Programming Paradigms
Written Exam (6 CPs)

06.07.2016

First name		Last name	
Student number		Signature	

Instructions for Students

- Write your name and student number on the exam sheet and on every solution sheet you hand in and also sign them.
- This is a closed book exam: the only resources allowed are blank paper and pens (do not use pencils).
- Write neatly and clearly. The clarity of your explanations will affect your grade.
- The duration of the exam is 2 hours.

Good luck!

Do not write in this space

Question	Marks	Achieved
1	24	
2	8	
3	12	
4	12	
5	14	
6	12	
7	8	
8	10	
Total	100	

Exercise 1 (24 marks)

a. (4 marks) Briefly describe the main difference between a compiled language and an interpreted language.

b. (4 marks) Briefly describe the concept of tail recursion, and why it is desirable to write tail recursive functions.

c. (4 marks) Is the following Ruby expression syntactically correct?

```
['2', 'times', '4', 'is not', "#{2+4}"].each { |x| puts x }
```

If no, explain what is wrong. If yes, what does the expression do?

d. (4 marks) What is the following Prolog program doing?

```
foo :-
    repeat,
    read(X),
    write(X),
    nl,
    X = 'quit',
    !.
```

e. (4 marks) Briefly explain how synchronous messaging is achieved in Erlang?

f. (4 marks) Consider the Haskell function `prod x y = x * y`. How is the function call `prod 2 4` evaluated? What is the name of this evaluation concept?

Exercise 2 (8 marks) Extend the Ruby class `Fixnum` with a method `square_root_times` that, if called for a number n and a code snippet, executes the code snippet $\lceil\sqrt{n}\rceil$ times. For example, `5.square_root_times{ puts 'hello world!' }` produces

```
hello world!
hello world!
hello world!
```

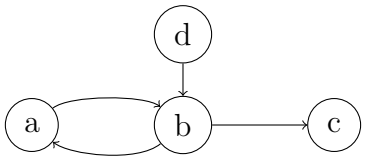
You are not allowed to use a built-in Ruby function to compute the square root of numbers.

Exercise 3 (12 marks) Write a Ruby function `prime_numbers` that has one input parameter `n` and returns an array containing the first `n` prime numbers (2 is the first prime number).

Exercise 4 (12 marks) Write a Prolog program `subst(X,L1,Y,L2)` that replaces all occurrences of `X` in list `L1` by two occurrences of `Y` to obtain list `L2`. For example, `subst(a,[a,b,c],1,L2)` instantiates `L2=[1,1,b,c]`.

Exercise 5 (14 marks) Given is a directed graph `G`, which is represented by a set of arc facts of the form `arc(X,Y)`. Here is an example:

```
arc(a,b).
arc(b,a).
arc(b,c).
arc(d,b).
```



Write a Prolog predicate `path(X,Y,P)` that computes a path `P` from a node `X` to a node `Y`, if such a path exists. An edge can occur only once in a path, i.e., cycles should be avoided. For instance, `path(a,c,P)` should return `P = [(a,b), (b,c)]`; path `P = [(a,b), (b,a), (a,b), (b,c)]` is not valid since arc `(a,b)` occurs twice. The order of the arcs in the returned path does not matter, i.e., `P = [(b,c), (a,b)]` is also ok.

Exercise 6 (12 marks) Write a function `loop` for an Erlang process that implements an accumulator for numbers and reacts as follows to messages: if a number is received, it is added to the accumulator; if `"reset"` is received, the accumulator is reset to zero; if `"result"` is received, the value of the accumulator is printed out; if `"exit"` is received, the process is stopped. In all cases a short log message is printed out.

Show the following steps on using the process:

- Start the process
- Send a message to increment the accumulator by 10
- Show the value of the accumulator
- Stop the process

Exercise 7 (8 marks) Look at the following recursive Haskell program.

a. (4 marks) Briefly describe what the program does.

```
mystery :: [a] -> Integer
mystery [] = 0
mystery (h:t) = 1 + (mystery t)
```

b. (4 marks) Transform this program into a tail-recursive one.

Exercise 8 (10 marks) Write a Haskell module that exports a function `noOfElem` that takes as input an element `x` and a list and returns the number of occurences of `x` in the list. For instance, `NoOfElem 1 [1,2,3,1]` returns `2`.

Solution 1

a. Compiled languages are translated into a from that can be run directly on a computer's processor. Usually the whole program is translated before it is run.

   Interpreted languages are processed by a higher-level virtual machine. Usually a program is translated on the fly, i.e., a statement is translated and then immediately executed.

b. A function is tail-recursive if there is no operation after the recursive call, that is, no operations are executed after the recursive call terminates. As a consequence, no data need to be stored on the stack that is needed when the recursive call terminates. Hence, different from non-tail-recursive function, for tail-recursive functions the stack does not grow with each recursive call.

c. The expression is syntactically correct. The result is: `2`
   ```
   times
   4
   is
   not
   6
   ```

d. This is a simple echo program. It reads from the standard input and shows the input on the standard output. When the user input is "quit", the program terminates.

e. At the receiver side:

   - Each receive clause will have to match the process ID of the requesting sender (in addition to the content of the message).
   - Each receive clause has to send a response to the sender (instead of/in addition to printing some result).

   At the sender side:

   - After sending a message, the sender has to wait for a response.

f. The function is evaluated in two steps:

   - The first input parameter is applied, i.e., `prod 2`, yielding a partially evaluated function `(\y -> 2 * y)`
   - The partially evaluated function is applied to the second argument, i.e., `(\y -> 2 * y) 4`, yielding `8`

   This is called curried functions.

Solution 2

```
class Fixnum
  def square_root_times
    i = 0
    while i * i < self
      i += 1
      yield
    end
  end
end
```

Solution 3

```
def prime_numbers( n )
  res = []
  number = 2
  count = 1
  while count <= n
    i = 2
    while i <= number
      break if number % i == 0
      i += 1
    end
    if i == number
      res.push(number)
      count += 1
    end
    number += 1
  end
  return res
end
```

Solution 4

```
subst(_, [], _, []).
subst(X, [X|L], Y, [Y,Y|M]) :-
    !,
    subst(X, L, Y, M).
subst(X, [Z|L], Y, [Z|M]) :-
    subst(X, L, Y, M).
```

Solution 5

```
path(X, Y, P) :- path0(X, Y, [], P).

path0(X, X, P, P).
path0(X, Y, P0, P) :-
    arc(X, Z),
    not( member((X,Z), P0) ),
    append(P0, [(X,Z)], P1),
    path0(Z, Y, P1, P).
```

Solution 6

```
-module(accumulator).
-export([loop/0]).

loop() -> loop( 0 ).

loop( Sum ) ->
  receive
    "result" ->
      io:format( "Result ~p~n", [Sum] ),
      loop( Sum );
    "reset" ->
      io:format( "Reset to 0 ~n" ),
      loop( 0 );
    "exit" ->
      io:format("Exit~n");
    Number ->
      io:format( "Increment by ~p~n", [Number] ),
      loop( Sum + Number )
  end.

Pid = spawn(fun accumulator:loop/1).
Pid ! 10.
Pid ! "result".
Pid ! "exit".
```

Solution 7

a. The program computes the length of an array.

b. Tail-recursive version: pass the length of the list seen so far as a second parameter; when the list is empty, the second parameter contains the length of the list.

```
module Tail (
  len
) where

len :: [a] -> Integer -> Integer
len [] x = x
len (h:t) x = len t (x+1)
```

Solution 8

```
module NoOfElem (noOfElem) where

noOfElem ::  Eq a => a -> [a] -> Int
noOfElem x [] = 0
noOfElem x (h:t) = (if h == x then 1 else 0) + noOfElem x t
```