Compilatoare

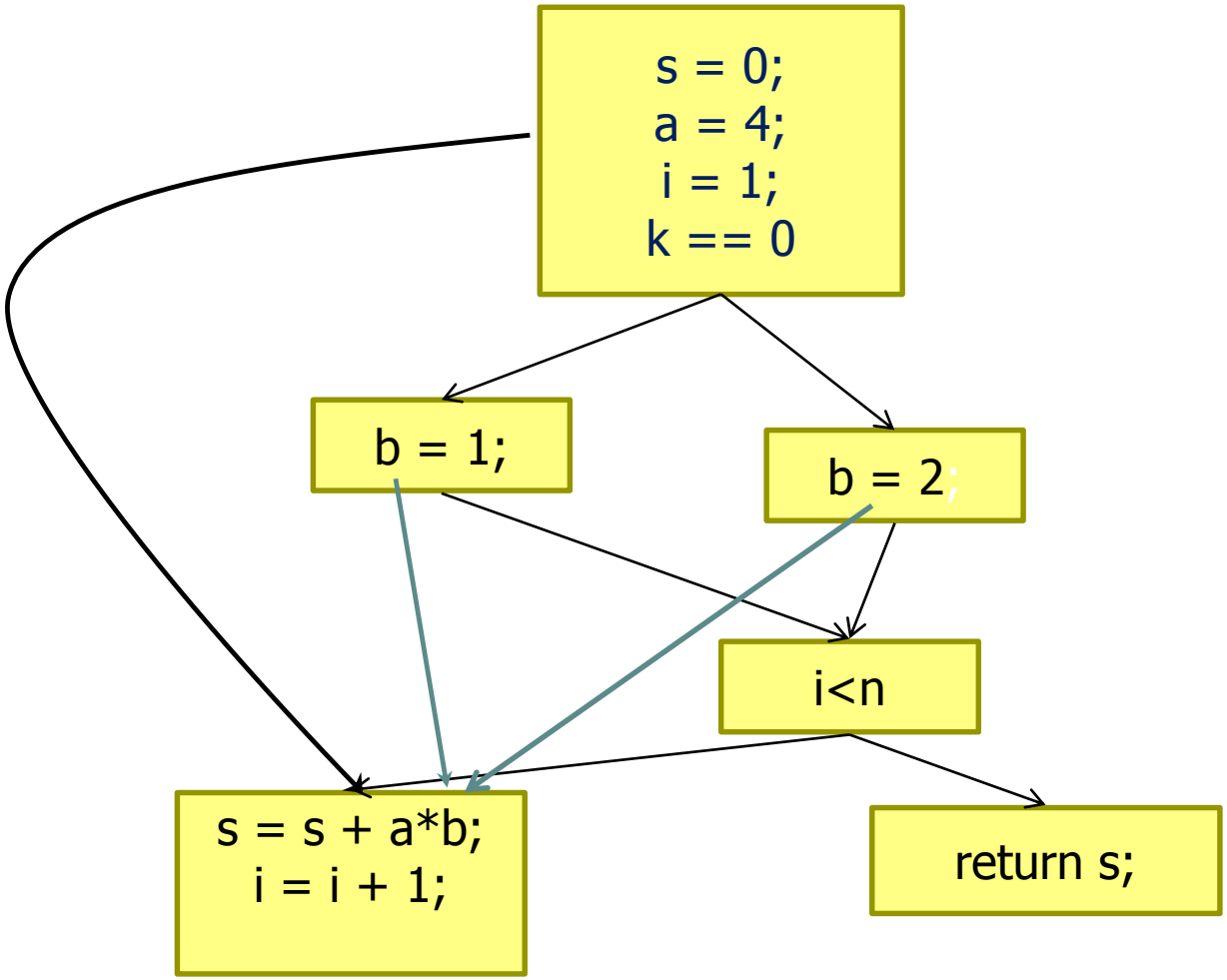
Analiza fluxului de date
Forma SSA

CPL
IR crunch

Review

- **Basic Block** – un set de instructiuni care se executa intotdeauna consecutiv
- **CFG** – Control Flow Graph
  - Fiecare muchie corespunde unui posibil flux de control al programului
- **Optimizare locala** – optimizare la nivelul unui singur basic block
- **Analiza globala** – analiza la nivelul CFG-ului

Reaching Definitions & Constant Propagation – exemplu



Is ***a*** constant in
s = s + a*b ?

YES

Is ***b*** constant in
s = s + a*b ?

NO

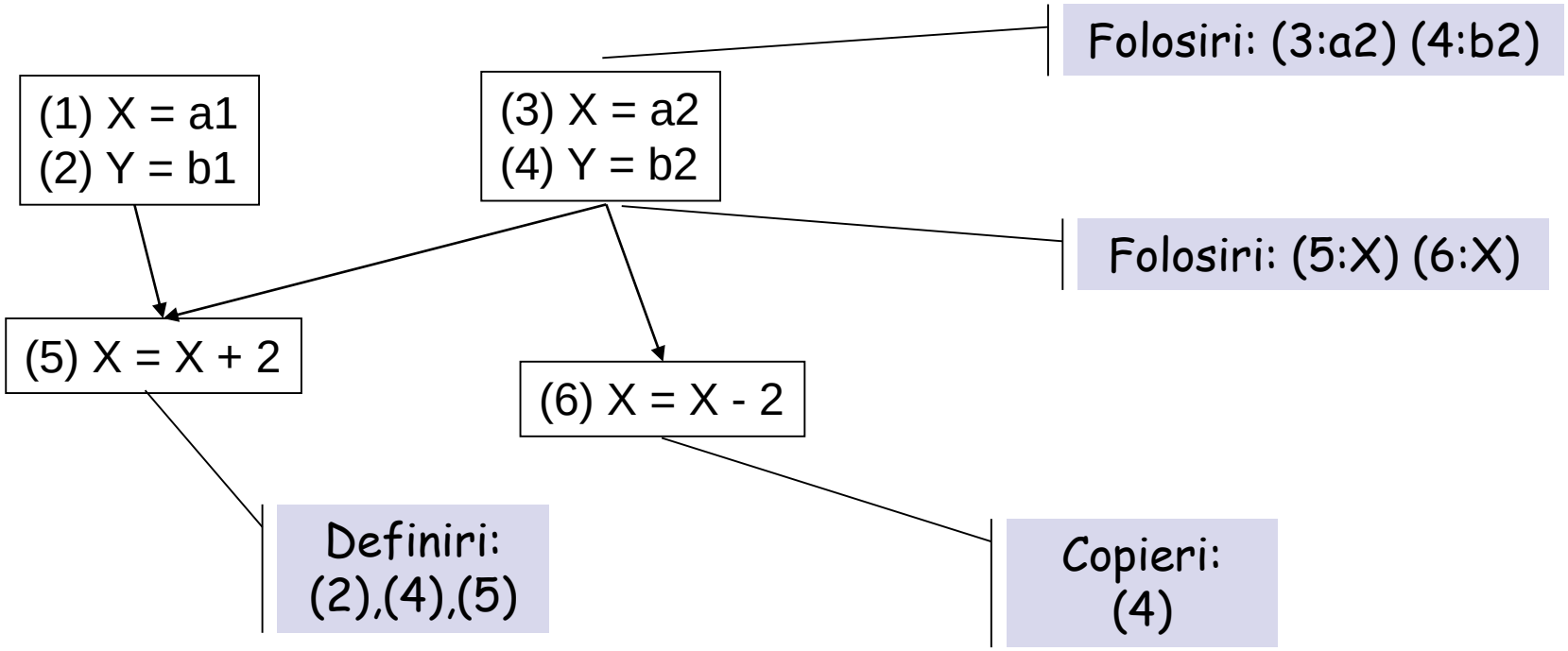
Cum rezolvam o problema de DFA

CPL
IR crunch

- Pasul 1: Care sunt valorile analizate? (cum ne reprezentam datele?)
- Pasul 2: Directia problemei? Daca o reducem la un BB, ne uitam “in jos” sau “in sus” pt. a afla raspunsul?
- Pasul 3: Ce se intampla cu informatia, pe un “if”? (meet vs. join)
- Pasul 4: Care e functia de flux a fiecarui bloc? (reprezentare: kill/def – cand e posibil)
- Pasul 5: Cum initializam informatia de flux? (hint: daca facem “OR” initializam cu 0, daca facem “AND” initializam cu 1)
- Pasul 6: Iteratia, pana ajungem la punct fix.

Analize de flux de date

- Definiri disponibile (“reaching definitions”)
- Folosiri expuse (“exposed uses”)
- Propagarea copierilor



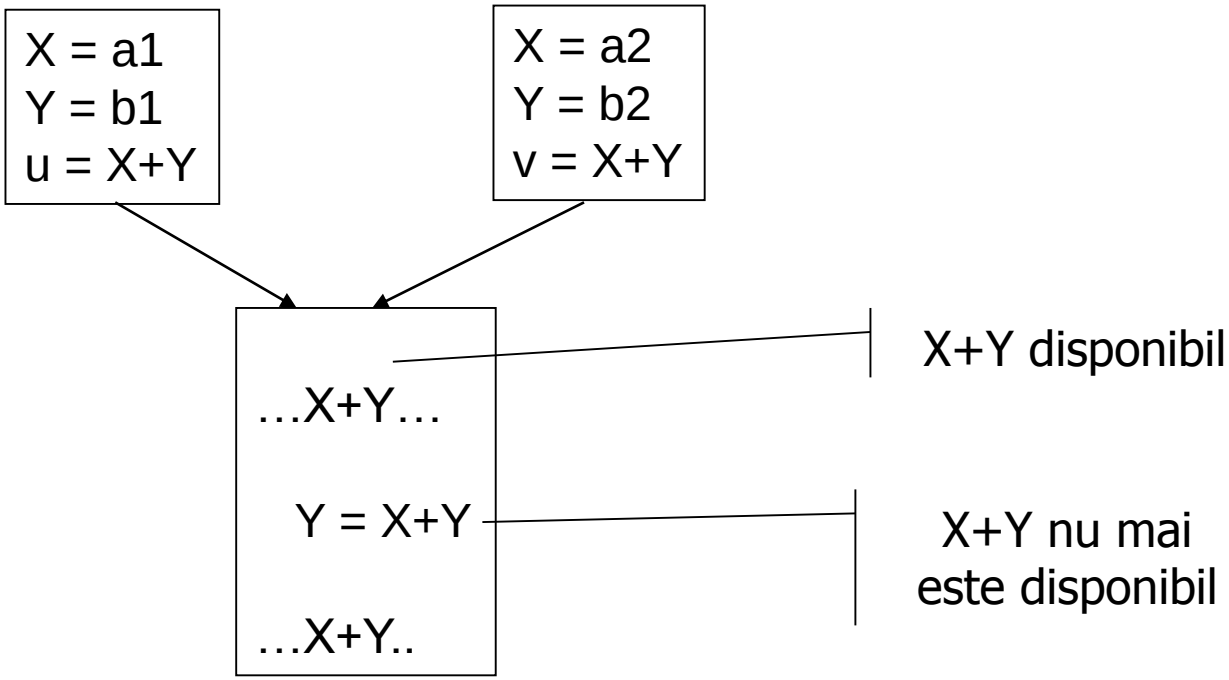
Alte analize de flux

- Exposed uses – reversul lui “reaching definitions”
  - Analiza inapoi
  - Cum se calculeaza functiile de flux?
  - Cum se combina informatiile din basic block-urile succesoare?

- Propagarea copierilor
  - Determina perechile de variabile care au intotdeauna aceeasi valoare la un punct al programului, in urma unei instructiuni de copiere.

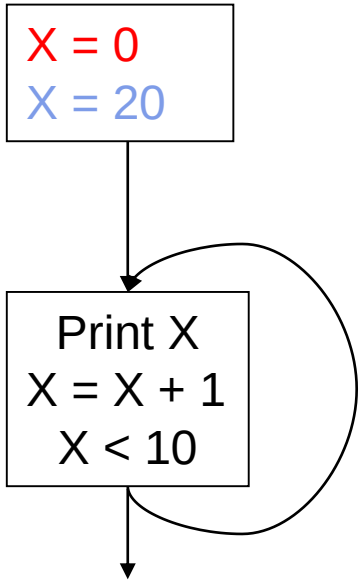
Alte analize de flux

- Expresiile disponibile
  - Determina expresiile care au fost deja calculate in program.
  - Utilizare: eliminarea calculelor redundante



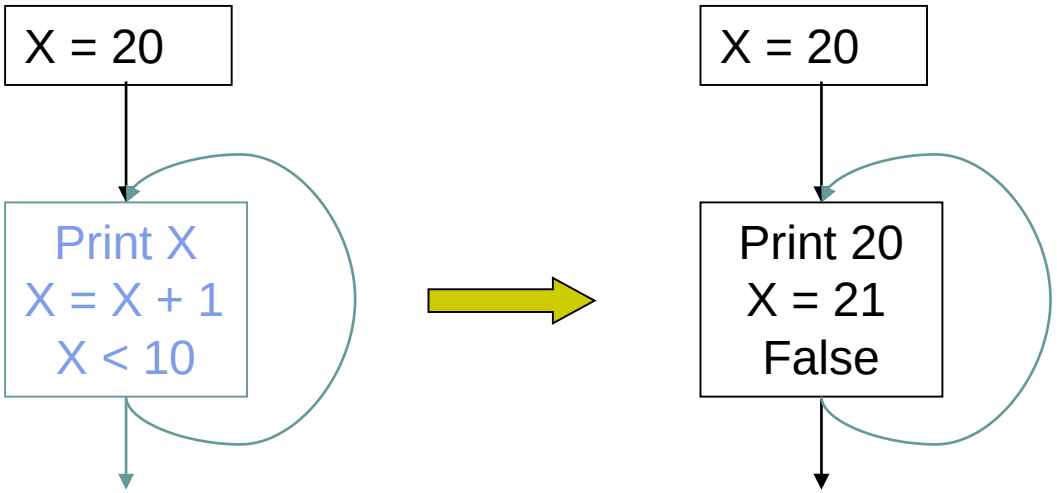
Alte analize de flux

- Propagarea constantelor
  - Determina perechile “variabila = valoare constanta“.
  - Doua valori speciale: “nedefinita“ si “nu e constanta“
    - const ∩ not-a-const -> not-a-const
    - const ∩ undef -> const
    - constA ∩ constA -> constA
    - constA ∩ constB -> not-a-const
  - Cate iteratii sunt necesare in exemplu?
  - Se poate simplifica pentru X = 20 ?
- Generalizare: domenii (“Range analysis“)
  - Determina multimea valorilor unei variabile



Alte analize de flux

- Propagarea conditionala a constantelor
  - Marcheaza muchiile din graf “vizitate”
  - Aplica operatorul “meet” doar pe predecesorii “vizitati”



Alte analize de flux

- Analiza variabilelor în viaţă
  - O variabilă *V* este *live* la sfârşitul unui basic block *n*, dacă există o cale fara definiri de la *n* la o folosire a variabilei V din alt bloc *n'*.
  - "live variable analysis problem" - determină setul de variabile care sunt "live" (în viaţă) pt orice punct din bloc.
  - Utilizari: alocarea registrilor

Alte analize de flux

- Eliminarea codului “mort”
  - Instructiune cu efecte laterale (nu poate fi eliminata)
    - Util(In, V) = Use(V) or (Util(Out,V) and not Def(V))
  - Instructiune fara efecte laterale (A = f(V))
    - Util(In, V) = (Util(Out,A) and Use(V))
      or (Util(Out,V) and not Def(V))

Metoda iterativă de DFA

- Stabileşte un set de functii de flux (pentru fiecare instructiune / basic block)
- Stabileşte un set de ecuaţii de flux (între basic blocks)
- Stabileşte valorile iniţiale
- Rezolvă/aplică iterativ ecuaţiile de flux, până când se ajunge la un **punct fix**.

Formalismul matematic

- Este algoritmul iterativ corect?
- Algoritmul converge catre o solutie?
- Este solutia exacta?

(Gary Kildall, 1972)

CPL
IR crunch

Formalismul matematic

- Latice – structura algebrica.
- Elementele laticei - proprietăti abstracte ale variabilelor, expresiilor sau altor componente din program.
- Fiecărui punct din program i se asociază un element de latice care memorează proprietăţile urmărite de analiză, in acel punct.
- Functiile de flux modeleaza efectul pe care îl are fiecare componentă a programului asupra elementelor de latice.
- Analiza calculeaza valorile laticei care rezolva ecuatiile date de functiile de flux.

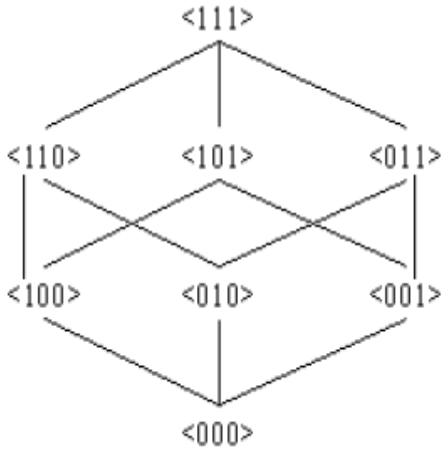
Formalismul matematic

- O latice L este formată dintr-o mulţime de valori si două operaţii pe care le vom nota ∩ ("meet") si U ("join") si care au următoarele proprietăţi:

1. Pentru orice $x, y \in L$ exista $z \in L$ si $w \in L$ unici, astfel încât $x \cap y = z$ şi $x \cup y = w$ ***(închidere)***
2. Pentru orice $x, y \in L$, $x \cap y = y \cap x$ si $x \cup y = y \cup x$ ***(comutativitate)***
3. Pentru orice $x, y, z \in L$, $(x \cap y) \cap z = x \cap (y \cap z)$ şi $(x \cup y) \cup z = x \cup (y \cup z)$ ***(asociativitate)***
4. Pentru orice $x, y \in L$, $(x \cap y) \cup y = y$ si $(x \cup y) \cap x = x$ ***(absorbtia)***
5. Exista doua elemente unice ale lui L, pe care le numim min (notat $\perp$) şi max (notat T), astfel încât $\forall x \in L$, $x \cap \perp = \perp$ si $x \cup T = T$ ***(unicitatea existentei elementelor de minim si maxim)***.
6. Numeroase latici sunt şi ***distributive***, adică pentru orice x, y, z ∈ L, avem: $(x \cap y) \cup z = (x \cup z) \cap (y \cup z)$ şi $(x \cup y) \cap z = (x \cap z) \cup (y \cap z)$.

Latici pt analiza de date

- In cazul analizei de date, cele mai multe dintre laticile folosite au ca elemente constituente vectori de biţi iar operaţiile de bază sunt reprezentate de operaţiile AND (meet) si OR (join) aplicate pe biţi.
- Elementul min,max – vectori de 0 / 1
- Pentru o problema particulara de analiza a fluxului de date, o functie de flux (f : L $\rightarrow$ L) modeleaza efectul unei parti de program asupra “datelor”



Semi-latice

- Algoritmul iterativ nu foloseste decat o singura operatie, $\cap$ (+ functiile de flux)

1. Pentru orice $x, y \in L$ exista $w \in L$ unic, astfel încât $x \cap y = w$
2. Pentru orice $x \in L$, $x \cap x = x$
3. Pentru orice $x, y \in L$, $x \cap y = y \cap x$
4. Pentru orice $x, y, z \in L$, $(x \cap y) \cap z = x \cap (y \cap z)$
5. Exista un element unic al lui L, max (notat T), astfel încât $\forall x \in L$, $x \cap T = x$
6. Optional, exista un element min (notat $\perp$) astfel incat $x \cap \perp = \perp$
7. Exista o relatie de ordine partiala, $x \subseteq y$ daca $x \cap y = x$

Algoritmul iterativ

- Directia inainte

```
out[entry] = init;
for each block B: out[B] = T;
while out changes
    for each block B
        in[B] = ⋂ out[P], P pred B
        out[B] = f_B(in[B]);
```

- Directia inapoi

```
in[exit] = init;
for each block B: in[B] = T;
while in changes
    for each block B
        out[B] = ⋂ in[S], S succ B
        in[B] = f_B(out[B]);
```

Functii monotone

- Operatiile U şi ∩ introduc o relaţie de ordine partiala pe elementele laticei
  - $x \subseteq y \Leftrightarrow x \cap y = x$. (+ op. dual)
  - proprietati
    - Reflexivitate : x<= x
    - Antisimetrie : daca y <=x si x<=y => x= y
    - Tranzitivitate : daca x<=y si y<=z => x <= z
- *Inaltimea* unei latici – lungimea maxima a unui lant x min $\subset$ x $\subset$... $\subset$ y $\subset$ max
- O funcţie ce mapeaza laticea pe ea insasi (f : L $\rightarrow$ L) este **monotona** dacă pentru $\forall$ x,y $\in$ L, x $\subseteq$ y => f(x) $\subseteq$ f(y)

Algoritmul iterativ

- Daca algoritmul iterativ converge, atunci rezultatul este corect - o solutie a ecuatiilor de flux.
- Daca toate functiile de flux sunt monotone si algoritmul converge, atunci solutia este maximala (Maximum Fixed Point)
- Daca functiile de flux sunt monotone si laticea are inaltime finita, algoritmii de analiza a fluxului de date se termina.
  - Range analysis – latice infinita

Corectitudine si precizie

- De ce e algoritmul corect?
  - Fiecare iteratie executa un pas pe o cale din CFG (conservator)
  - MFP – echivalent cu iterarea la infinit - acopera toate caile

- Cazul ideal vs. MOP vs. MFP
  - Ideal - toate caile posibile din program sunt executate, laticea contine proprietatile comune.

    $IN_B = \bigcap f_{Bn}(...f_{B1}(IN_{entry}))$ B1…BN cale executata pana la B
  - Meet Over Paths – toate caile din CFG sunt executate

    $IN_B = \bigcap f_{Bn}(...f_{B1}(IN_{entry}))$ B1…BN cale pana la B
  - Maximum Fixed Point - ce calculam noi

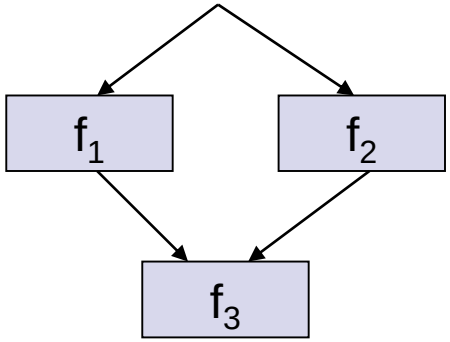
Functii distributive

- $f(x \cap y) = f(x) \cap f(y)$
- Daca functiile de flux sunt distributive, atunci MOP si MFP calculeaza acelasi lucru
- Functiile de flux de tipul

  $OUT_B(IN_B) = GEN_B \cup (IN_B - KILL_B)$
  sunt distributive.
  - GEN – setul de definitii generate in bloc
  - KILL – setul de definitii care dispar in bloc (Ex: o redefinire a unei variabile “omoara” definitia anterioara)

MFP vs. MOP

- MFP =MOP pt. functii de flux distributive
- Diferenta – de ex. propagarea constantelor

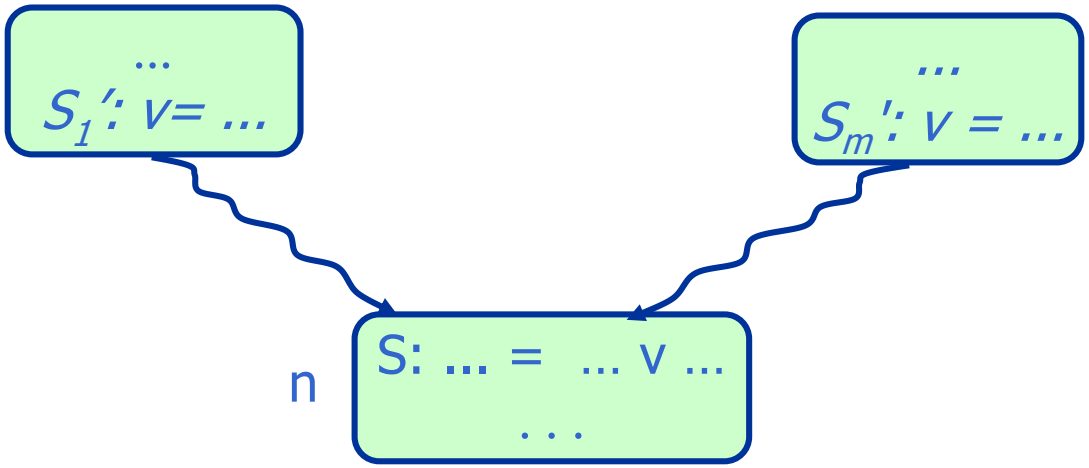


- MOP=f3(f1(entry)) ∩ f3(f2(entry))
- MFP=f3(f1(entry) ∩ f2(entry))

```
    bool a,b,x;
(1)
    a = false;
(2)
    b = false;
(3)
    if (x)
(4)
      {
(5)
      a = true
(6)
      }
    else
      {
(7)
      b = true
(8)
      }
(9)
    z = a or b
(10)
        if (z) ...
```

UD Chain

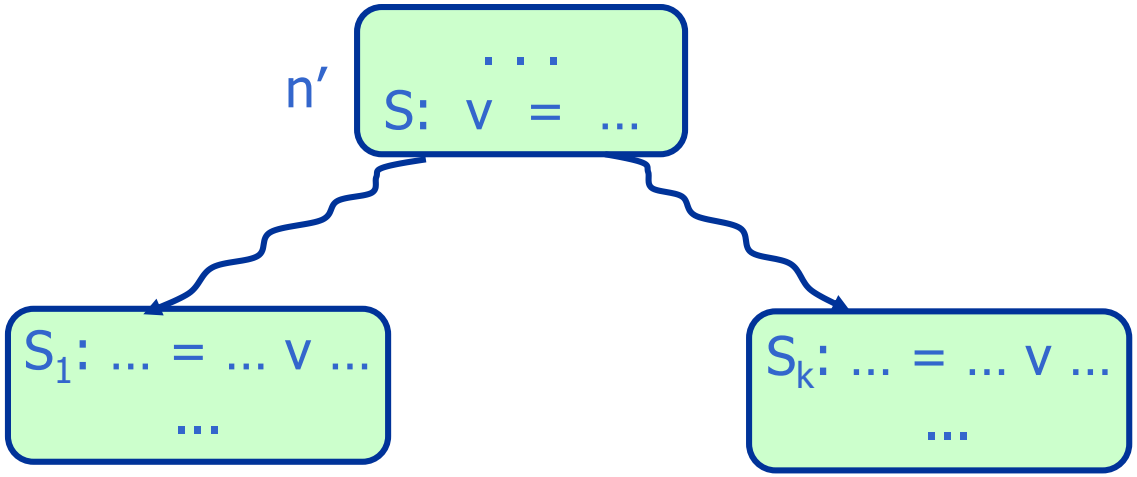
Un lant use-def(UD chain) e o lista a tuturor definirilor care pot ajunge la o folosire a unei variabile.



UD chain: UD(n, v) = (S_1', ..., S_m').

DU Chain

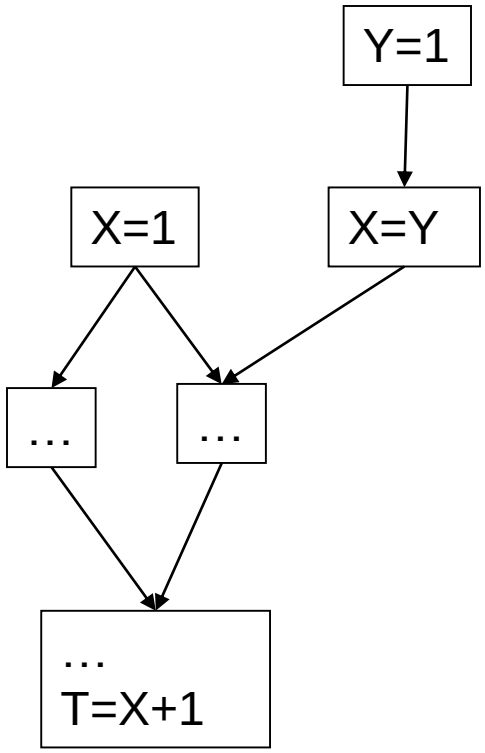
Un lant def-use (DU chain) e o lista a tuturor folosirilor la care se poate ajunge de la o anumita definire a unei variabile.

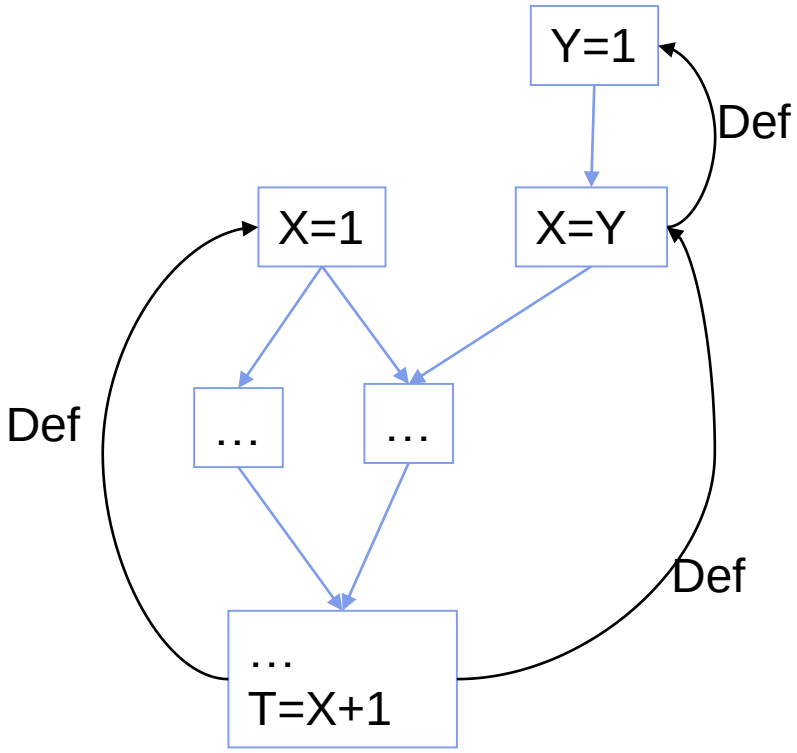


DU chain: DU(n', v) = (S_1, …, S_k).

Folosirea Def-Use Chains

- Propagarea constantelor – poate fi iterata pe CFG sau pe UD-chain (“sparse constant propagation”)





Aplicaţiile analizei def/use

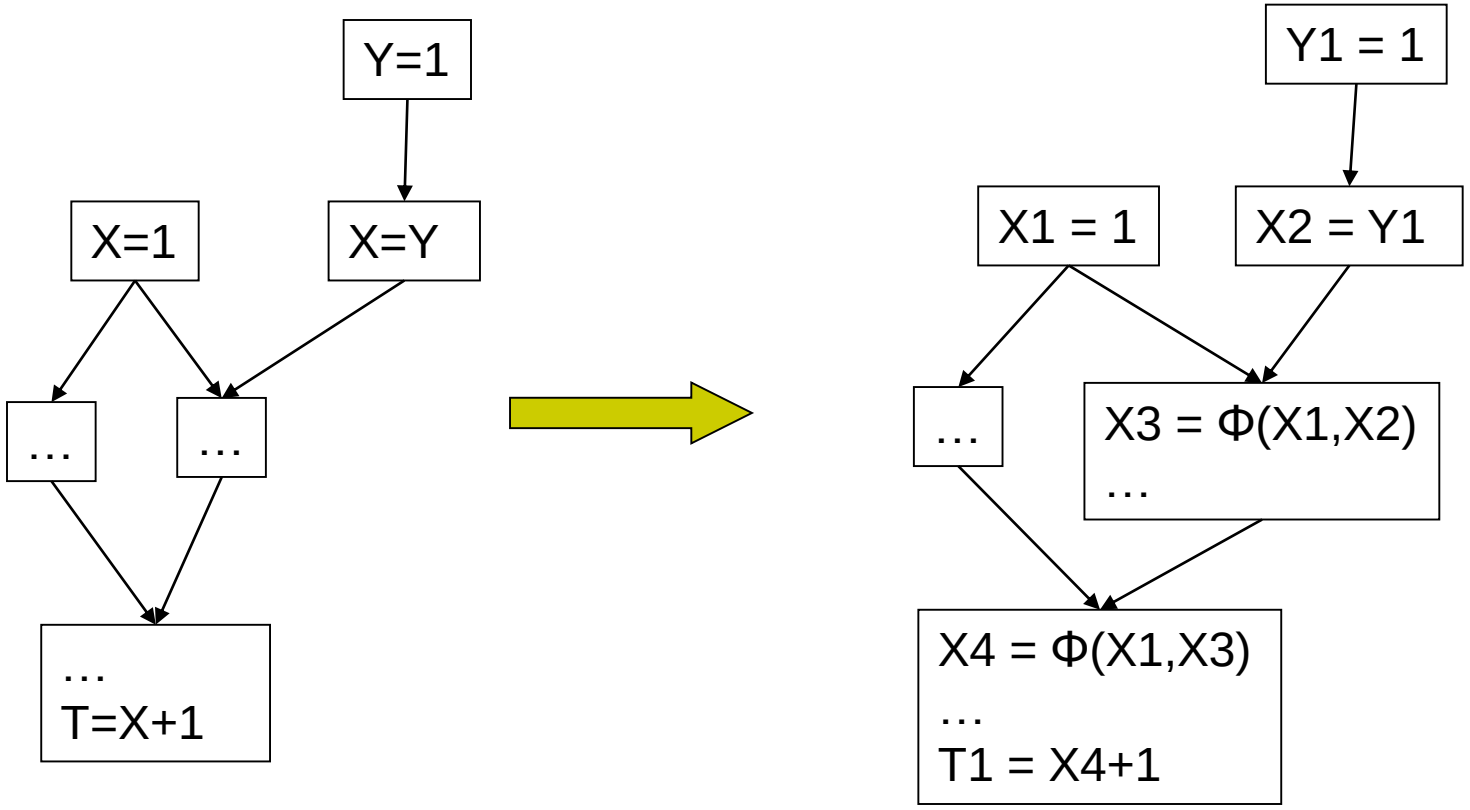
DU-chain - structura de date principala pentru optimizari

- Propagarea constantelor, copierilor
- Analiza variabilelor in viata si eliminarea codului inutil (“dead” code)
- Eliminarea calculelor redundante
- Detectia variabilelor de inductie (index de buclă)
- Descoperirea dependentelor, planificarea si paralelizarea instructiunilor.
- Analiza de alias
- …si multe alte analize pentru diverse transformari

… dar consuma memorie: M – definiri, N folosiri => O(M x N)

Forma SSA

- Reprezentare intermediara ce permite optimizari mai rapide decat folosind DU-chain:



CPL
IR crunch

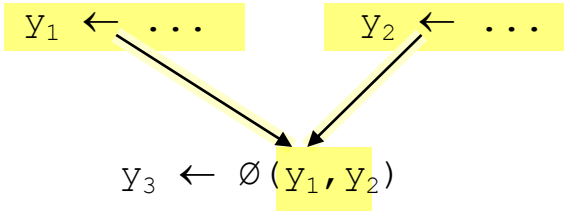
SSA

- SSA: Un program e in forma SSA daca si numai daca
  - Fiecare variabila e definita (static) exact o singura data, si
  - Fiecare folosire a unei variabile e dominata de definirea acelei variabile

Aceasta unica definire statica poate fi intr-o bucla si deci executata de multe ori. Astfel, chiar si intr-un program SSA o variabila poate fi definita in mod dinamic de mai multe ori.

Functia Ø

- Functia Ø e o copiere speciala care selecteaza unul din parametri.
- Alegerea parametrului e guvernata de arcul din CFG pe care s-a ajuns la blocul curent



- Procesoarele reale nu implementeaza functii Ø direct in hardware (este posibil?)

CPL
IR crunch

SSA: Motivatia

- Ofera o baza uniforma la nivel de IR pentru a rezolva o gama larga de probleme de flux de date
- Codifica informatie de flux de date + control
- Forma SSA poate fi construita si intretinuta eficient
- Multi algoritmi de analiza de date/optimizare pe forma SSA sunt mai eficienti (au complexitate mai scazuta) decat varianta pe CFG.

Forma SSA – exemplu

Forma SSA

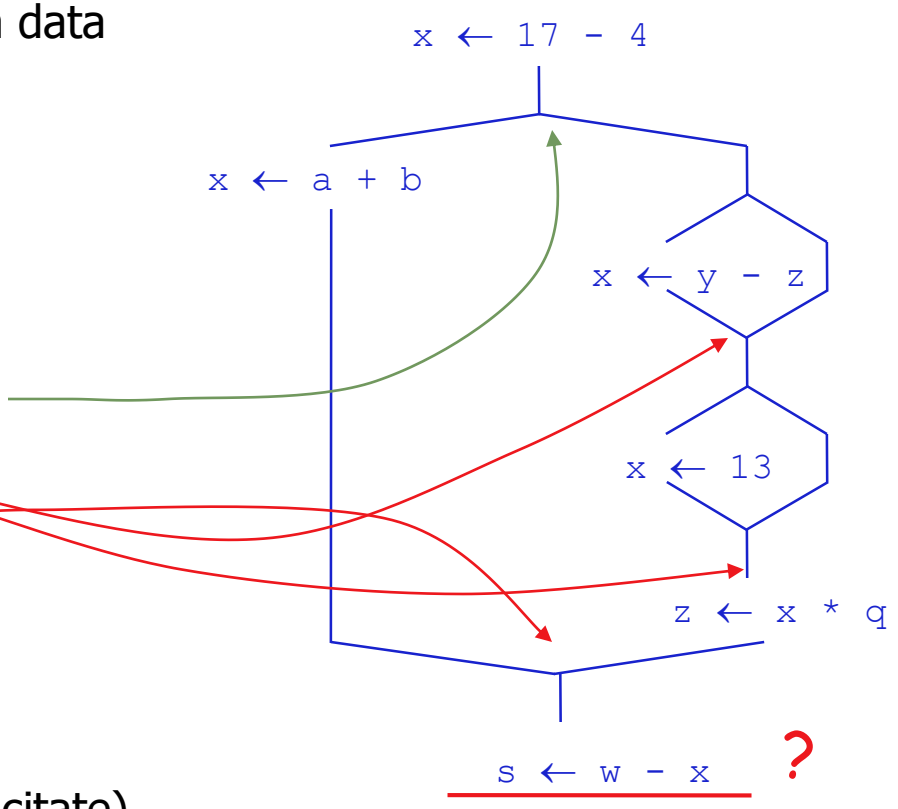
- Fiecare nume e definit exact o singura data
- Fiecare folosire refera un singur nume

Ce e mai greu

- Codul liniar e usor de transformat
- 'Splits' in CFG - usor de transformat
- 'Joins' in CFG - sunt mai problematice

Construirea formei SSA

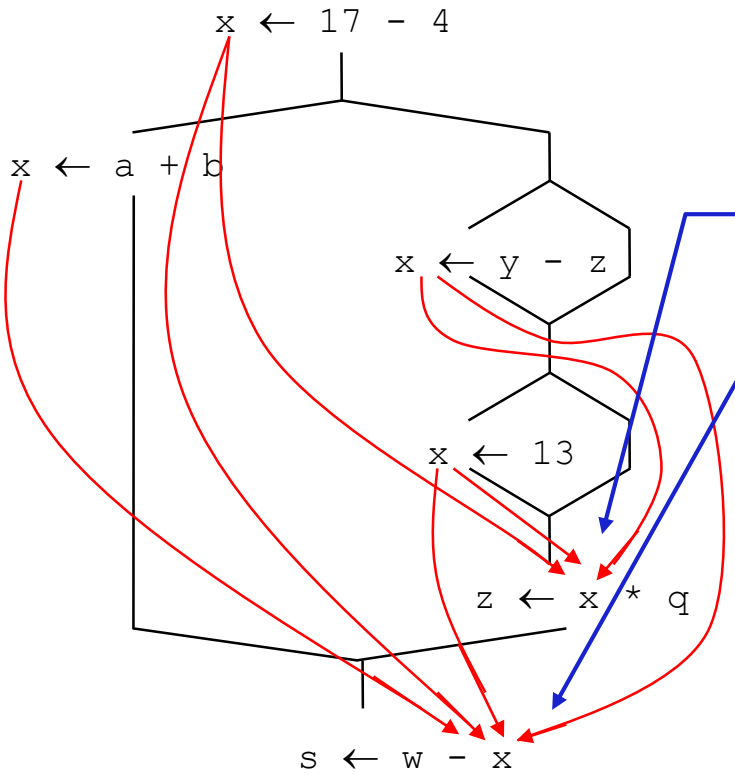
- Insereaza functii Ø in 'birth points'
- Redenumeste toate variabilele (pt. unicitate)



*

Birth Points

- Sa luam exemplul urmator:



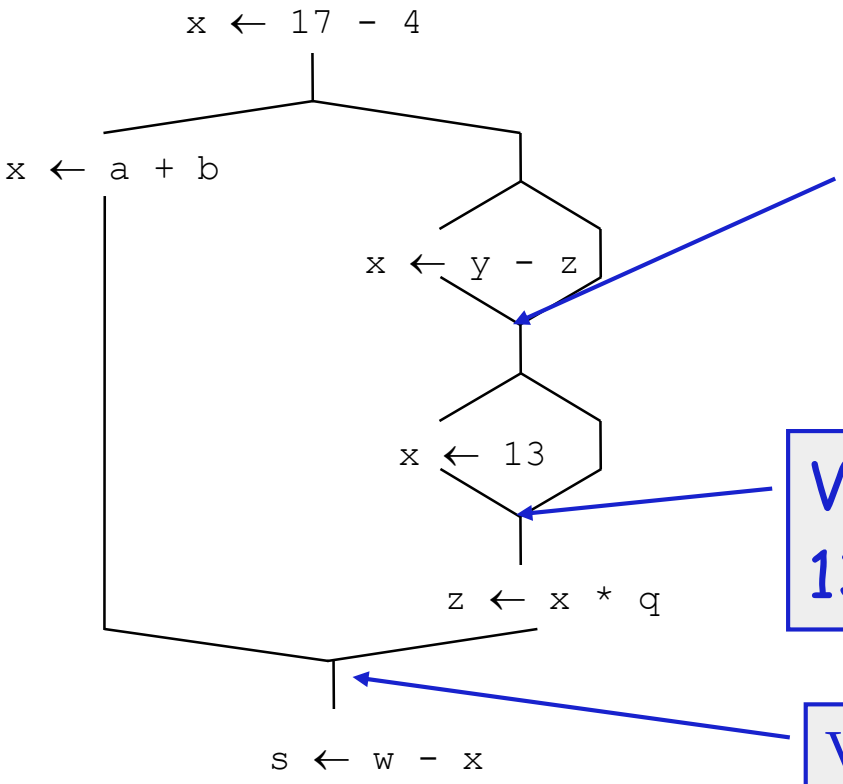
Variabila x apare peste tot si ia mai multe valori

- Aici, x poate fi 13, y-z, sau 17-4
- Aici, ar putea fi si a+b

Daca fiecare valoare are numele ei…

- Avem nevoie de o modalitate de a "uni" valorile distincte
- Valorile 'unite' se "nasc" la punctele de "join" din CFG *

Birth Points

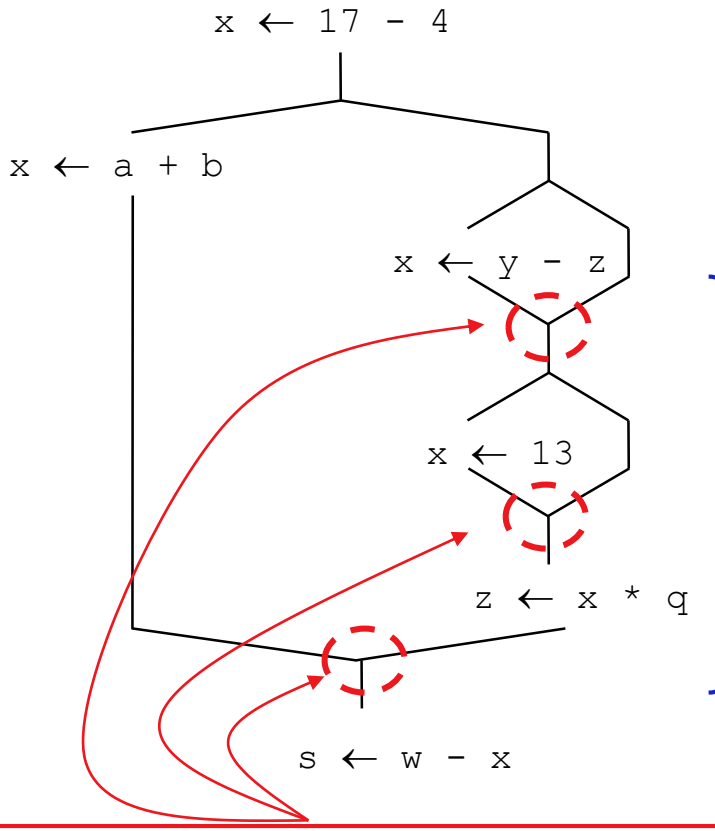


Valoare noua pt. x aici
17 - 4 sau y - z

Valoare noua pt. x aici
13 sau (17 - 4 sau y - z)

Valoare noua pt. x aici
a+b sau ((13 sau (17-4 sau y-z))

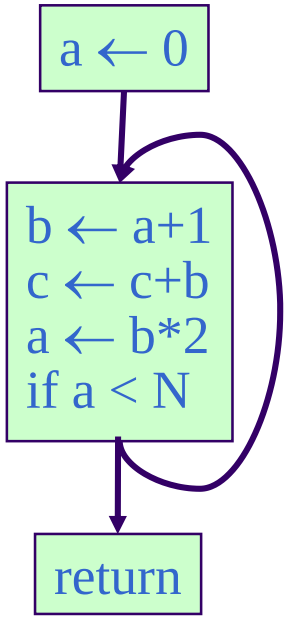
Birth Points

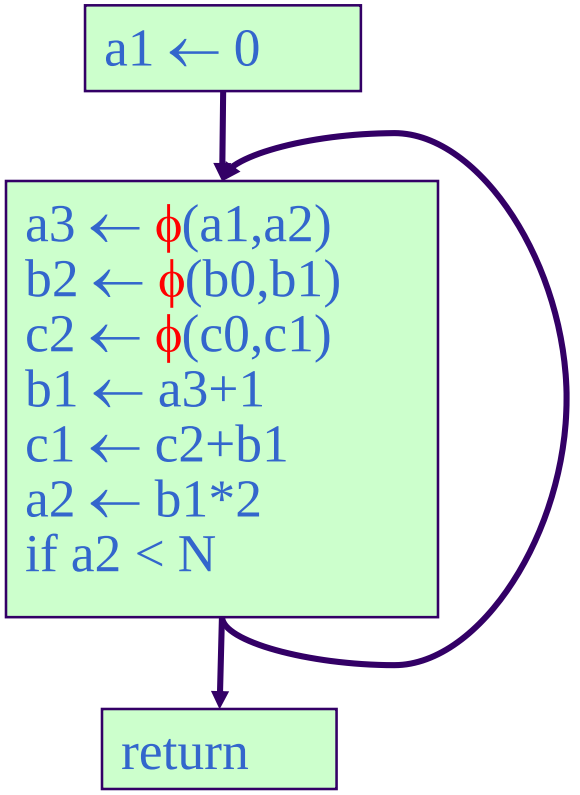


- Toate 'birth points' sunt join-uri pe CFG
- Nu toate join-urile sunt birth points
- Birth points sunt specifice fiecarei variabile …

birth points pentru valorile lui x

Un exemplu cu bucla





ϕ(b0,b1) nu e necesar, caci b1 nu e folosit. Dar faza care genereaza functiile ϕ nu stie asta. Functiile nenecesare sunt eliminate de dead code elimination.

Nota: doar a, c sunt folosite in bucla inainte de a fi de a fi redefinite.

Criterii pt inserarea functiilor ϕ

Am putea insera o functie ϕ pt fiecare variabila la fiecare *join* (punct cu mai mult de un predecesor). Dar ar fi un numar exagerat de mare.

Adaugam functii ϕ la 'birth point' – dar cum stim care sunt?

Intuitiv, adaugam o functie ϕ daca 2 definiri ajung la un punct z pe cai diferite

Criteriul convergentei cailor

Insereaza o functie ϕ pt. o variabila *a* la nodul *z* daca urmatoarele conditii sunt adevarate:

1. Exista un bloc *x* care defineste *a*
2. Exista un bloc $y \neq x$ care defineste *a*
3. Exista o cale nevida P_{xz} de la *x* la *z*
4. Exista o cale nevida P_{yz} de la *y* la *z*
5. Caile P_{xz} si P_{yz} nu au noduri in comun in afara de z
6. Nodul *z* nu apare si in P_{xz} si in P_{yz} inainte de sfarsit, (dar poate aparea in una dintre cai).

Blocul de start (Entry) contine o definire implicita a tuturor variabilelor.

Criteriul convergentei cailor

Functia ϕ e ea insasi o definitie.
De aceea, criteriul de mai sus trebuie iterat:

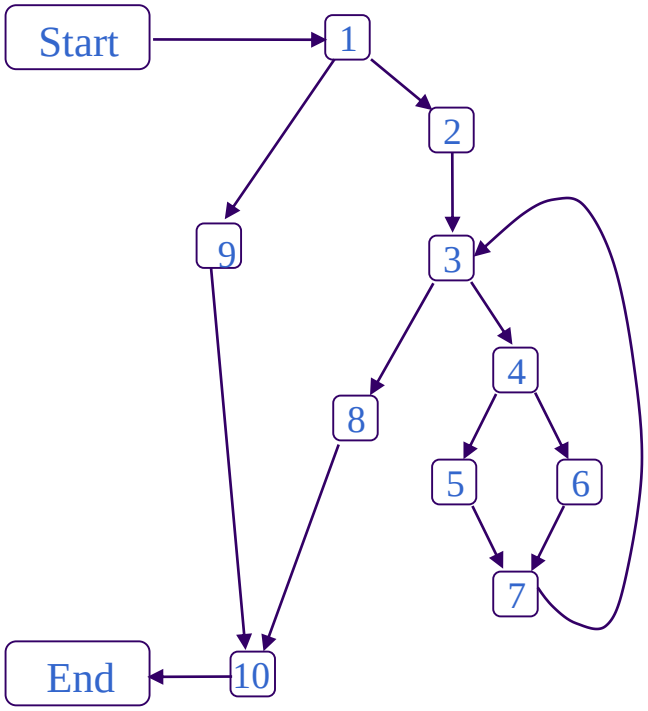
Cat timp exista noduri *x*, *y*, *z* indeplinind conditiile 1-5
si *z* nu contine o functie ϕ pentru *a*
adauga $a \leftarrow \phi(a, a, \ldots, a)$ la nodul *z*

Acest algoritm e extrem de costisitor, deoarece necesita examinarea tuturor perechilor (x,y,z) si a tuturor cailor x->y

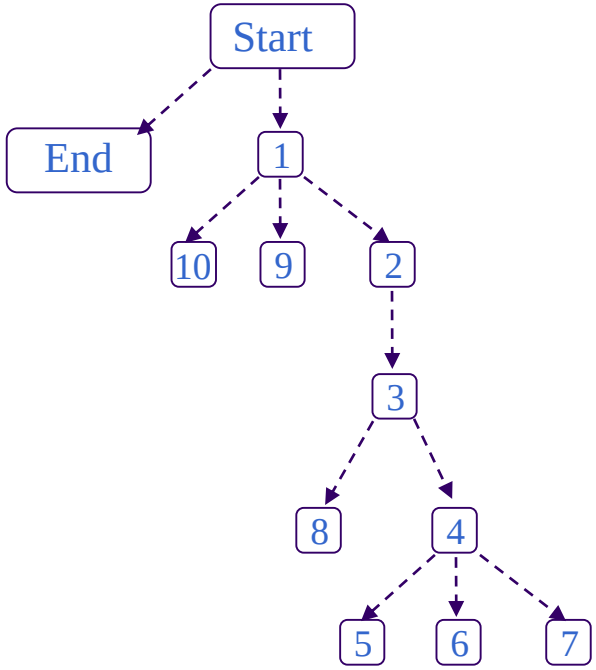
Algoritm mai eficient

- Foloseste notiunea de “Frontiera de dominare”
- Frontiera de dominare a blocului B: DF(B)
  - Setul de bb nedominate de B, care au cel putin un predecesor dominat de B.
  - Definitia se poate extinde la un set de noduri.
- Frontiera de dominare iterata a lui B, IDF(B)
  - Cel mai mic set de noduri care contine DF(B) si e punct fix relativ la aplicarea functiei DF.

Conceptul de DF



(a) CFG



(b) Arbore de dominare

Nota: Consideram o legatura implicita intre ‘start’ si ‘end’

DF(3) = {3, 10}

Definitia formala

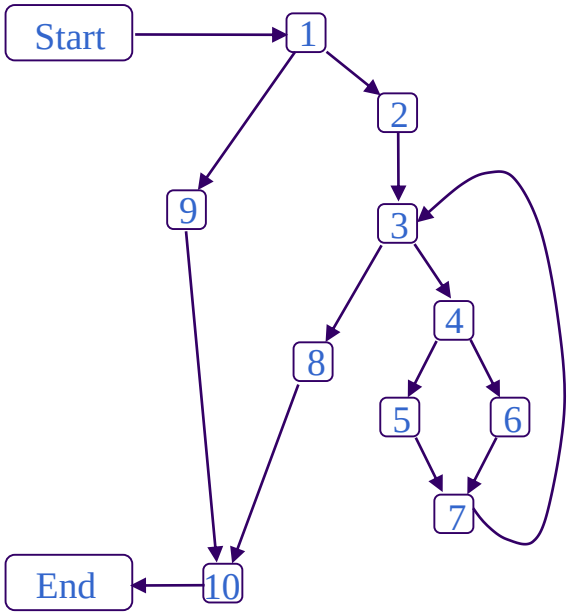
Frontiera de dominare DF (x) a unui nod x e setul de noduri y care au un predecesor z, cu proprietatea ca x dom z dar x ! sdom y.

Pentru un set de noduri S:

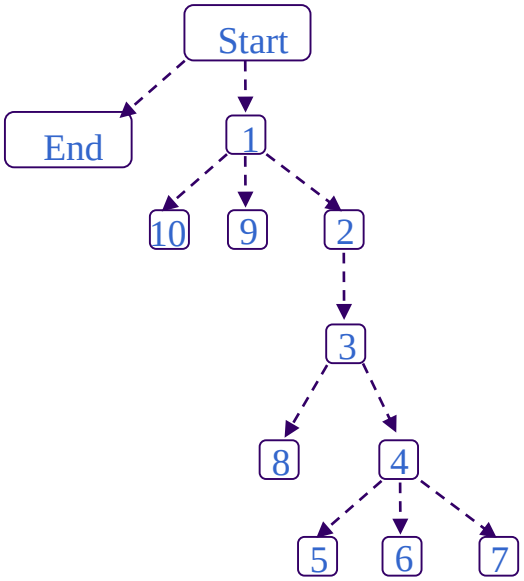
$$DF(S) = \cup (DF(x)), x \text{ in } S$$

Frontiera de dominare iterata *IDF(S)* pt un set de noduri S e multimea maximala din secventa:

$$IDF_1(S) = DF(S),$$
$$IDF_{i+1}(S) = DF(\cup (S, IDF_i(S))$$



(a)



(b)

DF(4) = {3}

IDF(4) = {3, 10, end}

IDF 1 (4) = DF{3}
IDF 2 (4) = DF{3 U *{3, 10}} = {3, 10, END}*
IDF 3 (4) = DF{3 U *{3, 10, END}} = {3, 10, END}*
In iteratia aceasta nu mai sunt schimbari, prin urmare
IDF(4) = {3, 10, END}

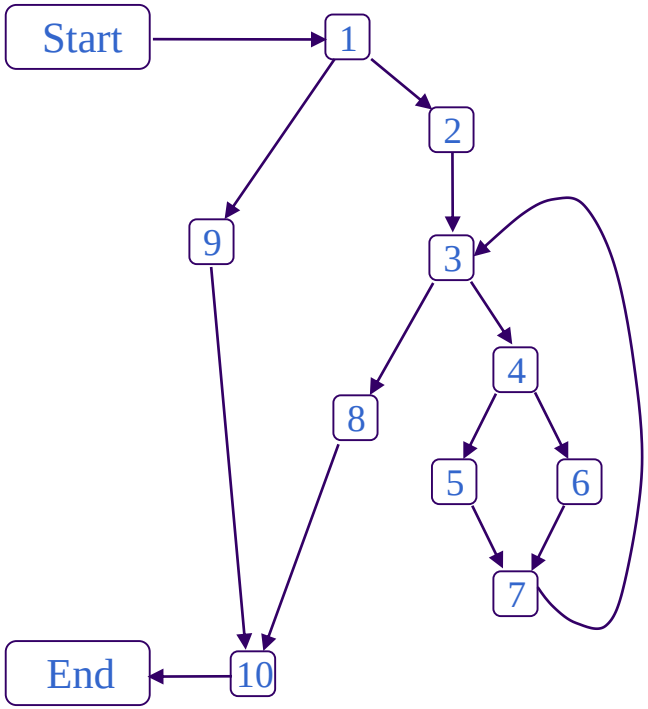
Cum se insereaza functii ϕ

1. Precalculeaza *DF(x)* pt. fiecare nod *x.*
2. Determina setul de noduri N_a. care contin definitia variabilei a
3. Calculeaza IDF(N_a) -> aici se insereaza fct. $\phi(a)$

Complexitatea calculului DF pentru un nod:

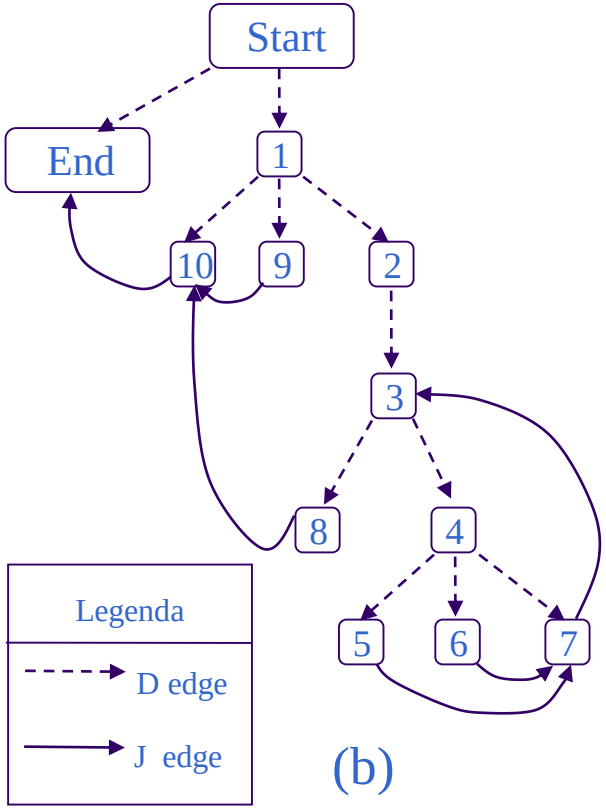
- implementare directa, O(N*N)
- Se poate si O(N)

Cum se calculeaza DF: Grafuri DJ



(a)

Graficul fluxului de control



(b)

Graful DJ

Calculul frontierei de dominare

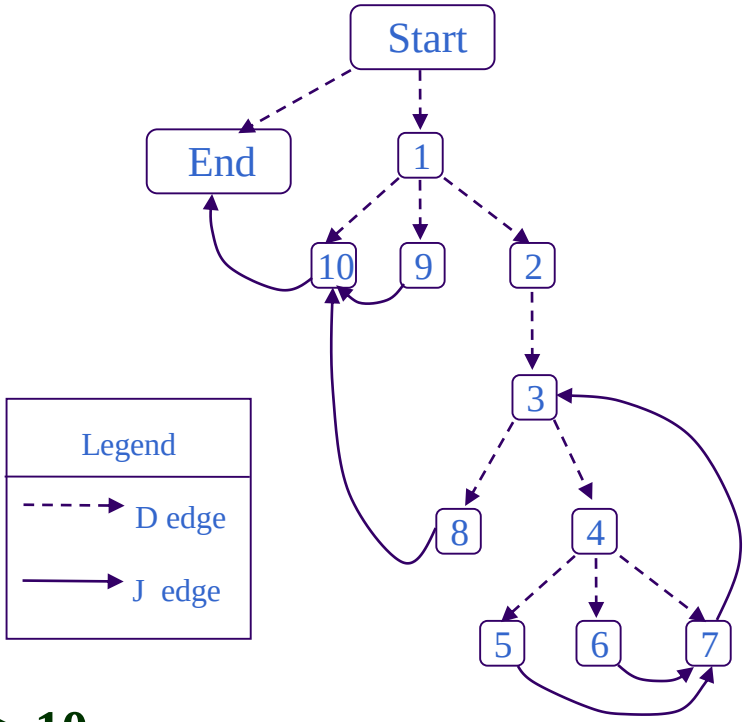
J-edge: o muchie x → y din CFG , x !sdom y

Formal:

DF(x) = Ø
For each y, x dom *y*
if*(y → z is J-edge)*
if*(z.level<= x.level)*
DF(x) = DF(x) U {*z*}

Complexitatea algoritmului e *0(*|*N*| + |*E*|*)*, cazul cel mai defavorabil

Exemplu



Note: legatura 8 -> 10
Level(10) < level(3). 10 este in DF(3) si IDF(3)

CPL
IR crunch

Eliminarea functiei ϕ

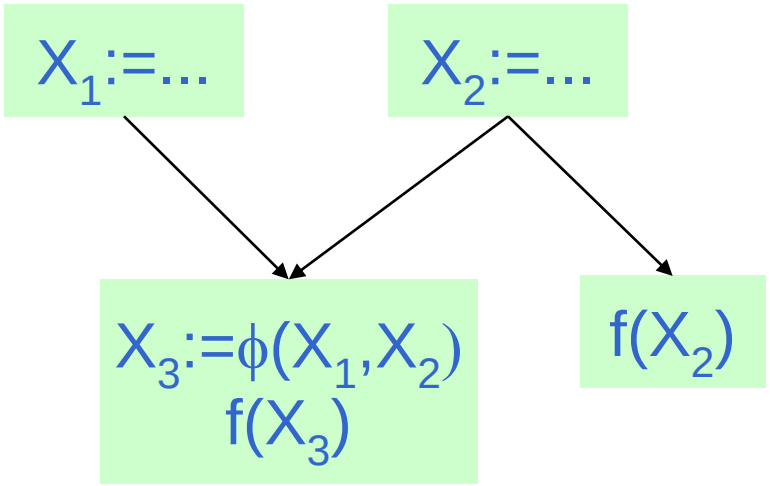
Cum implementam o functie ϕ care "stie" pe ce cale s-a ajuns la ea?

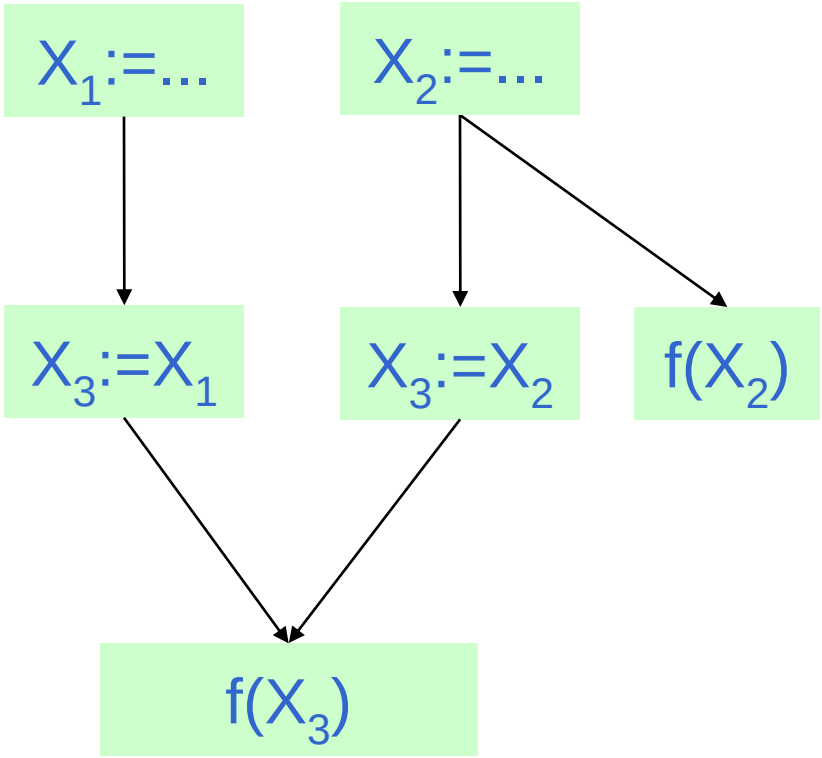
Rasp. 1: Nu o implementam! Functia ϕ e o conventie folosita doar pentru a conecta definiri cu folosiri in timpul optimizarilor, dar nu e implementata in practica.

(dupa optimizari ar putea sa nu mai fie posibil)

Rasp. 2: Ca sa 'scapam' de functiile ϕ, putem insera instructiuni "MOVE" pe toate caile de control.

Eliminarea functiei ϕ





Se poate folosi un singur registru pentru X_1,X_2,X_3?

Extragerea codului invariant

- O definire *d: t := x * y* este invarianta in bucla daca
  - x, y sunt constante, sau
  - toate definirile lui x si y se gasesc in afara buclei, sau
  - exista o singura definire a lui x (sau y) in bucla, si acea definire este invarianta.
- Conditii suficiente (conservatoare)

Extragerea codului invariant

- Codul invariant *d: t := x * y* poate fi mutat in pre-header daca
  - Definirile lui x si y se gasesc in afara buclei
  - Exista o singura definire a lui t in bucla
  - “d” domina toate iesirile din bucla in care t este in viata.
  - t nu este folosit la iesirea din pre-header
- Cum se scriu conditiile pentru SSA?
  - t nu are functii Φ in bucla

Extragerea codului invariant

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

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

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

CPL
IR crunch

Detectia variabilelor de inductie

- Variabila de inductie IV – “index” in bucla
  - Cresc la fiecare iteratie cu un pas constant
- Variabila de inductie de baza:
  - `I = I + c`, c este un invariant in bucla
- Variabila de inductie derivata
  - `J = I*a + b`, `I` este variabila de inductie, a,b sunt invarianti
- De ce?
  - Strength reduction
  - Analiza de dependenta

Variabile de inductie

```
int *a;
int i;
for (i=0; i<100; i++)
  a[i] = 100 - 2*i;
```

```
 i=0
L1:
 if i>=100 goto L2
 t1 = 2*i
 t2 = 100 - t1
 t3 = 4*i
 t4 = &a + t3
 *t4 = t2
 i=i+1
 goto L1
L2:
```

Detectia variabilelor de inductie

- Detectia variabilelor de baza
  - O singura definire in bucla, de tip I = I + c, c invariant
  - I = I * 1 + c. Notatie : I = (I, 1, c)
- Detectia variabilelor derivate
  - O singura definire in bucla, J = I*a + b, I este IV de baza, a,b sunt invarianti → J=(I,a,b)
  - O singura definire in bucla, K = J*a + b, J=(I,p,q), I nu este definit intre J si K
    - → K=(I, a*p, a*q+b)

CPL
IR crunch

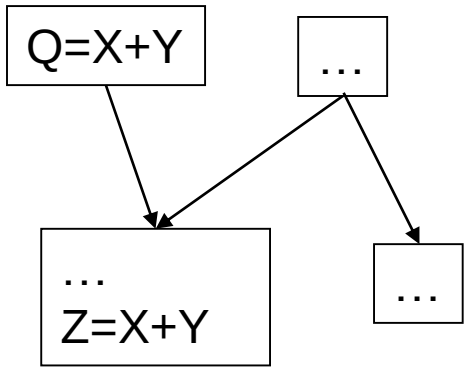
Eliminarea variabilelor de inductie

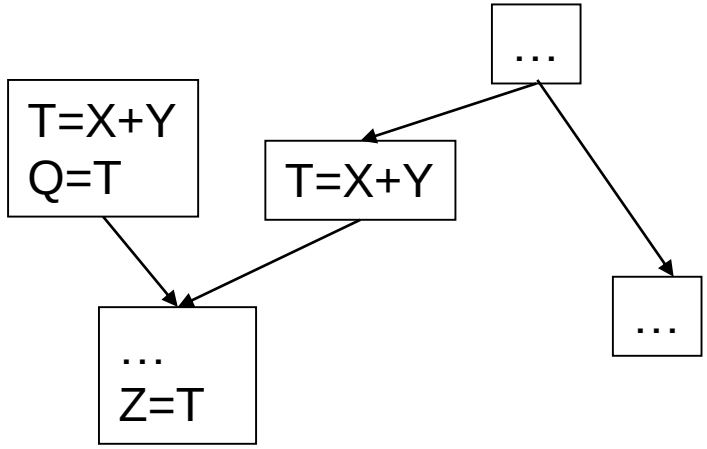
- Compunerea functiilor liniare e liniara
- O variabila tip J=(I,a,b)
  - In preheader: J = I * a + b (dupa definirea lui I)
  - In bucla: J = J + a
- Exercitiu - transformarea

Backup slides

Analize de flux complexe

- PRE – Eliminarea redundantei partiale
  - Include eliminarea subexpresiilor comune
  - Include scoaterea invariantilor in afara buclei
  - Implementare: Eliminarea muchiilor critice urmat de un set complex de analize de flux

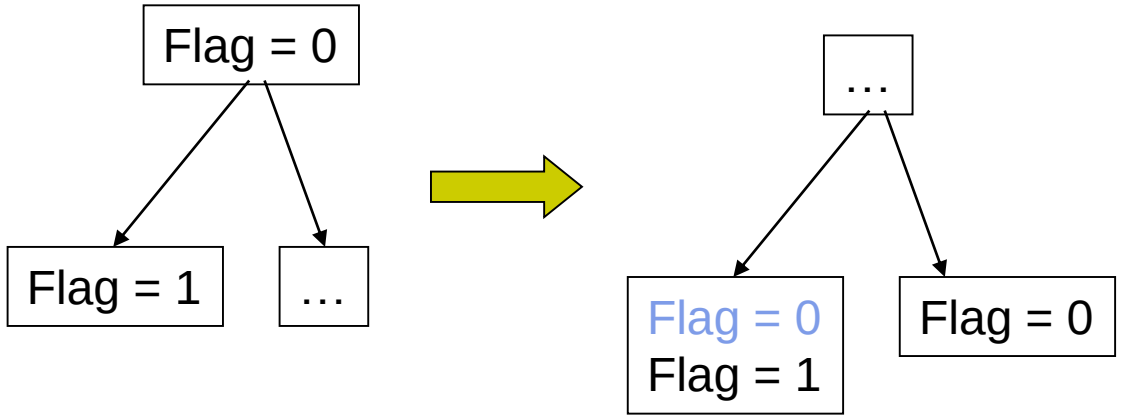




Analize de flux complexe

- “Partial dead code” → “True dead + True live”
- Determinarea punctului de inserare a instructiunilor

```
Flag = 0
If (..)
   {
   Flag = 1
   …
   }
Else { …}
```

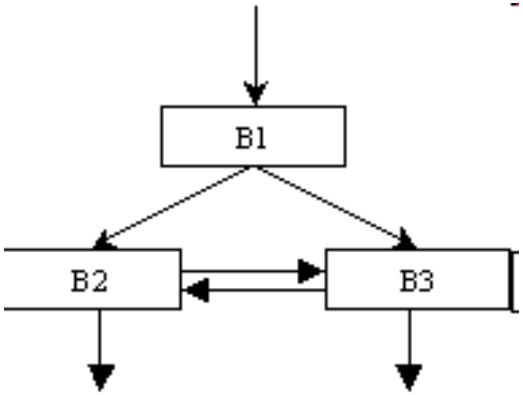


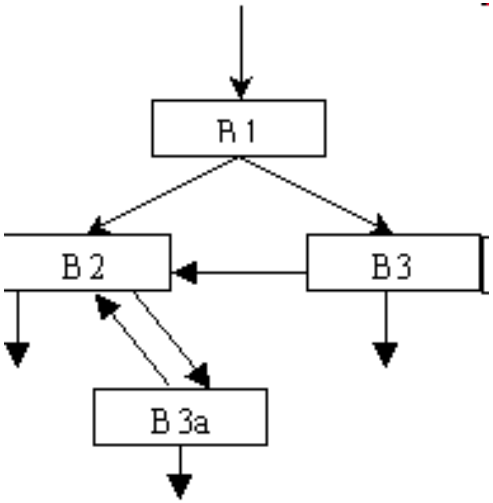
Analiza pe regiuni

- Alternativa la algoritmul iterativ
- Regiune – secventa de noduri dominate de un nod “header”
  - Daca m,n,h sunt noduri, n e in regiune, h e header, h dom n, h dom m, si exista cale m→n, atunci m e in regiune.
- Tipuri de regiuni
  - Aciclice (noduri = alte regiuni)
  - Bucle naturale (un arc inapoi)
  - Regiuni improprii

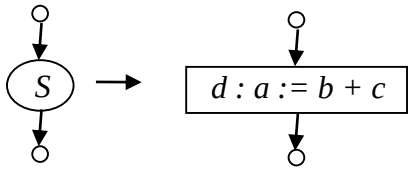
Node splitting

- Regiuni improprii – componente tare conexe ce nu sunt bucle naturale.
- Duplicarea nodurilor cu mai multi predecesori.
- Ce noduri duplicam?
- O euristica buna: “Making Graphs Reducible with Controlled Node Splitting”, Janssen & Corporaal



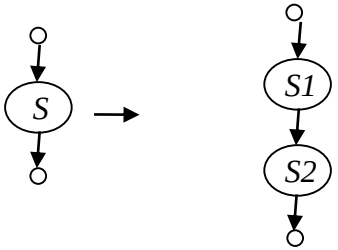


Analiza pe regiuni (RD)



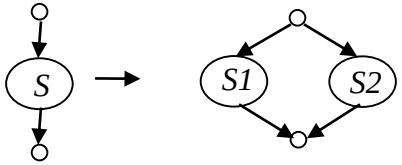
gen[S] = {d}

kill [S] = {orice def a:=...}



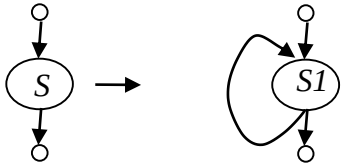
gen [S] = gen [S2] ∪ (gen [S1] - kill [S2])

kill [S] = kill [S1] ∪ kill [S2]



gen [S] = gen [S1] ∪ gen [S2]

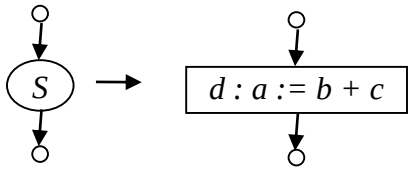
kill [S] = kill [S1] ∩ kill [S2]



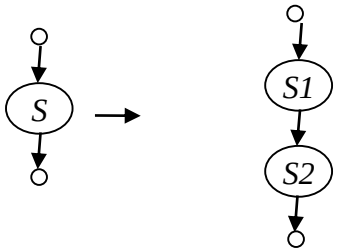
gen [S] = gen [S1]

kill [S] = kill [S1]

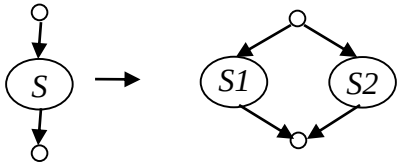
Analiza pe regiuni (RD)



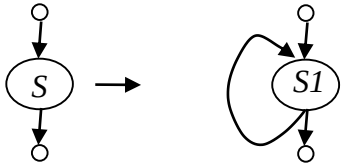
out [S] = gen [S] ∪ (in [S] - kill [S])



in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]



in [S1] = in [S]
in [S2] = in [S]
out [S] = out [S1] ∪ out [S2]



in [S1] = in [S] ∪ out [S1]
out [S]= out [S1]