Primena metode inverzne poljske notacije i interpolacije u simboličkim izračunavanjima

Doktorska disertacija

Sadržaj

Predgovor

Ova doktorska disertacija sadrži opis originalne metode za simbolička izračunavanja, zasnovane na inverznoj poljskoj notaciji. Opisana metoda je primenjena na niz problema iz različitih oblasti.

Disertacija je podeljena na pet glava. Svaka glava podeljena je na nekoliko odeljaka, dok su odeljci podeljeni na pododeljke. Numeracija odeljaka i pododeljaka izvršena je u okviru dotične glave, odnosno odeljka, i sastoji se od broja glave i odeljka, a kod pododeljaka i broja pododeljka. Numeracija formula, slika, tabela, lema, teorema i sličnih objekata vršena je nezavisno u okviru glave, tj. sadrži broj glave i redni broj objekta odgovarajuće vrste.

U prvoj glavi nalaze se uvodne napomene i naznake problema kojima ćemo se u nastavku baviti. Druga glava sadrži definicije i teoreme koje odslikavaju osobine inverzne poljske notacije a na kojima se ovaj metod zasniva. Tu je opisana i neposredna primena metode na eliminaciju nepotrebnih zagrada prilikom prevodjenja postfiksnog zapisa izraza u infiksni.

Treća glava sadrži opis primena naše metode na simboličko diferenciranje, algebarska izračunavanja i konstrukciju algoritma za triangulaciju konveksnih poligona.

U četvrtoj glavi se opisuje na koji način se naš metod može kombinovati sa numeričkom interpolacijom sa ciljem da se dobije algoritam za izračunavanje generalisanih inverza monomijalnih matrica.

Peta glava predstavlja zaključak.

Spisak korišćene literature nalazi se na kraju teksta sortiran prema prezimenu autora. U okviru teksta je referisanje literature vršeno rednim brojem sa kojim se dotična jedinica pojavljuje u tom spisku navedenim u uglastim zagradama.

Najzad, želim da se zahvalim mentoru, dr Predragu Stanimiroviću, redovnom profesoru Prirodno-matematičkog fakulteta u Nišu, na višegodišnjoj saradnji. Jedan od plodova te

saradnje jeste i ova disertacija. Takodje se zahvaljujem članovima komisije, dr Dušanu Tošiću, vanrednom profesoru Matematičkog fakulteta u Beogradu, i dr Miroslavu Ćiriću, redovnom profesoru Prirodno-matematičkog fakulteta u Nišu, na korisnim primedbama i sugestijama koje su značajno doprinele kvalitetu ovog teksta.

U Nišu, juna 2003. godine

mr Predrag Krtolica

Glava 1

Uvod

U ovoj glavi ćemo prikazati osnove koncepta inverzne poljske notacije, kao i neke napomene o oblastima primene našeg metoda zasnovanog na korišćenju ovog načina zapisivanja izraza različitog tipa.

1.1 Inverzna poljska notacija

Već godinama se inverzna poljska notacija može naći u mnogim udžbenicima iz računarstva (npr. [8], [30], [35], [36]). Prema ovoj notaciji se aritmetički izrazi zapisuju u *postfiksnom* obliku umesto u uobičajenom *infiksnom* obliku.

Na primer, infiksni izraz

$$a + b$$

se u inverznoj poljskoj notaciji zapisuje kao

$$ab + .$$

Takodje su poznati algoritmi za transformaciju postfiksnih izraza u infiksne i obrnuto (vidi npr. [8], [30], [35], [36]). Obično ovi algoritmi koriste magacin i mogu se iskazati na sledeći način:

Algoritam 1.1 (transformacija infiksnog izraza u postfiksni)

(1I) Polazeći sleva, ispitaj tekući element izraza.

(2I) Ako je tekući element operand, pošalji ga na izlaz i idi na korak (6I).

(3I) Ako je tekući element otvorena zagrada, upiši ga u magacin i idi na korak (6I).

(4I) Ako je tekući element operator tada uradi sledeće:

ako taj operator ima viši prioritet od vrha magacina upiši ga u magacin,

inače pročitaj operator iz magacina, pošalji ga na izlaz i ponovi korak (4I).

Ovde pretpostavljamo da zagrade imaju niži prioritet od svih operatora, a ako je magacin prazan onda ga tretiramo kao element sa najnižim prioritetom.

(5I) Ako je tekući element zatvorena zagrada, tada čitamo operatore iz magacina i šaljemo ih na izlaz sve dok ne pročitamo otvorenu zagradu (koju ne treba slati na izlaz).

(6I) Ako ima još elemenata u ulaznom izrazu koji nisu obradjeni tada uzimamo sledeći i idemo na korak (2I). Inače, pročitamo ostatak magacina, šaljemo ga na izlaz i završavamo sa obradom.

Algoritam 1.2 (transformacija postfiksnog izraza u infiksni)

(1P) Polazaći sleva, ispitaj tekući element izraza.

(2P) Ako je tekući element operand, upiši ga u magacin.

(3P) Ako je tekući element binarni operator, tada pročitamo dva operanda iz magacina, izvršimo odgovarajuću operaciju i upišemo rezultat u magacin; ali, ako je tekući element unarni operator, pročitamo samo jedan operand iz magacina, izvršimo odgovarajuću operaciju i upišemo rezultat u magacin [8].

(4P) Ako u ulaznom izrazu ima još elemenata uzimamo sledeći i idemo na korak (2P).

(5P) Ako je obradjen čitav ulazni niz tada je odgovarajući infiksni izraz na vrhu magacina; po njegovom čitanju algoritam se završava.

Zapazimo da u drugom algoritmu na kraju imamo vrednost odgovarajućeg izraza ako smo radili sa vrednostima operanada, ali ako radimo sa magacinom stringova dobićemo sumbolički infiksni izraz. U stvari, umesto operanada ili delova izraza magacin će sadržati pointere na odgovarajuće stringove. Na ovaj način smo u stanju da dobijemo simbolički izraz u infiksnom obliku.

Prikazani algoritmi se mogu proširiti tako da obradjuju n-arne operatore. Pri tom je potrebno poznavati listu tih operatora, kao i broj argumenata koji oni zahtevaju. Ilustracije radi, uvedimo operator fun sa tri argumenta - $fun(x, y, z)$. Sada se odgovarajući algoritmi mogu iskazati na sledeći način:

U algoritmu za transformisanje infiks izraza u postfiks uvešćemo dodatni korak, recimo (3Ia):

(3Ia) Ako je tekući element izraza zarez idi na korak (6I).

U algoritmu za transformisanje postfix izraza u infiks korak (3P) je izmenjen:

(3P) Ako je tekući element binarni operator, tada pročitaj dva operanda iz magacina, izvrši operaciju i upiši rezultat u magacin; ali, ako je tekući element unarni operator, pročitaj samo jedan operand iz magacina, izvrši operaciju i upiši rezultat u magacin. Ako je element n-arni operator (tj. trinarni u našem slučaju), pročitaj n operanada iz magacina, izvrši operaciju (tj. napravi string koji predstavlja poziv funkcije - $fun(x, y, z)$ ako su x, y i z tri operanda na vrhu magacina) i smesti rezultat u magacin.

Program koji transformiše proizvoljni infiks izraz u postfiks i obrnuto, u početku prihvata string koji predstavlja ulazni izraz a zatim vrši razdvajanje različitih elemenata izraza. Elementi izraza mogu biti operatori $+$, $-$, $*$, $/$, $\hat{}$ (operator stepenovanja na konstantnu celobrojnu vrednost), otvorene i zatvorene zagrade, unarni operatori, tj. standardna imena funkcija (*neg*, *plus*, *sin*, *cos*, *tan*, *ctg*, *log*, *exp*, *sqrt*; ova lista se lako može proširiti), i operandi koji mogu biti promenljive ili konstantne (celi brojevi ili realni u fikisnom zarezu). Posle razdvajanja dobijamo niz stringova koji predstavljaju elemente izraza. Posle toga se operator stepenovanja (npr. x^3) zamenjuju višestrukim množenjima ($x * x * x$).

Sada se poziva funkcija koja vrši transformaciju izraza u postfiksni oblik.

Druga funkcija transformiše dobijeni postfiksni izraz nazad u infiksni oblik. Podsetimo se da, umesto vrednosti operanada, koristimo pointere na stringove kako bi dobili simbolički infiks izraz.

Primer 1.1. *U ovom primeru ćemo prikazati primenu izloženih algoritama na izraz koji sadrži trinarni operator fun.*

```
You entered the following expression
x^2+x*cos(fun(x1,x2,x3))/x
This expression in postfix is
x x * x x1 x2 x3 fun cos * x / +
Now, we transform postfix expression back to infix
x*x+x*cos(fun(x1,x2,x3))/x
```

1.2 Oblasti moguće primene metode inverzne poljske notacije

Recimo nekoliko reči o oblastima u kojima se metod inverzne poljske notacije može sa uspehom primeniti. Napomenimo da ćemo se ovde dotaći onih oblasti na koje je primena ovog metoda razradjena u daljem tekstu, ali da to ne znači da ne postoje mogućnosti i za iznalaženje novih oblasti i problema u kojima ovaj pristup može da ima uspeha.

1.2.1 Simboličko diferenciranje

Automatsko diferenciranje je veoma važno u matematici i tehnici. Diferenciranje je jedan od najosnovnijih matematičkih alata sa širokom primenom u analizi i sintezi sistema, optimiziciji, upravljanju i dr. Opis sistema obično obuhvata linearne ili nelinearne jednačine, pa je za optimizaciju ili analizu često potreban gradijent, Hesijan ili Jakobijan, tj. neki izvod.

Grubo govoreći, postoje dva glavna pravca u automatskoj diferencijaciji: numerički i simbolički. Numerička diferencijacija pati od problema grešaka odsecanja i zaokruživanja [13], [14]. Sa druge strane, simbolička derivacija, kao deo automatskog diferenciranja, razmatra različite tehnike za odredjivanje izvoda matematičkog izraza koristeći jedino simboličke manipulacije nad ulaznim izrazom. Jasno je da ovde ne postoje greške odsecanja i zaokruživanja, ali simbolička derivacija može da da znatno složeniji izraz za izvod nego što je originalna funkcija, pri čemu se troši mnogo vremena i memorijskog prostora za manipulaciju formulama. Šta više, simbolička derivacija se ne može primeniti na one funkcije čije izračunavanje podrazumeva uslovna grananja [14]. Bez obzira na napredak u obe oblasti, numerička diferencijacija se smatra opštijim pristupom [13].

Ali, u nekim aplikacijama, kao što je to slučaj u veštačkoj inteligenciji ili ekspertnim sistemima, važno je raspolagati softverom koji je u stanju da obradi proizvoljne formule, poznate tek u vreme izvršavanja. Ovo je neophodno i u modeliranju i simulaciji ako želimo da razdvojimo onoga koji pravi model ili vrši simulaciju od programera.

Ugradjivanje nekih formula u kod bi nametalo izmenu koda i ponovno prevodjenje kada se promene formule koje se obradjuju. Numerička diferencijacija poseduje isti problem: korisnik i programer moraju biti ista osoba, ili naše sredstvo za diferencijaciju neće biti opšte primenljivo.

Ovi razlozi predstavljaju motivaciju za izradu alata za simboličko diferenciranje. Medjutim, ovaj problem nas je motivisao za čitav niz istraživanja na temu korišćenja inverzne poljske notacije u ovoj, ali i drugim oblastima. Naime, najčešći pristup u simboličkoj

derivaciji je da se napravi stablo izraza koji se diferencira i izvrše odredjene manipulacije nad tim stablom kako bi se dobilo stablo izraza koji predstavlja izvod ulazne formule. Izrada stabla izraza počinje njegovim pretvaranjem u postfiksnu notaciju. Takodje se rezultujuće stablo može pročitati tako da se dobije postfiksni oblik izvoda. Ova situacija nas je navela da postavimo sledeće pitanje. Ako se polazi od izraza koji je u inverznoj poljskoj notaciji i na kraju se, posle niza manipulacija nad dinamičkim strukturama podataka, dobija izvod u postfiksnom obliku, da li je moguće potrebnu obradu vršiti direktno nad izraznom u postfiksnom obliku? Rezultati u ovoj disertaciji daju potvrdan odgovor na to pitanje. Osim toga, ovakav pristup primenili smo i na druge oblasti čime smo pokazali opštost ove metode.

1.2.2 Algebarska izračunavanja

Ova oblast je možda i zahvalnija za primenu našeg metoda od simboličke diferencijacije. Razlog za ovo leži u činjenici da se ovde radi sa relativno jednostavnim izrazima koji sadrže mali broj različitih operacija i relativno ograničen skup operanada.

U daljem tekstu izložene su dve primene naše metode. Prva se odnosi na izračunavanje unarnih parfunkcija i inspirisana je radom [37]. Druga je "školska" i tiče se funkcija uparivanja iz teorije izračunljivosti [3].

1.2.3 Triangulacija poligona

Triangulacija konveksnih mnogouglova je stari i poznati problem. U literaturi postoje mnoga rešenja ovog problema. Ono što je ovde interesantno, i što je predstavljeno u trećoj glavi, jeste činjenica da smo naš metod iskoristili za konstrukciju algoritma koji radi sa celim brojevima i u suštini nije simbolički već numerički algoritam.

Glava 2

Osobine inverzne poljske notacije

U ovoj glavi su izloženi rezultati istraživanja osobina inverzne poljske notacije na kojima se zasniva naš metod ([22]). Uz to su dati i neki detalji implementacije primene ovih osobina na redukciju neopotrebnih zagrada prilikom prevodjenja postfiksnog izraza u infiksni oblik.

2.1 Osnovne definicije i teoreme

Pretpostavimo da imamo ulazni izraz transformisan u inverznu poljsku notaciju, pri čemu su svi njegovi *elementi* (promenljive, konstante i operatori) razdvojeni. Prema tome, radićemo sa nizom stringova koji predstavljaju elemente ulaznog izraza. Označimo ovaj niz sa $postfix$, gde je $postfix[i]$, za svako $i \geq 0$, string koji predstavlja jedan element izraza, odnosno promenljivu, konstantu ili operator.

Defincija 2.1. *Zahvat elementa $postfix[i]$ je broj njemu prethodnih elemenata koji čine operand(e) elementa $postfix[i]$. Zahvat elementa $postfix[i]$ označićemo sa*

$$GR(postfix[i]).$$

Kao što je to uobičajeno, ceo broj i zvaćemo indeksom elementa $postfix[i]$. Indeks i elementa $postfix[i]$ alternativno će biti označen sa $IND(postfix[i])$.

Komentar 2.1. *Element $postfix[i]$ u nizu $postfix$, koji predstavlja inverznu poljsku notaciju odgovarajućeg izraza, može biti operator ili prost operand (promenljiva ili konstanta). Smatraćemo svaki prosti operand 0-arnim operatorom, i pretpostaviti da je njegov zahvat jednak nuli.*

Primer 2.2. *Na primer, zahvat operatora + u ab+ je dva, jer su njemu prethodna dva elementa operandi operacije +. U nešto složenijem postfiks izrazu*

$$1\ \ 2\ \ x\ \ sqrt\ \ *\ \ /\ \ neg\ \ x\ \ sqrt\ \ x\ \ sqrt\ \ *\ \ / \tag{2.1}$$

koji predstavlja inverznu poljsku notaciju izraza

$$neg(1/(2 * sqrt(x)))/(sqrt(x) * sqrt(x))$$

zahvati operatora sadržanih u tom izrazu dati su u sledećoj tabeli:

postfix[3]=sqrt	*1*
*postfix[4]=**	*3*
postfix[5]=/	*5*
postfix[6]=neg	*6*
postfix[8]=sqrt	*1*
postfix[10]=sqrt	*1*
*postfix[11]= **	*4*
postfix[12]=/	*12*

Defincija 2.2. *Zahvaćeni elementi operatora $postfix[i]$ su oni elementi u nizu $postfix$ koji mu prethode sleva i koji formiraju njegove operande. Indeks krajnjeg levog medju njima nazvaćemo levom granicom zahvata. Levu granicu zahvata operatora $postfix[i]$ označićemo sa $LGB(postfix[i])$.*

Defincija 2.3. *Element $postfix[i]$ se naziva glavni element ili glava izraza koji čine $postfix[i]$ i njegovi zahvaćeni elementi.*

Komentar 2.2. *Proizvoljni element $postfix[i]$ se može posmatrati kao operator koji deluje na operande $arg_1, \ldots, arg_n$. Glave ovih operanada biće označene sa $op_1, \ldots, op_n$.*

Primer 2.3. *Razmotrimo izraz (2.1). Element $postfix[12] = /$ ima dva argumenta:*

$$arg_1 = 1\ \ 2\ \ x\ \ sqrt\ \ *\ \ /\ \ neg, \qquad arg_2 = x\ \ sqrt\ \ x\ \ sqrt\ \ *$$

Glave ovih argumenata su

$$op_1 = neg = postfix[6], \qquad op_2 = * = postfix[11].$$

Slično, operator $neg = postfix[5]$ ima za operand

$$arg_1 = 1 \quad 2 \quad x \quad sqrt \quad *$$

čija glava je

$$op_1 = / = postfix[5].$$

Lema 2.1. *Pretpostavimo da je $postfix[i]$ n-arni operator gde su glave njegovih operanada $op_1, \ldots, op_n$, respektivno. Tada važe sledeća tvrdjenja:*

(a) $GR(postfix[i]) = i - LGB(postfix[i])$.

(b) $GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k) = n + \sum_{k=1}^{n} (IND(op_k) - LGB(op_k))$.

(c) $LGB(postfix[i]) = i - n - \sum_{k=1}^{n} (IND(op_k) - LGB(op_k))$.

(d) $i = IND(postfix[i]) = n + \sum_{k=1}^{n} IND(op_k) + p$,
$LGB(postfix[i]) = \sum_{k=1}^{n} LGB(op_k) + p$,
za neki ceo broj p.

(e) $op_{n-j} = postfix\left[i - \sum_{k=1}^{j-1} GR(op_{n-k}) - j - 1\right], \quad j = 0, \ldots, n-1.$

Dokaz. **(a)** Sledi neposredno iz Definicija 2.1 i 2.2.

(b) Iz Definicija 2.1 – 2.3 je sasvim jasno da su glave $op_1, \ldots, op_n$ i njima zahvaćeni elementi u stvari zahvaćeni elementi elementa $postfix[i]$. Prema tome,

$$GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k).$$

Iz **(a)**, za svako $k = 1, \ldots, n$ zahvat od op_k jednak je

$$IND(op_k) - LGB(op_k).$$

Tada imamo

$$GR(postfix[i]) = n + \sum_{k=1}^{n} (IND(op_k) - LGB(op_k)).$$

(c) Ovaj deo je laka posledica iskaza pod **(a)** i **(b)**.

(d) Iz **(c)** dobijamo identitet

$$i - LGB(postfix[i]) = n + \sum_{k=1}^{n} IND(op_k) - \sum_{k=1}^{n} LGB(op_k).$$

Pošto su i i $LGB(postfix[i])$ kao i $IND(op_k)$ i $LGB(op_k)$, $k = 1, \ldots, n$ nenegativni celi brojevi, imamo da je

$$i = n + \sum_{k=1}^{n} IND(op_k) + p, \quad LGB(postfix[i]) = \sum_{k=1}^{n} LGB(op_k) + p,$$

gde je ceo broj p jednak

$$p = i - n - \sum_{k=1}^{n} IND(op_k) = LGB(postfix[i]) - \sum_{k=1}^{n} LGB(op_k).$$

(e) Za svako $j = 0, \ldots, n-1$, pozicija glave op_{n-j} se može dobiti oduzimanjem broja

$$\sum_{k=1}^{j-1} GR(op_{n-k}) + j + 1$$

od $i = IND(postfix[i])$. Broj

$$\sum_{k=1}^{j-1} GR(op_{n-k})$$

sadrži zahvate prethodnih glava op_{n-k}, $k = 1, \ldots, j-1$, kao i broj $j+1$ koji predstavlja lokacije elemenata op_{n-k}, $k = 1, \ldots, j-1$ kao i lokacije od op_{n-j} i $postfix[i]$. ■

Lema 2.2. *Ako je zahvat proizvoljenog n-arnog operatora $postfix[i]$ veći od n, tada je bar jedna od glava njegovih argumenata takodje operator.*

Dokaz. Pretpostavimo da je zahvat n-arnog operatora

$$postfix[i] = f(arg_1, \ldots, arg_n)$$

veći n. Ako su $op_1, \ldots, op_n$ glave argumenata $arg_1, \ldots, arg_n$, respektivno, tada, prema Lemi 2.1, imamo

$$GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k) > n.$$

Ovo implicira postojanje celog broja $k \in \{1, \ldots, n\}$ koji zadovoljava uslov $GR(op_k) > 0$. Prema tome, op_k je takodje operator. ■

U slučaju $n = 2$ dobijamo sledeće.

Posledica 2.1. *Ako je zahvat proizvoljnog binarnog operatora $postfix[i]$ veći od 2, tada je bar jedan od njemu prethodna dva elementa u inverznoj poljskoj notaciji ($postfix[i-1]$ i $postfix[i-2]$) takodje operator (unarni ili binarni).*

Teorema 2.1. *Neka je zahvat proizvoljnog binarnog operatora $postfix[i]$ veći od 2.*

(a) *Ako je razlika zahvata operatora $postfix[i]$ i zahvata prvog operatora koji mu prethodi jednaka 2, tada nije potrebno ubaciti zagrade oko najmanje jednog od dva operanda operatora $postfix[i]$. Preciznije,*

(i) ako je razlika indeksa i i indeksa prvog prethodnog operatora u odnosu na element $postfix[i]$ jednaka 1, tada nije potrebno ubaciti zagrade oko prvog izraza-operanda operatora $postfix[i]$;

(ii) ako je razlika indeksa i i indeksa prvog prethodnog operatora u odnosu na element $postfix[i]$ jednaka 2, tada nije potrebno ubaciti zagrade oko drugog izraza-operanda operatora $postfix[i]$.

(b) *U suprotnom slučaju, kada je razlika gore pomenutih zahvata veća od 2, tada zagrade treba umetnuti oko oba izraza-operanda. Izuzetak se pojavljuje u slučaju kada je jedan od izraza-operanada poziv unarnog operatora. U tom slučaju se zagrade mogu izostaviti.*

Dokaz. Neka operator $postfix[i]$ ima argumente čije su glave op_1 and op_2.

(a) U skladu sa Lemom 2.2, za slučaj $n = 2$, najmanje jedna od glava argumenata op_1 i op_2 je takodje operator. Iz dela **(e)** Leme 2.1 imamo da je $op_2 = postfix[i-1]$.

(i) Prema pretpostavci, $op_2 = postfix[i-1]$ je operator. Na osnovu Leme 2.1, dobijamo

$$2 = GR(postfix[i]) - GR(postfix[i-1]) = 2 + GR(op_1).$$

Prema tome, $GR(op_1) = 0$, što povlači da je op_1 prost operand.

(ii) U ovom slučaju, $op_1 = postfix[i-2]$ je operator. Prema Lemi 2.1, imamo

$$2 = GR(postfix[i]) - GR(postfix[i-2]) = 2 + GR(op_2).$$

To znači daje $GR(op_2) = 0$, odakle sledi da je $op_2 = postfix[i-1]$ prost operand.

(b) Pretpostavimo da je $op_2 = postfix[i-1]$ operator. Uz pomoć Leme 2.1 lako se dobija:

$$GR(postfix[i]) - GR(postfix[i-1]) = 2 + GR(op_1) > 2.$$

Odatle je $GR(op_1) > 0$, što znači da op_1 nije prost operand. Za slučaj kada je operand $op_1 = postfix[i-2]$ operator tvrdjenje se pokazuje na sličan način. ∎

2.2 Pravila simplifikacije i detalji implementacije

O ovom odeljku ćemo definisati nekoliko pravila za simplifikaciju infiks izraza koji se dobija iz odgovarajućeg postfiks izraza. Takodje ćemo opisati neke rutine koje primenjuju ova pravila.

Vrednost leve granice zahvata za $postfix[i]$, jednaka

$$LGB(postfix[i]) = i - n - \sum_{k=1}^{n}(IND(op_k) - LGB(op_k))$$

može se izračunati primenjujući sledeću efektivnu proceduru napisanu u pseudokodu koji podseća na jezik C++:

```
void left_bound(int z, int *ind)
{
do{
   *ind = *ind-1;
   if(is_unary_operator(postfix[*ind]))
        left_bound(1,ind);
   else
        if (is_binary_operator(postfix[*ind]) )
           left_bound(2, ind);
        else
           if ( is_nary_operator(postfix[*ind]) )
              left_bound(n,ind);
   z = z-1;
   }
while (z);
}
```

Parameter z je arnost operatora, dok je *ind* pointer na indeks operatora čiji se zahvat izračunava. U vreme poziva funkcije, $*ind$ ima vrednost indeksa i operatora $postfix[i]$ za koji izračunavamo levu granicu zahvata. Kada se funkcija izvrši, $*ind$ je indeks odgovarajuće leve granice zahvata. Funkcije **is_unary_operator**, **is_binary_operator** i **is_nary_operator** prepoznaju unarni, binarni i n-arni operator, respektivno.

Zahvat $GR(postfix[i])$ proizvoljnog elementa $postfix[i]$ izraza u inverznoj poljskoj notaciji može se izračunati na sledeći način:

```
int grasp(int i)
   {
   int* ptr_lgb;
   int start = i;
   ptr_lgb = &start;
   if ( is_unary_operator(postfix[i]) )
      left_bound(1,ptr_lgb);
   else
      if ( is_binary_operator(postfix[i]) )
          left_bound(2, ptr_lgb);
      else
          if ( is_nary_operator(postfix[i]) )
              left_bound(n, ptr_lgb);
   return (i - *ptr_lgb);
   }
```

Glavna pravila simplifikacije, koja se koriste za eliminaciju suvišnih zagrada u infiks izrazu, mogu se formulisati na sledeći način.

Pravilo 2.1. *(a) Ako je tekući operator $postfix[i]$ u inverznoj poljskoj notaciji izraza, tokom transformacije iz postfiksnog u infiksni oblik, binarni $+$, tada nije potrebno umetnuti zagrade oko njegovih operanada.*

(b) Ako je tekući operator $postfix[i]$ u inverznoj poljskoj notaciji izraza binarni $-$, tada važi sledeće:

(i) Zagrade nisu potrebne oko prvog argumenta;

(ii) Zagrade oko drugog argumenta su potrebne jedino u slučaju da je element $postfix[i-1]$ jedan od binarnih operatora $+$ ili $-$.

Razlog za (a) i prvi deo od (b) leži u činjenici da je $postfix[i]$ operator sa najmanjim prioritetom tako da svi mogući operatori u izrazima-operandima imaju isti ili viši prioritet, i mogu se primeniti pre tekućeg operatora. Ako izrazi-operandi sadrže operande istog prioriteta, zbog asocijativnosti zagrade su suvišne.

Drugi deo od (b) ima opravdanja u sledećem. Ako $postfix[i-1]$ nije jedan od binarnih operatora $+$ ili $-$, tada je drugi argument operatora $postfix[i]$ proizvod, količnik, ili poziv

unarnog operatora. Svi oni imaju viši prioritet od $postfix[i]$ i ovaj slučaj se svodi na slučaj (a).

Iz Teoreme 2.1 dobijamo sledeća pravila. U ovim pravilima pretpostavljamo da je $postfix[i]$ binarni operator sa dva operanda arg_1 i arg_2 čije su glave op_1 i op_2, respektivno.

Pravilo 2.2. *Neka je zahvat operatora* $postfix[i]$ *veći od* 2.

(i) Ako je $GR(postfix[i]) - GR(postfix[i-1]) = 2$ *i* $postfix[i-1]$ *je unarni ili binarni operator, tada nije potrebno umetati zagrade oko prvog izraza-operanda* arg_1*, koji je odredjen glavom* $op_1 = postfix[i - GR(postfix[i-1]) - 2]$.

(ii) Ako je $GR(postfix[i]) - GR(postfix[i-2]) = 2$ *i* $postfix[i-2]$ *je unarni ili binarni operator, tada nije potrebno umetati zagrade oko drugog izraza-operanda* arg_2*, odredjenog glavom* $op_2 = postfix[i-1]$.

(iii) Izuzetak kod slučaja (i) nastaje kada je $postfix[i] = *$ *i* $postfix[i-1] = *$ *ili* $postfix[i-1] = /$. *Takodje, izuzetak kod slučaja (ii) se javlja kada je* $postfix[i] = *$ *i* $postfix[i-2] = *$ *ili* $postfix[i-2] = /$. *Tada zagrade nisu potrebne oko oba operanda* arg_1 *i* arg_2. *Postoji još jedan izuzetak slučaja (ii), kada je* $postfix[i] = /$ *a* $postfix[i-2] = *$ *ili* $postfix[i-2] = /$. *Tada nema potrebe za zagradama oko oba argumenta.*

Pravilo 2.3. *Neka je zahvat proizvoljnog binarnog operatora* $postfix[i]$ *veći od* 2 *i razlika njegovog zahvata i zahvata prvog operatora koji mu prethodi veća od* 2. *Tada treba umetnuti zagrade oko oba izraza-operanda* arg_1 *i* arg_2. *Izuzeci se javljaju u sledećim slučajevima:*

(i) Jedan (ili oba) od izraza-operanada arg_1 *i* arg_2 *je poziv unarnog operatora, tj. bar jedna od glava* op_1, op_2 *je unarni operator. Tada zagrade treba izostaviti oko tog (ili oba) argumenta.*

(ii) Operator $postfix[i] = *$ *i jedna (ili obe) od glava njegovih argumenata je* $*$ *ili* $/$. *Tada se zagrade izostavljaju oko tog (ili oba) argumenta.*

(iii) Operator $postfix[i] = /$ *i* $op_1 = *$ *ili* $op_1 = /$. *Tada zagrade treba izostaviti oko prvog argumenta* arg_1.

Iz Definicije 2.1 neposredno sleduje Pravilo 2.4.

Pravilo 2.4. *Ako je* $postfix[i]$ *binarni operator i* $GR(postfix[i]) = 2$*, tada su oba njegova operanda,* arg_1 *i* arg_2*, prosti i zagrade se izostavljaju oko oba.*

Sledeća teorema pokazuje da Pravila 2.1.–2.4. pokrivaju sve moguće slučaje kada treba izostaviti ili umetnuti zagrade prilikom transformacije postfiks izraza u infiksni oblik.

Teorema 2.2. *Pravila* 2.1.–2.4. *uklanjaju sve nepotrebne zagrade.*

Dokaz. Kada govorimo o neophodnosti zagrada prilikom primene binarnih operacija, treba posmatrati jedino glavu izraza, označenu sa $head = postfix[i]$, i glave njenih argumenata $op_1 = postfix[i - GR(postfix[i-1]) - 2]$ i $op_2 = postfix[i-1]$. Posmatraćemo samo netrivijalne slučaje kada je *head* jedan od četiri aritmetička operatora. Svaka od glava op_1 i op_2 može biti jedan od četiri aritmetička operatora $+$, $-$, $*$ i $/$, ili neki od funkcionalnih operatora, ili prost operand.

Prema tome, postoje

$$6 \times 6 \times 4 = 144$$

različite mogućnosti.

Ako su oba argumenta arg_1 i arg_2 prosti, imamo $1 \times 1 \times 4 = 4$ različita slučaja (po jedan za svaki od aritmetičkih operatora), pokrivena Pravilom 2.4.

U

$$1 \times 5 \times 4 + 5 \times 1 \times 4 = 40$$

slučajeva, kada je tačno jedan od argumenat arg_1 i arg_2 prost, primenjujemo Pravilo 2.2.

U ostatku dokaza možemo, bez gubitka opštosti, da pretpostavimo da oba argumenta nisu prosti. Tada glave op_1 i op_2 mogu biti neki od četiri aritmetička operatora i, kao peta vrsta operatora, unarni funkcionalni operatori. Stoga imamo još

$$5 \times 5 \times 4 = 100$$

preostalih mogućnosti za op_1, op_2 i *head*.

Imamo $5 \times 5 \times 1 = 25$ slučajeva, kada je $head = +$, koji su pokriveni delom (a) Pravila 2.1. Sledećih $5 \times 5 \times 1 = 25$ slučajeva, kada je $head = -$, pokriveni su delom (b) 2.1.

U preostalih $5 \times 5 \times 2 = 50$ slučajeva je $head = *$ ili $head = /$. Dva slučaja, kada su op_1 i op_2 oba funkcionalni operatori, pokrivena su Pravilom 2.3.(i). Takodje je,

$$1 \times 4 \times 2 + 4 \times 1 \times 2 = 16$$

slučajeva, kada je tačno jedan od op_1 i op_2 funkcionalni operator, pokriveno Pravilom 2.3.(i).

Preostaju nam $4 \times 4 \times 2 = 32$ slučaja, $4 \times 4 \times 1 = 16$ kada je $head = *$ i $4 \times 4 \times 1 = 16$ kada je $head = /$.

Tada su moguće sledeće kombinacije.

- U 8 slučajeva imamo da je $op_1 = +|-$, $op_2 = +|-$ i $head = *|/$. Tada su zagrade neophodne oko oba argumenta. Ovi slučaji su pokriveni opštim delom Pravila 2.3.

- U 4 slučaja imamo da je $op_1 = +|-$, $op_2 = *|/$ i $head = *$. Tada se zagrade mogu izostaviti oko argumenta arg_2, na osnovu dela (ii) Pravila 2.3.

- Slično, u 4 slučaja imamo da je $op_1 = *|/$, $op_2 = +|-$ i $head = *$. Tada se zagrade izostavljaju oko argumenta arg_1, prema delu (ii) Pravila 2.3.

- U 4 slučaja imamo da je $op_1 = *|/$, $op_2 = *|/$ i $head = *$. Tada treba izostaviti zagrade oko oba argumenta arg_1 i arg_2. Ovi slučaji pokriveni su delom (ii) Pravila 2.3.

- U preostalih 12 slučaja je $head = /$. Ovi slučaji nastaju kada je

$$op_1 = +|-, \;\; op_2 = *|/ \text{ ili } op_1 = *|/, \;\; op_2 = +|- \;\; \text{ili } op_1 = *|/, \;\; op_2 = *|/.$$

Sada su zagrade potrebne oko oba argumenta, osim u 8 slučajeva, kada je $op_1 = *$ ili $op_1 = /$, i onda su zagrade oko arg_1 suvišne. Ovi slučaji su pokriveni delom (iii) Pravila 2.3.

Prema tome, možemo zaključiti da su svi mogući slučaji pokriveni Pravilima 2.1.–2.4., tako da ova pravila korektno odlučuju o tome da li su zagrade potrebne ili ne. ■

Sve mogućnosti za op_1, op_2 i $head$, kao i potreba za zagradama, prikazane su u Tabeli 2.1. Znakom f označavamo da je op_1 ili op_2 neki od unarnih funkcionalnih operatora, dok s označava prost operand.

Tabela 2.1.

glava	op_1	op_2	arg_1 ili (arg_1)	arg_2 ili (arg_2)	# slučaja
$+$	$+\|-\|*\|/\|f\|s$	$+\|-\|*\|/\|f\|s$	arg_1	arg_2	36
$-$	$+\|-\|*\|/\|f\|s$	$+\|-$	arg_1	(arg_2)	12
$-$	$+\|-\|*\|/\|f\|s$	$*\|/\|f\|s$	arg_1	arg_2	24
$*$	$+\|-$	$+\|-$	(arg_1)	(arg_2)	4
$*$	$+\|-$	$*\|/\|f\|s$	(arg_1)	arg_2	8
$*$	$*\|/\|f\|s$	$+\|-$	arg_1	(arg_2)	8
$*$	$*\|/\|f\|s$	$*\|/\|f\|s$	arg_1	arg_2	16
$/$	$+\|-$	$+\|-\|*\|/$	(arg_1)	(arg_2)	8
$/$	$*\|/\|f\|s$	$+\|-\|*\|/$	arg_1	(arg_2)	16
$/$	$+\|-$	$f\|s$	(arg_1)	arg_2	4
$/$	$*\|/\|f\|s$	$f\|s$	arg_1	arg_2	8

Pravila 2.1. do 2.4. mogu se primeniti u sledećim rutinama pseudokoda kako bi se izbeglo umetanje nepotrebnih zagrada.

Pretpostavimo da želimo da napravimo string `res` koji predstavlja primenu operatora $postfix[i]$ na njegove operande, označene sa `arg_1` and `arg_2`, dok `op_1` i `op_2` predstavljaju glave tih operanada. Funkcija `is_low_prior` prepoznaje kada je $postfix[i]$ binarni $+$ ili $-$.

Deo (a) Pravila 1 i Pravilo 4 mogu se implementirati kao u sledećem pseudokodu.

```
if (postfix[i]=='+' || GR(postfix[i]) == 2)
  {
  strcat(res, arg_1);
  strcat(res,postfix[i]);
  strcat(res, arg_2);
  }
```

Deo (b) Pravila 2.1. i Pravilo 2.4. mogu se implementirati kao u sledećem pseudokodu.

```
if (postfix[i]=='-' || GR(postfix[i]) > 2)
  {
  strcat(res, arg_1);
  strcat(str3,content(postfix[i]));
  if (postfix[i-1].  == '+' || postfix[i-1] == '-' )
          {
          strcat(res,"(");
          strcat(res, arg_2);
          strcat(res,")");
          }
  else
          strcat(res, arg_2);
  }
```

U niže prikazanom pseudokodu, koristeći Pravila 2.2. i 2.3., pravimo string `res` pri čemu se zagrade umeću samo kada su neophodne. Primetimo da smo ilustrovali upotrebu Pravila 2.2. i 2.2. samo za slučaj $postfix[i] = *$ (za deljenje je kod sasvim sličan ovome).

Takodje primetimo da je vrednost $GR(postfix[i])$ jednaka $grasp(i)$.

```
if (postfix[i] == '*')
  if (GR(postfix[i])-GR(postfix[i-1]) == 2)
   { /* Deo (i) Pravila 2 */
   op_2=postfix[i-1]; op_1=postfix[i-GR(op_2)-2];
   strcat(res,arg_1);
   strcat(res,postfix[i]);
   if (GR(op_2)>2 && is_not_unop(op_2) &&
          !(postfix[i-1] == '*' || postfix[i-1] == '/'))
     /* Deo (iii) Pravila 2*/
     { /* postfix[i-1] je operator */
     strcat(res,"(");
     strcat(res,arg_2);
     strcat(res,")");
     }
   else /* postfix[i-1] nije operator */
     strcat(res,arg_2);
   } /* Kraj dela (i) Pravila 2 */
else
   if (GR(postfix[i])-GR(postfix[i-2]) == 2)
      { /* Deo (ii) Pravila 2 */
      op_2=postfix[i-1]; op_1=postfix[i-2];
      if (is_not_unop(postfix[i-2]) &&
             !(postfix[i-2]=='*' || postfix[i-2]=='/'))
        /* Deo (iii) Pravila 2*/
            {
            strcat(res,"(");
            strcat(res,arg_1);
            strcat(res,")");
            }
      else
            strcat(res,arg_1);
      strcat(res,postfix[i]);
      strcat(res,arg_2);
      } /* Kraj dela (ii) Pravila 2 */
   else
      { /* Pravilo 3 */
      op_2=postfix[i-1]; op_1=postfix[i-GR(op_2)-2];
      if (is_not_unop(op_1) && !(op_1=='*' || op_1=='/'))
          {
```

```
            strcat(res,"(");
            strcat(res,arg_1);
            strcat(res,")");
            }
        else
            strcat(res,arg_1);
            strcat(res,postfix[i]);
            if (is_not_unop(op_2) && !(op_2=='*' || op_2=='/'))
                {
                strcat(res,"(");
                strcat(res,arg_2);
                strcat(res,")");
                }
            else
                strcat(res,arg_2);
    }
```

Primer 2.4. *Ovaj primer ilustruje kako ovaj softver vrši eliminaciju redundantnih zagrada kada se vrši transformacija postfiks izraza nazad u infiksni oblik, koristeći Pravila 2.1., 2.3. i 2.4.*

```
You entered the following expression
(((((x)*z)-y)/(y+z)))
This expression in postfix is
x z * y - y z + /
Now, we transform postfix expression back to infix
(x*z-y)/(y+z)
```

Najspoljnije zagrade su izostavljene na osnovu osobina inverzne poljske notacije. Primenom Pravila 2.4., od $(x)*z$ *dobijamo* $x*z$*. Dalje, primenom Pravila 2.1., od* $(x*z)-y$ *dobijamo* $x*z-y$*. Najzad, primena Pravila 2.3. nam kazuje da su zagrade neophodne u izrazima* $(x*z-y)$ *i* $(y+z)$*.*

Primer 2.5. *U ovom primeru pokazana je eliminacija suvišnih zagrada korišćenjem Pravila 2.2.*

```
You entered the following expression
(x)/(y+z)
This expression in postfix is
x y z + /
Now, we transform postfix expression back to infix
x/(y+z)
```

Primer 2.6. *U sledećoj tabeli prikazani su još neki primeri eliminacije nepotrebnih zagrada.*

Tabela 2.2.

Ulazni izraz	*Rezultujući izraz*	*Primenjena pravila*
$(((x)+(3))*((2*y)/((z)+1.23)))-4.37/(x)$	$(x+3)*2*y/(z+1.23)-4.37/x$	*3(ii)-(iii),4*
$((x*y)+(y-(2*(z))))/((x*3.14)+(y-4))$	$(x*y+y-2*z)/(x*3.14+y-4)$	*1(a)-(b),3,4*
$(cos(2*(x-y)+(x+(2*z))))*(x*(y)*z)$	$cos(2*(x-y)+x+2*z)*x*y*z$	*1,3(ii),4*
$(x)/(exp(x\hat{}2)/(y))*(x*0.5/(z))$	$x/(exp(x*x)/y)*x*0.5/z$	*2(i)-(ii),3(i)-(ii),4*
$sin((exp((x)+3)/((1.2)/y)))/(cos(y/z/w))$	$sin(exp(x+3)/(1.2/y))/cos(y/z/w)$	*2(iii),3(i),4*

Glava 3

Primene metode inverzne poljske notacije

U ovoj glavi ćemo prikazati neke primene našeg metoda koje se, uglavnom, tiču simboličkih izračunavanja. Pored simboličkog diferenciranja, prikazane su primene na algebarska izračunavanja i način konstrukcije algoritama za triangulaciju poligona zasnovan na korišćenju ove metode.

3.1 Simboličko diferenciranje

Simbolička derivacija je važan problem u mnogim oblastima računarstva (npr. u veštačkoj inteligenciji, optimizaciji, itd.) gde su ulazne formule poznate tek u vreme izvršenja programa. Obično metodi simboličke derivacije koriste stabla izraza ([6], [21], [25], [26], [30]). Tradicionalna procedura za simboličku derivaciju sastoji se od sledećih glavnih koraka:

Korak S1. Pretvori uobičajenu infiksnu formulu u inverznu poljsku notaciju.
Korak S2. Napravi stablo izraza ulazne formule.
Korak S3. Napravi stablo koje predstavlja izvod ulazne formule.
Korak S4. Pročitaj stablo izvoda modifikovanim *inorder* obilaskom čime se dobija izvod ulaznog izraza.

Mada je dinamička dodela memorije efikasan metod, ipak je potrebno platiti neku cenu u vremenu i memoriji prilikom kreiranja odgovarajućeg stabla izraza. Mogu se javiti i drugi problemi, kao što je *garbage collection* problem.

U ovom odeljku predlaže se metod simboličke derivacije koji "preskače" stabla izraza. Pomoću ovog metoda dobijamo inverznu poljsku notaciju (RPN) simboličkog izvoda

ulazne formule direktno iz njene RPN i RPN njenog izvoda. Preciznije, umesto koraka $S2$ i $S3$ konstruišemo RPN izvoda ulaznog izraza. Prelaz od RPN ulazne formule na RPN njenog izvoda je u stvari predmet ove glave. Korak $S4$ je takodje modifikovan. Formula koja predstavlja izvod generiše se iz njene RPN.

Primetimo da i mnogi raniji pristupi ovom problemu takodje ne koriste dinamičke strukture (npr. [9]).

Kao i ranije pretpostavićemo da je ulazni izraz transformisan u inverznu poljsku notaciju, gde su svi njegovi *elementi* (promenljive, konstante i operatori) razdvojeni.

Dozvoljeni operatori su $+$, $-$, $*$, / (binarni aritmetički operatori), i unarni operatori $plus()$, $neg()$ (predstavljaju unarni + i unarni $-$), $sin()$, $cos()$, $tan()$, $ctg()$, $log()$, $exp()$, $sqrt()$. Takodje je dozvoljen i operator ^ stepenovanja na celobrojnu konstantu. Konstante mogu biti celobrojne ili realne u fiksnom zarezu, dok se identifikatori promenljivih specificiraju kao što je to uobičajeno u programskim jezicima.

Naš metod derivacije zasniva se na osobinama inverzne poljske notacije proučavanim u [22]. Ove osobine se koriste prilikom prelaza od RPN date formule na RPN njenog izvoda i u odgovarajućim simplifikacijama.

3.1.1 Odredjivanje izvoda i njegova simplifikacija

Naš method za prelazak od RPN date formule na RPN njenog izvoda je rekurzivni algoritam koji obradjuje izraz u RPN odpozadi. Rekurzivna funkcija koja vrši takvu transformaciju ima prototip

```
void derive(int low, int upp, char* dx)
```

gde su `low` i `upp` donja i gornja granica indeksa, respektivno, za deo polja $postfix$, koji je predmet obrade. Takodje, `dx` je pointer na string koji predstavlja promenljivu po kojoj se vrši diferenciranje.

Poslednji element niza $postfix$ je zasigurno operator, binarni ili unarni (u [22] pokazano je kako se mogu obraditi n-arni operatori). Prvi poziv funkcije `derive` vrši se za parametre 0 i i, gde je i indeks poslednjeg elementa u RPN ulazne formule. Nadalje, rekurzivni pozivi funkcije `derive` slede pravila derivacije. Naš osnovni cilj je da konstruišemo niz stringova, označen sa *derivat*, koji sadrži RPN izvoda ulazne formule. Tokom konstrukcije niza *derivat* koristimo samo niz $postfix$. Primenjuju se pravila za derivaciju kompozicije funkcija, tako da se ovaj slučaj lako obradjuje.

U programu se koristi celobrojni niz *grasp*, čiji elementi su definisani sa

$$grasp[k] = GR(postfix[k]), \quad k = 1, \ldots, i.$$

Funkcija koja izračunava zahvat opisana je u [22], kao i u drugoj glavi.

Način primene rekurzivnih poziva prirodno zavisi od operatora $postfix[upp]$ koji predstavlja glavu formule koja se diferencira.

U ovom momentu, pretpostavimo da je $postfix[upp]$ binarni operator. Primenjujući rezultate Leme 2.1, zaključujemo da je glava njegovog drugog argumenta arg_2 jednaka

$$op_2 = postfix[upp - 1].$$

Zahvaćeni elementi glave op_2 uzeti su od

$$postfix[upp - 1 - grasp[upp - 1]]$$

do $postfix[upp - 1]$. Slično, glava prvog argumenta arg_1 od $postfix[upp]$ je

$$op_1 = postfix[upp - GR(op_2) - 2] = postfix[upp - 2 - grasp[upp - 1]].$$

Zahvaćeni elementi glave op_1 su od $postfix[low]$ do

$$postfix[upp - 2 - grasp[upp - 1]].$$

Za početak, opisaćemo derivaciju zbira ili razlike. Koristićemo uobičajeni znak $\pm$ za operator sume ili razlike. Prirodno se ovaj slučaj implementira kao što sledi.

```
derive(low, upp-2-grasp[upp-1], dx);
derive(upp-1-grasp[upp-1], upp-1, dx);
derivat[index++] = postfix[upp];
```

Objasnimo sada detalje ovog koda. Promenljiva *index* je globalna celobrojna promenljiva, inicijalno jednaka nuli, koja predstavlja indeks niza *derivat*.

Neka je RPN izraza x i y označena sa $\langle x\rangle$ i $\langle y\rangle$, respektivno. Takodje pretpostavimo da su RPN njihovih izvoda označene sa $\langle x'\rangle$ i $\langle y'\rangle$, respektivno. Tada je, u opštem slučaju, RPN izraza $(x \pm y)'$ označena sa $\langle x'\rangle\langle y'\rangle$" $\pm$ ". Konstrukcija niza *derivat*, zasnovana na nizu $postfix$, je u suštini sledeća transformacija:

$$\langle x\rangle\langle y\rangle\text{" } \pm \text{ "} \;\mapsto\; \langle x'\rangle\langle y'\rangle\text{" } \pm \text{ "}.$$

Pomoću izraza

```
derive(low, upp-2-grasp[upp-1], dx);
```

vrši se sledeća transformacija

$$\langle x\rangle \;\mapsto\; \langle x\rangle',$$

a sadržaj niza *derivat* je $\langle x' \rangle$.

Posle funkcijskog poziva

```
derive(upp-1-grasp[upp-1], upp-1, dx);
```

sadržaj niza *derivat* je $\langle x' \rangle \langle y' \rangle$.

Najzad, izraz

```
derivat[index++] = postfix[upp];
```

kompletira poljsku notaciju u obliku $\langle x' \rangle \langle y' \rangle$" $\pm$ ".

Medjutim, pokazuje se da nije lako vršiti simplifikaciju dobijenog izvoda dodatnom obradom, dok je to u slučaju metoda koja koriste stabla relativno lako. Simplifikacija se sastoji od eliminacije suvišnih operacija kao što su $x + 0$, $0 + x$, $x * 0$, $0 * x$, $x * 1$ i $1 * x$ i veoma je važna, ne samo radi jasnoće izlaza programa, već i za odredjivanje izvoda višeg reda (oni se mogu dobiti iterativnom primenom algoritma). Iz tih razloga smo se orijentisali da simplifikaciju vršimo u istom prolazu sa derivacijom.

U slučaju

`postfix[upp] == "+"` ili `postfix[upp] == "-"` ,

implementacija derivacije i simultana simplifikacija izvoda opisani su sledećim glavnim koracima.

Korak 1$\pm$. [Smeštanje sadržaja $\langle x' \rangle \langle y' \rangle$ u niz *derivat*, i pamćenje startnih pozicija izraza $\langle x' \rangle$ i $\langle y' \rangle$.]

```
stind = index;                          /* Startni indeks za ⟨x'⟩ */
derive(low, upp-2-grasp[upp-1], dx);    /* Smestanje ⟨x'⟩ */
stind1 = index;                         /* Startni indeks za ⟨y'⟩ */
derive(upp-1-grasp[upp-1], upp-1, dx);/* Smestanje ⟨y'⟩ */
```

Sada je sadržaj niza *derivat* jednak $\langle x' \rangle \langle y' \rangle$.

Korak 2$\pm$. [Dinamička simplifikacija.]

Slučaj 1. [$y' = 0$.] Ako je izvod drugog argumenta jednak $y' = 0$, tj. u slučaju

```
derivat[index-1] == "0"
```  (3.1)

RPN od $(x \pm y)'$ jednaka je $\langle x' \rangle$. Vršimo simplifikaciju

$$\langle x' \rangle \text{"0"} \ \mapsto \ \langle x' \rangle$$

u nizu *derivat*. Dovoljno je da se vratimo na poziciju prethodnog elementa:

```
index--;
```

Slučaj 2. [$x' = 0$.] Ovaj slučaj se detektuje pomoću izraza

```
derivat[stind1-1] == "0"
```
(3.2)

Dva podslučaja se mogu detektovati.

A. [$0+y' = y'$.] Ako je uslov (3.2) zadovoljen, i `postfix[upp] == "+"`, tada treba da izvršimo sledeću izmenu u nizu *derivat*:

$$"0"\langle y'\rangle \ \mapsto \ \langle y'\rangle.$$

Ponovo upisujemo $\langle y'\rangle$, počev od indeksa $stind = stind1 - 1$, prepisujući prvi argument koji je jednak nuli i izostavljajući operator "+" na kraju.

```
for (k = stind1-1; k < index-1; k++)
derivat[k] = derivat[k+1];
index = k+1;
```

B. [$0 - y' = -y'$.] Sada pretpostavimo da je uslov (3.2) zadovoljen i da je `postfix[upp] == "-"`. Tada se niz *derivat* menja prema

$$"0"\langle y'\rangle \ \mapsto \ \langle y'\rangle"neg".$$

U ovom slučaju je dovoljno ponovno upisati $\langle y'\rangle$, počev od indeksa $stind = stind1 - 1$, i smestiti operator "*neg*" na kraju.

```
for (k = stind1-1; k < index-1; k++)
derivat[k] = derivat[k+1];
index = k+1;
derivat[index++]="neg";
```

Korak 3±. [$x' \neq 0$ and $y' \neq 0$.] U opštem slučaju, kada oba uslova (3.1) i (3.2) nisu zadovoljena, RPN od $(x \pm y)'$ jednaka je $\langle x'\rangle\langle y'\rangle" \pm "$, i simplifikacija nije potrebna. Pošto je sadržaj niza *derivat* u stvari jednak $\langle x'\rangle\langle y'\rangle$, upisujemo

```
derivat[index++] = postfix[upp];
```

U slučaju `postfix[upp] == "*"`, izvod proizvoda $x * y$ može se implementirati na sličan način. U ovom slučaju, sadržaj dotičnog dela niza $postfix$ je $\langle x\rangle\langle y\rangle$" $*$ ". Naša namera je da smestimo sledeći sadržaj u niz *derivat*:

$$\langle x\rangle\langle y'\rangle\text{" }*\text{"}\langle x'\rangle\langle y\rangle\text{" }*\text{""} + \text{"}.$$

Algoritam bez simplifikacije je opisan u sledećem pseudokodu, pri čemu je sadržaj dela niza *derivat* koji se trenutno konstruiše opisan u komentarima.

```
for (k = low; k <= upp-2-grasp[upp-1]; k++)
derivat[index++] = postfix[k];           /* ⟨x⟩*/
derive(upp-1-grasp[upp-1], upp-1, dx);   /* ⟨x⟩⟨y'⟩ */
derivat[index++] = "*";                  /* ⟨x⟩⟨y'⟩ "*" */
derive(low, upp-2-grasp[upp-1], dx);     /* ⟨x⟩⟨y'⟩"*"⟨x'⟩ */
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
derivat[index++] = postfix[k];           /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩ */
derivat[index++] = "*";                  /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩"*" */
derivat[index++] = "+";                  /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩"*""+" */
```

Ali, kada vršimo i simplifikaciju, prethodni kod se menja u niže prikazani.

Korak 1.* [Smesti sadržaj $\langle x\rangle\langle y'\rangle$" $*$ " i zapamti počete pozicije izraza $\langle x\rangle$ i $\langle y'\rangle$.]
Početne pozicija izraza $\langle x\rangle$ i $\langle y'\rangle$ označene su sa $stind1$ i $stind2$, respektivno.

```
stind1 = index;                 /* Upamti pocetni indeks za ⟨x⟩ */
for (k = low; k <= upp-2-grasp[upp-1]; k++)
   derivat[index++] = postfix[k];                      /* ⟨x⟩ */
stind2 = index;                 /* Upamti pocetni indeks za ⟨y'⟩ */
derive(upp-1-grasp[upp-1], upp-1, dx);              /* ⟨x⟩⟨y'⟩ */
derivat[index++] = "*";                           /* ⟨x⟩⟨y'⟩"*" */
```

Korak 2.* [Simplifikacija izraza $\langle x\rangle\langle y'\rangle$" $*$ ".]

Slučaj 1. [Simplifikacija elemenata tipa
"0"$\langle y'\rangle$" $*$" $\mapsto$ "0" i $\langle x\rangle$"0"" $*$" $\mapsto$ "0".]

```
if (derivat[stind2-1] == "0" || derivat[index-2] == "0")
  {
  index = stind1;
  derivat[index++] = "0";
  }
```

Slučaj 2. [Simplifikacija oblika "1"$\langle y' \rangle$" $*$ " $\mapsto$ $\langle y' \rangle$.]

U ovom slučaju zadovoljen je uslov `derivat[stind2-1] == "1"` . Tada ponovo upisujemo $\langle y' \rangle$, počev od $stind2 - 1$, prepisujući prvi argument "1". Takodje izostavljamo operator " $*$ " na kraju.

```
if ( derivat[stind2-1] == "1" )
   {
   for(k = stind2-1; k <= index-2; k++)
       derivat[k]=derivat[k+1];
   index = k;
   }
```

Slučaj 3. [Simplifikacija oblika $\langle x \rangle$"1"" $*$ " $\mapsto$ $\langle x \rangle$.]
U ovom slučaju je zadovoljen uslov `derivat[index-2] == "1"`. Vraćamo se nazad za dva elementa:

```
if ( derivat[index-2] == "1" )
    index -= 2;
```

Označimo najjednostavniji oblik izraza $\langle x \rangle \langle y' \rangle$ "*" sa $\{\langle x \rangle \langle y' \rangle$" $*$ "$\}$.

Korak 3.* [Smesti sdaržaj $\{\langle x \rangle \langle y' \rangle$" $*$ "$\}$ $\langle x' \rangle \langle y \rangle$" $*$ " i zapamti startne pozicije izraza $\langle x \rangle$ i $\langle y' \rangle$.]

Startne pozicije izraza $\langle x \rangle$ i $\langle y' \rangle$ označene su sa $stind1$ i $stind2$, respektivno.

```
stind1 = index;                 /* Upamti pocetni indeks za ⟨x'⟩ */
derive(low, upp-2-grasp[upp-1], dx);      /* {⟨x⟩⟨y'⟩" * "}⟨x'⟩ */
stind2 = index;                 /* Upamti pocetni indeks za ⟨y⟩ */
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
  derivat[index++] = postfix[k];        /* {⟨x⟩⟨y'⟩" * "}⟨x'⟩⟨y⟩ */
derivat[index++] = "*";              /* {⟨x⟩⟨y'⟩" * "}⟨x'⟩⟨y⟩" * " */
```

Korak 4.* [Simplifikacija izraza $\{\langle x \rangle \langle y' \rangle$" $*$ "$\}$ $\langle x' \rangle \langle y \rangle$" $*$ ".]

Može se izvršiti slično kao u koracima $1*$, $2*$ i $3*$.

Korak 5.* [Simplifikacija izraza $\{\langle x \rangle \langle y' \rangle$" $*$ "$\}$ $\{\langle x' \rangle \langle y \rangle$" $*$ "$\}$" + ".]

Ova simplifikacija se može izvršiti slično kao u koracima $1\pm$ i $2\pm$, kod derivacije sume.

U slučaju deljenja pravila za derivaciju nameću sledeću transformaciju:

$$\langle x \rangle \langle y \rangle \text{"/"} \mapsto \langle x' \rangle \langle y \rangle \text{" * "} \langle x \rangle \langle y' \rangle \text{" * "" - "} \langle y \rangle \langle y \rangle \text{" * ""/"}.$$

```
derive(low,upp-2-grasp[upp-1], dx);
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
    derivat[index++] = postfix[k];
derivat[index++] = "*";
for (k = low; k <= upp-2-grasp[upp-1]; k++)
    derivat[index++] = postfix[k];
derive(upp-1-grasp[upp-1], upp-1, dx);
derivat[index++] = "*";
derivat[index++] = "-";
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
    derivat[index++] = postfix[k];
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
    derivat[index++] = postfix[k];
derivat[index++] = "*";
derivat[index++] = "/";
```

Simplifikacija se može izvršiti slično kao i ranije.

Sada pretpostavimo da je $postfix[upp]$ unarni operator. Prema Lemi 2.1, glava njegovog argumenta arg_1 je

$$op_1 = postfix[upp - 1].$$

Takodje, zahvaćeni elementi od op_1 su od $postfix[low]$ do $postfix[upp - 1]$. Prema tome potreban je samo jedan rekurzivni poziv.

```
derive(low, upp-1, dx);
```

U slučaju unarnih operatora (tj. poziva funkcija) primenjujemo pravila derivacije za složene funkcije. Trivijalni slučaj je za funkcije *plus* i *neg* koje predstavljaju unarni + i −.

Derivacija izraza koji sadrži operator *neg* implementirana je u sledećoj rutini:

```
derive(low, upp-1, dx);            /* ⟨x'⟩ */
derivat[index++] = postfix[upp];   /* ⟨x'⟩"neg" */
```

Simplifikacija "0" "*neg*" $\mapsto$ "0" implementirana je na sledeći način:

```
if (postfix[index-1] == "0")
  index--;
```

Derivacija izraza koji sadrži operator *plus* zahteva jedino

```
derive(low, upp-1, dx);     /* ⟨x'⟩ */
```

Ilustrujmo derivaciju unarnih operatora na još par primera. Za funkciju *sin* vršimo transformaciju

$$\langle x\rangle\text{"}sin\text{"} \;\mapsto\; \langle x\rangle\text{"}cos\text{"}\langle x'\rangle\text{"}*\text{"}.$$

Lako se vidi da se derivacija složenih funkcija vrši upotrebom pravila lanca.

Implementacija derivacije sa simplifikacijom opisana je sledećim koracima.

Korak 1. [Smesti sadržaj $\langle x\rangle$"cos"$\langle x'\rangle$ u niz *derivat*.]

```
for (k = low; k <= upp-1; k++)
      derivat[index++] = postfix[k]; /* ⟨x⟩ */
derivat[index++] = "cos";            /* ⟨x⟩"cos" */
derive(low, upp-1, dx);              /* ⟨x⟩"cos"⟨x'⟩ */
```

Korak 2. [Moguća simplifikacija.]

```
if ( derivat[index-1] == "1" )
     index--;              /* ⟨x⟩"cos""1" ↦ ⟨x⟩"cos" */
else
     if ( derivat[index-1] == "0" )
          { /* Argument funkcije sin je konstanta */
          index = low;
          derivat[index++] = "0"; /* ⟨x⟩"cos""0" ↦ "0" */
          }
     else  /* Opsti slucaj */
          derivat[index++] = "*"; /* ⟨x⟩"cos"⟨x'⟩"*" */
```

Derivacija logaritamske funkcije *log* se izvodi na sledeći način:

$$\langle x\rangle\text{"}log\text{"} \;\mapsto\; \langle x'\rangle\langle x\rangle\text{"}/\text{"}.$$

```
derive(low, upp-1, dx);                                  /* ⟨x'⟩ */
if ( derivat[index-1] == "0") )
      {                /* Argument funkcije log je konstanta */
      index = low;
      derivat[index++] = "0";                  /* ⟨log(x)'⟩ = "0" */
      }
else
      {
      for (k = low; k <= upp-1; k++)
          derivat[index++] = postfix[k];          /* ⟨x'⟩⟨x⟩ */
      derivat[index++] = "/";                   /* ⟨x'⟩⟨x⟩"/" */
      }
```

Za kvadratni koren imamo sledeću transformaciju:

$$\langle x\rangle\text{"}sqrt\text{"} \;\mapsto\; \langle x'\rangle\text{"2"}\langle x\rangle\text{"}sqrt\text{""}*\text{""/"}.$$

```
derive(low, upp-1, dx);
if ( derivat[index-1] == "0") )
  {             /* Argument funkcije sqrt je konstanta c*/
  index = low;
  derivat[index++] = "0"; /* ⟨sqrt(c)'⟩ = "0" */
  }
else
  { /* Opsti slucaj */
  derivat[index++] = "2";
  for (k = low; k <= upp-1; k++)
      derivat[index++] = postfix[k];
  derivat[index++] = "sqrt";
  derivat[index++] = "*";
  derivat[index++] = "/";
  }
```

Najzad, pretpostavimo da je $postfix[upp]$ prost operand. Tada postoje dve mogućnosti. Ako je $postfix[upp]$ jednak promenljivoj dx, njen izvod je 1, što se postiže sa

```
derivat[index++]="1";
```

Inače, izvod je jednak nuli:

```
derivat[index++]="0";
```

3.1.2 Poredjenja

U poredjenju sa metodima koje se zasnivaju na stablima izraza, možemo zaključiti da *zahvaćeni* elementi odgovaraju čvorovima podstabla (ne računajući koren podstabla). Takodje, $grasp[upp]$ jednak je broju čvorova u podstablu.

Šta više, simbolička diferencijacija zasnovana na RPN može se porediti sa poznatim tehnikama za diferencijaciju u jeziku LISP (vidi, npr. [1], [11], [12]). Označimo LISP funkciju za simboličku diferencijaciju sa *deriv*. Izraz

```
derive(low, upp-2-grasp[upp-1], dx);
derive(upp-1-grasp[upp-1], upp-1, dx);
```

odgovara rekurzivnim pozivima LISP funkcije *deriv* za prvi i drugi argument, respektivno:

```
(setq d1 (deriv (cadr x)) dx)
(setq d1 (deriv (caddr x))) dx)
```

U slučaju izraza $postfix[upp] == " + "$ ili $postfix[upp] == " - "$, izraz

```
derivat[index++] = postfix[upp];
```

ekvivalentan je jednom od izraza

```
(list '+ d1 d2)     (list '- d1 d2)
```

Slične analogije mogu se izvesti za sve ostale kako binarne, tako i unarne operatore. Razlog za primenu metoda zasnovanog na RPN je u tome što ovaj algoritam zahteva samo dva globalna niza, označena sa $postfix$ i $derivat$, pa su njegovi memorijski zahtevi najmanji mogući.

3.2 Algebarska izračunavanja

U ovom odeljku ćemo predstaviti dve slične primene našeg metoda na algebarska izračunavanja.

Prva primena tiče se izračunavanja unarnih parfunkcija. Detalji implementacije polugrupe unarnih funkcija nad parovima oblika $\langle a, b\rangle$ (tzv. *unarne parfunkcije*) istraživani su u [37].

Druga primena odnosi se na koncept funkcija uparivanja. Funkcije uparivanja su sredstvo za kodiranje parova brojeva jednim brojem korišćenjem primitivno rekurzivnih funkcija.

3.2.1 Izračunavanje unarnih parfunkcija

U [37] se razmatra algebra sa binarnom operacijom $\langle , \rangle$ i dve unarne operacije L i R za koje važe sledeće tri aksiome:

A_1 : $L(\langle a, b\rangle) = a$;
A_2 : $R(\langle a, b\rangle) = b$;
A_3 : $\langle L(x), R(x)\rangle = x$.

Parfunkcija je funkcija ove algebre u samu sebe i može se definisati korišćenjem operatora $\langle , \rangle$, L i R.

U [37] je opisana implementacija unarnih parfunkcija u jeziku MAPLE. Takodje, u [37] se jezik MAPLE koristi za automatsku verifikaciju jednačina potrebnih za opis polugrupe.

Ovaj odeljak sadrži opis algoritma za simboličko izračunavanje unarnih parfunkcija polugrupe iz [37].

Naš algoritam je implementiran u programskom jeziku C++ pod operativnim sistemom DOS (tako da su memorijska ograničenja ozbiljnija nego u slučaju Windows programa. Ovaj algoritam zasnovan je na metodi proširene inverzne poljske notacije uvedenoj u [22]. Metod iz [22] podešen je tako da radi sa izrazima koji sadrže operatore $\langle, \rangle$, L, R, i njihovu kompoziciju. Takodje su podržane imenovane kompozicije (označene velikim slovima I, S, B, M, D, T, C, E, A, i Q prema [37]). Argumenti ovih funkcija su promenljive označene malim slovima ili njihovi parovi napravljeni pomoću operatora $\langle, \rangle$. Članovi para mogu biti takodje parovi, kao što je slučaj i u [37].

Obično se ovakvi algoritmi implementiraju pomoću dinamičkih struktura podataka kao što su povezane liste ili stabla. Ako počnemo sa statičkim poljem treba da izvršimo odredjenu obradu koristeći dinamičke strukture podataka, dobijajući na kraju izlaz koji je smešten u novi statički niz. Medjutim, naš metod, podržan osobinama inverzne poljske notacije, istraživanim u [22], ne koristi više memorije nego što je potrebno za početni i rezultujući statički niz. Bez obzira na efikasnost dinamičke dodele memorije, na ovaj način štedimo nešto prostora i vremena.

Pretpostavljamo da obradjujemo izraze koji sadrže sledeće simbole: $\langle$, $\rangle$, , (zarez), (,), skup velikih slova (koja označavaju funkcije), i mala slova (koja označavaju argumente).

Naš metod zasniva se na inverznoj poljskoj notaciji (RPN) i sastoji se od tri različite faze. Prva faza predstavlja transformaciju ulaznog izraza u njegovu RPN. U drugoj fazi se implementira preslikavanje RPN ulaznog izraza u RPN rezultujućeg izraza. Najzad, u trećoj fazi konvertujemo RPN rezultujućeg izraza u uobičajeni oblik. Rezultat prve faze nazvaćemo *ulazni poljski izraz* (IPE), a rezultet druge faze *izlazni poljski izraz* (OPE).

U svrhu simboličkog izračunavanja unarnih parfunkcija napravićemo izmenu u standardnom algoritmu za transformaciju infiks izraza u postfiksni. Naglašavamo da se zarez smatra operatorom čiji je prioritet niži od prioriteta funkcijskih operatora kao što su L, R, i dr.

Razlike izmedju ovog algoritma i *Algoritma 1.1* su sledeće. Sada imamo dodatni korak *(3Ia)*:

(3Ia) Ako je tekući element $\langle$, upiši ga u magacin i idi na Korak (6I).

Korak (4I) se menja i može se iskazati kao:

(4I) Ako je tekući element operator tada, ako on ima viši prioritet od vrha magacina

upiši taj operator u magacin; inače pročitaj operator iz magacina, pošalji ga na izlaz, i ponovi Korak (4I).

Zapazimo da zagrade imaju niži prioritet od bilo kog operatora, kao i da se prazan magacin tretira kao element najnižeg prioriteta. Uz to, zarez ima poseban tretman: on se ponaša kao operator ali se izostavlja njegovo upisivanje u magacin.

Takodje ćemo dodati još jedan korak *(5Ia)*:

(5Ia) Ako je tekući element $\rangle$, pročitaj sve operatore iz magacina i pošalji ih na izlaz sve dok se ne naidje na $\langle$ i, umesto $\langle$, pošalji $\rangle$ na izlaz.

Koristeći ovaj algoritam dobijamo ulazni izraz pretvoren u postfiksni oblik (IPE) koji sadrži jedino slova (mala ili velika) i simbol $\rangle$.

Primer 3.1. *Evo nekoliko primera parfunkcija i njihovih RPN.*
$\langle\langle a, b\rangle, c\rangle \mapsto ab\rangle c\rangle$
$S(\langle L(x), R(\langle a, b\rangle)\rangle) \mapsto xLab\rangle R\rangle S$

Naš zadatak je da simbolički odredimo rezultujući izraz, takodje u postfiksnom obliku (OPE). Ova transformacija je suština druge faze i obavlja se direktno iz IPE, bez korišćenja dinamičkih struktura podataka.

Pre nego što pokažemo kako se to radi, objasnimo transformaciju iz postfiksne u uobičajenu formu koja se obavlja u trećoj fazi. Razlika u odnosu na *Algoritam 1.2* je u koraku (3P) koji se menja i sada glasi:

(3P) Ako je element funkcionalni operator (L, R, itd.), pročitaj jedan operand, primeni operaciju i smesti rezultat u magacin; ali, ako je element $\rangle$, tada pročitaj dva operanda iz magacina, napravi par i smesti rezultat u magacin.

Napomenimo još da se prethodnim algoritmom dobija simbolički izraz, obzirom da koristimo magacin karaktera. Primena operatora (pomenuta u *Koraku (3P)*) znači pravljenje stringova koji predstavljaju odgovarajući izraz.

Primer 3.2. *Prikažimo dva primera izraza u RPN i u uobičajenoj formi.*
$ab\rangle cd\rangle LB \mapsto B(L(\langle a, b\rangle, \langle c, d\rangle))$
$abc\rangle\rangle de\rangle\rangle \mapsto \langle\langle a, \langle b, c\rangle\rangle, \langle d, e\rangle\rangle$

Razmotrimo sada drugu fazu algoritma. Kada imamo algoritme za transformaciju izraza sa parfunkcijama u inverznu polsjku notaciju i obrnuto, preostaje problem implementacije simboličkog izračunavanja tih parfunkcija.

Obzirom na činjenicu da su svi izrazi ovakve vrste kompozicije operatora L, R i $\langle , \rangle$, razvićemo svaku imenovanu kompoziciju preko operatora L, R i $\langle , \rangle$, saglasno njihovim definicijama iz [37]. Prikažimo ovde te definicije:

$$\begin{aligned}
I &: x \mapsto x \\
S &: x \mapsto \langle R(x), L(x)\rangle \\
B &: x \mapsto \langle\langle L(x), L(R(x))\rangle, R(R(x))\rangle \\
M &: x \mapsto \langle L(L(x)), R(x)\rangle \\
D &: x \mapsto \langle x, x\rangle \\
T &: x \mapsto \langle\langle R(L(x)), L(L(x))\rangle, R(x)\rangle \\
C &: x \mapsto \langle\langle\langle L(L(x)), L(R(L(x)))\rangle, R(R(L(x)))\rangle, R(x)\rangle \\
E &: x \mapsto \langle\langle L(x), L(x)\rangle, R(x)\rangle \\
A &: x \mapsto \langle L(L(x)), \langle R(L(x)), R(x)\rangle\rangle \\
Q &: x \mapsto \langle\langle L(L(L(x))), \langle R(L(L(x))), R(L(x))\rangle\rangle, R(x)\rangle
\end{aligned}$$

pri čemu je x izraz koji može da sadrži parfunkcije i operator $\langle , \rangle$ i njihove kompozicije.

Prema našem algoritmu za transformaciju u postfiksnu formu, gornje definicije se mogu izraziti kao sledeće transformacije iz IPE u OPE:

$$\begin{aligned}
\tilde{x}I &\mapsto \tilde{x} \\
\tilde{x}S &\mapsto \tilde{x}R\tilde{x}L\rangle \\
\tilde{x}B &\mapsto \tilde{x}L\tilde{x}RL\rangle\tilde{x}RR\rangle \\
\tilde{x}M &\mapsto \tilde{x}LL\tilde{x}R\rangle \\
\tilde{x}D &\mapsto \tilde{x}\tilde{x}\rangle \\
\tilde{x}T &\mapsto \tilde{x}LR\tilde{x}LL\rangle\tilde{x}R\rangle \\
\tilde{x}C &\mapsto \tilde{x}LL\tilde{x}LRL\rangle\tilde{x}LRR\rangle\tilde{x}R\rangle \\
\tilde{x}E &\mapsto \tilde{x}L\tilde{x}L\rangle\tilde{x}R\rangle \\
\tilde{x}A &\mapsto \tilde{x}LL\tilde{x}LR\tilde{x}R\rangle\rangle \\
\tilde{x}Q &\mapsto \tilde{x}LLL\tilde{x}LLR\tilde{x}LR\rangle\rangle\tilde{x}R\rangle
\end{aligned} \tag{3.3}$$

gde je $\tilde{x}$ RPN izraza x, koji sadrži parfunkcije i operator $\langle , \rangle$ i njihovu kompoziciju.

U ovom momentu može se zaključiti šta je generalna ideja ovog algoritma. Primenom ovih transformacija razvićemo svaki IPE u odgovarajući OPE, koji se sastoji samo od operatora L, R i $\rangle$. Potom treba primeniti odgovarajuće operacije da bi odredili OPE. Medjutim, ako pokušamo da to uradimo direktno, gotovo sigurno će doći do pojave proznatog problema sa simboličkim izračunavanjima ([13], [14]). Dužina razvijenog izraza prilikom razvijanja svake kompozicije raste linearno, tako da iole duži izraz može da dovede do pada programa, bez obzira koliko memorije imamo na raspolaganju. Rešenje ovog problema je suštinsko za implementaciju algoritma.

Primer 3.3. *Možemo da izračunamo dužinu rezultujúg stringa kada se razvoj vrši na direktan način. Na primer, posmatrajmo jednu od 77 jednačina koje opisuju polugrupu iz [37]*

$$A = SBSBS. \tag{3.4}$$

Funkcija $A(x)$ razvijena po R, L i $\rangle$ ima sledeći oblik

$$\tilde{x}A \mapsto \tilde{x}LL\tilde{x}LR\tilde{x}R\rangle\rangle.$$

Ako označimo dužinu stringa $\tilde{x}$ sa size ($size \geq 1$), lako se može videti da je rezultujuća dužina razvijenog izraza $A(x)$ jednaka

$$3size + 7.$$

Za $size = 1$, imamo 10 za rezultujuću dužinu razvijenog izraza. Ako je jednačina (3.4) tačna (a jeste), kada razvijemo desnu stranu $S(B(S(B(S(x)))))$ i primenimo odgovarajuće operacije treba da dobijemo izraz iste dužine kao što je to izraz sa leve strane (štaviše, leva i desna strana treba da budu jednake). Ali, razmotrimo šta se dešava tokom razvoja desne strane jednačine (3.4).

Letimičan pogled na definicije $S(x)$ i $B(x)$ u (3.3) kazuje nam da svako S daje izraz dužine $2size + 3$, gde je size dužina njegovog argumenta, a svako B daje izraz dužine $3size + 7$.

Za kompoziciju $S(B(S(B(S(x)))))$ i za dužinu izraza x jednaku $size = 1$, može se izračunati da će razvijeni izraz imati dužinu 299!

U sličnom primeru jednačine

$$Q = TCTCT \tag{3.5}$$

možemo izračunati da je dužina razvijene desne strane jednačine (3.5) jednaka 1999!

Ovde su prikazani relativno "nevini" primeri gde je broj kompozicija pet. Dovoljno uporan čitalac može da se upusti u izračunavanje dužine razvijenog izraza sa 23 kompozicije kakvih ima u [37]!

Iz ovih razloga preduzećemo razvoj uz *dinamičku simplifikaciju*. Umesto direktnog razvoja uvek ćemo pokušavati da primenimo svaki operator (L, R ili $\rangle$) koji se dodaje stringu kako bi dobili maksimalnu moguću simplifikaciju izraza neposredno po uvećanju njegove dužine.

Ovaj razvoj vršimo pozivanjem funkcije `expand` čiji je prototip:

```
void expand(int low, int upp, char input[], char output[])
```

gde `low` i `upp` predstavljaju donju i gornju granicu indeksa elemenata niza koji se obradjuju, dok su `input` i `output` početni i rezultujući niz, respektivno (oba u postfiksnoj formi – zapravo, to su IPE i OPE). Na ovaj način, *glava* od IPE (vidi [22]) je element `input[upp]`.

Elementi niza `input` obradjuju se s desna na levo. Kada se prepozna funkcionalni operator vrši se rekurzivni poziv funkcije `expand`:

```
expand(low, upp-1, input, output);
```

Po završetku te funkcije, zavisno od operatora `input[upp]`, vršimo razvijanje koristeći sledeći kod (ovde ćemo prikazati kod samo za neke funkcije, jer su pravila za razvijanje slična kao i kod ostalih funkcija). Primetimo da se dinamička simplifikacija vrši pomoću funkcija `doL`, `doR` i `doAngle` koje rešavaju situacije kada se pojave operatori L, R i $\rangle$, respektivno. Ove funkcije će biti opisane kasnije.

Funkcije L i R u stvari ne zahtevaju posebno razvijanje, tako da, kada naidjemo na L, jednostavno uradimo sledeće:

```
strcpy(x,output);
calculate_grasp(x);
doL(0, strlen(x)-1, x, tmp);
strcpy(output,tmp);
```

Zapazimo da procedura *calculate_grasp(x)* odredjuje *zahvate* (vidi [22]) svih elemenata u nizu x.

Kada se javi operator R imamo sledeće

```
strcpy(x,output);
calculate_grasp(x);
doR(0, strlen(x)-1, x, tmp);
strcpy(output,tmp);
```

Funkcija I vraća njen argument, pa u tom slučaju jednostavno izostavimo da upišemo I u izlazni niz.

Nadalje ćemo prikazati kod samo za funkcije S i B, obzirom da se odatle lako može zaključiti kako se vrši razvoj za ostale funkcije. Na osnovu definicije funkcije S u (3.3), za njenu implementaciju koristimo sledeći kod:

```
strcpy(x,output);
calculate_grasp(x);
doR(low, strlen(x)-1, x, tmp);
strcpy(output,tmp);
doL(low, strlen(x)-1, x, tmp);
strcat(output,tmp);
calculate_grasp(output);
doAngle(low, strlen(output)-1, output, tmp);
strcpy(output,tmp);
```

Slično, za B imamo:

```
strcpy(x,output);
calculate_grasp(x);
doL(low, strlen(x)-1, x, tmp);
strcpy(output,tmp);
doR(low, strlen(x)-1, x, tmp);
calculate_grasp(tmp);
doL(low, strlen(tmp)-1, tmp, tmp);
strcat(output,tmp);
calculate_grasp(output);
doAngle(low, strlen(output)-1, output, tmp);
strcpy(output,tmp);
calculate_grasp(x);
doR(low, strlen(x)-1, x, tmp);
calculate_grasp(tmp);
doR(low, strlen(tmp)-1, tmp, tmp);
strcat(output,tmp);
calculate_grasp(output);
doAngle(low, strlen(output)-1, output, tmp);
strcpy(output,tmp);
```

U slučaju operatora $\rangle$ obavljamo sledeću obradu.

```
calculate_grasp(input);
expand(low, upp-2-grasp[upp-1], input, x);
strcpy(output,x);
calculate_grasp(input);
expand(upp-1-grasp[upp-1], upp-1, input, tmp);
strcat(output,tmp);
calculate_grasp(output);
doAngle(0, strlen(output)-1,output,tmp);
strcpy(output, tmp);
```

Kada je tekući element ulaznog niza malo slovo (tj. argument), ono se jednostavno prepiše u izlazni niz.

```
output[0] = input[upp];
```

Primetimo da je `grasp` celobrojni niz koji sadrži *zahvate* elemenata niza `input`. Takodje, sa `postfix` i `result` označavamo nizove koji predstavljaju odgovarajuće IPE i OPE unetog izraza, respektivno.

Pokažimo kako izgledaju funkcije `doL`, `doR` i `doAngle`. U stvari, `doL` i `doR` su slične, tako da ćemo prikazati samo `doL`. Ako je argument funkcije $L()$ par $\langle x, y\rangle$, tada je izlaz x; inače, rezultat je simbolički izraz $L(x)$ [22].

```
void doL(int low, int upp, char postfix[], char result[])
   {
   int i, j, k;
   i = upp;
   j = 0;
   if ( postfix[i] == '>' )
       for (k = i - grasp[i]; k < i-1-grasp[i-1]; k++)
             result[j++] = postfix[k];      /* ~x~y⟩L →~x */
   else
       {
       for (k = low; k <= i; k++)
             result[j++] = postfix[k];
       result[j++] = 'L';                  /* ~x →~xL */
       }
   result[j] = '\0';
   }
```

Funkcija `doAngle` ima zadatak da napravi par, ili, ako su uslovi Aksiome A_3 zadovoljeni, da napravi odgovarajuú simplifikaciju.

```
void doAngle(int low,int upp,char postfix[],char result[])
   {
   int i, j, k, l, same, len1, len2;
   i = upp;
   len1 = i-grasp[i]-1;
   len2 = grasp[i];
   same = 1;
   k = low;
   l = i-grasp[i];
   for (j = 0; j < len1; j++)
     if (postfix[k++] != postfix[l++])
         same = 0;
   if (postfix[i-1-grasp[i]] == 'L' && postfix[i] == 'R' &&
   len1 == len2 && same)
     {
     for (j = low; j < len1; j++)
     result[j] = postfix[j];
     result[j] = '\0';
     } /* ~xL~xR →~x */
   else
     {
     for (j = 0; j <= i; j++)
         result[j] = postfix[j];
     result[j++] = '>';
     result[j] = '\0';
       } /* ~x~y →~x~y⟩ */
   }
```

Posle razvoja sa dinamičkom simplifikacijom dobijamo OPE koji sadrži jedino operatore L, R i $\rangle$. Druga funkcija izvršava posao treće faze dajući rezultat u uobičajenom obliku. Ova funkcija, `doit`, ima prototip:

```
void doit(int low, int upp, char postfix[])
```

gde su `low` i `upp`, respektivno, donja i gornja granica indeksa niza koji se obradjuje, a `postfix` je niz u razvijenoj postfiksnoj formi (OPE) koji predstavlja izraz koji se ovom funkcijom obradjuje.

Obrada počinje elementom `postfix[low]` i ide nadalje na desno, prema algoritmu opisanom u ranije. Da ne bismo zamarali detaljima ovog koda objasnićemo samo jedno pitanje koje se tiče ove funkcije.

Kada operator L ili R treba da se primene na par, tada, u stvari, treba izdvojiti levi ili desni element para, respektivno. To izdvajanje elemenata para vršimo korišćenjem činjenice da je levi element para smešten sve do prvog zareza na koji naidjemo pri broju otvorenih i nezatvorenih ugaonih zagrada jednakom jedan. Primetimo da su svi izrazi u magacinu već u uobičajenoj formi. Za izdvajanje desnog elementa para tragamo sa desna za prvim zarezom pri broju otvorenih i nezatvorenih ugaonih zagrada jednakom jedan.

Verifikacija relacija grupe iz [37] je takodje implemetirana. Svih 77 jednačina smešta se u datoteku, izostavljajući zagrade i argumente u stilu [37]. Program čita jednu po jednu jednačinu iz datoteke, vrši njenu razvoj i redukciju, proizvodeći izlaz u kome se stavlja znak `==` izmedju leve i desne strane jednačine ako jednačina važi, ili znak `!=` ako jednačina ne važi. Čitava datoteka se obradjuje i proizvodi izlaz u svega par sekundi, bez ikakvih memorijski problema.

3.2.2 Simboličko izračunavanje funkcija uparivanja

Osobine funkcija uparivanja iskazane su u *Teoremi o funkcijama uparivanja* u [4].

Teorema 3.1. *Funkcije $\langle x, y\rangle$, $L(z)$, i $R(z)$ imaju sledeća svojstva:*

1. *One su primitivno rekurzivne;*
2. $L(\langle x, y\rangle) = x$, $R(\langle x, y\rangle) = y$;
3. $\langle L(z), R(z)\rangle = z$;
4. $L(z), R(z) \leq z$;

Takodje se u [3] i [4] funkcije uparivanja i njihovi inverzi definišu na sledeći način:

$$\begin{aligned} \langle x, y\rangle &= 2^x(2y+1) - 1, \\ x &= (z+1)_1, \\ y &= ((z+1)/2^x - 1)/2, \end{aligned} \tag{3.6}$$

gde je $(z+1)_1$ najveći ceo broj koji zadovoljava ${p_1}^x|(z+1)$, pri čemu je p_1 prvi prost broj (tj. 2).

U ovom odeljku ćemo opisati primenu našeg metoda na kodiranje parova brojeva. Ako se podsetimo definicija unarnih parfunkcija i funkcija uparivanja (aksiome 1–3 i Teorema 3.1, respektivno) videćemo da se jedina razlika u ponašanju funkcija L i R ogleda u tome da u poslednjem slučaju L i R ne ostaju neizračunati kada je argument ceo broj, ali da i dalje iz argumenta koji je par izdvajaju odgovarajći levi ili desni element, respektivno. Dalje, u ovom slučaju operator $\langle , \rangle$ vrši kodiranje na osnovu jednačine 3.1, umesto da napravi par kao u prethodnoj aplikaciji.

Za ovaj slučaj dozvolićemo da izrazi sadrže i četiri aritmetička operatora i operator stepenovanja ^, jer naš algoritam može da prepozna izraze kao što je $2^x(2y+1)-1$ kao argumente funkcija L i R, vraćajući x i y, respektivno. Primetimo da x i y takodje mogu biti izrazi, a ne samo promenljive. Uz to, argumenti mogu biti celi brojevi (ne samo simboli od jednog slova), tako da radimo sa nizom stringova.

Još uvek je veći deo algoritma sličan prethodno. Navešćemo glavne razlike koje se ogledaju u sledećem kodu.

U funkciji `doL` (kao i ranije dajemo kod samo za `doL` jer je `doR` sasvim slična) kada se prepozna ceo broj kao argument vrši se sledeća obrada.

```
y = atol(content(postfix[i]));
y += 1;
x = 0;
while (y % 2 == 0)
  {
  x += 1;
  y /= 2;
  }
result[0] = ltoa(x,tmp,10);
result[1].BecomeNull();
```

a kada se prepozna izraz oblika $2^x(2y+1)-1$ vrši se sledeće.

```
if (RecognizeCoding(i, postfix, left_ptr, right_ptr))
    {
    for (k = left-grasp[left-1] - 1; k < left; k++)
        {
        result[j] = postfix[k];
        j++;
        }
    result[j].BecomeNull();
    }
```

pri čemu funcija `RecognizeCoding` vraća 1 ako je šablon pomenutog izraza prepoznat, odnosno 0 u suprotnom. Takodje, ova funkcija ima paramtere `left_ptr` i `right_ptr` koji ukazuju na indeks `left` simbola ^ i indeks `right` simbola $*$ u fakotoru oblika $2 * y$, respektivno, tako da je izdavjanje stringova x ili y sasvim jednostavno.

U funkciji `doAngle` imamo sledeći dodatak. Kada se prepoznaju dva cela broja kao argumenti operatora $\rangle$ stavljamo:

```
n = atol(content(postfix[i-1]));
m = atol(content(postfix[i]));
y = power(2,n)*(2*m+1)-1;
result[0] = ltoa(y,tmp,10);
result[1].BecomeNull();
```

praveći kod za odgovarajući celobrojni par.

3.3 Konstrukcija algoritama za triangulaciju poligona

Triangulacija konveksnog mnogougla je sledeći problem. Za dati mnogougao naći broj mogućih deoba na trouglove pomoću njegovih diagonala bez preklapanja u tim deobama. To je klasičan problem rešen do sada na više načina.

Jedno rešenje iz [20] koristi bezkontekstnu gramatiku sa produkcijama:

$$S \to aSS, \quad S \to b, \tag{3.7}$$

gde su a i b terminalni, a S neterminalni simbol.

Triangulacija poligona zasniva se na sledećim principima:

(a) Neterminal S predstavlja orijentisani topološki segment koji se naziva *potencijalnim*.

(b) Niz aSS odgovara orijentisanom topološkom trouglu sa jednom realnom stranom a i dve potencijalne strane S i S.

(c) Smena $S \to aSS$ znači zamenu potencijalnih segmenata S trouglom aSS koji se sastoji od realne strane a sa definisanom orijentacijom i potencijalnim stranama S koje su stranice poligona koji se na taj način dobija.

(d) Smena $S \to b$ predstavlja zamenu potencijalne strane S realnom stranom b.

Ako primenimo $n-2$ puta prvu produkciju u (3.7) (drugim rečima, ako napravimo reč sa $n-2$ simbola a u njemu) a onda primenimo odgovarajući broj puta drugu produkciju iz (3.7), dobijamo jednu triangulaciju poligona sa n strana.

U [20] je pokazano da je broj svih mogućih triangulacija n-tougla $f(n)$ dat rekurentnom formulom

$$\begin{aligned} f(2) &= f(3) = 1, \\ f(n) &= \sum_{i=2}^{n-1} f(i)f(n-i+1) \end{aligned}$$

Naravno, implementacija ovog pristupa (posebno ako želimo da izlistamo sve trouglove preko njihovih temena) je poseban problem.

Inspirisani gore opisanim metodom, zamenićemo prvo pravilo gramatike u (3.7) i koristiti sledeća pravila za generisanje aritmetičkog izraza u inverznoj poljskoj notaciji.

$$S \rightarrow SS+ \tag{3.8}$$

$$S \rightarrow b \tag{3.9}$$

Najpre ćemo istraživati 1-1 i na preslikavanje $F_n : P_n \mapsto T_n$ podskupa izraza, generisanih uzastopnom primenom pravila (3.8) $n-2$ puta i pravila (3.9) $n-1$ puta, na skup triangulacija n-tougla, a zatim konstruisati jednostavni algoritam za triangulaciju poligona koji radi jedino sa celim brojevima, koristeći osobine inverzne poljske notacije i rezultate iz druge glave.

3.3.1 Aritmetički izrazi i triangulacije

Pošto produkcija $S \rightarrow aSS$ predstavlja konstrukciju trougla aSS sa realnom orijentisanom stranom a i dve potencijalne trane S, može se smatrati da produkcija $S \rightarrow SS+$ predstavlja konstrukciju trougla dobijenog zamenom realne strane a trougla aSS realnom stranom $+$, zadržavajući orijentaciju. Na ovaj način zamenjujemo terminal a znakom $+$ koji se može smatrati aritmetičkim operatorom sa dva S kao operandima.

Koristeći ovu korespondenciju, za svako $n \geq 3$ možemo posmatrati $F_n : R_n \mapsto T_n$, čiji domen je skup izraza dobijenih uzastopnom primenom pravila (3.8) $n-2$ puta i pravila (3.9) $n-1$ puta, a kodomen skup triangulacija n-tougla. Ali, nije teško pokazati da skup R_n sadrži višetstruke elemente. U sledećoj Lemi daćemo jedinstvenu karakterizaciju za svaki element iz R_n.

Korsiteći poznatu notaciju iz [36], sa V^n ćemo označiti sve stringove dužine n nad skupom V, a sa V^* označićemo skup zatorenja $\{\epsilon\} \cup V \cup V^2 \cup \cdots$, gde je ϵ prazna reč. Takodje, sa $\{b,+\}^*_{p,q}$ označavamo podksup skupa zatvorenja $\{b,+\}^*$ koji se sastoji od p pojavljivanja znaka b i q pojavljivanja znaka $+$. Lako se vidi da je $\{b,+\}^*_{0,0} = \{\epsilon\}$.

Lema 3.1. *Proizvoljni element r_n iz skupa R_n koji odgovara jednoj triangulaciji n-tougla jednoznačno je odredjen sledećim uslovima:*

(C1) Element je oblika

$$r_n = bb\alpha+, \quad \alpha \in \{b,+\}^*_{n-3,n-3},$$

(C2) Svaki početni deo izraza r_n (podstring uzastopnih znakova koji počinje prvim znakom) mora da bude oblika

$$\{b,+\}^*_{p,q} \quad p > q, \quad p = 1,\ldots,n-1,\ q = 1,\ldots,n-2.$$

Dokaz. Pokazaćemo ovo tvrdjenje indukcijom po n. Za $n = 3$ tvrdjenje je očigledno tačno, jer izraz $bb+$ odgovara trouglu i zadovoljava uslove $(C1)$ i $(C2)$. Pretpostavimo da je iskaz tačan za svako $n \leq k$ i pokažimo da važi i za $n = k+1$. Proizvoljna triangulacija $(k+1)$-tougla dobija se zamenom odgovarajuće strane u triangulaciji k-tougla novim trouglom. To znači da moramo najpre da primenimo pravilo (3.8) još jednom kao i pravilo (3.9) posle toga. Razmotrimo izraz

$$\begin{array}{l} bb+ \\ bb\alpha b\beta+,\ \alpha \in \{b,+\}^*_{p_\alpha,q_\alpha},\ \beta \in \{b,+\}^*_{p_\beta,q_\beta},\ p_\alpha{+}\,p_\beta{=}n{-}4,\ q_\alpha{+}q_\beta{=}n{-}3 \end{array} \tag{3.10}$$

koji zadovoljava uslov $(C2)$ i koji odgovara proizvoljnoj triangulaciji k-tougla. Svaki od ovih izraza zadovoljava uslov $(C1)$. Tada jedan od izraza

$$\begin{array}{l} bbb++ \\ bb{+}b+ \\ bbb+\alpha b\beta+ \\ bb+b\alpha b\beta+ \\ bb\alpha bb{+}\beta+, \\ \quad \alpha \in \{b,+\}^*_{p_\alpha,q_\alpha},\ \beta \in \{b,+\}^*_{p_\beta,q_\beta}, \\ \quad p_\alpha{+}p_\beta{=}n{-}4,\ q_\alpha{+}q_\beta{=}n{-}3 \end{array} \tag{3.11}$$

odredjuje odgovarajuću triangulaciju $(k+1)$-tougla, redom. Svaki od izraza iz (3.10) je jedinstven, tako da je svaki od izraza iz (3.11) jedinstveno odredjen. Pošto su sledeće transformacije valjane

$$\begin{aligned} bbb{+}{+} &= bb\gamma{+}, \quad \gamma = b{+} \in \{b,+\}^*_{1,1} \\ bb{+}b{+} &= bb\delta{+}, \quad \delta = {+}b \in \{b,+\}^*_{1,1} \\ \{bb\alpha bb{+}\beta{+}, \quad \alpha \in \{b,+\}^*_{p_\alpha,q_\alpha},\ \beta \in \{b,+\}^*_{p_\beta,q_\beta},\ p_\alpha + p_\beta = n-4,\ q_\alpha + q_\beta = n-3\} \\ &= \{bb\gamma{+},\ \gamma \in \{b,+\}^*_{n-2,n-2}\} \end{aligned}$$

zaključujemo da ovi izrazi takodje zadovoljavaju uslov $(C1)$. Koristeći iste transformacije i induktivnu hipotezu može se lako pokazati da ovaj izraz zadovoljava i uslov $(C2)$. ■

Sa P_n ćemo označiti skup izraza generisan gramatičkim pravilima (3.8) i (3.9) koji zadovoljavju uslove $(C1)$ i $(C2)$.

Posledica 3.1. *Zahvat glave izraza koji odgovara jednoj triangulaciji n-tougla jednak je* $2n-4$. *Glava zahvata* $n-1$ *znak* b *i* $n-3$ *znakova* $+$, *a početni deo zahvaćenih elemenata je oblika*

$$bb\{b,+\}^*_{p,q},\ p > q, p = 1,\ldots,n-3, q = 1,\ldots,n-3.$$

Dokaz. Iz Leme 3.1 sledi da je dužina izraza iz P_n koji odgovara proizvoljnoj triangulaciji n-tougla jednaka $2n-3$, da ima oblik $bb\{b,+\}^*_{n-3,n-3}$, i da je svaki njegov početni deo oblika $bb\{b,+\}^*_{p,q}$, $p > q$, $p = 1,\ldots,n-3$, $q = 1,\ldots,n-3$. Tada je, uz definicije 1.1 i 1.3, dokaz očigledan. ■

Lema 3.2. *Preslikavanje* $F_n : P_n \mapsto T_n$ *je dobro defininsano, 1-1 i na za svako* $n \geq 3$.

Dokaz. Najpre ćemo dokazati indukcijom da je preslikavanje F_n, $n \geq 3$, dobro definisano. Za $n = 3$, jedinstveni izraz $bb{+}$, dobijen jednom primenom pravila (3.8) i dve primene pravila (3.9), odgovara jednistvenoj i trivijalnoj triangulaciji trougla.

Pretpostavimo da tvrdjenje važi za $n = k$, $k \geq 3$.

Za slučaj $n = k+1$ razmotrimo proizvoljni izraz (3.11) generisan pravilom (3.8) primenjenim $k-1$ puta i pravilom (3.9) primenjenim k puta. U izrazu koji razmatramo ćemo zameniti pojavljivanje podniza $bb{+}$ sa b (koristeći obrnut smer pravila (3.9) and

(3.8)), i na taj način dobiti izraz oblika (3.10) generisan primenom pravila (3.8) $k-2$ puta i pravila (3.9) $k-1$ puta. Prema induktivnoj hipotezi on odgovara jedinstvenoj triangulaciji k-tougla. Ali, vraćanjem $bb+$ na prethodno mesto, dobijamo početni izraz (3.11) dodajući dodatni trougao na stranicu k-tougla koja odgovara transformisanom b. Na ovaj način dobijamo jedinstvenu triangulaciju $(k+1)$-tougla.

Na sličan način možemo, indukcijom, da pokažemo da je preslikavanje F_n 1-1.

Da bismo pokazali da je preslikavannje F_n na, konstruisaćemo efektivnu proceduru za dobijanje izraza iz P_n na osnovu date triangulacije. Pretpostavimo da je data jedna triangulacija sa orijentisanim stranicama i dijagonalama. Svakoj dijagonali i stranici AB dodelimo znak + a ostalim stranicama znak b. Posmatrajmo dve stranice koje definišu trougao nad stranicom AB. Označimo oznaku dolazne stranice sa L a odlazne stranice sa R. U opštem slučaju traženi izraz ima oblik $L+R$. Ako je L ili R jednako znaku +, treba ga okružiti zagradama i rekurzivno primeniti algoritam na odgovarajuću stranicu. Kada završimo rekurziju transformisaćemo dobijeni izraz u inverznu poljsku notaciju koja je očigledno element skupa P_n. ■

3.3.2 Konstrukcija algoritma

Defincija 3.1. *Pod nekomutativnom sumom dva cela broja a i b, u oznaci $a \oplus b$, podrazumevaćemo uobičajenu sumu $a+b$, ali ne i $b+a$.*

Defincija 3.2. *Pod višenivovskom dekompozicijom nekog parnog celog broja n podrazumevaćemo njegovu prezentaciju u obliku nekomutativne sume neparnih sabiraka ne većih od 3, napravljenu korišćenjem sledećih pravila:*

(1) Izrazi paran broj n u obliku nekomutativne sume dva neparna broja.

(2) U svakoj sumi nastavi sa dekompozicijom onih sabiraka većih od 3, ali umanjujući svaki od njih za 1 pre dalje dekompozicije. Sabirak a koji se umanjuje pre dalje dekompozicije biće označen u višenivovskoj dekompoziciji sa a_+. Vrednost ovog izraza je $a+1$.

Teorema 3.2. *Postoji bijekcija skupa T_n različitih triangulaciaj datog n-tougla na skup M_n različitih višenivovski dekompozicija broja $2n-4$.*

Dokaz. Pokazaćemo postojanje bijekcije medju skupovima P_n i M_n. Posmatrajmo izraz $\alpha+$, $\alpha \in \{b,+\}^*$ iz P_n koji odgovara proizvoljnoj triangulaciji n-tougla. Prema Posledici 2.1 zahvat njegove glave + je $2n-4$. Ovo je paran broj, tako da dužine njegovih argumenata arg_1 i arg_2 moraju biti obe parne ili obe neparne. Ali, one ne mogu biti parni brojevi, jer je najmanja dužina nekog argumenta uvek jednaka 1 (za argument S, tj. b posle transformacije $S \to b$), a svaki duži argument se dobija primenom pravila (3.8) koje uvećava dužinu izraza za dva.

Posmatrajmo glave op_1 i op_2 argumenata arg_1 i arg_2. Za slučaj da je $GR(op_1) = 1$ ili $GR(op_1) = 3$ i $GR(op_2) > 3$, zahvat glave se može izraziti kao

$$2n-4 = GR(op_1) \oplus (1 + GR(op_2)) = GR(op_1) \oplus (GR(op_2))_+.$$

Ovo daje višenivovske dekompozicije

$$2n-4 = 1 \oplus (2n-6)_+ \text{ i } 2n-4 = 3 \oplus (2n-8)_+,$$

respektivno.

Slično, za $GR(op_1) > 3$ i $GR(op_2) = 1$ ili $GR(op_2) = 3$, imamo

$$2n-4 = (1 + GR(op_1)) \oplus GR(op_2) = (GR(op_1))_+ \oplus GR(op_2).$$

Respektivno, možemo napisati

$$2n-4 = (2n-6)_+ \oplus 1 \text{ i } 2n-4 = (2n-8)_+ \oplus 3.$$

U slučaju $GR(op_1) = k_1$ i $GR(op_2) = k_2$, $k_1, k_2 \in \{5, 7, \ldots\}$, $k_1 + k_2 = 2n-4-2$, koristimo

$$2n-4 = (GR(op_1))_+ \oplus (GR(op_2))_+ = (k_1)_+ \oplus (k_2)_+$$

i primenjujemo dalju dekompoziciju na k_1 i k_2.

Najzad, jasno je da sa dekopozicijom treba nastaviti sve dok se ne dobiju argumenti ne veći od 3, jer oni odgovaraju jednom trouglu ($bb+$).

Obrnuto, kada imamo višenivovsku dekompoziciju broja $2n-4$, trivijalno je dobiti izraz iz skupa P_n, koji odgovara triangulaciji n-tougla.

Dokaz se može kopletirati korišćenjem Leme 2.2. ∎

Primer 3.4. *Za slučaj triangulacije petougla, broj* $2n-4 = 6$ *može se izraziti kao nekomutativna suma dva neparna cela broja na tri različita načina:*

$$1 \oplus 5 = 6 \qquad 3 \oplus 3 = 6 \qquad 5 \oplus 1 = 6$$

U prvoj i trećoj sumi postoje sabirci veći od 3. Njih treba dalje dekomponovati smanjujući njihove vrednosti za 1. Sve moguće dekompozicije su:

$$1 \oplus (1 \oplus 3)_+ = 6 \qquad 3 \oplus 3 = 6 \qquad (1 \oplus 3)_+ \oplus 1 = 6$$

$$1 \oplus (3 \oplus 1)_+ = 6 \qquad\qquad (3 \oplus 1)_+ \oplus 1 = 6$$

Sledećom tabelom iliustruvaémo korespodenciju izmedju skupa M_5 višenivovskih dekompozicija broja $2 * 5 - 4 = 6$ i izvodjenja odgovarajućih izraza iz P_5.

Tabela 3.1.

Višenivovska dekompozicija	*Izvodjenje odgovarajúég izraza iz P_5*
$1 \oplus (1 \oplus 3)_+ = 6$	$S \to SS+ \to SSS++ \to SSSS+++ \to bbbb+++$
$1 \oplus (3 \oplus 1)_+ = 6$	$S \to SS+ \to SSS++ \to SSS+\,S++ \to bbb+\,b++$
$3 \oplus 3 = 6$	$S \to SS+ \to SSS++ \to SS+\,SS++ \to bb+\,bb++$
$(3 \oplus 1)_+ \oplus 1 = 6$	$S \to SS+ \to SS+\,S+ \to SS+\,S+\,S+ \to bb+\,b+\,b+$
$(1 \oplus 3)_+ \oplus 1 = 6$	$S \to SS+ \to SS+\,S+ \to SSS++S+ \to bbb++b+$

Koristeći Teoremu 3.2 možemo da konstruišemo algoritam za triangulaciju n-tougla. Ovaj algoritam kao ulaz zahteva jedino broj temena mnogougla i, radeći samo sa celim brojevima, daje sve moguće triangulacije kao i liste odgovarajućih trouglova.

Algoritam 3.1

Počnimo od trougla označavajući i zapisujući njegova temena kao $A = 1$, $B = 2$ i $C = 3$ (pamteći orijentaciju trougla). Ovaj trougao odgovara izrazu $S \to SS+$. Pravimo dekompoziciju na prvom nivou (tj. razlažemo broj $2n - 4$ na dva neparna sabirka). Za svaki sabirak veći od 3 treba nastaviti sa njegovom dekompozicijom (u stvari, razlažemo vrednost koja je za jedan manja). Tokom dekompozicije temena tekućeg trougla A, B i C menjaju svoje vrednosti.

Ali, ako, na svakom nivou, dekomponujemo desni sabirak, tada to odgovara daljoj transformaciji desnog S iz $SS+$ ili daljoj transformaciji stranice (B, C). U ovom slučaju, teme A postaje bivše teme B, teme B postaje bivše teme C, a uvodi se novo teme C sa vrednošću koja je za jedan veća od minimalne medju bivšim temenima B i C. Dalje, svako teme u dotle nadjenoj triangulaciji čija je vrednost veća ili jednaka vrednosti novouvedenog temena treba da se uveća za 1.

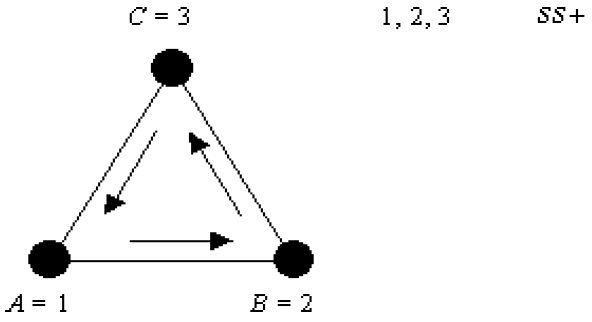
Ako dekomponujemo levi sabirak, tada to odgovara daljoj transformaciji levog S iz $SS+$ ili daljoj transformaciji stranice (C, A). U ovom slučaju teme A postaje bivše teme

C, teme B postaje bivše teme A, a uvodi se novo teme C na isti način kao i u prethodnom slučaju. Ali, ako neko od temena C ili A ima vrednost 1, uvodimo novo teme C sa vrednošću koja je za 1 veća od vrednosti maksimalne vrednosti medju vrednostima bivših temena A i C.

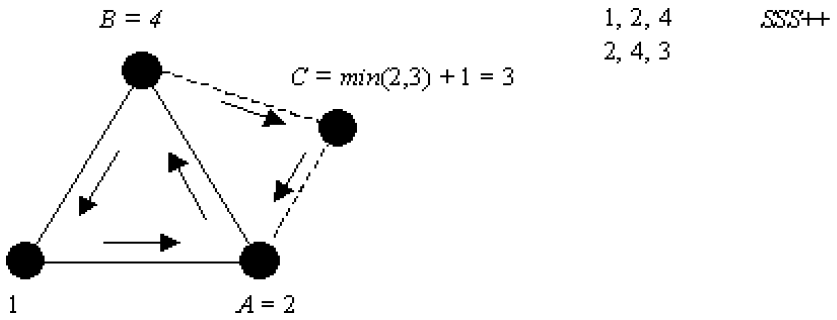
Za sabirke sa vrednošću 1 dalja dekompozicija nije moguća. Takodje, kada dobijemo sabirak sa vrednošću 3, pravimo trougao na opisani način prekidajući dalju dekompoziciju.

Prateći sve moguće "tokove" dekopozicije dobijamo sve triangulacije i liste trouglova.

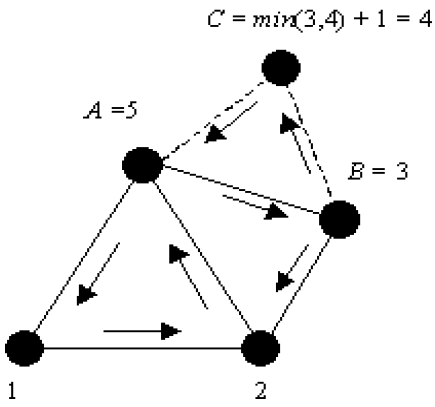
Primer 3.5. *U slučaju petougla počinjemo sa dekompozicijom vrednosti* $2n-4=6$ *kao što smo videli u Primeru 3.1. Na slikama 3.1 - 3.3 predstavljen je proces uspostavljanja trouglova i odgovarajući delovi izraza iz* P_5 *za "tok" dekompozicije* $6 = 1 \oplus 5 = 1 \oplus (1 \oplus 3)_+$.



Sl. 3.1



Sl. 3.2.

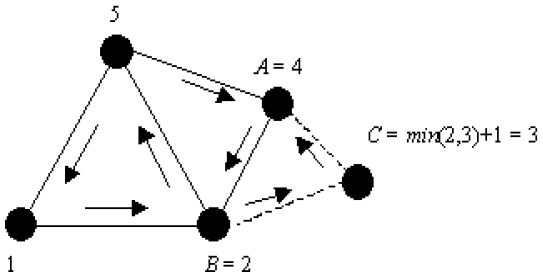


1, 2, 5
2, 5, 3
5, 3, 4

SSSS+++

Sl. 3.3.

Za tok $6 = 1 \oplus 5 = 1 \oplus (3 \oplus 1)_+$*, ovaj proces počinje na isti način kao i prethodni, ali umesto situacije sa slike 3.3 imamo onu sa slike 3.4.*

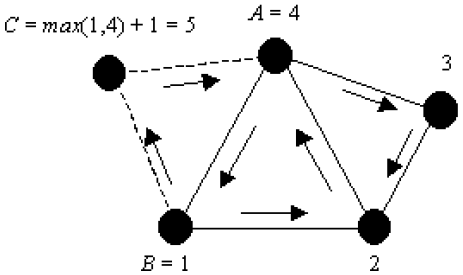


1, 2, 5
2, 5, 4
4, 2, 3

SSS+S++

Sl. 3.4.

Za tok $6 = 3 \oplus 3$ *treba posmatrati slike 3.1 i 3.2 a onda sliku 3.5. U ovom slučaju nema dekompozicije na sledećem nivou jer su oba sabirka jednaka 3. Jedino treba napraviti odgovarajuće trouglove.*

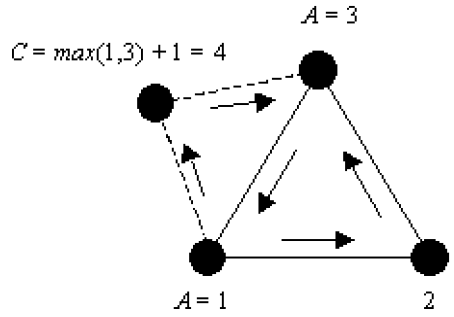


1, 2, 4
2, 4, 3
4, 1, 5

SS+SS++

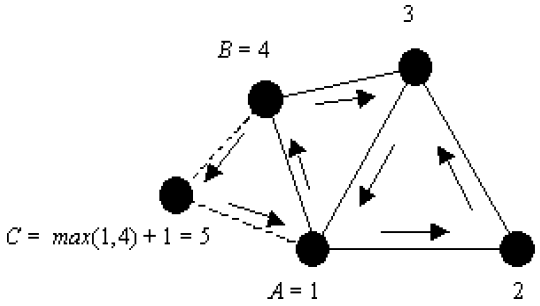
Sl. 3.5.

Za tok $6 = 5 \oplus 1 = (1 \oplus 3)_+ \oplus 1$ *treba posmatrati slike 3.1, 3.6 i 3.7, a za tok* $6 = 5 \oplus 1 = (3 \oplus 1)_+ \oplus 1$ *slike 3.1, 3.7 i 3.8.*



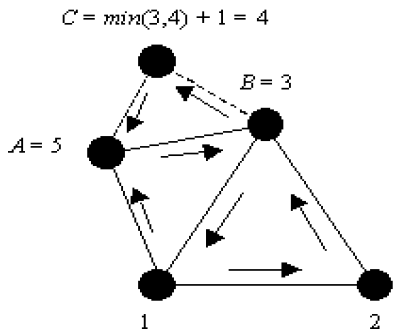
1, 2, 3 SS+S+
3, 1, 4

Sl. 3.6.



1, 2, 3 SSS++S+
1, 4, 3
1, 4, 5

Sl. 3.7.



1, 2, 3 SS+S+S+
1, 5, 3
5, 3, 4

Sl. 3.8.

Izlistani trouglovi se lako mogu sortirati prema oznakama temena a oznake se lako transformišu u velika slova (A, B, C, ...) i na taj način dobijaju liste trouglova na klasičan matematički način.

Na žalost, nije lako implementirati Algoritam 3.1. Zbog toga smo prinudjeni na nešto drugačiji pristup. Naime, najpre treba generisati sve postfiks izraze koji odgovaraju pojedinim triangulacijama, a onda odrediti po jednu triangulaciju za svaki od tih izraza.

Svi izrazi skupa P_n se mogu generisati koristeći rezultat Leme 3.1. i *backtracking* metod. Počinjemo od najvećeg dopuštenog izraza u leksikofrafskom smislu. To je izraz

$$(S)^{n-1}(+)^{n-2}.$$

U svakom sledećem koraku nalazimo prvi dozvoljeni izraz manji u leksikografskom smislu u odnosu na prethodno pronadjeni izraz. Na taj način možemo da pronadjemo sve dozvoljene izraze iz P_n.

Sledeći algoritam koristi prethodno generisanje izraza iz skupa P_n.

Algoritam 3.2.

Ovaj algoritam koristi sličnu ideju kao i Algoritam 3.1. Jedan pojedinačni izraz odgovara jednom toku dekompozicije, a time i jednoj pojedinačnoj triangulaciji. Koristimo zahvat operatora da "rekonstruišemo" višenivovsku dekompoziciju. Svaki zahvat veći od dva ukazuje na složeni argument, tj. dalju dekopoziciju odgovarajućeg celog broja. Potrebno nam je da znamo, kao što je to slučaj i kod Algoritma 3.1, koji od argumenata treba dekomponovati (levi ili desni ili oba). Na osnovu toga generisaćemo trouglove kao i ranije.

Za svaki od izraza iz P_n moramo da izračunamo zahvate njegovih elemenata. Da bi ubrzali naš algoritam izračunavaćemo zahvat nerekurzivno. To je olakšano činjenicom da imamo samo jednu vrstu operatora. U [22] (kao i u drugoj glavi) dat je algoritam za rekurzivno izračunavanje zahvata. U ovom slučaju zahvat se može izračunati pomoću sledećeg koda:

```
void calculate_grasp(exp& array)
{
int i = 0, k;
for (k = 0; k < 50; k++) grasp[k] = 0;
while ( array[i] )
  {
  if ( array[i] == '+' )
      grasp[i] = grasp[i-1]+grasp[i-grasp[i-1]-2]+2;
  i++;
  };
}
```

Primetimo da je `exp` tip polja koje sadrži odgovarajuće izraze i predstavlja niz znakova. U sledećoj tabeli prikazano je ponašanje ovog algoritma.

Tabela 3.2.

n	broj različitih triangulacija	Vreme obrade [s]
13	58786	2
14	208012	11
15	742900	48
16	2674440	195
17	9694845	790
18	35357670	3197

Iz Tabele 3.2 možemo zaključiti da Algoritam 3.2 pokazuje složenost linearno zavisnu od n. Ovo zapažanje je formalno izraženo u sledećoj teoremi.

Teorema 3.3. *Broj rekurzija potrebnih da se generiše pojedinačna triangulacija nekog n-tougla jednak je $n-2$.*

Dokaz. Na osnovu Posledice 3.1. imamo da je zahvat glave odgovarajućeg izraza iz P_n jednak $2n-4$. Drugim rečima, hoćemo da dokažemo da nam je potrebno dvostruko manje rekurzija. Za $n = 3$ zahvat glave odgovarajućeg izraza je dva i potreban nam je samo jedan poziv funkcije da bi generisali jedinstvenu triangulaciju trougla. Neka induktivna pretpostavka važi za svaki mnogougao sa $k < n$ temena i posmatrajmo neki n-tougao. Ako su l_1 i l_2 dužine argumenata za glavu odgovarajućeg izraza, onda je $l_1 + l_2 = 2n - 4$. Da bi se izvršile sve potrebne rekurzije za ove argumente (a oni odgovaraju mnogouglovima sa manje od n temena) potrebno je $(l_1 - 1)/2 = gr_1/2$ i $(l_2 - 1)/2 = gr_2/2$ rekurzivnih poziva respektivno, gde su gr_1 i gr_2 odgovarajući zahvati. Osim toga, potreban je još jedan poziv da bi se ta dva argumenta obradila. To ukupno daje

$$\frac{gr_1}{2} + \frac{gr_2}{2} + 1 = \frac{gr_1 + gr_2 + 2}{2} = \frac{l_1 + l_2}{2} = \frac{2n-4}{2} = n - 2$$

rekurzivnih poziva za kompletiranje jedne triangulacije koja odgovara datom n-touglu. ■

Glava 4

Generalisani inverzi polinomijalnih matrica i interpolacija

Ako je A nesingularna matrica, postoji jedinstvena inverzna matrica A^{-1}, tako da je $AA^{-1} = A^{-1}A = I$. U tom slučaju sistem linearnih jednačina $Ax = y$ ima jedinstveno rešenje $x = A^{-1}y$. Međutim, ako je A singularna ili pravougaona, A^{-1} ne postoji, ali to ne znači da rešenje sistema ne postoji. Čak i kod sistema koji nisu konzistentni, može se posmatrati približno rešenje dobijeno metodom najmanjih kvadrata.

Ove probleme, kao i mnoge druge u numeričkoj algebri, optimizaciji i sistemima upravljanja, teoriji igara i električnih kola, programiranju, statistici, ekonomiji i drugim oblastima, moguće je rešiti uvođenjem generalisanih inverza matrice ili linearnog operatora.

4.1 Osnovne definicije i teoreme

Neka je $\mathbb{C}$ skup kompleksnih brojeva, $\mathbb{C}^{m\times n}$ skup kompleksnih matrica dimenzija $m \times n$ i $\mathbb{C}_r^{m\times n} = \{X \in \mathbb{C}^{m\times n} : \text{rank}(X) = r\}$. Kao što je uobičajeno, $\mathbb{C}[s]$ (respektivno $\mathbb{C}(s)$) označava polinome (respektivno racionalne izraze) sa kompleksnim koeficijentima sa nepoznatom s. Matrice dimenzija $m \times n$ sa elementima u $\mathbb{C}[s]$ (respektivno $\mathbb{C}(s)$) označene su sa $\mathbb{C}[s]^{m\times n}$ (respektivno $\mathbb{C}(s)^{m\times n}$). Sa I_r označavamano jediničnu matricu reda r, a sa $\mathbb{O}$ odgovarajuću nula matricu.

Najpoznatiji uopšteni inverzi su Mur-Penrouzov (Moore-Penrose)i Drazinov.

Poznat je veći broj ekvivalentnih definicija Mur-Penorouzovog inverza. R. Penrouz (Penrose) je 1955. godine dokazao sledeću teoremu [27]:

Teorema 4.1. (Penrouz) *Za datu matricu $A \in \mathbb{C}^{m,n}$ postoji jedinstvena matrica $X \in \mathbb{C}^{n,m}$ koja ispunjava jednačine*

$$(1) \quad AXA = A \qquad (2) \quad XAX = X$$
$$(3) \quad (AX)^* = AX \qquad (4) \quad (XA)^* = XA$$

Penrouz je matricu X označio sa $A^\dagger$ i nazvao je *generalisani inverz matrice* A. On postoji za singularne i pravougaone matrice, a u slučaju regularne matrice je $A^\dagger = A^{-1}$. Za matricu $A^\dagger$ koristi se naziv Mur-Penrouzov inverz matrice A.

Teorema 4.2. (Mur) *Za datu matricu $A \in \mathbb{C}^{m,n}$ postoji jedinstvena matrica $X \in \mathbb{C}^{n\times m}$ tako da za pogodno izabrane matrice Y i Z važi:*

$$AXA = A\,, \qquad X = YA^* = A^*Z.$$

Osim toga je

$$XAX = X\,, \qquad (AX)^* = AX\,, \qquad (XA)^* = XA.$$

Defincija 4.1. [27] (Penrouzova definicija). *Za $A \in \mathbb{C}^{m\times n}$ generalisani inverz je jedinstvena matrica $A^\dagger \in \mathbb{C}^{n\times m}$ koja ispunjava jednačine* (1), (2), (3) *i* (4).

Ako je A kvadratna matrica takođe posmatramo sledeće jednačine:

$$(5) \quad AX = XA, \quad (1^k) \quad A^{k+1}X = A^k.$$

Defincija 4.2. *Za matricu $X = A^D$ kaže se da je Drazinov inverz od A ako su jednačine (1^k) (za neki pozitivni ceo broj k), (2) i (5) ispunjene.*

U [5] dat je algoritam za izračunavanje Mur-Penrouzovog inverza konstantnih kompleksnih matrica $A(s) \equiv A_0 \in \mathbb{C}^{m\times n}$ pomoću Leverije-Fadejevog (Leverrier-Faddeev) algoritma (takodje poznatog i kao Surio-Fram (Souriau-Frame) algoritam). Modifikacija Leverije-Fadejevog algoritma za izračunavanje Drazinovog inverza data je u [7]. Hartvig (Hartwig) u [10] nastavlja proučavanje ovog algoritma.

U [16] je opisana implementacija algoritma za izračunavanje Mur-Penrouzovog inverza singularnih racionalnih matrica u jeziku za simbolička izračunavanja MAPLE. U [17] i [19] koristi se reprezentacija i dva algoritma za izračunavanje Mur-Penrouzovog inverza neregularnih polinomijalnih matrica proizvoljnog reda. Odgovarajući algoritam za matricu polinoma dve promenljive može se naći u [18].

Algoritam za izračunavanje Drazinovog inverza racionalnih matrica predložen je u [31, 32], dok se algoritam za izračunavanje Drazinovog inverza polinomialnih matrica dat

u [15, 33]. Osim ovoga, u literaturi postoji veliki broj primena generalisanih inverza polinomijalnih matrica [16], [17, 18, 19].

Poznato je da je Leverije-Fadejev algoritam slabo uslovljen. Ova činjenica predstavlja motivaciju za korišćenje jezika za simboličko programiranje kao što je MATHEMATICA u [32] i [33].

Simbolička izračunavanja ne pate od grešaka odsecanja i zaokruživanja. Šta više, algoritmi koji se odnose na polinomijalne ili racionalne matrice primenljivi su na znatno širu klasu problema nego što je to slučaj sa algoritmima koji rade se konstantnim matricama. Takodje se ovi algoritmi mogu koristiti za konstrukciju test matrica kao i za verifikaciju nekih hipoteza. Najzad, ovi algoritmi se mogu direktno proveriti na test matricama. Medjutim, poznato je da simboličke implemetacije imaju odredjene nedostatke koji se ogledaju u znatnoj potrošnji vremena i memorijskog prostora. Iz tih razloga, u [15] i [33] predlažu se numerički algoritmi za izračunavanje Drazinovog inverza polinomijalnih matrica.

Motivacija za izradu algoritma koji sledi je sledeća. Do sada su u literaturi poznate različite modifikacije Leverije-Fadejevog algoritma za izračunavanje Mur-Penrouzovog inverza koje se mogu primeniti na kompleksne matrice [5], tj. racionalne ili polinomijalne matrice [16, 17, 18, 19, 34]. Poznate su i slične modifikacije ovog algoritma za izračunavanje Drazinovog pseudoinverza, primenljive na kompleksne matrice [7], odnosno racionalne ili polinomijalne matrice [15, 31, 32, 33]. Takodje se u [29] koristi Lagranžova (Lagrange) interpolacija za izračunavanje običnih matričnih inverza. Mi smo kombinovali ova dva metoda i primenili ih na izračunavanje Drazinovog ili Mur-Penrouzovog inverza monomijalnih matrica.

U narednom odeljku prikazaćemo algoritam za simboličko izračunavanje Mur-Penrouzovog i Drazinovog inverza date monomijalne matrice koji se zasniva na Lagranžovoj interpolacii preradjenoj za slučaj matričnih polinoma i Leverije-Fadejevom algoritmu. Implementacija je uradjena uprogramskom jeziku C++. Implementacija u jeziku MATHEMATICA je jednostavnija zbog mnogih alata za simboličko izračunavanje koji postoje u tom jeziku. Odgovarajuće programe u MATHEMATICI smo koristili radi poredjenja. Kao što ćemo videti, implemetacija u jeziku C++ je mnogo brža.

U sledećim odeljcima dali smo nekoliko ilustrativnih primera kao i odgovarajuća poredjenja.

4.2 Algoritam i implementacija

Osnovna ideja sastoji se u simboličkom izračunavanju Mur-Penrouzovog i Drazinovog inverza polinomijalnih matrica koristeći interpolaciju.

Izračunavaćemo Mur-Penrouzov i Drazinov inverze za nekoliko specifičnih realnih vrednosti promenljive s, a potom ćemo naći matrični interpolacioni polinom koji će nam dati odgovarajući generalisani inverz polinomijalne matrice.

Poredili smo rezultate dobijene ovom metodom sa onim koji se dobijaju implementacijom simboličkog odredjivanja generalisanih inverza na osnovu Leverije-Fadejevog algoritma u jeziku MATHEMATICA.

Algoritam 4.1.

Korak 1. Izaberi n vrednosti $s_1, \ldots, s_n$ promenljive s.

Korak 2. Odredi vrednosti u matrici $A(s)$ obzirom na izabrane vrednosti $s_1, \ldots, s_n$, pri čemu se dobijaju realne matrice $A(s_1), \ldots, A(s_n)$.

Korak 3. Izračunaj "interpolacione tačke"

$$A_k = A^g(s_k),\ k = 0, 1, \ldots, n,$$

gde $A^g(s_k)$ označava $A^\dagger(s_k)$ ili $A^D(s_k)$.

Korak 4. Konstruiši matrični interpolacioni polinom oblika

$$F(s) = \sum_{k=1}^{n} A_k l_k(s), \quad l_k(s) = \prod_{\nu=1,\ \nu\neq k}^{n} \frac{s - s_k}{s_k - s_\nu}$$

Razmotrimo neke detalje implementacije gore navedenih koraka.

Izračunavanje vrednosti $A(s_1), \ldots, A(s_n)$ zasnovano je na inverznoj poljskoj notaciji elemenata matrice $A(s)$. Podsetimo se da su elementi matrice polinomi. Posle konverzije tih polinoma u postfiksni oblilk u stanju smo da nadjemo njihove vrednosti u tačkama $s_1, \ldots, s_n$, koristeći dva magacina: jedan je magacin stringova a drugi je magacin numeričkih vrednosti.

U *Koraku 3*, koristeći Leverije-Fadejev algoritam, izračunavamo "interpolacione tačke", tj. realne matrice A_k.

Najzad, u *Koraku 4*, koristimo algoritam za polinomijalnu interpolaciju (na primer, iz [28]), koji je modifikovan za slučaj matričnih polinoma. Tako dobijamo matrice $C_k \in \mathbb{R}^{n\times m}$ za koje važi

$$\sum_{k=1}^{n} A_k l_k(s) = \sum_{k=0}^{n-1} C_k s^k.$$

Dobijene matrice C_k sadrže koeficijente polinoma koji su elementi traženog inverza. Iza toga se inverzna matrica može lako izraziti u polinomijalnom obliku.

4.3 Primeri

Primer 4.1. *Razmotrimo matricu iz* [38]

$$H_5(s) = \begin{bmatrix} a+1 & a+2 & a+2 & a+3 & a+4 \\ a+2 & a+2 & a+3 & a+4 & a+5 \\ a+2 & a+3 & a+4 & a+5 & a+6 \\ a+3 & a+4 & a+5 & a+5 & a+6 \\ a+4 & a+5 & a+6 & a+6 & a+7 \\ a & a+1 & a+1 & a+2 & a+3 \\ a-1 & a-1 & a-1 & a+1 & a+2 \end{bmatrix}.$$

Koristeći vrednosti $s_i = i, i = 1, 2$, dobijamo sledeći niz matrica koji predstavljaju koeficijente polinoma u Mur-Penrouzovom inverzu $H_5^\dagger(s)$. Primetimo da ≈ 0 znači da je odgovarajuća vrednost približna nuli (reda veličine 10^{-14} ili manjeg):

$$\begin{bmatrix} 0.333333 & 0.666667 & -1 & -0.333333 & 0.666667 & -0.666667 & 0.333333 \\ 1.33333 & -0.833333 & \approx 0 & -0.833333 & 0.666667 & -0.166667 & -0.166667 \\ -1.83333 & 0.333333 & \approx 0 & 1.83333 & -1.16667 & 1.16667 & -0.333333 \\ 1.33333 & -0.833333 & 3 & -2.83333 & 0.666667 & -2.16667 & -0.166667 \\ -0.666667 & 0.666667 & -2 & 1.66667 & -0.333333 & 1.33333 & 0.333333 \end{bmatrix},$$

$$\begin{bmatrix} \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & 0 \\ \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 \\ \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 & \approx 0 \\ 0.5 & \approx 0 & \approx 0 & -0.5 & 0.5 & -0.5 & \approx 0 \\ -0.5 & \approx 0 & 0 & 0.5 & -0.5 & 0.5 & \approx 0 \end{bmatrix}.$$

U [38] *tačne vrednosti pseudoinverzne matrice su:*

$$H_5^\dagger(s) = \frac{1}{12} \begin{bmatrix} 4 & 8 & -12 & -4 & 8 & -8 & 4 \\ 16 & -10 & 0 & -10 & 8 & -2 & -2 \\ -22 & 4 & 0 & 22 & -14 & 14 & -4 \\ 6a+16 & -10 & 36 & -6a-34 & 6a+8 & -6a-26 & -2 \\ -6a-8 & 8 & -24 & 6a+20 & -6a-4 & 6a+16 & 4 \end{bmatrix}.$$

Ako smatramo ekstremno male vrednosti (reda 10^{-14} i manjeg) nulama, može se zaključiti da je rezultat tačan (u okviru izvesne greške zaokruživanja).

Primer 4.2. *Posmatrajmo još jednu matricu iz* [38]

$$F_4(s) = \begin{bmatrix} s+4 & s+3 & s+2 & s+1 \\ s+3 & s+3 & s+2 & s+1 \\ s+2 & s+2 & s+1 & s \\ s+1 & s+1 & s & s-1 \end{bmatrix}.$$

Za vrednosti $s_i = i, i = 1, 2$, dobijamo sledeći niz matrica koje su koeficijenti polinoma u Mur-Penrouzovom inverzu $F_4^\dagger(s)$:

$$\begin{bmatrix} 1 & -0.833333 & -0.333333 & 0.166667 \\ -0.833333 & 0.777778 & 0.444444 & 0.111111 \\ -0.333333 & 0.444444 & 0.111111 & -0.222222 \\ 0.166667 & 0.111111 & -0.222222 & -0.555556 \end{bmatrix},$$

$$\begin{bmatrix} \approx 0 & \approx 0 & \approx 0 & \approx 0 \\ \approx 0 & -0.25 & \approx 0 & 0.25 \\ \approx 0 & \approx 0 & \approx 0 & 0 \\ \approx 0 & 0.25 & \approx 0 & -0.25 \end{bmatrix}.$$

Tačna pseudoinverzna matrica je ([38]*):*

$$F_4^\dagger(s) = \begin{bmatrix} 1 & -\frac{5}{6} & -\frac{1}{3} & \frac{1}{6} \\ -\frac{5}{6} & \frac{7}{9} - \frac{s}{4} & \frac{4}{9} & \frac{1}{9} + \frac{s}{4} \\ -\frac{1}{3} & \frac{4}{9} & \frac{1}{9} & -\frac{2}{9} \\ \frac{1}{6} & \frac{1}{9} + \frac{s}{4} & -\frac{2}{9} & -\frac{5}{9} - \frac{s}{4} \end{bmatrix}.$$

Primer 4.3. *Razmotrimo singularnu kvadratnu matricu iz* [38] *za $n = 7$.*

$$S_7(a) = \begin{bmatrix} a+1 & a & a & a & a & a & a+1 \\ a & a-1 & a & a & a & a & a \\ a & a & a+1 & a & a & a & a \\ a & a & a & a-1 & a & a & a \\ a & a & a & a & a+1 & a & a \\ a & a & a & a & a & a-1 & a \\ a+1 & a & a & a & a & a & a+1 \end{bmatrix}.$$

Za $s_i = i, i = 1, 2$, dobijamo sledeći niz matrica koje su koeficijenti polinoma u Mur-Penrouzovom inverzu $S_7^\dagger(a)$:

$$\begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 & 0 & 0.25 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0.25 & 0 & 0 & 0 & 0 & 0 & 0.25 \end{bmatrix},$$

$$\begin{bmatrix} -0.25 & 0.5 & -0.5 & 0.5 & -0.5 & 0.5 & -0.25 \\ 0.5 & -1 & 1 & -1 & 1 & -1 & 0.5 \\ -0.5 & 1 & -1 & 1 & -1 & 1 & -0.5 \\ 0.5 & -1 & 1 & -1 & 1 & -1 & 0.5 \\ -0.5 & 1 & -1 & 1 & -1 & 1 & -0.5 \\ 0.5 & -1 & 1 & -1 & 1 & -1 & 0.5 \\ -0.25 & 0.5 & -0.5 & 0.5 & -0.5 & 0.5 & -0.25 \end{bmatrix}.$$

U [38] *tačna pseudoinverzna matrica je:*

$$S_7^\dagger(a) = \frac{1}{4}\begin{bmatrix} -a+1 & 2a & -2a & 2a & -2a & 2a & -a+1 \\ 2a & -4a-4 & 4a & -4a & 4a & -4a & 2a \\ -2a & 4a & -4a+4 & 4a & -4a & 4a & -2a \\ 2a & -4a & -4a & -4a-4 & 4a & -4a & 2a \\ -2a & 4a & -4a & 4a & -4a+4 & 4a & -2a \\ 2a & -4a & -4a & 4a & -4a & -4a-4 & 2a \\ -a+1 & 2a & -2a & 2a & -2a & 2a & -a+1 \end{bmatrix}.$$

Primer 4.4. *Najzad, za matricu A_2 ranga dva i dimenzija 7×6 iz* [38]

$$A_2 = \begin{bmatrix} a+10 & a+9 & a+8 & a+7 & a+6 & a+5 \\ a+9 & a+8 & a+7 & a+6 & a+5 & a+4 \\ a+8 & a+7 & a+6 & a+5 & a+4 & a+3 \\ a+7 & a+6 & a+5 & a+4 & a+3 & a+2 \\ a+6 & a+5 & a+4 & a+3 & a+2 & a+1 \\ a+5 & a+4 & a+3 & a+2 & a+1 & a \\ a+4 & a+3 & a+2 & a+1 & a & a-1 \end{bmatrix}.$$

dobijamo

$$\begin{bmatrix} -0.0306122 & -0.0136054 & 0.00340136 & 0.0204082 & 0.037415 & 0.0544218 & 0.0714286 \\ -0.0112245 & -0.00340136 & 0.00442177 & 0.0122449 & 0.020068 & 0.0278912 & 0.0357143 \\ 0.00816327 & 0.00680272 & 0.00544218 & 0.00408163 & 0.00272109 & 0.00136054 & \approx 0 \\ 0.027551 & 0.0170068 & 0.00646259 & -0.00408163 & -0.0146259 & -0.0251701 & -0.0357143 \\ 0.0469388 & 0.0272109 & 0.00748299 & -0.0122449 & -0.0319728 & -0.0517007 & -0.0714286 \\ 0.0663265 & 0.037415 & 0.0085034 & -0.0204082 & -0.0493197 & -0.0782313 & -0.107143 \end{bmatrix},$$

$$\begin{bmatrix} -0.0153061 & -0.0102041 & -0.00510204 & \approx 0 & 0.00510204 & 0.0102041 & 0.0153061 \\ -0.00918367 & -0.00612245 & -0.00306122 & \approx 0 & 0.00306122 & 0.00612245 & 0.00918367 \\ -0.00306122 & -0.00204082 & -0.00102041 & \approx 0 & 0.00102041 & 0.00204082 & 0.00306122 \\ 0.00306122 & 0.00204082 & 0.00102041 & \approx 0 & -0.00102041 & -0.00204082 & -0.00306122 \\ 0.00918367 & 0.00612245 & 0.00306122 & \approx 0 & -0.00306122 & -0.00612245 & -0.00918367 \\ 0.0153061 & 0.0102041 & 0.00510204 & \approx 0 & -0.00510204 & -0.0102041 & -0.0153061 \end{bmatrix}.$$

U [38] *tačna pseudoinverzna matrica je:*

$$A_2^{\dagger} = \frac{1}{20580} \times$$

$$\begin{bmatrix} -315a-630 & -210a-280 & -105a+70 & 420 & 105a+770 & 210a+1120 & 315a+1470 \\ -189a-231 & -126a-70 & -63a+91 & 252 & 63a+413 & 126a+574 & 189a+735 \\ -63a-168 & -42a+140 & -21a+112 & 84 & 21a+56 & 42a+28 & 63a \\ 63a+567 & 42a+350 & 21a+133 & -84 & -21a-301 & -42a-518 & -63a-735 \\ 189a+966 & 126a+560 & 63a+154 & -252 & -63a-658 & -126a-1064 & -189a-1470 \\ 315a+1365 & 210a+770 & 105a+175 & -420 & -105a-1015 & -210a-1610 & -315a-2205 \end{bmatrix}.$$

4.4 Poredjenja

Programski paket MATHEMATICA ima mogućnost izračunavanja generalisanih inverza. U sledećoj tabeli prikazano je ponašanje našeg algoritma u odnosu na funkciju iz jezika MATHEMATICA na osnovu vremena izvršenja (u sekundama) prilikom izračunavanja inverza matrice S_n.

Tabela 4.1

n	Naš algoritam	Funkcija iz MATHEMATICE
70	1	10.391
75	2	12.859
80	3	15.188
85	3	18.640
90	3	21.625

Glava 5

Zaključak

Metod inverzne poljske notacije opisan u ovoj disertaciji omogućuje obavljanje simboličkih manipulacija nad različitm tipovima izraza. Pri tome su izbegnute dinamičke strukture podataka, kao što su povezane liste ili stabla, već se manipulacija obavlja direktno nad statičkim nizovima koji predstavljaju izraze u postfiksnoj notaciji.

Svakako da dinamičke strukture podataka omogućiju efikasno korišćenje memorije, ali smo "premošćavanjem" ovih faza dobili na jednostavnosti programiranja i brzini izvršenja. Problem utroška memorije, karakterističan za simbolička izračunavanja, uglavnom je uspešno prebrodjen.

Primenom ovog metoda na čitav niz problema pokazali smo njegovu opštost. Naravno, uvek postoje problemi kod kojih je primena ovakve metode uspešnija, kao i oni koji nisu naročito pogodni za njenu primenu.

Simboličko diferenciranje je, kao što je u uvodu rečeno, bio problem koji je iznedrio ovu metodu. Posebno uspešna je, po našem mišljenju, bila primena na izračunavanje parfunkcija, obzirom da mali broj operatora i jednostavni argumenti pogoduju ovoj metodi. Takodje je interesantan način na koji je metoda inverzne poljske notacije upotrebljena za konstrukciju algoritma za triangulaciju koji je u suštini više numerički nego simbolički.

U četvrtoj glavi videli smo nešto drugačiju primenu metode inverzne poljske notacije. Slična ideja je korišćena u radu [29] za izračunavanje običnih inverza polinomijalnih nesingularnih matrica. Polazeći od ove ideje i udružujući je sa Leverije-Fadejevim algoritmom, kao i našom metodom, došli smo do algoritma za izračunavanje generalisanih inverza monomijalnih matrica. Pri tome je ulaz u program matrica u simboličkom obliku i koja se evaluira za izabrane vrednosti nezavisno promenljive, kao što je to već opisano.

Literatura

[1] Abelson, H. and Sussman, G.J., *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts, 1985.

[2] Corman, T. H., Leiserson, C. E. and Rivest, R. L., *Introduction to Algorithms*, MIT Press, Cambridge, Massachusets, London, England, 1990.

[3] Cutland, N., *Computability: An Introduction to Recursive Functions Theory*, Cambridge University Press, Cambridge, London, Newyork, Sydney, 1986.

[4] Davies, M. D. and Weyuker, E. J., *Computability, Complexity, and Languages*, Academic Press, New York, London, Paris, 1983.

[5] Decell, H. P., *An application of the Cayley-Hamilton theorem to generalized matrix inversion*, SIAM Review, **7**, No. 4, (1965), pp. 526–528.

[6] Flanders, H.,*Automatic Differentiation of Composite Functions*, Automatic Differentiation of Algorithms: Theory, Implementation, and Application, Proceedings of the first SIAM Workshop on Automatic Differentiation (Andreas Griewank and George F. Corliss, eds.), Breckenridge, Colorado, January 6–8, 1991, pp. 95–99.

[7] Grevile, T. N. E., *The Souriau-Frame algorithm and the Drazin pseudoinverse*, Linear Algebra Appl. **6** (1973), pp. 205–208.

[8] Gries, D., *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, London, Sydney, Toronto, 1971.

[9] Hanson, J. W., Caviness, J. S. and Joseph, C., *Analytic Differnetiation By Computer*, Communications of the ACM, vol. 5, June, 1962, pp. 349–355.

[10] Hartwig, R. E., *More on the Souriau-Frame algorithm and the Drazin inverse*, SIAM J. Appl. Math., **31**, No. 1, (1976), pp. 42–46.

[11] Henessey, L.W., *Common LISP*, McGraw-Hill Book Company, 1989.

[12] Hyvönen, E. and Seppänen, J., *Introduction to LISP and functional programming*, Moskva, "Mir", 1990. (In Russian)

[13] Iri, M., *History of Automatic Differentiation and Rounding Error Estimation*, Automatic Differentiation of Algorithms: Theory, Implementation, and Application, Proceedings of the first SIAM Workshop on Automatic Differentiation, (Andreas Griewank and George F. Corliss, eds.), Breckenridge, Colorado, January 6–8, 1991, pp. 3–16.

[14] Iri, M. and Kubota, K., *Norms, Rounding Errors, Partial Derivatives and Fast Automatic Differentiation*, IEICE transactions, vol. E 74, no. 3, March 1991, pp. 463–471.

[15] Ji, J., *A finite algorithm for the Drazin inverse of a polynomial matrix*, Appl. Math. Comput. (to appear).

[16] Jones, J., Karampetakis, N. P. and Pugh, A. C., *The computation and application of the generalized inverse vai Maple*, J. Symbolic Computation **25** (1998), pp. 99–124.

[17] Karampetakis, N. P., *Computation of the generalized inverse of a polynomial matrix and applications*, Linear Algebra Appl. **252** (1997), pp. 35–60.

[18] Karampetakis, N. P., *Generalized inverses of two-variable polynomial matrices and applications*, Circuits Systems Signal Processing **16** (1997), pp. 439–453.

[19] Karampetakis, N. P. and Tzekis, P., *On the computation of the generalized inverse of a polynomial matrix*, Ima Journal of Mathematical Control and Information **18** (2001), pp. 83–97.

[20] Kross, M. and Lentin, A., *Notions sur les grammaries formelles*, Gauthier-Villars, 1967.

[21] Krtolica, P. V. and Stanković, M. S., *QADE – Program for Qualitative Analysis of Differential Equations*, Proc. of II Math. Conf. in Priština 1996 (Lj. D. Kočinac, ed.), Priština, 1997, pp. 229–243.

[22] Krtolica, P. V. and Stanimirović, P. S., *On Some Properties of Reverse Polish Notation*, FILOMAT, vol. 13, 1999, pp. 157–172.

[23] Krtolica, P. V. and Stanimirović, P. S., *Symbolic Derivation Without Using Expression Trees*, YUJOR, vol. 11, 2001, pp. 61–75.

[24] Krtolica, P. V. and Stanimirović, P. S., *Deducing about the Necessity of the Parenthesis*, FILOMAT, vol. 14, 2000, pp. 87–93.

[25] Parker, T. S. and Chua, L. O., *INSITE – A Software Toolkit for the Analysis of Nonlinear Dynamical Systems*, Proc. IEEE, vol. 75, pp. 1081–1089, August 1987.

[26] Parker, T. S. and Chua, L. O., *Practical Numerical Algorithms for Chaotic Systems*, Springer-Verlag, New York, 1989.

[27] Penrose, R., *A generalized inverse for matrices*, Proc. Cambridge Philos. Soc.,**51** (1955), 406–413

[28] Press, W. H., Teukolsky, S. A., Wetterling, W. T. and Flannery, B. P., *Numerical receipts in C*, Cambridge University Press, Cambridge (MA), 1992.

[29] Schuster, A. and Hippe, P., *Inversion of Polynomial Matrices by Interpolation*, IEEE Transaction on Automatic Control, Vol. 37, No. 3, (March 1992), pp. 363–365.

[30] Sedgewick, R., *Algorithms in C*, Addison-Wesley Publishing Company, Reading, MA, 1990.

[31] Stanimirović, P. S. and Karampetakis, N. P., *Symbolic implementation of Leverrier-Faddeev algorithm and applications*, 8th IEEE Medit. Conference on Control and Automation, Patra, Greece, 2000.

[32] Stanimirović, P. S. and Karampetakis, N. P., *On the computation of Drazin inverse of a rational matrix and applications*, Technical Report, Department of Mathematics, Aristotle University of Thessaloniki, Thessaloniki 54006, Greece, 2000.

[33] Stanimirović, P. S. and Tasić, M. B., *Drazin inverse of one-variable polynomial matrices*, Filomat **15** (2001), pp. 71–78.

[34] Stanimirović, P.S. *A finite algorithm for generalized inverses of polynomial and rational matrices*, Appl. Math. Comput., (to appear).

[35] Tanenbaum, A. S., *Structured Computer Organization*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[36] Tremblay, J.-P. and Sorenson, P.G., *The Theory and Practice of Compiler Writing*, McGraw-Hill Book Company, New York, 1985.

[37] Wald, B., *The Semigroup of Unary Pairfunctions, Verification of the Group Relations*, MapleTech, vol. 4, 1997., pp. 51–54.

[38] Zielke, G. *Report on test matrices for generalized inverses*, Computing **36** (1986), pp. 105–162.

[39] W. H. Press, S. A. Teukolsky, W. T. Wetterling, B. P. Flannery, *Numerical receipts in C*, Cambridge University Press, Cambridge (MA), 1992.