

## Yacc - generator parsera

VJEŽBE 11

ću je

skrenuti

- \* Korisnik priprema specifikaciju procesa pravila koja opisuju strukturu inputa, pa kada se prepoznaju ta pravila i rutinu niskog nivoa koja se bavi osnovnim inputom (leksičkom analizom inputa). Na osnovu te specifikacije YACC generiše funkciju za kontrolu tog procesa. To je funkcija koju zovemo **parser**. Ona obavjava sintaksnu analizu inputa, tj. u osnovi odgovara na pitanje da li ulazna niska (dio ulaznog toka do graničnika) pripada jeziku koji opisuju data gramatička pravila. Ta funkcija poziva ulaznu rutinu niskog nivoa koju je obezbedio korisnik (leksički analizator) da pokupi leksičke klase (tokene, lekseme) iz ulaznog toka. Te lekseme zatim organizuje prema pravilima koja struktura inputa mora da zadovoljava - gramatičkim pravilima; kada se prepozna neko od tih pravila (tj. desna strana nekog od tih pravila), izvršava se kod kojih je korisnik obezbedio **ta pravilo (akciju)**. Akcije mogu da vraćaju vrijednosti i da koriste vrijednosti drugih akcija. Ako parser uspije da strukturu ulazne niske usporedi sa najštom klasom - tj. početnim simbolom, tj. ako otkrije da postoji izvodenje te niske u toj gramatici, vratiće pozitivan odgovor, odnosno, obavjestiće korisnika da ta niska pripada datom jeziku.

### Osnovna specifikacija

- \* Svaka specifikaciona datoteka sastoji se od 3 sekcije: deklaracije, (gramatička) pravila i programi. Sekcije se odvajaju sa %%. Puna specifikacija ima oblike:

deklaracije

%%

pravila

%%

programi

a najmanja specifikacija ima oblike:

%%

pravila

jer ako nema sekcije programa ne mora se uključivati, ni drugi red sa procentima.

\* Sekcija deklaracije služi za deklaraciju imena (i globalnih C-ovskih promjenjivih). Imena se odnose na terminale (lekseme) ili na neterminale (uglavnom ono s lijeve strane gramatičkih pravila). Imena mogu biti razlicitih dužina i sastoje se od slova, cifara, tačaka i podukate, pri čemu moraju počinjati slovom. Velika i mala slova se razlikuju. Imena koja predstavljaju lekseme (terminale) moraju se deklarisati, na sljedeći način: u sekcijskoj deklaraciji upiše se

% token Ime1 Ime2 Ime3...

Suako imenue koje nije deklarisano u sekcijskoj deklaraciji smatraće se imenom neterminala. Svaki neterminal mora se pojaviti na lijevoj strani bar jednog pravila.

- Od svih neterminala, jedan, nazvan startni simbol, ima posebnu važnost. Parser se pravi da bi "prepoznao" startni simbol, te prema tome, startni simbol, predstavlja najveću i najopštiju strukturu koja je opisana gramatičkim pravilima. Po defaultu se uzima da je startni simbol lijeva strana prvega pravila navedenog u sekcijskim pravila, ali je poželjno eksplicitno deklarisati startni simbol u sekcijskoj deklaraciji, uz korištenje ključne riječi % start
- Korisnik može definisati i druge promjenjive koje će koristiti akcije. Deklaracije i definicije pojavljuju se u sekcijskoj deklaraciji, ograničene sa %{ i %}. Te promjenjive imaju globalan opseg važenja pa su poznate izrazima akcija i leksičkom analizatoru. Npr.

%{

int prom = 0;

%}

može se postaviti u sekcijsku deklaraciju i prom će biti dostupno svim akcijama.

- YACC parser koristi samo imena koja počinju sa "yy", tako da bi korisnik takva imena trebao izbjegavati.

\* Sekcija pravila  
sastoji se od gramatičkih pravila oblika

A : TIJELO;

Pri čemu je A ime neterminala, dvotacka predstavlja „strojicu“, a TIJELO je sekvenca 0 ili više imena i literala.

- Imena koja se koriste u tijelu gramatičkog pravila mogu predstavljati lekseme ili neterminale.
- Literali se sastoje od karaktera ograničenih jednostrukim navodnicima "", npr. '+'. Pri tome, kao i u C-u možemo koristiti i eskejp karakter (kosu crtu).
- Ako nekoliko pravila ima istu lijevu stranu, umjesto ponovnog ispisa te lijeve strane može se koristiti vertikalna crta.  
Tada se mogu razstaviti tacka-zarez.

Dakle, umjesto:

A : F G ;

A : X Y ;

moga i

A : F G

| X Y ;

- Ako neterminalu odgovara epsilon pravilo, to se može prikazati na sledeći način:

P : ;

### \* Akcije

- Sa svakim gramatičkim pravilom, korisnik može da poveže akcije koje će se izvršavati svaki put kada se u procesu očitavanja inputa prepozna to pravilo. Te akcije mogu vratiti vrijednosti i mogu dobijati vrijednosti od prethodnih pravila. Štaviše, i leksički analizator može vratiti vrijednosti leksema.

- Akcija je malakav izraz na C-u : input, output, poziv funkcija, promjena vrijednosti globalnih promjenjivih.

Akcija je određena sa jednim ili više izraza, ograničenih velikim zagradama { i }, npr.

A: (' B ')

{ printf ("Evo ode B"); }

Ili

X: y z

{ flag=26; }

- Kako bi se omogućila komunikacija između akcije i parsera, izrazi za akcije koriste mehanizam tзв. dolarskih (\$) promjenjivih.

Akcije za vraćanje vrijednosti koriste pseudo-promjenjive \$\$ koje postavljaju na neku vrijednost. Npr. akcija koja ne radi ništa drugo sem što vraća vrijednost 1 je

{ \$\$ = 1; }

- Za dobijanje vrijednosti koje vraćaju prethodne akcije i leksički analizator, akcija može koristiti promjenjive pseudo \$1, \$2, ... koje se odnose na vrijednosti koje redom vraćaju komponente desne strane pravila, s lijeva na desno. Tako, ako je pravilo

A: B C D;

ona \$2 ima vrijednost C.

Po defaultu, vrijednost pravila je vrijednost prve komponente u njemu (\$1). Tako, gramatička pravila oblika

A: B

ne moraju pratiti eksplisitne akcije.

- U dosadašnjim primjerima sve akcije su se nalazile na kraju pravila. Ponekad je poželjno preuzeti kontrolu prije nego što se pravilo u potpunosti parsira. YACC dozvoljava da se akcija upiše usred pravila. Pretpostavlja se da i ta akcija, kao komponenta pravila, vraća vrijednost, koja je dostupna pravilima koja su joj ždesna putem uobičajnog mehanizma, a ona može imati pristup vrijednostima koje vraćaju komponente koje su joj slijeva. Tako, pravilo:

A : B { \$\$=1; } C { x=\$2; y=\$3; }

ima sljedeći učinak: postavba x na 1 i y na vrijednost koju vraća c.

- Akcije koje ne završavaju pravila YACC u stvari tretira na sljedeći način: proizvede novo ime neterminala i novo pravilo koje povezuje to ime s komponentom desne strane. Unutrašnja akcija je akcija pokrenuta prepoznavanjem pridodataog pravila. YACC u stvari gornji primjer tretira na sljedeći način:

\$ ACT: /\* prazno \*/  
{ \$\$=1; }  
;

A : B \$ ACT C  
{ x=\$2; y=\$3 }  
;

## \* Leksički analizator

- Korisnik mora da obezbjedi leksički analizator tako što će očitavati ulazni tok i proslediti lekseme (tokene), zajedno s njihovim vrijednostima, ako je to potrebno, do parsera. Leksički analizator je cijelobrojna funkcija koja se naziva **yylex()** - to je naziv koji će parser očekivati, ako parсер dobijamo pomoću alata kao što je YACC, pa ćemo ga konstruirti u svakom slučaju. Funkcija **yylex()** vraća cijelobrojnu vrijednost koja predstavlja oznaku lekseme, odnosno vrstu tokena koji je očitan iz ulaznog toka. Ako je s tom leksemom povezana neka vrijednost, nju u okviru leksičkog analizatora treba povezati sa spojenošnjom promjenljivom **yyval()**, koja služi za prenos vrijednosti leksema do parsera.

- Parser i leksički analizator moraju se poklapati što se tiče brojeva leksema, odnosno oznaka za vrste tokena, kako bi mogla da postoji obastrano jednoznačna komunikacija između njih. Te brojeve može izabrati YACC - ili sam korisnik.

U svakom slučaju, da bi leksički analizator te brojeve vratio kao rezultat koristi se "#define" mehanizam C-a. Npr. pretpostavimo da je ime lekseme CIFRA definisano u sekciji deklaracija specifikacije (opisa) za YACC. Odgovarajuća sekcija leksičkog analizatora može izgledati ovako:

```
yylex()
{
    extern int yyval;
    int c;

    ...
    c = getchar();

    switch(c)
    {
        ...
        case '0':
        case '1':
        ...
        case '9': yyval = c - '0';
                    return (CIFRA);
    }
    ...
}
```

- Namjera je da se vrati broj lekseme CIFRA - oznaku da je iz ulaznog tока prepoznata niska koja odgovara pravilima te leksičke klase, kao i vrijednost koja je jednaka numeričkoj vrijednosti te cifre. Ako se kod leksičkog analizatora ubaci u sekciju programa specifikacije (opisa) za YACC, identifikator CIFRA će biti definisan kao broj lekseme vezane za CIFRE.
- Ovaj mehanizam daje jasne leksičke analizatore koje je lako modifikovati; jedini nedostatak predstavlja potreba da se izbjegne korištenje suhi imena leksema koja su rezervisana u C-u ili u parseru; npr. korištenje imena

Leksema kao što su "if" ili "while" će skoro sigurno dovesti do teških posledica kada se leksički analizator kompajlira. Ime "error" je rezervisano za obradu gresaka i takođe se ne smije koristiti nativno.

- Kao što je već pomenuto, brojevi leksema mogu biti izabrani od strane YACC-a, ili ih može izabrati korisnik. Default brojevi leksema za karaktere iz ASCII skupa karaktera predstavljaju njihova numenička vrijednost (redni broj) u ASCII skupu karaktera. Kako u ASCII skupu karaktera ima 256 karaktera (koji uključuju slova, cifre, interpunkcijske i specijalne znakove), ostalim imenima će se dodjeljivati brojevi od 257 pa nadalje.
- Da bi se leksemama dodjelivali brojevi, prva pojava imena lekseme u sekciji deklaracija mora biti neposredno prečena nenegativnim cijelim brojem. Uzima se da je taj cijeli broj broj lekseme imena ili literala. Imena i literali koji nisu definisani ovim mehanizmom zadržavaju svoje default brojeve. Važno je da brojevi leksema budu različiti.
- Iz istorijskih razlika, graničnik mora imati broj lekseme o ili negativan broj. Taj broj lekseme korisnik ne može redefinisati, prema tome, svi leksički analizatori trebaju biti spremni da vrata o ili negativan broj kao broj lekseme kada dođu do kraja ulaznog toka - time signaliziraju parseru da je input učitan i da treba biti upoređen sa najširim gramatičkom kategorijom - početnim simbolom.
- Veoma koristan alat za konstruisanje leksičkih analizatora je FLEX. Leksički analizatori, koji se dobijaju pomoću FLEX-a stvorenji su da rade u harmoniji sa parserima koje pravi YACC. Specifikacije za leksičke analizatore koriste regularne izraze umjesto gramatičkih pravila, ali su inače, kao što smo već vidjeli, prilično slične specifikacijama za parsere.

## Zadatak 1:

Napisati sintakski analizator za paskoliki jezik opisan gramatikom:

$\langle \text{naredba} \rangle ::= \langle \text{while\_naredba} \rangle$   
|  $\langle \text{begin\_naredba} \rangle$   
|  $\langle \text{naredba\_dodela} \rangle$

$\langle \text{while\_naredba} \rangle ::= \text{WHILE } \langle \text{liskaz} \rangle \text{ DO } \langle \text{naredba} \rangle$

$\langle \text{begin\_naredba} \rangle ::= \text{BEG } \langle \text{naredba} \rangle \text{ END SEMI}$

$\langle \text{naredba\_dodela} \rangle ::= \text{PROM } \text{DODELA } \langle \text{l2raz} \rangle \text{ SEMI}$

$\langle \text{naredba1} \rangle ::= \langle \text{naredba} \rangle | \langle \text{naredba} \rangle \langle \text{naredba} \rangle$

$\langle \text{liskaz} \rangle ::= \langle \text{l2raz} \rangle \text{ EQ } \langle \text{l2raz} \rangle$

$\langle \text{l2raz} \rangle ::= \text{PROM} | \text{BROJ}$

### Rešenje:

\* Primjetimo da leksički analizator treba da nas obavještava o pojavama sledećih leksema: „rezervisanih riječi“ - WHILE, DO, BEG, END, „znakova“ - SEMI (tacka-žarez), DODELA (duotacka-jednako) i EQ (jednako), te PROM (identifikatorima) i BROJ (brojevima). Kako želimo samo da odgovorimo na pitanje da li neka niska pripada jeziku definisanom tom gramatikom a ne i da evaluiramo te niske, nećemo tražiti od leksičkog analizatora da vraca vrijednosti prepoznatih leksema, već samo da obavještava parser o tome da ih je prepoznao.

\* Opis za FLEX koji ćemo koristiti za dobijanje traženog leksičkog analizatora:

go option noyywrap

slово [a-zA-Z]

cifra [0-9]

broj {cifra}+

prom {slovo}{ {cifra} | {slovo}}\*

```
0/0/0
while { return WHILE; }
do { return DO; }
begin { return BEG; }
end { return END; }
:= { return DODELA; }
= { return EQ; }
; { return SEMI; }
{ broj } { return BROJ; }
{ prom } { return PROM; }
```

\* Obratite pažnju da je treća sekcija, sekcija programa, u ovom opisu prazna (tj. nema je), za razliku od opisa za FLEX koji smo pisali u Poglavlju 1. To je zato što će funkcije `main()` i `yylex()`, koje smo šablonski ukločivali u tu sekciju, biti pozvane iz parsera, tako da nema potrebe da ih tu navodimo.

\* Kao i programi za rekurzivni spust koje smo koristili u konjukciji sa leksičkim analizatorima proizvedenim FLEX-om i parсер koji ćemo proizvesti YACC-om očekivamo da se leksički analizator nađe u datoteci `lex.yy.c`, pa zbog toga prvo moramo da „fleksujemo“ gore navedeni opis za FLEX (koji se nalazi u datoteci `pr1y.l`). Dakle, izvršimo sledeću naredbu:

```
flex pr1y.l
```

\* Tako smo dobili leksički analizator. Naravno, on sada ne može da se koristi sam za sebe (nema `main()`), već treba napraviti i parсер koji će ga „pozivati“.

Opis za parser za ovu gramatiku:

```
0/0 {
```

```
#include <stdio.h>
```

```
0/0 }
```

```
g0token WHILE DO BEG END SEMI EQ DODELA PROM BROJ
```

% % obdati posuje  
s : naredba { printf ("Uneta niska je sintaksno ispravna!");  
naredba : while\_naredba  
| begin\_naredba  
| naredba\_dodele  
;  
;

while\_naredba : WHILE ISKAZ DO naredba  
;

begin\_naredba : BEG naredba! END SEMI  
;

naredba\_dodele : PROM DODELA IZRAZ SEMI  
;

naredba! : naredba  
| naredba naredba!  
;  
;

ISKAZ : IZRAZ EQ IZRAZ  
;  
;

IZRAZ : BROJ  
| PROM  
;  
;

% %  
#include "putanja\lex.yy.c"

yyerror (char \*s)  
{ printf ("%s\n", s);

}

main()

{ return yyparse(); }

- \* Kako nismo koristili mehanizam "#define", leksemama će biti dodjeljeni default brojevi, tj. WHILE će dobiti broj 257, DO broj 258 ... itd. i funkcija yylex() će baš te brojeve proslediti parseru kad u ulaznom toku nađe na odgovarajuće lekseme.
- \* Obratite pažnju na sekciju programa ove specifikacije - ona uključuje tri stvari koje se, šablonski, nalaze u svakoj takvoj specifikaciji - direktivu za uključenje datoteke koja sadrži leksički analizator, funkciju za obradu grešaka i funkciju main(), koja u biti samo poziva funkciju yyparse(), koju ćemo dobiti kada YACC obradi tu specifikaciju.
- \* Specifikacija se nalazi u datoteci pr1y.y. Fakujemo je sa bison pr1.y.y.  
Trebali bi dobiti kratki izvještaj koji će nas obavjestiti o broju konfliktata.  
Otvorimo C-kod, kompajliramo ga i pokrenemo.  
u cmd : pr1y-tab.exe

D

Prihvata:

1.  $x := 23;$
2. while  $23 = 23$  do  $y := 14;$
3. begin  $y := 14;$  end;

## Dvostrukost i konflikti

\* Skup gramatičkih pravila je dvostruk ako postoji ulazna niska koja se može strukturirati na dva ili više različitih načina.

Npr. pravilo

Izraz : Izraz<sup>1</sup> - Izraz<sup>2</sup>

Predstavlja prirodan način izražavanja činjenice da je jedan od načina na koji se mogu formirati aritmetički izrazi spajanje druga dva izraza sa minusom između njih. Naučnost, to gramatičko pravilo ne određuje u potpunosti način na koji treba strukturirati sve kompleksne ulaze niske. Npr. za ulaz

Izraz - Izraz - Izraz

pravilo dozvoljava struktuiranje kao

(Izraz<sup>1</sup> - Izraz<sup>2</sup>) - Izraz<sup>3</sup>

ali i kao

Izraz - (Izraz<sup>1</sup> - Izraz<sup>2</sup>)

prvo predstavlja lijevu asocijativnost, a drugo desn.

\* Kada pokušava da izgradi parser, YACC prepoznaje takve dvostrukosti. Kada mu je dat ulaz oblika

Izraz - Izraz<sup>1</sup> - Izraz<sup>2</sup>

i kad je parser očitao drugi izraz s ulaza, ulaz koji je do tada vidi je

Izraz<sup>1</sup> - Izraz<sup>2</sup>

Koji odgovara desnoj strani gornjeg gramatičkog pravila. Parser bi mogao da redukuje input upotrebljavajući to pravilo - nakon toga, ulaz bi bio redukovani na Izraz (lijevu stranu gramatičkog pravila). Zatim bi parser očitao i preostali dio inputa

- Izraz

i opet redukovao. Učinak tih koraka doveo bi do rezultata koji odgovara lijevoj asocijativnosti.

Međutim, umjesto toga, kad parser vidi

Izraz<sup>1</sup> - Izraz<sup>2</sup>

mogaće da odloži neposrednu primjenu pravila i nastavi da očitava ulaz dok ne vidi

Izraz<sup>1</sup> - Izraz<sup>2</sup> - Izraz

Tu može da primjeni pravilo na desna tri simbola, redukujući ih na izraz, što ostavlja

### izraz - izraz

Sada se može još jednom redukovati po tom pravilu, a rezultat je interpretacija koja podrazumjeva desnu asocijativnost.

Tako, kad očita

### izraz - izraz

parser može da uradi dvije stvari, koje su obje dozvoljene - shift ili reduce akciju, i ne postoji način na koji on može donijeti tu odluku. To se naziva shift/reduce konflikt. Takođe, može se desiti da parser treba da doneše odluku između 2 dozvoljene redukcije - to se naziva reduce/reduce konflikt (ono što je parser očitao predstavlja desne strane više od jednog pravila). Primjetite da se nikad ne pojavljuje shift/shift konflikti (parser u datom trenutku može da šiftuje samo jedan simbol - nailazeći, i to može uraditi na samo jedan način).

\* Kada dođe do shift/reduce ili reduce/reduce konflikata, YACC ipak proizvede parser. To radi tako što kad god mora da doneše odluku izabere jedan od 2 dozvoljena koraka, primjenjujući dva pravila koja opisuju izvore koje se donose u takvoj situaciji, i koja se nazivaju pravilima za razrješavanje duosmisenosti:

1. U situaciji kada imamo shift/reduce konflikt, po defaultu se obavљa shift akcija.

2. U situaciji kada imamo reduce/reduce konflikt, default je da se redukuje po rangu gramatičkom pravila.

Pravilo 1 poulači da se redukcije odlazu kad god da to postoji mogućnost tako da se obavljaju sve moguće shift akcije.

Pravilo 2 daje korisniku dosta grub način kontrolisanja ponašanja parsera u toj situaciji, ali, ipak, reduce/reduce konflikte treba izbjegavati kad god je to moguće.

- \* Konflikti mogu biti posledice gresaka u ulazu ili u logičkoj postavci, ili toga da gramatička pravila iako su konzistentna, zahtjevaju kompleksniji parser od onoga koji može proizvesti YACC. Koristenje akcija unutar pravila takođe može izazvati konflikte ako se akcija mora izvršiti prije nego što je parser siguran koje pravilo prepoznaće. U tim slučajevima upotreba pravila koja razrešavaju duosmislenosti nije odgovarajuće rješenje, jer dovodi do netačnog parsera. Iz tog razloga YACC izužetava o broju shift/reduce i reduce/reduce konflikata koji su razrešeni upotrebom pravila 1, odnosno pravila 2.
- \* Uopšte, kad god je moguće primjeniti pravila koja razrešavaju duosmislenosti radi konstrukcije tačnog parsera, takođe je moguće i ponovo napisati gramatička pravila tako da se očita isti ulaz, ali bez konflikata. Iz tog razloga, većina ranjih generatora parsera smatrali su da konflikti predstavljaju greske. Iskustvo kreatora YACC-a pokazalo je da je to ponovno pisanje pravila radi eliminacije konflikata donekle neprirodno i da dovodi do sporijih parsera. Prema tome, neprirodno i da postoji konflikti. YACC će proizvesti parser čak i ako postoji konflikti.
- \* Kas primjer moći pravila koja razrešavaju duosmislenosti, razmotrićemo fragment iz programskog jezika koji uključuje "if then else" konstrukciju.

Izraz : IF '(' uslov ')' Izraz  
          | IF '(' uslov ')' Izraz ELSE Izraz  
          ;

U ovim pravilima IF i ELSE su lekseme (terminali, tokeni), "uslov" je nezavrsni simbol (neterminal) koji opisuje logičke izraze, a "Izraz" je neterminal koji opisuje izraze. Prvo pravilo ćemo zvati if-pravilo, a drugo if-else pravilo. Ta dva pravila formiraju duosmislenu konstrukciju, jer ulaz oblika:

IF (C1) IF (C2) S1 ELSE S2

može da se prema tim pravilima struktura na dva načina:

IF (C1) {

    IF (C2) S1

}

    ELSE S2

i

IF (C1) {

    IF (C2) S1

    ELSE S2

}

Druga interpretacija je ona koju tražimo kod većine programskih jezika koji ukučuju ova konstrukcije. Svaki ELSE je vezan s poslednjim IF koji mu prethodi, a koji još nije vezan sa nekim ELSE-om. U ovom primjeru, razmotrite situaciju do koje dolazi kada je parser na ulazu vidi

IF (C1) IF (C2) S1

i razmatra ELSE. Može odmah redukovati po jednostavnom if pravilu i dobiti

IF (C1) izraz

a onda učitati preostali ulaz

ELSE S2

i redukovati

IF (C1) izraz ELSE S2

po if-else pravilu. To dovodi do prve prikazane interpretacije tog ulaza.

S druge strane, ELSE se može šiftovati, nakon čega se učita S2 i desni dio

IF (C1) IF (C2) S1 ELSE S2

može se redukovati po if-else pravilu, što daje

## IF (c) izraz

a to se onda može redukovati po jednostavnom if pravilu. To dovodi do druge interpretacije ulaza koja je navedena gore, što je ono što se obično i želi postići.

- Opet parser može uraditi dvije dozvoljene stvari i postoji shift/reduce konflikt. Upotreba pravila koje razrješava duosmisljenost br.1. kaže parseru da izvrši shift akciju što dovodi do želenog rezultata.

## Prioritet

- \* Postoji jedna uobičajna situacija u kojoj pravila za razrješavanje duosmisljenosti nisu dovoljna, a to je parsiranje aritmetičkih izraza. Najveći dio obično korištenih konstrukcija za aritmetičke izraze može se prirodno opisati pojmom nivoa prioriteta za operatore, uz informacije o tome da li su lijevo ili desno asocijativni. Pokazuje se da se duosmislene gramatike s odgovarajućim pravilima za razrješavanje duosmisljenosti mogu da se koriste za stvaranje parsera koji su brži i lakši od parsera konstruisanih iz gramatika koje ne sadrže duosmisljenosti.
- \* Osnovna ideja uključuje pisanje gramatičkih pravila u obliku

izraz : izraz OP izraz

;

izraz : UNARNI izraz

za sve binarne i unarne operatore koji se traže.

To pravi vrlo duosmislenu gramatiku s mnogo konfliktova.

Radi uklanjanja duosmisljenosti, korisnik određuje prioritet, tj. jačinu okupljanja, za sve operatore, kao i asocijativnost binarnih operatora. Te informacije su dovoljne da bi yacc razrešio sve konflikte prema tim pravilima i da bi na kraju realizovao traženi prioritet i asocijativnost.

\* Prioritet i asocijativnost dodjebljuju se leksemama u sekciji deklaracije. To se radi unošenjem nekoliko linija koje počinju ključnim riječima  $\%o\ left$ ,  $\%o\ right$  ili  $\%o\nonassoc$  nakon kojih slijede liste leksema.

Sve lekseme u istom redu imaju isti nivo prioriteta i asocijativnost koja je naglašena ključnom riječju kojom taj red počinje, a redovi se nižu prema povećanju prioriteta. Prema tome

$\%o\ Left\ '+'\ '-'$

$\%o\ Left\ '*'\ '/'$

opisuje prioritet i asocijativnost četiri aritmetička operatorka,  $+$  i  $-$  su lijevo asocijativni i imaju manji prioritet od  $*$  i  $/$  koji su takođe lijevo asocijativni. Ključna riječ  $\%o\ right$  koristi za opis desno asocijativnih operatorka, a ključna riječ  $\%o\nonassoc$  za opis operatorka koji nisu asocijativni. Npr.

$\%o\ right\ '='$

$\%o\ left\ '+'\ '-'$

$\%o\ left\ '*'\ '/'$

$\%o\ %o$

Izraz : Izraz  $'='$  Izraz

| Izraz  $'+'$  Izraz

| Izraz  $'-'$  Izraz

| Izraz  $'*'$  Izraz

| Izraz  $'/'$  Izraz

| IME

| ;

dati opis možemo koristiti za strukturiranje inputa

$a = b = c * d - e - f * g$

na sledeći način

$a = (b = ((c * d) - e) - (f * g)))$

Kada se koristi taj mehanizam, unarni operatori moraju, u opštem slučaju, imati viši prioritet. Ponekad binarni i unarni operatori imaju istu simboličku reprezentaciju, ali različite nivoje prioriteta. Primjer za to imamo kod unarnog i binarnog - (minusa). Unarnom minusu se može dati ista snaga kao množenju, ili čak i viša, dok binarno minus ima nižu snagu od množenja. Klučna riječ % prec mijenja prioritetski nivo većan s određenim gramatičkim pravilom. % prec se pojavljuje odmahiza tijela gramatičkog pravila, prije akcije ili tačke-zarez, i prati ga ime lekseme ili literala. Upr. da bi unarni minus imao isti prioritet kao i množenje, pravila mogu izgledati ovako:

% Left '+' '-'

% Left '\*' '/'

90%

Izraz: Izraz '+' Izraz

| Izraz '-' Izraz

| Izraz '\*' Izraz

| Izraz '/' Izraz

| '-' Izraz % prec '\*'

| IME

;

Lekseme se mogu deklarisati sa % left, %right ili %nonassoc mogu, ali i ne moraju biti deklarisane i sa % token.

