Yet Another Compiler-Compiler (YACC)

Filip Marić
filip@matf.bg.ac.yu

Matematički fakultet
februar 2004.

Glava 1

YACC - opis sistema

1.1 Uvod

Yet another compiler-compiler (YACC) je alat razvijen još davnih sedamdesetih godina prošlog veka koji omogućava jednostavnu izgradnju rutina za prihvatanje i analizu tekstualnog ulaza programa. Naime, većina programa prihvata od svojih korisnika određeni tekstualni ulaz. Obično se vrlo precizno može definisati skup dopuštenih ulaza i time se implicitno definiše ulazni jezik programa (input language). Ulazni jezici programa variraju od prilično jednostavnih, poput niza brojeva, pa sve do prilično složenih kao što su, recimo, sami programski jezici. U daljem tekstu će proces prihvatanja i analiziranja tekstualnog ulaza programa biti nazivan *parsiranje*, dok će se skup rutina koje učestvuju u parsiranju jednim imenom nazivati *parser*[1]. Ručno konstruisanje parsera u slučajevima iole kompleksnijih ulaznih jezika predstavlja prilično mukotrpan posao, podložan greškama. Iz tog razloga, vremenom je razvijen niz alata, zajedničkim imenom zvanih kompilatorima-kompilatora (*compiler-compilers*), koji ovaj posao olakšavaju i delimično automatizuju. Iako deo imena *yet another* ukazuje da je u pitanju samo "još jedan" u nizu kompilatora-kompilatora, YACC se definitivno izdvojio i na ovom polju postao standard.

1.2 Korišćenje sistema YACC

Prilikom korišćenja sistema YACC, zadatak programera je da kontekst slobodnom gramatikom opiše ulazni jezik programa koji konstruiše, zatim da navede semantičke akcije koje predstavljaju programski kod koji se izvršava kada se prepozna određeni deo ulaza, i da uz to implementira rutinu nižeg nivoa koja predstavlja fazu leksičke analize čiji je zadatak da iz ulazne struje izdvaja tokene koji predstavljaju terminalne simbole u pomenutoj gramatici.

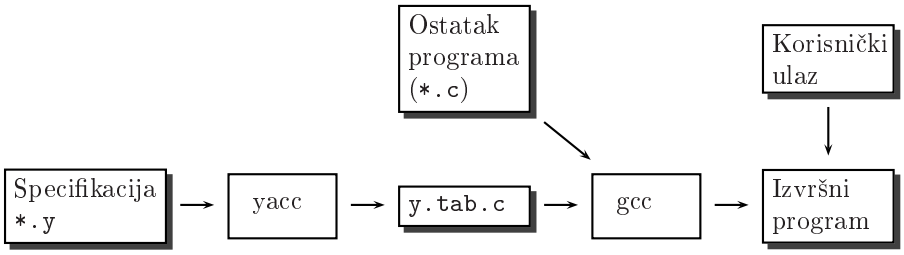
Iz nekoliko razloga možemo reći da je YACC alat koji je tesno povezan sa programskim jezikom C. Naime, sav programski kod koji pišemo unutar specifikacije se piše na ovom programskom jeziku. Rezultat rada YACC-a je datoteka koja se obično naziva

[1] Malo preciznije, termin parser obično označava samo rutine koje vrše sintaksnu analizu, dok se rutine koje vrše leksičku analizu nazivaju skenerima

`y.tab.c`

koja sadrži rutine koja vrši prihvatanje korisničkog ulaza tj. parsiranje. Rutine koje se nalaze u ovoj datoteci je potrebno prevesti nekim C prevodiocem i povezati sa ostatkom programa koji pišemo i tako dobiti izvršnu verziju programa. Dijalekt jezika C koji može da se koristi prilikom implementiranja svih ovih rutina zavisi samo od C prevodioca kojim ćemo prevoditi generisane datoteke.

Dakle, možemo reći da arhitektura YACC sistema odgovara slici 1.1.



Slika 1.1: Korišćenje YACC sistema

Da bismo dobili izvršnu verziju programa, potrebno je još implementirati funkcije

```
int yylex()
```

koja predstavlja leksički analizator i

```
void yyerror(char*,...)
```

koja služi za obradu nastalih grešaka.

Datoteka `y.tab.c` sadrži rutinu

```
int yyparse()
```

koja vrši parsiranje i koju je potrebno pozvati na mestu gde želimo da počne prihvatanje korisničkog ulaza. Ukoliko se parsiranje uspešno završi `yyparse` vraća vrednost 1, a ukoliko je prilikom parsiranja došlo do greške povratna vrednost je 0.

1.3 Osnovna struktura specifikacije

Ulazna datoteka YACC sistema (obično sa ekstenzijom .y) koja sadrži specifikaciju parsera se sastoji od tri velika dela medjusobno razdvojena simbolima

```
%%
```

kao što je prikazano na slici 1.2.

Deo `deklaracije` kao i deo `funkcije` može da ostane prazan. Beline u ulaznoj datoteci se zanemaruju prilikom njene obrade, dok se dodatni komentari mogu navesti između simbola `/*` i `*/`, slično kao u jeziku C.

```
deklaracije
%%
pravila
%%
funkcije
```

Slika 1.2: Osnovna struktura ulazne YACC datoteke

- Kao što samo ime kaže, odeljak `deklaracije` služi za deklarisanje osnovnih podataka koji se odnose na YACC, kao što je npr. navođenje spiska svih tokena gramatike koja opisuje ulazni jezik, navođenje prioriteta i asocijativnosti operatora, deklarisanje tipa atributa simbola date gramatike i slično. U ovom odeljku je moguće definisati i deklarisati razne globalne objekte (promenjive, funkcije, tipove, ...) na jeziku C koji se mogu koristiti u svim akcijama koje vrše obradu prihvaćenog ulaza. Taj C kod je potrebno navesti između

  ```
  %{ i %}
  ```

  "zagrada". Sve što je ovako ograđeno se bez ikakvih izmena prepisuje u rezultujuću (`y.tab.c`) datoteku koja sadrži rutine za parsiranje. Svi objekti navedeni na ovaj način imaju globalnu oblast važenja tj. to su objekti čija je oblast važenja datoteka `y.tab.c`.

- U odeljku `funkcije` se obično navode funkcije pisane u programskom jeziku C. Tekst koji se navodi u delu `funkcije` se takođe bez ikakve obrade dopiše na odgovarajuća mesta u `y.tab.c` datoteku. Ovde se obično navode rutine koje se koriste prilikom obrade ulaza i koje se mogu pozivati prilikom prihvatanja ulaza vođenog datim gramatičkim pravilima i njima pridruženim akcijama. Nije retko da se na ovom mestu navede i definicija funkcije `main()` kao i funkcije `yylex()` koja vrši fazu leksičke analize kako bi se prevođenjem generisane datoteke `y.tab.c` odmah dobila izvršna verzija programa.

- Centralni deo YACC-ove ulazne datoteke predstavlja odeljak `pravila` u kome se kontekst slobodnom gramatikom opisuje ulazni jezik programa i u kome se svakom gramatičkom pravilu pridružuju akcije koje predstavljaju kod koji se izvršava prilikom prepoznavanja dela ulaza opisanog tim pravilom. Svako gramatičko pravilo opisuje dopustive formate dela teksta na ulazu.

1.4 Navođenje gramatičkih pravila

Gramatička pravila se navode tako što se leva strana od desne razdvaja dvotačkom (`:`) i svako se pravilo završava simbolom tačka-zarez (`;`). YACC-ova sintaksa ne pravi razliku između imena tokena i neterminala i njihova imena mogu biti proizvoljno dugački nizovi slova, cifara, tačkica(`.`) i podvlaka (`_`) koji ne počinju cifrom. YACC pravi razliku između velikih i malih slova. Međutim,

nepisano pravilo, koje može značajno poboljšati čitljivost gramatičkih pravila, je da se neterminali pišu malim, a tokeni velikim slovima.

Dakle, primer pravila može biti

```
datum : DAN '.' IME_MESECA GODINA '.';
```

Na ovom mestu pretpostavljamo da `DAN`, `IME_MESECA`, i `GODINA` opisuju kategorije teksta sa ulaza koje su definisane na nekom drugom mestu. Simboli koji se navedu između navodnika (''), u ovom slučaju tačkice, se nazivaju literalima i predstavljaju tokene za koje se očekuje da se bukvalno pojave na odgovarajućem mestu ulaza. Dake, navedeno pravilo, uz takve dodatne definicije omogućava da se tekst `1. Januar 2004.` može smatrati datumom.

Ukoliko postoji više pravila sa zajedničkom levom stranom, ona se mogu objediniti korišćenjem uspravne crte (|) kao simbola alternacije. Npr. ukoliko je uz već navedeni format datuma dopušten i format opisan pravilom

```
datum : DAN '/' MESEC '/' GODINA ;
```

koji omogućava da se npr. `1/1/2004` može smatrati datumom, ova dva pravila se stapaju u jedno na sledeći način:

```
datum : DAN '.' IME_MESECA GODINA '.'
      | DAN '/' MESEC '/' GODINA
      ;
```

Naglasimo da se ε-pravila u gramatici predstavljaju na veoma uobičajeni način. Npr $A \to \varepsilon$ se predstavlja kao

```
A :
  ;
```

Da bi kontekstno slobodna gramatika bila u potpunosti definisana potrebno je razdvojiti terminalne simbole (tokene) od neterminalnih. YACC zahteva eksplicitno deklarisanje svakog tokena. Najlakši način da se ovo uradi je korišćenje direktive `%token` i to u delu `deklaracije` YACC datoteke tj. u delu pre prvog pojavljivanja simbola `%%`. Ova direktiva se koristi tako što se za njom navede lista imena simbola koje ćemo smatrati tokenima.

```
%token ime1, ime2, ... , imen
```

Sva imena koja se pojavljuju u gramatici, a koja nisu deklarisana kao tokeni se smatraju neterminalima i YACC zahteva da se svaki od njih pojavi na levoj strani bar jednog od navedih gramatičkih pravila.

Početni simbol (aksioma) gramatike se navodi korišćenjem direktive `%start` u odeljku `deklaracije`.

```
%start neterminal
```

Ukoliko se ova direktiva izostavi, aksiomom se smatra neterminal koji se nalazi na levoj strani prvog navedenog gramatičkog pravila.

1.5 Leksička analiza

1.5.1 Odnos između leksičkog i sintaksnog analizatora

Prilikom izgradnje parsera, prilično je bitno dobro podeliti posao između leksičkog i sintaksnog analizatora. Ovo je obično nešto što se može na više načina odraditi, tako je bitno prodiskutovati prednosti i nedostatke svakog od pristupa. Ukoliko se vratimo na primer definicije datuma iz poglavlja 1.4, postavlja se dilema da li uzeti da je `IME_MESECA` token ili neterminal? Ukoliko se odlučimo za neterminal, potrebno ga je posebno definisati što se može uraditi npr. ovako

```
IME_MESECA : 'J' 'a' 'n' 'u' 'a' 'r'
           | ...
           | 'D' 'e' 'c' 'e' 'm' 'b' 'a' 'r'
           ;
```

Iako se u ovom slučaju leksički analizator trivijalno gradi, jer je potrebno samo da čita i prepoznaje pojedinačna slova ulazne azbuke, ovaj pristup ima mnogo mana. Mnoštvo ovakvih gramatčkih pravila dovodi do previše kompleksnog rezultujućeg parsera i u vremenskom i u prostornom pogledu. Osim usporavanja procesa parsiranja, javljaju se problemi i prilikom pokušaja izmene specifikacije jer to dodatno komplikuje gramatiku i zahteva ponovno generisanje parsera.

Generalne smernice ukazuju na to da je poželjno što veći deo procesa prihvatanja ulaza realizovati kroz fazu leksičke analize tj. pogodnim odabirom azbuke tokena smanjiti broj i kompleksnost gramatičkih pravila.

Primeri tokena koji se obično javljaju u realnim parserima su `NUM` koji označava brojeve (ovo se u zavisnosti od potrebe može podeliti na `INT`, `REAL`, itd.), `ID` koji označava identifkikatore, posebni tokeni (npr. `IF`, `WHILE`,...) za ključne reči programskih jezika i slično. Jednokarakterski tokeni se obično ne imenuju već se na njih referiše preko literala tj. u gramatici se navode pod navodnicima (`''`).

1.5.2 Izgradnja leksičkog analizatora

Da bismo mogli da dobijemo potpuno funkcionalni program, neophodno je izgraditi leksički analizator čiji je zadatak da čita niz karaktera iz ulazne datoteke i prepoznaje jedan po jedan token. Ovo se realizuje preko funkcije

```
int yylex()
```

koja vraća celobrojni kod koji odgovara poslednjem prepoznatom tokenu.

Da bi sistem mogao da funkcioniše, neophodno je pridržavati se određenih konvencija. Naime, potrebno je imati u vidu činjenicu da YACC interno svakom gramatičkom simbolu dodeljuje celobrojni kod. Da bi sistem funkcionisao, neophodno je da se ovi kodovi poklapaju sa kodovima tokena koje vraća leksički analizator. Za svaki deklarisani token, YACC uvodi (`#define`) definiciju simbola sa njegovim imenom što omogućuje da se unutar leksičkog analizatora tokeni referišu simbolički. Standardno, prvom tokenu se dodeljuje kod 257 i kodovi redom rastu. Prvih 255 vrednosti je rezervisano za jednokarakterske simbole tj. literale koji se, podsetimo se, u gramatici navode između navodnika (`''`). Svakom ovakvom simbolu se pridružuje njegov ASCII kod i ovo je

potrebno ispoštovati u funkciji `yylex()`. Kraju ulaza se pridružuje poseban token koji se obično naziva *endmarker* i kome se pridružuje kod 0.

Imajući ovo u vidu, evo kako bi funkcija `yylex()` mogla da izgleda. U ovom primeru pretpostavljamo da je u odeljku deklaracija deklarisan token `NUM`.

```
int yylex()
{  /* Citamo karakter sa ulaza */
   char c=getchar();

   /* Pronalazimo jednokarakterske tokene */
   switch(c)
   {    case '(' :    case ')' :
        case '+' :    case '-' :
        case '*' :    case '/' :
        case '=' :
             return c;
       /* U slucaju kraja ulaza vracamo specijalnu vrednost 0 */
        case EOF    :
             return 0;
   }

   /* Prepoznajemo cele brojeve */
   if (isdigit(c))
   {    while(isdigit(c=getchar()));

        /* Vracamo suvisni karakter u ulaznu struju */
        ungetc(c,stdin);
        return NUM;
   }

   /* Prijavljujemo gresku u slucaju
      da ne mozemo da identifikujemo token*/
   yyerror("Greska : Nepoznati karakter %c", c);
   return 0;
}
```

Ukoliko se funkcija `yylex()` deklariše unutar dela `funkcije` ulazne YACC datoteke, njen se kod smešta u `y.tab.c` datoteku, tako da su joj sve simboličke konstante koje odgovaraju tokenima dostupne. Ukoliko se, pak, funkcija `yylex()` definiše u zasebnoj datoteci, potrebno je nekako uključiti pomenute deklaracije. Standardni mehanizam da se ovo uradi je da YACC, na osnovu ulazne specifikacije, pored datoteke `y.tab.c` koja sadrži ključne rutine parsera, automatski generiše i zaglavlje

`y.tab.h`

koja sadrži pomenute simboličke definicije tokena i eventualno još neke objekte koji predstavljaju interfejs parsera koji konstruišemo prema ostatku sistema, a o kojima će biti reči kasnije. U većini verzija YACC prevodioca se to radi navođenjem opcije

`-d`

prilikom prevođenja ulazne YACC datoteke koja sadrži specifikaciju parsera. Npr. posle kreiranja specifikacije parsera u datoteci `parser.y`, navođenjem komande

```
yacc -d parser.y
```

u komandnoj liniji operativnog sistema dobijamo datoteke `y.tab.c` i `y.tab.h`.

Ukoliko se leksički analizator navodi u posebnoj datoteci, dovoljno je na njenom početku uključiti zaglavlje `y.tab.h` i možemo nadalje koristiti simboličke definicije tokena.

1.6 Primer : Ispravnost aritmetičkih izraza

U ovom trenutku smo spremni da navedemo prvi kompletan primer. Znanje koje imamo nam omogućuje pisanje specifikacije parsera koji prihvataju dati ulaz i ispituju da li je taj ulaz pripada ulaznom jeziku opisanom datom specifikacijom.

Primer 1 *Napisati program koji za niz karaktera koji se unosi sa standardnog ulaza ispituje da li predstavlja ispravno formirani aritmetički izraz.*

Evo kako bi izgledao kompletan listing specifikacije sa ručno kodiranim leksičkim analizatorom.

```
/* Datoteka : ispravnost_izraza.y */

%{
/* Ova zaglavlja ukljucujemo za potrebe leksickog analizatora */
#include <stdio.h>
#include <ctype.h>

/* Funkcija za obradu gresaka */
#define yyerror printf

/* Prototipovi funkcija */
int main();
int yylex();
%}

%token NUM
%start expr
%%
    expr    :   expr '+' term
            |   term
            ;
    term    :   term '*' factor
            |   factor
            ;
    factor  :   NUM
            | '(' expr ')'
            ;
```

```
%%
/* Glavni program */
int main()
{   printf("Unesite aritmeticki izraz : ");
    if (yyparse()==0)
    {   printf("Uneti izraz je ispravan\n");
        return 0;
    }
    else
        return 1;
}

/* Leksicki analizator */
int yylex()
{  /* Citamo karakter sa ulaza */
   char c=getchar();

   /* Pronalazimo jednokarakterske tokene */
   switch(c)
   {    case '(' :
        case ')' :
        case '+' :
        case '*' :
             return c;
       /* Prelazak u novi red smatramo krajem ulaza */
        case '\n' :
             return 0;
   }

   /* Prepoznajemo cele brojeve */
   if (isdigit(c))
   {     while(isdigit(c=getchar()))
                 ;
        /* Vracamo suvisni karakter u ulaznu struju */
        ungetc(c,stdin);
        return NUM;
   }

   /* Prijavljujemo gresku u slucaju
      da ne mozemo da identifikujemo token*/
   yyerror("Greska : Nepoznati karakter %c\n", c);
   exit(1);
}
```

Da bi se od ove specifikacije dobila izvšna verzija programa potrebno je prvo pokrenuti YACC koji će da generiše `y.tab.c` datoteku, a zatim nju prevesti korišćenjem nekog C prevodioca. U daljem tekstu će biti pretpostavljeno da je operativni sistem koji koristimo Linux i da će C prevodilac biti `gcc` tako da je niz komandi koji dovodi do izvršne verzije programa

```
yacc ispravnost_izraza.y
```

```
gcc -oispravnost_izraza y.tab.c
```

Izvršna verzija programa ne radi puno toga. Od korisnika se ocekuje da unese aritmeticki izraz. Ukoliko se prilikom parsiranja naiđe na grešku ispisuje se podrazumevana poruka `syntax error` i u tom slučaju funkcija `yyparse()` vrednost različitu 0. Ukoliko je parsiranje uspelo, ova funkcija vraća 0 i u funkciji `main()` se ispisuje naša poruka da je parsiranje uspelo.

Primetimo kako je, umesto da se funkcija `yyerror` koja vrši prikazivanje poruka o greškama posebno definiše, napravljena jednostavna makro definicija koja sva pojavljivanja poziva ove funkcije menja pozivima funkcije `printf`. Ovo je bilo moguće uraditi zbog kompatibilnosti liste argumenata ove dve funkcije. Iako je ovo rešenje najlakše i veoma se često primenjuje, u slučaju da želimo iole bolju obradu grešaka ono nije zadovoljavajuće. U poglavlju **??** će biti prikazane naprednije tehnike za obradu grešaka.

1.7 Kako radi generisani parser?

Iako razumevanje rada već konstruisanog parsera nije neophodno za korišćenje samog sistema YACC, ono može biti prilično korisno za razumevanje grešaka koje mogu nastati prilikom zadavanja specifikacije. Razumevanje rada parsera takođe može pomoći prilikom razrešavanja konflikata koji mogu da nastanu, kao i pri otkrivanju i otklanjanju višeznačnosti ulazne gramatike.

Parseri koje YACC generiše rade tako što simuliraju rad potisnog automata koji vrši sintaknsu analizu naviše (bottom up). Da bi ovakav automat mogao da bude konstruisan neophodno je da je ulazni jezik u YACC specifikaciji opisan gramatikom koja pripada klasi LALR(1). Algoritam konstrukcije potisnog automata za datu LALR gramatiku je prilično kompleksan i na ovom mestu ga nećemo navoditi, a njegov se opis može pronaći u [2], [1].

Parser tj. potisni automat se odlikuje skupom svojih unutrašnjih stanja koja se kodiraju celim brojevima i stekom koji može da čuva cele brojeve koji predstavljaju stanja automata. Stanje koje se nalazi na vrhu steka se smatra tekućim.

Parser je takođe sposoban da pročita i upamti token koji sledi u ulaznoj struji i njega nazivamo preduvidni token (lookahead token). Ukoliko je preduvidni token potreban, a nije dostupan, parser poziva leksični analizator tj. funkciju `yylex()` na osnovu čijeg odgovora postavlja vrednost preduvidnog tokena. Prilikom početka parsiranja automat je u stanju 0, stek sadrži jedino 0, i nije pročitan preduvidni token.

Sistem YACC omogućuje korisnicima da na osnovu svoje specifikacije parsera generišu datoteku koja se obično naziva

```
y.output
```

i koja sadrži opis potisnog automata koji vrši parsiranje u obliku čitljivom za čoveka. Da bi se ovakva datoteka generisala potrebno je YACC pokrenuti sa parametrom koji se u većini verzija označava sa

```
-v
```

Pretpostavimo da smo u datoteci `primer.y` napisali sledeću specifikaciju parsera

```
%token INT ID NUM
%%
    program : deklaracija
            | definicija
            ;
deklaracija : INT ID
            ;
definicija  : INT ID '=' NUM
            ;
```

Ukoliko u komandnoj liniji operativnog sistema unesemo

```
yacc -v primer.y
```

sistem će generisati datoteku `y.output` sa sledećim sadržajem

```
   0  $accept : program $end
   1  program : deklaracija
   2          | definicija
   3  deklaracija : INT ID
   4  definicija : INT ID '=' NUM

state 0
    $accept : . program $end  (0)

    INT  shift 1
    .  error

    program  goto 2
    deklaracija  goto 3
    definicija  goto 4


state 1
    deklaracija : INT . ID  (3)
    definicija : INT . ID '=' NUM  (4)

    ID  shift 5
    .  error


state 2
    $accept : program . $end  (0)

    $end  accept


state 3
    program : deklaracija .  (1)
```

```
    .  reduce 1


state 4
    program : definicija .  (2)

    .  reduce 2


state 5
    deklaracija : INT ID .  (3)
    definicija : INT ID . '=' NUM  (4)

    '='  shift 6
    $end  reduce 3


state 6
    definicija : INT ID '=' . NUM  (4)

    NUM  shift 7
    .  error


state 7
    definicija : INT ID '=' NUM .  (4)

    .  reduce 4


6 terminals, 4 nonterminals, 5 grammar rules, 8 states
```

Svako od stanja sadrži u sebi skup tzv. ajtema (item) koji prestavljaju gramatička pravila sa umetnutim simbolom `'.'`. Neformalno govoreći, ovaj simbol služi da pokaže dokle se u tom trenutku stiglo sa čitanjem ulaza. Posmatrajmo stanje 1 u navedenom primeru. Ono sadrži sledeća dva ajtema

```
deklaracija : INT . ID  (3)
 definicija : INT . ID '=' NUM  (4)
```

Ovde brojevi u zagradi označavaju redni broj gramatičkog pravila i to u redosledu koji je dat na početku `y.output` datoteke. Ovo znači da je automat u stanju 1 ukoliko je krenuo da čita deklaraciju ili je krenuo da čita definiciju, pročitao je simbol `INT` i sada očekuje ostatak nekog od ovih pravila.

Posle skupa ajtema, svako stanje karakteriše i tzv. tablica prelaska. Tablica prelaska se sastoji od skupa pravila oblika

```
TOKEN akcija broj
```

Ovo znači da ako je u tekućem stanju preduvidni token jednak specifikovanom, izršava se navedena `akcija` i to sa parametrom `broj`. Ukoliko u nekom pravilu umesto oznake tokena nalazi tačkica (`'.'`), to označava da se ovo pravilo

primenjuje u slučaju da preduvidni token nije ni jedan od tokena navedenih u prethodnim pravilima tog stanja.

Neka stanja takođe sadrže i specifikaciju tzv. `goto` funkcije koja tekućem stanju i određenom neterminalu pridružuje novo stanje.

Automat je u stanju da izvršava četiri vrste akcija `shift`, `reduce`, `accept`, i `error`.

- `shift` (potiskivanje) - Ovo je akcija koju parser najčešće izvršava. Prilikom izvršavanja ove akcije, potrebno je obavezno konsultovati preduvidni token. U zavisnosti od tekućeg stanja automata i preduvidnog tokena parser, na osnovu svojih tablica prelaska, određuje sledeće stanje, postavlja ga na vrh steka i sinhronizuje vrednost preduvidnog tokena. Npr., ukoliko je parser u stanju 0 tj. na vrhu steka se nalazi broj 0, preduvidni token je `NUM`, i u tablici prelaska postoji akcija

  ```
  state 0
          .......
          INT   shift 1
  ```

  na vrh steka se postavlja 1 (iznad 0) i preduvidni token se sinhronizuje.

- `reduce` (svođenje) - ova akcija se može primeniti u trenutku kada je parser pročitao desnu stranu gramatičkog pravila i kada je spreman da je zameni neterminalom sa leve strane tog pravila. Ovo se ogleda tako što stanje sadrži ajtem kome se tačkica (`'.'`) nalazi iza poslednjeg simbola desne strane. Ponekad je potrebno konsultovati i preduvidni token kako bi se odlučilo da li se vrši svođenje i na osnovu kog gramatičkog pravila, ali obično je slučaj da ova informacija nije potrebna tako da se svođenje može vršiti i u slučaju nepoznatog preduvidnog tokena.

  Npr. ukoliko se automat nalazi u stanju 5, i preduvidni token je jednak endmarker-u, vrši se svođenje na osnovu pravila

  ```
  deklaracija : INT ID .  (3)
  ```

  dok se u stanju tri vrši redukcija na osnovu pravila

  ```
  program : deklaracija .  (1)
  ```

  šta god da je preduvidni token.

  Prilikom vršenja svođenja, sa steka se skida onoliko stanja koliko ima simbola sa desne strane pravila. To znači da nas akcija svođenja vraća u trenutak pre nego što je počelo prihvatanje niske izvedene iz desne strane pravila. U ovom trenutku bi parser trebalo da se ponaša kao da je video levu stranu upravo svedenog pravila i na osnovu ovog jednog neterminala je potrebno odrediti novo stanje. Međutim, nije potpuno jasno koje je to stanje jer to zavisi od toga u koji smo se trenutak vratili tj. u kom se stanju nalazio automat pre nego što je počeo da čita desnu stranu upravo svedenog pravila. To stanje nije teško odrediti jer se ono upravo nalazi na vrhu steka. Problem se sada lako razrešava konsultovanjem funkcije `goto`, jer nam ona upravo govori o tome u koje stanje treba preći, ukoliko je automat bio u određenom stanju i upravo je pročitana niska koja se izvodi iz datog neterminala.

- `accept` - (prihvatanje) Ova akcija označava da je ceo ulaz pročitan i da parser treba da uspešno završi rad. Ova se akcija poziva samo u slučaju da je preduvidni token endmarker.

- `error` - (greška) Ova akcija označava da parser ne može da nastavi parsiranje ulaza tj. da pročitani ulaz nije u skladu sa datom specifikacijom. Parser u ovom trenutku prijavljuje grešku i pokušava da se oporavi na načine koji će kasnije biti objašnjeni.

Imajući sve ovo u vidu, prikažimo rad parsera iz gornjeg primera na prihvatanju niske `int x=0;`

stek	ulaz	akcija
0	`INT ID = NUM $end`	`shift 1`
0 1	`ID = NUM $end`	`shift 5`
0 1 5	`= NUM $end`	`shift 6`
0 1 5 6	`NUM $end`	`shift 7`
0 1 5 6 7	`$end`	`reduce definicija : INT ID = NUM`
0	`$end`	(`goto definicija`)
0 4	`$end`	`reduce program : definicija`
0	`$end`	(`goto program`)
0 2	`$end`	`accept`

U početku rada je parser u stanju 0. Da bi parser doneo odluku koju akciju treba da izvrši potrebno je pročitati preduvidni token i leksički analizator ga postavlja na INT. Konsultovanjem tablica prelaska vidi se da se za ovaj preduvid i stanje nula vrši akcija `shift 1` tako da se na stek postavlja 1, a novi preduvidni token postaje `=`. Konsultovanjem tablica prelaska vidi se da se u stanju 1, pri preduvidnom tokenu `=` vrši akcija `shift 5`. Slično tako se dolazi do akcija `shift 6`, `shift 7` i dolazi se do situacije kada se na steku nalaze stanja 0, 1, 5, 6 i 7 i preduvidni karakter je endmarker. U stanju 7 se nailazi na akciju `reduce 4` koja se vrši bez obzira na to šta je preduvidni token, gde je 4 redni broj pravila `definicija : INT ID '=' NUM`. U tom trenutku se sa steka skida po jedno stanje za svaki od simbola sa desne strane ovog pravila i parser dolazi u stanje da se na steku nalazi simbol 0, ali u ovom trenutku je poznato da je pročitana definicija. U stanju 0, konsultovajući `goto` tablicu pronalazimo da se posle čitanja definicije prelazi u stanje 4. Stanje 4, definiše akciju `reduce 2`, gde je 2 pravilo `program : definicija`, tako da se sa steka skida simbol koji odgovara simbolu definicija i parser prelazi opet u stanje 0, sa znanjem da je pročitana niska izvedena iz neterminala `program`. Opet na osnovu tablice `goto` u stanju 0 nalazimo da je potrebno preći u stanje 2. U ovom stanju se sa preduvidom `endmarker` nalazi da je odgovarajuća akcija `accept` i parser uspešno završava rad.

1.7.1 Konflikti

Prilikom konstrukcije potisnog automata je moguće da se dogodi da u nekom stanju automat može da izvrši više akcija koje vode ka uspešnom prepoznavanju ulaznog jezika. Problem je u tome što izbor svake od ovih akcija obično vodi različitom drvetu izvođenja i samim tim različitom sintaksnom drvetu. Postoje dve vrste konflikta koje mogu da nastanu

1. `shift/reduce` *konflikti* - ova situacija nastaje kada u nekom stanju automat može da bira između akcije `shift` tj. potiskivanja stanja na stek na osnovu preduvidnog tokena i akcije `reduce` na osnovu nekog pravila. Primeri ovih konflikata će biti navedeni u poglavlju 1.8.

2. `reduce/reduce` *konflikti* - ova situacija može nastati ukoliko je parser u stanju da izvrši svođenje na osnovu dva ili više pravila. Tipični primeri nastajanja ovog konflikta su u slučajevima da dva različita neterminala mogu da izvedu istu nisku. Npr.

```
...
promenjiva : ID
           ;
izraz      :  ID
           |  NUM
           ....
           ;
```

Ukoliko, prilikom konstrukcije parsera, YACC naiđe na gramatiku koja dovodi do konfliktnih situacija one se razrešavaju na osnovu sledećih pravila

1. Sistem korisniku prijavljuje upozorenje da je došlo do konfliktne situacije

2. U slučaju `shift/reduce` konflikta podrazumevana akcija je `shift`

3. U slučaju `reduce/reduce` konflikta, vrši se svođenje na osnovu pravila koje je ranije navedeno u gramatici.

Prilikom pojave upozorenja u konfliktu, dobra praksa je generisati datoteku `y.output` i proveriti da li su sve podrazumevane akcije upravo ono što ste želeli. Pre opisa stanja u kome postoji određeni konflikt, nalazi se opis nastalog konflikta oblika npr.

```
4: shift/reduce conflict (shift 3, reduce 1) on '+'
```

Potrebno je određena količina iskustva kako bi se u potpunosti ovladalo pisanjem gramatika koje neće izazivati konflikte i ovladalo tehnikom uspešnog razrešavanja ovih konflikata.

1.8 Korišćenje višeznačnih (ambiguous) gramatika

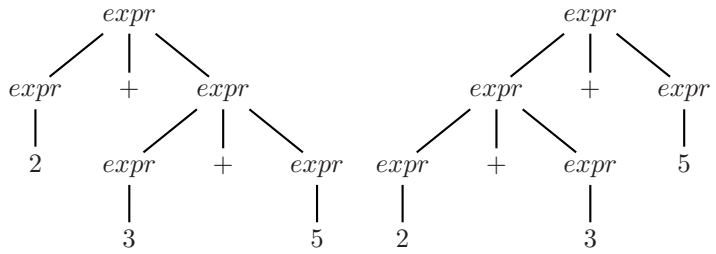
Iako gramatike koje su višeznačne prouzrokuju konflikte, postoje posebni mehanizmi da se ti konflikti otklone, bez menjanja samih gramatika. Razlog uvođenja ovakvih mehanizama, uz ostavljanje mogućnosti navođenja višeznačnih gramatika je to što se često neki konstrukti prilično jednostavno opisuju ovakvim gramatikama, dok svaka transformacija gramatike koja uklanja višeznačnost smanjuje čitljivost same gramatike, čini je mnogo kompleksijom što kao krajnji ishod ima glomazniji i sporiji rezultujući parser.

1.8.1 Koršćenje informacije o asocijativnosti tokena

Posmatrajmo primer sledeće jednostavne gramatike aritmetičkih izraza

```
expr : expr '+' expr
     | NUM
     ;
```

Jasno je da je ova gramatika višeznačna, jer npr. za nisku `2+3+5` postoji više različitih drveta izvođenja.



Slika 1.3: Drveta izvođenja za nisku 2+3+5

Nastali problem i njegovo rešenje je u tesnoj vezi sa vrstom asocijativnosti operatora `'+'` jer prvo drvo jasno odgovara desno asocijativnoj operaciji, dok drugo odgovara levo asocijativnoj operaciji.

Uočena višeznačnost se jasno ogleda kroz `shift/reduce` konflikt koji nastaje

```
4: shift/reduce conflict (shift 3, reduce 1) on '+'
  state 4

        expr : expr . '+' expr  (1)
        expr : expr '+' expr .  (1)

        '+'  shift 3
        $end  reduce 1
```

Podrazumevana akcija `shift` uvodi desnu asocijativnost, dok bi alternativna `reduce` akcija dovela do leve asocijativnosti.

YACC ostavlja korisnicima mogućnost navođenja asocijativnosti svakog tokena i to direktivama

`%left`	levo asocijativni token
`%right`	desno asocijativni token
`%nonassoc`	neasocijativni token

Ove direktive se navode u delu `deklaracije` datoteke koja sadrži YACC specifikaciju. Ukoliko se za neki token navodi vrsta asocijativnosti pomoću navedenih direktiva, nije potrebno navoditi njegovo ime pomoću `%token` deklaracije.

U slučaju nastalog konflikta pri pojavi tokena na ulazu koriste se sledeća pravila:

1. U slučaju da token nema definisanu asocijativnost prijavljuje se upozorenje da konflikt postoji, a konflikt se razrešava na osnovu podrazumevanih pravila navedenih u poglavlju 1.7.1.

2. U slučaju leve asocijativnosti konflikt se ne prijavljuje već se prednost daje akciji `reduce`.

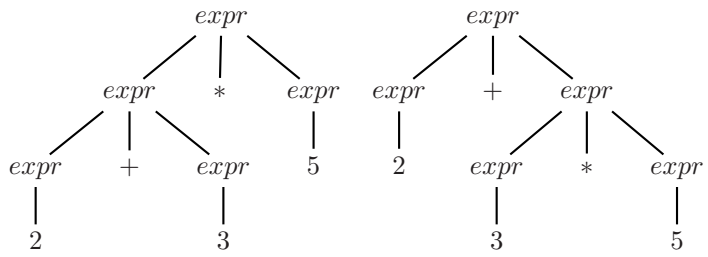
3. U slučaju desne asocijativnosti konflikt se ne prijavljuje već se prednost daje akciji `shift`.

1.8.2 Korišćenje informacije o prioritetu tokena

Dodajmo prethodnoj gramatici i pravilo koje se odnosi na množenje

```
expr : expr '+' expr
     | expr '*' expr
     | NUM
     ;
```

Uočimo nisku `2+3*5`. Za ovu nisku opet postoje dva drveta izvođenja



Slika 1.4: Drveta izvođenja za nisku 2+3*5

Ovaj put problem ne leži u asocijativnosti već u prioritetu operatora. Imajući u vidu uobičajena matematička pravila jasno je da je drugo drvo u ovom slučaju poželjnije, ali u samoj gramatici ta informacija nije nigde navedena.

Opet se ovakva situacija jasno ogleda u konfliktima koji nastaju

```
5: shift/reduce conflict (shift 3, reduce 1) on '+'
5: shift/reduce conflict (shift 4, reduce 1) on '*'
   state 5

        expr : expr . '+' expr  (1)
        expr : expr '+' expr .  (1)
        expr : expr . '*' expr  (2)

        '+'  shift 3
        '*'  shift 4
        $end  reduce 1


6: shift/reduce conflict (shift 3, reduce 2) on '+'
6: shift/reduce conflict (shift 4, reduce 2) on '*'
   state 6

        expr : expr . '+' expr  (1)
```

```
expr : expr . '*' expr  (2)
expr : expr '*' expr .  (2)

'+'  shift 3
'*'  shift 4
$end  reduce 2
```

Pored već viđenih konflikata (prvog u stanju 5. i drugog u stanju 6.) koji nastaju usled asocijativnosti, javila su se još dva nova koji oslikavaju problem nepoznavanja prioriteta operatora. Npr. ukoliko je parser u stanju

```
expr : expr '+' expr .  (1)
```

a na ulazu je token `*`, akcija `reduce` daje prioritet operatoru `+` dok podrazumevana akcija `shift` daje prioritet operatoru `*`, što je u skladu sa očekivanim ponašanjem. Međutim, ukoliko je parser u stanju

```
expr : expr '*' expr .  (2)
```

i na ulazu je `+`, podrazumevani `shift` daje prioritet sabiranju što svakako nije ono što želimo.

YACC omogućava određivanja prioriteta operatora direktno iz dela `definicija` u kome su tokeni bili deklarisani preko `%left`, `%right`, `%nonassoc` i `%token` direktiva. Svim tokenima deklarisanim pomoću jedne direktive se dodeljuje isti prioritet. Što su tokeni kasnije definisani dodeljuje im se veći prioritet. Imajući ovo u vidu, gramatici izraza iz primera je potrebno dodati sledeće informacije kako bi se svi konflikti pravilno razrešili

```
%right '='
%left  '+' '-'
%left  '*' '/'
%%
 expr : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | expr '=' expr
      | '(' expr ')'
      | NUM
      ;
```

Ovim je operator dodele `=` definisan kao desno asocijativan sa najnižim prioritetom. Svi ostali operatori su definisani kao levo asocijativni i smatra se da `+` i `-` imaju isti prioritet koji je manji od prioriteta operatora `*` i `/`, čiji je prioritet opet jednak. Korišćenjem zagrada `(` i `)` je ostavljena mogućnost eksplicitnog grupisanja. Na osnovu ovakve specifikacije, parser bi ulaz

```
a  =  b  =  c*(d+e)  -  f  -  g*h
```

struktuirao kao

```
a = ( b = ( ((c*(d+e))-f) - (g*h) ) )
```

Postavlja se pitanje kako se na osnovu ovakvih informacija razrešavaju `shift/reduce` konflikti koji nastaju. Svakom gramatičkom pravilu se takođe dodeljuje prioritet i asocijativnost i to tako što se za prioritet i asocijativnost pravila uzme prioritet i asocijativnost poslednjeg tokena koji se javlja sa njegove desne strane. Ukoliko se na desnoj strani ne pojavljuju tokeni, pravilu se ne dodeljuje prioritet ni asocijativnost. Kada se ovo zna, konflikti se razrešuju koristeći sledeća pravila:

1. U slučaju `reduce/reduce` ili `shift/reduce` konflikta u kome bilo token bilo pravilo koje u konfliktu učestvuje nema dodeljen prioritet, konflikt se prijavljuje, i razrešava se na osnovu podrazumevanih pravila navedenih u poglavlju 1.7.1.

2. U slučaju da su prioriteti poznati, konflikt se razrešava tako što se prednost da akciji (bilo `shift` bilo `reduce`) sa višim prioritetom

3. U slučaju da su prioriteti jednaki konflikt se razrešava na osnovu podataka o asocijativnosti kao što je opisano u poglavlju 1.8.1

Prioritet koji se dodeljuje pravilu se može promeniti korišćenjem za to namenjene `%prec` direktive. Ukoliko želimo da pravilu dodelimo prioritet određenog tokena na kraju desne strane pravila je potrebno navesti

```
%prec token
```

gde je `token` ime tokena čiji prioritet dodeljujemo pravilu.

Navešćemo sada dva primera u kojima se preko definisanje prioriteta razrešavaju konflikt, koji se jako često u praksi sreću.

Prvi primer se odnosi na upotrebu unarnih prefiksnih operatora u gramatikama izraza, kojima se po običaju dodeljuje jako visok prioritet. Ukoliko je token koji odgovara operatoru jedinstven, jedino što je potrebno je definisati taj token pri kraju definicija tokena. Međutim, veoma često se simbol unarnog prefiksnog operatora poklapa sa simbolom nekog binarnog infiksnog operatora koji ima relativno mali prioritet. To je naravno slučaj kod aritmetičkih operatora `-`. Evo kako se onda korišćenjem `%prec` direktive problem rešava

```
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
expr :    '('  expr  ')'
     |    expr  '+'  expr
     |    expr  '-'  expr
     |    expr  '*'  expr
     |    expr  '/'  expr
     |    '-'  expr  %prec  UMINUS
     ;
```

U ovom slučaju se poslednjem pravilu umesto malog prioriteta koji odgovara simbolu `'-'` dodeljuje veliki prioritet veštački uvedenog terminala `UMINUS`.

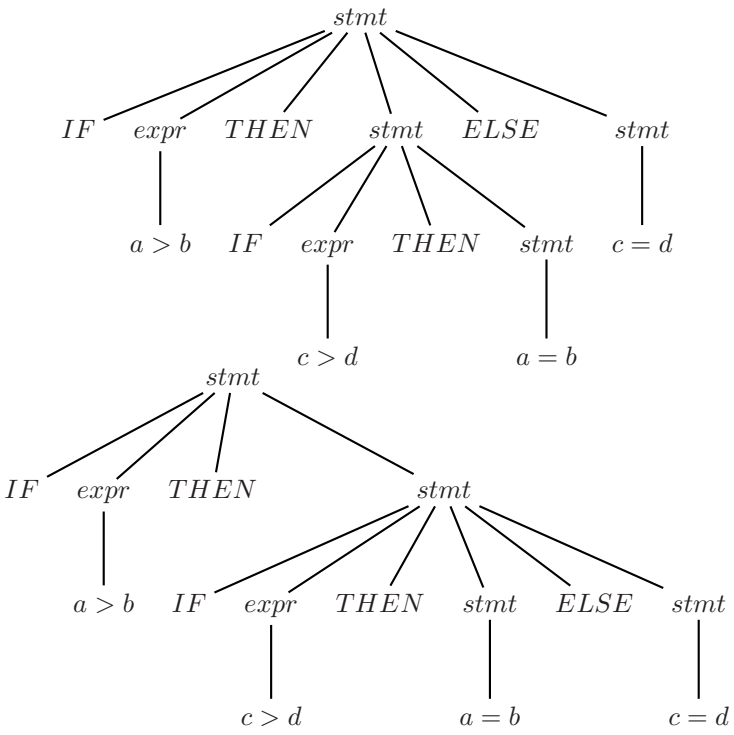
Drugi standardni primer korišćenja prioriteta je disambiguisanje IF-THEN-ELSE konstrukcije koja se često sreće u programskim jezicima. Naime, gramatika

```
stmt   :   IF expr THEN stmt
       |   IF expr THEN stmt ELSE stmt
       |   expr
       ;
```

je višeznačna, jer npr. niska

```
if a>b then if c>d then a=b else c=d
```

ima dva drveta izvođenja.



Slika 1.5: IF-THEN-ELSE višeznačnost

U slučaju stanja

```
6: shift/reduce conflict (shift 7, reduce 1) on ELSE state 6
        stmt : IF expr THEN stmt .  (1)
        stmt : IF expr THEN stmt . ELSE stmt  (2)

        ELSE  shift 7
        $end  reduce 1
```

javlja se konflikt i vidi se da je podrazumevana akcija `shift` što dovodi do toga se `ELSE` pridružuje poslednjem navedenom `IF`-u. Ovo ponašanje je upravo ono koje se javlja u modernim programskim jezicima i jasno je da je to ono što želimo. Ukoliko želimo da izbegnemo postojanje konflikta i upozorenje koje se tom prilikom javi, možemo da se poslužimo sledećom tehnikom. Osigurajmo da drugo pravilo ima veći prioritet od prvog. Poslednji terminal u prvom pravilu je `THEN`, a u drugom `ELSE` tako da jednostavno navođenje

```
%nonassoc THEN
%nonassoc ELSE
```

u odeljku definicija tokena razrešava problem. Ukoliko ne želimo da menjamo prioritet tokena `THEN` i `ELSE`, možemo da se poslužimo tehnikom uvođenja veštačkih tokena i korišćenja `%prec` direktive.

```
%nonassoc IF_THEN_PRIORITY
%nonassoc IF_THEN_ELSE_PRIORITY
%token IF THEN ELSE
%%
stmt    :   IF expr THEN stmt  %prec IF_THEN_PRIORITY
        |   IF expr THEN stmt ELSE stmt %prec IF_THEN_ELSE_PRIORITY
        |   expr
        ;
```

1.9 Semantičke akcije

U dosadašnjem tekstu smo se samo bavili problemom prihvatanja korisničkog ulaza, ali prilikom prihvatanja nismo vršili nikakvu obradu. Obrada ulaza se u YACC-u vrši umetanja semantičkih akcija u gramatiku.

Svakom gramatičkom pravilu je moguće pridružiti parče programskog koda koji nazivamo *akcija*. Akcija se izvršava u trenutku kada se, tokom prihvatanja ulaza, vrši svođenje (`reduce`) na osnovu odgovarajućeg pravila. Akcije se navode unutar vitičastih zagrada `{}` i sadrže proizvoljni kod u programskom jeziku C. Ovaj kod se skoro bez izmena umeće na odgovarajuća mesta rezultujuće `y.tab.c` datoteke. Npr. akcije koje vrše brojanje naredbi nekakvog programskog jezika bi mogle da budu

```
stmt : WHILE '(' condition ')' stmt
         { printf("Pojavila se WHILE naredba\n");
           broj_while++;
         }
     | IF '(' condition ')' stmt
         { printf("Pojavila se IF naredba\n");
           broj_if++;
         }
```

Iako to na prvi pogled nije očigledno, u suštini je postavljen zahtev da se akcije mogu navoditi samo na kraju gramatičkih pravila. Ukoliko se akcija navede negde "na sredini" pravila, npr.

```
A : B  { x=1; } C {y=1;}
  ;
```

gramatika se implicitno transformiše, umetanjem novih neterminala na mesto "unutrašnjih" akcija i novih ε pravila za tako uvedene neterminale, kako bi se ispoštovalo da se akcije nađu na kraju pravila. Pravilo iz gornjeg primera se transformiše u

```
A   : B POM C {y=1;}
    ;
```

```
POM :   {x=1;}
    ;
```

Imajući u vidu da se akcije pozivaju u trenutku svođenja (`reduce`) na osnovu odgovarajućeg gramatičkog pravila moguće je nedvosmisleno odrediti redosled njihovog primenjivanja.

1.10 Atributi i upotreba atributskog steka

Sistem YACC omogućava dodeljivanje određenih vrednosti (atributa) gramatičkim simbolima (tokenima i neterminalima) preko mehanizma atributskog steka (steka vrednosti). Naime, uz stek parsiranja (potisnu listu), YACC održava paralelno i atributski stek, koji je u svakom trenutku sinhronizovan sa potisnom listom. Svakom tokenu i neterminalu koji se javlja prilikom parsiranja odgovra određena vrednost na atributskom steku.

Prilikom svođenja na osnovu nekog gramatičkog pravila sa atributskog steka se "skida" onoliko vrednosti koliko je simbola sa desne strane pravila i na njega se postavlja vrednost koja odgovara neterminalu sa leve strane. Prilikom potiskivanja na osnovu nekog tokena, njegova vrednost se postavlja na atributski stek.

Tip ovih vrednosti je predstavljen simboličkom oznakom

`YYSTYPE`

koja podrazumevano označava celobrojni `int`. Najjednostavniji način promene tipa atributa je jednostavno navođenje pretprocesorske `#define YYSTYPE tip` ili `typedef tip YYSTYPE;` direkive u deo `deklaracije` ulazne YACC datoteke. U poglavlju 1.12 će pitanje promene tipa atributa biti detaljnije obrađeno.

Svakom tokenu je moguće pridružiti određenu vrednost (atribut). To se radi tako što se prilikom njegovog prepoznavanja u funkciji `yylex()` dodeli vrednost specijalnoj promenjivoj

`yylval`

Ova promenjiva je deklarisana unutar `y.tab.c` datoteke i njen tip odgovara tipu atributskog steka. Prilikom potiskivanja na osnovu nekog tokena vrednost promenjive `yylval` se postavlja na atributski stek i tako se postiže utisak da je preko ove promenjive dodeljena vrednost tokenu.

Vrednosti atributskog steka tj. vrednosti tokena i neterminala je moguće pristupiti direktno unutar akcija. To se realizuje malim proširivanjem jezika koji se koristi unutar akcija. Naime, uvode se specijalne pseudo-promenjive

`$$, $1, $2, $3, ...`

koje su tipa koji odgovara tipu atributskog steka. Dodeljivanjem vrednosti promenjivoj `$$`, dodeljujemo vrednost neterminalu sa leve strane pravila. Promenjiva `$1` odgovara vrednosti prvog gramatičkog simbola sa desne strane pravila, `$2` drugog itd. Na ovom mestu je važno podsetiti se da se umesto "unutrašnjih" akcija uvode novi neterminali, tako da brojanje simbola `$i` uključuje i akcije. Npr. ukoliko želimo da simbolu `A` dodelimo vrednost simbola `C` potrebno je navesti nešto ovako:

```
A  : B {printf("Unutrasnja akcija");} C {$$=$3;}
```

tj. zbog umetanja pomoćnog neterminala na mesto unutrašnje akcije, vrednosti simbola `C` odgovara `$3`.

Ono što na prvi pogled nije očigledno je da se svakom pravilu dodeljuje akcija, čak iako u specifikaciji ona nije navedena. Podrazumevana akcija koja se prilikom svođenja izvršava je

```
{$$=$1;}
```

tako da se neterminalu sa leve strane dodeljuje vrednost prvog simbola sa desne strane, ukoliko se drugačije ne navede. Ukoliko je ovo željeno ponašanje, pisanje akcije se može izostaviti, ali savet je da se zbog poboljšanja čitljivosti programa i ovakva akcija uvek eksplicitno navede. YACC ostavlja mogućnost da koristimo tzv. levi-kontekst, tj. pseudo-promenjive

```
$0, $-1, $-2, ...
```

Preko ovih promenjivih se pristupa vrednostima koje su dublje na steku od vrednosti prvog gramatičkog simbola desne strane pravila. Korišćenje levog konteksta spada u naprednije tehnike i trebalo bi ga izbegavati, ukoliko je moguće, dok se prilično ne ovlada sistemom. Navedimo jedan primer

```
recenica  :  pridev  imenica  glagol
          ;

pridev    :    OKRUGLA  {  $$ = okrugla_vr;  }
          |    PLAVA    {  $$ = plava_vr;    }
        . . .
          ;

imenica   :    LOPTA    {  $$ = lopta_vr;    }
          |    KOCKA    {  if( $0 == okrugla_vr )
                               printf( "Sta?\n" );
                           $$ = kocka_vr;
                        }
          ;
```

Imajući u vidu pravilo za rečenicu, jasno je da se `$0` odnosi na vrednost atributa pridruženog neterminalu `pridev`.

1.11 Primer : Kalkulator - prva verzija

U ovom trenutku možemo da konstruišemo kalkulator za celobrojnu aritmetiku.

Primer 2 *Napisati program koji sa standardnog ulaza čita aritmetički izraz u kome učestvuju celi brojevi i sve četiri aritmetičke operacije nad celim brojevima i, ukoliko je izraz ispravno unet, ispisuje na standardni izlaz njegovu vrednost.*

Umesto gramatike izraza iz prethodnog primera koristićemo višeznačnu gramatiku sa pravilima prioriteta i asocijativnosti kako je objašnjeno u poglavlju 1.8. Rečeno je da je atributski stek `int` tipa, što nam u ovom trenutku odgovara jer konstušemo kalkulator za celobrojnu aritmetiku. Jedini netrivijalni token će biti `NUM` i njegovu vrednost ćemo postavljati u fazi leksičke analize.

Akcije koje pridružujemo pravilima poštuju definiciju vrednosti izraza. Naime, ukoliko je izraz brojevna konstanta, njegova vrednost je jednaka vrednosti tog broja. To se oslikava u pravilu i akciji

```
expr :  NUM {$$=$1;} ;
```

pri čemu se na ovom mestu podrazumeva da je tokenu NUM u fazi leksičke analize dodeljena vrednost brojne konstante koju predstavlja, što je u programu stvarno i urađeno dodeljivanjem ove vrednosti promenjivoj `yylval`.

Akcije koje odgovaraju binarnim operatorima su oblika

```
expr : expr OP expr { $$=primeni_operaciju($1,$3); }
```

što govori da se vrednost izraza sa leve strane pravila sračunava primenom odgovarajuće operacije na već sračunate vrednosti izraza koji predstavljaju levi odnosno desni operand. Ovo odgovara sintetizovanju atributa na sintaksnom drvetu i odgovara sintaksnoj analizi naviše. U našem primeru za sve dopuštene operacije kalkulatora postoji operator jezika C koji je izračunava tako da se primena operacije svede na korišćenje odgovarajućeg C operatora (`$$=$1 OP $3;`).

Jedini problem koji je ostalo rešiti je kako na kraju ispisati vrednost sračunatog izraza. Rešenje je uvodenje novog, pomoćnog, neterminala koji će biti aksioma gramatike. Iako svako izvođenje počinje od ovog simbola, prilikom prihvatanja ulaza, svođenje na ovaj neterminal će se izvršiti poslednje, u trenutku kada je prihvaćen celokupni aritmetički izraz i kada je njegova vrednost već sračunata. U trenutku tog svođenja, dakle, jednostavno možemo ispisati vrednost unetog izraza.

Evo kako izgleda celokupna specifikacija

```
/* Datoteka : celobrojni_kalkulator.y */
%{
/* Ova zaglavlja ukljucujemo za potrebe leksickog analizatora */
#include <stdio.h>
#include <ctype.h>

/* Funkcija za obradu gresaka */
#define yyerror printf

/* Prototipovi funkcija */
int main();
int yylex();
%}

%left '+' '-'
%left '*' '/'
%left UMINUS
%start single_expression
%%
```

```
single_expression : expr
                   {printf("Vrednost izraza je : %d\n",$1);}
                  ;
expr :    '('  expr  ')'
           {$$=$2;}
     |    expr  '+'  expr
          {$$=$1+$3;}
     |    expr  '-'  expr
          {$$=$1-$3;}
     |    expr  '*'  expr
          {$$=$1*$3;}
     |    expr  '/'  expr
          {if ($3==0)
              yyerror("Deljenje nulom\n");
           else
              $$=$1/$3;
          }
     |    '-'  expr %prec  UMINUS
          {$$=-$2;}
     |    NUM
          {$$=$1;}
     ;
%%
/* Glavni program */
int main()
{   printf("Unesite aritmeticki izraz : ");
    return yyparse();
}

/* Leksicki analizator */
int yylex()
{  /* Citamo karakter sa ulaza */
   char c=getchar();

   /* Pronalazimo jednokarakterske tokene */
   switch(c)
   {    case '(' : case ')' :
        case '+' : case '-':
        case '*' : case '/':
             return c;
       /* Prelazak u novi red smatramo krajem ulaza */
        case '\n' :
             return 0;
   }

   /* Prepoznajemo cele brojeve */
   if (isdigit(c))
   {   /* Sracunavamo vrednost tokena NUM */
       yylval=c-'0';
```

```
        while(isdigit(c=getchar()))
            yylval=10*yylval+c-'0';

        /* Vracamo suvisni karakter u ulaznu struju */
        ungetc(c,stdin);
        return NUM;
    }

    /* Prijavljujemo gresku u slucaju
       da ne mozemo da identifikujemo token*/
    yyerror("Greska : Nepoznati karakter %c\n", c);
    return 0;
}
```

1.12 Promena tipa atributa

Iako je u poglavlju 1.10 kako se podrazumevani tip atributa može prilično jednostavno promeniti u odnosu na podrazumevani `int`, glavna snaga YACC-a se oseti tek kada se dozvoli mogućnost dodeljivanja različitih tipova različitim gramatičkim simbolima. Ovo se ostvaruje tako što se tip atributskog steka definiše kao unija nekoliko tipova. YACC obezbeđuje posebnu

`%union`

direktivu kojom se to može uraditi. Ova direktiva se navodi u delu `definicije` ulazne YACC datoteke i koristi se tako što se u vitičastim zagradama navedu članovi unije (u sintaksi jezika C).

```
%union  {
     telo unije ...
 }
```

Efekat ove deklaracije je umetanje

```
typedef union {
         telo unije
} YYSTYPE;
```

u datoteke `y.tab.c` i `y.tab.h` ukoliko se zahteva njeno generisanje.

Posao koji je dalje potrebno uraditi je povezivanje gramatičkih simbola sa pojedinim članovima unije, i preko njih im dodeliti odgovarajuće tipove. Ovo se postiže na različite načine, u zavisnosti od toga da li je u pitanju terminal ili neterminal. Za neterminale je uvedena posebna direktiva

`%type <ime_clana_unije> neterm1, neterm2, ...`

Ovim se postiže da se prilikom referisanja atributa preko pseudo-promenjivih `$i` uvek referiše na odgovarajući član unije. Slična konstrukcija, `<ime_clana_unije>` se koristi i za pridruživanja odgovarajućih članova unije tokenima, međutim ovom prilikom se ne koristi posebna direktiva već se navedena konstrukcija koristi uz `%token`, `%left`, `%right` i `%nonassoc` direktive.

```
%token <ime_clana_unije> token1, token2, ...
%left <ime_clana_unije> token1, token2, ...
%right <ime_clana_unije> token1, token2, ...
%nonassoc <ime_clana_unije> token1, token2, ...
```

Ukoliko se vrednost tokena postavlja preko `yylval` promenjive, potrebno je eksplicitno navesti kom članu unije se pristupa.

Prilikom korišćenja promenjivih `$i` unutar akcija, YACC automatski pristupa odgovarajućem polju unije na steku, i to na osnovu definicija tipova gramatičkih simbola. Ukoliko nismo dodelili eksplicitno tip nekom gramatičkom simbolu i pokušamo da koristimo pseudo-promenjivu koja se na njega odnosi, YACC prijavljuje grešku. Međutim, prilikom svake konkretne upotrebe promejivih `$i` moguće eksplicitno navesti na koje polje unije se referiše. To se radi navođenjem

```
$<ime_clana_unije>i
```

Ova tehnika se mora koristiti prilikom korišćenja levog konteksta.

1.13 Veza sa sistemom Lex

YACC se jako lako integriše sa sistemom *Lex* [?] koji služi za automatsko generisanje leksičkih analizatora tako da je veoma često slučaj da se funkcija `yylex()` automatski generiše korišćenjem *Lex*-a. Datoteka koju Lex generiše i koja sadrži implementaciju funkcije `int yylex()` se na većini sistema naziva `lex.yy.c`. Pošto se funkcija `yylex()` u ovom slučaju nalazi u odvojenoj datoteci od YACC rutina koje su, podsetimo se, smeštene u `y.tab.c`, potrebno je na početku Lex specifikacije uključiti zaglavlje `y.tab.h` koje, opet se podsetimo, YACC automatski generiše ako mu se navede parametar `-d`.

Primera radi, konstruišimo pomoću Lex-a leksički analizator za primer 1 naveden na strani 9.

Leksički analizator se specifikuje na sledeći način

```
/* Datoteka ispravnost_izraza.l */
{%
#include "y.tab.h"
%}
%%
[0-9]*   return NUM; [+*()]   return *yytext; "\n"     return 0;
.        printf("Greska : Nepoznati karakter %c\n", *yytext);
%%
```

YACC specifikaciju `ispravnost_izraza.y` je potrebno promeniti jedino tako što je potrebno ukloniti definiciju i deklaraciju funkcije `yylex()`.

Da bi se na osnovu ove dve datoteke dobio izvršni kod programa potrebno je generisati datoteke `y.tab.c` i `y.tab.h` korišćenjem `-d` parametra, zatim preko sistema Lex generisati datoteku `lex.yy.c` koja sadrži definiciju `yylex()`, i na kraju prevesti obe ove datoteke i linkovati ih zajedno. Dakle, da rezimiramo

```
yacc -d ispravnost_izraza.y
lex  ispravnost_izraza.l
gcc -oispravnost_izraza y.tab.c lex.yy.c
```

proizvodi izvršnu verziju programa.

U ovom slučaju je mnogo bolje koristiti alat `make`, jer time, na osnovu navedenih zavisnosti, izbegavamo bespotrebno prevođenje ažurnog koda. Za navedeni primer `Makefile` bi mogao da izgleda npr. ovako

```
PROJECT = ispravnost_izraza
CC   = gcc
YACC = yacc
LEX  = lex

$(PROJECT) : y.tab.o lex.yy.o
         $(CC) -o $(PROJECT) y.tab.o lex.yy.o

lex.yy.o : lex.yy.c y.tab.h
        $(CC) -c lex.yy.c

y.tab.o : y.tab.c
        $(CC) -c y.tab.c

lex.yy.c : $(PROJECT).l
        $(LEX) $(PROJECT).l

y.tab.c y.tab.h:
        $(YACC) -d $(PROJECT).y
```

`Makefile` je pisan prilično generički, tako da je za slične projekte potrebno izmeniti samo vrednost promenjive `PROJECT` i dodeliti joj ime aktuelnog projekta.

Osim što leksički analizator komunicira sa parserom tako što mu vraća kodove tokena, potrebno je ostvariti komunikaciju i na nivou postavljanja atributa tj. vrednosti tokena. U mnogim programima se može sresti sledeća tehnika:

Prilikom implementacije leksičkog analizatora vodi se računa samo o prepoznavanju tokena i vraćanju odgovarajućih kodova. Zatim se iz akcija u specifikaciji parsera referiše na promenjivu `yytext` koja je ugrađena promenjiva sistema Lex i koja predstavlja pokazivač na deo bafera koji sadrži tekuću leksemu. Da bi se uopšte moglo pristupati ovoj promenjivoj iz YACC akcija, u deo `deklaracije` se unese `extern char* yytext;` direktiva. Zatim se unutar akcija, direktno na osnovu lekseme sračunava vrednost određenog atributa. Ovom pristupu može da se nađe nekoliko mana.

- Potrebno je voditi računa o činjenici da se bafer leksičkog analizatora sa vremena na vreme osvežava, tako da deo bafera koji je sadržao određenu leksemu ne mora više da je sadrži, tako da pokušaj pamćenje lekseme jednostavnim pamćenjem pokazivača na deo bafera ne mora uvek da uspe. Zbog toga se umesto pamćenja pokazivača `yytext` zahteva kopiranje tekuće lekseme na neku drugu memorijsku poziciju (obično korišćenjem funkcije `strdup`) što nameće obavezu brisanja te memorije kada nam leksema više ne bude potrebna.

- Posmatrajmo sledeći primer

```
a : B {printf("%s",yytext); } C
```

```
| B D
;
```

Naivno bi se moglo pomisliti da će leksema koja odgovara tokenu `B` biti ispisana, zato što je to poslednji token koji je pročitan pre akcije. U većini slučajeva ovo pravilo je i ispoštovano. Međutim, osmotrimo detaljnije šta se u ovom konkretnom slučaju događa. Uvidom u generisani parser, vidi se sledeće stanje

```
state 1
     a : B . $$1 C  (2)
     a : B . D  (3)
     $$1 : .  (1)

     D  shift 3
     C  reduce 1
```

Parser nije u stanju da odluči da li da svede ε pravilo za pomoćni neterminal `$$1` sve dok ne sinhronizuje preduvidni karakter. U trenutku kada se to svođenje izvrši, parser je već pročitao token `C` tako da `yytext` u tom trenutku ukazuje na leksemu koja njemu odgovara i ona će biti odštampana, a ne leksema koja odgovara tokenu `B` kao što se u prvom trenutku mislilo.

Iz ovih razloga, autor ovog teksta preporučuje da se sve vrednosti tokena postavljaju direktno na atributski stek direktno iz leksičkog analizatora i to preko mehanizma promenjive `yylval` o kome je ranije bilo reči. Ukoliko se definiše unijski tip atributa, zaglavlje `y.tab.h` koje se koristi za komunikaciju između parsera i okruženja, u ovom slučaju leksičkog analizatora sadrži `extern` deklaraciju promenjive `yylval` zajedno sa deklaracijom unije. Ukoliko smo pak tip steka definisali prostim navođenjem `typdef` ili `#define` direktive za simbol `YYSTYPE`, unutar leksičkog analizatora moramo sami da napišemo `extern` deklaraciju za promenjivu `yylval`. Imajmo na umu da je simbol `YYSTYPE` lokalni za `y.tab.c` datoteku i da ga ne možemo koristiti unutar ostalih delova programa pa samim tim ni unutar Lex specifikacije već u pomenutoj `extern` deklaraciji moramo opet eksplicitno navesti tip atributskog steka.

Glava 2

Rešeni zadaci

2.1 Primer : Kalkulator sa polinomima

Primer 3 *Napraviti program koji sa ulaza čita niz aritmetičkih izraza nad polinomima i ispisuje njihove vrednosti. Polinomne konstante se zapisuju u obliku*

```
<cn, cn-1, ... , c0>
```

i predstavljaju polinom $c_n x^n + c_{n-1}x^{n-1} + \ldots + c_0$. *Dopuštena operacija je i pronalaženje izvoda polinoma*

Ulazni jezik dopušta i definisanje promenjivih polinomijalnog tipa i njihovu inicijalizaciju na vrednost datog izraza. Definisane promenjive mogu učestvovati u formiranju izraza.

Npr. za uneto

```
p=<1,1,1>;
p';
(<1,2,3>*3+<3,2,1>')*p;
```

program ispisuje

```
2.000000*x^1+1.000000
3.000000*x^4+15.000000*x^3+26.000000*x^2+23.000000*x^1+11.000000
```

Rešenje : Rešenje ovog zadatka je podeljeno na četiri datoteke.

- Datoteke `polinom.h` i `polinom.c` sadrže definiciju strukture `POLINOM` koja sadrži stepen i pokazivač na niz koeficijenata. U ovom datotekam je takođe definisan i niz funkcija za manipulaciju polinomima.
- Datoteka `poli_kalk.l` sadrži specifikaciju leksičkog analizatora za Lex.
- Datoteka `poli_kalk.y` sadrži specifikaciju parsera.

```
/* Datoteka : polinom.h
   U ovoj datoteci je definisana struktura POLINOM i
   deklarisano je nekoliko funkcija za manipulaciju polinomima
*/
```

```
/* Struktura POLINOM */
typedef struct
{  int stepen;
   double* koeficijenti;
} POLINOM;


/* Ispisivanje datog polinoma na standardni izlaz */
void     IspisiPolinom    (POLINOM* pln);

/* Uklanjanje datog polinoma iz memorije */
void     ObrisiPolinom    (POLINOM* pln);

/* Konstruisanje polinoma stepena 0 na osnovu
   datog koeficijenta */
POLINOM* NapraviPolinom   (double a);

/* Dodavanje koeficijenta polinomu na kraj
   tj. na mesto slobodnog clana, drugim recima,
   funkcija racuna i formira polinom x*pln+a */
POLINOM* DodajKoeficijent (POLINOM* pln,double a);

/* Pravljenje kopije datog polinoma */
POLINOM* KopirajPolinom   (POLINOM* polinom);

/* Izracunavanje izvoda polinoma */
POLINOM* NadjiIzvod       (POLINOM* pln);

/* Izracunavanje zbira dva polinoma */
POLINOM* Zbir             (POLINOM* p1,POLINOM* p2);

/* Izracunavanje razlike dva polinoma */
POLINOM* Razlika          (POLINOM* p1,POLINOM* p2);

/* Izracunavanje proizvoda dva polinoma */
POLINOM* Proizvod         (POLINOM* p1,POLINOM* p2);

/* Datoteka : polinom.c
   U ovoj datoteci su definisane funkcije za rad sa polinomima
   koje su deklarisane u zaglavlju polinom.h
*/

#include "polinom.h"

#define max(x,y) ((x)>(y)?(x):(y))

void IspisiPolinom(POLINOM* pln)
{   int i;
    for (i=pln->stepen; i>0; i--)
        printf("%f*x^%d+",pln->koeficijenti[i],i);
```

```
    printf("%f\n",pln->koeficijenti[0]);
}


void ObrisiPolinom(POLINOM* pln)
{  free(pln->koeficijenti);
   free(pln);
}

POLINOM* NapraviPolinom(double a)
{   POLINOM* pln=(POLINOM*)malloc(sizeof(pln));
    pln->stepen=0;
    pln->koeficijenti=(double*)malloc(sizeof(double));
    pln->koeficijenti[0]=a;
    return pln;
}

POLINOM* DodajKoeficijent(POLINOM* pln,double a)
{   int i;
    double* novikoeficijenti;
    pln->stepen++;
    novikoeficijenti=(double*)malloc((pln->stepen+1)*sizeof(double));
    for (i=0; i<pln->stepen; i++)
       novikoeficijenti[i+1]=pln->koeficijenti[i];
    novikoeficijenti[0]=a;
    free(pln->koeficijenti);
    pln->koeficijenti=novikoeficijenti;
    return pln;
}

POLINOM* KopirajPolinom(POLINOM* polinom)
{     int i;
      POLINOM* pln=(POLINOM*)malloc(sizeof(POLINOM));
      pln->stepen=polinom->stepen;
      pln->koeficijenti=
          (double*)malloc((pln->stepen+1)*sizeof(double));
      for (i=0; i<=pln->stepen; i++)
           pln->koeficijenti[i]=polinom->koeficijenti[i];
      return pln;
}

POLINOM* NadjiIzvod(POLINOM* polinom)
{   int i;
    POLINOM* izvod=(POLINOM*)malloc(sizeof(POLINOM));
    izvod->stepen=polinom->stepen-1;
    izvod->koeficijenti=
          (double*)malloc((izvod->stepen+1)*sizeof(double));
    for (i=1; i<=polinom->stepen; i++)
       izvod->koeficijenti[i-1]=polinom->koeficijenti[i]*i;
    return izvod;
```

```
}

POLINOM* Zbir(POLINOM* p1,POLINOM* p2)
{  int i;
   POLINOM* zbir=(POLINOM*)malloc(sizeof(POLINOM));
   zbir->stepen=max(p1->stepen,p2->stepen);
   zbir->koeficijenti=
         (double*)malloc((zbir->stepen+1)*sizeof(double));
   for (i=0; i<=zbir->stepen; i++)
        zbir->koeficijenti[i]=
              ((i<=p1->stepen)?p1->koeficijenti[i]:0)+
              ((i<=p2->stepen)?p2->koeficijenti[i]:0);
   return zbir;

}

POLINOM* Razlika(POLINOM* p1,POLINOM* p2)
{  int i;
   POLINOM* razlika=(POLINOM*)malloc(sizeof(POLINOM));
   razlika->stepen=max(p1->stepen,p2->stepen);
   razlika->koeficijenti=
            (double*)malloc((razlika->stepen+1)*sizeof(double));
   for (i=0; i<=razlika->stepen; i++)
        razlika->koeficijenti[i]=
                 ((i<=p1->stepen)?p1->koeficijenti[i]:0)-
                 ((i<=p2->stepen)?p2->koeficijenti[i]:0);
   return razlika;
}

POLINOM* Proizvod(POLINOM* p1,POLINOM* p2)
{  int i,j;
   POLINOM* proizvod=(POLINOM*)malloc(sizeof(POLINOM));
   proizvod->stepen=p1->stepen+p2->stepen;
   proizvod->koeficijenti=
            (double*)malloc((proizvod->stepen+1)*sizeof(double));

   for (i=0; i<=proizvod->stepen; i++)
       proizvod->koeficijenti[i]=0;

   for (i=0; i<=p1->stepen; i++)
       for (j=0; j<=p2->stepen; j++)
            proizvod->koeficijenti[i+j]+=
                     p1->koeficijenti[i]*p2->koeficijenti[j];

   return proizvod;
}

/* Datoteka : polinomi.l
   U ovoj datoteci je data specifikacija na osnovu koje
   Lex gradi lekicki analizator
```

```
*/
%{
#include "polinom.h"
#include "y.tab.h"
#include <math.h>
#include <string.h>
%}

CIFRA [0-9]
SLOVO [_a-zA-Z]
%%
{SLOVO}({SLOVO}|{CIFRA})* {
      yylval.string_v=strdup(yytext);
      return IME;
}

{CIFRA}+("."{CIFRA}+)?    {
      yylval.double_v=atof(yytext);
      return BROJ;
}

[-<>+*,()='\n]            {
      return *yytext;
}
[ \t]
%%

%{
/* Datoteka : polinomi.y
*/
#include <stdlib.h>
#include <string.h>
#include "polinom.h"

#define yyerror printf

/* Binarno pretrazivacko drvo koje sadrzi promenjive
   leksikografski sortirane. Vrednost svake promenjive
   je odredjena pokazivacem na polinom
*/
typedef struct _CvorPromenjive
{  /* ime promenjive */
   char* ime;
   /* vrednost promenive */
   POLINOM* polinom;
   /* potomci u drvetu */
   struct _CvorPromenjive *levi, *desni;
} CvorPromenjive;

/* Drvo promenjivih koje je u pocetku prazno */
```

```
CvorPromenjive *drvo=NULL;

/* Deklaracije funkcija za rad sa drvetom promenjivih */
void     DodajPromenjivu(CvorPromenjive **pdrvo,char* ime, POLINOM* pln);
POLINOM* PronadjiPromenjivu(CvorPromenjive* d,char *ime);
void     UkloniDrvo(CvorPromenjive* c);

%}

%union{
/* Vrednosti promenjivih i izraza su polinomi*/
POLINOM*  pPOLINOM_v;
/* Vrednost tokena ID je ime identifikatora */
char*     string_v;
/* Vrednsosti tokena BROJ je numericka vrednost
   odgovarajuce lekseme*/
double    double_v;
}

/* Definicija tokena uz postavljanje
   prioriteta i asocijativnosti */
%token <double_value> BROJ
%token <string_value> IME
%left '+','-'
%left '*'
%left "')"

/* Atributi neterminala nizkoef i izraz su pokazivaci
   na polinome koji predstavljaju njihove vrednosti */
%type <pPOLINOM_value> nizkoef,izraz
%%

/* Na ulazu se ocekuje niz dodela */
nizdodela  : nizdodela dodela
           |
           ;

dodela : IME '=' izraz ';'
             /* U drvo se dodaje promenjiva sa imenom tokena IME
                i upravo sintetizovanom vrednoscu izraza sa desne
                strane dodele */
             { DodajPromenjivu(&drvo,$1,$3); }
       | izraz '\n'
             /* Ispisujemo vrednost izraza i uklanjamo je */
             { IspisiPolinom($1);
               ObrisiPolinom($1);
             }
       | '\n' /* Preskacemo prazne redove */
       ;
```

```
izraz : izraz '+' izraz
             /* Konstruisemo zbir polinoma */
             { $$=Zbir($1,$3);

               /* Uklanjamo operande jer nam vise nisu potrebni */
               ObrisiPolinom($1);
               ObrisiPolinom($3);
             }

      | izraz '*' izraz
             /* Konstruisemo proizvod polinoma */
             { $$=Proizvod($1,$3);

               /* Uklanjamo operande jer nam vise nisu potrebni */
               ObrisiPolinom($1);
               ObrisiPolinom($3);
             }

      | izraz '-' izraz
             /* Konstruisemo razliku polinoma */
             { $$=Razlika($1,$3);

               /* Uklanjamo operande jer nam vise nisu potrebni */
               ObrisiPolinom($1);
               ObrisiPolinom($3);
             }

      | izraz "'"
             /* Pronalazi se izvod polinoma koji predstavlja vrednost
                izraza  koji prethodi simbolu ' */
             { $$=NadjiIzvod($1);

               /* Vrednost izraza nam vise nije potrebna */
               free($1);
             }

      | '(' izraz ')'
             /* Vrednost izraza je vrednost izraza u zagradama */
             { $$=$2; }

      | '<' nizkoef '>'
             /* Neterminal nizkoef u ovom trenuktu ukazuje na
                sintetizovani polinom koji odgovara datom nizu
                koeficijenata. Vrednost izraza je upravo ovaj
                polinom */
             { $$=$2; }

      | BROJ
             /* Kreira se polinom stepena 0 cija je vrednost
                jednaka vrednosti tokena BROJ postavljenoj tokom
```

```
               leksicke analize */
            { $$=NapraviPolinom($1);}

      | IME
            /* U drvetu promenjivih se trazi vrednost promenjive
               sa imenom tokena IME postavljenom tokom leksicke
               analize */
            { $$=PronadjiPromenjivu(drvo,$1);
              /* Ukoliko promenjiva ne postoji prijavljuje se greska */
              if($$==NULL)
              {  yyerror("Greska : promenjiva %s nije definisana! \n",$1);
                 exit(1);
              }

              /* Oslobadjamo string koji predstvlja
                 vrednost tokena BROJ */
              free($1);
           }
      ;

nizkoef :  BROJ
            /* Prilikom prvog koeficijenta kreira se
               polinom stepena 0 */
            {$$=NapraviPolinom($1);}
        | nizkoef ',' BROJ
            /* Zatim se dodaje jedan po jedan koeficijent */
            { $$=DodajKoeficijent($1,$3); }
        ;
%%

/* Funkcija dodaje promenjivu sa datim imenom u drvo promenjivih
   na ciji koren ukazuje pokazivac pdrvo. Vrednost promenjive se
   postavlja na dati polinom. Ukoliko promenjiva vec postoji u drvetu
   stara vrednost se uklanja */
void DodajPromenjivu(CvorPromenjive **pdrvo,char* ime, POLINOM* pln)
{   if ((*pdrvo)==NULL)
    {    (*pdrvo)=(CvorPromenjive*)malloc(sizeof(CvorPromenjive));
         (*pdrvo)->ime=strdup(ime);
         (*pdrvo)->polinom=pln;
         (*pdrvo)->levi=NULL;
         (*pdrvo)->desni=NULL;
    }
    else
    {   int cmp=strcmp((*pdrvo)->ime,ime);
        if (cmp==0)
        {   if ((*pdrvo)->polinom)
                  ObrisiPolinom((*pdrvo)->polinom);
            (*pdrvo)->polinom=pln;
        }
        else if (cmp<0)
```

```
                DodajPromenjivu(&((*pdrvo)->levi),ime,pln);
            else
                DodajPromenjivu(&((*pdrvo)->desni),ime,pln);
    }
}

/* Funkcija pokusava da pronadje polinom koji predstavlja
   vrednost promenjive sa datim imenom u datom drvetu promenjivih
   i vraca pokazivac na kopiju pronadjenog polinoma.
   Ukoliko promenjiva ne postoji u drvetu funkcija vraca NULL */
POLINOM* PronadjiPromenjivu(CvorPromenjive* d,char *ime)
{   if (d==NULL)
        return NULL;
    else
    {   int cmp=strcmp(d->ime,ime);
        if (cmp==0)
            return KopirajPolinom(d->polinom);
        else if (cmp<0)
                return PronadjiPromenjivu(d->levi,ime);
             else
                return PronadjiPromenjivu(d->desni,ime);
    }
}

/* Funkcija uklanja drvo promenjivih */
void UkloniDrvo(CvorPromenjive* c)
{   if (c!=NULL)
    {   UkloniDrvo(c->levi);
        UkloniDrvo(c->desni);
        ObrisiPolinom(c->polinom);
        free(c->ime);
        free(c);
    }
}

main()
{   /* Vrsimo parsiranje */
    yyparse();

    /* Po zavrsetku uklanjamo drvo promenjivih */
    UkloniDrvo(drvo);
}
```

Izvršna verzija programa se generiše na osnovu sledećeg `Makefile`-a

```
PROJECT = poli_kalk
CC   = gcc
YACC = yacc
LEX  = lex

$(PROJECT) : y.tab.o lex.yy.o polinom.o
```

```
    $(CC) -o $(PROJECT) y.tab.o lex.yy.o polinom.o

lex.yy.o : lex.yy.c y.tab.h polinom.h
    $(CC) -c lex.yy.c

y.tab.o : y.tab.c polinom.h
    $(CC) -c y.tab.c

polinom.o : polinom.c polinom.h
    $(CC) -c polinom.c

lex.yy.c : $(PROJECT).l
    $(LEX) $(PROJECT).l

y.tab.c y.tab.h: $(PROJECT).y
    $(YACC) -d $(PROJECT).y
```

2.2 Primer : "Čitljivo" prikazivanje aritmetičkih izraza

Primer 4 *Sa standardnog ulaza se unosi aritmetički izraz nad celim brojevima i jednoslovnim promenjivama. Napisati program koji na standardni izlaz ispisuje matricu karaktera u kojoj je dati izraz čitljivije prikazan, tj. sva deljenja su prikazana u obliku razlomaka. Npr. za uneto* `(a+b)/(c*d+f)-3*f+g/4` *program ispisuje*

```
 a+b          g
----- - 3*f + -
c*d+f         4
```

Zadatak rešavamo tako što rezultujuću matricu sintetizujemo na osnovu manjih matrica prateći sintaksnu strukturu izraza.

- Ukoliko je izraz elementaran tj. ako je u pitanju promenjiva ili konstanta, matrica karaktera je upravo tekst odgovarajuće lekseme.
- Ukoliko izraz predstavlja zbir, proizvod ili razliku dva operanda, rezultujuća matrica se dobija tako što se manje matrice operanada postave jedna do druge, odvojene karakterom operatora. Potrebno je još voditi računa o tome da, ukoliko operandi sadrže razlomačku crtu, ona bude u nivou tj. u istoj vrsti kao i karakter operatora. Ovo je razlog zbog čega uz svaku matricu karaktera čuvamo i podatak o položaju glavne razlomačke crte.
- Ukoliko izraz predstavlja količnik, rezultujuća matrica se dobija postavljanjem matrica koje odgovaraju operandima jedne iznad druge i umetanjem razlomačke crte, tj. niza karaktera - između njih. Potrebno je još voditi računa o tome, da ako imenilac ili brojilac već predstavlja razlomak, razlomačka crta mora biti duža, kako bi se već na prvi pogled uočilo koja razlomačka crta je "glavna".

Jedini problem koji je potrebno rešiti je ispisivanje zagrada. Kako bi zapis bio čitljiviji nećemo ispisivati zagrade kada to nije neophodno. To znači da je zagrade neophodno pisati samo u sledećim slučajevima:

- Ukoliko izraz predstavlja proizvod odnosno količnik (multiplikativna operacija), zagrade je potrebno staviti oko operanda kome je na vrhu sabiranje ili oduzimanje (aditivna operacija).
- Ukoliko izraz predstavlja razliku, zagrade je potrebno staviti oko umanjioca i to samo ako je na njegovom vrhu aditivna operacija.

Zbog ovoga, uz svaki izraz pamtimo informaciju o njegovoj vrsti, tj. o tome da li izraz predstavlja aditivnu, multiplikativnu operaciju ili je u pitanju elementaran izraz.

```
/* Datoteka : citljivi_izrazi.l */
%{
#include "y_tab.h"
%}

%%
[-+*/()]        return *yytext;
[0-9]+|[a-zA-Z] return LITERAL;
%%


/* Datoteka : citljivi_izrazi.y */
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Izraze delimo na elementarne (promenjive i konstante)
   aditivne (na "vrhu" je + ili -) i
   multiplikativne (na "vrhu" je * ili /) */
enum {ELEMENT, ADITIVNA, MULTIPLIKATIVNA};

/* Svakom izrazu odgovara matrica karaktera.
   Posto ne znamo kolike ove matrice mogu biti pravimo
   ih dinamickim */

typedef struct _mat
{  char **mat;
   int bv,bk;

   /* Vrsta matrice u kojoj se nalazi glavna
      razlomacka crta */
   int raz_crta;

   /* Zbog pravilnog postavljanja zagrada potrebno je
      znati i vrstu operacije na "vrhu" izraza */
```

```
   int vrsta_operacije;
} matrica;

/* Na atributski stek cemo stavljati pokazivace
   na matrice karaktera koje odgovaraju izrazima */
#define YYSTYPE matrica*

/* Funkcija za obradu greske */
#define yyerror printf

/* Lex promenjiva koja ukazuje na tekucu leksemu */
extern char* yytext;

/* Maksimum dva broja */
#define max(a,b) ((a>b)?(a):(b))
%}

%left '+','-'
%left '*','/'
%token LITERAL,'(',')'

%%

pocetak : izraz { ispisi($1); oslobodi($1);}

izraz : izraz '+' izraz
        { $$=matrica_pored_matrice($1,'+',$3);
          oslobodi($1);
          oslobodi($3); }
      | izraz '-' izraz
        { $$=matrica_pored_matrice($1,'-',$3);
          oslobodi($1);
          oslobodi($3); }
      | izraz '*' izraz
        { $$=matrica_pored_matrice($1,'*',$3);
          oslobodi($1);
          oslobodi($3); }
      | izraz '/' izraz
        { $$=matrica_iznad_matrice($1,$3);
          oslobodi($1);
          oslobodi($3); }
      | '(' izraz ')'
        { $$=$2; }
      | LITERAL
        { $$=napravi_matricu(yytext); }
      ;

%%

/* Funkcija alocira memoriju za matricu koja ima
```

```
   bv vrsta i bk kolona */
matrica* alociraj(int bv, int bk)
{   int i,j;
    matrica* pm=(matrica*) malloc(sizeof(matrica));
    pm->bv=bv;
    pm->bk=bk;
    pm->mat=(char**) malloc(bv*sizeof(char*));
    for (i=0; i<bv; i++)
    {  pm->mat[i]=(char*) malloc(bk*sizeof(char));
       for (j=0; j<bk; j++)
          pm->mat[i][j]=' ';
    }
    pm->raz_crta=0;
    pm->vrsta_operacije=ELEMENT;
    return pm;
}

/* Ispisuje se matrica karaktera */
void ispisi(matrica* pm)
{   int i,j;
    for (i=0; i<pm->bv; i++)
    {   for (j=0; j<pm->bk; j++)
           printf("%c",pm->mat[i][j]);
        printf("\n");
    }
}


/* Oslobadjamo strukturu matrice */
void oslobodi(matrica* pm)
{   int i,j;
    for (i=0; i<pm->bv; i++)
       free(pm->mat[i]);
    free(pm->mat);
    free(pm);
}

/* Pravimo "elementarnu" matricu na osnovu
   niza karaktera */
matrica* napravi_matricu(char* lit)
{   matrica *pm;
    pm=(matrica*) malloc(sizeof(matrica));
    if (pm==NULL)
    {   yyerror("Greska : neuspesno alociranje memorije ");
        exit(1);
    }
    pm->bv=1;
    pm->bk=strlen(lit);
    pm->mat=(char**) malloc(sizeof(char*));
    pm->mat[0]=strdup(lit);
```

```
    pm->raz_crta=0;
    pm->vrsta_operacije=ELEMENT;
    return pm;
}

/* Funkcija ubacuje blok matricu s u matricu d i to tako
   da gornji levi ugao bude na poziciji x,y matrice d */
void ubaci_mat_u_mat(matrica* d, matrica* s, int x, int y)
{   int i,j;
    for (i=0; i<s->bv; i++)
       for(j=0; j<s->bk; j++)
          d->mat[x+i][y+j]=s->mat[i][j];
}

/* Gradimo novu matricu tako sto postavljamo matricu l,
   zatim operator i onda matricu d jedno pored drugoga */
matrica* matrica_pored_matrice(matrica* l, char op, matrica* d)
{  int bv,bk;
   int bordura,zagrade_levo,zagrade_desno;
   int sirina_levog, sirina_desnog,
       pocetak_levog, kraj_levog,
       pocetak_desnog, kraj_desnog;
   matrica* pm;

   /* Ako matrice l i d sadrze razlomacku crtu ostavljamo
      jedno prazno mesto izmedju razlomaka i operatora */
   bordura=(l->raz_crta==0 &&
            d->raz_crta==0)?0:1;

   /* Da li ima potrebe stavljati matricu l u zagrade ? */
   zagrade_levo=((op=='*') && (l->vrsta_operacije==ADITIVNA))?1:0;

   /* Da li ima potrebe stavljati matricu d u zagrade ? */
   zagrade_desno=(((op=='*') && (d->vrsta_operacije==ADITIVNA)) ||
                  ((op=='-') && (d->vrsta_operacije==ADITIVNA)))?1:0;

   /* sirina_operanada ukljucujuci i zagrade*/
   sirina_levog  = zagrade_levo  + l->bk + zagrade_levo;
   sirina_desnog = zagrade_desno + d->bk + zagrade_desno;

   /* Pozicija na kojoj pocinji i zavrsavaju se operandi
      (ne racunajuci zagrade)*/
   pocetak_levog  = 0;
   kraj_levog     = zagrade_levo + l->bk;
   pocetak_desnog = sirina_levog+bordura+1+bordura;
   kraj_desnog    = pocetak_desnog+zagrade_desno+d->bk;

   /* Odredjujemo dimenzije nove matrice */
   bk=sirina_levog+
      bordura+1+bordura+
```

```
      sirina_desnog;

   bv=max(l->raz_crta,d->raz_crta)+
      1+
      max(l->bv-l->raz_crta-1,d->bv-d->raz_crta-1) ;

   /* Alociramo prostor */
   pm=alociraj(bv,bk);
   /* Odredjujemo polozaj glavne razlomacke crte */
   pm->raz_crta=max(l->raz_crta,d->raz_crta);

   /* Postavljamo polja strukture */
   pm->vrsta_operacije=(op=='*')?MULTIPLIKATIVNA:ADITIVNA;

   /* Operator postavljamo na odgovarajuce mesto */
   pm->mat[pm->raz_crta][sirina_levog+bordura]=op;
   /* Ukoliko je potrebno staviti zagrade postavljamo ih */
   if (zagrade_levo)
   {   pm->mat[pm->raz_crta][pocetak_levog]='(';
       pm->mat[pm->raz_crta][kraj_levog]=')';
   }

   if (zagrade_desno)
   {  pm->mat[pm->raz_crta][pocetak_desnog]='(';
      pm->mat[pm->raz_crta][kraj_desnog]=')';
   }
   /* Ugradjujemo vec postojece matrice l i d na odgovarajuce pozicije */
   ubaci_mat_u_mat(pm,l,pm->raz_crta-l->raz_crta,
                   pocetak_levog+zagrade_levo);
   ubaci_mat_u_mat(pm,d,pm->raz_crta-d->raz_crta,
                   pocetak_desnog+zagrade_desno);

   return pm;
}

/* Gradimo razlomak od dve matrice */
matrica* matrica_iznad_matrice(matrica* l, matrica* d)
{  int bv,bk,bordura;
   int i;
   matrica* pm;

   /* Ukoliko je ili brojilac ili imenilac razlomak,
      onda razlomacka crta mora biti duza */
   bordura=(l->raz_crta==0 &&
            d->raz_crta==0)?0:1;

   /* Odredjujemo broj vrsta i kolona nove matrice */
   bk=max(l->bk,d->bk)+2*bordura;
   bv=l->bv+1+d->bv;
```

```
   /* Alociramo memoriju */
   pm=alociraj(bv,bk);

   /* Ubacujemo brojilac */
   ubaci_mat_u_mat(pm,l,0,(bk-l->bk)/2);

   /* Gradimo razlomacku crtu */
   for (i=0; i<bk; i++)
     pm->mat[l->bv][i]='-';

   /* Ubacujemo imenilac */
   ubaci_mat_u_mat(pm,d,l->bv+1,(bk-d->bk)/2);

   /* Postavljamo ostala polja strukture */
   pm->raz_crta=l->bv;
   pm->vrsta_operacije=MULTIPLIKATIVNA;
   return pm;
}

main()
{  yyparse();
}
```

2.3 Primer : Pronalaženje izvoda funkcija

Primer 5 *Sa standardnog ulaza se unosi analitički izraz kojim je opisana neka elementarna funkcija, pri čemu je prilikom navodjenja funkcije dopušteno koristiti i operator ' koji služi za nalaženje izvoda funkcije. Za ovim operatorom može da sledi niz promenjivih po kojima se diferenciranje vrši. Ukoliko se ovaj niz ne navede, podrazumeva se da se diferencira po promenjivoj* `x`*. Napisati program koji za svaku ovako definisanu funkciju ispisuje njen donekle uprošćen oblik. Npr. Za uneto*

```
(sin(2*x))'
(x*y+y*x)'xy
```

Program ispisuje

```
cos(2*x)*2
2
```

Prilikom rešavanja ovog zadatka koristićemo tehniku eksplicitne izgradnje drveta apstraktne sintakse i definisanja niza funkcija koje transformisu takvo drvo.

Prvo što treba razrešiti je definisanje odgovarajuće memorijske strukture koja odgovara ovakvom jednom drvetu. Uočimo da će u tom drvetu biti potrebno čuvati čvorove sledeća četiri tipa

- *Čvorovi konstanti* - Odgovaraju listovima drveta u kojima se čuvaju brojevne konstante

- *Čvorovi promenjivih* - Odgovaraju listovima drveta u kojima se čuvaju promenjive
- *Čvorovi operacija* - Odgovaraju binarnim aritmetičkim operacijama. Ovakav čvor je okarakterisan identifikatorom vrste operacije kao i pokazivačima na dva podrveta koja predstavljaju argumente ove operacije
- *Čvorovi funkcija* - Odgovaraju analitčkim funkcijama jedne promenjive. Ovakav čvor je okarakterisan imenom funkcije kao i pokazivačem na podrvo koja predstavlja argument funkcije

Pošto drvo može da sadrži čvorove bilo koje od navedene četiri vrste, logično rešenje je predstaviti ga unijskim tipom podataka. Jedini problem koji je potrebno razrešiti je mogućnost identifikovanja vrste čvora. Za to iskoristimo sledeću tehniku. Na početku unije ostavimo prostor za posebno polje `TipCvora` koje će da služi za ovakvu identifikaciju. Medjutim, da bismo bili sigurni da se ovakav identifikator stvarno nalazi na početku memorijskog prostora unije, bez obzira o kojem se konkretnom čvoru radi, potrebno je sve četiri strukture koje odgovaraju različitim tipovima čvorova proširiti istim ovim identifikatorom i to na početku njihovog memorijskog prostora.

Funkcije za rad sa ovako definisanim drvetom su manje-više standardne i jedino je još potrebno naglasiti da je fukcija za uprošćavanje prilično jednostavno implementirana i da bi u slučaju realne primene ovakvog programa bilo potrebno još poraditi na ovoj funkciji.

Projekat je opet ogranizovan u četiri datoteke: `drvo_funkcije.h`, `drvo_funkcije.c`, `izvodi.l` i `izvodi.y`. `Makefile` pomoću koga se može dobiti izvršni program je veoma sličan onom iz primera 3 pa ga zbog toga ne navodimo.

```
/* Datoteka : drvo_funkcije.h
   Definicija strukture drveta koje sluzi za
   reprezentaciju analitickih funkcija i
   sadrzi deklaracije funkcija potrebnih za rad
   sa drvetom
*/

/* Postoje cetiri razlicita tipa cvorova u
   sintaksnom drvetu. Simboli ovde definisani
   sluze za identifikaciju pojedinacnog cvora. */
typedef enum {OPERACIJA, KONSTANTA,
              PROMENJIVA, FUNKCIJA} TipCvora;

/* Cvor koji odgovara konstantama */
typedef struct _CvorKonstante
{   /* Identifikator tipa. Za ove cvorove bi trebalo
       da ima vrednost KONSTANTA */
    TipCvora tip;

    /* Brojevna vrednost konstante */
    int vrednost;
} CvorKonstante;
```

```
/* Cvor koji odgovara promenjivama */
typedef struct _CvorPromenjive
{   /* Identifikator tipa. Za ove cvorove bi trebalo
       da ima vrednost PROMENJIVA */
    TipCvora tip;

    /* Ime promenjive */
    char ime;
} CvorPromenjive;

/* Cvor koji odgovara funkcijama */
typedef struct _CvorFunkcije
{    /* Identifikator tipa. Za ove cvorove bi trebalo
        da ima vrednost FUNKCIJA */
     TipCvora tip;

     /* Niska koja odgovara imenu funkcije */
     char* ime_funkcije;

     /* Pokazivac na drvo koje predstavlja argument
        funkcije */
     union _Cvor* argument;
} CvorFunkcije;


/* Cvor koji odgovara aritmetickim operacijama */
typedef struct _CvorOperacije
{    /* Identifikator tipa. Za ove cvorove bi trebalo
        da ima vrednost OPERACIJA */
     TipCvora tip;

     /* Karakter koji odgovara vrsti operacije
        '+' - sabiranje, '-' - oduzimanje,
        '*' - mnozenje,  '/' - deljenje
     */
     char vrsta_operacije;

     /* Pokazivaci na dva drveta koji predstavljaju
        argumente operacije */
     union _Cvor *argument_1,
                 *argument_2;
} CvorOperacije;

/* Unija koja objedinjuje sve pomenute tipove i
   predstavlja jedan cvor sintaksnog drveta */
typedef union _Cvor
{   /* Identifikator tipa. Na osnovu vrednosti
       ovog polja se odredjuje o kom tipu cvora
       se radi i kojem polju unije treba pristupati */
    TipCvora  tip;
```

```
   /* Cvor moze biti bilo koji od navedenih
      tipova cvorova */
    CvorOperacije   operacija;
    CvorKonstante   konstanta;
    CvorPromenjive  promenjiva;
    CvorFunkcije    funkcija;
} Cvor;

/* Deklaracije funkcija za rad sa ovakvim sintaksnim
   drvetom */


/* Funkcija pravi novi cvor konstante sa datom
   vrednoscu i vraca pokazivac na njega */
Cvor* NapraviCvorKonstante  (int);

/* Funkcija pravi novi cvor promenjive sa datim
   imenom i vraca pokazivac na njega */
Cvor* NapraviCvorPromenjive (char);

/* Funkcija pravi novi cvor operacije na osnovu
   date vrste operacije i dva drveta koja
   predstavljaju argumente i inkorporiraju se u
   drvo koje se gradi */
Cvor* NapraviCvorOperacije  (char, Cvor*, Cvor*);

/* Funkcija pravi novi cvor funkcije na osnovu
   datog imena funkcije i drveta koje predstavlja
   argument i inkorporiraj se u drvo koje se gradi */
Cvor* NapraviCvorFunkcije   (char*, Cvor*);

/* Funkcija ispisuje dato drvo */
void  Ispisi  (Cvor*);

/* Funkcija kopira celokupno drvo i vraca pokazivac
   na napravljenu kopiju */
Cvor* Kopiraj (Cvor*);

/* Funkcija uklanja celokupno drvo */
void  Obrisi  (Cvor*);

/* Funkcija na osnovu datog drveta gradi novo
   koje predstavlja izvod funkcije predstavljene
   prvim  drvetom */
Cvor* Izvod   (Cvor*,char);

/* Funkcija na osnovu datog drveta gradi novo
   koje predstavlja njegovo uproscenje */
Cvor* Uprosti (Cvor*);
```

```
/* Datoteka : drvo_funkcije.c
   Sadrzi definicije funkcija potrebnih za rad
   sa drvetom koje sluzi za reprezentaciju
   analitickih funkcija */

#include "drvo_funkcije.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

Cvor* NapraviCvorKonstante(int br)
{ Cvor* novi=malloc(sizeof(CvorKonstante));
  if (novi==NULL)
       printf("Neuspesno alocirana memorija\n");
   novi->tip=KONSTANTA;
   novi->konstanta.vrednost=br;
   return novi;
}

Cvor* NapraviCvorPromenjive(char ime)
{ Cvor* novi=malloc(sizeof(CvorPromenjive));
  if (novi==NULL)
       printf("Neuspesno alocirana memorija\n");
   novi->tip=PROMENJIVA;
   novi->promenjiva.ime=ime;
   return novi;
}

Cvor* NapraviCvorOperacije(char vo, Cvor* argument_1, Cvor* argument_2)
{
   Cvor* novi=malloc(sizeof(CvorOperacije));
   if (novi==NULL)
       printf("Neuspesno alocirana memorija\n");
   novi->tip=OPERACIJA;
   novi->operacija.vrsta_operacije=vo;
   novi->operacija.argument_1=argument_1;
   novi->operacija.argument_2=argument_2;
   return novi;
}

Cvor* NapraviCvorFunkcije(char* ime, Cvor* argument)
{  Cvor* novi=malloc(sizeof(CvorFunkcije));
   if (novi==NULL)
       printf("Neuspesno alocirana memorija\n");
   novi->tip=FUNKCIJA;
   novi->funkcija.ime_funkcije=ime;
   novi->funkcija.argument=argument;
   return novi;
}
```

```
void Ispisi(Cvor* c)
{  if (c!=NULL)
   {   switch(c->tip)
    {   case KONSTANTA:
                printf("%d",c->konstanta.vrednost);
                break;

            case PROMENJIVA:
                printf("%c",c->promenjiva.ime);
                break;

            case OPERACIJA:
                if (c->operacija.argument_1->tip==OPERACIJA)
                {    printf("(");
                     Ispisi(c->operacija.argument_1);
                     printf(")");
                }
                else /* Oslobadjanje spoljasnjih zagrada */
                     Ispisi(c->operacija.argument_1);

                printf("%c",c->operacija.vrsta_operacije);


                if (c->operacija.argument_2->tip==OPERACIJA)
                {    printf("(");
                     Ispisi(c->operacija.argument_2);
                     printf(")");
                }
                else /* Oslobadjanje spoljasnjih zagrada */
                     Ispisi(c->operacija.argument_2);

                break;

           case FUNKCIJA:
                printf("%s(",c->funkcija.ime_funkcije);
                Ispisi(c->funkcija.argument);
                printf(")");
    }
   }
}

void Obrisi(Cvor* c)
{    switch(c->tip)
     {   case KONSTANTA:
         case PROMENJIVA:
                free(c);
                break;
         case FUNKCIJA:
                free(c->funkcija.ime_funkcije);
                free(c);
```

```
                break;
        case OPERACIJA:
                Obrisi(c->operacija.argument_1);
                Obrisi(c->operacija.argument_2);
                free(c);
                break;
    }
}

Cvor* Kopiraj(Cvor* c)
{   switch(c->tip)
    {   case KONSTANTA:
        {   Cvor* novi=malloc(sizeof(CvorKonstante));
            if (novi==NULL)
                  printf("Neuspesno alocirana memorija\n");
            novi->tip=KONSTANTA;
            novi->konstanta=c->konstanta;
            return novi;
        }
        case PROMENJIVA:
        {   Cvor* novi=malloc(sizeof(CvorPromenjive));
            if (novi==NULL)
                  printf("Neuspesno alocirana memorija\n");
            novi->tip=PROMENJIVA;
            novi->promenjiva=c->promenjiva;
            return novi;
        }
        case OPERACIJA:
        {   Cvor* novi=malloc(sizeof(CvorOperacije));
            if (novi==NULL)
                  printf("Neuspesno alocirana memorija\n");
            novi->tip=OPERACIJA;
            novi->operacija.vrsta_operacije=c->operacija.vrsta_operacije;
            novi->operacija.argument_1=Kopiraj(c->operacija.argument_1);
            novi->operacija.argument_2=Kopiraj(c->operacija.argument_2);
            return novi;
        }

        case FUNKCIJA:
        {   Cvor* novi=malloc(sizeof(CvorFunkcije));
            if (novi==NULL)
                  printf("Neuspesno alocirana memorija\n");
            novi->tip=FUNKCIJA;
            novi->funkcija.ime_funkcije=
                  strdup(c->funkcija.ime_funkcije);
            novi->funkcija.argument=Kopiraj(c->funkcija.argument);
            return novi;
        }
    }
}
```

```
Cvor* Izvod(Cvor* c,char po_promenjivoj)
{    switch(c->tip)
     {   case KONSTANTA:
               /* Izvod konstante je 0 */
               return NapraviCvorKonstante(0);

         case PROMENJIVA:
              /* Izvod promenjive je 1 ako je u pitanju
                 promenjiva diferenciranja i 0 inace */
              return NapraviCvorKonstante(
                  (c->promenjiva.ime==po_promenjivoj)?1:0);

         case OPERACIJA:
             switch(c->operacija.vrsta_operacije)
             {  case '+':
                case '-':
                   /* Izvod zbira i razlike je jednak
                      zbiru odnosno razlici izvoda */
                   return NapraviCvorOperacije(
                     c->operacija.vrsta_operacije,
                     Izvod(c->operacija.argument_1,po_promenjivoj),
                     Izvod(c->operacija.argument_2,po_promenjivoj));

                case '*':
                   /* (f*g)'=f'*g+f*g' */
                   return NapraviCvorOperacije(
                     '+',
                     NapraviCvorOperacije(
                       '*', Izvod(c->operacija.argument_1,po_promenjivoj),
                            Kopiraj(c->operacija.argument_2)),
                     NapraviCvorOperacije(
                       '*', Kopiraj(c->operacija.argument_1),
                            Izvod(c->operacija.argument_2,po_promenjivoj)));

                case '/':
                   /* (f/g)'=(f'*g-f*g')/(g*g) */
                   return NapraviCvorOperacije(
                     '/',
                     NapraviCvorOperacije(
                       '-',
                       NapraviCvorOperacije(
                         '*',
                         Izvod(c->operacija.argument_1,po_promenjivoj),
                         Kopiraj(c->operacija.argument_2)),
                       NapraviCvorOperacije(
                         '*',
                         Kopiraj(c->operacija.argument_1),
                         Izvod(c->operacija.argument_2,po_promenjivoj))),
```

```
                    NapraviCvorOperacije(
                      '*',
                      Kopiraj(c->operacija.argument_2),
                      Kopiraj(c->operacija.argument_2)));
            }
            case FUNKCIJA:
            {   /* (sin(a)'=cos(a)*a' */
                if (strcmp(c->funkcija.ime_funkcije,"sin")==0)
                {    char* cos=strdup("cos");
                     return  NapraviCvorOperacije(
                       '*',
                       NapraviCvorFunkcije(
                          cos,
                          Kopiraj(c->funkcija.argument)),
                       Izvod(c->funkcija.argument,po_promenjivoj));
                }
                /* exp(a)'=exp(a)*a' */
                else if (strcmp(c->funkcija.ime_funkcije,"exp")==0)
                {    return  NapraviCvorOperacije(
                        '*',
                        Kopiraj(c),
                        Izvod(c->funkcija.argument,po_promenjivoj));
                }
                /* ostale funkcije nisu implementirane */
                else
                {  printf("Ne umem da nadjem izvod funkcije %s\n",
                           c->funkcija.ime_funkcije);
                   /* Umesto izvoda vracamo original */
                   return Kopiraj(c);
                }
            }
     }
}

Cvor* Uprosti(Cvor* c)
{   switch(c->tip)
    {   case OPERACIJA:
        {   /* Uproscavaju se oba argumenta i gradi se novi
               cvor operacije sa uproscenim argumentima */
            Cvor* u=NapraviCvorOperacije(
                        c->operacija.vrsta_operacije,
                        Uprosti(c->operacija.argument_1),
                        Uprosti(c->operacija.argument_2));

             /* Ukoliko su oba argumenta konstante, operacija se
                novom konstantom koja predstavlja sracunatu vrednost */
             if (u->operacija.argument_1->tip==KONSTANTA &&
                 u->operacija.argument_2->tip==KONSTANTA)
             {   switch(u->operacija.vrsta_operacije)
                 {   case '+' :
```

```
            {   /* const+const -> const */
                int rez=
                   u->operacija.argument_1->konstanta.vrednost+
                   u->operacija.argument_2->konstanta.vrednost;
                Obrisi(u);
                return NapraviCvorKonstante(rez);
            }
            case '-' :
            {   /* const-const -> const */
                int rez=
                   u->operacija.argument_1->konstanta.vrednost-
                   u->operacija.argument_2->konstanta.vrednost;
                Obrisi(u);
                return NapraviCvorKonstante(rez);
            }
            case '*' :
            {   /* const*const -> const */
                int rez=
                   u->operacija.argument_1->konstanta.vrednost*
                   u->operacija.argument_2->konstanta.vrednost;
                Obrisi(u);
                return NapraviCvorKonstante(rez);
            }
        }
    }

    if (u->operacija.argument_1->tip==KONSTANTA)
    {     if (u->operacija.argument_1->konstanta.vrednost==0)
          {      switch (u->operacija.vrsta_operacije)
                 {   /* 0+const -> const */
                     case '+' :
                     {   Cvor* rez=u->operacija.argument_2;
                         Obrisi(u->operacija.argument_1);
                         free(u);
                         return rez;
                     }
                     /* 0*const -> 0 */
                     case '*':
                     {   Obrisi(u);
                         return NapraviCvorKonstante(0);
                     }

                 }
          }
          if (u->operacija.argument_1->konstanta.vrednost==1)
          {      switch (u->operacija.vrsta_operacije)
                 {   /* const*1 -> const */
                     case '*' :
                     {   Cvor* rez=u->operacija.argument_2;
                         Obrisi(u->operacija.argument_1);
```

```
                                free(u);
                                return rez;
                            }
                        }
                    }
                }

                if (u->operacija.argument_2->tip==KONSTANTA)
                {     if (u->operacija.argument_2->konstanta.vrednost==0)
                      {       switch (u->operacija.vrsta_operacije)
                              {   /* const+0 -> const */
                                  case '+' :
                                  {   Cvor* rez=u->operacija.argument_1;
                                      Obrisi(u->operacija.argument_2);
                                      free(u);
                                      return rez;
                                  }
                                  /* const*0 -> 0 */
                                  case '*':
                                  {   Obrisi(u);
                                      return NapraviCvorKonstante(0);
                                  }
                              }
                      }

                      if (u->operacija.argument_2->konstanta.vrednost==1)
                      {       switch (u->operacija.vrsta_operacije)
                              {   /* const*1 -> const */
                                  case '*' :
                                  {   Cvor* rez=u->operacija.argument_1;
                                      Obrisi(u->operacija.argument_2);
                                      free(u);
                                      return rez;
                                  }
                              }
                      }

                }
                return u;

            }
            case FUNKCIJA:
            {   /* Rezultat je nova funkcija sa uproscenim
                   argumentom */
                return NapraviCvorFunkcije(
                        strdup(c->funkcija.ime_funkcije),
                        Uprosti(c->funkcija.argument);

            }
        }
```

```
    /* U ostalim slucajevima nema uproscavanja vec se samo
       kopira original */
    return Kopiraj(c);
}

%option noyywrap
%{
#include "drvo_funkcije.h"
#include "y.tab.h"
#include <string.h>
%}

%%
[0-9]+  { /* Brojevne konstante */
          /* Vrednost broja postavljamo na
             atributski stek */
          yylval.int_vrednost=atoi(yytext);
          return BROJ; }

[a-z]   { /* Jednokarakterske promenjive */
          /* Karakter postavljamo na
             atributski stek */
          yylval.char_vrednost=(*yytext);
          return PROM; }


"sin"|"cos"|"exp"|"ln" { /* Dopustene funkcije */
          /* Kopiju imena postavljamo na
             atributski stek */
          yylval.string_vrednost=strdup(yytext);
          return FJA;}

[-+*/()=\n'] { /* Literali */
          return *yytext; }

[ \t]   /* Preskacemo beline */
%%

%{
/* Funkcija za obradu gresaka */
#define yyerror printf

/* Definicije strukture koja predstavlja
   sintaksno drvo funkcija */
#include "drvo_funkcije.h"

/* Pomocne funkcije */
#include <stdlib.h>
#include <string.h>
%}
```

```
/* Na steku \' cemo \v cuvati
  - cele brojeve
  - karaktere
  - stringove
  - pokazivace na cvorove sintaksnog drveta
*/
%union
{   int   int_vrednost;
    char  char_vrednost;
    char* string_vrednost;
    Cvor* pCvor_vrednost;
};

/* Vrednosti na steku se koriste za komunikaciju
   sa leksickim analizatorom */
%token <int_vrednost>     BROJ
%token <char_vrednost>    PROM
%token <string_vrednost>  FJA

/* Neterminalu funkcija odgovara pokazivac na sintaksno
   drvo koje odgovara niski koja se iz njega izvodi */
%type <pCvor_vrednost> funkcija

/* Nizu promenjivih po kojima se data funkcija
   diferencira dodeljujemo nisku karaktera */
%type <string_vrednost> promenjive_diferenciranja


/* Definicija prioriteta i asocijativnosti operatora */
%left '+' '-'
%left '*' '/'
%left "'"

/* Pocetni simbol gramatike */
%start nizfunkcija
%%
/* Sa ulaza se unosi niz funkcija -
   svaka u posebnom redu */
nizfunkcija : nizfunkcija funkcija '\n'
                { /* Uproscavamo sintaksno drvo koje
                     je sintetizovano i koje odgovara
                     neterminalu funkcija */
                  Cvor* uproscena_fja=Uprosti($2);

                  /* Ispisuje se uproscena funkcija */
                  Ispisi(uproscena_fja);
                  putchar('\n');

                  /* Uklanja se originalno sintaksno drvo */
```

```
                    Obrisi($2);

                    /* Uklanja se uproscena funkcija */
                    Obrisi(uproscena_fja);
                  }
              |
              ;

/* Prilikom prihvatanja se sintetizuje sintaksno drvo */
funkcija   :    funkcija '+' funkcija
                  { $$=NapraviCvorOperacije('+',$1,$3); }
           |    funkcija '*' funkcija
                  { $$=NapraviCvorOperacije('*',$1,$3); }
           |    funkcija '-' funkcija
                  { $$=NapraviCvorOperacije('-',$1,$3); }
           |    funkcija '/' funkcija
                  { $$=NapraviCvorOperacije('/',$1,$3); }
           |    '(' funkcija ')'
                  { $$=$2; }
           |    FJA '(' funkcija ')'
                  { $$=NapraviCvorFunkcije($1,$3); }
           |    BROJ
                  { $$=NapraviCvorKonstante($1);  }
           |    PROM
                  { $$=NapraviCvorPromenjive($1); }
          /* Izvod funkcije */
           |    funkcija "'" promenjive_diferenciranja
                  {   /* Ukoliko nije navedena promenjiva
                         diferenciranja, diferencira se po
                         promenjivoj x '/
                      if ($3==NULL)
                      {    /* Pronalazimo izvod po promenjivoj x */
                           $$=Izvod($1,'x');
                           /* Uklanjamo originalno drvo */
                           Obrisi($1);
                      }
                      else
                      {    /* Diferenciramo, redom po navedenim
                              promenjivama */
                           int i;
                           for (i=0; i<strlen($3); i++)
                           {  /* Pronalazimo izvod po
                                 i-toj promenjivoj  */
                              Cvor* izvod=Izvod($1,$3[i]);
                              /* Uklanjamo originalno drvo */
                              Obrisi($1);

                              /* Izvod dodeljujemo neterminalu
                                 funkcija kako bismo ga dalje
                                 diferencirali */
```

```
                            $1=izvod;
                          }
                          /* Rezultat je pridruzen neterminalu
                             funkcija */
                          $$=$1;
                       }
                    }
       ;

promenjive_diferenciranja  : promenjive_diferenciranja PROM
                      {  /* Dodajemo karakter koji odgovara
                            tokenu PROM na kraj niske koja
                            odgovara promenjivim diferenciranja
                         */
                         /* Duzina prethodne nikse */
                         int old_len=($1==NULL)?0:strlen($1);
                         /* Vrsimo realokaciju memorije */
                         $$=realloc($1,old_len+1);
                         /* Postavljamo karakter */
                         $$[old_len]=$2;
                      }
                      |  /* eps */
                        {  /* Nisku inicijalizujemo na praznu */
                           $$=NULL;
                        }
                      ;

%%

int main()
{   /* Pokrecemo parsiranje */
    return yyparse();
}
```

2.4 Primer : Kompilator za minijaturni programski jezik

Primer 6 *Neka je dat asemblerski jezik, za fiktivni nula-adresni računar. Asembler podržava sledeće instrukcije*

`push` *x*	*postavlja x na stek*
`pop` *x*	*skida vrednost sa vrha steka i postavlja je u promenjivu x ako je navedena*
`add`	*skida dve vrednosti sa vrha steka i njihov zbir postavlja na stek*
`sub`	*skida dve vrednosti sa vrha steka i njihovu razliku postavlja na stek*
`mul`	*skida dve vrednosti sa vrha steka i njihov proizvod postavlja na stek*
`div`	*skida dve vrednosti sa vrha steka i njihov količnik postavlja na stek*
`jmp` *labela*	*bezuslovni skok na labelu labela*
`jz` *labela*	*skok na navedenu labelu ukoliko je na vrhu steka 0*
`jnz` *labela*	*skok na navedenu labelu ukoliko na vrhu steka nije 0*
`cmp`	*skida dve vrednosti sa vrha steka i postavlja 1 ako su jednake i 0 ako nisu*
`cmpLT`	*skida dve vrednosti sa vrha steka i postavlja 1 ako je prva manja od druge i 0 ako nije*
`cmpGT`	*skida dve vrednosti sa vrha steka i postavlja 1 ako je prva veća od druge i 0 ako nije*
`print`	*štampa vrednost sa vrha steka*
`stop`	*prekida izvršavanje programa*

Dopuštene labele su oblika `Lxxx :` *gde su* `x` *cifre.*

Neka je, dalje, dat mali programski jezik koji podržava samo celobrojni tip podataka i na njemu definisane operatore `+`, `-`, `*`, `/`, *prefiksni operator inkrementiranja* (`++`) *kao i relacijske operatore* `<`, `>`, `<=`, `>=`, `==`. *Svi operatori imaju isto značenje kao i u jeziku C. Od kontrolnih struktura jezik podržava uslovno grananje (if-then i if-then-else) tipa kao i while petlju pri čemu je sintaksa ista kao u jeziku C. Definisana je još i ugradjena funkcija* `print` *sa jednim celobrojnim argumentom koja služi sa štampanje.*

Napraviti program koji vrši prevodjenje sa datog programskog jezika na dati asembler. Npr.

```
x=0;
while (x<3)
{   x=x+1;
    print(x);
}
```

se prevodi u

```
    push 0
    pop x
L000:
    push x
    push 3
```

```
            compLT
            jz L001
            push x
            push 1
            add
            pop x
            push x
            print
            jmp L000
L001:
            stop
```

Iako je eksplicitna izgradnja sintaksnog drveta za dati jezik rešenje koje se obično primenjuje, zadatak je rešen koristeći elementarnije tehnike. Ulazni jezik je opisan kontekst slobodnom gramatikom i na odgovarajuća mesta su umetnute akcije koje generišu odgovarajući izlaz.

Označimo sa `asm<X>` asemblerski kod koji odgovara niski koja se izvodi iz neterminala `X`.

Atimetički izrazi oblika `expr1 OP expr2` se praktično prevode u "postfiksnu" notaciju

```
asm<expr1>
asm<expr2>
asm<OP>
```

pri čemu se promenjiva `x` prevodi u `push x`, i slično se brojevna konstanta `c` prevodi u `push c`.

Operator inkrementiranja se realizuje dodavanjem konstante 1.

Prilikom obrade kontrolnih struktura se koriste labele što uvodi potrebu za globalnom promenjivom `int free_label` koja će da sadrži broj koji odgovara sledećoj slobodnoj labeli.

Konstrukt `while (expr) stmt` se prevodi u

```
Lxxx :
      asm<expr>
      jz Lxxx+1
      asm<stmt>
      jmp Lxxx
Lxxx+1:
```

Ovo znači da se prilikom obrade svake while naredbe uvode dve nove labele. Prilikom generisanja `jmp` instukcije i sledeće labele, potrebno je poznavati broj `xxx` koji je bio odredjen na početku obrade pronadjene `while` naredbe. Pokušaj korišćenja promenjive `free_label` ili uvodjenja nove globalne promenjive u kojoj bismo upamtili vrednost `xxx` pada u vodu, jer je moguće "ugnježdjavanje" kontrolnih struktura. Jasno je, dakle, da je svakoj pojavi `while` konstrukta potrebno pridružiti odgovarajući broj `xxx`. U rešenju koje se ovde prikazuje, to je uradjeno korišćenjem atributskog steka. Naime, pomenutu vrednost labele `xxx` na steku pridružujemo tokenu `WHILE` i kasnije se na nju referišemo.

Slična je situacija i sa `if` konstruktima, medjutim, ovde se još javlja i problem višeznačnosti, pomenut u poglavlju 1.8. Konstrukt `if (expr) stmt` se prevodi u

```
                asm<expr>
                jz Lxxx
                asm<stmt>
          Lxxx:
```

dok se konstrukt `if (expr) stmt1 else stmt2` prevodi u

```
                asm<expr>
                jz Lxxx
                asm<stmt>
                jmp Lyyy
          Lxxx:
                asm<stmt>
          Lyyy:
```

Prikazano rešenje objedinjuje zajednički deo oba konstrukta u pravilo

```
stmt :  IF (expr) stmt opt_else
```

prilikom čije redukcije se generiše zajednički deo prevoda

```
                asm<expr>
                jz Lxxx
                asm<stmt>
```

Pri tom se vrednost broja `xxx` pridružuje tokenu `IF` na atributskom steku. Ostatak se razrešava koristeći pravilo

```
opt_else : /* eps */
         | ELSE stmt
         ;
```

Prilikom redukcija ovih pravila, generiše se ostatak prevoda. U oba slučaja je potrebno pristupiti celobrojnoj vrednosti `xxx` pridružene tokenu `IF` i to se radi korišćenjem levog konteksta što je opisano u poglavlju 1.10.

Leksički analizator je konstruisan prilično jednostavno, korišćenjem sistema *Lex*.

```
/* Datoteka mini_kompilator.l
   Opis leksickog analizatora koji je deo kompilatora za
   minijaturni programski jezik */
%option noyywrap
%{
#include "y.tab.h"
#include <string.h>
%}
%%
"while"                 return WHILE;
"if"                    return IF;
"else"                  return ELSE;
"print"                 return PRINT;
"++"                    return PP;
```

```
"--"                    return MM;
"=="                    return EQ;
"<="                    return LEQ;
">="                    return GEQ;
[-+*/().{}<>=;]         return *yytext;
[_a-zA-Z][_a-zA-Z0-9]*  {yylval.str_value=strdup(yytext); return ID;}
[1-9][0-9]*|"0"         {yylval.str_value=strdup(yytext); return NUM;}
[ \t\n]                 /* Beline zanemarujemo */
%%

/* Datoteka : mini_kompilator.y */
%{
/* Funkcija za obradu greske */
#define yyerror printf

/* Broj prve neiskoriscene labele */
int free_label=0;

%}

/* Specijalni token cija je jedina svrha da
   ima prioritet manji od tokena else i da
   razresi IF-THEN-ELSE viseznacnost koja nastaje
   u gramatici */
%nonassoc IF_THEN
%nonassoc ELSE

/* Prioriteti i asocijativnost operatora */
%right '='
%left '<' '>' EQ LEQ GEQ
%left '+' '-'
%left '*' '/'
%right PP MM

/* Tokenima dodeljujemo broj koji odgovara labeli
   koja je dodeljena odgovarajucoj naredbi */
%token <int_value> WHILE IF ELSE
%token PRINT

/* Tokenima dodeljujemo tekst odgovarajuce lekseme */
%token <str_value> NUM ID

/* Tip steka definisemo tako da na njega mozemo
   stavljati cele brojeve i stringove */
%union
{ int int_value;
  char* str_value;
}

%%
```

```
/* Instrukcija stop je poslednja u nizu */
program : stmts {printf("\tstop\n");}
        ;

stmts   : stmts stmt
        |
        ;

stmt :  ';' /* prazna naredba */
     |  expr ';' /* izraz */
     |  '{' stmts '}' /* blok naredbi */
     |   /* Naredba dodele se prevodi u :
            asm <expr>
            pop ID
        */

       ID '=' expr ';'  /* asm <expr> */
            { printf("\tpop %s\n",$1);
              free($1);
            }

     |  /* Funkcija printf se prevodi u
            asm <expr>
            print
        */
        PRINT '(' expr ')' ';'  /* asm<expr> */
            { printf("\tprint\n");  }

     |  /*    while (expr) stmt
                 se prevodi u
              Lxxx :
                     asm<expr>
                     jz Lxxx+1
                     asm<stmt>
                     jmp Lxxx
              Lxxx+1:
        */
         WHILE
            {   /* Tokenu WHILE dodeljujemo
                prvu slobodnu labelu (Lxxx) */
                $1=free_label;

                /* Ispisujemo labelu Lxxx: */
                printf("L%03d:\n",$1);

                /* Za obradu while konstrukta ce
                biti koriscene dve labele i odmah
                zbog toga rezervisemo i sledecu */
                free_label+=2;
```

```
            }
        '(' expr ')'  /* asm<expr> */
            {  /* Ispisujemo : jz Lxxx+1 */
               printf("\tjz L%03d\n",$1+1);
            }
        stmt /* asm<stmt> */
            { /* Ispisujemo jmp Lxxx  */
              printf("\tjmp L%03d\n",$1);
              /* Ispisujemo Lxxx+1 */
              printf("L%03d:\n",$1+1);
            }

    |  /* Zajednicki deo if-then i if-then-else grananja
          Viseznacnost se resava u pravilima za opt_else.
          U oba slucaja potrebno je ispisati
              asm<expr>
              jnz Lxxx
              asm<stmt>
          Tokenu IF se pridruzuje vrednsot xxx zbog mogucnosti
          kasnijeg referisanja
       */

       IF '(' expr ')'  /* asm<expr> */
            { /* Rezervisemo slobodnu labelu za Lxxx */
              $1=free_label++;
              printf("\tjnz L%03d\n",$1);
            }
       stmt /* asm<stmt> */
       opt_else /* Opciono else */

    ;

opt_else :
          /* Epsilon pravilo govori o tome da se radi
             o IF-THEN grananju sto znaci da se zapoceti
             asemblerski kod zavrsava definisanjem labele
                 Lxxx */
          {  /* Broj xxx je pridruzen tokenu IF i pristupamo
                mu koriscenjem levog konteksta */
             printf("L%03d:\n",$<int_value>-5);
          }
          /* Dajemo ovom pravilu manji prioritet od sledeceg */
          %prec IF_THEN
        | /* Grananje IF-THEN-ELSE TIPA. Zapoceti asemblerski
             kod dopunjavamo sa
                  jmp Lyyy
              Lxxx:
                  asm<stmt>
              Lyyy:
```

```
           */
           ELSE
              { /* Tokenu ELSE dodeljujemo vrednost broja Lyyy */
                $1=free_label++;
                printf("\tjmp L%03d\n",$1);
                /* Vrednost xxx se nalazi na steku i pridruzena je
                tokenu IF. Pristupamo joj koristeci levi kontekst */
                printf("L%03d:\n",$<int_value>-6);
              }
           stmt /* asm<stmt> */
              {  /* Vrednost yyy je pridruzena tokenu ELSE na steku */
                 printf("L%03d:\n",$1);
              }
         ;


/*      epxr op expr
          se prevodi u
        asm<expr1>
        asm<expr2>
        op
*/

expr    : expr '+' expr
             {  printf("\tadd\n");      }
        | expr '-' expr
             {  printf("\tsub\n");      }
        | expr '*' expr
             {  printf("\tmul\n");      }
        | expr '/' expr
             {  printf("\tdiv\n");      }
        | expr '<' expr
             {  printf("\tcompLT\n");   }
        | expr '>' expr
             {  printf("\tcompGT\n");   }
        | expr EQ  expr
             {  printf("\tcompEQ\n");   }
        | expr LEQ  expr
             {  printf("\tcompLEQ\n");  }
        | expr GEQ  expr
             {  printf("\tcompGEQ\n");  }
        | '(' expr ')'

        | ID PP
             {  printf("\tpush %s\n",$1);
                printf("\tpush %s\n",$1);
                printf("\tpush 1\n");
                printf("\tadd\n");
                printf("\tpop %s\n",$1);
                printf("\tpop\n");
```

```
                free($1);
            }
        | PP ID
            {  printf("\tpush %s\n",$2);
               printf("\tpush 1\n");
               printf("\tadd\n");
               printf("\tpop %s\n",$2);
               free($2);
            }


/* Tokenima ID i NUM je tokom leksicke analize dodeljen
   tekst odgovarajuce lekseme.
*/
        | ID
            {  printf("\tpush %s\n",$1);
               free($1);
            }
        | NUM
            {  printf("\tpush %s\n",$1);
               free($1);
            }
        ;



%%

main()
{
    return yyparse();
}
```

Bibliografija

[1] A. Aho, R. Sethi, J. Ulman, *Compilers : Princliples, Techniques and Tools.*

[2] A. Aho, J. Ulman, *The Theory of Parsing, Translation and Compiling.*

[3] Stephen C. Johnson, *Yacc: Yet Another Compiler-Compiler*

Sadržaj