

INF5110 – 4. mai 2011

Mer om kodegenerering, spesielt om registerallokering

Deles ut, og på planen:

Tilleggsnotat fra ASU: Bok av Aho, Sethi og Ullman

Stein Krogdahl, Ifi, UiO

Ellers står igjen i kurset:

Litt om Javas Byte-kode

Oppgaver til kodegenerering



Avsluttende om kodegenerering

Avsluttende pensum:

En del fra utdelt kap 9 fra Aho, Sethi og Ullmann's kompilatorbok ("Drage-boka") : Kap. 9.4, 9.5 og 9.6

Vi innfører en del begreper:

som vi ikke skal bruke i fullt monn. De er imidlertid viktige om man vil gå videre med litt optimalisering, og vi skal forsøke å gi begrepene en intuitiv begrunnelse

Som bakgrunnsmateriale:

anbefales man på det *sterkeste* å lese kap. 8.9 i Louden (dog er det ikke pensum)

Maskinen det oversettes til i notatet

- To-adresse-intruksjoner:

op source dest
↑ ↑ ↑
hva *mange forskjellige ulike former, se neste foil*

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

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 I

+ **Betingete hopp**

+ **Prosedyre kall**

++

Complex
Instruction
Set Computer

Reduced
Instruction
Set Computer

Mer er en CISC-maskin enn en RISC-maskin

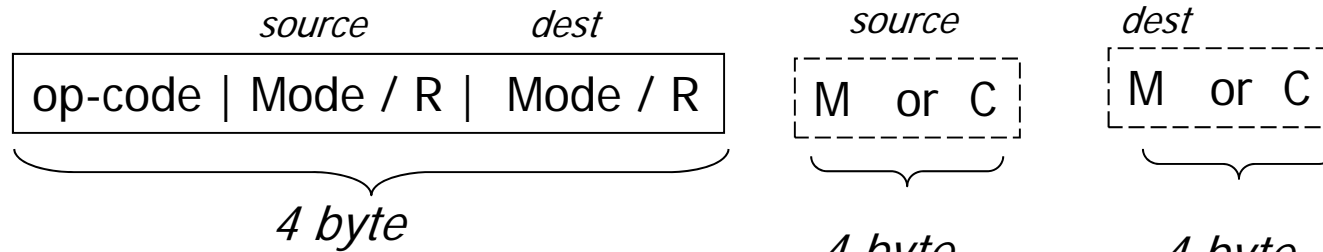
Instruksjonsformat og adressemodi – del 1

Kan se dette som eksempler på typiske adresseringsformer

Vi bruker instruksjonens lengde som "kost" av en instruksjon

Grunnkost for en fire-bytes instruksjon er 1

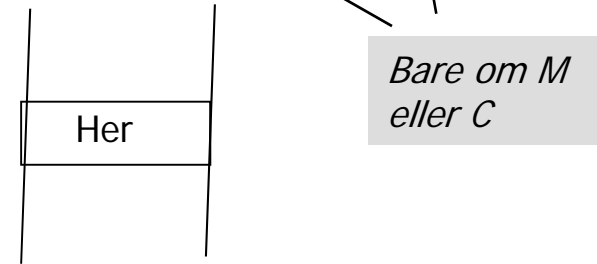
NB: Dette med hva en aksess til lageret "koster" er nå blitt veldig uforutsigelig pga. cacher etc.



+ Tilleggs-kost for hver adressering

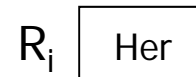
Adresseringsmodi:

1 Absolutt: M



Bare om M eller C

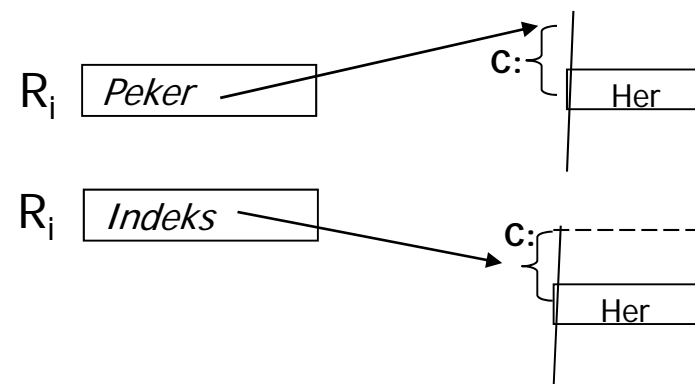
0 Register: R_i



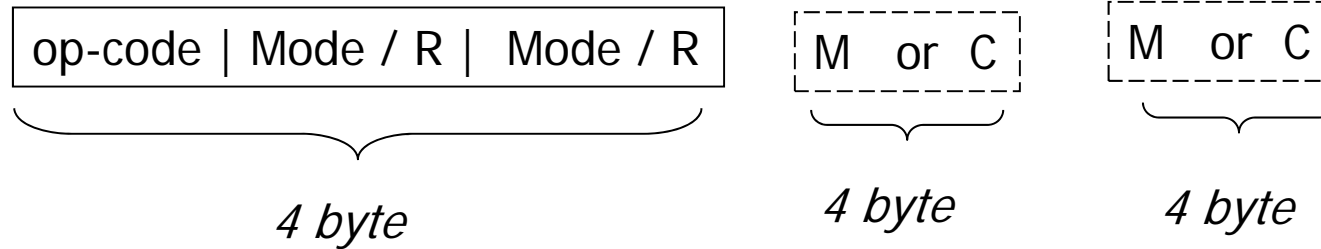
1 Indeksert : $C(RI)$

Grei på to måter:

- (1) Til å la R inneholde en objekt-peker og la C være en kompilator-kjent relativadresse i objektet
- (2) Til å la C være peker til en fastliggende array, og la R ha en indeks inn i denne

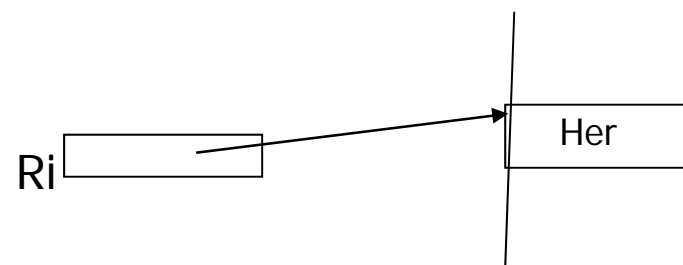


Instruksjonsformat og adressemodi – del 2



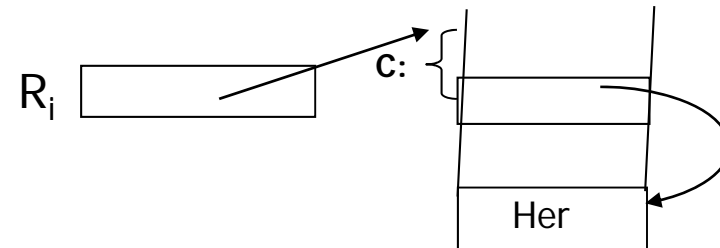
Adresseringsmodi:

0 Indirekte Register *R



1 Indirekte *C(R_i)

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



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

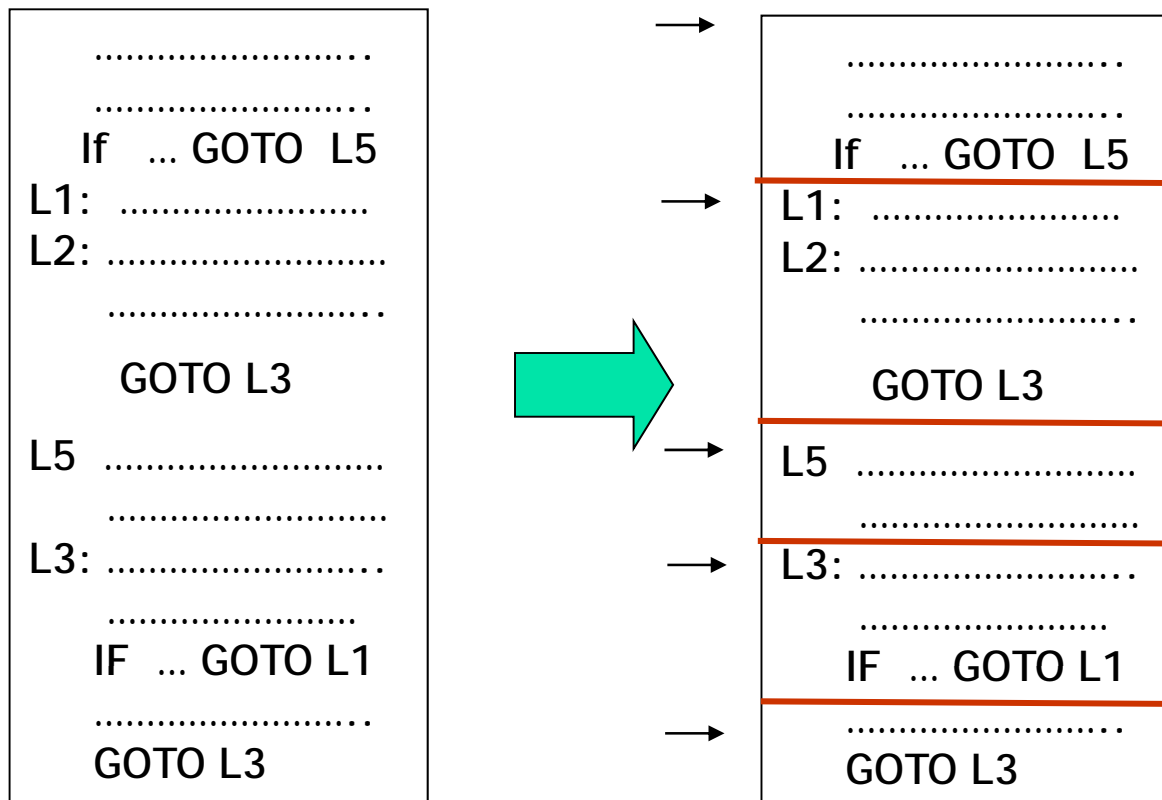


9.4 Basale blokker og Flyt-grafer

- 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 tydeligvis ingen som går til L2.

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

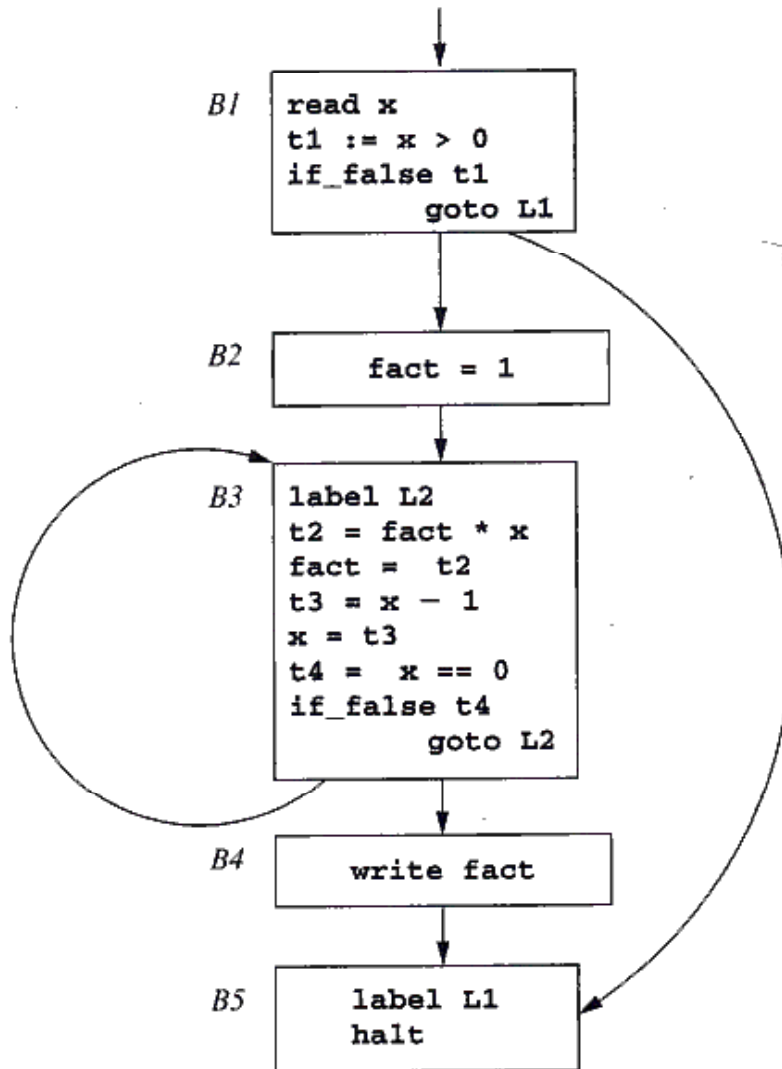


Flyt-graf

- 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_1 til 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. Flytgraf for én metode (“globalt nivå”)
3. Alle flytgrafene for hele programmet

Løkker i flyt-grafer

- Typisk bruk, for eksempel :

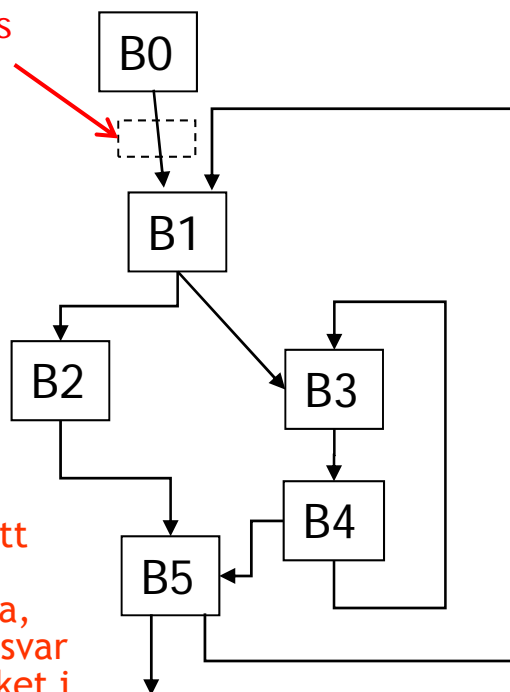
```
while (i<n) { i++; A[i] = 3*k; }
```

- 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. variabelen "i"**

Ny basal blokk der vi kan putte ting som "flyttes ut" av løkka, og gjøres på forhåndtning (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 $B_x \in L$ og $B_y \in L$, så går det en rettet vei fra B_x til B_y av lengde ≥ 1 (også om B_x og B_y er samme node!)

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

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":

```
x = v + w;  
...  
a = b + c;  
x = u + v      x = w  
              d = x + y
```

Stedet "i" er denne TA-instruksjonen
Er "x" i live her ?

Svaret er "ja", fordi det finnes en TA-setning "j" som *braker* "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)

- Men disse algoritmene er ikke pensum!



ASU, kap 9.5. Her ser man på:

Å genere "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 variabelens 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

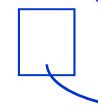


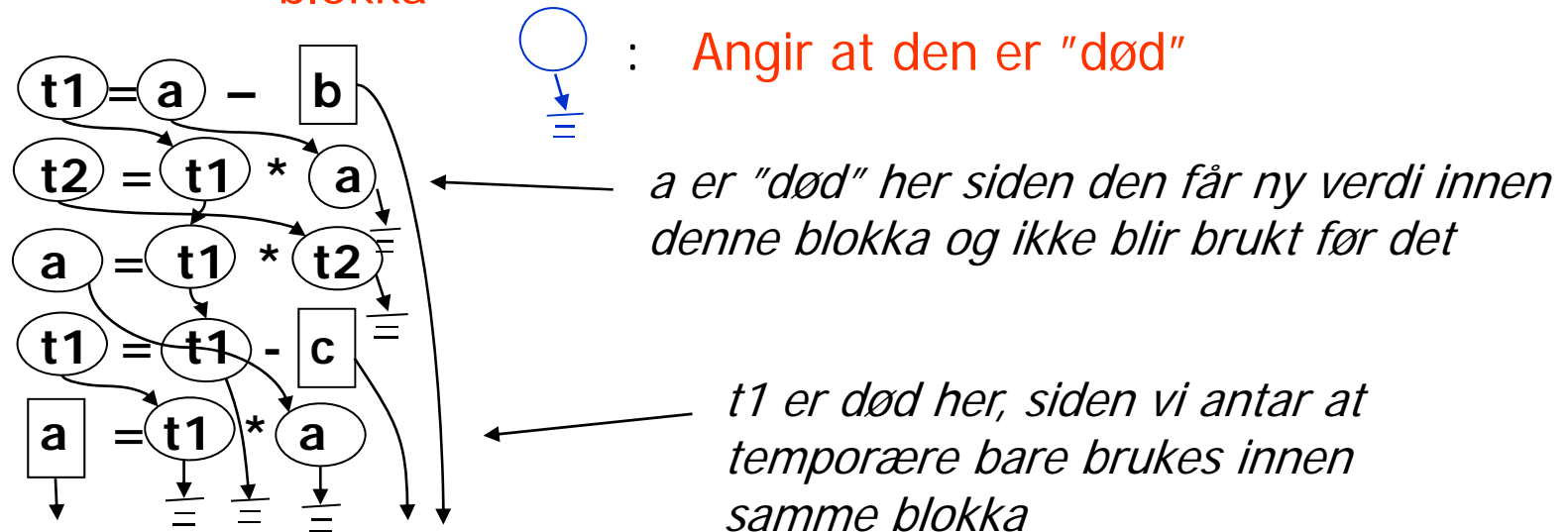
“Neste bruk” innen en basal blokk, og “i live”

- 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



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: programvariablene) merkes med 2, resten (de temporære) merkes 3

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

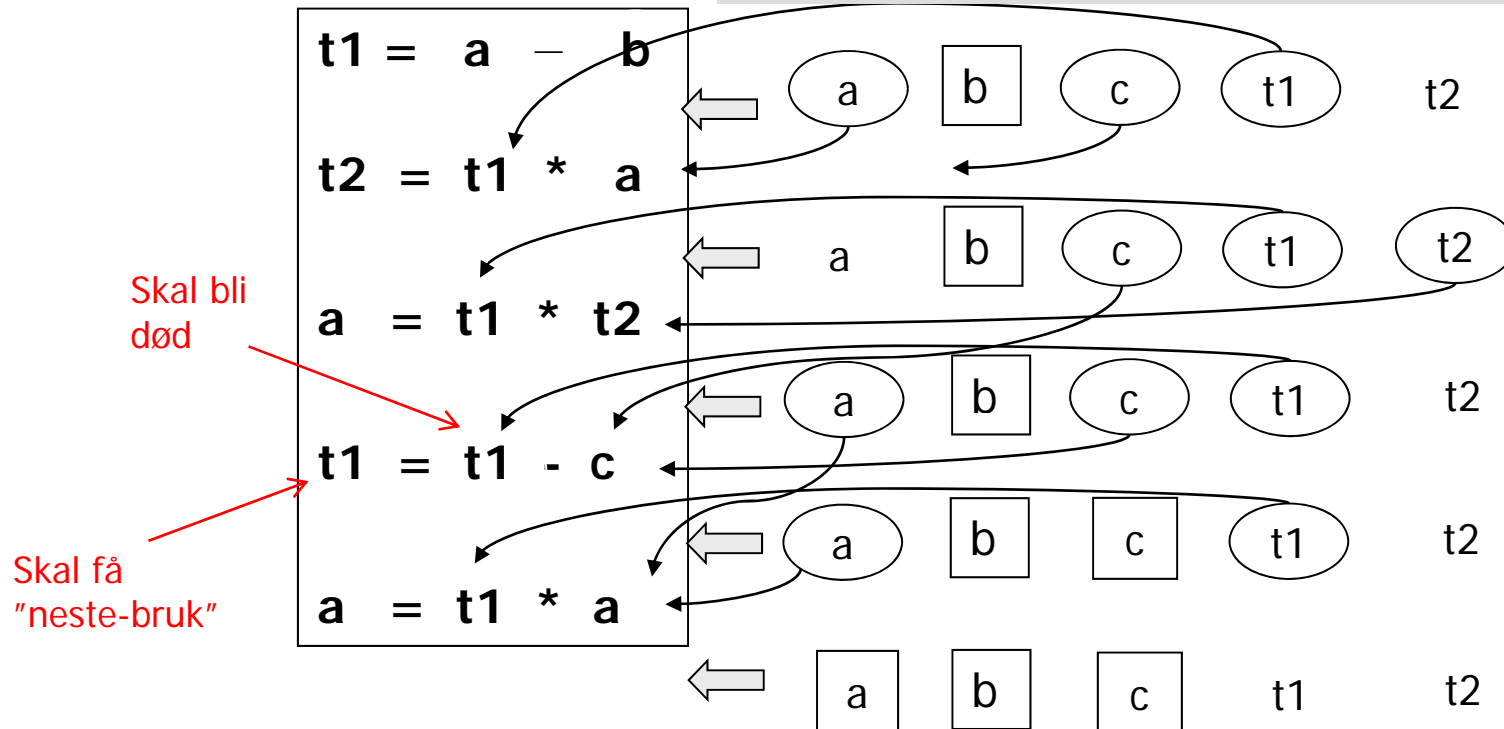
1. Merk x i TA-instr. slik x er merket i T
2. Forandre x sitt merke i T til 3 (altså død)
3. Merk y og z i TA-instr. slik de er merket i T
4. Forandre i T merkene for y og z til 1, med "neste bruk" satt til h.h.v y og z i TA-instruksjonen.

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)

Tabellen T på de forskjellige stadier:



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 variabelens 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: $X = Y \text{ op } Z$

(Med rettelser som også er angitt i notatet)

1. Finn et register for å holde resultatet:
 - $L = \text{getreg}("X = Y \text{ op } Z")$ // 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**
 - $Y' := \text{"beste lokasjon" der verdien av Y finnes}$
 - OG: generer: **MOV Y', L**
3. Sjekk adresse-deskriptoren for Z:
 $Z' := \text{"beste" lokasjon der verdien til Z ligger}$ // 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:
 - $X \text{ sin adr.deskr.} := \{L\}$, og X er ingen andre steder.
6. Hvis L er et register så oppdater register-deskr. for L:
 - $L \text{ sin reg.deskr.} := \{X\}$

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)