Predgovor

Kompajler je program koji prevodi ulazni program iz jednog programskog jezika u drugi. Da bi preveo program iz jednog jezika u drugi, kompajler prvo mora da učita program i da razume njegovu strukturu i značenje, a zatim treba da izgeneriše taj isti program ali na drugi način. Prednjim delom kompajlera se realizuje analiza ulaznog fajla, dok zadnji deo kompajlera realizuje sintezu.

U ovom diplomskom radu su realizovana prva dva obavezna dela svakog kompajlera – leksička i sintaksna analiza. Pošto su sintaksna i semantička analiza formalizovani postupci, za realizaciju leksičkog i sintaksnog analizatora je moguće koristiti razne generatore. U ovoj realizaciji su korišćeni generatori *JLex* (za leksički analizator) i *CUP* (za sintaksni analizator). Iako je u naučnom svetu rasprostranjeno mišljenje da generatori generišu neefikasan kod i da neracionalno koriste memoriju, ova dva genetatora prevazilaze ove probleme. Generator leksičkog analizatora *JLex* je, po nekim testovima proizvođača, čak brži od "ručno" pisanog leksičkog analizatora. Generator sintaksnog analizatora *CUP* koji generiše LALR sintaksni analizator prihvata skoro sve gramatike koje prihvata i zasad najopštiji LR sintaksni analizator, ali pri tome koristi mnogo manje memorije.

Leksički i sintaksni analizator je ovde realizovan za konkretni programski jezik "Tiger". Ovaj programski je osmislio profesor na Princeton univerzitetu - Andrew W. Appel. Osnovni razlog za nastanak programskog jezika Tiger je taj što je profesor Appel želeo da studentima prikaže princip realizacije kompajlera na jednom malom ali netrivijalnom jeziku.

Za realizaciju leksičkog i sintaksnog analizatora je korišćen programski jezik Java. Kao Java platforma je korišćen Java Development Kit 1.1.8. Java je moderan objektno - orjentisan programski jezik, koji je nastao tako što su njegovi tvorci ugradili dobre strane ostalih programskih jezika. Jedna od najboljih osobina Jave je ta što je Java program nezavisan od operativnog sistema, što znači da se program napisan u Javi može izvršavati na različitim operativnim sistemima bez ponovnog prevođenja.

Diplomski rad se sastoji od šest poglavlja u kojima je opisana ideja i realizacija leksičkog i sintaksnog analizatora za programski jezik Tiger. Izvorni kod programa nije navođen u celini zbog obimnosti, ali je priložen na disketi koja ide uz ovaj diplomski rad.

U prvom delu su opisani osnovni pojmovi i teorijske osnove vezane za konstrukciju kompajlera. Posebna pažnja je posvećena teoriji vezanoj za leksičku i sintaksnu analizu jer je to i tema ovog rada.

U drugom delu je detaljno opisan programski jezik Tiger. Iz razloga što je ovaj programski jezik relativno nov, ovde su opisane osobine jezika koje su bitne i za semantičku analizu (a ne samo za leksičku i sintaksnu analizu).

U trećem delu je opisana realizacija leksičkog analizatora. Pri opisu je prvo objašnjavana struktura specifikacije za *JLex*, a zatim je navođena i konkretna realizacija za programski jezik Tiger.

U četvrtom delu je opisana realizacija sintaksnog analizatora. Pri opisu je korišćen isti princip kao i u trećem poglavlju.

Peti deo se sastoji od test primera kojima se ilustruje da leksički i sintaksni analizator prihvata samo sintaksno ispravne programe.

Kratak zaključak o ovom diplomskom radu je dat u šestom delu.

Ovom prilikom želeo bih da se zahvalim svim članovima komisije na vremenu i trudu uloženom pri čitanju ovog rada, kao i mentoru dr Mirjani Ivanović koja me je zainteresovala za ovaj projekat. Takođe, želeo bih da se zahvalim majci Mirjani, na svesrdnoj pomoći i podršci tokom studija i izrade diplomskog rada, Tatjani Makuh kao i celoj porodici Makuh na strpljenju i podršci tokom studija.

Sadržaj

Predgovor *1*
Sadržaj *3*
1. Uvod *5*
1.1. Kompajleri i interpreteri *5*
1.2. Osnovna struktura kompajlera *6*
1.2.1. Leksička analiza *7*
1.2.2. Sintaksna analiza *8*
1.2.2.1. BNF *9*
1.2.2.2. EBNF *9*
1.2.2.3. Sintaksni dijagrami *9*
1.2.3. Semantička analiza *10*
1.2.4. Generisanje koda *10*
1.2.5. Optimizacija koda *10*
2. Programski jezik Tiger *12*
2.1. Leksičke napomene *12*
2.2. Deklaracije *12*
2.2.1. Tipovi podataka *12*
2.2.2. Promenljive *13*
2.2.3. Funkcije *14*
2.2.4. Pravila opsega *14*
2.3. Promenljive i izrazi *15*
2.3.1. L-vrednosti *15*
2.3.2. Izrazi *15*
2.3.3. Programi *18*
2.4. Standardna biblioteka *18*
3. Leksički analizator za programski jezik Tiger *19*
3.1. Ispisivanje poruka o greškama *20*
3.1.1. Klasa LineList *20*
3.1.2. Klasa ErrorMsg *20*
3.2. Struktura specifikacije leksičkog analizatora *21*
3.2.1. Korisnički kod *22*
3.2.2. JLex direktive *22*
3.2.2.1. Brojanje karaktera *23*
3.2.2.2. Brojanje linija *23*
3.2.2.3. Nekompatibilnost operativnih sistema *23*
3.2.2.4. Velika i mala slova *23*
3.2.2.5. Deklaracija stanja *23*

3.2.2.6. Kompatibilnost sa generatorom sintaksnog analizatora "CUP" *24*
3.2.2.7. Interni kod za leksički analizator *24*
3.2.2.8. Vraćanje vrednosti na kraju izvršavanja *25*
3.2.2.9. Definicije makroa *25*
3.2.3. Regularni izrazi *26*
3.2.3.1. Lista stanja *26*
3.2.3.2. Pravila za regularne izraze *27*
3.2.3.3. Akcije pridružene regularnom izrazu *28*
3.2.3.4. Regularni izrazi za programski jezik Tiger *28*
3.3. Generisanje leksičkog analizatora *31*
4. Sintaksni analizator za programski jezik Tiger *32*
4.1. Specifikacija sintakse *32*
4.1.1. Deklaracija paketa i uvozna lista *33*
4.1.2. Korisnički kod *33*
4.1.3. Lista simbola *34*
4.1.4. Deklaracija prioriteta i asocijacija *35*
4.1.5. Gramatika *36*
4.2. Oporavak od grešaka *39*
4.3. Generisanje i modifikacija sintaksnog analizatora *39*
5. Način rada sintaksnog analizatora i test primeri *42*
5.1. Program "Queens" *42*
5.2. Program "Merge" *43*
5.3. Ilustracija metoda "debug_parse" *45*
6. Zaključak *48*
Literatura *49*
Biografija *50*
Ključna dokumentacijska informacija *51*
Key words documentation *53*

1. Uvod

U ovom poglavlju biće ukratko spomenuti osnovni pojmovi vezani za konstrukciju kompajlera. To su uglavnom opšte poznati termini u računarstvu, te neće biti detaljno opisani.

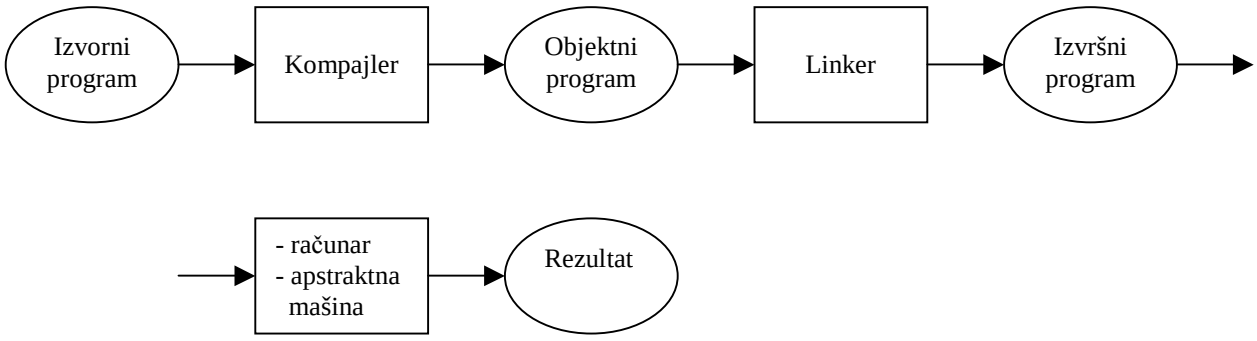
1.1. Kompajleri i interpreteri

Na početku bi trebalo napraviti razliku između programskog jezika i njegove realizacije. Programski jezik je skup pravila, koji postoji nezavisno od računara i nezavisno od toga da li je realizovan za neki računar ili nije. Nasuprot tome realizacija programskog jezika za neki računar obavezno se sastoji od prevodioca programskog jezika za taj računar, a može da sadrži i još neke mogućnosti: različite optimizacije, veliku biblioteku modula ili funkcija, ugrađeni "dibager" (pomoćni program za praćenje izvršavanja programa).

Prevodilac kao osnovni deo svake realizacije programskog jezika ima za zadatak da prevede izvorni program, zapisan u jednom programskom jeziku, u neki drugi jezik (može i mašinski). On može biti realizovan kao:

- Kompajler i
- Interpreter.

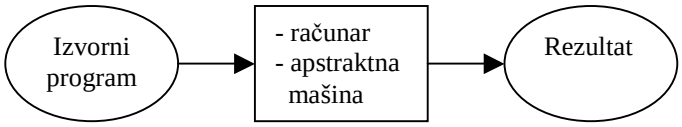
Prevođenje putem kompajlera ima sledeću strukturu:



Slika 1: Prevođenje putem kompajlera.

Kao što se iz slike vidi kompajler ima za zadatak da izvorni program prevede u objektni kod, koji zatim prihvata linker (koji je deo operativnog sistema) da bi napravio izvršni program za neki konkretan računar ili za neku apstraktnu mašinu.

Prevođenje putem interpretera ima sledeću strukturu:

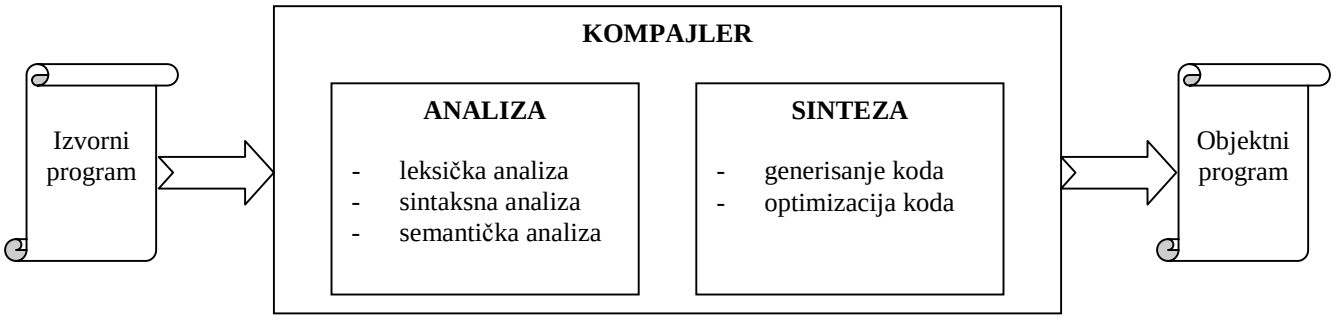


Slika 2: Prevođenje putem interpretera.

Kao što se iz slike vidi interpreter ne stvara objektnu formu već u isto vreme obrađuje internu formu programa i interpretira ga.

1.2. Osnovna struktura kompajlera

Pošto je kompajler u principu složen program stoga se njegova implementacija deli u više faza. Osnovna struktura kompajlera data je na slici 3.



Slika 3: Osnovna struktura kompajlera.

Kompajler ima dva osnovna dela i to su analiza i sinteza. U fazi analize ulazni fajl se analizira tj. analizira se njegova leksička, sintaksna i semantička ispravnost. U fazi sinteze vrši se generisanje nekog koda, a zatim se taj kod optimizuje.

1.2.1. Leksička analiza

Zadatak leksičke analize je da ulazni niz znakova podeli u pojedinačne smislene konstrukcije (identifikatore, stringove, operacije, itd) koji se koriste u narednom koraku kompajlera – sintaksnoj analizi. Pored ovog osnovnog zadatka u fazi leksičke analize se mogu realizovati i neke dodatne operacije kao što su:

- Izostavljanje elemenata programa koji nemaju značenje kao što su komentari, praznine, tabulatori, itd.
- Uspostavljanje veze između broja linije u kojoj se javila greška i poruke o grešci.
- Neke funkcije predprocesora kao što je ubacivanje tela makroa u poziv makroa.

Leksički analizator može biti realizovan kao poseban prolaz kroz ulazni fajl ili kao procedura koja se poziva iz sintaksnog analizatora. Drugi pristup je obično efikasniji, ali i prvi pristup ima svojih prednosti.

Razlikuju se tri načina realizacije leksičkog analizatora:

- Pisanje leksičkog analizatora u nekom asemblerskom jeziku
- Pisanje leksičkog analizatora korišćenjem nekog višeg programskog jezika
- Pisanje leksičkog analizatora korišćenjem nekog generatora leksičkih analizatora.

Prvi način je najteži, ali i najefikasniji, dok je treći način najlakši ali ovi generatori često proizvode neefikasan kod.

U osnovi svakog leksičkog analizatora stoji snažan matematički aparat – konačan automat. Konačni automat je matematički model koji služi za definisanje tokena da bi ih kasnije bilo lakše prepoznati u prevodiocima. U prevodiocima se mogu koristiti sledeći tipovi automata:

- Deterministički konačan automat
- Nedeterministički konačan automat
- Nedeterministički konačan automat sa praznim pravilom.

Formalno automat se definiše kao uređena petorka $(S, \Sigma, \delta, q_0, F)$ gde je:

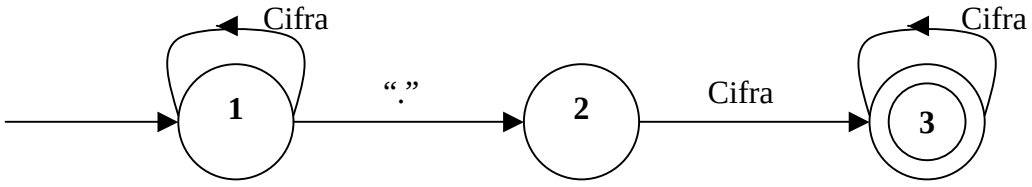
- S-konačan neprazan skup stanja automata
- Σ-ulazni alfabet automata
- δ-preslikavanje $S \rightarrow S$ koje predstavlja funkciju prelaska iz stanja u stanje
- $q_0 \in S$-početno stanje automata
- $F \subset S$-neprazan skup završnih stanja automata

Automat možemo zamisliti kao mašinu koja ima glavu za čitanje ulaznih karaktera i konačan skup stanja. Na početku čitanja automat se nalazi u početnom stanju. Promena stanja se vrši kada se očita naredni znak sa ulaza. Neka stanja automata se nazivaju završna stanja, i kada se automat nađe u tom stanju (i kada je kraj ulaznog stringa)

prekida se učitavanje znakova sa ulaza, a učitani niz znakova je niz znakova koje automat prihvata.

Automat se najčešće prikazuje kao grafički dijagram stanja.

Primer 1:
Automat za prepoznavanje realnog broja u nekom programskom jeziku bi izgledao ovako:



Slika 4: Automat koji prihvata realan broj.

Na slici 4. se vide svi osnovni elementi automata. Stanje automata se označava krugom u kome se obično upisuje broj tog stanja (da bi se stanja jedinstveno identifikovala). Početno stanje se prikazuje krugom u koga ulazi jedna prazna strelica. Prelazi stanja se prikazuju strelicama iznad kojih je napisan simbol koji prevodi automat iz stanja u stanje. Završno stanje se prikazuje duplim krugom.

1.2.2. Sintaksna analiza

Zadatak sintaksnog analizatora je da proveri sintaksnu ispravnost ulaznog programa, kao i da razčlani ulazni program na veće sintaksne celine: izraze, naredbe, procedure, itd. Pored toga sintaksni analizator bi trebao da otkrije što više grešaka u ulaznom fajlu kao i da se oporavi od njih.

Pošto je automat suviše slab da se opiše sintaksna ispravnost nekog programskog jezika, za potrebe sintaksne analize uvodi se još jedan matematički aparat – gramatike. Gramatika predstavlja formalizam za definiciju jezika.

U teoriji formalnih jezika gramatika se definiše kao uređena četvorka G=(N,T,s,P), gde je N-skup neterminalnih simbola odnosno simbola koji se mogu zameniti drugima, T-skup terminalnih simbola odnosno završnih simbola, s-početni neterminalni simbol koji se dalje razvija i P-skup pravila ili produkcija. Jezik generisan gramatikom predstavlja podskup od T*, ali samo onih reči koji se mogu izvesti iz početnog simbola s koristeći pravila izvođenja iz P.

Zbog velikog značaja koje imaju gramatike u specifikaciji programskih jezika, vremenom je razvijen niz formalizama za adekvatno predstavljanje pravila gramatike. Neki od najznačajnijih formalizama su:

- Bekusova normalna forma (BNF – Backus Normal Form)
- Proširena bekusova forma (EBNF – Extended Backus Normal Form)
- Sintaksni dijagrami.

1.2.2.1. BNF

U BNF skup pravila gramatike jezika se definiše skupom metaformula. Metaformula se sastoji od leve i desne strane koje su razdvojene metalingvističkom vezom "::=". Leva strana metaformule je metapromenljiva (neterminal), a desna strana je metaizraz.

Metaizraz predstavlja skup nizova metapromenljivih i metakonstanti (terminala) koji su razdvojeni metalingvističkom vezom " | " sa značenjem ili, a omogućava skraćeni zapis više pravila sa istom levom a različitom desnom stranom.

Metapromenljive se najčešće predstavljaju nizom slova i brojeva i znaka "_". Metakonstante se obično pišu pod znacima navoda.

Primer 2:
Ceo broj u nekom programskom jeziku se može prikazati na sledeći način:

Broj ::= Cifra Broj | Cifra.
Cifra::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

1.2.2.2. EBNF

Zapis pravila pomoću BNF je zbog pojave rekurzije, teško razumljiv za čoveka. Stoga je vremenom iz praktične potrebe došlo do njenog proširenja što je omogućilo sažetiju specifikaciju jezika. Proširenje skupa metasimbola BNF novim metasimbolima {(,),[,],{,}} dovodi do proširene Bekusove normalne forme – EBNF.

Metasimboli (i) se koriste da označe alternativni izbor. Metaizraz unutar uglastih zagrada [i] se ponavlja jednom ili nijednom. Metasimboli { i } se koriste da označe proizvoljan broj pojava metaizraza unutar zagrada.

Primer 3:
Neterminalni simbol "broj" iz primera 1. bi se mogao prikazati na sledeći način:

Broj ::= Cifra {Cifra}.
Cifra::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

1.2.2.3. Sintaksni dijagrami

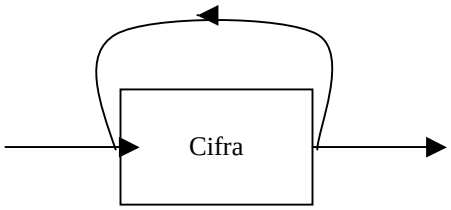
Sintaksni dijagram je grafički metod za opis sintakse programskih jezika, ekvivalentan BNF. Svakoj metaformuli jezika odgovara sintaksni dijagram koji je imenovan,

orjentisan, označen graf sa jednim ulazom i jednim izlazom.

Temena sintaksnog dijagrama označavaju:

- Metapromenljive, uokvirene pravougaonikom
- Metakonstante, uokvirene elipsom
- Račvanje označava alternativne puteve
- Kružna strelica oko nekog elementa označava višestruku pojavu tog elementa.

Primer 4:
Sintaksni dijagram za broj iz primera 2 i 3 je prikazan na slici 5.



Slika 5: Sintaksni dijagram za broj

1.2.3. Semantička analiza

Semantička analiza je faza u kompajleru u kojoj se proverava semantička ispravnost ulaznog programa. To praktično znači da semantička analiza određuje šta svaka konstrukcija u programu znači, povezuje promenljive sa njihovim deklaracijama i proverava tipove izraza. Semantička analiza se obično deli na:

- Analizu opsega – gde se određuje šta koje ime u programu predstavlja i
- Analizu tipa – gde se određuje da li je tip datog imena odgovarajući.

1.2.4. Generisanje koda

U ovoj fazi ulazni program se prevodi na neki drugi (ciljni) jezik, koji može biti i asemblerski ili mašinski jezik za neku stvarnu ili apstraktnu mašinu. Pri tome generisanje koda za neku apstraktnu mašinu je opštije jer su takvi programi mnogo prenosiviji sa platforme na platformu jer je dovoljno realizovati samo simulator za tu apstraktnu mašinu na datoj platformi.

1.2.5. Optimizacija koda

Optimizacija koda nije obavezan deo kompajlera, ali je poželjno da se izgenerisani kod

malo optimizuje radi efikasnijeg izvršavanja. Razlikujemo tri vrste optimizacija:

- Front-end optimizacija – ova optimizacija se izvodi na izvornom kodu radi bržeg izvršavanja (zamena stepenovanja množenjem, zamena konstantnih izraza nekom konstantom, itd).
- Međuoptimizacija – optimizacija generisanog međukoda
- Back-end optimizacija – ova optimizacija se izvodi na već izgenerisanom kodu. Najviše se postiže pravilnim izborom registara za dati procesor, jer je pristup registrima mnogo brži od pristupa memoriji.

2. Programski jezik Tiger

Programski jezik Tiger je mali, ali netrivijalan jezik sa ugnježđenim funkcijama, slogovima sa implicitnim pokazivačima, nizovima, celobrojnim i stringovskim promenljivima i sa nekoliko jednostavnih kontrolnih konstrukcija.

2.1. Leksičke napomene

Identifikator je niz slova, brojeva i znakova za podvlačenje, koji započinje slovom. Velika slova se razlikuju od malih.

Komentar može da se javi između bilo koja dva tokena. Komentari počinju sa /*, a završavaju se sa */ i mogu biti ugnježđeni.

2.2. Deklaracije

Deklaracija predstavlja niz deklaracija tipova, promenljivih i funkcija. Nikakvi specijalni znaci ne odvajaju niti okončavaju pojedine deklaracije.

Decs ::= *{Dec}.*
Dec ::= *tydec* |
vardec |
fundec.

2.2.1. Tipovi podataka

Sintaksa tipova i deklaracija tipova je sledeća:

tydec ::= *“type” id “=” ty.*
ty ::= *id |*
“{“ tyfields “}” |
“array of” id.
tyfields ::= *|*
id “:” type_id { “,” id “:” type_id }.

Dva imenovana tipa *int* i *string* su predefinisani. Dodatni nazivi tipova mogu biti definisani ili redefinisani (uključujući i predefinisane) pomoću deklaracije tipova.

Slogovi su definisani nabrajanjem njihovih polja u okviru vitičastih zagrada, kod kojih je svako polje opisano *fieldname “:” type_id*, gde je *type_id* identifikator definisan deklaracijom tipa.

Niz bilo kog imenovanog tipa može da bude deklarisan sa “*array of*” *type_id*. Dužina tipa nije specificirana kao deo deklaracije – svaki niz tog tipa može da ima drugačiju dužinu, a dužina će biti određena kreiranjem niza za vreme izvođenja.

Svaka deklaracija sloga ili niza kreira novi tip, nesaglasan sa svim drugim slogovima ili nizovima (čak iako su sva polja identična).

Skup tipova može da bude rekurzivan ili uzajamno rekurzivan. Uzajamno rekurzivni tipovi se deklarišu neprekidnim nizom deklaracija tipova, među kojima nema deklaracija promenljivih ili funkcija.

Različiti slogovi mogu da koriste iste nazive polja.

2.2.2. Promenljive

vardec ::= *“var” id “:=” exp |*
“var” id “:” type_id “:=” exp.

U kraćem obliku deklaracije promenljive dat je naziv promenljive, nakon čega sledi izraz koji predstavlja inicijalnu vrednost promenljive. U ovom slučaju tip promenljive se određuje iz tipa izraza.

U dužem obliku tip promenljive je eksplicitno zadat. Izraz mora biti istog tog tipa. Ukoliko je inicijalni izraz *nil*, tada se mora koristiti ovaj duži oblik.

Svaka deklaracija promenljive kreira jednu novu promenljivu, koja traje koliko i opseg deklaracije.

2.2.3. Funkcije

fundec ::= *“function” id “(“ tyfields”) =” exp |*
“function” id “(“ tyfields”) :” type_id “=” exp.

Prvi oblik predstavlja deklaraciju procedure, a drugi deklaraciju funkcije. Procedure ne vraćaju nikakvu vrednost za razliku od funkcija. Tip vrednosti koju vraća funkcija je naznačen iza dvotačke. Izraz *exp* predstavlja telo procedure ili funkcije, a *tyfields* opisuje imena i tipove parametara. Svi parametri se prosleđuju po vrednosti.

Funkcije mogu biti rekurzivne. Uzajamno rekurzivne funkcije i procedure se deklarišu neprekidnim nizom deklaracija funkcija.

2.2.4. Pravila opsega

U izrazu *“let”....vardec....”in” exps “end”* opseg promenljive traje od njene deklaracije pa do *end*.

U izrazu *“function” id “(“....id1 ”:” id2....”)=” exp* opseg parametra *id1* proseže se celim telom funkcije.

Opseg promenljive ili parametra uključuje i tela svih funkcija definisanih u tom opsegu. To znači da je pristup promenljivoj dozvoljen i u spoljnom opsegu.

U izrazu *“let”....tydec....”in” exps “end”* opseg imena tipa počinje na početku niza deklaracija tipova i traje do *end*. Ovim je omogućena i definicija rekurzivnih tipova.

U izrazu *“let”....fundec....”in” exps “end”* opseg imena funkcije počinje na početku niza deklaracija funkcija, a traje sve do *end*. Ovo uključuje i zaglavlja i tela svih funkcija u tom opsegu.

Postoje dva različita tipa imena: jedan za tipove, a drugi za funkcije i promenljive. Na primer tip *a* može biti istovremeno u opsegu sa promenljivom *a* ili sa funkcijom *a*. Međutim, promenljive i funkcije istog imena ne mogu biti istovremeno u istom opsegu (jedan će sakriti drugog).

2.3. Promenljive i izrazi

2.3.1. L-vrednosti

L-vrednost je lokacija čiju vrednost možemo pročitati ili promeniti. Promenljive, parametri procedura, polja slogova i elementi nizova predstavljaju l-vrednosti.

lvalue ::= *id* |
lvalue “.” id |
lvalue “[“ exp “]”.

Neterminal *id* se odnosi na promenljivu ili parametar dostupan preko pravila opsega. Tačkasta notacija dozvoljava selekciju polja nekog sloga odgovarajućeg naziva. Notacija pomoću uglastih zagrada omogućava selekciju odgovarajućeg elementa niza. Nizovi su indeksirani uzastopnim celim brojevima koji počinju od nule (sve do veličine niza umanjene za jedan).

2.3.2. Izrazi

L-vrednost, kada se koristi kao izraz, vraća sadržinu odgovarajuće lokacije.

Određeni izrazi ne vraćaju nikakvu vrednost: pozivi procedura, naredba dodele, if-then naredba, while naredba, break naredba i ponekad if-then-else naredba. Tako je na primer izraz *(a:=b) + c* sintaksno ispravan pa se greška otkriva tek na proveri tipova.

Izraz *nil* (rezervisana reč) označava vrednost koja može pripadati bilo kojem slogu. Ako slogovska promenljiva sadrži vrednost *nil*, tada pristup polju sloga predstavlja grešku. Izraz *nil* može biti upotrebljen samo u izrazima gde je moguće odrediti tip sloga.

Niz dva ili više izraza, unutar malih zagrada i razdvojenih tačka-zarezom (*exp; exp;...exp*) izračunava vrednost svih izraza u datom redosledu. Rezultat ove sekvence predstavlja rezultat poslednjeg izraza (ukoliko ga ima).

Niz decimalnih cifara predstavlja celobrojnu konstantu, koja predstavlja odgovarajući ceo broj. Stringovska konstanta je niz (od nula ili više) vidljivih simbola, razmaka ili specijalnih stringova oivičenih znacima navoda. Svaki specijalni string započinje znakom \ i stoji umesto niza karaktera. Samo sledeći specijalni stringovi su dozvoljeni (sve ostale upotrebe znaka \ su zabranjene):

\n	Znak za kraj reda.
\t	Znak za tab.
\ ^c	Kontrolni karakter, za neko pogodno *c*.
\ddd	Jedan karakter sa ASCII kodom ddd.
\”	Znak za navodnike.

\\ Znak \.

Izraz oblika:

id “( )” ili

id “(“ exp {“,” exp} “)”

ukazuje na primenu funkcije *id* na listu stvarnih parametara dobijenih izračunavanjem izraza sa leva na desno. Stvarni parametri moraju odgovarati formalnim parametrima u deklaraciji funkcije. Ako je *id* ime procedure onda telo te procedure ne sme da vrati nikakvu vrednost.

Ispred svakog celobrojnog izraza može da stoji znak – (unarni minus).

Izrazi oblika:

exp op exp

gde *op* može biti +,-,*,/ zahtevaju celobrojne izraze *exp* kao svoje argumente i daju celobrojne rezultate.

Izrazi oblika:

exp op exp

gde *op* može biti =, <, <=, >, >=, < > porede svoje operande i vraćaju rezultat 1, ako je relacija tačna, a 0 ako je relacija netačna. Ovi operatori mogu biti primenjeni na celobrojne operande. Operatori za jednakost ili nejednakost mogu takođe biti primenjeni na slogove ili nizove poredeći jednakost ili nejednakost po referenci (proveravaju da li su dva sloga iste pojave, a ne da li imaju istu sadržinu).

Operatori poređenja mogu takođe biti primenjeni na stringove. Dva stringa su jednaka ako su im sadržaji jednaki. Ostali operatori se odnose na leksikografski poredak.

Izrazi oblika:

exp op exp

gde *op* može biti & ili | predstavljaju operacije logičke konjukcije i disjunkcije. Izraz sa desne strane se ne izračunava ako se vrednost celokupnog izraza može odrediti samo iz vrednosti izraza sa desne strane (lenjo izračunavanje). Svaki broj veći od nule predstavlja tačnu vrednost, dok nula predstavlja vrednost-netačno.

Unarni minus ima najveći prioritet. Zatim, operatori * i / imaju sledeći po redu najveći prioritet, nakon koje dolaze + i - , potom relacioni operatori =, <, <=, >, >=, < > , onda &, a na kraju |.

Operatori *,/,- i + su svi levo asocijativni. Operatori poređenja nisu asocijativni pa je na primer izraz *a = b = c* sintaksno neispravan, dok je *(a = b) = c* sintaksno ispravan izraz.

Izraz oblika:

type_id “{“ id “=” exp { “,” id “=” exp } “}”

kreira novu pojavu sloga tipa *type_id*. Nazivi i tipovi polja moraju se slagati sa onima

koji su dati u definiciji sloga (i po broju i po redosledu i po tipu).

Izraz oblika:

type_id "[" exp1"] of" exp2

izračunava vrednosti *exp1* i *exp2* i kreira novi niz. Tip *type_id* mora biti deklarisan kao niz. Rezultat ovog izraza je novi niz tipa *type_id* indeksiran od nula do *exp1*-1, u kome je vrednost svakog elementa postavljena na inicijalnu vrednost *exp2*.

Kada slogovskoj ili nizovnoj promenljivoj *a* dodelimo vrednost *b* tada *a* pokazuje na isti slog ili niz kao i *b*. Buduće izmene sadržaja *a* će uticati i na *b* i obrnuto.

Naredba dodeljivanja ima sledeći oblik:

lvalue ":=" exp.

Ova naredba se izvršava na standardni način: prvo se izračunava lokacija *lvalue*, zatim se izračunava vrednost *exp* i na kraju se u *lvalue* postavlja vrednost *exp*. Sintaksno operator ":=" vezuje još slabije od |. Naredba dodeljivanja ne vraća nikakvu vrednost, tako da je izraz *(a := b) + c* semantički neispravan.

If naredba ima sledeći oblik:

"if" exp1 "then" exp2 ["else" exp3]

Prvo se izračunava vrednost izraza *exp1* , pa ako je ta vrednost različita od nule onda se izvršava izraz *exp2*, a ako je vrednost nula onda se izvršava izraz *exp3* (ukoliko postoji). Izrazi *exp2* i *exp3* moraju da vrate isti tip ili da ne vrate ništa.

Izraz:

"while" exp1 "do" exp2

izračunava izraz *exp1*, pa ako je rezultat različit od nule tada se izvršava izraz *exp2* i nakon toga se ceo while izraz ponovo izračunava.

For naredba u Tiger-u izgleda ovako:

"for" id ":=" exp1 "to" exp2 "do" exp3.

Ovom naredbom se izraz *exp3* izračunava za svaki ceo broj između *exp1* i *exp2*. Promenljiva *id* je nova promenljiva specijalno kreirana za for naredbu, čiji je opseg pokriva samo *exp3*. Izraz *exp3* ne sme da vrati nikakvu vrednost. Gornja i donja granica se računaju samo jednom i to pri ulasku u petlju. Ako je gornja granica veća od donje telo petlje se ne izvršava.

Naredba *break* prekida izvršavanje najbliže while ili for petlje. Nije dozvoljena pojava naredbe *break* koja nije unutar while ili for petlje.

Izraz:

"let" decs "in" expseq "end"

izračunava deklaracije *decs* i vezuje tipove, promenljive i funkcije čiji je opseg *expseq*. *expseq* je niz izraza, razdvojenih tačka-zarezom. Rezultat poslednjeg izraza u *expseq* je ujedno i rezultat celog let izraza.

2.3.3. Programi

Programi u programskom jeziku Tiger nemaju argumente. Program je samo izraz *exp*.

2.4. Standardna biblioteka

Sledeće funkcije su predefinisane:

```
function print(s : string)
```
Štampa string s na standardni izlaz.

```
function flush()
```
Ispisuje sadržaj izlaznog bafera na standardni izlaz.

```
function getchar() : string
```
Čita karakter sa standardnog ulaza.

```
function ord(s : string) : int
```
Vraća ASCII kod prvog karaktera u stringu `s`.

```
function chr(i : int) : string
```
Vraća karakter čiji je ASCII kod `i.`

```
function size(s : string) : int
```
Vraća dužinu stringa `s.`

```
function substring(s:string, first:int, n:int): string
```
Vraća podstring stringa `s` počevši od `first` dužine `n`.

```
function concat(s1: string,s2: string) : string
```
Spaja string `s1` sa stringom `s2.`

```
function not(i : int) : int
```
Vraća logičku vrednost da li je `i=0`.

```
function exit(i : int)
```
Završava izvršavanje sa kodom `i.`

3. Leksički analizator za programski jezik Tiger

U ovoj implementaciji kompajlera za programski jezik Tiger za realizaciju leksičkog analizatora korišćen je generator leksičkog analizatora "Jlex".

Iako je u prvoj glavi napomenuto da generatori leksičkog analizatora generišu neefikasan kod, to ne mora uvek da bude tako. Tvorci Jlex-a su vršili poređenje brzine leksičkog analizatora generisanog Jlex-om i nekog leksičkog analizatora koji je "ručno" pisan u programskom jeziku Java. Oni su merili vreme izvođenja leksičke analize na dve datoteke i evo kakve rezultate su dobili:

DUŽINA IZVORNOG FAJLA	LEKSIČKI ANALIZATOR REALIZOVAN POMOĆU JLEX-A	"RUČNO"PISAN LEKSIČKI ANALIZATOR U JAVI
177 linija	0.42 sekunde	0.53 sekunde
897 linija	0.98 sekunde	1.28 sekunde

Kao što se vidi iz tabele leksički analizator generisan pomoću Jlex-a se pokazao kao bolji i efikasniji.

Verovatno najpoznatiji generator leksičkog analizatora je Lex. On je napravljen za operativni sistem UNIX, a generiše kod u programskom jeziku C.

Jlex je upravo baziran na generatoru leksičkog analizatora Lex-u. On kao ulaz uzima fajl u kome je sadržana specifikacija nekog leksičkog analizatora a zatim generiše kod u Javi koji odgovara datom leksičkom analizatoru.

U ovom poglavlju će istovremeno biti objašnjena struktura specifikacionog fajla za Jlex, kao i realizacija konkretnog programskog jezika Tiger. Međutim, pre svega će prvo biti objašnjen paket za ispisivanje poruka o grešakama jer se on koristi i u ovom poglavlju, a i u sledećem.

3.1. Ispisivanje poruka o greškama

Sve rutine za ispisivanje poruka o greškama nalaze se u paketu `ErrorMsg`. Ovaj paket je bitan jer se poziva i iz leksičkog analizatora i iz sintaksnog analizatora. Struktura samog paketa je dosta jednostavna i sastoji se od dve klase: `LineList` i `ErrorMsg`.

3.1.1. Klasa LineList

Klasa LineList sadrži običnu implementaciju povezane liste i izgleda ovako:

```
class LineList {
  int head;
  LineList tail;
  LineList(int h, LineList t) {head=h; tail=t;}
}
```

U promenljivoj `head` se čuva pozicija zadnjeg karaktera u prošlom redu. Ovo je bitno jer klasa `ErrorMsg` dobija samo informaciju o poziciji karaktera kod koga je došlo do greške, a ne i o liniji u kojoj je došlo do greške. Klasa `LineList` nam služi da iz informacije o poziciji greške možemo izračunati i liniju u kojoj je došlo do greške. Informacija o zadnjem karakteru u redu se ažurira metodom `newline` (iz klase `ErrorMsg`), a ta metoda se poziva svaki put kad se naiđe na kraj reda.

3.1.2. Klasa ErrorMsg

Klasa `ErrorMsg` izgleda ovako:

```
public class ErrorMsg {
  private LineList linePos = new LineList(-1,null);
  private int lineNum=1;
  private String filename;
  public boolean anyErrors;
  public static int errorNum = 0;

  public ErrorMsg(String f) {
      filename=f;
  }

  public void newline(int pos) {
     lineNum++;
     linePos = new LineList(pos,linePos);
  }
  public void error(int pos, String msg) {
      int n = lineNum;
```

```
      LineList p = linePos;
       String sayPos="0.0";
       errorNum = errorNum + 1;
       anyErrors=true;

         while (p!=null) {
           if (p.head<pos) {
            sayPos = ":" + "(Line:" + String.valueOf(n) +
                     "." + "Column:" +
                     String.valueOf(pos-p.head) + ")";
            break;
           }
            p=p.tail; n--;
         }

       System.out.println(filename + ":" + sayPos +
                                    ": " + msg);
  }
}
```

Prpmenljive u ovoj klasi su sledeće:

- `linePos`-pokazuje na listu pozicija zadnjih karaktera u redu
- `lineNum`-broji redove u ulaznom fajlu
- `filename`-čuva ime ulaznog fajla (zbog ispisa)
- `anyErrors`-da li je bilo grešaka
- `errorNum`-broji greške.

Konstruktor ove klase ima samo jedan parametar, a to je ime ulaznog fajla što nam je bitno zbog ispisa.

Metod `newline` ima takođe samo jedan parametar-`pos` u kojem se prosleđuje pozicija zadnjeg karaktera u prethodnom redu. U ovoj metodi se povećava brojač linija-`lineNum`, a vrednost parametra pos se stavlja na početak liste.

Metoda `error` ovde radi glavni deo posla. Ona pregleda listu sve dok vrednost glave liste ne bude manje od prosleđene pozicije (i istovremeno smanjuje broj reda). Kada je glava liste manja od prosledjene pozicije znači da je nađena tražena linija. Tada se na ekran ispisuje ime ulaznog fajla, linija, kolona kao i neka dodatna poruka.

3.2. Struktura specifikacije leksičkog analizatora

Ulazni fajl, koji sadrži specifikaciju leksičkog analizatora, je organizovan u tri zasebna dela razdvojenih dvostrukim znakom za procenat (“%%”), koji moraju biti smešteni na početku linije. Ispravna Jlex specifikacija ima sledeću strukturu:

Korisnički kod
%%
JLex direktive
%%
Regularni izrazi

Prvi deo specifikacije – *Korisnički kod* – se kopira direktno u rezultujući izlazni fajl. Ovaj deo specifikacije omogućava prostor za implementaciju uslužnih klasa ili tipova podataka.

Drugi deo specifikacije – *JLex direktive* – služi za definiciju makroa, novih stanja, kao i za definiciju nekih dodatnih ponašanja leksičkog analizatora.

Treći deo specifikacije – *Regularni izrazi* – sadrži pravila za leksičku analizu. Svako pravilo se sastoji od tri dela: opcione liste stanja, regularnog izraza i akcije.

3.2.1. Korisnički kod

Korisnički kod prethodi prvom duplom znaku za procenat ("%%"). Ovaj kod se kopira direktno u izvorni fajl leksičkog analizatora koji generiše Jlex (na sam početak generisanog fajla). Na primer, ako generisani fajl treba da počne sa deklaracijom paketa ili sa uvoznom listom nekih klasa tada ove informacije treba staviti u korisnički kod. Pored ovoga u korisnički kod je moguće staviti i čitave definicije klasa.

Za specifikaciju leksičkog analizatora za programski jezik Tiger u odeljku za korisnički kod potrebno je samo navesti da sve izgenerisane klase pripadaju paketu `Parser`. Korisnički kod u ovom slučaju sadrži samo deklaraciju paketa i izgleda ovako:

```
package Parser;

%%
```

NAPOMENA: Neki obimniji izgled korisničkog koda moguće je pogledati u paketu Lexer na disketi koja ide uz ovaj diplomski rad. U ovom paketu leksički analizator je realizovan samostalno, nezavisno od sintaksnog analizatora

3.2.2. JLex direktive

Odeljak za JLex direktive počinje posle prve pojave "%%" i traje sve do sledeće pojave "%%". Svaka JLex direktiva bi trebalo da počinje na početku linije.

U ovom odeljku će biti opisane samo direktive koje su korišćene pri realizaciji leksičkog analizatora za programski jezik Tiger. Kompletan spisak direktiva se može naći na internet adresi http:/www.cs.princeton.edu/~appel/modern/java/JLex.

3.2.2.1. Brojanje karaktera

Brojanje karaktera je inicijalno isključeno, a aktivira se sledećom direktivom:
%char
Pozicija prvog karaktera u prepoznatom tokenu se čuva u celobrojnoj promenljivoj *yychar*.

3.2.2.2. Brojanje linija

Brojanje linija je takođe inicijalno isključeno, a aktivira se sledećom direktivom:
%line
Linija u kojoj je prepoznat token se čuva u celobrojnoj promenljivoj *yyline*.

3.2.2.3. Nekompatibilnost operativnih sistema

U operativnom sistemu UNIX, kraj reda se označava samo jednim karakterom ”/n” (newline), dok se u operativnim sistemima baziranim na DOS-u kraj reda označava sa dva karaktera “/r/n” (carriage return+newline). Direktiva oblika:
%notunix
ima za rezultat da se ili “/r” ili “/n” prepoznaju kao kraj reda.

3.2.2.4. Velika i mala slova

Direktiva:
%ignorecase
se zadaje kada želimo da napravimo leksički analizator koji će prepoznavati tokene bez obzira da li su napisana malim ili velikim slovima.

3.2.2.5. Deklaracija stanja

Leksička stanja se uvode da bi se kontrolisalo kada se koji regularni izraz prepoznaje. Stanja se deklarišu sledećom direktivom:
*%state **ime {,ime}***
Imena bi trebalo da budu standardni identifikatori tj. niz slova,cifara i “_” koji počinje sa slovom.

Pravila leksičkog analizatora počinju sa opcionom listom stanja. Ako je lista zadata, to pravilo se koristi samo ako je leksički analizator u nekom od tih stanja. Nasuprot tome, ako lista nije zadata pravilo se koristi bez obzira u ko je stanju leksički analizator.

JLex ima jedno inicijalno stanje *YYINITIAL*. Ako se ne uvode nova stanja, leksički analizator će ostati u inicijalnom stanju i prepoznavaće sve regularne izraze kako se očekuje. Međutim, ovde se stanja uvode da bi se prepoznali ugnježđeni komentari. Prevođenje leksičkog analizatora iz jednog stanja u drugo se vrši pozivom metoda *yybegin (**imeStanja**)*.

3.2.2.6. Kompatibilnost sa generatorom sintaksnog analizatora "CUP"

Java CUP je generator sintaksnog analizatora koji je korišćen za sintaksnu analizu programskog jezika Tiger. Da bi leksički i sintaksni analizator komunicirali moraju biti ispunjeni neki uslovi. Te uslove definiše direktiva oblika:
%cup
Rezultat ove direktive je:

- Da se glavna metoda leksičkog analizatora zove *next_token*
- Da leksički analizator implementira interfejs *java_cup.runtime.Scanner* (koji je definisan u Java CUP-u)
- Da glavna funkcija vraća vrednosti tipa *java_cup.runtime.Symbol* (koji je takođe definisan u Java CUP-u)

3.2.2.7. Interni kod za leksički analizator

Direktiva oblika:
%{
code
%}
omogućava korisniku da piše Java kod direktno u glavnu klasu leksičkog analizatora (*Ylex*). Ovo omogućava deklaraciju promenljivih i metoda koji će biti interni za generisanu klasu leksičkog analizatora.

Za leksički analizator programskog jezika Tiger interni kod izgleda ovako:

```
%{
  private ErrorMsg.ErrorMsg errorMsg;

  private int comment_count = 0;
 errorMsg.newline(yychar);
  }

  private java_cup.runtime.Symbol tok(int index) {
    return new java_cup.runtime.Symbol(index,yychar,
                                       yychar+yylength(),yytext());
  }

  Yylex(java.io.InputStream s, ErrorMsg.ErrorMsg e) {
  this(s);
```

```
  errorMsg=e;
  }
%}
```

Ovde je napravljen novi konstruktor za glavnu klasu leksičkog analizatora (*Yylex*). On postavlja lokalnu promenljivu `errorMsg` na već napravljenu instancu klase `ErrorMsg`, a zatim poziva inicijalni konstruktor klase `Yylex`.

Metoda `newline` poziva istoimenu metodu iz klase `ErrorMsg` sa parametrom `yychar` (u kome se nalazi pozicija prepoznatog tokena).

Metod `tok` poziva konstruktor klase `Symbol` sa parametrima: `index`, `yychar`, `yychar+yylength()` i `yytext()`. Metode `yylength()` i `yytext()` vraćaju dužinu i vrednost prepoznatog tokena.

Ovde je još deklarisana privatna promenljiva `comment_count` koja broji dubinu ugnježđenosti komentara. Ona se inicijalizuje na nula, a na kraju leksičke analize mora takođe da bude nula.

3.2.2.8. Vraćanje vrednosti na kraju izvršavanja

Direktiva oblika:
%eofval{
code
%eofval}
omogućava korisniku da piše Java kod koji će se izvršiti odmah nakon je dostignut EOF. Ovaj kod mora da vrati vrednost kompatibilnu sa tipom koji vraća leksički analizator.

U ovoj realizaciji leksičkog analizatora za programski jezik Tiger ovaj deo izgleda ovako:

```
%eofval{

      {if (comment_count>0)
           errorMsg.error (yychar,
                           “Unclosed comment at the end of file”);
       return tok(sym.EOF);
        }
%eofval}
```

Ovde se prvo proverava da li je promenljiva `comment_count` veća od nule, pa ako jeste ispisujemo grešku jer se leksički analizator nalazi još u komentaru.

3.2.2.9. Definicije makroa

Makroi se koriste za lakše zapisivanje nekih regularnih izraza. Definicija makroa izgleda ovako:

ime = definicija
Svaka definicija makroa se sastoji od imena makroa i od njemu pridružene definicije. Imena makroa predstavljaju standardne identifikatore, dok definicije predstavljaju regularne izraze (pravila za regularne izraze su opisana u sledećem poglavlju).

Definicije makroa mogu sadržati imena drugih makroa, ali uzajamno rekurzivne definicije su zabranjene.

Za leksički analizator programskog jezika Tiger potrebni su sledeći makroi:

```
ALPHA=[A-Za-z]
DIGIT=[0-9]
NONNEWLINE_WHITE_SPACE_CHAR=[\ \t\b\012]
WHITE_SPACE_CHAR=[\n\ \t\b\012]
STRING_TEXT=(\\\"|[^\n\"]|\\{WHITE_SPACE_CHAR}+\\)*
COMMENT_TEXT=([^/*\n]|[^*\n]"/"[^*\n]|[^/\n]"*"[^/\n]|
                      "*"[^/\n]|"/"[^*\n])*
```

Sama značenja ovih makroa će biti jasnija posle narednog poglavlja.

3.2.3. Regularni izrazi

Treći deo JLex specifikacije se sastoji od liste pravila za prepoznavanje tokena. Ova pravila se specificiraju regularnim izrazima. Pravila se sastoje od tri dela: opcione liste stanja, regularnog izraza i akcije pridružene tom regularnom izrazu.
[lista_stanja] reg_izraz { akcija }

Ako se ulaz podudara sa više različitih pravila, leksički analizator to rešava izborom pravila koji se podudara sa najdužim stringom. Ako se više izraza podudara sa ulaznim stringom iste dužine, leksički analizator izabira ono pravilo koje se javlja prvo u specifikaciji. Znači, pravila koja se pojavljuju ranije u specifikaciji imaju veći prioritet.

3.2.3.1. Lista stanja

Opciona lista stanja može prethoditi svakom regularnom izrazu. Ona ima sledeću formu:
<imeStanja {,imeStanja}>

Ako je lista stanja navedena leksički analizator će pokušati da upotrebi regularni izraz samo ako se nalazi u datom stanju. Nasuprot tome, ako lista stanja nije navedena onda će se taj regularni izraz upotrebiti bez obzira u kom se stanju nalazi leksički analizator.

3.2.3.2. Pravila za regularne izraze

Sledeći karakteri su metakarakteri i imaju specijalno značenje u JLex regularnim izrazima:

```
? * + | ( ) ^ $ / ; . =  [ ] { } " \
```

Osim ovih ostali samostalni karakteri predstavljaju sami sebe. Sledeći specijalni stringovi se mogu koristiti u regularnim izrazima:

`\b`	Backspace
`\n`	Newline
`\t`	Tab
`\f`	Formfeed
`\r`	Carriage return
`\ddd`	Karakter koji odgovara broju prikazanom preko tri oktalne cifre.
`\xdd`	Karakter koji odgovara broju prikazanom preko dve heksadecimalne cifre.
`\udddd`	Unicode karakter koji odgovara broju prikazanom preko četiri heksadecimalne cifre.
`\^C`	Kontrolni karakter
`\c`	Svaki karakter posle backslash-a je sam taj karakter

Dva uzastopna regularna izraza (*ef*) ukazuju na konkatenaciju tih izraza. U nastavku će biti navedene uloge i korišćenje metakaraktera.

- Oznaka za dolar ($) označava kraj linije. Ako je oznaka za dolar na kraju regularnog izraza, taj izraz se prepoznaje samo ako se nalazi na kraju linije.
- Tačka predstavlja regularni izraz koji prihvata svaki karakter osim newline.
- Metakarakteri gube svoje značenje ako se nalaze unutar duplih navodnika.
- Vitičaste zagrade ukazuju na poziv makroa sa imenom navedenim unutar vitičastih zagrada.
- Uspravna crta označava alternativu između dva regularna izraza (*e*|*f*).
- Zvezda (*) označava Klinijevo zatvorenje tj. prihvatanje nula ili više ponavljanja regularnih izraza.
- Znak plus (+) označava prihvatanje jednog ili više ponavljanje regularnog izraza.
- Upitnik (*?*) označava prihvatanje nijednog ili jednog regularnog izraza.
- Zagrade ( ) označavaju grupisanje regularnih izraza.
- Uglaste zagrade [ ] označavaju prihvatanje samo jednog karaktera od onih koji su navedeni unutar zagrada. Ako je prvi karakter posle otvorene zagrade (^), on označava da se prihvata bilo koji karakter osim onih koji su navedeni

unutar zagrada. Unutar uglastih zagrada je moguće koristiti sledeće metakaraktere:

{ ime }	Poziv makroa.
a - b	Skup karaktera od “a” do “b”
"..."	Svi metakarakteri između duplih navodnika gube svoje specijalno značenje.
\	Svi metakarakteri iza backslash-a (\) gube svoje specijalno značenje.

3.2.3.3. Akcije pridružene regularnom izrazu

Akcija, pridružena nekom regularnom, izrazu se sastoji od Java koda koji se nalazi u okviru vitičastih zagrada. Taj Java kod se kopira direktno u generisani fajl, a izvršava se odmah po prepoznavanju regularnog izraza. Obično se podrazumeva da ovaj kod vrati neku vrednost koju će prihvatiti sintaksni analizator. Ako ovaj kod ne vraća nikakvu vrednost leksički analizator će nastaviti da prepoznaje tokene sve dok ne naiđe na akciju u kojoj se vraća neka vrednost.

3.2.3.4. Regularni izrazi za programski jezik Tiger

Vrednosti koje se prosleđuju sintaksnom analizatoru su tipa *java_cup.runtime.Symbol*. Klasa *Symbol* ima sledeću strukturu:

```
package java_cup.runtime;

public class Symbol {
  public int sym;
  public int parse_state;
  boolean used_by_parser = false;
  public int left, right;
  public Object value;
}
```

Za lakše prosleđivanje je korišćena funkcija tok koja je opisana u poglavlju 3.2.2.7. Ova funkcija ima samo jedan celobrojan parametar u kome se prosleđuje indeks prepoznatog tokena. Indeks predstavlja internu reprezentaciju tokena u kompajleru. Vrednosti indeksa se nalaze u klasi *sym* (koju generiše sintaksni analizator). Klasa *sym* ima sledeću strukturu:

```
package Parser;

/** CUP generated class containing symbol constants. */
public class sym {
```

```
  /* terminals */
  public static final int FUNCTION = 2;
  public static final int EOF = 0;
  public static final int INT = 3;
  public static final int COLON = 6;
  public static final int DIVIDE = 5;
  public static final int GT = 4;
  public static final int OR = 8;
  public static final int ELSE = 7;
  public static final int NIL = 9;
  public static final int GE = 11;
  public static final int DO = 10;
  public static final int error = 1;
  public static final int LT = 12;
  public static final int OF = 13;
  public static final int UMINUS = 15;
  public static final int MINUS = 14;
.........
```

Rezervisane reči u programskom jeziku Tiger su: `WHILE, FOR, TO, BREAK, LET, IN, END, FUNCTION, VAR, TYPE, ARRAY, IF, THEN, ELSE, DO, OF i NIL`. Prepoznavanje rezervisanih reči se jednostavno specificira u JLex-u:

```
<YYINITIAL> "function"        { return tok(sym.FUNCTION); }
<YYINITIAL> "else"            { return tok(sym.ELSE); }
<YYINITIAL> "nil"             { return tok(sym.NIL); }
<YYINITIAL> "do"              { return tok(sym.DO); }
<YYINITIAL> "of"              { return tok(sym.OF); }
<YYINITIAL> "array"           { return tok(sym.ARRAY); }
<YYINITIAL> "type"            { return tok(sym.TYPE); }
.......
```

Specijalni simboli u programskom jezikuTiger su: `, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=`. Prepoznavanje specijalnih simbola se takođe jednostavno realizuje u JLex-u:

```
<YYINITIAL> ","   { return tok(sym.COMMA); }
<YYINITIAL> ":"   { return tok(sym.COLON); }
<YYINITIAL> ";"   { return tok(sym.SEMICOLON); }
<YYINITIAL> "("   { return tok(sym.LPAREN); }
<YYINITIAL> ")"   { return tok(sym.RPAREN); }
<YYINITIAL> "["   { return tok(sym.LBRACK); }
<YYINITIAL> "]"   { return tok(sym.RBRACK); }
.......
```

Kada prepoznamo neke delimitere ne treba ništa da se radi, jer oni ne predstavljaju elemente jezika. Za prepoznavanje delimitera se koristi makro *NONNEWLINE_WHITE_SPACE_CHAR*.

```
<YYINITIAL> {NONNEWLINE_WHITE_SPACE_CHAR}+ { }
```

Kada prepoznamo kraj reda tada treba pozvati metod *newline* iz klase *ErrorMsg*. Ovo treba raditi bilo da se nalazimo u stanju *YYINITIAL*, ili u stanju *COMMENT* (jer se redovi

broje u celom fajlu).

```
<YYINITIAL> \n { newline(); }

<COMMENT>   \n { newline(); }
```

Kada se leksički analitator nalazi u inicijalnom stanju, a naiđe se na početak komentara (/*), tada treba promeniti stanje leksičkog analizatora (pozivom metoda *yybegin(...)*). Međutim, ako se leksički analizator već nalazi u stanju *COMMENT* tada samo treba povećati vrednost promenljive *comment_count.*

```
<YYINITIAL> "/*" { yybegin(COMMENT);
                   comment_count = comment_count + 1; }

<COMMENT> "/*" { comment_count = comment_count + 1; }
```

Kada se leksički analizator nalazi u stanju *COMMENT*, a naiđe se na znak za kraj komentara (*/), tada treba smanjiti vrednost promenljive *comment_count*. Ako je posle toga vrednost promenljive jednaka nuli, tada se leksički analizator vraća u inicijalno stanje.

```
<COMMENT> "*/" {
      comment_count = comment_count - 1;
      Utility.assert(comment_count >= 0);
      if (comment_count == 0) {
            yybegin(YYINITIAL);
      }
}
```

NAPOMENA: U klasi Utility se nalaze neke pomoćne procedure koje nisu od značaja u razumevanju rada kompajlera.

Kada se leksički analizator nalazi u stanju *COMMENT* i kada se naiđe na regularan tekst komentara (koji je definisan makroom *COMMENT_TEXT*), tada ne treba raditi ništa jer se komentari zanemaruju.

```
<COMMENT> {COMMENT_TEXT} { }
```

Kada leksički analizator naiđe na regularan stringovski tekst (koji je definisan makroom *COMMENT_TEXT*) unutar dvostrukih navodnika, tada treba izbaciti navodnike i proslediti sintaksnom analizatoru simbol *STRING*, koji u promenljivoj *value* nosi sadržaj stringa (bez navodnika). Međutim, ako leksički analizator naiđe na stringovski tekst kome prethodi znak navoda, a iza teksta nema znaka navoda, tada treba prijaviti grešku a parseru vratiti simbol string sa vrednošću prepoznatog stringa.

```
<YYINITIAL> \"{STRING_TEXT}\" {
   String str =  yytext().substring(1,yytext().length()-1);
   Utility.assert(str.length() == yytext().length() - 2);
   return (new java_cup.runtime.Symbol(sym.STRING,yychar,
                                       yychar+yylength(),str));
```

```
}

<YYINITIAL> \"{STRING_TEXT} {
   String str =  yytext().substring(1,yytext().length());
   errorMsg.error(yychar,”Unclosed string”);

   return (new java_cup.runtime.Symbol(sym.STRING,yychar,
                                       yylength()+yychar,str));
}
```

Prepoznavanje brojeva (koji mogu biti samo celobrojni) i identifikatora je jednostavno. U ovu svrhu su korišćeni makroi *DIGIT* koji prepoznaje jednu cifaru i *ALPHA* koji prepoznaje jedno slovo ili cifaru.

```
<YYINITIAL> {DIGIT}+ {
      return (new java_cup.runtime.Symbol(sym.INT,yychar,
                                          yychar+yylength(),yytext()));
}

<YYINITIAL> {ALPHA}({ALPHA}|{DIGIT}|_)* {
      return (new java_cup.runtime.Symbol(sym.ID,yychar,
                                          yychar+yylength(),yytext()));
}
```

Ako u toku leksičke analize ništa od prethodno navedenog nije prepoznato to znači da se na ulazu nalazi nepoznati karakter. U tom slučaju na ekran se ispisuje poruka o grešci, a sintaksnom analizatoru se prosleđuje specijalni terminal *error* (koji će biti objašnjen u sledećoj glavi). U ovom slučaju se ne poziva metoda *errorMsg.error*, jer će se ona pozvati iz sintaksnog analizatora čim se primi specijalni terminal *error*. Ako bi i ovde pozvali metod *errorMsg.error*, onda bi se ova greška brojala dva puta.

```
<YYINITIAL,COMMENT> . {
     System.out.println("Illegal character: <" +
                        yytext() + ">");
     return tok(sym.error);
}
```

3.3. Generisanje leksičkog analizatora

Pod pretpostavkom da se u falju *tiger.lex* nalazi ispravna specifikacija leksičkog analizatora, tada se generisanje leksičkog analizatora izvodi sledećom naredbom:

```
java JLex.Main tiger.lex
```

Ako je specifikacija bila ispravna tada se u fajlu *tiger.lex.java* nalazi kod željenog leksičkog analizatora. Izvršna verzija se dobija pozivom Java kompajlera:

```
javac tiger.lex.java
```

4. Sintaksni analizator za programski jezik Tiger

Osnovna slabost LL(k) (Left-to-right parse, Leftmost-derivation, k-symbol lookahead) sintaksnih analizatora je ta što oni moraju predvideti koju produkciju treba uzeti znajući samo sledećih k tokena. Mnogo moćnija tehnika LR(k) (Left-to-right parse, Rightmost-derivation, k-symbol lookahead) sintaksnih analizatora je u mogućnosti da odloži odluku o izboru produkcije dok ne vidi sve ulazne tokene koji odgovaraju celoj desnoj strani neke produkcije. LALR(k) (Look-Ahead LR(k)) sintaksni analizatori su nešto oslabljeni LR(k) sintaksni analizatori, ali zadržavaju većinu dobrih osobina LR(k) sintaksnih analizatora. U praksi je ova razlika beznačajna, a ono što je bitno je da LALR(k) sintaksni analizatori koriste mnogo manje memorije.

U ovoj implementaciji kompajlera za programski jezik Tiger za realizaciju sintaksnog analizatora korišćen je generator sintaksnog analizatora "CUP" (Constructor of Useful Parsers). CUP je sistem za generisanje LALR sintaksnih analizatora iz jednostavne specifikacije. On ima istu ulogu kao i najčešće korišćeni generator YACC. Naime, CUP je napisan u Javi, koristi specifikaciju u koju je uključen Java kod i generiše sintaksni analizator koji je implementiran u Javi.

Kao i u prethodnoj glavi ovde će biti opisani samo oni elementi CUP-a koji koji su korišćeni u realizaciji sintaksnog analizatora za programski jezik Tiger. Za detaljniji opis generatora sintaksnog analizatora CUP-a se može pogledati na Internet adresi http://www.cs.princeton.edu/~appel/modern/java/cup.

4.1. Specifikacija sintakse

Specifikacija sintakse za CUP se sastoji iz sledećih pet delova:

- Deklaracija paketa i uvozna lista,
- Korisnički kod,
- Lista simbola,
- Deklaracija prioriteta i asocijacija i
- Gramatika.

Neki od ovih delova su opcioni. Svi navedeni delovi moraju biti baš u navedenom

rasporedu.

4.1.1. Deklaracija paketa i uvozna lista

Deklaracija paketa i uvozna lista je opcioni deo specifikacije sintakse. Oni imaju istu sintaksu i semantiku kao i u programskom jeziku Java.

Deklaracija paketa kazuje kojem će paketu pripadati generisane klase. Uvozne liste se direktno kopiraju u generisani kod i omogućavaju upotrebu imena iz drugih paketa u raznim delovima sintaksnog analizatora.

Deklaracija paketa i uvozna lista u specifikaciji za programski jezi Tiger izgledaju ovako:

```
package Parser;

import java_cup.runtime.*;
```

4.1.2. Korisnički kod

Posle deklaracije paketa i uvozne liste dolazi niz opcionih deklaracija koje dozvoljavaju korisniku da modifikuje generisani sintaksni analizator.

CUP generiše posebnu skrivenu klasu (kao deo sintaksnog analizatora) u koju se smeštaju sve akcije koje definiše korisnik. Deklaracija oblika:

action code {: ..... :};

omogućava smeštanje akcija baš u ovu klasu. Procedure i promenljive koje se koriste u akcijama u gramatici bi trebalo da se deklarišu u ovom odeljku.

Naredna deklaracija ima oblik:

parser code {: ..... :};

Ova deklaracija omogućava da se promenljive i procedure smeste direktno u glavnu klasu sintaksnog analizatora.

Sledeća deklaracija ima oblik:

init with {: ..... :};

Ova deklaracija omogućava da se implementira kod koji će se izvršiti pre nego što parser zatraži prvi token. Ovde je, na primer, moguće inicijalizovati leksički analizator.

Poslednja deklaracija ima oblik:

scan with {: ..... :}

i omgućava da se definiše na koji način će sintaksni analizator da traži sledeći token od leksičkog analizatora.

U slučaju sintaksnog analizatora za programski jezik Tiger potrebna je samo *parser code* deklaracija:

```
parser code {:

  public void syntax_error(Symbol current) {
    report_error("Syntax error (" + current.value + ")", current);
  }

  public void report_error(String message, Symbol info) {
      Main.errMsg.error(info.left, message);
  }
            :}
```

O metodama *synax_error* i *report_error* će biti reči kasnije te neće ovde biti posebno komentarisane.

4.1.3. Lista simbola

Lista simbola je prvi obavezan deo specifikacije sintakse. Ova deklaracija je neophodna za imenovanje i eventualno definisanje tipova svakog terminalnog ili neterminalnog simbola koji se pojavljuje u gramatici. Svaki terminalni ili neterminalni simbol je predstavljen objektom *Symbol*. Lista simbola ima sledeću sintaksu:

terminal [imeKlase] ime1, ime2, ... ;
non terminal [imeKlase] ime1, ime2, ... ;

Ime *imeKlase* označava tip vrednosti datih terminala ili neterminala. U slučaju da *imeKlase* nije zadato tada navedeni terminali ili neterminali nemaju nikakvu vrednost. Imena terminala i neterminala ne smeju biti rezervisane reči CUP-a: "*code*", "*action*", "*parser*", "*terminal*", "*non*", "*nonterminal*", "*init*", "*scan*", "*with*", "*start*", "*precedence*", "*left*", "*right*", "*nonassoc*", "*import*" i "*package*".

Lista simbola u sintaksnom analizatoru za programski jezik Tiger je:

```
terminal    FUNCTION, INT, GT, DIVIDE, COLON, ELSE, OR, NIL;
terminal    DO, GE, LT, OF, MINUS, UMINUS, ARRAY, TYPE, FOR, TO;
terminal    TIMES, OMMA, LE, IN, END, ASSIGN, STRING, DOT, LPAREN;
terminal    RPAREN, IF, SEMICOLON, ID, WHILE, LBRACK, RBRACK, NEQ;
terminal    VAR, BREAK, AND, PLUS, LBRACE, RBRACE, LET, THEN, EQ;

non terminal      program, exp, lvalue, sequence, expseq, neexpseq;
non terminal      function_call, explist, neexplist, expression;
non terminal      arithmetic_exp, aritm_op, comparison, comp_op;
non terminal      boolean_exp, bool_op, record_creation;
non terminal      initial, neinit, array_creation, assignment;
non terminal      if_exp, else_exp, while_exp, for_exp, break_exp;
non terminal      declarations, declaration, type_decl, type;
non terminal      netype_fields, type_id, var_decl, fun_decl;
non terminal      creation, statement, let_exp, type_fields, brack_exp;
```

4.1.4. Deklaracija prioriteta i asocijacija

U ovom delu, koji je takođe opcion, opisuju se prioriteti i asocijativnost terminala. Ovo je neophodno za sintaksnu analizu dvosmislenih gramatika (Gramatika je dvosmislena ako jedna rečenica može imati više stabala izvođenja). Deklaracija prioriteta i asocijacija ima sledeću sintaksu:

precedence left	*terminal{ , terminal };*
precedence right	*terminal{ , terminal };*
precedence nonassoc	*terminal{ , terminal };*

Terminali koji su navedeni u jednom redu (i razdvojeni zarezima) imaju isti prioritet. Redosled prioriteta, od najvišeg do najnižeg, je od dole na gore. Znači, terminali koji su u poslednjem redu deklaracije imaju najviši prioritet.

Prioritetima se razrešavaju razni konflikti. Na primer, za rečenicu *3+4*8* sintaksni analizator ne zna da li da *3+4* zameni sa neterminalom *exp* ili da stavi * na stek. Međutim, ako naznačimo da * ima veći prioritet od +, tada će *3+4* biti stavljeno na stek i množenje će se izvršiti pre sabiranja.

CUP dodeljuje svakom terminalu neki prioritet. Svaki terminal, koji nije u deklaraciji prioriteta, ima najmanji prioritet. CUP takođe dodeljuje prioritet i svakom pravilu gramatike. Taj prioritet je jednak prioritetu poslednjeg terminala u pravilu. U slučaju da pravilo nema terminala tada ono dobija najniži prioritet. Kada se pojavi konflikt sintaksni analizator određuje ko ima veći prioritet, terminal ili pravilo. Ako terminal ima veći prioritet onda se on stavlja na stek, a ako pravilo ima veći prioritet tada se vrši zamena niza elemenata sa vrha steka sa nekim neterminalom.

Asocijativnost se takođe koristi za razrešavanje konflikata, ali samo u slučaju jednakih prioriteta. Ako je asocijativnost nekog terminala *left*, tada će se u slučaju konflikta vršiti zamena. Na primer, ako je asocijativnost znaka + baš *left*, tada će se rečenica *3+4+5+6+7* analizirati standardno – prvo *3+4*, zatim *+5* i tako do kraja. Nasuprot ovome, ako je asocijativnost terminala *right* tada se taj terminal stavlja na stek, pa će se zamena vršiti sa desna na levo. U slučaju da je asocijativnost terminala + *right*, tada će se prvo računati *6+7* i tako na levo. Terminalu možemo dodeliti i osobinu *nonassoc*. Tada će svake dve uzastopne pojave terminala sa istim prioritetom rezultovati greškom. Ovo je korisno za operatore poređenja. Na primer, ako je terminal == definisan kao *nonassoc* tada će rečenica *6 == 7 == 8* prouzrokovati grešku.

Deklaracija prioriteta i asocijativnosti za programski jezik Tiger izgleda ovako:

```
precedence right ELSE;
precedence left OR;
precedence left AND;
precedence nonassoc EQ, NEQ, GE, GT, LE, LT;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
```

Treba napomenuti da je prva deklaracija navedena da bi se izraz iza terminala *else* odnosio na najbliži *if* izraz. Ostale deklaracije su jasne i proističu iz same definicije programskog jezika Tiger.

4.1.5. Gramatika

Poslednji deo CUP deklaracije predstavlja gramatika. Ovaj deo može početi sa deklaracijom početnog stanja:

*start with **neterminal**;*

Ovim se naznačava da je ***neterminal*** početni neterminal gramatike. Ako ova deklaracija nije navedena, tada se za početni neterminal uzima prvi neterminal u gramatici.

Svako pravilo gramatike se sastoji od leve strane, iza koje sledi znak *::=* i od desne strane. Leva strana predstavlja neterminal koji se opisuje pravilom, dok se desna strana sastoji od niza akcija, terminala, neterminala kao i direktive za dodelu prioriteta pravilu. U slučaju da postoji više pravila za jedan neterminal, tada se oni mogu razdvojiti znakom |.

U pravilima gramatike moguće je navesti i semantičke akcije. Semantičke akcije se navode između simbola *{: ..... :}* , a sadrže Java kod. Ove akcije se izvršavaju kada sintaksni analizator prepozna sve simbole koji se nalaze sa leve strane akcije.

Pravilima je moguće eksplicitno dodeliti prioritet. Ovo se postiže direktivom *%prec*. Dobar primer za ovu direktivu je unarni minus. Unarni minus ima prioritet veći od običnog minusa pa čak i od množenja. Međutm, leksički analizator vraća isti simbol i za običan i za unarni minus. Veći prioritet unarnog minusa se postiže *%prec* direktivom u samom pravilu:

```
arithmetic_exp    ::= MINUS exp  %prec UMINUS
```

Gramatika programskog jezika Tiger (napisana u sintaksi CUP-a) je data u nastavku. Početni simbol gramatike je *program*.

```
program     ::= exp     {: System.out.println("Parsing finished");
                           System.out.println(
                                    ErrorMsg.ErrorMsg.errorNum +
                                    " error(s) detected.");
                        :} ;

exp         ::= NIL
              | INT
              | STRING
              | lvalue
              | sequence
              | function_call
              | expression
```

```
              | creation
              | statement ;

lvalue      ::= ID
              | lvalue DOT ID
              | lvalue LBRACK exp RBRACK
              | brack_exp ;

brack_exp   ::= ID LBRACK exp RBRACK ;

sequence    ::= LPAREN expseq RPAREN ;

expseq      ::=
              | neexpseq ;

neexpseq    ::= exp
              | exp SEMICOLON neexpseq ;

function_call     ::= ID LPAREN explist RPAREN ;

explist     ::=
              | neexplist ;

neexplist   ::= exp
              | neexplist COMMA exp ;

expression  ::= arithmetic_exp
              | comparison
              | boolean_exp ;

arithmetic_exp    ::= MINUS exp  %prec UMINUS
              | exp aritm_op exp ;

aritm_op    ::= PLUS
              | MINUS
              | TIMES
              | DIVIDE ;

comparison  ::= exp comp_op exp ;

comp_op     ::= EQ
              | NEQ
              | LT
              | GT
              | LE
              | GE ;

boolean_exp ::= exp bool_op exp ;

bool_op     ::= AND
              | OR ;

creation    ::= record_creation
              | array_creation ;

record_creation   ::= ID LBRACE initial RBRACE ;
```

```
initial     ::=
              | neinit ;

neinit      ::= ID EQ exp
              | ID EQ exp COMMA neinit ;

array_creation    ::= brack_exp OF exp ;

statement   ::= assignment
              | if_exp
              | while_exp
              | for_exp
              | break_exp
              | let_exp ;

assignment  ::= lvalue ASSIGN exp ;

if_exp      ::= IF exp THEN exp else_exp;

else_exp    ::=
              | ELSE exp ;

while_exp   ::= WHILE exp DO exp ;

for_exp     ::= FOR ID ASSIGN  exp TO exp DO exp ;

break_exp   ::= BREAK ;

let_exp     ::= LET declarations IN expseq END ;

declarations      ::=
              | declaration declarations ;

declaration ::= type_decl
              | var_decl
              | fun_decl ;

type_decl   ::= TYPE ID EQ type ;

type        ::= type_id
              | LBRACE type_fields RBRACE
              | ARRAY OF type_id ;

type_fields ::=
              | netype_fields ;

netype_fields     ::= ID COLON type_id
              | ID COLON type_id COMMA netype_fields ;

type_id     ::= ID ;

var_decl    ::= VAR ID ASSIGN exp
              | VAR ID COLON type_id ASSIGN exp ;

fun_decl    ::= FUNCTION ID LPAREN type_fields RPAREN EQ exp
              | FUNCTION ID LPAREN type_fields RPAREN COLON
                type_id EQ exp ;
```

4.2. Oporavak od grešaka

Veoma važan aspekat pravljenja sintaksnog analizatora je oporavak od grešaka. Nekada se od kompajlera očekuje da u jednom prolazu otkrije što više grešaka, a ne da stane kod prve.

CUP koristi isti mehanizam za oporavak od grešaka kao i YACC. U principu on podržava specijalni simbol *error*. Ovaj simbol ima ulogu specijalnog neterminala koji, umesto da je definisam nekim pravilom, može da prihvati dugačak niz ulaznih simbola.

Specijalni simbol *error* se koristi jedino ako je sintaksna greška otkrivena. Kad se sintaksna greška otkrije, tada sintaksni analizator pokušava da zameni niz ulaznih tokena sa specijalnim simbolom *error*.

Kao primer se može uzeti pravilo iz gramatike:

```
neexpseq    ::= exp
              | exp SEMICOLON neexpseq
              | error SEMICOLON neexpseq ;
```

Ovaj primer ukazuje da, ako ni jedno pravilo za *exp* ne može da se iskoristi za ulazni string, tada se prijavljuje sintaksna greška i oporavak počinje. Oporavak od greške se izvršava tako što sintaksni analizator preskače sve ulazne tokene sve dok ne dođe do znaka-tačkazarez. Posle toga sintaksna analiza može uspešno da se nastavi jedino ako se prepozna narednih nekoliko tokena (inicijalno 3).

Specijalni simbol *error* bi mogao da se stavi u svako pravilo gramatike, međutim time se gubi struktuiranost i lep izgled gramatike.

4.3. Generisanje i modifikacija sintaksnog analizatora

Pod pretpostavkom da se u fajlu *tiger.cup* nalazi ispravna specifikacija sintaksnog analizatora, tada se generisanje vrši naredbom:

```
java java_cup.Main < tiger.cup
```

Svaki generisani sintaksni analizator se sastoji od tri klase. Klasa *sym* sadrži niz celobrojnih konstanti, po jednu za svaki terminal. Javna klasa *parser* sadrži implementaciju sintaksnog analizatora, dok skrivena klasa *CUP$action* sadrži sve akcije

koje je definisao korisnik u gramatici kao i one iz *action code* deklaracije.

Kompajliranje ovih klasa se vrši pozivom Java kompajlera:

```
javac sym.java
javac CUP$action.java
javac parser.java
```

Klasa *parser* sadrži generisani sintaksni analizator. Ona je podklasa klase *java_cup.runtime.lr_parser* koja implementira neki generalni sintaksni analizator. Klasa *parser* kao i njena nadklasa sadrže niz metoda koji mogu biti prepisani kako bi se izmenilo ponašanje sintaksnog analizatora. Ovde će biti opisana samo dva metoda koji su ranije spominjani.

```
public void report_error(String message, Object info)
```

Ovaj metod bi trebalo da se pozove svaki put kad se otkrije greška. Inicijalno, prvi parametar sadrži tekst poruke koji se ispisuje na *system.err* a drugi parametar se zanemaruje. Ovaj metod bi trebalo prepisati da bi se obezbedio bolji mehanizam za ispisivanje grešaka.

```
public void syntax_error(Symbol cur_token)
```

Ovaj metod se poziva iz sintaksnog analizatora čim se sintaksna greška otkrije. Inicijalna implementacija poziva `report_error("Syntax error", null);`.

Pri kreiranju klase *parser* bi trebalo naglasiti koji leksički analizator će se koristiti. To se lako postiže pozivom konstruktora za klasu *parser* koji kao argument prima vrednost tipa *Yylex*.

```
parser parser_obj = new parser( new Yylex(...));
```

Izvršavanje sintaksnog analizatora se postiže pozivom metoda `public Symbol parse()` ili `public Symbol debug_parse()`. Pozivom prvog metoda se inicira sintaksna analiza ulaznog fajla, dok poziv drugog metoda pored sintaksne analize omogućava i ispisivanje poruka. O kakvim porukama se radi biće ilustrovano u sledećem poglavlju.

Pozivanje sintaksnog analizatora za programski jezik Tiger se vrši u klasi *Main*:

```
package Parser;

public class Main {
  public static ErrorMsg.ErrorMsg errMsg = null;
  public static void main(String argv[]) throws java.io.IOException  {
      try {
       if (argv.length<1)
        System.out.println("Expected source file as parameter!");
       else {
          errMsg = new ErrorMsg.ErrorMsg(argv[0]);
```

```
            parser parser_obj = new parser( new Yylex(
                            new java.io.FileInputStream(argv[0]) ,
                            errMsg));
            System.out.println("Parsing... \n");
            java_cup.runtime.Symbol parse_tree = parser_obj.parse();
          }
        } catch (Exception e) {
             System.out.println("File not found");
          }
    }

}
```

5. Način rada sintaksnog analizatora i test primeri

U ovoj glavi dato je nekoliko test primera, u cilju dokazivanja da sintaksni analizator prihvata samo sintaksno ispravne programe.

U prva dva primera su prikazana dva Tiger programa u kojima se nalaze sve bitnije konstrukcije programskog jezika Tiger. U trećem primeruje ilustrovan rad metode *debug_parse*.

5.1. Program "Queens"

U fajlu *queens.tig* nalazi se Tiger program koji rešava problem kraljica. Naime, program treba da pronađe sve kombinacije postavljanja 8 kraljica na šahovsku tablu, ali tako da se one međusobno ne napadaju. Ovaj problem se rešava poznatim algoritmom – backtracking. Izvorni kod ovog programa izgleda ovako:

```
let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
       (for i := 0 to N-1
       do (for j := 0 to N-1
            do print(if col[i]=i then " O" else " .");
           print("\n"));
         print("\n"))

    function try(c:int) =
( /*  for i:= 0 to c do print("."); print("\n"); flush();*/
     if c=N
     then printboard()
     else for r := 0 to N-1
         do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
```

```
                then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                    col[c]:=r;
                      try(c+1);
                  row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

)
 in try(0)
end
```

Ovaj program sadrži deklaraciju promenljivih, tipova i funkcija. Od tipova podataka sadrži standardne tipove, kao i tip niza, dok od naredbi sadrži *let*-naredbu, *for*-naredbu i *if*-naredbu. Pored ovih konstrukcija ovaj program ilustruje i rekurziju.

Testiranje programa se izvodi sledećom naredbom:

```
java Parser.Main queens.tig
```

Izlaz koji daje sintaksni analizator izgleda ovako:

```
Parsing...

Parsing finished
0 error(s) detected.
```

5.2. Program “Merge”

Ovaj program učitava dve sortirane liste celih brojeva sa standardnog ulaza (elementi su odvojeni prazninama ili su u novom redu, a svaka lista bi trebalo da se završi tačkazarezom), a zatim ih sortira algoritmom merge-sort.

Izvorni kod programa izgleda ovako:

```
let

 type any = {any : int}
 var buffer := getchar()

function readint(any: any) : int =
 let var i := 0
     function isdigit(s : string) : int =
              ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
     function skipto() =
       while buffer=" " | buffer="\n"
         do buffer := getchar()
  in skipto();
     any.any := isdigit(buffer);
     while isdigit(buffer)
       do ( i := i*10+ord(buffer)-ord("0");
```

```
           buffer := getchar()
         );
     i
 end

 type list = {first: int, rest: list}

 function readlist() : list =
    let var any := any{any=0}
        var i := readint(any)
     in if any.any
         then list{first=i,rest=readlist()}
         else nil
    end

 function merge(a: list, b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first,rest=merge(a.rest,b)}
      else list{first=b.first,rest=merge(a,b.rest)}

 function printint(i: int) =
  let function f(i:int) = if i>0
           then (f(i/10);
             print(chr(i-i/10*10+ord("0"))))
   in if i<0 then (print("-"); f(-i))
      else if i>0 then f(i)
      else print("0")
  end

 function printlist(l: list) =
   if l=nil then print("\n")
   else ( printint(l.first);
          print(" ");
          printlist(l.rest)
        )

   var list1 := readlist()
   var list2 := (buffer:=getchar(); readlist())


  /* BODY OF MAIN PROGRAM */
 in printlist(merge(list1,list2))
end
```

U ovom programu su, pored navedenih konstrukcija, ilustrovane upotrebe: *while*-naredbe, slogovskog tipa podataka kao i implementacija povezane liste. Prikazana je i upotreba standardnih funkcija: *print*, *getchar*, *ord* i *chr*. Treba napomenuti da izraz

```
any := any{any=0}
```

ilustruje da jedno isto ime može označavati promenljivu, tip i polje sloga.

Testiranje programa se izvodi sledećom naredbom:

```
java Parser.Main merge.tig
```

Izlaz koji daje sintaksni analizator izgleda ovako:

```
Parsing...

Parsing finished
0 error(s) detected.
```

5.3. Ilustracija metoda “debug_parse”

Kao što je već napomenuto, izvršavanje sintaksnog analizatora se može postići pozivom metode *debug_parse*. Jedina metoda između ove metode i metode *parse* je što metoda *debug_parse* ispisuje poruke o toku sintaksne analize. Ove poruke sadrže informaciju o: trenutnom ulaznom tokenu, akciji koju izvršava sintaksni analizator (*Shift* ili *Reduce*) i o promeni stanja automata (koji je sastavni deo LALR(1) sintaksnog analizatora). Međutim, pošto su poruke koje ispisuje sintaksni analizator prilično velike, izvršavanje metode *debug_parse* će biti ilustrovano na jednom zaista kratkom primeru (*simple.tig)*:

```
let
  var i:=1
in
  i:=i+1
end
```

Pre analize ovog programa bi trebalo u fajlu *main.java* umesto naredbe

napisati naredbu

```
java_cup.runtime.Symbol parse_tree = parser_obj.debug_parse();
```

Sintaksna analiza programa *simple.tig* se poziva naredbom

```
java Parser.Main merge.tig
```

Izlaz koji se dobija pozivom metode *debug_parse* izgleda ovako:

```
Parsing...

# Initializing parser
# Current Symbol is #43
# Shift under term #43 to state #6
# Current token is #37
# Shift under term #37 to state #81
# Current token is #32
```

```
# Shift under term #32 to state #82
# Current token is #25
# Shift under term #25 to state #84
# Current token is #3
# Shift under term #3 to state #15
# Current token is #23
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 84, lhs sym 2 -> state 85
# Goto state #85
# Reduce with prod #84 [NT=37, SZ=4]
# Reduce rule: top state 6, lhs sym 37 -> state 74
# Goto state #74
# Reduce with prod #73 [NT=31, SZ=1]
# Reduce rule: top state 6, lhs sym 31 -> state 77
# Goto state #77
# Reduce with prod #70 [NT=30, SZ=0]
# Reduce rule: top state 77, lhs sym 30 -> state 114
# Goto state #114
# Reduce with prod #71 [NT=30, SZ=2]
# Reduce rule: top state 6, lhs sym 30 -> state 78
# Goto state #78
# Shift under term #23 to state #109
# Current token is #32
# Shift under term #32 to state #4
# Current token is #25
# Reduce with prod #11 [NT=3, SZ=1]
# Reduce rule: top state 109, lhs sym 3 -> state 30
# Goto state #30
# Shift under term #25 to state #34
# Current token is #32
# Shift under term #32 to state #4
# Current token is #40
# Reduce with prod #11 [NT=3, SZ=1]
# Reduce rule: top state 34, lhs sym 3 -> state 30
# Goto state #30
# Reduce with prod #5 [NT=2, SZ=1]
# Reduce rule: top state 34, lhs sym 2 -> state 35
# Goto state #35
# Shift under term #40 to state #43
# Current token is #3
# Reduce with prod #32 [NT=12, SZ=1]
# Reduce rule: top state 35, lhs sym 12 -> state 50
# Goto state #50
# Shift under term #3 to state #15
# Current token is #24
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 50, lhs sym 2 -> state 51
# Goto state #51
# Reduce with prod #31 [NT=11, SZ=3]
# Reduce rule: top state 34, lhs sym 11 -> state 24
# Goto state #24
# Reduce with prod #27 [NT=10, SZ=1]
# Reduce rule: top state 34, lhs sym 10 -> state 23
# Goto state #23
# Reduce with prod #8 [NT=2, SZ=1]
# Reduce rule: top state 34, lhs sym 2 -> state 35
# Goto state #35
```

```
# Reduce with prod #60 [NT=23, SZ=3]
# Reduce rule: top state 109, lhs sym 23 -> state 9
# Goto state #9
# Reduce with prod #54 [NT=22, SZ=1]
# Reduce rule: top state 109, lhs sym 22 -> state 11
# Goto state #11
# Reduce with prod #10 [NT=2, SZ=1]
# Reduce rule: top state 109, lhs sym 2 -> state 65
# Goto state #65
# Reduce with prod #19 [NT=6, SZ=1]
# Reduce rule: top state 109, lhs sym 6 -> state 64
# Goto state #64
# Reduce with prod #18 [NT=5, SZ=1]
# Reduce rule: top state 109, lhs sym 5 -> state 110
# Goto state #110
# Shift under term #24 to state #113
# Current token is #0
# Reduce with prod #67 [NT=29, SZ=5]
# Reduce rule: top state 0, lhs sym 29 -> state 26
# Goto state #26
# Reduce with prod #59 [NT=22, SZ=1]
# Reduce rule: top state 0, lhs sym 22 -> state 11
# Goto state #11
# Reduce with prod #10 [NT=2, SZ=1]
# Reduce rule: top state 0, lhs sym 2 -> state 8
# Goto state #8
Parsing finished
0 error(s) detected.
# Reduce with prod #0 [NT=1, SZ=1]
# Reduce rule: top state 0, lhs sym 1 -> state 25
# Goto state #25
# Shift under term #0 to state #59
# Current token is #0
# Reduce with prod #1 [NT=0, SZ=2]
# Reduce rule: top state 0, lhs sym 0 -> state -1
# Goto state #-1
```

6. Zaključak

U ovom radu je ilustrovana realizacija leksičkog i sintaksnog analizatora za programski jezik Tiger. Leksički i sintaksni analizator su realizovani korišćenjem generatora *JLex* i *CUP*. Ovde je prikazano kako realizacija leksičkog i sintaksnog analizatora može biti korektna i relativno jednostavna kada se koriste generatori. Međutim, dobrom izborom generatora (kao što su *JLex* i *CUP*) mogu se dobiti čak i efikasni leksički i sintaksni analizatori, koji se mogu koristiti i kod nekih komercijalnih kompajlera.

Prirodni nastavak ovog radaje realizacija semantičke analize kao i kreiranje apstraktnog sintaksnog drveta. Nakon ovoga moguće je realizovati interpreter za programski jezik Tiger ili kompajler koji će generisati Java bytecode ili kod za neku drugu apstraktnu ili stvarnu mašinu. Generisanje Java bytocode-a ima prednosti jer je taj kod nezavisan od operativnog sistema tj. može se izvršavati na različitim platformama.

Još jedan zanimljiv nastavak ovog rada mogao bi biti proširenje programskog jezika Tiger da bi on bio objektno-orjentisan ili funkcionalan ili čak i oba. Ovo proširenje je detaljno opisano u [1].

Literatura

[1] Andrew W. Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1998.

[2] Laura Lemay, Charles L. Perkins, Michael Morrison, Teach Yourself Java in 21 Days, Sams.net Publishing, 1996.

[3] Elliot Berk, JLex: A lexical analyzer generator for Java, Department of Computer Science, Princeton University, http://www.cs.princeton.edu/~appel/modern/java/JLex/main_java.html, July 27, 1999.

[4] Scott E. Hudson, CUP User's Manual, Georgia Institute of Technology, http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html, July, 1999.

[5] Zoran Budimac, Mirjana Ivanović, Đura Paunić, Programski jezik Modula-2, Univerzitet u Novom Sadu, Prirodno-matematički fakultet, Institut za matematiku, Novi Sad, 1996.

Biografija

Vladimir Kurbalija je rođen 12. maja 1977. u Novom Sadu, od majke Mirjane. Osnovnu školu i gimnaziju je završio u Novom Sadu. Tokom školovanja je više puta učestvovao na takmičenjima iz matematike i fizike. 1996. upisuje Prirodno-matematički fakultet u Novom Sadu. Tokom studiranja je više puta nagrađivan fakultetskim i univerzitetskim nagradama. Sve predviđene ispite je položio sa prosečnom ocenom 9.88.

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Ključna dokumentacijska informacija

Redni broj: RBR	
Identifikacioni broj: IBR	
Tip dokumentacije: TD	Monografska dokumentacija
Tip zapisa: TZ	Tekstualni štampani materijal
Vrsta rada: VR	Diplomski rad
Autor: AU	Vladimir Kurbalija
Mentor: MN	dr Mirjana Ivanović
Naslov rada: NR	Realizacija leksičkog i sintaksnog analizatora za programski jezik Tiger
Jezik publikacije: JP	srpski (latinica)
Jezik izvoda: JI	s/en
Zemlja publikovanja: ZP	SR Jugoslavija
Uže geografsko područje: UGP	Vojvodina
Godina: GO	2000
Izdavač: IZ	autorski reprint

Mesto i adresa: MA	Novi Sad, Trg D. Obradovića 4
Fizički opis rada: (broj poglavlja/strana/lit.citata/tabela/slika/grafika/priloga) FO	6/54/5/3/5/0/0
Naučna oblast: NO	Računarske nauke
Naučna disciplina: ND	Konstrukcija kompajlera
Predmetna odrednica/Ključne reči: PO	Pogramski jezici, Implementacija, Kompajleri
UDK	
Čuva se: ČU	
Važna napomena: VN	
Izvod: IZ	U ovom radu je ilustrovana realizacija leksičkog i sintaksnog analizatora za programski jezik Tiger. Leksički i sintaksni analizator su realizovani korišćenjem generatora *JLex* i *CUP*. Ovde je prikazano kako realizacija leksičkog i sintaksnog analizatora može biti korektna i relativno jednostavna kada se koriste generatori. Međutim, dobrom izborom generatora (kao što su *JLex* i *CUP*) mogu se dobiti čak i efikasni leksički i sintaksni analizatori, koji se mogu koristiti i kod nekih komercijalnih kompajlera.
Datum prihvatanja teme od strane NN veća: DP	oktobar 2000
Datum odbrane: DO	oktobar 2000
Članovi komisije: (Naučni stepen/ime i prezime/zvanje/fakultet) KO	
Predsednik:	dr Ratko Tošić, redovni profesor Prirodno-matematičkog fakulteta u Novom Sadu
Član:	dr Zoran Budimac, vanredni profesor Prirodno-matematičkog fakulteta u Novom Sadu
Član:	dr Miloš Racković, docent Prirodno-matematičkog fakulteta u Novom Sadu

University of Novi Sad
Faculty of natural sciences & mathematics
Key words documentation

Accession number: NO	
Identification number: INO	
Document type: DT	Monograph documentation
Type of record: TR	Textual printed material
Contents code: CC	Graduation thesis
Author: AU	Vladimir Kurbalija
Mentor: MN	dr Mirjana Ivanović
Title: TI	Implementation of lexical and syntax analyzers for programming language Tiger
Language of text: LT	Serbian (Latin)
Language of abstract: LA	en/s
Country of publication: CP	FR Yugoslavia
Locality of publication: LP	Vojvodina
Publication year: PY	2000
Publisher: PU	Author′s reprint

Publ. place: PP	Novi Sad, Trg D. Obradovića 4
Physical description: PD	6/54/5/3/5/0/0
Scientific field: SF	Computer science
Scientific discipline: SD	Compiler Construction
Subject/Key words: SKW	Programming languages, Implementation, Compilers
UC	
Holding data: HD	
Note: N	
Abstract: AB	Lexical and syntax analyzers for programming language Tiger are presented in this paper. They are implemented by using *JLex* & *CUP* generators. In this paper it is presented that the implementation of lexical and syntax analyzers may be correct and quite simple when we use generators. However, with a good choise of generators (like *JLex* & *CUP*) we can realize even high efficient analyzers, which can be used in some comercial compilers.
Accepted on Scientific board on: AS	October, 2000
Defended: DE	October, 2000
Thesis Defend board: DB	
President:	dr Ratko Tošić, full professor, Faculty of Natural Science and Mathematics, Novi Sad
Member:	dr Zoran Budimac, associate professor, Faculty of Natural Science and Mathematics, Novi Sad
Member:	dr Miloš Racković, assistant professor, Faculty of Natural Science and Mathematics, Novi Sad