MATEMATISKA INSTITUTIONEN
STOCKHOLMS UNIVERSITET
Avd. Beräkningsmatematik
Kursledare: Anders Mörtberg
Examinator: Lars Arvestad

Tentamensskrivning i
DA4003 Programmeringsparadigm
7.5 hp
2021-03-15

- **Del 1** består av ett antal frågor med varierande antal poäng (totalt 32) vilka ska lösas genom att man skriver kod i de olika programmeringsspråken i kursen.
- **Del 2** ges muntligen och består av flervalsfrågor där minst ett svarsalternativ är korrekt. Om man svarar fel eller inte har exakt rätt antal alternativ får man 0 poäng på frågan.
- **Skriv tydligt:** onödigt svårläst eller komplicerad kod leder till poängavdrag, så indentera och kommentera koden på ett lämpligt sätt.
- Inga externa bibliotek får användas om det inte står explicit i uppgiften.
- **Hjälpmedel:** Del 1 är öppen bok då det är hemtenta och samma regler kring hjälpmedel gäller som för labbarna. På Del 2 får man ha papper och penna.
- För att få godkänt måste man ha minst 16 poäng på Del 1 och 4 poäng på Del 2.
- **Betygsgränser:** E: 20, D: 24, C: 28, B: 32, A: 36, av maximala 40.
- **Inlämning:** lösningarna ska lämnas in på inlämningsmodulen under ”Tentamen” längst ner på kurssidan. Man ska ha en fil per uppgift och filnamnen ska vara de som står i uppgifterna. Börja alla filer med att skriva en kommentar med den anonyma koden som ni fått när ni registrerade er på tentan i Ladok. Denna kod är på formen på formen `XXX-XXXX-XXX`. Man ska **inte** skriva sitt riktiga namn någonstans i filen!
- Frågor om tentan ställs via mail till Anders (`anders.mortberg@math.su.se`). Han kommer vara tillgänglig under hela tentan och svara på frågor så fort som möjligt. Eventuella korrektioner och förtydliganden kommer publiceras på kursforumet.

Del 1: kodfrågor (32p totalt)

1. **Imperativ programmering:** Lösningarna på den här delen ska vara skrivna i C och läggas i `uppgift1.c`. På båda uppgifterna får man använda `stdio.h` och `stdlib.h`.

   a) Ett sätt att kryptera en text är att skriva bokstäverna i en matris med bestämt antal kolumner. Bokstäverna fylls i rad för rad och sen läser man ut det krypterade ordet genom att läsa kolumnvis.
   Exempel: för att kryptera `Secret text` med `3` kolumner skapar vi matrisen:

S	e	c
r	e	t
	t	e
x	t	

   Den krypterade texten är då `Sr xeettcte`.
   Skriv en funktion `void matrix_encrypt(char* s, int n, int cols)` där `s` är strängen vi ska kryptera, `n` är längden på strängen och `cols` är hur många kolumner matrisen ska ha. Funktionen ska använda arrayer och inte pekararitmetik för att lösa problemet. (5p)
   Tester: om man kompilerar och kör

   ```
   int main() {
     matrix_encrypt("Secret text",11,3);
     matrix_encrypt("Secret text",11,2);
     matrix_encrypt("Secrettext",10,5);
   }
   ```

   så ska utdata bli

   ```
   Sr xeettcte
   Sce eterttx
   Stetcerxet
   ```

Obs: om längden på `s` inte är jämnt delbar med `cols` så ska man inte få några extra mellanslag eller andra tecken i slutet av den krypterade strängen.

b) Skriv en version av `matrix_encrypt` som heter `matrix_encrypt_pointer` som endast använder pekare och pekararitmetik istället för arrayer.

Tester: om man kompilerar och kör

```
int main() {
  matrix_encrypt_pointer("Secret text",11,3);
  matrix_encrypt_pointer("Secret text",11,2);
  matrix_encrypt_pointer("Secrettext",10,5);
}
```

så ska man få samma utdata som ovan. (2p)

2. **Objektorienterad programmering:** Lösningarna på den här delen ska vara skrivna i Java och läggas i `uppgift2.java`. Man får använda `java.util.Random` för att lösa uppgiften.

På den här uppgiften ska ni implementera en klass `Dice` för tärningar med olika antal sidor och en subklass `Dice6` för vanliga sexsidiga tärningar. De två deluppgifterna är:

(a) Implementera klassen `Dice`. Den ska ha följande *privata* attribut:

- `int number_of_dice`: antal tärningar.
- `int number_of_sides`: antal sidor på tärningarna. Giltiga värden är 4, 6, 8, 12 och 20. Alla tärningar har samma antal sidor.

Och följande metoder:

- En konstruktor som sätter de två privata attributen. Konstruktorn ska verifiera att användaren försöker sätta ett giltigt antal sidor. Görs inte detta ska följande kodsnutt köras:

  ```
  throw new IllegalArgumentException("Incorrect number of side");
  ```

- `int get_number_of_dice()` och `int get_number_of_sides()`: hämta ut värdet på de privata attributen.
- `int roll_dice()`: returnera utfallet, dvs summan, av ett kast med tärningarna.
- `int roll_dice_many_times(int n)`: utfallet av att kasta tärningarna `n` gånger. (4p)

(b) Lägg till subklassen `Dice6` vilken ärver från `Dice` och representerar *en* sexsidig tärning. (1p)

Tester: För att testa era tärningar kan ni använda följande kodsnutt:

```
public class uppgift2 {
    public static void main(String[] args) {
        Dice TwoDice20 = new Dice(2,20);
        System.out.println(TwoDice20.roll_dice());
        System.out.println(TwoDice20.roll_dice());
        System.out.println(TwoDice20.roll_dice_many_times(10));

        Dice6 D6 = new Dice6();
        System.out.println(D6.roll_dice());
        System.out.println(D6.roll_dice());
        System.out.println(D6.roll_dice_many_times(20));
    }
}
```

Med detta ska utdata bli något stil med (**Obs:** då tärningskast är slumpmässiga kommer ni troligtvis få andra resultat från kasten):

```
6
17
123
3
2
73
```

3. **Webb och händelsestyrd programmering:** Kod som krävs för att lösa uppgiften finns att hämta på kurshemsidan i samma mapp som tentan. Så börja med att ladda ner filerna `uppgift3.html`, `style.css` och `uppgift3.js`. Lösningen på den här delen ska vara skriven i JavaScript och er uppgift är att skriva klart `uppgift3.js`. När ni är klara ska endast denna fil lämnas in (så ni ska inte ändra i HTML och CSS filerna).

   Uppgiften går ut på att implementera ett litet interaktivt spel som utspelar sig på ett 5×5 bräde med siffror. När man börjar spelet är alla siffror större än 0 och siffrorna längs kanterna går inte att klicka på. De 9 siffrorna i mitten går att klicka på och målet är att klicka på den ruta där summan på dess grannar är högst. Lyckas användaren göra detta byts värdet i rutan ut mot 0 och rutan går inte att klicka på igen. Man ska sedan hitta den ruta som nu har högst värde på summan av grannarna. Detta fortsätter tills spelaren klickat på alla rutor i rätt ordning och ett lämpligt meddelande skrivs sedan ut. Misslyckas man med att klicka på rätt ruta får man ett meddelande som säger det och får försöka igen.

   Filen `uppgift3.js` innehåller strukturen på programmet och de funktioner som finns implementerade är:

   - `init(g)`: rita startbrädet.
   - `get_grid()`: hämta nuvarande bräde som en array av arrayer.
   - `sum_neighbors(g,c,r)`: beräkna summan av de 8 grannarna till cellen `g[c][r]`. Här antas `g` vara ett bräde (dvs array av arrayer) och `c` samt `r` är heltal.

   De funktioner ni ska lägga till är:

   a) `is_complete(g)`: har användaren klickat på alla rutor i `g` i rätt ordning? Dvs, är värdet på de 9 rutorna i mitten alla 0? (1p)

   b) `best_value(g)`: beräkna det bästa grannvärdet i `g` (dvs den maximala summan av grannarna till de 9 rutorna i mitten).

      Obs: detta ska endast returnera maximala grannvärdet och inte vilken cell som har det grannvärdet då flera celler kan ha samma grannvärde.

      Tips: använd er av `sum_neighbors`. (2p)

   c) `klick()`: om användaren klickar på en av de 9 rutorna i mitten ska följande hända:

      i. Beräkna summan av grannarna till rutan som blivit klickad på.
      ii. Beräkna vilket det bästa grannvärdet på brädet är med hjälp av `best_value`.
      iii. Om dessa är lika har användaren klickat rätt och rutans värde ska sättas till 0, färgen ska bli samma som kantrutorna och användaren ska inte kunna klicka på rutan igen.
      iv. Om de skiljer sig ska meddelandet `You can do better!` skrivas ut.

      Denna funktion ska även testa, med hjälp av `is_complete(g)`, om användaren klarat spelet. Om så är fallet ska `Congratulations!` skrivas ut. (3p)

4. **Funktionell programmering:** Lösningarna på den här delen ska vara skrivna i Haskell och läggas i `uppgift4.hs`. Om man vill får man börja från sin lösning på labb 7, men det är inte nödvändigt.

   a) Nedan följer datatyperna för booleska- och heltalsuttryck från labb 7:

```
data BoolExpr = IntEq IntExpr IntExpr
              | BoolNot BoolExpr
  deriving (Eq,Show)

data IntExpr = Const Int
             | Var String
             | Add IntExpr IntExpr
             | Mul IntExpr IntExpr
  deriving (Eq,Show)
```

   Utöka `BoolExpr` med `Or` och `And` vilka båda tar in två `BoolExprs`. Ni ska även utöka

```
evalBoolExpr :: BoolExpr -> [(String,Int)] -> Bool
```

   med fall för dessa konstruktorer. De ska evalueras som boolesk ”*eller*” och ”*och*”. (2p)

b) Lägg nu även till konstruktorerna `IntGT` och `IntLT` till `BoolExpr`. Dessa två konstruktorer ska representera ">" och "<", och tar in två `IntExprs`. Utöka även `evalBoolExpr` med fall för `IntGT` och `IntLT`.

Tips: använd er av `evalIntExpr` från labben. Har man inte löst den uppgiften kan man låta `evalIntExpr` vara `undefined`. (2p)

c) Med hjälp av konstruktorerna ni har lagt till `BoolExpr` kan vi nu lägga in `>=` och `<=` som `BoolExprs` med hjälp av Haskellfunktioner:

```
-- Implements >= as a BoolExpr
geq :: IntExpr -> IntExpr -> BoolExpr
geq e1 e2 = undefined

-- Implements <= as a BoolExpr
leq :: IntExpr -> IntExpr -> BoolExpr
leq e1 e2 = undefined
```

Byt ut `undefined` mot lämplig kod ovan utan att lägga till något nytt i `BoolExpr`. (1p)

d) På labben hade vi även en datatyp för instruktioner i ett litet imperativt språk. Nedan följer en variant av språket med tilldelningar och while-loopar:

```
data Instr = Assign String IntExpr
           | While BoolExpr [Instr]
  deriving (Eq,Show)
```

En annan form av loopar som imperativa språk ofta har är for-loopar. Dessa kan alltid implementeras med hjälp av `Assign` och `While`, så vi behöver inte lägga till en separat konstruktor för dem i språket. Detta kan göras genom att lägga till en funktion:

```
for :: (String,IntExpr) -> BoolExpr -> Instr -> [Instr] -> [Instr]
for (x,e) cond upd body = undefined
```

Denna ska fungera så att vi får kod som representerar en for-loop på formen:

```
for (x = e; cond; upd)
    body
```

Här är alltså `x = e` en tilldelning som körs innan loopen börjar, `cond` är testet för när loopen ska sluta, `upd` är en instruktion som körs i slutet av varje iteration och `body` är kroppen i loopen. Er uppgift är nu att byta ut `undefined` i `for` så att vi får en lämplig lista av instruktioner (dvs ett program i vårat imperativa språk) som implementerar for-loopen. Man ska inte ändra i `Instr` utan istället implementera for-loopen med hjälp av `Assign` och `While`. (3p)

5. **Logikprogrammering:** Lösningarna på den här delen ska vara skrivna i Prolog och läggas i `uppgift5.pl`.

a) En användbar funktion i `Data.List` biblioteket i Haskell är `nub :: Eq a => [a] -> [a]`. Den tar bort dubletter ur listor:

```
> nub [1,2,3,1,2,1]
[1,2,3]
> nub [3,2,1,2,3,1,2,1]
[3,2,1]
> nub [1,1,1]
[1]
> nub []
[]
```

Skriv ett predikat `nub(XS,YS)` i Prolog så att `YS` är `XS` utan dubletter.

Obs: det spelar ingen roll i vilken ordning elementen i `YS` är. (2p)

b) Ett sätt att komprimera en lista är att byta ut närliggande likadana element mot tupler av elementet och hur många gånger det repeteras. Skriv ett `compress(XS,YS)` predikat i Prolog så att `YS` är den komprimerade versionen av `XS`.

Tester:

```
?- compress([],YS).
YS = [].

?- compress([a],YS).
YS = [(a, 1)] .

?- compress([a,a,b],YS).
YS = [(a, 2),  (b, 1)] .

?- compress([a,a,b,b,b,a],YS).
YS = [(a, 2),  (b, 3),  (a, 1)] .
```

(2p)

c) Skriv ett predikat `uncompress(XS,YS)` vilket relaterar en lista av par med element `X` och tal `n` med en lista där varje `X` repeterats `n` gånger.

Tester:

```
?- uncompress([],YS).
YS = [].

?- uncompress([(a, 1)],YS).
YS = [a] .

?- uncompress([(a, 2), (b, 1)],YS).
YS = [a, a, b] .

?- uncompress([(a, 2), (b, 3), (a, 1)],YS).
YS = [a, a, b, b, b, a] .
```

(2p)

Del 2: flervalsfrågor (1p per fråga, 8p totalt)

6. Vilka termer hänger specifikt ihop med funktionell programmering?

A. Rekursion
B. Klasser
C. Loopar
D. Högre ordningens funktioner
E. Snitt

7. Vad är en abstrakt klass?

A. En klass som inte går att implementera i Java.
B. En klass utan några metoder.
C. En klass som inte kan instansieras.
D. En klass som representerar en rekursiv datatyp.
E. Det finns inget som heter ”abstrakt klass”.

8. Vad används = till i Prolog?

A. Boolesk likhet
B. Tilldelning
C. Unifiering med ”occurs check”
D. Unifiering utan ”occurs check”
E. Det finns ingen = i Prolog, utan endast `=:=` och `is`.

9. Hur skapar man en funktion `foo` utan argument som returnerar ett flyttal i JavaScript?

A. `def foo():`

B. `function foo()`
C. `float foo()`
D. `float function foo()`
E. Det går inte, JavaScript har inte funktioner utan allt är klasser.

10. Vad returnerar en lexer?

A. Ett syntaxträd.
B. En lexikografiskt sorterad lista av ord.
C. Rad och kolumnnummer för alla ord i en fil.
D. En exekverbar fil.
E. En lista med tokens.

11. Vad kommer skrivas ut om man kör C-koden till höger?

A. `%d`
B. `4` då vi endast skriver ut första siffran av `x`.
C. `*p`
D. `42`
E. Det går inte att säga då `*p` är en godtycklig minnesadress.

```
int x = 42;
int *p;
p = &x;

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

12. Vad är typen på Haskellfunktionen:

```
f x y = (fst x, y : snd x)
```

A. `(a,[b]) -> c -> (a,[c])`
B. `(a,b) -> [b] -> (a,[b])`
C. `(a,b) -> [b] -> (a,b)`
D. `(a,[b]) -> b -> (a,[b])`
E. Ingen då koden leder till ett typfel.

13. Vilka påståenden nedan är sanna?

A. Tolkade (eng. *interpreted*) språk är alltid snabbare än kompilerade.
B. Java är ett statiskt typat språk.
C. JavaScript är en variant av Java som körs i webbläsaren.
D. Typklasser i Haskell är till för att göra det möjligt att skriva objektorienterad kod trots att Haskell är ett funktionellt språk.
E. C har manuell minneshantering (eng. *manual memory management*).