UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Exam in: **INF3110 Programming Languages**

Day of exam: **1. December 2009**

Exam hours: **14:30 – 17:30**

This examination paper consists of 10 pages including pages 8 and 9 that are used for answering parts of question 1, and page 10 that may be used for sketching.

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%) 2

Question 2 ML (weight 40%) 4

Question 3 Prolog (weight 20%) 5

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

NB.: ***1a, 1d,*** *and* ***1e*** *are answered by using pages 8 and 9, and deliver them together with the rest of the answers! Remember to fill in candidate number and date!*

Part I

The following program is in the C-like notation for ML-programs used in the book, expressing that each declaration introduces a block (by the enclosing `{…}`). The figure at page 8 shows the stack and closures in the situation when the call of `f` is on the top of the stack. For convenience the control links are omitted, as the layout of the stack in the figure reflects the control links. Execution starts by executing `h(g)`.

```
{ int i = 2;
  { int f(int k){return i*k}
    { int g(int j){return i*f(j);}
       { int h(int→int p) {
                 int i = 3;
                 return i+p(4);
               }
         h(g);
      }
    }
  }
}
```

1a

Fill in missing access links, missing links in the closures, and values of variables of activation records at the stage of execution right after `f` has returned its value and before `f` is popped of the stack.

1b

Given that we have static scoping, what is the value returned by `h(g)`?

Given that we have dynamic scoping, what is the value returned by `h(g)`?

Part II

Java array types are covariant with respect to the types of array elements, i.e. if B <: A, then B[] <: A[] (if B is a subtype of A then B[] is a subtype of A[]). This can be useful for creating functions that operate on many types of arrays. For example the following function takes in an array and swaps the first two elements in the array:

```
public void swapper(Object[] swappee) {
  if (swappee.length>1){
    Object temp = swappee[0];
    swappee[0]  = swappee[1];
    swappee[1]  = temp;
  }

}
```

`Object` is the class that is the superclass of all classes that do not have an explicit superclass.

The function work as it is and does not produce any type errors at compile time or run time.

1c

Suppose `a` is declared by `Shape[] a` to be an array of shapes, where `Shape` is a class. Explain why the principle "if B <: A, then B[] <: A[] " allows the type checker to accept the call `swapper(a)` at compile time.

1d

Suppose that `Shape[] a` is as in **1a**. Explain why the call `swapper(a)` and execution of the body of `swapper` will not cause a type error or exception at run time. Do this by explaining for each statement in the table at page 9 what are the types involved in the call and in the assignments.

1e

Java may insert run time checks to determine that all the objects are of the correct types. What run time checks are inserted in the *compiled* code for the swapper function and where? Answer this by filling in the table at page 9. Use the cast notation `(C)` in order to specify which run time type check (cast) that is involved in each assignment. One line is filled in, telling that at run time it is checked that the type of `swappee[0]` may be casted to `Object`, as the type of `temp` is `Object`.

1f

A friend of you suggest that Java arrays should follow contravariance instead of covariance (i.e. if B <: A, then A[] <: B[]). It is claimed that this will eliminate the need for run time type checks.

Write three lines or fewer of code that will compile fine under your friend's new type system, but will cause a runtime error. You may assume that you have two classes, A and B, that B is a subtype of A, and that B contains a method, `foo`, not found in A. Here are two declarations that you may assume before your three lines of code:

```
B b[];
A a[] = new A[10];
```

Question 2 ML (weight 40%)

2a

Write a function `removeElement` that gets a list (`L`) and a number (`i`) as inputs:

```
fun removeElement(L: list , i: int )= ...
```

The function searches for the input number in the list and removes all occurrences of the number `i` from the list `L`.

2b

Write a function `gcd` that gets two integers as inputs and calculate the greatest common divisor of them. Greatest common divisor is the greatest integer that divides both. (The inputs are positive numbers.)

```
fun gcd(i: int, j: int)= ...
```

2c

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

```
fun g(h,x) = h(h(x)) + 3
```

2d

Write a function `f` that gets a list `L=[x1, x2, ... Xn]` as input and returns this value:

```
(X1* 3) * (X2 * 3) * ...* (Xn * 3) + 8
```

```
fun f(L: list)= ...
```

2e

Reconsider the function `f` from **2d**. Can you use the exception handling mechanism in your function definition to make it more efficient?

If yes, show how.

2f

Define the function of problem **2d** using the higher order function "map".

Question 3 Prolog (weight 20%)

Merge sort is a comparison-based sorting algorithm invented by John von Neumann in 1945. The algorithm works as follows.

- An empty list or a singleton list (a list with exactly one element) is already sorted.
- In order to sort a list with two or more elements, split the list into two sublists of approx. equal length.
- Sort the two sublists by recursively applying the merge sort algorithm.
- Merge the two sorted lists into a single sorted list.

You will implement the merge sort algorithm as a Prolog program. First, we warm up with some easier questions.

3a

Write rules to define the following predicates.

- `empty_list/1` such that `empty_list(`*l*`)` is true if and only if *l* is an empty list
- `one_element_list/1` such that `one_element_list(`*l*`)` is true if and only if the list *l* has exactly one element
- `min_two_element_list/1` such that `min_two_element_list(`*l*`)` is true if and only if the list *l* has at least two elements

3b

Write rules to define the predicate `is_sorted/1` such that `is_sorted(`*l*`)` is true if and only if *l* is a sorted list of integers. A list is sorted if the first element is less than or equal to the second element, and the second element is less than or equal to the third, etc. Hint: recursion and integers comparison with =<.

3c

In order to implement merge sort, we need a predicate for splitting lists. We will split lists by extracting all members in odd positions to one list, and all members in even positions to the other. Write rules to define a predicate `split/3` such that `split(`*l*`,`*o*`,`*e*`)` splits the list *l* into the approximate equal size lists *o* and *e*. The list *o* shall contain the first, third, fifth, etc. element of *l*, and *e* shall contain the second, fourth, sixth, etc. element of *l*. Hint: recursion with three cases: an empty list, a list with exactly one element, a list with at least two elements.

Examples:

```
| ?- split([], O, E).

E = []
O = []

yes
| ?- split([1],O,E).

E = []
O = [1]

yes
| ?- split([0,1,2,3,4],O,E).

E = [1,3]
O = [0,2,4]

yes
```

3d

Write rules to define the predicate `merge/3` such that `merge(x,y,r)` merges two sorted lists *x* and *y* into a single sorted list *r*. Hint: use recursion to repeatedly take the larger of the heads of *x* and *y* until both are empty.

Examples:

```
| ?- merge([],[],R).

R = []

yes
| ?- merge([0],[],R).

R = [0]

yes
| ?- merge([],[0],R).

R = [0]

yes
| ?- merge([1,2,3],[0,1,2],R).

R = [0,1,1,2,2,3]

yes
```

3e

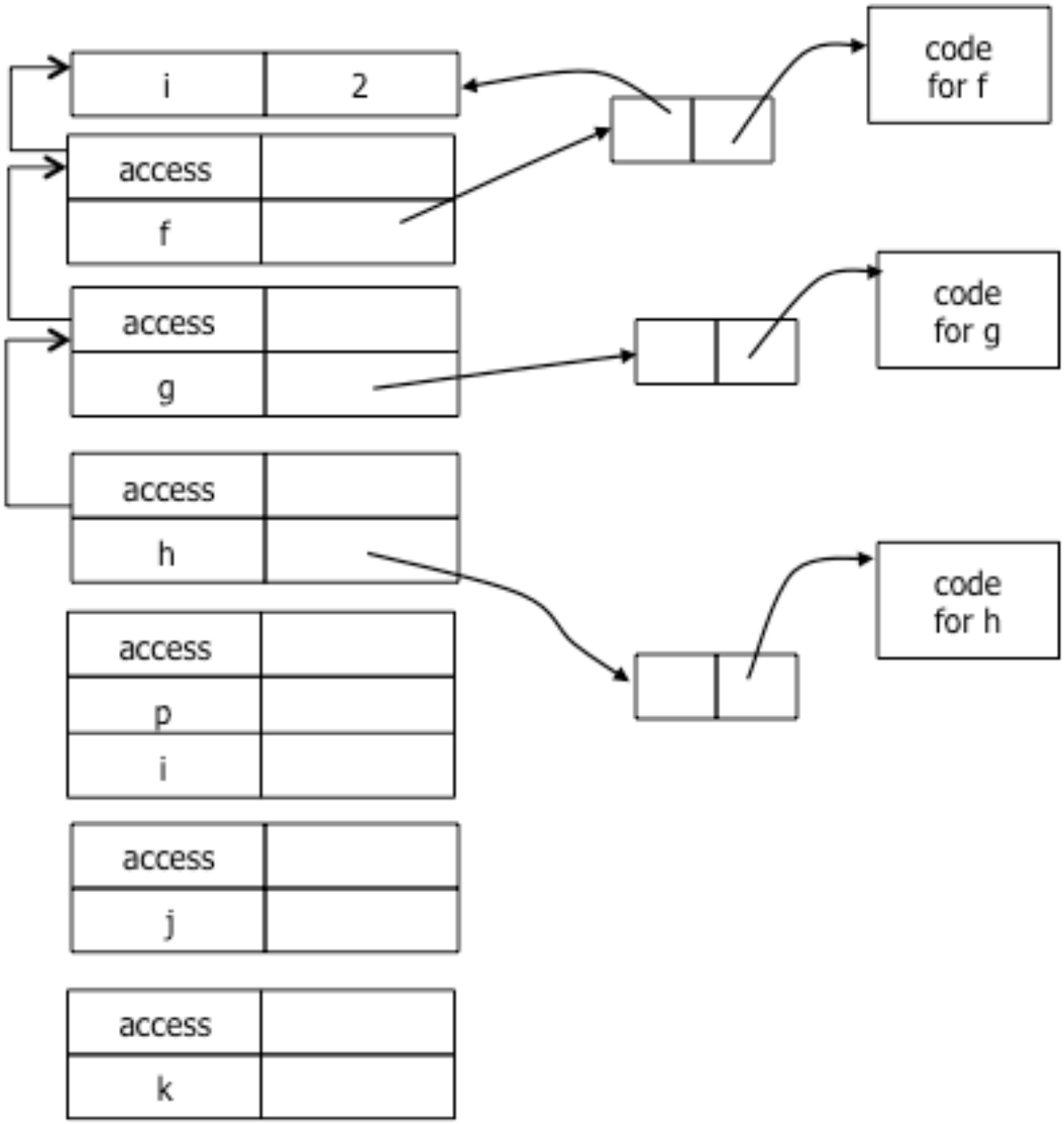
Write the predicate `mergesort/2` such that `mergesort(l,r)` sorts the list *l* into the list *r*. You should use the predicates `split/3` and `merge/3` which you defined in the previous questions. Hint: read the description of the algorithm at the beginning of this exercise to find out in which cases you have to split and sort recursively, and in which you don't.

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