Konstrukcija kompilatora - beleške sa predavanja

Contents

1 Kratak opis procesa kompilacije

Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika. Generalno, *leksika* se bavi opisivanjem osnovnih gradivnih elemenata jezika (ključnih reči, identifikatora, brojevnih konstanti, niski, …), a *sintaksa* načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije (izraze, naredbe, deklaracije, definicije, …). Pitanjem značenja ispravnih jezičkih konstrukcija bavi se *semantika*. Na primer, leksika jezika C definiše da je `if` ključna reč jezika, a da je `x` identifikator (ime promenljive), sintaksa definiše da se jedna naredba ispravno zapisuje tako što se iza ključne reči `if` u malim zagradama navede neki izraz koji ima logičku (tj. brojevnu) vrednost, a zatim neka druga naredba ili niz naredbi naveden u vitičastim zagraama, a semantika definiše da je ta naredba naredba grananja čiji je efekat to da se naredba ili niz naredbi navedeni iza uslova izvrše samo ako taj uslov ima vrednost tačno (tj. brojevnu vrednost različitu od nule).

Kompilatori su veoma složen softver. Kao što je obično slučaj u računarstvu složenost sistema se savlađuje podelom na manje celine koje imaju jasno definisane zadatke i međusobno sarađuju preko jasno definisanih interfejsa. Tako su kompilatori obično podeljeni na sledeće module.

- *Prednji deo* (engl. *front-end*), tj. *analiza* koja čita program zapisan na višem programskom jeziku, obrađuje ga i pohranjuje u obliku međureprezentacije tj. međukoda. Prednji deo se sastoji od sledećih komponenti:

  - Leksički analizator;
  - Sintaksički analizator;
  - Semantički analizator;
  - Generator međukoda.
- *Zadnji deo* (engl. *back-end*), ili *sinteza* koja prevodi međureprezentaciju (međukod) u ciljni jezik (najčešće, ali ne nužno mašinski jezik), moguće koristeći razne optimizacije. Zadnji deo se sastoji od sledećih komponenti:
  - Optimizator međukoda;
  - Generator ciljnog koda.

1.1 Leksička analiza

Osnovni leksički elementi prirodnih jezika su reči, pri čemu se razlikuje nekoliko različitih vrsta reči (imenice, glagoli, pridevi, ...). Zadatak leksičke analize prirodnog jezika je da identifikuje reči u rečenici i svrsta ih u odgovarajuće kategorije. Slično važi i za programske jezike. Programi se računaru zadaju predstavljeni nizom karaktera. Na primer, kod:

```
if (v < 120)
   s = s0 + v*t;
```

je, zapravo, samo niz karaktera: `if_(v < 120)\n\ts = s0 + v*t;\n`.

Pojedinačni karakteri se grupišu u nedeljive celine *lekseme* koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika i pridružuju im se *tokeni* koji opisuju leksičke kategorije kojima te lekseme pripadaju. Token je sintaksička klasa, kategorija, kao što je u prirodnom jeziku, na primer, kategorija imenica, glagola ili priloga. U programskom jeziku klase su identifikator, brojevna konstanta (celobrojna ili u pokretnom zarezu), matematički operator, itd. Leksema je konkretan primerak, konkretna instanca jednog tokena. Na primer, za token `identifikator`, primeri leksema su `a`, `A1`, `prvi_sabirak`, itd.

U prethodnom kodu, razlikuju se sledeće lekseme (reči) i njima pridruženi tokeni (kategorije).

leksema	token
`if`	ključna reč
`(`	zagrada
`v`	identifikator
`<`	operator
`120`	celobrojni literal
`)`	zagrada
`s`	identifikator
`=`	operator
`s0`	identifikator
`+`	operator

leksema	token
`v`	identifikator
`*`	operator
`t`	identifikator
`;`	interpunkcija

Dakle, leksička analiza je proces izdvajanja *leksema* i *tokena*, osnovnih jezičkih elemenata, iz niza ulaznih karaktera (na primer, karaktera koji čine program). Leksičku analizu vrše moduli kompilatora koji se nazivaju *leksički analizatori* (lekseri, skeneri). Oni obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom modulu (sintaksičkom analizatoru) koji nastavlja analizu teksta programa. Zapravo, leksički analizator najčešće radi na zahtev sintaksičkog analizatora tako što se sintaksički analizator obraća leksičkom analizatoru kada god mu zatreba naredni token.

Tokenima mogu biti pridruženi i neki dodatni *atributi*. Na primer, svakom tokenu `identifikator` može biti pridružen pokazivač na mesto u specijalnoj *tabeli simbola* koja sadrži informacije o pojedinačnim identifikatorima (npr. ime identifikatora, njegov tip, lokaciju u kodu na kome je deklarisan i slično), tokenu `brojevna_konstanta` može biti pridružena njena numerička vrednost i slično.

1.1.1 Regularni izrazi i konačni automati

Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole i slično. Ti obrasci za opisivanje tokena su obično *regularni izrazi*, a mehanizam za izdvajanje leksema iz ulaznog teksta zasniva se na *konačnim automatima*.

Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su: - *karakterske klase* - navode se između `[` i `]` i označavaju jedan od navedenih karaktera. Na primer, klasa `[0-9]` označava cifru. - *alternacija* - navodi se sa `|` i označava alternativne mogućnosti. Na primer, `a|b` označava slovo `a` ili slovo `b`. - *opciono pojavljivanje* - navodi se sa `?`. Npr. `a?` označava da slovo `a` može, a ne mora da se javi. - *ponavljanje* - navodi se sa `*` i označava da se nešto javlja nula ili više puta. Npr. `a*` označava niz od nula ili više slova `a`. - *pozitivno ponavljanje* - navodi se sa `+` i označava da se nešto javlja jedan ili više puta. Npr. `[0-9]+` označava neprazan niz cifara.

Razmotrimo identifikatore u programskom jeziku C. Govornim jezikom, identifikatore je moguće opisati kao "neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom". Ovo znači da je prvi karakter identifikatora ili slovo ili podvlaka, za čim sledi nula ili više slova, cifara ili podvlaka. Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:

`([a-zA-Z]|_)([a-zA-Z]|_|[0-9])*`

Ovaj izraz je moguće zapisati još jednostavnije kao:

`[a-zA-Z_][a-zA-Z_0-9]*`

Lex je program koji generiše leksere na programskom jeziku C, a na osnovu zadatog opisa tokena u obliku regularnih izraza. Na primer, jedan jednostavan opis leksičkog analizatora koji prepoznaje identifikatore, realne konstante, operatore i zagrade može biti sledeći.

```
%%
[a-zA-Z_][a-zA-Z_0-9]*   return IDENTIFIKATOR;
[+-]?([0-9]*[.])?[0-9]+  return REALNA_KONSTANTA;
[-+*/()]                 return *yytext;
[ \t\n]+                 /* beline se preskaču */
%%
```

1.2 Sintaksička analiza

Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica razmatraju ispravni programi. Sintaksa definiše formalne relacije između elemenata jezika, time pružajući strukturne opise ispravnih niski jezika. Sintaksa se bavi samo formom i strukturom jezika bez bilo kakvih razmatranja u vezi sa njihovim značenjem. Sintaksička struktura rečenica ili programa se može predstaviti u obliku stabla koje predstavlja međusobni odnos pojedinačnih tokena. Prikazani fragment koda je u jeziku C sintaksički ispravan i njegova sintaksička struktura se može predstaviti na sledeći način:

Dakle, taj fragment koda predstavlja `if-then` naredbu (iskaz) kojoj je uslov dat izrazom poređenja vrednosti promenljive `a` i konstante `3`, a telo predstavlja naredba dodele promenljivoj `x1` vrednosti izraza koji predstavlja zbir konstante `3` i proizvoda konstante `4` i vrednosti promenljive `a`.

Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva *sintaksički analizator* ili *parser*. Rezultat njegovog rada se najčeše isporučuje daljim fazama u obliku *sintakstičkog stabla* (kaže se i *stablo apstraktne sintakse* i *apstraktno sintakstičko stablo*).

1.2.1 Kontekstno slobodne gramatike

Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (npr. skup svih identifikatora, skup brojevnih literala, i slično). Međutim nije moguće konstruisati regularne izraze kojim bi se opisale neke konstrukcije koje se javljaju u programskim jezicima. Tako, na primer, nije moguće

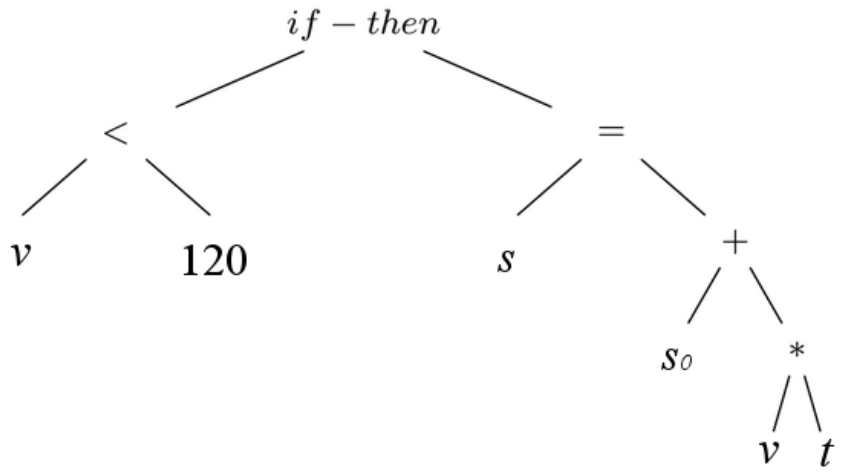


Figure 1: Sintaksičko stablo

regularnim izrazom opisati skup reči $\{a^n b^n \mid n > 0\} = \{ab, aabb, aaabbb, ...\}$. Takođe, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetički izrazi, tj. skup $\{a, a+a, a*a, a+a*a, a*(a+a), ...\}$.

Sintaksa jezika se obično opisuje gramatikama. U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći govorni jezik kao metajezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom *kontekstno-slobodne gramatike* (engl. *context free grammars*).

Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza. Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno-slobodnim gramatikama, tako da je kontekstno-slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (doduše regularni izrazi obično daju koncizniji opis).

Kontekstno-slobodne gramatike su određene skupom pravila. Sa leve strane pravila nalaze se tzv. pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako ju je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Pokazano je da je jezik identifikatora programskog jezika C regularan i da ga je moguće opisati regularnim izrazom. Sa druge strane, isti ovaj jezik je moguće

opisati i formalnom gramatikom. Razmotrimo naredna pravila (simbol ε označava praznu reč):

$$\begin{aligned} I &\to XZ \\ X &\to S \mid P \\ Z &\to YZ \mid \varepsilon \\ Y &\to S \mid P \mid C \\ S &\to \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\ P &\to \_ \\ C &\to \texttt{0} \mid \ldots \mid \texttt{9} \end{aligned}$$

Neterminal S odgovara slovima, neterminal P podvlaci, neterminal C ciframa, neterminal X slovu ili podvlaci, neterminal Y slovu, podvlaci ili cifri, a neterminal Z nizu simbola koji se mogu izvesti iz Y tj. nizu slova, podvlaka ili cifara.

Identifikator `x_1` je moguće izvesti kao

$$I \Rightarrow XZ \Rightarrow SZ \Rightarrow \texttt{x}Z \Rightarrow \texttt{x}YZ \Rightarrow \texttt{x}PZ \Rightarrow \texttt{x\_}Z \Rightarrow \texttt{x\_}YZ \Rightarrow \texttt{x\_}CZ \Rightarrow \texttt{x\_1}Z \Rightarrow \texttt{x\_1}.$$

Neka je gramatika određena skupom pravila:

$$\begin{aligned} E &\to E + E \\ &\mid E * E \\ &\mid (E) \\ &\mid a. \end{aligned}$$

Ova gramatika opisuje ispravne aritmetičke izraze u kojima su dopuštene operacije sabiranja i množenja. Npr. izraz $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

Međutim, isti izraz je moguće izvesti i kao

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

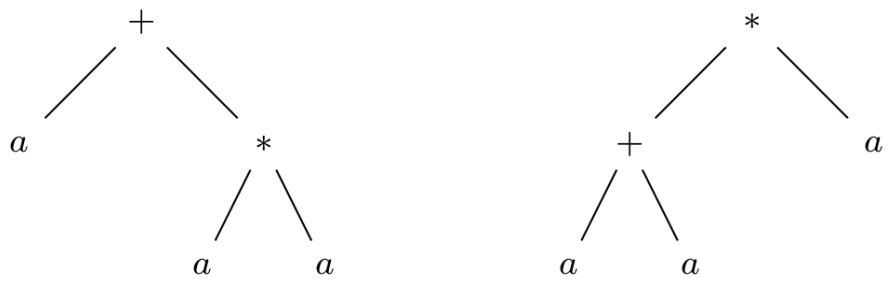


Figure 2: Dva drveta koja odgovaraju različitom prioritetu operatora

Prvo izvođenje odgovara levo, a drugo desno prikazanom sintaksičkom stablu:

Problem sa prethdnom gramatikom je to što se u njoj ne oslikava prioritet (niti asocijativnost) operatora i stoga se takve gramatike ne mogu direktno koristiti u sintaksičkoj analizi (pre njihovog korišćenja potrebno je na neki način precizirati prioritet i asocijativnost operatora).

Prethodni jezik možemo opisati i drugačijom gramatikom. Neka je gramatika zadata sledećim pravilima:

$$\begin{array}{rcl} E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (E) \\ & | & a \end{array}$$

I ova gramatika opisuje ispravne aritmetičke izraze. Na primer, niska $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E{+}T \Rightarrow T{+}T \Rightarrow F{+}T \Rightarrow a{+}T \Rightarrow a{+}T{*}F \Rightarrow a{+}F{*}F \Rightarrow a{+}a{*}F \Rightarrow a{+}a{*}a.$$

Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

Primetimo da je ova gramatika u određenom smislu preciznija od gramatike date u prethodnom primeru, s obzirom da je njome jednoznačno određen prioritet i asocijativnost operatora, što sa prethodnom gramatikom nije bio slučaj.

Kontekstno-slobodne gramatike se obično formiraju nad tokenima iz leksičke analize. Tako bi pravila za F kao izlaz iz rekurzije obuhvatala promenljive (npr. opisane tokenom *id*) i brojevne konstante (npr. opisane tokenom *num*). Za ulaznu nisku `s0 + v * t` leksički analizator bi prepoznao tokene `id + id * id` i to bi bilo prepoznato kao ispravan arimtetički izraz na osnovu gore navedene gramatike.

Kontekstno-slobodne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U kontekstno-slobodnim gramatikama sa leve strane pravila nalazi se uvek tačno jedan neterminalni simbol, a u opštem slučaju sa leve strane pravila može se nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

Na osnovu gramatike jezika formira se *potisni automat* na osnovu kojeg se jednostavno implementira program koji vrši sintakstičku analizu. Formiranje automata od gramatike se obično vrši automatski, uz korišćenje tzv. *generatora parsera* (engl. *parser generator*), poput sistema Yacc, Bison ili Antlr.

1.2.2 BNF, EBNF, Sintaksički dijagrami

Za opis sintakse koriste se i određene varijacije kontekstno-slobodnih gramatika. Najpoznatije od njih su *BNF (Bakus-Naurova forma)*, *EBNF (proširena Bakus-Naurova forma)* i *sintaksički dijagrami*.

BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksički dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok se BNF može veoma jednostavno objasniti, precizna definicija EBNF zahteva malo više rada i ona je data kroz ISO 14977 standard.

1.2.2.1 BNF

Metajezik pogodan za zapis pravila kontekstno-slobodnih gramatika je *BNF*. Prvu verziju jezika kreirao je Džon Bekus, a ubrzo zatim poboljšao je Piter Naur i ta poboljšana verzija je po prvi put upotrebljena da se formalno definiše sintaksa programskog jezika Algol 60. BNF je u početku označavala skraćenicu od *Bekusova normalna forma* (engl. *Backus Normal Form*), međutim na predlog Donalda Knuta, a kako bi se naglasio doprinos Naura, ime je izmenjeno u *Bekus-Naurova forma* (engl. *Backus Naur Form*) (čime je i jasno naglašeno da BNF nije normalna forma u smislu normalnih formi gramatika Čomskog).

U BNF notaciji, sintaksa objektnog jezika se opisuje pomoću konačnog skupa *metalingvističkih formula (MLF)* koje direktno odgovaraju pravilima kontekstno-slobodnih gramatika.

Svaka metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, tzv. "univerzalnim" metasimbolom (simbolom metajezika koji se koristi u svim MLF) `::=` koji se čita "po definiciji je", tj. MLF je oblika `A ::= a`, gde

je `A` metalingvistička promenljiva, a `a` metalingvistički izraz. Metalingvističke promenljive su fraze prirodnog jezika u uglastim zagradama (`<`, `>`), i one predstavljaju pojmove, tj. sintaksičke kategorije objektnog jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. U programskom jeziku, sintaksičke kategorije su, na primer, `<program>`, `<ceo broj>` i `<identifikator>`. U prirodnom jeziku, sintaksičke kategorije su, na primer, `<rec>` i `<recenica>`. Metalingvističke promenljive ne pripadaju objektnom jeziku. U nekim knjigama se umesto uglastih zagrada `<` i `>` metalingvističke promenljive označavaju korišćenjem podebljanih slova.

Metalingvističke konstante su simboli objektnog jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, npr. `if`, `then`, `begin`, `for`, itd.

Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksičkih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objektnom jeziku.

Metalingvistički izrazi se grade od metalingvističkih promenljivih i metalingvističkih konstanti primenom operacija konkatenacije i alternacije (`|`).

Metalingvistička formula `A ::= a` ima značenje: ML promenljiva `A` po definiciji je ML izraz `a`. Svaka metalingvistička promenljiva koja se pojavljuje u metalingvističkom izrazu `a` mora se definisati posebnom MLF.

Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:

```
<ceo broj> ::= <neoznacen ceo broj> | <znak broja> <neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra> |  <neoznacen ceo broj><cifra>
<cifra> ::= 0|1|2|3|4|5|6|7|8|9
<znak broja> ::= +|-
```

Gramatika aritmetičkih izraza može u BNF da se zapiše kao: ~~~~ ::= + | ::= * | ::= ( ) | ~~~~

1.2.2.2 EBNF, ISO 149777

EBNF, je skraćenica od *Extend Backus-Naur Form* tj. *proširena Bakus-Naurova forma*. Ovaj formalizam ne omogućava da se definiše bilo šta što nije moguće definisati korišćenjem BNF, ali uvodi skraćene zapise koji olakšavaju zapis gramatike i čine je čitljivijom. Nakon korišćenja BNF za definisanje sintakse Algola 60, pojavile su se mnoge njene modifikacije i proširenja.

EBNF proširuje BNF nekim elementima regularnih izraza:

- Vitičaste zagrade `{...}` okružuju elemente izraza koji se mogu ponavljati proizvoljan broj (nula ili više) puta. Alternativno, moguće je korišćenje i sufiksa `*`.

- Uglaste zagrade `[...]` okružuju opcione elemente u izrazima, tj. elemente koji se mogu pojaviti nula ili jedan put. Alternativno, moguće je korišćenje i sufiksa `?`.
- Sufiks `+` označava elemente izraza koji se mogu pojaviti jednom ili više puta.
- Obične male zagrade se koriste za grupisanje.

Svi ovi konstrukti se mogu izraziti i u BNF metajeziku, ali uz korišćenje znatno većeg broja MLF, što narušava čitljivost i jasnost (na primer, u BNF metajeziku opis izraza koji se ponavljaju odgovara rekurziji, a u EBNF metajeziku iteriranju). Izražajnost i BNF metajezika i EBNF metajezika jednaka je izražajnosti kontekst slobodnih gramatika, tj. sva tri ova formalizma mogu da opišu isti skup jezika.

Ponekad se usvaja konvencija da se terminali od jednog karaktera okružuju navodnicima `"` kako bi se razlikovali od metasimbola EBNF.

Korišćenjem EBNF identifikatori programskog jezika C se mogu definisati sa:

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> }
<slovo ili _> ::= "a"|...|"z"|"A"|...|"Z"|"_"
<cifra> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Korišćenjem EBNF, jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>}
<cifra> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Gramatika aritmetičkih izraza može u EBNF da se zapiše kao: ~~~ ::= {“+” } ::= {”*” } := “(” “)” | ~~~

Naredba grananja `if` sa opcionim pojavljivanjem `else` grane može se opisati sa:

```
<if_naredba> ::=  if "(" <bulovski_izraz> ")"
                     <niz_naredbi>
                  [ else
                     <niz_naredbi> ]
<niz_naredbi> ::= "{" <naredba> ";" { <naredba> ";" } "}"
```

S obzirom na veliki broj različitih proširenja BNF notacije, u jednom trenutku je postalo neophodno standardizovati notaciju. Verzija koja se danas standardno podrazumeva pod terminom EBNF je verzija koju je definisao i upotrebio Niklaus Virt u okviru definicije programskog jezika Pascal, 1977. Međunarodni komitet za standardizaciju definiše ovu notaciju u okviru standarda *ISO 14977*.

1.2.2.3 Sintaksički dijagrami

Sintaksički dijagrami ili *sintaksički grafovi* ili *pružni dijagrami* (engl. *railroad diagrams*) su grafička notacija za opis sintakse jezika koji su po izražajnosti ekvivalentni sa kontekstno slobodnim gramatikama tj. sa BNF. Primeri sintaksičkih dijagrama:

1.3 Semantička analiza

Jedan od osnovnih zadataka semantičke analize je *provera tipova* (engl. *typchecking*). Tokom provere tipova proverava se da li je svaka operacija primenjena na operande odgovarajućeg tipa (npr. indeksni pristup nizu obično zahteva da je indeks celobrojnog tipa i ako se pokuša pristup elementu niza sa navedenim indeksom koji je realnog tipa, prijavljuje se greška). U nekim slučajevima se u sintaksičko drvo umeću implicitne konverzije, gde je potrebno.

Razmatrajmo naredni fragment C koda.

```
float x, y;
x = 2 * y;
```

Semantički analizator prikuplja informacije o tipovima pojedinačnih promenljivih (obični ih skladišti u tablicu simbola). Nakon toga, proveravaju se tipovi operanada i pošto se zaključuje da se množe ceo broj i broj u pokretnom zarezu, ceo broj se konvertuje u realan. Najbolji način da se to uradi je da se u sintaksičkom drvetu čvor koji predstavlja celobrojnu konstantu 2 zameni čvorom koji predstavlja konstantu 2 zapisanu u realnom zarezu. Drugi način je da se u drvo umetne nov čvor koji predstavlja operaciju konverzije vrednosti iz celobrojnog u realni tip i da se prilikom generisanja koda na osnovu tog čvora generiše instrukcija koja tu konverziju vrši. U slučaju kada je potrebno izvršiti konverziju tipa promenljive, to će se skoro neizbežno desiti, a u slučaju da se ispod tog čvora nađe konstantna vrednost veoma verovatno je da će se konverzija izvršiti tokom faze optimizacije.

1.4 Generisanje međukoda

Većina kompilatora prevodi sintaksičko stablo provereno i dopunjeno tokom semantičke analize u određeni *međukod* (engl. *intermediate code*), koji se onda dalje analizira i optimizuje i na osnovu koga se u kasnijim fazama gradi rezultujući asemblerski i mašinski kod. Najčešći oblik međureprezentacije je tzv. *troadresni kod* u kome se javljaju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator. Otuda naziv troadresni: u svakoj instrukciji se navode "adrese" najviše dva operanda i rezultata operacije. Naredbe kontrole toka (`if`, `if-else`, `while`, `for`, `do-while`) se uklanjaju i svode na uslovne i bezuslovne skokove (predstavljenih često u obliku naredbi `goto`).

```
ifFalse v < 120 goto L
t1 = v * t
```

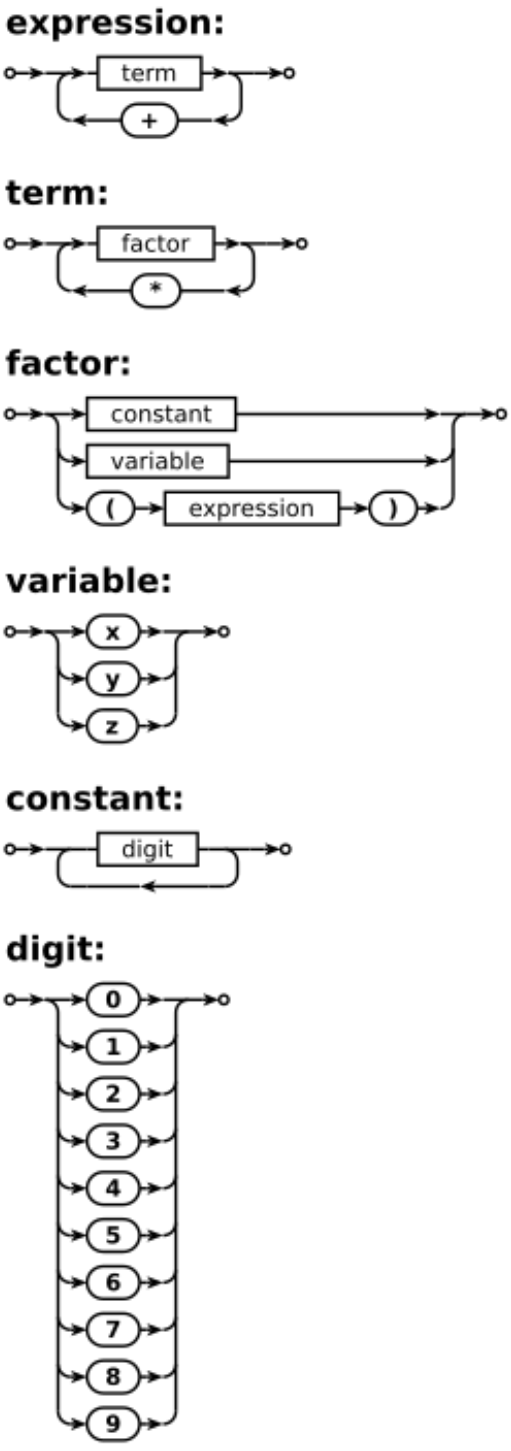


Figure 3: Sintaksički dijagrami za izraze

```
  s = s0 + t1
L:
```

Ako je, na primer, promenljiva `t` celobrojna, a `s` i `v` realne, onda bi se tokom semantičke analize u drvo ubacio čvor konverzije tipa i to bi se oslikalo i u međukodu.

```
  ifFalse v < 120 goto L
  t1 = int2float(t)
  t2 = v * t1
  s = s0 + t2
L:
```

Da bi se postigla troadresnost, u međukodu se vrednost svakog podizraza smešta u novouvedenu *privremenu promenljivu* (engl. *temporary*). U prethodnom primeru takve su `t1` i `t2`. Njihov broj je potencijalno neograničen, a tokom faze generisanja koda (tj. registarske alokacije) svim promenljivama se dodeljuju fizičke lokacije gde se one skladište (to su ili registri procesora ili lokacije u glavnoj memoriji).

1.5 Optimizacija

Optimizacija podrazumeva poboljšanje performansi koda, zadržavajući pri tom ekvivalentnost sa polaznim (optimizovani kod za iste ulaze mora da vrati iste izlaze kao i originalni kod i mora da proizvede iste sporedne efekte - npr. ako originalni kod nešto ispisuje na ekran ili skladišti na disk, to mora da uradi i optimizovani).

Optimizacija se najčešće odvija na dva nivoa: - optimizacija međukoda se generiše na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture. - optimizacija ciljnog koda se izvršava na samom kraju sinteze i zasniva se na detaljnom poznavanju ciljne arhitekture i asemblerskog i mašinskog jezika na kome se izražava ciljni program.

Navedimo nekoliko primera najčešćih optimizacija koje se vrše nad međukodom.

- Konstantni izrazi se mogu izračunati (engl. constant folding). Npr.

  ```
  x = 2 + 2          x = 4
  y = z * x          y = z * x
  ```

- Izbegava se upotreba promenljivih čija je vrednost konstantna (engl. constant propagation).

  ```
  x = 4              y = 4 * x
  y = z * x
  ```

- Operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže (engl. strength reduction).

```
y = 4 * x              y = x << 2
```

- Izbegava se vršenje istog izračunavanja više puta (engl. common subexpression elimination).

```
a = x + y          a = x + y
b = x - y          b = x - y
c = x + y          c = a
d = a + c          d = a + c
```

- Izbegava se uvođenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih (engl. copy propagation).

```
a = x + y          a = x + y
b = x - y          b = x - y
c = x + y          d = a + a
d = a + c
```

- Izračunavanja vrednosti promenljivih koje se dalje ne koriste, se eliminišu (engl. dead code elimination). Npr. ako se nakon prethodnog bloka koda koristi `d`, ali ne i `b`, tada se kod uprošćava na sledeći način.

```
a = x + y          a = x + y
b = x - y          d = a + a
d = a + a
```

Posebno je značajna optimizacija petlji, jer se kod u njima očekivano izvršava veći broj puta i čak i mala ušteda može biti značajnija nego neka veća ušteda u kodu koji se izvršava samo jednom. Jedna od najznačajnijih optimizacija petlji je izdvajanje izračunavanja vrednosti promenljivih koje su invarijantne za tu petlju (čija je vrednost ista u svakom koraku petlje) ispred same petlje. Pokažimo ovu ideju na C kodu (mada se optimizacija izvršava na nivou međukoda).

```
for (int i = 0; i < n; i++)          t1 = sin(alpha)
  a[i] = r[i] * sin(alpha);          for (int i = 0; i < n; i++)
                                         a[i] = r[i] * t1;
```

1.6 Generisanje koda

Tokom generisanja koda optimizovani međukod se prevodi u završni asemblerski tj. mašinski kod. U tom procesu potrebno je napraviti nekoliko značajnih odluka koje utiču na kvalitet generisanog koda. One se donose tokom faze odabira instrukcija (tada se određuje kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda), faze alokacije registara (tada se određuje lokacija na kojoj se svaka od promenljivih skladišti) i faze raspoređivanja instrukcija (tada se određuje redosled instrukcija koji doprinosi kvalitetnijem iskorišćavanju protočne obrade i paralelizacije na nivou instrukcija). Ove faze su prilično isprepletane. Najčešće se prvo radi odabir instrukcija dok se raspoređivanje instrukcija nekada rade i pre i posle alokacije registara.

1.6.1 Odabir instrukcija

Sekvenca troadresnih instrukcija se često može na više načina prevesti u realni asemblerski kod. Posmatrajmo C naredbu `a[i] = b + 1`, i pretpostavimo da se niz `a` i promenljiva `i` nalaze na steku i to na pozicijama određenim pomerajima C_a i C_i (to su konstante čije su vrednosti poznate kompilatoru) u odnosu na pokazivač početka okvira steka b_p, a da je promenljiva `b` globalna i da se nalazi na adresi M_b. Ta instrukcija se može predstaviti sledećim drvetom (čvor `ind` označava da se na tom mestu vrši indeksni pristup memorijskoj adresi).

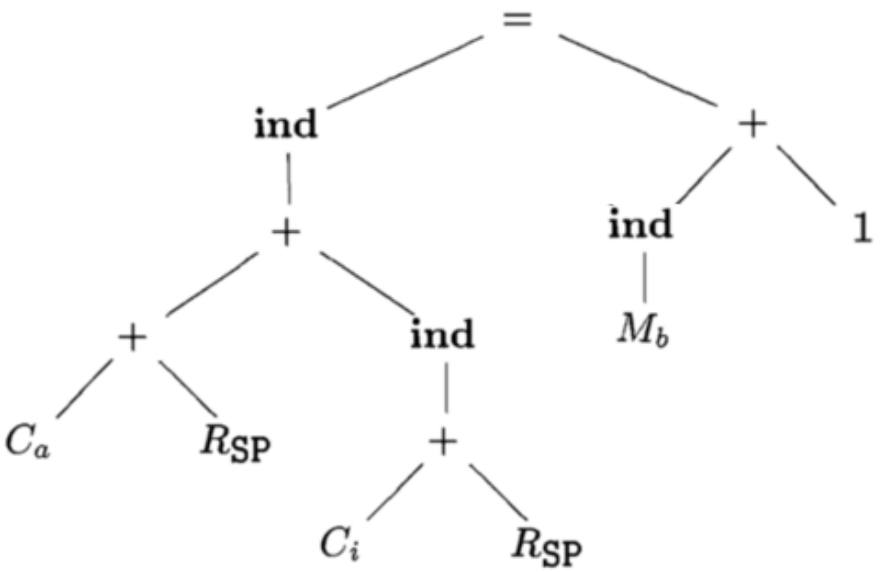


Figure 4: Drvo koje predstavlja troadresni kod potreban za izvršavanje naredbe `a[i] = b + 1`

Svakoj asemblerskoj instrukciji (uzevši u obzir i načine adresiranja) pridružuju se oblici drveta koji se njome mogu prekriti i odabir instrukcija teče tako što se na osnovu tih šablona pokušava prekriti celo drvo. Na slici su prikazani šabloni pridruženi instrukcijama `add ri, [rj + #Ca]`, `add ri, rj` i `inc ri`

Analizom raznih mogućnosti prekrivanja njegovih čvorova polaznog drveta asemblerskim naredbama možemo dobiti različite oblike asemblerskog koda. Na primer,

```
mov r0, #Ca          ; upisujemo pomeraj početka niza #Ca u registar r0
add r0, bp           ; uvećavamo r0 za vrednost pokazivača okvira steka bp
add r0, [bp + #Ci]   ; uvećavamo r0 za vrednost upisanu u memoriji na adresi određenoj zbir
mov r1, [Mb]         ; učitavamo sadržaj memorije sa adrese Mb u registar r1
inc r1               ; uvećavamo vrednost registra r1 za 1
mov [r0], r1         ; u memoriju na adresu određenu trenutnom vrednošću r0 upisujemo vredn
```

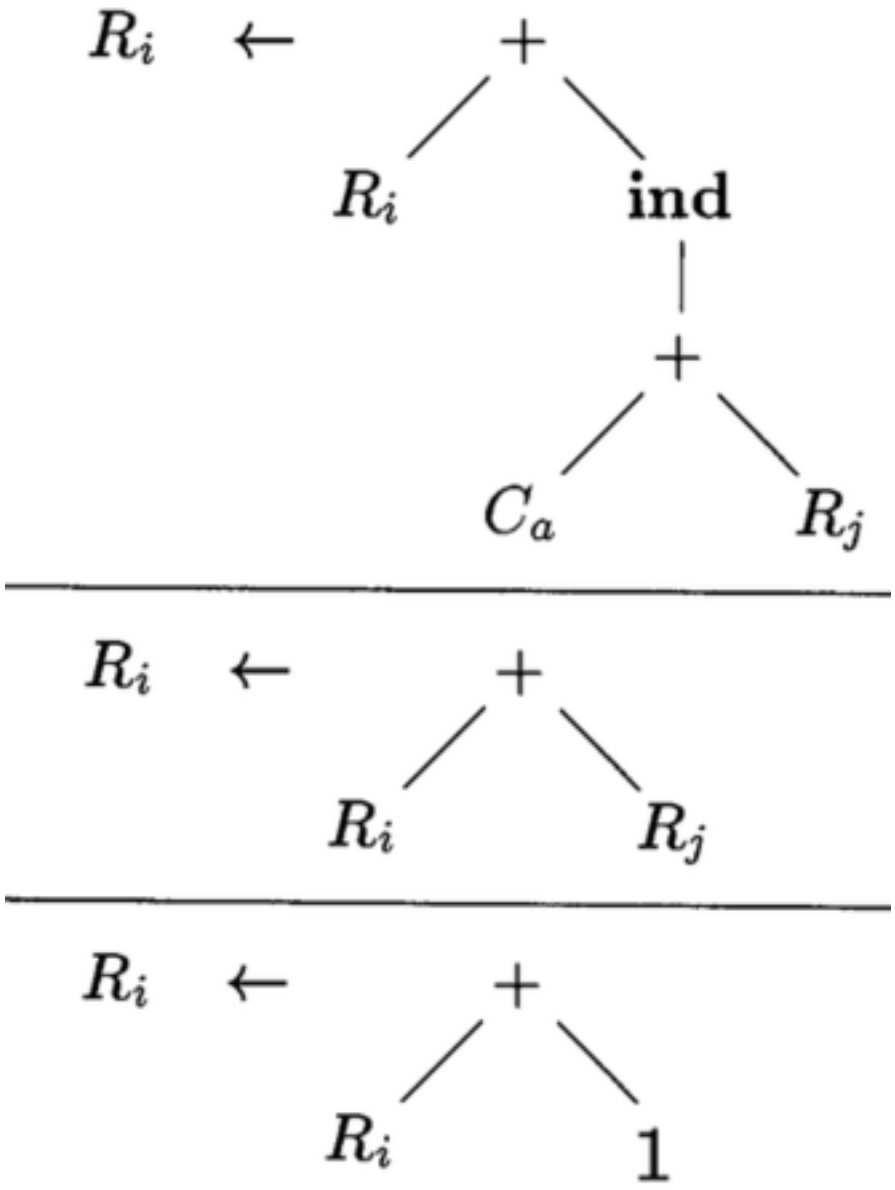


Figure 5: Šabloni pridruženi instrukcijama `add ri, [rj + #Ca]`, `add ri, rj` i `inc ri`

ili

```
mov r0, bp          ; upisujemo vrednost pokazivača bp na početak stek okvira u registar r0
add r0, #Ca         ; uvećavamo r0 za pomeraj početka niza
mov r1, bp          ; upisujemo vrednost pokazivača bp na početak stek okvira u r1
add r1, #Ci         ; uvećavamo r1 za pomeraj promenljive i
mov r1, [r1]        ; učitavamo iz memorije sadržaj sa adrese određene trenutnom vrednošću
add r0, r1          ; uvećavamo r0 za r1
mov r1, [Mb]        ; učitavamo sadržaj memorije sa adrese Mb u registar r1
add r1, 1           ; uvećavamo vrednost registra r1 za 1
mov [r0], r1        ; u memoriju na adresu određenu trenutnom vrednošću r0 upisujemo vredno
```

Zahvaljući korišćenju naprednijeg načina adresiranja (u instrukciji `add r0, [bp + #Ci]`) kao i efikasnije instrukcije uvećanja za 1 (`inc r1` umesto `add r1, 1`) realno je očekivati da je prvi kod efikasniji.

Obično se šabloni koji pokrivaju veće delove drveta favorizuju (gramzivi pristup, koji se koristi u algoritmima prekrivanja drveta kakav je maximal munc), ali postoje i algoritmi zasnovani na dinamičkom programiranju koji efikasno određuju optimalno pokrivanje.

1.6.2 Alokacija registara

Tokom faze registarske alokacije određuju se lokacije na kojima će biti skladištene vrednosti svih promenljivih koje se javljaju u međukodu (kako originalnih, tako i privremeno uvedenih tokom prevođenja u međukod). Cilj je da što više promenljivih bude skladišteno u registre procesora (jer je pristup registrima često za red veličine brži nego pristupa memoriji), međutim, to je često nemoguće, jer je broj registara ograničen i često prilično mali. Dve promenljive mogu biti smeštene u isti registar ako im se životni vek (period u kome se koriste) ne preklapa.

Posmatrajmo, na primer, naredni fragment koda i pretpostavimo da je na kraju izvršavanja potrebno poznavati samo vrednost promenljive `c` (ona se jedino koristi i nakon ovog koda). Za nju ćemo reći da je *živa* i nakon koda.

```
a = a + c
b = a + d
c = c + b
```

Analizom živosti promenljivih (engl. liveness analysis) možemo odrediti koje promenljive su žive pre svake instrukcije (njihove vrednosti je potrebno znati, jer se koristi u narednim instrukcijama). Analiza živosti se vrši unatrag instrukciju po instrukciju. Instrukcija oblika `x = y op z` izbacuje promenljivu `x` iz skupa živih (njenu vrednost nije potrebno znati pre instrukcije, jer se ona određuje tom instrukcijom), a u skup ubacuje `y` i `z` (njih je potrebno znati pre instrukcije da bi se ona mogla izvršiti). Po tom principu skupovi živih promenljivih se određuju na naredni način.

```
                {c, a, d}
a = a + c
                {c, a, d}
b = a + d
                {c, b}
c = c + b
                {c}
```

Može se primetiti da su `a`, `c` i `d` istovremeno žive (javljaju se u istom skupu), međutim, `b` je istovremeno živo samo sa `c`, ali ne sa `a` niti sa `d`. To znači da se `a` i `b` (ili alternativno `d` i `b` mogu smestiti u isti registar). Na primer, ako promenljive `a` i `b` smestimo u registar `r1`, promenljivu `c` u `r2`, a promenljivu `d` u `r3` možemo dobiti sledeći kod sa alociranim registrima.

```
r1 = r1 + r2
r1 = r1 + r3
r2 = r2 + r1
```

Analiza živosti je dosta kompleksnija kada se razmatraju programi složenije strukture (u kojima postoje skokovi tj. grananja i petlje), ali je princip sličan.

Nakon određivanja živosti promenljivih, jedna moguća tehnika za registarsku alokaciju je *bojenje grafa* (engl. *graph coloring*). Čvorovi grafa su promenljive, grana između promenljivih se postavlja ako su istovremeno žive, a onda se pokušava bojenje (nekom efikasnom heuristikom, jer je problem bojenja NP težak) sa onoliko boja koliko ima registara, tako da nikoja dva povezana čvora ne budu obojena u istu boju (jer boje odgovaraju registrima, a istovremeno žive promenljive ne smeju da budu u istom registru). Ako se ispostavi da se graf ne može obojiti, onda se bira neka promenljiva (poželjno neka koja se ne koristi često i izbegavaju se one koje su unutrašnjim petljama) i ona se smešta u memoriju (najčešće na programski stek). Pre svake instrukcije kojom se pristupa toj promenljivoj vrši se njeno učitavanje iz memorije (u troadresnom kodu dodaje se instrukcija učitavanja), a nakon svakog izračunavanja vrši se njen upis u memoriju (u troadresnom kodu se dodaje instrukcija upisa u memoriju). Time se skraćuje životni vek te promenljive u procesoru (svako njeno pojavljivanje sada faktički predstavlja novu promenljivu) čime se povećava mogućnost da se ostale promenljive uspešno rasporede.

1.6.3 Raspoređivanje instrukcija

Faza raspoređivanja instrukcija pokušava da doprinese brzini izvršavanja programa menjanjem redosleda instrukcija. Naime, nekim instrukcijama je moguće promeniti redosled izvršavanja bez promene semantike programa.

Jedan od ciljeva raspoređivanja je da se upotrebe pojedinačnih promenljivih lokalizuju u kodu, čime se povećava šansa da se registri oslobode dugog čuvanja vrednosti nekih promenljivih i da se upotrebe za čuvanje većeg broja

promenljivih.

Svaki savremeni brži procesor može da izvršava nekoliko instrukcija tokom jednog otkucaja sistemskog sata tj. ima osobinu *protočne obrade* (engl. *pipelining*). Dok se jedna instrukcija dovlači iz memorije u procesor, druga se dekodira, treća se već izvršava, a četvrtoj se već skladište rezultati.

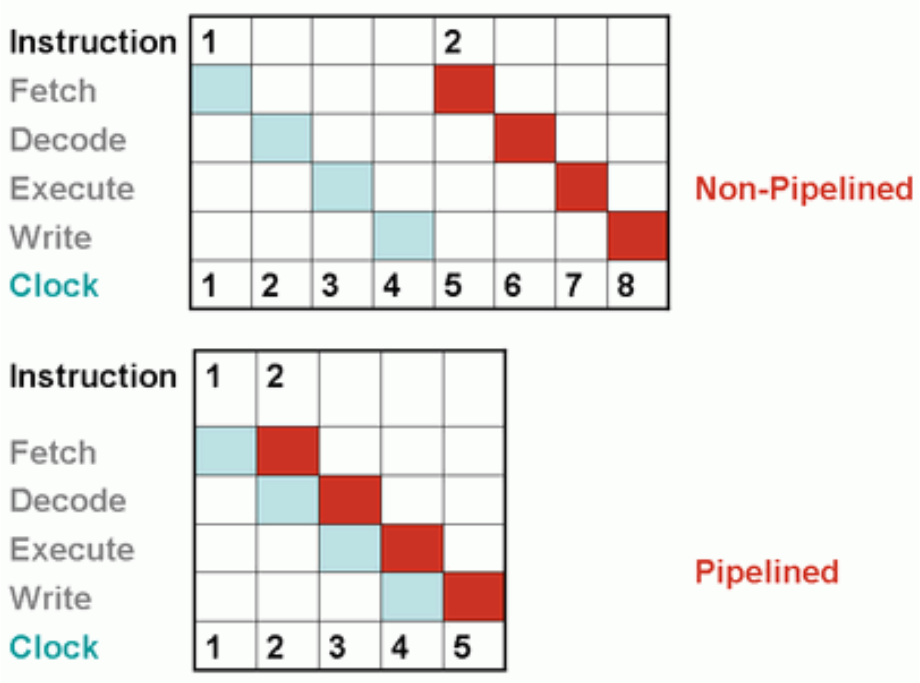


Figure 6: Protočna obrada

Promena redosleda instrukcija može uticati na to da se protočna obrada bolje iskoristi tj. da se izbegnu čekanja i zastoji u protočnoj obradi (engl. pipeline stalls) nastala zbog zavisnosti između susednih instrukcija. Na primer, kada je narednoj instrukciji potreban rezultat prethodne, njeno izvršvanje mora da pričeka da prethodna instrukcija završi sa upisom svojih rezultata. Raspoređivač u taj zastoj može umetnuti neku drugu instrukciju, nezavisnu od ovih i tako ubrzati ukupno izvršavanje svih instrukcija zajedno (ta pomerena instrukcija će biti izvršena dok bi procesor praktično čekao u prazno).

1.6.4 Serijalizacija

Kada su instrukcije odabrane, kada je izvršena alokacija registara i kada su instrukcije raspoređene, dobijeni kod se serijalizuje tj. zapisuje se u datoteke (objektne module) u obliku konkretnog mašinskog koda, čime se faza kompilacije

završava (nekada kompilator generiše asemblerski kod koji se onda asemblerom prevodi u mašinski). Nakon kompilacije pojedinačni objektni moduli se *povezuju* (engl. *linking*) u izršnu datoteku.

2 Stabla apstraktne sintakse

- Izrazi, naredbe, funkcije, programi itd. se mogu predstaviti u obliku *stabla apstraktne sintakse* (engl. *abstract syntax tree*, *AST*).
- Iako se zadnje faze kompilacije nekada mogu izvršavati paralelno sa sintaksičkom analizom, postupak se često značajno olakšava ako se rezultat sintaksičke analize predstavi eksplicitno sintaksičkim stablom.
- Naredni primer prikazuje AST za naredni fragment koda (koji implementira Euklidov algoritam sa oduzimanjem).

```
while (b != 0)
   if (a > b)
      a = a - b;
   else
      b = b - a;
return a;
```

2.1 Strukture podataka za reprezentovanje AST

- Stabla mogu imati više različitih vrsta čvorova (npr. čvor konstante, čvor promenljive, čvor binarnog operatora, čvor naredbe while, …)

2.1.1 Jezik C

- U jeziku C za svaku vrstu čvora definišemo posebnu strukturu podataka.
- Tip čvora definišemo kao strukturu koja sadrži podatak o tome koja je vrsta čvora u pitanju i uniju svih mogućih struktura pojedinačnih čvorova.
- Primer struktura podataka za reprezentovanje stabla za celobrojne aritmetičke izraze u jeziku C.

```
/* Cvor konstante */
struct nodeConst {
  int value;  /* Vrednost konstante */
};

/* Cvor promenljive */
```

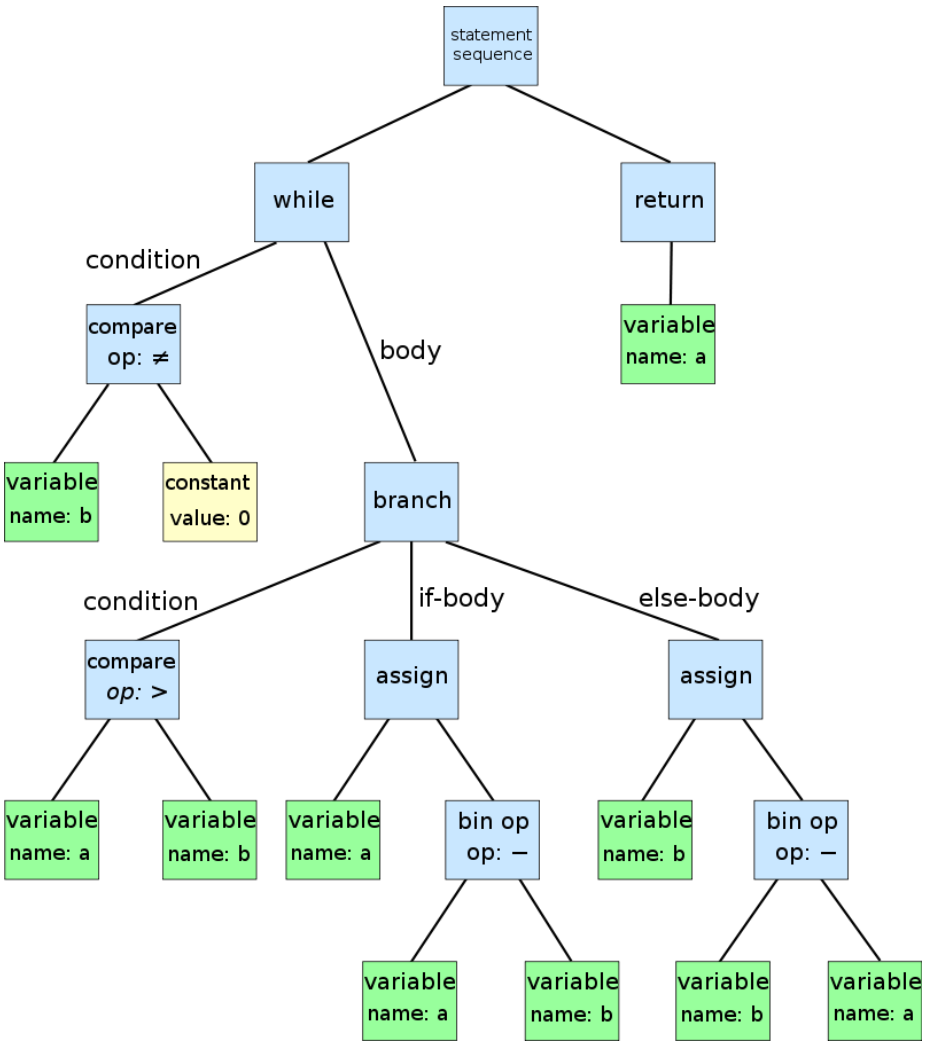


Figure 7: AST za Euklidov algoritam

```
struct nodeVar {
  char id[64]; /* Naziv promenljive */
};

/* Cvor operatora */
struct nodeOp {
  int operator;           /* Operator - ASCII kod ('+', '-', ...)
                                        ili konstanta (WHILE, ...) */
  int numOperands;        /* Broj operanada */
  struct cvor** operands; /* Niz operanada */
};

/* Oznaka vrste cvora */
typedef enum {CONST,  VAR,  OP} nodeType;
/* Struktura cvora */
typedef struct node {
  nodeType type;          /* Oznaka vrste */
  union {
    struct nodeConst const;
    struct nodeVar var;
    struct nodeOp op;
  } data;               /* Specificni podaci za razne vrste cvorova */
} node;
```

- Ako je `n` tipa `node*` pokazivač na čvor, tada vrstu čvora ispitujemo pomoću `n->type`, imenu promenljive u čvoru promenljive pristupamo pomoću `n->data.var.id` ili, još kraće, preko `n->var.id`, vrednosti konstante u čvoru konstante `n` pristupamo pomoću `n->data.const.value` ili još kraće pomoću `n->const.value`, dok se, na primer, prvom operandu čvora binarnog operatora pristupa pomoću `n->data.op.operands[0]` ili još kraće pomoću `n->op.operands[0]`.
- Sve ove definicije možemo objediniti. Pri tom se za vrste čvorova koje sadrže samo jedan podatak može izbeći kreiranje strukture.

```
typedef enum {CONST,  VAR,  OP} nodeType;
typedef struct node {
  nodeType type;
  union {
    int value;
    char id[64];
    struct nodeOp {
      int operator;
      int numOperands;
      struct cvor** operands;
    } op;
  } data;
```

```
} node;
```

- Pošto smo izbegli strukture za čvor promenljive i konstante, imenu promenljive u čvoru promenljive na koji ukazuje pokazivač `n` možemo pristupiti pomoću `n->id`, dok vrednosti konstante u čvoru konstante na koji ukazuje `n` možemo pristupiti pomoću `n->value`.
- Poželjno je obezbediti konstruktore - funkcije koje grade svaku pojedinačnu vrstu čvora.

```
node* makeNodeConst(int vrednost);
node* makeNodeVar(char* s);
node* makeNodeUnOperator(int operator, node* op1);
node* makeNodeBinOperator(int operator, node* op1, node* op2);
```

- Implementacija konstruktora.

```
node* makeNode(nodeType tip) {
  node* n = (node*)malloc(sizeof(node));
  /* obrada gresaka */
  n->type = type;
  return n;
}

node* makeNodeConst(int value) {
  node* n = makeNode(CONST);
  n->const.value = value;
  return n;
}

node* makeNodeVar(char* id) {
  node* n = makeNode(VAR);
  strncpy(n->var.id, s, 63);
  return n;
}

node* makeNodeOperator(int numOperands) {
  node* n = makeNode(OP);
  n->op.numOperanads = 1;
  n->op.operands = (node**)malloc(numOperanads * sizeof(node *));
  /* obrada gresaka */
}

node* makeNodeUnOperator(int operator, node* op1)  {
  node* n = makeNodeOperator(1);
  n->op.operands[0] = op1;
  return n;
}
```

```
node* makeNodeBinOperatora(int operator, node* op1, node* op2) {
  node* n = makeNodeOperator(2);
  n->op.operands[0] = op1;
  n->op.operands[1] = op2;
  return n;
}
```

- Stablo za izraz `3 + 4 * x` se onda može napraviti na sledeći način.

```
node* n = makeNodeBinOperator('+',
              makeNodeConst(3),
              makeNodeBinOperator('*',
                 makeNodeConst(4),
                 makeNodeVar("x")));
```

- Možemo definisati i funkciju za štampanje stabla (koristiće nam za debagovanje).

```
void print(nodeType* n);
```

- Implementacija može biti sledeća:

```
void print(nodeType* p) {
  /* Posao delegiramo pomoćnoj funkciji koja vrši uvlačenje
     ispisa u zavisnosti od nivoa čvora tj. od dubine rekurzije */
  print_(p, 0);
}

void print_(nodeType* p, int level) {
  /* U zavisnosti od dubine rekurzije (nivoa čvora u drvetu)
     vršimo uvlačenje */
  int k;
  for (k = 0; k < level; k++)
    putchar(' ');
  /* Analiziramo tip čvora */
  switch(p->type) {
  case CONST: /* cvor konstante */
    printf("Con: %d\n", p->con.value);      /* ispisujemo vredost */
    break;
  case VAR: /* cvor promenljive */
    printf("Id: %s\n", p->var.id);          /* ispisujemo naziv */
    break;
  case OP: /* cvor operatora */
    printf("Op %c\n", p->op.operator);      /* ispisujemo operator */
    for (k = 0; k < p->op.numOperands; k++) /* rekurzivno ispisujemo operande */
      print_(p->op.operand[k], level+1);
    break;
```

```
  }
}
```

2.1.2 Jezik Java

- U OO jezicima, drvo predstavljamo hijerarhijom klasa (obrazac Composite).

```
abstract class Expression {
    public void print() {
      printIndent(0);
    }

    protected void printIndent(int level) {
       for (int i = 0; i < level; i++)
          System.out.print("\t");
       printMe(level);
    }

    protected abstract void printMe(int level);
}

class Constant extends Expression {
    int value;

    public Constant(int value) {
        this.value = value;
    }

    protected void printMe(int level) {
        System.out.println("Const: " + value);
    }
}

class Variable extends Expression {
    String name;

    public Variable(String name) {
        this.name = name;
    }

    protected void printMe(int level) {
        System.out.println("Var: " + name);
    }
}
```

```
class Operator extends Expression {
    Expression[] operands;
    int operator;

    public Operator(char operator,
                    Expression op1,
                    Expression op2) {
        this.operator = operator;
        this.operands = new Expression[2];
        this.operands[0] = op1;
        this.operands[1] = op2;
    }

    protected void printMe(int level) {
        System.out.println("Op: " + operator);
        for (int i = 0; i < operands.length; i++)
            operands[i].printIndent(level+1);
    }
}
```

2.1.3 Jezik Haskell

- Implementacija tipova podataka za reprezentaciju stabla je neuporedivo jednostavnija u jeziku Haskell.

```
data OpType = PLUS | MinUS | TIMES | DIV
data Expr =
     Const Int
   | Var String
   | Op OpType [Expr]
```

- Konstruktore već imamo na raspolaganju.
- Stablo za izraz `3 + 4 * x` se onda može napraviti na sledeći način.

```
n = Op PLUS [
       Const 3,
       Op TIMES [
          Const 4,
          Var "x"
       ]
    ]
```

- I ovde možemo definisati funkciju za prikaz izraza.

```
print :: Expr -> String
print e = printIndent 0 e where
```

```
printIndent :: Int -> Expr -> String
printIndent n e = indent n ++ printMe n e
printMe :: Int -> Expr -> String
printMe n (Const k)   = "Const: " ++ show k
printMe n (Var x)     = "Var: " ++ x
printMe n (Op op ops) = "Op " ++ show op ++ "\n" ++
                        intercalate "\n" (map (printIndent (n+1)) ops)
```

2.2 Izgradnja apstraktnog stabla tokom sintaksičke analize

2.2.1 Lex/Yacc

- Ako koristimo C/C++ sintaksički analizator možemo izgraditi korišćenjem Yacc/Bison.

```
%left '+' '-'
%left '*' '/'

%union {
  int num_attr;
  char* str_attr;
  node* node_attr;
}

%token <str_attr> ID
%token <num_attr> BROJ

%type <node_attr> expr

%%

expr :
        expr '+' expr { $$ = makeNodeBinOperator('+', $1, $3); }
      | expr '-' expr { $$ = makeNodeBinOperator('-', $1, $3); }
      | expr '*' expr { $$ = makeNodeBinOperator('*', $1, $3); }
      | expr '/' expr { $$ = makeNodeBinOperator('/', $1, $3); }
      | NUM           { $$ = makeNodeConst($1); }
      | ID            { $$ = makeNodeVar($1); free($1); }
      ;

%%
```

2.2.2 Rekurzivni spust

- Možemo koristiti i tehniku rekurzivnog spusta.

- Koristimo LL(1) gramatiku koja je desno rekurzivna, ali želimo da u drvetu nametnemo levu asocijativnost.
- Moramo koristiti nasleđene atribute. Posmatrajmo sledeću atributsku gramatiku.

```
E : T { EP.u = T.v; } EP { E.v = EP.v; }
  ;

EP_1 : '+' T { EP_2.u = Op('+', EP_1.u, T.v); } EP_2 { EP_1.v = EP_2.v; }
     | '-' T { Ep_2.u = Op('-', EP_1.u, T.v); } EP_2 { EP_1.v = EP_2.v; }
     | eps   { EP_1.v = EP_1.u; }
     ;

T : F { TP.u = F.v; } TP { T.v = TP.v; }
  ;

TP_1 : '*' F { TP_2.u = Op('*', TP_1.u, F.v); } TP_2 { TP_1.v = TP_2.v; }
     | eps   { TP_1.v = TP_1.u; }
     ;

F : NUM { F.v = Const(NUM.v); }
  | ID  { F.v = Var(ID.name); }
  | '(' E ')' { F.v = E.v; }
  ;
```

- Prikažimo implementaciju sintaksičkog stabla na osnovu prethodne atributske gramatike na primeru jezika JAVA.

```
class Parser {
    /* Oznake tokena */
    static final int EOI = 0;
    static final int NUM = 1;
    static final int ID = 2;
    static final int PLUS = 3;
    static final int MinUS = 4;
    static final int ASTERISK = 5;
    static final int LP = 6;
    static final int RP = 7;

    /* Tekuca leksema */
    static String yylval_str;
    /* Vrednost tekuceg tokena */
    static int yylval_num;

    /* Leksicki analizator */
    static int yylex() throws java.io.IOException {
```

```
    char c = (char)System.in.read();
    if ('0' <= c && c <= '9') {
        yylval_num = c - '0';
        return NUM;
    }
    if ('a' <= c && c <= 'z') {
        yylval_str = Character.toString(c);
        return ID;
    }
    switch(c) {
        case '+':
            return PLUS;
        case '-':
            return MinUS;
        case '*':
            return ASTERISK;
        case '(':
            return LP;
        case ')':
            return RP;
    }
    return EOI;
}

/* LL(1) gramatika izraza */

/*  E : T EP      { NUM, ID, LP }
 *  EP : + T EP   { PLUS }
 *     : - T EP   { MinUS }
 *     | eps      { EOI, RP }
 *  T : F TP      { NUM, ID, LP }
 *  TP : * F TP   { ASTERISK }
 *     | eps      { PLUS, MinUS, EOI, RP}
 *  F : NUM       { NUM }
 *    | ID        { ID }
 *    | ( E )     { LP }
 */

static Expression E()
    throws java.io.IOException {
    System.out.println("E -> T EP");
    Expression t_attr = T();
    Expression ep_attr = EP(t_attr);
    return ep_attr;
}
```

```
static Expression EP(Expression prev)
    throws java.io.IOException {
    if (token == PLUS || token == MinUS) {
        System.out.println("EP -> +- T EP");

        char op = ' ';
        if (token == PLUS)
            op = '+';
        else if (token == MinUS)
            op = '-';

        token = yylex();
        Expression t_attr = T();
        Expression e = Op(op, prev, t_attr);
        Expression ep_attr = EP(e);
        return ep_attr;
    } else if (token == EOI || token == RP) {
        System.out.println("EP -> eps");
        return prev;
    } else {
        System.err.println("Greska");
    }
    return null;
}

static Expression T() throws java.io.IOException {
   System.out.println("T -> F TP");
   Expression f_attr = F();
   Expression tp_attr = TP(f_attr);
   return tp_attr;
}

static Expression TP(Expression prev) throws java.io.IOException {
    if (token == ASTERISK) {
        System.out.println("TP -> * F TP");
        token = yylex();
        Expression f_attr = F();
        Expression e = Op('*', prev, f_attr);
        Expression tp_attr = TP(e);
        return tp_attr;
    } else if (token == EOI || token == PLUS || token == MinUS || token == RP) {
        System.out.println("TP -> eps");
        return prev;
    } else {
        System.err.println("Greska");
    }
```

```
        return null;
    }

    static Expression F() throws java.io.IOException {
        if (token == NUM) {
            System.out.println("F -> NUM");
            token = yylex();
            Expression attr = new Constant(yylval_num);
            return attr;
        } else if (token == ID) {
            System.out.println("F -> ID");
            token = yylex();
            Expression attr = new Variable(yylval_str);
            return attr;
        } else if (token == LP) {
            System.out.println("F -> ( E )");
            token = yylex();
            Expression attr = E();
            if (token != RP)
                System.err.println("Greska");
            else
                token = yylex();
            return attr;
        } else
            System.err.println("Greska");
        return null;
    }

    public static void main(String[] args)  throws java.io.IOException  {
        token = yylex();
        Expression e = E();
        e.print();
    }
}
```

2.2.3 Haskell parsec

- Haskell pruža veoma moćnu biblioteku parsec za parsiranje.
- Primer definicije parsera za aritmetičke izraze.

```
import Control.Monad
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as Token
```

```haskell
-- Definicija tipa cvorova stabla
data OpType = PLUS | MinUS | TIMES | DIV deriving Show
data Expr =
     Const Integer
   | Var String
   | Op OpType [Expr] deriving Show

-- Konstruktori binarnih operacija (radi lakseg koriscenja)
plus  x y = Op PLUS [x, y]
minus x y = Op MinUS [x, y]
times x y = Op TIMES [x, y]

-- Definicija leksickog analizatora
lexer = Token.makeTokenParser languageDef
  where
    languageDef = emptyDef { Token.identStart = letter,
                             Token.identLetter = alphaNum,
                             Token.reservedOpNames = ["+", "-", "*"]
                           }

-- parseri koji citaju identifikatore, zagrade, brojeve i operatore
identifier = Token.identifier lexer
parens     = Token.parens     lexer
integer    = Token.integer    lexer
reservedOp = Token.reservedOp lexer

-- parser za izraze
exprParser :: Parser Expr
exprParser = buildExpressionParser operators term
  where
    -- operatori sa njihovom pozicijom, prioritetom odredjenim
    -- redosledom nabrajanja i asocijativnoscu
    operators =
     [[Infix (reservedOp "*" >> return times) AssocLeft],
      [Infix (reservedOp "+" >> return plus) AssocLeft,
       Infix (reservedOp "-" >> return minus) AssocLeft]]
    -- osnovni termovi u izrazima: izraz u zagradama, promenljiva, konstanta
    term = parens exprParser
       <|> liftM Var identifier
       <|> liftM Const integer

-- ulazna tacka - poziva se parser izraza na stringu i proverava da li
-- je parsiranje uspelo
parseString str =
  case parse exprParser "" str of
```

```
    Left e -> error $ show e
    Right r -> r
```

2.3 Usmereni aciklični grafovi (DAG)

- U složenim izrazima često dolazi do ponavljanja nekih podizraza (npr. $(x - y) * (x - y)$).
- Efikasnija reprezentacija se postiže ako se umesto stabla koristi *usmereni aciklički graf* (engl. *directed acyclic graph*, *DAG*).
- Primer `a + a*(b-c) + (b-c)*d`.

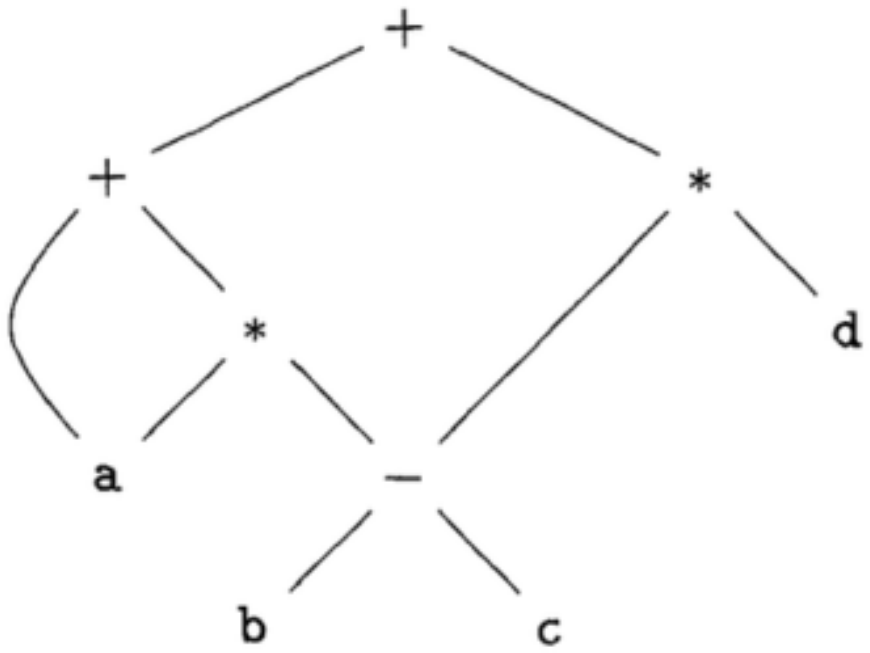


Figure 8: Primer usmerenog acikličnog grafa za izraz `a + a*(b-c) + (b-c)*d`

3 Interpretator koji izračunava vrednost izraza

- Vrednosti promenljivih čuvamo u mapi koja preslikava imena promenljivih u njihove vrednosti.

```
abstract class Expression {
    ...
```

```
    // vrednost izraza za date vrednosti promenljivih
    public abstract int interpret(Map<String, Integer> table);
}

class Constant extends Expression {
    ...

    // vrednost izraza za date vrednosti promenljivih
    public int interpret(Map<String, Integer> table) {
        return value;
    }
}

class Variable extends Expression {
    ...
    // vrednost izraza za date vrednosti promenljivih
    public int interpret(Map<String, Integer> table) {
        return table.get(name);
    }

}

class Operator extends Expression {
    ...

    // vrednost izraza za date vrednosti promenljivih
    public int interpret(Map<String, Integer> table) {
        int i0 = operands[0].interpret(table);
        int i1 = operands[1].interpret(table);
        return applyOp(i0, i1);
    }

    // primenjuje operator na vrednosti
    public abstract int applyOp(int x, int y);
}

// Operator sabiranja
class Plus extends Operator {
    ...

    // primena operatora +
    public int applyOp(int x, int y) {
        return x + y;
    }
}
```

```
// Operator množenja
class Times extends Operator {
    ...

    // primena operatora *
    public int applyOp(int x, int y) {
        return x * y;
    }
}
```

4 Generisanje koda za stek mašinu

- Simulator stek mašine možemo implementirati veoma jednostavno (npr. u jeziku C++).

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main() {
  stack<int> st;
  string s;
  while (cin >> s) {
    if (s == "push") {
      int x;
      cin >> x;
      st.push(x);
    } else if (s == "pop") {
      st.pop();
    } else if (s == "add") {
      int op2 = st.top();
      st.pop();
      int op1 = st.top();
      st.pop();
      st.push(op1 + op2);
    } else if (s == "mul") {
      int op2 = st.top();
      st.pop();
      int op1 = st.top();
      st.pop();
      st.push(op1 * op2);
    } else if (s == "print") {
      cout << st.top() << endl;
```

```
    }
  }
}

/*
   Komilacija za stek masinu sa sledecim instrukcijama:
     - push x - stavlja navedeni broj x na stek
     - add - skida dve vrednosti sa vrha steka i na stek stavlja njihov zbir
     - sub - skida dve vrednosti sa vrha steka i na stek stavlja njihovu razliku
     - mul - skida dve vrednosti sa vrha steka i na stek stavlja njihov proizvod

   - Pretpostavljamo da izraz sadrzi samo konstante i binarne operatore.
   - Kada kompiliramo cvor p dobijamo kod koji na stek dodaje jedan broj
     koji predstavlja vrednost cvora p .
   - Kompilacija se u ovom slucaju svodi na postfiksni obilazak drveta.
*/
void compileStackMachine(nodeType* p) {
  /* brojac slobodnih labela (za uslovne i bezuslovne skokove i
     kontrolu toka programa) */
  static int lbl;

  switch(p->type) {
  case typeCon:
    /* konstante postavljamo na stek */
    printf("\tpush %d\n", p->con.value);
    break;
  case typeId:
    /* promenljive postavljamo na stek */
    printf("\tpush [%s]\n", p->id.name);
  case typeOpr:
    if (p->opr.oper == PRinT) { /* naredba stampanja */
      /* vrsimo kompilaciju operanda i generisemo kod koji izracunava
         argument naredbe - nakon njegovog izracunavanja rezultat je
         na vrhu steka */
      compileStackMachine(p->opr.op[0]);
      /* generisemo (asemblersku) instrukciju stampanja */
      printf("\tprint\n");
    } else if (p->opr.oper == IF) { /* naredba if */
      /* rezervisemo labelu za kraj naredbe */
      int lblAfter = lbl++;
      /* generisemo kod koji izracunava uslov - rezultat je na vrhu
         steka */
      compileStackMachine(p->opr.op[0]);
      /* generisemo instrukciju koja proverava da li je na vrhu steka
         nula i ako jeste, skace na kraj tj. preskace telo naredbe
         grananja */
```

```
  printf("\tjz LBL%d\n", lblAfter);
  /* generisemo kod za telo naredbe grananja */
  compileStackMachine(p->opr.op[1]);
  /* generisemo labelu na kraju naredbe grananja */
  printf("LBL%d:\n", lblAfter);
} else if (p->opr.oper == WHILE) { /* petlja while  */
  /* rezervisemo dve labele: jednu za pocetak i jednu za kraj petlje */
  int lblBefore = lbl++;
  int lblAfter = lbl++;
  /* generisemo labelu na pocetku petlje */
  printf("LBL%d:\n", lblBefore);
  /* generisemo kod koji izracunava uslov petlje - rezultat je na
     vrhu steka */
  compileStackMachine(p->opr.op[0]);
  /* generisemo instrukciju koja proverava da li je na vrhu steka
     nula i ako jeste, skace na kraj petlje */
  printf("\tjz LBL%d\n", lblAfter);
  /* generisemo kod za telo petlje */
  compileStackMachine(p->opr.op[1]);
  /* generisemo bezuslovni skok na pocetak petlje */
  printf("\tjmp LBL%d\n", lblBefore);
  /* generisemo labelu na kraju petlje */
  printf("LBL%d:\n", lblAfter);
} else if (p->opr.oper == '=') { /* naredba dodele */
  /* generisemo kod koji izracunava desnu stranu - rezultat je na
     vrhu steka */
  compileStackMachine(p->opr.op[1]);
  /* generisemo kod koji vrednost sa steka prebacuje u memoriju,
     na mesto promenljive sa leve strane */
  printf("\tpop [%s]\n", p->opr.op[0]->id.name);
} else {
  if (p->opr.nops == 2) { /* aritmeticki binarni operator */
    /* vrsimo kompilaciju levog operanda */
    /* nakon izvrsavanja dobijenog koda na vrh steka ce biti dodata vrednost
       prvog operanda */
    compileStackMachine(p->opr.op[0]);
    /* vrsimo kompilaciju desnog operanda */
    /* nakon izvrsavanja dobijenog koda na vrh steka ce biti dodata vrednost
       drugog operanda */
    compileStackMachine(p->opr.op[1]);
    /* primenjujemo operator koji sa steka skida dve vrednosti (a to
       su vrednosti operanada) i na stek stavlja rezultat primene
       odgovarajuceg operatora na te dve vrednosti (a to ce biti
       vrednosti cvora p) */
    switch(p->opr.oper) {
    case '+':
```

```
          printf("\tadd\n");
          break;
        case '-':
          printf("\tsub\n");
          break;
        case '*':
          printf("\tmul\n");
          break;
        case '<':
          printf("\tcmpLT\n");
          break;
        case '>':
          printf("\tcmpGT\n");
          break;
        }
      }
    }
  }
}
```

5 Tablice simbola

- Tablica simbola treba da podržava operaciju umetanja (`insert`) i operaciju pronalaženja informacije na osnovu datog ključa (`lookup`). Pretpostavićemo da je ključ uvek tipa `char*`, dok pridružena informacija može biti bilo kog tipa (zato koristimo `void*`).
- Još jedna značajna funkcionalnost se odnosi na čuvanje informacije o *dosegu*, engl. *scope*) (istom ključu se mogu dodeliti različite informacije u zavisnosti od toga koji je doseg trenutno aktuelan). Dosezi su ugnežđeni i možemo smatrati da se ponašaju kao stek. Novi doseg se
- Interfejs tablice simbola može, dakle, biti sledeći.

```
#ifndef __SYMBOLTABLE_H__
#define __SYMBOLTABLE_H__

#include <string.h>

void insert(char* key, void* binding);
void* lookup(char* key);
void begin_scope();
void end_scope();

#endif
```

- Implementaciju možemo zasnovati na heširanju.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include "symboltable.h"

#define MAX_KEY 256
#define SIZE 109

#define MAX_STACK 256
static char* undo_stack[MAX_STACK];
static int undo_sp = 0;

void undo_stack_push(char* s) {
  undo_stack[undo_sp++] = s;
}

char* undo_stack_pop() {
  char *s = undo_stack[--undo_sp];
  return s;
}

extern void yyerror(char* err);

typedef struct _bucket {
  char key[MAX_KEY];
  struct _bucket* next;
  void* binding;
} bucket;

static bucket* table[SIZE];

unsigned hash(char* s) {
  unsigned h = 0;
  char* t;
  for (t = s; *t; t++)
    h = h * 65599 + *t;
  return h;
}

bucket* add_bucket(char* key, void* binding, bucket* next) {
  bucket* b = malloc(sizeof(bucket));
  if (b == NULL) yyerror("Alloc error\n");
  strcpy(b->key, key);
```

```
  b->binding = binding;
  b->next = next;
  return b;
}

void insert(char* key, void* binding) {
  undo_stack_push(strdup(key));
  unsigned i = hash(key) % SIZE;
  table[i] = add_bucket(key, binding, table[i]);
}

id* lookup(char* key) {
  unsigned i = hash(key) % SIZE;
  bucket* b;
  for (b = table[i]; b; b = b->next)
    if (strcmp(b->key, key) == 0)
      return b->binding;
  return NULL;
}

void delete(char* key) {
  unsigned i = hash(key) % SIZE;
  bucket* head = table[i];
  assert(strcmp(head->key, key) == 0);
  table[i] = head->next;
  free(head->binding);
  free(head);
}

void begin_scope() {
  undo_stack_push(NULL);
}

void end_scope() {
  while(1) {
    char *s = undo_stack_pop();
    if (s == NULL) break;
    delete(s);
    free(s);
  }
}

void* make_binding(int x) {
  int* b = (int*)malloc(sizeof(int));
  *b = x;
  return (void*)b;
```

```
}
```

6 Semantička analiza

- Osnovni zadatak je provera tipova.
- Posmatrajmo fragment jezika Pascal koji podržava tipove `char` i `integer`, pokazivački tip i tip jednodimenzionog niza date dimenzije. Obratimo pažnju i na doseg identifikatora (kao što je uobičajeno, konflikt identifikatore se razrešava tako što svaka deklaracija u unutrašnjem dosegu sakriva onu u spoljašnjem).

```
var
  x: ^integer;  (* pokazivac na int *)
  a: array [5] of integer; (* niz 5 int-ova *)
  b2: integer;  (* int *)
  c3: array [5] of array [7] of ^char; (* niz od 5 nizova od 7 pokazivaca na char *)
begin
  a[a[x^] + 10] + b2 + x^; (* OK - tipa int *)
  x; (* OK - tipa pokazivac na int *)
  a[b2]; (* OK - tipa int *)
  var
    y: ^integer; (* pokazivac na int *)
  begin
     a[y^] (* OK - tipa int *)
  end;
  b2; (* OK - tipa int *)
  var
    x: array[3] of char; (* niz od tri char-a *)
  begin
    x (* OK - niz od tri char-a *)
  end;
  x (* OK - pokazivac na int *)
end
```

- Tipovi se unutar kompilatora takođe predstavljaju drvetom (potpuno analogno izrazima, kako je opisano u delu AST).

```
enum types {T_ERROR, T_CHAR, T_inTEGER, T_POinTER, T_ARRAY};

typedef struct _type {
  enum types tag;
  union {
     struct _type* ptype;
     struct {
        int num;
```

```
        struct _type* ptype;
      } atype;
  };
} type;

type* error_type();
type* char_type();
type* integer_type();
type* pointer_type(type* ptype);
type* array_type(int num, type* ptype);

int is_integer(type* t);
int is_pointer(type* t);
int is_array(type* t);

type* copy_type(type* t);

void print_type(type* t);
```

- I implementacija ovih funkcija je jednostavna i odgovara funkcijama koje smo definisali prilikom rada sa drvetima izraza.

```
type* basic_type(enum types ty) {
   type* t = (type*)malloc(sizeof(type));
   /* Obrada greske alokacije */
   t->tag = ty;
   return t;
}

type* error_type() {
  return basic_type(T_ERROR);
}

type* char_type() {
  return basic_type(T_CHAR);
}

type* integer_type() {
  return basic_type(T_inTEGER);
}

type* pointer_type(type* ptype) {
  type* t = basic_type(T_POinTER);
  t->ptype = ptype;
  return t;
}
```

```
type* array_type(int num, type* ptype) {
  type* t = basic_type(T_ARRAY);
  t->atype.num = num;
  t->atype.ptype = ptype;
  return t;
}

int is_integer(type* t) {
  return t->tag == T_inTEGER;
}

int is_array(type* t) {
  return t->tag == T_ARRAY;
}

int is_pointer(type* t) {
  return t->tag == T_POinTER;
}

void print_type(type* t) {
  switch (t->tag) {
  case T_ERROR:
    printf("ERROR");
    break;
  case T_CHAR:
    printf("char");
    break;
  case T_inTEGER:
    printf("integer");
    break;
  case T_POinTER:
    printf("pointer to ");
    print_type(t->ptype);
    break;
  case T_ARRAY:
    printf("array of %d ", t->atype.num);
    print_type(t->atype.ptype);
    break;
  }
}

type* copy_type(type* t) {
  switch(t->tag) {
  case T_ERROR:
    return error_type();
  case T_CHAR:
```

```
    return char_type();
  case T_inTEGER:
    return integer_type();
  case T_POinTER:
    return pointer_type(copy_type(t->ptype));
  case T_ARRAY: {
    type* ptype = copy_type(t->atype.ptype);
    type* res = array_type(t->atype.num, ptype);
    return res;
    }
  }
}

void free_type(type* t) {
  switch(t->tag) {
  case T_POinTER:
    free_type(t->ptype);
    break;
  case T_ARRAY:
    free_type(t->atype.ptype);
    break;
  }
  free(t);
}
```

- Ključna funkcija semantičkog analizatora je ona koja vrši proveru tipa datog izraza.

```
type* typecheck(expr* e) {
  switch(e->tag) {
  case E_NUM:
    /* konstante u izrazima su isključivo tipa integer */
    return integer_type();
  case E_ID: {
    /* tip promenljive određujemo iz globalne tablice simbola */
    type* t = (type*)lookup(e->id);
    if (t == NULL)
      return error_type();
    return copy_type(t);
    }
  case E_OP: {
    switch (e->op.op) {
    /* sabiranje */
    case '+': {
      type* res;
      /* rekurzivno određujemo tipove operanada */
      type* t1 = typecheck(e->op.ops[0]);
```

```
    type* t2 = typecheck(e->op.ops[1]);
    /* ako su oba tipa integer i rezultat je integer */
    if (is_integer(t1) && is_integer(t2))
      res = integer_type();
    /* ostali tipovi se ne mogu sabirati */
    else
      res = error_type();
    free_type(t1); free_type(t2);
    return res;
    break;
    }
  /* dereferenciranje pokazivaca */
  case '^': {
    type* res;
    /* rekurzivno odredjujemo tip operanda */
    type* t1 = typecheck(e->op.ops[0]);
    /* ako je on pokazivacki tip */
    if (is_pointer(t1))
    /* tip izraza odgovara onom tipu na koji pokazivac pokazuje */
      res = copy_type(t1->ptype);
    /* ostali tipovi se ne mogu dereferencirati */
    else
      res = error_type();
    free_type(t1);
    return res;
    }
  /* indeksiranje niza */
  case ARRAY: {
    type* res;
    /* rekurzivno odredjujemo tipove operanda */
    type* t1 = typecheck(e->op.ops[0]);
    type* t2 = typecheck(e->op.ops[1]);
    /* prvi operand mora biti niz, a drugi integer */
    if (is_array(t1) && is_integer(t2))
      /* tip izraza je tip elemenata niza */
      res = copy_type(t1->atype.ptype);
    /* u drugim slucajevima indeksnin pristup nije dopusten */
    else
      res = error_type();
    free_type(t1); free_type(t2);
    return res;
    break;
    }
  }
  }
}
```

```
  return NULL;
}
```

7 Međukod

- Međukod (intermediate code IC, intermediate representation IR, intermediate lanugage IL)
- Prevođenje se vrši po sledećoj shemi: Izvorni kod → Međukod → Ciljni kod.
- Razni front-end i razni back-end mogu da koriste istu IR reprezentaciju, pa i optimizaciju nad IR. Primeri:
  - .NET u kojem se C#, VB, Visual C++ prevode na isti međukod.
  - LLVM koji daje podršku za veliki broj i frond-end i back-end sistema.

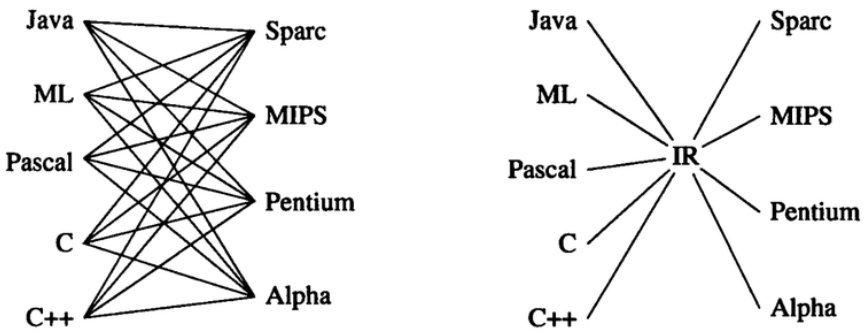


Figure 9: Shema kompilacije bez i sa IR

- Za međukod se kaže da je asembler visokog nivoa.
- Detaljniji je od izvornog koda, ali manje detaljan od ciljnog.
  - neograničen broj imena registara (privremene promenljive)
  - instrukcije višeg nivoa
  - nekada dostupne funkcije
  - nekada zadržani tipovi
- Primer LLVM (prikazani su originalni C kod i odgovarajući prevod na LLVM):

```
int mul_add(int x, int y, int z) {
   return x * y + z;
}
```

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

- Nekoliko vrsta IR:
  - stablo, DAG
  - postfiksna notacija
  - troadresni kod (TAC, 3AC)
  - graf toka kontrole na nivou bazičnih blokova (basic blocks, control flow graph)
  - SSA (Static Single Asignment Form)

7.1 Troadresni kod (TAC, 3AC)

- Svaka instrukcija najviše tri adrese (dva operanda i rezultat operacije)
- Dopušteni uslovni i bezuslovni skokovi
- Dopušteni pozivi funkcija (sa jednim argumentom)
- Uvode se imena podizraza (tzv. *privremene promenljive*, engl. *temporaries*)

```
i = 0;
while (i < 10)
  i += x + y * z;
print(i + 5);
```

```
(1)  i := 0
(2)  ifFalse i < 10 goto 7
(3)  t1 := y * z
(4)  t2 := x + t1
(5)  i := i + t2
(6)  goto 2
(7)  t3 := i + 5
(8)  print(t3)
```

- Pristup nizu zahteva da se neka izračunavanja rade u troadresnom kodu

```
a[i] = 3;
```

```
t1 := 4 * i;
t2 := a + t1;
memory[t2] := 3;
```

- Sintaksa jedne od mogućih varijanti troadresnog koda:

```
P : P S
  |
  ;

S : id := e op e     /* Dodela rezultata binarne operacije */
  | id := op e       /* Dodela rezultata unarne operacije */
  | id := e          /* Kopiranje */
  | push e           /* Pristup steku */
  | id := pop
  | L:                     /* Labela */
  | if e relop e goto L    /* Uslovni skok */
  | goto L                 /* Bezuslovni skok */
  | call p(e, ..., e)      /* poziv funkcije */
  | id := id[e]            /* nizovi */
  | id[e] := e
  | id := *id              /* pokazivaci */
  | id := &id
  ;

e : id
  | const
  ;
```

7.2 Prevođenje aritmetičkih izraza u TAC

Troadresni kod za izraze se obično generiše:

- na osnovu stabla apstraktne sintakse ili
- na osnovu DAG-a (usmerenog acikličkog grafa)

Na primer:

```
a = b*(-c) + b*(-c);
```

- Kod generisan na osnovu stabla:

```
t1 := -c;
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

- Kod generisan na osnovu DAG-a:

```
t1 := -c
t2 := b * t1
```

```
    t3 := t2 + t2
    a := t3
```

Postupak generisanja koda se može opisati i atributskom gramatikom (npr. obrada operatora sabiranja, konstanti i promenljivih):

```
e : e1 '+' e2 {
   e.place = newtemp();
   e.code = e1.code || e2.code ||
            gen(e.place, ':=', e1.place, '+', e2.place);
}

e : c {
   e.place = c;
   e.code = '';
}

e : x {
   e.place = x;
   e.code = '';
}
```

Atribut `place` predstavlja naziv promenljive u koju se smešta rezultat, a atribut `code` predstavlja string koji sadrži troadresni kod (funkcija `gen` gradi troadresni kod nadovezivanjem sitnijih delova).

7.3 Prevođenje logičkih izraza i kontrole toka u TAC

- Logički izrazi se prevode na dva načina:
  1. u troadresni kod koji izračunava njihovu logičku vrednost (uz primenu logičkih operatora u troadresnom kodu)
  2. u troadresni kod koji pomoću skokova određuje kontrolu toka (bez primene logičkih operatora u troadresnom kodu)
- Vrsta prevođenja zavisi od konteksta (npr. da li je logički izraz sa desne strane dodele ili je uslov neke naredbe).
- Treba obratiti pažnju na lenjo izračunavanje.

7.3.1 Prevođenje logičkih izraza u kod izračunava njihovu logičku vrednost

- Postupak analogan prevođenju aritmetičkih izraza.
- Primer u kome se logički izraz prevodi u logičku vrednost primenom logičkih operatora - ne uzima u obzir lenjo izračunavanje.

```
x = a || b && !c;
```

```
t1 := not c
t2 := b and t1
x  := a or t2
```

Pod pretpostavkom da se logički izrazi prevode u kod koji izračunava logičke vrednosti, prevođenje naredbi kontrole toka možemo opisati pomoću sledeće atributske gramatike.

```
S : while E do S1 {
   l_begin = newlabel();
   l_after = newlabel();
   S.code = label(l_begin) ||
            E.code ||
            gen('ifFalse', E.place, 'goto', l_after) ||
            S1.code ||
            gen('goto', l_begin) ||
            label(l_after);
}
```

7.3.1.1 Implementacija u C-u

- Implementacija u C-u compile3address.zip

```
/* Kompilacija u troadresni kod */
void compile3address(nodeType* p) {
  nodeType* res = compile3address_(p);
  freeNode(res);
}

/* Kompilacija u troadresni kod - ako se kompilira izraz, funkcija
   vraca novi cvor koji predstavlja konstantu ili promenljivu (originalnu
   ili privremenu) koja sadrzi rezultat izraza */
nodeType* compile3address_(nodeType* p) {
  /* brojac slobodnih labela */
  static int label;
  /* brojac slobodnih privremenih promenljivih */
  static int tempNum;
  switch (p->type) {
  case typeCon:
    /* konstanta predstavlja svoju vrednost */
    return copy(p);
  case typeId:
    /* promenljiva predstavlja svoju vrednost */
    return copy(p);
  case typeOpr:
```

```
if (p->opr.oper == PRinT) { /* naredba stampanja */
  /* kompiliramo argument funkcije print i kao rezultat dobijamo
     konstantu ili promenljivu koja ce sadrzati vrednost tog
     argumenta */
  nodeType* res = compile3address_(p->opr.op[0]);
  /* generisemo instrunkciju stampanja te konstante tj. promenljive */
  printf("print "); print(res); printf("\n");
  /* oslobadjamo memoriju zauzetu privremenim cvorom */
  freeNode(res);
  return NULL;
} if (p->opr.oper == ';') { /* nadovezivanje dve naredbe */
  /* kompiliramo prvu naredbu */
  compile3address_(p->opr.op[0]);
  /* kompiliramo drugu naredbu */
  compile3address_(p->opr.op[1]);
  /* naredbe nemaju vrednost, pa vracamo NULL */
  return NULL;
} else if (p->opr.oper == IF) { /* naredba grananja if */
  /* rezervisemo jednu slobodnu labelu za kraj naredbe grananja */
  int lblAfter = ++label;
  /* uslov i telo naredbe grananja */
  nodeType* cond = p->opr.op[0];
  nodeType* body = p->opr.op[1];

  /* kompiliramo levu i desnu stranu uslova naredbe if */
  nodeType* res0 = compile3address_(cond->opr.op[0]);
  nodeType* res1 = compile3address_(cond->opr.op[1]);

  /* res0 i res1 su konstante ili promenljive koje sadrze vrednosti
     leve i desne strane uslova naredbe if */

  /* generisemo instrukciju koja preskace telo naredbe if ako
     uslov nije ispujen */
  printf("ifFalse ");
  print(res0); printf(" %c ", cond->opr.oper); print(res1);
  printf(" goto labela%d\n", lblAfter);

  /* kompiliramo telo naredbe if - posto je to naredba, a ne
     izraz, povratna vrednost nije relevantna (ocekujemo da se
             vrati NULL) */
  compile3address_(body);

  /* generisemo labelu na kraju naredbe if */
  printf("labela%d:\n", lblAfter);

  /* if je naredba, a ne izraz, pa vracamo NULL */
```

```
  return NULL;
} else if (p->opr.oper == WHILE) { /* naredbe while */
  /* rezervisemo dve labele: jednu za pocetak i jednu za kraj */
  int lblBefore = ++label;
  int lblAfter = ++label;

  /* generisemo labelu na pocetku naredbe while */
  printf("labela%d:\n", lblBefore);

  /* uslov i telo naredbe grananja */
  nodeType* cond = p->opr.op[0];
  nodeType* body = p->opr.op[1];

  /* kompiliramo levu i desnu stranu uslova naredbe if */
  nodeType* res0 = compile3address_(cond->opr.op[0]);
  nodeType* res1 = compile3address_(cond->opr.op[1]);

  /* res0 i res1 su konstante ili promenljive koje sadrze vrednosti
 leve i desne strane uslova naredbe if */

  /* generisemo instrukciju koja preskace telo naredbe while ako
     uslov nije ispujen */
  printf("ifFalse ");
  print(res0); printf(" %c ", cond->opr.oper); print(res1);
  printf(" goto labela%d\n", lblAfter);

  /* kompiliramo telo naredbe while - posto je to naredba, a ne
     izraz, povratna vrednost nije relevantna (ocekujemo da se
             vrati NULL) */
  compile3address_(body);

  /* generisemo bezuslovni skok na pocetak petlje */
  printf("goto labela%d\n", lblBefore);

  /* generisemo labelu na kraju naredbe while */
  printf("labela%d:\n", lblAfter);

  /* while je naredba, a ne izraz, pa vracamo NULL */
  return NULL;
} else if (p->opr.oper == '=') { /* naredba dodele */
  /* kompiliramo desnu stranu */
  nodeType* res = compile3address_(p->opr.op[1]);
  /* rezultat na desnoj strani je smesten u konstantu ili
     promenljivu (originalnu ili privremenu) res */
  /* generisemo instrukciju dodele koja promenljivoj sa leve
     strane originalne dodele dodeljuje res */
```

```
      print(p->opr.op[0]); printf(" = "); print(res); printf("\n");
      /* oslobadjamo memoriju zauzetu pomocnim cvorom vracenom iz
         rekurzivnog poziva */
      freeNode(res);
      /* naredba dodele je naredba, a ne izraz, pa vracamo NULL */
      return NULL;
    } else {
       if (p->opr.nops == 2) { /* binarni aritmeticki operatori */
           /* rezervisemo novu privremenu promenljivu */
           int temp = tempNum++;
           /* odredjujemo joj ime */
           char tempName[MAX_ID + 1];
           sprintf(tempName, "t%d", temp);
           /* kompiliramo levi i desni operand aritmetickog operatora */
           nodeType* res1 = compile3address_(p->opr.op[0]);
           nodeType* res2 = compile3address_(p->opr.op[1]);
           /* res1 i res2 su konstante ili promenljive (originalne ili
              pomocne) koje  sadrze rezultat levog i desnog operanda */

           /* generisemo troadresnu dodelu */
           printf("%s := ", tempName);
           print(res1); printf(" %c ", p->opr.oper); print(res2); printf("\n");

           /* oslobadjamo memoriju zauzetu pomocnim cvorovima vracenim iz
              rekurzivnih poziva */
           freeNode(res1); freeNode(res2);

           /* rezultat se nalazi u pomocnoj promenljivoj */
           return id(tempName);
       }
    }
    break;
  }
}
```

7.3.2 Prevođenje logičkih izraza u skokove

- Primer uzima u obzir lenjo izračunavanje i prevodi izraz u kontrolu toka

```
if (a < b || (c < d && e < f)) {
   telo1
} else {
   telo2
}
```

- Prevod prethodnog programa.

```
  if a < b goto Ltrue
  if c < d goto L1
  goto Lfalse
L1:
  if e < f goto Ltrue
  goto Lfalse
Ltrue:
  ....    /* telo1 */
  goto LKraj
Lfalse:
  ....    /* telo2 */
LKraj:
```

- Primer u kome se logički izraz prevodi u skokove (uzima se u obzir lenjo izračunavanje).

```
x = a < b || c < d && e < f;
```

```
  if a < b goto Ltrue
  ifFalse c < d goto Lfalse
  ifFalse e < f goto Lfalse
LTrue:
  x := true
  goto Lkraj
LFalse:
  x := false
LKraj:
```

- Atributska gramatika kojom se opisuje prevođenje logičkih izraza i kontrole toka u skokove.
  - Naredbe imaju sintetisani atribut `code` koji sadrži rezultujući troadresni kod i nasleđeni atribut `Lnext` koji sadrži adresu labele koja označava mesto u kodu neposredno iza naredbe.
  - Logički izrazi imaju nasleđene atribute `Ltrue` i `Lfalse` koji sadrže adresu labela na kojima se nalazi kod na koji se prelazi ako je vrednost izraza tačno tj. netačno.

```
P : S {
   S.Lnext = newlabel()
   P.code = S.code || label(S.Lnext)
}

S : if B then S1 {
   B.Ltrue = newlabel()
   B.Lfalse = S.Lnext
   S1.Lnext = S.Lnext
   S.code = B.code || label(B.true) || S1.code
}
```

```
S : while B do S1 {
   begin = newlabel()
   B.Ltrue = newlabel();
   B.Lfalse = S.Lnext
   S1.next = begin
   S.code = label(begin) ||
            B.code ||
            label(B.Ltrue)
            S1.code ||
            gen('goto', begin)
}

S : S1 S2 {
   S1.next = newlabel()
   S2.next = S.next
   S.code = S1.code || label(S1.next) || S2.code
}

B : B1 || B2 {
   B1.Ltrue = B.Ltrue
   B1.Lfalse = newlabel()
   B2.Ltrue = B.Ltrue
   B2.Lfalse = B.Lfalse
   B.code = B1.code || label(B1.false) || B2.code
}

B : B1 && B2 {
   B1.Ltrue = newlabel()
   B1.Lfalse = B.Lfalse
   B2.Ltrue = B.Ltrue
   B2.Lfalse = B.Lfalse
   B.code = B1.code || label(B1.true) || B2.code
}

B : E1 rel E2 {
   B.code = E1.code || E2.code
            || gen('if' E1.place rel.op E2.place 'goto' B.Ltrue)
            || gen('goto' B.Lfalse)
}
```

- Primenimo prethodni algoritam na sledeći primer.

```
if (x < 100 || x > 200 && x != y)
   x = 0;
```

- Redosled operacije je sledeći:

- Naredba `S` (naredba `if`) dobija vrednost atributa `S.next = L1` (nova labela).
- Generiše se kod za naredbu `S`, a zatim se ispisuje labela `L1`.
- Generisanje koda za naredbu `S` teče na sledeći način.
  - Uslov `B` naredbe `if` dobija vrednosti atributa `B.Ltrue = L2` (nova labela) i `B.Lfalse = L1` (nju nasleđuje od `S.next`).
  - Slično, naredba `x = 0` dobija atribut `S1.next = L1`.
  - Prelazi se na generisanje koda za uslov `B`, zatim se generiše labela `L2` i nakon toga se prevodi `S1` (to daje `L2: x := 0`).
  - Generisanje koda za uslov `B` se vrši na sledeći način.
    * Pošto je u pitanju disjunkcija, prvi uslov `x < 100` dobija atribute `B1.Ltrue = L2` (nasleđuje je od `B`) i `B2.Lfalse = L3` (nova labela).
    * Slično drugi uslov `x > 200 && x != y` dobija atribut `B2.Ltrue = L2` i `B2.Lfalse = L1` (nasleđuje ih od `B`).
    * Generiše se kod za `B1` tj. `x < 100`, zatim se ispisuje labela `B1.Lfalse` tj. `L3` i na kraju se prelazi na generisanje koda za `B2`.
    * Generisanje koda za `B1` teče na sledeći način.
      · Generiše se `if x < 100 goto L2` (jer je to `B1.Ltrue` jednako `L2`) Generiše se `goto L3` (jer je `B1.Lfalse` jednako `L3`).
    * Generisanje koda za `B2` teče na sledeći način.
      · Pošto je u pitanju konjunkcija, prvi uslov `x > 200` dobija atribute `B3.Ltrue = L4` (nova labela) i `B3.Lfalse = L1` (nasleđuje je od `B2`).
      · Slično, drugi uslov `x != y` dobija atribute `B4.Ltrue = L2` i `B4.Lfalse = L1` (nasleđuje ih od `B2`).
      · Prevodi se kod za `B3`, zatim se postavlja labela `L4`, a zatim se prevodi kod za ‘B4.
      · Prevođenjem koda za `B3` se dobija `if x > 200 goto L4` (jer je `B3.Ltrue = L4`) i zatim `goto L1` jer je `B3.Lfalse = L1`.
      · Prevođenjem koda za `B4` se dobija `if x != y goto L2` (jer je `B4.Ltrue = L2`) i zatim `goto L1` jer je `B4.Lfalse = L1`.
- Dakle, prevod prethodnog programa je sledeći troadresni kod.

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x := 0
L1:
```

7.3.2.1 Implementacija u Haskell-u

- Prikažimo sada implementaciju programa koji generiše troadresni međukod od apstraktnog sintakstičkog stabla, zasnovanog na gore opisanom postupku.

```
import Data.List
import Control.Monad.State

-- Tipovi podataka za predstavljanje apstraktnost sintaksickog stabla
-- naseg imperativnog jezika

data ArithOp = Plus | Times
instance Show ArithOp where
  show Plus = "+"
  show Times = "*"

-- Aritmeticki izrazi
data ArithExpr =
    Const Int
  | Id String
  | Op ArithExpr ArithOp ArithExpr
   deriving Show

-- Relacijski operatori
data RelOp = Eq | Neq | Lt | Gt | Leq | Geq
-- Prikaz relacijskih operatora
instance Show RelOp where
  show Eq = "=="
  show Neq = "!="
  show Lt = "<"
  show Gt = ">"
  show Leq = "<="
  show Geq = ">="

-- Logicki izrazi
data BoolExpr =
    Or BoolExpr BoolExpr
  | And BoolExpr BoolExpr
  | Rel ArithExpr RelOp ArithExpr
    deriving Show

-- Naredbe
data Stmt =
    While BoolExpr Stmt
  | If BoolExpr Stmt
  | IfElse BoolExpr Stmt Stmt
```

```
  | Assign String ArithExpr
  | Seq Stmt Stmt
    deriving Show

-- Globalno stanje vodi racuna o brojevima labela i privremenih promenljivih
data Counters = Counters { nextLabel :: Int, nextTemp :: Int } deriving Show

-- Funkcija kojom ce se uzimati naredna slobodna labela (ona uvecava
-- globalni brojac labela)
newLabel :: State Counters Int
newLabel = state $ \(Counters l t) -> (l, Counters (l+1) t)

-- Funkcija kojom ce se uzimati naredno slobodno ime privremene
-- promenljive (ona uvecava globalni brojac privremenih promenljivih)
newTemp :: State Counters Int
newTemp = state $ \(Counters l t) -> (t, Counters l (t+1))


-- Uvlacenje troadresnih naredbi (radi preglednosti)
indent :: String -> String
indent s = "    " ++ s

-- Funkcija koja gradi ime privremene promenljive na osnovu njenog rednog broja
temp :: Int -> String
temp n = "t" ++ show n

-- funkcija koja gradi troadresnu instrukciju dodele
assign :: String -> ArithOp -> String -> String -> String
assign e1 op e2 res =
  res ++ " := " ++ e1 ++ " " ++ show op ++ " " ++ e2

-- Funkcija koja prevodi dati aritmeticki izraz u troadresni kod Ona
-- vraca dobijeni kod i ime promenljive u koju je na kraju smestena
-- vrednost izraza
compileArithExpr :: ArithExpr -> State Counters ([String], String)
compileArithExpr (Id x) = do return ([], x)
compileArithExpr (Const n) = do return ([], show n)
compileArithExpr (Op e1 op e2) = do
  (e1Code, e1Place) <- compileArithExpr e1
  (e2Code, e2Place) <- compileArithExpr e2
  t <- newTemp
  let code = e1Code ++ e2Code ++
             [indent $ assign e1Place op e2Place (temp t)]
  return (code, temp t)

-- Funkcije koje grade tekstualnu reprezentaciju troadresne kontrole toka
```

```haskell
-- Fukcija koja gradi tekst labele na osnovu njenog rednog broja
label :: Int -> String
label n = "L" ++ show n ++ ":"

-- Fukcija koja gradi tekst naredbe skoka na osnovu rednog broja
-- labele na koju se skace
goto :: Int -> String
goto n = "goto L" ++ show n

-- Funkcija koja gradi tekst narebe uslovnog skoka na osnovu datih
-- relacijskog operatora, njegovih operanada i rednog broja labele na
-- koju se skace
ifGoto :: String -> RelOp -> String -> Int -> String
ifGoto e1 op e2 l =
  "if " ++ e1 ++ " " ++ show op ++ " " ++ e2 ++ " " ++ "goto L" ++ show l


-- Funkcija vrsi prevodjenje naredbi u troadresni kod
compileStmt :: Stmt -> Int -> State Counters [String]

compileStmt (Assign x e) l_next = do
  (eCode, ePlace) <- compileArithExpr e
  let code = eCode ++ [indent $ "x = " ++ ePlace]
  return code

compileStmt (Seq s1 s2) l_next = do
  l1_next <- newLabel
  s1_code <- compileStmt s1 l1_next
  s2_code <- compileStmt s2 l_next
  let code = s1_code ++ [label l1_next] ++ s2_code
  return code

compileStmt (While b s1) l_next = do
  l_begin <- newLabel
  l_btrue <- newLabel
  bCode <- compileBoolExpr b l_btrue l_next
  s1Code <- compileStmt s1 l_next
  let code = [label l_begin] ++
             bCode ++
             [label l_btrue] ++
             s1Code ++
             [indent $ goto l_begin]
  return code

compileStmt (If b s1) l_next = do
```

```
  l_btrue <- newLabel
  bCode <- compileBoolExpr b l_btrue l_next
  s1Code <- compileStmt s1 l_next
  let code = bCode ++
             [label l_btrue] ++
             s1Code
  return code

compileStmt (IfElse b s1 s2) l_next = do
  l_btrue <- newLabel
  l_bfalse <- newLabel
  bCode <- compileBoolExpr b l_btrue l_bfalse
  s1Code <- compileStmt s1 l_next
  s2Code <- compileStmt s2 l_next
  let code = bCode ++
             [label l_btrue] ++
             s1Code ++
             [indent $ goto l_next] ++
             [label l_bfalse] ++
             s2Code
  return code

-- Funkcija vrsi prevodjenje logickih izraza u troadresni kod
compileBoolExpr ::  BoolExpr -> Int -> Int -> State Counters [String]

compileBoolExpr (Rel e1 op e2) l_true l_false = do
  (e1Code, e1Place) <- compileArithExpr e1;
  (e2Code, e2Place) <- compileArithExpr e2;
  let code = e1Code ++ e2Code ++
             [indent $ ifGoto e1Place op e2Place l_true] ++
             [indent $ goto l_false]
  return code

compileBoolExpr (Or b1 b2) l_true l_false = do
   l_b1false <- newLabel
   b1Code <- compileBoolExpr b1 l_true l_b1false
   b2Code <- compileBoolExpr b2 l_true l_false
   let code = b1Code ++ [label l_b1false] ++ b2Code
   return code

compileBoolExpr (And b1 b2) l_true l_false = do
   l_b1true <- newLabel
   b1Code <- compileBoolExpr b1 l_b1true l_false
   b2Code <- compileBoolExpr b2 l_true l_false
   let code = b1Code ++ [label l_b1true] ++ b2Code
   return code
```

```haskell
-- centralna funkcija prevodjenja
compile :: Stmt -> State Counters [String]
compile s = do
  l1 <- newLabel
  s_code <- compileStmt s l1
  let code = s_code ++ [label l1]
  return code

-- Funkcija koju korisnik poziva
compileProgram :: Stmt -> [String]
compileProgram s = fst $ runState (compile s) (Counters 1 1)


-- Testiramo rad programa na jednom malom primeru

--   while (x >= 10 || x < y)
--      x = z + x + 1;
--   y = 5;

prg1 :: Stmt
prg1 = Seq
         (While (Or (Rel (Id "x") Geq (Const 10)) (Rel (Id "x") Lt (Id "y")))
                (Assign "x" (Op (Id "z") Plus (Op (Id "x") Plus (Const 1)))))
         (Assign "y" (Const 5)))

main = putStrLn $ intercalate "\n" $ compileProgram prg1
```

8 Bazični blokovi i graf kontrole toka

- Kao priprema za dalje faze (optimizaciju, alokaciju registara, ...) troadresni kod se deli na manje celine *bazične blokove*, (engl. *basic blocks*) koje se povezuju u *graf kontrole toka* (engl. *control flow graph*, *CFG*).

8.1 Bazični blokovi

- Bazični blok sadrži niz troadresnih instrukcija koje se uvek izvršavaju sve zajedno.
- U bazični blok se ne može uskočiti GOTO instrukcijom tj. njegovo izvršavanje kreće od prve instrukcije i iz njega se ne može iskočiti (osim u slučaju poslednje instrukcije).

```
1:   i := 1
2:   j := 1
3:   t1 := 10 * i
4:   t2 := t1 + j
5:   t3 := 8 * t2
6:   t4 := t3 - 88
7:   a[t4] = 0.0
8:   j := j + 1
9:   if j <= 10 goto (3)
10:  i := i + 1
11:  if i <= 10 goto (2)
12:  i := 1
13:  t5 := i - 1
14:  t6 := 88 * t5
15:  a[t6] := 1.0
16:  i := i + 1
17:  if i <= 10 goto (13)
```

- Algoritam podele na bazične blokove se zasniva na određivanju *vođa blokova* tj. instrukcija kojima počinju blokovi. Svaki se blok onda prostire od instrukcije vođe (uključujući nju) do naredne instrukcije vođe (bez nje) ili do kraja koda.
  - Prva instrukcija u kodu je vođa (1 u datom primeru).
  - Instrukcija na koju se skače je vođa (2, 3 i 13 u datom primeru)
  - Instrukcija neposredno posle instrukcije skoka je vođa (10 i 12 u datom primeru).
- Na osnovu ovoga, podela prethodnog primera na blokove je sledeća.

```
1:   i := 1

2:   j := 1

3:   t1 := 10 * i
4:   t2 := t1 + j
5:   t3 := 8 * t2
6:   t4 := t3 - 88
7:   a[t4] = 0.0
8:   j := j + 1
9:   if j <= 10 goto (3)

10:  i := i + 1
11:  if i <= 10 goto (2)

12:  i := 1

13:  t5 := i - 1
```

```
14:  t6 := 88 * t5
15:  a[t6] := 1.0
16:  i := i + 1
17:  if i <= 10 goto (13)
```

- Zašo su važni osnovni blokovi? Zato što olakšavaju analizu koda (npr. u svrhu omptimizacije).

```
L:
   t := 2 * x;
   w := t + x;
   if w > 0 then goto L'
```

- Da li je moguće zameniti prve dve instrukcije sa `w := 3 * x`? Jeste, zato što se na `w := t + x` ne može skočiti tj. zato što smo sigurni da se pre `w := t + x` izvršila `t := 2 * x` (nismo mogli uskočiti u sredinu).

8.2 Graf kontrole toka

- Bazični blokovi povezani na osnovu skokova čine *graf kontrole toka* (engl. *control flow graph*, *CFG*)
- Umesto brojeva instrukcija grane u grafu kontrole toka sadrže brojeve bazičnih blokova na koje se skače.
- Petljama u programu odgovaraju petlje u grafu - postoji algoritam za detekciju petlji.
- Prilikom prevođenja za svaku funkciju se generiše zaseban graf kontrole toka.
- Graf kontrole toka se predstavlja korišćenjem bilo koje uobičajene reprezentacije grafa (pri čemu se u svakom čvoru čuva neka reprezentacija niza troadresnih instrukcija).

9 Oblik statičke jedinstvene dodele (SSA)

- Oblik statičke jedinstvene dodele (single static assignment, SSA) je svojstvo međukoda u kome se zahteva:
  - da se vrednost svakoj promenljivoj dodeljuje na tačno jednom mestu u programu
  - da je svaka promenljiva definisana pre nego što se upotrebi
- Naziv *statička* jedinstvena dodela potiče od toga što se promenljivoj vrednost dodeljuje na tačno jednom mestu u programu, ali ta instrukcija dodele može da se nalazi u petlji i da se izvršava više puta. Čisti funkcionalni jezici traže i više od ovoga - *dinamička* jedinstvena dodela podrazumeva

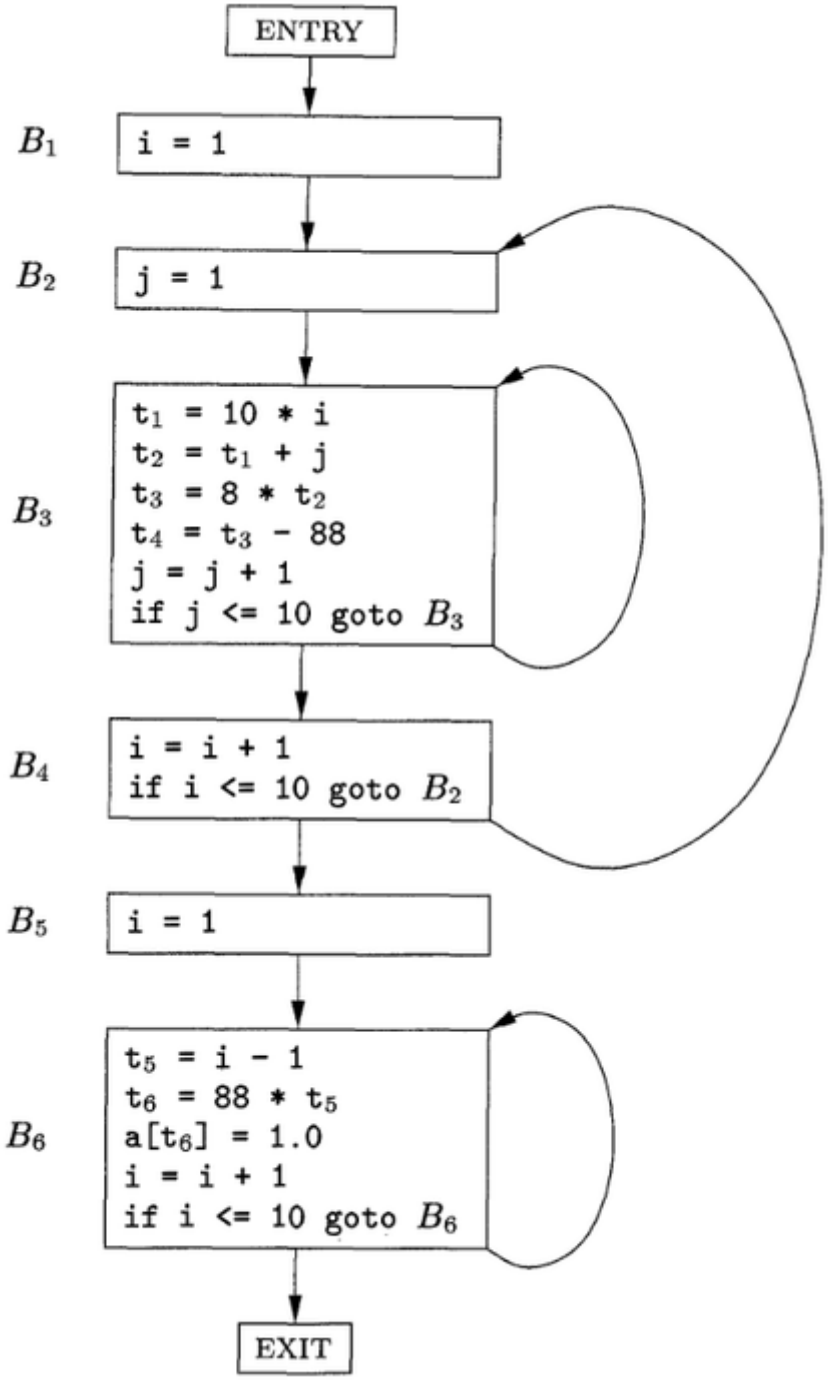


Figure 10: Graf kontrole toka za prethodni troadresni međukod

da se vrednost promenljivoj dodeljuje tačno jednom tokom izvršavanja programa (jednom kada se promenljivoj dodeli vrednost, ona se ne može promeniti).

- SSA je donekle vezana za funkcionalno programiranje i postoje algoritmi konverzije iz SSA u *medjureprezentaciju u obliku funkcionalnog programa* (engl. *functional intermediate form*) i obratno.
- SSA oblik međukoda olakšava mnoge dalje faze kompilacije (pre svega, optimizacije):
  - propagaciju konstanti
  - eliminaciju mrtvog koda
  - redukciju snage
  - alokaciju registara
  - ...
- Više reči o ovome u delu o optimizaciji.
- SSA oblik nam omogućava da veoma efikasno odredimo sledeće podatke:
  - za svaku instrukciju:
    * blok u kome se ona nalazi
    * prethodnu instrukciju u tom bloku
    * promenljivu koja se tom instrukcijom definiše (sa leve strane)
    * promenljive koje se koriste u toj instrukciji (sa desne strane)
  - za svaku promenljivu:
    * jedinstvenu instrukciju u kodu kojom se ona definiše
    * listu instrukcija u kojima se ona koristi
  - za svaki blok:
    * listu instrukcija
    * uređenu listu prethodnika
    * naslednika (jednog ili dva, u zavisnosti da li na kraju postoji uslovni skok)

9.1 Prevođenje u SSA

- Efikasna procedura prevođenja u SSA nije jednostavna. Navešćemo samo neke osnovne ideje.
- Postojeće promenljive se *verzionišu* - prilikom svake nove upotrebe uvodi se novi indeks.

```
x := 1
x := 2
y := x
```

```
x1 := 1
x2 := 1
```

```
y1 := x2
```

- Posmatrajmo sledeći primer sa više bazičnih blokova i složenijim grafom kontrole toka.

```
  x := 5
  x := x - 3
  ifFalse x < 3 goto L1
  y := x * 2
  goto L2
L1:
  y := x - 3
L2:
  w := x - y
  z := x + y
```

- Nakon verzionisanja dobijamo sledeći kod.

```
  x1 := 5
  x2 := x1 - 3
  ifFalse x2 < 3 goto L1
  y1 := x2 * 2
  goto L2
L1:
  y2 := x2 - 3
L2:
  w := x2 - y?
  z := x2 + y?
```

- Koja se vrednost promenljive `y` koristi u bloku `L2`? To zavisi od toga iz kog smo bloka došli u blok `L2`.
- SSA uvodi specijalnu instrukciju u međukod ϕ tj. `phi` (*phony*) koja opisuje ovakve situacije.

```
  x1 := 5
  x2 := x1 - 3
  ifFalse x2 < 3 goto L1
  y1 := x2 * 2
  goto L2
L1:
  y2 := x2 - 3
L2:
  y3 := phi(y1, y2)
  w := x2 - y3
  z := x2 + y3
```

- ϕ treba nekada uvesti i u slučaju petlji.

```
  a := 0
```

```
L1:
    b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

```
    a1 := 0
L1:
    a3 := phi(a1, a2)
    b1 := phi(b0, b2)
    c2 := phi(c0, c1)
    b2 := a3 + 1
    c1 := c2 + b2
    a2 := b2 * 2
    if a2 < n goto L1
    return c1
```

- Ostaje pitanje gde se ubacuju ϕ instrukcije?
  - Naivan način je da se ubacuju na svakom mestu gde u neki čvor u CFG ulaze bar dve grane, ali to je nepotrebno jer je moguće da se u oba čvora prethodnika koristi ista verzija promenljive.
  - Postoje efikasni algoritmi za određivanje gde je zaista potrebno postaviti ϕ (zasnovani su na analizi kroz koje blokove grafa kontrole bloka se može/mora proći da bi se stiglo do nekog bloka).
    * Čvor d *dominira* nad čvorom n (d je *dominator* čvora n) ako svaka putanja od ulaznog čvora do n mora da prođe kroz čvor d (pre nego što se izvrši n mora se proći kroz d).
    * Svaki čvor dominira nad samim sobom.
    * Čvor d *striktno dominira* nad n ako su različiti i d dominira nad n.
    * *Granica dominacije* (engl. *domination frontier*) čvora d je skup čvorova na kojima prestaje dominacija čvora d - to je skup svih čvorova n takvih da d dominira nad neposrednim prethodnikom čvora n, ali ne dominira stirktno nad n. Iz perspektive čvora d to su oni čvorovi u koje počinju da ulaze putanje koje ne moraju da prođu kroz d. Granicu dominacije je moguće efikasno odrediti Lengauer-Tardžanovim algoritmom.
    * Ako je neka promenljiva definisana u čvoru d, onda je za nju potrebno uvesti ϕ instrukcije tačno u čvorovima granice dominacije čvora d.
    * Postoje efikasni algoritmi izračunavanja granice dominacije, ali se time nećemo baviti.
    * Postoje i varijante u kojima se vrši optimizacija broja uvedenih ϕ funkcija (često u zavisnosti od toga kako se promenljive kasnije koriste).

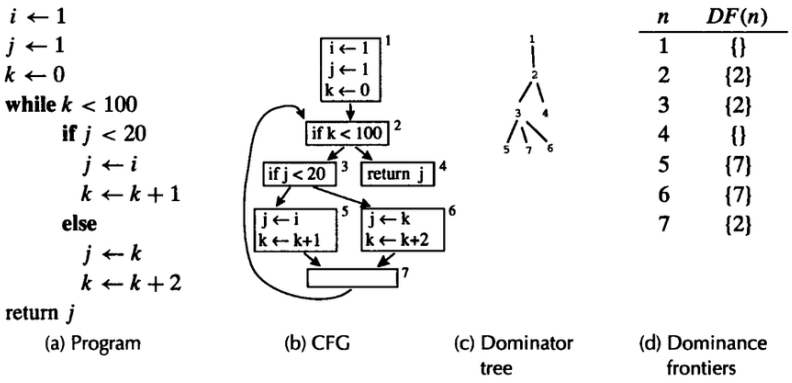


Figure 11: Prevođenje u SSA

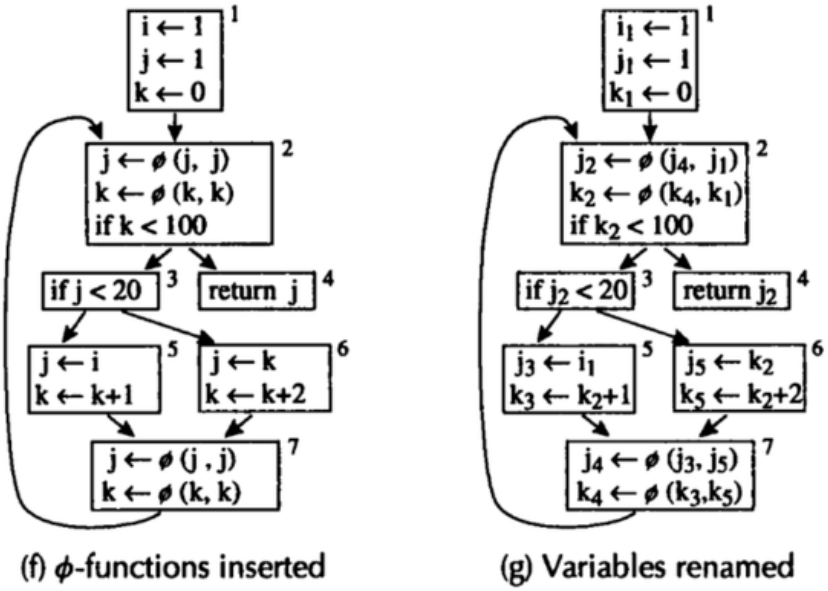


Figure 12: Prevođenje u SSA

- Nakon optimizacije, prilikom generisanja stvarnog koda, ϕ instrukcije se uklanjaju (time se onda gubi SSA oblik).
  - Najjednostavniji (ali ne i efikasan) način je izdizanje ϕ instrukcija u prethodne blokove.

```
  x1 := 5                          x1 := 5
  x2 := x1 - 3                     x2 := x1 - 3
  ifFalse x2 < 3 goto L1           ifFalse x2 < 3 goto L1
  y1 := x2 * 2                     y1 := x2 * 2
  goto L2                          y3 := y1
                                   goto L2
L1:                              L1:
  y2 := x2 - 3                     y2 := x2 - 3
                                   y3 := y2
L2:                              L2:
y3 := phi(y1, y2)                  w := x2 - y3
w := x2 - y3                       z := x2 + y3
z := x2 + y3
```

- LLVM koristi isključivo međukod u SSA obliku. Prikažimo na primeru ϕ instrukciju u LLVM.

```
int g(int x);

int f(int a, int b, bool flag) {
  if (flag)
    return g(a+b);
  else
    return a+b;
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
  entry:
    %c = add i32 %a, %b
    br i1 %flag, label %then, label %else

  then:
    %d = call i32 @g(i32 %c)
    ret i32 %d

  else:
    ret i32 %c
}
```

```
declare i32 @g(i32 %x)
```

```
define i32 @f(i32 %a, i32 %b, i1 %flag) {
  entry:
    %c = add i32 %a, %b
    br i1 %flag, label %then, label %else

  then:
    %d = call i32 @g(i32 %c)
    br label %end

  end:
    %result = phi i32  [ %entry, %c],
                       [ %then, %d ]
    ret i32 %result
}
```

10 Optimizacija

- Optimizacija je i dalje jedno od najizazovnijih pitanja kompilacije (puno istraživanja, praktičnog razvoja optimizatora, …).
- Termin nesretan (kod nije *optimalan*). Bolje reći "poboljašanje".
- Optimizuje se po različitim kriterijumima (procenjeno vreme izvršavanja, veličina koda, …)
- Potrebno je zadržati *ekvivalentnost* između polaznog i transformisanog koda (za iste vrednosti ulaznih podataka oba koda moraju da daju iste vrednosti izlaznih podataka). Kažemo da optimizacija mora da *očuva semantiku programa* (engl. *semantic preserving transformations*).
- Postoji balans između vremena potrebnog za optimizaciju i kvaliteta optimizovanog koda.
- Neke optimizacije su opšte i ne zavise od ciljne arhitekture (izvršavaju se na međureprezentaciji), a neke su specifične i zavise od ciljne arhitekture (izvršavaju se pre konačnog generisanja koda).
- Najkompleksnije optimizacije se obično ne rade (teško se implementiraju, troše puno vremena u kompilaciji, ne doprinose previše) - cilj je uvek maksimizirati dobit za najmanji trošak.
- Prednji deo kompilatora i generator međukoda mogu slobodno da generišu kod koji je prilično veliki i neefikasan (ali pravilnog oblika) računajući na to da će optimizator taj kod prilično pročistiti.
- Neke redundantnosti u programu su posledica činjenice da se programer izražava na jeziku visokog nivoa i od korisnika je zapravo sakrivena mogućnost da se izrazi na efikasniji način (računajući na to da će

efikasnost doneti kompilatorska optimizacija). Npr. C kod `x = a[j] * a[j]` se prevodi u troadresni kod u kome se za svaki pristup nizu iznova izračunava pomeraj u bajtovima (i to će sigurno biti naknadno optimizovano).

```
t1 := 4 * j
t2 := a[t1]
t3 := 4 * j
t4 := a[t3]
x := t2 * t4
```

- Optimizacija se vrši na nekoliko različitih nivoa:
  - Lokalna optimizacija (na nivou pojedinačnog bazičnog bloka)
  - Globalna optimizacija (na nivou celog grafa kontrole toka)
  - Interproceduralna optimizacija (na nivou celog programa tj. više funkcija)

10.1 Pripreme za optimizaciju

- Ključna razlika je u tome da li se vrši optimizacija nad običnim troadresnim kodom ili nad kodom u SSA (single static assignment) formi (pogledati primer kod eliminacije zajedničkih podizraza).
- Puno pročišćavanja elementarnih neefikasnosti se uradi u prvom prolazu optimizacije, a to je prevođenje u SSA.
- Na početku optimizacije se često radi *kanonikalizacija* (engl. *canonicalization*). U cilju lakše dalje obrade mnogi šabloni instrukcija se svode na kanonski oblik. Npr. u LLVM se na početku optimizacije vrši sledeća jednostavna transformacija koda (poređenje na različitost se menja u poređenje na jednakost, kako se u daljim optimizacijama nikad više ne bi morala obraćati pažnja na slučaj poređenja različitosti).

```
entry:
   %cmp = icmp ne 32 %argc, 3
   br i1 %cmp, label %if.then, label %if.end
```

```
entry:
   %cmp = icmp eq 32 %argc, 3
   br i1 %cmp, label %if.end, label %if.then
```

10.2 Lokalne optimizacije

- Ključna razlika je da li se koristi SSA. Danas svi značajni kompilatori koriste SSA.

- Mnoge lokalne optimizacije se vrše tako što se bazični blok transformiše u DAG, pa se onda novi kod generiše na osnovu DAG.
- Postoje različite vrste lokalnih optimizacija:
  - Možemo da elminišemo *lokalne zajedničke podizraze* (engl. *local common subexpression elimination*).
  - Možemo da elminišemo *mrtav kod* (engl. *dead code elminiation*) tj. kod koji izračunava vrednosti koje se ne koriste u daljem kodu.
  - Možemo primeniti *algebarske zakone* da uprostimo neke troadresne instrukcije. Npr. skuplje operacije poput množenja zamenjujemo jeftinijim, što nazivamo *redukcija snage* (engl. *strength reduction*).
  - Možemo da permutujemo redosled instrukcija (čime možemo da suzimo deo koda u kome je potrebno neku vrednost čuvati u registru ili u memoriji).

10.2.1 Algebarske transformacije

- Neke instrukcije se mogu obrisati:

```
x := x + 0
x := x * 1
```

- Neke instrukcije se mogu uprostiti (ovo se zove *redukcija snage*, engl. *strength reduction*):

```
x := x * 0          x := 0
y := y ^ 2          y := y * y
y := 2 * x          y := x + x
x := x * 8          x := x << 3
x := x * 15         t := x << 4; x := t - x;
```

- Ostaje pitanje da li je `<<` brže od `*`? Da li je kombinacija `<<` i `-` brža od `*`? Nekada jeste, nekada nije.

- Operacije na konstantama se mogu izvršiti tokom kompilacije umesto tokom izvršavanja (ovo se zove *uprošćavanje konstantni*, engl. *constant folding*).

```
x := 2 + 2                  x := 4
if 0 < 2 then goto L        goto L
if 2 < 0 then goto L        /* moze se obrisati */
```

- Potrebno je biti oprezan. Kompilacija i izvršavanje ne moraju da se vrše na istoj mašini, pa rezultati mogu biti različiti (različita širina reči ili različita preciznost realnih brojeva).

10.2.2 Eliminisanje zajedničkih podizraza

- Da li se u narednom kodu vrše neka suvišna izračunavanja?

```
a := b + c
b := a - d
c := b + c
d := a - d
```

  - Izrazi `b + c` i `a - d` se izračunavaju dva puta.
  - Izraz `b + c` se drugi put računa nad promenjenom vrednošću `b` i zato će vrednost u drugom izračunavanju biti drugačija nego u prvom.
  - Izraz `a - d` se drugi put izračunava nad istim operandima kao i prvi put i zato se drugo izračunavanje može izbeći.

- Kod se može uprostiti na sledeći način.

```
a := b + c
b := a - d
c := b + c
d := b
```

- Poslednja instrukcija može biti nepotrebna ako se `d` ne koristi nakon ovog koda. Slično, ako se `b` ne koristi, onda se kod može transformisati na sledeći način.

```
a := b + c
d := a - d
c := d + c
```

- Struktura zajedničkih podizraza se može identifikovati pomoću DAG.
- Pošto kod nije u SSA, potrebna je veoma fina analiza da li se vrednost promenljive menjala između dva naizgled identična izraza.
- Za svaku upotrebu promenljive u izrazu poželjno je znati prethodnu definiciju koja je relevantna (npr. u prethodnom primeru je u prvoj instrukciji relevantna ulazna vrednost promenljive `b`, a u trećoj vrednost iz druge instrukcije). Ovakve informacije se mogu izračunati analizom toka podataka o čemu će reči biti kasnije.
- U slučaju SSA potpuno smo sigurni da se vrednost promenljive nije mogla promeniti u toku jednog izvršavanja tog bazičnog bloka, tako da je sve identične izraze moguće elminisati.

10.2.3 Eliminacija mrtvog koda

- Dva osnovna oblika:
  - Eliminacija nedostižnih osnovnih blokova
  - Uklanjanje instrukcija koje definišu promenljive koje se ne koriste
- Uklanjanje koda čini program manjim, a manji programi su često brži (prostorna lokalnost, efekti keš-memorije)

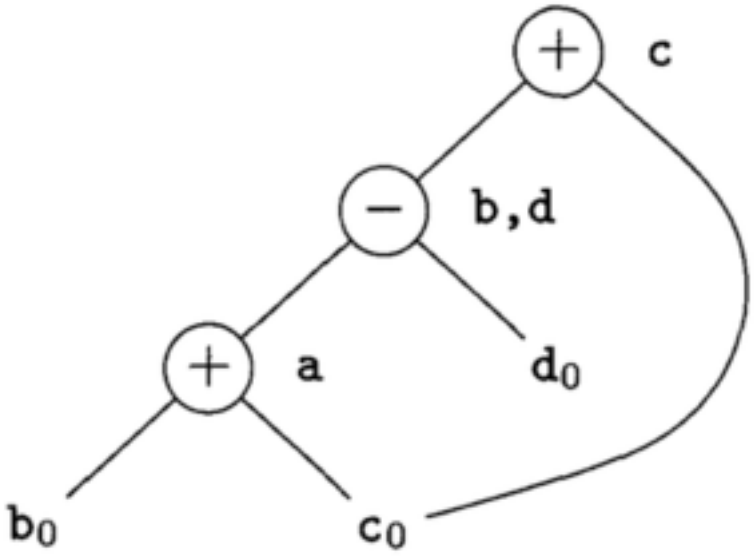


Figure 13: DAG za blok troadresnog koda

- Zašto nastaje nedostižan kod?
  - Uslovna kompilacija

    ```
    #define DEBUG 0

    if (DEBUG) {
      ...                    /* Ovaj kod ce biti mrtav */
    }
    ```

  - rezultat drugih optimizacija
- Kada je međukod u SSA obliku veoma lako nam je da odredimo koja definicija promenljive se koristi (svakoj promenljivoj odgovara tačno jedna definiciona instrukcija i poznata je lista instrukcija u kojoj se ta promenljiva koristi)
- Algoritam eliminacije možemo formulisati veoma jednostavno

  ```
  dok postoji neka promenljiva v koja se ne koristi i
      instrukcija kojom se definiše v ne proizvodi bočne efekte:
      obriši naredbu koja definiše v
      (iz koda ali i iz listi upotrebe promenljivih)
  ```

- Brisanjem instrukcije `v := x op y` moguće je da da definicije za `x` i `y`

postanu mrtve i da se one mogu ukloniti.

- Možemo formulisati naredni malo detaljniji algoritam uklanjanja mrtvog koda za SSA.

```
Neka je W lista svih promenljivih u SSA programu
dok W nije prazna:
  neka je v neka promenljiva uklonjena iz W
  ako je lista instrukcija u kojima se koristi v prazna:
     neka je S instrukcija u kojoj se definiše v
     ako S nema drugih efekata osim dodele promenljivoj v:
        obriši S iz programa
      za svaku promenljivu x koja se koristi sa desne strane S:
           obriši S iz liste njenih korišćenja
           dodaj x u W
```

- Pri eliminaciji mrtvog koda nekad treba biti obazriv. Nekada izvršavanje nekih aritmetičkih operacija može prouzrokovati izuzetak i prekid rada programa (npr. celobrojno deljenje nulom). Ako se to desi unutar mrtvog koda, eliminisanje tog mrtvog koda bi promenilo ponašanje programa, što se smatra nedopustivim, pa se u takvim slučajevima izbegava eliminisanje mrtvog koda. Mrtva instrukcija se može obrisati samo ako nema sporednih efekata.

10.2.4 Propagacija konstanti

- Instrukcije oblika `v := c` gde je `c` konstanta mogu prouzrokovati da se naredne upotrebe promenljive `v` zamene sa `c`.
  - Ponovo treba biti obazriv ako kod nije u SSA.
    ```
    a = 1
    b = a + 2
    a = a + 3
    c = a + 4
    ```
  - Ispravna propagacija konstanti je samo
    ```
    a = 1
    b = 1 + 2
    a = 1 + 3
    c = a + 4
    ```
  - Transformacija koja bi otišla korak dalje i zamenila poslednje korišćenje promenljive `a` ne bi bila ispravna (jer se vrednost `a` menja trećom instrukcijom).
    ```
    a = 1
    b = 1 + 2
    a = 1 + 3
    c = 1 + 4
    ```

  - U SSA je ovaj slučaj nemoguć i bezbedno je uvek zameniti sva pojavljivanja date promenljive konstantom.
  - Instrukcija `v := phi(c1, ..., cn)`, gde su sve `c1` do `cn` jednake, se može zameniti sa `v := c`.
  - Propagaciju konstanti u SSA možemo jednostavno postići narednim algoritmom.

```
neka je W lista svih instrukcija u SSA programu
dok W nije prazna:
   ukloni neku instrukciju S iz W
   ako je S oblika v := phi(c, ..., c) za neku konstantu c:
      zameni S sa v := c
   ako je S oblika v := c za neku konstantu c:
      obriši S iz programa
      za svaku instrukciju T u kojoj se koristi v:
         zameni v sa c u instrukciji T
         dodaj T u listu W
```

- Malim modifikacijama ovog algoritma možemo postići i lokalne optimizacije koje smo ranije pominjali:
  - instrukcije oblika `x := phi(y)` ili `x := y` mogu biti obrisane i `y` se može zameniti na svim mestima umesto `x` (*propagacija kopija*, engl. *copy propagation*)
  - instrukcije oblika `x := c1 op c2` za konstante `c1` i `c2` se mogu zameniti sa `x := c` gde se `c` izračunava za vreme kompilacije kao `c1 op c2` (*uprošćavanje konstanti*, engl. *constant folding*).
  - instrukcija grananja oblika `if c1 < c2 goto L` za konstante `c1` i `c2` se može ili zameniti sa `goto L` (ako se tokom kompilacije utvrdi da važi `c1 < c2`) ili se može obrisati (ako se utvrdi da to ne važi).
  - Uklanjanje naredbe može dovesti do toga da neki blok `L` postane nedostižan. Tada se mogu obrisati sve instrukcije iz njega vodeći računa o tome da se ažuriraju podaci o upotrebama promenljivih korišćenih u obrisanim instrukcijama i ažuriraju liste čvorova prethodnika i sledbenika u grafu.

10.2.5 Propagacija kopija

- Promenljive definisane u terminima drugih promenljivih su obično suvišne.

```
x := y
z := x + y
x := x + 2
w := z * x
```

```
x := y
z := y + y
x := y + 2
w := z * x
```

- Ponovo moramo voditi računa ako međukod nije u SSA obliku.
- Nakon propagacije konstanti incijalna definicija postaje mrtav kod i može se eliminisati.

```
z := y + y
x := y + 2
w := z * x
```

10.2.6 Kompozicija lokalnih transformacija

- Ni jedna lokalna optimizacija ne radi puno sama za sebe.
- Obično jedna optimizacija omogućava neku narednu.
- Optimizacije se izvršavaju jedna po jedna, dok god je to moguće.
- Ako je vreme kompilacije ograničeno, proces optimizacije se može zaustaviti u bilo kom trenutku.
- Razmotrimo naredni primer

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

- Propagacija kopiranja (konstanti)

```
a := 5
x := 2 * 5
y := x + 6
t := b * y
```

- Uprošćavanje konstantnih izraza

```
a := 5
x := 10
y := x + 6
t := b * y
```

- Propagacija konstanti

```
a := 5
x := 10
y := 10 + 6
t := b * y
```

- Uprošćavanje konstantnih izraza

```
a := 5
x := 10
```

```
y := 16
t := b * y
```

- Propagacija konstanti

```
a := 5
x := 10
y := 16
t := b * 16
```

- Redukcija snage

```
a := 5
x := 10
y := 16
t := b << 4
```

- Eliminacija mrtvog koda (ako se `a`, `x` i `y` ne koriste dalje)

```
t := b << 4
```

- Razmotrimo još jedan primer.

```
a := x ^ 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

- redukcija snage

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

- propagacija konstanti i promenljivih

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

- Uprošćavanje konstantnih izraza

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

- Propagacija konstanti

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := 6 * f
```

- Eliminacija zajedničkih podizraza

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := 6 * f
```

- Propagacija kopiranja promenljive

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

- Ako se `b`, `c`, `d` i `e` ne koriste, može se obrisati mrtav kod

```
a := x * x
f := a + a
g := 6 * f
```

- Može i dalje (ali to se teže primećuje i obično se ne bi izvršilo)

```
a := x * x
f := 2 * a
g := 6 * f
```

```
a := x * x
g := 12 * a
```

10.3 Globalna optimizacija

- Većina lokalnih optimizacija ima i svoje globalne varijante:
  - Globalna propagacija konstanti
  - Globalna eliminacija zajedničkih podizraza
  - ...
- Treba biti prilično pažljiv i pratiti tok podataka.
- Da li u narednom primeru možemo da zamenimo `x` sa `3`?

```
  x := 3
  if a < 10 goto L1
  z := x + 2
  x := x + 1
L1:
  y := 4 + x
```

- U redu je samo ovo:

```
  x := 3
  if a < 10 goto L1
  z := 3 + 2
  x := 3 + 1
L1:
  y := 4 + x
```

- U poslednjoj instrukciji `x` se ne sme zameniti sa `3`, jer se možda menja unutar bloka koji se izvršava ako nije `a < 10`.
- Primer globalnih optimizacija (particionisanje u algoritmu QuickSort).

```
i = m - 1; j = n; v = a[n];
while(1) {
   do i = i + 1; while (a[i] < v);
   do j = j - 1; while (a[j] > v);
   if (i >= j) break;
   x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[n]; a[n] = x;
```

- Prevođenje u troadresni međukod.

```
1:    i := m - 1
2:    j := n
3:    t1 := 4 * n
16:   t7 := 4 * i
17:   t8 := 4 * j
18:   t9 := a[t8]
```

```
4:    v := a[t1]
5:    i := i + 1
6:    t2 := 4 * i
7:    t3 := a[t2]
8:    if t3 < v goto (5)
9:    j := j - 1
10:   t4 := 4 * j
11:   t5 := a[t4]
12:   if t5 > v goto (9)
13:   if i >= j goto (23)
14:   t6 := 4 * i
15:   x := a[t6]
19:   a[t7] := t9
20:   t10 := 4 * j
21:   a[t10] := x
22:   goto (5)
23:   t11 := 4 * i
24:   x := a[t11]
25:   t12 := 4 * i
26:   t13 := 4 * n
27:   t14 = a[t13]
28:   a[t12] := t14
29:   t15 := 4*n
30:   a[t15] := x
```

- Lokalna optimizacija bloka B5 (eliminacija zajedničkih podizraza, propagacija kopija, eliminacija mrtvog koda).

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
```

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

- Posle lokalne optimizacije i dalje se u bloku B5 i B6 računaju vrednosti `4 * i`. Međutim, one su već izračunate u promenljivoj `t3` i ona se može koristiti zato što se ni na jednoj putanji od B2 do B5 i do B6 ne menja vrednost promenljive `i`. Slično se umesto ponovnog izračunavanja `4 * j` može upotrebiti `t4`, jer se ni na jednoj putanji od B3 do B5 i B6 ne menja vrednost `j`.
- Takođe se ni vrednost `a[j]` koja se nalazi u `t5` ne menja jer se na odgovrajućim putanjama kroz graf ne vrši izmena elemenata niza.
- Zato se blokovi B5 i B6 mogu značajno dodatno uprostiti.

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```

```
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := x
```

- Graf kontrole toka nakon ove globalne optimizacije izgleda ovako:
- Naglasimo da se zajednički podizraz `a[t1]` u blokovima B1 i B6 nije mogao eliminisati, jer se iako se vrednost promenljive `t1` ne menja, na putanji između tih blokova menja sadržaj niza `a`.

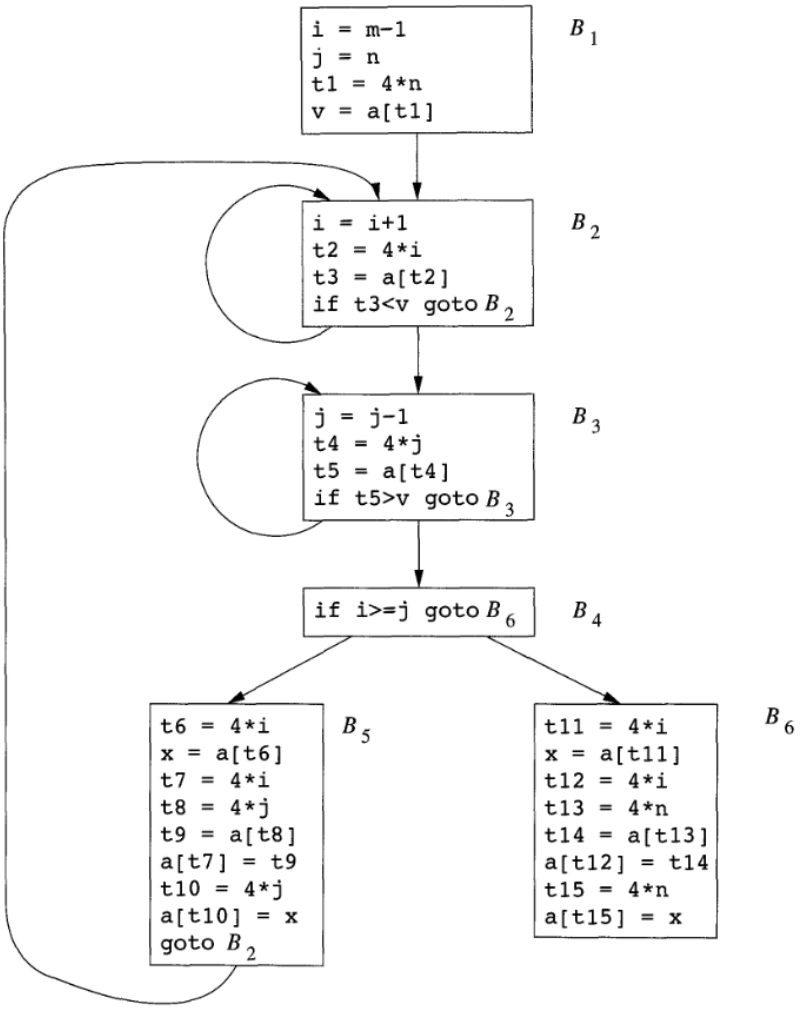


Figure 14: Graf kontrole toka za fragment QuickSort koda

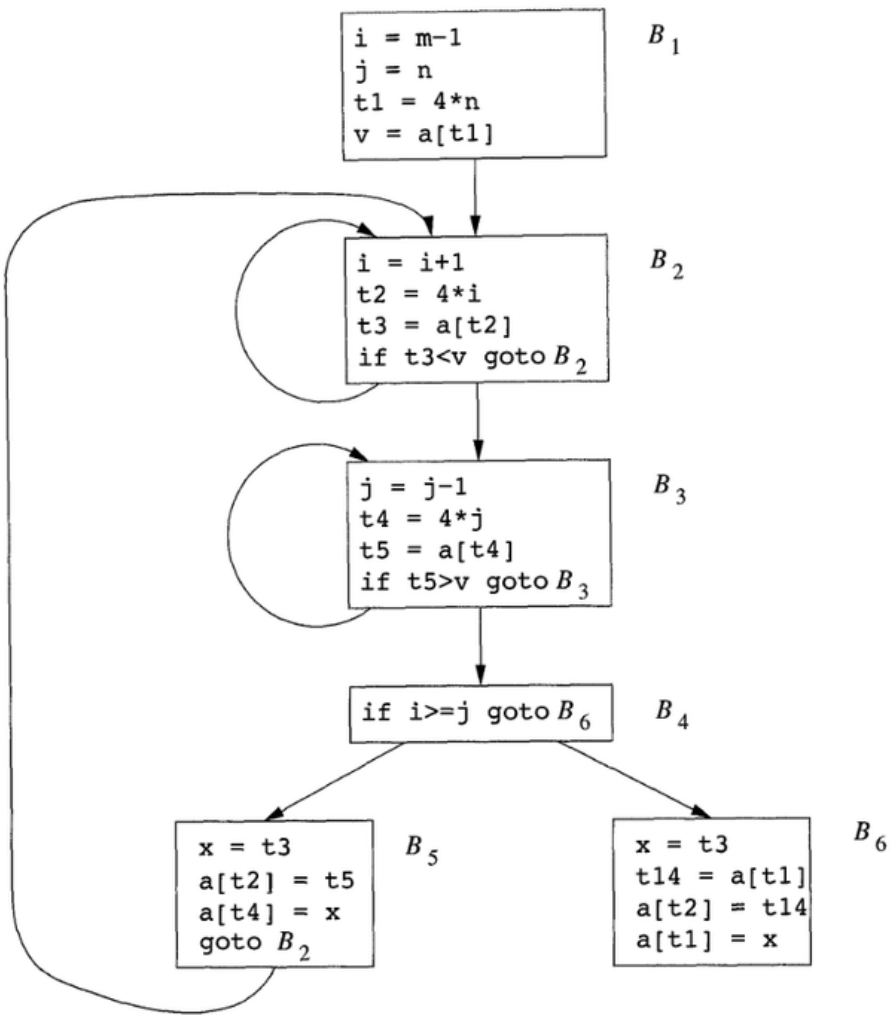


Figure 15: Graf kontrole toka za fragment QuickSort koda nakon globalne optimizacije

- Nakon lokalne propagacije kopija i brisanja novonastalog mrtvog koda blokovi B5 i B6 se dodatno mogu uprostiti u sledeći kod.

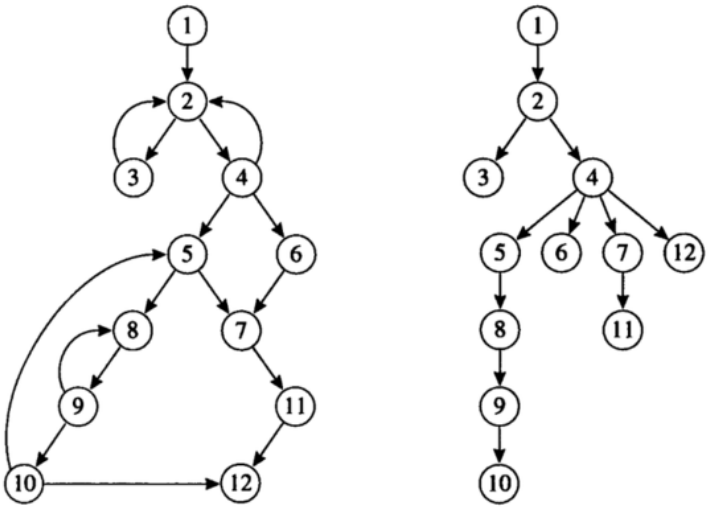
```
a[t2] := t5                t14 := a[t1]
a[t4] := t3                a[t2] := t14
goto B2                    a[t1] := t3
```

10.3.1 Optimizacija petlji

- Kod u petljama se očekivano izvršava veliki broj puta i stoga je jako bitno njega dobro optimizovati.
- Naročit značaj ima optimizacija kod unutar ugnežđenih petljih.

10.3.1.1 Šta su petlje u grafu i kako se otkrivaju?

- Petlja u grafu kontrole toka je skup čvorova S koji uključuje čvor zaglavlja h, tako da:
  - od svakog čvora u S se može stići do h,
  - od h se može stići do bilo kog čvora u S,
  - jedine putanje od čvorova van S do čvorova u S ulaze u h.
- Dakle petlja mora da ima jedinstven ulazni čvor (h), a može da ima više tačaka izlaza.
- Klasične kontrolne strukture (`for`, `while`, `do-while`, `repeat`, …) daju pravilne grafove kontrole toka. Upotreba `goto` može proizvesti nepravilne grafove koji se teže analiziraju i obrađuju.
- Pre optimizacije petlji potrebno ih je identifikovati u grafu. Algoritam je zasnovan na konceptu *dominacije* (vidi opis u delo o SSA). Podsetimo se, dominator je onaj čvor kroz koji se uvek mora proći da bi se stiglo do datog čvora.
  - Ako d_1 i d_2 dominiraju nad čvorom n, tada ili d_1 dominira nad d_2 ili d_2 dominira nad čvorom d_1.
  - Zato svaki čvor n (osim ulaznog čvora u graf) ima jedinstvenog *neposrednog dominatora*, $idom(n)$:
    * $idom(n)$ striktno dominira nad n,
    * $idom(n)$ ne dominira ni nad jednim drugim striktnim dominatorom od n.
  - Neposredni dominatori čine *drvo dominacije* (engl. *dominator tree*).



  - *Povratne grane* vode od nekog čvora n do čvora h koji njime dominira. *Prirodna petlja* za svaku povratnu granu je skup čvorova x kojima dominira h i od kojih se može stići do n (ali ne preko h). Na slici je prirodna petlja povratne grane $10 \rightarrow 5$ skup $\{5, 8, 9, 10\}$ i on sadrži ugnežđenu petlju $\{8, 9\}$.
- U mnogim optimizacijama petlje se deo koda iz petlje izvlači ispred petlje.
- U koji čvor se taj kod postavlja? Najlakše je uvesti novi čvor - *predzaglavlje petlje* (engl. *loop preheader*).

10.3.1.2 Pomeranje invarijantnog koda

- Pod pretpostavkom da se vrednost `limit` ne menja u petlji, naredni kod se može urzati ako se uradi naredna transformacija.

```
while (i <= limit - 2) ...
```

```
t = limit - 2;
while (i <= t) ...
```

- Ako se u petlji nalazi instrukcija oblika `t := a1 op a2` takva da `a1` i `a2` imaju istu vrednost u svakom prolazu kroz petlju, tada će i `t` imati istu vrednost i biće moguće izračunati ga samo jednom, umesto svaki put.
- Teško je precizno odrediti da li promenljive imaju istu vrednost svaki put. Zato se koristi konzervativna aproksimacija (ne smemo da tvrdimo da ima istu vrednost, ako nema, ali smemo da kažemo da nema, ako ima).
- Kažemo da je definicija `t := a1 op a2` *invarijantna* (engl. *loop invariant*) ako za svaki operand `ai` važi:
  - `ai` je konstanta ili
  - sve definicije `ai` koje dosežu do te instrukcije su van petlje
  - samo jedna definicija doseže do `ai` i ona je invarijantna.

- Na osnovu prethodnog moguće je formulisati iterativni algoritam za određivanje invarijantnih definicija (krećemo od onih koje sadrže samo konstante ili promenljive definisane van petlje, a onda proširujemo postupak uključujući i one promenljive koje su definisane definicijama za koje smo već utvrdili da su invarijantne).
- Invarijantne definicije se *izvlače* (engl. *hoist*) ispred petlje (najčešće u predzaglavlje).

```
    t := 0
L1:
    i := i + 1
    t := a + b
    M[i] := t
    if i < N goto L1
L2:
    x := t
```

```
    t := a + b
L1:
    i := i + 1
    M[i] := t
    if i < N goto L1
L2:
    x := t
```

- Izvlačenje se ne sme uvek vršiti!
  - Nekada se kod izvršava samo pod nekim uslovom i njegovo izvršavanje bez tog uslova može dovesti do neke greške.

```
    t := 0
L1:
    if i >= N goto L2
    i := i + 1
    t := a + b
    M[i] := t
    goto L1
L2:
    x := t
```

    * U prethodom kodu se `t := a + b` izvršava samo ako je `i < n`. Izvlačenje bi prouzrokovalo da se izvršava i kada je `i >= N`, a to može biti pogrešno.

  - Nekada se promenljiva menja tokom petlje.

```
    t := 0
L1:
    i := i + 1
    t := a + b
    M[i] := t
    t := 0
    M[j] := t
    if i < N goto L1
L2:
```

    * U prethodnom kodu se `t` menja instrukcijom `t := 0` unutar

petlje i mora se uvek ponovo postaviti na `a + b` (to nije moguće uraditi samo jednom pre petlje).

- Nekada se početna vrednost koristi u prvom prolasku kroz petlju, pre njene definicije.

```
    t := 0
L1:
    M[j] := t
    i := i + 1
    t := a + b
    M[i] := t
    if i < N goto L1
L2:
```

  * U prethodnom kodu se u prvom prolasku koristi vrednost `t := 0`.

- Mnogi od ovih problema nestaju ako je kod u SSA obliku.

10.3.1.3 Indukcione promenljive i redukcija snage

- Promenljiva `x` je indukciona ako postoji (pozitivna ili negativna) konstanta `c` takva da da se pri svakoj dodeli vrednosti `x` ona uveća za `c`.
- U tekućem primeru promenljive `i` i `t2` su indukcione u petlji oko bloka B2. Promenljiva `i` se uvećava za 1, a `t2` za 4.
- Indukcione promenljive često omogućavaju redukciju snage (zamenu množenja sabiranjem).
- Često se indukcione promenljive javljaju u parovima ili u grupama i obično se sve osim jedne mogu eliminisati.
- Postupak optimizacije često teče od unutrašnjih petlji.
- Nakon optimizacije indukcionih promenljivih, graf kontrole toka za tekući (QuickSort) primer izgleda ovako.

10.3.2 Odmotavanje petlji

- Nekada je telo petlje veoma jednostavno i većina vremena u izvršavanju petlje odlazi na testiranju uslova i ažuriranju promenljive.
- Ubrzanje se može dobiti ako se izvrši *odmotavanje petlje* (engl. loop unrolling).
- Odmotavanje samo po sebi obično ne donosi ubrzanje - smisao dolazi u kombinaciji sa drugim optimizacijama. Naredno odmotavanje ne donosi ubrzanje.

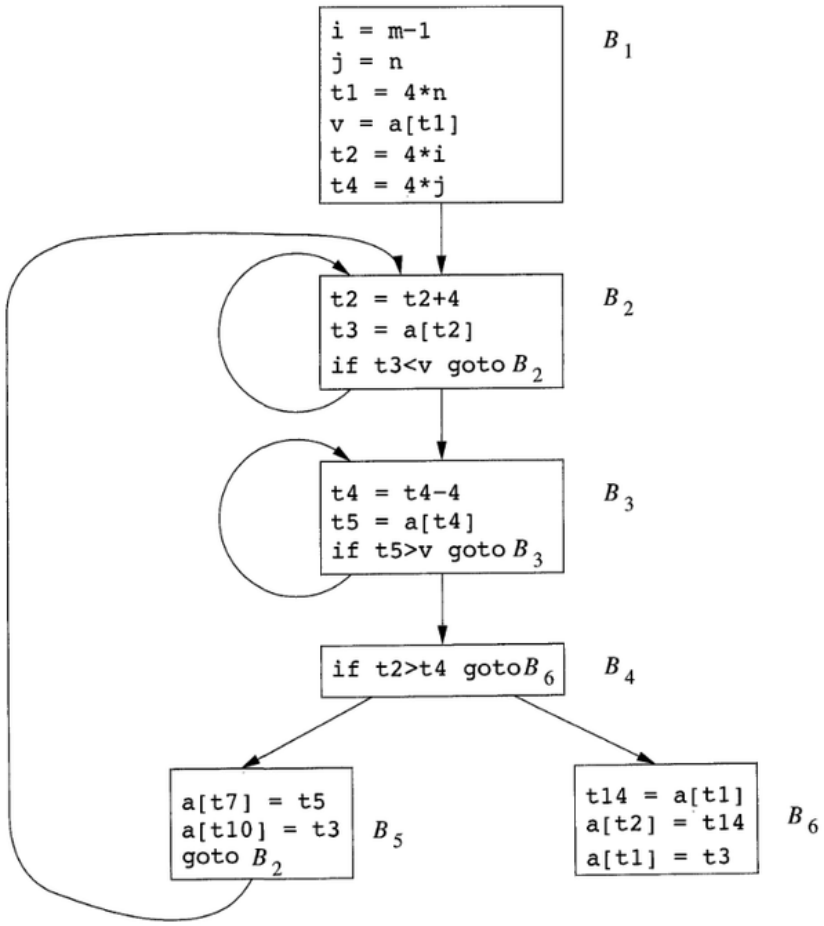


Figure 16: Graf kontrole toka za fragment QuickSort koda nakon optimizacije indukcionih promenljivih

```
L1:   x := M[i]
      s := s + x
      i := i + 4
      if i < n goto L1
```

```
L1:  x := M[i]
     x := s + x
     i := i + 4
     if i < n goto L1'
     goto L2
L1': x := M[i]
     x := s + x
     i := i + 4
     if i < n goto L1
L2:
```

```
     if i < n - 8 goto L1 else L2
L1:  x := M[i]
     s := s + x
     x := M[i + 4]
     s := s + x
     i := i + 8
     if i < n - 8 goto L1 else L2
L2:  x := M[i]
     s := s + x
     i := i + 4
     if i < n goto L2
```

10.4 Interproceduralna optimizacija

- Jedna od najzačajnijih trasformacija je *uvlačenje definicija funkcija* (engl. *inlining*)
- Potrebno je napraviti balans - premalo uvlačenja dovodi do skupih troškova vezanih za mehanizam poziva funkcija, dok previše uvlačenja uveća veličinu koda i dovodi do nefikasnosti usled njegovog sporog učitavanja iz memorije.
- Jednostavne funkcije se uvlače, a složenije ne.
- Ne možemo da znamo da li je neka funkcija jednostavna sve dok je ne optimizujemo. Kojim redom onda optimizovati funkcije?
- Napravi se graf poziva funkcija, identifikuju se komponente jake povezanosti, a onda se dobijene komponente obrađuju odozdo naviše.
- Iz poznavanja konteksta mozemo da uradimo inline samo neke grane u kodu (npr. ako je u nekoj funkciji grananje po nekom parametru, a kompilator ume da na neki način zaključi da vrednost tog parametra pripada određenom opsegu, tada može da uvuče samo tu granu koja odgovara tom opsegu).

11 Analize toka podataka

- Videli smo da je za sprovođenje nekih globalnih optimizacija potrebno imati dodatne informacije o programu i grafu kontrole toka. Na primer:
  - Da bi se sprovelo eliminisanje zajedničnikih podizraza, potrebno je znati da li se dva tekstualno identična izraza uvek (kroz sve moguće putanje programa) izračunavaju na istu vrednost.
  - Da bi se sprovelo eliminisanje mrtvog koda, potrebno je znati da neke definicije računaju vrednosti promenljivih ne koriste ni na jednoj putanji dalje kroz program.
  - Da bi se sprovela globalna eliminacija konstanti potrebno je da definicija kojom se promenljivoj dodeljuje neka konstantna vrednost doseže do upotrebe te promenljive kroz sve moguće putanje (tj. da se ni na jednoj putanji ta promenljiva ne definiše na neki drugačiji način).
- Na mnoga od pitanja poput onih gore moguće je dati odgovor nakon što se sprovede neka *analiza toka podataka* (engl. *dataflow analysis*).
- Postoje različite konkretne analize koje nam daju različite značajne informacije o programu i njegovom uticaju na podatke kroz sve moguće putanje, međutim, sve one se sprovode na donekle sličan način, pa ih ima smisla proučavati istovremeno.
- Neke od najpoznatijih analiza toka podataka su:
  - *dosezajuće definicije* (engl. *reaching definitions*) kojom se za svaku upotrebu promenljive određuju sve moguće definicije koje definišu tekuću vrednost te promenljive na mestu te upotrebe,
  - *živost promenljivih* (engl. *liveness*) kojom se u svakoj tački programa određuju promenljive čija se trenutna vrednost upotrebljava dalje, na nekoj putanji kroz program,
  - *raspoloživost izraza* (engl. *expression availability*) kojom se za svaku tačku u programu određuje da li je vrednost `x op y` izračunata i ažurna (nakon izračunavanja nisu menjane vrednosti `x` i `y`), bez obzira na koji način smo došli do te tačke,
  - *propagacija konstanti* (engl. *constant propagation*) kojom se za svaku upotrebu promenljive proverava da li je vrednost te promenljive jednaka nekoj konstanti (uvek istoj, bez obzira kojom putanjom smo stigli do tačku te upotrebe),
  - …

11.1 Opšta šema analize

- Svakoj tački u programu (mestu ispred instrukcije, mestu iza instrukcije, početku bazičnog bloka, kraju bazičnog bloka) dodeljuje se tzv. *vrednost toka podataka* (engl. *dataflow value*) koja predstavlja apstrakciju skupa

svih mogućih stanja programa (vrednosti promenljivih) koja mogu da važe u toj tački.

- Vrednost toka podataka zavisi od analize koje se vrše. Na primer, kod analize živosti vrednost toka podataka u svakoj tački je skup promenljivih koje su žive u toj tački (koriste se negde nadalje), dok je za doseg definicija vrednost toka podataka neki skup definicija promenljive koja se analizira.
- Vrednost toka podataka pre naredbe s označava se sa $in[s]$, a nakon naredbe s sa $out[s]$. Cilj analize toka podataka je odrediti ove vrednosti, ali tako da su zadovoljena sledeća ograničenja.
  - Semantička ograničenja naredbe izražene u terminima prenosne funkcije (eng. transfer function) koje na osnovu konkretne naredbe određuju vezu između vrednosti toka pre i posle naredbe.
  - Ograničenja formirana na osnovu kontrole toka programa i to unutar pojedinačnog bloka, ali i na osnovu grafa kontrole toka. Tok unutar bloka je obično trivijalan i on samo definiše da će vrednost toka pre prve naredbe nekog bloka biti jednak ulaznoj vrednosti u taj blok ($in[B] = in[s_0]$), da će vrednost posle poslednje naredbe bloka biti jednaka izlaznoj vrednosti iz tog bloka ($out[B] = out[s_n]$) i da će vrednosti za susedne naredbe biti povezane tako što će izlazna vrednost prethodne naredbe biti uvek jednaka ulaznoj vrednosti u sledeću ($out[s_i] = in[s_{i+1}]$). Ograničenja na osnovu grafa kontrole toka su obično složenija i ona povezuju izlazne vrednosti bloka prethodnika sa ulaznim vrednostima bloka sledbenika.
- Postoje dva osnovna oblika analize toka podataka (svaki je pogodan za neke konkretne analize):
  - analiza toka unapred (engl. forward flow) na osnovu ulaznih vrednosti u blok tj. vrednosti pre naredbi određuje izlazne vrednosti iz bloka tj. vrednosti posle naredbi.

$$out[s] = f_s(in[s])$$

$$in[B] = \bigoplus_{P-\text{prethodnik bloka B}} out[P]$$

    pri čemu $\bigoplus$ označava neku kombinaciju ovih vrednosti (najčešće je to unija ili presek).

  - analiza toka unatrag (engl. backward flow) na osnovu izlaznih vrednosti iz bloka tj. vrednosti posle naredbi određuje ulazne vrednosti u blok tj. vrednosti posle naredbi.

$$in[s] = f_s(out[s])$$

$$out[B] = \bigoplus_{S-\text{sledbenik bloka B}} in[S]$$

- Vrednosti toka podataka su često neki skupovi. Efekat svake pojedinačne naredbe je da se ti skupovi modifikuju tako što se neke vrednosti izbace iz trenutne vrednosti toka podataka (trenutnog skupa), a neke nove vrednosti ubace (svejedno je da li se gleda unapred ili unatrag). Tada se svakoj naredbi dodeljuje skup $gen[s]$ koji sadrži vrednosti koje treba dodati u skup i skup $kill[s]$ koji sadrži vrednosti koje treba izbaciti iz skupa. Važi da je tada $f_s(X) = gen[s] \cup (X \quad kill[s])$.
- Analiza je brža ako se prvo lokalno obradi svaki bazični blok B pojedinačno i izračunaju $gen[B]$ i $kill[B]$ i tek se tada pređe na globalnu analizu celog grafa. Na primer, prilikom analize unapred za bazični blok narebi $B = s_1 ... s_n$ važi da je

$$out[B] = gen[B] \cup (in[B] \quad kill[B])$$

$$kill[B] = kill[s_1] \cup ... \cup kill[s_n]$$

$$gen[B] = gen[s_n] \cup (gen[s_{n-1}] \; kill[s_n]) \cup ... \cup (gen[s_1] \; (kill[s_2] \cup ... \cup kill[s_n]))$$

  Veoma slične veze važe i u slučaju analize unatrag.
- Nekada se dešava da postoji više rešenja koja zadovoljavaju opisane uslove. Cilj je naći što "preciznija" rešenja, tj. rešenja koja omogućavaju što veće optimizacije, ali tako da se zadrži ekvivalentnost transformacije (rešenja moraju biti *konzervativna*).
- Jedno prilično kvalitetno rešenje možemo naći primenjujući na prethodne jednačine algoritam pronalaženja *najmanje fiksne tačke* (engl. *least fixed-point*).
- Ideja je da krenemo od neke inicijalne procene vrednosti toka podataka u svakom čvoru, a onda da vrednosti iterativno malo po malo ažuriramo sve dok se prvi put ne dođe da onih vrednosti koje zadovoljavaju sve jednačine.

11.2 Dosezajuće definicije

- Definicija d promenljive `x` (troadresna instrukcija oblika `x := ...`, najčešće `x := y op z`) doseže tačku p ako postoji putanja od tačke neposredno iza d do tačke p na kojoj ta definicija d nije ubijena tj. na toj putanji ne postoji druga definicija promenljive `x`. Ako definicija d doseže do tačke p,

onda je moguće da u tački p vrednost `x` bude baš ona dodeljena definicijom d.

- U nekim situacijama ne znamo da li troadresna instrukcija menja promenljivu `x` (npr. šaljemo `x` u funkciju po referenci). Zbog konzervativnosti moramo pretpostaviti da se na tom mestu vrednost `x` menja (jer se teoretski može promeniti).
- U većini primena dosezajućih definicija nije pogrešno da se navede da neka definicija doseže tačku p, čak i kada to nije slučaj, dok obratno može biti pogrešno.
- Dosezajuće definicije se mogu izračunati analizom unapred (krećemo od definicija i pratimo putanje unapred, sve dok ne naiđemo na nove definicije koje nadalje preuzimaju).
- Posmatrajmo naredbu `x := y op z`. Ona generiše novu definiciju d promenljive `x` i ubija sve dotadašnje definicije d' promenljive `x` (one ne dosežu dalje od ove tačke). Možemo reći da je za ovu naredbu d skup $gen[d] = \{d\}$, a skup $kill[d]$ skup svih drugih definicija promenljive `x` u programu. Za svaku naredbu s važi da je

$$out[s] = gen[s] \cup (in[s] \quad kill[s])$$

- U primeru

```
d1:   a := 3
d2:   a := 4
```

važi da je $kill = \{d_1, d_2\}$, dok je $gen = \{d_2\}$.

- Naredni primer ilustruje *gen* i *kill* za bazične blokove.
- Što se tiče kontrole toka, pošto definicija doseže neku tačku ako je doseže preko bilo koje putanje, važi

$$in[B] = \bigcup_{P-\text{prethodnik bloka B}} out[P]$$

- Podrazumevamo da postoji jedinsten ulazni blok (nazovimo ga *Entry*) i važi $out[Entry] = \emptyset$.
- Prikažimo korak po korak izračunavanje dostižnosti definicija na prethodnom primeru na osnovu algoritma najmanje fikne tačke. Na početku pretpostavljamo da su svi *in* i *out* skupovi prazni, a zatim ih malo po malo povećavamo, sve dok se prvi put ne dođe do stanja u kojem su sve jednačine zadovoljene.
  - Važi da je $in[B_1] = out[Entry] = \emptyset$, pa je $out[B_1] = gen[B_1] \cup (in[B_1] \quad kill[B_1]) = \{d_1, d_2, d_3\}$.

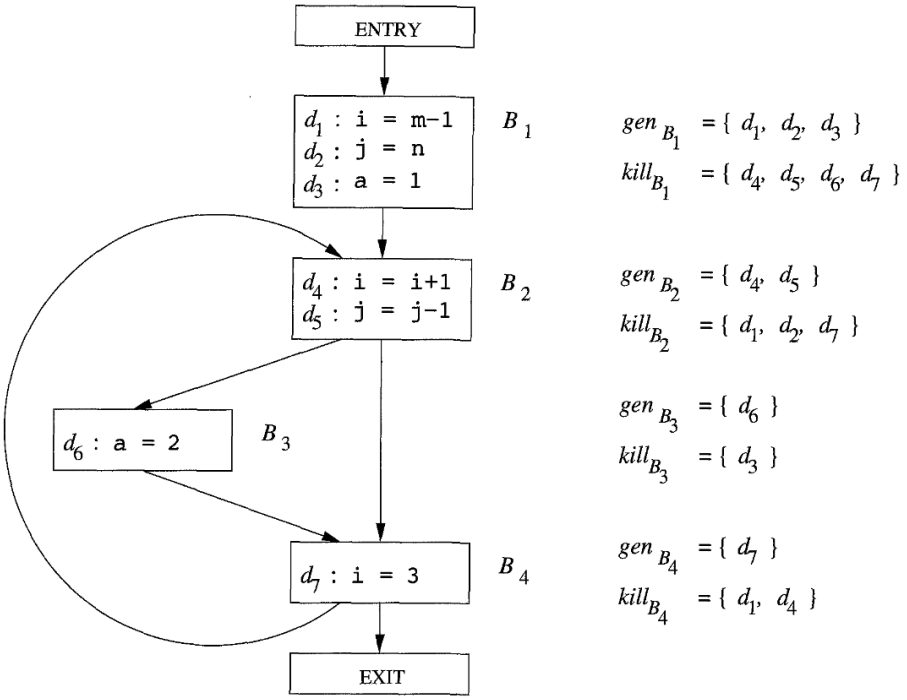


Figure 17: *gen* i *kill* za bazične blokove

- Pošto je inicijalno pretpostavljeno da je $out[B_4] = \emptyset$, važi da je $in[B_2] = out[B_1] \cup out[B_4] = \{d_1, d_2, d_3\}$.
- Važi da je $out[B_2] = gen[B_2] \cup (in[B_2] \setminus kill[B_2]) = \{d_4, d_5\} \cup (\{d_1, d_2, d_3\} \setminus \{d_1, d_2, d_7\}) = \{d_3, d_4, d_5\}$.
- Važi da je $in[B_3] = out[B_2] = \{d_3, d_4, d_5\}$.
- Važi da je $out[B_3] = gen[B_3] \cup (in[B_3] \setminus kill[B_3]) = \{d_6\} \cup (\{d_3, d_4, d_5\} \setminus \{d_3\}) = \{d_4, d_5, d_6\}$.
- Važi da je $in[B_4] = out[B_2] \cup out[B_3] = \{d_3, d_4, d_5, d_6\}$.
- Važi da je $out[B_4] = gen[B_4] \cup (in[B_4] \setminus kill[B_4]) = \{d_7\} \cup (\{d_3, d_4, d_5, d_6\} \setminus \{d_1, d_4\}) = \{d_3, d_5, d_6, d_7\}$.
- Činjenica da je $out[B_4]$ izračunat preciznije nego na početku nas tera da ažuriramo $in[B_2] = out[B_1] \cup out[B_4]$ i postavimo ga na $\{d_1, d_2, d_3, d_5, d_6, d_7\}$.
- Međutim, važi da je $out[B_2] = gen[B_2] \cup (in[B_2] \setminus kill[B_2]) = \{d_4, d_5\} \cup (\{d_1, d_2, d_3, d_5, d_6, d_7\} \setminus \{d_1, d_2, d_7\}) = \{d_3, d_4, d_5, d_6\}$.
- To je ujedno i $in[B_3]$, ali $out[B_3]$ i dalje ostaje $\{d_4, d_5, d_6\}$.
- Važi da je $in[B_4] = out[B_2] \cup out[B_3] = \{d_3, d_4, d_5, d_6\}$ i on se nije promenio.
- Na kraju važi da je $in[Exit] = out[B_4] = \{d_3, d_5, d_6, d_7\}$.
- Pošto nema daljih promena, dostignuta je fiksna tačka i u svakoj tački programa se znaju definicije koje je mogu dosegnuti.

- Rezultat možemo prikazati i na grafu kontrole toka.
- Postupak koji smo neformalno izvodili možemo formalizovati u obliku sledećeg algoritma.

```
out[Entry] = {}
for (svaki bazični blok B osim bloka Entry) out[B] = {}
repeat
   for (svaki bazični blok B osim bloka Entry)
      in[B] = Unija_{P je prethodnik B} out[P]
      out[B] = gen[B] unija (in[B] razlika kill[B])
until (nema promena na nekom out[B])
```

- Vrednosti toka podataka (skupove definicija) možemo predstaviti vektorima bitova. Tada se operacija unije ostvaruje operacijom bitovske disjunkcije, a operacija razlike ostvaruje operacijom bitovske konjunkcije sa bitovskim komplementom drugog operanda.
- Rad algoritma, pod pretpostavkom da se u petji `for` blokovi obilaze redosledom $B_1, B_2, B_3, B_4, Exit$, da se u vektorima bitova redom predstavl-

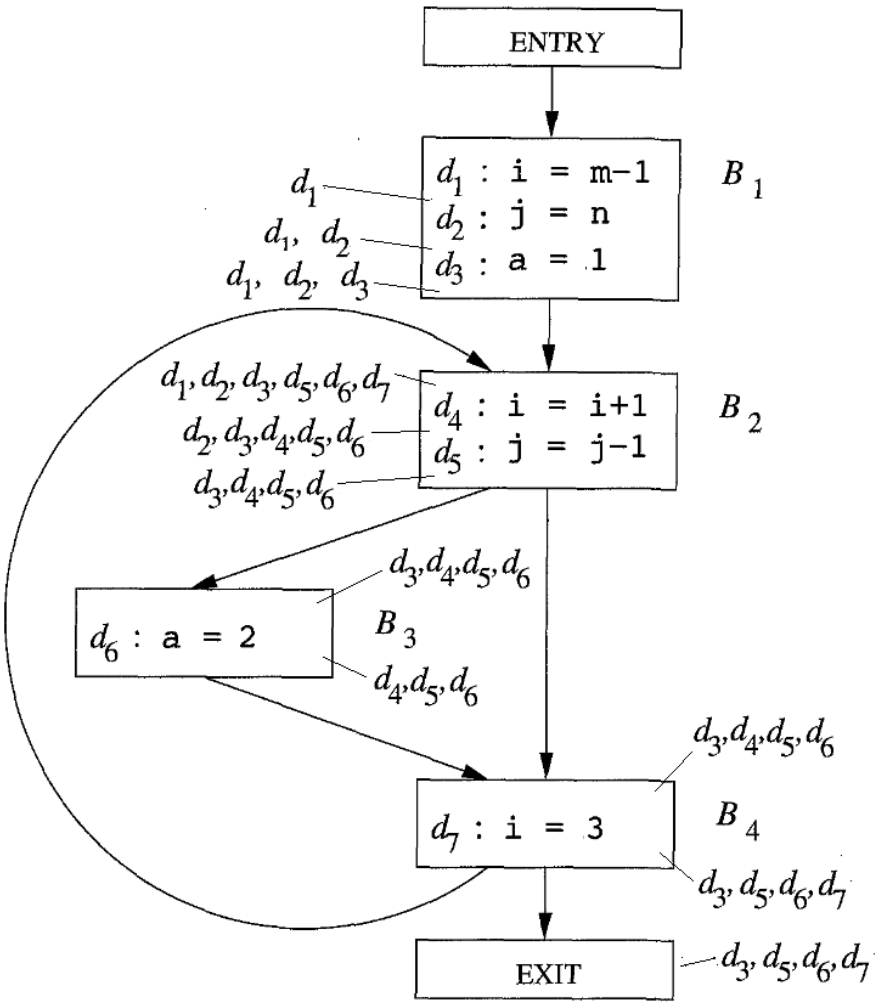


Figure 18: Rezultat analize dostižnosti definicija

jaju definicije od d_1 do d_7 i da kolone predstavljaju jednu po jednu iteraciju petlje `while` možemo prikazati sledećom tabelom.

Blok B	$out[B]^0$	$in[B]^1$	$out[B]^1$	$in[B]^2$	$out[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
$Exit$	000 0000	001 0111	001 0111	001 0111	001 0111

- Kod analiza unapred bolje je u unutrašnjoj petlji obilaziti čvorove u redosledu od ulaza ka izlazu iz grafa, dok je kod analize unazad bolje čvorove obilaziti u redosledu od izlaza ka ulazu iz grafa. Kada je graf aciklički, ovo garantuje završetak u samo jednom izvršavanju petlje `repeat`. Čak i kod cikličkih grafova ima smisla raditi u redosledu koji optrilike odgovara tome (koristi se algoritam kvazi-topološkog sortiranja zasnovan na DFS).

```
Topološko sortiranje:
N - broj čvorova
za svaki čvor i
 mark[i] := false
DFS(Entry)
```

```
DFS(i):
   if mark[i] = False:
      mark[i] := True
for (svaki sledbenik s čvora i)
      DFS(s)
   sorted[N] := i
   N := N - 1
```

- Unutrašnja petlja `for` se izvršava uvek za ceo graf iako su često promene u prethodnom koraku lokalizovane samo u nekom manjem delu grafa. Efikasniji algoritam možemo dobiti ako ažuriramo samo vrednosti kod kojih je bilo nekih promena.

```
neka je W skup svih bazičnih blokova
while (W nije prazan)
   ukloni proizvoljan čvor B iz W
   in[B] := Unija_{P je prethodnik B} out[P]
   out[B] := gen[B] unija (in[B] razlika kill[B])
   ako se out[B] promenio
     za svaki sledbenik S bloka B koji ne pripada W
        dodaj S u W
```

11.2.1 Lanci use-def

- Informacije o dosezajućim definicijama se često čuvaju u obliku *use-def* lanaca.
- To je struktura podataka koja povezuje svaku neke promenljive x sa listom definicija te promenljive koje mogu dosezati do te upotrebe.

- Na primer, za promenljivu `i` u definiciji d_4 važi da je *use-def*$(i) = \{d_1, d_7\}$, za promenljivu `j` u definiciji d_5 važi da je *use-def*$(j) = \{d_2, d_5\}$.

11.2.2 Upotreba informacija o dosezajućim definicijama

- Znanje o dosezajućim definicijama (npr. u obliku use-def lanaca) omogućava izvođenje jednostavne propagacije konstanti (o složenijoj tehnici detekcije i propagacije konstanti kasnije) i propagacije kopija.
- Pretpostavimo da postoji instrukcija d oblika `x := c` i druga instrukcija n oblika `y := x op z`. Ako je za `x` u instrukciji n važi da *use-def*(x) sadrži samo d, tada se u instrukciji n može izvršiti zamena `y := c op z`.
- Pretpostavimo da postoji instrukcija d oblika `x := t` i druga instrukcija n oblika `y := x op z`, tada ako nijedna druga definicija `t` ne doseže do n (tj. ako je *use-def*$(x) = \{d\}$), ali i ako ne postoji nijedna definicija promenljive `t` na bilo kojoj putanji između d i n, tada se n može zameniti sa $y := topz$.

11.3 Analiza živosti promenljivih

- Promenljiva x je živa u tački p ako se vrednost promenljive x u tački p koristi na nekoj putanji u grafu koja počinje u tački p.
- Živost promenljivih je veoma važna za:
  - eliminaciju mrtvog koda (dodele koje nisu žive se mogu eliminisati iz koda)
  - registarsku alokaciju (promenljiva koja se nalazi u registru i koja je mrtva se može odmah ukloniti iz registra)
- Živost se može utvrtiti analizom toka podataka unatrag (od upotreba promenljivih pa unatrag ka njihovim definicijama)
- Vrednost toka podataka za svaku tačku će biti skup živih promenljivih u toj tački.
- Kada se gleda odozgo naviše, svaka dodela oblika $x := yopz$ uklanja iz skupa živih promenljivih promenljivu x, a dodaje u skup živih promenljivih promenljive y i z. Dakle, ako sa $def[s]$ obeležimo promenljive koje se definišu naredbom s (u ovom primeru to je $\{x\}$), a sa $use[s]$ skup promenljivih koje se koriste u naredbi s (u ovom primeru to je $\{y, z\}$) važi da je $in[s] = use[s] \cup (out[s] \quad def[s])$. Primetimo da skup *use* ima ulogu *gen*, a *def* ima ulogu *kill*.
- Na osnovu ovoga se za svaki bazični blok izračunavaju skupovi $def[B]$ i $use[B]$ i važe jednakosti:

$$in[B] = use[B] \cup (out[B] \quad def[B])$$

$$out[B] = \bigcup_{S-\text{sledbenik bloka B}} in[S]$$

- Analiza može da krene od uslova da nijedna promenljiva nije živa nakon izlaska iz grafa ili od uslova da su na izlasku iz grafa žive neke promenljive (npr. ako graf predstavlja funkciju, na izlasku su žive one promenljive čiju vrednost funkcija vraća).
- Skupovi živih promenljivih u svakoj tački programa se mogu izračunati na sledećim algoritmom.

```
in[Exit] = {}
for (svaki bazični blok B osim bloka Exit) in[B] = {}
repeat
   for (svaki bazični blok B osim bloka exit)
      out[B] = Unija_{S je sledbenik B} in[S]
      in[B] = use[B] unija (out[B] razlika def[B])
until (nema promena na nekom in[B])
```

- Prikažimo rad algoritma na sledećem primeru.

```
B1: a := 3
    b := 5
    d := 4
    x := 100
    ifFalse a > b goto b3

B2: c := a + b
    d := 2

B3: c := 4
    t1 := b * d
    t2 := t1 + c
    return t2
```

  - Na kraju koda živa je samo promenljiva `t2`.
  - Prikažimo kako se vrši analiza poslednjeg bazičnog bloka. Između svake dve instrukcije prikažimo skup živih promenljivih.

```
                    {b, d} = in[B3]
B3: c := 4
                    {b, d, c}
    t1 := b * d
                    {t1, c}
    t2 := t1 + c
                    {t2} = out[B3]
    return t2
```

– Ovo nam govori da je pre bloka B_3 potrebno znati vrednosti promenljivih `b` i `d` i da se njihove vrednosti koriste za izračunavanje rešenja. Pošto se promenljiva `c` definiše u bloku B_3, njena ranija vrednost nije relevantna za izračunavanje krajnjeg rezultata (ranija vrednost nije živa, jer biva ubijena naredbom `c := 4`).

– Važi da je B_3 jedini sledbenik i bloka B_2. Zato je $out[B_2] = \{b, d\}$. Analizom unazad po bloku B_2 dobija se da je $in[B_2] = \{a, b\}$.

– Sledbenici bloka B_1 su B_2 i B_3, pa je $out[B1] = in[B_2] \cup in[B_3] = \{a, b, d\}$. Analizom unatrag po tom bloku dobijamo da je $in[B_1] = \emptyset$.

– Može se primetiti da x nikad nije živa promenljiva (pa se dodela `x := 100` može ukloniti). Slično ni vrednost `c := a + b` nije živa, pa se i ta dodela može ukloniti.

– Može se i primetiti da, na primer, promenljive `a`, `t1` i `t2` nikada nisu istovremeno žive, pa se za njihovo smeštanje može upotrebiti jedan isti registar. Na osnovu ovih uvida kod se može uprostiti na sledeći način.

```
B1: a := 3
    b := 5
    d := 4
    ifFalse a > b goto b3

B2: d := 2

B3: c := 4
    a := b * d
    a := a + c
    return a
```

- Prikažimo analizu živosti na još jednom kompleksnijem primeru (sa petljom).

```
B1:
    a := 0
B2:
    b := a + 1
    c := c + b
    a := b * 2
    if a < 100 goto B2
B3:
    return c
```

– Na kraju bloka B_3 živa je promenljiva c. Zato je $in[B_3] = \{c\}$.

– Sledbenici bloka B_2 su B_2 i B_3. Pretpostavili smo da je $in[B_2] = \emptyset$ i zato je $out[B_2] = in[B_3] \cup in[B_2] = \{c\}$.

– Na osnovu ovoga vršimo analizu bloka B_2 unatrag.

```
                        {a, c} = {a} + ({b, c} - {b})
  b := a + 1
                        {b, c} = {b, c} + ({b, c} - {c})
  c := c + b
                        {b, c} = {b} + ({a, c} - {a})
  a := b * 2
                        {a, c} = {a} + ({c} - {})
  if a < 100 goto B2
                        {c}
```

– Dobijamo da je $in[B_2] = \{a, c\}$, što je drugačije od naše početne procene da je $in[B_2] = \emptyset$. Stoga moramo ponovo da izračunamo $out[B_2] = in[B_2] \cup in[B_3]$ i dobijamo da je to jednako $\{a, c\}$.

– Ponovo vršimo analizu bloka B_2.

```
                        {a, c}
  b := a + 1
                        {b, c}
  c := c + b
                        {b, c}
  a := b * 2
                        {a, c}
  if a < 100 goto B2
                        {a, c}
```

– Ovaj put nije bilo promena i ne vrši se dalje ažuriranje.

– Važi da je $out[B_1] = in[B_2] = \{a, c\}$. Analizom bloka B_1 zaključujemo da je $in[B_1] = \{c\}$.

– Konačni rezultat analize se može prikazati na sledećem dijagramu.

```
B1:
                          {c}
     a := 0
                          {a, c}

  -------------------------------

                          {a, c}
B2:
     b := a + 1
                          {b, c}
     c := c + b
                          {b, c}
     a := b * 2
                          {a, c}
```

```
     if a < 100 goto B2
                          {a, c}

--------------------------------
                          {c}
B3:
     return c
                          {}
```

11.3.1 Lanci def-use

- Lanci def-use za svaku definiciju promenljive određuju skup instrukcija u kojima se ta definicija koristi.
- Promenljiva uvedena nekom definicijom je mrtva ako i samo ako joj je def-use lanac prazan.
- Def-use lanci se mogu odrediti veoma sličnom postupkom kao što je analiza živosti.

11.3.2 Upotreba živosti promenljivih

- Važna upotreba živosti je u procesu alokacije registara, o čemu će biti reči kasnije.
- Ako postoji instrukcija s oblika `a := b op c` (ili nekog sličnog u kojem se definiše `a`) takva da se `a` ne nalazi u skupu $out[s]$, tada je ova instrukcija mrtva i može se obrisati.

11.4 Dostupni izrazi

- Izraz `x op y` je dostupan u instrukciji n ako se na svakoj putanji od početnog čvora do čvora n u grafu kontrole toka programa vrednost izraza `x op y` izračunava bar jednom i ako ni na jednom putanji ne postoje nove definicije promenljivih `x` i `y` nakon poslednjeg izračunavanja `x op y` na toj putanji.
- Svaka instrukcija koji izračunava `x op y` generiše `x op y`, a svaka instrukcija koji definiše `x` ili `y` ubija `x op y`. Npr. `c := a + b` generiše `a + b`, ali `a := a + b` ne generiše `a + b` (jer je odmah nakon generisanja ubija). To se može formalizovati tako što je za definiciju s koje sa desne strane ima `x op y` kažemo da je $gen[s] = (xopy)$ $kill[s]$.
- Na osnovu ovoga možemo analizirati dostupnost izraza u narednom baznom bloku.

```
               {}
a := b + c
```

```
                {b+c}
b := a - d
                {a-d}
c := b + c
                {a-d}
d := a - d
                {}
```

Činjenica da je `a-d` dostupan pre `d := a-d` omogućava njegovu eliminaciju, dok čijenica da `b + c` nije dostupan pre `c := b + c` takvu eliminaciju onemogućava.

- Analiza dostupnosti teče unapred, slično kao kod analize dostižnosti definicija, međutim, pošto se traži definisanost na svim putanjama umesto unije se koristi presek.

$$in[B] = \bigcap_{P-\text{prethodnik čvora B}} out[P]$$

$$out[s] = gen[s] \cup (in[s] \quad kill[s])$$

- Inicijalno se postavlja da su svi *in* i *out* skupovi puni (a ne prazni) tj. univerzalni skupovi koji sadrže sve moguće izraze, osim što je $out[Entry] = \emptyset$.

11.4.1 Dosezajući izrazi

- Dosezajući izrazi (engl. reaching expressions)
- Izraz `x op y` u instrukciji s oblika `t := x op y` doseže instrukciju n ako postoji putanja od s do n na kojoj se ne vrši dodela promenljivama `x` niti `y` niti se ponovo računa `x op y`.
- Dosezajući izrazi se mogu odrediti analizom unatrag od instrukcije n ili tokom određivanja dostupnih izraza (to su oni izrazi na pojedinačnim putanjama koji zapravo čine izraz `x op y` dostupnim u n).

11.4.2 Upotreba dostupnih izraza (globalna eliminacija zajedničkih podizraza)

- Dostupni i dosezajući izrazi se koriste prilikom eliminacije zajedničkih podizraza.
- Ako postoji instrukcija n oblika `t := x op y` takva da je `x op y` dostupan izraz, tada se pronađu svi dosezajući izrazi tj. sve instrukcije s u kojima se računa `v := x op y`. Uvodi se nova pomoćna promenljiva `w` i sve

instrukcije s se menjaju sa `t := x op y` i zatim `v := w`. Time se obezbeđuje da se na svim putanjama pre n vrednost izraza nalazi u istoj promenljivoj `w`, pa se naredba `n` može zameniti naredbom `t := w` (bez uvođenja te nove privremene promenljive ne bismo znali koju promenljivu da uvedemo umesto `x op y` u n, jer se vrednost izraza, iako je dostupna, na različitim granama može nalaziti smeštena u različitim promenljivima.

- Razmotrimo naredni primer.

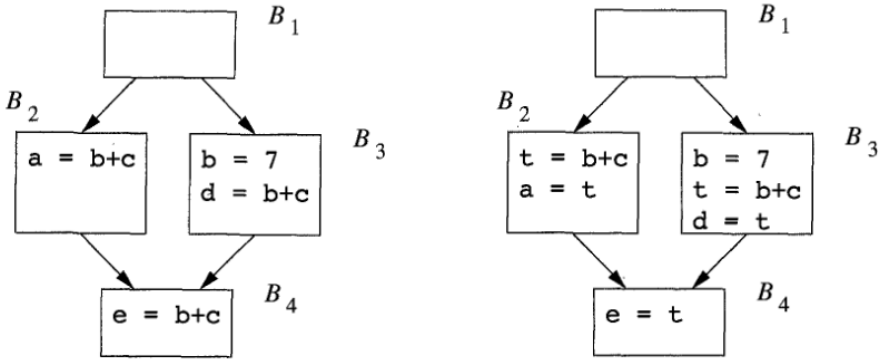


Figure 19: Primer globalne eliminacije zajedničkih podizraza

11.5 Rezime tri primera analize

.	Dosezajuće definicije	Žive promenljive	Dostupni izrazi
Domen	Skup definicija	Skup promenljivih	Skup izraza
Smer	unapred	unatrag	unapred
Transfer	$gen[B] \cup (X \quad kill[B])$	$use[B] \cup (X \ def[B])$	$gen[B] \cup (x$ $kill[B])$
Granica	$out[Entry] = \emptyset$	$in[Exit] = \emptyset$	$out[Entry] = \emptyset$
Jednačine	$out[B] = f_B(in[B])$	$in[B] = f_B(out[B])$	$out[B] =$ $f_B(in[B])$
Jednačine	$in[B] =$ $\bigcup_{P \in pred(B)} out[P]$	$out[B] =$ $\bigcup_{S \in succ(B)} in[S]$	$in[B] =$ $\bigcap_{P \in pred(B)} out[P]$
Inicijalizacija	$out[B] = \emptyset$	$in[B] = \emptyset$	$out[B] = U$

11.6 Teorijska uopštenja analize toka podataka

- Postoji teorijski okvir u kojem se mogu zajednički izraziti različite analize toka podataka.

- Na osnovu teorije dobijaju se odgovori na sledeća pitanja:
  - Pod kojim uslovima je analiza toka podataka korektna?
  - Koliko je precizno rešenje dobijeno iterativnim algoritmom?
  - Zašto iterativni algoritam konvergira?
  - Koje je značenje rešenja jednačina?
- Formulisanje apstraktnog okvira omogućava da se analiza prethodnih pitanja ne mora sprovoditi za svaku analizu toka podataka pojedinačno, već je moguće pozvati se na opšte teoreme.
- Apstraktni okvir analize toka podataka je uređena četvorka $(D, V, \wedge, F)$ koja se sastoji od:
  - Smera toka podataka D (unapred ili unatrag)
  - Polumreže koja uključuje domen V i operator spajanja (engl. meet) $\wedge$.
  - Familije transfer funkcija iz V u V koja uključuje i specijalne transfer funkcije za granične uslove (čvorove *Entry* ili *Exit*).
- Polumreže indukuju parcijalna uređenja dok operator $\wedge$ ima ulogu infimuma ($x \leq y$ akko $x \wedge y = x$).
- Familije skupova čine polumreže i u odnosu na $\cup$ (tada je najveći element $\emptyset$, a najmanji univerzalni skup U) i u odnosu na $\cap$ (tada je najveći element univerzalni skup U, a najmanji je $\emptyset$). U prvom slučaju je poredak $\supseteq$, a u drugom $\subseteq$.
- Najpreciznije je rešenje koje je najveće u parcijalnom poretku.
- Apstraktni okvir je monoton akko za svako $x, y \in V$ if $f \in F$ važi da iz $x \leq y$ sledi $f(x) \leq f(y)$ tj. ekvivalentno $f(x \wedge y) \leq f(x) \wedge f(y)$.
- Apstraktni okvir je distributivan akko za svako $x, y \in V$ if $f \in F$ važi da je $f(x \wedge y) = f(x) \wedge f(y)$. Svi distributivni okviri su monotoni (obratno ne mora da važi).
- Ako je $f(X) = G \cup (X \quad K)$, tada je okvir sigurno distributivan (a takvi su svi do sada viđeni primeri).
- Naredni algoritmi formulišu analizu toka podataka u apstraktnim okvirima i to unapred i unatrag (prikazujemo algoritam unapred).

```
out[Entry] := vEntry
for (svaki bazični blok B osim Entry) out[B] = Top
repeat
  for (svaki bazični blok B osim Entry)
     in[B] = /\_{P je prethodnik B} out[B]
     out[B] = fB(in[B])
until (bilo je promena u out)
```

- Može se dokazati sledeće.

  - Ako algoritam konvergira, *in* i *out* skupovi daju neko ispravno rešenje problema odgovarajućeg problema toka podataka.
  - Ako je okvir monoton, dobijeno rešenje je maksimalna fiksna tačka (engl. maximal fixed point, MFP), tj. vrednosti $out[B]$ i $in[B]$ za bilo koje drugo rešenje su uvek $\leq$ od vrednosti u rešenju koje algoritam pronalazi.
  - Ako je mreža problema monotona i konačne visine, algoritam će sigurno konvergirati.

- Može se definisati idealno rešenje koje u obzir uzima sve *moguće* (one kojima se stvarno može proći) putanje kroz graf i izračunava infimum svih tako pronađenih vrednosti (ono se ne može efektivno izračunati - slično kao halting problem). Svako rešenje koje je strogo veće od idealnog je korektno. Svako rešenje koje je manje ili jednako od idealnog je konzervativno tj. bezbedno. Ako je rešenje veće od idealnog, ono sigurno zanemaruje neku putanju i ta putanja može dovesti do nekorektnog ponašanja. Ako je rešenje manje od idealnog ono u obzir uzima sve putanje u programu i još neke koje ne postoje stvarno, ali to ne smeta.

- Aproksimacija idealnog rešenja je da se gledaju sve putanje kroz graf (ne samo one moguće). Ovo se zove susret-nad-putanjama (engl. meet-over-paths, MOP). Na primer, putanja do `x = 3` u narednom primeru nije moguća, ali otkrivanje toga koja je putanja moguća, a koja nije zahteva veoma kompleksnu analizu (neodlučivu u opštem slučaju).

  ```
  if (a < b)
    if (a > b)
      x = 3;
  ```

  Ova aproksimacija uvek daje rešenje koje je manje ili jednako od idealnog. Može se pokazati da efektivno opisani algoritam daje uvek rešenja koja su manja ili jednaka od MOP rešenja i stoga su rešenja koja daje algoritam uvek konzervativna i bezbedna.

11.7 Globalna propagacija konstanti

- Prikažimo interesantan okvir za analizu toka podataka koji se koristi za propagaciju konstanti.
  - Okvir ima neograničen skup mogućih vrednosti toka podataka (za razliku od svih do sada koji su imali samo konačno mnogo vrednosti).
  - Okvir nije distributivan.

- Mreža za analizu pojedinačne promenljive ima sledeće moguće vrednosti toka podataka:
  - sve konstante tipa podataka te promenljive,

– vrednost NAC (engl. not a constant) koja će biti dodeljivana promenljivama za koje se utvrdi da nemaju konstantnu vrednost (npr. jer joj se dodeljuje izraz u kome učestvuje druga promenljiva koja nije konstantna ili jer joj se na različitim putanjama dodelju različite konstante)
– vrednost $UNDEF$ koja označava da još ništa ne znamo o vrednosti ove promenljive.
– važi da je $UNDEF \wedge v = v$, $NAC \wedge v = NAC$, za konstantu c je $c \wedge c = c$, a za različite konstante c_1 i c_2 je $c_1 \wedge c_2 = NAC$.

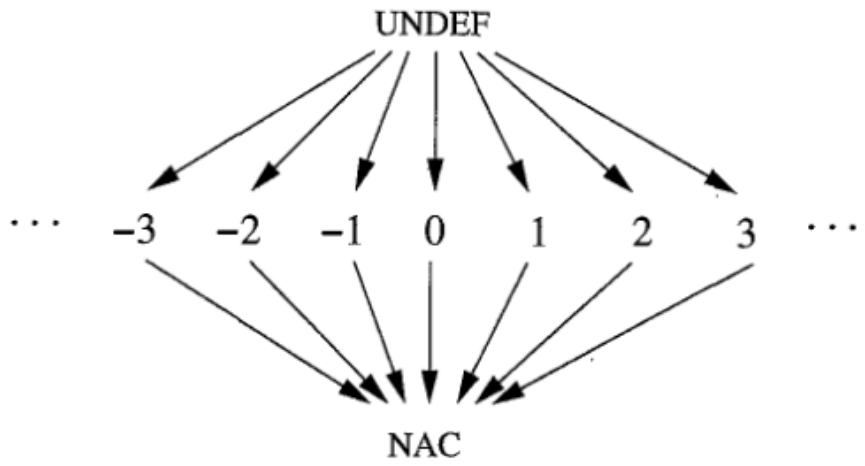


Figure 20: Mreža vrednosti okvira za globalnu propagaciju konstanti

- Funkcije transfera se mogu opisati na sledeći način:
  – Na početku analize je svakoj promenljivoj dodeljena vrednost $UNDEF$.
  – Ako s nije dodela vrednosti dodeljene vrednosti promenljivama se ne menjaju.
  – Ako je s dodela vrednosti promenljivoj `x` tada se vrednosti dodeljene drugim promenljivama, dok se vrednost promenljive `x` menja na sledeći način:
    * ako je dodela oblika `x := c` tada se promenljivoj `x` dodeljuje vrednost `c`.
    * ako je dodela oblika `x := y op z` ako su promenljivama `y` i `z` dodeljene konstantne vrednosti tada se promenljivoj `x` dodeljuje njihov zbir, ako je nekoj od njih dodeljena vrednost NAC, onda se i promenljivoj `x` dodeljuje vrednost NAC, dok se u suprotnom promenljivoj x dodeljuje vrednost $UNDEF$.
    * u suprotnom (npr. ako se `x` dodeljuje vrednost nekog poziva funkcije) se promenljivoj `x` dodeljuje vrednost NAC.

- Analiza je unapred i na početku su svim promenljivama dodeljene vrednosti $UNDEF$. Vrednosti ulaza u svaki blok se za svaku promenljivu računaju tako što se izračuna infimum izlaznih vrednosti za tu promenljivu duž svih blokova prethodnika tog bloka.
- Razmatrajmo vrednost promenljive `x` u narednom primeru. Da li je ona globalno konstantna? Kako se to prethodnom analizom utvrđuje?

```
    x := 3
    if b > 0 goto L1
    y := z + x
    goto L2
L1:
    y := 0
L2:
    a := 2 * x
    if a < b goto L1
```

  - Uradimo jednu iteraciju algoritma.

```
                        x = UNDEF
    x := 3
                        x = 3
    if b > 0 goto L1
                        x = 3
-----------------------------------------
                        x = 3
    y := z + x
                        x = 3
    goto L2
                        x = 3
-----------------------------------------
                        x = 3 /\ UNDEF = 3
L1:
    y := 0
                        x = 3
-----------------------------------------
                        x = 3 /\ 3 = 3
L2:
    a := 2 * x
                        x = 3
    if a < b goto L1
                        x = 3
```

  - Pošto se vrednost na izlazu iz poslednjeg bloka promenila, menja se vrednost na ulazu u blok `L1`. Međutim, to ne utiče na rezultat.
  - Zaključujemo da je `x` globalna konstanta i kod se može uprostiti. Nakon dodatnog izračunavanja konstantnih izraza i eliminisanja

mrtvog koda dobija se sledeći kod.

```
    if b > 0 goto L1
    y := z + 3
    goto L2
L1:
    y := 0
L2:
    if 6 < b goto L1
```

- Razmotrimo šta bi bilo da je primer malo drugačiji.

```
    x := 3
    if b > 0 goto L1
    y := z + x
    goto L2
L1:
    x := 0
L2:
    a := 2 * x
    if a < b goto L1
```

  – Sprovedimo prvu iteraciju algoritma

```
                              x = UNDEF
   x := 3
                              x = 3
   if b > 0 goto L1
                              x = 3
   ------------------------------------------

                              x = 3
   y := z + x
                              x = 3
   goto L2
                              x = 3
   ------------------------------------------
                              x = UNDEF /\ 3 = 3
L1:
   x := x + 2
                              x = NAC
   ------------------------------------------
                              x = 3 /\ NAC = NAC
L2:
   a := 2 * x
                              x = NAC
   if a < b goto L1
                              x = NAC
```

– Pošto se promenio *out* za poslednji blok menja se *in* za blok `L1` i ponovo vršimo izračunavanje.

```
                              x = UNDEF
   x := 3
                              x = 3
   if b > 0 goto L1
                              x = 3
-----------------------------------------


                              x = 3
   y := z + x
                              x = 3
   goto L2
                              x = 3
-----------------------------------------
                              x = NAC /\ 3 = NAC
L1:
   x := x + 2
                              x = NAC
-----------------------------------------
                              x = NAC /\ NAC = NAC
L2:
   a := 2 * x
                              x = NAC
   if a < b goto L1
                              x = NAC
```

– Možemo primetiti da `x` ima konstantnu vrednost 3 pre instrukcije `y := z + x`, ali ne i kasnije. Na tom mestu možemo uvrstiti konstantnu vrednost.

```
   x := 3
   if b > 0 goto L1
   y := z + 3
   goto L2
L1:
   x := 0
L2:
   a := 2 * x
   if a < b goto L1
```

- Razmotrimo i naredni, malo složeniji primer (koji izračunava elemente Fibonačijevog niza).

```
1:  n := 10
2:  older := 0
3:  old := 1
```

```
4:  result := 0
5:  if n <= 1 goto 14
6:  i := 2
7:  if i > n goto 13
8:  result := old + older
9:  older := old
10: old := result
11: i := i + 1
12: goto 7
13: return result
14: return n
```

– Sprovedimo prvu iteraciju algoritma.

```
                    {n: undef, old: undef, older: undef, result: undef, i: undef}
1:  n := 10
                    {n: 10, old: undef, older: undef, result: undef, i: undef}
2:  older := 0
                    {n: 10, old: undef, older: 0, result: undef, i: undef}
3:  old := 1
                    {n: 10, old: 1, older: 0, result: undef, i: undef}
4:  result := 0
                    {n: 10, old: 1, older: 0, result: 0, i: undef}
5:  if n <= 1 goto 14
                    {n: 10, old: 1, older: 0, result: 0, i: undef}
-------------------------------------------------------------------------------
                             nasledjeno od 5
                    {n: 10, old: 1, older: 0, result: 0, i: undef}
6:  i := 2
                    {n: 10, old: 1, older: 0, result: 0, i: 2}
-------------------------------------------------------------------------------
                             susret 6 i 12
                    {n: 10, old: 1, older: 0, result: 0, i: 2}
7:  if i > n goto 13
                    {n: 10, old: 1, older: 0, result: 0, i: 2}
-------------------------------------------------------------------------------
                             nasledjeno od 7
                    {n: 10, old: 1, older: 0, result: 0, i: 2}
8:  result := old + older
                    {n: 10, old: 1, older: 0, result: 1, i: 2}
9:  older := old
                    {n: 10, old: 1, older: 1, result: 1, i: 2}
10: old := result
                    {n: 10, old: 1, older: 1, result: 1, i: 2}
11: i := i + 1
                    {n: 10, old: 1, older: 1, result: 1, i: 3}
12: goto 7
```

```
                    {n: 10, old: 1, older: 1, result: 1, i: 3}
    ------------------------------------------------------------------------------
                              nasledjeno od 7
                    {n: 10, old: 1, older: 0, result: 0, i: 2}
    13: return result
                    {n: 10, old: 1, older: 0, result: 1, i: 2}
    ------------------------------------------------------------------------------
                              nasledjeno od 5
                    {n: 10, old: 1, older: 0, result: 0, i: undef}
    14: return n
                    {n: 10, old: 1, older: 0, result: 0, i: undef}
```

– Pošto se rezultat iza linije 12 promenio, moramo da pravimo još iteracija.

```
~~~
                   {n: undef, old: undef, older: undef, result: undef, i: undef}
1:  n := 10
                   {n: 10, old: undef, older: undef, result: undef, i: undef}
2:  older := 0
                   {n: 10, old: undef, older: 0, result: undef, i: undef}
3:  old := 1
                   {n: 10, old: 1, older: 0, result: undef, i: undef}
4:  result := 0
                   {n: 10, old: 1, older: 0, result: 0, i: undef}
5:  if n <= 1 goto 14
                   {n: 10, old: 1, older: 0, result: 0, i: undef}
------------------------------------------------------------------------------
                             nasledjeno od 5
                   {n: 10, old: 1, older: 0, result: 0, i: undef}
6:  i := 2
                      {n: 10, old: 1, older: 0, result: 0, i: 2}
------------------------------------------------------------------------------
                             susret 6 i 12
                   {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
7:  if i > n goto 13
                   {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
------------------------------------------------------------------------------
                             nasledjeno od 7
                   {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
8:  result := old + older
                   {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
9:  older := old
                   {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
10: old := result
                   {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
11: i := i + 1
```

```
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
12: goto 7
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
-------------------------------------------------------------------------------
                           nasledjeno od 7
                  {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
13: return result
                  {n: 10, old: 1, older: NAC, result: NAC, i: NAC}
-------------------------------------------------------------------------------
                           nasledjeno od 5
                  {n: 10, old: 1, older: 0, result: 0, i: undef}
14: return n
                  {n: 10, old: 1, older: 0, result: 0, i: undef}

~~~
```

– Pošto se ponovo rezultat iza linije 12 promenio, moramo da pravimo još iteracija.

```
                      {n: undef, old: undef, older: undef, result: undef, i: undef}
   1:  n := 10
                      {n: 10, old: undef, older: undef, result: undef, i: undef}
   2:  older := 0
                      {n: 10, old: undef, older: 0, result: undef, i: undef}
   3:  old := 1
                      {n: 10, old: 1, older: 0, result: undef, i: undef}
   4:  result := 0
                      {n: 10, old: 1, older: 0, result: 0, i: undef}
   5:  if n <= 1 goto 14
                      {n: 10, old: 1, older: 0, result: 0, i: undef}
   -------------------------------------------------------------------------------
                               nasledjeno od 5
                      {n: 10, old: 1, older: 0, result: 0, i: undef}
   6:  i := 2
                      {n: 10, old: 1, older: 0, result: 0, i: 2}
   -------------------------------------------------------------------------------
                               susret 6 i 12
                      {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
   7:  if i > n goto 13
                      {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
   -------------------------------------------------------------------------------
                               nasledjeno od 7
                      {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
   8:  result := old + older
                      {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
   9:  older := old
                      {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
```

```
10: old := result
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
11: i := i + 1
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
12: goto 7
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
-------------------------------------------------------------------------------
                           nasledjeno od 7
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
13: return result
                  {n: 10, old: NAC, older: NAC, result: NAC, i: NAC}
-------------------------------------------------------------------------------
                           nasledjeno od 5
                  {n: 10, old: 1, older: 0, result: 0, i: undef}
14: return n
                  {n: 10, old: 1, older: 0, result: 0, i: undef}
```

12 Alokacija registara

- Prilikom generisanja međukoda ne vodi se računa o tome gde će biti smeštene promenljive (originalne i privremene). Privremene promenljive se uvode gde god je to potrebno.
- Zadatak faze *alokacije registara* je da za svaku promenljivu odredi lokaciju na kojoj će ona biti smeštena.
  - lokacija može biti u registar procesora
  - lokacija može biti u memoriji (u stek okviru) Poželjno je da se što više promenljivih smesti u registre (jer im se može brže pristupiti).
- Postoje razlike između RISC i CISC arhitektura (kod RISC-a su tipično svi registri ekvivalentni, pa je alokacija registara jednostavnija, dok kod CISC-a često registri imaju neke svoje specifičnosti).
- Osnovno ograničenje: promenljive koje su istovremeno žive se moraju smestiti na različite lokacije!
- Nekada postoje i dodatna ogrančenja (npr. ako instrukcija `a := b op c` ne može da proizvede rezultat u registru `r1`, onda lokacija promenljive `a` ne može biti registar `r1`).

12.1 Bojenje grafa

- Pristup: konstruiše se *graf interferencije registara* (engl. *register interference graph*, *RIG*) u kojem su čvorovi promenljive, a grana između dve promenljive postoji akko su te dve promenljive u nekom trenutku istovremeno žive. Svakom registru se jednoznačno dodeli neka boja i tim bojama bojimo graf tako da nikoja dva susedna čvora nisu obojena istom bojom.

12.1.1 Bojenje uprošćavanjem

- I alokacija registara i bojenje grafa su NP kompletni problemi (nije poznato da li postoji efikasno rešenje - do sada nije pronađeno).
- Primenjuju se heuristike linearne složenosti koje u praksi daju dobre rezultate.
- Heuristika uprošćavanja se sastoji od sledećih koraka: `Build`, `Simplify`, `Spill`, `Select`, `Start over`.

`Build` Na osnovu analize živosti promenljivih, konstruiše se graf G interferencije registara.

`Simplify` Neka je K broj registara koji su nam na raspolaganju. Ako graf sadrži čvor m koji ima strogo manje od K suseda, tada, ako se može obojiti graf $G \setminus \{m\}$, tada se može obojiti i graf G. Zaista, susedi čvora m će biti obojeni sa strogo manje od K boja i postojaće sigurno boja kojom možemo da obojimo i čvor m. Ovo dovodi do rekurzivnog algoritma: uklanjamo jedan po jedan čvor stepena manjeg od K iz grafa (stavljamo ih na stek), sve dok se graf ne isprazni, a kasnije i vraćamo jedan po jedan i dodeljujemo im boje. Čvor se uklanja zajedno sa svojim granama, tako da se stepeni ostalih čvorova smanjuju što povećava šansu da se ostali čvorovi mogu ukloniti.

`Spill` Pretpostavimo da se u nekom trenutku ne može ukloniti ni jedan čvor, tj. da svi čvorovi grafa imaju *značajan stepen* tj. stepe veći ili jednak K. Tada se (heuristički) korak `Simplify` ne može primeniti. U tom trenutku se bira neki čvor koji se *prosipa*, tj. potencijalno se umesto u registar smešta u memoriju. *Optimistički* pristup uklanja taj čvor iz grafa i stavlja ga na stek (ne menjajući orginalni program). Naime, iako taj čvor ima više ili jednako od K suseda, moguće je da se desi da će neki njegovi susedi biti obojeni istim bojama i da će se za njega ipak naći slobodna boja (u tom slučaju se on zapravo neće prosuti u memoriju, već će moći da ostane u registru). Ako se to ne desi došlo je do *stvarnog prosipanja* koje zahteva izmenu programa (dodavanjem instrukcija pristupa memoriji) i pokretanje postupka iznova. Izbor promenljive koja se prosipa vrši se ponovo na osnovu heuristika. Osnovni kriterijumi su da je to promenljiva koja ima veliki stepen u grafu, ali koja se ne javlja u petljama u programu (naročito je bitno izbegavati prosipanje promenljivih u unutrašnjim petljama, jer bi njihovo stvarno prosipanje dovelo do jako velikog broja pristupa memoriji).

`Select` Korak `Select` je inverzni korak prethodnima. Kreće se od praznog grafa, skida se jedan po jedan čvor i dodeljuje mu se boja (ako je čvor bio dodat na stek u koraku `Simplify`, boju ćemo mu sigurno moći dodeliti, a ako je dodat u koraku `Spill`, to nije sigurno). Ako se nekom čvoru boja ne može dodeliti, on se proglašava za stvarno prosuti čvor i `Select` korak se nastavlja dalje, da bi se identifikovali drugih stvarno prosuti čvorovi.

`Start over` Ako graf nije uspešno obojen, tada se program menja tako da

se nakon definicije svake stvarno prosute promenljive umeće instrukcija njenog prebacivanja u memoriju, a pre svakog njenog korišćenja se umeće instrukcija njenog prebacivanja iz memorije. Na primer, ako je `a` stvarno prosuta promenljiva, kod se transformiše na sledeći način.

```
b := 3 + x          b := 3 + x
a := b * c          a := b * c
c := b + d          M[a_loc] := a
d := a + 2          c := c + d
c := d + b          a := M[a_loc]
b := -a             d := a + 2
                    c := d + b
                    a := M[a_loc]
                    b := -a
```

Time se životni vek (u procesoru) promenljive `a` skraćuje i oslobađaju se registri za smeštanje drugih promenljivh. Npr. promenljiva `c` u instrukciji `c := b + d` se nakon prebacivanja `a` u memoriju može prebaciti u registar u kojem je smešten `a` (jer je njegova vrednost smeštena u memoriju i ona nije živa, jer se u svakom korišćenju učitava iz memorije, tj. nakon `M[a_loc] := a`, ta promenljiva `a` više nije živa). Nakon izmene programa ceo algoritam se ponavlja.

12.1.2 Primer

Posmatrajmo naredni fragment koda i označene žive promenljive u svakoj tački.

```
                        {j, k}
g := mem[j + 12]
                        {j, g, k}
h := k - 1
                        {j, g, h}
f := g * h
                        {j, f}
e := mem[j + 8]
                        {e, j, f}
m := mem[j + 16]
                        {e, m, f}
b := mem[f]
                        {e, m, b}
c := e + 8
                        {c, m, b}
d := c
                        {d, m, b}
k := m + 4
                        {d, k, b}
```

```
j := b
```

{d, k, j}

Ovom programu odgovara naredni graf.

```
live-in: k j
        g := mem[j+12]
        h := k - 1
        f := g * h
        e := mem[j+8]
        m := mem[j+16]
        b := mem[f]
        c := e + 8
        d := c
        k := m + 4
        j := b
live-out: d k j
```

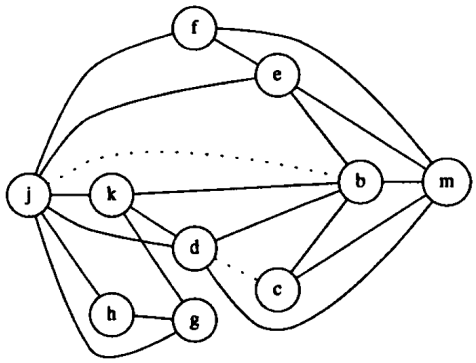


Figure 21: Primer programa i grafa

Pretpostavimo da na raspolaganju imamo 4 registra. U listu kandidata za izbacivanje možemo dodati čvorove `g`, `h`, `c` i `f`, pošto oni imaju manje od 4 suseda. Nakon uklanjanja čvorova `g` i `h` (i njihovog postavljanja na stek) i čvor `k` ima manje od 4 suseda i on se dodaje u listu. Nakon njegovog uklanjanja dobija se graf na narednoj slici.

Uklanjanje može da nastavi dalje i dobija se stanje na narednoj slici. Na kraju, kada se graf isprazni registre možemo dodeliti u obratnom redosledu.

12.2 Stapanje

- *Stapanje* (engl. *coalescing*) odmenjuje propagaciju kopija i eleminiše nepotrebne dodele.
- Kada se vrši dodela promenljive promenljivoj, one se mogu u grafu označiti isprekidanom linijom.
- U principu, svaki par čvorova grafa koji nije povezan granom se može stopiti, međutim, to bi dovelo verovatno do toga da se dobijeni graf ne može obojiti (jer bi novi čvor sadržao uniju grana polazna dva čvora).
- Primenjuju se strategije *sigurno stapanja* koje garantuje da se ako se graf može obojiti pre stapanja, onda se može obojiti i nakon stapanja.
- Postoje dve strategije sigurnog stapanja.
  - (Brigs) Dva čvora a i b se mogu stopiti ako je rezutujući čvor ab imati manje od K suseda značajnog stepena. Ovo ne može da pokvari obojivost (jer će nakon uklanjanja svih čvorova beznačajnog stepena, stopljeni čvor biti taj koji će moći da se ukloni u koraku `Simplify`)

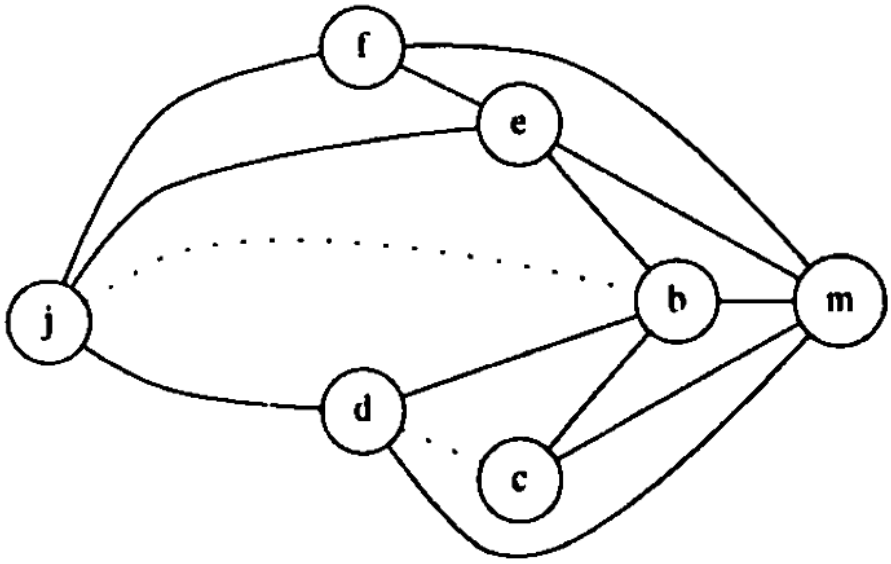


Figure 22: Graf nakon uklanjanja čvorova g, h i k

(a) stack	(b) assignment
m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4

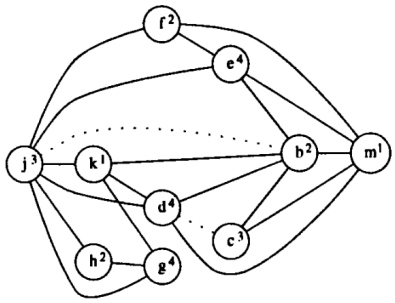


Figure 23: Stek i dobijeno bojenje

  - (Džordž) Dva čvora a i b se mogu stopiti ako se svaki sused čvora a značajnog stepena već u grafu povezan sa b.
- Korak stapanja `Coalesce` se izvodi u kombinaciji sa korakom `Simplify`, pri čemu se u `Simplify` prvo uklanjanju čvorovi koji ne učestvuju u propagaciji kopija (oni koji nemaju isprekidane linije), a u koraju `Coalesce` se vrši bezbedno stapanje onih čvorova koji su spojeni isprekidanim linijama (i iz koda se brišu odgovarajuće dodele kopija). Pošto se stapanjem uklanjaju isprekidane linije, stopljeni čvor može postati kandidat za `Simplify`. Ako nije moguće primeniti ni `Simplify` ni `Coalesce`, tada se korakom `Freeze` briše neka isprekidana linija (odričemo se mogućnosti da stopimo dve promenljive), u nadi da će to omogućiti `Simplify`. Tek kada `Freeze` nije više moguć, primenjuje se `Spill`.

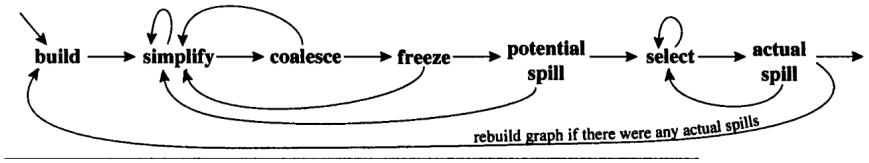


Figure 24: Dijagramski prikaz algoritma registarske alokacije sa stapanjem

13 Asemblerski i mašinski jezici

- Asemblerski i mašinski jezik su *mašinski zavisni* jezici i razlikuju se od jedne do druge arhitekture računara.
- Za PC računare najraspostranjenije su 32-bitna arhitektura x86 (piše se i x86-32) i njena naslednica, 64-bitna arhitektura x86-64 (piše se i x64 ili AMD64). Ove arhitekture koriste Inte-ovi i AMD-ovi procesori. One se razlikuju po mnogim aspektima, mada ima i dosta sličnosti.

13.1 Registri

- Širina registara je od 8 pa sve do 512bita. Koriste se termini byte (8 bita), word (16 bita), dword tj. double word (32 bita), qword tj. quad word (64 bita).
- x86-32 aritektura koristi sledeće registre opšte namene:
- x86-64 aritektura koristi sledeće registre opšte namene:
- Delovi širih registara dostupni su kao uži registri. Npr. u nižih 32 bita registra `rax` može se koristiti istovremeno kao registar `eax`. Ne mogu se koristiti istovremeno, jer dele isti prostor.

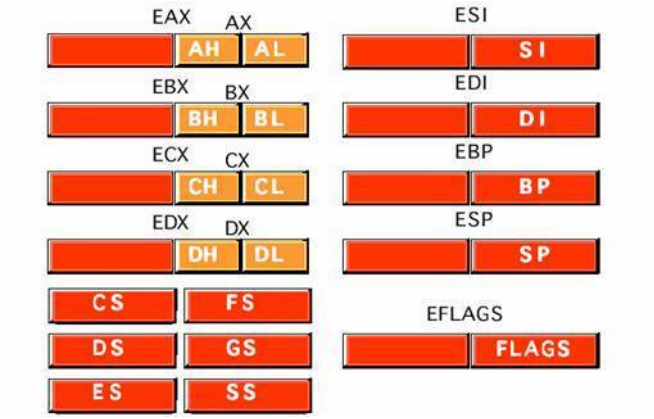


Figure 25: x86-32 registri opšte namene

13.2 Organizacija memorije

13.3 Instrukcije

13.4 Režimi adresiranja

13.5 Cisc, Risc

14 Aktivacioni slogovi

14.1 Stek okviri i njihov sadržaj

14.2 Konvencije pozivanja funkcija

14.2.1 Prenos parametara

14.2.2 Povratne vrednosti

14.2.3 Čuvanje registara

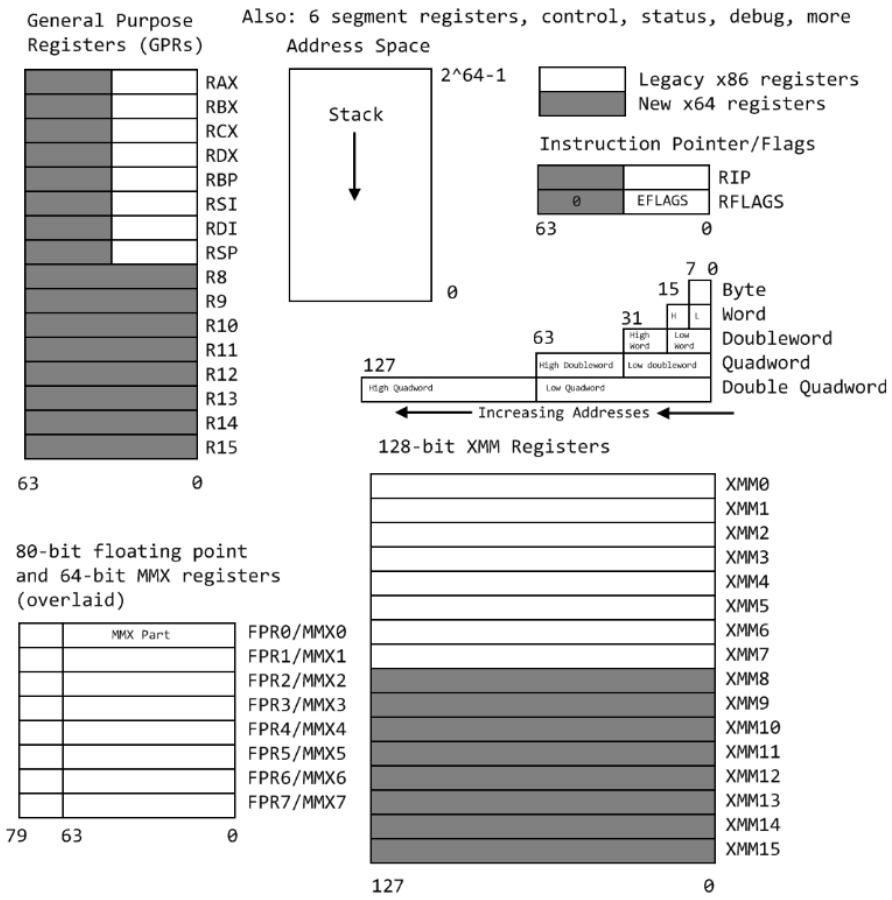


Figure 26: x86-64 registri

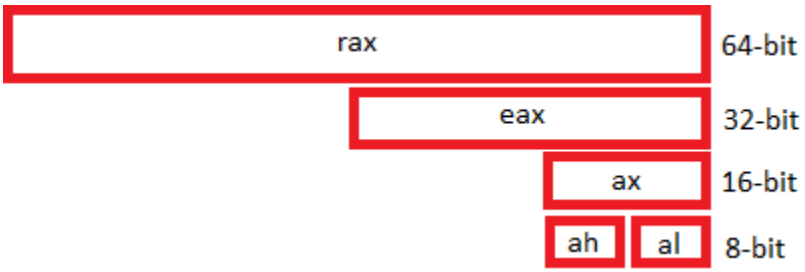


Figure 27: Deljenje prostora u registrima