Oppgaver til kodegenerering etc.
INF-5110, 2013

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.

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

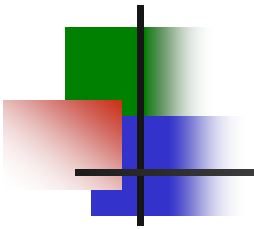
Setninger	Generert kode	Reg. deskriptorer	Adr. deskriptorer
		Alle Reg er ubrukte	
t = a - b	MOV a, R0 SUB b, R0	R0 inneholder t	t i R0
u = a – c	MOV a, R1 SUB c, R1	R0 inneholder t R1 inneholder u	t i R0 u i R1
v = t + u	ADD R1, R0	R0 inneholder v R1 inneholder u	v i R0 u i R1
d = v + u	ADD R1, R0	R0 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.)

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 den 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 om det er allerede i neste instruksjon (sik som hos oss) vil dette sikkert lønne seg.

- 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.



Oppgave 2
(se bakgrunn neste foiler)

- Vi har i pensum sett hvordan man i logisk uttrykk med operatorene *and*, *or* og *not,* kan bruke hopp (i stedet for å beregne den logiske verdien) og sørge for at disse går direkte til det stedet der denne logiske verdien skulle føre oss. Men ting blir ikke helt bra, for vi kan få slike situasjoner

```
...
jmp L
label L
...
```

- I slike tilfeller bør vi klare å få vekk "jmp L". Hvordan kan man få inn dette i vårt program?

- Hint: Vi gir en label-parameter til, som angir lablen akkurat etter det "du" skal lage kode til

Bakgrunn, oppgave 2
Gammel kodegenerering for If-setning til TA-kode

Her er vår gamle foil (fra ekstrapensum kap. 8) for å lage TA-kode for if-setninger, der hopp går direkte til riktig label. I tillegg til det skal vi altså i denne oppgaven bli kvitt de dumme hoppene som bare går til neste instruksjon. Det skal vi gjøre ved å ha nok en label-parameter, og den skal inneholde lablen som kalleren vet at kommer umiddelbart etter det «jeg» generer. Dermed kan jeg teste på om mitt siste hopp er til denne lablen, og i så fall la være å generere dette hoppet. De to neste foile viser hvordan det da blir.

```
……
case IfKind {
        String labT = genLabel();          // Skal hoppes til om, betingelse er True
        String labF = genLabel();          // Skal hoppes til om, betingelse er False
        genBoolCode(t.child[0], labT, labF); // Lag kode for det boolske uttrykket, med parametere!
        emit2(”lab”, labT);       //      True-hopp fra uttykket skal gå hit
        genCode(t.child[1]);   // kode for then-gren (nå uten label-parameter for break-setning)

        String labx = genLabel();      // Skal angi slutten av if-setningen, etter else-grenen
        if t.child[2] != null {            // Test på om det er else-gren?
            emit2(”ujp”, labx);              // Hopp over else-grenen
        }

        emit2(”label”, labF);      // False-hopp fra uttykket skal gå hit

        if t.child[2] != null {      // En gang til: test om det er else-gren? (litt plundrete programmering)
            genCode(t.child[2]); // Kode for else-gren (nå uten label-parameter)
            emit2(”label”, labx);      // Hopp over else-gren går hit
}           }
……
```

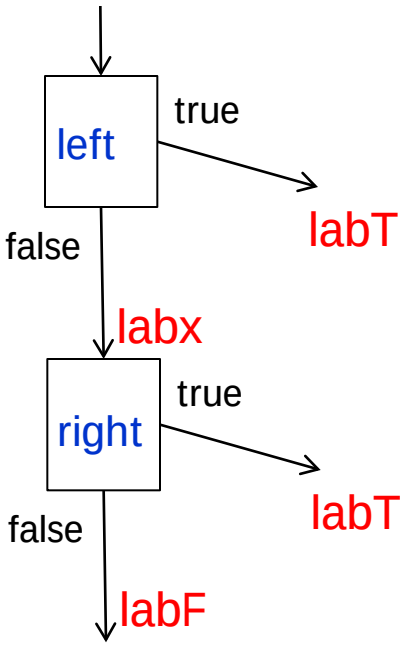
Mer bakgrunn, oppgave 2: Ikke helt bra versjon

Men heller ikke *denne* vil lage helt god kode!
Hvorfor?? (se helt nederst)

Vi bryr oss ikke med retur-navnet, siden de alltid vil hoppe ut

```
void genBoolCode(String labT, labF) {
        ...
        case ”||”: {
            String labx = genLabel();
            left.genBoolCode(labT, labx);
            emit2(”label”, labx);
            right.genBoolCode(labT, labF);
        }
        case ”&&”: {
            String labx = genLabel();
            left.genBoolCode(labx, labF);  // som over
            emit2(”label”, labx);
            right.genBoolCode(labT, labF);  // som over
        }
        case ”not”: { // Har bare ”left”-subtre
            left.genBoolCode(labF, labT);  // Ingen kode lages!!!
        }
        case ”<”: {
            String temp1, temp2, temp3; // temp3 skal holde den boolsk verdi for relasjonen
            temp1 = left.genIntCode(); temp2 = right.genIntCode();      temp3 = genLabel();
            emit4(temp3, temp1, «lt», temp2); // temp3 får (det boolske) svaret på relasjonen
            emit3(«jmp-false», temp3, labF);
            emit2(«ujp», labT);  // Denne er unødvendig dersom det som følger etter er stedet labT
                                 // Dette kan vi oppdage med en ekstra parameter som angir labelen bak
}       }                        // den konstrusksjonen man kaller kodegenererings-metoden for.
```

For ”||”:



Begynnende løsning på oppgave 2

Kodegenerering for if-setning uten “dumme hopp”

Altså, ny parameter, labAfter, til genBoolCode. Her må den gis som lablen ved starten av true-grenen, men det blir også eneste forandring

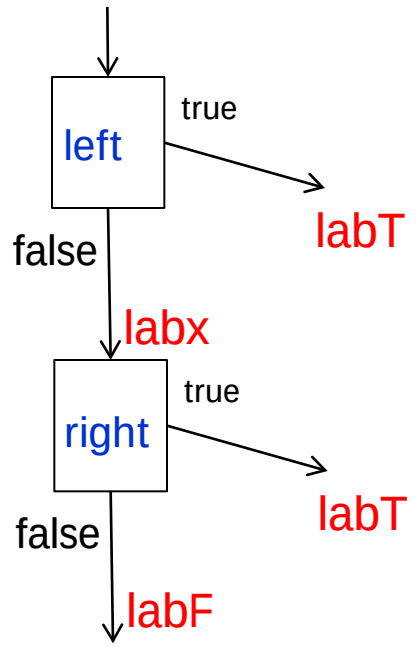
```
……
case IfKind {
        String labT = genLabel();        // Skal hoppes til om, betingelse er True
        String labF = genLabel();        // Skal hoppes til om, betingelse er False
        genBoolCode(t.child[0], labT, labF, labT);                    // ←----- NY!!
        emit2(”lab”, labT);      //     True-hopp fra det boolske uttykket skal gå hit
        genCode(t.child[1]);   // kode for then-gren (nå uten label-parameter for break-setning)

        String labx = genLabel();     // Skal angi slutten av if-setningen, etter else-grenen
        if t.child[2] != null {          // Test på om det er else-gren?
            emit2(”ujp”, labx);          // Hopp over else-grenen
        }

        emit2(”label”, labF);     // False-hopp fra det boolske uttykket skal gå hit

        if t.child[2] != null {     // En gang til: test om det er else-gren? (litt plundrete programmering)
            genCode(t.child[2]); // Kode for else-gren (nå uten label-parameter)
            emit2(”label”, labx);     // Hopp over else-gren går hit
}       }
……
```

Genere TA-kode for boolske uttrykk uten dumme hopp

```
void genBoolCode(String labT, labF, labAfter) {
    ...
    case "||": {
        String labx = genLabel();
        left.genBoolCode(labT, labxl labx);
        emit2(«label», labx);
        right.genBoolCode(labT, labF, labAfter)
    }
    case "&&": {
        String labx = genLabel();
        left.genBoolCode(labx, labF, labx);
        emit2(«label», labx);
        right.genBoolCode(labT, labF, labAfter);
    }
    case "not": { // Har bare "left"-subtre
        left.genBoolCode(labF, labT, labAfter);  // Ingen kode!!!
    }
    case "<": {
        String temp1, temp2, temp3; // temp3 skal holde den boolsk verdi for relasjonen
        temp1 = left.genIntCode(); temp2 = right.genIntCode();      temp3 = genLabel();
        emit4(temp3, temp1, «lt», temp2);
        emit3(«jmp-false», temp3, labF);
        if (labAfter != labT)  {emit2("ujp", labT); }  <---  NY test!
    }
}
```

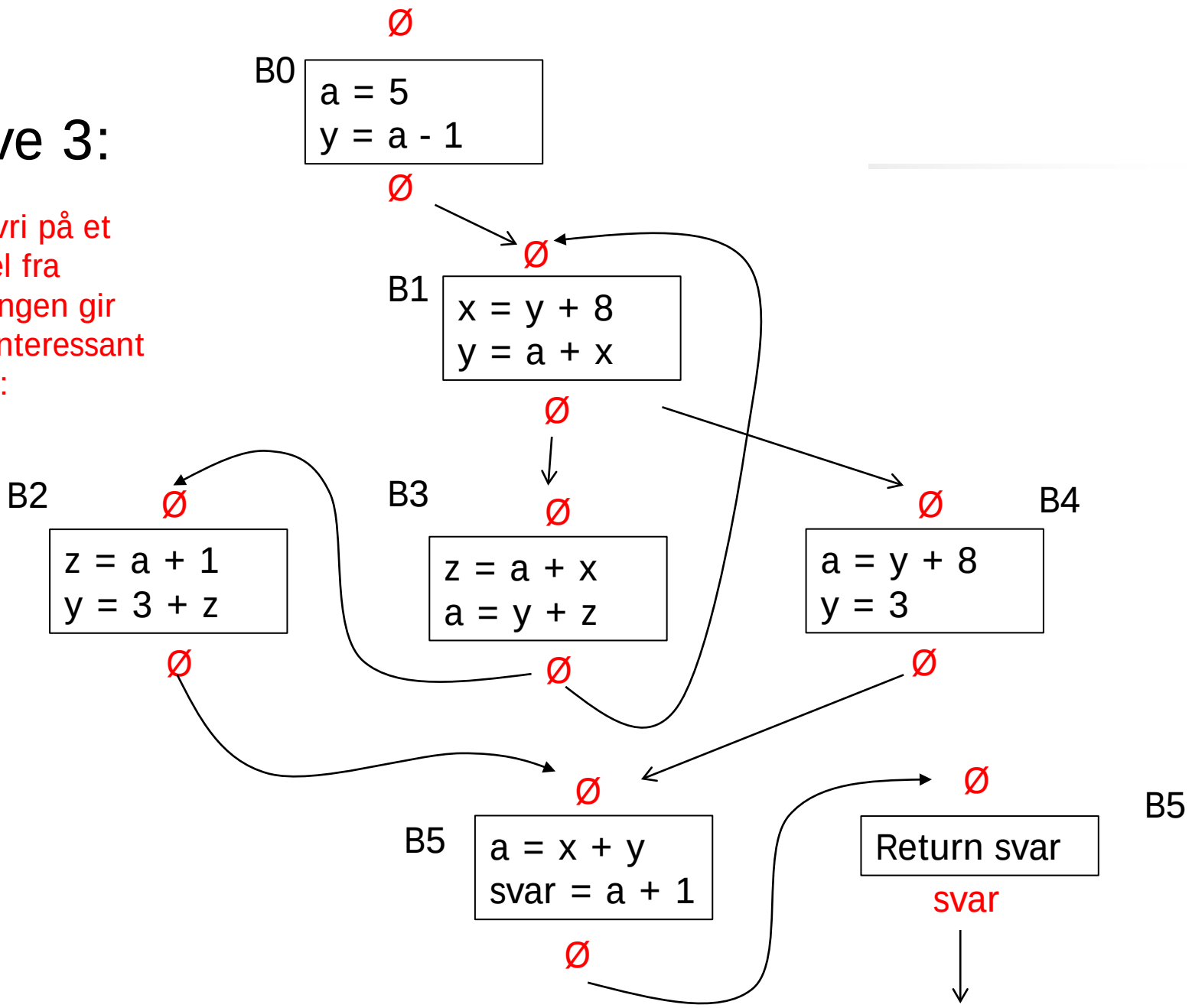
NY parameter!

For "||":



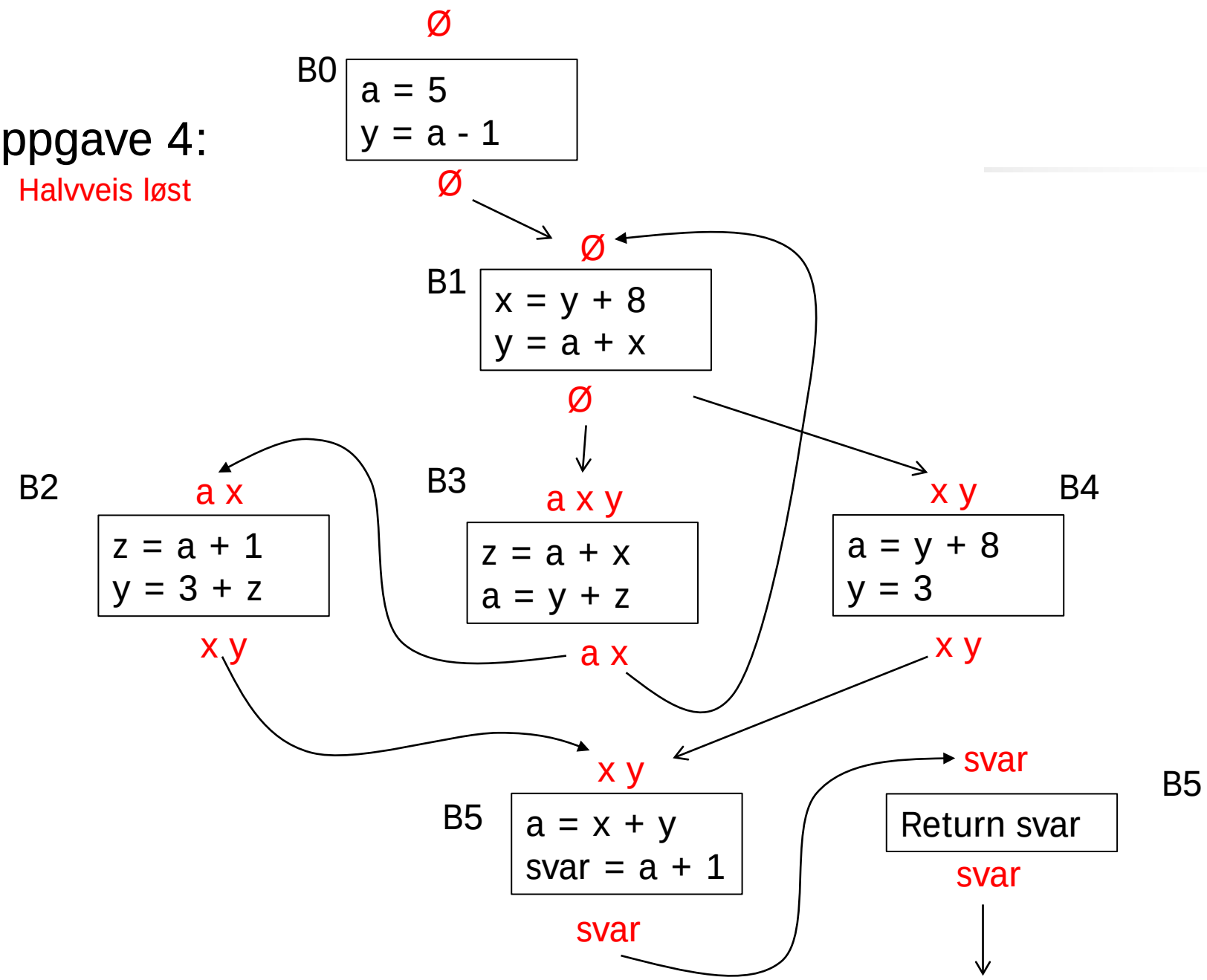
Oppgave 3:

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

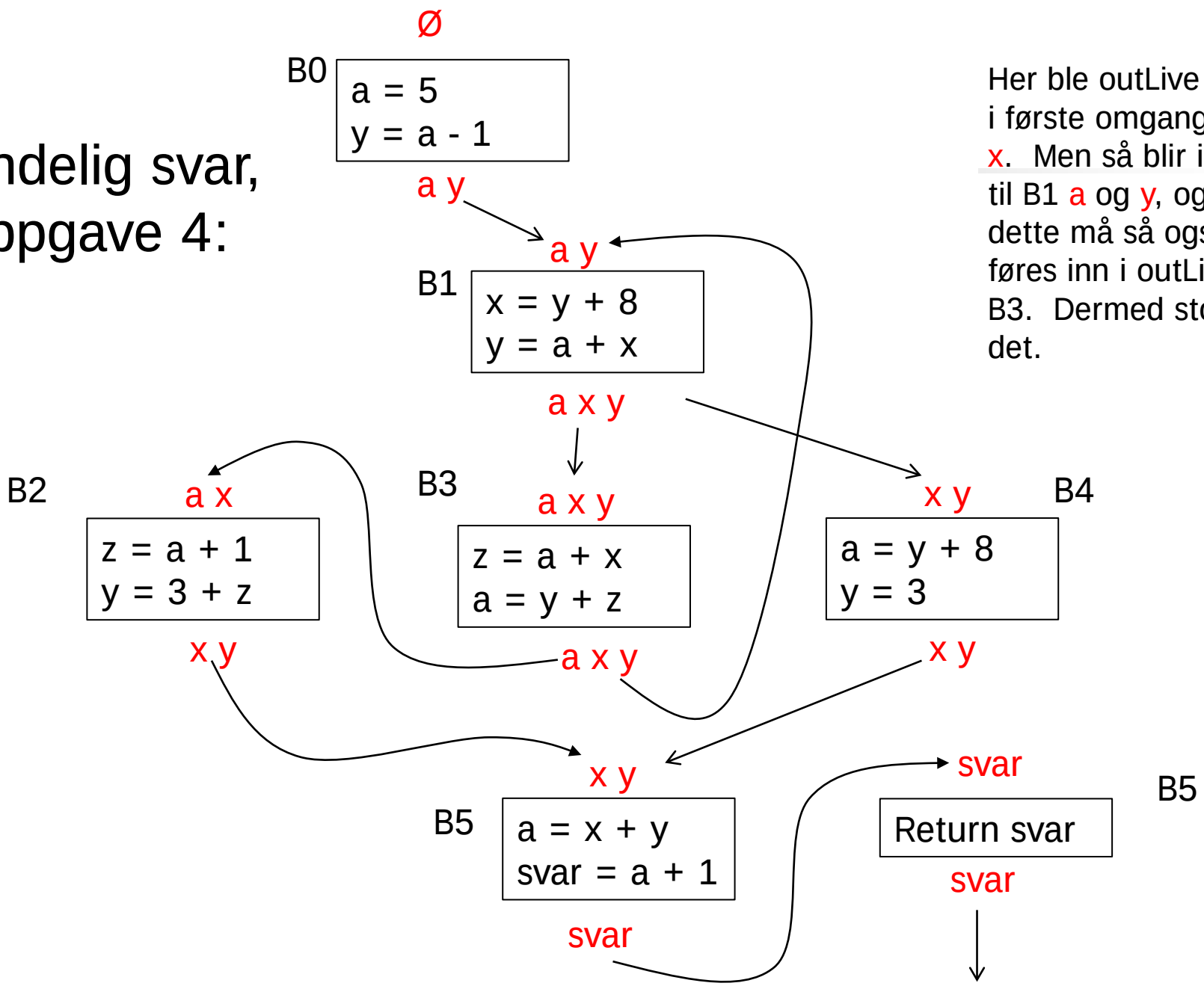


Oppgave 4:

Halvveis løst



Endelig svar, oppgave 4:



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 4, eksamen 2007

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
```

4a

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 2a, eksamen 2007

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
```

4a

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.

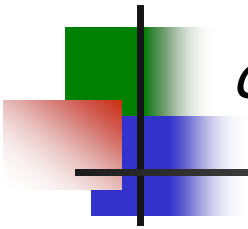
Svar:

a. Det kunne se ut som om denne var 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 fra eksamen 2009. Innledning:

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 4a og 4b, eksamen 2009.

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 4a og 4.b, eksamen 2009.

```
-------------------------------
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 4c, eksamen 2009

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.

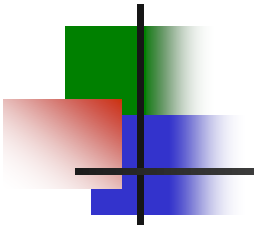
Formuler en generell regel som skal brukes lokalt på alle basale blokker for å sjekke om det produsenten av TA-kode sier er riktig, for et gitt TA-program. Vi antar her at alle variable er døde når programmet over stopper (etter siste instruksjon).

Oppgave 4.c, eksamen 2009

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: ”For alle temporærvariable som blir avlest (i boka: ”used”) i den basale blokken, så må de bli gitt en verdi (i boka: ”be defined”) før de blir avlest”. Man kunne alternativt si: ”Ingen temporærvariable må ha noen ’neste bruk’ (i boka: ”next use”) ved starten av den basale blokken”.



Oppgave 4d , eksamen 2009

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

Oppgave 4.d , eksamen 2009

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 4d, eksamen 2010

Arne har sett på kodegenerings-algoritemen 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.

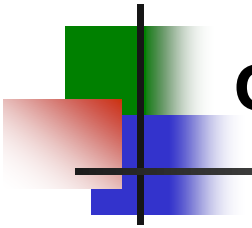
Svar 4d, eksamen 2010

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
```

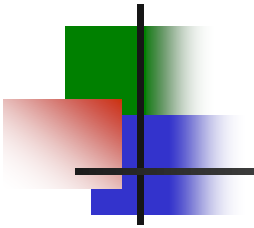
.



Oppgave 4 fra eksamen 2011. Innledning:

Vi skal her se på verifikasjon (omtrent som i en Java/JVM-loader) av en enkel type P-kode. Den har få instruksjoner, og alle verdier er heltall.

Vår P-kode utføres på vanlig måte, med en stakk med verdier under utførelsen. Under er v en programvariabel, og L er adressen til et sted i programmet.
Vår spesielle P-kode har følgende instruksjoner:

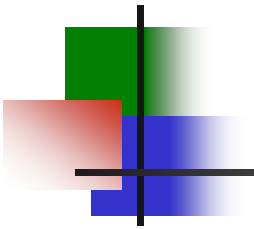


Til oppgave 4 fra eksamen 2011

Vår spesielle P-kode har følgende instruksjoner:

- lda v Henter adressen til variablen v opp på toppen av stakken. En adresse er også et heltall.
- ldv v Henter verdien av variablen v opp på toppen av stakken
- ldc k Henter konstanten k opp på stakken
- add Legger sammen de to øverste verdier på stakken, fjerner (popper) dem fra stakken og legger svaret på toppen av stakken.
- sto Her tolkes det som ligger på toppen av stakken som en verdi, og det nest øverst som en adresse. Instruksjonen kopierer verdien inn til den angitte adressen i lageret, og popper både verdien og adressen.
- jmp L Hopp til program-adressen L
- jge L (og likeledes: jgt L, jle L, jlt L, jeq L, jne L) Denne instruksjonen er litt enklere enn vanlig, nemlig slik: Om verdien på toppen av stakken er større eller lik 0 så hoppes det (og tilsvarende for de andre fem). Verdien på toppen av stakken poppes uansett om det hoppes eller ikke.
- lab L Angir at program-adressen L er på dette stedet i programmet.

Oppgave 4a fra Eksamen 2011



Vi tenker oss at vi skal lage en verifikator for programmer i vår P-kode (altså for sekvenser av P-instruksjoner).

Angi flest mulig ting som denne verifikatoren bør/kan teste angående et gitt slikt program. Forklar også i hvilken forstand et P-kode-program er ”riktig” om det passerer testen din.

Svar 4a, eksamen 2011

Alt som skal på stakken er her heltall (også adresser), så verifikatoren kan ikke se noen typemessig forskjell på forskjellige stakker. Det eneste den kan se på er størrelsen av stakken.

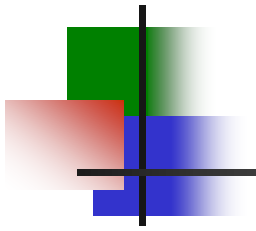
Vi vet at vi skal starte med tom stakk på toppen av programmet, og det er naturlig å sjekke følgende tre ting:

(1) At det alltid er nok elementer på stakken til å gjøre en angitt operasjon. For eksempel må de være minst to elementer på stakken for å gjøre en add-operasjon, og minst ett element for å gjøre en jge-instruksjon.

(2) Sjekke at stakken er tom ved slutten av programmet.

(3) Når man gjør et hopp til en gitt label L (eller kommer til lablen L fra forrige instruksjon) skal stakken alltid ha samme størrelse.

Om et program er kommet vel gjennom disse testene vet vi at stakken alltid vil ha de nødvendige heltall på stakken når en operasjon skal utføres, og det gjelder samme hvilken vei man går gjennom programmet.



Oppgave 4b fra eksamen 2011

Under står tre programmer i vår P-kode. Sjekk for hver av dem om de passerer testen din, og angi hva som eventuelt går galt om de ikke gjør det.

Program 1:

```
lda x
ldv y
ldv z
jge L1
add
add
ldc  5
lab L1
ldc 8
sto
```

Program 2:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
lab L1
sto
```

Program 3:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
ldv u
lab L1
sto
```

Svar 4.b, eksamen 2011

Program 1:

```
lda x
ldv y
ldv z
jge L1
add
add
ldc  5
lab L1
ldc 8
sto
```

Program 2:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
lab L1
sto
```

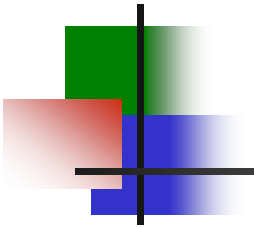
Program 3:

```
lda x
ldv y
ldv z
jge L1
ldc  5
add
ldv u
lab L1
sto
```

Det første programmet er galt fordi det bare vil være ett element på stakken ved den siste add-instruksjonen.

Det andre programmet er OK

Det tredje programmet er galt fordi de to veiene som fører til lablen L1 gir stakkdybde 3 (uten hopp) og 2 (med hopp).



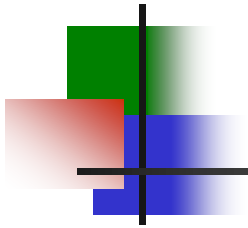
Oppgave 4c fra eksamen 2011

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 diskriptorer.

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

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 diskriptor 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).



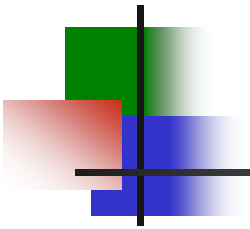
Svar oppgave 4c, eksamen 2011

Grovt sett:

Dersom språk-reglene sier at man må utføre uttrykk i rekkefølge fra venstre mot høyre, så må man lage 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 diskriptor. 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.



Oppgave 4d, eksamen 2011

*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 diskriptorene på stakken skal inneholde, og hva du eventuelt trenger av andre typer diskriptorer. Vi antar at vi kan tillate oss å lete gjennom alle kompilator-stakkens diskriptorer 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 diskriptorer.

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

Svar oppgave 4d, eksamen 2011

- *Diskriptorene på stakken bør hvertfall inneholde følgende:*

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

Diskriptorene 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-diskriptor 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 diskriptorene viktige.