NTNU

Institutt for datateknikk og informasjonsvitenskap

Midterm exam **TDT4165** Programming languages
Midtsemestereksamen **TDT4165** Programmeringsspråk

SOLUTIONS / LØSNINGER

Date / Dato	October 18th 2010 / 18. oktober 2010
Time / Tid	75 minutes
Language / Språk	English / Bokmål
Contact / Kontakt:	Hans Christian Falkenberg (997 21 309)
Support code / Hjelpemiddelkode:	C. No written / handwritten materials. Only specified, simple calculator. Ingen skrevne / håndskrevne hjelpemidler. Kun bestemt, enkel kalkulator.

The weighted sum of this and the final exam, with weights being 30% and 70% respectively, is compared to the final exam only (ie. weight 0% and 100% respectively). The best of these two sums will decide your grade.

- Each task have *zero, one or more* correct options.
- Correct options that are marked yield 2 points.
- Incorrect options that are marked yield minus 1 point.
- Marks must be made on the given answer sheet, which must have this identifier: 14
- All code snippets are Oz code.
- There are many tasks. Do the ones you find easy first and postpone those more difficult.
- Each option has been placed in random order, avoiding human bias.

Den vektede summen av denne og den endelige eksamen, med hhv. 30% og 70% vekt, sammenlignes med kun den endelige eksamen (dvs. vektet hhv. 0% og 100%). Den beste av disse to summene vil bestemme karakteren din.

- Hver oppgave har *null, ett eller flere* riktige alternativer.
- Riktige alternativer som er markerte gir 2 poeng.
- Uriktige alternativer som er markerte gir minus 1 poeng.
- Markeringer må skje på gitt svarark, som må ha følgende svararkidentifikator: 14
- Alle kodesnutter er Oz-kode.
- Det er mange oppgaver. Gjør de du synes er lette først og utsett de som er vanskeligere.
- Hvert alternativ har blitt plassert i tilfeldig rekkefølge for å unngå menneskelig skjevhet.

Task 1

([(`skip`, { })], { }) and ({ [(`skip`, { })] }, { }) are...
([(`skip`, { })], { }) og ({ [(`skip`, { })] }, { }) er...

a) valid **final** states of the **concurrent** and **sequential** abstract machine, respectively /
gyldige **slutt**-tilstander i hhv. den **samtidige** og **sekvensielle** abstrakte maskinen

b) valid initial states of the sequential and concurrent abstract machine, respectively /
gyldige start-tilstander i hhv. den sekvensielle og samtidige abstrakte maskinen
A single semantic statement with an empty environment: (`skip`, { })
This is within a list, the semantic stack.
On the right, this is within the multi-set of semantic stacks corresponding to threads.
The right empty curly brackets represent empty single-assignment stores.

c) valid **final** states of the **sequential** and **concurrent** abstract machine, respectively /
gyldige **slutt**-tilstander i hhv. den **sekvensielle** og **samtidige** abstrakte maskinen

d) valid **initial** states of the **concurrent** and **sequential** abstract machine, respectively /
gyldige **start**-tilstander i hhv. den **samtidige** og **sekvensielle** abstrakte maskinen

Task 2

Oz has... / Oz har...
a) strong scoping / sterke navneområder
This is a mix of the expressions «strong typing» and «dynamic scoping», hope you didn't get it mixed up :)
b) dynamic typing / dynamisk typing
Meaning the type of a variable is determined runtime (when it is bound to a value).
c) weak typing / svak typing
d) static typing / statisk typing
e) weak scoping / svake navneområder
f) strong typing / sterk typing
Meaning there is no automatic type conversation. Ie. operations (like +) cannot be performed on two variables of different type.
g) dynamic scoping / dynamiske navneområder
h) static scoping / statiske navneområder
Also called lexical scoping, this means the environment for a procedure is decided by where it is defined.

This is the syntax of the declarative kernel language defined in chapter 2.3 of CTMCP:
Dette er syntaksen for det deklarative kjernespråket definert i CTMCPs kapittel 2.3:

<statement> ::= `skip`
| *<statement>* *<statement>*
| `local` *<id>* `in` *<statement>* `end`
| *<id>* = *<id>*
| *<id>* = *<value>*
| `if` *<id>* `then` *<statement>* `else` *<statement>* `end`
| `case` *<id>* `of` *<pattern>* `then` *<statement>* `else` *<statement>* `end`
| '`{`' *<id>* { *<id>* }* '`}`'

<value> ::= *<number>* | *<record>* | *<procedure>*
<number> ::= *<integer>* | *<float>*
<pattern> ::= *<record>*
<record> ::= *<literal>*
| *<literal>* '`(`' { *<feature>* **:** *<id>* }* '`)`'
<procedure> ::= `proc` '`{`' `$` { *<id>* }* '`}`' *<statement>* `end`
<literal> ::= *<atom>* | *<bool>*
<feature> ::= *<atom>* | *<bool>* | *<integer>*
<bool> ::= `true` | `false`

<id> starts with an upper case letter, *<atom>* starts with a lower case letter, *<float>* has a dot and a fractional part while *<integer>* has no dot. Beyond that, the exact definitions of these are not important.

Task 3

Which are terminals in the grammar above?
Hvilke er terminaler i grammatikken ovenfor?

a) *<statement>* b) *<value>* c) *<bool>* d) *<pattern>*

All options are non-terminals. Examples of terminals are «true», «local» and «then».

Task 4

Which properties applies to the declarative kernel language with the syntax above?
Hvilke egenskaper gjelder for det deklarative kjernespråket med syntaksen ovenfor?

a) It allows recursive procedure calls / Det tillater rekursive prosedyrekall
A procedure contains a statement and a statement can be a procedure call.

b) It contains lots of syntactic sugar / Det inneholder mye syntaktisk sukker
It's the practical language that has lots of syntactic sugar for constructs on top of the declarative kernel language.

c) It's sequential / Det er sekvensielt
The kernel language with concurrency is introduced in chapter 4.

d) It cannot be extended to support exceptions / Det kan ikke utvides til å støtte unntak
Yes, it can – chapter 2.7 does this.

e) It supports the object-oriented paradigm well / Det støtter det objektorienterte paradigmet godt
Obviously not, there is no syntax for objects.

~~f) It's grammar is unambiguous / Dets grammatikk er utvetydig~~
(This option is removed. The intented answer was wrong and it was only vaguely midterm syllabus.)

Task 5

Which of the following are valid programs according to the grammar above?
Hvilke av de følgende er gyldige programmer i følge grammatikken ovenfor?

a) `local X in if X then skip else skip end` **`end`**

b) `skip skip skip`

c) ~~`declare`~~ **`local`** `Foo in Foo = 2 end`

d)
```
local
   R
in
   local VN in VN = nil
   local V3 in V3 = 3
   local V2 in V2 = 2
   local T3 in T3 = '|'(1:V3 2:VN)
   local L in L = '|'(1:V2 2:T3)
   case 2|3|nil L of H|T '|'(1:H 2:T) then
      R = H
   else
      skip
   end
   end end end end end
end
```

e) `local Y in Y = X end`

This is valid in the grammar. Even though the semantics require that X be declared, so it's not a valid Oz program (but that was not the question).

Needed **additions** *and* **subtractions** *to make all valid marked above.*

Task 6

Which strings can be generated by the following grammar?
Hvilke strenger kan genereres av den følgende grammatikken?

V = { k, j, m }
S = { a, e, o, u }
R = { (k, aja), (k, kk), (k, j), (j, om), (j, mu), (m, ε), (m, eej) }
v_s = k

a) ou	**k** -> ***kk*** -> ***j****k* -> ***om****k* -> *oεk* -> *o****j*** -> *o****mu*** -> *oεu* -> *ou*
b) kaak	*This contains variables and is thus not a string of the language*
c) jeej	*This contains variables and is thus not a string of the language*
d) aau	*k* -> ***kk*** -> ***aja****k* -> *ajak* -> *(j will produce a symbol)*
e) ε	*Only m can produce an empty string, but a symbol will always be produced with m*
f) uee	*... -> **ueej** -> (both rules for j contain a symbol)*
g) oeeu	***k*** -> ***j*** -> ***om*** -> *o****eej*** -> *oee****mu*** -> *oeeεu* -> *oeeu*

Task 7

Which are context-free languages?
Hvilke er kontekst-frie språk?

a) Context-sensitive languages / Kontekst-sensitive språk
Context-sensitive languages is a super-set of context-free languages.

b) V = { u, v }, S = { a, b }, R = { (v, va), (uvu, abba) }, v_s = u
The rule (uvu, abba) is not on the form (v, γ) where v ∈ V and γ ∈ (V ∪ S).*

c) V = { v }, S = { a, b }, R = { (v, a) }, v_s = v
All rules (well, there is just one rule) are on the form (v, γ).

d) Regular languages / Regulære språk
Regular languages is a sub-set of context-free languages.

e) V = { v }, S = { a, b }, R = { (v, va), (v, abba) }, v_s = v
All rules (both of them...) are on the form (v, γ).

Task 8

A grammar can be...
En grammatikk kan være...

a) written in Extended Backus-Naur form / skrevet i Extended Backus-Naur-form

b) ambiguous / tvetydig

c) stateful / tilstandsfull
It's programs that are stateful; and semantics that allow them to be.

d) incomplete / ufullstendig

e) semantics / semantikk
If anything, grammar is syntax.

Task 9

```
local Y X = 2 in
   Y = proc {$ A B C}
          if A < Limit then
             C = X * B
          else
             local P = Y in
                local Y in
                   {P A-1 B Y}
                   C = B * Y
                end
             end
          end
       end
   {Y 2 10 Result}
end
```

In the code above...
I koden ovenfor...

a) `Y` occurs as a formal parameter / forekommer `Y` som et formelt parameter
   `A`, `B` *and* `C` *are formal parameters.* `Y` *is used as an actual paramater.*

b) `X` occurs as a free identifier / forekommer `X` som en fri identifikator
   The second thing that happens is `X` *being declared, it cannot be free after that.*

c) `Y` occurs as a free identifier / forekommer `Y` som en fri identifikator
   The first thing that happens is `Y` *being declared, it cannot be free after that.*

d) `P` occurs as an external reference / forekommer `P` som en ekstern referanse
   `P` *is declared within the procedure.*

e) `Y` maps to at least three different variables (assuming the `else` clause would run) / peker `Y` på minst tre forskjellige variabler (om man antar at `else`-koden vil kjøre)
   Y is only declared twice.

f) `Limit` occurs as an external reference / forekommer `Limit` som en ekstern referanse
   `Limit` *is used within the procedure.*

g) `X` occurs as an external reference / forekommer `X` som en ekstern referanse
   `X` *is used within procedure and declared outside.*

h) `Limit` occurs as a free identifier / forekommer `Limit` som en fri identifikator
   `Limit` *is not declared in the code snippet.*

Task 10

What value would be shown by the following program if Oz used the other major scoping scheme?
Hvilken verdi ville blitt vist av det følgende programmet dersom Oz brukte den andre hovedtypen navneområder?

```
local P X = 4 in
   local X = 8 in
      P = proc {$}
             {Show X}
          end
   end
   local X = 7 in
      {P}
   end
end
```

a) 4
b) 7
c) 8
d) None; it fails when running / Ingen; det feiler når det kjører
e) None; it does not compile / Ingen; det kompilerer ikke

Task 11

What value is really shown by the program above?
Hvilken verdi vises egentlig av programmet ovenfor?

a) 7
b) 4
c) None; it would fail when running / Ingen; det ville feilet når det kjørte
d) None; it would not compile / Ingen; det ville ikke kompilert
e) 8

Task 12

What does this function do?
Hva gjør denne funksjonen?

```
fun {UnknownFunction L N T C}
   case L of nil then N
   [] X|Y then
      {C {T X}
       {UnknownFunction Y N T C}
      }
   end
end
```

a) FoldLeft b) Columns (transpose) c) Filter **d) FoldRight**

Task 13

Given the following abstract machine state, what will be shown (at least one option is correct)?
Gitt den følgende abstrakt maskin-tilstanden, hva vil vises (minst ett alternativ er riktig)?

([(<*s1*>, { X = v5 }), (<*s2*>, { X => v4, Y => v3 }), (<*s3*>, X => v2, Y => v1)],
{ v5=foo(5), v4=4, v3=3, v2=2, v1=1 })
<*s1*> = «raise X end»
<*s2*> = «catch foo(X) then {Show X#Y} raise X end end»
<*s3*> = «catch foo(Y) then {Show X#Y} end»

a) 4#1 b) 2#1 **c) 5#3** d) 5#3, 2#1 e) 4#3, 2#1 f) 4#3 g) 5#1

The second catch pattern will not match, since the raised value is the number from the first match, not a record. The X in the first catch's environment is hidden by the X in the pattern.

Task 14

What does this function do?
Hva gjør denne funksjonen?

```
fun {UnknownFunction X Y}
   (A#B)#(C#D) = X#Y
in
   B = C
   A#D
end
```

a) Filter **b) Append** c) Enumerate d) Zip

Of two diff lists.

Task 15

The function above is a...
Funksjonen ovenfor er en...

a) Producer / Produsent b) Transducer / Omformer c) Consumer / Konsument

Functions of all the kinds listed in the options to list/stream traversal.

Code snippet S1 / Kodesnutt S1:
```
proc {S1 Ys R}
   case Ys of nil then R = 0
   [] Yh|Yt then
      R = Yh + {S1 Yt $}
   end
end
```

Code snippet S2 / Kodesnutt S2:
```
fun {S2 L}
   case L of H|T then
      10 + H | {S2 T} | {S2 T}
   else nil
   end
end
```

Code snippet S3 / Kodesnutt S3:
```
fun {S3 Xs}
   local Result Support in
      proc {Support Xs Result}
         case Xs of nil then
            Result = nil
         [] Xh|Xt then
            local RestResult in
               Result = 10 + Xh | RestResult
               {Support Xt RestResult}
            end
         end
      end
      {Support Xs Result}
      Result
   end
end
```

Code snippet S4 / Kodesnutt S4:
```
fun {S4 L}
   case L of H|T then
      2 * H | {S4 T}
   else nil
   end
end
```

Code snippet S5 / Kodesnutt S5:
```
fun {S5 Ys R}
   case Ys of Yh|Yt then
      {S5 Yt Yh + R}
   else R end
end
```

Task 16

Which of the above code snippets have a tail-recursive procedure or function?
Hvilke av kodesnuttene ovenfor har en hale-rekursiv prosedyre eller funksjon?

a) S5 *Yh+R is done before the call, there is no stack growth.*
b) S3 *Obviously tail-recursive, the call is the last statement in Support.*
c) S1 *The addition must be done after the recursion completes and is put on the stack.*
d) S4 *The value to be returned (a list record) is created with an unbound variable before the call.*
e) S2 *The last recursive call is tail-recursive, but prevents the first from being tail-recursive.*

Task 17

Which of the above code snippets have a recursive procedure or function?
Hvilke av kodesnuttene ovenfor har en rekursiv prosedyre eller funksjon?

a) S5 **b) S4** **c) S2** **d) S3** **e) S1**
S1, S4 and S5 call themselves, Support calls itself, and S2 even calls itself twice.

Task 18

Which of the above code snippets can do an iterative computation?
Hvilke av kodesnuttene ovenfor kan gjøre en iterativ beregning?

a) S2 b) S1 **c) S4** **d) S5** **e) S3**
The ones that are tail-recursive.

Task 19

Which of these functions will well support implementation of the functionality of `S4`?
Hvilke av disse funksjonene vil støttet godt å implementere `S4` sin funksjonalitet?

a) Filter b) StreamFilter c) StreamMult d) Mult
e) Map **f) StreamMap**

```
Filter = fun { $ List Function } ... end
Map = fun { $ List Function } ... end
Mult = fun { $ List } ... end
```

Mult takes a list, that would just complicate matters. Below is how to do it with Map. Using StreamMap instead of Map will be identical, except StreamMap would to work in another thread (or several threads if it was implemented lazy).

```
fun {S1 L}
   {Map S1 fun {$ H} 2 + H end}
end
```

Task 20

What will `{S2 [1 2 3]}` return?
Hva vil `{S2 [1 2 3]}` returnere?

a) `[11 22 33 nil 33 nil 22 33 nil 33 nil]`
b) `[11 22 33]`
c) Nothing; it never returns / Ingenting; den returnerer aldri
d) `[11 22 53]`
e) `[11 [22 [33 nil] 33 nil] 22 [33 nil] 33 nil]`
None of the above...
`[11 [12 [13 nil] 13 nil] 12 [13 nil] 13 nil]`

Task 21

What can this program show (at least one option is correct)?
Hva kan dette programmet vise (minst ett alternativ er riktig)?

```
local X in
   proc {X I N}
      if I < N then
         thread {Show I} end
         {X thread I+1 end N}
      end
   end
   {X 1 4}
end
```

a) 1, 2, 3 *This is what is likely to be printed.*
b) 2, 3, 1 *The showing of 3 could very well be bypassed by the showing of 2 by scheduling.*
c) 1, 2, 4 *N will always be exactly one greater in each recursive call.*
d) 3, 2, 1 *Just put {Delay 1000*(3-I)} in front of {Show I} and this will happen.*
e) 1, 3, 3 *All calls of X will have different values of I.*
The fact that N+1 is computed in a separate thread has no effect on what can be shown (though it might affect probabilities ever so slightly).

Task 22

Which are other representations of `[a b c]`? / Hvilke er andre representasjoner av `[a b c]`?

a) `(a|b|c|_)#_`

This is potentially a diff list, but there are two different unbound variables. They may be bound in such a way that we get the diff list representing [1 2 3], but it could easily be bound another way too. If the variable on the right side of # were to be bound to something that the left side does not end in, it would not even be a diff list (eg. (1|2|3|nil)#4).

b) `[a b c]|nil`

This is a list of lists (well, a list with only one list, really).

c) `[a b c nil]`

d) `'|'(1:a 2:'|'(b '|'(c nil)))`

This is a mix of record and tuple syntax.

e) `'|'(a '|'(b '|'(c nil)))`

This is equal to option d), but with only tuple syntax.

f) `c|b|a|nil`

g) `a|b|c`

h) `(a|b|c|X)#X`

This is a diff list.

Task 23

What properties does the following stack data structure have?
Hvilke egenskap har den følgende stakk-datastrukturen?

```
fun {New}
   nil
end
fun {Push Stack Item}
   Item|Stack
end
fun {Pop Top|Rest ?Item}
   Item = Top
   Rest
end
fun {IsEmpty Stack}
   Stack == nil
end
```

a) Bundled / Buntet
b) Unbundled / Ubuntet
c) Embedded / Innebygd
d) Non-embedded / Ikke innebygd
e) Insecure / Usikker
f) Secure / Sikker

Answer sheet / Svarark

Candidate number / Kandidatnummer:

- There are different answer sheets. This is sheet 14. Make sure this matches your front page.
- Ring each letter corresponding to a correct option for the task.
- Watch out – options appear in random order!

- Det er forskjellige svarark. Dette er ark 14. Sørg for at dette stemmer overens med forsiden din.
- Sett en ring rundt hver bokstav som tilsvarer et riktig alternativ for oppgaven.
- Pass på – alternativene forekommer i tilfeldig rekkefølge!

Example	g	d	b	e	a	f	c	h
Task 1	d	e	a	h	f	g	c	b
Task 2	g	a	c	e	h	f	b	d
Task 3	a	d	b	f	h	c	e	g
Task 4	a	e	b	g	h	c	f	d
Task 5	a	f	b	g	e	c	d	h
Task 6	d	e	f	g	h	b	a	c
Task 7	c	h	d	b	e	a	g	f
Task 8	c	b	g	a	d	e	h	f
Task 9	f	b	c	g	h	a	e	d
Task 10	g	b	d	e	a	f	h	c
Task 11	b	a	g	c	h	f	d	e
Task 12	b	h	d	a	g	f	c	e
Task 13	h	f	b	g	a	d	c	e
Task 14	d	e	f	g	c	a	b	h
Task 15	e	b	d	a	g	h	f	c
Task 16	a	c	g	f	d	e	b	h
Task 17	c	a	d	e	h	f	g	b
Task 18	f	a	e	d	g	c	h	b
Task 19	g	d	f	c	a	h	e	b
Task 20	c	g	a	h	d	b	e	f
Task 21	f	e	a	h	b	c	d	g
Task 22	c	d	b	e	g	a	h	f
Task 23	c	g	d	b	e	a	h	f