Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2016-06-04
Sal (2)	G35 G37
Tid	8-12
Kurskod	TDDA69
Provkod	TENA
Kursnamn/benämning **Provnamn/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 or 076-777 28 70
Besöker salen ca klockan	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Antal exemplar i påsen	

Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2016-06-04
Sal (2)	G35 G37
Tid	8-12
Kurskod	TDDA69
Provkod	TENA
Kursnamn/benämning **Provnamn/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 or 076-777 28 70
Besöker salen ca klockan	ja
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 2016
2016-06-04 8-12
Time Limit: 4 hours

Examiner: Cyrille Berger
Tel: 076-777 28 70

This exam contains 5 pages (including this cover page) and 6 questions.
Total of points is 39p, the minimum for passing the exam is 19p, to get a four it is 26p and to get a five it is 32p.

No assistance.
Good luck!

1. (7 points) Programming paradigms.
   (a) (3 points) Explain the difference between functional programming and imperative programming.
   (b) (1 point) What is a pure function? Tell if pure and non-pure functions can be used in functional or imperative programming.
   (c) (3 points) For the following applications, select which programming paradigm (between functional or imperative) you would choose to use, and give an explanation of your choices:
      1. querying a database
      2. distributed numerical computations
      3. game

2. (6 points) Rewrite the following code using a recursion:
   (a) (2 points) Write a recursive function *has_digit(k, d)* that test if a number k contains the digit d at least once.

   Examples of use:

```
>>> has_digit(2147, 7)
True
>>> has_digit(2149, 7)
False
>>> has_digit(2747, 4)
True
>>> has_digit(123393, 4)
False
```

   (b) (4 points) Anonymous factorial

   In general, to write a recursive function, the function is given a name using a function definition (`def name`) or with an assignment statement. This allow to refer to the function within its own body.

   The recursive factorial function can be written as a single expression by using a conditional expression.

```
fact = lambda n: 1 if n == 1 else mul(n, fact(sub(n, 1)))
```

This implementation relies on the fact that `fact` has a name, that is referred in the body of the function `fact`.

In this question, you should define the `fact`(orial) function without giving it a name. Write an expression that computes n factorial using only *call expressions*, *conditional expressions*, and *lambda expressions* (no assignment or def statements). And you are **not** allowed to use *make_anonymous_factorial* in your return expression. The *sub* and *mul* functions from the operator module are the only built-in functions required to solve this problem:

```
from operator import sub, mul

def make_anonymous_factorial():
    """Return the value of an expression that computes factorial.

    >>> make_anonymous_factorial()(5)
    120
    """
    return 'YOUR_EXPRESSION_HERE'
```

Replace 'YOUR_EXPRESSION_HERE' in the code above with your implementation of a fully anonymous factorial.

3. (14 points) Environment model.

   (a) (2 points) Expressions can be evaluated in *normal* or *applicative* orders. Explain both orders. Do they always give the same result? If not, give an example that differentiates them.

   (b) (2 points) What are the problems with the *substitution* model and how is it solved by the *environment* model?

   (c) (1 point) Assume the expression below is evaluated in the order it is given.

```
function f(x)
{
  var y = g(2)(-1,1);
  g = h(g)
  return g(x+y)(4, 5);
}
function g(x)
{
  return function(y,z) { return z + (y * x); }
}
function h(f)
{
  return function(x) { return f(x+1); }
}
f(3)
```

What will the result be?

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

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

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

(g) (2 points) Does *f(3)* always return the same value? Explain your answer.

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. (5 points) Stack machines.

   In this question, we use a stack machine with the following instruction set:

   - *PUSH [constant_value]*: push the constant on the stack
   - *POP [number]*: pop a certain numbers of variables from the stack
   - *MUL*: pop two arguments from the stack, push the result of multiplying them
   - *SUB*: pop two arguments from the stack, push the result of subtracting them
   - *EQUAL*: pop two arguments from the stack, push true if they are equal, or false otherwise

- *LOAD [varname]*: push the value of variable
- *DCL [varname]*: declare the variable
- *STORE [varname]*: get the value, store the result and push the value
- *JMP [idx]*: jump to execute instruction at the given index
- *IFJMP [idx]*: pop the value and if true jump to [idx]
- *CALL [arguments]*: pop the function object and call it with the given number of arguments
- *RET*: return from a function call

(a) (2 points) Given the following factorial function:

```
1 var factorial = function(n)
2 {
3   if(n == 0) {
4     return 1
5   } else {
6     return n * factorial(n - 1);
7   }
8 }
```

Write the list of instructions that would define the factorial function on a stack machine with the provided instruction set.

Write the list of instructions that would call the factorial function.

For clarity, you should provide a number for each instruction in your answer, as shown in the following example:

1. LOAD 'k'
2. PUSH '5'
3. MUL
4. JMP 1

(b) (1 point) Explain what happen during a *CALL* instruction and how the *RET* instruction knows where to return.

(c) (2 points) What is the maximum depth of the stack for a call to `function(5)` ? List all the values in the stack.

6. (4 points) Logic Programming.

(a) (1 point) Give the answer(s) to the following query:

```
1 (fact (parent abraham barack))
2 (fact (parent abraham clinton))
3 (fact (parent delano herbert))
4 (fact (parent fillmore delano))
5 (fact (parent fillmore abraham))
6 (fact (parent fillmore grover))
7 (fact (grandparent (parent ?x ?y) (parent ?y ?z)))
8
9 (query (grandparent fillmore ?grandchild))
```

(b) (1 point) Explain how the query is executed.

(c) (2 points) Implement the predicate multiple in Prolog. It should decide whether a list contains the same element two or more times.

```
1 ( multiple ( a b d b c ) ) ;; is true
2 ( multiple ( a b c d ) ) ;; is false
```

You can use the append predicate without defining it.