Optimalizácia kódu a programu

Ján Šturc

Upresnenie

- Nejde o optimalizáciu (nájdenie najefektívnejšieho z pomedzi ekvivalentných programov).
  - Aj samotná ekvivalencia je nerozhodnuteľný problém.
- Sémantiku zachovávajúce transformácie, čo od nich očakávame ?
  1. Skrátenie času vykonávania programu
  2. Zníženie pamäťových nárokov
  3. Zníženie energetickej náročnosti (vyžiareného tepla) uplatňuje sa pri návrhu VLSI algoritmov.
- Klasické optimalizácie
  - Počet inštrukcií
  - Cena inštrukcií
- Moderné optimalizácie
  - Poradie inštrukcií (paralelizmus a latencia inštrukcií)
  - Umiestnenie operandov (registre, cahe, pamäť, …)

Úrovne optimalizácie

- Na úrovni algoritmu (lepší efektívnejší) algoritmus
  - Nie je predmetom kompilátorov
- Vysoká (strojovo nezávislá) na úrovni zdrojového jazyka a medzijazyka
- Nízka (strojovo závislá) na úrovni inštrukcií a architektúry počítača
- Inžinierské pravidlo palca:

  Algoritmická úroveň : nižšie úrovne = 50 : 50.
- Matematicky je to asi nezmysel
  - Všetko sa dá robiť ľubovolne zle.

Rozsah optimalizácie

- Lokálna (základný blok)
  - Je to najjednoduchšie
  - Dá sa robiť (aj sme to robili) súčasne s generovaním kódu.
- Globálna (intraprocedurálna)
  - Optimalizujú sa jednotlivé procedúry programu
  - Viac ako blok, ale ešte zvládnuteľné
  - Dôležité lebo cykly väčsinou presahujú jeden základný blok
- Interprocedurálna (celý program, modul)
  - Je to „skoro to isté“ ako globálna len vo väčšom rozsahu.
  - Umožňuje niektoré nové optimalizácie (napr. inlining, špecializáciu volania, ak procedúra nie je rekurzívna, náhrada opakovaného volania predvýpočtom a výberom, …).
  - Vyňatie volania procedúry pred cyklus.
  - Veľa kompilátorov nerobí interprocedurálnu optimalizáciu.

Optimalizácia cyklov 1

- „Mudrosť praktikov“
  - Viac ako 90% času strávi program v 4% kódu — v najvnútornejších cykloch.
- Možno pochybné, ale v každom prípade cykly stojí za to optimalizovať
- Zarážka (sentinel) – vysoko úrovňová optimalizácia

  while (i ≤ n **and** a[i]) **do** {S; i:= i + 1} nahraď
  a[n+1]:= **false**; **while** a[i] **do** {S; i:= i + 1}
- Reverzia cyklu

  for i:= 0 **to** n **do** S; nahraď **for** i:= n **downto** 0 **do** S;
  keď inštrukčný kód „vie“ testovať len na nulu.
- Rozvinutie cyklu

  for i:= 1 **to** 3 **do** S(i); nahraď S(1); S(2); S(3);
  Je to výhodné hlavne pre krátke cykly.

Optimalizácia cyklov 2

- Vyňatie „invariantu“ pred cyklus

  for (…) {S_1;Inv;S_2;} operandy (argumenty) výpočtu Inv sa v cykle nemenia. Nahraď Inv; **for** (…) {S_1;S_2;}

- Postponovanie príkazu do vetvy cyklu.
  - Ak výsledky nejakého zložitého príkazu S, majú následné použitie len v málo pravdepodobnej vetve cyklu, je vhodné „zašantročiť“ príkaz S do tejto vetvy.
  - Používa sa zriedka, lebo obvykle nevieme pravdepodobnosti vetiev cyklu.
  - Keď postponovanie nie je do cyklu, sa vždy vyplatí.
- Redukcia v sile operácii

  for i:= 0 **to** n **do** S(c×i); nahraď

  ub:= c×n; **for** t:= 0 **to** ub **by** c **do** S(t);

Optimalizácia cyklov 3

- Všeobecné optimalizácie
  - eliminácia spoločných podvýrazov
  - eliminácia mŕtveho kódu
  - skladanie konštánt

  získavajú v cykle na význame.
- Postponovanie výpočtu, ktorý nemá v cykle následné použitie za cyklus
  - Príklad:

```
for i:= 1 to n do
for j:= 1 to n do
    {A[i,j]:= 0; for k:= 1 to n do A[i,j]:= A[i,j] + B[i,k] × C[k,j] }
```

  - Nahraď

```
for i:= 1 to n do
for j:= 1 to n do { s:= 0;
                    for k:= 1 to n do s:= s + B[i,k] × C[k,j];
                    A[i,j]:= s                                  }
```

Pri dobrej optimalizácii cyklov asi nemá význam.

Optimalizácia cyklov 4

- Čiastočné rozvinutie (loop unroling)
  - Príklad (predpokladáme, že S(i) nemení i

```
i:= 1;
while i ≤ 100 do { S(i); i++;}
```

Nahradíme programom:

```
i:= 1;
while i ≤ 100 do { S(i); i++; S(i); i++;}
```

    - Variacie: S(0) pred cyklus, S(101) za cyklus, alebo štruktúra **loop break.**
- Spájanie cyklov (loop jamming)
  - Príklad

```
for i:= 1 to n do
for j:= 1 to n do A[i,j]:= 0;
for i:= 1 to n do A[i,i]:= 1;
```

nahradíme:

```
for i:= 1 to n do
    { for j:= 1 to n do A[i,j]:= 0;
      A[i,i]:= 1;                    }
```

Komplikuje použitie iných transformácií.

Redukcia vnorenia cyklov

Príklad

```
for i:= 1 to n do
for j:= 1 to n do A[i,j]:= 0;
```

nahradíme:

```
p:= &A[1,1]; ub:= p + n*n -1;
while p ≤ ub do {*p:= 0; p++;}
```

Optimalizácia cyklov 5

- Optimalizácie súvisiace s modernou architektúrou počítača (Nie sú v starších vydaniach dračej knihy. Len „fialový drak“.)
  - Aby operandy boli v dátovej cache.
  - Aby inštrukcie cyklu boli v inštrukčnej cache
  - Aby pri prúdovom spracovaní nedochádzalo k prestojom (stalls)
- Bude niečo v poslednej prednáške.
- Treba hľadať v literatúre
  - ACM Transactions on Programming Languages (TOPLAS)
  - Principles of Programming Languages
  - Software Practice and Experience.
- Firemná literatúra
  - Výrobcovia procesorov
  - Veľké softwareové firmy

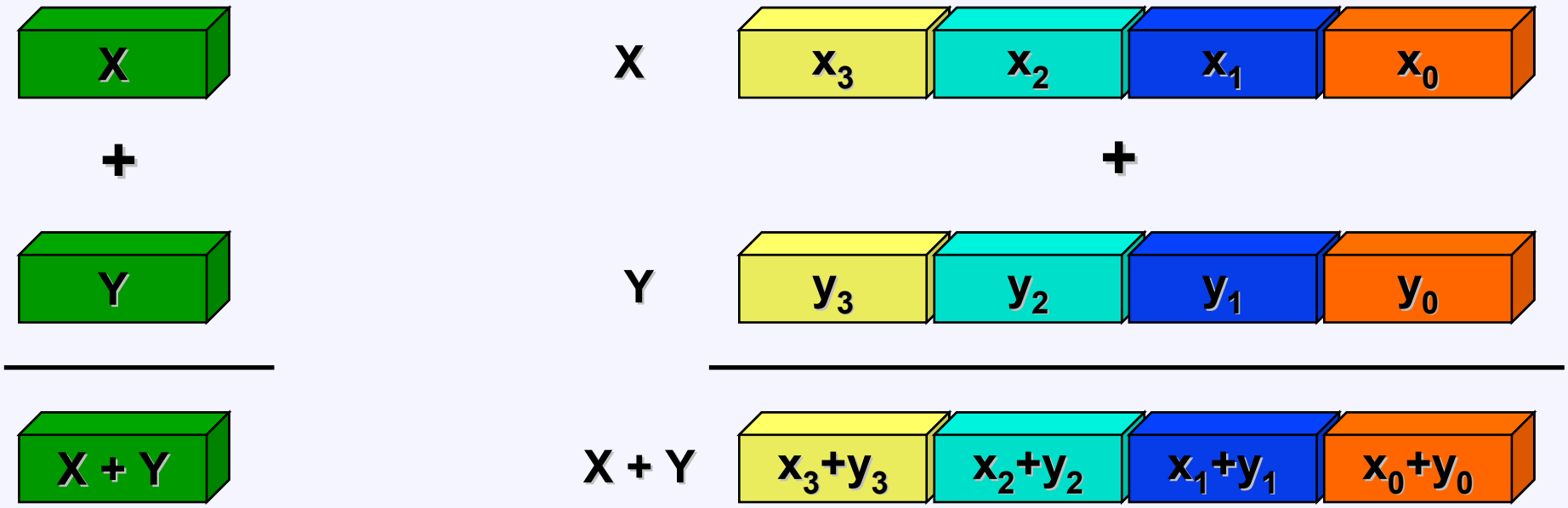
Príklady architektonických optimalizácii

- Preusporiadanie cyklu tak, aby sa robilo najmenej prenosov medzi cache a operačnou pamäťou
  - operácie s veľkými maticami
  - triedenie
- Úprava cyklu, aby sa dal využiť SIMD modul (MMX, SSE)
  - pozor na presnosť

SIMD – single instruction multiple data

zdroj Intel Corporation

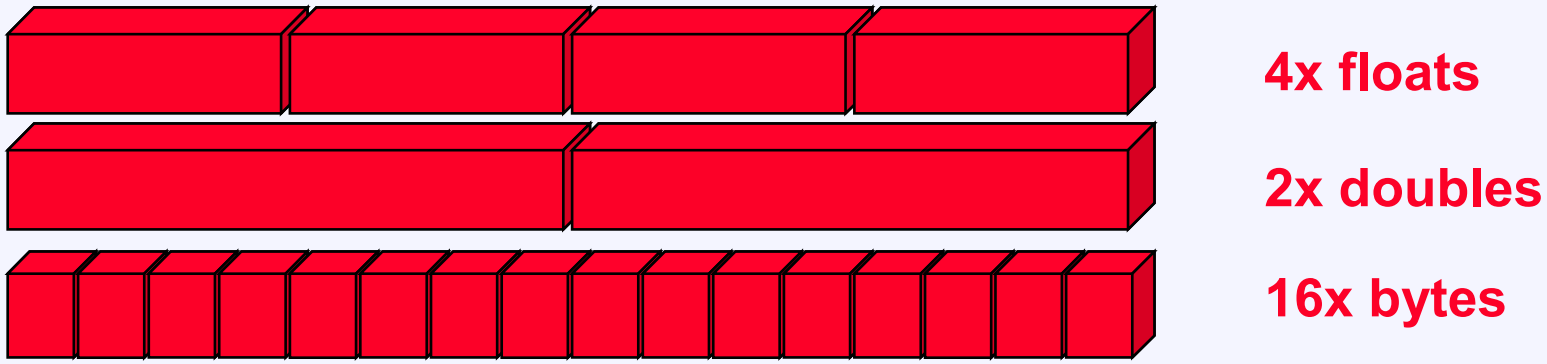
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - one operation produces multiple results



Všeobecnejšie riešenie je aritmetika s deleným prenosom (Grečný, Šturc 1967, Projekt RPP16, ÚTK SAV)

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another

Motivácie pre delený prenos, SSE a MMS

- Práca s textom
  - EBDIC a staré kódy – 6 bitov
  - ASCII – 8 bitov
  - GIER – 10 bit
  - Brinch Hansen (RC 4000) – 12 bit
- Multimédia
  - Vzorkovanie zvuku 12 alebo 16 bit je často dostatočná kvalita
  - Farebná hĺbka, 16, 24 alebo 32 bit
- Vzorkovanie technologických procesov
  - RC 4000 a RPP16 boli technologické (riadiace) počítače

Poznámka o dĺžke slova: dnes štandardne násobky 8.
Vtedy niektorí preferovali násobky 12.
Všeobecne 2^k3^l.

Dominancia v grafe a jej vlastnosti.

Definícia: Daný je graf G a počiatočný uzol n_0 (koreň). Hovoríme, že uzol d dominuje uzol n. Ak každá cesta v grafe G od počiatku n_0 do uzla n vedie cez uzol d.

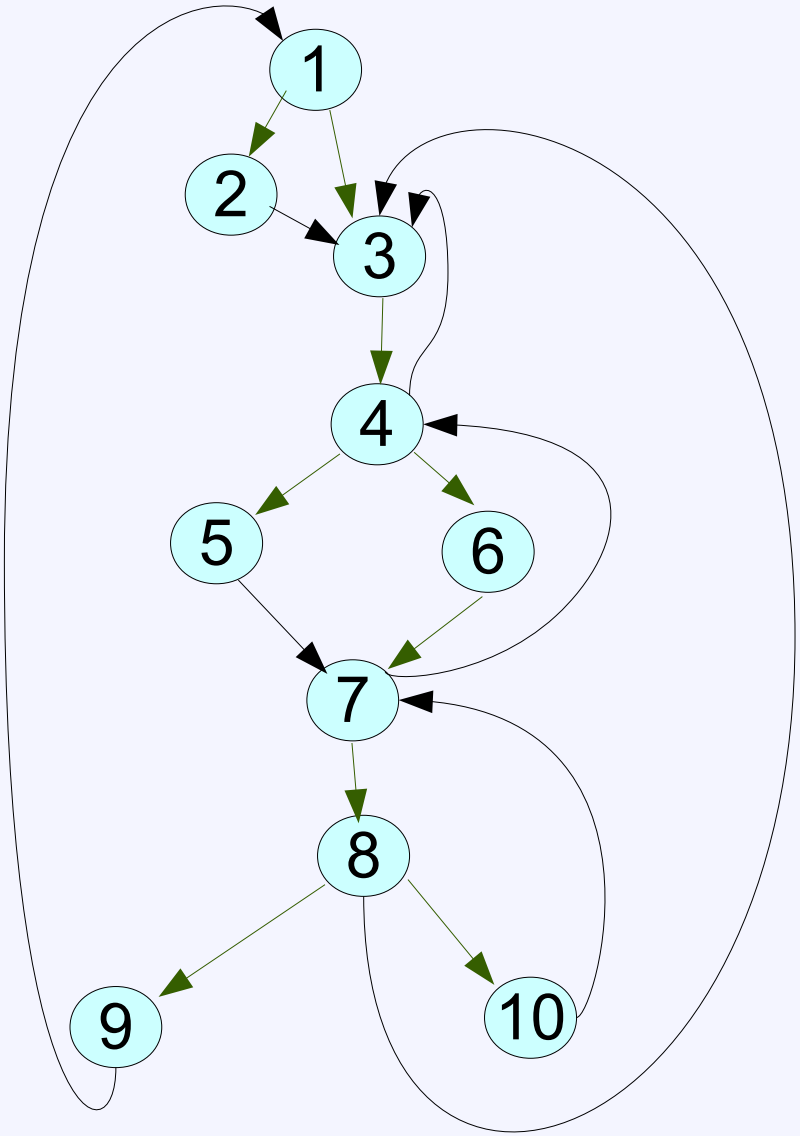
- Relácia dominancie $\succeq$ je reflexívne čiastočné usporiadanie na grafe G:
  1. Pre každý uzol n, $n \succeq n$.
  2. $p \succeq q$ a $q \succeq p$ implikuje $p = q$.
  3. $p \succeq q$ a $q \succeq r$ implikuje $p \succeq r$.
- Relácia dominancie rozdeluje uzly grafu G do dvoch tried
  - Dosiahnuteľné $n_0 \succeq n$.
  - Nedosiahnuteľné. Z hľadiska programu bezvýznamné. Môžeme ich vynechať.
- Ak G je súvislý graf, vynechaním uzla d a hrán s nim incidentných sa graf rozpadne na dve nesúvisle komponenty $\{n: d \succeq n \wedge d \neq n\}$ a G - $\{n: d \succeq n\}$.

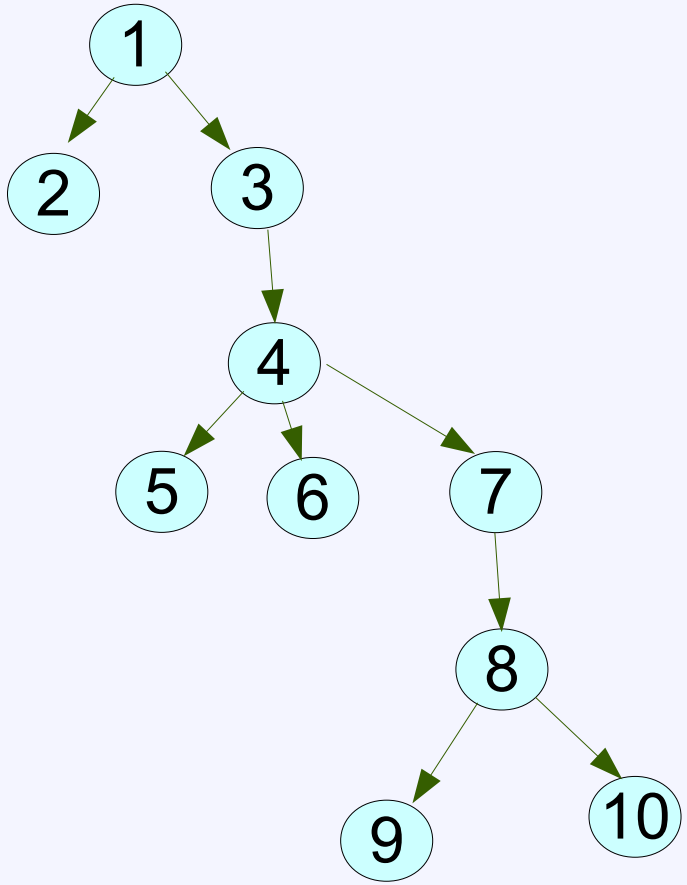
Ďalšie vlastnosti dominátorov

- Dominátory uzla n tvoria lineárne usporiadanú postupnosť podľa relácie dominancie $\succsim$.
- Táto postupnosť sa vyskytuje po každej ceste od počiatku n_0 po uzol n v rovnakom poradí.
- Označíme D(n) = {d: d $\succsim$ n} presnejšie $n_0=d_0 \succsim d_1 \ldots \succsim d_k=n$.
- Bezprostredný vlastný dominátor

  $d \succsim n \wedge d \neq n \wedge \neg\exists d'(d' \neq d \wedge d' \neq n \wedge d \succsim d' \succsim n)$.
- Graf G má strom dominátorov s koreňom n_0.
  - Strom dominátorov T_D je kostra grafu G.
  - S vlastnosťou, že dominátory uzla n sú práve jeho predchodcovia v strome dominátorov.

Príklad – strom dominátorov





Výpočet dominátorov

$D[n_0] := \{n_0\}$;
for each $n \in N - \{n_0\}$ **do** $D[n] := N$;
repeat nochange:= **true**;
for each $n \in N - \{n_0\}$ **do**
{ $D_{new} := \{n\} \cup \bigcap_{p \in pred(n)} D(p)$;
if $D[n] \neq D_{new}$ **then** { nochange:= **false**;
$D[n] := D_{new}$; }
}
until nochange;

Implementačné poznámky:

- D[n] bitový vektor (powerset N).

Worklist implementácia

$D[n_0] := \{n_0\}$;
for each $n \in N - \{n_0\}$ **do** $D[n] := N$;
worklist:= $succ(n_0)$;

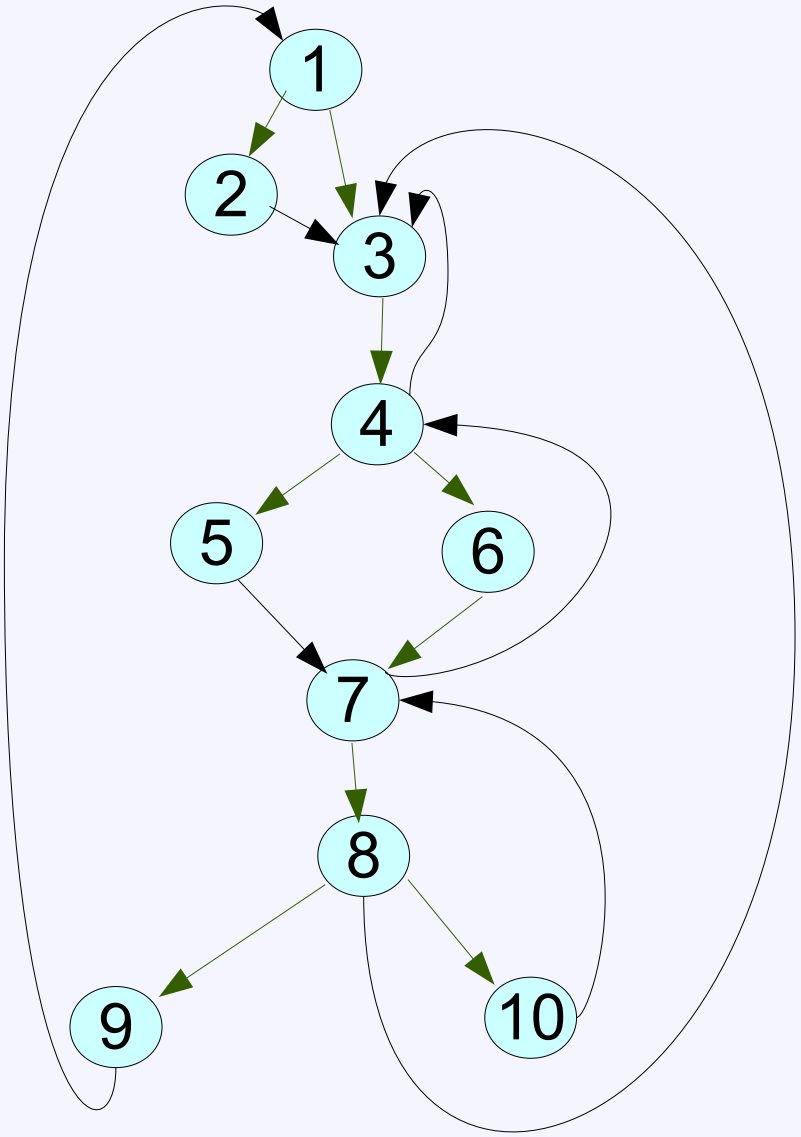
while worklist ≠ ∅ **do**
{ remove a node n from worklist;
$D_{new} := \{n\} \cup \bigcap_{p \in pred(n)} D(p)$;
if $D[n] \neq D_{new}$ **then** { $D[n] := D_{new}$;
worklist:= worklist ∪ $(succ(n) - \{n_0\})$; }
}

Prirodzený cyklus

- Spätná hrana od uzla n (chvost) k jeho dominátoru d (hlavička).
- Každej spätnej hrane zodpovedá cyklus – prirodzený cyklus pre túto hranu. Je to množina všetkých uzlov, z ktorých sa dá dostať do chvosta bez prejdenia hlavičkou.
- Algoritmus výpočtu prirodzeného cyklu

```
procedure insert(m);
if m ∉ Loop then { Loop:= Loop ∪ {m};
                   push(m, Stack);    }
procedure main(d,n);
{   Stack:= empty;
    Loop:= {d};
    insert(n);
    while not empty(Stack) do
        { m:= top(Stack); pop(Stack);
          for each p∈pred(m) do insert(p); }
 }
```

Príklad



Spätné hrany: $4 \rightarrow 3$, $7 \rightarrow 4$,
$8 \rightarrow 3$, $9 \rightarrow 1$, $10 \rightarrow 7$.

Prirodzené cykly:

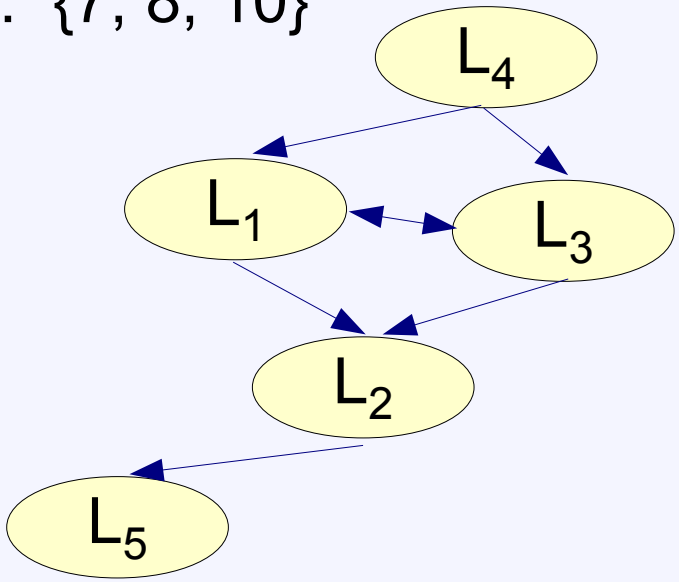
L_1: $4 \rightarrow 3$: {3,4,5,6,7,8,10}

L_2: $7 \rightarrow 4$: {4,5,6,7,8,10}

L_3: $8 \rightarrow 3$: {3,4,5,6,7,8,10}

L_4: $9 \rightarrow 1$: {1,2,3,4,5,6,7,8,9,10}

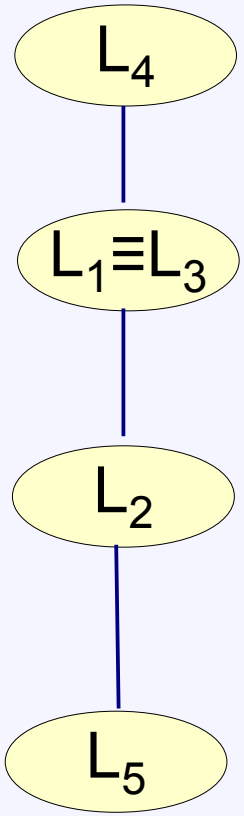
L_5: $10 \rightarrow 7$: {7, 8, 10}



Graf incidencie cyklov

- Uzlami grafu sú cykly, hrana medzi dvoma uzlami, ak majú aspoň jeden spoločný blok v grafe toku riadenia.
- Významný prípad cyklus je podmnožinou iného cyklu.
  - Loop nesting forest (Les zahniezdenia cyklov).
  - Často sa vyvára umelý uzol zo všetkých uzlov grafu toku riadenia (ak taký cyklus nexistuje). Potom sa les zahniezdenia cyklov stane stromom (loop nesting tree).
  - Pre „dobre štruktúrovaný“ program sú cykly buď disjunktné alebo zahniezdené (jeden je podmnožinou druhého)
- Zovšeobecnenie pre redukovateľné grafy
  - Cykly ktoré sú incidentné a nie je medzi nimi vzťah množinovej inklúzie považujeme za zahniezdené v poradí akom boli redukované. (Existujú aj zovšeobecnia pre neredukovateľné grafy.) Je to nejednoznačné (napr. L_1 a L_3).
- Listy stromu zahniezdenia = najvnútornejšie cykly

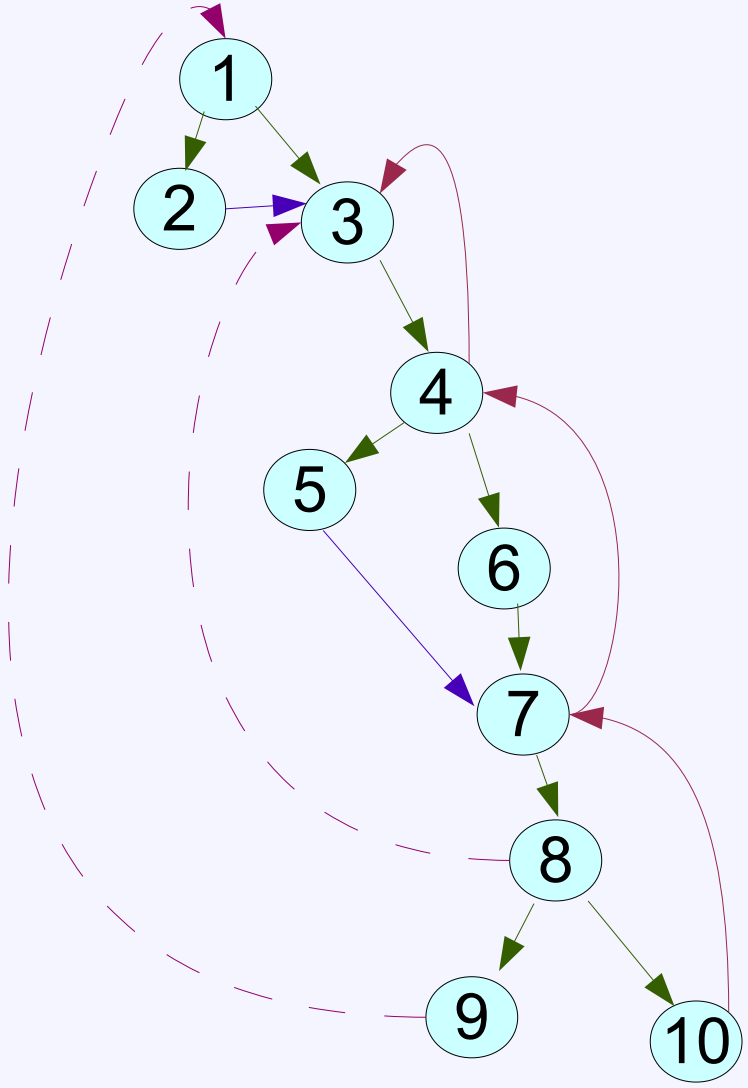
Strom zahniezdenia cyklov



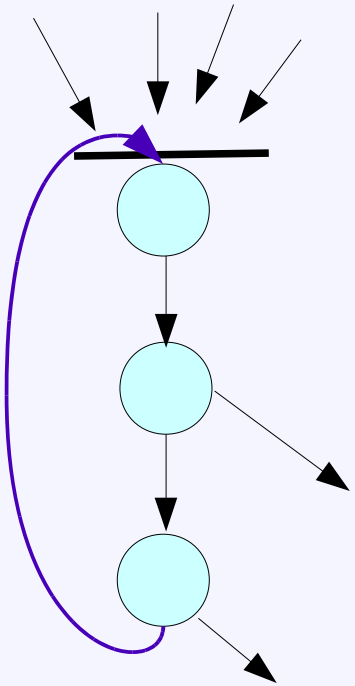
- Nakoniec jednoduchý obrázok
- Zdanie klame
- Najvnútornejší cyklus je $L_5 \equiv \{7,8,10\}$

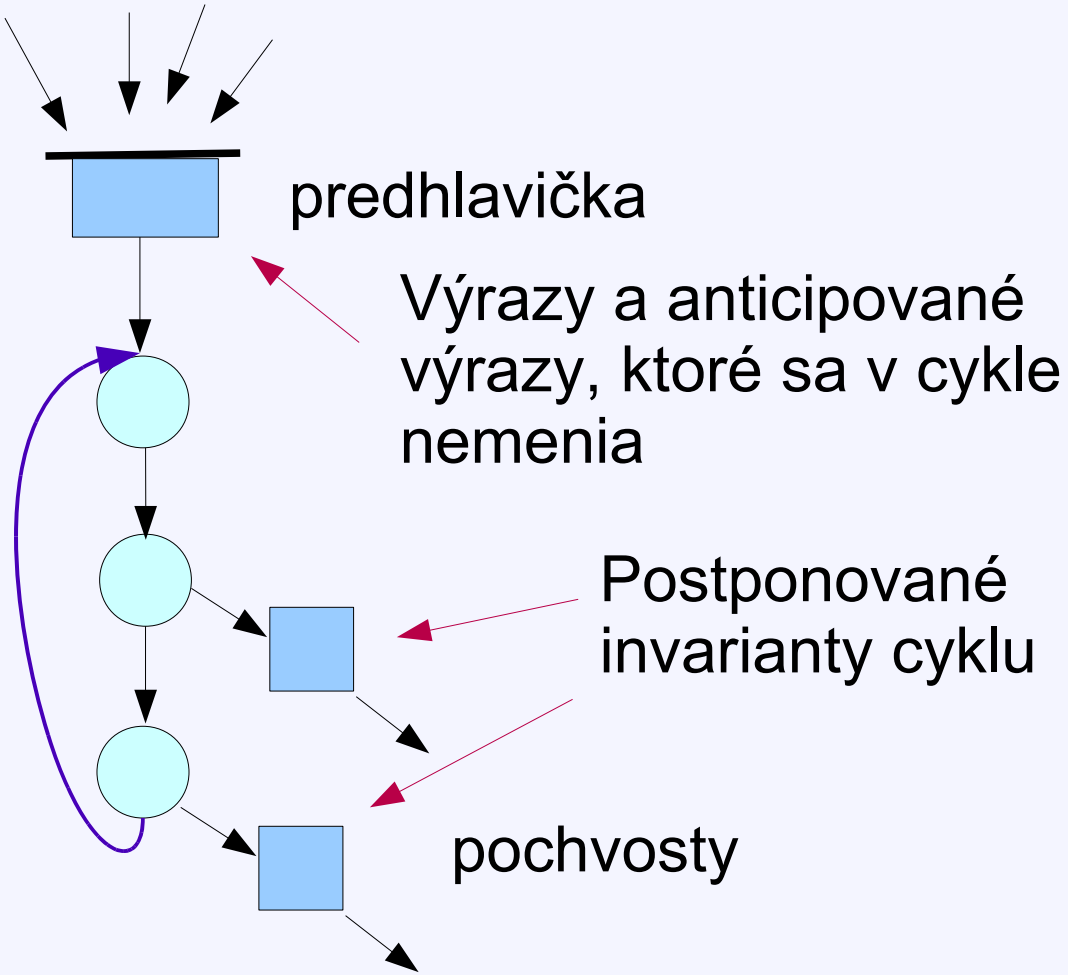
Iný spôsob určenia spätných hrán

```
procedure dfs(n);
{ mark(n);  //  visited
   for each successor s of n do
     if s not marked then
        { add edge n → s to T;
          dfs(s);                    }
   dfn[n]:= i;
   i:= i – 1;
}
procedure main;
{ i:= 0;
  for each node n of G do
     { unmark(n);  // unvisited
       i:= i + 1;                    }
  dfs(n0);
}
```



Presuny kódu (invariantov cyklu)





Identifikácia invariantov cyklu

- Zistiť výrazy ktoré sa v cykle nemenia
- Výpočet use-defifinition chains
  - a:= b + c je invariant v cykle, ak všetky definície, ktoré dosahujú (reaches) miesto tohto príkazu sú mimo cyklu.
- Anticipovaný invariant možno presunúť na akékoľvek miesto, ktoré dominuje všetky príkazy cyklu
- Miesto postponovaného invariantu
  - musí dominovať všetky výstupy, na ktorých má následné použitie (next use).
  - Ak také miesto neexistuje môže byť výhodné zopakovať výpočet tohto invariantu pre výstupy na ktorých ho treba.
- Uvážlivosť s transformáciami predlžujúcimi kód
  - Zdanlivá optimalizácia môže predĺžiť čas
  - Nadradený cyklus vďaka nej nevôjde do cache.

Globálne spoločné podvýrazy

- Vstup: Program rozdelený na bloky (graf toku riadenia), available expressions a reaching definitions.
- Výstup modifikovaný program.
- Metóda:
  1. Pre každý príkaz s: a:= b **op** c taký, že b a c sú dostupné na začiatku bloku B a v bloku B nie sú zmenené pred príkazom s.
  2. Nájdi všetky definície, ktoré dosahujú blok B a majú pravú stranu b **op** c.
  3. Vytvor novú dočasnú premennú t.
  4. Každý príkaz tvaru d:= b **op** c najdený v kroku 2 nahraď dvojicou príkazov: t:= b **op** c; d:= t.
  5. Nahraď príkaz s príkazom s': a:= t.

Deep common subexpessions:

x:= a + b	u:= a + b	Druhý príkaz až na druhý prechod.
y:= x*c	v:= u*c	