UNIVERSITETET I OSLO

Med svarforslag

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF5110 - Kompilatorteknikk
Eksamensdag :	Onsdag 5. juni 2013
Tid for eksamen :	14.30 - 18.30
Oppgavesettet er på :	8 sider (pluss vedlegg)
Vedlegg :	1 side (side 9 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 (35%)

Vi ser først på følgende grammatikker:

G1: $S \rightarrow (\ S \ ) \mid \varepsilon$

G2: $S \rightarrow (\ S \ ) \mid a$

Her er S eneste ikke-terminal og dermed også startsymbol. Symbolene ’(’, ’)’ og ’a’ er terminalsymboler (sammen med ’$’, som har den vanlige funksjonen).

1a

Er ett av eller begge språkene L(G1) og L(G2) regulære? Angi et regulært uttrykk for det/de som eventuelt er regulære.

Svar 1a

Ingen av grammatikkene er regulære. Dette kommer av at setningene vil ha formen ”((…()…))” eller ”((…(a)…))”, med like mange parenteser på begge sider. Dette å beskrive at to sekvenser av symboler i en setning skal være like lange (når det ikke er noen begrensning på lengden av sekvensene) kan ikke gjøres med regulære uttrykk (eller, en endelig automat kan ikke sjekkes at det er slik).

1b

Vi skal så se på en grammatikk G3 som minner om de over, men som er litt mer komplisert, og vi vil undersøke hvilke egenskaper den har.

G3:

$A \rightarrow (\ S \ ) \mid (\ B \ ]$

B → S | (B
S → (S) | ε

Her er A, B og S ikke-terminaler og A er startsymbol. Symbolene '(', ')' og ']' er terminalsymboler (sammen med '$', som har den vanlige funksjonen).

Angi *fire* setninger i språket L(G3), slik at de best mulig dekker de forskjellige setningstypene som forekommer i L(G3). Gi også, med ord, en felles, generell beskrivelse av setningene i L(G3).

Svar 1b

Setningene kan f.eks. være:

()
((()))
((()]
(]

Den tomme strengen er altså ikke i L(G3), og heller ikke "((()))]".

En generell beskrivelse kan være: "Setningene starter med en eller flere venstre-parenteser, og avsluttes enten med like mange høyre-parenteser, eller med ekte færre (gjerne null) høyre-parenteser samt en høyre-hakeparentes".

1c

Finn **First-** og **Follow-**mengdene til A, B og S i G3, med bruk av ε som i boka. Du behøver bare å oppgi svaret.

Svar 1c

	First	Follow
A	(	$
B	(ε	]
S	(ε	)]

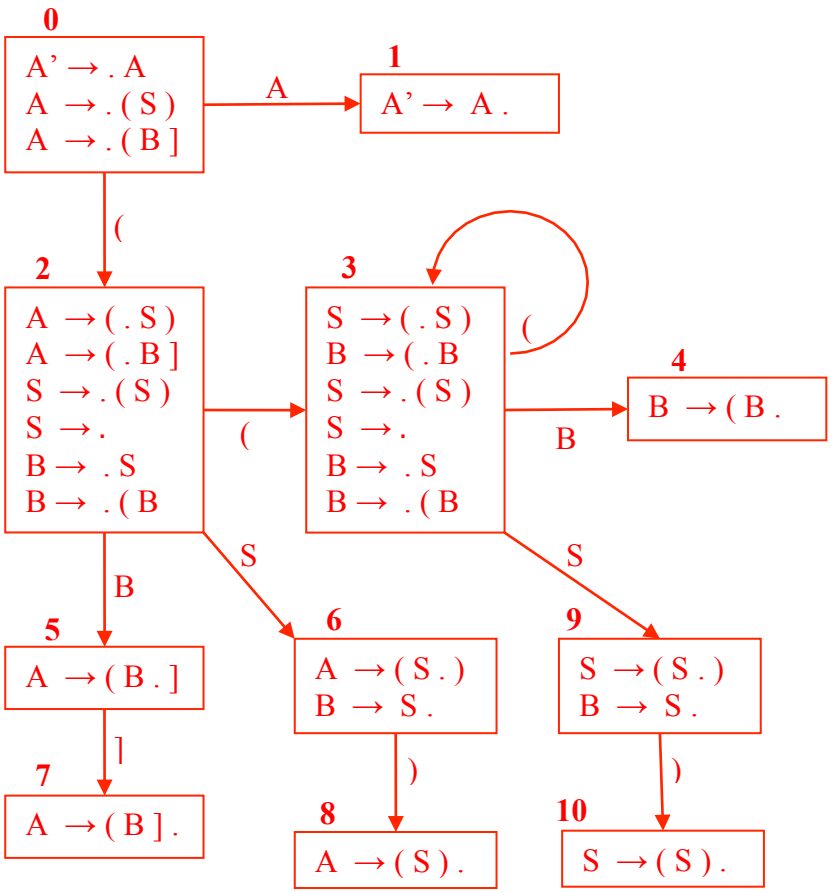
1d

Tegn opp LR(0)-DFAen til G3 (etter å ha innført et nytt startsymbol A' på vanlig måte). **Hint:** Det blir rundt 10 tilstander, og to av dem har 6 itemer. Vær nøye med å få med alle tillukninger og å slå sammen like tilstander.

Svar 1d

G3 på helt basal form, og med den ekstra ytterste produksjonen, er angitt ved siden av LR(0)-DFAen i svaret under

Det blir 11 tilstander, som angitt under. Vi nummererer også tilstandene for oppgave **1e**



Grammatikken:

A' → A
A → (S)
A → (B]
B → S
B → (B
S → (S)
S → ε

	First	Follow
A	(	$
B	(ε	]
S	(ε	)]

1e

Sett nummer på tilstandene fra 0 og oppover. Se gjennom alle tilstandene, og diskuter kort de som har (minst) én LR(0)-konflikt. Hvilke av disse har også en SLR(1)-konflikt? Er G3 en SLR-grammatikk?

Svar 1e

Her er det fire tilsander som har LR(0)-konflikter, nemlig 2, 3, 6 og 9. Det er itemet "S → ." som lager problemer alle fire steder. Dette sier at å redusere med "S → ε" er en mulighet, mens alle de andre itemene i alle disse tilstandene angir skift. Om man ser på etterfølger-mengder (og dermed med SLR(1)-filosofi) så lar de seg imidlertid løse. Etterfølgermengden til S er altså ')' og ']' og den til B er bare ']'

Tilst. 2 og 3: Her kan det bare skiftes for '(', mens reduksjon med "S → ε" bare er aktuelt for ')' og ']'. Altså SLR(1).

Tilst. 6 og 9: Her kan det bare skiftes for ')', mens det bare kan reduseres for ']'. Altså SLR(1).

1f

Du skal tegne opp deler av parserings-tabellen for G3 ut fra SLR(1)-filosofi, nemlig de to radene (linjene) som tilsvarer de to tilstander med 6 itemer. Dersom G3 ikke er SLR(1) skal du angi alle aktuelle alternativer i de rutene der det er SLR(1)-konflikt. Pass på å få med alle de riktige

symbolene langs øvre kant av tabellen.

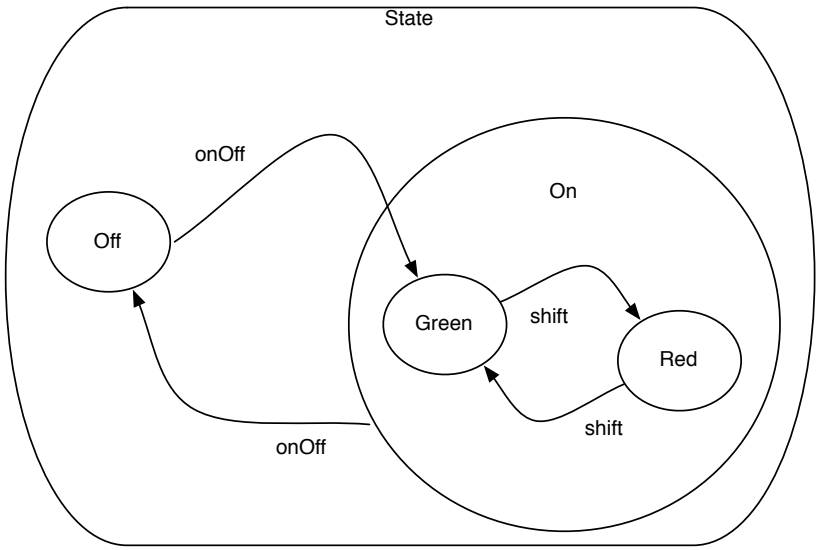
Svar 1f

Ut fra konklusjonen fra forrige oppgave, og LR(0)-DFA-en, blir tabellen for tilstandene 2 og 3 slik

	(	)	]	$	A	B	S
2	s3	r(S → ε)	r(S → ε)			5	6
3	s3	r(S → ε)	r(S → ε)			4	9

Oppgave 2 (20%)

Anta at vi har et objekt-orientert språk, hvor alle metoder i klasser er virtuelle slik at de kan redefineres i subklasser. En standard måte å implementerer tilstandsmaskiner i et slikt språk er å representere tilstandene ved objekter av klasser i et tilstandshierarki, med transisjoner som virtuelle metoder som redefineres i de forskjellige state subklasser. De følgende klasser implementerer en tilstandsmaskin for klassen `Switch` (på norsk: Bryter) med tilstandene `Off` og `On`. Tilstanden `On` inneholder nye tilstander (`Green` og `Red`) og transisjoner mellom disse gjøres ved kall på metoden `shift()`. En slik 'composite state' representeres ved en abstrakt klasse, som brukes som superklasse til alle tilstander som den skal inneholde: dermed gjelder de transisjoner som er definert for `On` også for tilstandene `Green` og `Red`.



```
class Switch{
    State state = new Off();
    int noOfShifts = 0;
    void onOff(){state.onOff()};
    void shift(){state.shift()};
    void print() {out.println(noOfShifts); noOfShifts = 0};
    void setState(State s) {state = s};
  }

  abstract class State{
    Switch s;
    void onOff (){};
    void shift(){};
  }
  class Off extends State{
    void onOff(){s.setState(new Green())}
  }
  abstract class On extends State{
    void onOff(){s.setState(new Off())}
  }
  class Green extends On{
    void shift(){
      s.noOfShifts=s.noOfShifts+1; s.setState(new Red())}
  }
  class Red extends On{
    void shift(){
      s.noOfShifts=s.noOfShifts+1; s.setState(new Green())}
  }
```

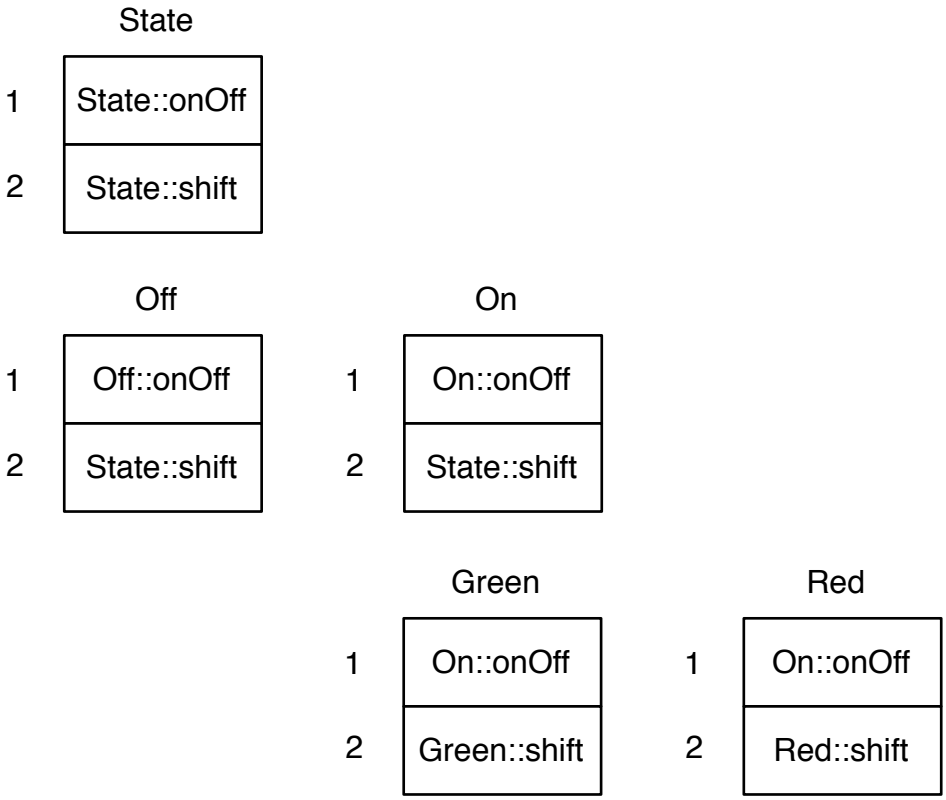
Alle tilstandsobjekter har en peker (`s`) til det `Switch`-objekt, som har tilstandsmaskinen. Vi har ikke tatt med hvordan denne settes, da det ikke er viktig for oppgaven.

Et program som bruker disse klasser vil lage et objekt av klassen `Switch` og kalle metodene `onOff` og `shift` på dette. Alt etter hvilken tilstand `Switch`–objektet er i (representert ved variablen `state`) vil de aktuelle redefinisjoner av transisjons-metodene bli utført.

2a

Lag virtuell-tabellene for alle klassene som trenger dette. For hvert element i tabellene skal du bruke notasjonen `C::m` for å angi hvilken metode som gjelder.

Svar 2a

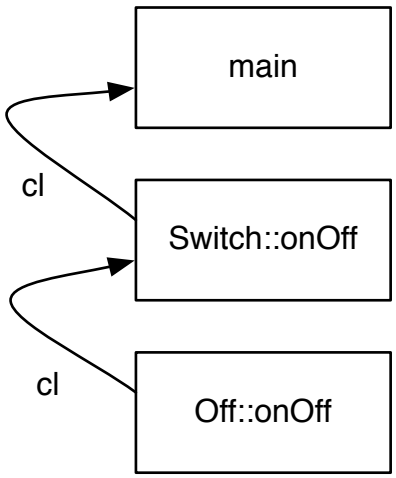


2b

Hvordan ser stakken ut rett før skift av tilstand (endring av variablen `state` i `Switch`-objektet) gitt at en metode `main` kaller metoden `onOff` på et `Switch`-objekt `aSwitch`:

```
aSwitch = new Switch();
aSwitch.onOff ()
```

Svar 2b



2c

I denne siste del av oppgaven har språket blitt utvidet slik at funksjoner nå kan være parametere til metoder, og at metoder kan ha lokale funksjoner. Funksjoner defineres som vanlige metoder.

Med denne utvidelse kan man få tilstandsmaskinen til ikke bare å skifte tilstand ved å kalle en transisjonsmetode, men samtidig få utført ting som brukeren av tilstandsmaskinen ønsker å få gjort. Dette gjøres ved å tilføye en funksjonsparameter til transisjons-metodene `onOff` og `shift` i `Switch`, overføre den til kallene på de tilsvarende metoder i `State` objektene, og kalle den som en del av disse:

```
class Switch{
    State state = new Off();
    int noOfShifts = 0;
    void onOff(void action()){state.onOff(action)};
    void shift(void action()){state.shift(action)};
    void print() {out.println(noOfShifts); noOfShifts = 0};
    void setState(State s){state = s};
  }
  abstract class State{
    Switch s;
    void onOff (void action()){};
    void shift(void action()){};
  }
  class Off extends State{
    void onOff(void action()){
    action(); s.setState(new Green())}
  }
  abstract class On extends State{
    void onOff(void action()){
      action(); s.setState(new Off())}
  }
  class Green extends On{
    void shift(void action()){
      s.noOfShifts=s.noOfShifts+1; action(); s.setState(new Red())}
  }
  class Red extends On{
    void shift(void action()){
      s.noOfShifts=s.noOfShifts+1; action(); s.setState(new Green())}
  }
```

Anta at vi har følgende `main`-metode, med en lokalt definert funksjon som overføres som parameter til `onOff` og `shift`:

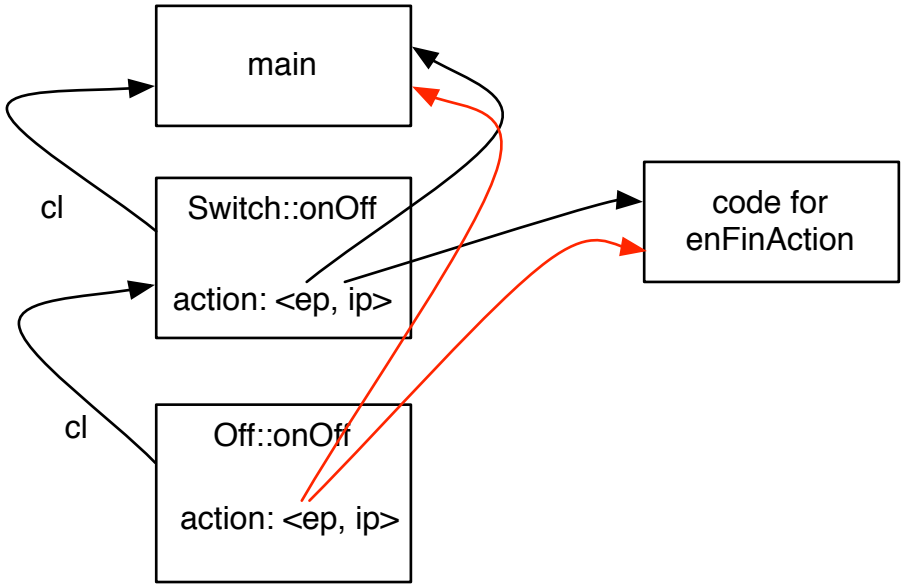
```
  void main{
    void enFinAction(){out.println('så skifter vi tilstand')};
    Switch aSwitch = new Switch();
    aSwitch.onOff(enFinAction);
  }
```

Spørsmålene til **2c** er:

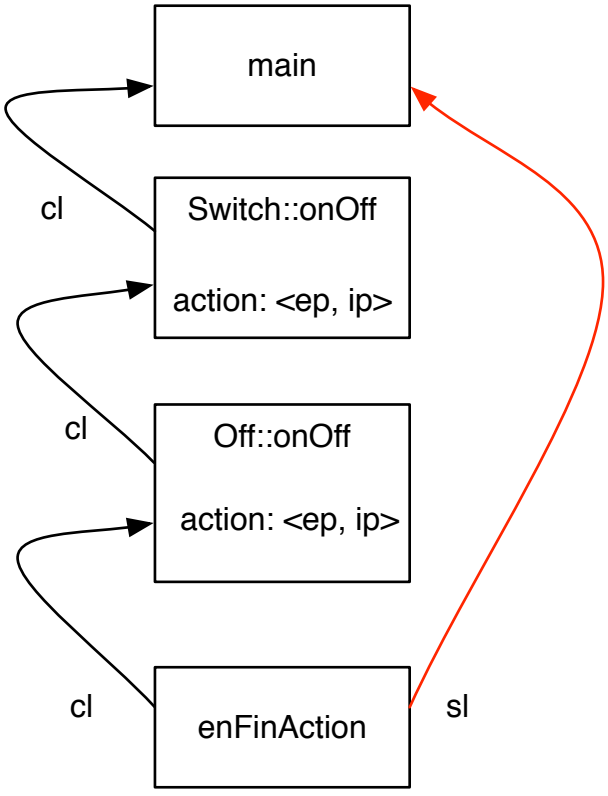
1) Skisser og forklar hvordan `action`-parameteren (til for eksempel metoden `onOff` i klassen `Switch`) implementeres, hvordan den overføres i kallet `state.onOff(action)`, og hvordan den kalles i utførelsen av `action()` i metoden `onOff` i klassen `Off`.
2) Tegn stakken (med control og static links) som den ser ut når `action()` i metoden `onOff` i klassen `Off` utføres.

Svar 2c

1)


main
cl
Switch::onOff
action: <ep, ip>
code for
enFinAction
cl
Off::onOff
action: <ep, ip>

2)


main
cl
Switch::onOff
action: <ep, ip>
cl
Off::onOff
action: <ep, ip>
cl
enFinAction
sl

Oppgave 3 (20%)

I stedet for å lage tilstandsmaskiner ved hjelp av klasser og subklasser, så vil vi i denne oppgaven lage et språk spesielt for å programmere tilstandsmaskiner. Vi har ikke tatt med alle detaljer i språket, bare de som er aktuelle for det vi skal se på. Definisjoner av events og bruk av dem i transisjoner er med i grammatikken, men de har ingen betydning for oppgaven og derfor ikke med i de semantiske regler.

Det følgende er et fragment av grammatikken for språket:

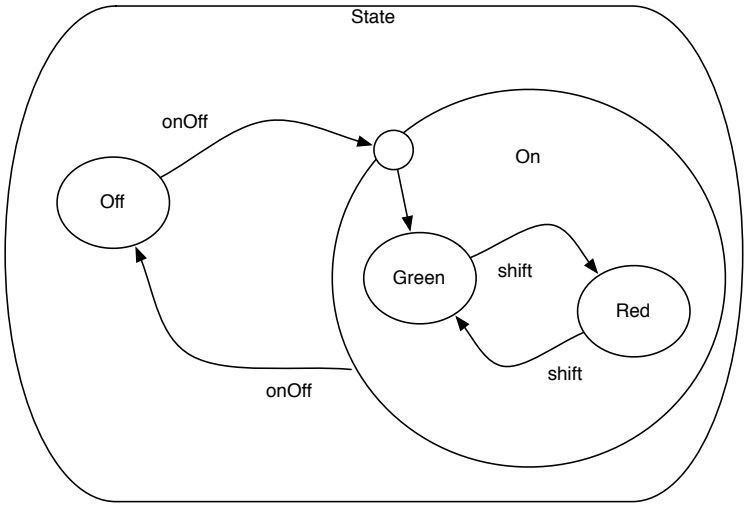
```
statemachine → state
state → state name ({events states transitions entryPoints})?

events → events, event
events → event
event → name

states → states, state
states → state

transitions → transitions, transition
transitions → transition
transition → on eventId from endPoint to endPoint

entryPoints →
entryPoints → entryPoints, entryPoint
entryPoints → entryPoint
entryPoint → entry name on name

endPoint → name
eventId → 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'.

En state er enten en 'simple state' (med bare et name) eller en 'composite state'. En transisjon defineres ved hvilket event som utløser transisjonen, samt 'source' og 'target' endPoint, som hver er et navn som identifiserer en simple state, en composite state eller et entryPoint. Et entryPoint er et punkt på en composite state, og det defineres ved to navn: (entryPoint navn, composite state navn).

Et krav i dette språket er at transisjoner ikke kan krysse grensen for en composite state som vi så i oppgave 2, hvor `onOff` fra tilstanden `Off` gikk direkte til tilstanden `Green` i tilstanden `On`. Reglene er som følger:

1. transisjoner kan bare være mellom states på samme nivå eller mellom states og entryPoints på states på samme nivå
2. 'target' for en transisjon kan ikke være en composite state, bare en 'simple state' eller et entryPoint på en composite state.



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

Du kan bruke følgende funksjoner:

`insert(name, kind, level):`
Setter inn i symboltabellen sammenhengen mellom *`name`* (på en state eller entryPoint) og (*`kind, level`*)

`lookupKind(name):` Gir kind (dvs simple, composite eller entryPoint) for *`name`*.

`lookupLevel(name):` Gir nivå for state eller entryPoint med navnet *`name`*.

Besvar dette spørsmålet ved å bruke vedlegget side 9.

Svar 3

Grammar Rule	Semantic Rule
statemachine → state	*state.level=0*
state → **state *name***	*insert(**name**.name, simple, state.level)*
state → **state *name*** **{***events states transitions entryPoints***}**	*insert(**name**.name, composite, state.level)* *states.level=state.level+1* *transitions.level=state.level+1* *entryPoints.level=state.level+1*
$states_1$ → $states_2$, *state*	$states_2.level=states_1.level$ $state.level=states_1.level$
states → state	*state.level=states.level*
$transitions_1$ → $transitions_2$, *transition*	$transitions_2.level= transitions_1.level$ $transition.level= transitions_1.level$
transitions → *transition*	*transition.level= transitions.level*
transition → **on** *eventId* **from** $endPoint_1$ **to** $endPoint_2$	**if** $lookupKind(endPoint_2.name)=$ *composite* **then** *notOK* **else** **if** $lookupKind(endPoint_1.name)=$ *simple* **and** $lookupKind(endPoint_2.name)=$ *simple* **then if** $lookupLevel(endPoint_1.name)=$ $lookupLevel(endPoint_2.name)$ **then** *OK* **else** **if** *(*$lookupKind(endPoint_1.name)=$ *simple* **or** *composite)* **and** $lookupKind(endPoint_2.name)=$ *entryPoint* **then if** $lookupLevel(endPoint_1.name)=$ $lookupLevel(endPoint_2.name)-1$ **then** *OK* **else** **if** $lookupKind(endPoint_1.name)=$ *entryPoint* **and** $lookupKind(endPoint_2.name)=$ *simple* **then** **if** $lookupLevel(endPoint_2.name)=$ *transition.level* **then** *OK* **else** *notOK*
$entryPoints_1$ → $entryPoints_2$, *entryPoint*	$entryPoints_2.level= entryPoints_1.level$ $entryPoint.level= entryPoints_1.level$
entryPoints → *entryPoint*	*entryPoint.level= entryPoints.level*
entryPoint → **entry** $name_1$ **on** $name_2$	*insert(***name**$_1$.name, entryPoint, entryPoint.level)*
endPoint → ***name***	*endPoint.name =* ***name****.name*

Oppgave 4 (25%)

4a

Vi deler opp en metode i et program i basale blokker, og tegner opp flytgrafen for metoden. Til slutt finner vi (f.eks. med den gjennomgåtte iterasjons-metoden) hvilke variable som er i live ("live") foran og bak hver basal blokk (altså *inLive* og *outLive* for hver blokk). Svar så på spørsmålene:

(1) Hvordan kan du nå finne TA-instruksjoner (om noen) som helt sikkert ikke vil ha noen betydning for utførelsen av programmet?

(2) Hvordan kan vi se om det er variable (og eventuelt hvilke) som kan bli brukt (avlest) før de har blitt gitt en verdi i programmet?

Svar 4a

(1) Se på en TA-instruksjon i en blokk B som er tilordning til en variabel v. Denne kan fjernes dersom

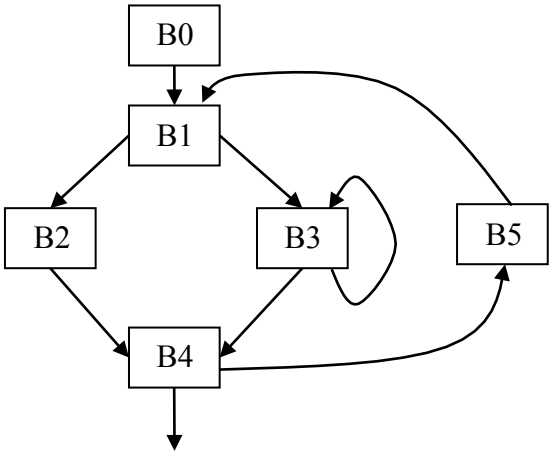
  (a) det ikke er noe bruk av v senere i blokka B, og

  (b) v ikke forekommer i outLive(B)

(2) Om det er variable i inLive(B0), der B0 er startblokka, så vil disse variablene (for en alle annen utførelse av programmet) kunne bli brukt før de får verdi.

4b

Vi ser på følgende flytgraf:



Knut mener at det i denne flytgrafen er følgende løkker (slik løkker defineres i forbindelse med kodegenerering):

B1, B2, B4, B5
B1, B3, B4, B5
B1, B2, B3, B4, B5

Astrid er ikke enig. Hvem har rett? Forklar! Om Astrid har rett, angi de riktige løkkene i flytgrafen over.

Svar 4b

Det er Astrid som har rett. Hverken {B1, B2, B4, B5} eller {B1, B3, B4, B5} er løkker i bokas forstand, siden man kan komme utenfra og inn i dem til to forskjellige blokker, nemlig slik:

Man kan komme utenfra og inn i {B1, B2, B4, B5} til B1 (fra B0) og til B5 (fra B3)
Man kan komme utenfra og inn i {B1, B3, B4, B5} til B1 (fra B0) og til B5 (fra B2).

Dessuten har Knut ikke fått med seg at blokken B3 alene utgjør en løkke. De riktige løkkene er altså:

B1, B2, B3, B4, B5
B3

4c

I blokk B2 i flytgrafen i 4.b står TA-instruksjonene

…
k = j + x
k = k* k
…

Vi lurer på om denne beregningen kan flyttes ut av den minste løkka L som B2 er med i (for å spare eksekveringstid).

(1) Hva må du da sjekke i de forskjellige basale blokkene før du eventuelt kan gjøre en slik flytting, og i nøyaktig hvilke blokker må slik sjekk gjøres?

Konkret vil en slik flytting innebære at vi legger til følgende et sted utenfor løkka L:

k' = j + x k' er en ny variabel
k' = k'* k'

Dessuten erstatter vi den første sekvensen (i B2) med k = k'.

(2) Hvor, utenfor løkka L, er det rimelig å legge den TA-sekvensen over som gir verdi til k'?

Svar 4c

(1) Man må sjekke at k, j og x ikke får en ny verdi noen steder i den minste omsluttende løkka L, og det er altså i: {B1, B2, B3, B4, B5}.

(2) Man må legge disse TA-instruksjonene i en ny "kunstig" basal blokk som har etterfølger B1, og som har som forgjengere alle blokker utenfor løkka L som har (eller hadde) B1 som etterfølger. Dette gir her en ny blokk mellom B0 og B1. Merk at kanten fra B5 til B1 ikke skal "gå innom" denne blokka, siden B5 er med i løkka.

4d

Vi vil bruke den kodegenererings-metoden (inklusive metoden *getreg*) som står i notatet tatt fra kap. 9. i ASU-boken (og i foilene) til å generere maskinkode (av samme type som i notatet) for følgende basale blokk av TA-instruksjoner:

e = a – b
f = a – c
d = f – e

Her er alle variable vanlige programvariable, og de antas alle å være i live etter blokka.
Til forskjell fra i notatet skal vi nå anta at maskinen bare har *ett* register R0. Du kan anta at analysen som gir informasjon om *next-use* etc. for den aktuelle blokka er utført før kodegenereringen starter.

Hva blir den genererte sekvensen av maskin-instruksjoner? Vis hvilke maskin-instruksjoner som stammer fra hvilken TA-instruksjon. Du behøver ikke angi de formelle deskriptorene, men du skal skrive noen kommentarer med tilsvarende innhold til høyre for programmet.

Svar 4c

Dette blir litt fiklete kode, fordi det bare er ett register. Vi får:

```
e = a – b       MOV   a   R0
                SUB   b   R0   // Som i notatet, med e nå i R0. Nå er alle registre opptatt

f = a – c       MOV   R0  e    // f har en neste-bruk. Frigjør det eneste registeret som finnes
                MOV   a   R0
                SUB   c   R0   // f er nå i R0

d = f – e       MOV  R0   f    //  f er i live etter blokken og må derfor saves.
                SUB   e   R0   //  f er allerede i riktig posisjon for operasjonen, altså i R0.
                               //  Etterpå er d i R0

Avslutning      MOV   R0  d    //Alle de andre er i sine ”hjemmeposisjoner”
```