Programming Paradigms
Written Exam (6 CPs)

20.09.2017

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

Exercise	Marks	Achieved
1	20	
2	8	
3	8	
4	10	
5	10	
6	14	
7	16	
8	14	
Total	100	

Exercise 1 (20 marks)

a. (4 marks) Briefly describe the concept of abstract data types and the advantages they introduced with respect to imperative/procedural programming.

b. (4 marks) What does the following Ruby-code print?

```
def like_map(array)
  result = []
  array.each do |element|
    result << (yield element)
  end
  result
end

x = like_map([1, 2, 3]) do |number|
  number * 2
end

print x
```

c. (4 marks) The box model of Prolog execution is a simple way to show the control flow. Briefly sketch and describe the box model.

d. (4 marks) Consider the Haskell function `plus x y = x + y`. How is the function call `plus 20 4` evaluated? What is the name of this evaluation concept?

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

Exercise 2 (8 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). Your program should use at least one code block.

Exercise 3 (8 marks) Write a Ruby class `Animal` with a property

- `kind`: a string that holds the type of the animal

and the following instance methods:

- `eat`: takes a parameter `food` and prints a message that the animal is eating `food`
- `sleep` and `wake`: these two methods do not have any arguments; instead, they will set an instance variable `@state` to the string `"asleep"` and `"awake"`, respectively

Write a second Ruby class `Person` with the following characteristics:

- Inherits from `Animal`
- Automatically sets `@kind` to `"person"`
- Adds 3 new instance variables: `age`, `gender`, `name`
- Overrides the `eat` method so that a person cannot eat a `"person"`

Exercise 4 (10 marks) Write a Prolog program `drop_kth(K, L, R)`, which removes the element at position `K` from the list `L` and returns the resulting list in `R`. For instance, `drop_kth(3, [a,b,c,d,e], R)` succeeds with `R = [a,b,d,e]`.

Exercise 5 (10 marks) The following Prolog knowledge base describes a small social network using the `friend` relation that represents a direct friendship between two persons.

```
friend(tom,tim).
friend(tom,alf).
friend(alf,ann).
friend(alf,joe).
friend(joe,sue).
friend(joe,tim).
friend(sue,ann).
```

Write a predicate `friends_dist(X,Y,D)` which tells whether `X` and `Y` are connected by friendship relations at a distance of `D`. For instance, `friends_dist(tom,Y,2)` succeeds and instantiates `Y=ann` and `Y=joe`.

Exercise 6 (14 marks) Write a Haskell module that exports a function `split`, which splits a list at a given position. The list and the split position are given as input parameter; the function returns a pair consisting of the two parts of the list. For instance, `split [1,2,3,4,5] 2` returns the two parts `([1,2], [3,4,5])`.

Exercise 7 (16 marks) Write a Haskell module that exports a function `diffAB` which takes as input a list and two elements `a`, `b` of the list and returns the difference between the number of occurences of `a` and `b` in the list. For instance, `diffAB [3,4,2,3,3] 3 4` returns `2`.

Exercise 8 (14 marks) Write an Erlang module that exports a function `loop` for a 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 `"sum"` is received, the value of the accumulator is printed out; if `"exit"` is received, the process is stopped; for all other messages, an error message is shown. In all cases, a corresponding message is printed.

Moreover, show the following steps:

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

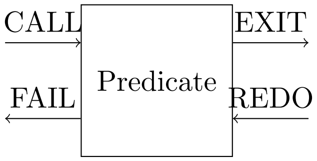
Solution 1

a. The procedural approach in imperative programming was taken further by introducing abstract data types (ADT). In ADTs, everything related to a type is encapsulated in one bundle, most importantly data itself and operations on the data. This is also known as information hiding and has several advantages: data can only be accessed via a specified operations/interface; the actual representation/implementation is hidden and can easily be changed/replaced without affecting the rest of the program; the code becomes more portable.

b. The result is `[2, 4, 6]`. The `like_map()` method takes and array and a code block as arguments. `like_map()` iterates over each element of the array, yields the code block, and appends the result to the result array. `like_map()` behaves like the `Array#map` method.

c. The box model provides a simple way to show the control flow of a Prolog program. A box represents the invocation of a single predicate. The box has four ports (with associated events):

- CALL: The first call of a predicate; control enters into the box
- EXIT: The goal has been proven
- REDO: The system comes back to a goal, trying ot re-satisfy it, i.e., backtracking
- FAIL: The goal/predicate fails



d. The function is evaluated in two steps:

- The first input parameter is applied, i.e., `plus 20`, yielding a partially evaluated function `(\y -> 20 + y)`
- The partially evaluated function is applied to the second argument, i.e., `(\y -> 20 + y) 4`, yielding `24`

This is called curried functions.

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.

Solution 2

```
def prime_numbers( n )
  res = []
  num = 2
  while res.length < n
    isprime = true
    2.upto(num-1) { |i|
      isprime = false if num % i == 0
    }
    res.push(num) if isprime
    num += 1
  end
  return res
end
```

Solution 3

```
class Animal
  @kind
  @state

  def initialize(kind)
    @kind = kind
  end

  def eat(food)
    print "Animal eats: #{food}\n"
  end

  def sleep
    @state = "asleep"
  end

  def wake
    @state = "awake"
  end
end

class Person < Animal
  @age
  @gender
  @name

  def initialize(age,gender,name)
    super("person")
    @age = age
    @gender = gender
    @name = name
  end

  def eat(food)
    print "Animal eats: #{food}\n" if food != "person"
  end
end
```

Solution 4

```
drop_kth( 1, [_|Xs], Xs ).
drop_kth( K, [X|Xs], [X|Ys]) :-
    K > 1,
    K1 is K - 1,
    drop_kth( K1, Xs, Ys ).
```

Solution 5

```
friends_dist( X, Y, 1 ) :- friend( X, Y ).
friends_dist( X, Y, D ) :-
    friend( X, Z ),
    D1 is D - 1,
    friends_dist( Z, Y, D1 ).
```

Solution 6

```
module List (
       split
) where
```

```
split :: [a] -> Int -> ([a], [a])
split xs n
    | n < 0         = ([], xs)
    | n > length xs = (xs, [])
    | otherwise     = split2 ([], xs) n

split2 :: ([a], [a]) -> Int -> ([a], [a])
split2 (xs, ys) 0 = (xs, ys)
split2 (xs, (y:ys)) n = split2 (xs ++ [y], ys) (n - 1)
```

Solution 7

```
module diffAB (
       diffAB
) where

diffAB :: Eq a => [a] -> a -> a -> Int
diffAB xs a b = diffAB2 xs a b 0

diffAB2 :: Eq a => [a] -> a -> a -> Int -> Int
diffAB2 [] a b n    = n
diffAB2 (x:xs) a b n
    | x == a        = diffAB2 xs a b (n+1)
    | x == b        = diffAB2 xs a b (n-1)
    | otherwise     = diffAB2 xs a b n
```

Solution 8

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

loop() -> loop( 0 ).

loop( Sum ) ->
  receive
    "sum" ->
      io:format( "Sum ~p~n", [Sum] ),
      loop( Sum );
    "reset" ->
      io:format( "Reset to 0 ~n" ),
      loop( 0 );
    "exit" ->
```

```
      io:format("Exit~n");
    N when is_number(N) ->
      io:format( "Increment by ~p~n", [N] ),
      loop( Sum + N );
    _ ->
      io:format( "Invalid message~n" ),
      loop( Sum )
  end.

Pid = spawn(fun accumulator:loop/0).
Pid ! 10.
Pid ! "sum".
Pid ! "exit".
```