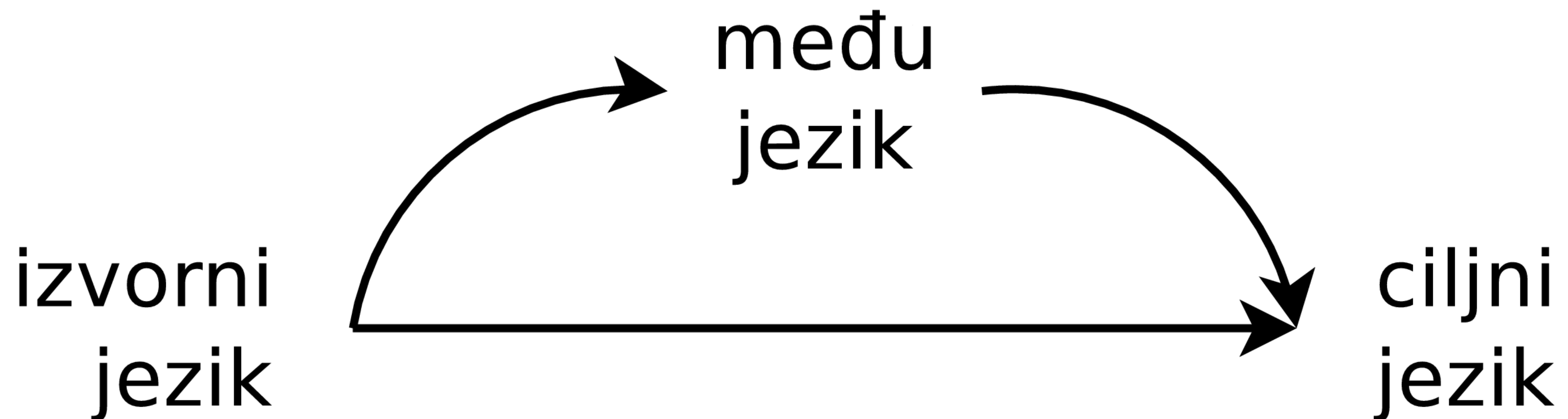
Međukod

- Međujezik - jezik između izvornog jezika i ciljnog jezika
  - IL = *intermediate language*
  - na nivou apstrakcije
    - više detalja od izvornog jezika (npr: registri)
    - manje detalja od ciljnog jezika
  - većina kompajlera ima IL



Međukod

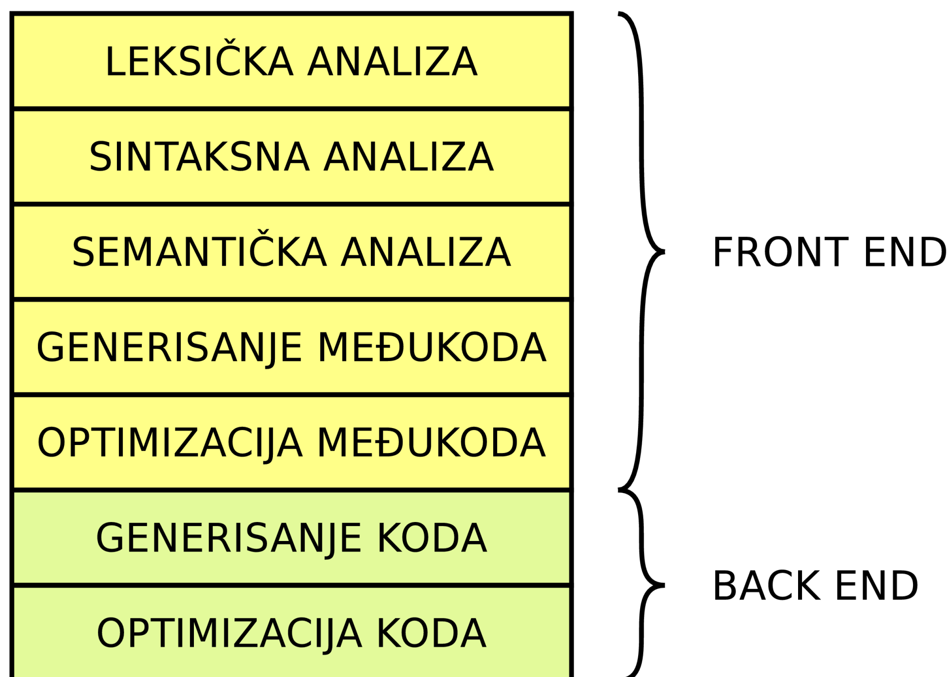
- **Faze kompajliranja**
  - leksička analiza
  - sintaksna analiza
  - semantička analiza
  - generisanje koda
  - optimizacija
- Faze kompajliranja se grupišu na
  - faze zavisne od izvornog jezika (analiza) pripadaju prednjem modulu kompajlera ***(front end)*** i
  - faze zavisne od ciljnog jezika (sinteza) pripadaju zadnjem modulu kompajlera ***(back end)***

Međukod

- Praktična važnost podele kompajlera na prednji i zadnji modul
  - prevođenje sa jednog izvornog jezika na više ciljnih jezika
    - 1 prednji modul i više zadnjih modula
  - prevođenje sa više izvornih jezika na jedan ciljni jezik
    - više prednjih i 1 zadnji modul
- Podela kompajlera na prednji i zadnji modul je važna za olakšavanje prilagođavanja kompajlera novim zahtevima i novim okolnostima
- Komunikacija prednjeg i zadnjeg modula kompajlera podrazumeva uvođenje **međukoda** *(intermediate code)*
  - *međukod* predstavlja
    - (1) ciljni jezik za prednji modul kompajlera, a
    - (2) izvorni jezik za zadnji modul kompajlera
- Međukod je jezik hipotetskog računara (apstraktne mašine)

Međukod

- Uvođenje međukoda dovodi do pojave faze generisanja međukoda i podele faze optimizacije na dve zasebne faze
- Za međukod je važno da olakša implementaciju faza generisanja međukoda i koda, kao i faza optimizacije
- Pregled faza kompajliranja:



Međukod

- Kompajler može sadržati samo prednji modul
  - tada je njegov ciljni jezik međukod,
  - koga izvršava (interpretira) poseban program – **interpreter**

Upotreba stabla parsiranja

- Funkcionisanje prednjeg modula kompajlera je zasnovano na stablu parsiranja
- Stablo parsiranja sadrži reprezentaciju kompletnog i ispravnog (validnog) programa u izvornom jeziku
- Stablo parsiranja omogućuje:
  - prevođenje
  - interpretiranje
  - primenu procesa suprotnog parsiranju - *unparse*, i prikazivanje korisniku raznih osobina programa (u obliku običnog teksta, u obliku *XML*-a, u nekom grafičkom obliku, ...)
  - proveru da li su sve promenljive inicijalizovane pre upotrebe (samo neki jezici ovo definišu kao deo semantičkih pravila, ali mnogi ne)
  - ...

Vrste međukoda

- Međukod može imati oblik
  - **sintaksnog stabla**
  - **postfiksne** (poljske) **notacije** (primenjena kod *Pascal* kompajlera) ili **prefiksne notacije**
  - **troadresnog koda** koji odgovara hipotetskom asemblerskom jeziku (primenjen kod *C* kompajlera)

Vrste međukoda – sintaksno stablo

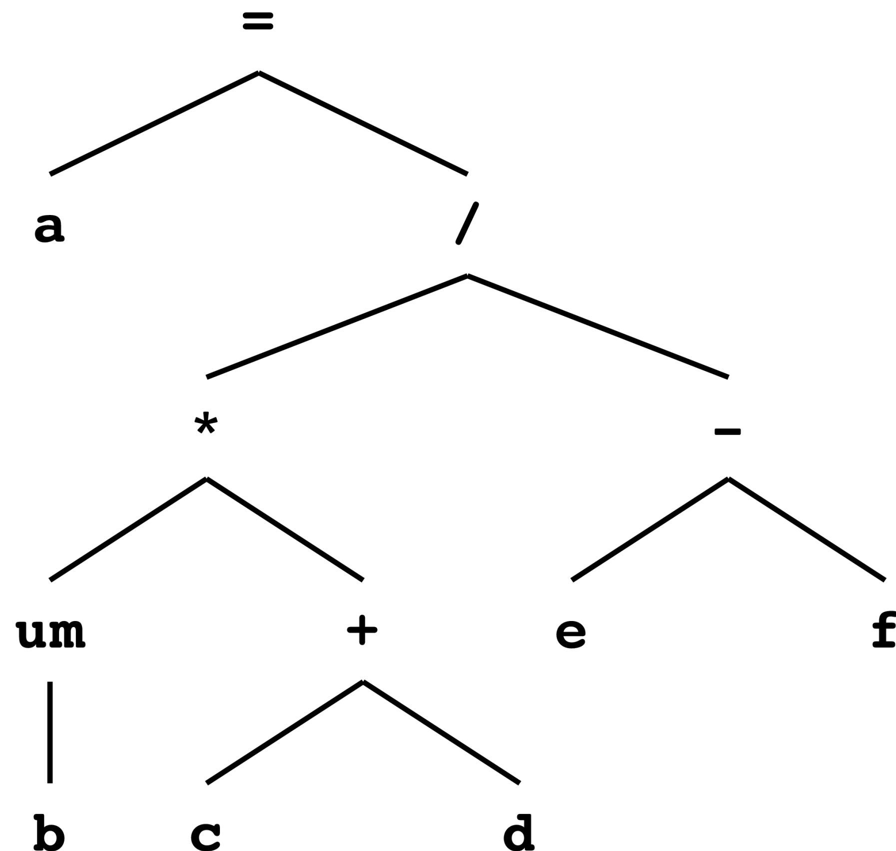
- Sintaksno stablo = Apstraktno sintaksno stablo
  - je prezentacija sintaksne strukture izvornog koda u obliku stabla
  - izražava prirodnu hijerarhijsku strukturu programa pokazujući redosled izvršavanja operacija programa
  - svaki čvor stabla označava pojam ili simbol koji se pojavljuje u izvornom kodu
  - sintaksa je apstraktna u smislu da ne odražava svaki detalj koji se pojavljuje u pravoj sintaksi i po tome se razlikuje od konkretnog stabla parsiranja
    - stablo parsiranja pokazuje postupak izvođenja programa iz gramatike
    - sintaksno stablo je kondenzovano stablo parsiranja
- Apstraktna sintaksna stabla se koriste i u sistemima za analizu i transformaciju programa

Vrste međukoda – sintaksno stablo

- Za iskaz:

  a = –b * (c + d) / (e – f)

  Sintaksno stablo izgleda:

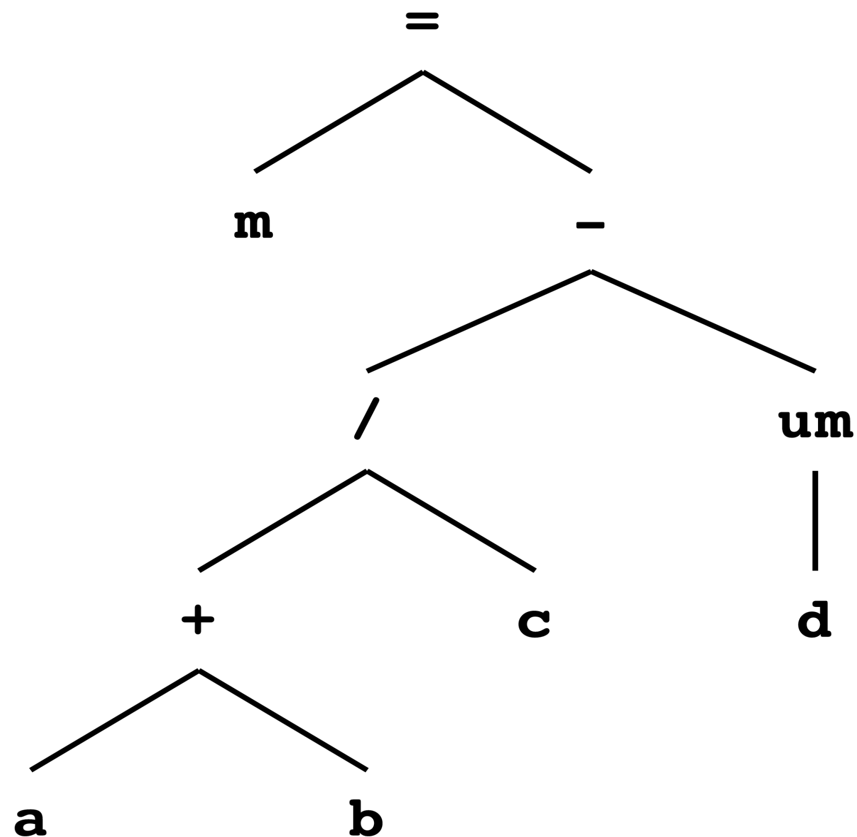


Vrste međukoda – sintaksno stablo

- Za iskaz:

```
m = (a + b) * c – –d
```

Sintaksno stablo izgleda:



Vrste međukoda – postfiksna notacija

- Postfiksna notacija ili Obrnuta Poljska notacija (*Reverse Polish notation - RPN*)
  - prilagođena je apstraktnoj stek mašini
  - uniformno tretira sve operatore
  - ne zahteva zagrade
  - operator sledi iza svih svojih operanada
    - za razliku od Poljske notacije, gde se prvo navodi operator
- Naziv “Poljska” se odnosi na nacionalnost logičara koji je izmislio Poljsku notaciju 1920-tih
- U računarstvu, postfiksna notacija se često koristi u programskim jezicima zasnovanim na steku

Vrste međukoda – postfiksna notacija

- za iskaz (koji koristi infiksnu notaciju):

```
a = -b * (c + d) / (e – f)
```

odgovarajuća postfiksna notacija izgleda

```
a b unarni_minus c d + * e f - / =
```

- iskazu u infiksnoj notaciji:

```
m = (a + b) * c - -d
```

odgovara iskaz u postfiksnoj notaciji:

```
m a b + c * d unarni_minus - =
```

Vrste međukoda – postfiksna notacija

- Pomenuti primer postfiksne notacije može biti izražen pomoću IJVM naredbi:

```
a = -b * (c + d) / (e - f);

BIPUSH   0
ILOAD    b
ISUB
ILOAD    c
ILOAD    d
IADD
IMUL
ILOAD    e
ILOAD    f
ISUB
IDIV
ISTORE   a
```

Vrste međukoda – troadresni kod

- Troadresni kod (*three-address code TAC* ili *3AC*)
  - ne mora imati simbolički oblik hipotetskog asemblerskog jezika,
  - nego može imati numerički oblik, gde pojedini brojevi predstavljaju kod naredbe i kodove operanada
- Ako kao kodovi operanada služe indeksi elemenata tabele simbola, tada se generatoru koda prepušta da zauzima memoriju za operande
- Numerički oblik troadresnog koda je zgodniji za zadnji modul kompajlera
- Neke varijante 2-, 3- ili 4-adresnog koda se često koriste kao međukod (*intermediate representation*) jer se dobro prevode (mapiraju) na većinu asemblerskih jezika

Vrste međukoda – troadresni kod

- Svaka instrukcija se može opisati kao grupa od 4 elementa:

  `(operator, operand1, operand2, result)`

- Svaka naredba ima uopštenu formu:

  `result := operand1 operator operand2`

  kao na primer:

  `x := y op z`

  - gde su **`x`**, **`y`** i **`z`**
    - promenljive,
    - konstante ili
    - privremene promenljive izgenerisane od strane kompajlera, a
  - **`op`** je bilo koji operator (npr. operator dodele)

Vrste međukoda – troadresni kod

- Izrazi koji sadrže više od jedne operacije, kao što je:

```
p := x + y * z
```

- nisu pogodni za predstavljenje kao jedna instrukcija troadresnog koda. One se dekomponuju u ekvivalentnu sekvencu instrukcija:

```
t_1 := y * z
p := x + t_1
```

  - rezultat svake operacije dobija ime
- Osnovna osobina troadresnog koda je to da svaka instrukcija implementira samo jednu operaciju, i da *source* i *destination* mogu da se odnose na bilo koji slobodan registar

Vrste međukoda – troadresni kod

```
a = b * c + b * d;
```

```
_t1 = b * c;
_t2= b * d;
_t3 = _t1 + _t2;
a = _t3;
```

```
m = (a + b) * c - -d
```

```
_t1 = a + b;
_t2 = _t1 * c;
_t3 = 0 – d;
_t4 = _t2 – t3;
m = _t4;
```

- Sa leve strane je aritmetički izraz, a sa desne strane je ekvivalentni TAC
- Privremene promenljive **_t1**, **_t2,** **_t3** i **_t4** je napravio kompajler da bi broj adresa sveo do 3 (1 instrukcija = 1 operacija)

Vrste međukoda – troadresni kod

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

```
      _t1 = b + c;
      _t2 = a < _t1;
      IfZ _t2 Goto _L0;
      _t3 = a - c;
      a = _t3;
_L0:  _t4 = b * c;
      c = _t4;
```

- Naravno, malo je komplikovanije prevesti grananje, petlje i pozive funkcija
- Prikazan je primer prevođenja **if** naredbe

```
IfZ _t2 Goto _L0;
```

znači : idi na labelu **_L0** ako je vrednost **_t2** nula

Generisanje međukoda

- Generisanje međukoda je vrlo slično generisanju koda
  - IL može koristiti neograničen broj registara
    - olakšano generisanje koda
  - u asemblerskom jeziku postoji ograničen broj registara
- Ranije navedeni primeri generisanja koda su zapravo primeri generisanja međukoda
- Međujezik koji koristi MICKO je *high-level* asemblerski jezik: hipotetski asemblerski jezik, što znači troadresni kod

Optimizacija međukoda

- Pre optimizacije međukoda
  - koju vrši kompajler
- moguća je optimizacija izvornog koda
  - koja je u nadležnosti programera

Optimizacija izvornog koda

- Za ovu optimizaciju neophodno je analizirati izvorni kod
- Analiza programa može biti
  - statička (analiza izvornog koda) i
  - dinamička (analiza izvršavanja programa)
    - razmatraju se performanse programa, u smislu
      - brzine izvršavanja i
      - zauzeća memorije
    - zasniva se na podacima koji su prikupljeni tokom izvršavanja programa
  - analizom performansi se utvrđuju mesta u programu na čije izvršavanje odlazi najveći deo vremena izvršavanja programa
  - cilj: odrediti delove programa koje treba optimizovati
    - bilo po pitanju brzine, bilo po pitanju zauzeća memorije

Profajler

- **Profajler** (*profiler*) je alat koji obavlja analizu performansi merenjem broja poziva funkcija i trajanja svakog poziva
- Izlaz profajlera se sastoji od niza zabeleženih događaja (*trace*), odnosno statistički obrađenih rezltata analize (*profile*)
- Da bi prikupili podatke, profajleri koriste različite tehnike:
  - korišćenje prekida
  - ubacivanje koda u neke sistemske pozive (*hook*)
  - ubacivanje koda za merenje u program koji se analizira (modifikacija izvršnog oblika analiziranog programa)
- Primer profajlera je *GNU profiler* (*gprof*)

gprof

- Pre korišćenja *gprof*-a, program se mora prevesti sa podrškom za analizu

```
gcc -g -pg -o program program.c
```

- Nakon prevođenja, program se izvršava

```
./program
```

  Tokom njegovog izvršavanja se generiše datoteka **`gmon.out`** u kojoj se nalaze informacije prikupljene tokom rada programa

- Kada su podaci o izvršavanju programa prikupljeni, poziva se *gprof* koji vrši statističku analizu zabeleženih podataka i na ekranu ispisuje rezultate analize

```
gprof program
```

  (podrazumeva se da se u tekućem direktorijumu nalazi datoteka **`gmon.out`**)

izlaz *gprof-a*

- *gprof* može prikazati više različitih analiza, od kojih su najznačajnije:
  - ukupno vreme izvršavanja svake funkcije (*flat profile*)
  - vreme izvršavanja funkcija zajedno sa svim funkcijama koje je ona pozivala (*call graph*)
  - određivanje broja izvršavanja svake linije koda ponaosob (*annotaded source*) (ovo se postiže dodavanjem opcije **`-A`** prilikom pozivanja *gprof*-a)
- Ukoliko se *gprof* pozove bez dodatnih opcija, biće prikazani *flat profile* i *call graph*

flat profile

- Primer koda za analizu:

```
#include <stdio.h>

void function1() {
    int i, j;
    for(i=0; i < 10000; i++)
        j += i;
}
void function2() {
    int i, j;
    function1();
    for(i=0; i < 20000; i++)
        j = i;
}
```

```
int main() {
    int i, j;
    for (i = 0; i < 100; i++)
        function1();
    for(i = 0; i < 5000; i++)
        function2();
    return 0;
  }
```

flat profile

- Primer *flat profile* izlaza:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %    cumulative   self              self     total
 time    seconds   seconds    calls  ms/call  ms/call   name
 62.70      3.82      3.82     5000     0.76     1.21  function2
 37.79      6.12      2.30     5100     0.45     0.45  function1
```

- Funkcije su sortirane prvo po vremenu provedenom u njima, zatim po broju poziva i na kraju po svom imenu

flat profile

- Značenje kolona je sledeće:
  - **`% time`** – procenat ukupnog vremena izvršavanja koji je proveden u funkciji
  - **`cumulative seconds`** – broj sekundi provedenih u tekućoj funkciji i svim funkcijama u tabeli pre nje
  - **`self seconds`** - broj sekundi provedenih u funkciji
  - **`calls`** – broj poziva funkcije
  - **`self ms/calls`** – prosečno vreme trajanja svakog poziva, gledajući samo vreme provedeno u funkciji
  - **`total ms/calls`** – prosečno vreme trajanja svakog poziva, gledajući vreme provedeno u funkciji i funkcijama koje je ona pozivala
  - **`name`** – ime funkcije

Optimizacija međukoda

- U modernim kompajlerima, ovo je najkompleksnija faza
- Optimizacija međukoda je mašinski nezavisna optimizacija
- Podrazumeva transformaciju međukoda koja
  - **ne menja značenje koda** (programa)
  - ali merljivo **poboljšava korišćenje resursa**
    - ubrzava njegovo izvršavanje ili
    - smanjuje memorijske zahteve ili
    - smanjuje veličinu koda ili
    - smanjuje pristup disku (baze podataka) ili
    - smanjuje potrošnju energije / baterije
- Kompajleri koji obavljaju ovakve transformacije se nazivaju optimizujući kompajleri *(optimizing compilers)*

Optimizacija međukoda

- Optimizacija međukoda se zasniva na
  - **analizi upravljačkog toka** *(control-flow analisys*)
  - otkrivanju **baznih blokova** *(basic blocks)*
- Bazni blok (BB) je sekvenca instrukcija sa jednom tačkom ulaza i jednom tačkom izlaza
  - maksimalan broj instrukcija (najduža sekvenca instrukcija)
  - instrukcije se uvek izvršavaju od prve do poslednje
  - ne sadrži labele (osim u prvoj instrukciji)
  - ne sadrži skokove (osim u poslednjoj instrukciji)
- Ideja je da:
  - nema ulaska (*jump in*) u BB (osim na početku BB)
  - nema izlaska (*jump out*) iz BB (osim na kraju BB)
  - tok izvršavanje teče od prve do poslednje instrukcije bez zaustavljanja

Optimizacija međukoda

- **dijagrama toka** *(control-flow graph)* je
  - usmereni graf
  - čvorovi su bazni blokovi
  - spojnice su usmerene
    - pokazuju **redosled izvršavanja** *(flow of control)*
    - postoje od BBA ka BBB ako se izvršavanje nastavlja od poslednje instrukcije iz A ka prvoj instrukciji u B
  - pokazuje tok izvršavanja između baznih blokova
    - unutar BB tok izvršavanja ide od prve do poslednje instrukcije
  - opisuje jednu funkciju

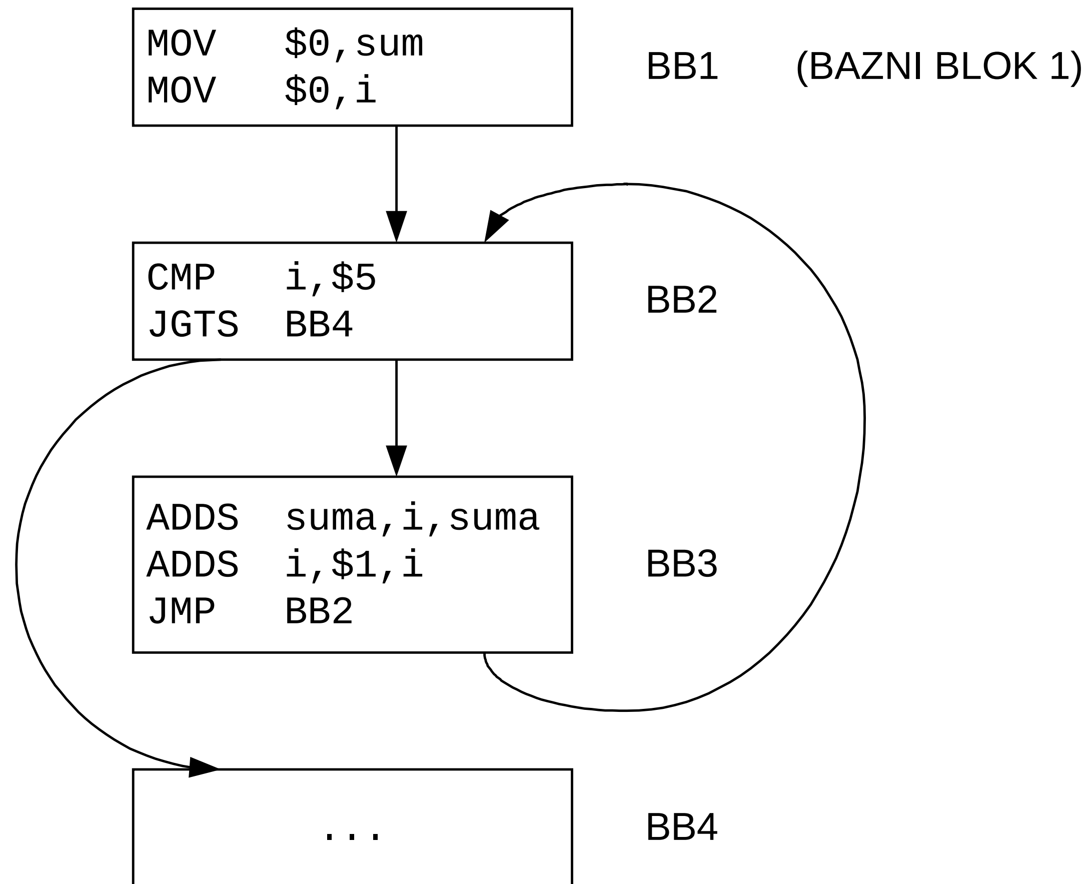
Optimizacija međukoda

- Na primer, međukodu:

```
        MOV     $0,suma
        MOV     $0,i
for0:
        CMP     i,$5
        JGTS    exit0
        ADDS    suma,i,suma
        ADDS    i,$1,i
        JMP     for0
exit0:
```

Optimizacija međukoda

odgovara dijagram toka:



Optimizacija međukoda - primer

- Koliko baznih blokova ima (odgovor: 5)
- Na kojim linijama počinje i završava svaki od baznih blokova?

```
@if2:                               1  ┐
          CMPS    c,d               2  │ BB1
          JGTS    @true2            3  ┘
@false2:                            4  ┐
          CMPS    a,c               5  │ BB2
          JLES    @false3           6  ┘
@true2:                             7  ┐
          MOV     $1,a              8  │ BB3
          JMP     @exit2            9  ┘
@false3:                            10 ┐
          MOV     $2,a              11 ┘ BB4
@exit2:                             12 } BB5
```

Optimizacija međukoda

- Granularizacija optimizacije
  1. **lokalne optimizacije**
     - primenjuju se u jednom baznom bloku
     - većina kompajlera ih radi
  2. **globalne optimizacije**
     - primenjuju se u jednom dijagramu toka (telu funkcije)
     - puno kompajlera ih radi
  3. **među-proceduralne optimizacije**
     - primenjuju se između procedura
     - nekoliko kompajlera ih radi
  - najviše se isplati raditi lokalnu optimizaciju
  - cilj: maksimum dobiti za minimalnu cenu
     - kompleksnost implementacije
     - vreme kompajliranja
     - koliko se isplati

Optimizacija međukoda - lokalna

- **Lokalna optimizacija**
  - najjednostavniji oblik optimizacije
  - optimizuje jedan BB
- **Algebarske transformacije**
  - neke naredbe mogu biti uklonjene
    - `x = x + 0`
    - `x = x * 1`
  - neke naredbe mogu biti pojednostavljene
    - `x = x * 0` → `x = 0`
      - traje jednako dugo, ali pospešuje sledeće optimizacije
    - `y = y ** 2` → `y = y * y`
      - stepenovanje nije *built-in* instrukcija već poziv funkcije
    - `x = x * 8` → `x = x << 3`
      - množenje stepenom 2-ke se menja bit-šiftovanjem
      - na nekim mašinama `<<` je brže od `*`, ali ne na svim

Optimizacija međukoda - lokalna

- **slaganje konstanti** (*constant folding*): neke operacije nad konstantama mogu biti izračunate u vreme kompajliranja
  - `x = y op z`
  - ako su `y` i `z` konstante, onda `y op z` može biti izračunato u vreme kompajliranja
  - `x = 2 + 2  →   x = 4`
  - `if 2 < 0 jump L`
  - može biti obrisana jer je uslov `2 < 0` uvek `false`
  - slaganje konstanti je jedan od najčešćih i najbitnijih optimizacija koje kompajler sprovodi

Optimizacija međukoda - lokalna

- nedostupne naredbe
  - ne postoji skok na tu naredbu
  - ne može se „propasti“ na izvršavanje te naredbe
  - nedostupan kod se može obrisati
    - time se smanjuje kod programa

Optimizacija međukoda - lokalna

- kako i zašto se pojavljuju nedostupne naredbe?
  - *then* telo postaje nedostupno kada nema dibagiranja

```
#define DEBUG 0
if(DEBUG) then  →   if(0) then
   . . .                . . .      * nedostupno
```

  - upotreba biblioteke
    - biblioteke sadrži stotine funkcija, a upotrebi se njen mali deo, recimo 3 funkcije
    - ostatak funkcija se može ukloniti iz *binary* datoteke
  - rezultat drugih optimizacija

Optimizacija međukoda - lokalna

- Suvišni kod je onaj koji ima ponovljen efekat i nepotreban je
  1) **zajednički podizraz** (*common subexpression*) koji je prethodno izračunat i još uvek je dostupan
     a)
     ```
     y = x * 2;
     return x * 2;
     ```
     - drugi izraz `x*2` je suvišan i može se zameniti referencom na promenljivu `y`
     - ako se `y` nigde dalje ne koristi, dodela `y = x * 2;` može biti uklonjena

     b)
     ```
     x = y + z                    x = y + z
     …               →
     m = y + z                    m = x
     ```
     - uslov: da se vrednosti `x`, `y` i `z` ne menjaju nigde u … kodu
     - često se primenjuje u kompajlerima

Optimizacija međukoda - lokalna

- u nastavku koda, svaka upotreba **`m`** se može zameniti sa **`x`**
- (i svako **`a`** se može zameniti sa **`b`**)

```
x = y + z              x = y + z
a = b          =>      a = b
m = y + z              m = x
n = 2 * a              n = 2 * b
```

- ovaj postupak se zove propagacija kopije (*copy propagation*
  - ukoliko je u pitanja u konstanta, onda se zove propagacija konstante *(constant propagation)*
- sam postupak nije poboljšao kod, ali
- daje rezultate u spoju sa drugim optimizacijama
- na primer: ako se **`a`** ne koristi dalje u programu, onda se naredba **`a = b`** može ukloniti

Optimizacija međukoda - lokalna

- primer:

```
a = 5                       a = 5
x = 2 * a         =>        x = 10
y = x + 6                   y = 16
m = x * y                   m = 160
```

- propagacija konstante: **a** se svugde zamenjuje konstanom **5**
- u drugoj naredbi je sada moguće primeniti constant folding (**2 * 5**)
- sada smo dobili novu konstantu (**10**) kao vrednost za **x**, pa dalje možemo propagirati tu konstantu:

y = 10 + 6 => slaganje konstanti **=> y = 16**

m = 10 * y => propagacija kopije **=> m = 10 * 16**

=> m = 160

Optimizacija međukoda - lokalna

2) nedostupan kod (*unreachable code*)

- kod koji se nikada neće izvršiti, jer tok izvršavanja nikad neće stići do njega

```
return x + y;
int z = x * y;
```

- do definicije **z** promenljive se nikada ne stigne, jer se pre toga izvrši **return**
- definicija **z** promenljive se može izbaciti

Optimizacija međukoda - lokalna

3) mrtav kod (*dead code*)

= kod koji se izvršava ali nema efekta na ostatak koda

```
z = x / y
```

- izraz koji se dodeljuje **`z`** se nikada neće iskoristiti ako se **`z`** ne koristi nigde dalje u programu
- samo deljenje može dovesti do pojave izuzetka
- optimizacija ovakve naredbe izbacuje
- primer: naredba **`a = b`** može da se izbaci ako se **`a`** ne koristi nigde dalje u kodu

```
b = z + y              b = z + y
a = b          =>
x = 2 * b              x = 2 * b
```

Optimizacija međukoda - lokalna

- Svaka optimizacija, sama za sebe, čini malo poboljšanje koda
- Optimizacije se najčešće dopunjuju
  - primena jedne optimizacije omogući drugu
- Optimizujući kompajleri ponavljaju optimizacije dok ne dođu do stanja kada poboljšanje više nije moguće
  - optimizacija može biti i zaustavljena u nekom trenutku vremena, da bi se ograničilo vreme kompajliranja

Optimizacija međukoda – lokalna

Početni kod:

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

Algebarske transformacije:

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

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

Optimizacija međukoda – lokalna

Propagacija kopije:

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

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

Optimizacija međukoda – lokalna

Slaganje konstanti:

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

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

Optimizacija međukoda – lokalna

Uklanjanje zajedničkih podizraza:

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

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

Optimizacija međukoda – lokalna

Propagacija kopije:

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

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

Optimizacija međukoda – lokalna

Uklanjanje mrtvog koda:

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

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

```
f = 2 * a
g = 12 * a
```

Optimizacija međukoda - lokalna

- Koje su validne lokalne optimizacije za dati BB? Pretpostavka je da su samo **`g`** i **`x`** referencirane izvan ovog BB.

a) propagacija kopije: Linija 4 postaje **`d := a * b`**.

b) uklanjanje zajedničkih podizraza: Linija 5 postaje **`e := d`**.

c) uklanjanje mrtvog koda: Linija 3 je uklonjena.

d) Posle nekoliko iteracija optimizacije, ceo blok koda se može redukovati na **`g := 5`**.

```
1   a := 1
2   b := 3
3   c := a + x
4   d := a * 3
5   e := b * 3
6   f := a + b
7   g := e - f
```

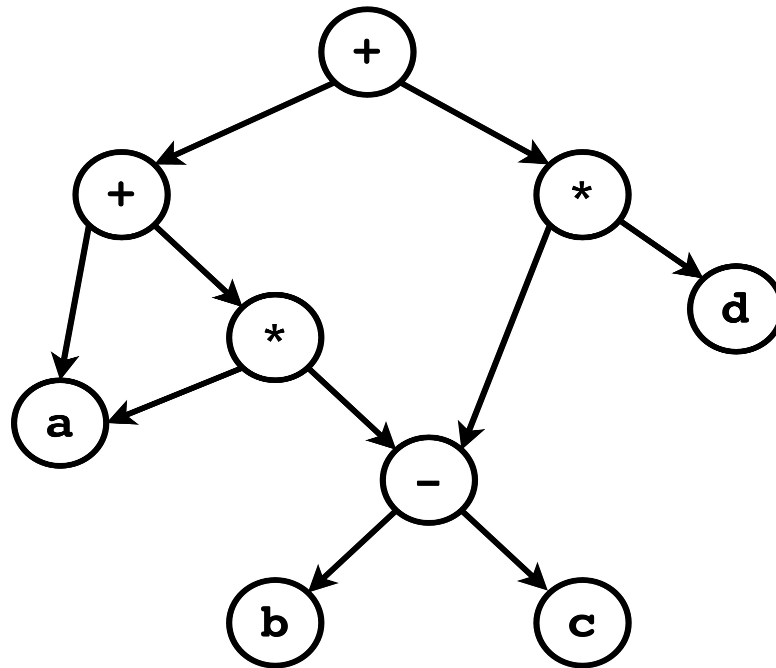
Optimizacija međukoda

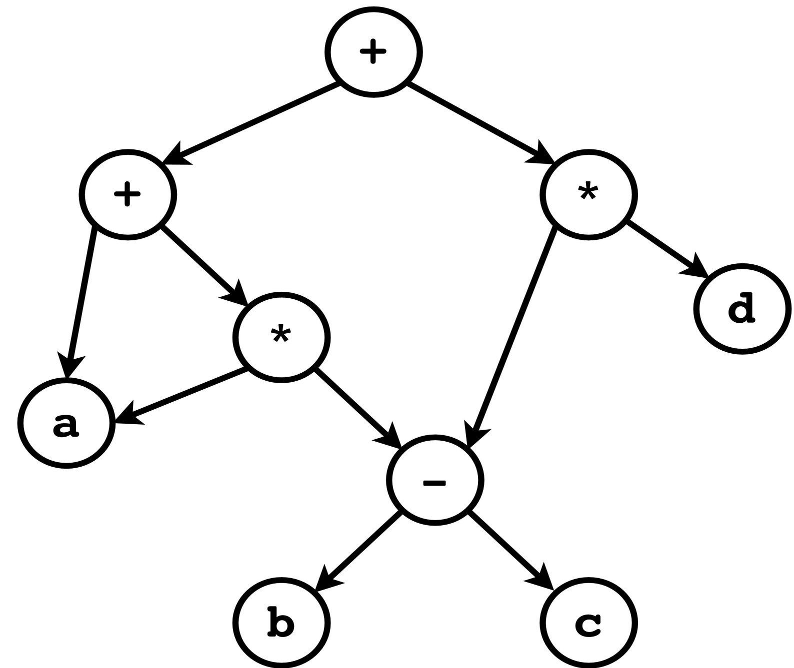
- Za otkrivanje zajedničkih podizraza korisno je crtati **usmerene aciklične grafove** baznih blokova
- *directed acyclic graph – DAG*
  - usmereni graf koji ne sadrži petlje
  - koristi se za prikazivanje redosleda događaja
- DAG se koristi za lako otkrivanje zajedničkih podizraza
- DAG ima po 1 čvor za svaki podizraz
  - unutrašnji čvor predstavlja operator a njegovi čvorovi potomci su operandi
  - čvor može imati više roditelja

Optimizacija međukoda

- Slika pokazuje stablo i odgovarajući DAG za izraz:

a + a * (b – c) + (b – c) * d





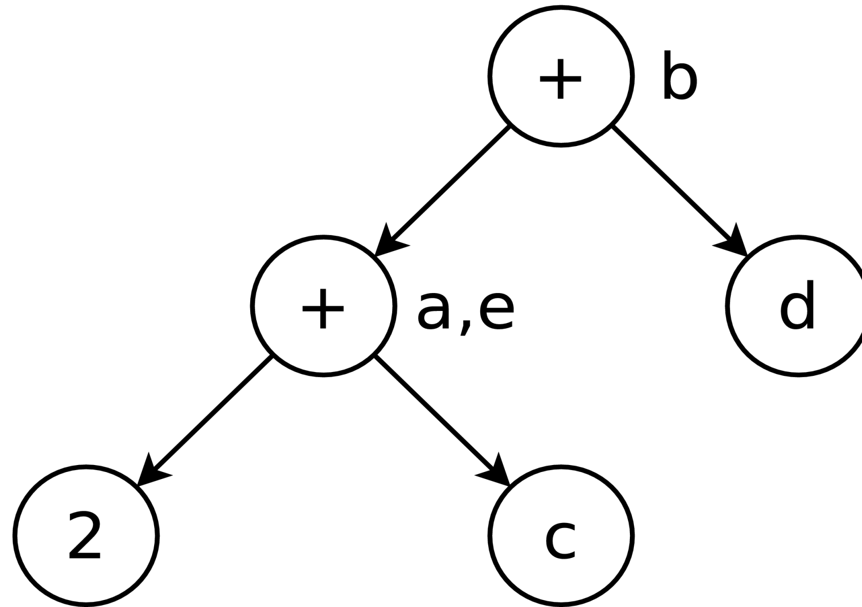
- List **a** ima 2 roditelja jer je **a** zajednički podizraz za 2 podizraza: za prvo sadbiranje i prvo množenje
- Obe pojave zajedničkog podizraza **b – c** su predstavljene istim čvorom, koji ima 2 roditelja

Optimizacija međukoda

- Za bazni blok

```
ADDS   $2,c,a
ADDS   a,d,b
ADDS   $2,c,e
```

usmereni aciklični graf izgleda:



- Iz grafa je očigledno da se promenljivim **a** i **e** pridružuje vrednost zajedničkog podizraza, pa se bazni blok može transformisati u:

```
ADDS  $2,c,a
ADDS  a,d,b
MOV   a,e
```

Optimizacija međukoda

- Primer suvišne naredbe je naredba koja pridružuje vrednost promenljivoj u baznom bloku od koga se ona više ne koristi
- Ako se promenljiva **e** ne koristi iza baznog bloka iz prethodnog primera, taj bazni blok se može transformisati u

```
ADDS  $2,c,a
ADDS  a,d,b
```

- Suvišne naredbe se obično javljaju nakon pojedinih transformacija baznih blokova u toku optimizacije

Optimizacija međukoda

- Optimizacija petlji ima za cilj da smanji vreme izvršavanja petlje
  - smanjenjiti broj naredbi u telu petlje
    - naredbe koje ne zavise od petlje idu ispred tela petlje
  - npr: ako se u petlji pristupa svim elementima niza **`int n[10]`**

```
a = n[i – 5];
```

    - tada petlja sadrži računanje adresa pojedinih elemenata:

```
adresa_n + (i - 5) * 4
```

    - podrazumeva se:
      - da je **`adresa_n`** adresa prvog elementa niza **`n`**
      - da **`adresa_n`** nije poznata u kompilaciji
      - da svaki element niza zauzima 4 bajta
    - Prethodni izraz se može transformisati u:

```
i * 4 + (adresa_n - 20)
```

    - a računanje izraza iz zagrade se može pomeriti ispred petlje, tako da se samo njegova vrednost koristi u petlji

Optimizacija međukoda – *peephole*

- Posebna vrsta optimizacije je **parcijalna** *(peephole)* **optimizacija**
  - posmatraju se kratke sekvence uzastopnih naredbi i zamenjuju kraćim i bržim sekvencama
  - sekvenca naredbi se posmatra kroz zamišljeni prorez *(peephole)*,
    - pri čemu se prorez pomera preko naredbi od početka ka kraju programa
  - parcijalna optimizacija je najdelotvornija kada se uzastopno višestruko ponavlja

Optimizacija međukoda – *peephole*

- u toku parcijalne optimizacije traže se pojave unapred zadanih slučajeva kao što su:
  - suvišne naredbe
  - suvišni skokovi
  - nedostupne naredbe
  - mrtve naredbe
- uoče se tipični slučajevi neefikasnog međukoda
- za svaki od slučajeva se sprovodi parcijalna optimizacija da bi se pojave dotičnog slučaja pronašle i uklonile

Optimizacija međukoda

- Primeri **suvišnih naredbi** (* označava suvišnu naredbu):
  - naredba poređenja koja sledi iza identične naredbe, a da između njih nije bilo izmena uslovnih *(condition code)* bita iz status registra

```
CMPU   a,b
JEQ    @false0
CMPU   a,b          *
JLEU   @false1
```

  - naredba punjenja *(load)* registra koja sledi iza identične naredbe, a da između njih nije bilo izmena sadržaja registra i obrnuto *(store)*

```
MOV    $7, %1
ADDU   %0, %1, %2
MOV    $7, %1       *
```

Optimizacija međukoda

- Primeri **suvišnih naredbi**:
  - ponovno izračunavanje izraza

```
y = x * 2;
return x * 2;      *
```

  - uzastopne *load* i *store* naredbe koje se odnose na isti registar i istu memorijsku lokaciju

```
MOV  %0,a
MOV  a,%0          *
```

Optimizacija međukoda

- Primeri **nedostupnih naredbi**
  - naredbe koje slede iza naredbe bezuslovnog ili uslovnog skoka, a nisu cilj neke druge naredbe skoka

```
   JEQ    labela1
   JNE    labela2
   MOV    $12, %0    *
labela1:
   …
labela2:
   …
```

  - naredbe iza **`return`** iskaza u funkciji

```
   return x + y;
   int z = x * y;    *
```

Optimizacija međukoda

- Primeri mrtvog koda
  - naredbe koje se izvršavaju ali nemaju efekta na ostatak koda

```
a = x + y;
z = 2;                  *
return x * y;
```

Optimizacija međukoda - primer

```
        MULS    $2,%2,%2        alg. transformacija
        MOV     $5,n
lab1:
        ADDS    %2,n,%2
        MOV     $5,n            suvišna
        JGTS    lab2
        JLTS    lab3
        JEQ     lab4
        MOV     $1,q            mrtva
        SUBS    n,%2,%2         nedostupna
```

Optimizacija međukoda

- Suvišne i nedostupne naredbe se izbacuju
- Operandi naredbi suvišnih skokova se modifikuju
  - suvišan skok je naredba skoka čiji je cilj naredba bezuslovnog skoka

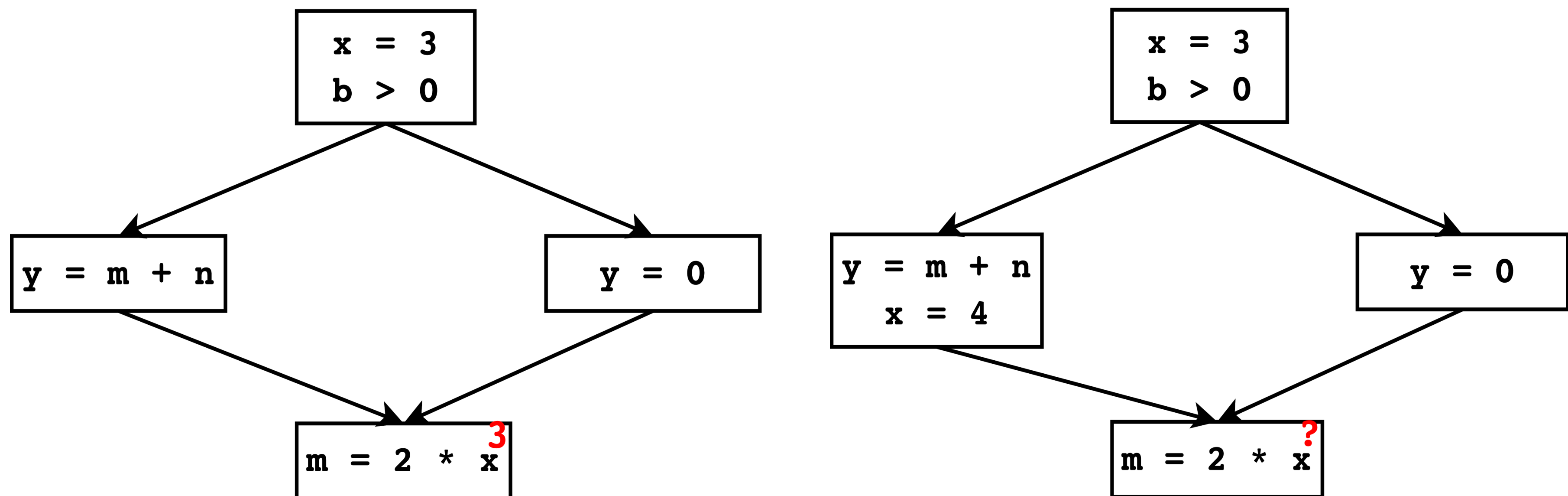
```
    JEQ    labela1
    ...
labela1:
    JMP    labela2
```

modifikuju se tako da se izbegavaju suvišni skokovi:

```
    JEQ    labela2
    ...
labela1:
    JMP    labela2
```

Optimizacija međukoda - globalna

- Odluka o transformaciji baznih blokova se donosi na osnovu:
  - poznavanja načina korišćenja promenljivih u programu
  - u prvom primeru moguće je propagirati konstantu **`x = 3`**
  - u drugom primeru nije moguća propagacija konstante
- globalno propagiranje konstante **`k`** je moguće pod uslovom da je na svakoj putanji do upotrebe `x`, poslednja dodela **`x = k`**



Optimizacija međukoda - globalna

- Uslov korektnosti primene optimizacije nije lako proveriti
- „sve putanje“ uključuje i
  - putanje oko petlji
  - putanje kroz grananja
- Zbog provere uslova obavlja se **globalna analiza toka podataka** *(global dataflow analisys)*
  - analiza celog dijagrama toka
- Cilj ovakve analize:
  - da se za svaku promenljivu odredi poslednji bazni blok u kome se ona koristi (u kome se preuzima njena vrednost)
- Rezultati analize toka podataka se čuvaju u tabeli simbola