UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Exam in:	**INF3110 Programming Languages**
Day of exam:	**2. December 2010**
Exam hours:	**14:30 – 18:30**
This examination paper consists of 8 pages	
Appendices:	**No**
Permitted materials:	**All printed and written**

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%)

Part I

1a

The following is a fragment of code in a language with static scoping:

```
{
  int x=0;
  void p(int y) {
    y=1;
    x=0;
  };
  p(x)
}
```

It defines a variable x, a function p with formal parameter y, and the function p is called with x as actual parameter.

What will the value of x be after the call p(x) for these different cases of the parameter y:

Specification of the parameter y	**by-value**	**by-reference**	**by-value-result**	**by name**
Value of x after the call p(x)				

Answer

Specification of the parameter y	**by-value**	**by-reference**	**by-value-result**	**by name**
Value of x after the call p(x)	0	0	1	0

1b

In the following code fragment the p has become a method of a class, and it has been extended with a statement assigning a new A object to the rA in the enclosing scope.

```
{
   class A {
     int x=0;
     void A(int i) {x=i};
     void p(int y) {
       rA = new A(2);
       y=1;
       x=0;
     };
  }
  A rA = new A(1);
  rA.p(rA.x)
}
```

What will the value of rA.x be after the call rA.p(rA.x) for these different cases of the parameter `y`:

Specification of the parameter `y`	**by-value**	**by-reference**	**by-value-result**	**by name**
Value of `rA.x` after the call `rA.p(rA.x)`				

Answer

Specification of the parameter `y`	**by-value**	**by-reference**	**by-value-result**	**by name**
Value of `rA.x` after the call `rA.p(rA.x)`	2	2	2	1

Part II

1c

The Observer pattern is an example of a pattern where two classes (and their anticipated subclasses) are dependent on each other. Assume that we have a language with virtual classes: A virtual class is specified so that it can only be redefined to subclasses of a bound class. This may be used to specify the dependency between the Observer and Subject classes, and to specify the requirement on the type of events:

```
{
   class Event {...}
   class Observer {
     virtual class S extends Subject;
     virtual class E extends Event;

     public void notify(E e, S s){...}
   }
   class Subject {
     virtual class O extends Observer;
     virtual class E extends Event;
     observers () O;
     public void register(O o){...}
     public void notifyObservers(E e, S s){
       ... observers(i).notify(this, e); ...}
   }
   Observer someObserver;
}
```

`Observer` objects call `register` on the `Subject` objects they want to observe. Each `Subject` object notifies the observers (kept in an array observers) when an event has happened. `Subject` and `Observer` are dependant on each other, and the virtual classes express that subclasses of `Observer` and `Subject` are similarly dependant on each other.

For a given type of events, the subclasses of `Observer` and `Subject` are specified with redefinitions of the virtual classes in order to reflect that `WindowObserver` objects only observe `WindowSubject` objects and that `WindowSubject` objects only are observed by `WindowObserver` objects:

```
class WindowEvent extends Event {...}

class WindowObserver extends Observer{
  class S is WindowSubject;
  class E is WindowEvent;
  String label;
  ...
}
class WindowSubject extends Subject {
  class O is WindowObserver;
  class E is WindowEvent;
  String label;

  public void windowMethod(){...}
  ...
}
```

Consider the following two statements as part of the `windowMethod`, assuming that the `someObserver` in the enclosing scope is visible:

```
... print(someObserver.label);
... print(observers(i).label);
```

Insert casting or castings in order to get the required type checking done.

Answer:

```
... print((WindowObserver)someObserver.label);
// someObserver is typed by Observer, and Observer does not have label

... print(observers(i).label);
// no run-time type check
```

1d

We now assume that we do not have virtual classes. The pattern and an application of it will then look as follows:

```
class Event {...}
class Observer {
  public void notify(Event e, Subject s){...}
}
class Subject {
  observers ()Observer;
  public void register(Observer o){...}
  public void notifyObservers(Event e, Subject s){
    ... observers(i).notify(this, e); ...}
}
Observer someObserver;


class WindowEvent extends Event {...}

class WindowObserver extends Observer {
```

```
    String label;
    ...
  }
  class WindowSubject extends Subject {
    String label;

    public void windowMethod(){...}

    ...
  }
```

Consider the following code as part of the windowMethod:

```
  ... print(someObserver.label);
  ... print(observers(i).label);
```

Insert the casting or castings in order to get the required type checking done.

Answer:

```
  ... print((WindowObserver)someObserver.label);

  ... print((WindowObserver)observers(i).label);
```

Question 2 ML (weight 40%)

2a

Let's define a filter function that only keeps list elements that satisfy a given predicate:

```
fun filter p nil     = nil
|   filter p (x::xs) = if (p x) then (x :: (filter p xs)) else (filter p xs)
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

1) What is the result of evaluating the expression:

```
      filter (fn x => x > 2) (map length [[1,2],[3,4,5],[6]])
      Answer: val it = [3] : int list
```

2) What is the type of the result? **Answer: int list**

3) What is the result of evaluating the expression:

```
      filter (fn x => x > 2) (map length [[[9,8],[7,6,5,4],[3]],[[6]]])

      Answer: val it = [3] : int list
```

4) Given the following function

```
      fun f x = if x < 2 then x = 0 else f (x-2)

      val f = fn : int -> bool
```

What is the result of evaluating the expression `f 1001`? **Answer: false**

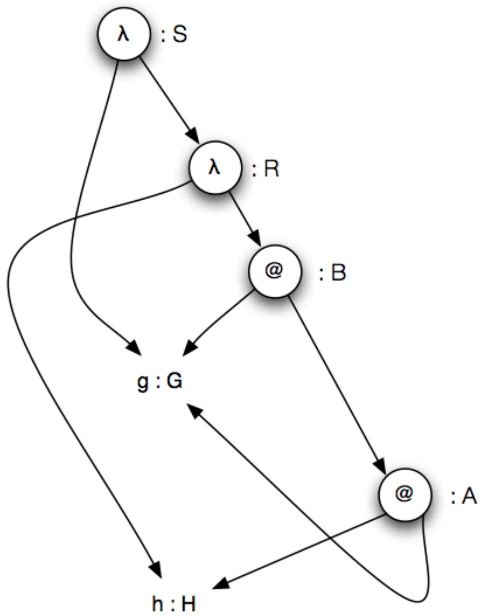
2b

Draw the type inference parse tree of the following function and describe the type inference steps of the function according to this tree. What is the type of f?

```
fun f = fn g => fn h => g(h(g))
```

Hint: the final result will not contain concrete types such as `int` or `string`, but only type variables.

Answer:



H = G -> A
G = A -> B
R = H -> B
S = G -> R = (A->B) -> (H->B) = (A->B) -> ((A->B)->B)
val f = fn : ('a -> 'b) -> (('a -> 'b) -> 'a) -> 'b

2c

Let's consider foraging lemmings ("lemen" in Norwegian). Each lemming has a non-negative `int`-value indicating its hunger. When a lemming eats nuts, this will satiate him corresponding to the (non-negative) number of nuts. We model this the following way in ML:

```
datatype lemen = Lemen of int ;

datatype noeter = Noeter of int ;
```

Given are also the two functions "driver" (Norwegian for doing something), and "spise" (to eat):

```
fun driver f (x::nil) = nil                    (lemmings don't like to eat alone!)

|   driver f (x::xs) = (f x) :: (driver f xs) ;

fun spise (Lemen i) (Noeter j) = if j > i then Lemen 0

                                               else Lemen (i - j) ;
```

2c.1) What is the result of evaluating the following expression?

```
driver (fn x => spise x (Noeter(3))) [Lemen 5, Lemen 2, Lemen 2] ;
```

SOLUTION: val it = [Lemen 2,Lemen 0] : lemen list

2c.2) Lemmings collect and hide their nuts in summer in their nest, which is a cave system. A cave system has caves with food, empty caves, and branches into other cave systems left and right.

```
datatype nest = Cave of noeter
              | EmptyCave
              | Branch of nest * nest ;
```

Let's also define the ML-notion of "direction":

```
datatype direction = Left | Right ;
```

Define a function "`hide`" of type

```
val hide = fn : nest -> noeter -> direction list -> nest
```

The arguments are a nest, collected nuts, and a path of type `direction list`.

The path elements of either "Left" or "Right" indicate, which branch to take. The result is an updated nest, where the food has been deposited in the chamber that the path pointed to.

Assume that a path is always in so far valid, as that it will always end in an *empty* cave, which correspondingly should be updated to a cave holding the given amount of nuts. The remainder of the nest stays unchanged.

SOLUTION:

fun hide EmptyCave f nil = Cave f

| hide (Branch (l,r)) f (Left::ds) = Branch(hide l f ds, r)

| hide (Branch (l,r)) f (Right::ds) = Branch(l, hide r f ds) ;

2c.3) Define the function

```
val maxCave = fn : nest -> nest
```

which should return the cave with the most nuts in a nest. If the nest does not contain a cave with nuts, `EmptyCave` should be returned.

SOLUTION:

```
fun maxCave EmptyCave     = EmptyCave
|  maxCave (Cave(f))     = Cave(f)
|  maxCave (Branch(l,r)) = max (maxCave l, maxCave r) ;


fun max EmptyCave y = y
|  max (Cave(i)) EmptyCave = Cave(i)
|  max (Cave(Noeter(i))) (Cave(Noeter(j))) = if i > j then Cave(Noeter(i))
                                             else Cave(Noeter(j)) ;
```

Question 3 Prolog (weight 20%)

Peano's encoding of the natural numbers uses the constant `zero`, and a unary successor function "`s`". We can use this encoding in Prolog. For example, the number 3 is represented as `s(s(s(zero)))`.

We extend this notion with an additional symbol "`p`" for the predecessor of a number. With zero, s, and p, we can represent every integer value: `p(p(zero))` for example represents the value -2.

Alas, now every number can be represented by infinitely many terms! The terms `s(zero)`, `s(p(s(zero)))`, and `s(p(p(s(s(zero)))))` represent the number 1. We call a term normalized, if it does not use the symbols `p` and `s` at the same time. This means that `s(p(s(zero)))` and `s(p(p(s(s(zero)))))` are not normalized. Every integer has a unique normalized representation.

3a) Implement a unary Prolog predicate `normalized` that determines if the term representation of a number is normalized. Use two helper predicates that check separately whether a term contains only `zero` or `s(s(..(zero)..))`, or only `zero` or `p(p(..(zero)..))`.

SOLUTION

```
normalized(X) :- normalizedS(X).
normalized(X) :- normalizedP(X).

normalizedS(zero).
normalizedS(s(X)) :- normalizedS(X).

normalizedP(zero).
normalizedP(p(X)) :- normalizedP(X).
```

3b) Implement the binary Prolog predicate `normalize` that normalizes a number in our term representation. For example, it should be true that `normalize(s(p(s(zero))), s(zero))`.

SOLUTION (two possibilities)

```
normalize(zero,zero).
normalize(s(X),Y)       :- normalize(X,p(Y)).
normalize(s(X),s(s(Y))) :- normalize(X,s(Y)).
normalize(s(X),s(zero)) :- normalize(X,zero).
normalize(p(X),Y)       :- normalize(X,s(Y)).
```

normalize(p(X),p(p(Y))) :- normalize(X,p(Y)).

normalize(p(X),p(zero)) :- normalize(X,zero).

normalize(X,Y) :- norm(X,zero,X).

norm(zero,X,X).

norm(p(X),zero,Z) :- norm(X,p(zero),Z).

norm(p(X),s(Y),Z) :- norm(X,Y,Z).

norm(p(X),p(Y),Z) :- norm(X,p(p(Y)),Z).

norm(s(X),zero,Z) :- norm(X,s(zero),Z).

norm(s(X),s(Y),Z) :- norm(X,s(s(Y)),Z).

norm(s(X),p(Y),Z) :- norm(X,Y,Z).

3c) Implement a ternary Prolog predicate `plus` that adds two numbers in our term representation. You do not need to normalize the result.

Hint: use the equalities zero+x = x, s(x)+y = s(x+y), p(x)+y = p(x+y).

SOLUTION:

plus(zero,X,X).

plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

plus(p(X),Y,p(Y)) :- plus(X,Y,Z).

3d) Combine the predicates for normalization and addition into a single predicate `normPlus` that calculates the normalized result of addition.

You do not need to have completed assignments b) and c) for this.

SOLUTION:

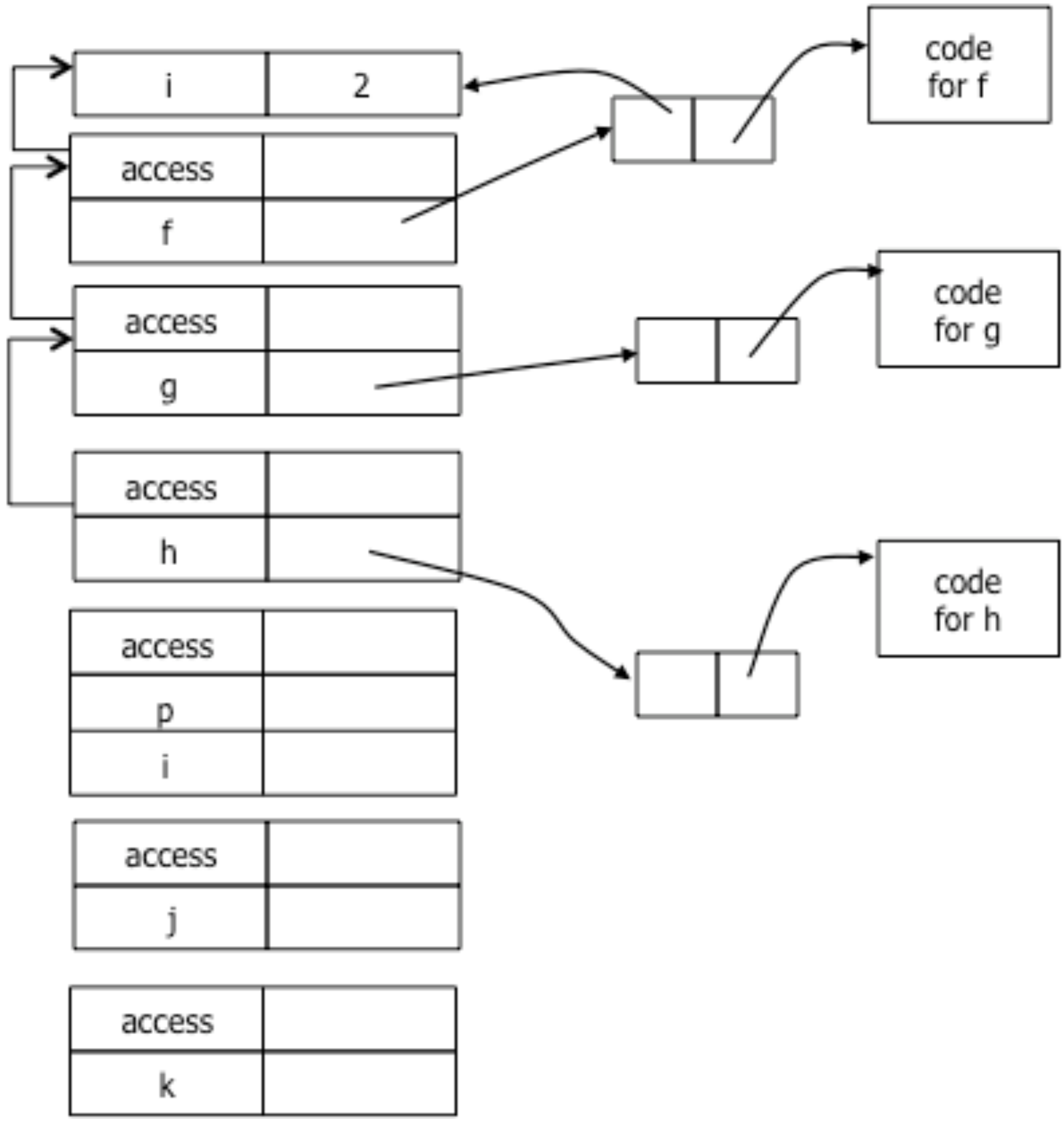
normPlus(X,Y,A) :- plus(X,Y,Z), normalize(Z,A).

Page for answering Question 1a

Candidate no: ....

Date: ....

1a



Page for answering Questions 1d & 1e

Candidate no: ....

Date: ....

1d

	Explanation
`swapper(a)`	

`Object temp = swappee[0];`	`Temp has type Object`
	`swappee[0] will have type Shape`
	`Shape <: Object, so no runtime check is needed`

`swappee[0]  = swappee[1];`	
`swappee[1]  = temp;`	

1e

	Run time types and type checks
`Object temp = swappee[0];`	`Object temp = (Object)swappee[0];`
`swappee[0]  = swappee[1];`	
`swappee[1]  = temp;`	

Extra page for sketching the answer to Question 1a