Oppgaver til kodegenerering etc.
INF-5110, 16. mai, 2014

Oppgave 1: Vi skal se på koden generert av TA-instruksjonene til høyre i figur 9.10 i det utdelte notatet, side 539

a) Se på detaljene i hvorfor maskinkoden ble slik den ble ut fra algoritmen.

b) Påvis at det finnes en bedre kodesekvens for de samme TA-setningene enn den angitte, som er generert av notatets algoritme.

c) Diskuter hvordan vi kunne forandre denne kodegenererings-algoritmen, slik at den gir bedre kode i dette og andre tilfeller.

Ekstra-oppgave om vi får tid:

d) Vi får oppgitt en mer komplisert sekvens av TA-instruksjoner som utgjør en basal blokk. Oversett denne etter algoritmen, og vurder om variasjonene vi fant i punkt 1c) kan hjelpe. Se egen foil.

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)

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

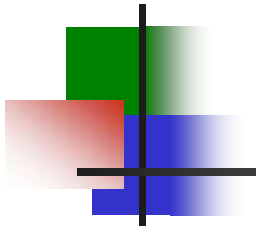
MOV Y X

ADD X X

Opprinnelig verdi av X ødelegges

Til oppgave 1:
Den genererte koden, samt bruk av deskriptorer

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 hjemmeposisjon
Avslutning av basal blokk	MOV R0, d	Alle Reg er ubrukte	(Alle prog.variable i hjemmepos.)

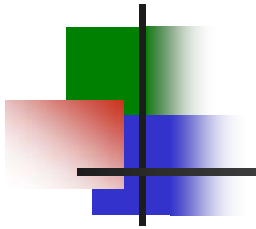


Svar oppgave 1c:

Noen mulige forbedringer på kodegenererings-algoritmen (som er gjengitt på de to neste foilene, direkte fra pensum)

- Når det er en symmetrisk operasjon (f.eks. * eller +) så kan man, i tillegg til å forsøke med x = y op z, også forsøke med x = z op y. Kanskje ligger verdiene til y og z slik til i registre at den siste gir bedre kode enn den første.

- Om man henter opp en verdi fra lageret til et register, og denne verdien vil bli ødelagt av operasjonen den skal inngå i (f.eks. slik som for a i første instruksjon i vårt eksempel), og denne verdien har en neste-bruk i blokka, så kan det lønne seg å kopiere denne verdien over i et ledig register (om slikt finnes). Man kan også i forbindelse med registrering av neste-bruk etc. registrere hvor *langt* det er til neste bruk, og dersom det er allerede i neste instruksjon (sik som hos oss) vil dette sikkert lønne seg.



Svar oppgave 1c (fortsatt):
Noen mulige forbedringer på kodegenererings-algoritmen (som er gjengitt på de to neste foilene, direkte fra pensum)

- Vi kan altså også gjøre flyt-analyse av flytgrafen for hele metoden og finne hvilke variable som virkelig er i live etter hver blokk. Da kan vi slippe mye «tilbake-storing» av verdier på slutten av blokka.

- Vi kan også forsøke å bytte om på to (eller flere) TA-instruksjoner som er naboer, men som ikke bruker noen av de samme variablene. Kanskje kan dette gi en bedre maskinkode-sekvens?

Ekstraoppgave: Oppgave 1d):

Vi ser på følgende sekvens av TA-instruksjoner. Denne skal oversettes til maskininstruksjoner etter bokas algoritme, men diskuter på veien variasjoner foreslått på 1c). Vi antar at maskinen har to registre, og at alle operasjonene er slik at destination må være et register (ikke for MOV!). TA-sekvensen er:

d = a – b

a = c + a

u = c / b

b = u + a

d = d + u

Svar: Neste foiler

Svar: Ekstraoppgave 1d)

Rett fram oversettelse etter algoritmen gir:

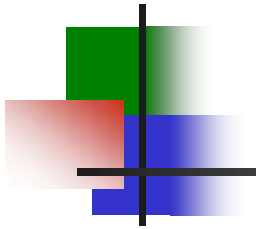
1. d = a – b
   ```
   MOV a  R0
   SUB  b  R0          d is in R0
   ```
2. a = c + a
   ```
   MOV c  R1
   ADD a  R1           d er i R0, a er i R1
   ```
3. u = c / b

   (c er ikke i noe register, men dest. må ha et register. Vi må frigjøre R0 (d) eller R1 (a). Vi velger R0 (d), fordi det er lengst til det skal brukes. Verdien av a skal brukes allerede i neste instruksjon)
   ```
   MOV R0  d
   MOV c  R0
   DIV   b  R0         u er i R0, a er i R1 (d er hjemme)
   ```
4. b = u + a

   (både u og a ligger i registre, men u har en neste-bruk og bør bevares. Vi frigjør R1, for å kunne bruke dette til dest.)
   ```
   MOV  R1  a
   MOV  R0 R1
   ADD   a   R1        u er i R0,  b er i R1 (a er hjemme)
   ```
5. d = d + u

   (vi må ha d i register før operasjonen, og må frigjøre et. Vi velger R1 (b), fordi den ikke har noen neste-bruk)
   ```
   MOV  R1  b
   MOV  d   R1
   ADD  R0  R1         u is in R0, d is in R1
   ```

Save alive variables (program variables)
```
MOV R1 d               (u er temporær, og behøver ikke saves)
```



Svar: Ekstraoppgave 1d)

Vurdering av forbedringer I

1. d = a – b

   Om vi hadde brukt at a skal brukes allerede i neste instr, kunne vi reddet dens verdi i R1

   ```
   MOV  a  R0
   MOV  R0 R1
   SUB   b  R0          d is in R0
   ```

2. a = c + a

   Om vi også ser på «a = a + c» så viser det seg at denne er bedre, og det gir

   ```
   ADD   c  R1          d er i R0, a er i R1
   ```

3. u = c / b

   Her ser det ut til å være lite lurt å gjøre

   (c er ikke i noe register, men dest. må ha et register. Vi må frigjøre R0 (d) eller R1 (a). Vi velger R0 (d), fordi det er lengst til det skal brukes. Verdien av a skal brukes allerede i neste instruksjon)

   ```
   MOV   R0  d
   MOV   c  R0
   DIV    b  R0          u er i R0, a er i R1 (d er hjemme)
   ```

Svar: Ekstraoppgave 1d)
Vurdering av forbedringer II

```
                    …
                    MOV   R0  d
                    MOV   c  R0
                    DIV    b  R0          u er i R0, a er i R1 (d er hjemme)
```

4. b = u + a

Om vi her ser på «b = a + u» i stedet, så får vi bedre kode:
(a har ingen neste-bruk, men vi må anta den er i live. Så det er greit å gjøre operasjonen i R1, men vi må save verdien av a først)

```
MOV  R1  a
ADD   R0  R1          u er i R0,  b er i R1 (a er hjemme)
```

5. d = d + u

Om vi her i stedet ser på «d = u + d» så får vi bedre kode
(Dette blir da veldig rett fram)

```
ADD   d  R0          d is in R0, b er i R1
```

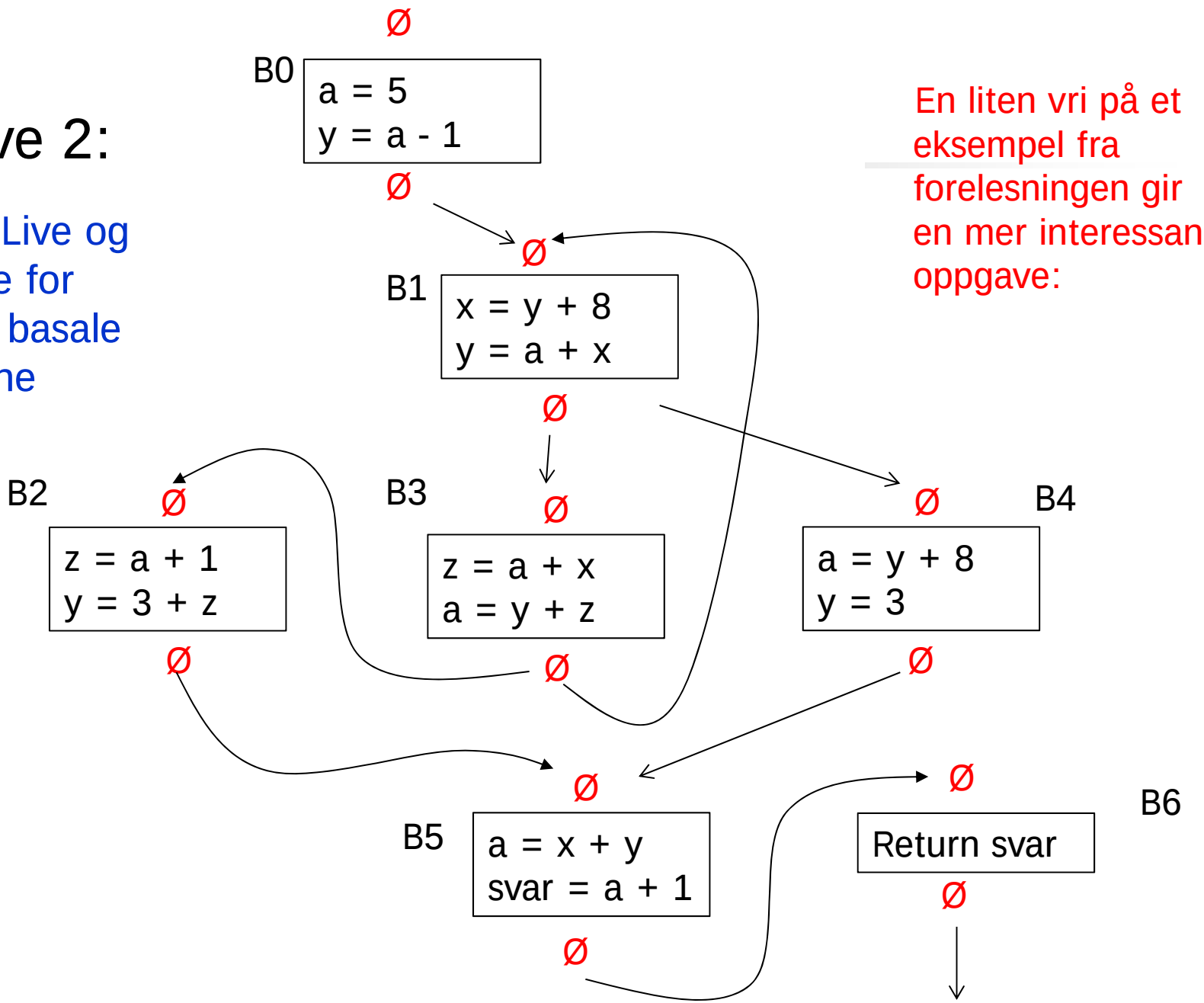
Save alive variables (program variables)

Vi får nå to variabler å save:

```
MOV R0  b
MOV R1  d
```

Oppgave 2:

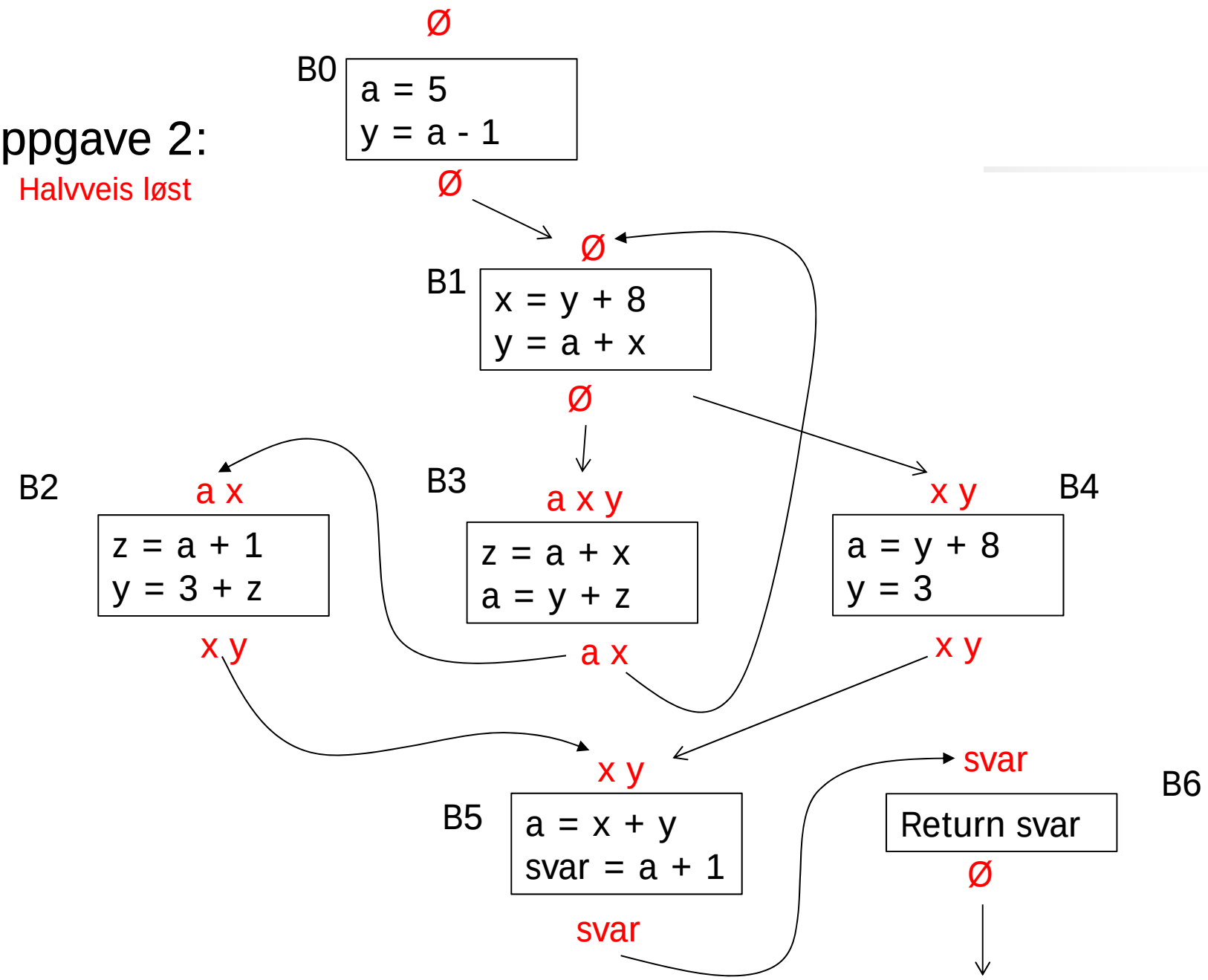
Finn InLive og OutLive for alle de basale blokkene



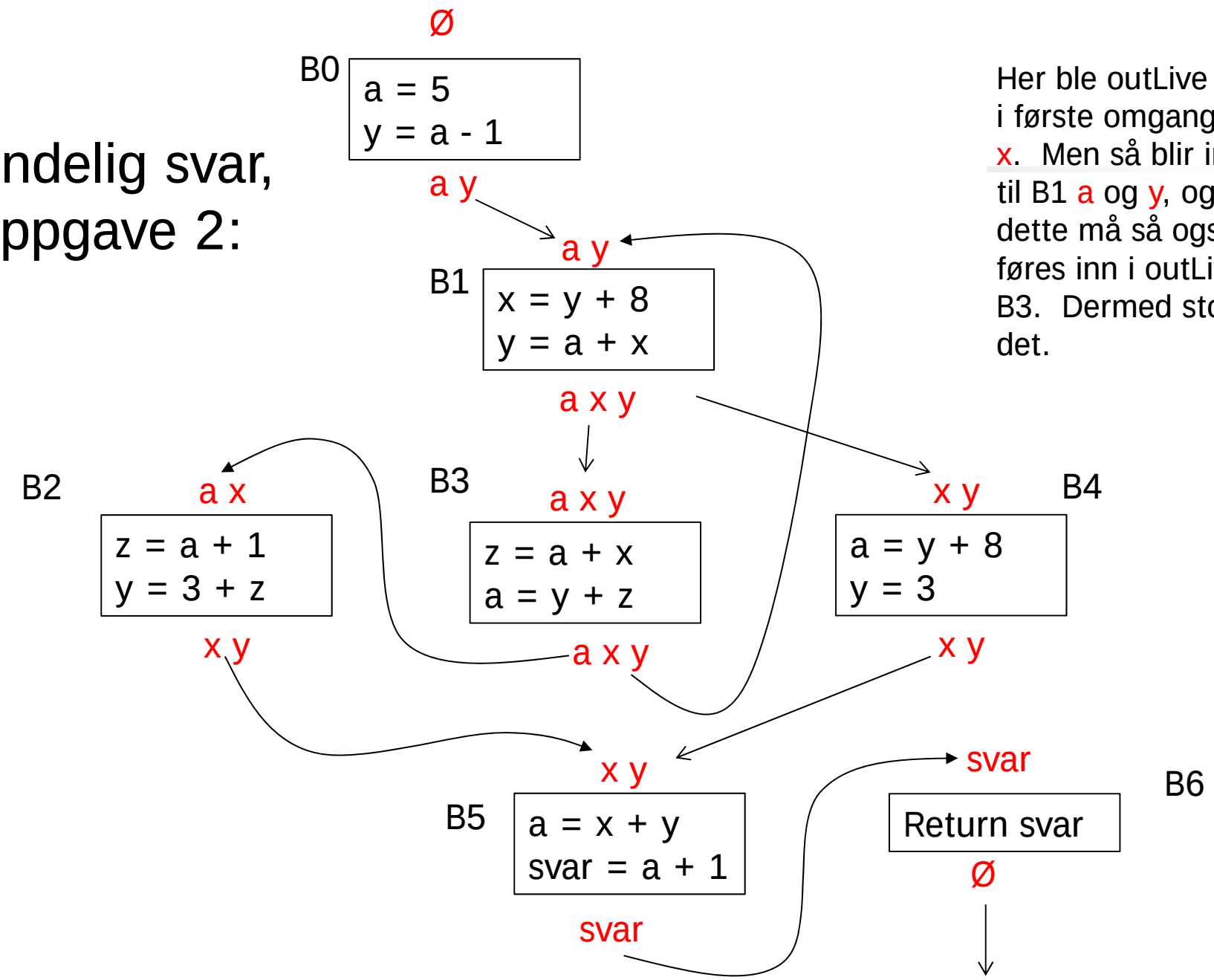
En liten vri på et eksempel fra forelesningen gir en mer interessant oppgave:

Oppgave 2:

Halvveis løst



Endelig svar, oppgave 2:



Her ble outLive til B3 i første omgang a og x. Men så blir inLive til B1 a og y, og dette må så også føres inn i outLive til B3. Dermed stopper det.

Oppgave 3 (Eksamen 2007, opg 4a)

Gitt følgende program, der alle setningene er tre-addresse-instruksjoner, bortsett fra at vi også tillater if- og while-setninger på vanlig måte. Instruksjonene ”x = input” og ”output x” regnes som vanlige tre-addresse-instruksjoner, med den opplagte betydning. Vi antar at ingen variable er i live ved slutten av programmet.

```
1:   a = input
2:   b = input
3:   d = a + b
4:   c = a * d
5:   if ( b < 5 ) {
6:      while ( b < 0 ) {
7:         a = b + 2
8:         b = b + 1
9:      }
10:     d = 2 * b
11: } else {
12:     d = b * 3
13:     a = d - b
14: }
15: output a
16: output d
```

Angi for hver av variablene *a, b, c* og *d* om de er i live eller ikke *umiddelbart etter* linje 4. Gi en kort forklaring for hver av variablene.

Svarforslag, oppgave 3 (Eksamen 2007, opg 4a)

Gitt følgende program, der alle setningene er tre-addresse-instruksjoner, bortsett fra at vi også tillater if- og while-setninger på vanlig måte. Instruksjonene ”x = input” og ”output x” regnes som vanlige tre-addresse-instruksjoner, med den opplagte betydning. Vi antar at ingen variable er i live ved slutten av programmet.

```
1:   a = input
2:   b = input
3:   d = a + b
4:   c = a * d
5:   if ( b < 5 ) {
6:      while ( b < 0 ) {
7:         a = b + 2
8:         b = b + 1
9:      }
10:     d = 2 * b
11: } else {
12:     d = b * 3
13:     a = d - b
14: }
15: output a
16: output d
```

Angi for hver av variablene *a, b, c* og *d* om de er i live eller ikke *umiddelbart etter* linje 4. Gi en kort forklaring for hver av variablene.

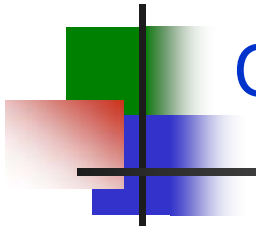
Svar:

a. Det kunne se ut som om denne er død, siden den settes (”defineres”) både i den ene og den andre grenen av if-setningen, uten å bli brukt først (linje 7 og linje 13). Men, dersom while-løkke går null ganger vil den verdien som **a** hadde etter linje 4 være den som brukes i linje 15. Altså er den i live.

b. Denne er i høyeste grad live, siden den brukes allerede i linje 5

c. Denne brukes i det hele tatt ikke etter linje 4, og er derfor ikke i live (altså død).

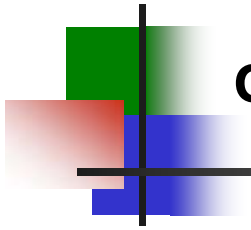
d. Denne er ikke i live, siden den helt sikkert settes i begge if-grener, uten å bli brukt først (linje 10 og 12).



Oppgave 4 (Eksamen 2009, 4a, 4b, 4c og 4d)

Gitt følgende sekvens av treadresse-instruksjoner (TA-instruksjoner):

```
1     a  =  input
2     b  =  input
3     t1 =  a + b
4     t2 =  a * 2
5     c  =  t1 + t2
6     if  a < c goto 8
7     t2 =  a + b
8     b  =  25
9     c  =  b + c
10    d  =  a – b
11    if  t2 = 0 goto 17
12    d  =  a + b
13    t1 =  b – c
14    c  =  d – t1
15    if  c < d  goto 3
16    c  =  a + b
17    output  c
18    output  d
```



Oppgave 4 (Eksamen 2009, 4a og 4b)

4a Angi hvor det starter en ny basal blokk (”basic block”).

4b Gi hver basal blokk nedover i programmet navn B1, B2, ... osv, og tegn opp flytgrafen (uten koden, men bare med navnet inni hver node).

Svar ,oppgave 4 (Eksamen 2009, 4a og 4b)

```
------------------------------
1    a  =   input              B1
2    b  =   input
------------------------------
3    t1 =  a + b               B2
4    t2 =  a * 2
5    c  =  t1 + t2
6    if  a < c goto 8
------------------------------
7    t2 =  a + b               B3
------------------------------
8    b  =  25                  B4
9     c  =  b + c
10   d  =  a – b
11   if  t2 = 0 goto 17
------------------------------
```

```
------------------------------
12   d  =  a + b               B5
13   t1 =  b – c
14   c  =  d – t1
15   if  c < d  goto 3
------------------------------
16 c  =  a + b                 B6
------------------------------
17   output  c                 B7
18   output  d
```

Oppgave 4 (Eksamen 2009, opg 4c)

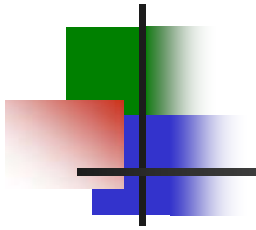
Oppgaven:

Den som produserer TA-kode påstår at den er slik at temporære variable alltid er døde på slutten av hver basal blokk, og ved starten av programmet, selv om de samme temporærvariable altså blir brukt i flere basale blokker, slik som over.

Formuler en generell regel som skal brukes lokalt på alle basale blokker for å sjekke om det produsenten av TA-kode sier er riktig, for et gitt TA-program. Vi antar her at alle variable er døde når programmet over stopper (etter siste instruksjon).

Svar 4.c:

Reglen kan være: ”Alle temporærvariable som blir *avlest* (i boka: ”used”) i en basale blokk, må i denne blokken bli *gitt en verdi* (i boka: ”be defined”) før de blir avlest ”. Man kunne alternativt si: ”Ingen temporærvariable må ha noen «neste bruk» (”next use”) ved starten av noen basale blokk”.



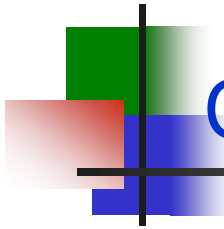
Oppgave 4 (Eksamen 2009, opg 4d)

Oppgaven:

Bruk reglen du fant under **4.c** til å undersøke hvordan dette forholder seg i den TA-koden som er angitt over. De temporære variablene heter t1, t2.

Svar 4.d

Det er én blokk der denne reglen ikke følges, nemlig i B4. Der brukes t2 i linje 11 uten å ha fått verdi først.



Oppgave 5 (Eksamen 2010, opg 4d)

Oppgaven:

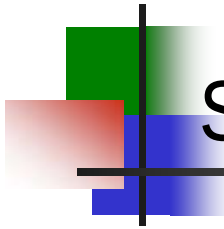
Arne har sett på kodegenerings-algoritmen på slutten av det utdelte heftet (fra kap. 9 i ASU). Han mener da at for de to treadresse-instruksjonene:

t1 = a – b
t2 = b – c

så vil algoritmen produsere instruksjonene under. Han har antatt at det er to registre, og at begge er tomme ved starten

```
MOV a, R0
MOV b, R1
SUB  R1, R0
SUB  c, R1
```

Ellen er uenig. Hvem har rett? Forklar.

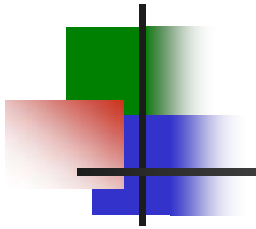


Svar: Oppgave 5 (Eksamen 2010, opg 4d)

Nei, algoritmen vil ikke generere dette. Den vil faktisk (slik som i eksempelet på side 539 i det utleverte notat) alltid hente den siste operanden (hhv b og c) direkte fra ”hjemme-posisjon” i lageret dersom den ikke allerede er i et register.

Det var vel ikke krav om å skrive hva den ville generere, men det ville altså bli:

```
MOV a, R0
SUB  b, R0
MOV b, R1
SUB  c, R1
```



Oppgave 6 (Eksamen 2011, opg 4c)

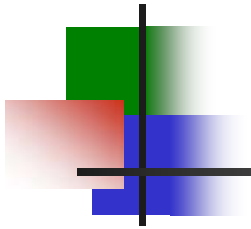
Oppgaven:

Vi vil oversette vår P-kode til maskinkode for en maskin der alle operasjoner (inkl. sammenlikninger) må gjøres mellom verdier som ligger i registre, og der kopiering mellom lageret og registre bare kan gjøres med egne LOAD- og STORE-instruksjoner. Under oversettelsen har vi en stakk med deskriptorer.

Vi skal se på det å oversette P-instruksjonen ”ldv v”.

(Denne «push’er» altså verdien av variablen v til toppen av stakken.) Spørsmålet er om det da er fornuftigst å produsere en LOAD-instruksjon som henter verdien av variabelen ”v” opp i et register, eller om det er best bare å legge en deskriptor på stakken som sier at denne verdien ligger i variabelen ”v”.

Drøft dette ut fra forskjellige forutsetninger, f.eks. ut fra hva språket som vi oversetter fra lover om rekkefølgen ved beregning av uttrykk (men også ut fra andre ting som du mener er aktuelle).



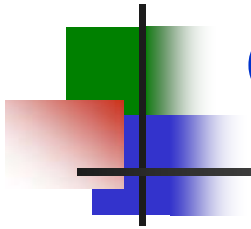
Svar: Oppgave 6 (Eksamen 2011, opg 4c)

Grovt sett:

- Dersom språk-reglene sier at man må utføre uttrykk i rekkefølge fra venstre mot høyre, så må man i det generelle tilfellet lage en LOAD-instruksjon fra v (program-variabel) til et register med en gang.

- Om man er fri til å beregne uttrykket i vilkårlig rekkefølge er det lurt å lage en deskriptor. Da står man fritt til å vente med opphentingen til en operasjon faktisk trenger den verdien, slik at den ikke har tatt opp et register fram til den faktisk skal brukes.

- Her kan man optimalisere mye ved f.eks. å sjekke om det finnes noen prosedyrekall i det aktuelle uttrykket slik at verdier på variable kan forandre seg. Om det ikke det ikke finnes noe slikt kall man i alle tilfelle vente med å hente verdien (lage en LOAD-instr) til man trenger verdien (altså å legge en deskriptor på stakken).



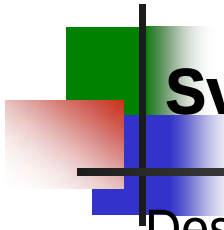
Oppgave 6 (Eksamen 2011, opg 4d)

Oppgaven:

Vi vil igjen oversette vår P-kode til maskinkode, slik som i oppgave **4c**, og vi skal anta at vi skal oversette én og én basal blokk, og at alle registre skal tømmes på kontrollert måte etter utførelsen av en basal blokk.

Spørsmålet her er hvilke data deskriptorene på stakken skal inneholde, og hva du eventuelt trenger av andre typer deskriptorer. Vi antar at vi kan tillate oss å lete gjennom alle kompilator-stakkens deskriptorer hver gang vi lurer på hvor visse verdier er etc., slik at informasjon vi kan finne på denne måten ikke behøver å lagres i ekstra deskriptorer.

Forklar også kort hvordan du kan finne den informasjonen du trenger under kodegenereringen, og hvordan du eventuelt vil bruke de ekstra deskriptorene du vil ha.



Svar: Oppgave 6 (Eksamen 2011, opg 4c)

Deskriptorene på stakken bør i hvert fall inneholde følgende:

- Om det er en konstant (og da hvilken),
- Om det er verdien til en program-variabel (og da hvilken variabel)
- Om det er en verdi som ligger i et register (og i så fall hvilket).

Deskriptorene på stakken vil generelt ikke inneholde nok informasjon til å holde orden på hva som er i hvilke registere, om en variabel-verdi er i sin ”hjemmeposisjon”, etc.

Dette blir opplagt når man ser at stakken kan bli tom mellom setningene inne i den basale blokken, og at det da fremdeles kan være variabel-verdier som ligger i registre i påvente av at verdiene kanskje skal brukes en gang til.

Det fører til at man får omtrent de samme behov som i kodegenererings-algoritmen i boka, og at det kan være greit med både en register-deskriptor og en adresse-deskriptor. Vi skal jo også, ved slutten av den basale blokka, sette alle verdier tilbake til deres ”hjemmeposisjon” i sine variable, og da er disse deskriptorene viktige.