Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2019-06-05
Sal (1)	TER2(8)
Tid	8-12
Utb. kod	TDDA69
Modul	TENA
Utb. kodnamn/benämning **Modulnamn/benämning**	Data- och programstrukturer Tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Cyrille Berger
Telefon under skrivtiden	013-284023 eller 0767772870
Besöker salen ca klockan	ca. ~~15:00~~ 9 - 9 30
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Antal exemplar i påsen	

TDDA69
Spring 2019
2019-06-05 8-12
Time Limit: 4 hours

Examiner: Cyrille Berger
Tel: 076-777 28 70

This exam contains 4 pages (including this cover page) and 6 questions.
Total of points is 29p, the minimum for passing the exam is 14p, to get a four it is 19p and to get a five it is 24p.

No assistance.

Good luck!

1. (6 points) Programming paradigms and concepts.
   (a) (4 points) Draw a diagram showing the relation between the following programming paradigms:
      - First-order functional programming
      - Functional programming
      - Logic programming
      - Imperative programming
      - Sequential object-oriented programming
      - Declarative concurrent programming

      The relation between those programming paradigms could be (not all of them are necesserary usefull, and some might appear several times in the diagram):
      - +procedure
      - +closure
      - +cell(state)
      - +unification
      - +thread
      - +search
      - +port

      The diagram should be a graph where the nodes are the programming paradigms and the edges are the relations.
   (b) (2 points) Which programming paradigm would you use for a programming language that will primarly be used to query a database? Explain your choice.

2. (2 points) Write a recursive function that uses Newton's method to calculate cube roots.
   Given a guess g_n for the cube root of x an improved guess is given by:

$$g_{n+1} = \frac{1}{3} \cdot (\frac{x}{g^2} + 2 \cdot g_n) \tag{1}$$

   When $|g_{n+1} - g_n| < \epsilon$ the solution is found.
   You can start with $g_0 = x$.
   Only **if**, **else**, arithmetic operators and recursive calls are allowed.

3. (8 points) Environment diagram.

   Assume the expression below is evaluated in the order it is given.

```
1  function f(x)
2  {
3    return h(g)(x+1)(4, 5);
4  }
5  function g(x)
6  {
7    return function(y,z) { return z + (y * x); }
8  }
9  function h(f)
10 {
11   return function(x) { return f(x+3); }
12 }
13 f(5)
```

   (a) (1 point) What will the result be?

   (b) (3 points) Draw a diagram that captures what is going on according to the environment model of evaluation.

   (c) (2 points) Mark the important structures and explain why, and in what order, they are created and (can be) removed.

   (d) (2 points) Use the diagram to show the result of the evaluation.

4. (3 points) Macros.

   What is printed when executing the following code?

```
1  def skipper(f, n=None):
2    if n is None:
3        return lambda n : skipper(f, n)
4    else:
5      if n % 2 == 0:
6        retval = f(n)
7      else:
8        retval = n * skipper(f, n-1)
9
10     return retval
11
12 calls = 0
13
14 @skipper
15 def fact(n):
16   global calls
17   calls += 1
18   if n < 1:
19     return 1
```

```
20      else:
21        return n * fact(n-1)
22
23  print(fact(4))
24  print(calls)
```

5. (3 points) Regular expressions.
   (a) (1 point) Given the following regular expression:

```
1  /(ab+c)*/
```

   Where + is one or more occurence, * is zero or more occurence and () is used for grouping. Which of the following strings matches:

```
1  var a = "abc";
2  var b = "ac";
3  var c = "";
4  var d = "abbbbc";
5  var e = "abbbcabc"
```

   (b) (2 points) Explain how the regular expression is executed, using a diagram.

6. (7 points) Concurrent Programming.
   (a) (1 point) The following class defines an account:

```
1  class Account:
2    def __init__(self, balance):
3      self.balance = balance
4    def withdraw(self, amount):
5      """Withdraw money from the account."""
6        if amount > self.balance:
7            return 'Insufficient funds'
8        self.balance = self.balance - amount
```

   We want to use in a multi-threaded banking system:

```
1  account = Account(100)
2  thread.start_new_thread(Account.withdraw, (account, 20))
3  thread.start_new_thread(Account.withdraw, (account, 25))
4  print(account.balance)
```

   What is the expected result? Explain why with the current implementation the result can be different.

   (b) (2 points) In Python, you can create a mutex with `mutex = threading.Lock()`, acquire the mutex with `mutex.acquire()` and release it with `mutex.release()`. Provide a modification of the `Account.withdraw` function to guarantee that we obtain the correct result

(c) (2 points) A common mistake with mutex is to forget to unlock it. What solution(s) would you implement in a programming language to help developers avoid this problem?

(d) (2 points) Why declarative concurrency (and streams) cannot be used to model client/server applications?