MED SVARFORSLAG

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110 - Kompilatorteknikk
Eksamensdag :	Onsdag 1. juni 2011
Tid for eksamen :	14.30 - 18.30
Oppgavesettet er på :	7 sider (pluss vedlegg)
Vedlegg :	1 side (side 8 rives ut, fylles ut og leveres i "hvit" besvarelse)
Tillatte hjelpemidler :	Alle trykte og skrevne

Les gjennom ***hele*** oppgavesettet før du begynner å løse oppgavene. Dersom du savner opplysninger i oppgavene, kan du selv legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så tilfelle rede for disse forutsetningene og antagelsene.

Oppgave 1 (25%)

Vi skal se på et antall grammatikker, nemlig følgende:

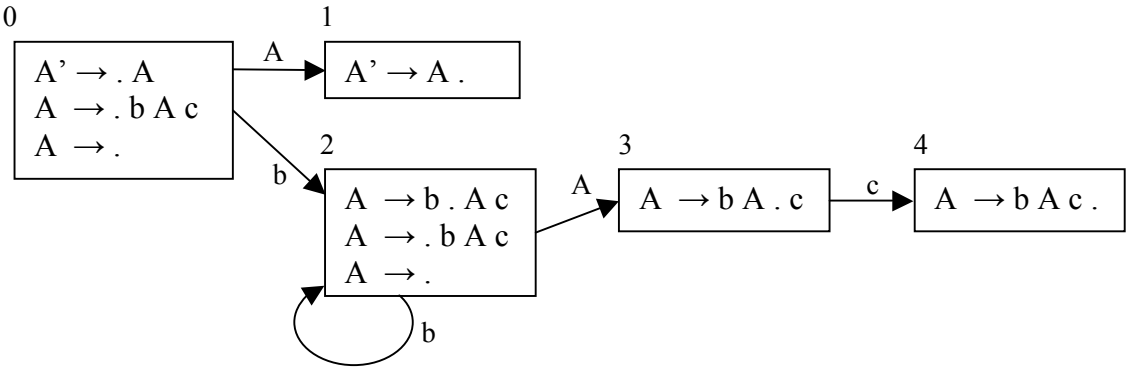
i. $A \rightarrow b\ A\ c \mid \varepsilon$
ii. $A \rightarrow b\ A\ b \mid b$
iii. $A \rightarrow b\ A\ b \mid c$

1a

Her er A startsymbol og eneste ikke-terminal, mens b og c er terminaler. For hver av de tre grammatikkene: Beregn First og Follow til A, tegn LR(0)-DFA'en (etter utvidelse med $A' \rightarrow A$), og angi om den er en SLR-grammatikk eller ikke. Angi også hvilke av grammatikkene, om noen, som er LR(0)-grammatikker.

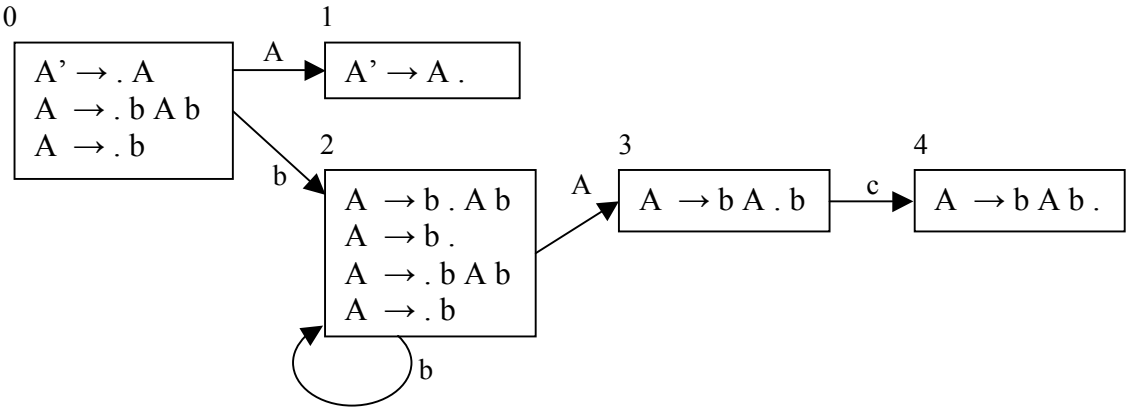
Svar 1a

Grammatikk i: $A \rightarrow b\ A\ c \mid \varepsilon$
First(A) = { b, ε } Follow(A) = { c, $ }



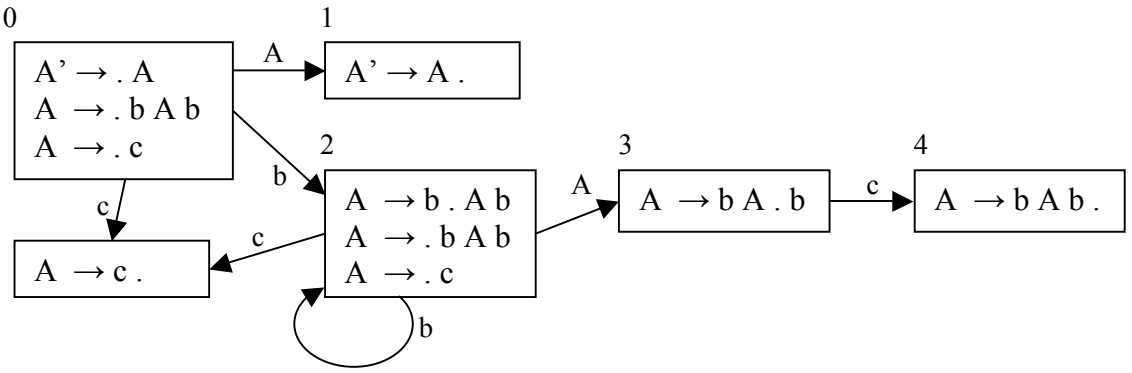
SLR(1): Det kunne være tilstand 2 som gav problemer, men siden b (eneste terminal-kant ut av tilst. 2) ikke er med i etterfølger-mengden til A, så går det bra.
LR(0): Nei. Både i tilstand 0 og 2 må man lese neste tegn for å avgjøre hva man skal gjøre.

Grammatikk ii: A → b A b | b
First(A) = { b } Follow(A) = { b, $ }



SLR(1): Nei. Siden b er med i etterfølgermengden til A vet vi ikke om vi skal redusere eller skifte i tilstand 2.
LR(0): Nei. Når den ikke er SLR(1) er den heller ikke LR(0)

Grammatikk iii: A → b A b | c
First(A) = { b, c } Follow(A) = { b, $ }



SLR(1): Ja, den er faktisk LR(0) som angitt under, og da er den også SLR(1)
LR(0): Ja, for i hver tilstand er det bare snakk om enten å skifte eller å redusere.

1b
For hver av de tre grammatikkene, avgjør om grammatikken er LR(1). Det er her mulig å avgjøre dette og forklare det uten å tegne opp LR(1)-DFA'en, men det er også en OK besvarelse om du tegner opp denne og avgjør det ut fra den.

Svar 1b
Grammatikk i: Siden grammatikken er SLR(1) er den også LR(1)

Grammatikk ii: Denne er ikke LR(1). Under en parsering etter denne grammatikken skal man gå over fra å skifte til å redusere når vi har lest inn den midterste b-en, men vi kan ikke generelt vite når vi er der om vi bare kan lese ett tegn framover.

Grammatikk iii: Siden grammatikken er LR(0) er den også LR(1)

1c
Angi for hver av grammatikkene over om det språket de genererer er regulært, og for de som er det skal du angi et regulært uttrykk for språket. For de grammatikker du mener *ikke* genererer et regulært språk, forklar hvorfor.

Svar 1c
Grammatikk i: Den generelle formen for en setning er først null eller flere b-er, som er fulgt av like mange c-er, og dette språket er ikke regulært. Det er vel OK bare å vise til at det er angitt på foilene et sted (i en fasit?) at språk der man skal ha like mange "sånne" som "slike" (f.eks. "…(((((())))))…") ikke er regulære.
Ellers kan man f.eks. argumentere slik: Et språk er som kjent regulært hvis og bare hvis man kan parsere det ved bare å huske en *endelig* mengde informasjon om den delen man *har* lest (eller være i én av et endelig antall tilstander) når man leser fra venstre mot høyre. Men i grammatikk i må man kunne huske hvor mange b-er man har lest når man finner første c, og det antallet kan være ubergrenset stort.

Grammatikk ii: Den generelle setningen her er rett og slett et odde antall b-er. Dette språket er regulært siden det kan beskrives ved følgende regulære uttrykk b (b b)*

Grammatikk iii: Den generelle setningen her er to sekvenser med like mange b-er, og med en c i midten. Dette språket er ikke regulært, av samme grunn som for grammatikk i.

1d
Tegn opp en parseringstabell for grammatikk i, og sørg for at den blir uten konflikter. Angi så en steg-for-steg LR-analyse av setningen "bbcc", på samme måte som øverst på side 213 i boka (tabell 5.8).

Svar 1d
Ut fra LR(0)-DFA'en og de SLR(1)-betrakninger som er gjort over, får vi følgende entydige tabell:

	b	c	$	A
0	s2		r(A→ε)	1
1			accept	
2	s2	r(A→ε)	r(A→ε)	3
3		s4		
4		r(A→bAc)	r(A→bAc)	

Parseringen blir da som følger:

$ 0	b b c c $	
$ 0 b 2	b c c $	
$ 0 b 2 b 2	c c $	Så skjer det mest interessante:
$ 0 b 2 b 2 A 3	c c $	
$ 0 b 2 b 2 A 3 c 4	c $	
$ 0 b 2 A 3	c $	
$ 0 b 2 A 3 c 4	$	
$ 0 A 1	$	Og dette gir "accept"

Oppgave 2 (20%)

Anta at vi har et objekt-orientert språk, hvor en virtuell metode i en klasse kan redefineres ("overriding") i en subklasse av denne klassen. En virtuell metode deklareres med en `virtual` modifier, mens en redefinisjon deklareres med modifieren `redef`. Metoder uten `virtual` er vanlige metoder og kan altså ikke redefineres. Merk at dette ikke er helt som i Java. I Java er alle metoder virtuelle, mens her gjelder det bare de som har modifieren `virtual`.

Det følgende er klasser definert i dette språket:

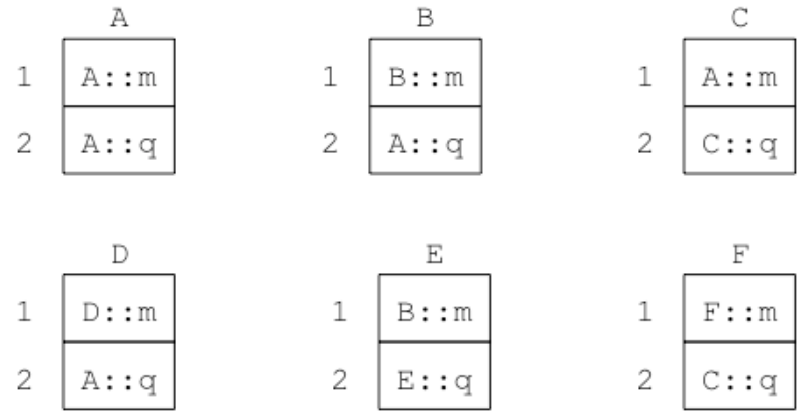
```
class A {
  virtual void m(int x,y){...}
  void p(){...}
  virtual void q(){...}
}
class B extends A{
  redef void m(int x,y){...}
  void r(){...}
}
class C extends A{
  redef void q(){...}
}
class D extends B{
  redef void m(int x,y){...}
}
class E extends B{
  redef void q(){...}

class F extends C{
  redef void m(int x,y){...}
```

2a
Vi antar nå først at klassen for et gitt objekt bestemmer, på vanlig måte, hvilken versjon av en virtuell metode som kalles.

Lag virtuell-tabellene for klassene `A`, `B`, `C`, `D`,`E` og `F`. For hvert element i tabellene skal du bruke notasjonen `A::m` for å angi hvilken metode som gjelder. Indeksen i disse tabeller starter på 1.

Svar 2a



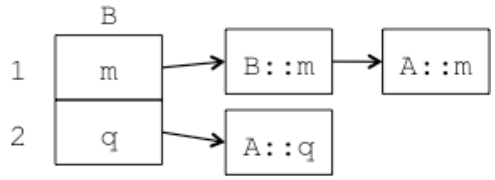
2b
I resten av oppgaven skal vi for virtuelle metoder ha den semantikk at en redefinert virtuell metode, for eksempel `m`, skal, som det første den gjør, kalle den tilsvarende virtuelle eller redefinerte metode (dvs `m`) i den nærmeste superklasse som har en slik, før den utfører sin egen body. Dette vil i sin tur føre til at redefinerte eller virtuelle metoder `m` i videre superklasser utføres.

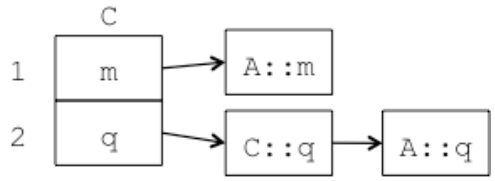
Dette kunne man implementere ved å sette inn det riktige kall som første statement i bodies på redefinerte metoder. Men semantikken rundt parameteroverføring skal her være litt spesiell slik at denne enkle måten ikke vil fungere. Parametrene som gis med i det opprinnelige kallet skal nemlig gå direkte som parametre til den metoden som utføres først, altså den som er merket `virtual` i programmet. Når denne er ferdig utført skal de verdiene som da står i dens parametervariable overføres som aktuelle parametre til den neste dypere redefinerte metode, osv. Dette gjør at stakken av kall må settes opp først, og de aktuelle parametre må gis til den første virtuelle metode som skal utføres.

Hvis for eksempel `m` kalles med `m(1,2)` på et `D`-objekt, så skal stakken settes opp og de aktuelle parametre gis til aktiveringsblokken tilsvarende `A::m`, og utførelsen skal starte med utførelsen av `A::m`. Ved exit av `A::m` skal verdiene av parametrene `x` og `y` gis som aktuelle til den versjon av `m` som da skal utføres.

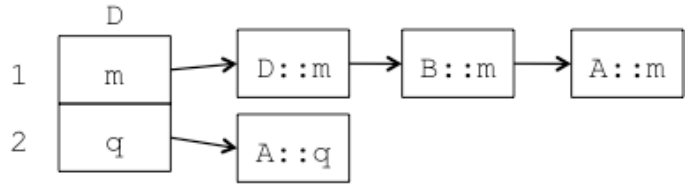
For å implementere denne nye semantikk utvides virtuell-tabellen, slik at det for hvert indeks blir en liste av metode-angivelser. Denne listen vil dermed angi sekvensen av de metoder som skal kalles.

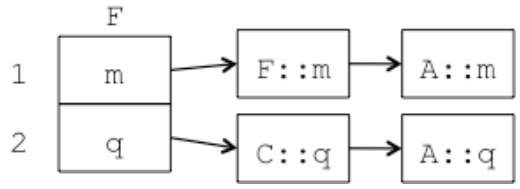
Tegn de nye virtuell-tabeller for klassene `D` og `F`. Tabellene for `B` og `C` er gitt under. For metode-angivelser brukes samme notasjon som før.





Svar 2b





2c

I denne del av oppgaven skal du skissere hvordan stakken i **2b** kan lages ved hjelp av de nye virtuell-tabeller. Du kan anta at du har en run-time rutine `makeActivationRecord(metode)`. Denne tar som parameter en metode-angivelse (for eksempel `A::m`) med nok informasjon til å lage en aktiveringsblokk med riktig størrelse, men du skal skissere hvordan control-link og retur-adresse settes i aktiveringsblokken.

Anta at den aktuelle virtuell-tabell holdes i en variabel med navn `vt`, den aktuelle metoden holdes i en variabel `method`, og funksjonen `index(method)` gir deg index i `vt`. Anta videre at inngangen i tabellen gir en peker til første metode-angivelse, og at hver av disse har en `next` peker. For metode-angivelsen som svarer til den virtuelle metode hvor den defineres for første gang (i eksemplet her `m` i `A`) er denne peker `none`.

Du kan gjerne illustrere resultatet for et kall `m(1,2)` på et `D`-objekt, men om resten er riktig er det OK uten.

Svar 2c

```
caller = fp;
method = vt(index(method));
while method =/= none do {
  ar = makeActivationRecord(method);
  ar.conrolLink = caller;
  ar.returAdresse = første statement i metoden som tilsvarer fp
  method = method.next;
  caller = ar;
}
```

2d

Overføring av parametre mellom de enkelte utførelser av en virtuell metode kan åpenbart ikke gjøres som en del av det å sette opp stakken i **2c**, men må gjøres ved bl.a. å sette inn ekstra kode i metoder merket `virtual` eller `redef`. Du må også innføre en ekstra variabel i de aktuelle aktiveringsblokkene. Hvilken kode skal settes inn og hvor? Koden kan gjøre bruk av alle deler av den aktuelle aktiveringsblokk.

Svar 2d

Utfør følgende kode, som det siste før exit, i alle aktiveringer bortsett fra den dybeste:

```
controlLink.x = x
controlLink.y = y
```

Oppgave 3 (30%)

Det følgende er et fragment av en grammatikk for et språk med klasser. En klasse kan *ikke* ha noen superklasse, men den må implementere en eller flere interfacer:

class → **class** ***name*** **implements** *interfaces* { *decls* }
decls → *decls* **;** *decl* | *decl*
decl → *variable-decl* | *method-decl*
method-decl → *type* ***name*** **(** *params* **)** *body*
type → **int** | **bool** | **void**
interfaces → *interfaces*, *interface* | *interface*
interface → ***name***

Ord i kursiv er ikke-terminaler, ord og tegn i **fet skrift** er terminal-symboler, mens ***name*** representerer et navn som scanneren leverer. Det kan antas at ***name*** har attributtet 'name'.

Det som er spesielt med dette språk er at de av en klasses metoder som har samme navn som en av interfacene klassen implementerer er 'konstruktører' for klassen. Det kan i klassen gjerne være flere metoder, som har samme navn som et interface, men da med forskjellige parametre; dette er imidlertid ikke temaet i denne oppgave. Generering av objekter har formen "new <class-name>.<interface-name>(<actual parameters>)", da forskjellige klasser kan implementere samme interfacen.

En krav i dette språket er at konstruktører må være spesifisert med typen void, og det er dette kravet som skal sjekkes med de semantiske regler du skal sette opp.

Lag semantiske regler for dette kravet i følgende fragment av en attributtgrammatikk.

For å lage reglene kan du bruke de funksjoner og de mengder du trenger, bare du definerer dem.

Besvar dette spørsmålet ved å bruke vedlegget side 8.

Grammar Rule	Semantic Rule
class → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}**	
$decls_1$ → $decls_2$ **;** *decl*	
decls → *decl*	
decl → *method-decl*	
method-decl → *type* ***name*** **(** *params* **)** *body*	
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void
$interfaces_1$ → $interfaces_2$, *interface*	
interfaces → *interface*	
interface → ***name***	*interface*.interfaceName = ***name***

Svar 3

Grammar Rule	Semantic Rule
class → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}**	*decls*.setOfInterfaceNames = *interfaces*.setOfInterfaceNames
$decls_1$ → $decls_2$ **;** *decl*	$decls_2$.setOfInterfaceNames = $decls_1$.setOfInterfaceNames *decl.setOfInterfaceNames* = $decls_1$.setOfInterfaceNames
decls → *decl*	*decl.setOfInterfaceNames* = *decls*.setOfInterfaceNames
decl → *method-decl*	*method-decl.setOfInterfaceNames* = *decl*.setOfInterfaceNames
method-decl → *type* ***name*** **(** *params* **)** *body*	***if*** *method-decl*.setOfInterfaceName.*has(****name***.name*)* ***then if*** *(****not****(type*.type = void*))****then*** *error("constructor not of type void")*
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void
$interfaces_1$ → $interfaces_2$, *interface*	$interfaces_1$.setOfInterfaceNames= $interfaces_2$.setOfInterfaceNames + *[interface*.interfaceName*]*
interfaces → *interface*	*interfaces*.setOfinterfaceNames.insert*(* *interface*.interfaceName*)*
interface → ***name***	*interface*.interfaceName= ***name***

Oppgave 4 (25%) (Jeg kunne her tenke meg at hver av oppgavene får samme % *etter* oppdeling av 4a i 4a1 og 4a2, altså 5% på hver).

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:

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.

4a *(Jeg ser at denne burde vært delt i to oppgaver, og under kommer først den opprinnelige oppgaven, og så den oppdelte utgaven (delt i 4a1 og 4a2). Merk at den siste setningen i den opprinnelige er tatt med i 4a1, men den ble vel egentlig et litt dummy spørsmål)*
(Opprinnelig 4a:) 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, og skisser hvilke datastrukturer m.m. du vil bruke for å utføre testen. Beskriv tingene direkte i forhold til vår spesielle P-kode. Et program skal både starte og avslutte med tom stakk. Du kan anta at programmets hoppinstruksjoner faktisk går til en instruksjon i programmet, så dette behøver du ikke teste

Forklar også i hvilken forstand et P-kode-program er ”riktig” om det passerer testen din.

4a1 *(Oppdelt utgave)*
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 4a1
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:

- 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.
- Sjekke at stakken er tom ved slutten av programmet.
- 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 denne testen 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.

4b2 *(Oppdelt utgave)*
Skisser hvilke datastrukturer m.m. du vil bruke for å utføre testen. Beskriv tingene direkte i forhold til vår spesielle P-kode. Et program skal både starte og avslutte med tom stakk. Du kan anta at programmets hoppinstruksjoner faktisk går til en instruksjon i programmet, så dette behøver du ikke teste.

Svar 4a2 *(Det var jo egentlig bare spørsmål etter datastrukturen, men for å forstå den må man jo*

også kjenne bruken. Om datastrukturen og antydninger av hvordan den skal brukes henger sånn rimelig sammen, så skal vi vel ikke forlange så mye mer)
Det er det siste punktet i svaret på 4a1 som krever litt ekstra, og man bør ved hver "lab L" i programmet gjøre plass til en liten data-record som her kalles S(L) ("status ved L) som har én variabel SD (stakkdybde, som fra starten er -1) og en boolsk variabel "SV" (som fra starten er FALSE, og som sier om man har sjekket programmet videre fra L). Dessuten trenger vi en global mengde av labler som vi kaller "LabelKurven".

Utføring av verifikasjonen: De to første punktene over går greit å sjekke ved bare løpende å holde greie på stakkdybden mens man "utfører" programmet. Selve testen startes så med tom stakk, og man begynner å "utføre programmet" fra toppen f.eks. slik som angitt under. (Dette er egentlig bare et søk gjennom alle kantene i en graf, der de betingede hoppene er noder med to ut-kanter, og der labler er noder med flere inn-kanter. Søket kan også godt gjøres rekursivt, men under gjøres det mer som bredde først. Hoved-vitsen er altså å sjekke at alle veier til hver node har samme stakk-dybde):

- Man har altså en løpende stakkdybde (heltall), og ved hver instruksjon gjør man de forandringer av denne som den aktuelle instruksjonen foreskriver (og man sjekker hele tiden at det alltid er nok elementer på stakken til den aktuelle instruksjonen, ellers forkastes programmet).

- Ved hver "lab L"-instruksjon gjør man følgende test: Anta at stakkdybden langs den veien du har fulgt vil være *sd* ved L. Dersom lablens *SD* er -1, setter vi *SD* til *sd* (dette er første gang vi hører om denne lablen, og vi angir at vi nå har sett en vei dit som har stakkdybde *SD=sd*). Ellers tester vi at *sd* er lik *SD*, og om det ikke stemmer skal programmet forkastes (det finnes da to veier til lablen som har forskjellige *SD*).

  Til slutt sjekker vi om *S(L).SV* er FALSE, og i så fall setter vi den til TRUE, og fortsetter på vanlig måte. Om *S(L).SV* er TRUE avslutter vi sjekken på dette punkt (det er gjort allerede), og henter en label L (med *S(L).SV* = FALSE) fra "Label-Kurven". Om kurven er tom er vi ferdig med hele sjekken, ellers går vi til L, setter vår løpende stakkdybde til S(L).SD, setter S(L).SV til TRUE, og fortsetter på vanlig måte.

- Om man kommer til et betinget hopp til L ser man først på S(L).SD og gjør en test av denne slik som angitt i forrige punkt. *Om S(L).SV* = FALSE legger vi L i LabelKurven (vi må starte herfra senere). Deretter fortsetter man på vanlig måte med instruksjonen etter det betingede hoppet.

- Om man kommer til et ubetinget hopp "jmp L" gjør man også først den samme testen som angitt over. *Om S(L).SV* er FALSE setter vi den til TRUE og fortsetter sjekken fra L med den aktuelle stakkdybden. Om *S(L).SV* er TRUE gjør vi som over: Vi avslutter sjekken på dette punkt, og henter en label L (med *S(L).SV* = FALSE) fra "Label-Kurven". Om kurven er tom er vi ferdig med hele sjekken, ellers går vi til L setter vår løpende stakkdybde til *S(L).SD*, setter *S(L).SV* til TRUE, og fortsetter på vanlig måte.

4b
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 4b
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).

4c
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 4c
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.

4d
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 4d
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), eller
- 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.

Lykke til!

Stein Krogdahl og *Birger Møller-Pedersen*

Vedlegg til besvarelse av Oppgave 3

Kandidat nr: .....

Dato: .....

Grammar Rule	Semantic Rule
class → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}**	
$decls_1$ → $decls_2$ **;** *decl*	
decls → *decl*	
decl → *method-decl*	
method-decl → *type* ***name*** **(** *params* **)** *body*	
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void
$interfaces_1$ → $interfaces_2$, *interface*	
interfaces → *interface*	
interface → ***name***	*interface*.interfaceName = ***name***