura sintaksnog stabla i korištenjem posebnih funkcija za rad s malim dijelovima memorije. Time se uklanja neodgovarajuće memorijsko poravnavanje i znatno smanjuje segmentacija memorije. Memorijsko poravnavanje postavlja elemente struktura na memorijska mjesta koja su višekratnici neke vrijednosti (npr. 4 ili 8), bez obzira na njihovu stvarnu veličinu. Segmentacija memorije uzrokovana je niskom učinkovitošću uobičajenih funkcija za rad sa memorijom pri korištenju malih dijelova memorije. Često zauzimanje i brisanje malih dijelova memorije također povećava segmentaciju radne memorije.

Može se zaključiti da postupci opisani u ovom radu omogućavaju izgradnju učinkovite jezično orjentirane razvojne okoline. Poboljšanja su potrebna prvenstveno u inkrementalnom sintaksnom analizatoru, koji se može poboljšati na način da inteligentnije obrađuje specijalne slučajeve kod analize rubnih područja velikih sintaksnih struktura, te da učinkovitije koristi radnu memoriju računala.

6.

ZAKLJUČAK

U ovom radu izlažu se i ispituju postupci koji omogućavaju izgradnju učinkovite jezično orjentirane razvojne okoline. Detaljno je opisan način izrade općenitog inkrementalnog leksičkog analizatora, te inkrementalnog sintaksnog analizatora. Izložen je i postupak obrade neispravnih područja u ulaznoj datoteci, te postupak oporavka inkrementalnih analizatora od greške.

Opisani postupci prebacuju jedan dio analize iz jezičnog procesora u razvojnu okolinu, čime ista postaje jezično svjesna. Time je omogućeno trenutno upozoravanje korisnika na leksičke i sintaksne greške, te brže uređivanje i lakše mijenjanje jezičnih struktura.

Izrađen je pokazni program koji koristi postupke i strukture podataka izložene u ovome radu. Na njemu je pokazana ispravnost i učinkovitost postupaka. Pokazni program učinkovito osvježava unutrašnju reprezentaciju programa u skladu s promjenama koje nastaju tijekom uređivanja programa u razvojnoj okolini. Brzina osvježavanja je zadovoljavajuća i kod većih ulaznih datoteka.

Pokazni program radi leksičku analizu korištenjem automata zadanog u odvojenoj datoteci. Time je moguća promjena leksike ulaznog jezika bez mijenjanja izvornog kôda pokaznog programa. Pokazni program radi za proizvoljnu LR(1) gramatiku koju je također moguće promijeniti bez mijenjanja izvornog kôda pokaznog programa. Informacije o ulaznom programu, koje su rezultat rada inkrementalnog leksičkog i sintaksnog analizatora, mogu se iskoristiti za dodavanje brojnih jezično svjesnih svojstava razvojnoj okolini. Pokazna razvojna okolina je nadograđena sa tri takva svojstva: bojanje leksičkih jedinki, označavanje neispravnih sintaksnih struktura, te označavanje pripadne sintaksne strukture za trenutnu poziciju pokazivača mjesta unosa. U prilogu je priložen cijeli izvorni kôd pokaznog programa.

Analizom rada pokaznog programa pokazano je da su postupci dovoljno učinkoviti. Dan je zaključak u kojem smjeru bi trebala ići daljnja poboljšanja. Potrebno je riješiti problem posebnih slučajeva (kod kojih dolazi do analize cijele ulazne datoteke). Također je potrebno dodatno smanjiti memorijske zahtjeve inkrementalnog sintaksnog analizatora.

Dio koji je bitan za problematiku jezično orjentiranih razvojnih okolina, a nije obrađen unutar ovog rada, jesu postupci inkrementalne semantičke analize. To područje je izuzetno složeno i za sada još nije prikladno riješeno u stručnoj literaturi. Problem koji se javlja i prije izrade samog semantičkog analizatora jest način jedinstvene prezentacije semantike nekog programskog jezika. Proizvoljan programski jezik možemo leksički i sintaksno točno opisati korištenjem poznatih formalizama (DKA, LR(1) gramatika). Međutim, ne postoji zadovoljavajući i općeprihvaćeni formalizam za opis semantike programskog jezika.

Možemo zaključiti da u ovom području računarske znanosti postoji još mnogo tema za daljnja istraživanja, te mnogo prostora za daljnja unaprjeđenja u budućim jezično orjentiranim razvojnim okolinama.

7.

LITERATURA

[1] S. Srbljić: *"Jezični procesori II: Analiza izvornog i sinteza ciljnog programa"*, Element, Zagreb, 2002.

[2] A.A. Khwaja, J.E. Urban: *"Syntax-directed editing environments: issues and features"*, Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice, 14-16 Feb, Indianapolis, 1993, pp.230-237.

[3] Alice: The Personal Pascal, Looking Glass Software Limited, 1985.
URL: http://www.templetons.com/brad/alice.html

[4] HoTMetaL, SoftQuad Software, 1999.
URL: http://www.softquad.com

[5] Microsoft Visual BASIC 6.0, Microsoft Corporation., 1998.
URL: http://www.microsoft.com

[6] S. Srbljić: *"Jezični procesori I: Uvod u teoriju formalnih jezika, automata i gramatika"*, Element, Zagreb, 2000.

[7] C. Petzold: *"Programming Windows 95"*, Microsoft Press, Redmond, Washington, 1996.

[8] R. Sedgewick: *"Algorithms in C"*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[9] Microsoft Visual Studio .NET, Microsoft Corporation., 2002.
URL: http://www.microsoft.com

[10] T. A. Wagner: *"Practical Algorithms for Incremental Software Development Environments"*, EECS Department, University of California, Berkeley, 1998.

[11] Ž. Švedić: *"Dinamička izgradnja sintaksnog stabla"*, seminarski rad, Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave, Zagreb, 2001.

[12] Microsoft Visual C++ 6.0, Microsoft Corporation., 1998.
URL: http://www.microsoft.com

8.

PRILOG A: Gramatika pokaznog jezika

```
[nezavrsni]
<PROGRAM>       0
<USES_DIO>      0
<DEKL_L>        0
<GLAVNI_DIO>    0
<BLOK>          0
<USES_L>        0
<DEKL>          0
<VAR_DEKL>      0
<FUN_DEKL>      0
<FN_ARG_L_EPS>  0
<FN_ARG_L>      0
<VISE_TIPOVA>   0
<JEDAN_TIP>     0
<IDN_NIZ>       0
<TIP>           0
<NAR_NIZ>       0
<NAR>           0
<IF_NAR>        0
<FUN_POZIV>     0
<PRIDRUZI>      0
<ARG_L_EPS>     0
<ARG_L>         0
<IZRAZ>         0
<IF_NAST>       0
<SUMA>          0
<UMNOZAK>       0
<ZAGRADE>       0
<POREDBE>       0
<FOR_NAR>       0
<TO_DOWNTO>     0
<REPEAT_NAR>    0
<WHILE_NAR>     0
<RETURN_NAR>    0

[zavrsni]
uses
var
function
integer
string
char
begin
end
```

```
if
then
else
endif
for
to
downto
do
endfor
repeat
until
while
endwhile
return
<
>
>=
<=
=
!=
.
,
;
(
)
:
:=
+
-
*
/
IDN
ASM
STR
KARAKTER
DEK
REALNI
HEX
OKT
BIN

[akcijski]

[pocetni]
<PROGRAM>

[produkcije]
<PROGRAM>       -> <USES_DIO> <DEKL_L> <GLAVNI_DIO>

<GLAVNI_DIO>    -> <BLOK> .

<USES_DIO>      -> uses IDN <USES_L>
                -> eps

<USES_L>        -> , IDN <USES_L>
                -> ;

<DEKL_L>        -> <DEKL> <DEKL_L>
                -> eps

<DEKL>          -> <VAR_DEKL>
                -> <FUN_DEKL>

<VAR_DEKL>      -> var <JEDAN_TIP> ; <VISE_TIPOVA>
```

```
<VISE_TIPOVA>   -> <JEDAN_TIP> ; <VISE_TIPOVA>
                -> eps

<JEDAN_TIP>     -> IDN <IDN_NIZ> : <TIP>

<IDN_NIZ>       -> , IDN <IDN_NIZ>
                -> eps

<FUN_DEKL>      -> function IDN ( <FN_ARG_L_EPS> ) : <TIP> ; <DEKL_L> <BLOK>
;

<FN_ARG_L_EPS>  -> <FN_ARG_L>
                -> eps

<FN_ARG_L>      -> <JEDAN_TIP> ; <FN_ARG_L>
                -> <JEDAN_TIP>

<TIP>   -> integer
        -> string
        -> char

<BLOK>  -> begin <NAR_NIZ> end

<NAR_NIZ>       -> <NAR> ; <NAR_NIZ>
                -> eps

<NAR>           -> <IF_NAR>
                -> <FUN_POZIV>
                -> <PRIDRUZI>
                -> <FOR_NAR>
                -> <REPEAT_NAR>
                -> <WHILE_NAR>
                -> <RETURN_NAR>

<FUN_POZIV>     -> IDN ( <ARG_L_EPS> )

<ARG_L_EPS>     -> <ARG_L>
                -> eps

<ARG_L>         -> <IZRAZ> , <ARG_L>
                -> <IZRAZ>

<PRIDRUZI>      -> IDN := <IZRAZ>

<IF_NAR>        -> if <IZRAZ> then <NAR_NIZ> <IF_NAST> endif

<IF_NAST>       -> else <NAR_NIZ>
                -> eps

<IZRAZ>         -> <POREDBE>

<POREDBE>       -> <POREDBE> = <SUMA>
                -> <POREDBE> != <SUMA>
                -> <POREDBE> < <SUMA>
                -> <POREDBE> > <SUMA>
                -> <POREDBE> <= <SUMA>
                -> <POREDBE> >= <SUMA>
                -> <SUMA>

<SUMA>          -> <SUMA> + <UMNOZAK>
                -> <SUMA> - <UMNOZAK>
                -> <UMNOZAK>
```

```
<UMNOZAK>       -> <UMNOZAK> * <ZAGRADE>
                -> <UMNOZAK> / <ZAGRADE>
                -> <ZAGRADE>

<ZAGRADE>       -> ( <IZRAZ> )
                -> <FUN_POZIV>
                -> IDN
                -> STR
                -> KARAKTER
                -> DEK
                -> REALNI
                -> HEX
                -> OKT
                -> BIN

<FOR_NAR>       -> for IDN := <IZRAZ> <TO_DOWNTO> <IZRAZ> do <NAR_NIZ> endfor

<TO_DOWNTO>     -> to
                -> downto

<REPEAT_NAR>    -> repeat <NAR_NIZ> until <IZRAZ>

<WHILE_NAR>     -> while <IZRAZ> do <NAR_NIZ> endwhile

<RETURN_NAR>    -> return
                -> return <IZRAZ>

[kraj]
```

9.

PRILOG B: Izvorni kôd pokaznog programa

9.1. Datoteke inkrementalnog leksičkog analizatora

9.1.1. TDatoteka.h

```
//
//   TDatoteka - template za containere u obliku datoteke
//

#if !defined(TDATOTEKA_H)
#define TDATOTEKA_H

////////////////////////////////////////////////////////////////////

template< class T >
class TDatoteka
{
public:
     virtual ~TDatoteka() { }

     class Iterator
     {
     public:
          Iterator(): BrRed(-1), BrStup(-1), pDat(NULL) {}

          Iterator(int red, int stup, TDatoteka< T > *pd):
               BrRed(red), BrStup(stup), pDat(pd)
          {
               PopraviAkoTreba();
          }

          void Postavi(int red, int stup, TDatoteka< T > *pd=NULL)
          {
               BrRed = red;
               BrStup = stup;

               if(pd!=NULL) pDat = pd;

               PopraviAkoTreba();
          }
```

```
int VratiRed() { return BrRed; }

int VratiStup() { return BrStup; }

TDatoteka< T > * VratiDat() { return pDat; }

Iterator & operator=(Iterator & rhs)
{
     BrRed = rhs.BrRed;
     BrStup = rhs.BrStup;
     pDat = rhs.pDat;

     return *this;
}

bool Eof()
{
     if( BrRed==-1 && BrStup==-1 ) return true;
     else return false;
}

void PopraviAkoTreba()
{
     if( BrRed>=pDat->VratiBrojRedova() ||
          BrStup>=pDat->VratiVelReda(BrRed) )
          BrRed = BrStup = -1;
}

bool operator!=(Iterator & right)
{
     if( BrRed!=right.BrRed ||
          BrStup!=right.BrStup ||
          pDat!=right.pDat)
          return true;

     return false;
}

Iterator operator++()
{
     BrStup++;

     while( BrStup >= pDat->VratiVelReda(BrRed) )
     {
          BrRed++;
          if( BrRed >= pDat->VratiBrojRedova() )
          {
               BrRed = BrStup = -1;
               break;
          }

          BrStup=0;
     }

     return *this;
}

Iterator operator--()
{
     BrStup--;

     while( BrStup<0 )
     {
          BrRed--;
          if( BrRed<0 )
          {
               BrRed = BrStup = -1;
               break;
          }

          BrStup = pDat->VratiVelReda(BrRed)-1;
     }
```

```
            return *this;
        }

        Iterator operator+=(int pomak)
        {
            throw string("operator+= nije implementiran");

            return *this;
        }

        T & operator*()
        {
            return pDat->BrzoVratiElNaPoz(BrRed,BrStup);
        }

    protected:
        int BrRed;
        int BrStup;
        TDatoteka< T > *pDat;
    };

    friend Iterator;

    virtual int VratiBrojRedova()=0;
    virtual int VratiVelReda(int koji)=0;

    void ProvjeriRed(int red)
    {
        if( red<0 || red>=VratiBrojRedova() )
        {
            stringstream ss;
            ss << "ProvjeriRed(" << red << ")";
            throw string(ss.str());
        }
    }

    void ProvjeriPoz(int red, int stupac)
    {
        ProvjeriRed(red);

        if( stupac<0 || stupac>=VratiVelReda(red) )
        {
            stringstream ss;
            ss << "ProvjeriPoz(" << red << "," << stupac << ")";
            throw string(ss.str());
        }
    }

    T & VratiElNaPoz(int red, int stupac)
    {
        ProvjeriPoz(red,stupac);

        return BrzoVratiElNaPoz(red,stupac);
    }

    int VratiBrojElem()
    {
        int i, suma;

        for(i=suma=0 ; i<VratiBrojRedova() ; i++)
            suma += VratiVelReda(i);

        return suma;
    }

protected:
    virtual T & BrzoVratiElNaPoz(int red, int stupac)=0;
};

//////////////////////////////////////////////////////////////////
```

```
#endif
```

9.1.2. CCharDatoteka.h

```
//
//  CCharDatoteka
//

#if !defined(CCHARDATOTEKA_H)
#define CCHARDATOTEKA_H

#include "TDatoteka.h"

////////////////////////////////////////////////////////////////////

class CCharDatoteka: public TDatoteka< char >
{
public:

    CCharDatoteka(): DuljNajReda(0)
    {
        Dat.resize(1);
    }

    CCharDatoteka(const string &ime_dat): DuljNajReda(0)
    {
        Ucitaj(ime_dat);
    }

    void Ucitaj(const string &ime_dat)
    {
        ifstream in(ime_dat.c_str(), ios_base::binary);

        if(in.fail())
        {
            stringstream ss;
            ss << "Ne mogu otvoriti " << ime_dat << " za citanje";
            throw string(ss.str());
        }

        Ucitaj(in);

        in.close();
    }

    void Ucitaj(istream &in)
    {
        char buf[1024];
        int red;

        Dat.resize(0);
        Dat.resize(1);

        for(red=0;;red++)
        {
            in.getline(buf,1023);
            if(in.fail()) break;

            for(int stupac=0 ; buf[stupac] ; stupac++)
                Dat[red].push_back( buf[stupac] );

            if(stupac>DuljNajReda) DuljNajReda=stupac;

            if(!in.eof())
            {
                Dat[red].push_back( '\n' );
                Dat.push_back( my_vect<char>() );
```

```
            }
        }
    }

    int VratiBrojRedova()
    {
        return Dat.size();
    }

    int VratiVelReda(int koji)
    {
        return Dat[koji].size();
    }

    int VratiVelRedaBezLFiCR(int koji)
    {
        int vel;

        for( vel=VratiVelReda(koji) ; vel>0 ; vel-- )
        {
            char ch = VratiElNaPoz(koji,vel-1);

            if( ch!='\r' && ch!='\n' ) break;
        }

        return vel;
    }

    virtual pair<int,int> Umetni(int red, int stupac, char ch)
    {
        ProvjeriRed(red);

        if( stupac<0 || stupac>VratiVelReda(red) )
            throw string("Neispravan stupac u Umetni()");

        Dat[red].insert( Dat[red].begin()+stupac, ch );

        if( Dat[red].size()>DuljNajReda )
            DuljNajReda = Dat[red].size();

        return pair<int,int>(red,red);
    }

    virtual pair<int,int> Obrisi(int red, int stupac)
    {
        ProvjeriPoz(red,stupac);

        Dat[red].erase( Dat[red].begin()+stupac );

        return pair<int,int>(red,red);
    }

    virtual int SpojiSaSljedecimRedom(int koji)
    {
        int KolikoZnObrisano=0;

        ProvjeriRed(koji);

        while( Dat[koji].size()>0 &&
                 (Dat[koji].back()=='\r' || Dat[koji].back()=='\n') )
        {
            Dat[koji].pop_back();
            KolikoZnObrisano++;
        }

        ProvjeriRed(koji+1);

        Dat[koji].insert(
            Dat[koji].end(),
            Dat[koji+1].begin(),
            Dat[koji+1].end() );
```

```
        Dat.erase( Dat.begin()+koji+1 );

        if( Dat[koji].size()>DuljNajReda )
            DuljNajReda = Dat[koji].size();

        return KolikoZnObrisano;
    }

    virtual int PrelomiRed(int red, int stupac)
    {
        // provjeri red
        ProvjeriRed(red);

        // ovo je malo razlicito od ProvjeriPoz()
        if( stupac<0 || stupac>VratiVelReda(red) )
            throw string("Neispravan stupac u PrelomiRed()");

        // ubaci novi red
        my_vect< my_vect< char > >::iterator it =
            Dat.insert( Dat.begin()+red+1 );

        my_vect<char>::iterator poc = Dat[red].begin()+stupac;
        my_vect<char>::iterator kraj = Dat[red].end();

        // stavi u taj red zadnji dio trenutnog
        it->insert( it->begin(), poc, kraj );

        // u trenutnom obrisi zadnji dio

        Dat[red].erase( poc, kraj );

        // dodaj CR i LF u stari red
        Dat[red].push_back('\r');
        Dat[red].push_back('\n');

        // dodali smo dva znaka
        return 2;
    }

    int VratiDuljinuNajduzegReda()
    {
        return DuljNajReda;
    }

protected:

    my_vect< my_vect< char > > Dat;

    int DuljNajReda;

    char & BrzoVratiElNaPoz(int red, int stupac)
    {
        return Dat[red][stupac];
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.1.3. CCharLexDat.h

```
//
//  CCharLexDat
//

#if !defined(CCHARLEXDAT_H)
```

```
#define CCHARLEXDAT_H

#include "LexSucelje.h"

//////////////////////////////////////////////////////////////////

class CCharLexDat: public CCharDatoteka
{
public:
     CCharLexDat(): bNapravioLexAn(true)
     {
          pLex = new CLexAnalizator( *this );
     }

     CCharLexDat(const string &ime_dat):
          CCharDatoteka(ime_dat), bNapravioLexAn(true)
     {
          pLex = new CLexAnalizator( *this );
     }

     CCharLexDat( CLexAnalizator *plex ): bNapravioLexAn(false), pLex(plex)
     {
     }

     CCharLexDat(const string &ime_dat, CLexAnalizator *plex):
          CCharDatoteka(ime_dat), bNapravioLexAn(false), pLex(plex)
     {
     }

     ~CCharLexDat()
     {
          if( bNapravioLexAn )
               delete pLex;
     }

     pair<int,int> Umetni(int red, int stupac, char ch)
     {
          CCharDatoteka::Umetni(red,stupac,ch);

          int tm1=1,tm2=0;

          return pLex->ObradiUmetniObrisi(red,stupac,tm1,tm2);
     }

     pair<int,int> Obrisi(int red, int stupac)
     {
          CCharDatoteka::Obrisi(red,stupac);

          int tm1=0,tm2=1;

          return pLex->ObradiUmetniObrisi(red,stupac,tm1,tm2);
     }

     int SpojiSaSljedecimRedom(int koji)
     {
          int stup = Dat[koji].size()-1;

          int BrObrisZn =   CCharDatoteka::SpojiSaSljedecimRedom(koji);
          int tm=0;

          pLex->ObradiUmetniObrisi(koji,stup,tm,BrObrisZn);

          return BrObrisZn;
     }

     int PrelomiRed(int red, int stupac)
     {
          int BrUmetnZn = CCharDatoteka::PrelomiRed(red,stupac), tm=0;

          pLex->ObradiUmetniObrisi(red,stupac,BrUmetnZn,tm);

          return BrUmetnZn;
```

```
    }

    CLexAnalizator *pLex;

protected:
    bool bNapravioLexAn;
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.1.4. CLexDatoteka.h

```
//
//  CLexDatoteka
//

#if !defined(CLEXDATOTEKA_H)
#define CLEXDATOTEKA_H

#include "LexSucelje.h"
#include "CTransliterator.h"
#include "CLexDKA.h"

//////////////////////////////////////////////////////////////////////

class CToken: public COpcenitiToken
{
public:

    CToken(TIPTOKENA tip, int brzn, char *tokstr): Tip(tip), BrZn(brzn)
    {
        PostaviTokStr(tokstr);
    }

    int VratiTip()
    {
        return Tip;
    }

    int VratiBrZn()
    {
        return BrZn;
    }

    void PostaviBrZn( int br )
    {

        BrZn = br;
    }

    const char* VratiTokStr()
    {
        return TokStr.c_str();
    }

    void PostaviTokStr(const char *tokstr)
    {
        TokStr = tokstr;
    }

protected:
    TIPTOKENA Tip;
    int BrZn;
    string TokStr;
};
```

```
////////////////////////////////////////////////////////////////////

class CTokenDatoteka: public TDatoteka< CToken >
{
     friend class CLexDatoteka;

public:
     CTokenDatoteka()
     {
          // jedan red
          Dat.resize(1);
     }

//  todo: ovo nece trebati u jednom trenutku

     void DebugUDatoteku()
     {
          static int brbris=0;
          char buffer[20];
          string fname("TEMP/lex_brisi");

          if(brbris==0)
               system("del /Q temp\\*.*");

          fname += itoa( brbris, buffer, 10);
          fname += ".txt";

          ofstream ofs( fname.c_str() );

          IspisiDbg(ofs);

          ofs.close();

          brbris++;
     }

     void IspisiDbg( ostream & out )
     {
          my_vect< my_vect< CToken > >::iterator datit;

          for(datit=Dat.begin();datit!=Dat.end();datit++)
          {
               my_vect< CToken >::iterator redit;

               for(redit=datit->begin();redit!=datit->end();redit++)
                    out << ImeTokena[redit->VratiTip()] << ":"
                         /*<< redit->VratiBrZn() << " \""
                         << redit->VratiTokStr() << "\" "*/;

               out << endl;
          }
     }

     int VratiBrojRedova()
     {
          return Dat.size();
     }

     int VratiVelReda(int koji)
     {
          return Dat[koji].size();
     }

protected:
     my_vect< my_vect< CToken > > Dat;

     CToken & BrzoVratiElNaPoz(int red, int stupac)
     {
          return Dat[red][stupac];
     }
};
```

```
//////////////////////////////////////////////////////////////////////

class CLexDatoteka: public CTokenDatoteka, public CLexDodatniPodaci
{
    friend class CLexAnalizator;

public:
    CLexDatoteka(CCharDatoteka & refchdat):
        RefChDat(refchdat)
    {
        // alociraj DKA ako treba
        if( BrReferenciNaLex==0 )
        {
            if( pLexDKA!=NULL )
                throw string("Nesto ne valja u CLexDatoteka(...)");

            pLexDKA = new CLexDKA( "Podaci/JednostavanJezik.dka",
                                   new CTransliterator("Podaci/JednostavanJezik.tr")
);
        }

        // jedna vise referenca na DKA
        BrReferenciNaLex++;

        // u kojemu se cijela nalazi neanalizirana datoteka
        Dat[0].push_back( CToken(NEPOZNATO,RefChDat.VratiBrojElem(),"") );
        Dat[0].push_back( CToken(ZNKRNIZA,0,"") );

        // analiziraj cijelu datoteku
        int tm1=0,tm2=0;

        Analiziraj( Iterator(0,0,this), tm1, tm2 );
    }

    ~CLexDatoteka()
    {
        BrReferenciNaLex--;

        if( BrReferenciNaLex==0 )
        {
            delete pLexDKA->trans;
            delete pLexDKA;
            pLexDKA = NULL;
        }
    }

    class CChTokenIterator
    {
    public:
        CChTokenIterator(Iterator tokit, int poz): TokIt(tokit), PozUTok(poz)
        {
            UskladiChIterator();
        }

        CChTokenIterator(int red, int stup, CLexDatoteka *plexd):
            ChIt(red,stup,&(plexd->RefChDat)), TokIt(red,0,plexd)
        {
            UskladiTokIterator();
        }

        Iterator TokIt;
        int PozUTok;
        CCharDatoteka::Iterator ChIt;

        void UskladiChIterator()
        {
            int BrIspred;

            Iterator TmTokIt( TokIt.VratiRed(), 0, TokIt.VratiDat() );

            for( BrIspred=0; TmTokIt!=TokIt; ++TmTokIt )
```

```
                BrIspred+=(*TmTokIt).VratiBrZn();

            CLexDatoteka * plexdat =
                dynamic_cast<CLexDatoteka *>( TokIt.VratiDat() );

            ChIt.Postavi( TokIt.VratiRed(), BrIspred, &(plexdat->RefChDat) );
        }

        void UskladiTokIterator()
        {
            PozUTok = ChIt.VratiStup();

            while( PozUTok!=0 && PozUTok >= (*TokIt).VratiBrZn() )
            {
                PozUTok -= (*TokIt).VratiBrZn();
                ++TokIt;
            }
        }

        bool operator!=(CChTokenIterator & right)
        {
            if( ChIt!=right.ChIt ||
                TokIt!=right.TokIt ||
                PozUTok!=right.PozUTok)
                return true;

            return false;
        }

        CChTokenIterator operator++()
        {
            ++PozUTok;
            if( PozUTok==(*TokIt).VratiBrZn() )
            {
                PremotajTokNaKraj(TokIt);
                PozUTok=0;
            }

            ++ChIt;

            return *this;
        }


        CChTokenIterator operator--()
        {
            --PozUTok;
            if( PozUTok==-1 )
            {
                PremotajTokNaPocetak(--TokIt);

                PozUTok += (*TokIt).VratiBrZn();
            }

            --ChIt;

            return *this;
        }

        bool Eof()
        {
            return ChIt.Eof();
        }

    };

    pair<int,int> ObradiObrisi(int &red, int &stupac, int &KolikoUmetnutihZn, int
&KolikoObrisanihZn)
    {
        CChTokenIterator it(red,stupac,this);

        Iterator tit(it.TokIt);
```

```
        // ako je Eof onda je brisanje na kraju reda
        if( it.ChIt.Eof() )
        {
            tit.Postavi( red, Dat[red].size()-1 );

            if( (*tit).VratiTip()==ZNKRNIZA ) --tit;
        }

        PremotajTokNaPocetak(tit);

        (*tit).PostaviBrZn( (*tit).VratiBrZn() - KolikoObrisanihZn );

        if( (*tit).VratiBrZn()<0 )
            throw string("Obrisano vise od jednog tokena u ObradiObrisi()");

        KolikoObrisanihZn = 0;

        NapraviNepoznatiInterval(tit,KolikoUmetnutihZn,KolikoObrisanihZn);

        red = tit.VratiRed();
        stupac = tit.VratiStup();

        return Analiziraj(tit,KolikoUmetnutihZn,KolikoObrisanihZn);
    }

    pair<int,int> ObradiUmetni(int &red, int &stupac, int &KolikoUmetnutihZn, int
&KolikoObrisanihZn)
    {
        CChTokenIterator it(red,stupac,this);

        Iterator tit(it.TokIt);

        // ako je na pocetku tokena
        if( it.PozUTok==0 && (*tit).VratiTip()!=NASTAVAK)
        {
            // onda dodaj novi token velicine KolikoUmetnutihZn
            Dat[tit.VratiRed()].insert(
                Dat[tit.VratiRed()].begin() + tit.VratiStup(),
                CToken(NEPOZNATO,KolikoUmetnutihZn,"") );
        }
        else
        {
            // inace shvati to kao promjenu unutar tokena
            PremotajTokNaPocetak(tit);
            (*tit).PostaviBrZn( (*tit).VratiBrZn() + KolikoUmetnutihZn );
        }

        KolikoUmetnutihZn=0;

        NapraviNepoznatiInterval(tit,KolikoUmetnutihZn,KolikoObrisanihZn);

        red = tit.VratiRed();
        stupac = tit.VratiStup();

        return Analiziraj(tit,KolikoUmetnutihZn,KolikoObrisanihZn);
    }

    static void PremotajTokNaPocetak( Iterator & it )
    {
        while( !(it.VratiRed()==0&&it.VratiStup()==0) &&
                (*it).VratiTip()==NASTAVAK )
            --it;
    }

    static void PremotajTokNaKraj( Iterator & it )
    {
        do ++it; while( !it.Eof() && (*it).VratiTip()==NASTAVAK );
    }

    CCharDatoteka & RefChDat;
```

```
    int BrNovihTokena;

protected:
    static CLexDKA *pLexDKA;
    static int BrReferenciNaLex;

    pair<int,int> Analiziraj( Iterator itNeispravanToken, int &BrUmet, int &BrObris )
    {
        int izraz,ulaz,stanje,PocRed;

        CChTokenIterator pocetak(itNeispravanToken,0);
        CChTokenIterator zavrsetak(pocetak), posljednji(pocetak);

        PocRed=pocetak.ChIt.VratiRed();

        // ovo sluzi za pohranu napravljenih tokena
        CTokenDatoteka ParsTok;

        BrNovihTokena = 0;

        // koristimo algoritam iz skripte
        for(;!pocetak.Eof();)
        {
            // ovo sam ja dodao radi inkrementalnog algoritma
            //  (ako je prodjeno nepoznato podrucije i na pocetku smo
            //   novog tokena, onda slobodno mozemo prekinuti)
            if( (*(pocetak.TokIt)).VratiTip()!=NEPOZNATO && pocetak.PozUTok==0)
                break;

            stanje = izraz = 0;

            for(;;)
                switch( pLexDKA->tabela[stanje][pLexDKA->BrUlaza] )
                {
                default: // postoje izrazi koji zavrsavaju ovdje
                    izraz = pLexDKA->tabela[stanje][pLexDKA->BrUlaza];
                    posljednji = zavrsetak;
                    // nema break pa propada prema dolje!

                case 0:      // ne postoje izrazi koji zavrsavaju ovdje
                    if(zavrsetak.Eof())
                        goto van;

                    ulaz = *(zavrsetak.ChIt);
                    ++zavrsetak;
                    ulaz = pLexDKA->trans->TranslateCh(ulaz);
                    stanje = pLexDKA->tabela[stanje][ulaz];
                    break;

                case -1: // greska
                    goto van;
                }
    van: ;

            if(izraz==0)
            {
                CChTokenIterator tmit(pocetak);

                if( pocetak!=zavrsetak && ++tmit != zavrsetak
                    // ovo sam naknadno dodao
                    && (!zavrsetak.Eof()) )
                    --zavrsetak;

                DodajTokenU( ParsTok, GRESKA, pocetak, zavrsetak );

                pocetak = posljednji = zavrsetak;
            }
            else
            {
                DodajTokenU( ParsTok, izraz-1, pocetak, posljednji );

                pocetak = zavrsetak = posljednji;
```

```
            }
        }

        // sada treba [itNeispravanToken, posljednji) zamjeniti sa ParsTok

        // ovo treba zbog brisanja cijele datoteke
        if( (*(posljednji.TokIt)).VratiTip()==NEPOZNATO )
        {
            Iterator tmit(posljednji.TokIt);

            BrObris += ObrisiInterval( itNeispravanToken, ++tmit );
        }
        else
            BrObris += ObrisiInterval( itNeispravanToken, posljednji.TokIt );

        BrUmet += ParsTok.VratiBrojElem();

        UmetniInterval( itNeispravanToken, ParsTok );

        if( posljednji.ChIt.Eof() )
        {
            CCharDatoteka * pchdat =
                dynamic_cast<CCharDatoteka *>( posljednji.ChIt.VratiDat() );

            return pair<int,int>( PocRed, pchdat->VratiBrojRedova()-1 );
        }
        else
            return pair<int,int>( PocRed, posljednji.ChIt.VratiRed() );
    }

    void DodajTokenU( CTokenDatoteka & sto, int koji,
            CChTokenIterator poc, CChTokenIterator kraj)
    {
        int kr_red, kr_stup;

        BrNovihTokena++;

        if( kraj.Eof() )
        {
            CCharDatoteka * pchdat =
                dynamic_cast<CCharDatoteka *>( kraj.ChIt.VratiDat() );

            kr_red = _cpp_max( pchdat->VratiBrojRedova()-1, 0 );
            kr_stup = pchdat->VratiVelReda(kr_red);
        }
        else
        {
            kr_red=kraj.ChIt.VratiRed();
            kr_stup=kraj.ChIt.VratiStup();
        }

        // na kraj zadnjeg reda dodaj token tog tipa
        sto.Dat.back().push_back( CToken( (TIPTOKENA)koji, 0, "" ) );

        // pamti koji je to token
        CTokenDatoteka::Iterator
            tit( sto.Dat.size()-1, sto.Dat.back().size()-1, &sto );

        // postoje dva slucaja
        if( poc.ChIt.VratiRed()==kr_red )
        {
            (*tit).PostaviBrZn( kr_stup - poc.ChIt.VratiStup() );
        }
        else
        {
            int velicina = RefChDat.VratiVelReda( poc.ChIt.VratiRed() ) -
                    poc.ChIt.VratiStup();

            my_vect< CToken > TmRed;

            TmRed.push_back( CToken(NASTAVAK,0,"") );
```

```
            for( int i=poc.ChIt.VratiRed()+1; i<kr_red; i++)
            {
                // dodaj red i inicijaliziraj token u tom redu
                sto.Dat.push_back( TmRed );
                velicina += RefChDat.VratiVelReda(i);
                sto.Dat.back()[0].PostaviBrZn( RefChDat.VratiVelReda(i) );
            }

            if( kr_stup!=0 )
            {
                sto.Dat.push_back( TmRed );
                velicina += kr_stup;
                sto.Dat.back()[0].PostaviBrZn( kr_stup );
            }
            else
                sto.Dat.push_back( my_vect< CToken >() );

            (*tit).PostaviBrZn( velicina );

        }

        // todo: ovdje se postavlja string
        //  to usporava leksicku

        string tms;
        int i;

        for(i=0;i<(*tit).VratiBrZn();i++)
        {
            tms += (*(poc.ChIt));
            ++(poc.ChIt);
        }

        (*tit).PostaviTokStr( tms.c_str() );
    }

    int ObrisiInterval( Iterator poc, Iterator kraj )
    {
        int BrObrisanih=0, BrNepoznatih=0;

        my_vect< my_vect< CToken > >::iterator pocit, krit, tmit;

        pocit = Dat.begin() + poc.VratiRed();

        krit = Dat.begin() + kraj.VratiRed();

        if( poc!=kraj )
        {
            // nepoznati token je ili prvi u intervalu..
            if( (*poc).VratiTip()==NEPOZNATO )
                BrNepoznatih=1;
            else
            {
                // ..ili je zadnji u intervalu
                Iterator tmit(kraj);
                --tmit;
                if( tmit!=poc && (*tmit).VratiTip()==NEPOZNATO )
                    BrNepoznatih=1;
            }
        }

        // postoje dva slucaja
        if( pocit==krit )
        {
            BrObrisanih += kraj.VratiStup() - poc.VratiStup();

            // obrisi potrebno u tom redu
            pocit->erase(
                pocit->begin()+poc.VratiStup(),
                pocit->begin()+kraj.VratiStup() );
        }
        else
```

```
        {
            // obrisi do kraja prvog reda

            BrObrisanih += pocit->end() - (pocit->begin()+poc.VratiStup()) ;

            pocit->erase(
                pocit->begin()+poc.VratiStup(),
                pocit->end() );

            // spoji redove

            BrObrisanih -= krit->end() - (krit->begin()+kraj.VratiStup()) ;

            pocit->insert(
                pocit->end(),
                krit->begin()+kraj.VratiStup(),
                krit->end() );

            // obrisi sve redove do krajnjeg reda ukljucivo

            for( tmit=pocit+1; tmit!=krit+1; tmit++ )
                BrObrisanih += tmit->size();

            Dat.erase( pocit+1, krit+1 );
        }

        if( BrObrisanih - BrNepoznatih >= 0 )
            BrObrisanih -= BrNepoznatih;

        return BrObrisanih;
    }

    void PrelomiRed(int red, int stupac)
    {
        // provjeri red
        ProvjeriRed(red);

        // ovo je malo razlicito od ProvjeriPoz()
        if( stupac<0 || stupac>VratiVelReda(red) )
            throw string("Neispravan stupac u PrelomiRed()");

        // ubaci novi red
        my_vect< my_vect< CToken > >::iterator it =
            Dat.insert( Dat.begin()+red+1 );

        my_vect<CToken>::iterator poc = Dat[red].begin()+stupac;
        my_vect<CToken>::iterator kraj = Dat[red].end();

        // stavi u taj red zadnji dio trenutnog
        it->insert( it->begin(), poc, kraj );

        // u trenutnom obrisi zadnji dio
        Dat[red].erase( poc, kraj );
    }

    void UmetniInterval( Iterator gdje, CTokenDatoteka & sto )
    {
        my_vect< my_vect< CToken > >::iterator it, tdit;

        if( sto.Dat.size()>1 )
            PrelomiRed( gdje.VratiRed(), gdje.VratiStup() );

        it = Dat.begin() + gdje.VratiRed();

        tdit = sto.Dat.begin();

        // umetni na kraj prvog reda
        it->insert(
            it->begin()+gdje.VratiStup(),
            tdit->begin(),
            tdit->end() );
```

```
        // umetni ostale redove
        tdit++;

        if( tdit < sto.Dat.end()-1 )
            Dat.insert( it+1, tdit, sto.Dat.end()-1 );

        // umetni na pocetak zadnjeg reda
        if( sto.Dat.size()>1 )
        {
            tdit = sto.Dat.end()-1;

            it = Dat.begin() + gdje.VratiRed() + sto.Dat.size()-1;

            it->insert( it->begin(),
                tdit->begin(),
                tdit->end() );
        }
    }

    void NapraviNepoznatiInterval( Iterator & it, int &BrUmet, int &BrObris  )
    {
        int vel;

        Iterator poc(it), kraj(it);

        // prvo nadjemo kraj (tj. onaj iza kraja) trenutnog tokena
        PremotajTokNaKraj(kraj);

        // velicina trenutnog tokena
        vel = (*poc).VratiBrZn();

        // prebacimo se na prethodni token
        if( !(poc.VratiRed()==0&&poc.VratiStup()==0) )
        {
            --poc;

            // on isto moze biti u vise redova
            PremotajTokNaPocetak(poc);

            // dodajemo velicinu prethodnog tokena
            vel += (*poc).VratiBrZn();
        }

        // vrsimo zamjenu oba tokena (ili mozda jednog) sa nepoznatim tokenom

        // ovo treba kod brisanja zadnjeg znaka u datoteci
        if( !kraj.Eof() )
            BrObris += ObrisiInterval( poc, kraj );

        // umecemo nepoznati token samo ako je velicine vece od jedan
        if( vel>0 )
        {
            CTokenDatoteka TmDat;

            TmDat.Dat[0].push_back( CToken(NEPOZNATO,vel,"") );

            UmetniInterval( poc, TmDat );
        }

        // moramo promjeniti i iterator
        it = poc;
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.1.5. CLexDatoteka.cpp

```
//
//  CLexDatoteka
//

#include "StdAfx.h"

#include "CLexDatoteka.h"

//////////////////////////////////////////////////////////////////////

CLexDKA *CLexDatoteka::pLexDKA=NULL;
int CLexDatoteka::BrReferenciNaLex=0;

//////////////////////////////////////////////////////////////////////

CLexAnalizator::CLexAnalizator(): pLexDodPod(NULL)
{
}

CLexAnalizator::CLexAnalizator(CCharDatoteka & refchdat)
{
    Inicijalizacija(refchdat);
}

void CLexAnalizator::Inicijalizacija(CCharDatoteka & refchdat)
{
    pLexDodPod = new CLexDatoteka(refchdat);
}

CLexAnalizator::~CLexAnalizator()
{
    // castanje mora biti jer destruktor nije virtual!
    delete ((CLexDatoteka *)pLexDodPod);
}

int CLexAnalizator::VratiBrRazlicitihTok()
{
    return NASTAVAK;
}

int CLexAnalizator::VratiBrNovihTokena()
{
    return ((CLexDatoteka *)pLexDodPod)->BrNovihTokena;
}

pair<int,int> CLexAnalizator::ObradiUmetniObrisi(int &red, int &stupac, int
&KolikoUmetnutihZn, int &KolikoObrisanihZn)
{
    if(KolikoUmetnutihZn==0)
        return ((CLexDatoteka *)pLexDodPod)->ObradiObrisi( red, stupac,
KolikoUmetnutihZn, KolikoObrisanihZn );
    else if(KolikoObrisanihZn==0)
return ((CLexDatoteka *)pLexDodPod)->ObradiUmetni( red, stupac, KolikoUmetnutihZn,
 KolikoObrisanihZn );


else
        throw string("ObradiUmetniObrisi pozvano sa umetanjem i brisanjem");
}

void CLexAnalizator::NapuniInterval(int red, int OdStup, int DoStup, NIZCHTIP & niz)
{
    CLexDatoteka::CChTokenIterator chtit(red,OdStup,(CLexDatoteka *)pLexDodPod);

    CLexDatoteka::Iterator tmit( chtit.TokIt );

    CLexDatoteka::PremotajTokNaPocetak( tmit );

    for(int tip=(*tmit).VratiTip() ; OdStup<DoStup ; OdStup++ )
    {
        niz.push_back( pair<char,int>( *(chtit.ChIt), tip ) );
```

```
        ++chtit;

        if(chtit.PozUTok==0)
            tip=(*(chtit.TokIt)).VratiTip();
    }
}

int CLexAnalizator::VratiBrojRedova()
{
    return ((CLexDatoteka *)pLexDodPod)->VratiBrojRedova();
}

int CLexAnalizator::VratiVelReda(int koji)
{
    return ((CLexDatoteka *)pLexDodPod)->VratiVelReda(koji);
}

COpcenitiToken & CLexAnalizator::BrzoVratiElNaPoz(int red, int stupac)
{
    return ((CLexDatoteka *)pLexDodPod)->BrzoVratiElNaPoz(red,stupac);
}

//////////////////////////////////////////////////////////////////////
```

9.1.6. CLexDKA.h

```
//
//  CLexDKA.h
//

#if !defined(CLEXDKA_H)
#define CLEXDKA_H

#include "CTransliterator.h"

////////////////////////////// CLexDKA ///////////////////////////////

class CLexDKA     // klasa koja sadrzava funkcionalnost
{                     //  leksickog analizatora
public:
    // inicijalizacija zahtjeva ime datoteke sa definicijom automata,
    //  pokazivac na konkretni transliterator
    CLexDKA(const char* DefFileName, CTransliterator* transliterator);
    // i naravno destruktor
    ~CLexDKA();

    CTransliterator* trans;    // pointer na transliterator koji automat koristi

    int** tabela;         // DKA tabela -> svaka celija tablice je indeks u nizu
prijelaza
    int BrStanja, BrUlaza;// tj. broj redova i broj stupaca

private:
    void alociraj_tabelu(int ***tab, int koliko);    // ovo interno koristimo
    void oslobodi_tabelu(int ***tab);                //  za rukovanje tabelama
};

////////////////////////////// kraj //////////////////////////////////

#endif
```

9.1.7. CLexDKA.cpp

```
#include "StdAfx.h"

#include "CLexDKA.h"

////////////////////////////// CLexDKA ///////////////////////////////

/*
    format datoteke sa definicijom automata:
        N -> broj razlicitih stanja
        ime_grupe1 ime_grupe2 ...
        indeks_stanja X sljed_stanje_za_grupu1 ...
        .
        .
    NAPOMENA:    X=0      - neprihvatljivo
                 X>1      - prihvatljivo (red. br. reg. iz.)
                 X=-1 - stanje greske (prazno stanje)
*/

CLexDKA::CLexDKA(const char* DefFileName, CTransliterator* transliterator)
{
    int i,j,stanje,BrIzraza;
    char tms[17];
    int* ulaz;

    trans=transliterator;

    // broj ulaznih znakova ne pise u datoteci nego se dobiva od transliteratora
    BrUlaza=trans->VratiBrojRazlicitih();
    // stupci u ulaznoj datoteci ne moraju biti navedeni u istom
    //  poretku kao i u datoteci za transliteraciju - za pamcenje
    //   poretka sluzi ulaz[]
    ulaz=new int[BrUlaza];

    ifstream af(DefFileName);

    if(af.fail())
    {
        stringstream ss;
        ss << "Ne mogu otvoriti " << DefFileName;
        throw string(ss.str());
    }

    af >> BrStanja;

    // nakon sto znamo broj stanja i broj ulaza mozemo alocirati tabelu
    alociraj_tabelu(&tabela,BrStanja);

    // ucitavamo imena transliteriranih znakova i pamtimo na
    //  koji stupac se odnose
    for(i=0;i<BrUlaza;i++)
    {
        af >> tms;

        ulaz[i]=trans->TranslateName(tms);

        if(ulaz[i]==-1)
        {
            stringstream ss;
            ss << "greska u datoteci -> nema imena: " << tms;
            throw string(ss.str());
        }
    }

    // ovdje ucitavamo red po red datoteke i punimo tabelu
    for(i=0;i<BrStanja;i++)
    {
        af >> stanje >> BrIzraza;
```

```
            tabela[stanje][BrUlaza]=BrIzraza;

            for(j=0;j<BrUlaza;j++)
                af >> tabela[stanje][ulaz[j]];
        }

        // cistimo iza sebe
        delete ulaz;

        af.close();
}

// ovom funkcijom alociramo dvodimenzionalnu tabelu prijelaza
void CLexDKA::alociraj_tabelu(int ***tab, int koliko)
{
        int i;

        *tab=new int* [koliko];

        for(i=0;i<koliko;i++)
            (*tab)[i]=new int[BrUlaza+1];
}

// destruktor
CLexDKA::~CLexDKA()
{
        oslobodi_tabelu(&tabela);
}

// dealokacija dvodimenzionalne tabele
void CLexDKA::oslobodi_tabelu(int ***tab)   // brisemo dinamicki alociranu memoriju
{
        int i;

        for(i=0;i<BrStanja;i++) delete (*tab)[i];

        delete *tab;
}

////////////////////////////// kraj //////////////////////////////
```

9.1.8. CTransliterator.h

```
//
//  CTransliterator.h
//

#if !defined(CTRANSLITERATOR_H)
#define CTRANSLITERATOR_H

////////////////////////// Transliterator //////////////////////////

class CTransliterator // klasa koja je zaduzena za transliteraciju
{
public:
        CTransliterator(const char* TransFileName); // transliteracija je definirana u
datoteci
        ~CTransliterator();

        int VratiBrojRazlicitih() { return BrRazlicitih; }
        int TranslateCh(unsigned char ch) { return CharTable[ch]; }   // transliterira
znak
        // grupe ulaznih znakova nazivamo imenima
        int TranslateName(const char* name);    // navedena funkcija konvertira imena u
indekse
```

```
    char* GetName(const int index);             // i obratno

protected:
    int BrRazlicitih;     // broj razlicitih grupa
    char** NameTable;     // imena razlicitih grupa
    int CharTable[256];       // tabela transliteracije
};

////////////////////////////// kraj //////////////////////////////

#endif
```

9.1.9. CTransliterator.cpp

```
#include "StdAfx.h"

#include "CTransliterator.h"

////////////////////////// Transliterator //////////////////////////

/*
    format datoteke za transliteraciju:
        N -> broj razlicitih grupa
        od_zn do_zn ime_grupe ->   ako hocemo samo jedan znak u grupi
                                        pisemo npr. ' ' ' ' SPACE
                                        ili 32 32 SPACE
                                        ili 32 ' ' SPACE
                                        ili ' ' 32 SPACE
        .
        .
*/

CTransliterator::CTransliterator(const char* TransFileName)
{
    char a[2][17],name[17];
    int i,j,k,zn[2];

    // otvaramo ulaznu datoteku
    ifstream tf(TransFileName);

    if(tf.fail())
    {
        stringstream ss;
        ss << "Ne mogu otvoriti " << TransFileName;
        throw  string(ss.str());
    }

    // citamo broj simbola (vise znakova transliteriramo u jedan simbol)
    tf >> j;

    // alociramo tabelu imena simbola
    NameTable=new char* [j];

    for(i=0;i<256;i++) CharTable[i]=-1;     // -1 oznacava gresku -> znak koji nije
definiran

    // ucitavamo red po red dok ne naidemo na EOF
    for(BrRazlicitih=0;;)
    {
        tf >> a[0] >> a[1] >> name;

        if(tf.fail()) break;

        // ako taj simbol ne postoji moramo ga dodati u tabelu simbola
        if((k=TranslateName(name))==-1)
        {
            k=BrRazlicitih;
```

```
            NameTable[k]=new char [strlen(name)+1];

            strcpy(NameTable[k], name);

            BrRazlicitih++;
        }

        // svaki od dva ulazna znaka moramo prevesti
        for(i=0;i<2;i++)
        {
            zn[i]=atoi(a[i]);

            // znakovi mogu biti ASCII kodovi
            if(zn[i]>0)
                if(zn[i]<=255)
                    continue;
                else {
                    throw string("CTransliterator::nepravilna \
                        datoteka: preveliki indeks");
                }

            j=strlen(a[i]);

            if(j==1&&a[i][0]=='0') continue;

            // ili mogu biti zadani u obliku 'c' (c je char)
            if(j!=3||a[i][0]!='\''||a[i][2]!='\'') {
                throw string("CTransliterator::nepravilna \
                        datoteka: neispravno zadani znak");
            }

            zn[i]=a[i][1];
        }

        // interval u tabeli izmedu ta dva znaka transliteriramo
        for(j=zn[0];j<=zn[1];j++) CharTable[j]=k;
    }

    tf.close();
}

// destruktor brise sve dinamicki alocirane podatke
CTransliterator::~CTransliterator()
{
    for(;BrRazlicitih>0;BrRazlicitih--)
        delete NameTable[BrRazlicitih-1];

    delete NameTable;
}

// konvertira imena grupa u indekse
int CTransliterator::TranslateName(const char* name)
{
    int i;

    for(i=0;i<BrRazlicitih;i++)
        if(strcmp(NameTable[i],name)==0) return i;

    return -1;
}

// i obratno
char* CTransliterator::GetName(const int index)
{
    if(index<0||index>=BrRazlicitih)
        throw string("CTransliterator::GetName");

    return NameTable[index];
}

////////////////////////////// kraj ////////////////////////////////
```

9.1.10. Tokeni.h

```
//
//  Tokeni
//

#if !defined(TOKENI_H)
#define TOKENI_H

////////////////////////////////////////////////////////////////////

enum TIPTOKENA { KROS=0, IDN, ASM, STR, KARAKTER, DEK,
                      REALNI, HEX, OKT, BIN, ODBACI, GRESKA,
                      NASTAVAK, ZNKRNIZA, NEPOZNATO };

extern char ImeTokena[15][10];

////////////////////////////////////////////////////////////////////

#endif
```

9.1.11. Tokeni.cpp

```
//
//  Tokeni
//

#include "StdAfx.h"

#include "Tokeni.h"

////////////////////////////////////////////////////////////////////

char ImeTokena[15][10] = { "KROS", "IDN", "ASM", "STR", "KARAKTER", "DEK",
                           "REALNI", "HEX", "OKT", "BIN", "ODBACI", "GRESKA",
                           "NASTAVAK", "ZNKRNIZA", "NEPOZNATO" };

////////////////////////////////////////////////////////////////////
```

9.1.12. LexSucelje.h

```
//
//  LexSucelje
//

#if !defined(LEXSUCELJE_H)
#define LEXSUCELJE_H

#include "TDatoteka.h"
#include "CCharDatoteka.h"
```

```
#include "Tokeni.h"

//////////////////////////////////////////////////////////////////////

class COpcenitiToken
{
public:
     virtual int VratiTip()=0;
     virtual const char* VratiTokStr()=0;
};

//////////////////////////////////////////////////////////////////////

class CLexDodatniPodaci { };

//////////////////////////////////////////////////////////////////////

class CLexAnalizator: public TDatoteka< COpcenitiToken >
{
public:
     CLexAnalizator();
     CLexAnalizator(CCharDatoteka & refchdat);
     void Inicijalizacija(CCharDatoteka & refchdat);
     virtual ~CLexAnalizator();

     int VratiBrojRedova();
     int VratiVelReda(int koji);

     int VratiBrRazlicitihTok();

     int VratiBrNovihTokena();

     virtual pair<int,int> ObradiUmetniObrisi(int &red, int &stupac, int
&KolikoUmetnutihZn, int &KolikoObrisanihZn);

     typedef my_vect< pair< char, int> > NIZCHTIP;

     void NapuniInterval(int red, int OdStup, int DoStup, NIZCHTIP & niz);

     CLexDodatniPodaci *pLexDodPod;

protected:
     COpcenitiToken & BrzoVratiElNaPoz(int red, int stupac);
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.2. Datoteke inkrementalnog sintaksnog analizatora

9.2.1. SynSucelje.h

```
//
//  SynSucelje
//

#if !defined(SYNSUCELJE_H)
#define SYNSUCELJE_H

#include "LexSucelje.h"
```

```
//////////////////////////////////////////////////////////////////////

class CSynDodatniPodaci { };

//////////////////////////////////////////////////////////////////////

class CSynAnalizator
{
public:
     CSynAnalizator(CLexAnalizator & reflexana);
     void Inicijalizacija(CLexAnalizator & reflexana);
     virtual ~CSynAnalizator();

     virtual void ObradiUmetniObrisi(int red, int stupac, int KolikoUmet, int
KolikoObris);

     virtual bool VratiUspjesanPopravak();

     struct SCh3Svojstva
     {
          SCh3Svojstva(char ch, int tip, bool ispravan, int generacija):
               Ch(ch), Tip(tip), bSinIspravan(ispravan), KurGeneracija(generacija) {}

          char Ch;
          int Tip;
          bool bSinIspravan;
          int KurGeneracija;
     };

     typedef my_vect< SCh3Svojstva > NIZCH3SVOJSTVA;

     void NapuniInterval(int red, int OdStup, int DoStup, NIZCH3SVOJSTVA & niz);

     void NovaKurGen(int red, int stup, int generacija);

protected:

     CSynDodatniPodaci *pSynDodPod;
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.2.2. SynSucelje.cpp

```
//
//  SynSucelje.cpp
//

#include "StdAfx.h"

#include "SynSucelje.h"

#include "IzDynEdit/ZParser.h"

#include "CLexDatoteka.h"

//////////////////////////////////////////////////////////////////////

CSynAnalizator::CSynAnalizator(CLexAnalizator & reflexana)
{
     Inicijalizacija(reflexana);
}

void CSynAnalizator::Inicijalizacija(CLexAnalizator & reflexana)
```

```
{
     pSynDodPod = new CDinamickaDatoteka(reflexana);
}

CSynAnalizator::~CSynAnalizator()
{
     // castanje mora biti jer destruktor nije virtual!
     delete ((CDinamickaDatoteka *)pSynDodPod);
}

void CSynAnalizator::ObradiUmetniObrisi(int red, int stupac, int KolikoUmet, int
KolikoObris)
{
     ((CDinamickaDatoteka *)pSynDodPod)-
>UmetniObrisi(red,stupac,KolikoUmet,KolikoObris);
}

bool CSynAnalizator::VratiUspjesanPopravak()
{
     return ((CDinamickaDatoteka *)pSynDodPod)->DinPars.UspjesnoParsiranje;
}
void CSynAnalizator::NapuniInterval(int red, int OdStup, int DoStup, NIZCH3SVOJSTVA &
niz)
{
     CDinamickaDatoteka *pDD = (CDinamickaDatoteka *)pSynDodPod;
     GR_IF( pDD==NULL );

     CLexDatoteka *pLD = (CLexDatoteka *)(pDD->RefLexAna.pLexDodPod);
     GR_IF( pLD==NULL );

     CLexDatoteka::CChTokenIterator chtit(red,OdStup,pLD);

     CLexDatoteka::Iterator tmit( chtit.TokIt );

     pDD->Premotaj( tmit.VratiRed(), tmit.VratiStup() );

     CLexDatoteka::PremotajTokNaPocetak( tmit );

     CZZnak *TmZZnak = pDD->ZnakNaPoz( tmit.VratiRed(), tmit.VratiStup() );

     for( ; OdStup<DoStup ; OdStup++ )
     {
          niz.push_back( SCh3Svojstva( *(chtit.ChIt), TmZZnak->Tip,
               TmZZnak->bSinkroniziran, TmZZnak->Generacija) );

          ++chtit;

          if(chtit.PozUTok==0)
          {
               pDD->PomakniNaSljedeci(false);
               TmZZnak = pDD->pTrZnak;
          }
     }
}

void CSynAnalizator::NovaKurGen(int red, int stup, int generacija)
{
     CDinamickaDatoteka *pDD = (CDinamickaDatoteka *)pSynDodPod;
     GR_IF( pDD==NULL );

     CLexDatoteka *pLD = (CLexDatoteka *)(pDD->RefLexAna.pLexDodPod);
     GR_IF( pLD==NULL );

     CLexDatoteka::CChTokenIterator chtit(red,stup,pLD);

     CLexDatoteka::Iterator tmit( chtit.TokIt );

     CLexDatoteka::PremotajTokNaPocetak( tmit );

     pDD->Premotaj( tmit.VratiRed(), tmit.VratiStup() );

     if( pDD->pTrZnak->bSinkroniziran )
```

```
        pDD->NovaGeneracija( pDD->pTrZnak->PozStablo, generacija );
}

//////////////////////////////////////////////////////////////////////
```

9.2.3. CLexSynDat.h

```
//
//  CLexSynDat
//

#if !defined(CLEXSYNDAT_H)
#define CLEXSYNDAT_H

#include "CCharLexDat.h"
#include "LexSucelje.h"
#include "SynSucelje.h"

//////////////////////////////////////////////////////////////////////

class CLexSynDat: public CLexAnalizator
{
public:
    CLexSynDat()
    {
        pChLexDat = new CCharLexDat( this );
        Inicijalizacija( *pChLexDat );

        pSyn = new CSynAnalizator( *this );
    }

    CLexSynDat(const string &ime_dat)
    {
        pChLexDat = new CCharLexDat( ime_dat, this );
        Inicijalizacija( *pChLexDat );

        pSyn = new CSynAnalizator( *this );
    }

    ~CLexSynDat()
    {
        delete pChLexDat;
        delete pSyn;
    }

    pair<int,int> ObradiUmetniObrisi(int &red, int &stupac, int &KolikoUmetnutihZn,
int &KolikoObrisanihZn)
    {
        pair<int,int> ret =

    CLexAnalizator::ObradiUmetniObrisi(red,stupac,KolikoUmetnutihZn,KolikoObrisanihZn)
;

        pSyn->ObradiUmetniObrisi(red,stupac,KolikoUmetnutihZn,KolikoObrisanihZn);

        return ret;
    }

    CCharLexDat *pChLexDat;
    CSynAnalizator *pSyn;
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.2.4. ZnReference.h

```
//
//       ZnReference.h
//

#if !defined(ZNREFERENCE_H)
#define ZNREFERENCE_H

#include "ZDKA.h"

////////////////////////////////////////////////////////////////////

class CReferencaNaZnak: public COsnovnaKlasa
{
public:
     CReferencaNaZnak(CZnakGramatike *pzg):
       pZG(pzg)/*, hPozUTree(NULL)*/, bNovo(true) { }
/* todo: TreeProzor
     HTREEITEM hPozUTree;
*/
     CZnakGramatike *pZG;

     bool bNovo;

     // mogli bi nekako i bez ovoga, ali moramo imati barem jednu
     //  virtualnu funkciju radi dynamic_cast<>()
     virtual CZStanje *VratiStanjePrije()=0;
};

////////////////////////////////////////////////////////////////////

class CReferencaNaZavrsni: public CReferencaNaZnak
{
public:
     CReferencaNaZavrsni(CZStanje *pStPr, CZStanje *pStPrRed, CZavrsniZnak *pzz, class
CZZnak *pzznak):
          pStanjePrije(pStPr), pZadStPrijeRedukcija(pStPrRed), pZZnak(pzznak),
               CReferencaNaZnak(pzz), BrIspod(1) { }

     class CZZnak *pZZnak;

     CZStanje *pZadStPrijeRedukcija;
     short int BrIspod;

     CZStanje *pStanjePrije;
     CZStanje *VratiStanjePrije() { return  pStanjePrije; }
};

////////////////////////////////////////////////////////////////////

class CAtributnaReferenca: public CReferencaNaZnak
{
public:
     CAtributnaReferenca(CZnakGramatike *pzg): CReferencaNaZnak(pzg) { }
     // todo: ovdje idu atributi
};

////////////////////////////////////////////////////////////////////

class CReferencaNaNezavrsni: public CAtributnaReferenca
{
public:
     CReferencaNaNezavrsni(CZStanje *pSt, CNezavrsniZnak *pnz,
          class CCvorSinStabla *pdijete, int brispod): pStanjePrije(pSt),
          pDijete(pdijete), BrIspod(brispod), CAtributnaReferenca(pnz) { }
```

```
     class CCvorSinStabla *pDijete;
     short int BrIspod;

     CZStanje *pStanjePrije;
     CZStanje *VratiStanjePrije() { return  pStanjePrije; }
};

//////////////////////////////////////////////////////////////////////

class CReferencaNaAkcijski: public CAtributnaReferenca
{
public:
     CReferencaNaAkcijski(CAkcijskiZnak *paz): CAtributnaReferenca(paz) { }


     CZStanje *VratiStanjePrije()
     {
          GR_INT_CLASS << "referenca na akcijski nema stanje prije" << kraj;
          return NULL;
     }
};

//////////////////////////////////////////////////////////////////////

class CPokNaZnakUCvoru: public COsnovnaKlasa
{
public:
     CPokNaZnakUCvoru(): pCvor(NULL), PozZnak(-1) { }
     CPokNaZnakUCvoru(class CCvorSinStabla *pcvor, int pozznak):
          pCvor(pcvor), PozZnak(pozznak) { }
     CPokNaZnakUCvoru(const CPokNaZnakUCvoru &pz):
          pCvor(pz.pCvor), PozZnak(pz.PozZnak) { }

     class CCvorSinStabla *pCvor;
     short int PozZnak;

     friend const bool operator==(const CPokNaZnakUCvoru & left,
          const CPokNaZnakUCvoru & right)
     {
          return ( left.pCvor == right.pCvor && left.PozZnak == right.PozZnak );
     }

     CReferencaNaZnak *RefZnak();
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.2.5. ZnReference.cpp

```
#include "stdafx.h"

#include "ZnReference.h"

#include "ZParser.h"

//////////////////////////////////////////////////////////////////////

CReferencaNaZnak *CPokNaZnakUCvoru::RefZnak()
{
     GR_IF( PozZnak>=pCvor->ZnProd.size() || PozZnak<0 );

     return pCvor->ZnProd[ PozZnak ];
}
```

```
////////////////////////////////////////////////////////////////////
```

9.2.6. ZParser.h

```
//
//       ZParser
//

#if !defined(ZPARSER_H)
#define ZPARSER_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ZDKA.h"

#include "../LexSucelje.h"

#include "ZnReference.h"

#include "../SynSucelje.h"

////////////////////////////////////////////////////////////////////

typedef my_vect<CReferencaNaZnak *> NIZPTRREFZN;

class CCvorSinStabla: public COsnovnaKlasa
{
public:
     CCvorSinStabla(): pProd(NULL) { }

     CPokNaZnakUCvoru Otac;
     CKonNeoProdukcija *pProd;

     NIZPTRREFZN ZnProd;
};

////////////////////////////////////////////////////////////////////

class CZStablo: public COsnovnaKlasa
{
public:
     CZStablo(): pRefNaOrgDna(NULL)
     {
          // ovo sluzi za jednostavniji ZamjeniSa()
          pGlava = new CCvorSinStabla();

          // ovo sluzi:
          //   - za poziciju stoga kod umetanja na kraj
          //   - za pamcenje zadnjeg stanja 'zavrsnog pomaka'
          //   - kao nesto na sto pokazuje kraj datoteke
          pRefKraj = new CReferencaNaZavrsni(NULL,NULL,NULL,NULL);

          pGlava->ZnProd.push_back( pRefKraj );
     }

     ~CZStablo() { ObrisiPodstablo(pGlava); }

     CCvorSinStabla *pGlava;
     CReferencaNaZavrsni *pRefKraj;
     CReferencaNaZnak *pRefNaOrgDna;

     void ObrisiPodstablo(CCvorSinStabla *pVrh)
     {
```

```
        if(pVrh==NULL) // zbog ZamjeniSa()
            return;

        NIZPTRREFZN::iterator it;

        for(it=pVrh->ZnProd.begin();it!=pVrh->ZnProd.end();it++)
        {
            CReferencaNaNezavrsni *pRefNZ =
                dynamic_cast<CReferencaNaNezavrsni *>(*it);

            if(pRefNZ!=NULL)
                ObrisiPodstablo(pRefNZ->pDijete);

            if( pRefNaOrgDna==NULL )
                delete *it;

            if( *it==pRefNaOrgDna )
                pRefNaOrgDna=NULL;
        }

        if( pRefNaOrgDna==NULL )
            delete pVrh;
    }

    void IspisiPodstablo(CCvorSinStabla *pVrh, ostream &out)
    {
        GR_IF(pVrh==NULL);

        NIZPTRREFZN::iterator it;

        for(it=pVrh->ZnProd.begin();it!=pVrh->ZnProd.end();it++)
        {
            CReferencaNaNezavrsni *pRefNZ =
                dynamic_cast<CReferencaNaNezavrsni *>(*it);

            if(pRefNZ!=NULL)
                IspisiPodstablo(pRefNZ->pDijete,out);
            else
            {
                CReferencaNaZavrsni *pRefZZ =
                    dynamic_cast<CReferencaNaZavrsni *>(*it);

                if(pRefZZ!=NULL)
                {
                    out << "StavljamNaStog( ";
                    pRefZZ->pZG->Ispisi(out);
                    out << ")" << endl;
                }
            }
        }

        pVrh->pProd->Ispisi(out);
    }

    void OznaciPodstablo(CCvorSinStabla *pVrh, int markacija);
};

////////////////////////////////////////////////////////////////////

class CZStogStablo: public CZStablo
{
public:

    ~CZStogStablo() { PocistiStog(); }

    CPokNaZnakUCvoru PokTrazenZnak;

    // NAPOMENA: ovo je stog koji pokazuje na puno mjesto!
    stack<CReferencaNaZnak *> RefStog;

    void PocistiStog()
    {
```

```
    // ovo je komplicirano zbog pRefNaOrgDna -- moramo nezavrsne brisati
    //  obratnim redosljedom!

    stack<CReferencaNaNezavrsni *> tmstack;
    CReferencaNaNezavrsni *pRefNZ;

    while( !RefStog.empty() )
    {
        CReferencaNaZnak *pRefZn = RefStog.top();

        pRefNZ = dynamic_cast<CReferencaNaNezavrsni *>(pRefZn);

        if(pRefNZ!=NULL)
            tmstack.push(pRefNZ);
        else
            delete pRefZn;

        RefStog.pop();
    }

    while( !tmstack.empty() )
    {
        pRefNZ = tmstack.top();

        ObrisiPodstablo(pRefNZ->pDijete);

        delete pRefNZ;

        tmstack.pop();
    }
}

void Push(CReferencaNaZnak *pRefZn)
{
    RefStog.push(pRefZn);
}

CReferencaNaZnak *Pop()
{
    CReferencaNaZnak *rez;

    if( RefStog.empty() )
        return PopIzStabla();

    rez=RefStog.top();
    RefStog.pop();

    return rez;
}

void PostaviDno(CPokNaZnakUCvoru NovoDno)
{
    Dno = NovoDno;
    bPrviPomakDna = true;
}

void ZaSljedZavrsniPostaviPocStanje(CPokNaZnakUCvoru od, CZStanje *pSt)
{
    CReferencaNaZavrsni *pRefZZ;

    do {
        od.PozZnak++;

        if(od.PozZnak == od.pCvor->ZnProd.size())
            od = od.pCvor->Otac;

        pRefZZ = dynamic_cast<CReferencaNaZavrsni *>(od.RefZnak());

    } while(pRefZZ==NULL);

    pRefZZ->pZadStPrijeRedukcija = pSt;
}
```

```
    // NAPOMENA: ovo je stog koji pokazuje na prazno mjesto!
    CPokNaZnakUCvoru Dno;

protected:

    bool bPrviPomakDna;


    void PomakniDnoNaPrethodni()
    {
        for(;;)
        {
            Dno.PozZnak--;

            if(Dno.PozZnak==-1)
            {
                Dno = Dno.pCvor->Otac;

                if( Dno==PokTrazenZnak ) // trazeni mora biti iznad 'dna' stoga
                    PokTrazenZnak = Dno.pCvor->Otac;

                continue;
            }

            if( dynamic_cast<CReferencaNaAkcijski *>(Dno.RefZnak() ) == NULL )
                break;
        }
    }

    CReferencaNaZnak *PopIzStabla()
    {
        GR_IF( Dno.pCvor==NULL );

        PomakniDnoNaPrethodni();
        if(bPrviPomakDna)
            SrediPrviPomakDna();

        return Dno.RefZnak();
    }

    // Buduci da se radi o LR(1) parseru, redukcija gleda znak unaprijed. Zbog toga
    //  moramo ukloniti efekte svih produkcija neposredno prije prve pozicije dna.
    void SrediPrviPomakDna()
    {
        for(;;)
        {
            CReferencaNaNezavrsni *pRefNZ =
                dynamic_cast<CReferencaNaNezavrsni *>( Dno.RefZnak() );

            if(pRefNZ==NULL) // ako je nije ref. na nez. onda je zavrsni -- sve O.K.
                break;
            else if(pRefNZ->BrIspod!=0)
            {
                Dno.pCvor = pRefNZ->pDijete;
                Dno.PozZnak = Dno.pCvor->ZnProd.size()-1;

                if( dynamic_cast<CReferencaNaAkcijski *>(Dno.RefZnak() ) != NULL )
                    PomakniDnoNaPrethodni();
            }
            else
                PomakniDnoNaPrethodni();
        }

        pRefNaOrgDna = Dno.RefZnak();

        bPrviPomakDna = false;
    }
};

//////////////////////////////////////////////////////////////////
```

```
class CDinamickaDatoteka;

class CDinamickiParser: public COsnovnaKlasa
{
friend CDinamickaDatoteka;

public:
     bool UspjesnoParsiranje;

     CDinamickiParser(): pDinDat(NULL)
     {

     }

     void Inicijaliziraj(CZDKA *pzdka)
     {
          pZDKA = pzdka;

          // inicijalizacija stabla
          StogStab.pGlava->ZnProd.insert(
               StogStab.pGlava->ZnProd.begin(),
               new CReferencaNaNezavrsni( pZDKA->pPocSt, pZDKA->pGram->pPocNez,
                    NULL, 0) );
     }

     void ParsirajSve();

     void PromjeniBrojZnakovaIspod(CPokNaZnakUCvoru koji, int koliko)
     {
          if(koji.pCvor==NULL || koliko==0 )
               return;

          CReferencaNaNezavrsni *pRefNZ =
               dynamic_cast<CReferencaNaNezavrsni *>( koji.RefZnak() );

          GR_IF( pRefNZ==NULL );

          pRefNZ->BrIspod += koliko;

          PromjeniBrojZnakovaIspod( koji.pCvor->Otac, koliko);
     }

     pair<int, int> Popravi(int BrPromZn=0);

     void Ispisi(ostream &out)
     {
          StogStab.IspisiPodstablo( dynamic_cast<CReferencaNaNezavrsni *>(
               StogStab.pGlava->ZnProd.front())->pDijete, out );
     }

     CZDKA *pZDKA;
/*   todo: ovo je za TreeProzor
     CTreeProzor *pTreePr;

     void IspisiPodstabloUTreeProzor(CCvorSinStabla *pVrh, HTREEITEM gdje)
     {
          GR_IF(pVrh==NULL);

          NIZPTRREFZN::iterator it = pVrh->ZnProd.begin();

          if(it==pVrh->ZnProd.end())
               pTreePr->UbaciItem( gdje, "eps", (LPARAM)NULL);
          else
               do {
                    stringstream tms;

                    (*it)->pZG->Ispisi(tms);

                    (*it)->hPozUTree = pTreePr->UbaciItem( gdje,
                         (char *)tms.str().c_str(), (LPARAM) (*it) );

                    CReferencaNaNezavrsni *pRefNZ =
```

```
                    dynamic_cast<CReferencaNaNezavrsni *>(*it);

                if(pRefNZ!=NULL)
                    IspisiPodstabloUTreeProzor( pRefNZ->pDijete, pRefNZ->hPozUTree);

                it++;
            } while(it!=pVrh->ZnProd.end());
    }
*/
    enum REZ_PARSIRANJA { USPJESNO=0, NE_POSTOJI_ZAVRSNI, NEOCEKIVANI_KRAJ,
        VISAK_ZNAKOVA_ILI_SINTAKSA, NEISPRAVNA_SINTAKSA };

protected:

    CDinamickaDatoteka *pDinDat;

    CZStogStablo StogStab;

    REZ_PARSIRANJA Parsiraj();

    bool MozesPremotatiNaPocCvora(CPokNaZnakUCvoru koji)
    {
        for(;;)
        {
            if(koji.PozZnak==0)
                return true;

            koji.PozZnak--;

            CReferencaNaZnak *pRefZG = koji.RefZnak();

            if( dynamic_cast<CReferencaNaZavrsni *>(pRefZG)!=NULL )
                return false;
            else {
                CReferencaNaNezavrsni *pRefNZ =
                    dynamic_cast<CReferencaNaNezavrsni *>(pRefZG);

                if( pRefNZ!=NULL )
                {
                    if( pRefNZ->BrIspod != 0)
                        return false;
                }
            }
        }
    }

    bool GotovPopravak(CZStanje *pStPrijeRed);

    void ZamjeniSa(CPokNaZnakUCvoru koji,  CReferencaNaNezavrsni *pSaKojim)
    {
        GR_IF( koji.PozZnak>=koji.pCvor->ZnProd.size() || koji.PozZnak<0 );

        NIZPTRREFZN::iterator it = koji.pCvor->ZnProd.begin()+koji.PozZnak;

        CReferencaNaNezavrsni *pRefNZ = dynamic_cast<CReferencaNaNezavrsni *>(*it);

        // popravak prikaza..
/* todo: TreeProzor
        if(pRefNZ->pDijete!=NULL)
            pTreePr->ObrisiDjecu(pRefNZ->hPozUTree);

        pSaKojim->hPozUTree = pRefNZ->hPozUTree;

        pTreePr->PromjeniLParamZaItem( pSaKojim->hPozUTree, (LPARAM) pSaKojim );

        IspisiPodstabloUTreeProzor(pSaKojim->pDijete, pSaKojim->hPozUTree);
*/
        // ..kraj popravka prikaza

        it = koji.pCvor->ZnProd.erase( it );

        koji.pCvor->ZnProd.insert( it, pSaKojim );
```

```
        pSaKojim->pDijete->Otac = koji;

        StogStab.ObrisiPodstablo(pRefNZ->pDijete);

        delete pRefNZ;

        // nakon svega, mozemo promjeniti ulaznu datoteku
        PoveziPodstablo(pSaKojim->pDijete);
    }


    void PoveziPodstablo(CCvorSinStabla *pVrh);

    CZStanje *Reduciraj(CKonNeoProdukcija *pProd);
};

////////////////////////////////////////////////////////////////////

class CZZnak: public COsnovnaKlasa
{
public:
    CZZnak(TIPTOKENA tip, string tokstr): Tip(tip),
        TokStr(tokstr), bSinkroniziran(false), Generacija(-1) { }
    CZZnak(const CZZnak &zz): Tip(zz.Tip), TokStr(zz.TokStr),
        PozStablo(zz.PozStablo), bSinkroniziran(false),
        Generacija(zz.Generacija) { }

    TIPTOKENA Tip;
    string TokStr;

    CPokNaZnakUCvoru PozStablo;
    bool bSinkroniziran;

    int Generacija;
};

typedef  my_vect<CZZnak *> NIZPTRZZNAK;

////////////////////////////////////////////////////////////////////

class CZRed: public COsnovnaKlasa
{
public:
    CZRed(const CZRed &zr): NizZn(zr.NizZn), bImaCR(false) { }

    CZRed(): bImaCR(false) { }

    NIZPTRZZNAK NizZn;

    // za podrsku CR/LF (DOS) i LF (UNIX) datotekama -- redove zapisuje bez
    //  jednoga i drugoga. Za LF znamo da mora doci, a da li ima CR pamtimo
    //   za svaki red
    bool bImaCR;
};

////////////////////////////////////////////////////////////////////

class CAkcijskiProba: public CSkupAkcija
{
public:
    CAkcijskiProba()
    {
        Dodaj( CAkcija("{radi}") );
        Dodaj( CAkcija("{brisi}") );
    }
};

////////////////////////////////////////////////////////////////////

class CDinamickaDatoteka: public COsnovnaKlasa, public CSynDodatniPodaci
{
```

```
friend CDinamickiParser;

public:

    typedef my_vect<CZRed> NIZZRED;

    CDinamickaDatoteka( CLexAnalizator & reflexana):
        RefLexAna(reflexana), pTrZnak(NULL),
        BrRed(-1), BrStupac(-1), BrZnakova(0)
    {
        // alociraj CZDKA i ostale ako treba
        if( BrReferenciNaZDKA==0 )
        {
            if( pZDKA!=NULL )
                throw string("Nesto ne valja u CDinamickaDatoteka(...)");

            CAkcijskiProba *pAkc = new CAkcijskiProba();

            CKonNeoGramatika *pGram = new CKonNeoGramatika(
                "Podaci/gramatika.dat",*pAkc);

            pGram->IzracunajSvePraznoce();
            pGram->IzracunajFirstSkupove();

            pZDKA = new CZDKA(*pGram);

            pZDKA->IzgradiCijeliDKA();
/*
            ofstream tmf("temp/stanja.txt");

            pZDKA->Ispisi(tmf,true);
*/      }

        // jedna vise referenca na ZDKA
        BrReferenciNaZDKA++;

        DinPars.Inicijaliziraj(pZDKA);

        DinPars.pDinDat=this;

        // inicijalizacija datoteke preuzeta iz Ucitaj()

        NizRed.push_back( CZRed() );

        CZZnak *pTmZ = new CZZnak(ZNKRNIZA,"\0");

        BrZnakova = 0;

        pTmZ->PozStablo.pCvor = DinPars.StogStab.pGlava;
        pTmZ->PozStablo.PozZnak = 1;
        pTmZ->bSinkroniziran = true;

        NizRed[ NizRed.size()-1 ].NizZn.push_back( pTmZ );

        Premotaj(0,0);

        // inicijalizacija iz leksike

        // ovdje ide VratiBrojElem()-1 jer ne dodajemo znak za kraj
        UmetniObrisi(0,0,RefLexAna.VratiBrojElem()-1,0,true);

        DinPars.ParsirajSve();
    }

    ~CDinamickaDatoteka()
    {
        NIZZRED::iterator it;
        NIZPTRZZNAK::iterator znit;

        for(it=NizRed.begin();it!=NizRed.end();it++)
            for(znit=it->NizZn.begin();znit!=it->NizZn.end();znit++)
                delete *znit;
```

```
        // obrisi i ZDKA ako treba

        BrReferenciNaZDKA--;

        if( BrReferenciNaZDKA==0 )
        {
            delete pZDKA->pGram->pSkAkc;
            delete pZDKA->pGram;
            delete pZDKA;

            pZDKA = NULL;
        }

    }

    void Ispisi(ostream &out)
    {
        Premotaj(0,0);

        for(; pTrZnak->Tip != ZNKRNIZA ; PomakniNaSljedeci() )
            out << pTrZnak->TokStr;
    }

    // todo: ovo je samo za debug
    void DebugUDatoteku()
    {
        static int brbris=0;
        char buffer[20];
        string fname("TEMP/syn_bris");

        fname += itoa( brbris, buffer, 10);
        fname += ".txt";

        ofstream ofs( fname.c_str() );

        IspisiDbg(ofs);

        ofs.close();

        brbris++;
    }

    void IspisiDbg( ostream & out )
    {
        NIZZRED::iterator datit;

        for(datit=NizRed.begin();datit!=NizRed.end();datit++)
        {
            NIZPTRZZNAK::iterator redit;

            for(redit=datit->NizZn.begin();redit!=datit->NizZn.end();redit++)
            {
                out << ImeTokena[ (*redit)->Tip ] << " ";

                if( (*redit)->Tip==ODBACI || (*redit)->Tip==NASTAVAK )
                    out << "--- ";
                else
                    out << (*redit)->TokStr << " ";
            }

            out << endl;
        }
    }

    void ProvjeriJednakost()
    {
        NIZZRED::iterator datit;

        CLexAnalizator::Iterator lexit(0,0,&RefLexAna);

        for(datit=NizRed.begin();datit!=NizRed.end();datit++)
```

```
        {
             NIZPTRZZNAK::iterator redit;

             for(redit=datit->NizZn.begin();redit!=datit->NizZn.end();redit++,++lexit)
                  if( (*redit)->Tip != (*lexit).VratiTip() )
                  {
                       if( (*lexit).VratiTip()!=ZNKRNIZA ||
                            (*redit)->Tip != ZNKRNIZA)
                       {
                            stringstream ss;
                            ss << "Tip se ne podudara. ";
                            ss << "Tip od (*redit)->Tip je ";
                            ss << ImeTokena[ (*redit)->Tip ] << endl;
                            ss << "a od (*lexit).VratiTip() je " << ImeTokena[
(*lexit).VratiTip() ] << endl;
                            ss << "red datit-NizRed.begin(): " << int(datit-
NizRed.begin()) << endl;
                            ss << "stupac redit-datit->NizZn.begin(): " << int(redit-
datit->NizZn.begin()) << endl;
                            ss << "red lexit.VratiRed(): " << lexit.VratiRed() << endl;
                            ss << "stupac lexit.VratiStup(): " << lexit.VratiStup() <<
endl;
                            throw string(ss.str());
                       }
                  }
                  else if( datit-NizRed.begin() != lexit.VratiRed() )
                       {
                            stringstream ss;
                            ss << "Red se ne podudara. ";
                            ss << "Tip od (*redit)->Tip je ";
                            ss << ImeTokena[ (*redit)->Tip ] << endl;
                            ss << "a od (*lexit).VratiTip() je " << ImeTokena[
(*lexit).VratiTip() ] << endl;
                            ss << "red datit-NizRed.begin(): " << int(datit-
NizRed.begin()) << endl;
                            ss << "stupac redit-datit->NizZn.begin(): " << int(redit-
datit->NizZn.begin()) << endl;
                            ss << "red lexit.VratiRed(): " << lexit.VratiRed() << endl;
                            ss << "stupac lexit.VratiStup(): " << lexit.VratiStup() <<
endl;
                            throw string(ss.str());
                       }
                  else if( redit-datit->NizZn.begin() != lexit.VratiStup() )
                       throw string("Stupac se ne podudara");
        }
    }

    int VratiBrojRedova()
    {
         return NizRed.size();
    }

    int VratiVelReda(int koji)
    {
         GR_IF(koji<0||koji>=NizRed.size());

         int len = NizRed[koji].NizZn.size();

         if(koji == NizRed.size()-1)
              len--;

         return len;
    }

    CZZnak *ZnakNaPoz(int red, int stupac, bool bFatalErr=true)
    {
         if(red<0||stupac<0||red>=NizRed.size()||stupac>=NizRed[red].NizZn.size())
         {
              if(bFatalErr)
                   GR_INT_CLASS << "ne postoji znak na toj poziciji"<< kraj;
              else
                   return NULL;
```

```
        }

        return NizRed[red].NizZn[stupac];
    }

    pair<int, int> UmetniObrisi(int red, int stupac, int BrUmet, int BrObris, bool
bPrvoPars=false)
    {
        // prva verzija:
        int i;

        // ODBACI i NASTAVAK ne brojimo kao tokene
        int korektor=0;

        // obrisi:

        for(i=0;i<BrObris;i++)
        {
            Premotaj(red,stupac);

            CPokNaZnakUCvoru TmPok = pTrZnak->PozStablo;
            int TmTip = pTrZnak->Tip;

            NIZPTRZZNAK::iterator it;

            delete NizRed[red].NizZn[stupac];

            // brise znak, vraca iterator na znak iza obrisanog
            it = NizRed[red].NizZn.erase( NizRed[red].NizZn.begin()+stupac );

            // smanjuje broj znakova
            BrZnakova--;

            // nakon ovog, pTrZnak pokazuje na znak iza obrisanog
            if( it!=NizRed[red].NizZn.end() )
                pTrZnak = *it;
            else
            {
                SpojiSaSljedecimRedom(red);
                Premotaj(red,stupac);
            }

            if( TmTip==ODBACI || TmTip==NASTAVAK )
            {
                korektor--;
                continue;
            }

            // treba preskociti sve ODBACI ili NASTAVAK znakove
            if( pTrZnak->Tip==ODBACI || pTrZnak->Tip==NASTAVAK )
                PomakniNaSljedeci();

            // cvor vise ne pokazuje na znak iza obrisanog..
            if( pTrZnak->PozStablo.pCvor!=NULL )
            {
                CReferencaNaZavrsni *pRefZZ = dynamic_cast<CReferencaNaZavrsni *>(
                    pTrZnak->PozStablo.RefZnak() );

                pRefZZ->pZZnak = NULL;
            }

            // ..znak iza obrisanog pokazuje na cvor od obrisanog..
            pTrZnak->PozStablo = TmPok;
            pTrZnak->bSinkroniziran = false;

            // ..i, naravno, cvor od obrisanog pokazuje na novi znak
            if( TmPok.pCvor!=NULL && TmPok.pCvor!=DinPars.StogStab.pGlava )
                dynamic_cast<CReferencaNaZavrsni *>( TmPok.RefZnak() )->pZZnak =
pTrZnak;
        }

        // umetni:
```

```
        CLexAnalizator::Iterator lexit(red,stupac,&RefLexAna);

        for(i=0;i<BrUmet-1;i++)
            ++lexit;

        for(i=0;i<BrUmet;i++,--lexit)
        {
            TIPTOKENA tip = (TIPTOKENA) (*lexit).VratiTip();
            string tmp_s = (*lexit).VratiTokStr();


    GR_IF(red<0||stupac<0||red>=NizRed.size()||stupac>NizRed[red].NizZn.size());

            // mislim da bi ovo trebalo rijesiti problem podudarnosti redova
            if( lexit.VratiStup()==RefLexAna.VratiVelReda(lexit.VratiRed())-1 )
                PrelomiRed(red,stupac);

            // ako je skroz na kraju reda..
            if(stupac==NizRed[red].NizZn.size())
            {
                // ..moramo naci sljedeci znak u nekom od sljedecih redova
                BrRed=red;
                BrStupac=stupac;
                PomakniNaSljedeci();
            }
            else // inace samo premotaj i preskoci ODBACI i NASTAVAK
            {
                Premotaj(red,stupac);

                // treba preskociti sve ODBACI ili NASTAVAK znakove
                if( pTrZnak->Tip==ODBACI || pTrZnak->Tip==NASTAVAK )
                    PomakniNaSljedeci();
            }

            // cvor znaka ispred kojeg umecemo
            CPokNaZnakUCvoru TmPok = pTrZnak->PozStablo;

            if( tip==ODBACI || tip==NASTAVAK )
            {
                korektor++;
            }
            else
            {
                // trenutni znak (ispred kojeg umecemo) vise ne pokazuje na cvor X..
                pTrZnak->PozStablo.pCvor = NULL;
                pTrZnak->PozStablo.PozZnak = -1;
            }

            // umetanje novog znaka
            NizRed[red].NizZn.insert(
                NizRed[red].NizZn.begin()+stupac, new CZZnak(tip,tmp_s) );

            // povecavamo broj znakova
            BrZnakova++;

            // premotavanje na umetnuti znak
            Premotaj(red,stupac);

            if( tip==ODBACI || tip==NASTAVAK )
            {
                pTrZnak->bSinkroniziran = true;
                continue;
            }

            // ..nego umetnuti znak pokazuje na cvor X..
            pTrZnak->PozStablo = TmPok;
            pTrZnak->bSinkroniziran = false;

            // ..a naravno, i cvor pokazuje na novi znak
            if( TmPok.pCvor!=NULL && TmPok.pCvor!=DinPars.StogStab.pGlava )
```

```
                dynamic_cast<CReferencaNaZavrsni *>( TmPok.RefZnak() )->pZZnak =
pTrZnak;
        }

        // pozovi parsiranje:

        if(!bPrvoPars)
        {
            pair<int,int> tmpar = DinPars.Popravi( BrUmet-BrObris-korektor );

            if( tmpar.first > red )
                tmpar.first = red;

            return tmpar;
        }
        else
        {
            BrZnakova -= korektor;

            return pair<int,int>(red,red);
        }
    }

    void OznaciUTreeView(int red, int stupac)
    {
        if(red<0||stupac<0||red>=NizRed.size()||stupac>=NizRed[red].NizZn.size())
            return;

        CZZnak *pZZnak = NizRed[red].NizZn[stupac];
/*todo: TreeProzor
        if(pZZnak->PozStablo.pCvor!=NULL && DinPars.pTreePr->bOsvjeziTree)
            DinPars.pTreePr->OznaciItem( pZZnak->PozStablo.RefZnak()->hPozUTree );
*/  }

    void SpojiSaSljedecimRedom(int koji)
    {
        GR_IF(koji<0||koji>=NizRed.size()-1);

        NizRed[koji].NizZn.insert(
            NizRed[koji].NizZn.end(),
            NizRed[koji+1].NizZn.begin(),
            NizRed[koji+1].NizZn.end() );

        NizRed.erase( NizRed.begin()+koji+1 );
    }

    void PrelomiRed(int red, int stupac)
    {
        GR_IF(red<0||stupac<0||red>=NizRed.size()||stupac>NizRed[red].NizZn.size());

        // ubaci novi red
        NIZZRED::iterator it = NizRed.insert( NizRed.begin()+red+1 );

        NIZPTRZZNAK::iterator poc = NizRed[red].NizZn.begin()+stupac;
        NIZPTRZZNAK::iterator kraj = NizRed[red].NizZn.end();

        // stavi u taj red zadnji dio trenutnog
        it->NizZn.insert( it->NizZn.begin(), poc, kraj );

        // u trenutnom obrisi zadnji dio
        NizRed[red].NizZn.erase( poc, kraj );
    }

    CDinamickiParser DinPars;

    CZZnak *pTrZnak;

    void Premotaj(int red, int stupac)
    {
        GR_IF(red<0||stupac<0||red>=NizRed.size()||stupac>=NizRed[red].NizZn.size());

        BrRed=red;
```

```
        BrStupac=stupac;

        pTrZnak = NizRed[BrRed].NizZn[BrStupac];
    }

    CLexAnalizator &RefLexAna;

    void PomakniNaSljedeci(bool bPreskociTokene=true)
    {
        do
        {
            BrStupac++;

            for(;BrStupac >= NizRed[BrRed].NizZn.size();)
            {
                BrRed++;
                BrStupac=0;
            }

            pTrZnak = NizRed[BrRed].NizZn[BrStupac];
        }
        while( bPreskociTokene &&
            (pTrZnak->Tip==ODBACI || pTrZnak->Tip==NASTAVAK) );
    }

    bool PomakniNaPrethodni()
    {
        do
        {
            BrStupac--;

            for(;BrStupac < 0;)
            {
                BrRed--;
                if(BrRed<0)
                {
                    BrRed=0;
                    return false;
                }

                BrStupac = NizRed[BrRed].NizZn.size()-1;
            }

            pTrZnak = NizRed[BrRed].NizZn[BrStupac];
        }
        while( pTrZnak->Tip==ODBACI || pTrZnak->Tip==NASTAVAK );

        return true;
    }

    void NovaGeneracija( CPokNaZnakUCvoru &pozst, int generacija )
    {
        if( pozst.pCvor==NULL )
            return;

        DinPars.StogStab.OznaciPodstablo( pozst.pCvor, generacija );
    }

private:

    NIZZRED NizRed;

    int BrRed;
    int BrStupac;
    int BrZnakova;

    static CZDKA *pZDKA;
    static int BrReferenciNaZDKA;
};

//////////////////////////////////////////////////////////////////
```

```
#endif
```

9.2.7. ZParser.cpp

```
#include "stdafx.h"

#include "ZParser.h"

//////////////////////////////////////////////////////////////////////

void CZStablo::OznaciPodstablo(CCvorSinStabla *pVrh, int markacija)
{
     GR_IF(pVrh==NULL);

     NIZPTRREFZN::iterator it;

     for(it=pVrh->ZnProd.begin();it!=pVrh->ZnProd.end();it++)
     {
          CReferencaNaNezavrsni *pRefNZ =
               dynamic_cast<CReferencaNaNezavrsni *>(*it);

          if(pRefNZ!=NULL)
               OznaciPodstablo(pRefNZ->pDijete,markacija);
          else
          {
               CReferencaNaZavrsni *pRefZZ =
                    dynamic_cast<CReferencaNaZavrsni *>(*it);

               if(pRefZZ!=NULL && pRefZZ->pZZnak!=NULL)
                    pRefZZ->pZZnak->Generacija = markacija;
          }
     }
}

//////////////////////////////////////////////////////////////////////

CZDKA *CDinamickaDatoteka::pZDKA=NULL;

int CDinamickaDatoteka::BrReferenciNaZDKA=0;

//////////////////////////////////////////////////////////////////////

void CDinamickiParser::ParsirajSve()
{
     // krecemo od glave
     CPokNaZnakUCvoru koji(StogStab.pGlava,0);

     CReferencaNaNezavrsni *pRefNZ =
          dynamic_cast<CReferencaNaNezavrsni *>(koji.RefZnak());

     // moramo ubaciti pocetni u prikaz stabla
/* todo: TreeProzor
     if(pRefNZ->hPozUTree==NULL)
     {
          stringstream tms;

          pRefNZ->pZG->Ispisi(tms);
          pRefNZ->hPozUTree = pTreePr->UbaciItem( TVI_ROOT,
                                   (char *)tms.str().c_str(), (LPARAM) pRefNZ );

          pTreePr->OznaciItem( pRefNZ->hPozUTree );
     }
*/
     // inicijalizacija ulaza
     pDinDat->Premotaj(0,0);

     if( pDinDat->pTrZnak->Tip==ODBACI || pDinDat->pTrZnak->Tip==NASTAVAK )
```

```
        pDinDat->PomakniNaSljedeci();

    pDinDat->pTrZnak->PozStablo = koji;
    pDinDat->pTrZnak->bSinkroniziran = false;

    Popravi( pDinDat->BrZnakova );
}

////////////////////////////////////////////////////////////////////

pair<int, int> CDinamickiParser::Popravi(int BrPromZn)
{
    // vracamo od kojeg do kojeg reda smo popravljali (uspjesno ili neuspjesno)
    int PrviPopRed, ZadnjiPopRed, RedZahtjeva;

    // provjera stoga
    GR_IF(!StogStab.RefStog.empty());

    // trenutni red u kojem smo
    RedZahtjeva = pDinDat->BrRed;
/*
    if( BrPromZn==0 )
        return pair<int, int>(RedZahtjeva, RedZahtjeva);
*/
    if( pDinDat->pTrZnak->Tip==ODBACI || pDinDat->pTrZnak->Tip==NASTAVAK )
        pDinDat->PomakniNaSljedeci();

    // premotaj do prvog koji nije NULL..
    while( pDinDat->pTrZnak->PozStablo.pCvor==NULL )
        if( pDinDat->PomakniNaPrethodni()==false )
            GR_INT_CLASS << "sve do pocetka datoteke su NULL pokazivaci" << kraj;

    CPokNaZnakUCvoru koji = pDinDat->pTrZnak->PozStablo;

    // ..i za njega promjeni rekurzivno BrIspod
    if( koji.pCvor!=StogStab.pGlava )
    {
        dynamic_cast<CReferencaNaZavrsni *>( koji.RefZnak() )
            ->BrIspod += BrPromZn;

        PromjeniBrojZnakovaIspod( koji.pCvor->Otac, BrPromZn );
    }
    else
        PromjeniBrojZnakovaIspod( CPokNaZnakUCvoru(StogStab.pGlava,0), BrPromZn );

    // premotaj do pocetka nesinkroniziranog bloka
    for(; pDinDat->PomakniNaPrethodni() && !pDinDat->pTrZnak->bSinkroniziran ;) ;

    pDinDat->PomakniNaSljedeci();

    PrviPopRed = pDinDat->BrRed;

    // zapamti koji znak trazimo

    koji = pDinDat->pTrZnak->PozStablo;

    if( koji.pCvor!=StogStab.pGlava )
        StogStab.PokTrazenZnak = koji.pCvor->Otac;
    else
        // ovo treba zbog ParsirajSve() i dodavanja na kraj datoteke
        StogStab.PokTrazenZnak = CPokNaZnakUCvoru(StogStab.pGlava,0);

    StogStab.PostaviDno( koji );

    // inicijalizacija DKA
    CReferencaNaZavrsni *pRefZZ =
        dynamic_cast<CReferencaNaZavrsni *>(koji.RefZnak());

    if(pRefZZ!=NULL)
        pZDKA->pTrSt = pRefZZ->pZadStPrijeRedukcija;
    else
        pZDKA->pTrSt = pZDKA->pPocSt;
```

```
     // e sad mozemo..

     if( Parsiraj()==USPJESNO && pDinDat->pTrZnak->Tip == ZNKRNIZA )
     {
          pDinDat->pTrZnak->bSinkroniziran = true;
          pDinDat->pTrZnak->PozStablo = CPokNaZnakUCvoru(StogStab.pGlava,1);
     }

     ZadnjiPopRed = pDinDat->BrRed;

     return pair<int, int>(PrviPopRed, _MAX(ZadnjiPopRed,RedZahtjeva));
}

////////////////////////////////////////////////////////////////////

CDinamickiParser::REZ_PARSIRANJA CDinamickiParser::Parsiraj()
{
     CKonNeoProdukcija *pProd;
     CZavrsniZnak *pZZ;
     CZStanje *pStaroStPrijeRedukcija;
     CZStanje *pStaroSt;

     UspjesnoParsiranje=false;

     pStaroStPrijeRedukcija = pZDKA->pTrSt;

     for(;;pDinDat->PomakniNaSljedeci())
     {
          pZZ = pZDKA->pGram->NadjiZavrsni( pDinDat->pTrZnak->Tip,
               pDinDat->pTrZnak->TokStr );

          for(;;)
          {
               pStaroSt = pZDKA->pTrSt;

               if( pZZ==NULL || !pZDKA->NapraviPrijelaz( pZZ, &pProd ) )
               {
                    StogStab.PocistiStog();

                    if(pZZ==NULL)
                         return NE_POSTOJI_ZAVRSNI;
                    else if( pDinDat->pTrZnak->Tip == ZNKRNIZA )
                         return NEOCEKIVANI_KRAJ;
                    else if( pZDKA->NapraviPrijelaz(
                         &(pZDKA->pGram->ZnKrNiza), &pProd ) )
                         return VISAK_ZNAKOVA_ILI_SINTAKSA;
                    else
                         return NEISPRAVNA_SINTAKSA;
               }

               if(pProd==NULL)
               { // ako je normalan prijelaz
                    pDinDat->pTrZnak->bSinkroniziran = false;

                    StogStab.Push( new CReferencaNaZavrsni(pStaroSt,
                         pStaroStPrijeRedukcija, pZZ, pDinDat->pTrZnak) );

                    pStaroStPrijeRedukcija = pZDKA->pTrSt;

                    break;
               }

               // inace je redukcija..

               CZStanje *pRedStanje = Reduciraj(pProd);

               if( GotovPopravak(pStaroStPrijeRedukcija) )
               {
                    UspjesnoParsiranje=true;
                    return USPJESNO; // popravio je stablo, zavrsi sa parsiranjem
               }
```

```
            else
                pZDKA->NapraviRedukciju( pRedStanje, pProd->LStrana );
        }
    }
}

////////////////////////////////////////////////////////////////////

bool CDinamickiParser::GotovPopravak(CZStanje *pStPrijeRed)
{
    if( StogStab.RefStog.size()!=1 )
        return false;

    if( pDinDat->pTrZnak->bSinkroniziran==false &&
         pDinDat->pTrZnak->Tip != ZNKRNIZA )
        return false;

    CReferencaNaNezavrsni *pRefNZStog =
        dynamic_cast<CReferencaNaNezavrsni *>(StogStab.RefStog.top());
    GR_IF(pRefNZStog==NULL);

    CReferencaNaNezavrsni *pRefNZTr;
    CPokNaZnakUCvoru x;

    if( StogStab.PokTrazenZnak == CPokNaZnakUCvoru(StogStab.pGlava,0) )
    {
        x = StogStab.PokTrazenZnak;

        pRefNZTr = dynamic_cast<CReferencaNaNezavrsni *>( x.RefZnak() );
        GR_IF(pRefNZTr==NULL);

        if( pRefNZTr->BrIspod != pRefNZStog->BrIspod
            || pRefNZTr->pZG != pRefNZStog->pZG )
            return false; // nisu jednaki
    }
    else
    {
        bool bFlag = false;

        for(x = StogStab.Dno;;)
        {
            // ako nije na 'pocetku' onda sigurno nije
            if(!MozesPremotatiNaPocCvora(x))
                return false;

            // nadji oca
            x = x.pCvor->Otac;

            // ako dosli do vrha stabla onda sigurno nije
            if(x.pCvor==NULL)
                return false;

            // ovdje ide provjera jednakosti..
            pRefNZTr = dynamic_cast<CReferencaNaNezavrsni *>( x.RefZnak() );
            GR_IF(pRefNZTr==NULL);

/* // ovo trenutno ne vrijedi jer neki cvorovi imaju BrIspod=-1
            if( pRefNZTr->BrIspod > pRefNZStog->BrIspod ) // ako smo se popeli
                return false; // 'previsoko' u stablo, mozes prekinuti
*/
            if( x == StogStab.PokTrazenZnak )
                bFlag = true;

            if( bFlag
                && pRefNZTr->BrIspod == pRefNZStog->BrIspod
                && pRefNZTr->pZG == pRefNZStog->pZG )
                break; // jednaki su!
        }
    }

    ZamjeniSa( x, pRefNZStog );
```

```
     StogStab.Pop();

     StogStab.ZaSljedZavrsniPostaviPocStanje( x, pStPrijeRed );

     return true;
}

//////////////////////////////////////////////////////////////////////

void CDinamickiParser::PoveziPodstablo(CCvorSinStabla *pVrh)
{
     GR_IF(pVrh==NULL);

     NIZPTRREFZN::iterator it;

     for(it=pVrh->ZnProd.begin();it!=pVrh->ZnProd.end();it++)
     {
          CReferencaNaNezavrsni *pRefNZ =
               dynamic_cast<CReferencaNaNezavrsni *>(*it);

          if(pRefNZ!=NULL)
          {
               if( (*it)->bNovo )
               {
                    PoveziPodstablo( pRefNZ->pDijete );
                    (*it)->bNovo = false;
               }

               pRefNZ->pDijete->Otac.pCvor = pVrh;
               pRefNZ->pDijete->Otac.PozZnak = it - pVrh->ZnProd.begin();
          }
          else
          {
               CReferencaNaZavrsni *pRefZZ =
                    dynamic_cast<CReferencaNaZavrsni *>(*it);

               if( pRefZZ != NULL && pRefZZ->pZZnak != NULL )
               {
                    pRefZZ->pZZnak->PozStablo.pCvor = pVrh;
                    pRefZZ->pZZnak->PozStablo.PozZnak = it - pVrh->ZnProd.begin();

                    if( (*it)->bNovo )
                    {
                         pRefZZ->pZZnak->bSinkroniziran = true;

                         (*it)->bNovo = false;
                    }
               }
          }
     }
}

//////////////////////////////////////////////////////////////////////

CZStanje * CDinamickiParser::Reduciraj(CKonNeoProdukcija *pProd)
{
     CCvorSinStabla *pCvor;

     CReferencaNaZnak *pRefZn;
     int SumaIspod=0;
     // ovo ce se modificirati kasnije u procesu ako nije istina
     CZStanje *rezSt = pZDKA->pTrSt;

     pCvor = new CCvorSinStabla();

     pCvor->pProd = pProd;

     pCvor->ZnProd.reserve( pProd->DStrana.size() );

     NIZPTRZNGR::reverse_iterator rit;

     for(rit=pProd->DStrana.rbegin();rit!=pProd->DStrana.rend();rit++)
```

```
    {
        CAkcijskiZnak *pRefAZ = dynamic_cast<CAkcijskiZnak *>(*rit);

        if( pRefAZ==NULL)
        {
            pRefZn = StogStab.Pop();
            rezSt = pRefZn->VratiStanjePrije();

            IFDEBUG if( pRefZn->pZG != *rit )
                GR_INT_CLASS << "nemoguca redukcija (DKA != stog)" << kraj;

            pCvor->ZnProd.insert( pCvor->ZnProd.begin(), pRefZn );

            CReferencaNaZavrsni *pRefZZ =
                dynamic_cast<CReferencaNaZavrsni *>(pRefZn);

            if( pRefZZ!=NULL )
                SumaIspod += pRefZZ->BrIspod;
            else
                SumaIspod += (dynamic_cast<CReferencaNaNezavrsni *>
                    (pRefZn))->BrIspod;
        }
        else
        {
            pCvor->ZnProd.insert( pCvor->ZnProd.begin(),
                new CReferencaNaAkcijski(pRefAZ) );
        }
    }


    StogStab.Push( new CReferencaNaNezavrsni(rezSt, pProd->LStrana, pCvor, SumaIspod)
);

    return rezSt;
}

//////////////////////////////////////////////////////////////////////
```

9.3. Datoteke klasa povezanih sa prikazom i uporabom gramatike

9.3.1. Globalno.h

```
//
//      Globalne funkcije i definicije
//

#if !defined(GLOBALNO_H)
#define GLOBALNO_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

//////////////////////////////////////////////////////////////////////

template<class _T>
    int DodajSet(set<_T> & left, set<_T> & right)
```

```
{
     int brojac=0;

     // iterator kroz set
     set<_T>::iterator tmit;

     // dodaj u left sve iz right
     for(tmit=right.begin();tmit!=right.end();tmit++)
          if(left.insert(*tmit).second)
               brojac++;

     return brojac;
}

////////////////////////////////

template<class _T>
     set<_T> & operator+=(set<_T> & left, set<_T> & right)
{
     DodajSet(left,right);

     return left;
}

////////////////////////////////

template<class _T>
     bool Podskup(set<_T> & podskup, set<_T> & odSkupa)
{
     set<_T>::iterator tmit;

     for(tmit=podskup.begin();tmit!=podskup.end();tmit++)
          if( odSkupa.find( *tmit ) == odSkupa.end() )
               return false;

     return true;
}

////////////////////////////////////////////////////////////////////

#endif
```

9.3.2. ZZnakovi.h

```
//
//       Znakovi
//

#if !defined(ZZNAKOVI_H)
#define ZZNAKOVI_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ZAkcije.h"

#include "../Tokeni.h"

////////////////////////////////////////////////////////////////////

class CZnakGramatike: public COsnovnaKlasa
{
public:
     virtual void Ispisi(ostream &out, bool bSve=false)=0;
};
```

```
typedef my_vect<CZnakGramatike *> NIZPTRZNGR;

//////////////////////////////////////////////////////////////////

class CZavrsniZnak: public CZnakGramatike
{
public:
     CZavrsniZnak(TIPTOKENA tip, const string &ime): Tip(tip), Ime(ime) { }
     CZavrsniZnak(const string &ime): Tip(NEPOZNATO), Ime(ime) { }
     CZavrsniZnak(const CZavrsniZnak &zz): Tip(zz.Tip), Ime(zz.Ime) { }

     TIPTOKENA Tip;
     string Ime;

     virtual void Ispisi(ostream &out, bool bSve=false)
     {
          out << '\'' << Ime << "\' ";
     }

     friend const bool operator<(const CZavrsniZnak & left, const CZavrsniZnak & right)
     {
          return ( left.Ime < right.Ime );
     }

     friend const bool operator==(const CZavrsniZnak & left, const CZavrsniZnak &
right)
     {
          return ( left.Ime == right.Ime );
     }
};

typedef set<CZavrsniZnak> SETZZ;
typedef set<CZavrsniZnak *> SETPTRZZ;

//////////////////////////////////////////////////////////////////

class CZnakSaImenom: public CZnakGramatike
{
public:
     CZnakSaImenom(): BrAtrib(0) { }
     CZnakSaImenom(const string &ime, int bratrib): Ime(ime), BrAtrib(bratrib) { }

     string Ime;
     int BrAtrib;

     virtual void Ispisi(ostream &out, bool bSve=false)
     {
          out << Ime << " ";
          if(bSve) out << BrAtrib << " ";
     }

     friend const bool operator<(const CZnakSaImenom & left, const CZnakSaImenom &
right)
     {
          return ( left.Ime < right.Ime );
     }
};

//////////////////////////////////////////////////////////////////

typedef set<class CNezavrsniZnak *> SETPTRNZ;

template<class _T>
     class TDinRekVarijabla: public COsnovnaKlasa // dinamicka rekurzivna varijabla
{
public:
     TDinRekVarijabla(): bPoznata(false) {}
     TDinRekVarijabla(const TDinRekVarijabla<class _T> &iz): bPoznata(iz.bPoznata)
     {
          Ignorirao=iz.Ignorirao;
          Var=iz.Var;
     }
```

```
    _T Var;
    bool bPoznata;
    SETPTRNZ Ignorirao;

    void Ispisi(ostream &out)
    {
        out << "|" << ((bPoznata)?"true":"false") << "|";

        SETPTRNZ::iterator it;
        for(it=Ignorirao.begin();it!=Ignorirao.end();it++)
            (*it)->Ispisi(out);

        out << "] ";
    }
};

//////////////////////////////////////////////////////////////////

typedef set<class CKonNeoProdukcija *> SETPTRKNPROD;

class CNezavrsniZnak: public CZnakSaImenom
{
public:
    CNezavrsniZnak(const string &ime, int bratrib): CZnakSaImenom(ime,bratrib) { }

    CNezavrsniZnak(const CNezavrsniZnak &nz):
        CZnakSaImenom(nz.Ime,nz.BrAtrib), SkPtrProd(nz.SkPtrProd)
    {
        Prazan=nz.Prazan;
        First=nz.First;
    }

    TDinRekVarijabla<bool> Prazan;
    TDinRekVarijabla<SETPTRZZ> First;

    SETPTRKNPROD SkPtrProd;

    virtual void Ispisi(ostream &out, bool bSve=false)
    {
        CZnakSaImenom::Ispisi(out,bSve);

        if(bSve)
        {
            out << "[" << ((Prazan.Var)?"prazan":"neprazan");
            Prazan.Ispisi(out);

            out << "[";
            SETPTRZZ::iterator it;
            for(it=First.Var.begin();it!=First.Var.end();it++)
                (*it)->Ispisi(out);

            First.Ispisi(out);

            out << endl;
        }
    }
};

typedef set<CNezavrsniZnak> SETNZ;

//////////////////////////////////////////////////////////////////

class CAkcijskiZnak: public CZnakSaImenom
{
public:
    CAkcijskiZnak(const string &ime, int bratrib, CAkcija *pakc):
      CZnakSaImenom(ime,bratrib), pAkc(pakc) { }

    CAkcijskiZnak(const CAkcijskiZnak &az):
      CZnakSaImenom(az.Ime,az.BrAtrib), pAkc(az.pAkc) { }
```

```
    CAkcija *pAkc;
};

typedef set<CAkcijskiZnak> SETAZ;

//////////////////////////////////////////////////////////////////////

#endif
```

9.3.3. ZAkcije.h

```
//
//      Akcije
//

#if !defined(ZAKCIJE_H)
#define ZAKCIJE_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Greske.h"

//////////////////////////////////////////////////////////////////////

class CAkcija: public COsnovnaKlasa
{
public:
    CAkcija(string ime): bKoristiSe(false), Ime(ime) { }
    CAkcija(const CAkcija &akc): bKoristiSe(akc.bKoristiSe), Ime(akc.Ime) { }

    void Izvrsi() { };

    bool bKoristiSe;
    string Ime;

    friend const bool operator<(const CAkcija & left, const CAkcija & right)
    {
        return ( left.Ime < right.Ime );
    }

protected:
    // todo: pointer na funkciju
};

//////////////////////////////////////////////////////////////////////

class CSkupAkcija: public COsnovnaKlasa
{
public:
    typedef map<string,CAkcija> MAPAKCIJA;

    CAkcija *Nadji(string &ime)
    {
        MAPAKCIJA::iterator it;

        it=MapAkc.find(ime);
        if(it==MapAkc.end()) return NULL;

        return &(it->second);
    }

    void Dodaj(CAkcija &akcija)
    {
        if( MapAkc.insert( MAPAKCIJA::value_type(akcija.Ime, akcija) ).second ==
false)
```

```
            GR << "Pokusaj dodavanja akcije koja vec postoji" << kraj;
    }

protected:
    MAPAKCIJA MapAkc;
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.3.4. ZDKA.h

```
//
//       Produkcijski DKA
//

#if !defined(ZDKA_H)
#define ZDKA_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ZKonNeoGr.h"

//////////////////////////////////////////////////////////////////////

class CLRStavka: public COsnovnaKlasa
{
public:
    CLRStavka(CKonNeoProdukcija *pprod, int tocpoz, SETPTRZZ &poslje):
        pProd(pprod), TocPoz(tocpoz), Poslje(poslje) {}

    CLRStavka(const CLRStavka &st):
        pProd(st.pProd), TocPoz(st.TocPoz), Poslje(st.Poslje) {}

    CKonNeoProdukcija *pProd;
    int TocPoz;

    SETPTRZZ Poslje;

    friend const bool operator < (const CLRStavka &left, const CLRStavka &right)
    {
        if( left.pProd < right.pProd ) return true;
        if( left.pProd > right.pProd ) return false;

        if( left.TocPoz < right.TocPoz ) return true;
        if( left.TocPoz > right.TocPoz ) return false;

        if( left.Poslje < right.Poslje ) return true;
        if( left.Poslje > right.Poslje ) return false;

        return false;
    }

    void Ispisi(ostream &out)
    {
        pProd->LStrana->Ispisi(out);

        out << "-> ";

        int i;

        for(i=0;i<TocPoz;i++)
            pProd->DStrana[i]->Ispisi(out);
```

```
        out << ". ";

        for(;i<pProd->DStrana.size();i++)
            pProd->DStrana[i]->Ispisi(out);

        SETPTRZZ::iterator zzit;

        out << "  { ";
        for(zzit=Poslje.begin();zzit!=Poslje.end();zzit++)
            (*zzit)->Ispisi(out);

        out << "}" << endl;
    }
};

////////////////////////////////////////////////////////////////////

class CSljedecaAkcija: public COsnovnaKlasa
{
public:
    CSljedecaAkcija(class CZStanje *pstanje): bRedukcija(false) { Ptr.pStanje=pstanje;
}
    CSljedecaAkcija(CKonNeoProdukcija *pprod): bRedukcija(true) { Ptr.pProd=pprod; }
    CSljedecaAkcija(const CSljedecaAkcija &sakc):
        bRedukcija(sakc.bRedukcija) { Ptr=sakc.Ptr; }

    bool bRedukcija;

    union UPtrStPtrProd
    {
        class CZStanje *pStanje;
        CKonNeoProdukcija *pProd;
    }Ptr;
};

typedef map<CZavrsniZnak *, CSljedecaAkcija> PZZ2AKC;
typedef map<CNezavrsniZnak *, CZStanje * > PNZ2STANJE;
typedef set<CLRStavka> SETLRSTAV;

////////////////////////////////////////////////////////////////////

class CZStanje: public COsnovnaKlasa
{
public:
    CZStanje(): Indeks(-1) { }
    CZStanje(const CZStanje &zs): SkStavki(zs.SkStavki), PriNaZavZn(zs.PriNaZavZn),
        PriNaNezZn(zs.PriNaNezZn), Indeks(zs.Indeks) { }

    SETLRSTAV SkStavki;

    PZZ2AKC PriNaZavZn;
    PNZ2STANJE PriNaNezZn;

    int Indeks;

    friend const bool operator < (const CZStanje &left, const CZStanje &right)
    {
        return ( left.SkStavki < right.SkStavki );
    }

    void Ispisi(ostream &out, bool bSve=false)
    {
        SETLRSTAV::iterator it;

        out << "* stanje " << Indeks << " *" << endl;

        for(it=SkStavki.begin();it!=SkStavki.end();it++)
            it->Ispisi(out);

        if(bSve)
        {
            PZZ2AKC::iterator prchit;
```

```
            for(prchit=PriNaZavZn.begin();prchit!=PriNaZavZn.end();prchit++)
            {
                out << "--- ";
                prchit->first->Ispisi(out);
                out << "---> ";

                IspisiSljedAkc( out, prchit->second );
            }

            PNZ2STANJE::iterator pnzit;

            for(pnzit=PriNaNezZn.begin();pnzit!=PriNaNezZn.end();pnzit++)
            {
                out << "--- ";
                pnzit->first->Ispisi(out);
                out << " ---> stanje " << pnzit->second->Indeks << endl;
            }
        }

        out << endl;
    }

protected:
    void IspisiSljedAkc(ostream &out, CSljedecaAkcija &SA)
    {
        if(SA.bRedukcija)
        {
            out << "redukcija: ";
            SA.Ptr.pProd->Ispisi(out);
        }
        else
            out << "stanje " << SA.Ptr.pStanje->Indeks << endl;
    }
};

//////////////////////////////////////////////////////////////////////

class CZDKA: public COsnovnaKlasa
{
public:
    CZDKA(CKonNeoGramatika &gram): pGram(&gram), BrStanja(0)
    {
        // stavi pocetno stanje u SkStanja

        SETNZ::const_iterator nzit;

        nzit = pGram->SkNez.find( *(pGram->pPocNez) );

        SETPTRKNPROD::const_iterator prit = nzit->SkPtrProd.begin();

        if( prit == nzit->SkPtrProd.end() )
            GR << "nema pocetnih produkcija" << kraj;

        pPocSt = new CZStanje;
        pPocSt->Indeks = BrStanja++;
        SETPTRZZ Prazno;
        Prazno.insert( &(pGram->ZnKrNiza) );

        do {
            DodajStavku( pPocSt, *prit, 0, Prazno );
            prit++;
        } while( prit != nzit->SkPtrProd.end() );

        SkStanja.insert(pPocSt);

        pTrSt = pPocSt;
    }

    ~CZDKA()
    {
        SETPTRZST::iterator it;
```

```
        for(it=SkStanja.begin();it!=SkStanja.end();it++)
            delete *it;
    }

    CKonNeoGramatika *pGram;

    CZStanje *pTrSt;
    CZStanje *pPocSt;

    bool NapraviPrijelaz(CZavrsniZnak *pZZ, CKonNeoProdukcija **pProd)
    {
        PZZ2AKC::iterator it;

        // jesmo li vec obradjivali akciju za taj znak
        it=pTrSt->PriNaZavZn.find(pZZ);

        if(it==pTrSt->PriNaZavZn.end()) // ako nismo odredi koja je akcija
        {
            // ..pravimo novo stanje
            //  NAPOMENA: novo stanje ce biti 'sirovo' stanje koje se brzo kreira
znaci
            //   ne provjerava se reduciraj/reduciraj i reduciraj/pomakni
proturijecje
            CZStanje *pNovoSt = NovoStanje(pTrSt,pZZ);

            if(pNovoSt==NULL) // novo stanje je prazno?
            {
                // jel to redukcija?
                CKonNeoProdukcija *pTmProd = KojaRedukcija(pTrSt,pZZ);

                if(pTmProd==NULL) // ako nema redukcije..
                {
                    return false; //  ne postoji nikakav prijelaz za taj znak!
                }
                else // inace biljezimo redukciju
                    it = pTrSt->PriNaZavZn.insert(
                        PZZ2AKC::value_type( pZZ, CSljedecaAkcija( pTmProd ) )
).first;

                // NAPOMENA: Prvo provjeravamo novo stanje a tek onda redukciju zbog
toga
                //  jer pravilo kaze da u pomakni/reduciraj konfliktu prvo ide
pomakni!
            }
            else
            {
                SETPTRZST::iterator stit;

                // postoji li to stanje vec u skupu stanja
                stit=SkStanja.find(pNovoSt);
                if(stit==SkStanja.end()) // ne postoji..
                {
                    if(!pGram->bNejednoznacnaGram) ProvjeriStanje( pNovoSt );
                    stit=SkStanja.insert( pNovoSt ).first; // ..dodaj
                }
                else // inace obrisi
                {
                    delete pNovoSt;
                    BrStanja--;
                }

                it = pTrSt->PriNaZavZn.insert(
                    PZZ2AKC::value_type( pZZ, CSljedecaAkcija( *stit ) ) ).first;
            }
        }
/*
   ovdje treba prouciti sta je sta

        if(it==pTrSt->PriNaZavZn.end()) // ako nismo odredi koja je akcija
        {
            // jel to redukcija?
```

```
            CKonNeoProdukcija *pTmProd = KojaRedukcija(pTrSt,pZZ);

            if(pTmProd==NULL) // ako nema redukcije..
            {
                // ..pravimo novo stanje
                //  NAPOMENA: novo stanje ce biti 'sirovo' stanje koje se brzo kreira
znaci
                //   ne provjerava se reduciraj/reduciraj i reduciraj/pomakni
proturijecje
                CZStanje *pNovoSt = NovoStanje(pTrSt,pZZ);

                if(pNovoSt==NULL) // novo stanje je prazno?
                    return false; //  ne postoji nikakav prijelaz za taj znak!

                SETPTRZST::iterator stit;

                // postoji li to stanje vec u skupu stanja
                stit=SkStanja.find(pNovoSt);
                if(stit==SkStanja.end()) // ne postoji..
                {
                    ProvjeriStanje( pNovoSt );
                    stit=SkStanja.insert( pNovoSt ).first; // ..dodaj
                }
                else // inace obrisi
                {
                    delete pNovoSt;
                    BrStanja--;
                }

                it = pTrSt->PriNaZavZn.insert(
                    PZZ2AKC::value_type( pZZ, CSljedecaAkcija( *stit ) ) ).first;
            }
            else // inace biljezimo redukciju
                it = pTrSt->PriNaZavZn.insert(
                    PZZ2AKC::value_type( pZZ, CSljedecaAkcija( pTmProd ) ) ).first;
        }
*/
        if(!it->second.bRedukcija) // ako akcija nije redukcija..
        {
            // ..onda je prijelaz u sljedece stanje
            pTrSt=it->second.Ptr.pStanje;
            *pProd=NULL; // nema redukcije
        }
        else
        {
            // inace je redukcija
            *pProd=it->second.Ptr.pProd;
        }

        return true;
    }

    void NapraviRedukciju(CZStanje *pSt, CNezavrsniZnak *pNZ)
    {
        pTrSt=pSt;

        PNZ2STANJE::iterator it;

        // jesmo li vec obradjivali prijelaz za taj nezavrsni znak
        it=pTrSt->PriNaNezZn.find(pNZ);

        if(it==pTrSt->PriNaNezZn.end()) // ako nismo, odredi koji je to prijelaz
        {
            // ..pravimo novo stanje
            //  NAPOMENA: novo stanje ce biti 'sirovo' stanje koje se brzo kreira
znaci
            //   ne provjerava se reduciraj/reduciraj i reduciraj/pomakni
proturijecje
            CZStanje *pNovoSt = NovoStanje(pTrSt,pNZ);

            IFDEBUG if(pNovoSt==NULL)
```

```
// ne smije nikako biti NULL - neka greska u parseru!

                GR_INT_CLASS << "Ne mogu napraviti redukciju (pogresno stanje?)" <<
kraj;

            SETPTRZST::iterator stit;

            // postoji li to stanje vec u skupu stanja
            stit=SkStanja.find(pNovoSt);
            if(stit==SkStanja.end()) // ne postoji..
            {
                if(!pGram->bNejednoznacnaGram) ProvjeriStanje( pNovoSt );
                stit=SkStanja.insert( pNovoSt ).first; // ..dodaj
            }
            else
            {
                delete pNovoSt;
                BrStanja--;
            }

            it = pTrSt->PriNaNezZn.insert(
                PNZ2STANJE::value_type( pNZ, *stit ) ).first;
        }

        pTrSt=it->second;
    }

    void IzgradiCijeliDKA()
    {
        CZStanje *pStaro=pTrSt;

        CKonNeoProdukcija *pBezVeze;

        SETPTRZST::iterator stanjeit;
        SETLRSTAV::iterator stavkait;

        set<int> Napravljeno;

        do {
            for(stanjeit=SkStanja.begin();stanjeit!=SkStanja.end();stanjeit++)
            {
                if( Napravljeno.insert((*stanjeit)->Indeks).second == false )
                    continue; // vec je napravljeno

                for(stavkait=(*stanjeit)->SkStavki.begin();
                        stavkait!=(*stanjeit)->SkStavki.end(); stavkait++)
                {
                    pTrSt=*stanjeit;  // ovo mora biti tu jer se mijenja u
                                      //  NapraviPrijelaz i ostalim funkc.

                    if( stavkait->pProd->DStrana.size() > stavkait->TocPoz )
                    {
                        CZnakGramatike *pZnak =
                            stavkait->pProd->DStrana[ stavkait->TocPoz ];

                        CZavrsniZnak *pZZ = dynamic_cast<CZavrsniZnak *>(pZnak);

                        if(pZZ!=NULL)
                        {
                            if(!NapraviPrijelaz( pZZ, &pBezVeze ))
                                GR_INT_CLASS << "ne moze napraviti prijelaz" <<
kraj;
                        }
                        else
                        {
                            CNezavrsniZnak *pNZ = dynamic_cast<CNezavrsniZnak
*>(pZnak);

                            if(pNZ!=NULL)
                                NapraviRedukciju( *stanjeit, pNZ );
                            else
```

```
                                    GR_INT_CLASS << "tocka na neispravnom mjestu" <<
kraj;
                            }
                        }
                        else if( stavkait->pProd->DStrana.size() == stavkait->TocPoz )
                        {
                            SETPTRZZ::iterator zzit = stavkait->Poslje.begin();

                            for(;zzit!=stavkait->Poslje.end();zzit++)
                            {
                                pTrSt=*stanjeit;  // ovo mora biti tu jer se mijenja u
                                                  //  NapraviPrijelaz i ostalim
funkc.

                                if(!NapraviPrijelaz( (*zzit), &pBezVeze ))
                                    GR_INT_CLASS << "ne moze napraviti prijelaz!!!" <<
kraj;
                            }
                        }
                    }
                }
            } while( Napravljeno.size() < SkStanja.size() );

            pTrSt=pStaro;
        }

        void Ispisi(ostream &out, bool bSve=false)
        {
            out << "*** CZDKA - broj stanja: " << SkStanja.size() << " ***" << endl;

            my_vect<CZStanje> TmStanja;

            SETPTRZST::iterator sit;

            for(sit=SkStanja.begin();sit!=SkStanja.end();sit++)
                TmStanja.push_back(**sit);

            sort( TmStanja.begin(), TmStanja.end(), UsporediIndeks );

            my_vect<CZStanje>::iterator vit;

            for(vit=TmStanja.begin();vit!=TmStanja.end();vit++)
                vit->Ispisi(out,bSve);
        }

protected:
        struct SZStanjeLess : public binary_function<CZStanje *, CZStanje *, bool>
        {
            bool operator()(const CZStanje *x, const CZStanje *y) const
            {
                return ( *x < *y );
            }
        };

        typedef  set<CZStanje *, SZStanjeLess> SETPTRZST;

        SETPTRZST SkStanja;

        int BrStanja;

        static bool UsporediIndeks(CZStanje &left, CZStanje &right)
        {
            return left.Indeks < right.Indeks;
        }

        // provjerava da li se u stanju *pSt za zavrsni znak *pZZ moze
        //  moze primjeniti neka redukcija: vrati ptr_na_pr ako moze, inace NULL
        CKonNeoProdukcija *KojaRedukcija(CZStanje *pSt, CZavrsniZnak *pZZ)
        {
            CKonNeoProdukcija *pTmProd=NULL;

            SETLRSTAV::iterator it;
```

```
        for(it=pSt->SkStavki.begin();it!=pSt->SkStavki.end();it++)
            if( it->pProd->DStrana.size() == it->TocPoz )
                if( it->Poslje.find( pZZ ) != it->Poslje.end() )
                {
                    if(!pGram->bNejednoznacnaGram)
                        return it->pProd;
                    else // ovo razrijesava reduciraj/reduciraj proturijecije
                        if( pTmProd==NULL ||
                            it->pProd->RedBr < pTmProd->RedBr )
                        {
                            pTmProd = it->pProd;
                        }
                }

        return pTmProd;
    }

    // pokusava napraviti novo stanje za prijelaz zbog znaka *pZnak, ako ne moze onda
NULL
    //  NAPOMENA: novo stanje ce biti 'sirovo' stanje koje se brzo kreira znaci
    //   ne provjerava se reduciraj/reduciraj i reduciraj/pomakni proturijecje
    CZStanje *NovoStanje(CZStanje *pSt, CZnakGramatike *pZnak)
    {
        CZStanje *pNovo;

        pNovo=new CZStanje;
        pNovo->Indeks = BrStanja++;

        SETLRSTAV::iterator it;


        for(it=pSt->SkStavki.begin();it!=pSt->SkStavki.end();it++)
            if( it->pProd->DStrana.size() > it->TocPoz )
                if( it->pProd->DStrana[ it->TocPoz ] == pZnak )
                    DodajStavku( pNovo, it->pProd, it->TocPoz+1, it->Poslje );

        if( pNovo->SkStavki.size() == 0 ) // ako je prazno stanje
        {
            delete pNovo;
            BrStanja--;
            return NULL;
        }

        return pNovo;
    }

    pair< SETLRSTAV::iterator, bool >
        NadjiSlicnuStavku(CZStanje *pStanje, CKonNeoProdukcija *pProd, int TocPoz)
    {
    // todo: ovo se moze ubrzati

        SETLRSTAV::iterator stit;

        // idemo kroz sve stavke
        for(stit=pStanje->SkStavki.begin();stit!=pStanje->SkStavki.end();stit++)
            if( stit->pProd==pProd && stit->TocPoz==TocPoz)
                return pair< SETLRSTAV::iterator, bool >( stit, true );

        return pair< SETLRSTAV::iterator, bool >( stit, false );
    }

    void DodajStavku(CZStanje *pStanje, CKonNeoProdukcija *pProd, int TocPoz, SETPTRZZ
&Poslje)
    {
        TocPoz = PozicijaBezAkcijskih(pProd,TocPoz);

        pair< SETLRSTAV::iterator, bool > rez;

        rez = NadjiSlicnuStavku( pStanje, pProd, TocPoz );

        if(rez.second) // ako postoji slicna stavka
```

```
        {
            int stara_vel = rez.first->Poslje.size();

            rez.first->Poslje += Poslje;

            if( rez.first->Poslje.size()==stara_vel ) // ako nismo nista dodali
                rez.second = false; // nemoj nista kasnije raditi
        }
        else
        {
            rez = pStanje->SkStavki.insert( CLRStavka(pProd, TocPoz, Poslje) );

            GR_IF(rez.second==false);
        }

        if(rez.second)    // ako smo nesto novo dodali
            ProsiriAkoJeNezavrsni( pStanje, &(*(rez.first)) );
    }

    int PozicijaBezAkcijskih(CKonNeoProdukcija *pProd, int TocPoz)
    {
        for(; pProd->DStrana.size() > TocPoz ; TocPoz++)
            if( dynamic_cast<CAkcijskiZnak *>( pProd->DStrana[TocPoz] ) == NULL)
                return TocPoz;

        return TocPoz;
    }

    void ProsiriAkoJeNezavrsni(CZStanje *pStanje, CLRStavka *pStavka)
    {
        if(pStavka->pProd->DStrana.size() <= pStavka->TocPoz) return;

        CNezavrsniZnak *pNZ = dynamic_cast<CNezavrsniZnak *>(
            pStavka->pProd->DStrana[ pStavka->TocPoz ] );

        if(pNZ!=NULL)
        {
            SETPTRZZ First;

            // racunaj first=FIRST(beta) (vidi skriptu str.164)
            // dodaj na first Poslje ako je beta prazan

            if( pGram->PuniFirstVratiPrazan(
                     pStavka->pProd->DStrana.begin() + pStavka->TocPoz + 1,
                     pStavka->pProd->DStrana.end(),
                     First ) == true )
                First+=pStavka->Poslje;

            // dodaje sve stavke koje proizlaze iz pNZ
            SETNZ::const_iterator nzit;

            nzit = pGram->SkNez.find( *pNZ );

            SETPTRKNPROD::const_iterator prit = nzit->SkPtrProd.begin();

            if( prit == nzit->SkPtrProd.end() )
                GR << "nijedna produkcija ne proizlazi iz nezavrsnog znaka" << kraj;

            do {
                DodajStavku( pStanje, *prit, 0, First );
                prit++;
            } while( prit != nzit->SkPtrProd.end() );
        }
    }

    void ProvjeriStanje(CZStanje *pSt)
    {
        SETPTRZZ SkSvihPosljRedu;

        SETLRSTAV::iterator stit;

        for(stit=pSt->SkStavki.begin();stit!=pSt->SkStavki.end();stit++)
```

```
        {
            IFDEBUG if( stit->pProd->DStrana.size() < stit->TocPoz )
                GR_INT_CLASS << "greska u negdje u stanjima" << kraj;

            // provjeri..
            if( stit->pProd->DStrana.size() == stit->TocPoz )
                if( DodajSet(SkSvihPosljRedu, stit->Poslje) != stit->Poslje.size() )
                { // ..reduciraj/reduciraj proturijecije!
                    stit->pProd->Ispisi(*cgreska,true);

                    GR << "pri obradi navedene produkcije je otkriveno " <<
                        "reduciraj/reduciraj proturijecije u gramatici" << kraj;
                }
        }

        for(stit=pSt->SkStavki.begin();stit!=pSt->SkStavki.end();stit++)
        {
            // provjeri..
            if( stit->pProd->DStrana.size() > stit->TocPoz )
            { // ..reduciraj/pomakni proturijecije:
                SETPTRZZ first = pGram->KojiFirst(
                    stit->pProd->DStrana[stit->TocPoz] );

                if( DodajSet(first,SkSvihPosljRedu) != SkSvihPosljRedu.size() )
                {
                    stit->pProd->Ispisi(*cgreska,true);
                    pSt->Ispisi(*cgreska);

                    GR << "pri obradi navedene produkcije je otkriveno " <<
                        "reduciraj/pomakni proturijecije u gramatici" << kraj;
                }
            }
        }
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.3.5. ZKonNeoGr.h

```
//
//      Konteksno-neovisna gramatika
//

#if !defined(ZKONNEOGR_H)
#define ZKONNEOGR_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ZZnakovi.h"

#include "Greske.h"
#include "Globalno.h"

//////////////////////////////////////////////////////////////////////

class CPozicijaAtributa: public COsnovnaKlasa
{
public:
    CPozicijaAtributa(): PozZn(-1), PozAtr(-1) { }
    CPozicijaAtributa(const CPozicijaAtributa &pa):
        PozZn(pa.PozZn), PozAtr(pa.PozAtr) { }
    CPozicijaAtributa(string & str) { Pridruzi(str); }
```

```
    int PozZn;        // -1 -> greska, 0 -> znak na lijevoj strani, 1,2,3 -> znak na
desnoj strani
    int PozAtr;       // pocevsi od nule

    void Pridruzi(string &str)
    {
        string::size_type tpoz;

        tpoz=str.find('.');
        if(tpoz==string::npos)
            GR << "Nema tocke u poziciji atributa" << kraj;

        PozZn=atol( str.c_str() );

        if(PozZn==0 && str[0]!='0')
            GR << "Pozicija znaka nije broj" << kraj;

        PozAtr=atol( str.c_str()+tpoz+1 );

        if(PozAtr==0 && str[tpoz+1]!='0')
            GR << "Pozicija atributa nije broj" << kraj;
    }

    void Ispisi(ostream &out, bool bSve=false)
    {
        out << PozZn << "." << PozAtr << " ";
    }
};

typedef my_vect<CPozicijaAtributa> NIZPOZATR;

//////////////////////////////////////////////////////////////////////

class CPraviloPreslikavanja: public COsnovnaKlasa
{
public:
    CPraviloPreslikavanja(): PozAkt(-1) { }
    CPraviloPreslikavanja(const CPraviloPreslikavanja &pp):
        Iz(pp.Iz), U(pp.U), PozAkt(pp.PozAkt) { }

    CPozicijaAtributa Iz;
    NIZPOZATR U;

    int PozAkt;       // poslje kojeg znaka na desnoj strani ce se aktivirati ( 0 ->
prije svih )

    void Ispisi(ostream &out, bool bSve=false)
    {
        if(bSve)
            out << "\taktivacija(" << PozAkt << ")\t";
        else
            out << "\t\t";

        Iz.Ispisi(out,bSve);

        out << "=> ";

        NIZPOZATR::iterator it;

        for(it=U.begin();it!=U.end();it++)
            it->Ispisi(out,bSve);

        out << endl;
    }
};

//////////////////////////////////////////////////////////////////////

class CKonNeoProdukcija: public COsnovnaKlasa
{
public:
```

```
    CKonNeoProdukcija(int redbr): RedBr(redbr) { }
    CKonNeoProdukcija(const CKonNeoProdukcija &kp):
         LStrana(kp.LStrana), DStrana(kp.DStrana),
              SkPrPres(kp.SkPrPres), RedBr(kp.RedBr) { }

    CNezavrsniZnak *LStrana;
    NIZPTRZNGR DStrana;

    typedef my_vect<CPraviloPreslikavanja> NIZPRPRES;

    NIZPRPRES SkPrPres;

    void Ispisi(ostream &out, bool bSve=false)
    {
         LStrana->Ispisi(out);

         out << "-> ";

         NIZPTRZNGR::iterator it;

         if(DStrana.size()==0)
              out << "eps ";
         else
              for(it=DStrana.begin();it!=DStrana.end();it++)
                   (*it)->Ispisi(out);

         if(bSve)
         {
              out << "\n";

              NIZPRPRES::iterator it;

              for(it=SkPrPres.begin();it!=SkPrPres.end();it++)
                   it->Ispisi(out,bSve);
         }

         out << endl;
    }

    // ovo je novo dodano

    int RedBr; // 0 -> najveci prioritet

    friend const bool operator<(const CKonNeoProdukcija & left, const
CKonNeoProdukcija & right)
    {
         if( left.RedBr < right.RedBr ) return true;

         return false;
    }

/*
    friend const bool operator<(const CKonNeoProdukcija & left, const
CKonNeoProdukcija & right)
    {
         if( *(left.LStrana) < *(right.LStrana) ) return true;
         if( *(right.LStrana) < *(left.LStrana) ) return false;

         NIZPTRZNGR::const_iterator itright=right.DStrana.begin();
         NIZPTRZNGR::const_iterator itleft=left.DStrana.begin();

         for(; itleft!=left.DStrana.end() && itright!=right.DStrana.end() ; itleft++,
itright++)
         {
              if( *itleft < *itright ) return true;
              if( *itright < *itleft ) return false;
         }

         if( left.DStrana.size() < right.DStrana.size() ) return true;
         if( left.DStrana.size() > right.DStrana.size() ) return false;

         return false;
```

```
    }
*/
};

typedef set<CKonNeoProdukcija> SETKNPROD;

//////////////////////////////////////////////////////////////////

class CZDatoteka: public COsnovnaKlasa
{
public:
    CZDatoteka();
    CZDatoteka(const string &ime) { Otvori(ime); }
    ~CZDatoteka() { if(InStr.is_open()) Zatvori(); }

    void Otvori(const string &ime)
    {
        InStr.open(ime.c_str(),ios_base::binary);
        if(InStr.fail())
            GR << "Ne mogu otvoriti datoteku " << ime << " za citanje" << kraj;

        NoviRed();
    }

    void Zatvori()
    {
        InStr.close();
    }

    bool NoviRed(bool bFatalErr=true)
    {
        InStr >> ws;
        InStr.getline(Red,255);
        if(InStr.fail())
            if(bFatalErr) GR << "Neocekivani kraj datoteke ili predugacak red" <<
kraj;
            else return false;

        TrPoz=0;

        return true;
    }

    bool UcitajStr(string &str, bool bFatalErr=true)
    {
        char tm[256];
        int i;

        for(i=0; Red[TrPoz] && !isspace(Red[TrPoz]) ;i++,TrPoz++)
            tm[i]=Red[TrPoz];

        tm[i]=0;

        for(; isspace(Red[TrPoz]) && Red[TrPoz] ;TrPoz++) ;

        str.assign(tm);

        if(str=="")
            if(bFatalErr) GR << "Nema vise nista u trenutnom redu" << kraj;
            else return false;

        return true;
    }

    bool UcitajInt(int &i, bool bFatalErr=true)
    {
        string str;

        if(UcitajStr(str,bFatalErr)==false) return false;

        i=atoi(str.c_str());
```

```
        if(i==0 && str[0]!='0')
            if(bFatalErr) GR << "Sljedeci token u datoteci nije cijeli broj" << kraj;
            else return false;

        return true;
    }

protected:
    ifstream InStr;
    char Red[256];
    int TrPoz;
};

//////////////////////////////////////////////////////////////////

class CKonNeoGramatika: public COsnovnaKlasa
{
public:
    CKonNeoGramatika(const string &ime_dat, CSkupAkcija &sk_akc):
        pSkAkc(&sk_akc), ZnKrNiza(ZNKRNIZA,""), bNejednoznacnaGram(false)
    {
        CZDatoteka zd(ime_dat);

        Ucitaj(zd);
    }

    void Ucitaj(CZDatoteka &zd)
    {
        string str;
        int bratr;
        CNezavrsniZnak *tmnez;

        zd.UcitajStr(str);

        if(str!="[nezavrsni]")
            GR << "Neispravan format datoteke" << kraj;

        for(;;) // nezavrsni
        {
            zd.NoviRed();


            zd.UcitajStr(str);
            if(str=="[zavrsni]") break;

            zd.UcitajInt(bratr);

            if( SkNez.insert(CNezavrsniZnak(str,bratr)).second == false )
                GR << "Nezavrsni " << str << " vec postoji u datoteci sa gramatikom"
<< kraj;
        }

        for(;;) // zavrsni
        {
            zd.NoviRed();

            zd.UcitajStr(str);
            if(str=="[akcijski]") break;

            pair<SETZZ::iterator,bool> rez = SkZav.insert(CZavrsniZnak(str));
            if( rez.second == false )
                GR << "Zavrsni " << str << " vec postoji u datoteci sa gramatikom" <<
kraj;
            else {
                if( str == "IDN" )
                    (rez.first)->Tip = IDN;
                else if( str == "ASM" )
                    (rez.first)->Tip = ASM;
                else if( str == "STR" )
                    (rez.first)->Tip = STR;
                else if( str == "KARAKTER" )
                    (rez.first)->Tip = KARAKTER;
```

```
                else if( str == "DEK" )
                    (rez.first)->Tip = DEK;
                else if( str == "REALNI" )
                    (rez.first)->Tip = REALNI;
                else if( str == "HEX" )
                    (rez.first)->Tip = HEX;
                else if( str == "OKT" )
                    (rez.first)->Tip = OKT;
                else if( str == "BIN" )
                    (rez.first)->Tip = BIN;
                else
                    (rez.first)->Tip = KROS;
            }
        }

        for(;;) // akcijski
        {
            zd.NoviRed();

            zd.UcitajStr(str);
            if(str=="[pocetni]") break;

            zd.UcitajInt(bratr);

            CAkcija *pakc=pSkAkc->Nadji(str);

            if(pakc==NULL)
                GR << "Ne postoji akcija " << str << " " << kraj;

            if(pakc->bKoristiSe)
                GR << "Akcijski znak " << str << " visestruko naveden u ulaznoj
datoteci" << kraj;

            pakc->bKoristiSe=true;

            if( SkAkc.insert(CAkcijskiZnak(str,bratr,pakc)).second == false)
                GR << "Akcijski " << str << " vec postoji u datoteci sa gramatikom"
<< kraj;
        }

        // pocetni

        zd.NoviRed();

        zd.UcitajStr(str);

        SETNZ::iterator it;

        it=SkNez.find( CNezavrsniZnak(str,0) );

        if(it==SkNez.end())
            GR << "Pocetni znak u datoteci mora biti jedan od nezavrsnih" << kraj;

        pPocNez=&(*it);

        // produkcije
        zd.NoviRed();

        zd.UcitajStr(str);
        if(str!="[produkcije]")
            GR << "Neispravan format datoteke" << kraj;

        zd.NoviRed();

        zd.UcitajStr(str);    // nezavrsni znak na ljevoj strani

        bool bKrajLeksike=false;

        for(bool bNijeKraj=true;bNijeKraj;) // jedna po jedna produkcija
        {
            it=SkNez.find( CNezavrsniZnak(str,0) );
```

```
            if(it==SkNez.end())
                GR << "Produkcija u datoteci mora pocinjati sa postojecim nezavrsnim"
<< kraj;

            tmnez=&(*it);

            zd.UcitajStr(str);

            if(str!="->")
                GR << "Neispravan format datoteke (nedostaje ->)" << kraj;

            for(bool bNijeNovaPr=true; bNijeKraj & bNijeNovaPr ;)// svaka produkcija
moze imati vise podprodukcija u datoteci
            {
                CKonNeoProdukcija tmprod( SkProd.size() ); // koje su ustvari
odvojene inacice u memoriji

                tmprod.LStrana=tmnez; // lijeva strana produkcije

                zd.UcitajStr(str);

                if(str!="eps")
                    for(;;) // znak po znak desne strane produkcije
                    {
                        switch(str[0])
                        {
                        case '<':    // nezavrsni
                            if(str.length()<3) goto l_zavrsni;

                            it=SkNez.find( CNezavrsniZnak(str,0) );
                            if(it==SkNez.end())
                                GR << "Nepostojeci nezavrsni znak " << str << " u
produkciji" << kraj;
                            tmprod.DStrana.push_back( &(*it) );
                            break;

                        case '{':    // akcijski
                            {
                                SETAZ::iterator it=SkAkc.find(
CAkcijskiZnak(str,0,NULL) );
                                if(it==SkAkc.end())
                                    GR << "Nepostojeci akcijski znak " << str << "
u produkciji" << kraj;
                                tmprod.DStrana.push_back( &(*it) );
                            }
                            break;

                        default: // inace je zavrsni
l_zavrsni:                          if(bKrajLeksike)
                                    GR << "nakon [gotova_leksika] ne mogu ici zavrsni
znakovi" << kraj;

                            {
                                SETZZ::iterator it=SkZav.find( CZavrsniZnak(str) );

                                if(it==SkZav.end())
                                {

                                    tmprod.Ispisi(*cgreska);

                                    GR << "Nepostojeci zavrsni znak " << str <<
                                        " u prethodnoj produkciji" << kraj;
                                }

                                tmprod.DStrana.push_back( &(*it) );
                            }
                        }

                        if(!zd.UcitajStr(str,false)) break;
                    }
```

```
                for(; bNijeKraj & bNijeNovaPr ;)    // ucitavamo pravila
preslikavanja
                {
                    zd.NoviRed();
                    zd.UcitajStr(str);
                    if(str=="->")
                        break;
                    else if(str=="[kraj]") {
                        bNijeKraj=false;
                        break;
                    } else if(str=="[gotova_leksika]") {
                        bKrajLeksike=true;
                        bNijeNovaPr=false;
                        zd.NoviRed();
                        zd.UcitajStr(str);
                        break;
                    } else if(str[0]=='<') {
                        bNijeNovaPr=false;
                        break;
                    }

                    // pojedino pravilo preslikavanja

                    CPraviloPreslikavanja tm_ppres;

                    tm_ppres.Iz.Pridruzi(str);

                    zd.UcitajStr(str);

                    if(str!="=>")
                        GR << "Neispravan format datoteke (nedostaje =>)" << kraj;

                    zd.UcitajStr(str);

                    for(;;)
                    {
                        tm_ppres.U.push_back( CPozicijaAtributa(str) );

                        if(!zd.UcitajStr(str,false)) break;

                        if(str=="[kraj]") { bNijeKraj=false; break; }
                    }

                    tmprod.SkPrPres.push_back( tm_ppres );
                }

                SETKNPROD::iterator prit;

                prit=SkProd.insert( tmprod ).first;

                tmnez->SkPtrProd.insert( &(*prit) );
            }
        }
    }

    void Ispisi(ostream &out, bool bSve=false)
    {
        out << "[nezavrsni]" << endl;

        SETNZ::iterator nit;

        for(nit=SkNez.begin();nit!=SkNez.end();nit++)
            nit->Ispisi(out,bSve);

        out << "\n\n" << "[zavrsni]" << endl;

        SETZZ::iterator zit;

        for(zit=SkZav.begin();zit!=SkZav.end();zit++)
            zit->Ispisi(out,bSve);

        out << "\n\n" << "[akcijski]" << endl;
```

```
    SETAZ::iterator ait;

    for(ait=SkAkc.begin();ait!=SkAkc.end();ait++)
        ait->Ispisi(out,bSve);

    out << "\n\n" << "[pocetni]" << endl;

    pPocNez->Ispisi(out);

    out << "\n\n" << "[produkcije]" << endl;

    SETKNPROD::iterator prit;

    for(prit=SkProd.begin();prit!=SkProd.end();prit++)
        prit->Ispisi(out,bSve);

    out << "[kraj]" << endl;
}

CZavrsniZnak *NadjiZavrsni(TIPTOKENA tip, const string &ime)
{
    static CZavrsniZnak *m[10];
    static bool fl=false;

    if(fl==false)
    {
        const char tok_str[][10]={"KROS", "IDN", "ASM", "STR",
            "KARAKTER", "DEK", "REALNI", "HEX", "OKT", "BIN" };
        int i;

        for(i=1;i<10;i++)
        {
            SETZZ::iterator it;
            it = SkZav.find( CZavrsniZnak(tok_str[i]) );
            if( it==SkZav.end() )
                GR << "nesto ne valja u NadjiZavrsni" << kraj;
            else
                m[i] = &(*it);
        }

        fl=true;
    }

    switch(tip)
    {
    case KROS:
        {
            SETZZ::iterator tit;
            tit = SkZav.find( CZavrsniZnak(ime) );
            if( tit==SkZav.end() )
                return NULL;
            else
                return &(*tit);
        }
    case ZNKRNIZA:
        return &ZnKrNiza;

    case IDN:
        return m[1];

    case ASM:
        return m[2];

    case STR:
        return m[3];

    case KARAKTER:
        return m[4];

    case DEK:
        return m[5];
```

```
        case REALNI:
            return m[6];

        case HEX:
            return m[7];

        case OKT:
            return m[8];

        case BIN:
            return m[9];

        default:
            return NULL;
        }
    }

    void IzracunajSvePraznoce()
    {
        SETNZ::iterator it;

        for(it=SkNez.begin();it!=SkNez.end();it++)
            IsPrazan( &(*it) );
    }

    bool IsPrazan(CZnakGramatike *pZnak)
    {
        bool rez;
        SETPTRNZ TmIgnoriraj,TmIgnorirao;

        rez = VratiPrazan(pZnak,TmIgnoriraj,TmIgnorirao);
        IFDEBUG if( TmIgnorirao.size() != 0 )
            GR_INT_CLASS << "pogreska u funkcijama za praznocu" << kraj;

        return rez;
    }

    void IzracunajFirstSkupove()
    {
        SETNZ::iterator it;

        for(it=SkNez.begin();it!=SkNez.end();it++)
            KojiFirst( &(*it) );
    }

    SETPTRZZ KojiFirst(CZnakGramatike *pZnak)
    {
        SETPTRZZ rez;
        SETPTRNZ TmIgnoriraj, TmIgnorirao;

        rez = VratiFirst( pZnak, TmIgnoriraj, TmIgnorirao );

        IFDEBUG if( TmIgnorirao.size() != 0 )
            GR_INT_CLASS << "pogreska u funkcijama za first" << kraj;

        return rez;
    }

    bool PuniFirstVratiPrazan(NIZPTRZNGR::iterator it, NIZPTRZNGR::iterator kraj,
        SETPTRZZ &First)
    {
        for(;it!=kraj;it++)
        {
            First += KojiFirst(*it);

            if( ! IsPrazan(*it) ) return false;
        }

        return true;
    }
```

```
    CSkupAkcija *pSkAkc;

    SETNZ SkNez;
    SETZZ SkZav;
    SETAZ SkAkc;
    SETKNPROD SkProd;

    CNezavrsniZnak *pPocNez;

    CZavrsniZnak ZnKrNiza;  // tj. _|_ (vidi skriptu)

    // ovo je novo dodano
    bool bNejednoznacnaGram;

protected:

    bool VratiPrazan(CZnakGramatike *pZnak, SETPTRNZ &Ignoriraj, SETPTRNZ &Ignorirao)
    {
        Ignorirao.clear();

        if( dynamic_cast<CZavrsniZnak *>(pZnak) != NULL ) return false;
        else if( dynamic_cast<CAkcijskiZnak *>(pZnak) != NULL ) return true;
        else
        { // inace je sigurno nezavrsni
            CNezavrsniZnak *pNZ = dynamic_cast<CNezavrsniZnak *>(pZnak);

            if( ! pNZ->Prazan.bPoznata ||
                ! Podskup( pNZ->Prazan.Ignorirao, Ignoriraj ) )
            { // inace je moramo izracunati
                IzracunajPrazan(pNZ,Ignoriraj);
            }

            Ignorirao += pNZ->Prazan.Ignorirao;
            return pNZ->Prazan.Var;
        }
    }

    void IzracunajPrazan( CNezavrsniZnak *pNZ, SETPTRNZ Ignoriraj )
    {
        pNZ->Prazan.bPoznata=true; // pocinjemo sa racunanjem
        pNZ->Prazan.Ignorirao.clear();

        Ignoriraj.insert( pNZ );   // ubuduce izbjegavaj i ovaj znak

        SETPTRNZ TmIgnorirao;

        // idi kroz sve produkcije..
        SETPTRKNPROD::iterator prit = pNZ->SkPtrProd.begin();

        if( prit == pNZ->SkPtrProd.end() )
            GR << "nijedna produkcija ne proizlazi iz nezavrsnog znaka" << kraj;

        do {
            if( IzracunajPrazan(
                     (*prit)->DStrana.begin(), (*prit)->DStrana.end(),
                     Ignoriraj, TmIgnorirao) )
            { // ako je neka produkcija prazna..
                pNZ->Prazan.Var=true;          // ..onda je i taj znak
                pNZ->Prazan.Ignorirao.clear(); // u to smo definitivno sigurni!
                return;
            }
            else // nesigurnost se povecava
                pNZ->Prazan.Ignorirao+=TmIgnorirao;

            prit++;
        } while( prit != pNZ->SkPtrProd.end() );

        // ako smo dosli do ovdje onda nismo sigurni da je prazan

        pNZ->Prazan.Var=false;
        pNZ->Prazan.Ignorirao.erase( pNZ );
    }
```

```
    bool IzracunajPrazan( NIZPTRZNGR::iterator it, NIZPTRZNGR::iterator kraj,
        SETPTRNZ &Ignoriraj, SETPTRNZ &Ignorirao )
    {
        bool bPrazan=true;    // pretpostavimo da je prazan
        Ignorirao.clear();

        SETPTRNZ TmIgnorirao;
        CNezavrsniZnak *pNZ;

        // idemo kroz sve znakove
        for(;it!=kraj;it++)
        {
            if( (pNZ=dynamic_cast<CNezavrsniZnak *>(*it)) != NULL )
                if( Ignoriraj.find(pNZ) != Ignoriraj.end() ) // taj znak se ignorira
                {
                    Ignorirao.insert(pNZ);
                    continue;
                }

            if( VratiPrazan( *it, Ignoriraj, TmIgnorirao) == false )
            {
                if( TmIgnorirao.size() == 0 )
                    return false;// sigurno nije prazan
                else {
                    Ignorirao+=TmIgnorirao;
                    bPrazan=false;
                }
            }
        }

        if( bPrazan && Ignorirao.size()==0 )
            return true; // sigurno je prazan

        return false;// vrati da je neprazan ali sa rezervom (Ignoriraj)
    }

    SETPTRZZ VratiFirst(CZnakGramatike *pZnak, SETPTRNZ &Ignoriraj, SETPTRNZ
&Ignorirao)
    {
        Ignorirao.clear();

        CZavrsniZnak *pZZ = dynamic_cast<CZavrsniZnak *>(pZnak);

        if( pZZ != NULL )
        {
            SETPTRZZ rez;
            rez.insert( pZZ );
            return rez;
        }
        else if( dynamic_cast<CAkcijskiZnak *>(pZnak) != NULL )
            return SETPTRZZ(); // bas nista :)
        else
        { // inace je sigurno nezavrsni
            CNezavrsniZnak *pNZ = dynamic_cast<CNezavrsniZnak *>(pZnak);

            if( ! pNZ->First.bPoznata ||
                ! Podskup( pNZ->First.Ignorirao, Ignoriraj ) )
            { // inace moramo izracunati
                IzracunajFirst(pNZ,Ignoriraj);
            }

            Ignorirao += pNZ->First.Ignorirao;
            return pNZ->First.Var;
        }
    }

    void IzracunajFirst( CNezavrsniZnak *pNZ, SETPTRNZ Ignoriraj )
    {
        pNZ->First.bPoznata=true;  // pocinjemo sa racunanjem
        pNZ->First.Ignorirao.clear();
```

```
        Ignoriraj.insert( pNZ );   // ubuduce izbjegavaj i ovaj znak

        SETPTRNZ TmIgnorirao;

        // idi kroz sve produkcije..
        SETPTRKNPROD::iterator prit = pNZ->SkPtrProd.begin();

        if( prit == pNZ->SkPtrProd.end() )
             GR << "nijedna produkcija ne proizlazi iz nezavrsnog znaka" << kraj;

        do {
             pNZ->First.Var += IzracunajFirst(
                      (*prit)->DStrana.begin(), (*prit)->DStrana.end(),
                      Ignoriraj, TmIgnorirao);

             pNZ->First.Ignorirao += TmIgnorirao;

             prit++;
        } while( prit != pNZ->SkPtrProd.end() );

        pNZ->First.Ignorirao.erase( pNZ );
    }

    SETPTRZZ IzracunajFirst( NIZPTRZNGR::iterator it, NIZPTRZNGR::iterator kraj,
        SETPTRNZ &Ignoriraj, SETPTRNZ &Ignorirao )
    {
        SETPTRZZ rez;

        Ignorirao.clear();

        SETPTRNZ TmIgnorirao;
        CNezavrsniZnak *pNZ;

        // idemo kroz sve znakove
        for(;it!=kraj;it++)
        {
             if( (pNZ=dynamic_cast<CNezavrsniZnak *>(*it)) != NULL &&
                  Ignoriraj.find(pNZ) != Ignoriraj.end() ) // taj znak se ignorira
             {
                  Ignorirao.insert(pNZ);
             }
             else
             {
                  rez += VratiFirst( *it, Ignoriraj, TmIgnorirao );
                  Ignorirao += TmIgnorirao;
             }

             if( ! IsPrazan(*it) ) break; // ako je neprazan prekidaj
        }

        return rez;
    }
};

//////////////////////////////////////////////////////////////////

#endif
```

9.4. Datoteke sa funkcijama za obradu grešaka

9.4.1. Greske.h

```
//
//       Globalne funkcije za obradu gresaka
//

#if !defined(GRESKE_H)
#define GRESKE_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

//////////////////////////////////////////////////////////////////////

extern stringstream *cgreska;

//////////////////////////////////////////////////////////////////////

#ifdef _DEBUG
     #define IFDEBUG if(1)
#else
     #define IFDEBUG if(0)
#endif

//////////////////////////////////////////////////////////////////////

char *Path2Name(char *path);


#define GR (*cgreska << "Greska: ")

#define GR_INT (*cgreska << "Greska u " << Path2Name(__FILE__) << "(" << __LINE__ <<
"): ")

#define GR_INT_CLASS (*cgreska << "Greska u " << Path2Name(__FILE__) << "(" <<
__LINE__ << ") [" << typeid(*this).name() << "]: ")

#define GR_IF(x) IFDEBUG if(x) GR_INT << #x << kraj

template<class E, class Tr> inline
     basic_ostream<E, Tr>& __cdecl kraj(basic_ostream<E, Tr>& O)
     {
          stringstream tm;
          tm << O.rdbuf() << " !\n";
          MessageBox(NULL,tm.str().c_str(),"Fatalna greska",MB_OK|MB_ICONEXCLAMATION);
          exit(0);
          return (O);
     }

//////////////////////////////////////////////////////////////////////

class COsnovnaKlasa
{
public:
     COsnovnaKlasa();
     ~COsnovnaKlasa();

     static int KolikoObjekata() { return BrObjekata; }

private:
     COsnovnaKlasa(const COsnovnaKlasa &o);

protected:
     static int BrObjekata;
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.4.2. Greske.cpp

```
#include "stdafx.h"

#include "Greske.h"

//////////////////////////////////////////////////////////////////////
stringstream moje_greske;
stringstream *cgreska=&moje_greske;

//////////////////////////////////////////////////////////////////////


char *Path2Name(char *path)
{
     int i;

     for(i=strlen(path)-1;i>=0;i--)
          if(path[i]=='\\') return path+i+1;

     return path;
}

//////////////////////////////////////////////////////////////////////

COsnovnaKlasa::COsnovnaKlasa()
{
     BrObjekata++;
}

COsnovnaKlasa::~COsnovnaKlasa()
{
     BrObjekata--;
}

COsnovnaKlasa::COsnovnaKlasa(const COsnovnaKlasa &o)
{
     BrObjekata++;
}

int COsnovnaKlasa::BrObjekata=0;

//////////////////////////////////////////////////////////////////////
```

9.5. Datoteke grafičkog sučelja

9.5.1. CZSWindow.h

```
//
//  CZSWindow
//

#if !defined(CZSWINDOW_H)
#define CZSWINDOW_H

//////////////////////////////////////////////////////////////////////
```

```
class CZSWindow
{
public:

     virtual ~CZSWindow()
     {
          DestroyWindow(m_hwnd);
     }

     HWND m_hwnd;

     WPARAM Radi()
     {
          MSG msg;

          while( GetMessage(&msg, NULL, 0, 0) )
          {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }

          return msg.wParam;
     }

protected:

     static LONG TmAdrCZSWindow;

     static LRESULT CALLBACK SetAdrWndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
          SetWindowLong(hwnd,0,TmAdrCZSWindow);

          TmAdrCZSWindow = 0;

          SetWindowLong(hwnd,GWL_WNDPROC,(LONG)IndirectWndProc);

          return IndirectWndProc(hwnd, iMsg, wParam, lParam);
     }

     static LRESULT CALLBACK IndirectWndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
     {
          CZSWindow *pWin = (CZSWindow *) GetWindowLong(hwnd,0) ;

          if( iMsg == WM_CREATE )
          {
               // buduci da se WM_CREATE salje za vrijeme CreateWindow()
               //  jos uvijek nije postavljen m_hwnd. Zbog toga ovo MORA
               //   biti ovdje, a m_hwnd se ne mora drugdje ni postavljati
               pWin->m_hwnd = hwnd;
          }

          return pWin->WndProc(hwnd,iMsg,wParam,lParam);
     }

     struct SWndProcArgs
     {
          HWND hwnd;
          UINT iMsg;
          WPARAM wParam;
          LPARAM lParam;
     };

     LRESULT VratiDefault( SWndProcArgs * Args )
     {
          return DefWindowProc( Args->hwnd, Args->iMsg, Args->wParam, Args->lParam);
     }

     HINSTANCE VratiHInstance( SWndProcArgs * Args )
     {
```

```
        return (HINSTANCE)GetWindowLong( Args->hwnd, GWL_HINSTANCE );
    }

    virtual LRESULT OnWMCreate( LPCREATESTRUCT pCS, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMSize( WPARAM fwSizeType, WORD nWidth, WORD nHeight,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMActivate( WORD fActive, WORD fMinimized, HWND hwndPrevious,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMNotify( int idCtrl, LPNMHDR pnmh, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMSetFocus( HWND hwndLoseFocus, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMKillFocus( HWND hwndGetFocus, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMKeyDown( int nVirtKey, LPARAM lKeyData, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMPaint( HDC wPar_hdc, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMChar( TCHAR chCharCode, LPARAM lKeyData, SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMLButtonDown( WPARAM fwKeys, WORD xPos, WORD yPos, SWndProcArgs
*Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMMouseWheel( WORD fwKeys, short zDelta, short xPos, short yPos,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMVScroll( int nScrollCode, short int nPos, HWND hwndScrollBar,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMHScroll( int nScrollCode, short int nPos, HWND hwndScrollBar,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
```

```
    }

    virtual LRESULT OnWMCommand( WORD wNotifyCode, WORD wID, HWND hwndCtl,
SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT OnWMDestroy( SWndProcArgs *Args )
    {
        return VratiDefault(Args);
    }

    virtual LRESULT WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
        try
        {
            SWndProcArgs Args = {hwnd, iMsg, wParam, lParam} ;

            switch (iMsg)
            {
            case WM_CREATE:
                return OnWMCreate( (LPCREATESTRUCT)lParam, &Args );

            case WM_SIZE:
                return OnWMSize( wParam, LOWORD(lParam), HIWORD(lParam), &Args );

            case WM_ACTIVATE:
                return OnWMActivate( LOWORD(wParam), HIWORD(wParam), (HWND)lParam,
&Args );

            case WM_NOTIFY:
                return OnWMNotify( (int)wParam, (LPNMHDR)lParam, &Args );

            case WM_SETFOCUS:
                return OnWMSetFocus( (HWND)wParam, &Args );

            case WM_KILLFOCUS:
                return OnWMKillFocus( (HWND)wParam, &Args );

            case WM_KEYDOWN:
                return OnWMKeyDown( (int)wParam, lParam, &Args );

            case WM_PAINT:
                return OnWMPaint( (HDC)wParam, &Args );

            case WM_CHAR:
                return OnWMChar( (TCHAR)wParam, lParam, &Args );

            case WM_LBUTTONDOWN:
                return OnWMLButtonDown( wParam, LOWORD(lParam), HIWORD(lParam), &Args
);

            case WM_MOUSEWHEEL:
                return OnWMMouseWheel( LOWORD(wParam), HIWORD(wParam),
                    LOWORD(lParam), HIWORD(lParam), &Args);

            case WM_VSCROLL:
                return OnWMVScroll( LOWORD(wParam), HIWORD(wParam), (HWND)lParam,
&Args );

            case WM_HSCROLL:
                return OnWMHScroll( LOWORD(wParam), HIWORD(wParam), (HWND)lParam,
&Args );

            case WM_COMMAND:
                return OnWMCommand( HIWORD(wParam), LOWORD(wParam), (HWND)lParam,
&Args );

            case WM_DESTROY:
                return OnWMDestroy( &Args );
            }
```

```
            return VratiDefault(&Args);
        }
        catch( string s )
        {
            s += "!";
            MessageBox( NULL, s.c_str(), "Fatalna greska (WndProc)", MB_OK );
        }
        catch( const char *msg )
        {
            string s;
            s += msg;
            s += "!";
            s += " (char) ";
            MessageBox( NULL, s.c_str(), "Fatalna greska (WndProc)", MB_OK );
        }
        catch( ... )
        {
            MessageBox( NULL, "Vracena iznimka nepoznatog tipa! (WndProc)",
                "Fatalna greska", MB_OK );
        }

        exit(0);
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.5.2. CZSWindow.cpp

```
//
//  CZSWindow
//

#include "StdAfx.h"

#include "CZSWindow.h"

//////////////////////////////////////////////////////////////////////

LONG CZSWindow::TmAdrCZSWindow = 0;

//////////////////////////////////////////////////////////////////////
```

9.5.3. CAboutDijalog.h

```
//
//  CAboutDijalog
//

#if !defined(CABOUTDIJALOG_H)
#define CABOUTDIJALOG_H

//////////////////////////////////////////////////////////////////////

class CAboutDijalog
{
```

```
public:
     CAboutDijalog( HINSTANCE hInstance, HWND hParentWnd )
     {
          DialogBox( hInstance, "AboutDijalog", hParentWnd, DlgProc );
     }

protected:

     static BOOL CALLBACK DlgProc( HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam )
     {
          switch(iMsg)
          {
          case WM_INITDIALOG:
               return TRUE;

          case WM_COMMAND:
               switch (LOWORD (wParam))
               {
               case IDOK:
               case IDCANCEL:
                    EndDialog( hDlg, 0);
                    return TRUE;
               }
               break ;
          }

          return FALSE ;
     }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.5.4. CDatView.h

```
//
//   CDatView
//

#if !defined(CDATVIEW_H)
#define CDATVIEW_H

#include "CZSWindow.h"

#include "CCharDatoteka.h"

#define IDC_DATVIEW 100

//////////////////////////////////////////////////////////////////////

class CDatView: public CZSWindow
{
public:

     CCharDatoteka & RefChDat;

     CDatView( CCharDatoteka & chdat ): RefChDat(chdat) { }

     CDatView( CCharDatoteka & chdat, HINSTANCE hInstance, HWND hParentWnd,
          int xPos, int yPos, int Sirina, int Visina ): RefChDat(chdat)
     {
          InicijalizirajProzor( "DatView",
               hInstance, hParentWnd,
               xPos, yPos, Sirina, Visina );
     }
```

```
    int RedKursor;
    int StupacKursor;

protected:

    void InicijalizirajProzor( char *ImeKlase,
        HINSTANCE hInstance, HWND hParentWnd,
        int xPos, int yPos, int Sirina, int Visina )
    {
        BojaPozadine = GetNearestColor( GetDC(NULL), RGB(255,255,215) );

        WNDCLASSEX wcex;

        // ako klasa vec nije registrirana..
        if( GetClassInfoEx( hInstance, ImeKlase, &wcex ) == 0)
        {
            // ..onda je registriraj
            wcex.cbSize        = sizeof(wcex);
            wcex.style         = CS_HREDRAW|CS_VREDRAW ;

            wcex.lpfnWndProc   = SetAdrWndProc;

            wcex.cbClsExtra    = 0;

            wcex.cbWndExtra    = 4;

            wcex.hInstance     = hInstance;
            wcex.hIcon         = NULL;
            wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);

            wcex.hbrBackground = CreateSolidBrush( BojaPozadine ) ;

            wcex.lpszMenuName  = NULL;
            wcex.lpszClassName = ImeKlase;
            wcex.hIconSm       = NULL;

            RegisterClassEx(&wcex);
        }

        TmAdrCZSWindow = (LONG)this;

        CreateWindowEx( WS_EX_CLIENTEDGE, ImeKlase, NULL,
            WS_CHILDWINDOW | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL,
            xPos, yPos, Sirina, Visina,
            hParentWnd, NULL, hInstance, NULL);
    }

    COLORREF BojaPozadine;

    int SirinaZnaka;
    int VisinaZnaka;

    int SirinaProzPix;
    int VisinaProzPix;

    int SirinaProzZn;
    int VisinaProzZn;

    int PrviVidRed;
    int PrviVidStup;

    void PostaviKursor()
    {
        SetCaretPos( (StupacKursor-PrviVidStup)*SirinaZnaka,
            (RedKursor-PrviVidRed)*VisinaZnaka );
    }

    bool PomakniAkoJeIzvan()
    {
        bool bPomakIzvan = false;

        int VelRed = RefChDat.VratiVelRedaBezLFiCR( RedKursor );
```

```
    if( StupacKursor >= VelRed)
        StupacKursor = VelRed;

    if(StupacKursor < PrviVidStup)
    {
        PrviVidStup = StupacKursor;
        bPomakIzvan = true;
    }
    else if(StupacKursor+1 >= PrviVidStup+SirinaProzZn)
    {
        PrviVidStup = StupacKursor-SirinaProzZn+2;
        bPomakIzvan = true;
    }

    if(RedKursor < PrviVidRed)
    {
        PrviVidRed = RedKursor;
        bPomakIzvan = true;
    }
    else if(RedKursor+1 >= PrviVidRed+VisinaProzZn)
    {
        PrviVidRed = RedKursor-VisinaProzZn+2;
        bPomakIzvan = true;
    }

    return bPomakIzvan;
}

void RectOdTrRedDoDna(RECT & PodCrtanja)
{
    PodCrtanja.top = (RedKursor-PrviVidRed)*VisinaZnaka;
    PodCrtanja.bottom = VisinaProzPix;
    PodCrtanja.left = 0;
    PodCrtanja.right = SirinaProzPix;
}

void RectOdRedDoRed(RECT & PodCrtanja, int Od, int Do)
{
    PodCrtanja.top = (Od-PrviVidRed)*VisinaZnaka;
    PodCrtanja.bottom = (Do+1-PrviVidRed)*VisinaZnaka;
    PodCrtanja.left = 0;
    PodCrtanja.right = SirinaProzPix;
}

int VScrollMax;
int HScrollMax;

void OsvjeziKlizneTrake()
{
    SCROLLINFO scinf;

    scinf.cbSize = sizeof(scinf);
    scinf.fMask = SIF_ALL;
    scinf.nMin = 0;

    VScrollMax = _cpp_max( 0, RefChDat.VratiBrojRedova() );
    PrviVidRed = _cpp_min( PrviVidRed, VScrollMax );

    HScrollMax = _cpp_max( 0, RefChDat.VratiDuljinuNajduzegReda() );
    PrviVidStup = _cpp_min( PrviVidStup, HScrollMax );

    scinf.nMax = VScrollMax;
    scinf.nPage = VisinaProzZn;
    scinf.nPos = PrviVidRed;

    SetScrollInfo( m_hwnd, SB_VERT, &scinf, FALSE);

    scinf.nMax = HScrollMax;
    scinf.nPage = SirinaProzZn;
    scinf.nPos = PrviVidStup;
```

```
        SetScrollInfo( m_hwnd, SB_HORZ, &scinf, FALSE);
    }

    ////////////////////////////////// message handleri:

    LRESULT OnWMCreate( LPCREATESTRUCT pCS, SWndProcArgs *Args )
    {
        PrviVidRed = PrviVidStup = 0;
        RedKursor = StupacKursor = 0;

        TEXTMETRIC tm;

        HDC hdc = GetDC(m_hwnd);

        SelectObject(hdc, GetStockObject (SYSTEM_FIXED_FONT));
        GetTextMetrics(hdc, &tm) ;

        SirinaZnaka = tm.tmAveCharWidth;
        VisinaZnaka = tm.tmHeight;

        ReleaseDC(m_hwnd, hdc);

        return 0;
    }

    LRESULT OnWMSize( WPARAM fwSizeType, WORD nWidth, WORD nHeight, SWndProcArgs *Args
)
    {
        // obtain window size in pixels
        SirinaProzPix = nWidth;
        VisinaProzPix = nHeight;

        // calculate window size in characters
        SirinaProzZn = (SirinaProzPix + SirinaZnaka - 1) / SirinaZnaka;
        VisinaProzZn = (VisinaProzPix + VisinaZnaka - 1) / VisinaZnaka;

        // klizne trake
        OsvjeziKlizneTrake();

        if(m_hwnd == GetFocus())
            PostaviKursor();

        return 0;
    }

    LRESULT OnWMSetFocus( HWND hwndLoseFocus, SWndProcArgs *Args )
    {
        // create and show the caret
        CreateCaret(m_hwnd, NULL, SirinaZnaka, VisinaZnaka);
        PostaviKursor();
        ShowCaret(m_hwnd);

        SendMessage( GetParent(m_hwnd), WM_COMMAND, IDC_DATVIEW, 0);

        return 0;
    }

    LRESULT OnWMKillFocus( HWND hwndGetFocus, SWndProcArgs *Args )
    {
        // hide and destroy the caret
        HideCaret(m_hwnd);
        DestroyCaret();

        return 0;
    }

    LRESULT OnWMLButtonDown( WPARAM fwKeys, WORD xPos, WORD yPos, SWndProcArgs *Args )
    {
        SetFocus( m_hwnd );

        RedKursor = _cpp_min(
            PrviVidRed + yPos/VisinaZnaka,
```

```
            RefChDat.VratiBrojRedova()-1 );

        StupacKursor = _cpp_min(
            PrviVidStup + xPos/SirinaZnaka,
            RefChDat.VratiVelRedaBezLFiCR( RedKursor ) );

        PostaviKursor();

        SendMessage( GetParent(m_hwnd), WM_COMMAND, IDC_DATVIEW, 0);

    // todo: da se vidi trenutna sintaksna jedinka,
    //  trenutno treba iscrtavati cijeli prozor
        InvalidateRect(m_hwnd,NULL,true);

        return 0;
    }

    LRESULT OnWMMouseWheel( WORD fwKeys, short zDelta, short xPos, short yPos,
SWndProcArgs *Args )
    {
        static int SumZ=0;
        int OldPoz = PrviVidRed;

        SumZ += zDelta;

        PrviVidRed -= (SumZ/WHEEL_DELTA)*2;
        SumZ %= WHEEL_DELTA;

        if( PrviVidRed<0 ) PrviVidRed=0;
        else if( PrviVidRed>VScrollMax ) PrviVidRed=VScrollMax;

        int pomak = PrviVidRed-OldPoz;

        ScrollWindow( m_hwnd, 0, -VisinaZnaka*pomak, NULL, NULL );
        UpdateWindow( m_hwnd );

        return 0;
    }

    LRESULT OnWMKeyDown( int nVirtKey, LPARAM lKeyData, SWndProcArgs *Args )
    {
        bool bTrebaPaint;
        RECT *pPod=NULL;

        switch(nVirtKey)
        {
        case VK_HOME:
            StupacKursor = 0;
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_END:
            StupacKursor = RefChDat.VratiVelRedaBezLFiCR(RedKursor) ;
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_PRIOR:
            // PageUp
            {
                int OldPrviRed = PrviVidRed;
                PrviVidRed = _cpp_max( PrviVidRed-VisinaProzZn, 0 );
                RedKursor -= OldPrviRed-PrviVidRed;
                PomakniAkoJeIzvan();
                bTrebaPaint = true;
            }
            break;

        case VK_NEXT:
            // PageDown
            {
                int zadnji = RefChDat.VratiBrojRedova()-1;
                PrviVidRed = _cpp_min( PrviVidRed+VisinaProzZn, zadnji );
```

```
                RedKursor = _cpp_min( RedKursor+VisinaProzZn, zadnji );
                PomakniAkoJeIzvan();
                bTrebaPaint = true;
            }
            break;

        case VK_LEFT:
            if(StupacKursor > 0)
                StupacKursor--;
            else if(RedKursor > 0) {
                RedKursor--;
                StupacKursor = RefChDat.VratiVelRedaBezLFiCR(RedKursor);
            }
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_RIGHT:
            if(StupacKursor < RefChDat.VratiVelRedaBezLFiCR(RedKursor))
                StupacKursor++;
            else if(RedKursor < RefChDat.VratiBrojRedova()-1) {
                RedKursor++;
                StupacKursor = 0;
            }
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_UP:
            RedKursor = _cpp_max(RedKursor-1, 0);
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_DOWN :
            RedKursor = _cpp_min(RedKursor+1, RefChDat.VratiBrojRedova()-1) ;
            bTrebaPaint = PomakniAkoJeIzvan();
            break;

        case VK_DELETE:
            {
                RECT PodCrtanja;

                if( StupacKursor == RefChDat.VratiVelRedaBezLFiCR(RedKursor) )
                {
                    if(RedKursor != RefChDat.VratiBrojRedova()-1)
                        RefChDat.SpojiSaSljedecimRedom(RedKursor);

                    RectOdTrRedDoDna(PodCrtanja);
                }
                else
                {
                    pair<int,int> IntRed = RefChDat.Obrisi(RedKursor, StupacKursor);
                    RectOdRedDoRed(PodCrtanja, IntRed.first, IntRed.second);
                }

                pPod = &PodCrtanja;
            }
            PomakniAkoJeIzvan();
            OsvjeziKlizneTrake();
            bTrebaPaint = true;
            break;

        default:
            return 0;
        }

        PostaviKursor();

        SendMessage( GetParent(m_hwnd), WM_COMMAND, IDC_DATVIEW, 0);

/*  // todo: da se vidi trenutna sintaksna jedinka,
    //  trenutno treba iscrtavati cijeli prozor

        if( bTrebaPaint )
```

```
            InvalidateRect(m_hwnd,pPod,true);
*/
        InvalidateRect(m_hwnd,NULL,true);

        return 0;
    }

    LRESULT OnWMChar( TCHAR chCharCode, LPARAM lKeyData, SWndProcArgs *Args )
    {
        int i;

        // ako se znak pocne ponavljati..
        for(i=0 ; i<(int)LOWORD(lKeyData) ; i++)
        {
            // obrada pojedinog znaka..
            switch(chCharCode)
            {
            case '\b':        // backspace
                if( StupacKursor!=0 || RedKursor!=0 )
                {
                    SendMessage(m_hwnd, WM_KEYDOWN, VK_LEFT, 1L);
                    SendMessage(m_hwnd, WM_KEYDOWN, VK_DELETE, 1L);
                }
                break;

            case '\t':        // tab
                do
                    SendMessage(m_hwnd, WM_CHAR, ' ', 1L);
                while(StupacKursor%4 != 0);

                break;

            case '\n':        // line feed
                MessageBox(NULL,"Ovo je okinulo line feed","vidi OnWMChar()",MB_OK);
                break;

            case '\r':        // carriage return
                {
                    // treba ponovo crtati donji dio prozora
                    RECT PodCrtanja;

                    RectOdTrRedDoDna(PodCrtanja);

                    // promjeni datoteku
                    RefChDat.PrelomiRed(RedKursor,StupacKursor);

                    // pomakni kursor
                    SendMessage(m_hwnd, WM_KEYDOWN, VK_HOME, 1L);
                    SendMessage(m_hwnd, WM_KEYDOWN, VK_DOWN, 1L);

                    OsvjeziKlizneTrake();

                    InvalidateRect(m_hwnd,&PodCrtanja,true);

                    SendMessage( GetParent(m_hwnd), WM_COMMAND, IDC_DATVIEW, 0);
                }
                break;

            default:     // character codes
                {
                    // ubaci znak
                    pair<int,int> IntRed = RefChDat.Umetni(

                        RedKursor, StupacKursor, chCharCode );

                    // pomakni se za jedno mjesto
                    SendMessage(m_hwnd, WM_KEYDOWN, VK_RIGHT, 1L);

                    OsvjeziKlizneTrake();

                    // ponovo iscrtaj taj red
                    RECT PodCrtanja;
```

```
                    RectOdRedDoRed(PodCrtanja,IntRed.first,IntRed.second);

                    InvalidateRect(m_hwnd,&PodCrtanja,true);

                    SendMessage( GetParent(m_hwnd), WM_COMMAND, IDC_DATVIEW, 0);
                }
                break;
            }
        }

        PostaviKursor();

    // todo: da se vidi trenutna sintaksna jedinka,
    //  trenutno treba iscrtavati cijeli prozor
        InvalidateRect(m_hwnd,NULL,true);

        return 0;
    }

    LRESULT OnWMPaint( HDC wPar_hdc, SWndProcArgs *Args )
    {
        char ch;
        int ZadnjiRed, ZadnjiStup, y, x;
        PAINTSTRUCT ps;

        SetScrollPos( m_hwnd, SB_VERT, PrviVidRed, TRUE );
        SetScrollPos( m_hwnd, SB_HORZ, PrviVidStup, TRUE );

        HDC hdc = BeginPaint(m_hwnd, &ps);

        SetBkColor( hdc, BojaPozadine );

        FillRect( hdc, &(ps.rcPaint), CreateSolidBrush(BojaPozadine) );

        int NeisPodZnTop  = ps.rcPaint.top / VisinaZnaka;
        int NeisPodZnBottom = (ps.rcPaint.bottom+VisinaZnaka-1) / VisinaZnaka;
        int NeisPodZnLeft = ps.rcPaint.left / SirinaZnaka;
        int NeisPodZnRight    = (ps.rcPaint.right+VisinaZnaka-1) / SirinaZnaka;

        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT) );

        ZadnjiRed = _cpp_min( RefChDat.VratiBrojRedova(),
            PrviVidRed+NeisPodZnBottom ) ;

        for( y=PrviVidRed+NeisPodZnTop ; y<ZadnjiRed ; y++)
        {
            ZadnjiStup = _cpp_min( RefChDat.VratiVelRedaBezLFiCR(y),
                PrviVidStup+NeisPodZnRight );

            for( x=PrviVidStup+NeisPodZnLeft ; x<ZadnjiStup ; x++)
            {
                ch = RefChDat.VratiElNaPoz(y,x);

                SetTextColor( hdc, RGB(0,0,0) );

                TextOut( hdc,
                    (x-PrviVidStup)*SirinaZnaka,
                    (y-PrviVidRed)*VisinaZnaka, &ch, 1);
            }
        }

        EndPaint (m_hwnd, &ps) ;

        return 0;
    }

    int ObradiScrollBar( int nScrollCode, short int nPos,
        int & Poz, int MaxPoz, int WinSize )
    {
        int OldPoz = Poz;
```

```
        switch(nScrollCode)
        {
        case SB_TOP:
             Poz = 0;
             break;

        case SB_BOTTOM:
             Poz = MaxPoz;
             break;

        case SB_LINEUP:
             Poz--;
             break;

        case SB_LINEDOWN:
             Poz++;
             break;

        case SB_PAGEUP:
             Poz -= WinSize;
             break;

        case SB_PAGEDOWN:
             Poz += WinSize;
             break;

        case SB_THUMBTRACK:
             Poz = nPos;
             break;

        default:
             break;
        }

        if( Poz<0 ) Poz=0;
        else if( Poz>MaxPoz ) Poz=MaxPoz;

        return Poz-OldPoz;
    }

    LRESULT OnWMVScroll( int nScrollCode, short int nPos, HWND hwndScrollBar,
SWndProcArgs *Args )
    {
        int pomak = ObradiScrollBar( nScrollCode, nPos,
                         PrviVidRed, VScrollMax, VisinaProzZn);

        ScrollWindow( m_hwnd, 0, -VisinaZnaka*pomak, NULL, NULL );
        UpdateWindow( m_hwnd );

        return 0;
    }

    LRESULT OnWMHScroll( int nScrollCode, short int nPos, HWND hwndScrollBar,
SWndProcArgs *Args )
    {
        int pomak = ObradiScrollBar( nScrollCode, nPos,
                         PrviVidStup, HScrollMax, SirinaProzZn);

        ScrollWindow( m_hwnd, -SirinaZnaka*pomak, 0, NULL, NULL );
        UpdateWindow( m_hwnd );

        return 0;
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.5.5. CLexView.h

```
//
//  CLexView
//

#if !defined(CLEXVIEW_H)
#define CLEXVIEW_H

#include "CDatView.h"

#include "CCharLexDat.h"

////////////////////////////////////////////////////////////////////

class CLexView: public CDatView
{
public:

     CCharLexDat & RefChLexDat;

     CLexView( CCharLexDat & chlexdat ):
          RefChLexDat(chlexdat), CDatView(chlexdat),
LexPal("podaci/JednostavanJezik.pal")
     {
          if( RefChLexDat.pLex->VratiBrRazlicitihTok() != LexPal.Velicina() )
               throw string("Razlicit broj tokena u paleti i leksickom analizatoru");
     }

     CLexView( CCharLexDat & chlexdat, HINSTANCE hInstance, HWND hParentWnd,
          int xPos, int yPos, int Sirina, int Visina ):
          RefChLexDat(chlexdat), CDatView(chlexdat),
LexPal("podaci/JednostavanJezik.pal")
     {
          InicijalizirajProzor( "LexView",
               hInstance, hParentWnd,
               xPos, yPos, Sirina, Visina );

          if( RefChLexDat.pLex->VratiBrRazlicitihTok() != LexPal.Velicina() )
               throw string("Razlicit broj tokena u paleti i leksickom analizatoru");
     }

     class CLexPaletaBoja
     {
     public:
          CLexPaletaBoja(const string &ime_dat)
          {
               ifstream ins(ime_dat.c_str());

               if(ins.fail())
                    throw string("Ne mogu otvoriti datoteku za paletu");

               Ucitaj(ins);

               ins.close();
          }

          CLexPaletaBoja(istream & ins)
          {
               Ucitaj(ins);
          }

          void Ucitaj(istream & ins)
          {
               int i,n,r,g,b;

               ins >> n;

               for(i=0;i<n;i++)
               {
```

```
                ins >> r >> g >> b;
                paleta.push_back( RGB(r,g,b) );
            }
        }

        int Velicina()
        {
            return paleta.size();
        }

        COLORREF & operator[](int koji)
        {
            return paleta[koji];
        }

    protected:
        my_vect<COLORREF> paleta;
    };

protected:

    CLexPaletaBoja LexPal;

    LRESULT OnWMPaint( HDC wPar_hdc, SWndProcArgs *Args )
    {
        int ZadnjiRed, ZadnjiStup, y, x;
        PAINTSTRUCT ps;

        SetScrollPos( m_hwnd, SB_VERT, PrviVidRed, TRUE );
        SetScrollPos( m_hwnd, SB_HORZ, PrviVidStup, TRUE );

        HDC hdc = BeginPaint(m_hwnd, &ps);

        SetBkColor( hdc, BojaPozadine );

        FillRect( hdc, &(ps.rcPaint), CreateSolidBrush(BojaPozadine) );

        int NeisPodZnTop  = ps.rcPaint.top / VisinaZnaka;
        int NeisPodZnBottom = (ps.rcPaint.bottom+VisinaZnaka-1) / VisinaZnaka;
        int NeisPodZnLeft = ps.rcPaint.left / SirinaZnaka;
        int NeisPodZnRight    = (ps.rcPaint.right+VisinaZnaka-1) / SirinaZnaka;

        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT) );

        ZadnjiRed = _cpp_min( RefChDat.VratiBrojRedova(),
            PrviVidRed+NeisPodZnBottom ) ;

        for( y=PrviVidRed+NeisPodZnTop ; y<ZadnjiRed ; y++)
        {
            x=PrviVidStup+NeisPodZnLeft;

            ZadnjiStup = _cpp_min( RefChDat.VratiVelRedaBezLFiCR(y),
                PrviVidStup+NeisPodZnRight );

            CLexAnalizator::NIZCHTIP niz;

            RefChLexDat.pLex->NapuniInterval( y, x, ZadnjiStup, niz );

            CLexAnalizator::NIZCHTIP::iterator it;

            for( it=niz.begin() ; x<ZadnjiStup ; x++, it++ )
            {
                SetTextColor( hdc, LexPal[ it->second ] );

                TextOut( hdc, (x-PrviVidStup)*SirinaZnaka,
                    (y-PrviVidRed)*VisinaZnaka, &(it->first), 1);
            }
        }

        EndPaint (m_hwnd, &ps) ;

        return 0;
```

```
    }
};

//////////////////////////////////////////////////////////////////

#endif
```

9.5.6. CSynView.h

```
//
//  CSynView
//

#if !defined(CSYNVIEW_H)
#define CSYNVIEW_H

#include "CLexView.h"

#include "CLexSynDat.h"

//////////////////////////////////////////////////////////////////

class CSynView: public CLexView
{
public:

    CLexSynDat & RefLexSynDat;

    CSynView( CLexSynDat & lexsyndat, HINSTANCE hInstance, HWND hParentWnd,
        int xPos, int yPos, int Sirina, int Visina ):
        RefLexSynDat(lexsyndat), CLexView( *(lexsyndat.pChLexDat) )
    {
        InicijalizirajProzor( "SynView",
            hInstance, hParentWnd,
            xPos, yPos, Sirina, Visina );
    }

protected:

    LRESULT OnWMPaint( HDC wPar_hdc, SWndProcArgs *Args )
    {
        int ZadnjiRed, ZadnjiStup, y, x;
        PAINTSTRUCT ps;
        static int TrGeneracija = 0;

        TrGeneracija++;

        SetScrollPos( m_hwnd, SB_VERT, PrviVidRed, TRUE );
        SetScrollPos( m_hwnd, SB_HORZ, PrviVidStup, TRUE );

        HDC hdc = BeginPaint(m_hwnd, &ps);

        SetBkColor( hdc, BojaPozadine );

        FillRect( hdc, &(ps.rcPaint), CreateSolidBrush(BojaPozadine) );

        int NeisPodZnTop  = ps.rcPaint.top / VisinaZnaka;
        int NeisPodZnBottom = (ps.rcPaint.bottom+VisinaZnaka-1) / VisinaZnaka;
        int NeisPodZnLeft = ps.rcPaint.left / SirinaZnaka;
        int NeisPodZnRight    = (ps.rcPaint.right+VisinaZnaka-1) / SirinaZnaka;

        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT) );

        ZadnjiRed = _cpp_min( RefChDat.VratiBrojRedova(),
            PrviVidRed+NeisPodZnBottom ) ;

        RefLexSynDat.pSyn->NovaKurGen( RedKursor, StupacKursor, TrGeneracija );
```

```
        HPEN hPen = (HPEN) SelectObject( hdc, CreatePen(PS_DASH, 0, RGB(255,0,0)) );

        for( y=PrviVidRed+NeisPodZnTop ; y<ZadnjiRed ; y++)
        {
             x=PrviVidStup+NeisPodZnLeft;

             ZadnjiStup = _cpp_min( RefChDat.VratiVelRedaBezLFiCR(y),
                  PrviVidStup+NeisPodZnRight );

             CSynAnalizator::NIZCH3SVOJSTVA niz;

             RefLexSynDat.pSyn->NapuniInterval( y, x, ZadnjiStup, niz );

             CSynAnalizator::NIZCH3SVOJSTVA::iterator it;

             int tmy = (y-PrviVidRed+1)*VisinaZnaka - 1;

             for( it=niz.begin() ; x<ZadnjiStup ; x++, it++ )
             {
                  if( it->Tip!=NASTAVAK )
                       SetTextColor( hdc, LexPal[ it->Tip ] );

                  if( it->KurGeneracija==TrGeneracija )
                       SetBkColor( hdc, RGB(202,217,244) );
                  else
                       SetBkColor( hdc, BojaPozadine );

                  TextOut( hdc, (x-PrviVidStup)*SirinaZnaka,
                       (y-PrviVidRed)*VisinaZnaka, &(it->Ch), 1);

                  if(it->bSinIspravan==false)
                  {
                       MoveToEx( hdc, (x-PrviVidStup)*SirinaZnaka, tmy, NULL );
                       LineTo( hdc, (x-PrviVidStup+1)*SirinaZnaka, tmy );
                  }
             }

        }

        DeleteObject( SelectObject( hdc, hPen ) );

        EndPaint (m_hwnd, &ps) ;

        return 0;
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.5.7. GlavniMeni.h

```
//
//  GlavniMeni
//

#if !defined(CGLAVNIMENI_H)
#define CGLAVNIMENI_H

//////////////////////////////////////////////////////////////////////

#define IDM_EXIT 1
#define IDM_ABOUT 10

//////////////////////////////////////////////////////////////////////
```

```
#endif
```

9.5.8. CGlavniProzor.h

```
//
//  CGlavniProzor
//

#if !defined(CGLAVNIPROZOR_H)
#define CGLAVNIPROZOR_H

#include "CZSWindow.h"
#include "GlavniMeni.h"
#include "CAboutDijalog.h"
#include "CSynView.h"

#define JORO_VER "0.21"

//////////////////////////////////////////////////////////////////////

class CGlavniProzor: public CZSWindow
{
public:

    CGlavniProzor( HINSTANCE hInstance, int iCmdShow,
        int xPos, int yPos, int Sirina, int Visina )
    {
        // inicijalizacija prozora

        char ImeKlase[] = "GlavProz";

        WNDCLASSEX wcex;

        // ako klasa vec nije registrirana..
        if( GetClassInfoEx( hInstance, ImeKlase, &wcex ) == 0)
        {
            // ..onda je registriraj
            wcex.cbSize        = sizeof(wcex);
            wcex.style         = CS_HREDRAW|CS_VREDRAW ;

            wcex.lpfnWndProc   = SetAdrWndProc;

            wcex.cbClsExtra    = 0;

            wcex.cbWndExtra    = 4;

            wcex.hInstance     = hInstance;
            wcex.hIcon         = (HICON)LoadImage(hInstance, "IkonaPrograma",
                                    IMAGE_ICON, 32, 32, LR_DEFAULTCOLOR);
            wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
            wcex.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH );
            wcex.lpszMenuName  = "GlavniMeni";
            wcex.lpszClassName = ImeKlase;
            wcex.hIconSm       = (HICON)LoadImage(hInstance, "IkonaPrograma",
                                    IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR);

            RegisterClassEx(&wcex);
        }

        TmAdrCZSWindow = (LONG)this;

        CreateWindow( ImeKlase, " Jezicno Orjentirana Razvojna Okolina - ver
"JORO_VER,
            WS_OVERLAPPEDWINDOW,
            xPos, yPos, Sirina, Visina,
```

```
            NULL, NULL, hInstance, NULL) ;

        ShowWindow(m_hwnd, iCmdShow);
        UpdateWindow(m_hwnd);

        SetFocus( (*ViewProzori.begin())->m_hwnd );
    }

    ~CGlavniProzor()
    {
        CZSWindow::~CZSWindow();

        list< CSynView * >::iterator vit;

        for(vit=ViewProzori.begin();vit!=ViewProzori.end();vit++)
            delete *vit;

        list< CLexSynDat * >::iterator cit;

        for(cit=TmDat.begin();cit!=TmDat.end();cit++)
            delete *cit;
    }

protected:

    list< CSynView * > ViewProzori;

    list< CLexSynDat * > TmDat;

    HWND hStatusnaLinija;

    LRESULT OnWMCreate( LPCREATESTRUCT pCS, SWndProcArgs *Args )
    {
        // inicijalizacija datoteke
        CLexSynDat *pLSDat = new CLexSynDat("Podaci\\prog.txt");

        TmDat.push_back( pLSDat );

        // inicijalizacija prozora
        CSynView *pSV = new CSynView( *pLSDat, VratiHInstance(Args),
            m_hwnd, 0, 0, 0, 0 );

        ViewProzori.push_back( pSV );

        // inicijalizacija statusne trake
        hStatusnaLinija = CreateStatusWindow(
                WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
                CCS_BOTTOM | SBARS_SIZEGRIP,
                "statusna linija",
                m_hwnd, 2);

        return 0;
    }

    LRESULT OnWMSize( WPARAM fwSizeType, WORD nWidth, WORD nHeight, SWndProcArgs *Args
)
    {
        // moramo rucno mijenjati poziciju statusne linije
        RECT StatRect;

        GetWindowRect( hStatusnaLinija, &StatRect);

        int VisinaStatusneLinije = StatRect.bottom - StatRect.top;

        nHeight -= VisinaStatusneLinije;

        MoveWindow( hStatusnaLinija, 0, nHeight,
            nWidth, VisinaStatusneLinije, TRUE);

        // a onda i velicinu view prozora
        MoveWindow( (*(ViewProzori.begin()))->m_hwnd,
            0, 0, nWidth, nHeight, TRUE );
```

```
        return 0;
    }

    LRESULT OnWMCommand( WORD wNotifyCode, WORD wID, HWND hwndCtl, SWndProcArgs *Args
)
    {
        HMENU hMenu = GetMenu( m_hwnd );

        switch( wID )
        {
        case IDM_EXIT:
            return OnWMDestroy(Args);

        case IDM_ABOUT:
            {
                CAboutDijalog about( VratiHInstance(Args), m_hwnd );
            }
            return 0;

        case IDC_DATVIEW:
            {
                stringstream tmss;

                tmss << "red: " << (*(ViewProzori.begin()))->RedKursor <<
                    "  stupac: " << (*(ViewProzori.begin()))->StupacKursor <<
                    "  broj novih leksickih jedinki: " << (*(ViewProzori.begin()))-
>RefChLexDat.pLex->VratiBrNovihTokena() <<
                    "  uspjesan popravak: ";

                if( (*(ViewProzori.begin()))->RefLexSynDat.pSyn-
>VratiUspjesanPopravak() )
                    tmss << "DA";
                else
                    tmss << "NE";

                SetWindowText( hStatusnaLinija, tmss.str().c_str() );
            }
            return 0;

        default:
            throw string("Selektirana nepostojeca stavka menija");
        }

        return VratiDefault(Args);
    }

    LRESULT OnWMDestroy( SWndProcArgs *Args )
    {
        PostQuitMessage(0);

        return 0;
    }

};

//////////////////////////////////////////////////////////////////////

#endif
```

9.6. Ostale datoteke

9.6.1. StdAfx.h

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN        // Exclude rarely-used stuff from Windows headers

#define _WIN32_WINNT 0x400

#include <windows.h>

#include <commctrl.h>

// reference additional headers your program requires here

#pragma warning(disable: 4786) // kod koristenja STL VC++ kompajler stalno prijavljuje
                                  //  warning 4786 - ova linija iskljucuje navedeni

#include <typeinfo>       // koristimo STL (Standard Template Library)
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <list>
#include <map>
#include <set>
#include <stack>
#include <algorithm>
#include <strstream>

using namespace std;  // takoder zbog STL-a

#include "MyVect.h"

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
```

9.6.2. StdAfx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
//   JORO.pch will be the pre-compiled header
//   stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// reference any additional headers you need in STDAFX.H
// and not in this file
```

9.6.3. MyVect.h

```
#if !defined(MYVECT_H)
#define MYVECT_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <vector>

using namespace std;

//////////////////////////////////////////////////////////////////////

template<class _Ty, class _A = allocator<_Ty> >
    class my_vect: public vector<_Ty,_A>
{
public:
    explicit my_vect(const _A& _Al = _A())
    {
        allocator=_Al;
        _First=0;
        _Last=0;
        _End=0;
    }

    explicit my_vect(size_type _N, const _Ty& _V = _Ty(), const _A& _Al = _A())

    {
        allocator=_Al;
        _First = allocator.allocate(_N, (void *)0);
        _Ufill(_First, _N, _V);
        _Last = _First + _N;
        _End = _Last;
    }

    my_vect(const _Myt& _X)
    {
        allocator=_X.allocator;
        _First = allocator.allocate(_X.size(), (void *)0);

        _Last = _Ucopy(_X.begin(), _X.end(), _First);
        _End = _Last;
    }

    my_vect(_It _F, _It _L, const _A& _Al = _A())
    {
        allocator=_Al;
        _First=0;
        _Last=0;
        _End=0;
        insert(begin(), _F, _L);
    }

    ~my_vect()
    {
        _Destroy(_First, _Last);
        allocator.deallocate(_First, _End - _First);
        _First = 0, _Last = 0, _End = 0;
    }

    reference operator[](size_type pos)
    {
#ifdef _DEBUG
        return vector<_Ty,_A>::at(pos);
#else
        return vector<_Ty,_A>::operator[](pos);
#endif
    }

    const_reference  operator[](size_type pos) const
    {
#ifdef _DEBUG
```

```
        return vector<_Ty,_A>::at(pos);
#else
        return vector<_Ty,_A>::operator[](pos);
#endif
    }
};

//////////////////////////////////////////////////////////////////////

#endif
```

9.6.4. JORO.cpp

```
// JORO.cpp : Defines the entry point for the application.
//

#include "stdafx.h"

#include "CGlavniProzor.h"

//////////////////////////////////////////////////////////////////////

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int iCmdShow )
{
    try
    {
        InitCommonControls();

        CGlavniProzor gp( hInstance, iCmdShow, 300, 100, 700, 500 );

        gp.Radi();
    }
    catch( string s )
    {
        s += "!";
        MessageBox( NULL, s.c_str(), "Fatalna greska (WinMain)", MB_OK );
    }
    catch( const char *msg )
    {
        string s;
        s += msg;
        s += "!";
        s += " (char) ";
        MessageBox( NULL, s.c_str(), "Fatalna greska (WinMain)", MB_OK );
    }
    catch( ... )
    {
        MessageBox( NULL, "Vracena iznimka nepoznatog tipa! (WinMain)",
            "Fatalna greska", MB_OK );
    }

    return 0;
}

//////////////////////////////////////////////////////////////////////
```