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

Eksamen i TDT4165 Programmeringsspråk

(med rette- eller løsningsforslag)
09.00-13.00, 15. desember 2011

Språk: Bokmål

Faglig kontakt under eksamen:

- Øystein Nytrø Tlf 91897606

Hjelpemidler: C. Bare godkjent kalkulator tillatt. Ingen handskrevne eller trykte hjelpemidler.

Eksamen har 18 deloppgaver med lik vekt. Svar kort og presist. Om noe er uklart, forklar hva du forutsetter. **Alle programmer skal skrives i Oz.**

Rettevegledning *. . . foreløpig og kan endre seg fram til alle besvarelser er rettet og endelig sensur er falt.* ■

Del 1 Beregningsmodell og semantikk

Oppg. 1.1. Hvilke utvidelser av beregningsmodellene i Oz kjenner du? Hva skiller dem fra hverandre?

Rettevegledning *Se forelesning 22, oppsummering, som gjennomgår dette.*

B Den **deklarative, sekvensielle** *utførelsesmodellen fra kapittel 2 av CTMCP.*

Noen mulige utvidelser:

E Unntakshåndtering `try ..... catch ...then ...end`, `raise ... end`

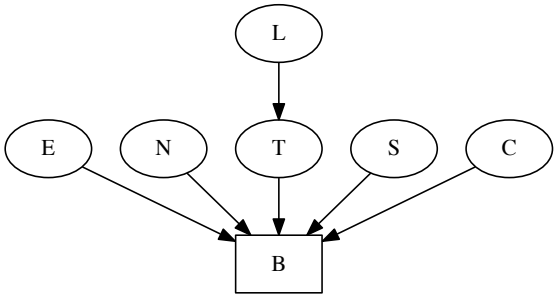
N Navn og bare lesbare variable : `NewName, !!`

T Samtidighet med tråder: `thread ...end`

L Lat utførelse: `ByNeed, lazy`

S Muterbar/endrebar tilstand (celler): `NewCell, Exchange`

C Ikkedeterministisk valg: `choice, fail, Solve`



■

Oppg. 1.2. Skriv de semantiske reglene som utvider beregningsmodellen med unntaksbehandling (Hint: `try, catch, raise`).

Rettevegledning *Se forelesning 8:*

Ved semantisk uttrykk

(`try` ⟨*statement*⟩$_1$ `catch` ⟨*id*⟩ `then` ⟨*statement*⟩$_2$ `end`, E)

Gjør:

1. *Dytt* (`catch` ⟨*id*⟩ `then` ⟨*statement*⟩$_2$ `end`, E)
2. *Dytt* (⟨*statement*⟩$_1$, E))

Ved:

(`raise` ⟨*id*⟩`end`, E)

Gjør:

1. *Hvis stakken er tom, stopp og rapporter ufanget unntak.*
2. *Ellers, hvis toppsetning ikke er en* `catch`*-setning, gå til 1.*
3. *Toppsetningen skal være på formen:* (`catch` ⟨*id*⟩$_c$ `then` ⟨*statement*⟩$_c$`end`, E$_c$).
   Dytt (⟨*statement*⟩$_c$, E$_c$ + {⟨*id*⟩$_c$ → E(⟨*id*⟩)}

Ved:

(`catch` ⟨*id*⟩ `then` ⟨*statement*⟩`end`, E)

Gjør:

1. *Ingenting, ekvivalent med* `skip`. *Et* ⟨*statement*⟩ *i en* `catch`*-setning utføres bare når stakken gjennomsøkes etter en* `raise`.

■

Oppg. 1.3. Er samtidige (concurrent) programmer med unntak deklarative? Forklar.

Rettevegledning *Flere forklaringer. Gode eksempler i f.eks. forelesning 12. Generelt: Unntakshåndtering alene er (observasjonsmessig) deklarativt. Samtidighet (concurrency) med tråder og delte logiske variable er ikke (observasjonsmessig) deklarativt hvis det er mulig at unifikasjonsfeil ikke fører til at hele programmet terminerer. Kombinasjonen av unntak og samtidige tråder gjør at en unifikasjonsfeil kan fanges, og at ikke-deterministiske, ellers feilende program, lykkes med uforutsigbart resultat, som mao. er ikke-deklarativt.* ■

Oppg. 1.4. Begrepet binding brukes (litt uheldig) på to forskjellige fenomener: (a) på en variabelidentifikator som er bundet, i motsetning til fri, i et leksikalt (tekstlig) skop. (b) på en logisk variabel som er bundet, i motsetning til ubundet, til en verdi eller annen variabel i et dynamisk (ved kjøretid) skop? Forklar, og gi eksempler på forskjellen med små programmer.

Rettevegledning *(a) I setningen* `local X in X=1 end` *er* `X` *bundet. I setningen* `X=1` *er X ubundet. Denne betydningen refererer til om en variabel er deklarert i eller utenfor et tekstlig skop. (b) Logiske variable har en levetid under programutførelse hvor de ved deklarasjon er ubundne, men kan bindes til andre variable eller (delvis bundne) verdier. I programmet*

```
1  local X Y in
2  X = Y
3  Y = 2
4  end
```

er X og Y ubundne i linje 1, bundne til hverandre i linje 2 og begge bundne til `2` *i linje 3.* ■

Oppg. 1.5. Hva er lat utførelse og hvordan implementeres det i Oz? Definer semantikken for `ByNeed`.

Rettevegledning

Lat utførelse er et viktig programmeringsparadigme som tillater at beregning ikke gjøres før det er nødvendig. Det tillater deklarative strømmer (og rekursive strukturer), og det forenkler modularisering og synkronisering. Se forelesning 13 og CTMCP s. 281.

Ved: ({`ByNeed` ⟨*id*⟩$_1$ ⟨*id*⟩$_2$}, E) *Gjør:*

1. *Hvis* E(⟨*id*⟩$_2$) *er ubundet, legg et par til trigger-lageret:* (E(⟨*id*⟩$_1$), E(⟨*id*⟩$_2$))
2. *Ellers, lag en ny trådstakk og dytt:* ({⟨*id*⟩$_1$ ⟨*id*⟩$_2$}, E)

Når det er behov for en verdi v og triggerlager inneholder et par (v', v), så:

1. *Fjern (v', v) fra triggerlageret.*
2. *Lag en ny trådstakk og dytt: ({$\langle id \rangle_1$ $\langle id \rangle_2$}, {$\langle id \rangle_1 \rightarrow v'$, $\langle id \rangle_2 \rightarrow v$})*
   (hvor $\langle id \rangle_1$ og $\langle id \rangle_2$ er to nye, distinkte *navn).*

■

Oppg. 1.6. Gitt programmet:

```
local Z
  fun lazy {F1 X} X+Z end
  fun lazy {F2 Y} Z=1 Y+Z end
in
  {Browse {F1 1} + {F2 2} }
end
```

Hvilket, om noe, tall vil skrives ut? Er programmet deterministisk og/eller deklarativt? Forklar.

Rettevegledning *Det vil skrives ut* `5`. *Programmet er deterministisk og deklarativt fordi det blir utført, og det kan bare produsere ett resulat.* ■

Oppg. 1.7. Gitt programmet over, forklar hva som vil skje om vi har lat utførelse og dataflyt-variabler, men ikke samtidighet (concurrency)?

Rettevegledning *Lathet og dataflytvariable uten samtidighet vil gi vranglås, fordi F1 vil vente på at* `Z` *får en verdi, og* `F2` *vil ikke utføres før* `F1` *og dermed blir ikke* `Z` *bundet.*

Lat utførelse forutsetter (implisitt) samtidighet, fordi den late funksjonen utføres i en egen tråd. Så det er også riktig å si at latskap ikke er mulig uten samtidighet. ■

Del 2 Tilstand og abstraksjon

Oppg. 2.1. Programmer funksjonen `{SumList NumList}`, hvor f.eks. `{SumList [1 2 3]}` returnerer `6` rent deklarativt, og halerekursivt, med bruk av en indre funksjon `{SumList2 NumList Accu}` hvor `Accu` akkumulerer summen så langt.

Rettevegledning

```
declare SumList SumList2
fun {SumList NumList}
   fun {SumList2 NumList Accu}
      case NumList
      of nil then Accu
        [] H|T then {SumList2 T H+Accu}
      end
   end
   {SumList2 NumList 0}
end
```

■

Oppg. 2.2. Programmer `{SumListS NumList}` som gir samme resultat som `{SumList NumList}`, men som bruker ekstern akkumulator implementert som celle (hint: `NewCell`).

Rettevegledning *For eksempel:*

```
declare SumList
fun {SumList NumList}
      Result = {NewCell 0}
      Input = {NewCell NumList}
      proc {ISum}
        case @Input of nil then skip
[] Number | Numbers then
```

```
   Result := @Result + Number
   Input := Numbers
   {ISum}
   end
      end
      in
   {ISum}
            @Result
   end
```

■

Oppg. 2.3. Forklar hvilke forskjellige typer av parameteroverføring som finnes, og hvordan de kan simuleres/implementeres i Oz.

Rettevegledning *Se CTMCP s. 430–435. I hovedsak:*

Call by reference *Prosedyren har tilgang til variabelen som aktuelt parameter. Standard i Oz, dvs. variabelen kan bindes i prosedyren.*

Call by variable *Spesialtilfelle av Call by reference. Prosedyren kan f.eks. kopiere en referanse til en variabel, for deretter å kunne bruke den lokale variabelen som et alias.*

Call by value *Verdien av aktuelt parameter overføres, men prosedyren kan ikke binde/endre det aktuelle parameteret.*

Call by value-result *Effekt som i Call by reference, men implementeres ved å kopiere verdien, og når prosedyren terminerer, binde eller oppdatere det aktuelle parameteret. Det aktuelle parameteret vil altså ikke endres mens prosedyren er aktiv.*

Call by name *Aktuelt parameter evalueres i sitt definisjonsnavnerom først ved behov. Implementeres ved at det lages en kontinuasjon, thunk, som beregner parameterverdien når den trengs. Call by name er lat utførelse uten memorering.*

Call by need *En variant av Call by name, hvor thunk-en kalles bare en gang, og ikke hver gang det formelle parameteret brukes. Call by need er det samme som lat evaluering med memorering.*

■

Oppg. 2.4. I Oz er en port en kommunikasjonskanal som implementerer en datastrøm og har operasjonene `{Send Port X}` og `{NewPort Stream ?Port}`. Vis hvordan dette kan implementeres ved hjelp av en celle.

Rettevegledning *Åpen og upakket løsning:*

```
declare
proc {NewPort Stream Port}
   P = {NewCell Stream}
end
proc {Send Port X}
   S=@Port S1 in
   S = X|S1
   Port := S1
end
```

■

Del 3 Høyere ordens og relasjonell programmering

Oppg. 3.1. Implementer `{Filter List Criterion}` som tar en liste og en boolsk funksjon (med ett parameter) som parameter og returnerer en liste som inneholder alle elementene som funksjonen er sann for. Vis hvordan du kan bruke den for å filtrere ut alle partall av en liste.

Rettevegledning

```
declare Filter
fun {Filter List Criterion}
```

```
   case List of Head|Tail then
      if {Criterion Head} then
         Head|{Filter Tail Criterion}
      else {Filter Tail Criterion} end
   else nil end end

{Browse {Filter [10 9 8 7 6 5 4 3 2 1 0] fun {$ V} (V mod 2) == 1 end}
```

■

Oppg. 3.2. Lag en funksjon med ett parameter som må være boolsk funksjon med ett parameter, og returnerer en filterfunksjon. Vis et eksempel på bruk.

Rettevegledning *Svært enkelt, om du forstår prosedyreabstraksjon...*

```
declare MakeFilter
fun {MakeFilter Criterion}
   fun {$ List}
      case List of Head|Tail then
      if {Criterion Head} then
         Head|{Filter Tail Criterion}
      else {Filter Tail Criterion} end
      else nil end
   end
end

declare IsOdd
fun {IsOdd V} (V mod 2) == 1 end

{Browse {{MakeFilter IsOdd} [0 1 2 3 4 5 6]}}
```

■

Oppg. 3.3. 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`. Vis et eksempel på kall, og hva resultatet blir.

Rettevegledning *Det er mange mulige løsninger, men de fleste skriver det eksemplet de har sett før, som likner på løsningsforslaget. Primærreferansen for permutasjonsalgoritmer er 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 permuterte resten. Mange glemte kall med* `SolveAll`*, et brukseksempel og resultat.*

```
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
```

■

Del 4 Grammatikker og parsing

I denne oppgaven skal du lage en rekursiv nedstigningsparser i Oz for noen `SELECT`-setninger i SQL definert i grammatikken:

⟨*select stmt*⟩ ::= 'select' ⟨*name list*⟩ 'from' ⟨*name list*⟩ 'where' ⟨*expr*⟩
⟨*name list*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ ',' ⟨*name list*⟩
⟨*expr*⟩ ::= ⟨*name*⟩ '=' ⟨*value*⟩

Anta at `'select'`, `'from'`, `'where'`, `','`, og `'='` er blitt redusert til leksemer av en leksikalsk analysator (eng: *tokenizer*) som også har laget post-verdier `nm(L)` for ⟨*name*⟩ og `vl(V)` for ⟨*value*⟩, hvor `L` er et atom som representerer identifikatoren og `V` er et heltall. Ikke bry deg om feilhåndtering i parseren.

Oppg. 4.1. Er grammatikken egnet for rekursiv nedstigningsparsing? Forklar. Hvis ikke, transformer grammatikken slik at den er egnet.

Rettevegledning ⟨*name list*⟩ *har to alternativer som begge starter med* ⟨*name*⟩*. Denne må venstre-faktoriseres:*

⟨*name list*⟩ ::= ⟨*name*⟩ *[',' ⟨name list⟩]*

■

Oppg. 4.2. Lag en funksjon `{Expr In ?Out}` som kjenner igjen uttrykk definert ved ⟨*expr*⟩ fra en leksemliste `In`. F.eks. kan `{Expr [nm(i) '=' vl(4) og bortetter] Out}` returnere `equal(i 4)` og binde `Out` til `[og bortetter]`[1].

Rettevegledning

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

declare N
{Browse {Expr [nm(jon) '=' vl(4)] N}}
{Browse N}
```

■

Oppg. 4.3. Programmer funksjonen `{NameList In ?Out}` i Oz som parser ⟨*name list*⟩. For eksempel kan `{NameList [nm(f1) ',' nm(f2) where ever] Out}` returnere `[nm(f1) nm(f2)]`, og binde resten av lista `Out` til `[where ever]`[2]. Det at en ⟨*name list*⟩ representeres som en liste i Oz er bare et forslag.

Rettevegledning

```
declare
fun {NamList In Ut}
   case In
   of nm(N)|R then
      case R
      of ','|R2 then nm(N)|{NamList R2 Ut}
      else Ut=R nm(N)|nil
      end
   else Ut=In nil
   end
end

declare N
{Browse {NamList [nm(f1) ',' nm(f2) ',' nm(f3)] N}}
{Browse N}
```

■

Oppg. 4.4. Lag tilsvarende `{SelectStmt In ?Out}`, som parser hele ⟨*select stmt*⟩. Med `In =`

[1] Det som kommer etter ⟨*expr*⟩ behøver ikke være syntaktisk korrekt.
[2] Se forrige fotnote

```
['select' nm(f1) ',' nm(f2) 'from' nm(tab) 'where' nm(f1) '=' vl(4) osb],
```

skal den binde `Out` til `[osb]` og returnere et syntaks-tre som inneholder informasjonen fra den delen av leksem-lista som ble gjenkjent, for eksempel `sel([nm(f1) nm(f2)] [nm(tab)] equal(f1 4))` eller en annen posttype du finner at passer.

Rettevegledning

```
declare Lx=['select' nm(f1) ',' nm(f2)
                'from' nm(tab)
                'where' nm(f1) '=' vl(4)
                ]

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

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

■