INF5110 – 9. mai 2012

Stein Krogdahl, Ifi, UiO

1. Tilleggsnotat fra bok av Aho, Sethi og Ullman
(legges ut på undervisnings-planen)
2. Eksempel på global data-analyse

Ellers står igjen i kurset (tirsdag 15/5):
1. Litt om behandling av generiske klasser etc.
2. Oppgaver til kodegenerering

Onsdag 16/5: Ikke undervisning!
Tirsdag 22/5: Gjennomgang av fjorårets eksamen etc.

Maskinen det oversettes til i notatet

Detaljer ikke viktige, bare eksempel på instruksjoner

- To-adresse-instruksjoner:

op source dest

source og *dest* angir et register eller en lagercelle (og i ett tilfelle en verdi)

hva

mange forskjellige ulike former, se neste foil

ADD a b
SUB a b

Merk: Beregner b – a og legger svaret i b.

OG: I maskinen skissert på s. 12 i Louden er det omvendt!

MUL a b
.....
GOTO l
+ Betingete hopp
+ Prosedyrekall
++

Complex Instruction Set Computer

Reduced Instruction Set Computer

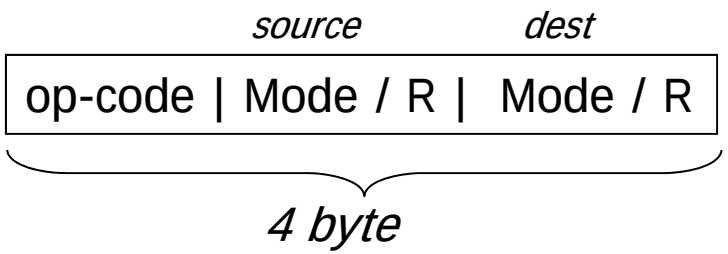
Mer er en CISC-maskin enn en RISC-maskin

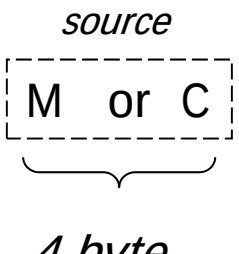
Instruksjonsformat og adressemodi – del 1

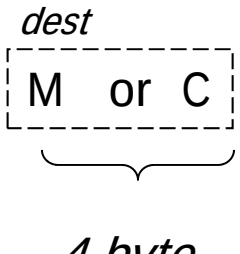
Kan se dette som eksempler på typiske addresseringsformer

Vi bruker instruksjonens lengde som "kost" av en instruksjon

Grunnkost for en fire-bytes instruksjon er 1



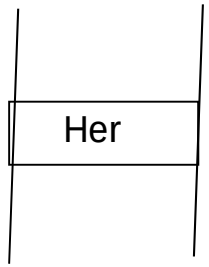




\+ Tilleggskost for hver adressering

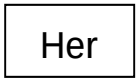
Adresseringsmodi:

1 Absolutt: M



Bare om M eller C

0 Register: Ri



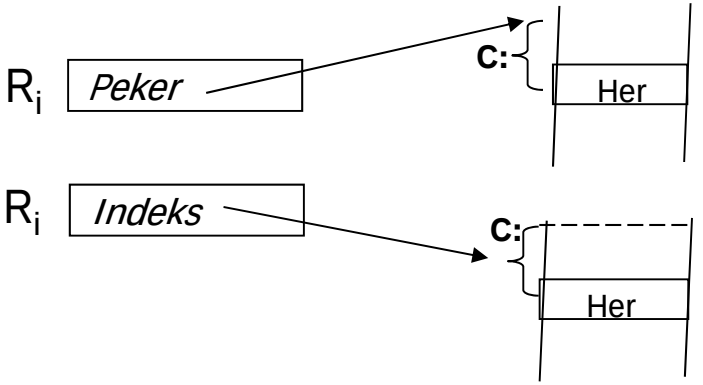
NB: Dette med hva en aksess til lageret "koster" er nå blitt veldig uforutsigelig pga. cacher etc.

1 Indeksert : C(RI)

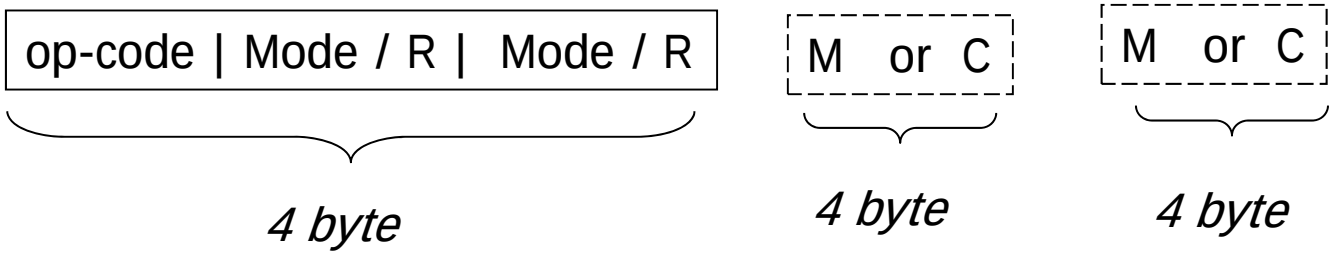
Grei på to måter:

(1) Til å la R inneholde en objekt-peker og la C være en kompilatorkjent relativadresse i objektet

(2) Til å la C være peker til en fastliggende array, og la R ha en indeks inn i denne

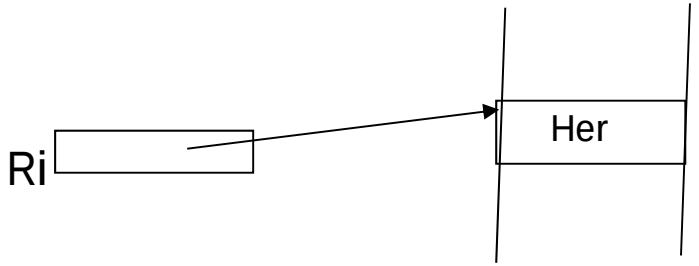


Instruksjonsformat og adressemodi – del 2



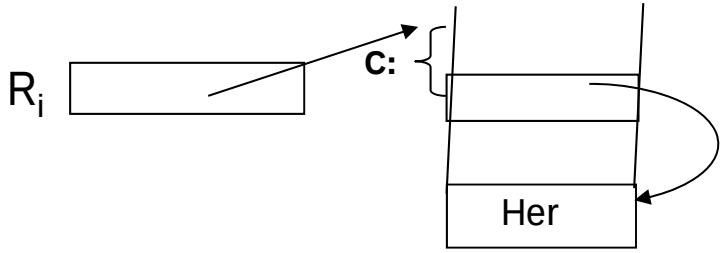
Adresseringsmodi:

0 Indirekte Register *R

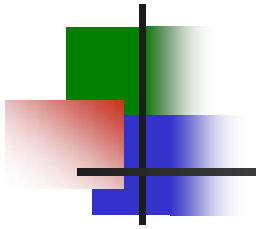


1 Indirekte *C(Ri)

Kan brukes til å hoppe to steg av gangen langs en linket liste, f.eks ved følging av "lang" access link



1 Literal #M bare for source (spesiell, men veldig brukbar!)



9.4 Basale blokker og Flyt-grafer

- En Flyt-graf er en graf der nodene er sekvenser av treadresse-instruksjoner (eller en annen type lavnivå kode)

- Nodene i grafen er kalt *Basale Blokker*
  - En basal blokk er en sekvens av treadresse-instruksjoner som er slik at:

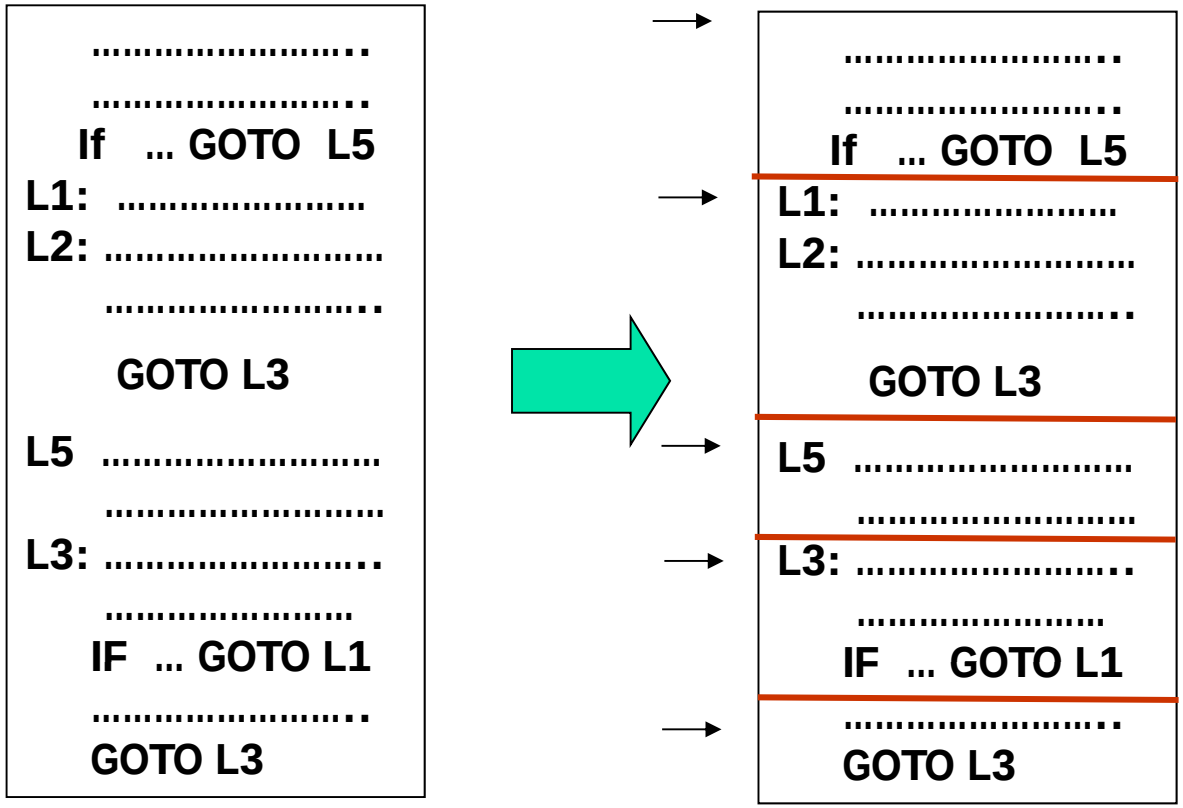
    Programkontrollen alltid kommer inn i første setning og går sekvensielt gjennom hver setning uten stopp eller sidesprang. Eneste unntak er at siste setning kan være et hopp – eventuelt et betinget hopp.

- Kantene i grafen representerer de mulige veier programflyten-kontrollen kan ta

- Innen en Basal Blokk er det lett å holde oversikt over hvor verdier etc. er, og lett å gjøre ”abstrakt interpretasjon” = ”statisk simulerig”

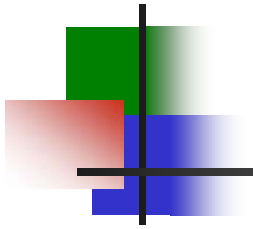
Eksempel på oppdeling i basale blokker

- Basale blokker: Fra og med en ”leder” fram til neste, eller slutt
- Algoritme for å finne alle ledere:
  - Første setning er leder
  - en ”goto i”, gjør setning ”i” til en leder
  - setninger etter ”goto ..” er ledere



Det er tydligvis ingen som går til L2.

Metodekall tenker vi ikke på. Kan behandles litt forskj. avh. av formål.



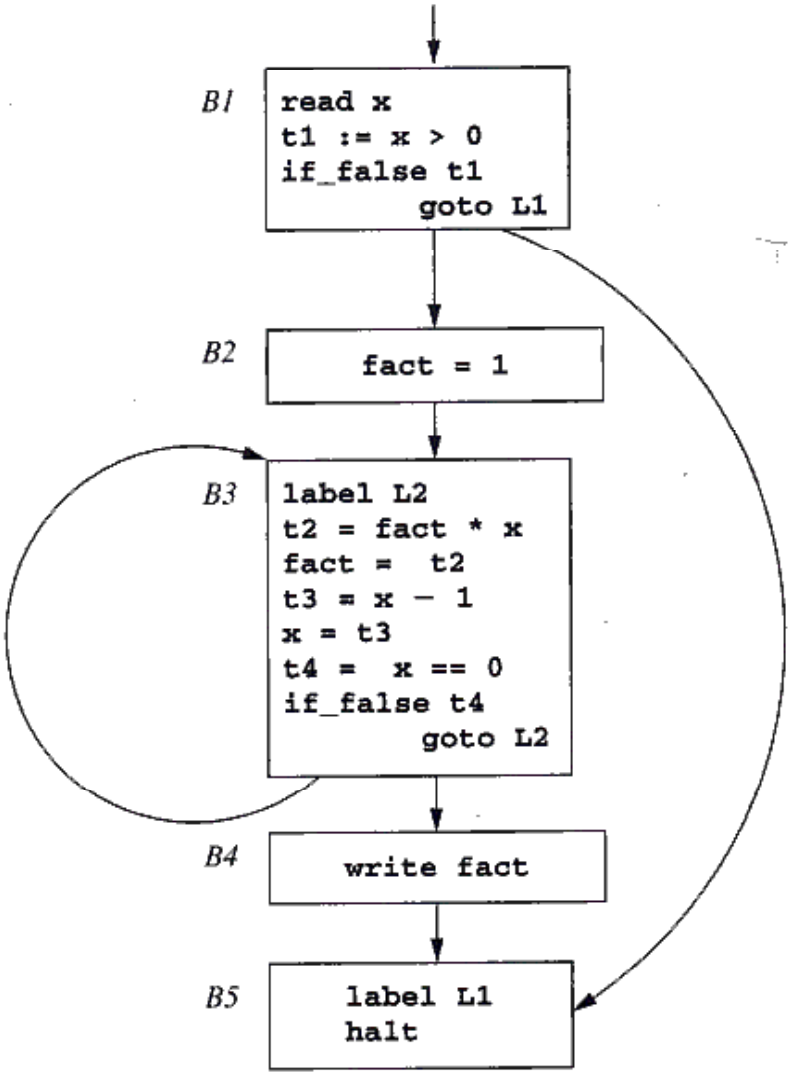
Flyt-graf

- Nodene i flyt-grafen er de basale blokkene med en initiell node (den første basale blokken i programmet).

- Det er en rettet kant fra blokk B_1 til B_2 hvis B_2 kan følge direkte etter B_1 i en eller annen utførelse. Det vil si at det enten er:
  - En betinget eller ubetinget goto fra siste setning i B_1til første setning i B_2 eller
  - B_2 følger direkte etter B_1 i programmet, og B_1 ender ikke i en ubetinget goto.

- Vi sier at B_1 er en *forgjenger* til B_2 og B_2 er en *etterfølger* til B_1

Flytgraf fra Louden 8.9
Oftest: En flytgraf for hver metode



En goto-setning eller if-goto-setning vil alltid være siste setning i sin basale blokk (men ikke alle basale blokker slutter slik!)

Metodekall kan passes inn i dette på litt forskjellig måter, avh. av grafens bruk:

- Danne egne basale blokker
- Kan ligge først i en basal blokk?

Kode kan analyseres/arbeides med (f.eks. til optimalisering) på tre nivåer:

1. Inne i én basal blokk
2. Flytgrafen for én metode ("globalt nivå")
3. Alle flytgrafene for hele programmet

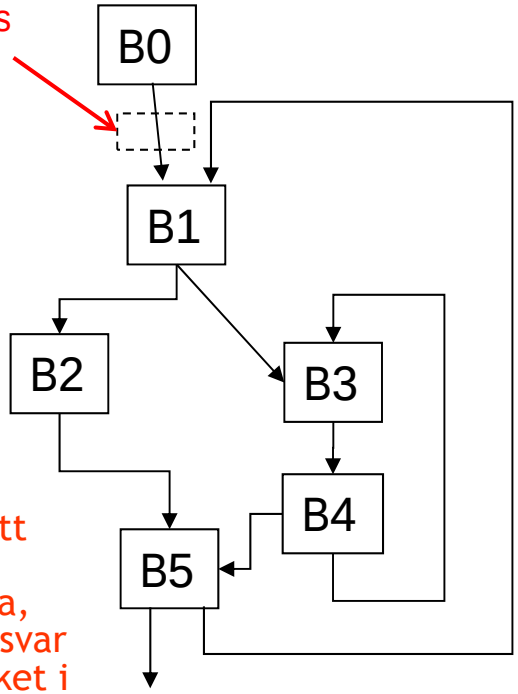
Løkker i flyt-grafer

- Typisk bruk, for eksempel :

  while (i<n) { i++; A[i] = 3*k; }

  - Kan vi flytte beregninger ut av løkka? Ja, beregning av ”3*k”
  - Kan kanskje holde mye brukte variable i registre mens vi er i løkka? F.eks. variablen ”i”

Ny basal blokk der vi kan putte ting som ”flyttes ut” av løkka, og gjøres på forhåndting (f.eks. k’=k*3)



Hvorfor vi *ikke* vil ha {B1,B2,B5} som lovlig løkke?:

Om vi da ville sjekke om en gitt variabel k ble forandret i løkka, kunne vi få feil svar om vi bare sjekket i B1, B2 og B5. Blokkene B3 og B4 må også undersøkes

En løkke er et utplukk L av noder slik at:

1) Alle-til-alle-vei: Dersom Bx $\in$L og By $\in$L, så går det en rettet vei fra Bx til By av lengde $\geq$ 1 *(også om Bx og By er samme node!)*

2) L har bare én ”inngang”: Det finnes bare én B$\in$L slik at Bn $\rightarrow$ B og Bn$\notin$L.

Begrunnelse for 2) er rent praktisk: Ett sted å initialisere løkka & Ett sted om vi skal flytte noe ”ut av løkka” (stiplet boks).

Eksempler:

{B3,B4} og {B1,B2,B3,B4,B5} er løkker

{B1,B2,B5} er ikke løkke (se forklaring til venstre)

Hva er ”liveness” (”i live”) ?

- Begrepet er *uavhengig* av basale blokker (Ikke klart i notatet)
- Defineres i 9.4 og brukes i 9.5
  - Terminologi:

```
a := x + y;
if (x<a) goto L;
```

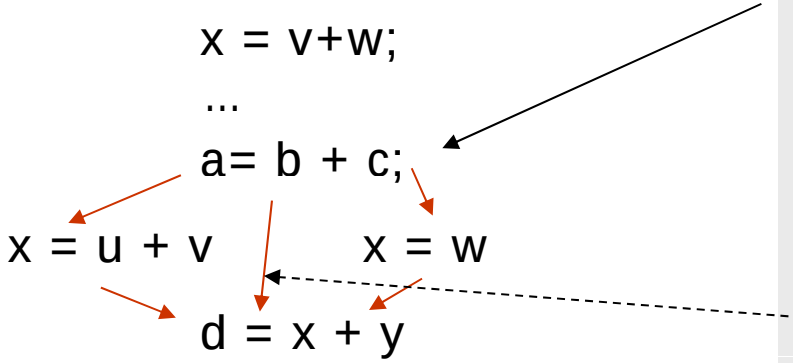
Her ”defineres” **a**, og ”brukes” **x** og **y**

Her ”brukes” **x** og **a**.

Intuitiv definisjon:

En variabel x er ”levende” (eller ”i live”) på et gitt sted i programmet dersom den verdien den der har kan bli brukt senere i *en eller annen utførelse.*

Teknisk definisjon av: Er ”x” levende på stedet ”i”:

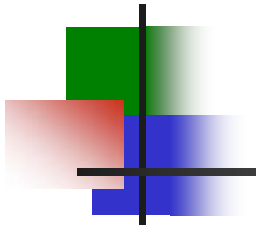


Stedet ”i” er denne TA-instruksjonen

Er ”x” i live her ?

Svaret er ”ja”, fordi det finnes en TA-setning ”j” som *bruker* ”x”, og *det er minst én eksekveringsvei fra ”i” til ”j” uten noen tilordning til* ”x”.

Definisjon: En variabel som ikke er ”levende” på et gitt punkt, sies å være ”død” på dette punktet (og dens verdien behøver da ikke lagres)



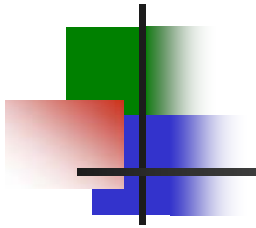
Andre def. som kan være interessante for optimalisering

Global dataflyt-analyse. Eksempler:

- Gitt en TA-instruksjon der x brukes:
  - Finn alle de tilordninger (definisjoner) der denne verdien på x kan være satt
- Gitt en tilordning der x blir satt:
  - Finn alle de steder der denne verdien av x kan bli brukt

Disse og liknende sammenhenger kan ”lett” bergenes ved en kompletterings-algoritme på de basale blokkene (omtrent som når vi finner First og Follow for grammatikker)

- Vi skal se på en slik algoritme

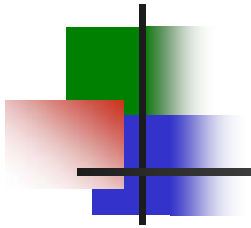


ASU, kap 9.5. Her ser man på:
Å generere ”lur” kode for én og én basal blokk

- Innen en basal blokk er det lett å holde orden på hvilke verdier som er i hvilke registre etc. ned gjennom blokka

- Filosofien for metoden vi ser på:
  - Mellom hver basal blokk sørger vi for:
    - Alle verdier av program-variable ligger i variablens lokasjon ”ute i hovedlageret”.
    - Vi antar også et TA-koden er laget slik at temporære variable ikke skal bære verdier fra én basal blokk til en annen. Temporære variable er altså døde ved begynnelsen og slutten av hver basal blokk

  - Det kan også hende at programvariable er døde ved slutten av en basal blokk.
    - Om vi skal få oversikt over dette må vi gjøre *global dataflytanalyse*
    - Men det gjør vi ikke her. Vi må derfor anta at alle *program-variable* er i live ved slutten av en basal blokk.
    - Og vi antar også at mellomkoden er laget slik at alle *temporære variable* er døde ved slutten (og derved ved begynnelsen) av hver blokk

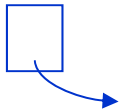


”Neste bruk” innen en basal blokk, og ”i live”

- Før man gjør kodegenerering for en basal blokk er det lurt å skaffe seg oversikt over bruk av variable (temporære og andre) i blokka:
  - En variabel-forekomst kan ha en ”neste-bruk” i blokka (derved ”i live”):
    - Def: Den verdien den her har blir brukt senere i samme basale blokka.
    - Da kan det f.eks. være lurt å la den bli værende i et register
  - En variabel-forekomst som ikke har noen ”neste-bruk”, kan fremdeles være ”i live”:
    - Da: Verdien i variabelen blir verken brukt eller gitt ny verdi senere i blokka
    - Men denne verdien kan bli (eller blir helt sikkert) brukt i andre blokker senere.
    - Dette gjelder i vår setting bare program-variable (ikke temporære)
  - En variabel-forekomst er død.
    - Gjelder temporære variable som ikke blir referert mer i blokka
    - Gjelder alle variable som blir gitt ny verdi lenger ned i blokka, og som ikke brukes før det.

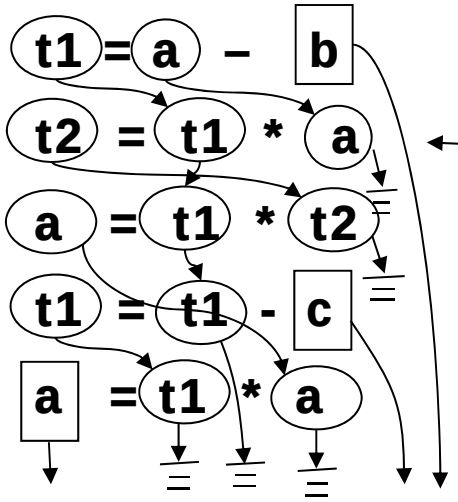
Eksempel på informasjon om ”neste bruk” og ”i live” innen en basal blokk

: Angir at den har ”neste bruk” i blokka (og hvor, men dette brukes ikke av vår algoritme). Slike er ”i live”



: Angir at den er ”i live”, men uten noen ”neste bruk” i blokka

: Angir at den er ”død”



a er ”død” her siden den får ny verdi innen denne blokka og ikke blir brukt før det

t1 er død her, siden vi antar at temporære bare brukes innen samme blokka

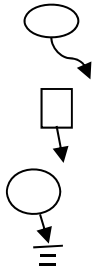
Vi antar altså: Programvariablene **a, b, c** er i live etter blokka.

Kommentarer:

1. Global ”dataflyt-analyse” finner de variable som faktisk er i live etter en basal blokk.
2. Temporære brukes også ofte på tvers av basale blokker (for eksempel etter flytting)

Algoritme for å finne informasjon om ”neste bruk” og ”i live”

Vi har en tabell T over alle variable i blokka, der hver variabel kan merkes som:



1) ”i live”, og har en angitt ”neste bruk” i blokka
2) ”i live”, men uten ”neste bruk” i blokka
3) ”død” (og dermed ingen ”neste bruk”)

Initialisering av tabellen T:

De variablene som er ”i live” ved slutten av blokka (her: program-variablene) merkes med 2, resten (de temporære) merkes 3

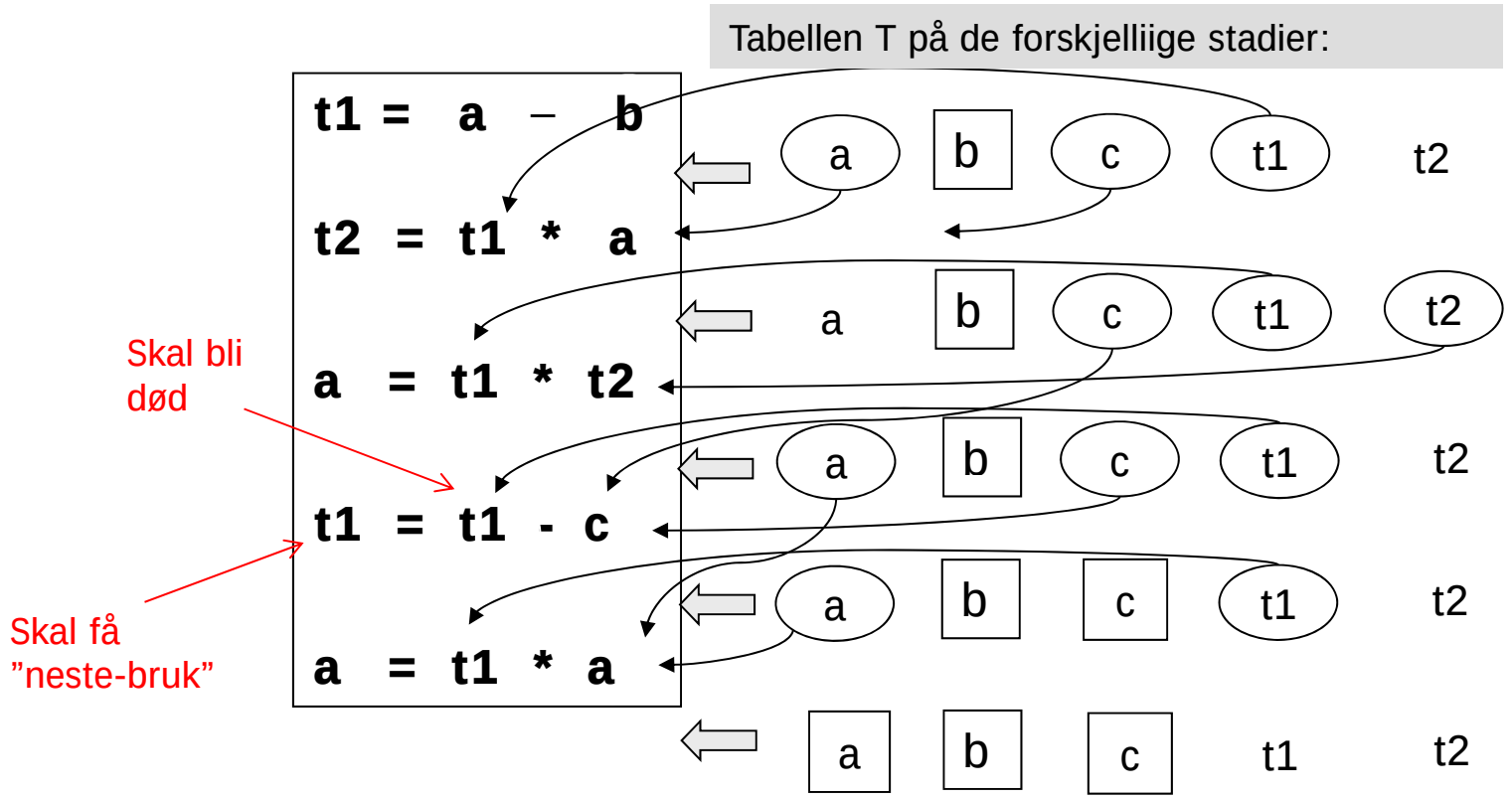
Steget (gjentas for hver TA-instr. ”x = y op z”, fra siste til første):

1. Merk x i TA-instr. slik x er merket i T
2. Forandre x sitt merke i T til 3 (altså død)
3. Merk y og z i TA-instr. slik de er merket i T
4. Forandre i T merkene for y og z til 1, med ”neste bruk” satt til h.h.v y og z i TA-instruksjonen.

Må skille mellom 1. og 3. for at `a = a + b;` skal bli riktig.
(Trykkfeil som er rettet i det utdelte notat)

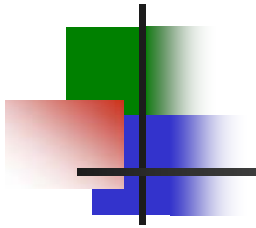
Algoritme for å finne ”neste bruk” og ”i live” innen en basic blokk

- Gå bakfra og hold greie på status til alle variable:
  (N.B. Trykkfeil i bokas algoritme – rettet i det utdelte)



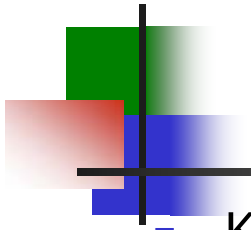
Døde variable er tegnet uten ring eller firkant rundt.

I nederste linje (initialiseringen) antar at vi at bare progr. variable overlever fra en blokk til en annen.



Kodegenererings-algoritme

- Altså, en enkel algoritme: Lager maskinkode for én og én Basal Blokk
- Algoritmen lager kode som, innefor hver Basal Blokk, holder beregnede verdier i registre så langt det er ønskelig og mulig (spesielt viktig om de har ”neste bruk” i denne BB)
- Når programkontrollen går mellom de Basale Blokkene så skal samtlige variabel-verdier ligge i sine respektive hukommelses-plasser (mens temporære variable ikke ”er i live”, og dermed ikke behøver lagres)
- Kodegenerering for hver Basal Blokk blir da:
  - Utfør algoritmen for å finne ”neste bruk” og ”i live” (går altså baklengs)
  - Det genereres kode for én og én treadresse-setning av gangen, i tur og orden fra første til siste setning
  - OG husk: Etter siste setning genereres kode for å legge verdier fra registre tilbake til sine respektive hukommelses-plasser der det er nødvendig.
- Noen mangler ved algoritmen (som i stor grad lett kan rettes opp)
  - Variable som kun blir *lest* innenfor en Basal Blokk blir *aldri* lagt i registre, selv ikke om det er gjentatte referanser til variabelen
  - I enkleste utgave utnytter den ikke på kommutativitet for + og *



Register- og adresse-deskriptorer

- Kodegenerator-algoritmen bruker deskriptorer for å holde greie på hva som er i registre og i program-variablene:
  - En Register-deskriptor for hvert register holder greie på hva som for tiden er i registerene. Ved starten skal alle register-deskriptorer angi at registeret er ledig. Generelt angir register-deskriptoren enten at registeret er ledig eller at det inneholder verdien til en eller flere angitte variable.
  - En Adresse-deskriptor holder greie på hvor verdien av en variabel finnes i øyeblikket. Den kan være i ett eller flere registre, og/eller i variabels lager-lokasjon.
    - Disse desriptorene opprettes etter hvert som det blir ”snakk om” variablene. At det ikke er noen adresse-deskriptor for ’x’ betyr:
      - x er programvariabel: Verdien ligger (bare) i variablens lager-lokasjon
      - x er temporær variabel: Variabelen er ikke i bruk nå
  - Informasjonen er redundant – dvs. vi har begge deskriptor-typene (adresse og register) ”bare” for å få raske oppslag. Kunne greid oss med én av dem.

Typisk bruk av deskriptorene

Setninger	Generert kode	Reg. deskriptorer	Adr. deskriptorer
		Alle Reg er ubrukte	
t = a - b	MOV a, R0 SUB b, R0	R0 inneholder t	t i R0
u = a – c	MOV a, R1 SUB c, R1	R0 inneholder t R1 inneholder u	t i R0 u i R1
v = t + u	ADD R1, R0	R0 inneholder v R1 inneholder u	v i R0 u i R1
d = v + u	ADD R1, R0	R0 inneholder d	d i R0 og ikke i hukommelsen
Avslutning av basal blokk	MOV R0, d	Alle Reg er ubrukte	(Alle prog.variable i hukommelsen)

Kodegenerering for: **X = Y op Z**

(Med rettelser som også er angitt i notatet)

1. Finn et register for å holde resultatet:
   - L = getreg(”X = Y op Z”) // Helst et sted Y allerede er
2. Sørg for at verdien av Y faktisk er i L:
   - Hvis Y er i L, oppdater adressediskr. til Y: Y ikke lenger i L **else**
   - Y’ := ”beste lokasjon” der verdien av Y finnes
   - OG: generer: **MOV Y’, L**
3. Sjekk adresse-deskriptoren for Z:

   Z’ := ”beste” lokasjon der verdien til Z ligger // Helst et register
   - Generer så ”hovedinstruksjonen”: **OP Z’ , L**
4. For hver av Y og Z: Om den er død og er i et register

   Oppdater i så fall register-deskriptoren:

   Registrene inneholder nå ikke lenger Y og/eller Z
5. Oppdaterer deskriptorer i forhold X:
   - X sin adr.deskr. := {L}, og X er ingen andre steder.
6. Hvis L er et register så oppdater register-deskr. for L:
   - L sin reg.deskr. := {X}

Getreg (”X = Y op Z”)

Instruksjonen som utfører operasjonen vi få Y som target-adresse

1. Hvis Y ikke er ”i live” etter ”X = Y op Z”, og Y er alene i R_i:
   - Return(R_i) (punkt 1 kan lett forfines en god del) **else**

2. Hvis det finnes et tomt register R_i : Return (R_i) **else**

3. Hvis X har en ”neste bruk” eller X er lik Z eller operatoren ellers krever et register:
   - Velg et (okkupert) register R
   - Hvis verdien i R ikke også ligger ”hjemme” i hukommelsen:
     - Generer **MOV R, mem** // mem er lagerlokasjonen for R-verdien
     - Oppdater adresse-deskriptor for **mem**
   - return (R) **else**

4. return (X), altså lever hukommelses-plassen til X (må kanskje opprettes om X er en temp-variabel)

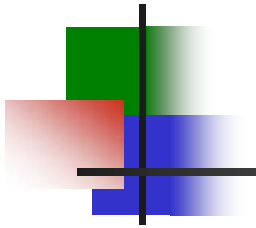
Opprinnelig verdi av X ødelegges

NB: For at X = Y + X skal funke, måtte pnk. 3 modifiseres, ellers ville vi fått:

```
MOV  Y  X
ADD  X  X
```

Eksempel på kode-generering (samme som en tidligere foil)

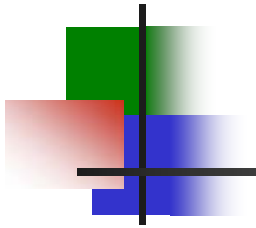
Setninger	Generert kode	Reg. deskriptorer	Adr. deskriptorer
		Alle Reg er ubrukte	
t = a - b	MOV a, R0 SUB b, R0	R0 inneholder t	t i R0
u = a – c	MOV a, R1 SUB c, R1	R0 inneholder t R1 inneholder u	t i R0 u i R1
v = t + u	ADD R1, R0	R0 inneholder v R1 inneholder u	v i R0 u i R1
d = v + u	ADD R1, R0	R0 inneholder d	d i R0 og ikke i hukommelsen
Avslutning av basal blokk	MOV R0, d	Alle Reg er ubrukte	(Alle prog.variable i hukommelsen)



Eksempel på global analyse:
Hvilke variable er *egentlig* i live etter en basal blokk?

NB: Ér pensum, men bare beskrevet her!

- Vi har tidligere antatt at *alle* programvariable er i live etter hver basal blokk (og ingen temporær-variable).
- Den analysen vi ser på her vil gi svar på hvilke variable som *faktisk* er i live etter hver blokk (altså: De der det er mulig at deres verdi vil bli brukt videre, og vi kan godt her også inkludere temporære variable).
- Ved "global analyse" ser vi på hele flytgrafen for *en metode*
- Vi kunne gjort analysen på enkeltinstruksjoner, men det ville gått sent
- I stedet ser vi på hver basal blokk, og koker den informasjonen vi trenger om hver av blokkene ned til to Ja/Nei-verdier, nemlig følgende:
  - Får variablen en ny verdi i denne basale blokken?
  - Blir den verdien som er i variablen ved inngang av en blokk brukt i den basale blokken?
- Disse to verdiene lagrer vi så for hver blokk, og de vil være grunnlaget for analysealgoritmen.



Fire typer basale blokker (for hver variabel)

- Fra forrige foil:
  - Får variablen en ny verdi i denne basale blokken?
  - Blir den verdien som er i variablen ved inngang brukt i den basale blokken?
- Vi får da fire typer basale blokker. Eksempler:
  - Ja, ja: Blir brukt, men får etterpå ny verdi (kanskje flere ganger)

    Variablen x i denne blokka: a = x + y

    x = u + v
  - Ja, nei: Får ny verdi før den eventuelt brukes

    Variablen x i denne blokka: x = u + v

    a = x + y
  - Nei, ja: Blir brukt, men verdien forandres ikke gjennom blokka

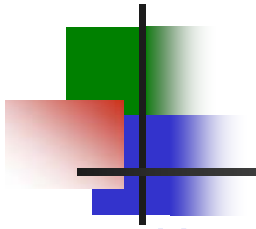
    Variablen x i denne blokka: a = x + y

    y = u + v
  - Nei, nei: Blir hverken brukt eller får ny verdi

    Variablen x i denne blokka: a = w + y

    x = u + v

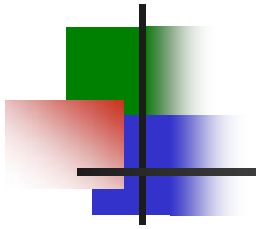
Data for kompletterings-algoritmen



- Hver basal blokk har, ut fra strukturen på flytgrafen, en mengde av (direkte) *etterfølgere* og *forgjengere* (mengder av basale blokker), og disse vil altså være konstant gjennom algoritmen!
  - Begge disse mengdene kan inkludere blokken selv

- Under algoritmen har hver basal blokk to mengder av variable knyttet til seg (og disse vil stadig få flere elementer gjennom algoritmen)
  - InLive: Dette skal bli mengen av variable som er i live helt i starten av den aktuelle blokken. Initielt er denne mengden tom for alle blokker.
  - OutLive: Dette skal bli mengden av variable som er i live helt på slutten av den aktuelle blokka. Det er *den* vi egentlig er interessert i (for å kunne initialisere algoritmen som finner NextUse etc. med så få variable som mulig).

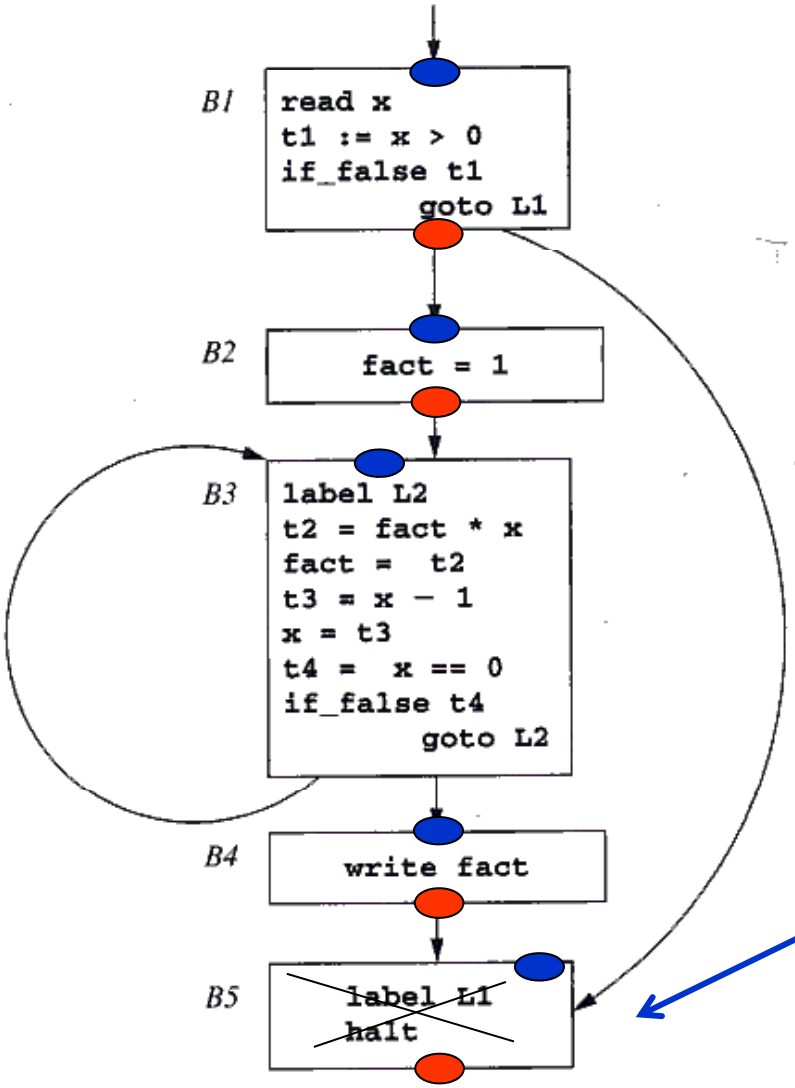
    Også disse skal fra starten være tomme, bortsett fra i avslutnings-blokka (return-setningen ser vi som en egen blokk). Den skal ha OutLive lik variablen som angis etter return (NB: vi antar ingen sideeffekter i metoden!)



Kompletterings-algoritme:

- Hver basal blokk har mengde av (direkte) *forgjengere* (basale blokker), og denne vil være konstant gjennom algoritmen!
  - Denne mengden kan inkludere blokken selv!
- Hver basal blokk har to mengder av variable *inLive* og *outLive*, og disse *vil* forandre seg under algoritmen
  - Disse er fra starten tomme, bortsett fra avslutnings-blokka (return-setningen som sees som egen blokk) som har outLive lik variablene som inngår i det returnerte uttrykk (og antar ingen sideeffekter!)
- Eksempel neste side

Eksempel på forgjenger-mengder m.m.



Forgjenger-mengdene er:

- For B1: ingen (starten av alg.)
- For B2: B1
- For B3: B2 og B3
- For B4: B3
- For B5: B1 og B1

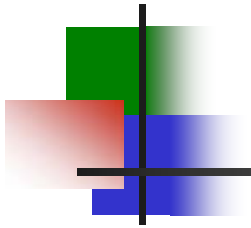
: InLive

: OutLive

Denne blokken regner vi i stedet med at inneholder:

return (fact)

Dette må da være en TA-instruksjon



Kompletterings-algoritmen:
(eksempel kommer som oppgave 15/5-12)

- Selve algoritmen går på at vi utfører følgende kompletteringssteg så lenge en av de tre er mulig for en eller variabel og en eller annen blokk:
  - Baklengs gjennom blokka: Dersom en variabel er med i outLive til en blokk, og blokka ikke gir variablen ny verdi, så skal den også være med i inLive til den blokka
  - Brukes i blokka: Dersom inn-verdien av en variabel brukes i blokka så skal den være med i inLive til blokka
  - Tilbakeføring til forgjenger: Dersom en variabel er med i inLive til en blokk, så skal den også være med i outLive for alle dens forgjengere

  - Man må altså overbevise seg om at det som legges inn i en av mengdene InLive eller OutLive ved disse skrittene *må* faktisk være med i de respektive mengdene.
  - Det som står i OutLive-mengdene når algoritmen stopper kan vi bruke som initialisering til algoritmen som finner next-use etc.