Front page

NTNU

Faculty of Information Technology and Electrical Engineering

Department of Computer Science

Examination paper for TDT4165 Programming Languages

This document is generated and mangled by Inspera. The GRADING AID parts are inserted manually. The electronic representation and embedded grading aid in Inspera is authoritative in case of discrepancies. Inspera cannot do basic arithmetic, so some tasks count maximally 9.96 instead of 10.

Examination date: December 13, 2021

Examination time (from-to): 09.00 – 12.00

Permitted examination support material: Code E: No support material allowed

Academic contact during examination: Øystein Nytrø   Phone: 91 89 76 06

OTHER INFORMATION

This course has only an english version of the exam.

This examination has 15 tasks. All tasks have the same weight. Wrong answers are not scored negatively. There is an ungraded text-entry at the end of the exam that you can use for comments

Get an overview of the question set before you start answering the questions.

Read the questions carefully and make your own assumptions. If a question is unclear/vague, make your own assumptions and specify them in your answer. Only contact academic contact in case of errors or insufficiencies in the question set. Address an invigilator if you wish to contact the academic contact. Write down the question in advance.

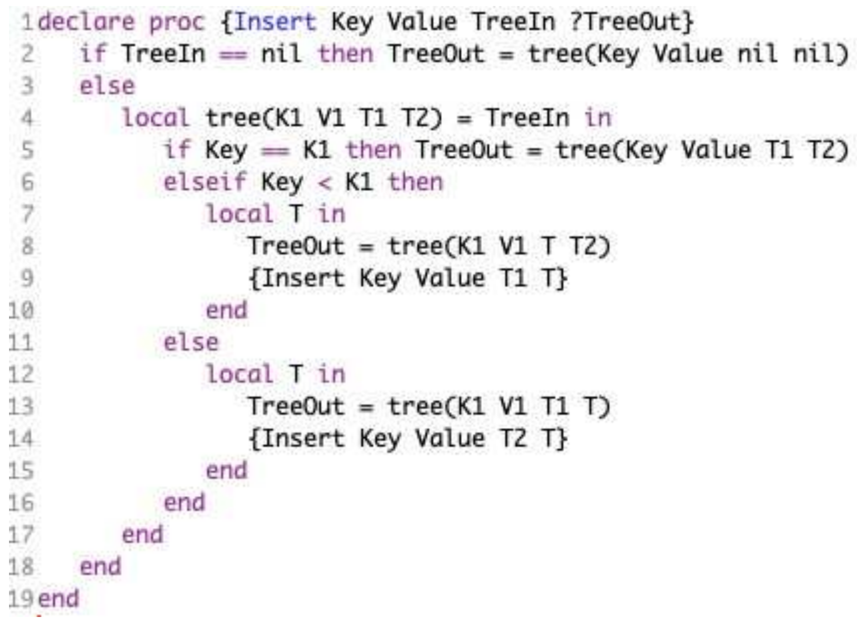
Notifications: If there is a need to send a message to the candidates during the exam (e.g. if there is an error in the question set), this will be done by sending a notification in Inspera. A dialogue box will appear. You can re-read the notification by clicking the bell icon in the top right-hand corner of the screen.

Withdrawing from the exam: If you become ill or wish to submit a blank test/withdraw from the exam for another reason, go to the menu in the top right-hand corner and click “Submit blank”. This cannot be undone, even if the test is still open.

Access to your answers: After the exam, you can find your answers in the archive in Inspera. Be aware that it may take a working day until any hand-written material is available in the archive.

1 Pattern matching

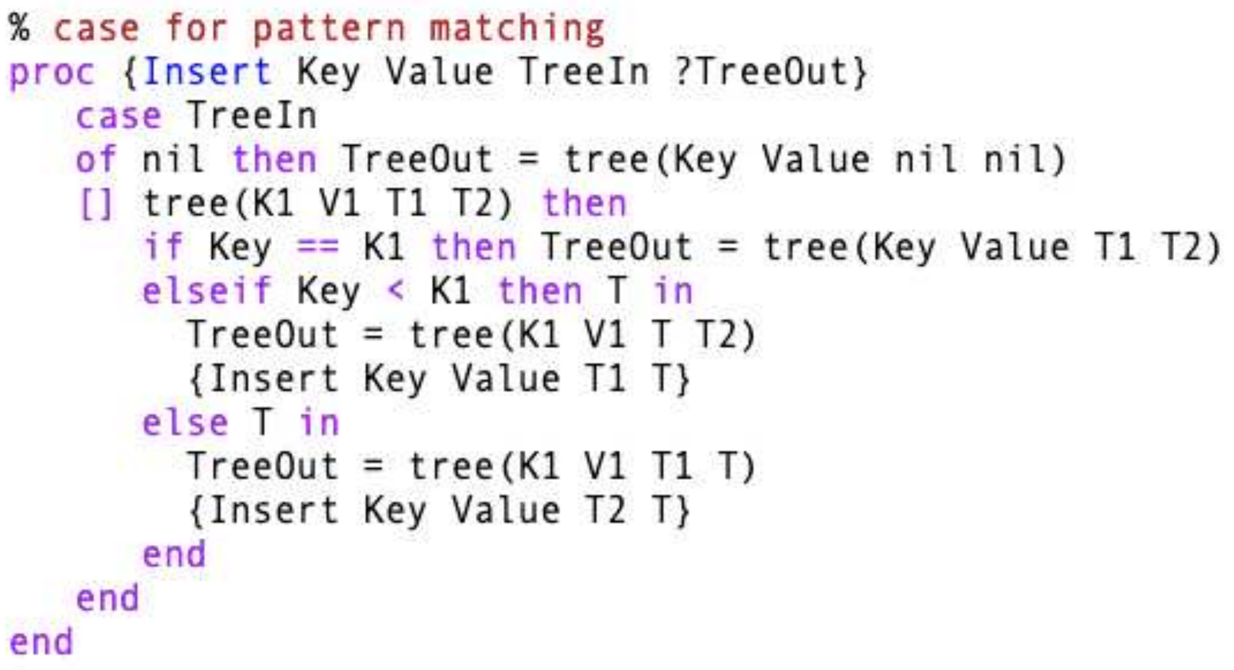
The program shown employs pattern matching on records. Simplify by using pattern-matching in case-constructs.



```
declare proc {Insert Key Value TreeIn ?TreeOut}
   if TreeIn == nil then TreeOut = tree(Key Value nil nil)
   else
      local tree(K1 V1 T1 T2) = TreeIn in
         if Key == K1 then TreeOut = tree(Key Value T1 T2)
         elseif Key < K1 then
            local T in
               TreeOut = tree(K1 V1 T T2)
               {Insert Key Value T1 T}
            end
         else
            local T in
               TreeOut = tree(K1 V1 T1 T)
               {Insert Key Value T2 T}
            end
         end
      end
   end
end
```

Grading aid:

Correct code is shown, so the main point is to show proper use of at least one case sentence utilizing pattern matching and implicit declaration of local variables in the pattern. See Mozart tutorial: http://mozart2.org/mozart-v1/doc-1.4.0/tutorial/node5.html#label34. A possible solution:



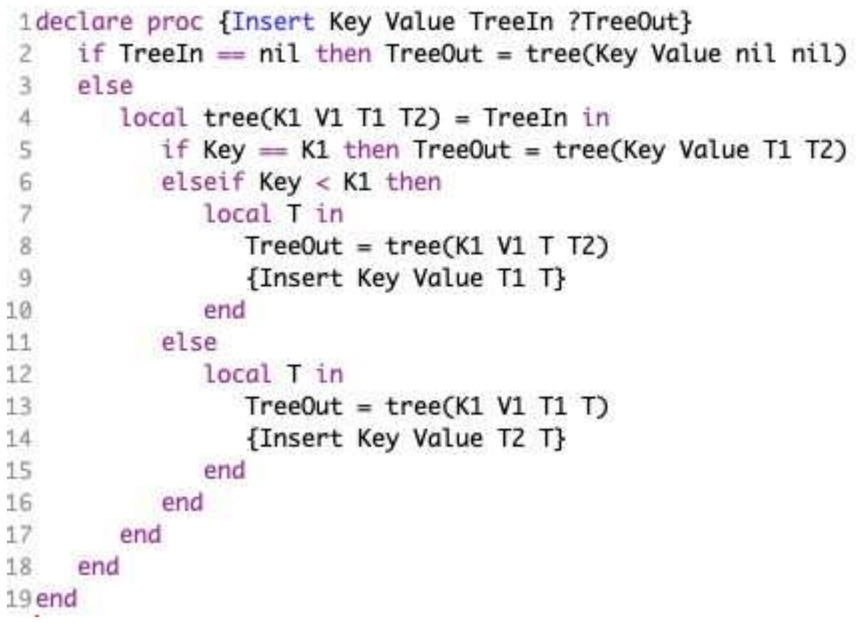
```
% case for pattern matching
proc {Insert Key Value TreeIn ?TreeOut}
   case TreeIn
   of nil then TreeOut = tree(Key Value nil nil)
   [] tree(K1 V1 T1 T2) then
      if Key == K1 then TreeOut = tree(Key Value T1 T2)
      elseif Key < K1 then T in
        TreeOut = tree(K1 V1 T T2)
        {Insert Key Value T1 T}
      else T in
        TreeOut = tree(K1 V1 T1 T)
        {Insert Key Value T2 T}
      end
   end
end
```

Maximum marks: 10

2 Record reasoning

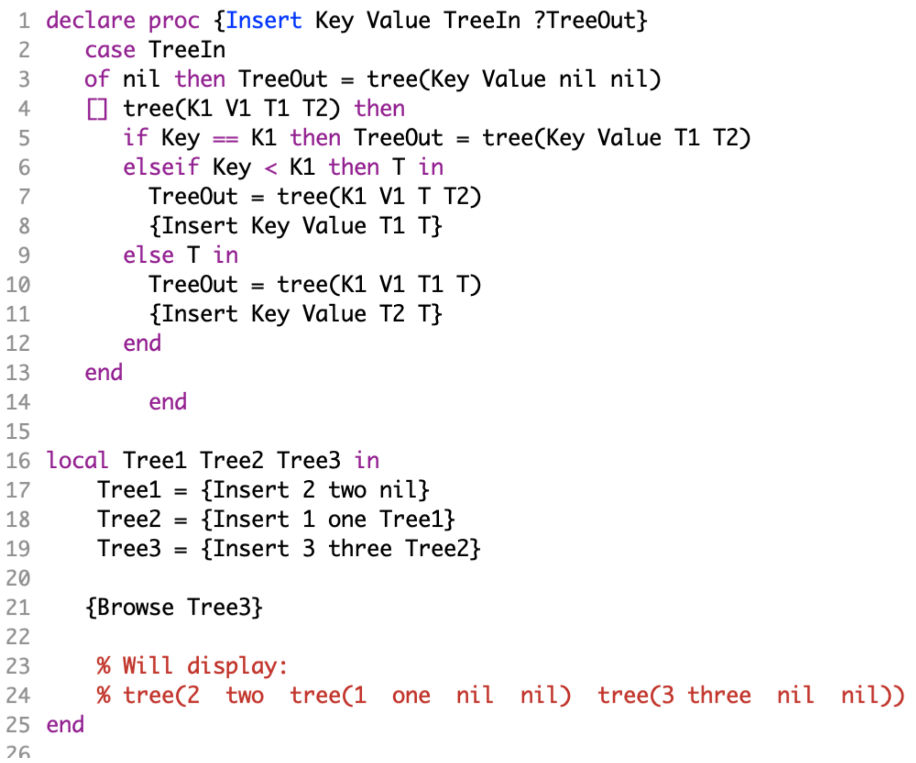
The program shown will insert values in a binary, key-sorted tree structure.
Write a program that

1. uses the procedure Insert to make a balanced tree with the keys 1, 2 and 3 and corresponding values one, two and three
2. shows the resulting TreeOut with a Browse-statement.
3. Include the value shown by the Browse-statment in a comment!



```
declare proc {Insert Key Value TreeIn ?TreeOut}
   if TreeIn == nil then TreeOut = tree(Key Value nil nil)
   else
      local tree(K1 V1 T1 T2) = TreeIn in
         if Key == K1 then TreeOut = tree(Key Value T1 T2)
         elseif Key < K1 then
            local T in
               TreeOut = tree(K1 V1 T T2)
               {Insert Key Value T1 T}
            end
         else
            local T in
               TreeOut = tree(K1 V1 T1 T)
               {Insert Key Value T2 T}
            end
         end
      end
   end
end
```

Grading aid:
All three tasks must be completed. A possible solution:



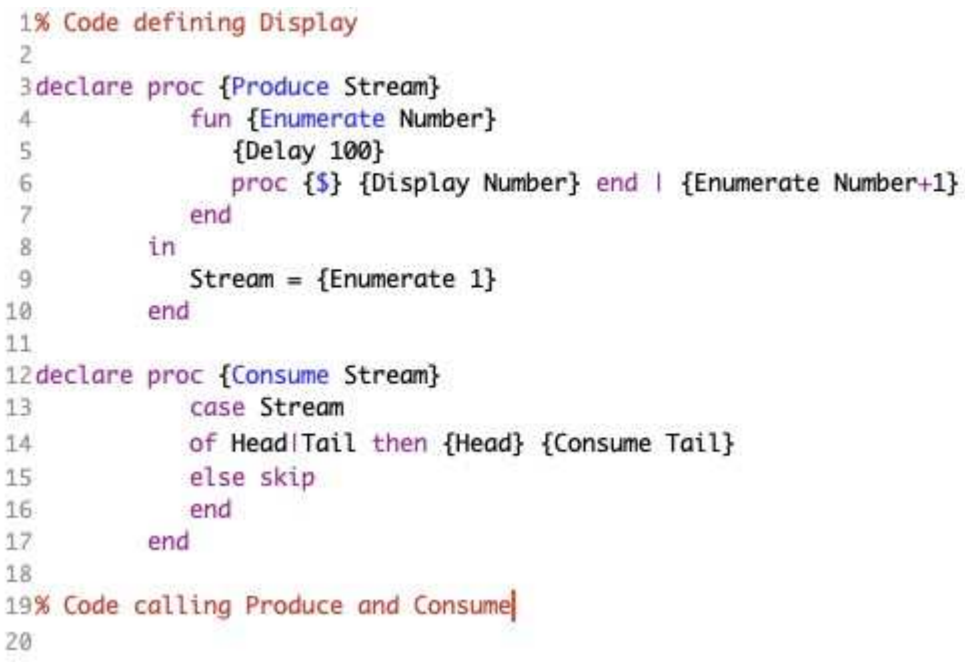
```
declare proc {Insert Key Value TreeIn ?TreeOut}
   case TreeIn
   of nil then TreeOut = tree(Key Value nil nil)
   [] tree(K1 V1 T1 T2) then
      if Key == K1 then TreeOut = tree(Key Value T1 T2)
      elseif Key < K1 then T in
        TreeOut = tree(K1 V1 T T2)
        {Insert Key Value T1 T}
      else T in
        TreeOut = tree(K1 V1 T1 T)
        {Insert Key Value T2 T}
      end
   end
        end

local Tree1 Tree2 Tree3 in
    Tree1 = {Insert 2 two nil}
    Tree2 = {Insert 1 one Tree1}
    Tree3 = {Insert 3 three Tree2}

   {Browse Tree3}

    % Will display:
    % tree(2  two  tree(1  one  nil  nil)  tree(3 three  nil  nil))
end
```

ta

Maximum marks: 10

3 Thread sequencing MC

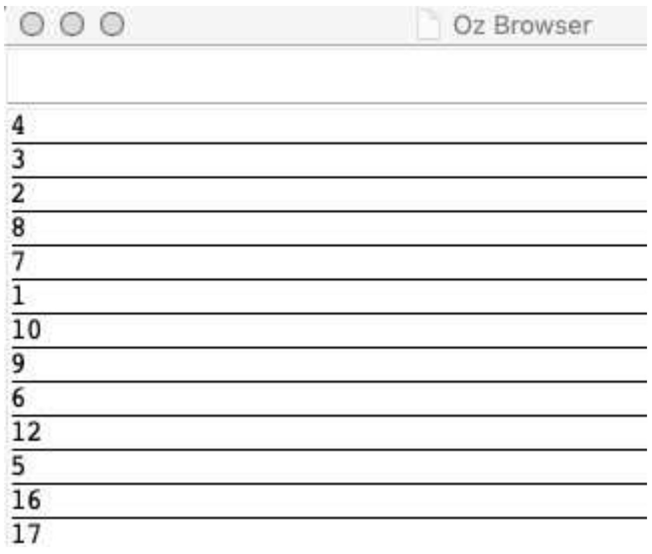
Given the following code



```
% Code defining Display

declare proc {Produce Stream}
            fun {Enumerate Number}
               {Delay 100}
               proc {$} {Display Number} end | {Enumerate Number+1}
            end
         in
            Stream = {Enumerate 1}
         end

declare proc {Consume Stream}
            case Stream
            of Head|Tail then {Head} {Consume Tail}
            else skip
            end
         end

% Code calling Produce and Consume
```

{Delay N} freezes the thread for N milliseconds. {Delay {OS.rand} mod 1000} freezes the thread a random number of milliseconds between 0 and 1000.

Which of the alternative code snippets defining Display and calling Produce and Consume shown in the alternatives could produce an output like below:



```
4
3
2
8
7
1
10
9
6
12
5
16
17
```

and so on...

Alternative code snippets:

1.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  {Produce Numbers}
  {Consume Numbers}
end
```

2.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
end
```

3.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
  thread {Consume Numbers} end
end
```

4.
```
proc {Display Number}
   thread {Delay {OS.rand} mod 1000} {Browse Number} end
end
local Numbers in
  {Produce Numbers}
  {Consume Numbers}
end
```

5.
```
proc {Display Number}
   thread {Delay {OS.rand} mod 1000} {Browse Number} end
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
end
```

Select one alternative

Snippet 4

Snippet 2

Snippet 5

Snippet 1

Snippet 3

Maximum marks: 10

Grading comments:

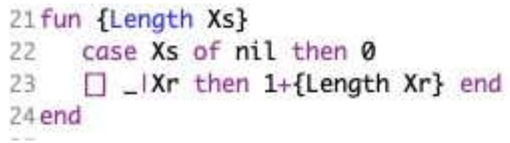
Only alternative 5 may give a random sequence.

The alternatives are:

1. No output ({Consume Numbers} is unreachable).
2. The sequence 1, 2, 3, . . . (guaranteed by unthreaded Display).
3. A sequence such as 1, 2, 1, 3, . . . , where each number occurs twice and a number n output by one of the consumers can never precede any number m < n output by the same consumer.
4. No output ({Consume Numbers} is unreachable).
5. A more or less random sequence, with each number occurring once, and with larger numbers more likely to appear later in the sequence.

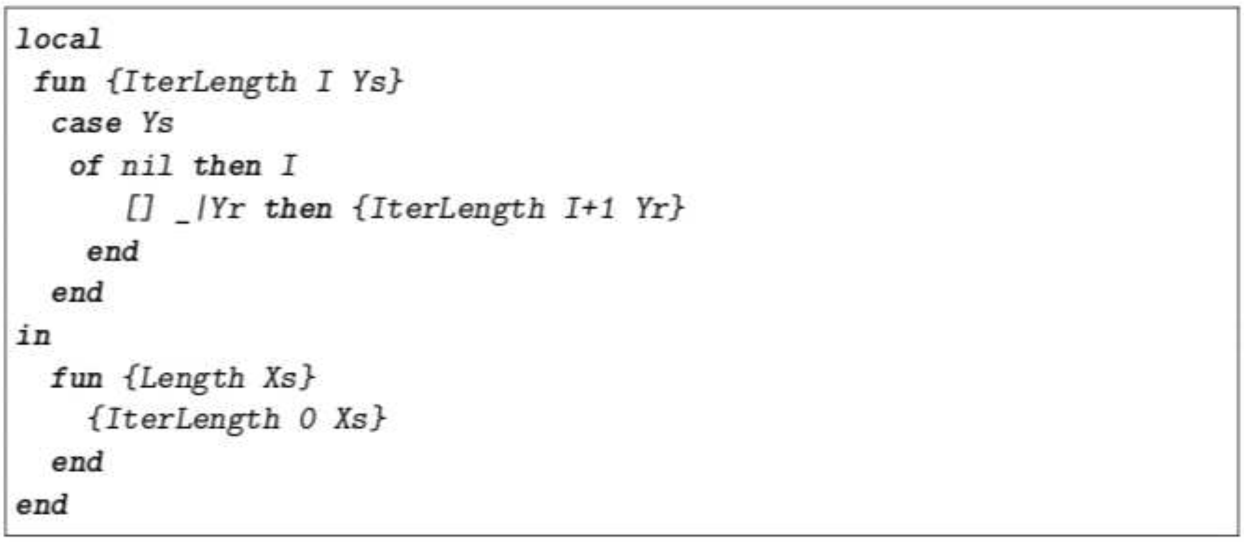
4 Iterative Length

Implement an iterative version of Length in Oz:



```
fun {Length Xs}
   case Xs of nil then 0
   [] _|Xr then 1+{Length Xr} end
end
```

Grading aid (CTMCP 3.4.2.4)

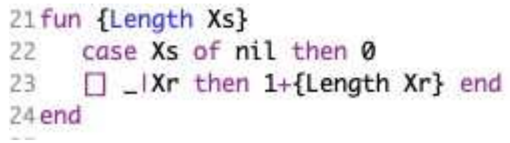


```
local
 fun {IterLength I Ys}
  case Ys
   of nil then I
      [] _|Yr then {IterLength I+1 Yr}
    end
  end
in
  fun {Length Xs}
    {IterLength 0 Xs}
  end
end
```

Maximum marks: 10

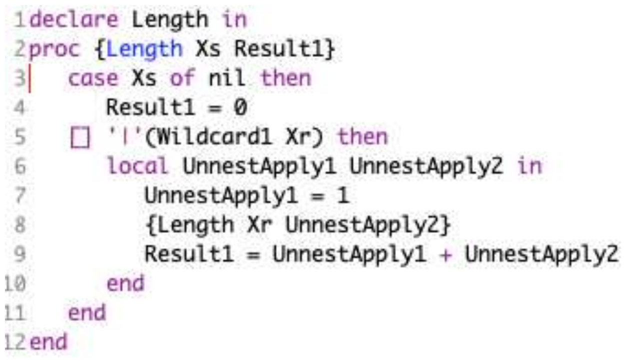
5 Fun to Proc

In the core language, functions are replaced by procedures. Write the equivalent procedure for the recursive Length function:



```
fun {Length Xs}
   case Xs of nil then 0
   [] _|Xr then 1+{Length Xr} end
end
```

Grading aid
It was not required to convert into core language, but below is the result of Oz doing that:

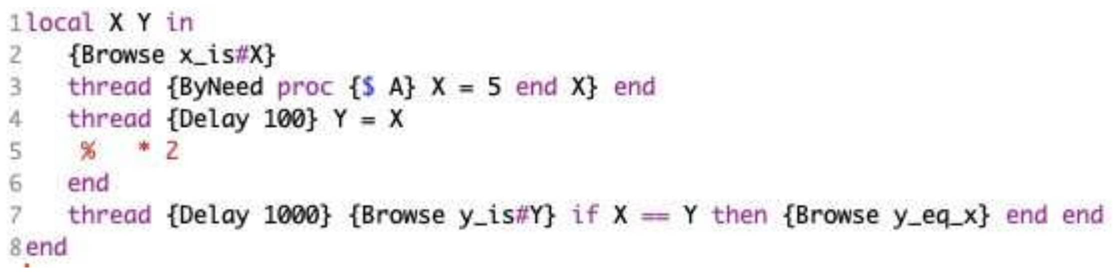


```
declare Length in
proc {Length Xs Result1}
   case Xs of nil then
      Result1 = 0
   [] '|'(Wildcard1 Xr) then
      local UnnestApply1 UnnestApply2 in
         UnnestApply1 = 1
         {Length Xr UnnestApply2}
         Result1 = UnnestApply1 + UnnestApply2
      end
   end
end
```

Maximum marks: 10

6 ByNeed and TreadsMC

Consider the following code snippet in Oz. The interpreter will produce specific output with delay between different degree of instantiation of unbound variables ("_").



```
local X Y in
   {Browse x_is#X}
   thread {ByNeed proc {$ A} X = 5 end X} end
   thread {Delay 100} Y = X
    %   * 2
   end
   thread {Delay 1000} {Browse y_is#Y} if X == Y then {Browse y_eq_x} end end
end
```

Based on the specific sequence of output, which of the statements below would be the most correct?

Select one alternative:

Exchanging 'y_eq_x' for '1+X' (note capital X) would not change the first occurring output.

The output indicates that at least one of the explicit three threads is still suspended after the longest delay.

The first occurring output does not change because it is not needed.

Changing the delay would make the computation non-deterministic.

The first occurring output does not change because it is subject to the first occurring unification.

If you remove the comment "%" before "* 2", making the second statement in the second thread become "Y = X * 2", what is the most correct statement:

Select one alternative

None of the other solutions

'x_is#_' and 'x_eq_y' is printed immediately.

'x_is#5' and 'y_is#10' is printed eventually.

'x_is#_' and 'y_is#10' is printed and remains so after all thread have terminated.

'x_is#_' and 'y_is#10' is printed immediately.

Maximum marks: 10

7 Oz and grammar comprehension

Consider the following simplified and incomplete grammar for a fragment of Oz:

```
1.    <s> ::= skip
2.    | <s> <s>
3.    | local <x> in <s> end
4.    | case <x> of <...> then <...> { [] <.. >} end
5.    | <x> = <x>
6.    | if <x> then <s> else <s> end
```

<s> is a statement, and is also the start symbol.

<x> is an identifier

<...> is left unspecified

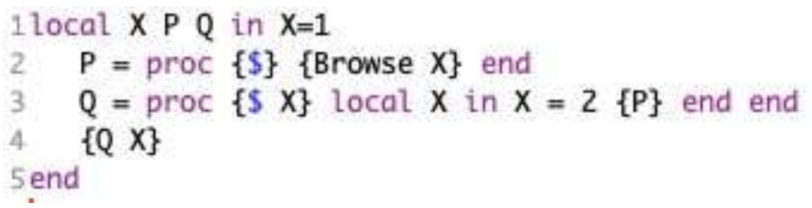
Please match statements with line numbers.

	4	2	3	6	5	1
This clause represents a statment that always reduces the size of the semantic stack						
This clause makes the grammar ambiguious						
This suspendable statement may directly change the environment of the next semantic stack content.						
This clause allows statements to be interpreted as expressions						
This clause represents a non-suspendable statement						
This clause represents a statement that only affects the single assignment store						

Maximum marks: 9.96

8 Execution stack state

Consider the following Oz program:



```
local X P Q in X=1
  P = proc {$} {Browse X} end
  Q = proc {$ X} local X in X = 2 {P} end end
  {Q X}
end
```

Show the state of the abstract machine during an execution of the program just before the statement {Q X} is executed. (Ignore the identifier Browse when you show the content of environments).

Grading aid:

The state of the abstract machine during an execution of the above program (translated into the kernel language) just before the statement `{Q X}` is executed is as follows:

$$([(\texttt{\{Q X\}}, \{Q \rightarrow v_3, X \rightarrow v_1\})],$$
$$\{ v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{X \rightarrow v_1\}),$$
$$v_3 = (\texttt{proc \{\$ X\} local X in X = 2 \{P\} end end}, \{P \rightarrow v_2\}) \})$$

where v_i are variables in the single assignment store.

Maximum marks: 10

9 Thread join

The following is a fragment of Scala code:

object ThreadStuff extends App {
val t = thread { log(“one”) }
log(“two”)   log(“three”)
t.join()   log(“four”)
}

What is correct?

Select one alternative:

The sequence cannot be predicted

None of the other answers

The log will show the sequence "one two three four"

The log will show the sequence "two three one four"

The log will always show "one" before "four"

Maximum marks: 10

10 Race

A race condition in a concurrent program is:
Select one alternative:

The situation that threads are deadlocked

The situation that the effect, or output, of concurrent computation depends on execution scheduling

The situation(s) immediately before a deadlock occurs.

The situation in which different threads depletes each others resources

A conditional test that a thread is not waiting

Maximum marks: 10

11 ExecutionContext

The abstract “execute” method of Scala’s ExecutionContext trait
Select one alternative:

has a default global execution context

makes concurrency deterministic

does not reduce memory usage to achieve concurrency

takes a runnable and executes it immediately

None of the other answers

Maximum marks: 10

12 Identifier scopes in Oz etc.

Which completion of "Oz has..." is true?

Select one alternative:

no scope rules.

static typing.

no typing.

dynamic scoping.

static scoping.

Maximum marks: 10

13 Church-multiplication

In Learn Prolog Now, so-called Church encoding of numerals is introduced with the following definition of predicates:

```
numeral(0).                          % 0 is a numeral
numeral(succ(X)) :- numeral(X).    % if X is a numeral, so is succ(X).
% succ(0) represents 1, succ(succ(0)) == 2, and so forth
```

```
add(0,Y,Y). add(succ(X),Y,succ(Z)) :-
add(X,Y,Z).
```

Define the Prolog predicate mult(A,B,R) that defines muliplication of numerals with the above representation, so that R = A * B.

Grading aid:

```
mult(0,_,0).
mult(succ(X),Y,R) :-    mult(X,Y,S),    add(S,Y,R).
```

Maximum marks: 10

14 Query comprehension

In Learn Prolog Now, so-called Church encoding of numerals is introduced with the following definition of predicates:

```
numeral(0).                          % 0 is a numeral
numeral(succ(X)) :- numeral(X).   % if X is a numeral, so is succ(X).
% succ(0) represents 1, succ(succ(0)) == 2, and so forth
```

```
add(0,Y,Y). add(succ(X),Y,succ(Z)) :-
add(X,Y,Z).
```

Match queries in Prolog to the right answers:
Please match the values:

	R = succ(succ(0))	false	R = succ(0)	R = 0
?- numeral(succ(succ(1))).				
?- add(0,succ(0),R)				
?- add(succ(R),succ(R),succ(succ(_))).				
?- add(R,succ(succ(0)),succ(succ(succ(succ(0))))).				
?- add(succ(R),X,succ(succ(R))).				

Maximum marks: 10

15 Explain CLPFD

Given the following interaction with a Prolog interpreter, match the line numbers to the right answers

```
1    :- use_module(library(clpfd)).

2    test(X, Y) :-
3    X in 0..10,
4    Y #> 4,
5    X #> Y.

6  ?- test(X, Y).

7  X in 6..10,
8  Y#=<X+-1,
9  Y in 5..9
```

Match explanation to the line numbers

	1	4	6	5	3	9	8	72
A directive to the interpreter								
X in the answer is constrained to an interval.								
The head of a defined clause (rule)								
A query to the interpreter.								
Two variables in the answer are constrained by a relationship.								
The variable must be constrained to bounded interval								

Maximum marks: 9.96

16 Comment

If you have additional comments or remarks to any of the questions, this is the place to write them!

Please refer to question number if relevant:

Maximum marks: 0