Compilatoare

Optimizari

Analiza Fluxului de Control

Optimizari

- Optimizarea – transformarea unui cod pentru a-l face mai bun:
  - Timp de executie
  - Memorie ocupata
  - Putere consumata
- Consistenta semantica – nu trebuie schimbat rezultatul sau efectele laterale
  - Exceptie – codul care e din start gresit (‘garbage in – garbage out”). De ex., overflow-ul pe integer are valoare nedeterminata in C.
- Optimizarea e domeniul unde se face cercetare in ziua de azi (in teoria compilatoarelor). Scanarea, parsarea, analiza semantica si generarea de cod (neoptimizat) sunt bine intelese si in general destul de directe/usoare.

Calitatile unui compilator

- Performanta codului rezultat
  - Ideal: cel putin la fel de buna ca a unui programator in limbaj de asamblare.
- Facilitatile de limbaj implementate (Compatibilitate cu ANSI C, gcc, extensii de limbaj)
- Compilatoarele sunt programe!
  - Stabilitate, timp de compilare, memoria necesara; performanta predictibila.
  - Nu toate optimizarile sunt folosite in practica; pot fi dificil de implementat, cu un cost mare ca timp/memorie, sau aduc castiguri mici.

Ce se poate optimiza

- In codul intermediar
  - Inlocuirea calculelor redundante cu valorile deja calculate
  - Mutarea instructiunilor in locuri executate mai rar
  - Specializarea codului general
  - Detectia si eliminarea codului nefolosit
- In codul obiect
  - Inlocuirea unei operatii costisitoare cu unele mai simple
  - Inlocuirea unei secvente de instructiuni cu unele mai puternice
  - Ascunderea latentei, folosirea paralelismului din arhitecturi
  - Folosirea eficienta a resurselor (registri, cache)

Selectia optimizarilor

"High-quality optimization is more of an art than a science"

- Performanta rezultatului depinde de tipul si ordinea executiei optimizarilor.
  - Un pas de optimizare poate crea sau distruge oportunitati pentru pasul urmator.
  - Optimizarile se pot suprapune (Ex: numerotarea valorilor gaseste subexpresii comune)
- Nu este nevoie de rezultatul perfect. Doar unul bun.
  - Probleme de optimizare sunt NP – majoritatea algoritmilor folositi sunt euristici. Se pot gasi in general cazuri particulare cand o optimizare produce cod mai prost; totusi, pe medie, o optimizare imbunatateste performanta
- Optimizam codul cel mai frecvent intalnit, dar trebuie mentinuta corectitudinea si pe cazurile 'rare' (*memcpy vs. memmove*)

Limitarile optimizarilor

- Codul eficient nu depinde doar de compilator, ci mai ales de programator
  - Nici un compilator nu va inlocui BubbleSort cu Quicksort!!
- In termeni de complexitate (O(x)), in general un compilator poate imbunatati doar factorii constanti (dar si acestia sunt importanti)

Lasati compilatorul sa optimizeze!

- Optimizari la nivel de functie. Calcule cu variabile locale. Acces la elementele tablourilor. Propagarea constantelor. Subexpresii comune. Strength reduction.
- Alocarea variabilelor in registre. Selecţia si planificarea instrucţiunilor. Optimizarea salturilor.
- Functii de biblioteca eficiente (ex: STL)
- Scrieti cod cat mai simplu, structurat.
  - Folosirea cuvantului cheie 'register' e de mult timp inutila.
  - Folosirea de pointeri in loc de array poate duce chiar la rezultate mai proaste.
  - Folosirea de 'short' in loc de 'int' pt. variabile scalare/indici de bucla poate produce rezultate mai proaste.

Ce poate face programatorul

- #1: Reducerea complexitatii algoritmilor!
- Unele optimizari nu sunt larg implementate
  - Variabile referite prin pointeri (analiza de alias). Copierea valorilor in variabile locale?
  - Analiza interprocedurala. Inlining, efecte laterale.
  - Transformari de cicluri (cache, paralelizare)
- Date despre executie; profiling.
- Ajutor pt. compilator prin specificarea unei semantici mai stricte (const, restrict, static).
- Optimizari la nivelul intregului program.

Mai departe in curs

- Optimizari la nivel de bloc
- Optimizari la nivel de functie
- Optimizari de nivel scazut (alocarea registrilor, planificarea instructiunilor).

Daca e timp

- Optimizari de bucle si paralelizarea codului.
- Optimizari la nivel de program (alias,inlining)

Referinta: Steven Muchnick, *Advanced Compiler Design Implementation*

CPL IR crunch

Cum optimizam

- Pentru a putea face optimizari, primul pas e sa facem analize
  - Flux de control – care este structura programului
  - Flux de date – cum utilizeaza programul variabilele / valorile
  - Alias – ce obiecte sunt referite de pointerii din program
  - De dependenta – cum depind (in cadrul unei bucle) referirile la acelasi obiect
  - De inductie – cauta variabilele care se modifica in bucla intr-o maniera predictibila, descopera forma spatiului de iteratie
  - Etc.

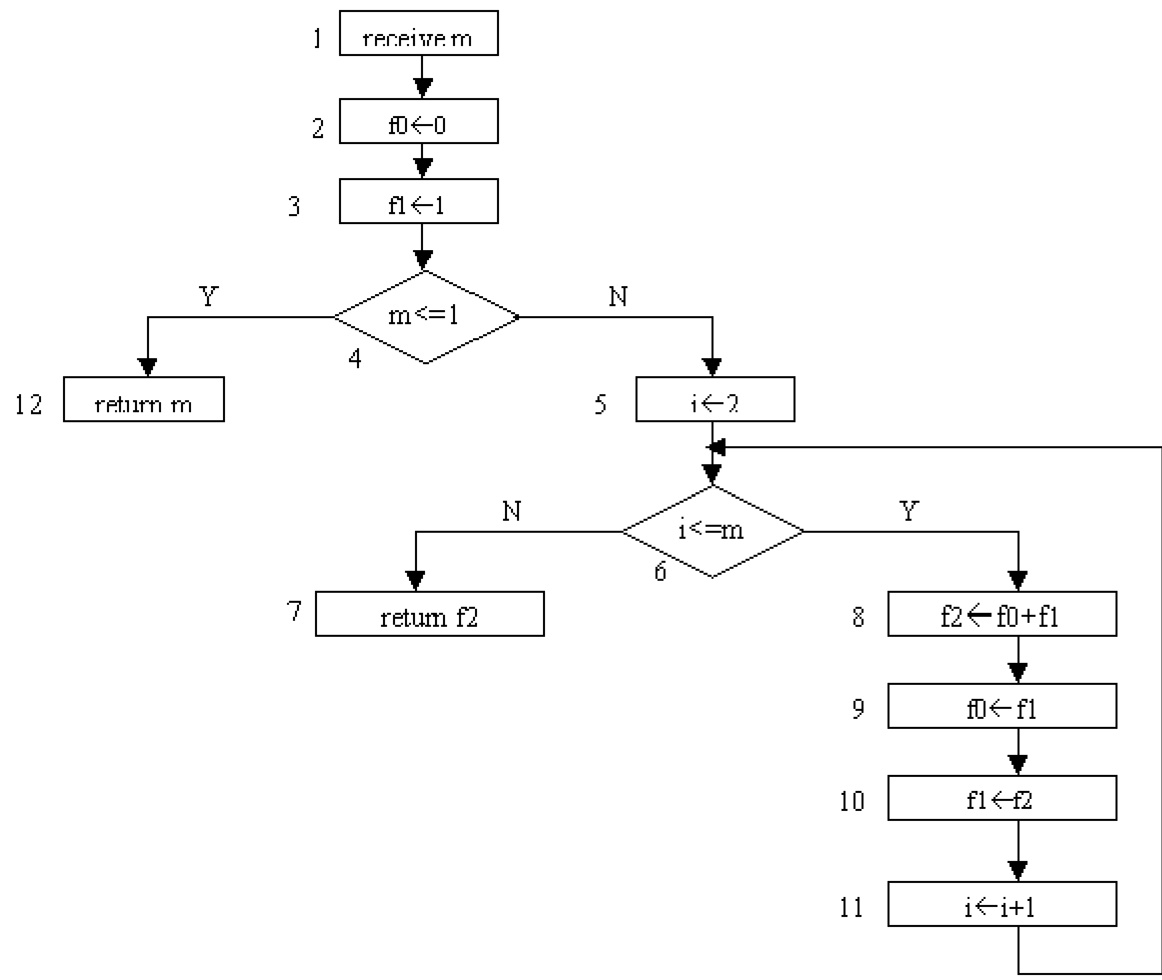
CPL IR crunch

Control Flow Analysis

- Descopera structura fluxului de control
  - Poate fi evidenta sau nu in programul sursa
  - Goto, exceptii
- Basic block – secventa de instructiuni cu o singura intrare si o singura iesire
  - Executia primei instructiuni garanteaza executarea tuturor instructiunilor din bloc.
- Prima instructiune – 'leader'. Basic block – secventa dintre doi leaderi.

CPL
IR crunch

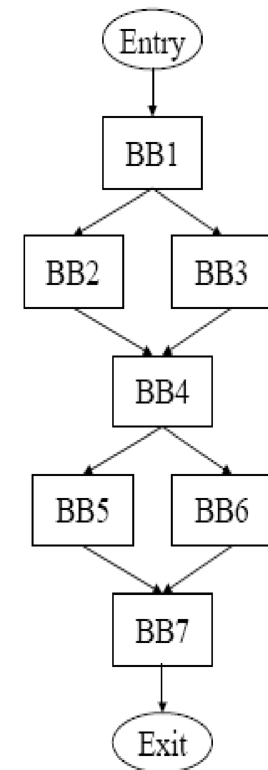
Control Flow Graph



1 receive m
2 f0 <- 0
3 f1 <- 1
4 if m <= 1 goto L3
5 i <- 2
6 L1: if i <= m goto L2
7 return f2
8 L2: f2 <- f0+f1
9 f0 <- f1
10 f1 <- f2
11 i <- i+1
12 goto L1
13 L3: return m

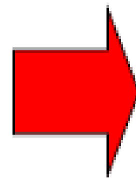
Control Flow Graph

- Graf orientat
- Fiecare nod e un basic block
- Exista arc intre 2 noduri BB1 si BB2 daca BB2 poate urma imediat dupa BB1, in cursul executiei programulu
- 2 noduri “false” Entry&Exit
- Apeluri de functii? exit()? Exceptii?



CFG – exemplu, If

```
x = z – 2;
y = 2 * z;
if (c) {
    x = x + 1;
    y = y + 1;
}
else {
    x = x – 1;
    y = y – 1;
}
z = x + y
```



B1

```
x = z – 2;
y = 2 * z;
if (c) B2 else B3
```

then (taken) → B2

```
x = x + 1;
y = y + 1;
goto B4
```

else (fallthrough) → B3

```
x = x – 1;
y = y – 1;
```

B4

```
z = z + y
```

Impartirea in blocuri

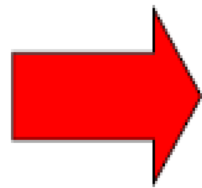
- Care sunt basic block-uri – pe secventa de mai jos?

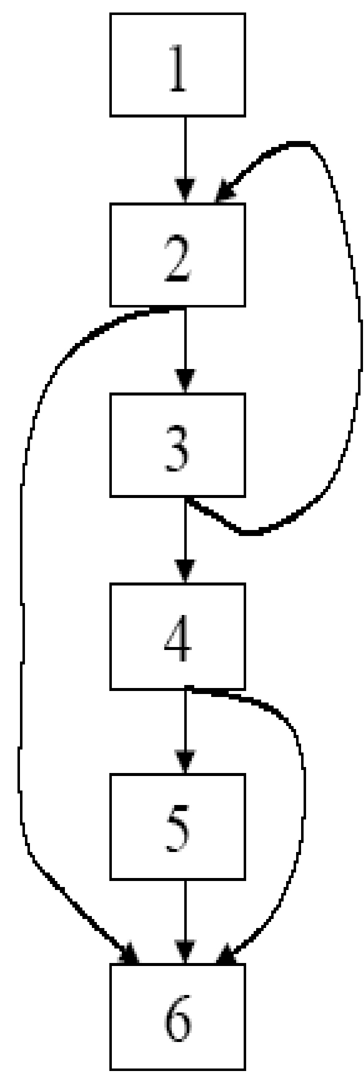
L1: r7 = load(r8)
L2: r1 = r2 + r3
L3: beq r1, 0, L10
L4: r4 = r5 * r6
L5: r1 = r1 + 1
L6: beq r1 100 L2
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: r7 = r7 & 3
L10: r9 = load (r3)
L11: store(r9, r1)

- Cum arata CFG-ul?

Impartirea in blocuri

1	L1: r7 = load(r8)
2	L2: r1 = r2 + r3 L3: beq r1, 0, L10
3	L4: r4 = r5 * r6 L5: r1 = r1 + 1 L6: beq r1 100 L2
4	L7: beq r2 100 L10
5	L8: r5 = r9 + 1 L9: r7 = r7 & 3
6	L10: r9 = load (r3) L11: store(r9, r1)



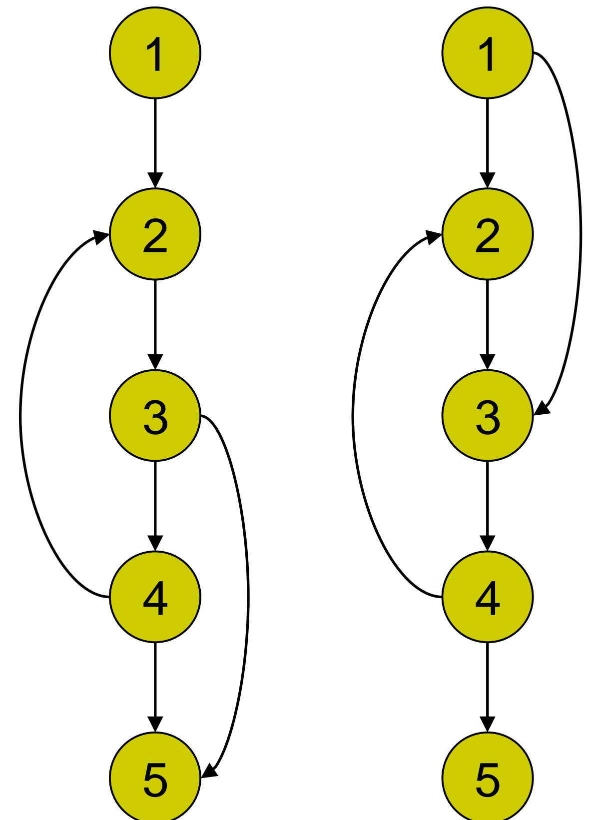


Structura, pe CFG

- ‘code unreachable’ – daca nu are predecesori
- ‘if’ – nod cu 2 succesori.
- Bucla – componenta tare conexa.
  - Ne intereseaza in general buclele “naturale” (cu un singur punct de intrare in bucla

Bucle naturale

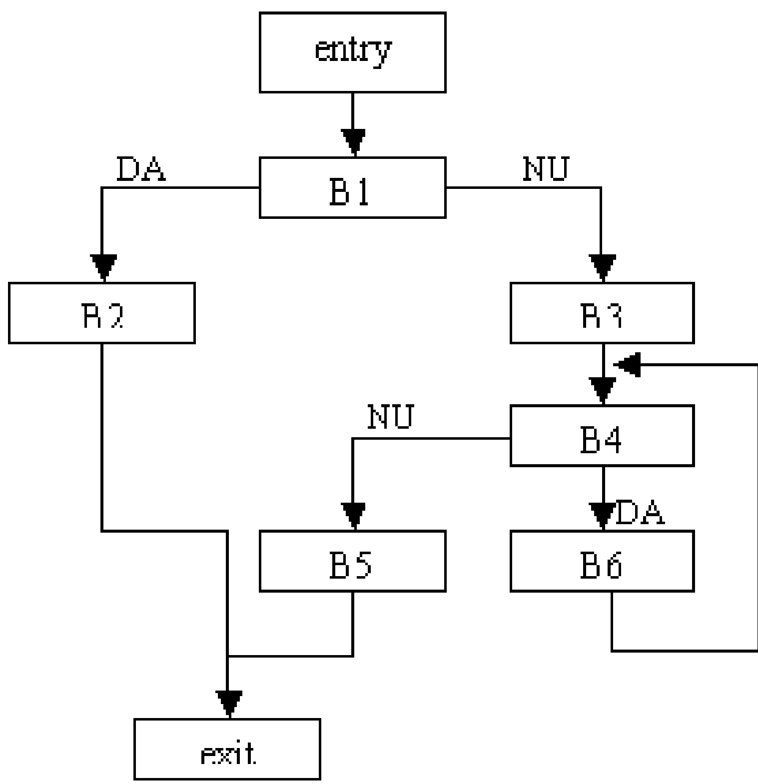
- Nu orice “circuit” in CFG e ‘bucla naturala’
  - O singura intrare
  - Arcele formeaza cel putin un circuit
- Ne intereseaza daca executia unui bloc garanteaza executia altui bloc

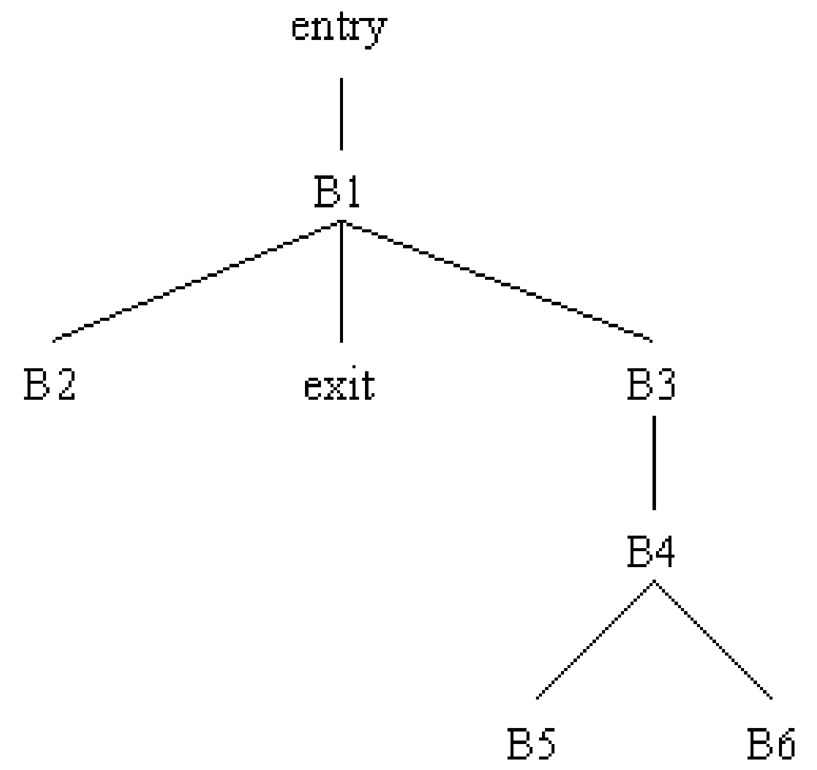


Dominare, post-dominare

- X dom Y daca orice cale de la Entry la Y contine X
  - Proprietati
    - X dom X
    - X dom Y si Y dom Z => X dom Z
    - X dom Z si Y dom Z => X dom Y SAU Y dom X
- Y pdom X daca orice cale de la X la Exit contine Y (Y dom X pe "cfg-ul inversat")
- X idom Y daca Xdom Y, X!=Y si nu exista Z!=X,Y a.i X dom Z si Z dom Y
  - "idom" creaza o relatie de arbore intre noduri – "arborele de dominare"

Arbore de dominare





CPL IR crunch

Algoritm

- Initializare
  - »Dom(entry) = entry
  - »Dom(everything else) = all nodes

- Calcul iterativ

```
»while change, do
    change = false
    for each BB (except entry)
       tmp(BB) = BB + {intersect Dom(P), ∀ P, predecesor al lui BB }
       if (tmp(BB) != dom(BB))
          dom(BB) = tmp(BB)
          change = true
```

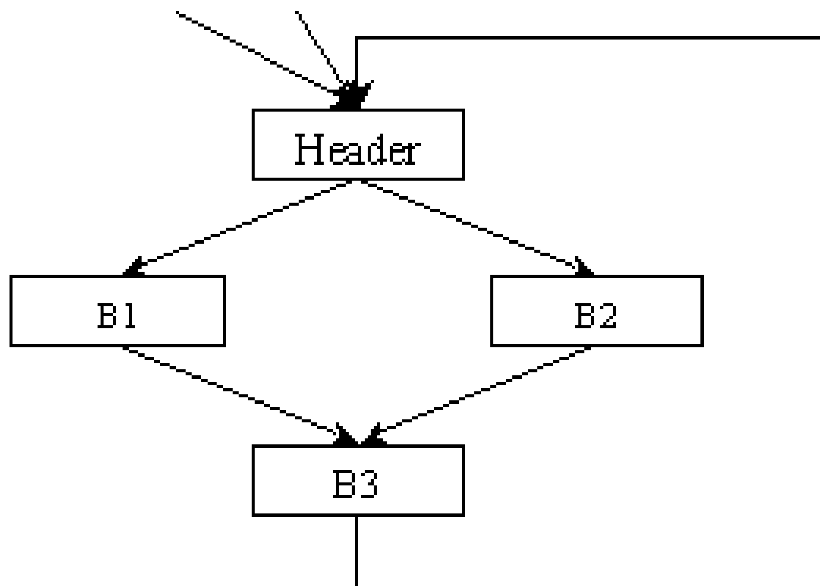
CPL IR crunch

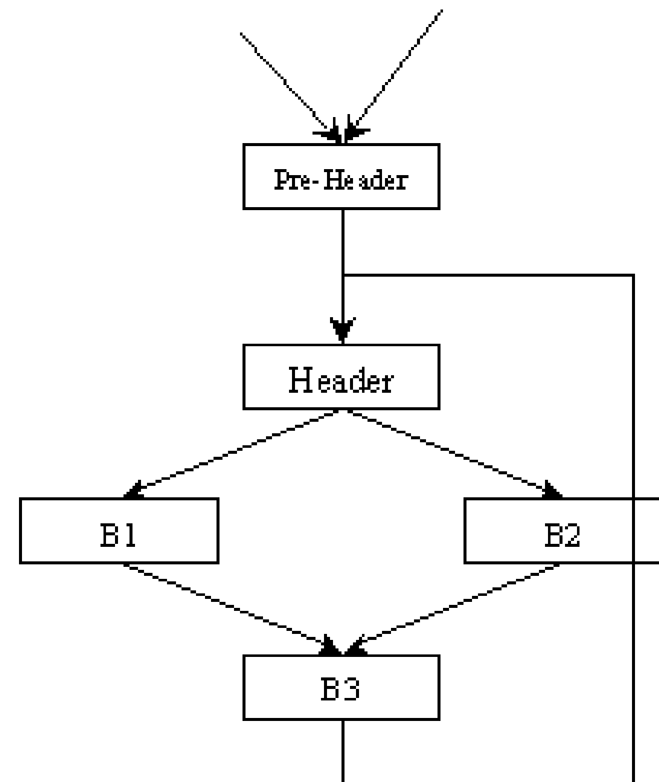
Detectia Buclelor Naturale

- Arc inapoi: un arc al carei “destinatie” domina “sursa”.
- Header: “destinatia” unui arc inapoi, domina toate nodurile din bucla naturala
- Bucla Naturala – def. formala:
  - Determinata de un arc inapoi n → h
  - Nodul h : loop header; h dom n
  - Setul de noduri x, pentru care exista o cale in CFG P=x→…→n , ce nu include h
- Restul buclelor -> “irreducible regions”

Preheader

Pentru optimizari ce scot cod in afara buclei:

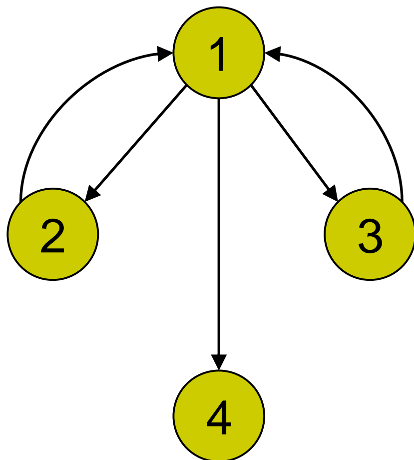


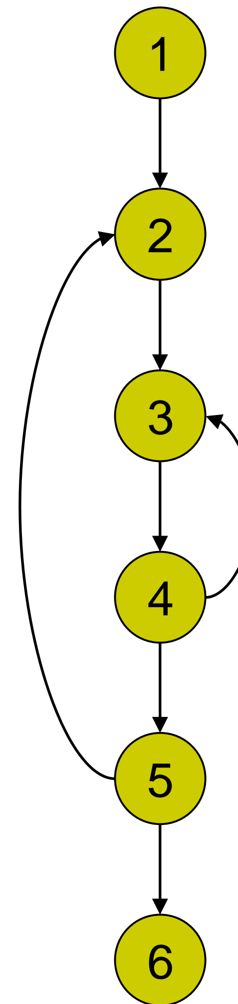


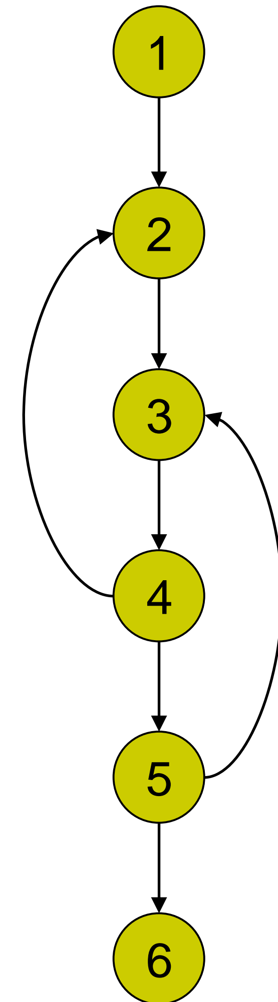
CPL IR crunch

Bucle imbricate

- Daca au headere diferite, sunt disjuncte sau imbricate
- Mai dificil daca au acelasi header
  - Se trateaza in general ca o singura bucla

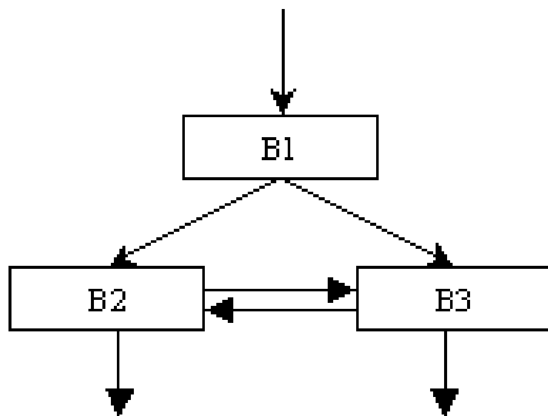


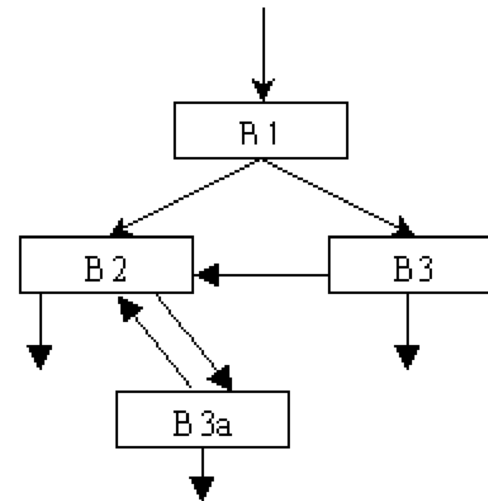




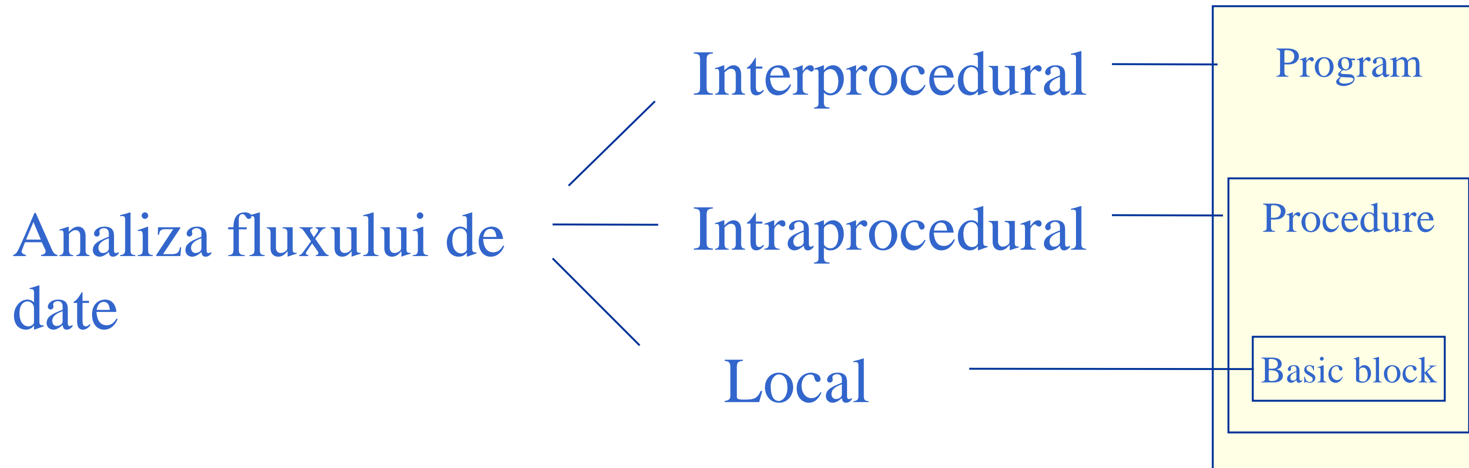
Regiuni improprii

- Node splitting
- Optimizari conservative
- Analize iterative





Analize si optimizari



- Se bazeaza pe analiza fluxului de control
- Determina cum sunt utilizate valorile in program ("fluxul de date")

Constant folding

- Evaluarea la momentul compilarii a expresiilor cu operanzi constanti;
- Daca intalnim o expresie de genul **10 + 2 * 3**, putem calcula rezultatul '16' in mod direct.
- Similar, "**if a > 0 goto L1 else goto L2**" poate fi inlocuita cu 'goto L1' daca stim ca a == 0
  1. inlocuim "a > 0" cu "true"
  2. o optimizare separata va elimina codul "unreachable"
- Pot aparea la nivelul codului intermediar:
  - A[3] → *(A + 3*sizeof(int)) → *(A+12)

Constant folding (2)

- Trebuie respectate regulile limbajului compilat (nu a limbajului in care e scris compilatorul!) si regulile procesorului destinatie (nu regulile procesorului care ruleaza compilatorul!)
  - De ex., daca facem ‘constant folding’ pe un tip ‘float’ si variabilele erau de fapt ‘double’ e gresit
  - Uneori, evaluarea unor expresii constante poate genera exceptii (overflow), trebuie avut grija la asta
  - ‘int’ pe procesorul destinatie poate fi pe 16 biti
- Uneori, ‘constant folding’ permite programatorilor sa foloseasca expresii in loc de constante (de exemplu in ‘case’ sau dimensiunea unui array). Acesta e exemplu de “zona gri” unde analiza semantica /optimizarea influenteaza analiza sintactica.

Propagarea constantelor

- Daca o variabila are valoare constanta, ea poate fi inlocuita cu constanta in toate expresiile unde apare
- Exemplu a=2; c=a+b; d=a+1;
- Propagarea constantelor poate crea oportunitati pentru constant folding (si reciproc)
- E foarte importanta in special pe arhitecturi RISC fiindca muta constantele unde sunt folosite, reducand nr. de registri folositi si uneori si nr. de instructiuni
  - De ex. MIPS are un mod de adresare [reg+offset], dar nu [reg+reg]. Propagarea constantelor in acest caz poate elimina o instructiune 'add'.
- Dezavantaje
  - pentru constante mari creste dimensiune codului;
  - unele arhitecturi nu pot incarca orice constanta dintr-o singura instructiune.

CPL IR crunch

Simplificari algebrice si reasociere

- Pe baza proprietăţilor algebrice ale operaţiilor aritmetice şi logice se pot elimina instrucţiuni care nu au nici un efect sau se pot simplifica expresii.
- Exemple de instrucţiuni care pot să fie eliminate:
  - x = x + 0
  - x = x * 1
- Exemple de instrucţiuni care pot să fie simplificate
  - x = x * 0   sau x = 0/x => x = 0
  - x = b || false => x = b
  - x = b && true => x = b
- Reasocierea poate expune oportunitati de ‘constant folding’
  - A = 15 + b - 10 -> A = b + (15 – 10)

Strength reduction (reducerea ‘puterii’ operatorilor)

- Aceasta optimizare consta in inlocuirea unor operatori ‘scumpi’ cu unii mai ‘ieftini’
- Care e mai eficient?
  - i*2 = 2*i = i+i = i << 1
  - i/2 = (int)(i*0.5) = i>>1
  - i % 16 = i & 15
  - 0-i = -i
  - f*2 = 2.0 * f = f + f
  - f/2.0 = f*0.5
- Combinare de strength reduction si simplificari algebrice: i*9 = i<<3 +i.

Strength reduction (2)

- Strength reduction se face de multe ori ca optimizare de bucla

```
i=0;
while (i < 100) {
    arr[i] = 0;
    i = i + 1;
    }
```

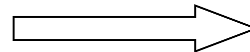
```
ptr=arr;
while (ptr < arr+400) {
        *ptr = 0;
        ptr = ptr + 4;
    }
```

- Inmultirea implicita cu 4 devine adunare.
- Indexul i variaza liniar? Analiza variabilelor de inductie

Propagarea copierilor

- Similara cu propagarea constantelor, dar pentru valori neconstante:

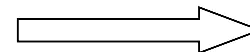
```
tmp2 = tmp1 ;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3 ;
tmp5 = tmp3 * tmp2 ;
c = tmp5 + tmp4 ;
```



```
tmp2 = tmp1 ;
tmp3 = tmp1 * tmp1;
tmp4 = tmp3 ;
tmp5 = tmp3 * tmp1 ;
c = tmp5 + tmp3 ;
```

- Se poate si 'invers'

```
tmp1 = Call _Binky ;
a = tmp1;
tmp2 = Call _Winky ;
b = tmp2 ;
tmp3 = a * b ;
c = tmp3 ;
```



```
a = Call _Binky ;
b = Call _Winky ;
c = a * b ;
```

Eliminarea codului ‘mort’

- Daca rezultatul unei instructiuni nu e folosit, instructiunea poate fi considerata ‘moarta’ si poate fi stearsa.
- Cu grija!
  - tmp1 = call print; // functie cu efecte laterale
  - tmp1 = tmp2 / 0; // exceptii
- ‘cod mort’ poate sa apara pe sursa originala, dar e mai probabil sa apara ca urmare a optimizarilor
- ‘dead code’ – se executa, dar nu are efect vs. ‘unreachable code’ – nu se executa niciodata

CPL IR crunch

Eliminarea subexpresiilor comune

- Doua operatii sunt comune daca produc acelasi rezultat – e mai eficient sa-l calculam o data si sa-l referim direct a doua oara
  - (tine un registru in plus in viata….)

```
x = 21 * -x;
y = (x*x)+(x/y);
z = (x/y)/(x*x);
```

- Translatare directa:

```
tmp1 = 21 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
```

- Dupa propagarea constantelor, eliminarea subexpresiilor comune:

```
tmp2 = -x ;
x = 21 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
z = tmp5 / tmp3 ;
```

Exemplu de aplicare a optimizarilor

Început	Transformări algebrice	Copy propagation	Constant folding
a = x ** 2	a = x * x	a = x * x	a = x * x
b = 3	b = 3	b = 3	b = 3
c = x	c = x	c = x	c = x
d = c * c	d = c * c	d = x * x	d = x * x
e = b * 2	e = b << 1	e = 3 << 1	e = 6
f = a + d	f = a + d	f = a + d	f = a + d
g = e * f	g = e * f	g = e * f	g = e * f

Eliminare subexpresii	Copy propagation	Eliminare cod inutil
a = x * x	a = x * x	a = x * x
b = 3	b = 3	
c = x	c = x	
d = a	d = a	
e = 6	e = 6	
f = a + d	f = a + a	f = a + a
g = e * f	g = 6 * f	g = 6 * f

Numerotarea valorilor

- Imbina mai multe optimizari (propagarea copierilor, eliminarea subexpresiilor comune)
- Se atribuie câte un număr distinct pentru fiecare valoare calculată.
- Două expresii Ei şi Ej au numerele asociate egale numai dacă se poate demonstra că cele două expresii sunt egale pentru orice operanzi.
- Două expresii sunt redundante dacă acelaşi operator se aplică asupra aceleaşi valori asociate.
- Pentru a realiza corespondenţa dintre variabile, constante şi valori calculate şi valorile numerice asociate se utilizează o tabelă hash. Pentru variabile şi constante cheia poate să fie chiar şirul de caractere respectiv.

Numerotarea valorilor(alg)

pentru fiecare expresie e de forma rezultate = op1 oper op2 din bloc

se obţin valorile numerice pentru op1 şi op2

se construieşte o cheie pentru tabela hash pe baza operatorului şi a valorilor numerice ale operanzilor

dacă cheia este deja în tabelă

se înlocuieşte expresia cu o operaţie de copiere

altfel

se introduce cheia în tabelă

se asociază cheii o nouă valoare numerică

se înregistrează valoarea numerică a rezultatului

CPL IR crunch

Numerotarea valorilor (ex)

$a = b + c$	$a^3 = b^1 + c^2$	$a = b + c$
$b = a - d$	$b^5 = a^3 - d^4$	$b = a - d$
$c = b + c$	$c^6 = b^5 + c^2$	$c = b + c$
$d = a - d$	$d^5 = a^3 - d^4$	$d = b$

- Se constată că cele utilizări ale expresiei b + c conduc la valori diferite spre deosebire de expresia a – d care are aceeaşi valoare în cele două apariţii.
- Algoritmul se poate extinde ţinând de exemplu cont de faptul că pentru operaţii comutative valoarea numerică asociată ar trebui să fie aceeaşi indiferent de ordinea operanzilor.
- O altă aplicaţie a numerelor asociate expresiilor poate să fie reprezentată de identificarea unor operaţii care nu au nici un efect, aşa cum este de exemplu x * 1.

CPL IR crunch

Numerotarea valorilor (alg2)

pentru fiecare expresie e de forma rezultate = op1 oper op2 din bloc
    se obţin valorile numerice pentru op1 şi op2
    dacă operanzii sunt constante
        se evaluează şi se înlocuiesc referinţele ulterioare cu
        rezultatul obţinut
    dacă operaţia este o identitate
        înlocuieşte cu o copiere
    dacă operaţia este comutativă
        sortează operanzii în ordinea numerelor asociate
    se construieşte o cheie pentru tabela hash pe baza operatorului şi a
        valorilor numerice ale operanzilor
    dacă cheia este deja în tabelă
            se înlocuieşte expresia cu o operaţie de copiere
            se înregistrează valoarea numerică a rezultatului
    altfel
            se introduce cheia în tabelă
            se asociază cheii o nouă valoare numerică
            se înregistrează valoarea numerică a rezultatului