# UNIVERSITETET I OSLO

## Det matematisk-naturvitenskapelige fakultet

| | |
|---|---|
| **Exam in:** | **INF3110 Programming Languages** |
| **Day of exam:** | **December 12, 2013** |
| **Exam hours:** | **14:30 – 18:30** |
| **This examination paper consists of 9 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 answer.*

Good luck!

**with answers**

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

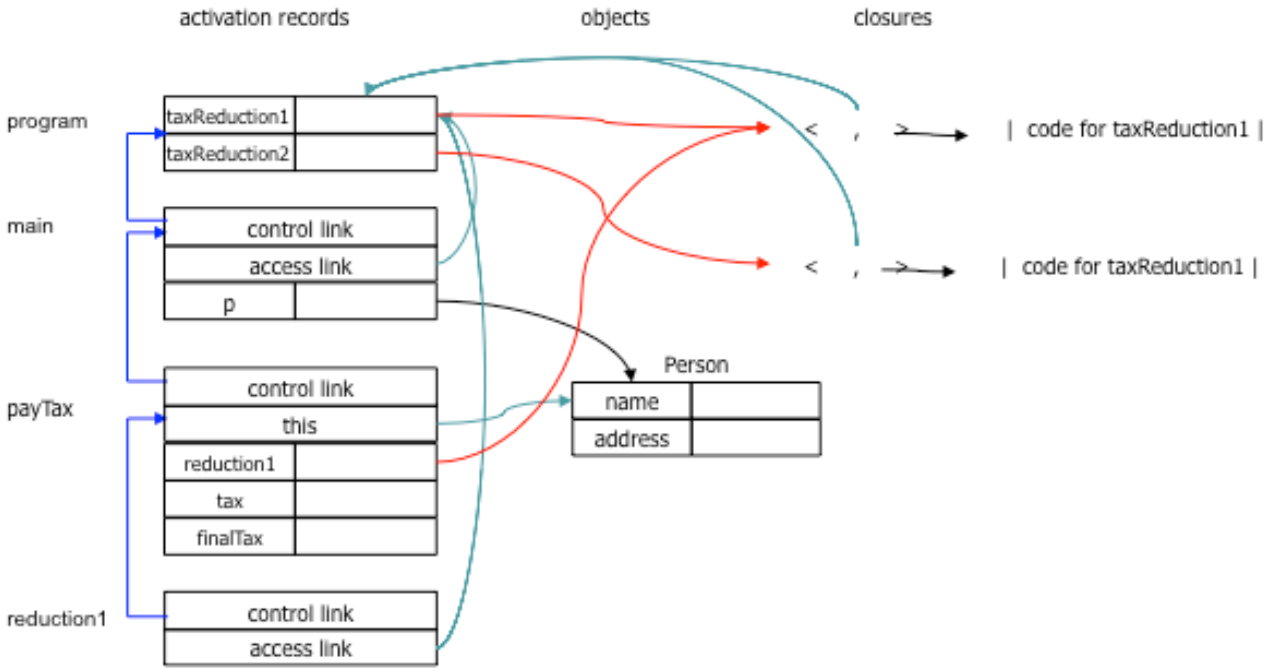
In all parts of Question 1 we assume that we have a statically scoped language with blocks (`{...}`) that may have variables, functions, interfaces and classes (with variables and methods). Functions and methods may have functions as parameters. A program has an outermost block, and all names defined in the outermost block are visible according to usual scope rules. A program is executed by first allocating an activation record for the outermost block and then executing the `main` function.

## 1a

The code below is a program in this language. It is not intended to be complete, and it includes only things that are required in order to answer this question.

```
{
  int taxReduction1(String address) {...}
  int taxReduction2(String address) {...}

  interface Addressable {
    String address();
    void printLabel();
  }
  interface Taxable {
    int income()
    void payTax(int reduction(String));
  }
  class Person implements Addressable, Taxable {
    String name;
    String address;
    String name(){return name;}
    String address(){return address;}

    void printLabel(){...}

    void payTax(int reduction(String)){
      int tax, finalTax;
      ... // compute tax
      finalTax = tax - reduction(address());
      ... // pay finalTax
    }
    public void Person(String pname) {name=pname;}
  }

  void main() {
    Person p = new Person("Birger");
    // setting the address and income of p
    p.printLabel();
    ...
    p.payTax(taxReduction1);
    ...
  }
}
```

Draw the run-time stack and the `Person` object as they are while executing the call `reduction(address())` in `payTax`. Include all links between the activation records, the variables and parameters in the activation records, and illustrate/explain how both the formal and actual parameter to `payTax` are handled. You may assume that the static link for a method activation record is the object that has the method.




## 1b

In this part of Question 1 we change the `printLabel` to have a parameter `quality`, and we add two variables to the outermost program block. Labels may be printed in two qualities (1 or 2), and there is a maximum number of prints with quality 1. In `main` we now going to have a list of `Person` objects.

Given the program sketch below (just including the changes), does the `quality` parameter has to be a 'by reference' parameter (the `?` would then be e.g. `ref`), or will it do with a 'by value' parameter (the `?` would then be e.g. `val`). Explain shortly why and then how you would represent the parameter in an activation record for a call of `printLabel`.

```
{
   ... // as above
   int noOfPrints=0;
   int maxQuality1;
   class Person implements Addressable, Taxable {... // as above

     void printLabel(? int quality){

        if (quality==1){...} else {...}; // print label
        noOfPrints=noOfPrints+1;
        if (noOfPrints>maxQuality1){quality=2};
      }
   }
   void main() {
     List<Person> persons;
     int printQuality=1;

     // fill the persons list with Person objects
     // set maxQuality1

     for (Person p: persons){ p.printLabel(printQuality);}
   }
}
```

By-reference

The parameters must be a link (address) of a variable.

## 1c

In this part we add two methods to class `Person`, and define `VisitingPerson` as a subclass of `Person` in the same scope as `Person`. In this language methods are by default virtual, and redefinitions must have the same signature (no variance) as the virtual method in the superclass. The method `sameAddress` is thus redefined in `VisitingPerson`:

```
class Person implements Addressable, Taxable {
  // as above
  boolean sameAddress(Person p) {
    return ((address() = p.address());
  }
  boolean samePerson(Person p) {
    return (name = p.name()& address() = p.address());
  }
}

class VisitingPerson extends Person {
  String visitingAddress;
  String visitingAddress(){return visitingAddress;}

  boolean sameAddress(Person p) {
    return ((address() = p.address())
         | (visitingAddress() = p.visitingAddress()));
  }

}
```

1. What kind of compile-time type error will this code give, and where?

`p.visitingAddress()`: Person does not define a `visitingAddress` method.

2. How would you change the program to avoid this type error?

`(VisitingPerson)p.visitingAddress()`

## 1d

In this part we try to turn the interfaces into classes in order to be able to specify the behaviour of `printLabel`. Class `Person` is now defined to have variables with types `Addressable` and `Taxable` instead of implementing the corresponding interfaces. We just consider the details of `Addressable`, and for this part of Question 1 we drop the parameter to `printLabel`. We assume that the `addr` variable of `Person` is visible, so that the `main` method can now directly call `printLabel` of a `Person p` by the call '`p.addr.printLabel()`'. The address of a `Person p` will similarly be accessible by `p.addr.address()`.

```
{
  // as above

  class Addressable {
    String address;
    String address(){return address;}
    void printLabel(){System.out.println(address());}
  }
  class Taxable {
    void payTax(int reduction(String)){...}
  }
  class Person {
    String name;
    String address;
    String name(){return name;}
    String address(){return address;}

    public Addressable addr = new Addressable();
    public Taxable tax = new Taxable();

    public void Person(String pname, paddress) {
      name=pname; addr.address=paddress;
    }
  }
  void main() {
    Person p = new Person("Birger", "Røahagan 33A");
    p.addr.printLabel();
    ...
    p.tax.payTax(taxReduction1);
    ...
  }
}
```

The problem with this solution is that `printLabel` will only print the `address` of a `Person`, while we would like it to print both `name` and `address` of a `Person`. It is not possible to define `printLabel` within class `Addressable` so that it prints the `name`, as `name` is not visible from class `Addressable`.

Sketch a solution, given that `printLabel` still has to be defined in `Addressable` (as above), and that `main` has the call `p.addr.printLabel()`. The solution must print the `name`. You may assume that a method `m` that is redefined in a subclass may call the `m` of the superclass by `super.m()`, and that the language supports anonymous classes.

```
  class Person {
    String name;
    String name(){return name;}

    Addressable addr = new Addressable(){
      System.out.println(name());super.printLabel();
    }


}
```

# Question 2. ML (weight 40%)

## 2 (a)

Assume the following definition:

```
(* val fold = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun fold f y nil = y
|   fold f y (x::xs) = fold f (f (x, y)) xs;
```

Give the result of evaluating the following *two* expressions:

```
fold (fn (x,y) => x + y) 0 [1,2,3]


((fn f => f 2) (fn x => fn y => 2 * y)) 0
```

Is the above definition of `fold` tail-recursive or not? YES or NO.

```
val it = 6 : int

val it = 0 : int
```

## 2 (b+c+d)

The following datatype can be used to describe terms over constructors and variables as we would find them e.g. in Prolog:

```
datatype term = Var of string                (* Variable *)
              | C of (string * term list); (* Constructor, e.g. f(g(X),Y) *)
```

A substitution can correspondingly be modelled as a function with the following type from one term into another:

```
type subst = term -> term;
```

Let us assume these definitions in the following.

## 2b

Given a pair `(string * term)`, representing a variable and the term it is replaced with, now define the function:

```
val apply = fn : (string * term) -> subst
```

which returns a function which will recursively apply this particular substitution on a term.

```
fun apply (v, t) : subst = fn u => case u of Var z     => if v = z then t else (Var z)
                                   |         C (cn,ts) => C (cn, map (apply (v,t)) ts);
```

## 2c+d

With the above definitions, the unification function takes two terms and returns a substitution or fails with an exception:

```
val unify = fn : term -> term -> subst
```

With these definitions, a variable can be unified with any term, yielding a substitution from that variable to the term, whereas two constructor terms can only be unified if their constructors are the

same and their sub-terms can be pair-wise unified.

We are thus neglecting the so-called "occurs check", which in a real implementation would be used to avoid infinite unification when substituting a variable with a term *again* containing that variable.

Implement the function "`unify`", using a helper-function "`fold2`" with the signature:

```
val fold2 = fn : term list -> term list -> subst
```

which unifies two lists of terms recursively: if both lists are empty, the resulting substitution is the identity-function. Otherwise, it obtains a substitution from unifying the two heads of the lists, applies this substitution to the two remainder lists, and recursively calculates the resulting substitution. Note that for the final result, the intermediate substitution and the recursively calculated substitution must be composed.

In the case that the two lists of the helper function have different sizes, e.g. when trying to unify `f(X)` with `f(a,b)`, or when the two constructors do not match, raise an exception "Fail".

The definition of "`unify`" can be quite concise, since most of the actual work can be delegated to "`fold2`" when matching two constructors, and to "`apply`" when matching a variable against a term.

Examples:

1) The following fails with an exception as expected, since we are effectively trying to calculate the unifier for `c(X,d(Y))` and `c(d(Z))`, and the arities do not match:

```
unify (C ("c", [Var "X", C("d", [Var "Y"])]))
      (C ("c", [C("d",[Var "Z"])]))
uncaught exception Fail
```

2) As a substitution is a function that cannot be printed as a result by the SML interpreter, we have to ask for the mapping of a particular variable. Unifying `c(X,d(X))` and `c(d(Z),K)`, `X` will be substituted with `d(Z)`:

```
(unify (C ("c", [Var "X", C("d", [Var "X"])]))
       (C ("c", [C("d",[Var "Z"]), Var "K"])))
(Var "X");
val it = C ("d",[Var "Z"]) : term
```

Whereas asking for `(Var "K")` yields `d(d(Z))`:

```
val it = C ("d",[C ("d",[Var "Z"])]) : term
```

```
(* 2c: Define val fold2 = fn : term list -> term list -> subst *)
  fun fold2 nil nil = (fn x => x)
  |   fold2 (x::xs) (y::ys) = let val s = unify x y
                              in (fold2 (map s xs) (map s ys)) o s end
  |   fold2 _ _ = raise Fail (* wrong number of args *)
(* 2d: Define val unify = fn : term -> term -> subst *)
  and unify (Var x) y             = apply (x,y)
  |   unify (C (c,ts)) (C (d,us)) = if c = d then fold2 ts us
                                    else raise Fail
  |   unify y (Var x)             = apply (x,y);
```

## 2e

Design one or more datatypes that enable you to model a Prolog program as a list of rules (clauses). Recall that a clause has a so-called atom as *head*, consisting of the name of the predicate and a list of terms, and an optional *body*, that is a list of atoms. Refer to the `term`-datatype from above where necessary.

```
datatype atom = A of (string * term list);
      datatype rule = R of (atom * atom list);
```

## 2f

Represent the following Prolog program as a list of rules in your datatype(s) from question 2(e):

```
p(X).

p(d(Y)) :- p(c).
```

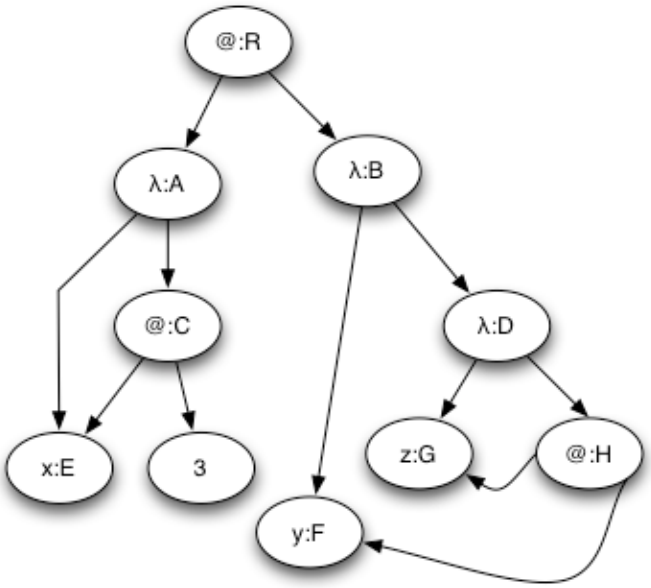
```
[R(A("p",[Var "X"]),[]),                                (* 1st rule (fact) *)
  R(A("p",[C("d",[Var "Y"])]), [A("p",[C("c",[])])])(* 2nd rule *)
```

## 2g

Use the following graph to apply the type inference algorithm to the expression, and give the type of the expression, or in the case of a type error, explain how you detect this.

```
(fn x => x 3) (fn y => fn z => z y)
```

Use the given node-labels (for better readability given in upper-case) and assume that the literal `3` has type `int` to derive the equations, and describe your steps.

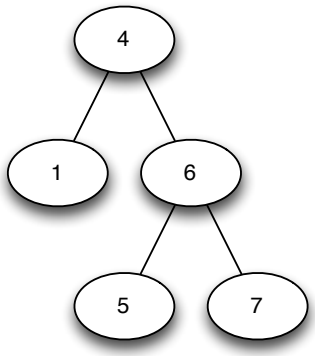


```
A = B -> R
A = E -> C
B = F -> D
E = int -> C
D = G -> H
G = F -> H

Further:
E -> C = B -> R             C=R
B=F->(G->H) = F -> ((F->H) -> H)
E=int -> R

Then:
(int -> R) -> R = B -> R            B=int->R
int->R = F -> ((F->H) -> H)

Then:  F=int    R=(int->H) -> H
```

# Question 3. Prolog (weight 20%)

## 3a

Binary trees can be represented in Prolog as terms in the following way: Let `X` be a term. Then `leaf(X)` is a tree with a single leaf containing a value `X`. Given two trees `L` and `R`, let `node(L,X,R)` be the binary tree with a root containing the term `X`, and sub-trees `L` and `R`. The following example shows a binary tree and its representation as Prolog term:




```
node(leaf(1),4,node(leaf(5),6,leaf(7)))
```

A depth-first traversal of a tree returns the element from a leaf, and for a node first visits the left sub-tree, then the value in the node, and then recursively the remaining right sub-tree. For our tree, the elements would be visited in order: `1,4,5,6,7`.

Write a binary predicate `toList` in Prolog, where `toList(X,Y)` is true when `X` is a binary tree and `Y` a list which contains the values of `X` in the exact order the depth-first traversal visits them. For our example, the following query holds:

```
toList(node(leaf(1),4,node(leaf(5),6,leaf(7))),[1,4,5,6,7]).
```

Feel free to use the predicate `append/3`, where `append(X,Y,Z)` is true when the list `Z` starts with the list `X`, followed by the list `Y`.

```
toList ( leaf (N),[N]).
toList ( node (X,N,Y),ZS):- toList (X,XS),
                            toList (Y,YS),
                            append (XS ,[N|YS],ZS ).
```

## 3b

Write a ternary predicate `del`, which takes as first argument an element `X` and in the second argument a list `XS`. The third argument should contain the list that you obtain by removing one (arbitrary) occurrence of `X` in `XS`. Define it in such a way that backtracking enumerates all possibilities to remove `X` from `XS`!

As an example, the query `del(2,[3,2,1,2,3],ZS)` should return the two possible answer substitutions `ZS=[3,1,2,3]` and `ZS=[3,2,1,3]`.

```
del(X,[X|XS],XS).
del(X,[Y|XS],[Y|YS]) :- del(X,XS,YS).
```

## 3c

Write a ternary predicate `delN`, which takes as first argument a list `XS` and in the second argument a non-negative number. The third argument should contain the list `XS` with the n-th element removed (where indices start from 1, i.e., for 0 there is nothing to remove).
As an example, the query `delN(['fee','foe','fum'],2,ZS)` should return the result `ZS=['fee', 'fum'].`

```
delN(XS,0,XS).
delN([],_,[]).
delN([_X|XS],1,XS).
delN([X|XS],N,[X|ZS]) :- N > 1, M is N-1, delN(XS,M,ZS
```