TDT4165 PROGRAMMING LANGUAGES

Fall 2012

Exercise 01
Introduksjon til Oz

Denne øvingen vil gjøre deg kjent med Emacs og Mozart. Du må kunne bruke disse verktøyene for å gjøre resten av øvingene.

Mozart er installert på `selje`. Om du vil gjøre denne øvingen på din egen datamaskin må du installere Emacs og Mozart på din egen datamaskin ved å følge instruksjonene på hjemmesiden til faget.

Det kan være lurt å se gjennom første kapittel i læreboken mens du gjør denne øvingen.

Task 1 Ditt første Oz-program

Start Mozart ved å skrive `oz` (selje) eller ved å velge det fra Start-menyen (Windows). Et Emacs-vindu vil dukke opp. Bruk dette vinduet til å kommunisere med Mozart. I Emacs blir de åpne vinduene kalt *buffers*.

Når du starter Mozart vil man skrive kode i den øverste bufferen, mens output fra kompilatoren vil komme i den nederste. Prøv å skrive følgende kode:

```
{Show 'Hello World!'}
```

Velg `Oz` fra menyen over bufferne. Om du bruker `eos` kan du trykke F10 for å aktivere menylinjen og deretter trykke shift+o. Her kan du se de forskjellige kommandoene du kan gi Oz. Velg `Feed Buffer` for å mate inn koden din. Du vil legge merke til at det ikke skjer så mye. Så hvor ble resultatet av? All utskrift fra `Show` kommer i en annen buffer kalt `Emulator`. For å se denne kan du enten velge den fra `Buffers` på menylinjen eller gå tilbake til Oz-menyen og velge `Show/Hide` etterfulgt av `Emulator`.

En annen å få utskriftsdata på er ved å bruke kommandoen

`Browse`. Endre koden til:

```
{Browse 'Hello World!'}
```

Om du trykker på `Feed Buffer` nå vil ingenting bli skrevet ut i emulatoren. I stedet will et vindu kalt `Oz Browser` poppe opp og skrive ut teksten. Det er flere forskjeller mellom `Show` og `Browse` som du snart vil se.

Task 2 Deklarering av variabler I

`declare` er et reservert ord i Oz. Ved å bruke det kan du lage noe interne variable som man kan gi variable. Oz bruker dynamisk typing som betyr at du ikke trenger å spesifiere om variabelen skal inneholde heltall, tekst eller noe annet. Oz finner det ut når det kjører programmet. Du kan deklarere en variabel og tilegne den en verdi med et steg eller gjøre det hver for seg.

Legg merke til at når man først at tilegnet en variabel i Oz en verdi så kan man ikke tilegne den en ny verdi senere. Eksempel:

```
declare
X = 'if I have been assigned this value'
X = 'I cannot be assigned this one' % static analysis error
```

Det er mulig å deklarere en ny X, men dette vil bare gjemme den gamle variabelen og ikke forandre verdien.

a) Gjør om på koden under sånn at man ikke regner ut X direkte, men lager to andre variabler Y og Z og bruker dem til å regne ut X.

```
declare
X = 42 * 23
{Browse X}
```

Kjør denne koden:

b)
```
declare
X Y
X = 'dette er magisk'
{Browse Y}
Y = X
```

Hvordan tror du `Browse` kan vite verdien til Y selv om den blir kalt før Y blir tilegnet en verdi? Hvordan kan man bruke dette?

Task 3 Funksjoner og prosedyrer

`fun` er et reservert ord man bruker til å lage funksjoner. Dette er det samme som metoder i Java på den måten at de tar argumenter og returnerer en verdi. Under er et eksempel på hvordan det fungerer.

```
declare
fun {Max X Y}
   if X>Y then
      X
   else
      Y
   end
end
```

Oz bruker ikke en spesiell `return` kommando som Java. Det er verdien av uttrykket på den siste linjen som blir returnert.

For å kalle en funksjon bruker man denne syntaksen:

```
{Function-name Arg1 Arg2 .. ArgN}
```

Om vi for eksempel vil vite om 4 eller 7 er størst bruker vi:

```
{Browse {Max 4 7}}
```

`proc` brukes til å lage en prosedyre. De fungerer som funksjoner bortsett fra at du ikke returnerer en verdi.

a) Skriv en funksjon {Min Number1 Number2} som returnerer det minste tallet.

b) Skriv en funksjon {IsBigger Number Threshold} som returnerer sann om Number parameteren er større enn Threshold-parameteren og usann om ikke.

Task 4 Deklarering av variabler II

Om vi vil lage variabler som bare er synlige for deler av programmet, for eksempel en funksjon, kan man gjøre det slik:

```
local
   Variable1 Variable2 .. VariableN
in
   % your code here
end
```

En kortere variant som bruker syntaktisk sukker:

```
Variable1 Variable2 .. VariableN in
   % your code here
```

Variablene som blir deklarert med `local` er lokale innenfor skopet de ble deklarert i.

Skriv en prosedyre {Circle R} som kalkulerer arealet, diameteren og radiusen til en sirkel med radius r og lagrer resultatet i tre variable. Deretter skal du skrive ut resutatet. Bruke utrykket $A = \pi * R^2$, $D = 2R$ og $O = \pi * D$.

Task 5 Rekursjon

Rekursjon er en metode for å kalkulere et svar ved å la funksjonen kalle seg selv direkte eller indirekte helt til man finner svaret. Dette er en parallell til matematisk induksjon. Rekursjon er et av de viktige konseptene i deklarativ programmering, og du vil bruke det mye i senere øvinger.

Denne funksjonen kalkulerer fakultet av et tall (n!):

```
fun {Factorial N}
   if N==0 then
      1
   else
      N * {Factorial N-1}
   end
end
```

a) Skriv en funksjon {SumTo FirstInteget LastInteger} som kalkulerer summen av heltallene mellom FirstInteger og LastInteger (inklusive).

Eksempel: Kjøringen av {SumTo 0 2} bør returnere 3. Kjøringen av {SumTo 3 5} bør returnere 12.

b) Du skal nå lage en ny funksjon kalt {Max X Y} som i likhet med den forrige returnerer det største av to tall X og Y. Forskjellen er at den eneste testen du har lov til å utføre er om tallet er lik null eller ikke. Med andre ord `if N==0 then ... else ...end`. Du kan anta at X og Y er større enn 0.

Du kan bruke følgende ligninger i funksjonen.

- $max(n, m) = m$, hvis $n = 0$.
- $max(n, m) = n$, hvis $m = 0$.
- $max(n, m) = 1 + max(n - 1, m - 1)$, generelt.

Task 6 Lister - teori

Lister er en av de mest viktige datastrukturene i Oz. De brukes til å representere sekvenser av elementer og er ofte brukt med rekursjon for å bygge svaret gradvis. Lister kan bli represert på mange måter. Listene under er alle like.

- List = [1 2 3]
- List = 1|2|3|nil
- List = '|'(1 '|'(2 '|'(3 nil)))

Legg merke til at en komplett liste alltid har `nil` som sitt siste element.

For å hente ut data fra ei liste kan vi bruke prikk-notasjon. Problemet med dette er at man bare kan referere til hodet (head) eller halen (tail) av listen. Dette betyr at man bare kan hente ut det første elementet eller halen av listen. I listene over vil `List.1` returnere 1, mens `List.2` vil returnere `[2 3]`. Om vi vil ha det tredje elementet må vi skrive `[List.2.2.1`, som er tungvint.

En bedre løsning er å bruke mønstergjenkjenning med `case` uttrykk. For å hente ut hodet og halen av listen skriver vi:

```
case List of Head|Tail then
   % code that uses Head and/or Tail
end
```

Det er også mulig å gjøre mer avansert mønstergjenkjenning, for eksempel slik:

```
case List of Element1|Element2|Element3|Rest then
   % code that uses the elements
end
```

Task 7 Lister - praksis

For å forberede deg til øving 2 skal du nå skrive noen funksjoner for å håndtere lister. Du kan ikke bruke de innebygde funksjonene i Oz som gjør akkurat det samme. Hint: Mange av oppgavene kan løses med å kontruere lister gjennom rekursjon. Om du trenger eksempler finnes de i læreboken.

a) {Length Xs} returnerer lengden til listen Xs

b) {Take Xs N} returnerer en liste over de N første elementene av listen Xs. Om N er større enn antall elementer i Xs skal hele listen returneres i stedet.

c) {Drop Xs N} returnerer Xs uten de første N elementene. Om N er større enn antall elementer i Xs skal nil returneres.

d) {Append Xs Ys} returnerer en liste med alle elementene i Xs etterfulgt av alle elementene i Ys

e) {Member Xs Y} returnerer sann om Y finnes i Xs og usann ellers

f) {Position Xs Y} returnerer posisjonen til det første tilfellet av Y i Xs. Du kan gå ut i fra at Y finnes i listen. For eksempel, {Position [1 3 5] 3} returnerer 2.

Task 8 Lekser

En *lekser* (engelsk: lexer, lexical analyzer, tokenizer) er et program som utfører leksikalsk analyse. Det tar inn en sekvens med leksemer og returnerer en liste med tokens.

Implementer funksjonen {Tokenize L} som tar en sekvens av leksemer og returnerer en sekvens av tokens for et lite subsett av det deklarative sekvensielle kjernespråket til Oz.

Funksjonen skal kunne klassifisere ordene `local`, `in`, `if`, `then`, `else`, `end`, identifikatorer, atomer, binære aritmetiske operatorer (`+`, `-`, `*`, `/`), unifikasjonsoperatoren `=`, og sammenlikningsoperatoren `==`.

`Tokenize` skal ta inn en liste med leksemer som strenger og gi ut igjen en liste med tokener.

Se Oz-dokumentasjonen for mer om strenger:
3.8 Lists
2.1 Strings

For eksempel:

```
{Tokenize ["local" "X" "in" "if" "x" "end"]}
 evaluates to
[key("local") id("X") key("in") key("if") atom("x") key("end")]
```

eller

```
{Tokenize ["local" "X" "in" "if" "x" "end"]}
evalutaes to
['local' lex(var:"X") 'in' 'if' lex(atom:"x") 'end']
```

Vi ser ikke på (makro)syntaksen her. Eksemplene over er gode eksempler, men de er ikke syntaktisk riktige. Legg også merke til at variable starter med stor bokstav, mens atomer starter med små.

Hint: Det kan hjelpe å bruke funksjonen List.member (eller Member) som tester om en liste inneholder et objekt:

```
{Member 1 [1]} % evaluates to true
{Member 2 [1]} % evaluates to false
```

Det kan også være greit å vite at strenger i Oz er lister av enkelttegn og at enkelttegn er tallverdier:

```
"abc" == [&a &b &c] ==  [97 98 99]

&a >= &b % false
&a =< &b % true
```

Du må spesifisere hva som skjer om et ulovlig leksem blir funnet, for eksempel `?` eller `&`.