Norges Teknisk-Naturvitenskapelige Universitet
Fakultet for Informasjonsteknologi, matematikk og elektroteknikk
Institutt for Datateknikk og Informasjonsvitenskap

LØSNINGSFORSLAG TIL
EKSAMEN I TDT 4165
PROGRAMMERINGSSPRÅK

Lørdag 28. Mai 2005, 9.00–13.00

BOKMÅL

Faglig kontakt under eksamen:
Per Holager (foreleser), Tlf 996 17 836
Per Kristian Lehre, Tlf 913 60 757

Hjelpemiddelkode: C
Tillatte trykte og håndskrevne hjelpemidler: Ingen
Bestemt, enkel kalkulator er tillatt.

Oppgaven er kvalitetssikret av Martin Thorsen Ranang.

Les hele oppgavesettet før du begynner på besvarelsen. Svar kort og konsist. Dersom svaret er uklart eller lengre enn nødvendig, trekker dette ned.

Flervalg

Fyll inn svarene på denne oppgaven på svararket på siste side.

Kun ett av svaralternativene på hver deloppgave er riktig.

Oppgave 1: (15 %)

a) Hvilket programmeringsspråk er følgende programfragment skrevet i:

```
size::BinTree a -> Integer
size Empty = 0
size(Node va lt rt)=1+(size lt)+(size rt)
```

1. Prolog
2. Haskell
3. Java
4. C++

Løsning: 2. Haskell

b) Hva menes med begrepet *memoization*?

1. At grensesnittet til komponenten skal være uavhengig av beregningsmodellen som benyttes for å implementere den.
2. Å ta vare på resultatene fra funksjonskall slik at fremtidige kall kan håndteres raskere.
3. At objekter kan dele data uten noen bestemte forbehold når alle objektene kjører i samme tråd.
4. Å bruke en ivrig (eng. *eager*) versjon av optimalisering fremfor en lat (eng. *lazy*) versjon.

Løsning: 2 - Å ta vare på resultatene fra funksjonskall.

c) Gitt følgende to Oz-funksjoner for beregning av fakultetsfunksjonen $n!$ (husk at $0! = 1$):

```
declare
fun {Fac1 N}
   if N>1 then {Fac1 N-1}*N
   else 1
   end
end
```

```
declare
fun {Fac2 N}
   if N<1 then 1
   else N*{Fac1 N-1}
   end
end
```

Hvilke utsagn er sant:

1. `Fac1` beregner ikke funksjonen korrekt.
2. `Fac2` beregner ikke funksjonen korrekt.
3. `Fac1` og `Fac2` er omtrent like raske.
4. `Fac1` er ikke høyre-rekursiv (halerekursiv). Den er derfor tregere enn `Fac2`.

Løsning: 3 - `Fac1` og `Fac2` er omtrentelig like raske.

d) Hva kan sies om typesystemet i Oz om man bruker terminologien i læreboka?

1. Svak (eng. *weak*) og dynamisk.
2. Svak (eng. *weak*) og statisk.
3. Sterk (eng. *strong*) og dynamisk
4. Sterk (eng. *strong*) og statisk.

Løsning: 3 - Sterk og dynamisk

e) Hva er korrekt strukturell, operasjonell, semantikk (eng. *structural operational semantics*) fra kapittel 13 for `if`-utsagn i tilfellet der betingelsen etter `if` holder:

1. $\dfrac{\texttt{if } X \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end} \;\|\; S_1\{X \to x\}}{\sigma \wedge x = \texttt{true} \;\|\; \sigma \wedge x = \texttt{true}}$

2. $\dfrac{\texttt{if } x \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end} \;\|\; S_1}{\sigma \;\|\; \sigma \wedge x = \texttt{true}}$

3. $\dfrac{\texttt{if } x \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end} \;\|\; S_1}{\sigma \;\|\; \sigma}$ dersom $\sigma \models x = \texttt{true}$.

4. $\dfrac{\texttt{if } X \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end} \;\|\; S_1\{X \to x\}}{\sigma \wedge X = \texttt{true} \;\|\; \sigma \wedge X = \texttt{true}}$

Løsning: 3

f) Strukturell, operasjonell semantikk for mislykte verdier (eng. *failed values*) består ikke av følgende reduksjon: $\dfrac{\{\texttt{FailedValue } x\ x_f\} \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma \wedge x_f = failed(x)}$, fordi

1. det ville ikke gitt en korrekt feilhåndtering når x_f allerede er bundet.
2. utførselen ville stoppet og lageret ville ikke endret seg.
3. utførselen ville ikke stoppet umiddelbart.
4. resultatet av funksjonen *failed* kan ikke bli tilordnet til x_f fordi det ikke er en boolsk verdi.

Løsning: 1 - det ville ikke gitt en korrekt feilhåndtering når...

g) Hva er den effektive datatypen (i "generell" typenotasjon fra forelesningene) av parametrene i metoden i følgende Java kode-fragment:

```
public class Cc { public double pr(double p1, Cc p2) ... }
```

1. `p1: ref double, and p2: ref Cc`
2. `p1: ref double, and p2: Cc`
3. `p1: double, and p2: ref Cc`
4. `p1: double, and p2: Cc`

Løsning: 3 - `p1: double, and p2: ref Cc`

h) Hva er den effektive datatypen (i "generell" typenotasjon fra forelesningene) av variabelen `x` deklarert av følgende C++ kode-fragment:

```
int *x(char*)
```

1. `ref func(ref char)int`
2. `ref [0..255] int`
3. `[ref char]ref int`
4. `func(ref char)ref int`

Løsning: 4 - `func(ref char)ref int`

i) Hvilke av følgende utsagn om trådet tilstand (dvs. akkumulator-parameter) er minst korrekt?

1. reduserer ofte behovet for antall prosedyreargumenter.
2. er et alternativ til eksplisitt tilstand.
3. er en deklarativ programmeringsteknikk.
4. er nyttig for å konstruere effektiv programmering.

Løsning: 1 - reduserer ofte behovet for antall prosedyreargumenter.

j) Hvilket programmeringsspråk er følgende kodefragment skrevet i:

```
nyrootsyn(BM,NN,Gender):-nrootsyn(BM,NN),nyroot(NN,Gender),!.
```

1. Prolog
2. Haskell
3. Java
4. C++

Løsning: 1 - Prolog

Programmering

Oppgave 2: (8 %)

Gitt følgende Javaklasse for en enkelt lenket liste av strenger,

```
public class LiStr { String str; LiStr next; ... }
```

Skriv en Javametode (`public void printList()`) for denne klassen som traverserer listen og skriver ut strengene (ved hjelp av `System.out.println(str)`). Begrens deg til enkel-tilordningsparadigmet (eng. *single assignment paradigm*), og unngå spesielt `for`- og `while`-løkker.

(Dersom du ikke kjenner Java, bruk et annet velkjent imperativt språk bortsett fra Oz. Husk å angi hvilket språk du velger.)

Løsning: Et fullstendig program, inkludert forespurte metode:

```
public class LiStr {
       String str;
       LiStr next;

       LiStr(String stR, LiStr nexT) {
              str = stR;
              next= nexT;
       }
```

```
        public void printList() {
                System.out.println(str);
                if (next!=null)
                        next.printList();
        }

        public static void main(String[] a) {
                LiStr ls = new LiStr("Pål", new LiStr("sine",
                                        new LiStr("høner", null)));
                ls.printList();
        }
}
```

Relasjonsmodellen

Oppgave 3: (25 %)

Skriv en funksjon i relasjonsmodellen som genererer alle permutasjoner av en gitt innputliste av vilkårlig størrelse. Vis hvordan funksjonen du definerer skal kalles med `SolveAll`-funksjonen definert i læreboka.

Merk: Du kan benytte den innebygde funksjonen `{Append List1 List2}`.

Løsning:

```
declare

fun{Perm N A L}
        choice   % with current 1., all perm.s of rest:
                if A>1
                then L.1|{Perm A-1 A-1 L.2}
                else [L.1]
                end
        []       % rotate list and repeat:
                if N>1
                then {Perm N-1 A {Append L.2 [L.1]}}
                else fail
                end
        end
end
```

En annen, muligens mer intuitiv løsning:

```
declare
```

```
fun {Permutations List}
   case List
   of nil then nil
   [] _|_ then Rest in {ChooseOne List Rest}|{Permutations Rest}
   end
end

fun {ChooseOne List Rest}
   case List
   of [H] then Rest=nil H
   [] H|T then choice Rest=T H
       [] Rest2 in Rest=H|Rest2 {ChooseOne T Rest2}
       end
   end
end

List = [a b c]
Sols = {SolveAll fun {$} {Permutations List} end}
```

Parser

Oppgave 4: (26 %)

Gitt følgende forenklede fragment av Javasyntaksen:

```
<term>       ::= <field term> | <call term> .
<field term> ::= 'id' | <term> '.' 'id' .
<call term>  ::=  <field term> '(' <param> ')' .
<param>      ::=  ϵ | <term> .
```

Terminalsymbolet `'id'` representerer en identifikator. Leksikal-analysatoren vil fremskaffe de faktiske navnene som atomer (på den vanlige måten). Symbolet ϵ representerer den tomme strengen.

Følgende er syntaks for et semantisk tre (læreboken: *abstract syntax tree*) for en parser i Oz for syntaksen ovenfor:

```
<term>       ::= <field term> | <call term> | nil .
<field term> ::= field(id: <id's text>  nxt: <term> ) .
<call term>  ::= call(id: <id's text>  prm:<term>  nxt: <term> )
```

Det vil si at hvis innputt til en leksikalanalysator er

```
Nils.venn(Jens).farge
```

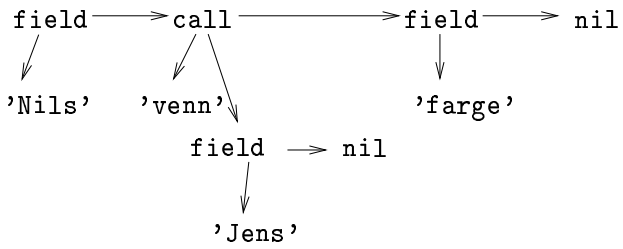
så blir token-lista som går inn i parseren

```
[ 'id'('Nils') '.' 'id'('venn') '(' 'id'('Jens') ')'
 '.' 'id'('farge') ]
```

Om man ser bort fra rekkefølgen på feltene, skal dette gi returverdi

```
field(id:'Nils'
     nxt:call(id:'venn'
              prm:field(id:'Jens' nxt:nil)
              nxt:field(id:'farge'  nxt:nil)
  )      )
```

Alternativt,



Kanter til neste er her tegnet horisontalt mot høyre.

a) Er dette syntaks-fragmentet et passende utgangspunkt for å skrive en rekursiv nedstigningsparser? Begrunn svaret. Dersom den ikke er passende, konstruere en passende, ekvivalent grammatikk. Forsøk å gjøre den så enkel som mulig.

Løsning: Den er ikke passende på grunn av venstrerekursjon for `<term>` via `<field term>` og muligens også `<call term>`.

Sett inn for `<field term>` og `<indexed term>` i definisjonen av `<term>`:

```
<term>      ::= 'id'
                | 'id' '(' <param> ')'
                | <term> '.' 'id'
                | <term> '.' 'id' '(' <param> ')' .
<param>     ::=  | <term> .
```

Faktoriser, bruk paranteser og hakeparanteser, transformer venstrerekursjon til høyrerekursjon:

```
<term>      ::=  ('id' ['(' <param> ')']) <term rest> .
<term rest> ::=  | ('.' 'id' ['(' <param> ')']) <term rest> .
<param>     ::=  | <term> .
```

Merk at det som kan følge `'.'` i `<term rest>` er det samme som definisjonen av `<term>`; erstatt dette og forenkler:

```
<term>      ::=  'id' ['(' <param> ')']['.' <term>]
<param>     ::=  | <term> .
```

b) Skriv en rekursiv nedstigningsparser i Oz for dette språket. Parseren skal returnere `true` når parsingen lykkes og `false` når parsingen mislykkes.

Løsning:

```
declare Param
proc {Term In Out} R3 in
       case In
       of 'id'(N)|'('|R1 then R2 in
              {Param R1 R2}
              R2 = ')'|R3
       [] 'id'(N)|R1 then
              R1=R3
       end
       case R3
       of '.'|R4 then {Term R4 Out}
       else Out=R3
       end
end

proc {Param In Out}
       case In
       of 'id'(_)|_ then
              {Term In Out}
       else Out=In
       end
end

declare X=
        [ 'id'('Nils') '.' 'id'('venn') '(' 'id'('Jens') ')'
         '.' 'id'('farge') ]
```

c) Utvid svaret ditt på **b)** ovenfor til å generere semantiske trær på den gitte formen.

Løsning:

```
declare Param
fun{Term In Out} R3 Tr Nx in
       case In
```

```
       of 'id'(N)|'('|R1 then R2 in
              Tr=call(id:N  prm:{Param R1 R2} nxt:Nx)
              R2 = ')'|R3
       [] 'id'(N)|R1 then
              Tr=field(id:N  nxt:Nx)
              R1=R3
       end
       case R3
       of '.'|R4 then Nx={Term R4 Out}
       else Out=R3  Nx=nil
       end
       Tr
end

fun{Param In Out}
       {Show Param\#In}
       case In
       of 'id'(_)|_ then
              {Term In Out}
       else Out=In  nil
       end
end

declare X = [ 'id'('Nils') '.' 'id'('Venn') '(' 'id'('Jens') ')'
               '.' 'id'('farge') ]

declare Y
{Browse {Term X Y}}
{Browse Y}
```

Tolker

Oppgave 5: (26 %)

Introduksjon

I denne oppgaven skal vi se på en tolker for et veldig enkelt imperativt språk. Den konkrete syntaksen (eng. *concrete syntax*), dvs. syntaksen for det som går inn i parseren er ikke viktig for denne oppgaven. Vi antar at det finnes en parser som genererer semantiske trær (i læreboken: abstract syntax trees) på formen definert av syntaksen i Figur 1. Som du ser så er de semantiske trærne strukturerte som nøstede recorder i Oz. Appendikset inneholder en tolker for disse trærne. Din oppgave er å utvide og vurdere hvordan man kan utvide det semantiske treet og tolkeren.

En uformell beskrivelse av det semantiske treet og dets semantikk

Et program (`<Program>` i Fig. 1) representeres som en liste av setninger (`<Statement>` i Fig. 1). Setningene utføres i den rekkefølgen de forekommer i listen. Det er to typer setninger: `assign(I E)` lager en binding i lageret fra variabelen med identifikator `I` (`<Identifier>` i Fig. 1) og verdien av uttrykket `E` (`<Expression>` i Fig. 1), og `print(E)` skriver ut verdien av uttrykket `E` til skjerm. En variabel oppstår i lageret første gang den bindes til en verdi. Forholdet mellom identifikatorer og variabler er altså permanent. En variabel kan bindes til en verdi et vilkårlig antall ganger, også til verdier av forskjellig type. Tilgjengelige typer av verdier er integere (`<Integer>` i Fig. 1) og boolske (`<Boolean>` i Fig. 1). Når en variabel bindes på nytt, erstatter den nye verdien den gamle. Semantikken til uttrykkene går frem av syntaksen.

Tolkeren

Tolkeren kan ansees som en formell definisjon av semantikken. Den består av hovedfunksjonen `Execute` og noen hjelpefunksjoner.

Funksjonen `Execute` tar som innputt en utførselstilstand og returnerer listen av utførselstilstandene som oppstår under utførsel, ett steg om gangen, inntil utførselen er ferdig. Hvert utførselssteg representeres ved en record som er definert i Figur 2.

En utførselstilstand består av listen av gjenværende setninger og gjeldende lager. Når tolkingen starter, blir `Execute` kalt med en utførselstilstand som inneholder hele programmet og et tomt lager. I løpet av utførselen vil `Execute` bli kalt rekursivt med utførelsestilstanden som ble returnert fra forrige steg. Utførselen avsluttes når listen av gjenværende setninger er `nil`.

Funksjonen `Evaluate` tar som innputt et uttrykk og et lager, og returnerer verdien av setningen for det lageret.

Funksjonen `Bind` tar som innputt en identifikator, en verdi og et lager og returnerer lageret som oppstår når man binder identifikatoren til verdien i lageret.

Funksjonen `Lookup` tar som innputt en identifikator og et lager, og returnerer verdien som identifikatoren er bundet til i lageret.

Eksempel

Anta vi skal bruke tolkeren til å utføre:

```
SimpleProgram = [assign('x' 2) assign('x' add('x' 2)) print('x')]
Trace = {Execute state(SimpleProgram empty)}
```

Da vil svaret `4` vises i browser-vinduet, `Trace` vil inneholde følgende:

```
[state([assign(x 2) assign(x add(x 2)) print(x)] empty)
 state([assign(x add(x 2)) print(x)] bind(x 2 empty))
 state([print(x)] bind(x 4 empty))
 state(nil bind(x 4 empty))]
```

Delproblemer

a) Vi ønsker å utvide språket med `if`-konstruksjoner. En `if`-setning (`<Statement>`) består av et betingelsesuttrykk (`<Expression>`), en `then`-setning (`<Statement>`) og en `else`-setning (`<Statement>`). Når det utføres, vil først betingelsesuttrykket utføres. Hvis resultatet er sant, vil `then`-setningen utføres. Ellers vil `else`-setningen utføres. Finn opp enn passende utvidelse til det semantiske treet for å håndtere `if`-konstruksjoner. Beskriv den som en utvidelse av syntaksen. Skriv Oz-code som får tolkeren å utføre `if`-konstruksjoner korrekt. Indiker linjenummeret i syntaksen (i Fig. 1) og tolkeren der tilleggene dine skal plasseres.

Løsning:

Sett inn følgende etter linje 3 i Figur 1:

```
| if(<Expression> <Statement> <Statement>)
```

Sett inn følgende etter linje 13 i tolkeren:

```
[] 'if'(Expression Statement1 Statement2) then
   ChosenStatement = if {Evaluate Expression Store} then
                        Statement1 else Statement2 end in
   State|{Execute state(ChosenStatement|RestOfStack Store)}
```

b) Vi ønsker å utvide språket med `while`-løkker. En `while`-setning (`<Statement>`) inneholder et betingelsesuttrykk (`<Expression>`) og en hovedsetning (`<Statement>`). Når det utføres, vil betingelsesuttrykket evalueres. Dersom resultatet er sant, vil hovedsetningen utføres og deretter blir `while`-setningen utført igjen. Ellers vil utførselen fortsette med resten av programmet. Finn opp en passende utvidelse av det semantiske treet for å representere `while`-løkker og vis utvidelsen av syntaksen. Skriv Oz-kode som tillater tolkeren å utføre `while`-løkker. Indiker linjenummeret i syntaksen og tolkeren der tilleggene dine skal plasseres.

Løsning:

Sett inn følgende rett etter det som ble satt inn i Fig. 1 i forrige oppgave. (Det blir linje 5.)

```
| while(<Expression> <Statement>)
```

Sett inn følgende etter det som ble satt inn i tolkeren i forrige oppgave. (Det blir linje 18-22.)

```
[] 'while'(Expression Statement1) then
   if {Evaluate Expression Store} then
      State|{Execute state(Statement1|Statement|RestOfStack Store)}
   else State|{Execute state(RestOfStack Store)}
   end
```

c) Beskriv uformelt, ikke med programkode eller grammatikkregler, hvordan du vil utvide det semantiske treet og tolkeren med følgende egenskaper:

- Dynamisk typesjekking
- Statisk typesjekking

Løsning: Dynamisk typesjekking er på en måte allerde implementert i tolkeren i og med at det vil komme system-unntak fra Oz-emulatoren hvis det en operasjon forsøkes utført med inkompatible typer på parameterne. Hvis vi skulle gjøre typesjekkingen mer eksplisitt i tolkeren kunne vi pakke verdiene inn i records hvor merkelappene viser typen. Da kunne vi bruke if-setninger i alternativene i case-setningen funksjonen for evaluering av uttrykk og sendt et unntak til hovedfunksjonen ved typefeil. Begge mulighetene her er OK svar, men et toppsvar bør ha med det andre og kanskje helst begge.

Statisk typesjekking kunne implementeres ved å legge inn typedeklarasjoner av variable i syntaksen og lage en typesjekkingsfunksjon som kalles med det semantiske treet før det skal utføres og sjekker om typingen er riktig. Typesjekkingsfunksjonen vil måtte vite om innputt- og utputt-typer til uttrykksoperasjonene, huske typedeklarasjoner for variablene og kunne dedusere typen til et nøstet uttrykk.

d) Oppførselen til tolkeren i denne oppgaven ligner oppførselen til den abstrakte maskinen som beskriver semantikken i kapittel 2 i læreboka. Den abstrakte maskinen bruker 'sammenheng' (eng: *environments*), mens denne tolkeren ser ut til å fungere tilfredsstillende uten. Hvorfor?

Løsning: Kjernespråket i kaptittel 2 har prosedyrer. Dette betyr at et variabelnavn kan referere til forskjellig variabel på forskjellige steder i programteksten, og for å håndtere dette bruker den abstrakte maskinen omgivelser (eng: *environments*).

```
<Program>      ::= "["{<Statement>}"]"
<Statement>    ::= assign(<Identifier> <Expression>)
                 | print(<Expression>)
<Expression>   ::= add(<Expression> <Expression)
                 | subtract(<Expression> <Expression>)
                 | multiply(<Expression> <Expression>)
                 | divide(<Expression> <Expression>)
                 | equals(<Expression> <Expression>)
                 | lessthan(<Expression> <Expression>)
                 | greaterthan(<Expression> <Expression>)
                 | <Identifier>
                 | <Value>
                 | 'not'(<Expression>)
<Value>        ::= <Integer>|<Boolean>
<Identifier>   ::= <Letter>{<Letter>}
<Letter>       ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<Integer>      ::= <NonzeroDigit>{<Digit>}
<Digit>        ::= 0|<NonzeroDigit>
<NonzeroDigit> ::= 1|2|3|4|5|6|7|8|9
<Boolean>      ::= true|false
```

Figur 1: Syntaks for det semantiske treet.

```
<State> ::= state(<Program> <Store>)
<Store> ::= bind(<Identifier> <Value> <Store>)|empty
```

Figur 2: Syntaks for representasjon av utførselstilstandene.

Appendiks : Tolker for semantisk tre

```
declare

fun {Execute State}
   case State
   of state(nil _) then [State]
   [] state(Statement|RestOfStack Store) then
      case Statement
      of assign(Identifier Expression) then
         State|{Execute state(RestOfStack
                              {Bind Identifier {Evaluate Expression Store} Store})}
      [] print(Expression) then
         {Browse {Evaluate Expression Store}}
         State|{Execute state(RestOfStack Store)}
      else raise invalidStatement(Statement) end
      end
   else raise invalidState(State) end
   end
end

fun {Evaluate Expr Store}
   case Expr
   of add(A B) then {Evaluate A Store}+{Evaluate B Store}
   [] subtract(A B) then {Evaluate A Store}-{Evaluate B Store}
   [] multiply(A B) then {Evaluate A Store}*{Evaluate B Store}
   [] divide(A B) then {Evaluate A Store} div {Evaluate B Store}
   [] equals(A B) then {Evaluate A Store}=={Evaluate B Store}
   [] greaterthan(A B) then {Evaluate A Store}>{Evaluate B Store}
   [] lessthan(A B) then {Evaluate A Store}<{Evaluate B Store}
   [] 'not'(A) then {Evaluate A Store}==false
   else if {IsAtom Expr} then {Lookup Expr Store}
        elseif {IsNumber Expr} orelse Expr==false orelse Expr==true then Expr
        else raise couldntEvaluate(Expr Store) end
        end
   end
end
```

```
44 fun {Bind Identifier Value Store}
45    case Store
46    of empty then bind(Identifier Value Store)
47    [] bind(OldIdentifier OldValue RestOfStore)
48    then if OldIdentifier==Identifier then bind(Identifier Value RestOfStore)
49         else bind(OldIdentifier OldValue {Bind Identifier Value RestOfStore})
50         end
51    else raise invalidStore(Store) end
52    end
53 end
54
55 fun {Lookup Identifier Store}
56    case Store
57    of empty then raise identifierNotInStore(Identifier Store) end
58    [] bind(BoundIdentifier Value RestOfStore) then
59       if BoundIdentifier==Identifier then Value
60       else {Lookup Identifier RestOfStore}
61       end
62    else raise invalidStore(Store) end
63    end
64 end
```

Svarark for Oppgave 1, Flervalg

Fyll inn studentnummeret og dine svar på Oppgave 1 på denne siden.

Studentnummer: .....

	1.	2.	3.	4
a)				
b)				
c)				
d)				
e)				
f)				
g)				
h)				
i)				
j)				

Husk å levere inn denne siden sammen med resten av din besvarelse!

EKSAMENSSLUTT