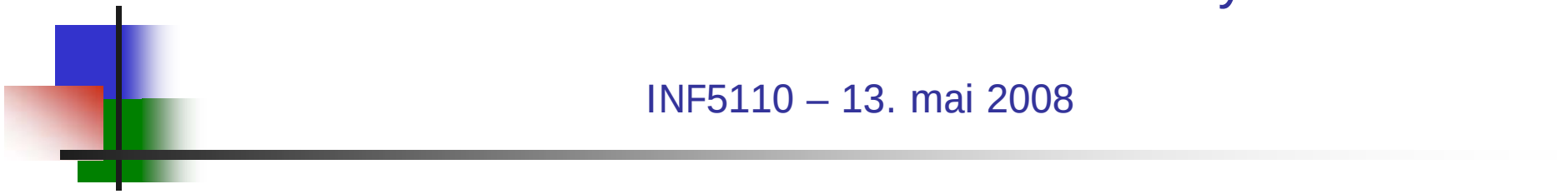
Avsluttende om kodegenerering: Tilleggsnotat fra AHU
Samt: En oversikt over bruk av Javas Byte-kode



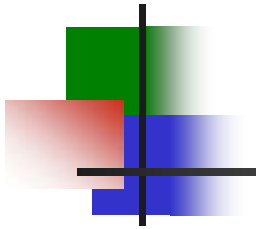
INF5110 – 13. mai 2008

Stein Krogdahl,
Ifi UiO

Endelig pensumliste kommer i morgen 14/5

Program framover:

Torsdag 22. mai: Gjennomgåelse av fjorårets eksamen
(Ligger på forelesningsplanen. Forsøk å løse den på tre timer)

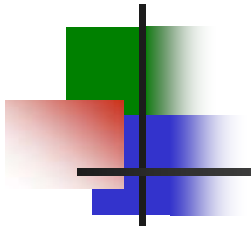


ASU, kap 9.5:
Vi generer kode for én og én basal blokk

- Da er det lett å holde orden på hvilke verdier som er i hvilke registre etc. ned gjennom blokka

- Mellom hver basal blokk sørger vi for:
  - Alle verdier av program-variable ligger ”ute” i den variablens lokasjon 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, og må derfor anta at at alle program-variable er i live ved slutten av en basal blokk.



”Neste bruk” og ”i live” innen én basal blokk

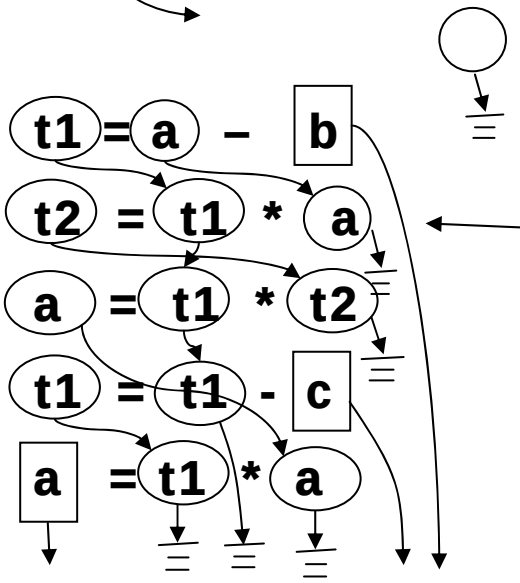
- Før man gjør kodegenerering for en basal blokk er det lurt å skaffe seg oversikt over bruk av variable (temporære og andre) i blokka:
  - En variabel-forekomst kan ha en ”neste-bruk” (derved ”i live”):
    - Def: Den verdien den har blir brukt senere i samme basale blokka.
    - Da kan det være lurt 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 den kan bli/blir brukt i senere blokker
    - 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” (og hvor, med det brukes ikke av vår algoritme). Disse er ”i live”

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

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, siden temporære bare brukes innen samme blokka

Altså : I live etter blokka: ***a,b,c***

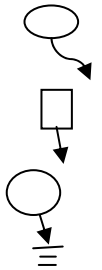
Dvs., programvariablene.

Kommentarer:

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

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:

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

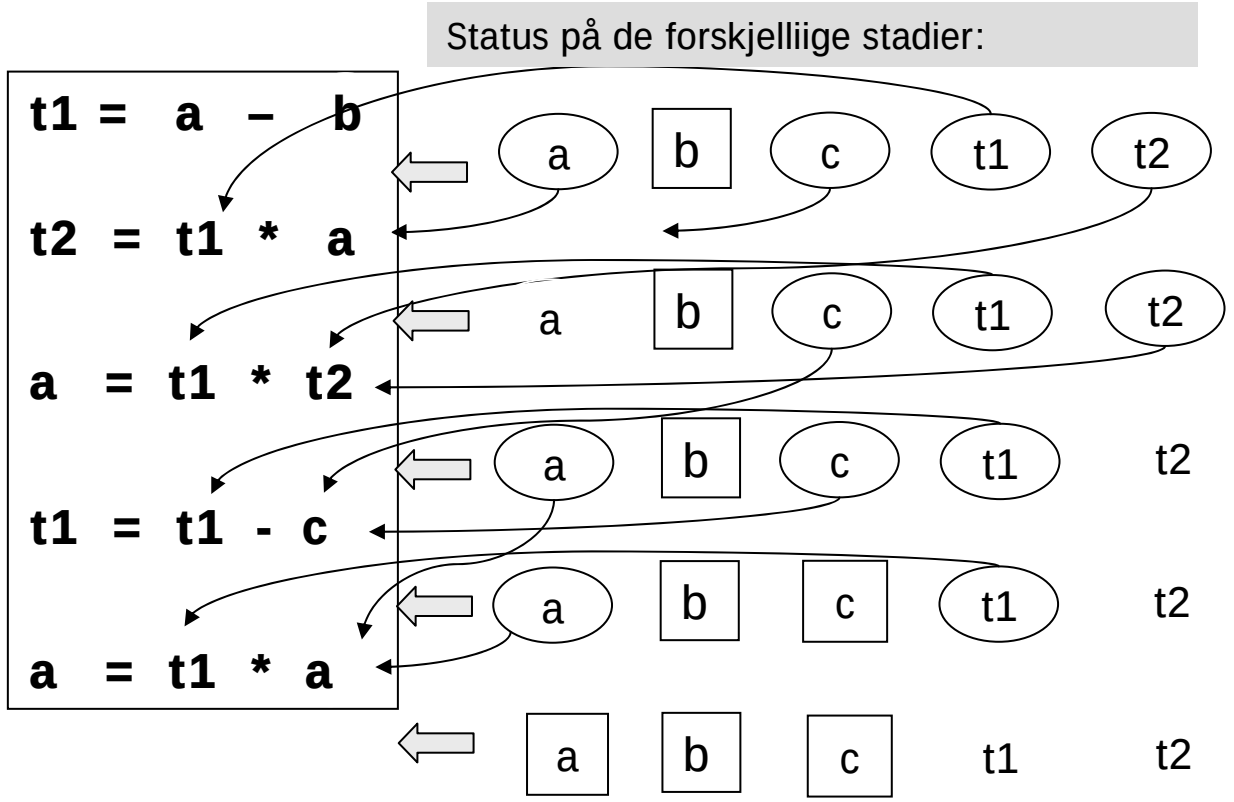
Steget (gjentas for hver TA-instr. fra sist til første):

1. Merk x i S slik x er merket i T
2. Forandre x sitt merke i T til 3
3. Merk y og z i S 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 S.

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

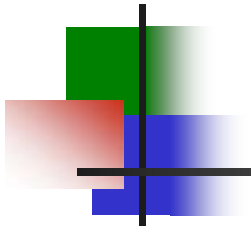
Algoritme for å finne ”neste bruk” og ”i live” innen en basic blokk

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



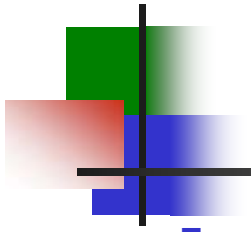
Døde variable er tegnet uten ring rundt.

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



Kodegenererings-algoritme

- Enkel algoritme: Lager maskinkode for én og én Basal Blokk
- Algoritmen holder (innefor hver Basale Blokk) beregnede verdier i registre så langt det er ønskelig og mulig (spesielt viktig om de har ”neste bruk”)
- Når programkontrollen går mellom de Basale Blokkene så ligger samtlige variabel-verdier i sine respektive hukommelses-plasser (mens temporære variable ikke ”er i live”)
- Kodegenerering for hver Basal Blokk blir da:
  - Utfør algoritmen for å finne ”neste bruk” og ”i live”
  - 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 legges verdier fra registre tilbake til sine respektive hukommelses-plasser
- Noen mangler ved algoritmen (som i stor grad lett kan rettes opp)
  - Variable som kun blir *lest* innenfor en Basal Blokk blir ikke lagt i registre, selv om det er mange 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 at registeret er ledig eller at det inneholder verdien til et antall 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
  - 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 a, 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**

(litt rettet i forhold til det utdelte)

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 Z finns // 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 ha 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 av R ikke ligger 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)

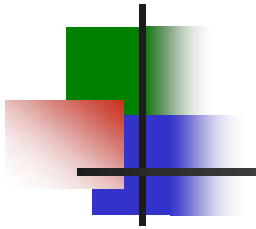
Opprinnelig verdi av X ødelegges

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 a, 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)

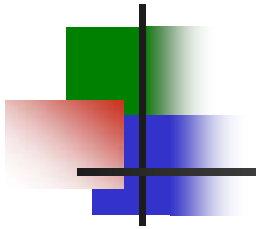


Litt om Javas class-filer og byte-kode

Ikke pensum, men anbefales lest som bakgrunnsmateriale

- Disse formatene ble planlagt fra start som en del av hele Java-ideen
  - Byte-koden gir portabilitet ved at den utføres av en interpretator (som må skrives for hver enkelt maskin, gjerne i C eller C++)
  - Gir, sammen med et standard bibliotek, et enhetlig grensesnitt til operativsystemet, grafikk, osv. fra Java
  - Samme Java-kompilator kan dermed brukes på alle maskiner
  - For effektivitet: Byte-koden blir nå ofte oversatt til maskinkode før utførelse, enten i en egen kompilerings-operasjon, eller som JIT-kompilering som en del av ”loadinga”.

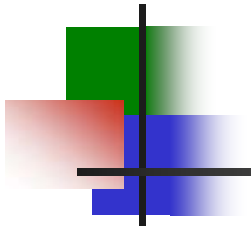
- Likner mye på den lavnivå-koden dere oversetter til i Oblig 2
  - Men hos dere blir også ”Loading” gjort samtiding som koden blir laget.
  - I Javas byte-kode blir alle navn beholdt på tekstlig form i den byte-koden som legges på fil (som klasse-filer)
  - Først når denne taes inn av loaderen blir den laget om til binær kode.



Litt om Javas class-filer og byte-kode

- Formatet av class-filene inneholder både
  - Den utførbare koden (som byte-kode = sekvenser av byte-instruksjoner),
  - Og hele strukturen av klassen, med info om navn, variable, metoder, parametere, typer, etc.
  - Disse to informasjonstypene ligger tradisjonelt på to forskjellige filer: For C og C++, på .c-filer (som oversettes til maskinkode) og på .h-filer

- En class-fil leses derved i to sammenhenger i forbindelse med kompilering/kjøring:
  - Når en annen klasse som referer til denne (f.eks. en subklasse) kompileres: Da ser man mest på struktur-delen av klasse-filen
  - Når klassen skal loades: Da ser man mest på byte-koden for hver av metodene i klassen

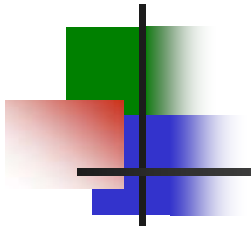


Formatet av Javas class-filer

- På hver class-fil er det bare beskrivelse av én klasse eller ett grensesnitt

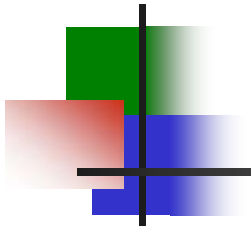
- class-filene har all informasjon om:
  - Navn på klassen, og på superklassen og implementerte grensesnitt
  - Hvilke variable klassen har, ved navn, type og synlighet
  - Hvilke metoder den har, ved navn, type, synlighet, parametere (med typer), og byte-kode-sekvens

- class-filene har også et ”navne-område” eller et ”konstant-område”
  - Her ligger alle tekster, navn, og tall-verdier (på tekstlig form) som brukes i programmet, pent etter hverandre.
  - Når disse skal brukes ellers i klassefila (f.eks. i selve byte-koden) så angir man bare indeksen til dette navnet i navne-området.



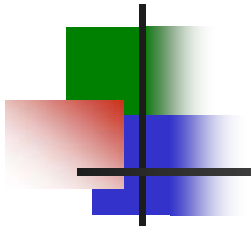
Formatet av byte-koden

- Byte-koden har samme idé som P-koden, ved at arbeids-dataene skal ligge på en stakk under utførelsen
- Funksjons-delen av instruksjonen er på én byte, altså plass til 256 instruksjoner (som faktisk er litt lite)
- Byte-instruksjonenes ”adressefelt” (der dette trengs) angis ved fullt navn (tekstlig), type og klasse-tilhørighet
  - Det eneste unntaket fra dette er lokale variable i metoder. Disse angis som relativ-adresse (i byte) i aktiverings-blokken.
- Som antydet på forrige foil: Det blir mange navn det stadig skal refereres til. Derfor ligger navnene bare én gang hver i et eget ”navne-område”, og de angis andre steder bare ved en indeks inn i dette området



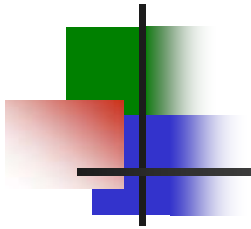
Hva foregår i en Java Virtual Machine (JVM)

- En JVM kan enten *interpretere* eller *oversette til maskinkode* (JIT, eller vanlig)
- Den består av en *loader*, en *verifikator*, samt av en *interpretator* eller en *oversetter*
- Loaderen starter med å lese inn og behandle den angitte class-fila
- Leser så etter hvert inn alle class-filer som det referers til fra denne, osv.
- Lager en ”descriptor” for hver klasse.
  - Denne vil ligge fast under den kommende utførelse av programmet
  - Alle objekter har en peker til descriptoren for sin klasse
  - Descriptoren inneholder virtuell-tabellen for klassen, en peker til descriptoren for superklassen, etc.
  - Dersom debugging eller refleksivitet: Descriptoren inneholder også info om alle variable/metoder
  - Descriptoren kan også ha en peker til et sted der selve Java-koden ligger



Mer om: Hva foregår i en JVM

- Loaderen gjør ”allokering” av variable i klassene, dvs.:
  - Går gjennom byte-kode-sekvensen og gjør hver av de tekstlige operandene om til relativadresser, og alle klasse-angivelser (f.eks. i casting og ved ”new C...”) om til pekere til klassens descriptor
  - Merk at allokering av én klasse må gjøres *før* allokering for dens subklasser
- Dersom interpretering:
  - Legger sekvensen av byte-instruksjoner (nå med tall både for funksjonsangivelse og adresser) ut i et passelig format
  - Starter å interpretere dette
- Dersom oversetting:
  - Oversetter sekvensen byte-instruksjoner til maskinkode for gitt maskin
  - Kan også gjøres som en vanlig ”forhåndskompilering”, eller en JIT-kompilering (Just-In-Time) i forbindelse med at programmet skal startes opp.
- Kan også gjøre noe midt i mellom:
  - Interpretere først, men etter hvert oversette de metoder som brukes mest.
- Finnes også Java-kompilatorer som hopper over hele byte-kode-steget
  - Får da en tradisjonell kompilator, som kan lage effektiv kode
  - Men det er mer problematisk å koble seg til Javas standard-bibliotek etc.



Verifikatoren

- Denne kan gå gjennom klassefiler og sjekke at de er konsistente
  - Spesielt sjekke at bytekoden er konsistent (se under)
  - Og at den ikke gjør noe den ikke har autorisasjon til
  - Dette er spesielt aktuelt for class-filer som hentes inn over nettet

- Hva gjør verifikatoren med byte-koden:
  - ”Simulerer” hele tiden hva som vil ligge på stakken under utførelsen
  - Sjekker at det som ligger på toppen av stakken hele tiden stemmer typemessig etc. med den instruksjonen som skal utføres
  - Sjekker at når det gjøres hopp så er stakken helt lik der det hoppes fra og der det hoppes til.
  - Sjekker at de operasjonene som gjøres er lovlige (i henhold til autorisasjon)

Typisk Byte-kode, ferdig til interpretering:

Merk: Også funsjonskodene (f.eks. *sipush, iconst_2 og goto*) er nå gjort om til tall-koder (mellom 0 og 255)

Java-program:

```
outer:
for (int i = 2; i < 1000; i++) {
   for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
   }
   System.out.println (i);
}
```

```
0: iconst_2
1: istore_1    // "i" har reladr 1
2: iload_1
3: sipush 1000
6: if_icmpge 44
9: iconst_2
10: istore_2  // "j" har reladr 2
11: iload_2
12: iload_1
13: if_icmpge 31
16: iload_1
17: iload_2
18: irem # remainder
19: ifne 25
22: goto 38
25: iinc_2, 1
28: goto 11
31: getstatic #84;  // Området for println() ?
34: iload_1
35: invokevirtual #85; // println()
38: iinc 1, 1
41: goto 2
44: return
```