UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Exam in:	**INF3110 Programming Languages**
Day of exam:	**December 3, 2012**
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!

Contents

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

1a

NB.: This part of question 1 is answered by filling in links in the figure at page 9, and deliver it together with the rest of the answers! Remember to fill in candidate number and date!

Assume that we have a Java like language (with static scoping) that has methods as parameters to classes. A method as parameter becomes a method of the class with the parameter. Only methods with no return type and with no parameters are allowed. Actual parameters are provided as part of the generation of objects: actual parameters can be methods that are visible in the scope containing the generation of the object.

Below is a simple example according to these rules. The class `Figure` has a single formal method parameter `whenMoved`; this is in method `moveTo` called as if it was a local method of class `Figure`. It is called when the figure has been moved. The actual parameter (`gotThere`) defined in `SpeakingCanvas` will say a sentence (with an increasing volume) for each move. The method `moveFigureAround` will have some kind of control structure (indicated by … as it is not important here) that sets the value of `p` and move the `Figure f`.

The default constructor for a class with method parameters will take a list of methods as actual parameter and bind the formal method parameters to these actual methods – one does thus not need to specify such a constructor. Details like constructors and the detailed definition of the class `Point` are not included.

```
class Program {
  public static void main(String[] args) {
    SpeakingCanvas sc = new SpeakingCanvas();
    ...; sc.moveFigureAround(); ...
  }
}
class Point {int x,y; ...}
class Figure(void whenMoved()){
  Point center;
  public void moveTo(Point p){
    center = p;
    whenMoved();
  }
}
class SpeakingCanvas {
  int volume = 0;
  Figure f = new Figure(gotThere);

  public void gotThere(){
    volume = volume + 1;          // *
    sound('got there', volume)
  }
  public void moveFigureAround(){
    Point p;
    ...
    f.moveTo(p);
    ...
  }
}
```

The program execution is started by execution of the `main` method in the class `Program`.

Fill in access and control links of activation records at the stage of execution when the activation record for `gotThere` is on top, the activation record for `moveTo` is the second to the top of the run time stack, and the statement marked with * has been executed. Fill in also the links of the only closure that is needed, and indicate how the parameter `whenMoved` is represented. Note that we assume that access links may not only be links to activation records, but also to objects of classes in which methods are defined.

Answer: See last page.

1b

Now we extended the mechanism so that a method as parameter also may have parameters, with the rule that a method is an allowed actual parameter if the types of the parameters of the method are the *same* types or *subtypes* of the corresponding parameter types of the formal parameter method specification.

The following example is therefore legal: `gotThere(VolumePoint vp)` is a valid actual parameter, because the type of the parameter `vp` is a subtype of the parameter type `Point` specified for `whenMoved`. The language also has to be extended such that the variables in the `main` method (here `gp`, `gvp` and `sc`) are visible in all other classes.

```
class Program {
  public static void main(String[] args) {
    Point gp = new Point(1,2);
    VolumePoint gvp = new VolumePoint(1,2,3);
    SpeakingCanvas sc = new SpeakingCanvas ();
    ...; sc.moveFigureAround(); ...
  }
}
class Point {int x,y; ...}
class VolumePoint extends Point{
  int vol;
  int getVolume {return vol;}
}

class Figure (void whenMoved(Point)){
  Point center;
  public bool equals(Point p) {...}
  public void moveTo(Point p){
    center = p;
    whenMoved(gp);
  }
}
class SpeakingCanvas{
  int volume = 0;
  Figure f = new Figure(gotThere);

  public void gotThere(VolumePoint vp){
    int v = vp.getVolume();
    if (f.equals(vp) then {
      volume = volume + v;
      sound('got there', volume);
    }
  }
  public void moveFigureAround(){
    Point p;
    ...
```

```
        f.moveTo(p);
        ...
    }
}
```

b1)

1. The call `whenMoved(gp)` leads to a runtime type error. Which kind of type error is this? Why is it not possible to statically type check this program.

The call `whenMoved(gp)` via the call `f.moveTo(p)` will imply that `gotThere` is called with `gp` that denotes a `Point` object, and `gotThere` attempts to call the `getVolume` method.

2. Would it be possible to solve this by explicit casting? Answer Yes or No, with a few lines of explanation.

No. The problem arises from the assignment of `gp` to the formal parameter `vp` as a result of the call `whenMoved(gp)` via the call `f.moveTo()`. An explicit casting should have been `whenMoved((VolumePoint)gp)`, but that would rule out calls `whenMoved(ap)` where `ap` denotes a `Point` object, and that should be allowed in cases where `f` is given an actual parameter with a same parameter type, i.e. `Point`.

3. Would another value of `gp` make the program not have a runtime type error? If Yes, please indicate the value, if No, explain why.

Yes. `gp = new VolumePoint()`

b2)

This is not illustrated in the code above, but the idea with a method as parameter to class `Figure` is that e.g. a class `PlayingCanvas` may have a method similar to `gotThere` in `SpeakingCanvas` that plays a tune instead of saying a sentence. The class `Figure` will work for both Canvas classes by providing different actual parameters as part of generating the `Figure` -object.

How would you achieve the same effect without using methods as parameters, but only mechanisms like classes, subclasses and overriding of methods? Sketch a solution by sketching the classes `Figure`, `SpeakingCanvas` and `PlayingCanvas`. You may need to introduce one or more new classes. Include in the sketch how this line of code

```
Figure f = new Figure(gotThere);
```

will be in the new solution.

```
class Figure {
  Canvas c;
  void Figure(Canvas cp) {c=cp}
  Point center;
  public bool equals(Point p) {...}
  public void moveTo(Point p){
    center = p;
    c.whenMoved(gp);
  }
}

Class Canvas {
  int volume = 0;
  Figure f = new Figure(this);

  public abstract void whenMoved (VolumePoint vp);

  public void moveFigureAround(){
    Point p;
    ...
    f.moveTo(p);
    ...
  }

}

class SpeakingCanvas extends Canvas {

  public void whenMoved (VolumePoint vp){
    int v = vp.getVolume();
    if (f.equals(vp) then {
      volume = volume + v;
      speak('got there', volume);
    }
  }
}
class PlayingCanvas extends Canvas {

  public void whenMoved (VolumePoint vp){
    int v = vp.getVolume();
    if (f.equals(vp) then {
      volume = volume + v;
      sound('got there', volume);
    }
  }
}
```

Question 2. ML (weight 40%)

2 (a)

Assume the following definition:

```
fun map : ('a -> 'b) -> 'a list -> 'b list
fun map f nil     = nil
  | map f (x::xs) = (f x) :: map f xs ;
```

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

```
map (fn x => x * 2) [1,2,3]
[2,4,6]

(fn y => y 2) (fn y => 2 * y)
4
```

2 (b+c+d+e)

Given the two following datatypes, one for expression trees, and one for a stack-based language:

```
datatype exp =
    Const of int
  | Neg of exp
  | Add of (exp*exp);

datatype stacklang =
    Push of int
  | Op of (int list -> int list);
```

We can then for example write an expression resembling the term "~(3+5)":

```
val t = Neg (Add ((Const 3), (Const 5)) : exp
```

Instead of evaluating this term directly, we would like to convert it into its stack language-representation first, a list of stack-operations: all arguments to arithmetic operations must first be pushed onto the stack (here, a list of integers). An operation will remove as many elements from the top of the stack as it needs, and put the result back. For example, the above expression can be converted into:

```
val st = [Push 3, Push 5, Op fAdd, Op fNeg] : stacklang list
```

2b

Define the necessary operations `fAdd` and `fNeg` of type `int list -> int list`. Use pattern matching to take the required number of arguments from the stack, and return the updated stack. It is okay for an operation to crash when there are not enough arguments on the stack.

See 2c.

2c

Define the function `convert : exp -> stacklang list`, which turns an expression into its corresponding list of stack operations, using `fAdd` and `fNeg` from 2b) as illustrated in the example.

```
fun cvt (e : exp) : stack list =
 case e of Const i => [Push i]
 | Neg e => (cvt e) @ [Op (fn (s1::ss) => (~ s1)::ss)]
 | Add (e1,e2) => (cvt e1) @ (cvt e2) @ [Op (fn (s1::s2::ss) => (s1 + s2)::ss)]
 ;
```

2d

Define the function `evalStack : stacklang * int list -> int list` which, given a *single* statement and an input stack, computes the resulting stack.

```
fun eval ((stmt : stack), (st : int list)) : int list =
  case stmt of Push i => i::st
             | Op f => f st
             ;
```

2e

Define the function `evalExp : exp -> int list`, which *first* uses `convert` (from 2c) to turn the expression into the stack language, and *then* execute that intermediate stacklang-program element-wise with `evalStack` (from 2d), starting with an empty input-stack.

Use the infix function composition operator `o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b`, where `f o g = fn x => f (g x)`, read as "f after g", to define this evaluation function, and the built-in function `foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`, which "collapses" a list element-wise using a binary function and start value, defined as:

```
fun foldl f z nil     = z
  | foldl f z (x::xs) = foldl f (f (x, z)) xs;
```

Use only functions defined above (2c&d), and the predefined functions `foldl` and `o`. Tip: You may want to think about how you could use `foldl` to evaluate a stacklang-program first, and then use this insight in the final solution to handle an `exp`.

```
val evalExp : exp -> int list =
  (foldl (eval) []) o cvt;
```

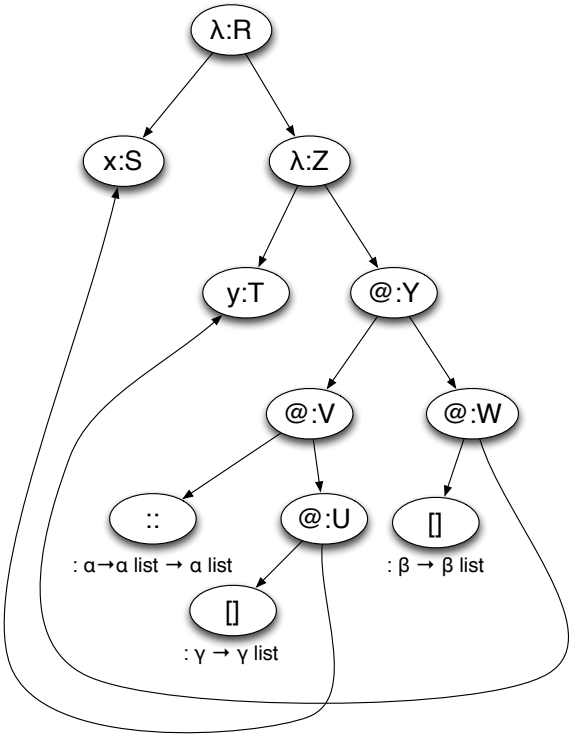
2f

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.

```
fun f x = fn y => [x] :: [y]
```

Use the given node-labels (for better readability given in upper-case) and assumed signatures (with Greek letters as type variables) to derive the equations, and describe your steps. Note that the

polymorphic function "`[]`", which constructs a singleton list, occurs *twice*, each time with different type variables in its signature.



1) γ → γ list = S → U

2) α→α list → α list = U → V

3) β → β list = T → W

4) V = W→Y

5) Z = T→ Y

6) R = S→ Z

6+5) R = S→T→Y

1) S= γ, U= γ list, 1+2) α= γ list

3) T= β, W= β list

2+4) U= α , W= α list, Y= α list

3/2+4) α= β

R= γ → γ list → γ list list

Question 3. Prolog (weight 20%)

3a

Define a predicate `append3` which has four arguments `Xs`, `Ys`, `Zs`, `Rs`, and which is fulfilled if `Rs` is the result of appending the lists `Xs`, `Ys`, and `Zs`.

Please define it "from scratch," i.e. using three rules that mention only the append3 predicate. Do *not* use the built-in append predicate for two lists, or any other auxiliary predicate.

Examples:

```
?- append3([1,2], [a,b], [3,4], Rs).
Rs = [1,2,a,b,3,4].

?- append3([], [1,2], []).
Rs = [1,2].
```

The second example mistakenly has only three arguments. Should have been an extra Rs at the end. To a student who understood the question, this should have been obvious, but if there are problems that might be a consequence of this mistake, we can adjust the scoring.

```
append3([],[],Zs,Zs).
append3([],[Y|Ys],Zs,[Y|Rs]) :- append3([],Ys,Zs,Rs).
append3([X|Xs],Ys,Zs,[X|Rs]) :- append3(Xs,Ys,Zs,Rs).
```

3b

Define a predicate `sublist(Xs,Ys)`, which is fulfilled if `Xs` is a sublist of `Ys`, i.e. if `Ys` starts with some (possibly 0) arbitrary elements, followed by all elements of `Xs`, in order, followed by some more (possibly 0) arbitrary elements. Please use `append3` as auxiliary predicate in your definition.

Examples:

```
?- sublist([a,b],[1,2,a,b,3]).
yes

?- sublist(Xs,[a,b]).
Xs = [a,b] ? ;
Xs = [b] ? ;
Xs = [] ? ;
Xs = [a] ? ;
No

sublist(Xs,Ys) :- append3(_,Xs,_,Ys).
```

3c

Let a collection of "list rewrite rules" be given as facts `rewrite(To,From)`. For instance

```
rewrite([a,b],[b,a]).
rewrite([c,c],[c]).
```

E.g., the first of these facts means that any occurrence of `[a,b]` in a list may be replaced by `[b,a]`, etc.

They are just examples, your code should work for any set of such `rewrite` facts.

The first argument, `From`, is called the left side of the rewrite rule, and the second argument, `To`, is called the right side.

Implement a predicate `onestep(Xs,Ys)` that is true if `Xs` and `Ys` are lists, and Ys is obtained by finding one occurrence of a left side of some rewrite rule in `Xs`, and replacing it by the corresponding right side.

For instance, with the two rewrite rules above, we would get

```
?- onestep([a,b,c,c,b,a],Ys).
Ys = [b,a,c,c,b,a] ? ; % because of the first rule
Ys = [a,b,c,b,a] ? ;    % because of the second rule
no
```

(underlining added to show where some elements were replaced)

Hint: use `append3` in your definition.

```
onestep(L,K) :- rewrite(From,To),
                append3(Pre,From,Post,L),
                append3(Pre, To, Post, K).
```

3d

Now use `onestep` to define a predicate `manystep(Xs,Ys)` that is true if

`Ys` can be obtained from `Xs` by applying 0 or more rewriting steps.

For instance:

```
?- manystep([a,b,c,c],Rs).
Rs = [a,b,c,c] ? ; % no rules applied
Rs = [b,a,c,c] ? ; % first rule
Rs = [b,a,c] ? ;   % first, then second rule
Rs = [a,b,c] ? ;   % second rule
Rs = [b,a,c] ? ;   % second, then first rule
no
```

Note: it doesn't matter if the some results are repeated.

```
manystep(L,L).
manystep(L,N) :- onestep(L,M), manystep(M,N).
```

3e

Based on `manystep`, define a predicate `normalform(Xs,Ys)` that is true if

- ☐ `Ys` can be obtained from `Xs` by applying 0 or more rewriting steps.
- ☐ `Ys` cannot be rewritten further, i.e. No rewrite rule can be applied to `Ys`.

For instance:

```
?- normalform([a,b,c,c,b,a],Rs).
Rs = [b,a,c,b,a] ? ; % applied first, then second rule
Rs = [b,a,c,b,a] ? ; % applied second, then first rule
no
```

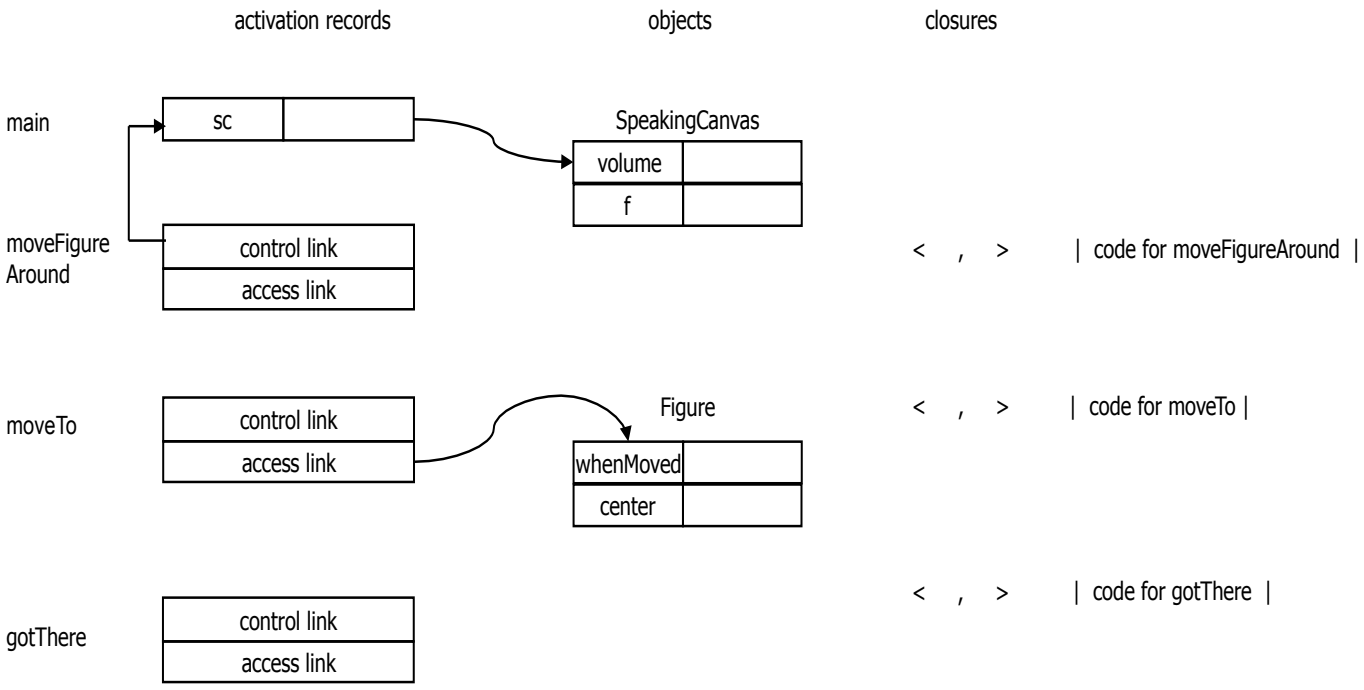
Hint: there are several possibilities to do this, some with and some without auxiliary functions in addition to the ones already defined. To capture that *no* further rewriting is possible, you can use either cut (`!`) or Prolog's negation by failure (`\+`).

```
normalform(L,N) :- manystep(L,N), (\+ onestep(N,_)).
```

Page for answering Question 1a

Candidate no: .....

Date: .....



Answering Question 1a

Candidate no: .....

Date: .....