Skriftlig eksamen i Datalogi

Modul 1

Sommer 1998

Opgavesættet består af 4 opgaver, der ved bedømmelsen tillægges følgende vægte:

Opgave 1	24%
Opgave 2	35%
Opgave 3	15%
Opgave 4	26%

Alle sædvanlige hjælpemidler er tilladt. Der må ikke medbringes datamaskine. I tilfælde af unøjagtigheder i opgaveteksterne forventes det, at deltagerne selv præciserer besvarelsernes forudsætninger.

Opgavesættet består af en forside og 11 paginerede sider. Kontroller at din kopi er fuldstændig.

Opgave 1: Objektorienteret programmering (24%)

Nedenfor er vist et udkast til en samling Java-klasser, der er beregnet til katalogisering af dyr. Klassen `Animal` er en fælles overklasse for de to klasser `Carnivore` og `Herbivore`, der repræsenterer henholdsvis mængden af kødædende og planteædende dyr.

```
class Animal {
    Animal(String n) {
        /* Kode ikke medtaget. Se spørgsmål 1.1 */
    }

    static void printAnimals() {
        /* Kode ikke medtaget. Se spørgsmål 1.3 */
    }

    static Animal firstAnimal;

    String name;
    Animal next;
    public String toString() { return name; }
}

class Herbivore extends Animal {
    Herbivore(String n, int g) {
        /* Kode ikke medtaget. Se spørgsmål 1.2 */
    }

    int grassNeeded;
}

class Carnivore extends Animal {
    Carnivore(String n, int m) {
        /* Kode ikke medtaget */
    }

    int meatNeeded;
}
```

Hvert `Animal`-objekt er forsynet med en tekststreng `name`, der angiver dyrets navn. Referencen `next` benyttes til opbevaring af `Animal`-objekterne i en envejsliste, idet den statiske reference `firstAnimal` peger på det forreste objekt i denne liste, og `next` angiver objektets efterfølger i listen.

Spørgsmål 1.1 Programmér en konstruktør for klassen `Animal`, som tildeler `name` en værdi og indsætter det genererede objekt forrest i envejslisten.

For hvert planteædende dyr (`Herbivore`-objekt) angives dets daglige fødebehov ved hjælp af attributten `grassNeeded`.

Spørgsmål 1.2 Programmér en konstruktør for klassen `Herbivore`, som tildeler værdier til `name` og `grassNeeded`, og som indsætter objektet i envejslisten ved hjælp af konstruktøren for `Animal`.

Spørgsmål 1.3 Programmér metoden `printAnimals`, der gennemløber envejslisten af dyr og udskriver deres navne.

Spørgsmål 1.4 Programmér en metode `printHerbivores`, der kun udskriver navnene på de af dyrene i envejslisten, der er planteædere (`Herbivore`-objekter).

Klassen `Herbivore` benyttes nu som overklasse for klassen `Giraffe`, der repræsenterer mængden af giraffer. Attributten `neckLength` angiver halslængden for en giraf.

```
class Giraffe extends Herbivore {
    Giraffe(String n, int g, double nL) {
        /* Kode ikke medtaget */
    }

    double neckLength;
}
```

Lad der være givet følgende erklæringer:

```
Animal a;
Herbivore h;
Carnivore c;
Giraffe g1, g2;
```

Spørgsmål 1.5 Antag at objekterne `a`, `h`, `c`, `g1` og `g2` eksisterer. Hvilke af følgende sætninger vil da give fejl under oversættelsen?

(1) `a = g1;`
(2) `a = h;`
(3) `g1 = a;`
(4) `g1 = g2;`
(5) `g1 = (Giraffe) h;`
(6) `g1 = (Carnivore) h;`
(7) `g1 = (Giraffe) c;`

Nedenfor ses et interface, `Sortable`, samt en klasse `Sort` til sortering af en tabel af `Sortable`-objekter i stigende orden.

```
interface Sortable {
    boolean lessThan(Sortable s);
}

class Sort {
    static void selectionSort(Sortable a[]) {
        for (int i = 0; i < a.length - 1; i++) {
            int min = i;
            for (int j = i+1; j < a.length; j++)
                if (a[j].lessThan(a[i]))
                    min = j;
            Sortable temp = a[min];
            a[min] = a[i];
            a[i] = temp;
        }
    }
}
```

Spørgsmål 1.6 Angiv hvilke ændringer der skal foretages i klassen `Giraffe` på side 2, hvis dette `interface` skal kunne benyttes til at sortere en tabel af `Giraffe`-objekter i stigende orden efter deres halslængde.

Opgave 2: Parsetræer (35%)

Spørgsmål 2.1 Omskriv nedenstående udtryk til omvendt polsk (postfix) notation.

```
2 * (5 - 3) + 1
```

Et parsetræ for et udtryk er et binært træ, der opbygges ved følgende simple rekursive regel: "Lad operatoren være rod. Lad træet, der svarer til den venstre operand, være venstre undertræ for roden, og lad træet, der svarer til den højre operand være højre undertræ for roden".

Spørgsmål 2.2 Tegn et parsetræ for udtrykket ovenfor.

På næste side ses et udkast til en samling Java-klasser, der skal benyttes til at repræsentere udtryk i form af parsetræer. Som operander tillades heltal. Som operatorer tillades `+`, `-` og `*` (addition, subtraktion og multiplikation).

Spørgsmål 2.3 Programmér en Java-klasse, `Div`, som tillader brugen af operatoren `/` (division).

Spørgsmål 2.4 Programmér en kodestump i Java, der benytter klasserne til at oprette et parsetræ for udtrykket i spørgsmål 2.1.

Spørgsmål 2.5 Lad `root` pege på roden i dette parsetræ. Hvad udskrives da ved udførelse af følgende sætninger?

```
root.print();
System.out.println();
System.out.println("Value = " + root.eval());
```

```
abstract class Node {
    abstract int eval();
    abstract void print();
}

abstract class Operator extends Node {
    Node left, right;
    void print() {
        System.out.print("(");
        left.print();
        System.out.print(this);
        right.print();
        System.out.print(")");
    }
}

class Plus extends Operator {
    Plus(Node l, Node r) { left = l; right = r; }
    int eval() { return left.eval() + right.eval(); }
    public String toString() { return "+"; }
}

class Minus extends Operator {
    Minus(Node l, Node r) { left = l; right = r; }
    int eval() { return left.eval() - right.eval(); }
    public String toString() { return "-"; }

}

class Mult extends Operator {
    Mult(Node l, Node r) { left = l; right = r; }
    int eval() { return left.eval() * right.eval(); }
    public String toString() { return "*"; }
}

abstract class Operand extends Node {
    void print() { System.out.print(this); }
}

class Constant extends Operand {
    int value;
    Constant(int v) { value = v; }
    int eval() { return value; }
    public String toString() {
        return Integer.toString(value);
    }
}
```

Tilladte udtryk kan defineres ved hjælp af nedenstående grammatik.

```
<expression> ::= <term> |
                 <term> + <expression> |
                 <term> - <expression>

<term> ::= <factor> |
           <factor> * <term> |
           <factor> / <term>

<factor> ::= <constant> |
             ( <expression> )
```

`<constant>` er en sekvens af cifre (og repræsenterer dermed ikke-negative heltal).

Grammatikken kan benyttes næsten direkte til at programmere en metode til at undersøge, om en tekststreng indeholder et lovligt udtryk (`expression`). I klassen `Parser` nedenfor er implementeret en sådan metode, `parse`. Metoden syntaksanalyserer den tekststreng, der overføres som parameter. Hvis tekststrengen repræsenterer et lovligt udtryk, opbygges og returneres et parsetræ for udtrykket; i modsat fald afbrydes programafviklingen med en fejludskrift.

```
class Parser {
    static final int PLUS = 0, MINUS = 1, MULT = 2,
                     DIV = 3, LPAR = 4, RPAR = 5,
                     CONST = 6, EOS = 7;
    int token;
    int value;
    StringTokenizer str;

    Node parse(String s) {
        str = new StringTokenizer(s, "+-*/() ", true);
        getToken();
        return expression();
    }

    Node expression() {
        Node t = term();
        while (token == PLUS || token == MINUS)
            if (token == PLUS)
                { getToken(); t = new Plus(t, term()); }
            else
                { getToken(); t = new Minus(t, term()); }
        return t;
    }

    Node term() { /* se spørgsmål 2.7 */ }



                                          // fortsættes på næste side
```

```
    Node factor() {
        Node t = null;
        if (token == CONST)
            t = new Constant(value);
        else if (token == LPAR) {
            getToken();
            t = expression();
            if (token != RPAR)
                error("missing right paranthesis");
        }
        else
            error("illegal factor " + token);
        getToken();
        return t;
    }

    void error(String s) {
        throw new RuntimeException(s);
    }

    void getToken() {
        String s;
        try {
            s = str.nextToken();
        }
        catch(NoSuchElementException e) {
            token = EOS;
            return;
        }
        if (s.equals(" ")) getToken();
        else if (s.equals("+")) token = PLUS;
        else if (s.equals("-")) token = MINUS;
        else if (s.equals("*")) token = MULT;
        else if (s.equals("/")) token = DIV;
        else if (s.equals("(")) token = LPAR;
        else if (s.equals(")")) token = RPAR;
        else {
            try{
                value = Integer.parseInt(s);
                token = CONST;
             }
            catch(NumberFormatException e) {
                error("constant expected");
            }
        }
    }
}
```

Spørgsmål 2.6 Programmér metoden `term`.

Spørgsmål 2.7 Skriv en programstump, der benytter klassen `Parser` til at beregne og udskrive værdien af udtrykket

2 * (5 - 3) + 1

Opgave 3: Sortering (15%)

Quicksort er en effektiv metode til sortering. Sortering af en tabel af heltalsvariabler ved hjælp af Quicksort kan i Java programmeres på følgende måde.

```
public void quicksort(int a[]) {
    quicksort(a, 0, a.length-1);
}

private void quicksort(int a[], int i, int j) {
    if (i < j){
        int m = partition(a, i, j);
        quicksort(a, i, m-1);
        quicksort(a, m+1, j);
    }
}
```

Her er `partition` en hjælpemetode, der omordner et delsegment `(a[i], a[i+1], ..., a[j])` af tabellen `a`, således at følgende 3 betingelser er opfyldt:

(1) Elementet `a[m]` er på sin endelige og korrekte plads i tabellen.
(2) Intet af elementerne `a[i], ..., a[m-1]` er større end `a[m]`.
(3) Intet af elementerne `a[m+1], ..., a[j]` er mindre end `a[m]`.

At programmere en fejlfri udgave af `partition` er imidlertid ikke nogen simpel opgave. Nedenstående udgave er således ikke korrekt.

```
private int partition(int a[], int i, int j) {
    int v = a[(i+j)/2];
    while (i < j) {
        while (a[i] < v) i++;
        while (a[j] > v) j--;
        swap(a, i, j);
    }
    return i;
}

private void swap(int a[], int i, int j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;
}
```

Spørgsmål 3.1 Angiv et sæt af værdier `a[1], ..., a[n]` for hvilke kaldet `partition(a, 1, n)` ikke terminerer.

Nedenfor ses en skitse til en anden udgave af `partition`.

```
private int partition(int a[], int i, int j) {   //  1
    while (true) {                               //  2
        while (i < j && a[i] < a[j]) j--;        //  3
        if (i >= j) return i;                    //  4
        ???                                      //  5
        while (i < j && a[i] < a[j]) i++;        //  6
        if (i >= j) return j;                    //  7
        ???                                      //  8
    }                                            //  9
}                                                // 10
```

Spørgsmål 3.2 Angiv den manglende kode ved spørgsmålstegnene i linjerne 5 og 8, således at metoden bliver korrekt.

Rekursionen i `quicksort` kan let elimineres ved brug af eksplicit stak. Nedenstående klasse, `Frame`, kan benyttes til at repræsentere hvert metodekald i form af en post indeholdende værdierne af metodens lokale variable (`i`, `j` og `m`) samt en angivelse af returpunktet for metodekaldet (`PC`). Pegeren `prev` udpeger det underliggende stakelement.

```
class Frame {
    int i, j, m, PC;
    Frame prev;

    Frame(int i, int j, Frame prev) {
        this.i = i; this.j = j; this.prev = prev; PC = 1;
    }
}
```

Nedenfor er angivet en næsten færdig version af `quicksort`, hvor rekursion er elimineret ved hjælp af klassen `Frame`.

```
private void quicksort(int a[], int i, int j) {
    Frame top = new Frame(i, j, null);
    while (top != null) {
        switch(top.PC) {
        case 1: if (top.i >= top.j)
                    top = top.prev;
                else {
                    ???  /* se spørgsmål 3.3 */
                }
                continue;
        case 2: top.PC = 3;
                top = new Frame(top.m + 1, top.j, top);
                continue;
        case 3: top = top.prev;
        }
    }
}
```

Spørgsmål 3.3 Angiv den manglende kode ved spørgsmålstegnene, således at metoden bliver korrekt.

Opgave 4: OOA-modellering (26%)

De praktiserende læger i Happykøbing sender deres patienter til et privat laboratorium for at få taget blodprøver. Lægerne skriver en rekvisition med patientens stamdata (cpr. nummer, navn, adresse), lægens navn og adresse, samt en specifikation af de analyser der ønskes taget. For nemheds skyld antager vi, at de forskellige mulige analyser kaldes A1 til AN. Patienten ringer selv til laboratoriet for at bestille tid og medbringer selv rekvisitionen til laboratoriet, hvor den samme laborant tager blodprøverne, foretager de specificerede analyser og påfører resultaterne på rekvisitionen. Laboratoriet benytter postvæsenet til at sende resultatet af analyserne til lægen.

Der har været mange problemer med denne arbejdsgang. Dels synes læger og patienter, at der går for lang tid inden lægen modtager analyseresultaterne, dels har der været mange eksempler på, at der er foretaget forkerte analyser.

De praktiserende læger og laboratoriet er derfor blevet enige om at bestille en analyse med henblik på at indføre et edb-baseret system, der kan sikre færre problemer. Du skal foretage denne analyse ved hjælp af OOA. Sammen med 2 læger og 3 repræsentanter fra laboratoriet er du kommet frem til følgende foreløbige systemdefinition udtrykt i BATOFF:

Betingelser: Edb-systemet skal benyttes af lægerne, der skal kunne bestille tid til patienter og de ønskede analyser via systemet, mens patienten er i konsultationen. Laboratoriet skal kunne overføre analyseresultaterne elektronisk til lægerne. Lægerne bruger i forvejen en elektronisk patientjounal, mens laboratoriet har et papir-baseret system og derfor ikke kan forventes at have erfaringer med edb.

Anvendelsesområde: Lægerne og laboratoriet skal kunne udveksle oplysninger om patienterne. Lægerne skal sende patientens stamdata (cpr. nummer, navn, adresse), eget navn og post- og e-mailadresse, samt en specifikation af de analyser der ønskes taget. Laboratoriet skal sende analyseresultaterne til lægen.

Teknologi: De eksisterende PC'er hos lægerne skal benyttes, mens laboratoriet skal have nyt udstyr. Af hensyn til sikkerheden skal der etableret et lukket netværk mellem lægerne og laboratoriet.

Objektsystem: Patient, læge, tidsbestilling, rekvisition, laboratorium, laborant.

Funktionalitet: Støtte til tidsbestilling og kommunikation mellem læger og laboratorium om patienter, rekvisitioner og analyseresultater. Desuden skal det være muligt at finde ud af hvilken laborant der har foretaget hvilke blodprøver og analyser for hvilken patient. På sigt overvejes det endeligt at føre kontrol med laboranternes tidsforbrug.

Filosofi: Et værktøj til de arbejdsopgaver, der er involveret i at bestille tid til patienterne og udveksle de nødvendige oplysninger. Systemet skal også kunne udveksle data med lægernes elektroniske patientjournal.

Spørgsmål 4.1: Lav et klassediagram for objektsystemet og begrund diagrammet.

Spørgsmål 4.2: Lav et tilstandsdiagram for rekvisition og begrund diagrammet.

NB. Hvis du foretager afgrænsninger eller gør dig yderligere forudsætninger end de beskrevne, skal du beskrive disse i opgavebesvarelsen.