Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2015-08-20
Sal (1)	G35
Tid	14-18
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 2015
2015-08-20 14-18
Time Limit: 4 hours

Examiner: Cyrille Berger
Tel: 076-777 28 70

This exam contains 7 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. (4 points) Programming paradigms.
   (a) (3 points) Explain the difference between imperative programming and object-oriented programming.
   (b) (1 point) Explain the main differences between weak typing and strong typing.

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:

```
1 from operator import sub, mul
2
3 def make_anonymous_factorial():
4     """Return the value of an expression that computes factorial.
5
6     >>> make_anonymous_factorial()(5)
7     120
8     """
9     return 'YOUR_EXPRESSION_HERE'
```

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

3. (11 points) Intervals, data abstraction.

   Alyssa P. Hacker is designing a system to help people solve engineering problems. One feature she wants to provide in her system is the ability to manipulate inexact quantities (such as measured parameters of physical devices) with known precision, so that when computations are done with such approximate quantities the results will be numbers of known precision.

   Alyssa's idea is to implement interval arithmetic as a set of arithmetic operations for combining "intervals" (objects that represent the range of possible values of an inexact quantity). The result of adding, subtracting, multiplying, or dividing two intervals is itself an interval, representing the range of the result.

   Alyssa postulates the existence of an abstract object called an "interval" that has two endpoints: a lower bound and an upper bound. She also presumes that, given the endpoints of an interval, she can construct the interval using the data constructor interval. Using the constructor and selectors, she defines the following operations:

```
def str_interval(x):
    """Return a string representation of interval x.

    >>> str_interval(interval(-1, 2))
    '-1 to 2'
    """
    return '{0} to {1}'.format(lower_bound(x), upper_bound(x))

def add_interval(x, y):
    """Return an interval that contains the sum of any value in
    interval x and any value in interval y.

    >>> str_interval(add_interval(interval(-1, 2), interval(4, 8)))
    '3 to 10'
    """
    lower = lower_bound(x) + lower_bound(y)
    upper = upper_bound(x) + upper_bound(y)
    return interval(lower, upper)

def mul_interval(x, y):
    """Return the interval that contains the product of any value
    in x and any value in y.

    >>> str_interval(mul_interval(interval(-1, 2), interval(4, 8)))
    '-8 to 16'
    """
    p1 = lower_bound(x) * lower_bound(y)
    p2 = lower_bound(x) * upper_bound(y)
    p3 = upper_bound(x) * lower_bound(y)
    p4 = upper_bound(x) * upper_bound(y)
    return interval(min(p1, p2, p3, p4), max(p1, p2, p3, p4))
```

(a) (1 point) Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Define the constructor and selectors in terms of two-element lists:

```
def interval(a, b):
    """Construct an interval from a to b."""
    "*** YOUR CODE HERE ***"

def lower_bound(x):
    """Return the lower bound of interval x."""
    "*** YOUR CODE HERE ***"

def upper_bound(x):
    """Return the upper bound of interval x."""
    "*** YOUR CODE HERE ***"
```

(b) (2 points) Alyssa needs an implementation of the division and she ask you to provide

one. She suggests that the division can be implemented, by multiplying by the reciprocal of y. Ben Bitdiddle, an expert systems programmer, will be reviewing your code and wan that you that it is not clear what it means to divide by an interval that spans zero. Make sure to add an assert statement to your code to ensure that no such interval is used as a divisor:

```
def div_interval(x, y):
    """Return the interval that contains the quotient of any value in x
    divided by any value in y.

    Division is implemented as the multiplication of x by the reciprocal
    of y.

    >>> str_interval(div_interval(interval(-1, 2), interval(4, 8)))
    '-0.25 to 0.5'
    >>> str_interval(div_interval(interval(4, 8), interval(-1, 2)))
    AssertionError
    """
    "*** YOUR CODE HERE ***"
```

(c) (2 points) Using reasoning analogous to Alyssa's, define a subtraction function for intervals:

```
def sub_interval(x, y):
    """Return the interval that contains the difference between any value in x
    and any value in y.

    >>> str_interval(sub_interval(interval(-1, 2), interval(4, 8)))
    '-9 to -2'
    """
    "*** YOUR CODE HERE ***"
```

(d) (3 points) After considerable work, Alyssa P. Hacker delivers her finished system. Several years later, after she has forgotten all about it, she gets a frenzied call from an irate user, Lem E. Tweakit. It seems that Lem has noticed that the formula for parallel resistors can be written in two algebraically equivalent ways:

```
par1(r1, r2) = (r1 * r2) / (r1 + r2)
```

or

```
par2(r1, r2) = 1 / (1/r1 + 1/r2)
```

He has written the following two programs, each of which computes the parallel_resistors formula differently:

```
def par1(r1, r2):
    return div_interval(mul_interval(r1, r2), add_interval(r1, r2))

def par2(r1, r2):
    one = interval(1, 1)
    rep_r1 = div_interval(one, r1)
    rep_r2 = div_interval(one, r2)
    return div_interval(one, add_interval(rep_r1, rep_r2))
```

Lem complains that Alyssa's program gives different answers for the two ways of computing. This is a serious complaint:

```
# These two intervals give different results for parallel resistors:
a = make_center_percent(1, 1)
b = make_center_percent(2, 1)
print(str_interval(par1(a, b)), '!=', str_interval(par2(a, b)))
```

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that the problem is multiple references to the same interval.

The Multiple References Problem: a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated.

Thus, she says, par2 is a better program for parallel resistances than par1. Is she right? Why?

(e) (3 points) Write a function quadratic that returns the interval of all values f(t) such that t is in the argument interval x and f(t) is a quadratic function

```
f(t) = a*t*t + b*t + c
```

Make sure that your implementation returns the smallest such interval, one that does not suffer from the multiple references problem.

Hint: the derivative f'(t) = 2*a*t + b, and so the extreme point of the quadratic is -b/(2*a):

```
def quadratic(x, a, b, c):
    """Return the interval that is the range of the quadratic defined by
    coefficients a, b, and c, for domain interval x.

    >>> str_interval(quadratic(interval(0, 2), -2, 3, -1))
    '-3 to 0.125'
    >>> str_interval(quadratic(interval(1, 3), 2, -3, 1))
    '0 to 10'
    """
    "*** YOUR CODE HERE ***"
```

4. (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.

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:

```
var factorial = function(n)
{
  if(n == 0) {
    return 1
  } else {
    return n * factorial(n - 1);
  }
}
```

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

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. (5 points) Concurrent Programming.

(a) (1 point) The following class defines an account:

```
class Account:
  def __init__(self, balance):
    self.balance = balance
  def withdraw(self, amount):
    """Withdraw money from the account."""
      if amount > self.balance:
          return 'Insufficient funds'
      self.balance = self.balance - amount
```

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

```
account = Account(100)
thread.start_new_thread(Account.withdraw, (account, 20))
thread.start_new_thread(Account.withdraw, (account, 25))
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?