Gamla tentor på delkurs Objektorienterad Programmering

Omtentamen 10. april 2003

Tillåtna hjälpmedel: Litteratur, egna anteckningar

Detta dokument innehåller 9 tentamensfrågor från tidigare kurstilfällen, inkl. kommentarer till tre ov dessa (ordinarie tentamen från 28 maj 2001). Kursen hade då ingen separat tentamen för funktionsprogrammering, så att dessa tentor inte är någon modell för kursens nuvarande form. Speciellt hör uppgifter 1, 4 och 7 inte hemma i en kurs i objektorienterad programmering.

Tentamen består av 3 uppgifter, varje uppgift kan ge 10 poäng. Deluppgifternas poäng anges inom [hakparenteser]. Observera att poänggivningen inte nödvändigtvis avspeglar deluppgiftens svårhetsgrad. Uppgifterna kan besvaras på svenska eller engelska.

Behandla högst en uppgift per papper (det går bra att behandla deluppgifter på samma papper). Skriv bara på ena sidan och markera varje sida med dina initialer. Skriv läsligt.

► a)[0] Hur många uppgifter får man behandla per papper?

► b)[0] Får man skriva på baksidan?

Det går bra att referera till kursböckerna av Schmidt, Hansen och Rischel eller Rosen. Referenser måste ha formen [S: *något*], [HR: *något*] eller [R: *något*], där *något* är ett avsnittsnummer, en kodbit, en räknad uppgift, etc. Skriv till exempel "genom att använda metoden **validate** från [S: Figure 7 i Chapter 6]". Vill du referera till något annat, måste du ange bokens titel, författere, och förlag. Skriv till exempel "genom att använda metoden *validate* från [Barry Cornelius: Understanding Java, Addison–Wesley 2001, p. 254]".

Ibland hänger vissa uppgifter ihop. Observera att det går bra att referera till svaret på tidigare uppgifter, även om du inte har löst dem.

Det går bra att deklarera och använda ytterligare funktioner om det behövs.

Exercise 1

This considers expressions over a variable x, for example $((x * x) * (x + 4))$ or $(x + 1)$. To be precise, an Object is an **ExpressionOverX** if

1. it is a constant,
2. it is the variable x,
3. it is a structure (e_1 op e_2), where e_1 and e_2 are **ExpressionOverX** and 'op' is either '+' or '$*$.'

We can represent an **ExpressionOverX** is Java using the following classes:

```
abstract class ExpressionOverX { }

class Constant extends ExpressionOverX
{ int  value;
  Constant(int v) { value= v; }
}

class X extends ExpressionOverX { }

class Expr extends ExpressionOverX
{ ExpressionOverX e1, e2;
  char  op;
  Expr(ExpressionOverX left, char  c, ExpressionOverX right)
  { e1= left;
    op= c;
    e2= right
  }
}
```

For example, the expression '$(x + 2)$' is represented by the following ExpressionOverX

```
new Expr(new X(), '+', new Constant(2))
```

▶ a)[1] Construct an object of class **ExpressionOverX** that represents the expression $((x * x) * (x + 2))$.

▶ b)[3] Add a method

```
int  evaluateAt(int val)
```

that evaluates this expression given a value for x. For example, if e is the object you constructed in the previous exercise, then

e.evaluateAt(3)

should return 45 (because $((3 * 3) * (3 + 2)) = 45$).

It is well-known that we can *find the derivative of* (sv. *derivera*, *avleda*) an expression using the following rules,

- derivative(k)= 0, for any constant k,
- derivative((x))= 1,

- derivative($f+g$)= derivative(f)+derivative(g),
- derivative($f*g$)= derivative(f)*g + f*derivative(g).

For example,

$$\text{derivative}(((2+x)*18)) = (((0+1)*18)+((2+x)*0)).$$

Obviously, the latter expression can be simplified using high-school algebra to 18, but we will not be concerned with simplification in this question.

► c)[3] Write a method

```
static ExpressionOverX derivative(ExpressionOverX e)
```

that computes the derivative of a given ExpressionOverX.

En expression is *linear* if its derivative is constant, i.e., does not depend on x. For example, the expression $3*x+5$ is linear (its derivative is 3), but the expression $3*x*x+5$ is not (its derivative is $3*x$). The following method checks is a given expression contains any occurences of the variable x:

```
static boolean containsAnX(ExpressionOverX e)
{ if (e instanceof Const) return false;
  else if (e instanceof X) return true;
  else { ExpressionOverX left= ((Expr) e).e1;
         ExpressionOverX right= ((Expr) e).e2;
         return containsAnX(left) || containsAnX(right);
       }
}
```

► d)[3] Professor Precipitat claims that the following expression decides if a given expression e is linear:

```
!containsAnX(derivative(e))
```

Is the good professor right? If yes, give a proof; if no, give a counterexample.

Exercise 2

The following program computes the remainder[1] of two numbers. For example, the remainder of dividing 27 by 4 is 3, because $27 = 6 * 4 + 3$.

The body of the while-loop in line 6 is missing; line numbers have been added to facilitate reference.

```
1. /** Compute the remainder of two given integers.
0.  * @param $a$ an integer
0.  * @param $d$ a positive integer
0.  * @return $r$ such that $\exists q\colon a = dq + r$ and $0 \leq r < d$. */
1. static int remainder(int $a$, int $d$)
2. { int $r$= $a$;
3.   int $q$= 0;
4.   while ($r$ >= $d$)
5.     // loop invariant: $a = dq + r \wedge r \geq 0$
6      { loop body
7.     }
8.   return $r$
9. }
```

▶ a)[1] Is the precondition for a sufficient? If yes, give a proof. If no, give a counterexample and strengthen the precondition for a.

▶ b)[3] Assuming that the body of the **while**-loop maintains the invariant, show that the method is partially correct.

▶ c)[3] Write the body of the **while**-loop so that it maintains the invariant and terminates.

▶ d)[3] Change the loop condition in line 4 to

```
4'. while ($d * (q + 1) <= a$)
```

Is the invariant in line 5 sufficient to prove partial correctness for the new programme? If yes, do it. If no, construct a new invariant and prove partial correctness. (*Remark*: You may change the loop body if you want, but you don't need to.)

[1] rest ved heltalsdivision

Exercise 3

Consider the following method

```
static int throwDie()
{ return (int) (Math.random()*6 + 1); }
```

Recall that the 'random()' method of class 'Math' returns a random float f with $0 \leq f < 1$, so 'throwDie' returns a random number from 1 to 6, like a six-sided die (*en tärning*).

We now invoke throwDie from a JButton and show the result in a JFrame. Recall that the ActionListener interfaces promises to implement 'actionPerformed', which will be invoked when the relevant JButton is pressed.

```
1. JButton B= new JButton();
2. B.addActionListener(new new DieResult());

3. public class DieResult extends JFrame implements ActionListener
   { int number;
     public DieResult()
     { setVisible(true); }
     public void paint(Graphics g)
     { g.drawString("Die shows "+number); }
     public void actionPerformed(Event e)
     { number= throwDie();
       repaint();
     }
     throwDie goes here
   }
```

► a)[1] What will the DieResult frame show before the user pushes B? Add a relevant (but simple) invariant to the class DieResult to improve the situation and change the code accordingly.

► b)[4] Rewrite the above application so that it corresponds to an MVC architecture. Especially, introduce a new class 'Die' as the model. You may not change lines 1-3, and don't have to separate the controller from the view.

► c)[1] Draw a class diagram for you answer to the previous question.

► d)[3] Change the architecture so that the button B becomes the controller and DieResult becomes the view (with no controller duties). *Hint:* Define a new class that extends JButton.

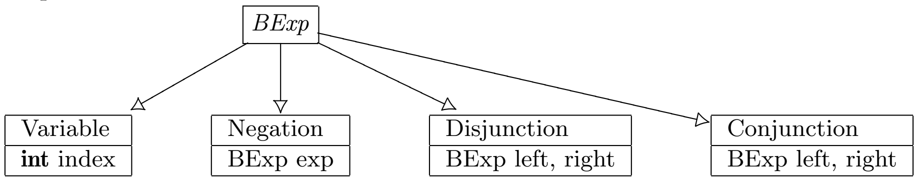
► e)[1] Draw a class diagram for you answer to the previous question.

Exercise 4 (5 May 2001)

This exercise considers Boolean expressions over n variables $x_0, \ldots, x_{n-1}$, like $x_1 \vee (x_0 \wedge \neg x_2)$. To be precise, an Object is a BExp (for *Boolean expression*) if it is

1. a variable, like x_0 or x_6,
2. a negation ('not') $\neg e$, where e is a BExp,
3. a disjunction ('or') $e_1 \vee e_2$, where e_1 and e_2 are BExp's,
4. a conjunction ('and') $e_1 \wedge e_2$, where e_1 and e_2 are BExp's.

We can represent a Boolean expression in Java using an abstract class for BExp and extensions for each of the four cases:



For example, the Boolean expression $x_{15} \wedge \neg x_2$ would be constructed like this:

new Conjunction(**new** Variable(15), **new** Negation(**new** Variable(2)))

To be quite precise, here is the Java code for most of the classes, with their constructors, in all boring detail.

```
public abstract class BExp { }

public class Variable extends BExp
{ int  index;
  public Variable(int i) { index= i; }
}

public class Negation extends BExp
{ BExp exp;
  public Negation(BExp e) { exp= e; }
}

public class Conjunction extends BExp
{ BExp left, right;
  public Conjunction(BExp e1, BExp e2)
  { left= e1;
    right= e2;
  }
}
```

▶ a)[1] Write the missing class 'Disjunction'.

▶ b)[1] Create an object corresponding to the expression $x_1 \vee (x_0 \wedge \neg x_2)$.

Given an array x of Boolean values, we can evaluate a BExp by setting $x_i = x[i]$. For example, if $x = [\textbf{false}, \textbf{true}, \textbf{true}]$ then $x_0 = \textbf{false}$ and $x_1 = x_2 = \textbf{true}$, so the expression $x_1 \vee (x_0 \wedge \neg x_2)$ evaluates to **true** because

$$\begin{aligned} x_1 \vee (x_0 \wedge \neg x_2) &= \textbf{true} \vee (\textbf{false} \wedge \neg\textbf{true}) \\ &= \textbf{true} \vee (\textbf{false} \wedge \textbf{false}) = \textbf{true} \vee \textbf{false} = \textbf{true} \end{aligned}$$

▶ c)[4] Add a method

> **boolean** evaluate(**boolean** [ ] x)

to BExp that evalues the expression with respect to the values in x. For example, if e is the BExp corresponding to $x_1 \vee (x_0 \wedge \neg x_2)$ then e.evaluate([**false**, **true**, **true**]) should return **true**.

A Boolean Expression is on *normal form* if negations are only used direcly on variables. For example, $x_1 \vee (x_0 \wedge \neg x_2)$ is on normal form (the only negated expression is the variable x_2), but $x_1 \vee \neg(x_0 \wedge x_2)$ is not (because negation is used on $(x_0 \wedge x_2)$, which is a conjunction, not a variable).

▶ d)[2] Write a static method

> **static boolean** isOnNormalForm(BExp e)

that decides if the given expression is on normal form.

▶ e)[2] (Harder) Write a method

> BExp toNormalForm(BExp e)

that transforms a boolean expression to an equivalent Boolean expression on normal form. *Hint:* Recall $\neg(\neg x) = x$ and De Morgan's laws: $\neg(x \vee y) = \neg x \wedge \neg y$ and $\neg(x \wedge y) = \neg x \vee \neg y$.

Exercise 5 (5 May 2001)

The *integer square root* (isr) of a number $a \geq 1$ is the positive integer x satisfying

$$x^2 \leq a < (x+1)^2.$$

For example, isr(25) = 5, isr(27) = 5, and isr(35) = 5.

▶ a)[1] Find isr(1), isr(2), and isr(15).

Here is a simple algorithm that finds integer square roots:

```
/** Computes the integer square root of a given number.
* @param $a$ an integer $\geq 1$
* @return isr($a$).
*/
static int isr(int a)
{ int $x$=1;
  while ($(x+1)*(x+1) \leq a$)
        // loop invariant: $x^2 \leq a$
        { loop body }
  return $x$;
}
```

▶ b)[2] Write the loop body so that it satisfies the invariant and terminates (you do not have to prove that.) Your algorithm may take time $O(\sqrt{a})$ (you do not have to prove that).

▶ c)[4] Assuming that the loop body satisfies the invariant, prove that isr is partially correct.

Here is a faster algorithm for the same problem that runs in time $\Theta(\log a)$ (in comparison, the algorithm you probably found in question b probably runs in time $\Theta(\sqrt{a})$).

```
/** Computes the integer square root of a given number.
* @param $a$ a positive integer $\geq 1$
* @return isr($a$).
*/
static int fastIsr(int a)
{ int $x$= 1;
  int $y$= $a+1$;
  while ($x+1 \neq y$)
        { int $d$= $(x+y)/2$;
          if ($d*d \leq a$) $x$= $d$;
          else $y$= $d$;
        }
  return $x$;
}
```

▶ d)[3] (Harder) Prove that fastIsr is partially correct. Explain briefly why the algorithm terminates (don't give a proof).

Exercise 6 (5 May 2001)

The `filter` function in Haskell is defined like this (see [T, Chap. 9.2]):

```
filter:: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x        = x : filter p xs
  | otherwise =     filter p xs
```

We will implement a similar mechanism in Java. First, a simple example.

Let w be a ConsList (see [S, Chap. 12.6]) whose objects are known to be Strings. We want to filter out all strings that don't start with 'T'. For example, if w represents the list

[“James”, “Tiberius”, “Kirk”, “Tom”, “Paris”, “Tuvok”],

the we want the filtered result to be the ConsList

[“Tiberius”, “Tom”, “Tuvok”].

The method can be written using a structural recursion pattern:

```
Conslist filterT(ConsList w)
{ ConsList answer;
  if (w instanceof  Nil)
      { answer= ···; }
  else { Object h= ((Cons) w).head();
         ConsList t= ((Cons) w).tail();
         ConsList tail_answer= ···;
         String headString= (String) h;
         if (headString.startsWith('T'))
            ⋮
       }
  return answer;
}
```

► a)[3] Complete the method filterT.

► b)[2] Write filterT as a Haskell function. You *must* use Haskell's `filter` function (defined above) and a property

```
startsWithT :: String -> Bool
```

To handle filterings in Java with the same generality as in Haskell, we introduce an interface

```
interface Filter
{ boolean remainsInList(Object o); }
```

to handle the selection—remainsInList(o) shall return **true** for all objects that are to remain in the list. For example, here is a filter for removing Strings that don't start with 'T':

```
class  TFilter implements Filter
{ boolean remainsInList(Object o)
  { return  (((String) o).startsWith('T')); }
}
```

▶ c)[1] Implement a **class** EvenFilter **implements** Filter, for removing odd numbers[2] from a list of Integers.

▶ d)[4] Write a method

ConsList filterList(ConsList *w*, Filter *f*)

that filters the list *w* using the filter *f*. For example,

filterList(*w*, **new** TFilter())

should give the same result as filterT(*w*).

[2]udda tal

Comments for ex 6–8 (exam 010528)

1c The nicest way is to add an abstract method

```
boolean evaluate(boolean [] x);
```

to BExp and then add a (concrete) method to each of the subclasses. Each of these methods is very short. Here, for example, is the method added to class Negation:

```
boolean evaluate(boolean [] x)
{ return !exp.evaluate(x); }
```

1d Here is a straightforward solution using the structural recursion pattern (and lots of casting).

```
static boolean isOnNormalForm(BExp e)
{ if (e instanceof Variable) return true;
  else if  (e instanceof Negation)
    return (((Negation) e).exp instanceof Variable);
  else if (e instanceof Disjunction)
    return isOnNormalForm(((Disjunction) e).left) &&
           isOnNormalForm(((Disjunction) e).right);
  else
    return isOnNormalForm(((Conjunction) e).left) &&
           isOnNormalForm(((Conjunction) e.right);
}
```

1e Check if your solution works for $\neg(\neg x \vee y)$, this catches a popular mistake.

2b *Read the exercise* – you are *not* asked to prove invariance. Here is the complete answer: "$x = x + 1;$ ".

2c *Read the exercise* – you are *not* asked to prove invariance.

2d This exercise contains three parts: (i) find a good invariant and prove that it entails partial correctness, (ii) prove that the loop body satisfies the invariant, (iii) give a termination argument. Such an argument might be, "The value of $y - x$ decreases in every iteration and never reaches 0".

3b Here is an (correct) implementation of `filterT` in Haskell that earns you 0 points:

```
filterT [] = []
filterT (x:xs)
  | startsWithT x = x : filterT xs
  | otherwise     =     filterT xs
```

This answer is *wrong* because it doesn't use the `filter` function.

3d Probably inspired by the pattern for filterT, many students handed in a solution that contained the line

```
String headString= (String) h;
```

This is a mistake and would produce a type error. You shouldn't make any extra asssumptions about the type of w.

Exercise 7 (29 August 2001)

This exercise studies genealogical trees (*stamträd*). A *person* has a name and zero or more children, who are themselves persons.

```
public class Person
{ String  name;
  Person[ ] children;
  public Person(String s, Person[ ] p)
  { . . . }
}
```

▶ a)[1] Finish the constructor for Person.

▶ b)[1] Create an object corresponding to the Person 'Finwë', who has two children 'Finarfin' and 'Fingolfin'.

▶ c)[2] Extend Person with a method

int getNumGrandchildren()

that returns the number of a Person's grandchildren.

A person is *lonely* if he has exactly one child.[3]

▶ d)[3] Write a method

static boolean hasLonelySuccessor(**Person** *p*)

that returns 'true' if and only if there is a lonely person in *p*'s genealogical tree.

▶ e)[3] Write a method

static void makeHappy(**Person** *p*)

that adds a new child (called 'TV') to every lonely person in *p*'s genealogical tree.

[3]persons with no children at all are assumed to be very young or have full-time jobs, and are not considered lonely in this exercise

Exercise 8 (29 August 2001)

Here is a simple algorithm than raises a number to a given power. For example, on input $a = 3$ and $n = 11$ it will compute $3^{11} = 177147$.

```
    /** Raises a number to a power.
    * @param a real number
    * @param n ≥ 1, integer
    * @return a^n.
    */
1.  static float power(float a, int n)
2.  { int i= 1;
3.    float b= a;
3.    while (i != n)
4.          // loop invariant: b = a^i
5.          { loop body }
6.    return b;
7.  }
```

► a)[2] Write the loop body so that it satisfies the invariant and terminates (you do not have to prove that.) The loop body may not invoke any other methods (like Math.pow for example).

► b)[2] Assuming that the loop body satisfies the invariant, prove that the method 'power' is partially correct.

► c)[1] Are the preconditions on n necessary? If yes, give test values for n that show this. If no, replace the precondition by a weaker one.

Here is a faster algorithm for the same problem that runs in time $\Theta(\log n)$.

```
    /** Raises a number to a power.
    * @param a, real number
    * @param n ≥ 0, integer
    * @return a^n.
    */
1.  static int fastPower(float a, int n)
2.  { float p= 1;
3.    float q= a;
4.    int i= n;
5.    while (i > 0)
6.          { if (i%2 == 1)   { p= p * q; }
7.            q= q * q;
8.            i= i/2;    // integer division!
9.          }
10.   return p;
11. }
```

Remember that '%' is Java's modulo operation, so that line 6 simply means 'if i is odd then...'. Also remember that '/' is integer division, so that line 9 computes $\lfloor i/2 \rfloor$.

► d)[1] Run the algorithm on $a = 2$, $n = 11$. Show the values of p, q, and i or each iteration.

- ▶ e)[1] Professor Precipitat suggests to test this algorithm for $a = 1$ and $n = 32$. Criticise the good professor and suggest a better pair of values (you may still perform only one test!).
- ▶ f)[3] (Hard) Prove that fastPower is partially correct. This involves finding a good invariant. Explain briefly why the algorithm terminates (don't give a proof).

Exercise 9 (28 August 2001)

The `foldr` function in Haskell is defined like this (see [T, Chap. 9.3]):

```
foldr:: (a -> a -> a) -> a -> [a] -> a
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

We will implement a similar mechanism in Java. First, a simple example.

Let w be a ConsList (see [S, Chap. 12.6]) whose objects are known to be Strings. We want to construct the string consisting of all first letters. For example, if w represents the list

[“Seven”, “Paris”, “Odo”, “Cochran”, “Kirk”],

then we want the result to be “SPOCK”. If w is empty then we return the empty string.

The method can be written using a structural recursion pattern:

```
Conslist firstLetters(ConsList w)
{ ConsList answer;
  if (w instanceof Nil)
      { answer= ···; }
  else { Object h= ((Cons) w).head();
         ConsList t= ((Cons) w).tail();
         ConsList tail_answer= ···;
         String headString= (String) h;
         ⋮
      }
  return answer;
}
```

► a)[4] Complete the method firstLetters.

► b)[2] Write firstLetters as a Haskell function. You *must* use Haskell's `foldr` function (defined above).

To handle folding in Java with the same generality as in Haskell, we introduce an interface

```
interface FoldableOperation
{ Object combine(Object o1, Object o2); }
```

The idea is that combine(o_1, o_2) shall perform the binary operation to be folded into the list. For example, here is a FoldableOperation for concatenating the first letters of two strings:

```
class FirstLetters implements FoldableOperation
{ Object combine(Object o1, Object o2)
  { String s1= (String) o1;
    String s2= (String) o2;
    return s1.substring(0,1) + s2.substring(0,1);
  }
}
```

► c)[1] Implement a

> **class** Multiply **implements** FoldableOperation

for multiplying the numbers in a list of Integers. (*Careful:* use Integers, not **int**'s.)

► d)[3] Write a method

> ConsList fold(ConsList w, Object s, FoldableOperation f)

that folds f into the list w. For example,

> filterList(w, `""`, **new** FirstLetters())

should give the same result as filterT(w). Also, if the list v corresponds to the Integers 5,2, and 3, then

> filterList(v, **new** Integer(1), **new** Multiply())

should return the Integer 30.