MATEMATISKA INSTITUTIONEN
STOCKHOLMS UNIVERSITET
Avd. Beräkningsmatematik
Kursledare och examinator: Anders Mörtberg

Tentamensskrivning i
DA4003 Programmeringsparadigm
7.5 hp
2026-03-09

- **Del 1** består av 8 flervalsfrågor där minst ett svarsalternativ är korrekt. Om man svarar fel eller inte har exakt rätt antal alternativ får man 0 poäng på frågan.
- **Del 2** består av ett antal frågor med varierande antal poäng vilka ska lösas genom att man skriver kod i de olika programmeringsspråken i kursen.
- **Skriv tydligt.** Svårlästa svar riskerar 0 poäng.
- Inga externa bibliotek får användas om det inte står explicit i uppgiften.
- Skriv bara på en sida av varje papper.
- För att få godkänt måste man ha minst 4 poäng på Del 1, har man inte det rättas inte Del 2.
- **Hjälpmedel:** Ett A4 med så mycket information du vill. Du får skriva på båda sidorna.
- **Betygsgränser:** E: 15, D: 18, C: 21, B: 24, A: 27, av maximala 30.

Del 1: flervalsfrågor (1p per fråga, 8p totalt)

Var snäll och samla svaren på del 1 på ett svarspapper.

1. Vad skrivs ut av C-koden till höger?

```
int *p = malloc(5 * sizeof(int));

for (int i = 0; i < 5; i++) {
    *(p + i) = i;
}

p += *(p + 2);

printf("%d", *p);
```

A. `0`
B. `1`
C. `2`
D. `3`
E. `4`

2. Vilket/Vilka påståenden nedan är sanna?

A. Variabler i JavaScript kan byta typ efter att man deklarerat dem.
B. Java är ett statiskt typat språk.
C. Java är ett dynamiskt typat språk.
D. Typfel hittas vid kompilering för statiskt typade språk.
E. En polymorf funktion är en funktion som fungerar för argument med godtyckliga typer.

3. Vad är typen på den anonyma Haskellfunktionen `\f g xs -> filter f (map g xs)` ?

A. `(b -> Bool) -> (a -> b) -> [a] -> [Bool]`
B. `(b -> Bool) -> (b -> a) -> [b] -> [Bool]`
C. `(b -> c) -> (a -> b) -> [a] -> [c]`
D. `(b -> Bool) -> (a -> b) -> [a] -> [b]`
E. `(a -> b) -> (a -> Bool) -> [a] -> [b]`

4. Vad gäller för Javakoden till höger?

```
class Volvo extends Car {
    private String owner;

    public Volvo(String o) {
        super("Volvo");
        owner = o;
    }
    public String get_owner() {
        return owner;
    }
}
```

A. Klassen `Car` är en superklass till `Volvo`
B. Klassen `Volvo` är en superklass till `Car`
C. Konstruktorn i `Car` tar in en sträng som argument
D. `owner` är en klassvariabel
E. `owner` är en instansvariabel

5. Vad blir resultatet av `foo (*) [1,2,3,4]` givet Haskellkoden nedan?

```
foo :: (a -> a -> a) -> [a] -> [a]
foo f (x:y:xs) = f x y : foo f (y:xs)
foo f _ = []
```

A. `[1,2,3,4]`
B. `[1,2,6,12]`
C. `[1,2,6,12,4]`
D. `[2,6,12,24]`
E. `[2,6,12]`

6. Hur många lösningar (d.v.s. tilldelningar till `X`, `Y` och `Z`) kommer Prolog generera för ett anrop till `p(X,Y,Z)` givet koden till höger?

```
p(X,Y,Z):- q(X), !, r(Y), s(Z).

q(1).
q(2).

r(3).
r(4).

s(5).
s(6).
```

A. 1
B. 2
C. 3
D. 4
E. 5

7. Vad blir resultatet av att köra Haskellkoden:

```
filter (\(x,y) -> x >= y) [(1,1),(1,2),(2,1)]
```

A. `[(2,1)]`
B. `[(1,1),(1,2)]`
C. `[(1,1),(2,1)]`
D. `[(1,1),(1,2),(2,1)]`
E. `[True,False,True]`

8. Givet C-koden till höger, vad är värdet på `x` när koden körts?

```
int x = 1;

for (int i = 1; i <= 5; i++) {
    if (x > i) {
        goto end;
    } else {
        x += x;
    }
}
end:
```

A. `0`
B. `1`
C. `2`
D. `3`
E. `4`

Del 2: kodfrågor (22p totalt)

Var snäll använd ett papper till varje uppgift i del 2. Lösningarna på deluppgifterna för varje programmeringsspråk kan skrivas på samma papper.

9. **Imperativ programmering i C**

   (a) Skriv en funktion `int* filter_positive(int* p, int n)` som tar in en pekare `p` som pekar på `n` tal i minnet och returnerar en pekare till en sekvens där det första värdet är antal positiva tal i sekvensen och resten av värdena är de positiva talen i sekvensen. För sekvensen `1,-2,3,-4` ska resultatet alltså vara en pekare som pekar på `2,1,3` då det är två positiva tal i sekvensen, d.v.s. `1` och `3`. (3p)

   (b) Skriv kod som visar hur `filter_positive` kan användas på `1,-2,3,-4` samt hur man sedan kan använda resultatet för att beräkna summan av de positiva talen. Kodsnutten ska alltså göra följande:

   i. Skapa en pekare till sekvensen `1,-2,3,-4`.
   ii. Anropa `filter_positive` med pekaren och `4` som indata.
   iii. Använd resultatet för att beräkna summan av de positiva talen och sedan skriva ut resultatet.

   För poäng måste koden summera resultatet från `filter_positive` i steg iii. och man får **inte** hårdkoda utskrift av summan, d.v.s. `4`. (2p)

10. **Objektorienterad programmering i Java**

    Här ska ni skriva klasser för att representera varor i en mataffär.

    (a) Börja med att skriva en klass `Item` med privata instansattribut `name` av typ `String` och `price` av typ `Double`. Klassen ska ha en lämplig konstruktor samt getters för båda instansattributen, men bara setter för `price`. Man ska alltså kunna hämta ut både `name` och `price`, men bara ändra på `price`. (2p)

    (b) Lägg till en klass `Produce` som ärver `Item` och representerar frukt och grönt. Den ska ha privata instansattribut `price_per_kg` och `weight` av typ `Double` vilka representerar pris per kg och vikt (i kg). Man behöver inte skriva getters och setters, men konstruktorn för `Produce` ska ta in namnet på varan, pris per kg och vikten, och sen ska de fyra instansattributen sättas på lämpligt sätt, d.v.s. priset ska sättas till pris per kg gånger vikten och de andra instansattributen ska sättas till det som användaren anger. (2p)

    Tips: Kom ihåg att instansattribut i superklassen kan sättas med hjälp av `super(...)`.

    (c) Hur skapar man en bananklase som kostar 21 kr/kg och väger 2kg? (1p)

11. **Funktionell programmering i Haskell**

    (a) Skriv en funktion `addPositive :: [Int] -> Int` som tar in en lista med heltal och returnerar summan av de positiva talen i listan. Negativa tal ska alltså ignoreras. (2p)

    Exempelanvändning:

```
> addPositive [1,-2,3,-4]
4
> addPositive []
0
```

    (b) Skriv en funktion `findIndex :: Eq a => a -> [a] -> Int` som tar in ett element och en lista av godtycklig typ `a` som kan jämföras för likhet. Funktionen ska returnera första index som elementet finns på i listan. Om elementet inte finns i listan ska `-1` returneras. (2p)

    Exempelanvändning:

```
> findIndex 2 [0..5]
2
> findIndex 'j' "hej"
2
> findIndex 'x' "hej"
-1
> findIndex 100 [1..10]
-1
```

(c) Skriv din egen version av den inbyggda funktionen `zip :: [a] -> [b] -> [(a,b)]`. Om en av listorna har fler element än den andra ska resterande element ignoreras då de inte går att para ihop med något. För poäng får man självfallet inte använda sig av den inbyggda `zip` funktionen. (2p)

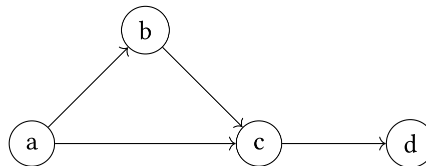
Exempelkörning:

```
> zip [1,2,3] "hej"
[(1,'h'),(2,'e'),(3,'j')]
> zip [1,2,3] [1,2,3,4,5]
[(1,1),(2,2),(3,3)]
> zip "hej" "oj"
[('h','o'),('e','j')]
```

12. Logikprogrammering i Prolog Vi kan beskriva en riktad graf i Prolog genom att ge dess kantrelation ”`edge`” på följande sätt:

```
edge(a,b).
edge(b,c).
edge(a,c).
edge(c,d).
```

Den här grafen är acyklisk (d.v.s. det är en “directed acyclic graph”, DAG) och kan ritas som:



Det faktum att grafen är acyklisk gör uppgifterna nedan lättare då vi inte behöver oroa oss för oändliga looper p.g.a. cykler. För poäng ska man skriva generell kod som funkar för alla acykliska grafer, d.v.s. inte bara för den här specifika grafen.

(a) Skriv ett predikat `connected(X,Y)` som avgör om `X` och `Y` sitter ihop med en sekvens av noll eller fler kanter i grafen, d.v.s. alla noder sitter ihop med sig själva och med alla noder som det finns en sekvens av kanter till. För full poäng ska snitt (`!`) användas på lämpligt sätt så att man direkt bara får ett svar även om `X` och `Y` går att koppla ihop på flera sätt (som t.ex. är fallet för `a` och `d`). (2p)

Exempelanvändning:

```
?- connected(a,d).
true.
?- connected(a,a).
true.
?- connected(a,c).
true.
?- connected(c,d).
true.
?- connected(c,a).
false.
```

(b) Skrev ett predikat `path(X,Y,NS)` där `X` och `Y` är noder och `NS` är en lista av noder som kopplar ihop `X` och `Y` längs en sekvens av kanter. (2p)

Exempelanvändning:

```
?- path(a,d,NS).
NS = [a, b, c, d] ;
NS = [a, c, d] ;
false.
?- path(a,a,NS).
NS = [a] ;
```

```
false.
?- path(b,d,NS).
NS = [b, c, d] ;
false.
?- path(c,a,NS).
false.
```

Obs: Här ska man kunna få ut alla vägar mellan `X` och `Y`, så snitt bör inte användas.

(c) Skrev ett predikat `distance(X,Y,N)` där `X` och `Y` är noder och `N` ett tal som räknar hur många kanter som kopplar ihop `X` och `Y` längs de olika möjliga vägarna som de sitter ihop på. (2p)

Tips: man får använda sig av `path` ovan även om man inte löst den uppgiften. Kanske kan den inbyggda relationen `length` vara till hjälp? Den funkar så att om man skriver `length(XS,N)` så är `XS` en lista och `N` längden på listan.

Exempelanvändning:

```
?- distance(a,d,N).
N = 3 ;
N = 2 ;
false.
?- distance(a,a,N).
N = 0 ;
false.
?- distance(b,d,N).
N = 2 ;
false.
?- distance(c,a,N).
false.
```