Norges Teknisk-Naturvitenskapelige Universitet
Fakultet for Informasjonsteknologi, Matematikk og Elektroteknikk
Institutt for Datateknikk og Informasjonsvitenskap

Eksamen i TDT4165 Programmeringsspråk

(med rette- eller løsningsforslag)
Fredag 18 Desember 2009

Laget av: Øystein Nytrø

Godkjent av: Flere

Eksamen har 5 oppgaver, hver oppgave har ett eller flere spørsmål.

For å få maksimalt antall poeng bør du svare riktig på alle spørsmål i alle oppgaver. Dersom du svarer feil eller lar være å svare på ett spørsmål, blir resultatet ditt tilsvarende redusert. Ulike oppgaver gir ulikt antall poeng; se oppgavene for detaljer. Alle kode-eksempler er skrevet i Oz. Kode du skriver skal også være i Oz.

Svar kort og klart, uten tekst som ikke er relevant eller ikke bidrar til svaret. Gjør og beskriv nødvendige antakelser.

Løsning/Rettevegledning Eksamen bestod av varianter og modifikasjoner av oppgaver som har vært gitt tidligere eller burde være kjent fra øvinger og forelesninger. Hensikten med eksamen var ikke å teste evne til å sette seg inn i kompliserte oppgaver, men heller å teste grunnleggende kunnskaper over et bredt spekter. Resultatet var at mange gjorde det forholdsvis bra, og mange hadde god kontroll på semantikken i kjernespråket og implementasjon av unntakshåndtering, lat evaluering, tråder og relasjonell programmering. Eksamen inneholdt ikke en spesifikk lur/vanskelig oppgave. Det førte til relativt mange A-besvarelser. Den viktigste metalærdommen er:

- Finn, og svar på, alle delspørsmål. Selv om det bare er en bisetning.
- Notater og øvinger er også pensum.
- Emner som ble gitt stor oppmerksomhet i forelesning er gjerne eksamensrelevante...
- Uformelle forklaringer, ikke gjort ved hjelp av formalnotasjonen fra boka, må også være presise i terminologi og komplette.

Del 1 Lat utførelse (25%)

Oppg. 1.1. Hva er *lat utførelse*? Forklar hensikten med lat utførelse.

Løsning/Rettevegledning Lat utførelse er behovsdrevet. En setning utføres bare når resultatet er etterspurt, og ikke nødvendigvis i programmets tekstlige rekkefølge. Lat utførelse er nyttig for å implementere beregninger med uendelige datastrukturer (f.eks. strømmer) eller for å implementere behovsdrevet parallellitet.

Oppg. 1.2. En lat funksjon i Oz defineres med det reserverte ordet `lazy`. Men `lazy` er bare syntaksmelis, kjernespråket bruker `ByNeed`. Vis hvordan definisjonen under kan oversettes til kjernespråket.

```
fun lazy {SumSquares N1 N2}
   N1*N1+N2*N2
end
```

Hint: Ikke bare `lazy` er syntaksmelis!

Løsning/Rettevegledning

```
SumSquares = proc {$ N1 N2 ?Result}
  local Compute
    in Compute = proc {$ ?R}
      local M1 in local M2 in
         M1 = {Number.'*' N1 N1}
         M1 = {Number.'*' N2 N2}
         R =    {Number.'+' M1 M2}
    end end end
  {ByNeed Compute Result}
  end
end
```

Oppg. 1.3. Med lat utførelse, implementer funksjonen `ListEvenIntegers` som returnerer *alle* partallene, fra og med 0 (`[0 2 4 ...]`). `ListEvenIntegers` skal virke slik:

```
EvenIntegers = {ListEvenIntegers}
{Browse {ListItem EvenIntegers 1}}
{Browse {ListItem EvenIntegers 5}}
```

med `0` og `8` som utskrift. Funksjonen `ListItem` har en liste og et positivt heltall som argument, og funksjonsverdien er elementet på plassen angitt av heltallet. Første element har plass 1. Implementer `ListItem` med bruk av `lazy` eller `ByNeed`.

Løsning/Rettevegledning `Browse` vil ikke trigge en lat evaluering, så oppgaven er uheldig/feil formulert. `ListEvenIntegers` og hjelpefunksjonen `ListItem` kan implementeres:

```
fun {ListEvenIntegers}
  fun lazy {Enumerate N}
    N|{Enumerate N+2}
  end
in
  {Enumerate 0}
end

fun {ListItem List N}
  case List
  of Head|Tail then
    if N == 1 then Head
    else {ListItem Tail N-1}
    end
  end
end
```

Oppg. 1.4. Hvilke egenskaper må den abstrakte maskinen ha for å støtte lat utførelse? Grei ut om semantikken til lat utførelse generelt, og mer spesielt:

(a) Hvordan utføres ({`ByNeed` ⟨x⟩ ⟨y⟩}, E)? (⟨x⟩ og ⟨y⟩ er identifikatorer, og E er en omgivelse.)

Løsning/Rettevegledning Den abstrakte maskinen må ha et utløser-minne (trigger). Detaljene for semantikken er angitt på s. 282 i Seif & Haridi.

(b) Hva skjer om man trenger en variabel som en lat funksjon er anvendt på? Tolk *trenger* i denne sammenhengen. Gi to eksempler, ett med en variabel som trengs, og ett med en som ikke trengs.

Løsning/Rettevegledning En anvendelse av en lat funksjon er det samme som en *by-need* anvendelse av en prosedyre med ett argument på en ubundet variable som bindes som et

resultat av anvendelsen (etter behov). En variabel trengs om verdien trengs i en beregning/operasjon. Om variabelen er bundet, hentes verdien fra skrivengangs-minnet; Om variabelen ikke er bundet, men finnes som andreelement i et par i trigger-minnet, startes en ny tråd. Hvis variabelen er ubundet og det ikke finnes en tilsvarende trigger, så suspenderes beregningen. I programmet under trengs `Var` i første linje, men ikke i andre:

```
{Browse Var+0}
{Browse Var}
```

jfr. feilformuleringen i første deloppgave...

(c) Forklar de to minnehåndteringsreglene som er nødvendige for å håndtere variabel-rekkevidde (reachability) i lat utførelse.

Løsning/Rettevegledning En variabel er nåbar om den forekommer som første element i et par i utløser-minne, og det andre elementet er nåbart. Om en variabel ikke er nåbar, så må alle par hvor variabelen forekommer som andre element fjernes fra utløser-minne.

Oppg. 1.5. Forklar forskjellen i utførelse for:

```
thread {Procedure Argument} end
{ByNeed Procedure Argument}
```

Løsning/Rettevegledning Den første linjen resulterer i start av en ny tråd som vil anvende prosedyren på argumentet (når det er trådens tur). Den andre linjen resulterer i en tråd om argumentet er bundet, eller en utløser, men ikke en tråd, om argumentet ikke er bundet.

Del 2 Grammatikk og parsing (25%)

Gitt grammatikken

```
<select stmt> ::= 'select' <name list> 'from' <name list> 'where' <expr>
<name list>   ::= <name> | <name> ',' <name list>
<expr>        ::= <name> '=' <value>
```

Gå ut fra at en leksikalsk analysator oversetter terminalordene `'select'`, `'from'`, `'where'`, `','`, og `'='` til atomer i Oz, og `<name>` som `name(L)` og `<value>` til `val(V)` hvor `L` er et atom som svarer til navnet og `V` er et heltall. En setning representeres som en liste av slike verdier.

Oppg. 2.1. Egner grammatikken seg for enn rekursiv nedstigningsparser? Forklar! Vis og forklar de språkbevarende transformasjonene som skal til for at grammatikken egner seg til rekursiv nedstigningsparsing.

Løsning/Rettevegledning Grammatikkens andre regel må venstrefaktoriseres. Resultatet blir enten en ny `<name list>`-regel eller f.eks.

```
<name list> ::= <name> [',' <name list>]
```

Oppg. 2.2. Lag funksjonen `{Expr In ?Out}` som kjenner igjen setninger avledet fra ikke-terminalordet `<expr>` fra starten av lista med token `In`, f.eks. `[name(iter) '=' val(37000) ...]`. `Out` er resten av token som ikke er brukt `...`, og funksjonsverdien er en representasjon av parse- eller syntakstreet for setningen.

Løsning/Rettevegledning

```
declare
fun {Expr In Ut}
   case In
   of name(N)|'='|val(V)|R then Ut=R erLik(N V)
   end
end
```

```
declare N

{Browse {Expr [name(jon) '=' val(4)] N}}
{Browse N}
```

Oppg. 2.3. Lag tilsvarende en funksjon `{NameList In ?Out}` som kjenner igjen setningsformer avledet fra `<name list>`.

Løsning/Rettevegledning For eksempel:

```
declare
fun {NameList In Out}
   case In
   of name(N)|R then
      case R
      of ','|R2 then navn(N)|{NameList R2 Out}
      else Out=R navn(N)|nil
      end
   else Out=In nil
   end
end

declare N
{Browse {NameList [name(olav) ',' name(jens) ',' name(helge)] N}}
{Browse N}
```

Oppg. 2.4. Lag også en funksjon `{SelectStmt In ?Out}` som kjenner igjen setningformer utledet fra `<select stmt>`.

Løsning/Rettevegledning

```
declare
fun {Select I1 Ut} S I2 F I3 B in
   if I1.1=='select' then
      S={NameList I1.2 'from'|I2}
      F={NameList I2 'where'|I3}
      B={Expr I3 Ut}
      select(S F B)
   end
end

declare N
{Browse {Select Lx N}}
{Browse N}
```

Oppg. 2.5. Skriv resultatet av å anvende (og Browse) `SelectStmt` på

```
['select' name(f1) ',' name(f2)
 'from' name(tab)
 'where' name(f1) '=' val(4) tidelibom ],
```

Løsning/Rettevegledning Det er ikke gitt noen krav til funksjonsverdi, - så resultatet av å `Browse` kommer an på hvilken representasjon du har valgt... Men argumentet som blir bundet til resten av lista bør inneholde `tidelibom`.

Del 3 Relasjonsprogrammering (15%)

Oppg. 3.1. Skriv en funksjon som utnytter relasjonsmodellen til å produsere alle lister som er permutasjoner av en vilkårlig liste. Vis hvordan funksjonen kan kalles med `SolveAll`. Hint: Funksjonen {`Append`

`List1 List2}`!

Løsning/Rettevegledning Det er mange løsninger som er mulig, og hintet var ikke spesielt nyttig[1]. Er du interessert i permutasjonsalgoritmer og - generatorer, så er en primærreferanse Donald Knuth: *The Art of Computer Programming - Volume 4 Fascicle 2 - Generating All Tuples and Permutations*, 2005. Løsningen baserer seg på å fjerne ett vilkårlig element fra listen, legge det først i konkatenering med den permutere resten. Testingen kan gjøres mer kompakt, om enn mer kryptisk.

```
declare
fun {Perm List}
   case List
   of [_] then List
   [] _|_ then Rest in {Pick List Rest}|{Permutations Rest}
   end
end

fun {Pick List Rest}
   H|T = List in
   choice Rest = T H
   [] Rest2 in Rest = H|Rest2 {Pick T Rest2}
   end
end
```

Oppg. 3.2. Forklar hvilke egenskaper i kjernespråket som relasjonsprogrammering er avhengig av.

Løsning/Rettevegledning Det viktigste poenget er muligheten til å etablere tråder i egne beregningsrom, som er underrom av et annet rom. Hvert rom har et sett av beskrankninger, vokterbetingelser, som avgjør om tråden i rommet kan kjøre eller ikke, eventuelt om den skal termineres. Hvert slikt delrom har mulighet for å ha egne, lokale variable, og variabelbindinger som kan inkludere variable i overliggende rom. Disse kan oppheves om beregningsrommet termineres (med sin tråd). Kjernespråket er utvidet med `choice` for å opprette et slikt underrom, med voktere, bindinger, variable og tråder. `fail` er en eksplisitt terminering av tråden i delrommet og sletting av bindinger og annet som var i delrommet.

Del 4 Unntak(10%)

Tenk deg en kjøring av programmet under:

```
local X Y Z in
   X = Y
   try
      X = 1 Y = 2 Z = 3
   catch Exception then
      skip
   end
   {Browse X#Y#Z}
end
```

Oppg. 4.1. Blir det skrevet ut noe, og i tilfelle, hva?

Løsning/Rettevegledning `1#1#_` skrives ut!

Oppg. 4.2. Forklar hvordan `try-catch` setningen utføres på den abstrakte maskinen, og bruk dette til å grunngi forrige svar. Formell forklaring er gild, men uformell forklaring er også greit.

[1] Ikke engang for det store relasjonelle programmeringsdyret, som noen mente var relevant i løsningen.

Løsning/Rettevegledning `try-catch` er beskrevet i avsnitt 2.6 i CTMCP. Generelt, når et unntak kastes, så poppes stakken til den kjørende tråden. I eksemplet over er det bare en tråd. Første leksikalske (tekstlige) `catch`-setning blir utført, men bindinger i skrive-lageret blir IKKE omgjort. I programmet over blir unntaket kastet idet `Y`, som er bundet til `X`, forsøkes bundet til `2`. `Z` forblir ubundet.

Del 5 Funksjonell programmering (25%)

Løsning/Rettevegledning Rettekommentar: Svært mange forskjellige knotete måter å løse et knotete problem: Se etter rekursjonsstruktur som tilsvarer typen. Noen komplikasjoner rundt at bare ikke-blad har en verdi. Ubundne variable ble lemfeldig håndtert. Antakelsene var mange og frivole. I denne oppgaven skal vi bruke en datastruktur gitt av grammatikken under:

```
<Tree>    ::= leaf | tree(val:<Value> left:<Tree> right:<Tree>)
<Value>   ::= ...
```

Setningene i språket er post-verdier i `Oz`. `<Value>` står for et heltall eller en variabel. I eksemplet under så er `Tree1` bundet til en postverdi som er språklig velforma.

```
Tree1 = tree(val:1
             left:tree(val:2
                       left:leaf
                       right:leaf)
             right:tree(val:3
                        left:leaf
                        right:tree(val:4
                                   right:leaf
                                   left:leaf)))
```

Oppg. 5.1. Skriv en funksjon {`TreeSum Tree`} som summerer alle verdiene i treet. For eksempel er {`TreeSum Tree1`} = 10. Alle ubundne verdier skal forbli ubundne, men regnes som 0.

Oppg. 5.2. Kjører løsningen på forrige problem problem med konstant stakkstørrelse? Forklar.

Oppg. 5.3. Definer {`Tfold F U Tree`} som traverserer et tre som over, og nytter en funksjon med tre argumenter på verdien til en node og verdien av å nytte funksjonen på høyre og venstre deltre. Blader har verdien U. Ubundne variable skal ikke bindes, men regnes å ha verdien U. For eksempel:

```
{Tfold  fun {$ X Y Z} X + Y + Z end 0 Tree1}    =  10,
{Tfold fun {$ X Y Z} X * Y * Z end 1 Tree1}     =  24
{Tfold fun {$ X Y Z} o(X Y Z) end leaf Tree1}   =  o(1 o(2 leaf leaf)
                                                       o(3 leaf o(4 leaf leaf)))
```