UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Exam in:	**INF3110 Programming Languages**
Day of exam:	**December 12, 2016**
Exam hours:	**14:30 – 18:30**
This examination paper consists of 8 pages.	
Appendices:	**No**
Permitted materials:	**All printed and written, including the textbook**

Make sure that your copy of this examination paper is complete before answering.

This exam consists of 3 questions that may be answered independently. If you think the text of the questions is unclear, make your own assumptions/interpretations, but be sure to write these down as part of the answers.

Good luck!

Question 1. Runtime-systems, scoping, types (weight 40%)

Consider the following EBNF definition of a small language:

```
program           ::= <decl>+

decl              ::= <varDecl> | <procDecl>
varDecl           ::= "var" <name> ":" <typename> "=" <expr> ";"
procDecl          ::= "procedure" <name> "(" <formalParamList>? ")"
                        "begin" <blockBody> "end"
formalParamList   ::= <formalParameter> ( "," <formalParameter> )*
formalParameter   ::= <name> ":" <typename>
blockBody         ::= <varDecl>* <statement>*

statement         ::= <printStatement>
                      | <callStatement>
                      | <assignStatment>
                      | <ifStatement>

printStatement    ::= "print" <expr> ";"
callStatement     ::= "call" <name> "(" <actualParamList>  ")" ";"
assignStatment    ::= <name> ":=" <expr> ";"
ifStatement       ::= "if" <expr> "begin" <blockBody> "end"
actualParamList   ::= <expr> ( "," <expr> )*

expr              ::= <name> | <integer> | "true" | "false"
```

In the productions above, nonterminals are enclosed within angled brackets (< and >), and terminals are quoted. Plus (+) is used for one or more repetitions, asterisk (*) for zero or more, and question mark (?) for zero or one. Regular parentheses are used for grouping symbols in the grammar. The nonterminals *name*, *typename* and *integer* are not specified in any further detail, but it will suffice to say that *name* and *typename* are strings of letters and numbers, and *integer* denotes “ordinary” integer numbers.

1a)

Below is a small program in the language defined by the grammar above:

```
var x : int = 0;
var y : int = 1;

procedure M(x : int)
begin
    var y : int = 42;
    x := 1;
    call N(x);
end

procedure N(x : int)
begin
    var z : int = 2;
    if true begin
        var z : int = 42;
    end

    print x;
    print y;
    print z;
end

procedure Main()
begin
    call M(x);
end
```

Explain what this program does; describe what will be printed, and why. Is there more than one reasonable interpretation? Explain your reasoning. Assume that the program execution starts with the execution of the procedure `Main()`, and that `print <expr>` will print the value of the expression to the console/terminal. Furthermore, assume that the program is legal and valid (it has no errors) in the language in question.

1b)

Draw an abstract syntax tree for the procedure `M` from 1a).

1c)

Draw a runtime stack for the program from 1a) with activation records containing variables, parameters, access links and control links when the program has just executed the `print z` statement. Explain briefly the assumptions you make about the semantics of the language.

1d)

Extend the grammar provided at the top of this exercise to allow for procedures within procedures. E.g., the following should be allowed, where we have introduced new procedures `P` and `Q`:

```
procedure N(x : int)
begin
    var z : int = 2;

    procedure Q(v : int)
    begin
        print v;
    end

    if true begin
        var z : int = 42;
        procedure P() begin print z; end
        call P();
    end

    print x;
    print y;
    print z;

    call Q(y);
end
```

You only need to write out the productions that you modify.

1e)

Assume now, that in addition to procedures within procedures, we also allow procedures as parameters to other procedures. Explain briefly how this can be implemented in the runtime system.

1f)

Modify the procedure named `N` from 1d) so that this procedure now has a new formal parameter `FP` that is a procedure, and so that `N` performs a recursive call to itself with the procedure `P` as the actual parameter. Write down the new program, and the runtime stack right after the first recursive call to `N`, so that the activation block for this call is on the stack. (As in 1a, we assume that program execution starts with the `Main()` procedure.)

You only need to write down the parts of the program you modify. You do not need to write any modified version of the grammar of the language in this exercise, just assume a suitable syntax for this purpose. If the program ends up with infinite recursion (non-termination), that is OK.

Briefly state any further assumptions that you make about the language.

Question 2. ML (weight 40%)

2a

Evaluate the following ML expressions:

```
a) List.foldr (-) 4 [3,2,1]

b) List.foldr (fn (y,x) => 2*x + y) 4 [1,2,3]
```

2b

Assume the standard definition of `foldl`:

```
fun foldl (f: 'a*'b->'b) (acc: 'b) (l: 'a list): 'b =
  case l of
    [] => acc
  | x::xs => foldl f (f(x,acc)) xs
```

With the help of `foldl`, define the function

```
val myReverse = fn : 'a list -> 'a list
```

which returns the input list in reverse order.

2c

1) Define a function that sums up all values in a list of integers (and consider also question (4) below):

```
val sum = fn : int list -> int
```

2) Define a function that finds the largest element in a list of integers:

```
val max = fn : int list -> int
```

3) Define the function

```
val summax = fn : int list list -> int
```

which takes a list of lists of integers and sums up the largest values from each list. Example:

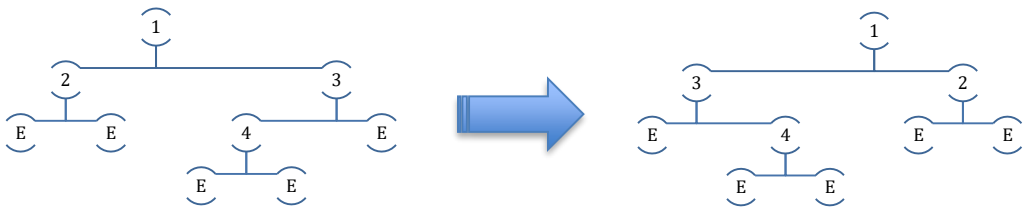
```
summax [[1,2],[4,2],[3,6,5,9]] = 15
```

4) Give function `sum` from the question (2c1) above as a tail-recursive function (if you have not done so immediately – you do not have to answer both questions separately if you directly give the tail-recursive version)! If you cannot answer question (2c1), explain the difference between a tail-recursive and a naïve implementation of a recursive function in a few sentences.

2d

1) Define a datatype for binary trees, `'a tree`, with a constructor `Empty`, and a constructor `Node` with a value of type `'a` and two subtrees.
2) Define a function `val mirror = fn : 'a tree -> 'a` tree that returns a mirrored version of the tree passed as an argument. That is, for a node, each subtree must be mirrored.

   Examples ("E" for `Empty`):



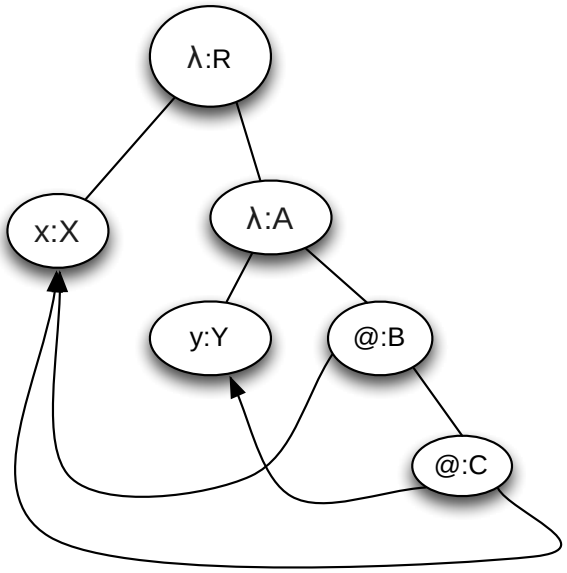
3) Define a function `val sym = fn : 'a tree -> 'a tree -> bool` which returns true, if the two trees are symmetric (that is, one is a mirror of the other), or false otherwise.

2e

Calculate the type for the following expression according to the ML type inference algorithm:

```
fn x => fn y => x (y x).
```

Use the provided type variables in the parse graph below. Derive the corresponding equations for the parse graph, and solve the resulting equation system to obtain the type of the root node R.



Question 3. Prolog (weight 20%)

3a

Give PROLOG's answer (that is, the substitutions for all variables in a query, if the terms can be unified) for each of the queries below, or write "`no`" if no solution exists **and** give the reason. Be careful with the last question, the == is no typo!

```
1. g(a,X,Y) = g(Z,c,f(Z)).

2. g(a,X,Y) = g(Y,c,f(X)).

3. f(X,Y,d) == f(Y,h(X),Z).
```

3b

Define the predicate `rotate(L,N,R)` which rotates cyclically a list L with N positions to the left, i.e. the following queries succeed:

```
rotate([a,b,c,d], 1, [b,c,d,a]).
rotate([a,b,c,d], 6, [c,d,a,b]).
rotate([a,b,c,d], 3, [d,a,b,c]).
```