

DD1361 Programmeringsparadigm

Laborationer 2016



Innehåll

| | |
|-------------------------------------|----|
| Labbkvitto | 1 |
| Allmänna Instruktioner | 2 |
| Labb F1: Uppvärmning i Haskell | 3 |
| Labb F2: Molekylärbiologi i Haskell | 5 |
| Labb F3: Evolutionära träd och I/O | 9 |
| Labb L1: Uppvärmning i Prolog | 14 |
| Labb L2: Konspirationsdetektion | 16 |
| Labb L3: Redovisningsschemaläggning | 18 |
| Labb S1: Reguljära Uttryck | 22 |
| Labb S2: Sköldpaddegrafik | 24 |

Senast uppdaterad: 28 oktober 2016

Labbkvitto

DD1361 Programmeringsparadigm 2016

Underskrifter är **giltiga i 6 månader**. Det är *ditt ansvar* att kontrollera att dina labbresultat rapporterats in i rapp-systemet och att snarast kontakta kursledaren om så inte gjorts.

På kursen tillämpas **CSC-skolans hederskodex**. Jag intygar att jag har läst och förstått denna hederskodex, samt att jag har läst och förstått de allmänna labb-instruktionerna på nästa sida i labb-kompendiet.

.....

.....

Signatur

Namnförtydligande

| Labb | Datum | Handledare |
|-------------|-------|------------|
| F1 | | |
| F2 | | |
| F3 | | |
| L1 | | |
| L2 | | |
| L3 | | |
| S1 | | |
| S2 | | |
| S3 | | |
| S4 | | |
| X1 | | |
| X2 | | |
| Inet | | |

Fet stil indikerar vilka labbar som är obligatoriska.

Allmänna Instruktioner

1 Kattis-systemet

De flesta av labbarna använder sig av systemet **Kattis** för automatisk rättning av er kod. För att stifta en första bekantskap med systemet kan du kolla på [hjälp-sidan](#) för ditt favorit-språk.

2 Git

I kursen använder vi KTH:s Gitlab-installation för att arbeta med labbar och lämna in dem för redovisning.

För detaljerad information om detta se [sidan om git](#) på kurs-hemsidan.

3 Kod/dokumentations-krav

Utöver att er kod ska bli godkänd av Kattis krävs följande:

1. Det ska vara tydligt dokumenterat i kommentar högst upp i koden vilka som har skrivit koden. Detta gäller *alla* inskickningar ni gör till Kattis, och är inte något ni kan lägga till i slutet när ni väl fått er kod att bli godkänd av Kattis.
2. Själva koden ska vara ordentligt kommenterad. Syftet med olika funktioner/predikat som ni definerar ska förklaras.

4 Tidsfrister

Se sidan om [laborationer](#) på kurs-hemsidan för information om vilka tidsfrister som gäller för att göra färdigt de olika typerna av labbar.

5 Kösystem

För hjälp på labb-passen använder vi kösystemet Stay A While, <http://queue.csc.kth.se/#/queue/Progp>.

Labb F1: Uppvärmning i Haskell

Problem-ID på Kattis: [kth:prog:warmup](#)

I denna labb ska du konstruera några enkla funktioner i Haskell. Alla funktioner du definierar i denna labb ska ligga i en modul som heter `F1`. I ditt git-repo för labben finns ett kod-skelett som du kan utgå ifrån, som innehåller triviala kodstubbar (som såklart inte gör rätt) för samtliga deluppgifter i labben.

1 Fibonacci-talen

Fibonacci-talen är en talföljd som definieras så här:

$$F(n) = \begin{cases} 0 & \text{om } n = 0 \\ 1 & \text{om } n = 1 \\ F(n-1) + F(n-2) & \text{om } n > 1 \end{cases}$$

Uppgift Skriv en funktion `fib` som tar ett heltal n och returnerar $F(n)$. Du behöver inte hantera negativa tal. Funktionen ska klara att beräkna $F(n)$ för n upp till 30 på en bråkdel av en sekund. Hur lång tid tar det att beräkna $F(35)$? $F(40)$?

Tips Lättast är att definiera funktionen med flera ekvationer (analogt med definitionen ovan).

Exempel

```
fib(7) ska returnera 13
```

```
fib(17) ska returnera 1597
```

2 Rövarspråket

I *rövarspråket* dubbleras man alla konsonanter och lägger ett "o" emellan, se exempel nedan. (För den här uppgiften ignorerar vi de specialfall som ibland tillämpas där t.ex. "x" behandlas som "ks".)

Uppgift Skriv en funktion `rovarsprak` som tar en sträng och returnerar en ny sträng där varje konsonant x har ersatts av strängen xox . Skriv också en funktion `karpsravor` som gör det omvända, dvs tar en sträng på rövarspråk och "avkodar" den.

Funktionerna behöver bara hantera strängar med gemener (inga mellanslag, siffror, stora bokstäver, eller andra tecken), och behöver inte hantera åäö. Funktionen `karpsravor` behöver bara fungera på strängar som verkligen tillhör rövarspråket, ingen felhantering behövs för felaktig indata.

Funktionerna ska gå i linjär tid och hantera strängar på upp till 100 000 tecken inom en bråkdel av en sekund.

Tips Ni vill antagligen skriva en funktion som avgör om ett givet tecken är vokal eller konsonant. Funktionen `elem` kan vara en praktisk byggsten för detta. I den här uppgiften anser vi "y" vara en vokal (som i svenskan).

Exempel

```
rovarsprak("progp") ska returnera poprorogogpop
```

```
rovarsprak("cirkus") ska returnera cocirorkokusos
```

```
karpsravor("hohejoj") ska returnera hej
karpsravor("fofunonkottotionon") ska returnera funktion
```

3 Medellängd

Uppgift Skriv en funktion `medellangd` som tar en text (String) som indata och returnerar ett tal (Double) med medellängden på orden i texten.

Ett ord definieras vi som en sammanhängande delsträng av bokstäver ur alfabetet, stora eller små. Alla blanka tecken, kommatering, siffror, etc, är ord-delande.

Funktionen ska gå i linjär tid och hantera texter på upp till 100 000 tecken inom en bråkdel av en sekund.

Tips Funktionen `isAlpha :: Char -> Bool` returnerar sant på just de tecken som finns i alfabetet. För att komma åt `isAlpha` måste du importera modulen `Data.Char`.

En möjlig ansats är att först stycka upp texten i ord och sedan beräkna antal ord samt totala längden på orden.

Exempel

```
medellangd("No, I am definitely not a pie!") ska returnera 3.14285714...
medellangd("w0w such t3xt...") ska returnera 1.8
```

4 Listskyffling

Vi är intresserade av att kasta om elementen i en lista enligt följande: först tar vi varannat element (första, tredje, femte, etc). Vi upprepar sedan detta på elementen som återstår (dvs tar andra, sjätte, tionde, etc). Detta upprepas så länge det fortfarande finns element kvar. Om vi t.ex. börjar med listan (1, 2, 3, 4, 5, 6, 7, 8, 9) kommer vi i första vändan få (1, 3, 5, 7, 9), och elementen (2, 4, 6, 8) återstår. I andra vändan lägger vi till (2, 6), och bara (4, 8) återstår. I tredje vändan lägger vi bara till 4, och bara 8 återstår. I fjärde och sista vändan lägger vi slutligen till 8, och slutresultatet blir listan (1, 3, 5, 7, 9, 2, 6, 4, 8).

Uppgift Skriv en funktion `skyffla` som tar en lista som indata och returnerar en omkastad lista enligt beskrivningen ovan.

Funktionen ska fungera på alla typer av listor.

Funktionen ska kunna hantera listor på upp till 5 000 element inom en bråkdel av en sekund (var försiktig med “++”-operatorm!).

Exempel

```
skyffla(["kasta", "ord", "om"]) ska returnera ["kasta", "om", "ord"]
skyffla([3.4, 2.3, 5, 185, 23]) ska returnera [3.4, 5, 23, 2.3, 185]
skyffla([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]) ska returnera [1, 3, 5, 7, 9, 11, 2, 6, 10, 4, 12, 8]
skyffla([1,2..5000]) ska returnera sitt svar inom en bråkdel av en sekund.
```

Labb F2: Molekylärbiologi i Haskell

Problem-ID på Kattis: [kth:progp:f2](#)

I denna labb ska du konstruera verktyg för att arbeta med molekylärbiologi i Haskell. Alla funktioner du definierar i denna labb ska ligga i en modul som heter F2.

1 Exempeldata och testning

För att hjälpa till på traven i testningen av din kod tillhandahålls en hjälpfil `molbio.hs` i ditt git-repo för labben. Den filen definierar en modul kallad `Molbio` som importerar din modul `F2`. Tanken är att du, om det passar dig, laddar `molbio.hs` i `ghci` och där kan testa din modul. Filen definierar följande dataset:

figur Ett mycket litet exempel på DNA-data, återfinns också i figur 1.

simple,sample Två små exempel på DNA-data.

foxp4 Sex proteiner från några ryggradsdjur.

fam1-fam5 Fem uppsättningar nukleära hormonreceptorer från ett flertal olika arter.

I filen finns också några funktioner för att köra snabba test av några av de olika funktioner du ska implementera i uppgifterna nedan. Mer information finner du i kommentarerna i filen.

2 Molekylära sekvenser

Det är främst två sorters molekyler som molekylärbiologer tittar på: DNA och proteiner. Båda har en linjär struktur som gör att man representerar dem som strängar, oftast benämnda "sekvenser". DNA har välkänd struktur över fyra byggstenar, nukleotiderna A, C, G och T, och en DNA-sekvens kan därför se ut som t.ex. `ATTATCGGCTCT`. Proteinsekvenser är uppbyggda av 20 byggstenar, aminosyror, som brukar representeras med bokstäverna `ARNDCSEQGHILKMFPSTWYV`.¹

Längder på både DNA och proteiner kan variera starkt, men man måste kunna representera sekvenser som är från några tiotal symboler långa till över 10^4 symboler.

En vanlig operation på *par* av sekvenser är att beräkna deras *evolutionära avstånd*. Att bara räkna mutationer är också vanligt, men det måttet är inte proportionellt mot tiden, så därför används statistiska modeller för sekvensers evolution.

Enligt en känd och enkel modell som kallas *Jukes-Cantor* låter man avståndet $d_{a,b}$ mellan två DNA-sekvenser a och b (av samma längd) vara

$$d_{a,b} = -\frac{3}{4} \ln(1 - 4\alpha/3)$$

där α är andelen positioner där sekvenserna skiljer sig åt (det *normaliserade Hamming-avståndet* mellan sekvenserna). Formeln fungerar dock inte bra om sekvenserna skiljer sig åt mer än väntat, så om $\alpha > 0.74$ låter man $d_{a,b} = 3.3$.

Det finns en nästan likadan modell ("Poisson-modellen") för proteinsekvenser där man sätter avståndet till

$$d_{a,b} = -\frac{19}{20} \ln(1 - 20\alpha/19)$$

för $\alpha \leq 0.94$ och $d_{a,b} = 3.7$ annars. Parametrarna är alltså ändrade för att reflektera det större alfabetet hos proteinsekvenser.

¹Borde inte aminosyroras förkortningar `ARNDCSEQGHILKMFPSTWYV` stå i bokstavsordning? Det gör de: A, R, och N representerar till exempel aminosyror Alanin, Arginin, och asparagin.

Uppgifter

1. Skapa en datatyp `MolSeq` för molekylära sekvenser som anger sekvensnamn, sekvens (en sträng), och om det är DNA eller protein som sekvensen beskriver. Du behöver inte begränsa vilka bokstäver som får finnas i en DNA/protein-sträng.
2. Skriv en funktion `string2seq` med typsignaturen `String -> String -> MolSeq`. Dess första argument är ett namn och andra argument är en sekvens. Denna funktion ska automatiskt skilja på DNA och protein, genom att kontrollera om en sekvens bara innehåller A, C, G, samt T och då utgå ifrån att det är DNA.
3. Skriv tre funktioner `seqName`, `seqSequence`, `seqLength` som tar en `MolSeq` och returnerar namn, sekvens, respektive sekvenslängd. Du ska inte behöva duplicera din kod beroende på om det är DNA eller protein!
4. Implementera `seqDistance :: MolSeq -> MolSeq -> Double` som jämför två DNA-sekvenser eller två proteinsekvenser och returnerar deras evolutionära avstånd.

Om man försöker jämföra DNA med protein ska det signaleras ett fel med hjälp av funktionen `error`.

Du kan anta att de två sekvenserna har samma längd, och behöver inte hantera fallet att de har olika längd.

3 Profiler och sekvenser

Profiler används för att sammanfatta utseendet hos en mängd relaterade sekvenser. De är intressanta därför att man har funnit att om man vill söka efter likheter så är det bättre att söka med en profil, som sammanfattar liknande gener/proteiner, än att söka enskilda sekvenser. Vanligen används profiler för att sammanfatta viktiga delar av sekvenser, men i den här programmeringsövningen förenklar vi uppgiften till att arbeta med hela sekvenser.

En profil för en uppsättning DNA- eller protein-sekvenser är en matris $M = (m_{i,j})$ där element $m_{i,j}$ är frekvensen av bokstaven i på position j . Om alla sekvenser man studerar börjar med "A", då ska vi ha att $m_{A,0} = 1$. Om hälften av sekvenserna har "A" i position 1, och den andra hälften har "C", då ska vi ha $m_{A,1} = m_{C,1} = 0.5$. Figur 1 har ett exempel på hur man går från sekvenser till profil och exemplets data finns i `molbio.hs`.

| | | | | |
|--|---------|--|---------|--|
| ACATAA AAGTCA ACGTGC AAGTTC ACGTAA | → $C =$ | $\begin{matrix} & & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} A \\ C \\ G \\ T \end{matrix} & \begin{pmatrix} 5 & 2 & 1 & 0 & 2 & 3 \\ 0 & 3 & 0 & 0 & 1 & 2 \\ 0 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 1 & 0 \end{pmatrix} \end{matrix}$ | → $M =$ | $\begin{matrix} & & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} A \\ C \\ G \\ T \end{matrix} & \begin{pmatrix} 1 & 0.4 & 0.2 & 0 & 0.4 & 0.6 \\ 0 & 0.6 & 0 & 0 & 0.2 & 0.4 \\ 0 & 0 & 0.8 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 1 & 0.2 & 0 \end{pmatrix} \end{matrix}$ |
|--|---------|--|---------|--|

Figur 1: Ett exempel på hur fem DNA-sekvenser av längd sex omvandlas till en profil. Matrisen C räknar hur många gånger varje bokstav används i varje position. Matrisen M skapas från C genom att dela varje element i C med antalet sekvenser.

Det finns flera sätt man kan mäta avståndet (eller skillnaden) mellan två profiler. Ett sätt är att räkna ut den totala elementvisa skillnaden. Låt $M = (m_{i,j})$ och $M' = (m'_{i,j})$ vara två profiler över n positioner. Deras avstånd kan då skrivas

$$d(M, M') = \sum_{i \in \{A,C,G,T\}} \sum_{j=0}^{n-1} |m_{i,j} - m'_{i,j}|$$

```

nucleotides = "ACGT"
aminoacids = sort "ARNDCSEQHILKMFPSTWYVX"

makeProfileMatrix :: [MolSeq] -> ???
makeProfileMatrix [] = error "Empty_sequence_list"
makeProfileMatrix sl = res
  where
    t = seqType (head sl)
    defaults =
      if (t == DNA) then
        zip nucleotides (replicate (length nucleotides) 0) -- Rad (i)
      else
        zip aminoacids (replicate (length aminoacids) 0) -- Rad (ii)
    strs = map seqSequence sl -- Rad (iii)
    tmp1 = map (map (\x -> ((head x), (length x))) . group . sort)
           (transpose strs) -- Rad (iv)
    equalFst a b = (fst a) == (fst b)
    res = map sort (map (\l -> unionBy equalFst l defaults) tmp1)

```

Figur 2: Hjälpkod för att konstruera profilmatrix

Man summerar alltså över såväl alfabetet samt positionerna.

Om man skapar en profil för protein-sekvenser arbetar man med matriser som har 20 rader istället för 4, en rad för var och en av de tjugo aminosyrorna (ARNDCSEQHILKMFPSTWYV).

Uppgifter

1. Skapa en datatyp `Profile` för att lagra profiler. Datatypen ska lagra information om den profil som lagras med hjälp av matrisen M (enligt beskrivningen ovan), det är en profil för DNA eller protein, hur många sekvenser profilen är byggd ifrån, och ett namn på profilen.
2. Skriv en funktion `molseqs2profile :: String -> [MolSeq] -> Profile` som returnerar en profil från de givna sekvenserna med den givna strängen som namn. Som hjälp för att skapa profil-matrisen har du koden i figur 2. Vid redovisning ska du kunna förklara exakt hur den fungerar, speciellt raderna (i)-(iv). Skriv gärna kommentarer direkt in i koden inför redovisningen, för så här kryptiskt ska det ju inte se ut!
3. Skriv en funktion `profileName :: Profile -> String` som returnerar en profils namn, och en funktion `profileFrequency :: Profile -> Int -> Char -> Double` som tar en profil p , en heltalsposition i , och ett tecken c , och returnerar den relativa frekvensen för tecken c på position i i profilen p (med andra ord, värdet på elementet $m_{c,i}$ i profilens matris M).
4. Skriv `profileDistance :: Profile -> Profile -> Double`. Avståndet mellan två profiler M och M' mäts med hjälp av funktionen $d(M, M')$ beskriven ovan.

4 Generell beräkning av avståndsmatriser

Du har nu definierat två relaterade datatyper, `MolSeq` och `Profile`. De är i grunden olika, men en operation som att beräkna avståndet mellan två objekt, till till exempel, förenar dem även om de två implementationerna är olika. Eftersom vi har två skilda datatyper men med liknande funktioner, kan det vara praktiskt att skapa en typklass för att samla dem.

Vid studier av såväl molekylära sekvenser som profiler vill man ibland räkna ut alla parvisa avstånd och sammanfatta dessa i en *avståndsmatrix*. Eftersom en typklass kan samla generella metoder kan man skriva en sådan funktion i typklassen istället för att implementera den särskilt för de två datatyperna.

En avståndsmatrix kan representeras på många sätt, men i ett funktionellt språk är det ofta bra att ha en listrepresentation. Den representation du ska använda här är en lista av tripplar på formen (namn1, namn2, avstånd).

Uppgifter

1. Implementera typklassen `Evol` och låt `MolSeq` och `Profile` bli instanser av `Evol`. Alla instanser av `Evol` ska implementera en funktion `distance` som mäter avstånd mellan två `Evol`, och en funktion `name` som ger namnet på en `Evol`. Finns det någon mer funktion som man bör implementera i `Evol`?
2. Implementera funktionen `distanceMatrix` i `Evol` som tar en lista av någon typ som tillhör klassen `Evol`, och returnerar alla par av avstånd. Den här funktionen ska sedan automatiskt vara definierad för både listor av `MolSeq` och listor av `Profile`.

Som nämndes ska avståndsmatrisen som returneras representeras som en lista av tripler på formen (namn1, namn2, avstånd). Denna ska komma i följande ordning: först kommer avstånden från det första elementet till alla andra. Sedan kommer avstånden från det andra elementet till alla andra utom det första (eftersom det redan angetts). Och så vidare. T.ex.: om vi har fyra `MolSeq`-objekt `A`, `B`, `C`, `D` och skickar in listan `[A, B, C, D]`, så ska `distanceMatrix` returnera listan

`[(A, A, ·), (A, B, ·), (A, C, ·), (A, D, ·), (B, B, ·), (B, C, ·), (B, D, ·), (C, C, ·), (C, D, ·), (D, D, ·)]`

(fast med samtliga “·” utbytta mot avståndet mellan respektive objekt).

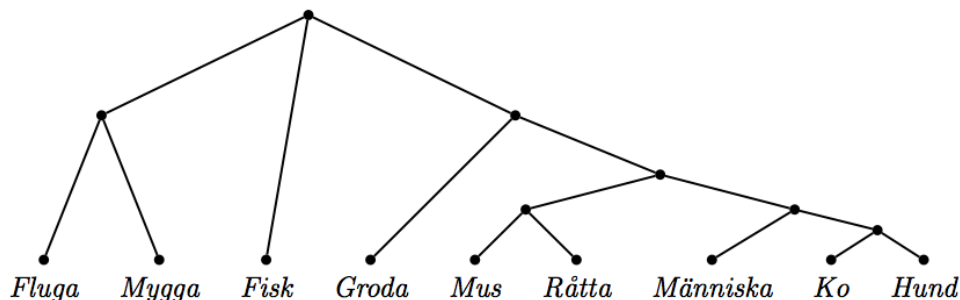
Labb F3: Evolutionära träd och I/O

Problem-ID på Kattis: [kth:progp:f3](#)

Det här är en laboration som dels testar din förmåga att arbeta funktionellt genom att överföra en abstrakt beskriven algoritm till ett funktionellt program, och dels låter dig skriva ett fullständigt Haskell-program med inläsning och utskrifter – operationer som har sido-effekter. Vi fortsätter arbeta med problem från molekylärbiologin och det är meningen att det du skriver ska bygga på laboration F2 – du kan antagligen med fördel använda din lösning från F2 utan ändringar, och skriva din lösning på denna labb i en ny modul F3 som importerar F2.

1 Bakgrund

Det finns många anledningar till att vara intresserad av hur arter och/eller gener har uppkommit, bland annat är det för många viktigt att helt enkelt förstå hur olika arter har uppstått. Inom medicin kan kunskap om geners utveckling ge kunskap om hur de fungerar och vilken funktion de har. En grundläggande fråga är då hur man rekonstruerar det evolutionära träd, en fylogeni, som gav upphov till sekvenserna? Figur 1 ger ett exempel på en fylogeni. Indata är alltså en mängd sekvenser, DNA eller protein, och utdata är ett träd där alla inre hörn har grad 3.



Figur 1: Exempelfylogeni för en hypotetisk gen funnen i nio arter. Lägg märke till att det här trädets inte är avsett att påstå något om var evolutionen började, dvs vilken punkt i trädets som är äldst. Man säger att trädets är orotat. (Med vår övriga kunskap om de olika arterna kan vi dock vara ganska säkra på att en eventuell rot skulle ligga på kanten mellan insekterna och fisken i den här specifika fylogenin).

2 Algoritmen Neighbor Joining

Den vanligaste och mest kända algoritmen för att återskapa träd givet avståndsdata är Neighbor Joining (NJ). Det är en iterativ algoritm som successivt väljer ut par av löv och slår ihop dem till ett nytt hörn. Vilket par man väljer är avgörande för att resultatet ska bli bra, och NJ garanterar faktiskt inte att det är det bästa trädets (i meningen: passar bäst med avstånden) som returneras: NJ är en girig heuristik.

Låt F_1 vara den mängd hörn som ska vara löv i det träd T vi bygger. Låt D_1 vara avståndsmatrisen över F_1 . I varje iteration i av NJ kommer vi att välja ut två element a, b i F_i och kombinera ihop dessa till ett träd. Detta skapar en ny mängd F_{i+1} och avståndsmatris D_{i+1} .

Urvalsfunktionen S

Vi väljer de två element $a, b \in F_i$ som minimerar följande urvalsfunktionen S :

$$S_i(x, y) = (|F_i| - 2)D_i(x, y) - \sum_{z \in F_i} (D_i(x, z) + D_i(y, z))$$

Funktionen ser vid en första anblick en smula underlig ut, men man kan göra en tolkning av den. Den andra termen mäter hur långt bort övriga hörn ligger från x och y , och den första termen mäter hur nära

x och y ligger varandra, skalat med en faktor för att göra de två termerna jämförbara. Det S kan sägas välja ut är alltså de två hörn som ligger längst ifrån de andra.

Pseudokod

Vi använder oss av en förenklad version av NJ. Den som tittar på andra beskrivningar av NJ kommer att finna att steg 2d är lite hårigare än vad som ges här. Låt F_1 och D_1 vara indata.

1. $i \leftarrow 1$
2. Så länge $|F_i| > 3$:
 - (a) Hitta det par $a, b \in F_i$ som minimerar $S_i(a, b)$
 - (b) Skapa ett nytt träd T_i där träden a och b är barn till en nyskapad nod.
 - (c) $F_{i+1} \leftarrow F_i \cup \{T_i\} \setminus \{a, b\}$ – Lägg till det nya trädet till F och ta bort de gamla
 - (d) Skapa D_{i+1} från D_i enligt

$$D_{i+1}(x, y) = D_i(x, y) \quad \text{för } x, y \in F_{i+1} \setminus \{T_i\}$$

$$D_{i+1}(x, T_i) = D_{i+1}(T_i, x) = \frac{D_i(x, a) + D_i(x, b)}{2} \quad \text{för } x \in F_{i+1} \setminus \{T_i\}$$

- (e) $i \leftarrow i + 1$

3. Skapa ett nytt träd T där de tre kvarvarande träden i F_i är barn till en nyskapad nod
4. Returnera T

Exempelkörning

Antag att vi har de fem sekvenserna a, b, \dots, e som beskrivs i exempel-data 1 nedan. Använder vi vår kod från F2 för att beräkna avståndsmatrisen D_1 för dessa får vi:

$$D_1 \approx \begin{matrix} & a & b & c & d & e \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \begin{pmatrix} 0 & 0.304 & 0.824 & 0.520 & 0.824 \\ & 0 & 3.300 & 1.344 & 0.824 \\ & & 0 & 0.137 & 0.304 \\ & & & 0 & 0.137 \\ & & & & 0 \end{pmatrix} \end{matrix}.$$

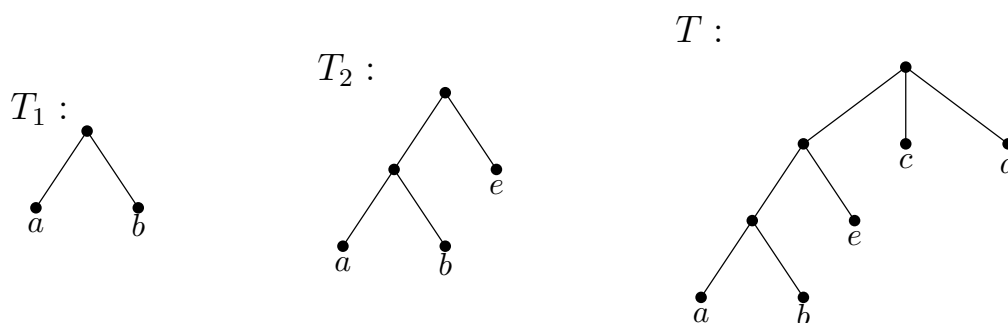
I algoritmens första iteration har vi "träden" $F_1 = \{a, b, c, d, e\}$, alla bestående av en enda nod. Urvalsfunktionen S_1 för första iterationen får följande värden:

$$S_1 \approx \begin{matrix} & b & c & d & e \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{pmatrix} -7.331 & -4.565 & -3.049 & -2.089 \\ & -0.437 & -3.878 & -5.389 \\ & & -6.292 & -5.741 \\ & & & -3.816 \end{pmatrix} \end{matrix}.$$

Den minimeras alltså av paret (a, b) , så vi skapar ett nytt träd T_1 som består av en ny nod med a och b som barn. Den nya avståndsmatrisen D_2 över de aktiva träden $F_2 = \{T_1, c, d, e\}$ och nya urvalsfunktionen S_2 kommer se ut som följer:

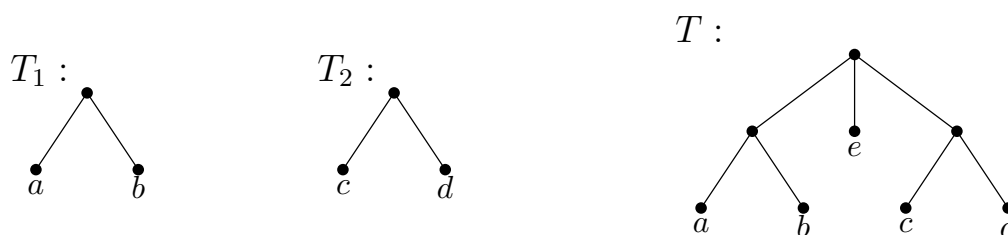
$$D_2 \approx \begin{matrix} & T_1 & c & d & e \\ \begin{matrix} T_1 \\ c \\ d \\ e \end{matrix} & \begin{pmatrix} 0 & 2.062 & 0.932 & 0.824 \\ & 0 & 0.137 & 0.304 \\ & & 0 & 0.137 \\ & & & 0 \end{pmatrix}, \quad S_2 \approx \begin{matrix} & c & d & e \\ \begin{matrix} T_1 \\ c \\ d \end{matrix} & \begin{pmatrix} -2.197 & -3.159 & -3.435 \\ & -3.435 & -3.159 \\ & & -2.197 \end{pmatrix} \end{matrix}.$$

Urvalsfunktionen S_2 minimeras av endera paret (c, d) eller paret (T_1, e) , och vi kan välja vilket som helst av dem. Låt oss säga att vi väljer det senare, dvs (T_1, e) . Vi bildar då ett träd T_2 som består av en ny nod med T_1 och e som barn. Vi har nu bara tre aktiva träd kvar $F_3 = \{T_2, c, d\}$, och vi går till steg 3 i algoritmen och skapar trädet T bestående av en ny nod med T_2 , c , och d som barn. Figur 2 illustrerar de olika träden som byggs upp under körningen.



Figur 2: Träden som byggs upp när algoritmen kör på första exempel-fallet

Hade vi i iteration 2 istället valt att para ihop (c, d) istället för (T_1, e) hade vi fått resultatet som visas i figur 3. Eftersom vi betraktar träden som orotade så är slutresultatet i själva verket samma träd som T från figur 2 även om det ritats annorlunda pga att vi kopplade ihop noderna i en annan ordning.



Figur 3: Alternativ körning på första exempel-fallet

3 Indata

Ditt program ska läsa en uppsättning DNA-sekvenser på standard input (till end of file). Vi kommer använda ett enkelt format där varje sekvens beskrivs av en rad med namnet på sekvensen följt av själva sekvensen, åtskiljda av mellanslag (varken namnet eller sekvensen kommer att innehålla några mellanslag).

Du kan anta följande om indata-strömmen:

- Alla sekvenser kommer att ha samma längd.
- Den innehåller minst 3 och högst 32 sekvenser.
- Varje sekvens är minst 1 och högst 1000 tecken lång.
- Namnen består bara av tecknen 'a'-'z', 'A'-'Z', och '0'-'9', och är mellan 1 och 15 tecken långa.
- Sekvenserna är DNA-sekvenser dvs består bara av tecknen ACGT.

4 Utdata

Ditt program ska skriva ut ett evolutionärt träd på standard output. Ett evolutionärt träd (eller fylogeni), skrivs ofta på ett format som kallas Newick. Det träd som visas i figur 2 kan då se ut så här:

```
(( (a, b), e), c, d)
```

Så länge trädet saknar rot kan man skriva trädet på flera ekvivalenta sätt. Detta träd kan alltså även skrivas på följande sätt:

```
((a,b),e,(c,d))
(a,b,((c,d),e))
(b,a,((c,d),e))
```

och många fler. Ett löv i trädet representeras alltså av en sträng med lövets/sekvensens namn. Ett inre hörn i trädet representeras med hjälp av parenteser runt om två komma-åtskilda delträd. På den översta nivån använder vi parenteser runt tre delträd.

Alla ekvivalenta sätt att formatera trädet kommer att godkännas.

Tips

Börja med I/O-delen: skriva ett program som läser in indatasekvenserna, beräknar deras avståndsmatris (D_1) med hjälp av din lösning från F2, och sedan skriver ut avståndsmatrisen. Håll delarna av programmet som är inkapslade i IO-monaden minimala – du ska inte behöva gå in i din kod från F2 och ändra den!

I den här uppgiften är det bra att använda `ghc` för att kompilera Haskell-koden till ett körbart program (utöver att använda `ghci` för att testa olika funktioner i ert program). I terminalen i Unix kan man sedan använda omdirigering av standard input för att skicka en fil till programmet. Om ni t.ex. har sparat ned första exempel-fallet till filen `sample1.in` och kompilerat ert program till en körbar binär `Main` så ska ni i terminalen kunna skriva

```
> ./Main < sample1.in
```

för att köra ert program på det aktuella indata.

Implementera sedan något sätt att representera träden som används i algoritmen, och utskrift av dessa. Modulerna `Data.Either` eller `Data.Maybe` kan vara behändiga här – kolla upp dessa!

Med kringarbetet avklarat kan man gripa sig an själva algoritmen. Du vill nog ha något sätt att representera aktuellt tillstånd i algoritmen (mängden F_i av träd vi har kvar att knyta ihop och avståndsmatrisen D_i), och en funktion som givet avståndsmatrisen D_i och två element $x, y \in F_i$ beräknar urvalsfunktionen $S_i(x, y)$.

Det är antagligen bra att följa den ovan givna algoritmens struktur ganska nära. Indata kommer vara relativt litet, högst 32 sekvenser, vilket innebär att din implementation inte behöver vara speciellt effektiv och du kan använda naiva lösningar till de olika delproblemen. Om du vill är det såklart tillåtet att göra en smartare, snabbare implementation – modulerna `Data.Set` och `Data.Map` kan vara en bra början till detta.

Sample Input 1

```
a AACCGGTT
b AACCGGGG
c AATTTTTT
d AACTTTTT
e AACTTTTG
```

Sample Output 1

```
(( (a,b) , e) , c, d)
```

Sample Input 2

```
C CCCCCCCCCC
Go GGGGGGGGGG
Python GAGGCACCGGGG
Java ACACAACCCCC
Prolog TTTCATCTTTTT
Haskell CGCGCGTTTGTT
```

Sample Output 2

```
((Prolog,Haskell) , (Go,Python) , (C,Java))
```

Sample Input 3

```
1 AAA
2 CCC
3 GGG
```

Sample Output 3

```
(1, 2, 3)
```

Labb L1: Uppvärmning i Prolog

Problem-ID på Kattis: [kth:prog:warmup](#)

I denna labb återvänder vi till uppgifterna från labb F1. Du ska nu lösa dessa även i Prolog. I ditt git-repo för labben finns ett kod-skelett som du kan utgå ifrån, som innehåller triviala kodstubar (som såklart inte gör rätt) för samtliga deluppgifter i labben.

Nedan följer specifikationer för de prolog-predikat du ska skriva. Om du har glömt bort vad som ska beräknas i de olika deluppgifterna, se labb F1.

Vanlig fallgrop

I samtliga uppgifter ska predikaten ni skriver bara generera varje lösning en gång. Med andra ord, ska t.ex. `rovarsprak([100], X)` generera $X = [100, 111, 100]$ som lösning exakt en gång, och `skyffla([1, 2], X)` generera $X = [1, 2]$ exakt en gång. Om er lösning genererar samma svar flera gånger så kommer lösningen inte bli godkänd av Kattis.

1 Fibonacci-talen

Skriv ett predikat `fib(+N, ?F)` som är sant om och endast om F är det N 'te Fibonacci-talet. Målet `fib(30, F)` ska alltså ha exakt en lösning, nämligen att F är det trettionde Fibonacci-talet.

Att diskutera vid redovisning. Jämför hur stora Fibonacci-tal du kan beräkna med din Prolog-lösning jämfört med din Haskell-lösning.

2 Rövarspråket

Skriv ett predikat `rovarsprak(?Text, ?RovarText)` som är sant om och endast om `RovarText` är `Text` översatt till rövarspråket.

De två strängarna `Text` och `RovarText` kommer att representeras som listor av tecken-koder. Strängen "hej" representeras t.ex. av listan `[104, 101, 106]` (ASCII-koderna för 'h', 'e', och 'j'). Det finns många olika predikat för att arbeta med sådana listor (manualen är din vän). Ett praktiskt predikat att känna till är `writelf(+Atom)` som skriver ut listan formaterad som en sträng (`writelf([104, 101, 106])` skriver alltså ut "hej").

Predikatet `rovarsprak` ska gå att använda "åt båda hållen", d.v.s. `rovarsprak(T, [104, 111, 104, 101, 106, 111, 106])` (vilken är strängen som representeras av andra argumentet?) ska unifiera `T` med `[104, 101, 106]`.

3 Medellängd

Skriv ett predikat `medellangd(+Text, ?AvgLen)` som är sant om och endast om `AvgLen` är medellängden av texten `Text`. Texten `Text` representeras precis som i förra uppgiften av en lista med tecken-koder.

Tips. Det inbyggda predikatet `char_code` kan komma till användning, kolla upp det!

4 Listskyffling

Skriv ett predikat `skyffla(+Lista, ?Skyfflad)` som är sant om och endast om `Skyfflad` är `Lista` efter omskyffling.

Att diskutera vid redovisning. Kan man använda ditt predikat för att “avskyffla” en lista (dvs gå tillbaka till ursprunget från en omskyfflad lista)?

Tips. Det inbyggda predikatet `append` kan komma till användning.

Labb L2: Konspirationsdetektion

Problem-ID på Kattis: [kth:progp:l2](#)

1 Bakgrund

Foliehattarnas Riksförbund har beställt en programvara för att söka efter möjliga konspirationer i sociala nätverk. Ett socialt nätverk i den här uppgiften består av en uppsättning personer, och information om vilka par av personer som är bekanta med varandra.

En konspiration består av ett flertal personer: en mängd konspiratörer samt en spindel i nätet, med följande egenskaper:

- Spindeln i nätet är bekant med alla konspiratörer (men kanske även andra personer som inte är konspiratörer).
- Ingen av konspiratörerna är bekant med någon annan konspiratör (eftersom detta skulle dra misstanke till sig).
- Alla personer som inte redan är konspiratörer eller spindeln i nätet känner någon av konspiratörerna (så att konspirationen har inflytande över hela nätverket).

Databas

Det sociala nätverket du ska söka i kommer vara definierat via följande två predikat.

person(?X) är sant om X är en person (så “`person(X)`.” kan användas för att lista alla personer i databasen).

knows(?X, ?Y) är sant om X känner Y . Observera att vi i den här uppgiften betraktar bekantskapsrelationen som symmetrisk – om X känner Y så är Y bekant med X – men i den Prolog-databas som ditt program kommer arbeta med kommer bara en av de två ordningarna vara definierad.

Databasen kommer att definiera ett relativt litet socialt nätverk, det kommer att ha högst 32 personer.

Exempeldata och testning

Till er hjälp finns en uppsättning exempel-databaser inlagda i ert git-repo för labben, i katalogen “`examples/`”. I `README.md`-filen i ert git-repo finns detaljerade instruktioner om hur ni kan provköra med exempel-databaserna samt hur man använder `trace`, om ni inte redan gjort detta.

2 Uppgift

1. Skriv ett predikat `spider(?X)` som är sant om X kan vara spindeln i nätet i en konspiration. När predikatet anropas med en okänd ska det generera varje möjlig spindel i nätet. Det spelar ingen roll i vilken ordning de genereras, men samma person ska inte genereras flera gånger (se exempel-databasen “`example2.pl`”).
2. Konstruera en egen exempel-databas för problemet där ert program tar väsentlig tid på sig (säg, mer än 10 sekunder). Ert exempel får inte vara för stort – högst 500 personer (vilket ju är mycket större än de 32 personer som används i Kattis-testfallen), och ju mindre desto bättre.

Vid redovisning ska ni även kunna förklara hur sökningen efter lösningar fungerar i ert program (t.ex. på er egenkonstruerade databas), med hjälp av `trace`.

3 Vägledning

En bra början är att skriva predikat för de olika del-egenskaperna vi är intresserade av, t.ex.:

- Ett predikat som kollar om två personer känner varandra (dvs om antingen $\text{knows}(X, Y)$ eller $\text{knows}(Y, X)$ är sant).
- Ett predikat som givet en lista med personer kollar om det finns någon person utanför listan som inte känner någon i listan.
- Etc

Givet dessa kan en första lösning på problemet ha följande struktur, enligt “generate-and-test”-metoden.

1. Låt s vara en person och K en lista med (andra) personer.
2. Testa om K kan vara konspiratörerna med s som spindeln i nätet.

Utöver predikaten för att verifiera de önskade egenskaperna behöver du alltså skriva ett predikat som genererar alla möjliga delmängder av personer, ungefär på samma sätt som vi i exemplet från föreläsningen med permutations-sortering har ett predikat som genererar alla permutationer av en lista.

Du kan skicka in denna lösning till Kattis för att se hur långt den klarar sig. Du borde då klara ca de 20 första testfallen i Kattis (och sedan få Time Limit Exceeded), om du inte gör det så är det antagligen något som du gjort konstigt som vore värt att fixa innan du går vidare med en mer komplicerad lösning.

Hur ska vi ändra lösningen så att den blir snabbare? Istället för att generera hela listan K och sedan testa om den uppfyller villkoren för att vara konspiratörerna kan man kontrollera vissa av villkoren medans listan genereras. På det sättet kommer man tidigt i genereringen upptäcka att en partiell lista av konspiratörer inte kommer leda till en lösning, och därmed slippa ägna tid åt att utforska den. Följande är ett bra sätt att utföra genereringen som ska vara tillräckligt snabbt för att klara testfallen på Kattis:

1. I varje steg har vi en lista K med personer som vi valt ut som konspiratörer hittills, och en lista P med personer som är ytterligare potentiella konspiratörer. Initialt är K tom, och P alla personer som känner s (personen som vi försöker identifiera som spindel).

Så länge P inte är tom har vi två möjliga sätt att (rekursivt) gå vidare: vi tar ut den första personen x i P :

Fall 1: x ska vara en konspiratör. Vi skapar en ny lista K' genom att lägga till x till listan K , och en ny lista P' genom att ta bort x , och alla personer som känner x , ur P (eftersom inga av dessa längre kan vara konspiratörer).

Fall 2: x ska inte vara en konspiratör. Vi låter då $K' = K$, och skapar en ny lista P' genom att ta bort x ur P .

I båda fallen fortsätter vi sedan rekursivt med de nya listorna K' och P' .

När listan P är tom har vi genererat färdigt K och kontrollerar huruvida det är en lösning.

2. I varje steg kontrollerar vi att alla personer i databasen som inte är med i K eller P känner minst en person i minst en av listorna K och P . Om det finns en person som inte är med i K eller P och inte heller känner någon i K eller P så kommer den här grenen av sökningen inte att leda till någon lösning (Fundera på varför! På redovisningen ska du kunna förklara detta), och vi kan därför avbryta.

För att få din lösning att fungera kommer du att behöva förstå *hur* Prolog letar efter en lösning. Prologs trace-verktyg är mycket användbart för detta – om du inte redan använde det till labb L1 så ska du använda det nu. Den första exempel-databasen har lite instruktioner för hur man kan använda trace.

Labb L3: Redovisningsschemaläggning

I denna labb ska du skriva en villkorsprogrammerings-modell för att schemalägga redovisningar av labbar. Modelleringspråket du ska använda är **MiniZinc**. MiniZinc och villkorsprogrammering ligger mycket närmare forskningsfronten än vad vanliga programmeringspråk gör, och därför är det i en del avseenden inte lika välpolerat och färdig-utarbetat som vanliga programspråk.

Du ska alltså förvänta dig att det kommer vara besvärligare att hitta utförlig dokumentation än det är för t.ex. Haskell eller Prolog.

Komma igång med MiniZinc

MiniZinc finns installerat i CSC:s Ubuntu-miljö. Om du använder egen dator kan du ladda hem MiniZinc för ditt OS från MiniZincs hemsida.

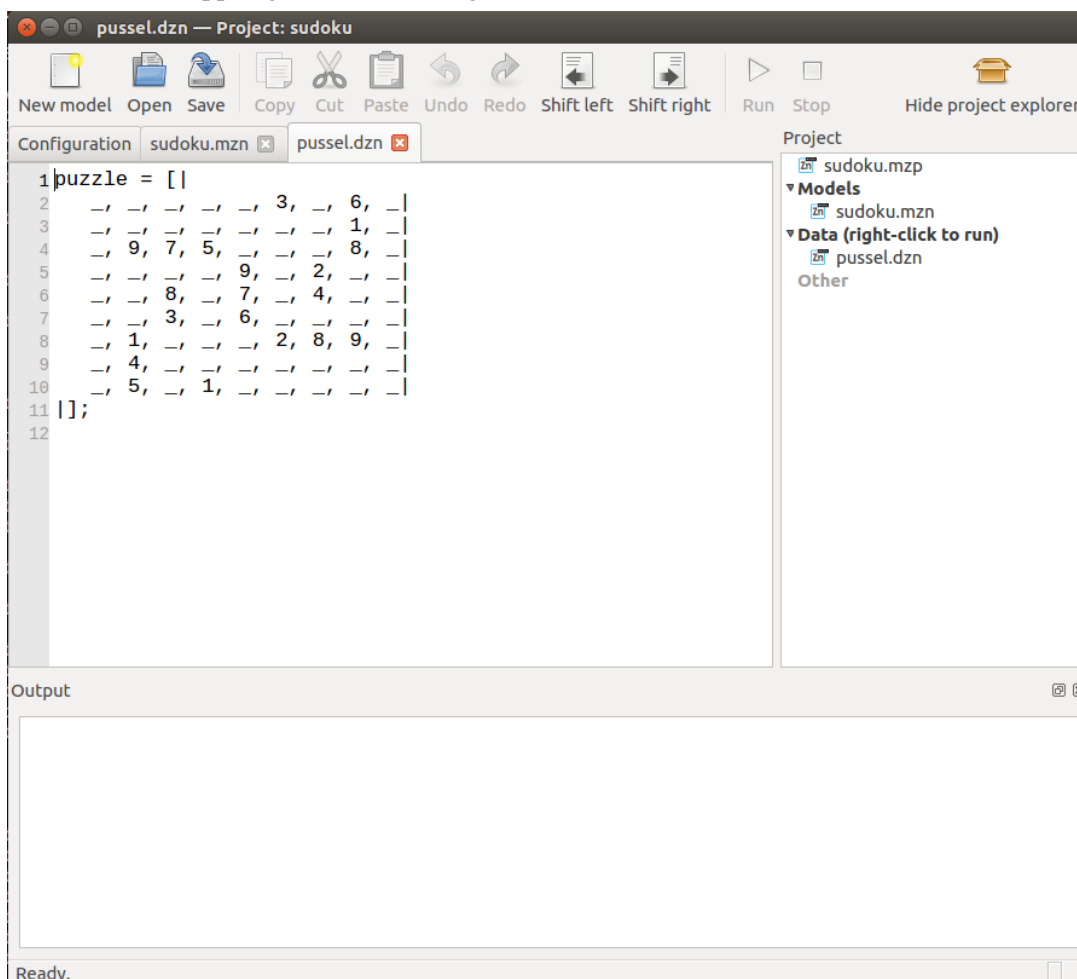
I ditt git-repo för labben finns (under katalogen `sudoku/`) Sudoku-lösningsexemplet från gästföreläsningen om villkorsprogrammering (11 oktober). Det består av tre filer:

1. `sudoku.mzn` är själva modellen,
2. `pussel.dzn` är en konkret Sudoku-instans,
3. `sudoku.mzp` är en projektfil för MiniZinc-IDE:t.

Du kan öppna `sudoku`-projektet i MiniZinc-IDE:t från terminal med kommandot

```
username@dator:~/path$ minizinc-ide sudoku.mzp
```

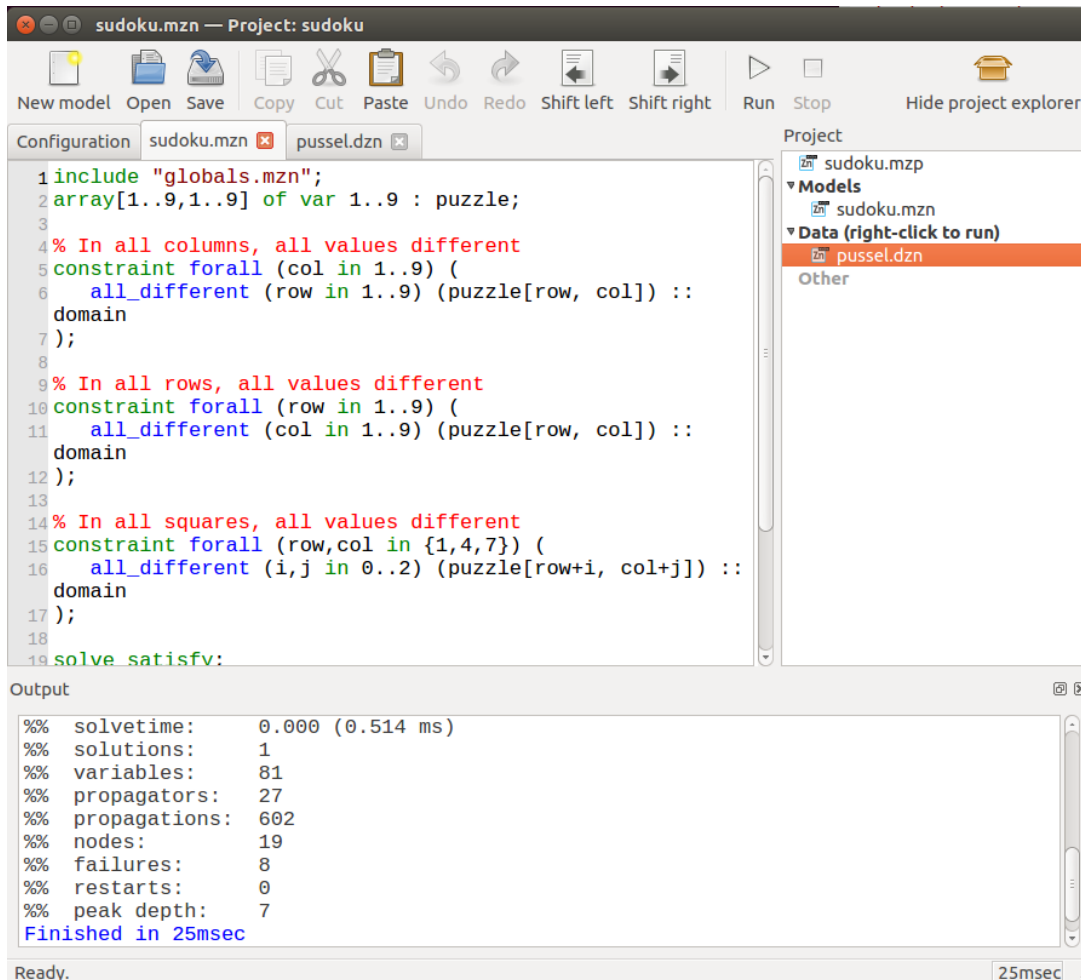
Du borde då få upp något som ser ut ungefär så här:



För att köra sudoku-modellen och lösa pusslet:

1. Navigera till fiken som har modellen `sudoku.mzn`,
2. Högerklicka på `pussel.dzn` i projektvyn till höger (om du inte ser denna, klicka på “Show project explorer” längst upp till höger).
3. Klicka “Run model with this data”.

Det dyker då upp lite info i textboxen längst ner:



The screenshot shows the MiniZinc IDE interface. The main editor displays the following code:

```
1 include "globals.mzn";
2 array[1..9,1..9] of var 1..9 : puzzle;
3
4 % In all columns, all values different
5 constraint forall (col in 1..9) (
6   all_different (row in 1..9) (puzzle[row, col]) ::
7   domain
8 );
9 % In all rows, all values different
10 constraint forall (row in 1..9) (
11   all_different (col in 1..9) (puzzle[row, col]) ::
12   domain
13 );
14 % In all squares, all values different
15 constraint forall (row,col in {1,4,7}) (
16   all_different (i,j in 0..2) (puzzle[row+i, col+j]) ::
17   domain
18 );
19 solve satisfy;
```

The right-hand side shows a project explorer with the following structure:

- Project
 - sudoku.mzp
 - sudoku.mzn
 - Data (right-click to run)
 - pussel.dzn
 - Other

The bottom output window displays the following execution statistics:

```
%% solvetime: 0.000 (0.514 ms)
%% solutions: 1
%% variables: 81
%% propagators: 27
%% propagations: 602
%% nodes: 19
%% failures: 8
%% restarts: 0
%% peak depth: 7
Finished in 25msec
```

The status bar at the bottom indicates "Ready." and "25msec".

Om du scollar upp i textboxen, eller förstorar den, kommer du högre upp hitta själva lösningen.

Fliken “Configuration” i MiniZinc har en rad olika inställningar man kan pillra med för att justera hur MiniZinc arbetar på olika sätt, det är en god idé att orientera sig bland dessa och se vilka inställningar som finns.

MiniZinc har en bra [tutorial](#) som man kan titta på för att bekanta sig närmare med modelleringspråket. En annan användbar resurs är [dokumentationen](#) över alla olika inbyggda funktioner och villkor som finns i MiniZinc.

Redovisnings-schemaläggning

Vi vill automatisera schemaläggning av redovisning av labbar i en kurs. Kursen har ett antal olika labbar, och studenter har skickat in lösningar på dessa. De data som är givna är en mängd labb-lösningar, samt hur många labbhandledare som är tillgängliga, och hur många redovisningstider som finns för varje labbhandledare.

Varje *labb-lösning* specificeras av två parametrar:

1. Vilken labb som lösts
2. Vilken delmängd studenter som gjort lösningen (typiskt två studenter då de jobbar i par).

Antal handledare och redovisningstider ges helt enkelt av varsitt heltal.

Givet dessa data vill vi schemalägga en mängd redovisningar. Varje *redovisning* specificeras av fyra parametrar:

1. Vilken labb som redovisas
2. Vilka studenter som ingår i denna redovisningsgrupp
3. Vilken handledare som de ska redovisa för
4. Vilken tid de ska redovisa

Följande villkor gäller för redovisningspassen:

- Varje student ska få redovisa exakt de labbar hen löst, och inte behöva redovisa någon labb mer än en gång.
- Redovisningsgrupperna ska bestå av grupper om 2 studenter (i undantagsfall en grupp per labb med 3 studenter, om det är ett udda antal studenter som gjort den labben).
- Varje student ska redovisa varje labb tillsammans med en annan student (eller två andra, i undantagsfallet med en redovisningsgrupp av storlek 3) än den/de studenter som hen gjort labben tillsammans med. Med andra ord: om två studenter jobbat tillsammans på en lab ska de inte hamna i samma redovisningsgrupp för den labben (men om de båda ska redovisa någon annan labb och inte jobbat tillsammans på den så är det OK om de hamnar i samma redovisningsgrupp för den andra labben).
- En labbhandledare kan inte vara inbokad på två olika redovisningar vid samma tidpunkt.
- En student kan inte vara inbokad på två olika redovisningar vid samma tidpunkt.

Modell-skelett

I katalogen `scheduling/` i ditt git-repo finns ett MiniZinc-projekt `scheduling.mzp` för labben. Detta projekt innehåller ett skelett till en modell, som definierar alla problemets datavariabler och lösningsvariabler, samt innehåller kod för utskrift av lösningen. Det du behöver göra är att konstruera de villkor som lösningen ska uppfylla.

Skelettet definierar även flera olika hjälpvariabler som kan vara användbara i en lösning. Du kan såklart välja huruvida du vill eller inte vill använda dessa, och huruvida du vill definiera ytterligare hjälpvariabler.

I slutändan ska de variabler som definieras i kodskelettet ges en korrekt tilldelning, så att modellen producerar utskrift enligt det givna utskriftsformatet.

Datafiler

Det givna projektet har innehåller även en uppsättning datafiler. De är indelade i tre olika “svårighetsgrader”: easy, medium och hard. För att bli godkänd på labben krävs att din modell klarar samtliga easy-instanser, och minst hälften (avrundat uppåt) av alla medium-instanser. Din modell behöver inte klara någon av hard-instanserna (kursledarens modell gör inte det!), dessa tillhandahålls bara som en extra utmaning för den som är intresserad.

Att “klara” en instans här betyder att en lösning ska hittas (eller icke-satisfierbarhet upptäckas) inom högst någon minut på en normal dator.

Dokumentation

Du ska skriva ett kort dokument som beskriver vad som gjorts. Detta ska innehålla:

1. Namn på vem/vilka som skrivit lösningen.
2. Vilka av testfallen du klarar och vilka lösningar du får, samt hur många “nodes” som användes och hur mycket CPU-tid varje testfall tog att lösa.
3. Reflektion över labben och resultatet, och dina intryck av villkorsprogrammering i allmänhet och minizinc i synnerhet.

Rapporten kan vara en enkel txt-fil eller en pdf. *Word-dokument eller liknande format är ej OK.* En typisk längd är 1 sida men det är inget krav att rapporten måste vara så lång.

Labb S1: Reguljära Uttryck

Problem-ID på Kattis: [kth:progp:s1](#)

Reguljära uttryck och deras varianter är mycket praktiska vid vardaglig programmering. I denna laboration ska du konstruera reguljära uttryck för lite olika strängsökningsproblem. För att kommunicera dina reguljära uttryck till Kattis ska du använda programspråket Python. Du kommer inte behöva skriva någon avancerad Python-kod, så du behöver inte ha använt Python tidigare. **Dina funktioner måste ligga i en fil som heter `s1.py`** annars kommer du få Run Time Error (“ImportError”) i Kattis.

I ditt git-repo för labben hittar du ett kodskelett, i vilket ett flertal funktioner definieras. I skelettet returnerar alla funktionerna en tom sträng, men de ska i din lösning returnera strängar som innehåller olika regex för att lösa de olika deluppgifterna nedan. T.ex. ska alltså den första funktionen, `dna()`, returnera ett regex för att matcha DNA-sekvenser. Kodskelettet innehåller även lite kod för att testa din lösning, se kommentarer i kodskelettet för hur du kan gå tillväga med detta.

I två av uppgifterna ska det reguljära uttryck du konstruerar bero på en söksträng som skickas som indata. Här kan du behöva skriva lite minimal Python-kod (Python-manualens [tutorial om strängar](#) är nog till hjälp om du aldrig använt Python förut).

De regex du konstruerar får vara högst 250 tecken långa (detta är en generöst tilltagen gräns), förutom i de två uppgifterna som tar en söksträng som indata. Om du i någon av de andra uppgifterna returnerar ett för långt regex kommer din inskickning att få ett Run Time Error i Kattis. I de två uppgifterna med en söksträng som indata finns ingen specifik övre gräns på hur långt ditt regex får vara, men om det är för långt och komplicerat kommer din lösning att få Time Limit Exceeded.

Matchningen som kommer att utföras med de regex du konstruerar är att den kommer söka efter någon del av strängen som matchar ditt uttryck. Det betyder att i uppgifter där kravet är att hela strängen ska uppfylla något villkor så måste du använda de speciella regex-symbolerna “^” och “\$”. Du kan läsa mer om dessa, samt om vilken regex-funktionalitet som finns i Python i allmänhet, [här](#).

Uppgifterna är ungefär sorterade efter kursledarens subjektiva åsikt om deras svårighetsgrad, och Kattis kommer att testa uppgifterna i samma ordning. När du är klar med första uppgiften kan du alltså skicka in din lösning och se om du klarar alla testfall som hör till första uppgiften, och så vidare.

Uppgifter

1. DNA

Skriv ett regex som matchar en sträng om och endast om den är en DNA-sekvens, dvs bara består av tecknen ACGT (endast stora bokstäver, ej acgt).

2. Sorterade tal

Skriv ett regex som matchar en sträng över tecknen 0-9 om och endast om tecknen strängen är sorterade i fallande ordning. Till exempel ska “42”, “9876543210”, och “000” matchas, men “4711”, “11119”, “123”, och “777a” inte matchas.

3. Sök efter given sträng – del 1

Skriv ett regex som matchar en sträng s om och endast en given söksträng x förekommer som *delsträng* i s . Om söksträngen x är “progp” ska alltså t.ex. strängarna “popororpopogpepor” och “progprog” matchas, men inte “PROGP”, “programmeringsparadigm”, eller “inda”. Du kan anta att indatasträngen x bara består av bokstäver och siffror.

4. Sök efter given sträng – del 2

I den här uppgiften kan du ha användning av metoden `string.join` (exempel [här](#)).

Skriv ett regex som matchar en sträng s om och endast en given söksträng x förekommer som *delsekvens* i s , dvs om vi genom att ta bort några tecken ur s kan bilda x . Om söksträngen x är “progp” ska alltså alla strängar som matchade i exemplet för del 1 fortfarande matcha, men nu ska

även t.ex. “programmeringsparadigm” och “p r o g p” matcha (men inte “inda” eller “poprg”). Du kan anta att indatasträngen x bara består av bokstäver och siffror.

5. Ekvationer utan parenteser

Eftersom reguljära uttryck (och även regex) inte kan användas för att kolla om en uppsättning parenteser är balanserade så kan vi inte skriva regex för att matcha allmänna ekvationer. Men vi kan skriva ett regex för att matcha aritmetiska uttryck och ekvationer som inte tillåts innehålla parenteser, och det ska vi göra nu.

De aritmetiska uttrycken vi vill matcha består av ett eller flera heltal, åtskiljda av någon av operatorerna för de fyra räknesätten: +, -, *, /. Heltalen kan ha inledande nollor (matchande exempel 4 nedan). I början av ett uttryck kan det finnas ett plus- eller minustecken för att explicit säga att första talet är positivt eller negativt (matchande exempel 2, 3, 5 nedan), men vi tillåter inte detta på tal i mitten av uttryck (icke-matchande exempel 2 nedan). En ekvation är två uttryck separerade av ett likhetstecken. Bara ett likhetstecken kan förekomma (icke-matchande exempel 4 nedan).

| Strängar som ska matchas | Strängar som inte ska matchas |
|--------------------------|-------------------------------|
| 1589+232 | 5*x |
| -12*53+1-2/5 | 18/-35 |
| 18=+17/25 | *23 |
| 000=0 | 7=7=7 |
| +1+2+3=-5*2/3 | 3.14159265358 |

6. Parenteser med begränsat djup

Reguljära uttryck kan inte användas för att beskriva balanserade parentesuttryck i allmänhet, men om vi begränsar oss till parentesuttryck med begränsat djup kan vi göra det. Med “djupet” för ett parentesuttryck menar vi det maximala antalet nästlade parentespar. Djupet för “()” är 1, och djupet för “()()()” är 3.

Skriv ett regex för att känna igen balanserade parentesuttryck som har djup högst 5. Till exempel ska strängarna “()()”, “((((())))”, “()((()()))” matcha, men strängarna “()()”, “((((())))” och “(x)” inte matcha.

Tänk på att “(” och “)” har speciell betydelse i regex, och att du måste använda “\” och “\)” för att matcha vänster- och höger-parentestecken.

7. Sorterade tal igen

Skriv ett regex som matchar en sträng över tecknen 0-9 om och endast om det finns tre intilliggande siffror någonstans i talet som är sorterade i strikt stigande ordning. Till exempel ska “123”, “9876456000”, “123456789” och “91370” matcha, men “111”, “415263”, “xyz123xyz” ska inte matchas.

(Tips: börja med att skriva ett reguljärt uttryck för tre siffror i stigande ordning där den mittersta siffran är t.ex. “4”, och fundera sedan på hur detta kan användas.)

Att diskutera vid redovisning.

Kan vi göra en variant av lösningen på uppgift 4 där glappen mellan bokstäverna måste vara lika långa? Isåfall ungefär hur? I denna variant skulle “p123r123xyzgooop” alltså innehålla söksträngen “prog” eftersom den återfinns med ett glapp på 3 tecken mellan varje bokstav i söksträngen, men “p123r123o123g12p” skulle inte anses innehålla “prog” eftersom glappet mellan “g” och “p” inte är lika stort som övriga.

Kan vi kombinera lösningarna för uppgifterna 5 och 6 för att skriva ett regex för att matcha aritmetiska uttryck och ekvationer som tillåts innehålla parentesuttryck upp till ett begränsat djup? Isåfall ungefär hur?

Kan vi generalisera lösningen på uppgift 7 och skriva ett regex som matchar strängar med fyra, intilliggande siffror istället för tre? Och vidare till fem, sex, etc intilliggande sorterade siffror? Isåfall ungefär hur?

Labb S2: Sköldpaddegrafik

Problem-ID på Kattis: [kth:progp:s2](#)

I denna labb ska du implementera en parser för ett enkelt programmeringsspråk för grafik, baserat på det klassiska programmeringsspråket **Logo** (som du inte behöver känna till sedan tidigare eller ens när du är klar med den här labben). Du får använda vilket programmeringsspråk du vill bland de som finns på Kattis, men du får inte använda inbyggda bibliotek/verktyg för att konstruera parsers (t.ex. DCG i Prolog) utan ska implementera denna “från scratch” med rekursiv medåkning. Däremot får du gärna använda inbyggda bibliotek/verktyg för reguljära uttryck för din lexikala analys, om du vill (även om språket som ska parsas är såpass enkelt att det inte finns något egentligt behov av det).

Sköldpaddegrafik

Till vår hjälp har vi en sköldpadda (låt oss kalla den Leona) med en penna. Vi kan instruera Leona att gå till olika platser och linjer ritas då längs vägen Leona går. Instruktionerna till Leona ges som “program” i Leona-språket. Språket har följande instruktionsuppsättning:

| | |
|----------------|---|
| FORW d | Leona går framåt d punkter (för ett <i>positivt</i> heltal d). |
| BACK d | Leona går bakåt d punkter (för ett <i>positivt</i> heltal d). |
| LEFT θ | Leona svänger vänster θ grader (för ett <i>positivt</i> heltal θ), utan att flytta sig från sin nuvarande position. |
| RIGHT θ | Leona svänger höger θ grader (för ett <i>positivt</i> heltal θ), utan att flytta sig från sin nuvarande position. |
| DOWN | Leona sänker ned pennan så att den lämnar spår efter sig när Leona rör sig |
| UP | Leona höjer pennan så att inget spår lämnas när Leona rör sig |
| COLOR c | byter färg på pennan till färgen c . Färg specas på hex-format, t.ex. #FFA500 för orange (om du är osäker på vad detta innebär så är din favoritsökmotor din vän, som vanligt). |
| REP r <REPS> | Leona upprepar <REPS> r gånger (för ett <i>positivt</i> heltal r). <REPS> är en sekvens av en eller flera instruktioner, omgivna av citationstecken (“”). Om sekvensen bara består av en enda instruktion är citationstecknen valfria. |

Språket är case insensitive – i både kommando-namn och beskrivning av färger kan små och stora bokstäver blandas. Kommandon i språket avslutas med punkt (‘.’), med undantag för REP-kommandon, efter dessa har man inga punkter (däremot ska varje kommando i REP-sekvensen avslutas med punkt). Kommentarer kan skrivas i språket med procenttecken (‘%’), allt som står efter ett procenttecken på en rad anses vara en kommentar. All whitespace (mellanslag, tabbar och nyradstecken) är ekvivalent förutom i kommentarer (där nyrad betyder “slut på kommentar”). Det måste finnas whitespace mellan ett kommando och dess parameter (t.ex. mellan RIGHT och θ), i övrig är all whitespace optional.

Leona startar på positionen (0,0) och är vänd i riktning mot punkten (1,0). Pennan är initialt blå (#0000FF) och i upphöjt läge.

Notera att även om alla indataparametrar är heltal så kan Leona hamna på koordinater som inte är heltal. Om vi t.ex. från startläget utför LEFT 30 . FORWARD 2 . kommer Leona att befinna sig på positionen $(\sqrt{3}, 1) \approx (1.732, 1)$.

Uppgifter

Det övergripande målet med uppgiften är att skriva en *översättare* för Leona-språket, som översätter ett program på Leona-språket till en lista med linjesegment givna i kartesiska koordinater. För varje instruktion där Leona går framåt eller bakåt och pennan är nedsänkt ska du alltså konstruera ett linjesegment från punkten (x_1, y_1) där Leona startar till punkten (x_2, y_2) där Leona stannar.

För att göra detta ska du utföra följande uppgifter:

1. Konstruera en formell grammatik för Leona-språket. Detta inkluderar att välja en lämplig nivå på hur indatafilen ska delas upp i tokens, d.v.s. vilka delar av parsningen som ska skötas i det lexikala analys-steget, och vilka delar som ska skötas av själva parsern. Senare ska en rekursiv medåknings-parser skrivas enligt grammatiken, så försök se till att grammatiken är lämpad för detta (annars kommer den antagligen behöva modifieras).
2. Som första steg i en parser för din grammatik, skriv en *lexikal analysator* som delar upp indatafilen i tokens.
3. Skriv en *parser* för Leona-språket med rekursiv medåkning. Parsern ska ta sekvensen av tokens som produceras av den lexikala analysatorn, och producera ett syntaxträd.
4. Skriv kod för att *exekvera* det givna programmet genom att översätta det syntax-träd som produceras av parsern till en lista med linjesegment.
5. Slå ihop lexikal analys, parsning, och exekvering till ett fullständigt program som läser ett Leona-program och konstruerar linjesegmenten. Se nedan för detaljerad information om hur indata ska läsas och utdata skrivas.

Observera att dessa del-uppgifter ska ses som krav. Du **ska** konstruera en grammatik för språket, du **ska** skriva en parser som använder rekursiv medåkning, och du **ska** separera de olika stegen (lexikal analys, parsning, exekvering).

Vägledning

För att göra översättningen behöver du kunna beräkna vilken position Leona befinner sig på. Låt oss påminna om följande grundläggande trigonometriska faktum: om Leona befinner sig på koordinaterna (x, y) , är vänd i riktning v (antal grader moturs från rakt högerut), och går d punkter framåt, så kommer Leonas nya position att vara $(x + d \cos(\pi v/180), y + d \sin(\pi v/180))$

Det kan vara lockande att angripa den här uppgiften genom att helt enkelt börja koda. Tips: gör inte det! Om du börjar i rätt ände, med att konstruera en grammatik och göra uppgiften “by the book”, löper du mycket mindre risk att fastna på någon av de många detaljer och knepigheter som finns i uppgiften, vilket kommer bespara dig tid i det långa loppet.

Indata

Indata består av ett Leona-program och ges på standard input (`System.in` in Java – vid behov, se Kattis-hjälpen för information om vad detta betyder).

Du kan anta att om det givna programmet är syntaktiskt korrekt så kommer alla alla tal som är obegränsade i språkdefinitionen (avståndsparametrar d och repetitionsparametrar r) vara högst 10^5 , och att det totala antalet instruktioner som utförs när programmet körs kommer vara högst $2 \cdot 10^5$ (dessa är alltså garantier på indata, inget du behöver kontrollera).

Indatafilen är högst 1 MB stor.

Utdata

- Om det givna Leona-programmet är syntaktiskt felaktigt ska följande meddelande skrivas ut, där r är den rad på vilken (första) syntaxfelet finns:

Syntaxfel på rad r

Se förtydliganden i exempel-fallen nedan om vilken rad som anses vara raden för första syntaxfelet.

- Annars, om det givna programmet är syntaktiskt korrekt, ska en lista med linjesegment som ritas av programmet skrivas ut. Segmenten ska skrivas ut i samma ordning som de ritas av Leona, och varje segment skrivs ut på en ny rad, på följande format:

$c \ x_1 \ y_1 \ x_2 \ y_2$

Här är c färgen linjesegmentet har (i hex-format precis som i språket), (x_1, y_1) är startpunkten för linjesegmentet (den punkt där Leona började när segmentet ritades) och (x_2, y_2) är slutpunkten för linjesegmentet (den punkt där Leona slutade). Koordinaterna ska vara korrekta upp till en noggrannhet på 10^{-3} (om `double`-variabler används och skrivs ut dem med 4 eller fler decimaler ska detta inte orsaka några bekymmer).

Sample Input 1

```
% Det här är en kommentar
% Nu ritar vi en kvadrat
DOWN.
FORW 1. LEFT 90.
FORW 1. LEFT 90.
FORW 1. LEFT 90.
FORW 1. LEFT 90.
```

Sample Output 1

```
#0000FF 0.0000 0.0000 1.0000 0.0000
#0000FF 1.0000 0.0000 1.0000 1.0000
#0000FF 1.0000 1.0000 0.0000 1.0000
#0000FF 0.0000 1.0000 0.0000 0.0000
```

Sample Input 2

```
% Space runt punkt valfritt.
DOWN . UP.DOWN. DOWN.
% Rader kan vara tomma

% radbrytning/space/tabbar för
% att göra koden mer läslig.
REP 3 "COLOR #FF0000.
    FORW 1. LEFT 10.
    COLOR #000000.
    FORW 2. LEFT 20."
% Eller oläslig
    COLOR
% färgval på gång
#111111.
REP 1 BACK 1.
```

Sample Output 2

```
#FF0000 0.0000 0.0000 1.0000 0.0000
#000000 1.0000 0.0000 2.9696 0.3473
#FF0000 2.9696 0.3473 3.8356 0.8473
#000000 3.8356 0.8473 5.3677 2.1329
#FF0000 5.3677 2.1329 5.8677 2.9989
#000000 5.8677 2.9989 6.5518 4.8783
#111111 6.5518 4.8783 6.5518 3.8783
```

Sample Input 3

```
% Syntaxfel: felaktig färgsyntax
COLOR 05AB34.
FORW 1.
```

Sample Output 3

```
Syntaxfel på rad 2
```

Sample Input 4

```
% Oavslutad loop
REP 5 "DOWN. FORW 1. LEFT 10.
```

Sample Output 4

```
Syntaxfel på rad 2
```

Sample Input 5

```
% Syntaxfel: ej heltal
FORW 2,3.
```

Sample Output 5

```
Syntaxfel på rad 2
```

Sample Input 6

```
%&(CDH*(
FORW
#123456.
&C(*N&(*#NRC
```

Sample Output 6

```
Syntaxfel på rad 3
```

Sample Input 7

```
% Måste vara whitespace mellan
% kommando och parameter
DOWN. COLOR#000000.
```

Sample Output 7

```
Syntaxfel på rad 3
```

Sample Input 8

```
% Syntaxfel: saknas punkt.
DOWN
% Om filen tar slut mitt i ett kommando
% så anses felet ligga på sista raden
% i filen där det förekom någon kod
```

Sample Output 8

```
Syntaxfel på rad 2
```

Sample Input 9

```
% Måste vara space mellan argument
REP 5"FORW 1."
% Detta inte OK heller
REP 5FORW 1.
```

Sample Output 9

```
Syntaxfel på rad 2
```

Sample Input 10

```
% Ta 8 steg framåt
REP 2 REP 4 FORW 1.
REP% Repetition på gång
2% Två gånger
"%Snart kommer kommandon
DOWN% Kommentera mera
.% Avsluta down-kommando
FORW 1
LEFT 1. % Oj, glömde punkt efter FORW-kommando
"
```

Sample Output 10

```
Syntaxfel på rad 9
```

Sample Input 11

```
% Nästlad loop 1
REP 2 "UP. FORW 10. DOWN. REP 3 "LEFT 120. FORW 1.""
% Nästlad loop 2
REP 3 "REP 2 "RIGHT 2. FORW 1."
      COLOR #FF0000. FORW 10. COLOR #0000FF."
% COLOR #000000. % Bortkommenterat färgbyte
BACK 10.
% Upper/lower case ignoreras
% Detta gäller även hex-tecknen A-F i färgerna i utdata,
% det spelar ingen roll om du använder stora eller små
% bokstäver eller en blandning.
color #AbCdEf. left 70. forW 10.
```

Sample Output 11

```
#0000FF 10.0000 0.0000 9.5000 0.8660
#0000FF 9.5000 0.8660 9.0000 0.0000
#0000FF 9.0000 0.0000 10.0000 0.0000
#0000FF 20.0000 0.0000 19.5000 0.8660
#0000FF 19.5000 0.8660 19.0000 0.0000
#0000FF 19.0000 0.0000 20.0000 0.0000
#0000FF 20.0000 0.0000 20.9994 -0.0349
#0000FF 20.9994 -0.0349 21.9970 -0.1047
#FF0000 21.9970 -0.1047 31.9726 -0.8022
#0000FF 31.9726 -0.8022 32.9671 -0.9067
#0000FF 32.9671 -0.9067 33.9574 -1.0459
#FF0000 33.9574 -1.0459 43.8601 -2.4377
#0000FF 43.8601 -2.4377 44.8449 -2.6113
#0000FF 44.8449 -2.6113 45.8230 -2.8192
#FF0000 45.8230 -2.8192 55.6045 -4.8983
#0000FF 55.6045 -4.8983 45.8230 -2.8192
#ABCDEF 45.8230 -2.8192 51.1222 5.6613
```