DD1362 Programmeringsparadigm

Laborationer 2022

Innehåll

Senast uppdaterad: 19 januari 2022

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 Githubinstallation 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 definierar ska förklaras.

Labb F1: Uppvärmning i Haskell

Problem-ID på Kattis: kth.progp.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. Man ska klara minst $F(73)$ lokalt för att få godkänt.

Tips Att endast implementera rekursionen som den är given blir för långsamt.

Exempel

`fib(7)` ska returnera 13

`fib(17)` ska returnera 1597

2 Rövarspråket

I *rövarspråket* dubblerar 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 åääö. 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("fofunonkoktotionon")` 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 definierar 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 "++"-operatorn!).

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, aminosyrorna, som brukar representeras med bokstäverna `ARNDCEQGHILKMFPSTWYV`.[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.

[1] Borde inte aminosyrornas förkortningar ARNDCEQGHILKMFPSTWYV stå i bokstavsordning? Det gör de: A, R, och N representerar till exempel aminosyrorna 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.

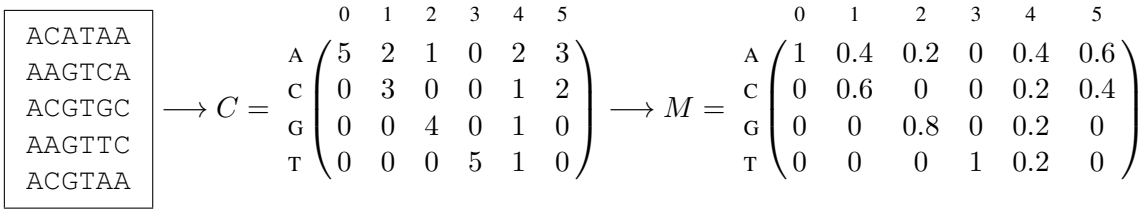
   Om man försöker jämföra DNA med protein ska det signaleras ett fel med hjälp av funktionen error.

   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.



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

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

```
nucleotides = "ACGT"
aminoacids = sort "ARNDCEQGHILKMFPSTWYV"

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 profilmatris

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 (`ARNDCEQGHILKMFPSTWYV`).

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 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 *avståndsmatris*. 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 avståndsmatris 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 tripplar 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. e.ex.: om vi har fyra `MolSeq`-objekt A, B, C, D och skickar in listan $[A, B, C, D]$, så ska `distanceMatrix` returnera listan

   $$[(A, A, \cdot), (A, B, \cdot), (A, C, \cdot), (A, D, \cdot), (B, B, \cdot), (B, C, \cdot), (B, D, \cdot), (C, C, \cdot), (C, D, \cdot), (D, D, \cdot)]$$

   (fast med samtliga "$\cdot$" utbytta mot avståndet mellan respektive objekt).

Labb F3: Monader

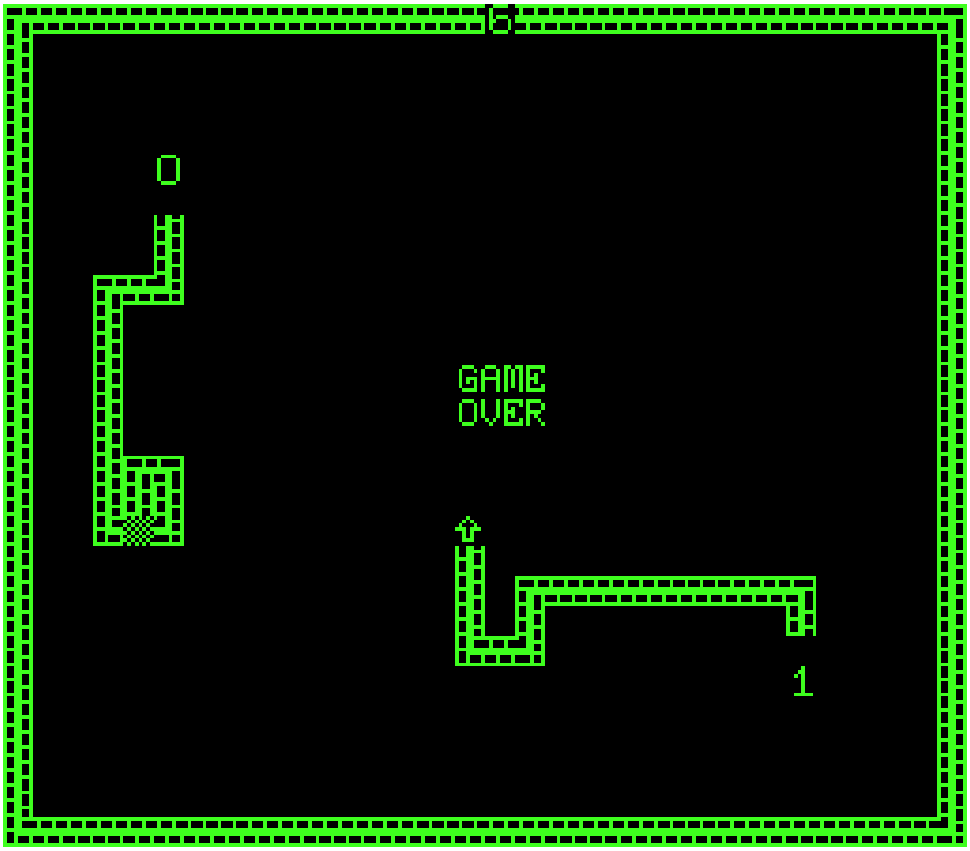
Denna labb testas inte på Kattis.

Många tror att Haskell är för begränsat för att göra ett komplett program med indata och grafisk presentation så nu är det dags att bevisa att de har fel med ett enkelt spel och samtidigt få träna på Monader, ett design pattern som används för att hantera I/O och state.

1 Bakgrund

1976 släppte Gremlin arkadspelet Blockade. Spelet börjar med att visa en rektangulär spelplan indelad i rutor där spelarnas startpositioner representeras med en pil som är riktad uppåt, nedåt, till höger eller till vänster. I varje tidssteg byggs en mur vid pilens position och pilen tar ett steg i sin riktning. Spelarna kan genom att trycka på en av fyra knappar ändra riktning på pilen och målet är att bygga en så lång mur som möjligt utan att muren kör in i den omslutande rektangeln, den andra spelarens mur, eller sig själv. Om du behöver ytterligare beskrivning av spelet så finns det filmer på youtube.

Spelet har inspirerat andra spel som Snake och Tron men den här labben handlar om att implementera en enkel version av originalet.



Figur 1: Spelet Blockade från 1976 (Bildkälla: Wikimedia commons).

2 Tillåtna förenklingar från originalet

Spelet behöver inte ha avancerad grafik som originalet utan ASCII-grafik går utmärkt. Murar kan representeras med ett '#'-tecken och pilens olika riktningar kan representeras med lämpliga tecken som mindre än, större än, hustak och bokstaven "v". Färger är överkurs. Det är varken nödvändigt eller önskvärt att implementera spelets soundtrack. Game Over behöver inte skrivas ut på spelplanen, men det behöver skrivas ut längst ner vid avslut. Spelet kan se ut som illustrationen på nästa sida.

Använd ncurses, hscurses eller ett liknande bibliotek för att ge spelet ett pseudografiskt användargränssnitt. Det är tillåtet men inte nödvändigt att använda OpenGL eller liknande för att ge spelet ett grafiskt användargränssnitt.

3 Krav

Vissa förenklingar gör uppgiften såpass enkel att man inte behöver förstå hur man använder monader fullt ut, så de är förbjudna. Exempel nedan.

- Det är inte tillåtet att kräva att användaren trycker på en tangent för att driva spelet framåt. När spelet startats ska murarna växa i en riktning tills någon av spelarna trycker i en annan riktning.
- Det är inte tillåtet att kräva att spelarna trycker på enter eller retur efter sin tangentnedtryckning
- Det är inte tillåtet att skriva ut hela spelplanen om och om igen i terminalen för att visa förändringen. Använd ncurses, hscurses eller liknande.

4 Inför redovisning

Här är en checklista inför redovisning:

- Var beredd att visa upp spelet i en terminal. Spela en kort match med din labbkamrat eller labbassistenten.
- Var beredd på att förklara all kod som kör spelet.
- Se till att alla funktioner är väldokumenterade och klokt namngiva.

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 sökträng 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 sökträng 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 sökträng 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 sökträng x förekommer som *delsträng* i s. Om sökträngen x är "progp" ska alltså t.ex. strängarna "popororpoprogpepor" och "progprogp" 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 sökträng x förekommer som *delsekvens* i s, dvs om vi genom att ta bort några tecken ur s kan bilda x. Om sökträngen 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 “p123r123oxyzgoooop” alltså innehålla sökströngen “progp” eftersom den återfinns med ett glapp på 3 tecken mellan varje bokstav i sökströngen, men “p123r123o123g12p” skulle inte anses innehålla “progp” 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 intill-liggande 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 vars syfte är 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). Du får gärna utgå ifrån Java-kodexemplen för rekursiv medåkning från föreläsningarna om du vill.

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 *positivt* heltal d).
`BACK` d	Leona går bakåt d punkter (för ett *positivt* heltal d).
`LEFT` θ	Leona svänger vänster θ grader (för ett *positivt* heltal θ), utan att flytta sig från sin nuvarande position.
`RIGHT` θ	Leona svänger höger θ grader (för ett *positivt* 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 *positivt* 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), samt mellan r-argumentet till `REP` och `<REPS>`-delen. I övrigt är all whitespace godtycklig.

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. För att hjälpa till på traven med detta ger vi ett förslag på hur indata borde styckas upp i tokens nedan. Du behöver inte följa detta förslag om du inte vill men om du har problem att få din lösning korrekt rekommenderas att du följer denna vägledning. Grammatiken behöver uttryckas på BNF-form eller liknande för godkänt. 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

Trigonometri: 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 $(x + d\cos(\pi v/180), y + d\sin(\pi v/180))$

Lexikal analys: Ett förslag på token-typer att använda för tokenisering (lexikal analys) av indata är att använda följande 13 token-typer:

- FORW, BACK, LEFT, RIGHT, DOWN, UP, COLOR, REP: tokens som representerar respektive kommando.
- PERIOD, QUOTE: token som representerar punkt respektive citat-tecken.
- DECIMAL: token som representerar ett positivt heltal.
- HEX: token som representerar en hex-färg.
- ERROR: allt annat innehåll – token som representerar ett syntax-fel på lexikal nivå.

Det kan vara lockande att även ha med en token-typ för whitespace (eftersom detta måste finnas på vissa ställen), men detta tenderar att leda till obegripliga buggar. Istället kan tokens för de kommandon som har argument definieras som "kommando-namnet följt av ett whitespace". En sträng som "`FORW 12`" skulle då ge token-sekvensen FORW, NUMBER, medan strängen "`FORW12`" skulle ge ett ERROR-token. Kravet på whitespace mellan r-argumentet till `REP` och `<REPS>`-delen kan hanteras på ett liknande sätt men är lite mer komplicerat. Eftersom vi i den lexikala analysen inte ser skillnad på ett tal som är ett argument till t.ex. `FORW` och ett tal som är ett argument till `REP` så måste vi tillåta att ett tal följas av andra saker än whitespace, men vissa kombinationer som t.ex. "`42"`" och "`42F`" borde resultera i ett ERROR-token.

Rekursiv medåkning: Kom ihåg att en rekursiv medåkningsparser ska ha en funktion för varje icke-slutsymbol i din grammatik. Dessa funktioner ska inte ta några argument, och returnera ett parseträd. Om du börjar skriva funktioner som tar in olika argument som håller reda på någonting om vad som hänt hittills (t.ex. en boolean-flagga som anger huruvida vi befinner oss inne i en REP-sats eller inte) så har du gjort fel, och kommer behöva göra om din lösning! Detsamma gäller om du istället för funktionsargument har lite globala variabler eller klass-variabler som håller reda på något slags tillstånd. Det *enda* tillstånd (state) som får finnas i parsern är ett Lexer-objekt som håller reda på vilket indata-token vi för närvarande befinner oss på.

Gör delarna i rätt ordning: 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` i 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 svaret skrivs ut med 4 eller fler decimaler ska det inte bli avrundningsfel).

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/tabb 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
```

Sample Input 12

```
DOWN .
% OBS! Denna fil har ett
% tabb-tecken ('\t') i första
% REP-satsen.  Om du kör den
% genom att göra copy-paste
% kommer tabb-tecknet inte
% testas korrekt.
REP     2 REP % raaadbryyyt
1 "REP 2 REP 2 "FORW 1."
 LEFT 45."
```

Sample Output 12

```
#0000FF 0.0000 0.0000 1.0000 0.0000
#0000FF 1.0000 0.0000 2.0000 0.0000
#0000FF 2.0000 0.0000 3.0000 0.0000
#0000FF 3.0000 0.0000 4.0000 0.0000
#0000FF 4.0000 0.0000 4.7071 0.7071
#0000FF 4.7071 0.7071 5.4142 1.4142
#0000FF 5.4142 1.4142 6.1213 2.1213
#0000FF 6.1213 2.1213 6.8284 2.8284
```

Labb S3: Automatanalys

Problem-ID på Kattis: kth.progp.s3

I den här labben ska du skriva kod för att analysera en ändlig automat (deterministic finite automaton, DFA). Mer specifikt ska du skriva ett program som genererar "alla" (upp till en övre gräns) strängar som accepteras av en given automat.

Labben är primärt avsedd att utföras i Java. Det är möjligt att använda något annat språk men i detta fall kommer du behöva göra lite mer saker själv (se avsnittet "Använda annat språk" nedan).

Du ska implementera en klass `DFA` som representerar en automat. Klassen ska ha följande metoder:

- `public DFA(int stateCount, int startState);`
  Konstruktor som skapar en automat med `stateCount` antal tillstånd, där tillstånd nummer `startState` är starttillstånd. Tillstånden numreras från 0 till `stateCount` − 1.
- `public void setAccepting(int state);`
  Anger att tillståndet `state` är ett accepterande tillstånd.
- `public void addTransition(int from, int to, char sym);`
  Anger att det finns en övergång från `from` till `to` med tecknet `sym`.
- `public List<String> getAcceptingStrings(int maxCount);`
  Metod som returnerar upp till `maxCount` olika strängar som automaten accepterar. Om automaten accepterar färre (eller lika med) `maxCount` strängar ska alla strängar som automaten accepterar returneras. Om automaten accepterar fler strängar ska exakt `maxCount` olika strängar returneras. I det senare fallet får metoden i princip returnera vilka accepterande strängar som helst (det behöver t.ex. inte vara de första i alfabetisk ordning, eller något sådant), men av tekniska skäl får de returnerade strängarna inte vara allt för långa (se "Begränsningar" nedan). Listan som returneras behöver inte vara sorterad, strängarna kan returneras i godtycklig ordning.

Testkod

I ditt git-repo för labben finns ett kod-skelett, inklusive en main-klass och några testfall, som du kan använda för att provköra din lösning. Mer information finns i den medföljande README-filen.

Till Kattis ska du bara skicka in din lösning `DFA.java`, inte main-klassen `Main.java` eller inläsningsrutinerna i `Kattio.java` (som main-klassen använder sig av).

Använda annat språk

Om du väldigt gärna vill använda något annat språk än Java så är det tillåtet (men det måste vara ett språk som finns på Kattis). I detta fall behöver du själv hantera inläsning av automaten och utskrift av svaret genom att konvertera `Main.java` från Java-skelettet till det språk du vill använda.

Begränsningar

I Kattis-testerna kommer automaten inte att ha mer än 50 tillstånd, och parametern `maxCount` kommer inte överstiga 1000. De tecken som kommer användas i Kattis-testerna är `a-z`, `A-Z`, och `0-9`. (Dessa begränsningar är inte något ni ska hård-koda i er lösning, utan en vägledning om ni har problem att bli godkända i Kattis och vill begränsa vad ni testar.)

Strängarna som `getAcceptingStrings` returnerar får vara högst 5000 tecken långa (detta kommer alltid vara tillräckligt med väldigt god marginal).

Du kan anta att alla anrop till din klass är korrekta och *behöver inte* skriva någon speciell felhantering. T.ex. kommer alltså parametrarna `from` eller `to` i `addTransition` alltid ges värden som är mellan 0 och `stateCount` $-$ 1, och från varje tillstånd och tecken kommer det finnas högst en övergång från tillståndet för det tecknet.

Labb Inet: Internet/sockets

Denna laboration går ut på att implementera ett enkelt spel för minst två samtidiga spelare med varsin klient.

Syftet med labben är att studera internet-orienterad programmering och då speciellt kommunikation via sockets (det finns en officiell tutorial om sockets att läsa för den som är intresserad) och konstruktion av protokoll. Det är särskilt viktigt att du kan beskriva hur kommunikationen fungerar och därför är det viktigt att ni kan specificera protokollet som programmen använder i detalj.

De flesta studenter skriver denna labb i Python eller Java men det kan vara lärorikt att skriva i t.ex. Go eller Erlang som är konstruerade för att hantera parallellprogrammering. Erlang är däremot inte så bra på att hantera strängar och man kan behöva skriva en liten frontend i Java eller Python för användargränssnittet.

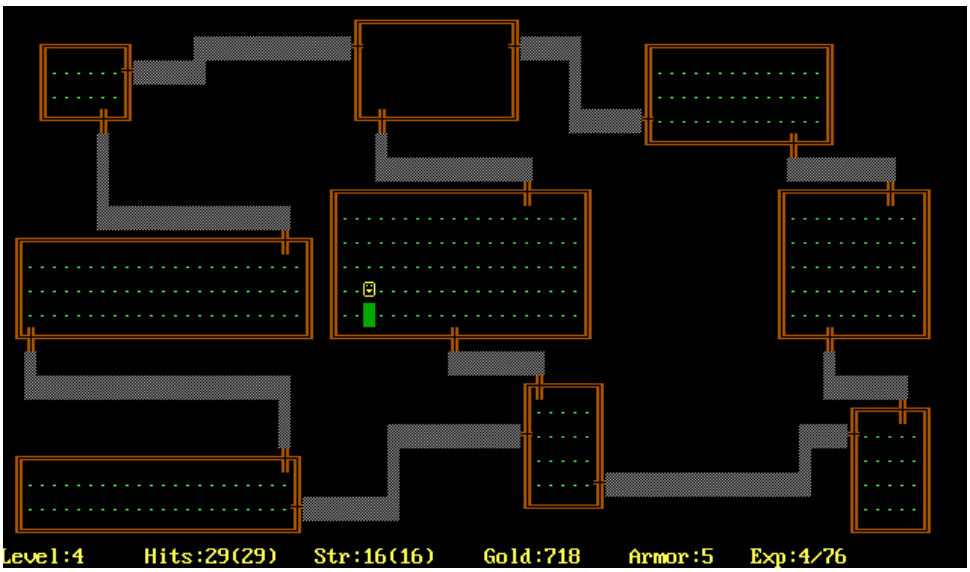
(Denna labb rättas inte på Kattis.)

Bakgrund

1980 kom spelet Rogue och det har gett upphov till en hel uppsjö med Rogueliknande spel. Att implementera en Rogueklon har i alla tider varit den mest klassiska av klassiska programmeringsuppgifter och nu har den letat sig till den här kursen också, men i mycket nedbantad form. Att vinna i Rogueliknande spel handlar om att besegra monster eller andra spelare, ta deras saker och använda dessa för att låsa upp hinder och ta sig vidare till någon slags mål. Vi vill inte begränsa er kreativitet för mycket i vad detta mål är, men samtidigt sätta vissa gränser så att spelen och protokollen varken blir för avancerade (så att uppgiften inte blir klar i tid) eller för enkla (så att de inte utmanar er att lära er kursinnehållet).

Uppgift

Uppgiften går ut på att implementera (1) klient och (2) server samt (3) ett väldokumenterat protokoll för hur klienten och servern kommunicerar med varandra. Skriv både klient- och serverdelen så att kraven på nästa sida uppfylls.



Figur 1: Bild ur spelet Rogue från 1980 (Källa: Wikimedia commons).

Krav

Klienten och servern behöver uppfylla följande krav.

Systemet

1. Spelarna behöver kunna flyttas omkring i 4 riktningar genom att trycka på piltangenterna eller klicka på knappar i ett gränssnitt.
2. Det ska finnas föremål som kan plockas upp och då ska dessa försvinna från spelplanen för övriga spelare. Åtminstone vissa föremål behöver vara relevanta för målet.
3. Spelarna behöver kunna bära på saker som plockats upp från spelbrädet.
4. Det ska finnas hinderrutor '#'. Om en spelare försöker flytta sig in i ett hinder så ska spelet fortsätta men utan att utföra förflyttningen.
5. Spelarna blockerar varandra, så om en spelare försöker gå dit som en annan spelare redan står så ska spelet fortsätta men utan att utföra förflyttningen.
6. Det ska finnas ett mål med spelet.
   (a) Målet måste involvera båda spelarna.
   (b) Målet får till exempel vara att besegra den andra spelaren.
   (c) Målet får till exempel bygga på samarbete.
   (d) När målet är uppfyllt ska spelet avslutas med en text (eventuellt olika) till båda spelarna.
7. Koden ska struktureras i funktioner och klasser om du använder det. Ingen funktion eller klass får bli för stor (mört, mört, mört, blåval-mönstret).
8. Koden ska dokumenteras bra.

Spelet bedöms helt på sina tekniska meriter. Ni behöver inte...

1. implementera inloggning eller någon form av persistens (databas).
2. implementera ett avancerat grafiskt användargränssnitt. Teckengrafik i ncurses går bra.
3. ha ett stort spel med flera våningar, olika nycklar och många olika monster.
4. skriva en story till spelet.

Det är inte tillåtet att...

1. använda web sockets för protokollet.
2. förenkla blockerande rutor eller spelare genom att göra dessa till omedelbara vinster eller förluster.
3. förenkla målet så att en spelare kan göra allt som krävs för att vinna utan att interagera med någon annan spelare.
4. förenkla labben genom att bara göra labb F3 spelbar via Internet.

Protokollet

Protokollet ska dokumenteras väl. Dokumentationen ska:

1. Innehålla en utförlig beskrivning av datan som skickas mellan server och klient.
2. Innehålla ett tillståndsdiagram (state diagram) som beskriver vilka tillstånd klienten och servern kan ha samt vad som gör att de övergår från ett tillstånd till ett annat.
3. Vara så detaljerat att en godtycklig student som har klarat kursen kan bygga en ny klient som fungerar med din server utifrån endast protokollet.

Protokollet ska dessutom uppfylla följande krav:

1. Protokollet ska antingen skicka text (ASCII, ISO8859-1 eller helst UTF8) eller vara binärt.
2. Protokollet måste vara på operativsystemsnivå och inte på webbnivå (web sockets är en helt annan sorts utmaning än denna labb).
3. Ha viss datasäkerhet. Det ska inte gå att krascha servern (eller klienten) genom att skicka trasig data. Ni behöver inte skydda er mot timingattacker.
4. Protokollet får inte vara programspråkspecifikt. Det är till exempel inte tillåtet att förvänta sig att mottagaren automatiskt kan deserialisera ett Python-objekt.
5. Servern behöver ha viss robusthet. Den ska till exempel kunna hantera att en klient disconnectar.
6. Protokollet ska delas upp i olika funktioner/metoder.

Tips

1. Använd ncurses för klienten och en enkel meny för servern. Servern kan logga med utskrifter i terminalen.
2. Låt klienten vara en tunn klient och lägg istället mer logik på servern. Det är tillåtet att strömma all grafik från servern.

Vid redovisning

1. ... ska du ha tre terminalfönster uppe med en server och minst två spelklienter som körs.
2. ... ha ditt protokoll redo för att kunna visa upp det.
3. ... ska du kunna redogöra för olika tillstånd i klient och serverdelen.
4. ... ska du ha en editor uppe med all källkod.
5. ... ska du kunna redogöra för alla detaljer i koden.

Labb X1: Jämförelse av Paradigm

I denna labb ska du lösa flera olika mindre uppgifter, i flera olika programmeringsparadigm.
De tre uppgifter du ska lösa är följande Kattis-uppgifter:

- Bus Numbers
- Calculating Dart Scores
- Peragrams

De tre paradigm som du ska använda är:

- imperativ/objektorienterad programmering, där du får välja mellan C, C++, Java och Rust.
- funktionell programmering, i form av Haskell.
- logikprogrammering, i form av Prolog.

Du ska lösa ***var och en av de tre uppgifterna i minst två paradigm***, och du ska använda ***var och en av de tre paradigmen för att lösa minst två uppgifter***. Totalt ska du alltså att ha skrivit minst 6 st. program (men de är alla ganska små).
Om ni arbetar tillsammans, observera följande: som vanligt är grundprincipen att ni ska lösa uppgifterna tillsammans (inte bara dela upp de olika uppgifterna sinsemellan och göra dem på varsitt håll), och som vanligt gäller att båda i gruppen ska ha full koll på all kod som ni vill redovisa.

Sidospår: Prolog

I/O

I/O i Prolog kan vara lite krångligt, och är inte tänkt att vara fokus för den här labben. Därför tillhandahålls en fil kattio.pl som innehåller inläsningsrutiner för att läsa in tal och strängar. Filen innehåller dokumentation om hur man använder den, men för att ge ett exempel tillhandahålls även en lösning på Kattis-problemet A Different Problem här: different.pl.
(Rutinerna i kattio.pl går säkert att förbättra så att de blir lite snabbare, även om det inte är relevant för de tre uppgifterna i denna labb. Om du tycker detta är kul och kommer på sätt att förbättra rutinerna får du gärna informera Per om det!)

Kompilering

För att kompilera ett prolog-program till en körbar binär på samma sätt som Kattis gör kan man skriva följande i terminalen:

```
swipl -O -q -g main -t halt -o {binär} -c {källkods-filer}
```

Där `{binär}` är vad man vill att binären som man sedan kör ska heta, och `{källkods-filer}` är en lista med de filer man vill kompilera. Den körbara binären kan man sedan provköra på sedvanligt sätt från terminalen, och använda "<" för att dirigera in en indatafil som indata till programmet.

Dokumentation

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

1. Namn på vem/vilka som skrivit den.

2. För vart och ett av de tre problemen:

   (a) Vilka paradigm du valde att använda på denna, och varför.

   (b) Kort reflektion över resultatet – var det mycket enklare att lösa problemet i det ena paradigmet, isåfall varför, kunde man ta lösningen i ett av språken och direkt översätta till ett annat, etc.

3. Avslutningsvis en kort allmän reflektion över labben – styrkor/svagheter hos de olika paradigmen, etc.

Rapporten kan vara en enkel txt-fil eller en pdf. *Word-dokument eller liknande format är ej OK*. En typisk längd är 1-2 sidor men det är inget krav att rapporten måste hålla sig inom detta intervall .

Labb X2: Programspråkstidsresa

I denna labb ska vi titta lite på hur programmeringsspråken utvecklats över tiden, genom att skriva program för en uppgift (beräkning av så kallade Bernoulli-tal) i några olika språk från olika tidsperioder.

Lite programspråksbakgrund

Vad som var den första datorn och det första programspråket beror lite på hur man definierar dessa termer. Man brukar säga att världens första programmerare var grevinnan Ada Lovelace, som på mitten av 1800-talet konstruerade en algoritm för att beräkna Bernoulli-tal på Charles Babbages Analytical Engine (den första mekaniska datorn – som dock aldrig byggdes utan stannade på koncept-stadiet).

Att konstruera algoritmer för Babbages Analytical Engine hade inte så mycket gemensamt med dagens algoritmkonstruktion. Det skulle dröja till 1940-talet, då de första elektroniska datorerna började dyka upp, tills det började dyka upp språk som liknar det vi idag kallar för programmeringsspråk.

En längre men aningen ironisk beskrivning av programmeringens historia finns här.

Bernoulli-tal

Din uppgift är att göra vad Ada Lovelace gjorde för Babbages Analytical Engine – att beräkna Bernoullital – men att göra det i några olika historiska programmeringsspråk.

Bernoulli-talen är en klassisk serie tal med många användningsområden. Även om dessa tal är har väldigt många intressanta egenskaper så behöver man inte någon faktiskt kunskap om dem för att göra labben, det enda man behöver veta är att det n:te Bernoulli-talet, B_n, kan beräknas med hjälp av följande pseudokod[2]:

```
1: function B(n)
2:     B[0] ← 1
3:     for m ← 1 to n do
4:         B[m] ← 0
5:         for k ← 0 to m − 1 do
6:             B[m] ← B[m] − BINOM(m + 1, k) · B[k]
7:         B[m] ← B[m]/(m + 1)
8:     return B[n]

9: function BINOM(n, k)
10:    r ← 1
11:    for i ← 1 to k do
12:        r ← r · (n − i + 1)/i
13:    return r
```

Funktionen BINOM beräknar de så kallade *binomialtalen* $\binom{n}{k}$. Om man inte redan känner till dessa kommer man att få lära sig mer om dem när man läser diskret matematik.

Implementerar man denna algoritm och anropar $B(4)$ så borde man få svaret ≈ -0.033333 ($-1/30$ för att vara mer exakt). Beroende på vilket språk och vilken datatyp för flyttal man använder får man overflow-problem när n blir större. Använder man double-variabler borde det hända någonstans runt $n = 30$.

[2] Det finns några olika uppsättningar Bernoulli-tal, denna algoritm beräknar vad som ibland kallas de "första Bernoullitalen".

Språk

Du ska implementera beräkning av Bernoulli-tal i följande språk. Exakt hur programmet ska fungera får du välja själv, det kan t.ex. skriva ut en tabell med de 20 första Bernoulli-talen, eller be användaren mata in n och sedan mata ut det n:te Bernoulli-talet.

Informationen nedan om språken är avsiktligt ganska sparsam om hur man kommer igång med språken (med undantag för COBOL). *En del av uppgiften är att själv söka reda på tillräckligt mycket info för att kunna komma igång och skriva ett enkelt program i respektive språk.*

50/60-tal: COBOL

COBOL (COmmon Business Oriented Language) utvecklades i slutet av 1950-talet och början av 1960-talet (även om viss modernisering och vidareutveckling skett sedan dess). Mycket ont har sagts om COBOL, det mesta välförtjänt, men det brukar sägas att det fortfarande idag existerar stora komplexa monolitiska system (ofta hos regeringar/militär/stora företag) skrivna för länge sedan i COBOL, för dyra för att byta ut mot något annat. Det producerades faktiskt, så sent som 2014, debattinlägg om att återinföra COBOL-undervisning[3].

Paradigm-mässigt är COBOL helt imperativt, och nästan direkt svårt att skriva strukturerad kod i (moderniseringar av språket har dock lagt till koncept som objekt-orientering).

Det finns många dialekter av COBOL. Ditt program ska fungera i OpenCobol/GNU Cobol, som finns installerat på i CSC:s Ubuntu-miljö (kommandot för kompilatorn är `cobc`). Wikipedia-sidan om GNUCobol har lite exempel för att komma igång. En stor mängd mer komplicerade exempel finns här (man behöver kommentera bort första raden i dessa exempel-program och istället använda kommandoradsparametern "`-free`" till `cobc`).

70-tal: Smalltalk

Smalltalk var ett av de allra första objekt-orienterade språken, och har haft ett stort inflytande på dagens moderna objekt-orienterade språk. Språket kan sägas ta objektorientering till absurdum – i Smalltalk är *allt* objekt, och alla beräkningar sker genom att meddelanden skickas mellan objekt.

T.ex. beräkningen "5 + 4" betraktas i Smalltalk som "skicka meddelandet '+' med parameter 4 till objektet 5".

Precis som när det gäller COBOL finns många olika varianter och implementationer av Smalltalk. I CSC:s Ubuntu-miljö finns GNU Smalltalk installerat, och ditt program ska funka i detta. Kompilatorn/interpretatorn startas med kommandot `gst`.

80-tal: Erlang

Erlang utvecklades på Ericsson under 1980-talet. Det har de senaste åren börjat bli populärt igen, mycket tack vare att samtidighet (eng. concurrency) är inbyggt i språket. Samtidighet gör det enkelt att använda för att skriva distribuerade program, något som ju numera med moln etc har blivit allt viktigare.

Syntaxmässigt lånar Erlang mycket från Prolog, men paradigmmässigt är det snarast ett funktionellt språk.

Erlang finns installerat i CSC:s Ubuntu-miljö, med kommandona `erl` och `erlc` (emulator och kompilator, på samma sätt som det finns `ghci` och `ghc` för Haskell).

[3] Skrivet av en Cobol-försäljare, så kanske mer underhållning än seriöst debattinlägg.

90-tal: PHP

PHP dök upp i mitten på 1990-talet och är främst ett språk för server-side-programmering av websidor. Paradigm-mässigt är det imperativt och objektorienterat. PHP är på många sätt vår tids Cobol. Till skillnad från de flesta andra moderna språk var det aldrig avsett att ens bli ett programmeringsspråk, och har växt fram lite hipp som happ utan tanke eller design. Allting från funktionsnamn till semantik är ofta godtyckligt, inkonsekvent och inte sällan förvånande (här finns några exempel). Det är svårt att programmera i PHP en längre tid utan att bli upprörd över språket. Många är de diatriber över språket man kan hitta online, två läsvärda exempel är The PHP Singularity och PHP: a fractal of bad design.

Trots detta är PHP idag ett av de vanligaste språken (om inte det vanligaste) för webb-server-programmering, antagligen för att det är väldigt lätt att komma igång med, och dess breda användning gör att det, precis som Cobol, antagligen aldrig helt kommer gå att ta kål på.

PHP finns installerat i CSC:s Ubuntu-miljö och PHP-program kan köras direkt i terminalen utan web, med kommandot `php`. Er lösning ska gå att köra på detta sättet snarare än via en web-sida.

00-tal: Clojure

Clojure dök upp första gången 2007 (första publika versionen. Första stabila versionen kom 2009). Det är ett funktionellt språk baserat på det klassiska språket Lisp (som skapades redan på 50-talet). Bland Clojures features kan nämnas transaktionellt minne. Det finns många bra videor online med Clojures skapare och evangelist Rich Hickey, se t.ex. Clojure Made Simple eller Clojure for Java Programmers Part 1.

Clojure finns installerat i CSC:s Ubuntu-miljö, med kommandot `clojure`.

10-tal: Rust

Rust såg dagens ljus 2010. Det är ett multiparadigmspråk som påminner mycket om C++ men dess minnesmodell använder en borrow checker för att uppnå jämförbar prestanda på ett mycket enklare sätt. Rust lär du dig enklast genom Rustboken.

Rust finns installerat i CSC:s Ubuntu-miljö, med kommandot `rustc`.

Dokumentation

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

1. Namn på vem/vilka som skrivit den.
2. För vart och ett av språken, en kort reflektion över implementationen – t.ex. vad som var lätt, vad som var svårt, vad du tyckte om språket, hur det skulle vara att göra ett större program/projekt i språket. (Själva algoritmen som ska implementeras är ju tämligen enkel, så det som kan vara svårt är typiskt mer språk/kompilator-specifika detaljer.)
3. Avslutningsvis en kort allmän reflektion över labben – sammanfattande intryck om de olika språken och ev. andra tankar.

Rapporten kan vara en enkel txt-fil eller en pdf. *Word-dokument eller liknande format är ej OK*. En typisk längd är 1-2 sidor men det är inget krav att rapporten måste hålla sig inom detta intervall.