Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Exam in TDT4165
Programming languages

December 19, 2012

SUGGESTED SOLUTIONS / LØSNINGSFORLAG (v.1)

Language: English
Contact during the exam:

- Øystein Nytrø Tlf 91897606

Exam aids: C. Only approved calculator allowed. No written or printed material permitted. There is only one version of this exam. This one... The exam has 17 subtasks with equal weight. Short and succinct answers are preferred. Do not risk negating a correct answer by exam-bloat! When in doubt about interpretation, state your assumptions.
All programs should be written in Oz.

Part 1 Computational model and semantics

Task 1.1. What is the difference between definitional and observational declarativity? Explain by means of explicit Oz-code examples.

Solution

Definitional declarativity: A programming language that guarantees delarativeness (e.g. Oz, Haskell).

```
declare
fun {Def A B}
   A + B
end
```

Observational declarativity: The accessible parts of a program unit (its interface) cannot be used to assert that the unit is not declarative. I.e. 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 (e.g. Java).

```
declare
fun {Obs A B}
   X Y in
   X = {NewCell 0}
   thread
      X := @X+A
   end
   thread
      X := @X+B
      Y = @X
   end
   Y
end
```

```
{Browse {Obs 3 4}}
```

Task 1.2. Explain the new semantic rules and other changes necessary to extend the computation model with relational computation.

Solution

The relational model of computation covered in Ch. 9 is an extension of the declarative sequential model of computation from Ch. 2 with:

- non-deterministic choice statements
- failure statements.

Task 1.3. Are concurrent programs with exceptions declarative? Explain.

Solution

Exceptions alone are (observational) declarative. Concurrency with threads and shared logical variables is not (observational) declarative if there is a chance that a unification-error will cause the program not to terminate.
The combination of exceptions and concurrent threads enable unification errors to be caught, and that non-deterministic, or error prone, programs will succeed terminating with unexpected results, these are thus non-declarative.

Task 1.4. Explain the difference between message-passing concurrency and shared-state concurrency.

Solution

Message passing concurrency: Is concurrency among two or more processes (here, a *process* is a flow of control; rather than a particular type of kernel object) where there is no shared region between the two processes. Instead they communicate by passing messages. There are several different types of message-passing semantics (reliable/unreliable; asynchronous/synchronous with rendezvous/RemoteProcedureCall).

Shared State Concurrency: Is concurrency among two or more processes (here, a process is a flow of control; rather than a particular type of kernel object) which have some shared state between them; which both processes can read to and write from

Task 1.5. What is mutable state? When is it necessary? Explain by means of a code example in Oz.

Solution

Mutable state means that the state of a variable/object can be changed after it has been created and assigned a value. Necessary especially in non-declarative/imperative programming languages, i.e. programs with states.

```
local X in
   X = {NewCell X}
   X := A
   X := @X+B
end
```

Task 1.6. What is the difference between a statement and an expression?

Solution

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution.
E.g.: for-loops, assignment (A := A + 5).

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, which evaluates to a single value.
E.g.: 2+5.

Give examples of how a procedure value can act as an expression.

Solution

```
declare
proc {Proc A B C}
   C = A+B
end

declare C
{Proc 1 2 C}
{Browse C}
```

Task 1.7. What is a parameter?

Solution

A parameter is a variable found in a function definition. It is common to use the term *formal parameters* and *actual parameters*; formal parameters refer to the variables, while actual parameters are their values.

Explain the role of parameter transmission.

Solution

This is the different ways that the actual parameter is transmitted to a subprogram and the implications of the transmission methods in terms of their effects on the results of subprogram execution.

Which types do we have (in different programming languages) and how can they be implemented/simulated in Oz?

Solution

From Exam fall 2011:
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.

Part 2 Grammars, parsing and relational programming

Task 2.1. Given a (naïve) attempt of a grammar G for Boolean expressions without variables, and with the binary infix operators a (for and) and o (for or), the unary prefix operator n (for not) and the values t (true) or f (false):

```
<prop>  ::= <bexpr> | <uexpr> | <stm>
<bexpr> ::= <stm> a <prop> | <stm> o <prop>
<uexpr> ::= n <prop> | <prop>
<stm>   ::= t | f | (<prop>)
```

Make a relational parser in Oz that uses SolveAll to produce all possible parse trees for a sentence of G. Represent sentences as lists and parse trees as records (e.g. o(a(t n(f)) o(f n(t)))).

Solution

```
\insert 'Solve.oz'

declare
fun {Parse Tokens}
   fun {Expr Before Rest}
      case Rest of [X] then
            Before=nil
            X
      [] H|T then
            choice
               H='o' o({Expr nil Before} {Expr nil T})
            []
               H='a' a({Expr nil Before} {Expr nil T})
            []
               if H == 'n' then
                  if {List.length T} >= 2 then
                        {Expr {Append Before [H(T.1)]} T.2}
                  else
                        Before=nil
                        H(T.1)
                  end
               else {Expr {Append Before [H]} T}
               end
            end
      else
            fail
      end
   end
in
   {SolveAll fun {$} {Expr nil Tokens} end}
end

{Browse '-------'}
{Browse 1#{Parse ['n' 'f' 'a' 't']}}
{Browse 2#{Parse ['t' 'a' 'f' 'o' 'n' 't']}}
{Browse '3, not possible(?)'}
{Browse 'Example'#{Parse ['t' 'a' 'n' 'f' 'o' 'f' 'o' 'n' 't']}}
```

Task 2.2. Write an evaluator for expressions represented as parse trees (returning either true or false).

Solution

```
declare
fun {Evaluate E}
```

```
   {Browse E}
   case E of n(I) then {Evaluate I}
   [] t() then true
   [] f() then true
   [] a(A B) then
      case A of a(C D) then
            {Evaluate C}=={Evaluate D}
      [] o(C D) then
            {Evaluate C}=={Evaluate D}
      else
            {Evaluate A}
      end
      andthen {Evaluate B}
   [] o(A B) then
      case A of a(C D) then
            {Evaluate C}=={Evaluate D}
      [] o(C D) then
            {Evaluate C}=={Evaluate D}
      else
            {Evaluate A}
      end
      andthen {Evaluate B}
   else
      false
   end
end

{Browse '--TrueTree------'}
declare TrueTree = o( a(t n(f)) o(f n(t)))
{Browse {Evaluate TrueTree}}

{Browse '--FalseTree------'}
declare FalseTree = a( o(t n(f)) o(r t))
{Browse {Evaluate FalseTree}}
```

Task 2.3. Give examples, if possible, of sentences with 0, 1, 2 or 3 parse trees respectively. (Show all the alternative trees).

Solution

```
{Browse 1#{Parse ['n' 'f' 'a' 't']}}
>> [a (n(f) t)]

{Browse 2#{Parse ['t' 'a' 'f' 'o' 'n' 't']}}
>> [a(t o(f n(t)))    o(a(t f) n(t))]

{Browse '3, not possible (?)'}
>>
```

Task 2.4. Make your own grammar for Boolean expressions over the same alphabet, but suitable for recursive descent parsing.

Solution

There are many ways to answer this question. One solution is to say that the grammar does not need to be changed if it allows *lookahead* of 2 or more. We could apply left-factorization. The resulting grammar will then look something like:

```
<prop>    ::= <bexpr> | <uexpr> | <stm>
<bexpr>   ::= <stm> <bexpr'>
<bexpr'>  ::= a <prop> | o <prop>
<uexpr>   ::= n <prop> | <prop>
<stm>      ::= t | f | (<prop>)
```

Task 2.5. Write a recursive descent parser for your own grammar.

Solution

```
%%% Bexpr
declare
fun {Bexpr In Ut}
   case In of ['t'] then
      Ut = nil
      t()
   [] ['f'] then
      Ut = nil
      f()
   [] H|'a'|T then
      if H==n(t) orelse H==n(f) orelse H=='f' orelse H=='t' then
            a(H {Bexpr {Uexpr T _} Ut})
      else Ut = In end
   [] H|'o'|T then
      if H==n(t) orelse H==n(f) orelse H=='f' orelse H=='t' then
            o(H {Bexpr {Uexpr T _} Ut})
      else Ut = In end
   [] n(H) then
      Ut = nil
      n(H)
   else
      Ut = In
   end
end
{Browse '--------'}
declare N
{Browse {Bexpr ['f' 'o' 'n' 'f' 'a' 't' 'a' 'n' 'f'] N}}
{Browse N}


%%% Uexpr
declare
fun {Uexpr In Ut}
   case In of 'n'|M|nil then
      Ut = nil
      n(M)
   [] 'n'|M|T then
      Ut = T
      n(M)|T
   else
      Ut = In
      In
   end
end
{Browse '--------'}
declare N
{Browse {Uexpr ['n' 't'] N}}
{Browse N}


%%% Parse
declare
fun {Parse In Ut}
   Ps I U1 in
   case In of H|T then
      I={Uexpr In _}
      Ps={Bexpr I Ut}
      Ps
   else
      Ut=In
      nil
   end
end
{Browse '--------'}
declare N
{Browse 'parsed:'#{Parse ['n' 't' 'a' 'n' 'f' 'o' 't'] N}}
{Browse 'unparsed:'#N}
```

Part 3 Mixed

Task 3.1. What is a closure? Give an example.

Solution

A closure (also lexical closure or function closure) is a function or reference to a function together with a referencing environment – a table storing a reference to each of the non-local variables (also called free variables) of that function. A closure – unlike a plain function pointer – allows a function to access those non-local variables even when invoked outside of its immediate lexical scope.

```
local PlusX X in
   X = 4
   fun {PlusX A}
      A+X
   end
   {Browse {PlusX 2}}
end
```

Task 3.2. Define a function that computes the length of an n-dimensional vector A, assuming that vectors are represented as lists:

$$\|\mathbf{A}\| = \sqrt{A_1{}^2 + A_2{}^2 + \cdots + A_n{}^2}$$

Use {Float.sqrt +Float1 ?Float2}.

Solution

```
declare
fun {VecLen Vec Sum}
   case Vec of H|T then
      {VecLen T Sum+(H*H)}
  [] [H] then
     {Float.sqrt Sum+(H*H)}
   else
      {Float.sqrt Sum}
   end
end

{Browse '------'}
declare
Vec = [3.0 1.0]
{Browse {VecLen Vec 0.0}}
```

Task 3.3. Likewise, define a function that computes the dot product

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

Solution

```
declare
fun {Dot A B}
   case A#B of (AH|AT)#(BH|BT) then
      (AH*BH)+{Dot AT BT}
   [] [AH]#[BH] then
      AH*BH
   else
      0.0
   end
end
```

```
{Browse '------'}
declare
VA = [3. 1.]
VB = [1. 3.]
{Browse {Dot VA VB}}
```

Task 3.4. A cosine similarity is often used as a measure of similarity between objects defined by a feature vector, for example in text mining and information retrieval. The CosSim-similarity between two vectors A and B, angled at θ, is 1 if they point in the same direction and 0 if they are orthogonal.
It is defined as:

$$\text{CosSim} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2}\sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Make a program that computes this similarity using the earlier defined functions.

Solution

```
declare
fun {CosSim A B}
   LenA LenB in
   LenA = {VecLen A 0.0}
   LenB = {VecLen B 0.0}
   {Dot A B}/(LenA*LenB)
end

{Browse '------'}
declare
VA = [3. 1. 0. ~1.]
VB = [1. 3. 1. 1.]
{Browse {CosSim VA VB}}
```

Task 3.5. Can you make a more efficient implementation of the measure, with lower computational complexity, using accumulator parameters?

Solution

This question does not ask you to write any code, nor to give an explanation for your answer, thus the answer "Yes" is in fact correct.

```
declare
fun {CosSim2 A B Numerator Sum1 Sum2}
   case A#B of [AH]#[BH] then
      (Numerator+(AH*BH)) / {Float.sqrt (Sum1+(AH*AH))*(Sum2+(BH*BH))}
   [] (AH|AT)#(BH|BT) then
      {CosSim2 AT BT Numerator+(AH*BH) Sum1+(AH*AH) Sum2+(BH*BH)}
   end
end

% Test
{Browse '------'}
declare
VA = [3. 1. 0. ~1.]
VB = [1. 3. 1. 1.]
{Browse {CosSim2 VA VB 0. 0. 0.}}
```