MED SVARFORSLAG

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110 - Kompilatorteknikk
Eksamensdag :	Onsdag 6. juni 2012
Tid for eksamen :	14.30 - 18.30
Oppgavesettet er på :	6 sider (pluss vedlegg)
Vedlegg :	1 side (side 7 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. Deler av oppgave 3 besvares ved bruk av vedlegg.

Oppgave 1 (25%)

Vi skal se på følgende grammatikk G1:

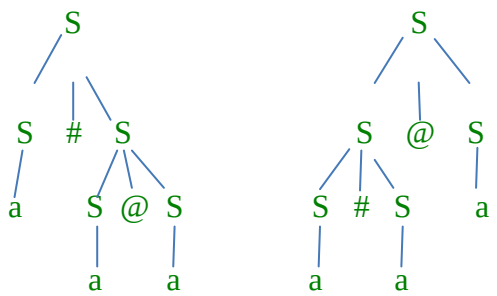
S → a | S # S | S @ S

Her er S startsymbol og eneste ikke-terminal, mens a, # og @ (samt avslutnings-symbolet $) er teminalsymboler.

1.a
Gi en konkret begrunnelse for at G1 er flertydig.

Svar 1.a

Definisjonen av at en grammatikk er *flertydig* er at det finnes en setning generert av grammatikken (altså i L(G1)) som har minst to konkrete syntakstrær (parseringstrær). Vi legger legger merketil at denne likner veldig på flertydig utgave av uttrykk med + og *, og vi kan her se på setningen: a # a @ a, som har de to syntakstrærne under. Man kunne like gjerne brukt f.eks.: a @ a @ a



S → a | S # S | S @ S

1.b
Anta at:

- Operasjonen # har lav presedens, og er høyreassosiativ
- Operasjonen @ har høy presedens, og er venstreassosiativ

Angi en ny grammatikk G2 som er entydig, som beskriver samme språket som G1 og som gir et syntaks-tre som følger de to reglene over. Du kan innføre nye ikke-terminaler, og du behøver ikke argumentere for at G2 er entydig ut over å vise til at den likner tilsvarende entydige grammatikker i pensum.

Svar 1.b

Det er to rimelige svar, som er like gode:

```
S -> T # S | T          S -> T # S | T
T -> T @ F | F          T -> T @ a | a
F -> a
```

Disse, og spesielt det til venstre, er satt opp etter samme prinsipp som på side 119 i læreboka

1.c
Vi ser på grammatikkene G1, G2, samt følgende grammatikk G3 (der + er et nytt terminal-symbol):

S → a | S # S | S @ S | + S +

Angi for hver av språkene L(G1), L(G2) og L(G3) om de *er* eller *ikke* er regulære. Forklar, og angi et regulært uttrykk for de som eventuelt er regulære.

Svar 1.c

G1 og G2 er jo samme språket, og det er regulært og kan beskrives f.eks. slik: a ((# | @) a)*

For G3 er alle setninger som kan dannes med bare «+» og «a» følgende:

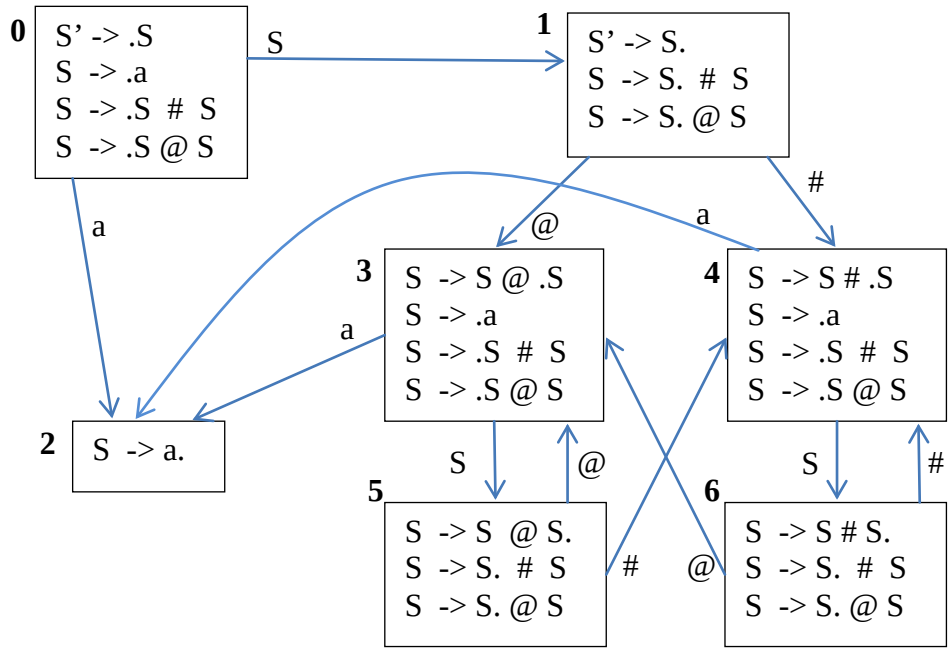
a +a+ ++a++ +++a+++ …

Det må her altså være like mange «+»-er både foran og bak «a»-en. Det er kjent fra pensum at et slikt språk ikke er regulært. Ser man på hele språket til G3 vil setningene også inneholde «@» og «#», men «+»-ene vil komme inn på en liknende måte som over, så språket er ikke regulært.

1.d
Tegn opp LR(0)-DFA'en til den flertydige grammatikken G1 (med vanlig bruk av S').

Svar 1.d

G1:
S' → S
S → a | S # S | S @ S



1.e
Beregn First og Follow til S i G1 (med vanlig bruk av $). Angi så hvilke tilstander i DFA'en fra **1.d** som har:

A. konflikter som *ikke* kan løses med LR(0)-betrakninger, men som *kan* løses med SLR(1)-betrakninger. Forklar.
B. konflikter som ikke kan løses med SLR(1)-betrakninger. Forklar.

Svar 1.e

Gjelder alltid for S'

First(S) = { a }
Follow (S) = { # @ $ }

First(S') = First(S)
Follow(S') = { $ }

G1:
S' → S
S → a | S # S | S @ S

A. Det er en LR(0)-konflikt i tilstand 1, som kan løses i SLR(1) ved at man reduserer ("accept" er reduksjon med S' -> S) ved $, og skifter ved # og @
B. Det er LR(0)-konflikter både i tilstand 5 og 6. Disse kan ikke løses med SLR-betrakninger siden det er aktuelt både å skifte og redusere for # og @ (men om det kommer $ vet vi at det skal reduseres).

Kommentar: Vi *visste* at det her ville bli konflikter som ikke er løselige av noen type LR-betraktninger, siden grammatikken er flertydig.

1.f
For tilstandene under punkt B i spørsmål **1.e**, angi hvordan du ville løse konfliktene i disse «for hånd», om du skal få den presedensen og assosiativiteten som er angitt i **1.b**?

Svar 1.e

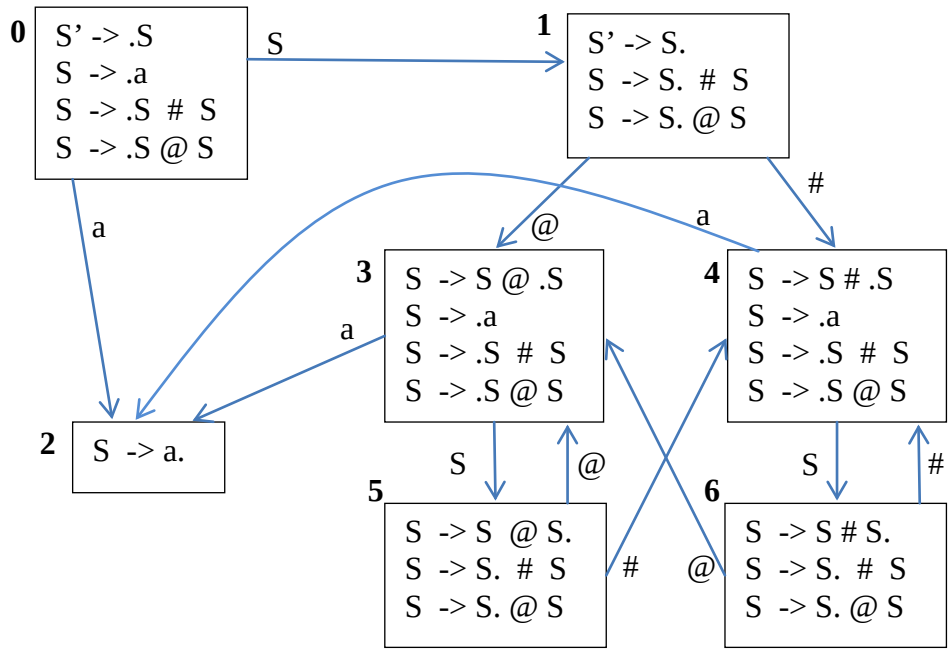
Tilsand 5: Her ligger nå S @ S på toppen av stakken. Derfor:
For #: Reduser, siden @ binder sterkere enn #
For @: Reduser, fordi @ er venstreassosiativ
For $: Reduser (ingen konflikt)

Tilsand 6: Her ligger nå S # S på toppen av stakken. Derfor:
For #: Skift, siden # er høyreassossiativ
For @: Skift, fordi @ binder sterkere enn #
For $: Reduser (ingen konflikt)

1.g
Sett opp en SLR(1)-parseringstabell for L(G1) ut fra svaret på spørsmålene **1.d** og **1.f**. Tabellen skal altså ha maks én aksjon i hver rute, og den resulterende syntaksanalysen skal altså følge reglene fra **1.b**.

Svar 1.g Her er LR(0)-DFA'en fra 1.d:

G1:
S' → S
S → a | S # S | S @ S



	a	#	@	$	S
0	s2				1
1		s4	s3	acc.	
2		r(S -> a)	r(S -> a)	r(S -> a)	
3	s2				5
4	s2				6
5		r(S->S @ S)	r(S->S @ S)	r(S->S@S)	
6		s4	s3	r(S->S # S)	

Oppgave 2 (25%)

Oppgave 3 (25%)

Oppgave 4 (25%)

En metode med to value-parametere er oversatt til følgende sekvens av TA-instruksjoner. Den eneste typen i språket er heltall.

```
x = <verdien av første aktuelle parameter>      (Du kan skrive dette slik: «x = par1»)
y = <verdien av andre aktuelle parameter>       (Tilsvarende)
z = x + y
label  L1
u = x + 1
x = u + y
if (y < x) goto L4          // Vi antar at det finnes en TA-instruksjon av denne formen
y = u + 1
u = y + x
label  L2
z = 5
if (x < u) goto L1
x = u + y
v = x + y
u = v + 1
goto L5
label  L4
v = z + 3
x = y + u
label  L5
z = v + 4
z = x + z
return z
```

4.a
Del programmet opp i basale blokker, og tegn opp flyt-grafen for programmet. Sett navnene B0, B1, osv. på blokkene. Svar, se lenger ned.

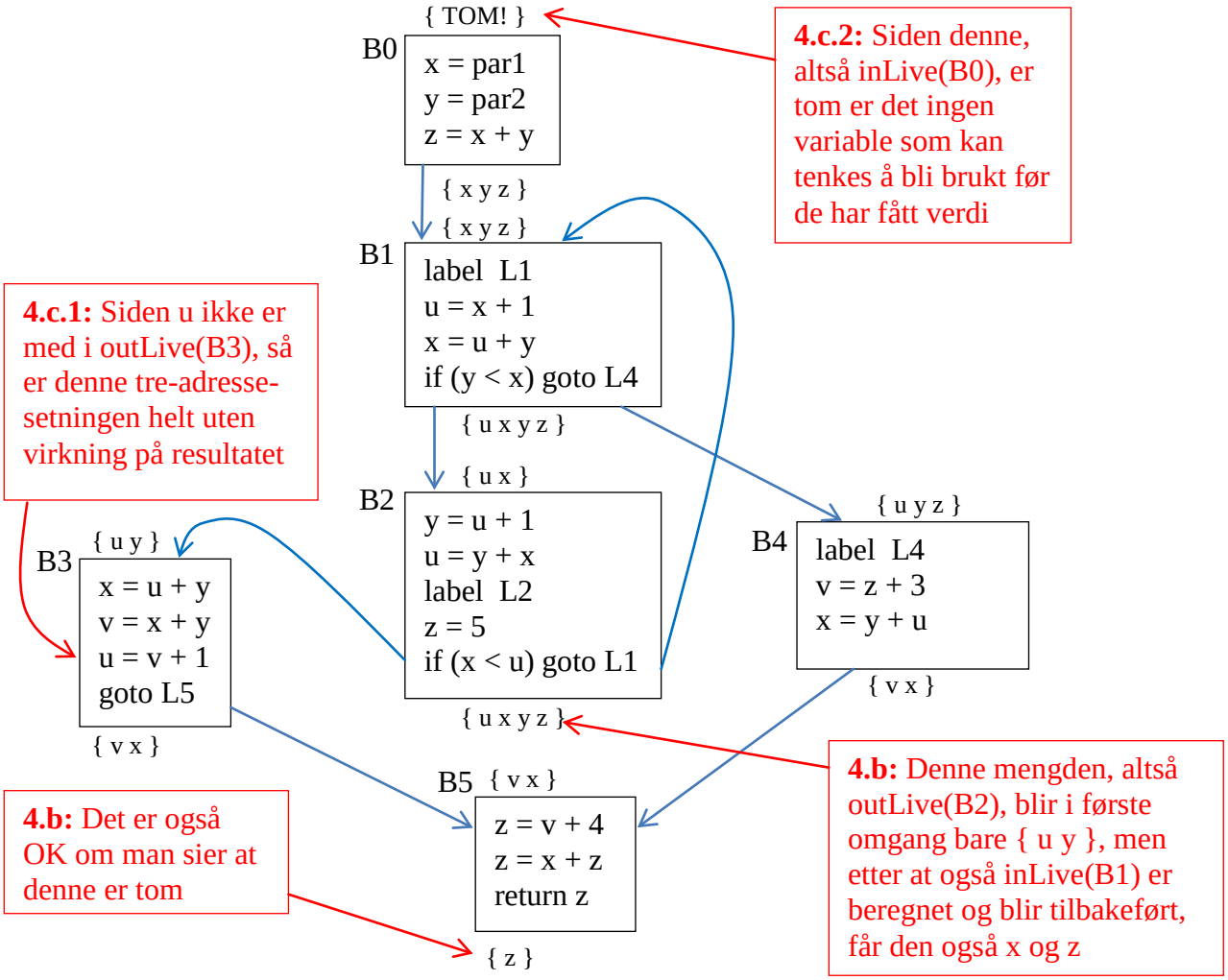
4.b
For hver basal blokk, finn hvilke variable som faktisk er i live både foran og etter blokka (altså *inLive* og *outLive* for blokka). Du kan bruke metoden fra pensum til å finne svaret, eller gjøre egne betraktninger. Det er greit å enten gi svaret direkte på flytgrafen fra **4.a**, eller du kan tegne opp flyt-grafen en gang til (gjerne uten kode i nodene) med alle mengdene inLive og outLive satt på der de hører hjemme. Svar, se lenger ned.

4.c
Ut fra informasjonen fra **4.b** og detaljene i TA-instruksjonene er det mulig å se

(1) om noen av TA-instruksjonene i programmet kan fjernes uten at det forandrer sluttresultatet når programmet utføres. Angi i så fall disse
(2) om det er variable som, i en eller annen eksekvering, kan bli brukt før de har fått verdi. Angi i så fall disse.

Om du ikke har fått til **4.b** kan du likevel forsøke å svare på denne oppgaven enten direkte fra programmet, eller fra flyt-grafen. Svar, se lenger ned.

Svar 4.a, 4.b og 4.c



4.d (Er uavhengig av det over)
I pensum er det diskutert hvordan man kan lage TA-kode for kortsluttede boolske uttrykk som står som betingelser i if- eller while-setninger. Dette gjøres rekursivt, og den rekursive kodegenererings-metoden har to label-parametere som koden skal hoppe til når man vet at det lokale uttrykket er h.h.v. **true** eller **false**. Programmet for dette er gjengitt under.

Vi oversatte i pensum til TA-kode og ikke til P-kode bl.a. for å slippe å tenke på at det under beregning av uttykket kan være noe på stakken ved hopp, som kanskje ikke stemmer med det stedet det hoppes til. Vi skal her se nærmere på hvor stort dette problemet blir, og hvordan vi eventuelt kan korrigere for det.

Som svar forklar først i detalj hvordan ting vil forholde seg med stakkdybder under uttrykksberegningen, og skissér så hvordan dette kan håndteres under kodegenereringen, gjerne ved å henvise til koden under. **Merk:** Vi antar at P-koden skal lages slik at stakken under kjøring er tom mellom setninger, og at den derved er tom når beregningen av det boolske uttrykket starter.

Hint: Det er sikkert lurt å se på hvordan P-koden blir for noen konkrete boolske uttrykk.

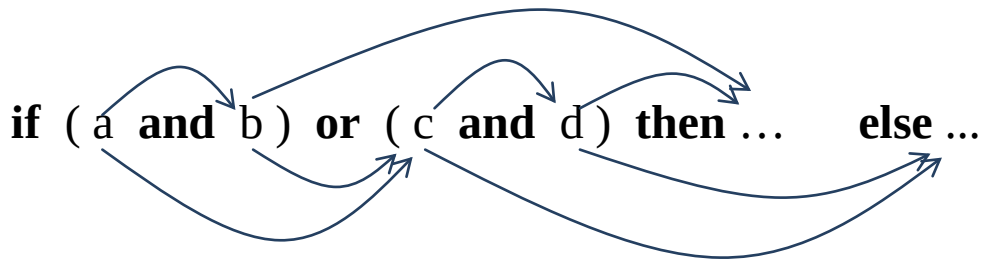
Program fra pensum. Det genererer TA-kode for logiske uttrykk i if- og while-setninger:

```
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);
            emit2(”label”, labx);
            right.genBoolCode(labT, labF);
        }
        case ”not”: {                                   // Har bare ”left”-subtre
            left.genBoolCode(labF, labT);
        }
        case ”<”: {
            String temp1, temp2, temp3;
            temp1 = left.genIntCode();  temp2 = right.genIntCode();
            temp3 = genLabel();
            emit4(temp3, temp1, «lt», temp2);   // Lager instruksjonen:   ”temp3 = temp1 < temp2”
            emit3(«jmp-false», temp3, labF);
            emit2(«ujp», labT);
        }
    }
```

Svar 4.d

Saken her er at så lenge man skal generere kode som *kortslutter* de boolske uttrykkene så blir det ingen problemer med stakkdybden i det hele tatt. Det kommer av at man, så fort det er en boolsk verdi på stakken, kommer til å teste om denne er **true/false**, og ut fra svaret enten hoppe eller fortsette rett til neste instruksjon. Teste-instruksjonen er slik at den i begge tilfelle vil ta bort det som er på toppen av stakken, og dermed vil det aldri bygge seg opp noe på stakken.

Vi kan se på følgende eksempel, der a, b c og d er boolske variable. Beregningen vil da følge pilene, ut fra at alle grener som ligger over teksten er **true**-grener, og de under er **false**-grener:



Vi antar altså at stakken er tom når vi starter på if-setningen. Ved hver variabel blir verdien av denne variabelen pushet på stakken, men forsvinner igjen i og med **true/false**-testen. Dermed vil sakken alltid være tom når kontrollen går langs en kant.

Dette betyr at en kodegenererings-prosedyre som skal lage P-kode kan lages etter nøyaktig samme mal som den angitt i oppgaven som lager TA-kode. Vi behøver ikke tenke på stakkdybden i det hele tatt.