NTNU

Institutt for datateknikk
og informasjonsvitenskap

Final exam **TDT4165** Programming languages
Endelig eksamen **TDT4165** Programmeringsspråk

SUGGESTED SOLUTION / LØSNINGSFORSLAG

Date / Dato	December 1st 2010 / 1. desember 2010
Time / Tid	4 hours
Language / Språk	English / Bokmål
Contact / Kontakt:	Hans Christian Falkenberg (997 21 309)
Reviewer / Gjennomgått av:	Øystein Nytrø
Support code / Hjelpemiddelkode:	C. No written / handwritten materials. Only specified, simple calculator. Ingen skrevne / håndskrevne hjelpemidler. Kun bestemt, enkel kalkulator.

- The weighted sum of the midterm exam and this, with weights being 30% and 70% respectively, is compared to this exam only (ie. weight 0% and 100% respectively). The better of these two sums will decide your grade.
- The acronym 'CTMCP' is used to refer to the curriculum book (by van Roy and Haridi).
- If you skip or answer incorrectly a question, your score will be reduced correspondingly. Different tasks contribute differently to the total score; see each task for details.
- All code examples are given in Oz. When asked to write code, write Oz code.
- Use concise answers, without text that is irrelevant or does not contribute to the answer.
- You may also disagree with what is stated in CTMCP or with what was explained during the lectures, but you should give convincing arguments in such cases.
- On average, there is somewhat more than 6 minutes available per subtask.

- Den vektede summen av midtsemestereksamen og denne, 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.
- Akronymet 'CTMCP' brukes for å referere til pensumboka (av van Roy og Haridi).
- Om du utelater eller svarer feil på et spørsmål reduseres poengene dine tilsvarende. Forskjellige oppgaver bidrar forskjellig til totalsummen; se hver enkelt oppgave for detaljer.
- Alle kodeeksempler er gitt i Oz. Når du blir bedt om å skrive kode, skriv Oz-kode.
- Bruk konsise svar, uten tekst som er irrelevant eller ikke bidrar til svaret.
- Du kan være uenig i hva som er oppgitt i CTMCP eller forklart i forelesningene, men du må i så fall oppgi overbevisende argumenter.
- I gjennomsnitt er det noe mer enn 6 minutter tilgjengelig per deloppgave.

This is the grammar for the declarative kernel language (DSKL) defined in chapter 2.3 of CTMCP:
Dette er grammatikken for det deklarative kjernespråket (DSKL) 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 (keywords must be enclosed in apostrophes), <*float*> has a dot and a fractional part while <*integer*> has no dot. Beyond that, the exact definitions of these are not important.

Task 1 – Syntax and semantics (15%)

1.a Briefly explain the meaning of «syntax» in our context.
Forklar kort betydningen av «syntaks» i vår kontekst.

A syntax explains the allowed structure in a language, usually by using a grammar. Extra info: It decides how groups of symbols (tokens) can fit together to make a program (macrosyntax) and also how a token can be built up (microsyntax).

1.b Briefly explain the meaning of «semantics» in our context.
Forklar kort betydningen av «semantikk» i vår kontekst.

Semantics are about how elements how the syntax are to be understood in terms of executing the program.

1.c Explain the parts of the abstract machine for DSKL.
Forklar bestanddelene i den abstrakte maskinen for DSKL.

Semantick stack – A stack of semantic statements, where the topmost element is the next one to be executed. A semantic statement is a statement and an environment (a mapping from identifiers to variables).
Single assignment store – All variables ever declared and their values (for those that are not unbound).
The *execution state* of the machine is a snapshot of the semantic stack and the single assignment store.

1.d Give an example of syntactic sugar for DSKL. Explain both the syntax and semantics.
Gi et eksempel på syntaktisk sukker for DSKL. Forklar både syntaks og semantikk.

(Obviously there are many other examples that may yield a full score.)

Allowing multiple values to be declared at once.

Changing syntax rule

`<statement> ::= local <id> in <statement> end`

to

`<statement> ::= local { <id> }+ in <statement> end`

The semantics is easily explained by translating into the kernel language:

`local` X_0 X_1 ... X_N `in <statement> end`

is translated to

```
local X0 in
   local X1 in
      ...
         local XN in
            <statement>
         end
      ...
   end
end
```

1.e Give an example of a linguistic abstraction for DSKL. Explain both the syntax and semantics.
Gi et eksempel på en lingvistisk abstraksjon for DSKL. Forklar både syntaks og semantikk.

(Exactly when something should be classified as a linguistic abstraction rather than syntactic sugar is not clear cut, borderland examples will be accepted.)

Functions – unlike procedures, they always return a value.

Assuming how to use an expression has already been defined, we add the syntax rules

`<expression> ::= fun '{' $ { <id> }* '}' [ <statement> ] <expression> end`

`<expression> ::= '{' <id> { <id> }* '}'`

Since actual parameters for a procedure call can be unbound at call time and bound during execution, procedures can bind the actual parameters to one or more return values. This makes a translation explanation of the function semantics straight forward:

`Foo = fun {$` P_0 P_1 ... P_N`} <statement> <expression> end`

`Bar = {Foo` A_0 A_1 ... A_N`}`

is translated to

`Foo = proc {$` P_0 P_1 ... P_N `Ret} <statement> Ret = <expression> end`

`{Foo` A_0 A_1 ... A_N `Bar}`

1.f Explain how to add support exceptions, both syntactically and semantically.
Forklar hvordan støtte for unntak kan legges til, både syntaktisk og semantisk.

Extending DSKL syntax:

`<statement> ::= try <statement>`$_1$ `catch <id>`$_C$ `then <statement>`$_2$ `end`

`<statement> ::= raise <id>`$_R$ `end`

Semantics is explained in terms of the abstract machine:

When encountering a semantic statement with `try` and environment E, put two semantic statements on the stack: First `catch ...` and then `<statement>`$_1$., both with environment E.

A semantic statement with `catch ...` is the same as the `skip` statement.

When encountering a semantic statement with raise, pop semantic statements until a `catch ...` is encountered. If one is encounter, put `<statement>`$_2$ with the environment from the catch extended with `<id>`$_C$ = E_{RAISE}(`<id>`$_R$).

1.g With exceptions, is it still the same computation model / kernel language?
Med unntak, er det fremdeles den samme beregningsmodellen / kjernespråket?

No, the semantics cannot be translated into any of the existing features of the DSKL.

Task 2 – Declarativity (10%)

2.a Name the three characteristics from CTMCP's definition of a (definitionally) declarative program unit.
Nevn de tre egenskapene fra CTMCPs definisjon av en (definisjonsmessig) deklarativ programenhet.

Stateless. Independent. Deterministic.

2.b What is observational declarativity?
Hva er observerbar deklarativitet?

That the accessible parts of a program unit (its interface) cannot be used to assert that the unit is not declarative. Ie. its behavior is declarative, so far as external units know, even though the implementation may use non-declarative techniques.
Another way to say it: Any single use of the program unit observed will always yield the same output for any given input.

2.c Can a concurrent program be declarative? Write a few words about why / why not.
Kan et samtidighetsprogram være deklarativt? Skriv noen få ord om hvorfor / hvorfor ikke.

Yes. Oz's dataflow (single-assignment) varaiables guarantee that a run of the program will always bind the variables the same way or always fail with a unification error. Ie. the only possible non-declarative behavior would be to bind variables in a different order, to change the computation. Indeed, variables may be bound in different order, but it will not change the computation (ie. it is still deterministic), because the program will block at the same points when a variable is needed.
Answering no can also give full score, if you argue that the above is only observational declarativity and that the program is no longer definitionally declarative when looking at its individual parts. Ie. actual parameters in a procedure call may be unbound when the procedure returns in one run of the program, but bound when the procedure returns in another run.

2.d Give a code example with a function that is observationally declarative, but not (definitionally) declarative.
Gi et kodeeksempel med en funksjon som er observerbar deklarativ, men ikke (definisjonsmessig) deklarativ.

`Foo` is observationally declarative.

```
local
   Cell = {NewCell 0}
in
   Foo = fun {$}
            Cell := @Cell + 1
            42
         end
end
```

This is a nonsense example with a counter that counts the number of invocations of `Foo`. It is observationally declarative because this counter isn't used for anything. Another nonsense example would be to have a function that waits a random time before returning (random being both non-deterministic and an external dependency).

An example that would make more sense (much sense, in fact) would be to use memoization as an optimization. Kudos if you have such an example :)

Task 3 – Functional and higher-order programming (30%)

3.a Which of these code snippets use higher order programming (zero, one or more)?
Hvilke av disse kodesnuttene bruker høyere ordens programmering (null, en eller flere)?

(i)

```
local
   Pow = fun {$ X Y}
            if Y == 1 then X
            else X * {Pow X Y-1} end
         end
in
   {Show {Pow 2 10}}
end
```

(ii)

```
local
   Show2 = proc {$ Z X Y}
              {Show {Z X Y}}
           end
in
   {Show2 Pow 2 10}
end
```

(iii)

```
local
   PP = fun {$}
           fun {$ X Y Z}
              {Pow X {Pow Y Z}}
           end
        end
in
   {Show {{PowPow} 1 2 3}}
end
```

All.
i: The function is a value that is unified with / assigned to Pow.
ii: The Pow function is passed as an argument to Show2.
iii: PP returns a function value. *(PowPow should be replaced by PP, this was announced at the exam)*

3.b What is higher-order programming?
Hva er høyere ordens programmering?

To use functions as first-class objects / values.

3.c Explain three of these concepts: Procedural abstraction, genericity, instantiation, embedding.
Forklar tre av disse konseptene: Prosedyral abstraksjon, generiskhet, instansiering, innebygging.

(Any three of the below will give a full score).
Procedural abstraction – Giving a group of statements a name that can be used to run them.
Genericity – Taking out statements of a procedure and using an argument to run arbitrary other arguments in their stead.
Instantiation – Returning a new more specific procedure (ie. one with one less argument).
Embedding – Storing a procedure in a data structure.

3.d What is a closure?
Hva er en tillukning?

A procedure value – which consists of the procedure definition and the environment where it was defined.

3.e What will the following program show?
Hva vil det følgende programmet vise?

```
local
   fun {Zip List1 List2}
      case List1#List2
      of (Head1|Tail1)#(Head2|Tail2) then
         (Head1#Head2)|{Zip Tail1 Tail2}
      else nil end end
in
   {Show {Zip [1 2 3] [10 20 30]}}
end
```

```
[1#10 2#20 3#30]
```

3.f Implement `fun {Unzip Splitter List}` (which returns a tuple with two lists) and `proc {TupleSplit Zipped Elem1 Elem2}`. `Splitter` is a three argument procedure that splits the first argument and binds the result to the last two arguments. `TupleSplit` is such a procedure, and it should split elements that were created by the `Zip` implementation above. `Unzip` must use `Splitter` to create two elements for each element of `List`.

Implementer `fun {Unzip Splitter List}` (som returnerer en tuppel med to lister) og `proc {TupleSplit Zipped Elem1 Elem2}`. `Splitter` er en tre-arguments-prosedyre som deler opp det første argumentet og binder resultatet til de to siste argumentene. `TupleSplit` er en slik prosedyre, og den skal dele elementer som ble laget av `Zip`-implementasjonen ovenfor. `Unzip` må bruke `Splitter` for å lage to elementer per element i `List`.

```
proc {TupleSplit Zipped ?Elem1 ?Elem2}
   Elem1#Elem2 = Zipped
end
fun {Unzip Splitter List}
   case List of nil then nil#nil
   [] H|T then X Y A B in
      {Splitter H X Y}
      A#B = {Unzip Splitter T}
      (X|A)#(Y|B)
   end
end
```

3.g Write a line of code to show how `Unzip` and `TupleSplit` can be used. Assume that `List = {Zip [1 2 3] [10 20 30]}` has already been run and that `List` is in scope.

Skriv en linje kode for å vise hvordan `Unzip` og `TupleSplit` kan brukes. Anta at `List = {Zip [1 2 3] [10 20 30]}` allerede har blitt kjørt og at `List` er i navneområdet.

```
local L1#L2 = {Unzip TupleSplit List} in … end
```

The next two subtasks were mistakenly labelled the same as the previous two.
The candidates were told to either use 3.f2 and 3.g2 for the next two subtasks,
or continue the numbering with 3.h, so that the last subtask would end up as 3.l.

3.f2 What does this function do; ie. what should its name be instead of `Foo`?
Hva gjør denne funksjonen; altså hva burde den hete i stedet for `Foo`?

```
fun {Foo M}
   case M of nil|_ then
      nil
   else
      {Map M fun {$ H|_} H end} | {Foo {Map M fun {$ _|T} T end}}
   end
end
```

It transposes a matrix. Name it `Transpose` or `Columns` (as in that it retrieves a list of columns, assuming that the matrix was represented as a list of rows).

3.g2 Give an example of input and output for `Foo` that shows how it works.
Gi et eksempel på innputt og utputt for `Foo` som viser hvordan den virker.

```
[ [1 2 3]                 [ [1 10 100]
  [10 20 30]          =>    [2 20 200]
  [100 200 300]             [3 30 300]
]                         ]
```

3.h What does `FoldRight` do?
Hva gjør `FoldRight`?

Collapses a list by merging the current value (for some initial value) with the rightmost remaining element.

3.i Implement `FoldRight`.
Implementer `FoldRight`.

```
fun {FoldRight List Fun Init}
   case List of nil then Init
   [] H|T then
      {Fun H {FoldRight T Fun Init}}
   end
end
```

3.j Implement `fun {SumList List}`, using `FoldRight`.
Implementer `fun {SumList List}` ved å bruke `FoldRight`.

```
fun {SumList List}
   {FoldRight List fun {$ X Y} X+Y end 0}
end
```

Task 4 – Message based concurrency (15%)

```
declare
fun {NewPortObject InitialState Function}
   Stream
in
   thread _={FoldLeft Stream Function InitialState} end
   {NewPort Stream}
end
NPO = NewPortObject
```

To send a value to a port, use `{Send Port Value}`.
For å sende en verdi til en port, bruk `{Send Port Value}`.

4.a What happens when you send something to a port?
Hva hender når du sender noe til en port?

The stream associated with the port is extended with the value you sent. The port is updated to use the new, unbound end of the stream.

4.b What extensions of the abstract machine for DSKL are required to explain the computation model for the code above?
Hvilke utvidelser må gjøres av den abstrakte maskinen for DSKL for å forklare beregningsmodellen for koden ovenfor?

Multiset of semantic stacks – Each semantic stack represents a thread. The multiset replace the existing single semantic stack.
Mutable store – Has pairs of all allocated ports and their current stream ends.

4.c Use `NPO` to implement a server that counts how many messages it has received. `{MakeServer MyProc Test}` shall return a new instance of the server. For each message received, the server should call `Test` with the message as an argument. Iff `Test` returns `true`, the server should call `MyProc` with the number of messages received so far.

Bruk `NPO` for å implementere en tjener som teller hvor mange meldinger den har mottatt. `{MakeServer MyProc Test}` skal returnere en ny instans av tjeneren. For hver melding skal tjeneren kalle `Test` med meldingen som argument. Bare dersom `Test` returnerer `true`, skal tjeneren kalle `MyProc` med antallet meldinger mottatt hittil.

Example use / eksempel på bruk:

```
local
   ServerA = {MakeServer Show fun {$ Msg} Msg==show end}
   ServerB = {MakeServer Show fun {$ Msg} Msg==show end}
in
   {Send ServerA foo}
   {Send ServerA bar}
   {Send ServerA show} % server A will now show 3
   {Send ServerB show} % server B will now show 1
end
```

```
fun {MakeServer MyProc Test}
   fun {Server Count Msg}
      NewCount = Count + 1 in
      if {Test Msg} then
         {MyProc NewCount}
      end
      NewCount
   end
in
   {NPO 0 Server}
end
```

4.d Can `MakeServer` be used so that the client can do arbitrary work on the server?
Kan `MakeServer` brukes slik at klienten kan gjøre vilkårlig arbeid på tjeneren?

Yes, `Test` can do anything it likes and it will be run on the server.

Task 5 – Relational programming (15%)

```
local
   fun {Bar X}
      local Y in
         Y = choice c [] d end
         (Y \= X) = true
         X#Y
      end
   end
   fun {Foo}
      choice {Bar a} [] {Bar b} [] {Bar c} end
   end
in
   {Show {SolveAll Foo}}
end
```

5.a What is shown by the above code if `SolveAll` uses depth-first search (leftmost choice first)?
Hva vises av koden ovenfor dersom `SolveAll` bruker dybde-først søk (venstre choice først)?

```
[a#c a#d b#c b#d c#d]
```

5.b What is shown if `SolveAll` uses breadth-first search (leftmost choice first)?
Hva vises dersom `SolveAll` bruker bredde-først søk (venstre choice først)?

```
[a#c b#c a#d b#d c#d]
```

5.c Make a program that use the relational computation model to find all possible pairings of people that want to go to the movies together.
Lag et program som bruker den relasjonelle beregningsmodellen for å finne alle mulige sammenkoblinger av folk som vil gå på kino sammen.

Example input / eksempelinnputt:
`Wishes = wishes(`
`alice:[bob charlie dennis]`
`bob:[alice eva foxy]`
`charlie:[alice foxy]`
`dennis:[alice foxy]`
`eva:[bob charlie dennis]`
`foxy:[charlie dennis])`
 	
Output / utputt:
`[[alice#charlie bob#eva dennis#foxy]`
`[alice#dennis bob#eva charlie#foxy]]`
	Example input / eksempelinnputt:
	`Wishes = wishes(`
	`alice:[raymond]`
	`bob:[raymond]`
	`charlie:[raymond]`
	`raymond:nil)`
	
	Output / utputt:
	`nil`

You may find the following functions useful / Du kan kanskje benytte de følgende funksjonene:

`{Member Element List}`	Returns true iff `Element` occurs in `List`.
`{Filter List Function}`	Returns a new list with elements filtered by `Function`.
`{Arity Record}`	Returns a list of all the features in `Record`.

```
declare
fun {Pick List}
   Head|Tail = List in
   choice Head [] {Pick Tail} end
end
fun {Match Wishes People Matched}
   case People of nil then nil
   [] P|Rest then W in
      if {Member P Matched} then
         {Match Wishes Rest Matched}
      else
         W = {Pick Wishes.P}
         {Member W Matched} = false
         {Member P Wishes.W} = true
         P#W | {Match Wishes Rest P|W|Matched}
      end
   end
end
fun {Calc}
   {Match Wishes {Arity Wishes} nil}
end
{Show {SolveAll Calc}}
```

Task 6 – Various (15%)

6.a A data structure can be stateless or stateful. What are the remaining four of the six properties discussed in this course?
En datastruktur kan være tilstandsløs eller tilstandsfull. Hva er de gjenværende fire av de seks egenskapene som har blitt forklart i dette kurset?

Secure, insecure (or open), bundled and unbundled.

6.b Implement a counter data structure that use cells to hold state.
Implementer en teller-datastruktur som bruker celler for å holde på tilstand.

Hint: `{NewCell Init ?Cell}`, `{Exchange Cell ?OldValue NewValue}`.

(The interface of the counter data structure was intentionally not specified, to easily allow different answers to the next subtask. Anything that appears to be counting or keep count will be accepted.)

```
declare
fun {Counter}
   C Increase Get in
   {NewCell 0 C}
   proc {Increase}
      Old in
      {Exchange C Old Old}
      {Exchange C _ Old+1}
   end
   fun {Get}
      Old in
      {Exchange C Old Old}
      Old
   end
   counter(increase:Increase get:Get)
end
```

6.c Which two of the other properties does your counter implementation have?
Hvilke to av de andre egenskapene har din teller-implementasjon?

Secure, bundled. *(Obviously, your implementation may have other properties.)*

6.d What is the difference between passing parameters by reference and by value?
Explain it in terms of memory addresses.

Hva er forskjellen på parametersending med referanse eller verdi?
Forklar det ved hjelp av minneadresser.

Passing by reference means that the function will operate on the memory address of the actual parameter. (This requires, of course, that the actual parameter is a variable that *has* a memory address, as opposed to being a value stored only in the call argument list.)

Passing by value means that a copy of the actual parameter is done. The same value now exists on two different memory addresses and only the newly allocated address is what the function use (and only the function knows of it).

6.e What is lazy evaluation?
Hva er lat evaluering?

Postponing of evaluating a value until it is actually needed. This means that the value must be specified in terms of something that can be evaluated (eg. a function), and that the function must be called to compute the value the first time it is needed (and the value will be known after that).

6.f What is its counterpart called?
Hva kalles dets motstykke?

Eager evaluation.

6.g Give a code example that illustrates the difference between the two.
Gi et kodeeksempel som illustrerer forskjellen mellom de to.

```
fun {Foo} {Show foo} 1 end
fun lazy {Bar} {Show bar} 2 end
fun {Baz} {Show baz} 3 end
fun {Calc A B C}
   A + B + C
end
{Show {Calc {Foo} {Bar} {Baz}}}
```

This will show

```
foo
baz
bar
6
```

which demonstrates that the value from the call to Bar is only evaluated when it is actually needed in the calculation, whereas the other two calls are evaluated before Calc is called.