TDP007 - Tenta

2022-03-24

Regler

- All kod efterrättas på denna tenta
- Vid toalettbesök ska vakt informeras.
- All form av kontakt mellan studenter under tentamens gång är strängt förbjuden.
- Böcker och anteckningssidor kan komma att granskas av assistent, vakt eller examinator under tentamens gång.
- Frågor om specifika uppgifter eller om tentamen i stort ska ställas via tentasystemets kommunikationsklient.
- Systemfrågor kan ställas till jourhavande genom att räcka upp handen
- Endast uppgifter inskickade före tentamenstidens slut rättas.

Hjälpmedel	En Rubybok (exempelvis the pickaxe) Ett A4-ark med egna anteckningar Tillgång till webresurser: ruby-docs, rubular och tidig version av kursboken

Information

Betygssättning vid tentamen

Tentamen består av ett antal uppgifter på varierande nivå. Uppgifter som uppfyller specifikationen samt följer god sed och konventioner krävs för maxpoäng på en uppgift. Avvikelser från ovanstående ger avdrag.

Tentan består av uppgifter (några indelade i deluppgifter) som totalt kan ge 50 poäng.

Ruby-docs

På tentan har du tillgång till referenssidorna på `https://ruby-doc.org/` (både core och std-lib finns tillgänglig) via webbläsaren.

Rubular

På tentan har du tillgång till sidan `https://rubular.com/`.

Givna filer

Eventuella givna filer finns länkade från kurssidan.

Avslutning

När du skickat in alla uppgifter och känner dig färdig kan du stänga tetaklienten och logga ut.

Uppgift 1 - Icke-deteministisk programmering (7p)

I den givna filen `amb_test.rb` hittar du den problemlösare vi arbetat med under laborationsserien. I denna uppgift ska du använda den problemlösaren för att hitta vilka stolar personalen i Pontus arbetsrum kan välja. I arbetsrummet finns 4 stolar med olika inställningar av höjd. Dessa höjder är: `61, 40, 54 och 50`. Det finns 4 personer i rummet som alla behöver en stol att sitta på och man får inte ändra höjden på stolarna. Dessa personer kan max sitta i en stol som är en viss höjd `p1: 59, p2: 66, p3: 45 och p4: 54`. Ingen person kan sitta i samma stol som en annan. Ändra den givna filen så att den skriver ut alla möjliga kombinationer av personer och stolar.

Körexempel

```
p1: 54, p2: 61, p3: 40, p4: 50
p1: 50, p2: 61, p3: 40, p4: 54
Det var alla möjligheter.
```

I körexemplet ovan kan alltså `p1` sitta i stolen som är `54` hög eller `50` hög om alla personer ska få en stol.

Antaganden:

- Du kan anta att stolarnas höjder är unika (ingen stol har exakt samma höjd som en annan).
- Du kan anta att det finns minst lika många stolar som personer
- För full poäng bör programmet vara så allmänt skrivet att man kan lägga till fler stolar, det är alltså lämpligt att lagra stolarna i en `Array`. Men du kan utgå ifrån att det alltid kommer vara 4 personer och att 4 stolar kommer väljas ut av de tillgängliga stolarna.

Uppgift 2 - Sax-parsning av XML (9p)

I den givna filen `sax_example.rb` hittar du ett exempel på hur man kan arbeta med en sax_parser i ruby. I den givna filen `orientering.xml` hittar du en xml-fil som specificerar en orienteringstävling. I xml-filer av denna typ så finns det ett antal intressanta datum och tider. De datum och tider som är intressanta för oss är – där dokumentet har modifierats, här är ett exempel på intressant information:

```
<DateInfo type="modified">

  <Date>2022-03-06</Date>

  <Clock>17:34</Clock>

</DateInfo>
```

Ditt jobb är att med hjälp av sax parsning läsa ut datumet och tiden för alla `DateInfo`-element där `type` är satt som `modified`. Din implementation och sax parsern skall lagra informationen för varje `DateInfo`-element i form av en `Hash`. Alla dessa hashar skall sedan lagras i klassen i en `Array` och motsvarande access-operator eller getter skall skapas för att hämta ut denna `Array`

Körexempel:

```
[{:date=>"2022-03-06", :clock=>"17:34"},
 {:date=>"2022-01-04", :clock=>"19:52"}]
```

- Formateringen av utskriften är inte viktig, inte heller att nycklarna i hashen är symboler
- Lösningen skall vara någorlunda robust, även om du kan utgå från att formatet på andra filer följer formatet i denna fil så är det inte säkert att andra filer har exakt samma element på samma ställe.

Uppgift 3 - Dom-parsning av XML och regepx (13p)

Del a - Dom-parsning (7p)

I den givna filen `orientering.xml` finns ett antal banor specificerade. Dessa banor representeras av elementet `<Course>`. Vissa av dessa element har ett attribut som bestämmer om de skall synas eller inte. I denna uppgift ska du gå igenom alla banor som har attributet `visibility` satt som `'show'`. För varje bana som uppfyller detta kriterium skall du skriva ut banans id och sedan gå igenom alla element som ligger i banan av typen `LegLength` och skriva ut: genomsnittlig längd, total längd och antalet etapper (se körexemplet). Endast två banor i den givna filen uppfyller detta kriterium, men du behöver ta hänsyn till att de kan finnas godtycklig många sådana banor.

Körexempel:

```
Found a course, id: 0
Average leg length: 419
Total length: 7552
Number of legs: 18

Found a course, id: 5
Average leg length: 360
Total length: 3242
Number of legs: 9
```

- Det kan vara frestande att läsa ut den totala längden ur elementet `<CourseLength>` men på grund av slarvig inmatning stämmer inte dessa alltid. Det är därför viktigt att du summerar längden av alla etapperna (som finns i `LegLength`).

Del b - Regepx (6p)

I filen `orientering.xml` finns det information om kartan i elementet som heter `Map`. Din uppgift är att läsa ut datumet `Date` och skalan `Scale` ur alla element `Map` med regular expressions. I den givna filen finns det 2 sådana element men din lösning ska vara robust nog för att hantera ett godtyckligt antal. Programmet skall sedan lagra informationen i object av typen `Hash` och lagra dessa i en `Array`. Till sist ska du skriva ut resultatet (se körexemplet).

```
[{:date=>"1999-05-01", :scale=>"16000"},
 {:date=>"1995-04-06", :scale=>"15000"}]
```

- Formatet på utskriften spelar ingen roll men typerna ska stämma (det är ok om nycklarna i `Hash`arna är av annan datatyp).
- Du kan vilja använda `Regexp::MULTILINE` i din lösning som gör att `.` också matchar `newline`. Nedan följer 2 alternativ för att sätta den inställningen på ett regexp:

  ```
  expression = Regexp.new("abc", Regexp::MULTILINE)
  expression = /abc/m
  ```

- Det kan vara en bra idé att först försöka hitta hela `Map`-element med regexp och sedan söka i varje sådan sträng med regexp igen, istället för att försöka lösa allting med ett regexp.

Uppgift 4 - Parser (15p)

I filen `rdparse.rb` finns den parser som vi arbetat med i kursen, ditt jobb i denna uppgift är att utöka/ändra denna parser så att det givna språket nedan fungerar enligt beskrivning.

Det rykande färska språket STRIMLA (String Manipluation Language) är ett enkelt interpreterat språk för att manipulera textsträngar på olika sätt. Ett uttryck i STRIMLA kan konstrueras ut på fem olika sätt, vilket illustreras med följande exempel:

```
[Hejsan]       En enkel sträng anges med hakparenteser och innehåller
               bokstäver och/eller mellanslag.

3*[banan]      Detta skapar en sträng som innehåller banan tre gånger,
               d.v.s. "bananbananbanan".

2>[abc]        Flyttar respektive tecken två steg framåt i alfabetet, d.v.s.
               resultatet är "cde".
               (Lös enkelt, wrapping behöver inte fungera.)
               (String.ord och Integer.chr kan vara intressanta)

[foo]+[fie]    Slår ihop två strängar, d.v.s. "foofie".

(2*[oh])+[hai] Parenteser kan användas för att bestämma
               prioriteten. I det här fallet blir resultatet "ohohhai".

Alla strängar måste omges med hakparenteser. Alla tal måste vara positiva.
```

Del a (10p)

Din uppgift är att först konstruera en grammatik för STRIMLA - där `STRING`, `INTEGER` och symboler kan vara terminaler - och därefter - med utgångspunkt från parsern i rdparse.rb - konstruera en parser för STRIMLA som kan hantera det språk som beskrivs ovan. Språket får inte utökas på något sätt som gör parsern enklare.

Följande exempel visar hur det kan se ut:

```
4*[boom]
=> boomboomboomboom
[Hello ]+[there]+[ buddy]
=> Hello there buddy
2*[a]+[b]
=> aab
2*([a]+[b])
=> abab
2>[a]+[b]
=> cb
3*2>[a]+[b]
=> cccb
```

Tips: Tänk på att operationer som använder heltal (`+*>`) bara behöver fungera om heltalet är till vänster. Vi kan alltså anta att högra operanden alltid är en sträng i vår implementation.

Del b (5p)

En grammatik kallas tvetydig(=icke entydig) om samma sträng kan tolkas/parsas på fler än ett sätt enligt grammatiken.

```
Med grammatiken
<summa> ::= <tal>
          |  <summa> + <summa>
kan 1+2+3 tolkas enligt <summa> + <summa> både som att 1 är första och
2+3 är andra <summa> och som att 1+2 är första och 3 är andra <summa>.
```

Är din grammatik entydig eller tvetydig? Motivera. Hur hade din grammatik kunnat göras enklare eller tvingats bli mer komplicerad om den tilläts/tvangs ha motsatt egenskap?

Uppgift 5 - DSL (6p)

I den givna filen `dsl.rb` finns en konfigurationsfil. Ditt jobb är att implementera detta domänspecifika språk med hjälp av `method_missing` i ruby. I den givna filen `dsl_reader.rb` finns lite given kod som du kan utgå ifrån. Det är viktigt att det fungerar att lägga till godtyckliga alternativ i `dsl.rb` och att ditt program hanterar detta. I detta språk står alltid ett alternativ på en ensam rad och det till vänster är alternativet och det till höger är vad alternativet skall sättas till.

Högersidan kommer alltid vara en text utan citattecken. Vänstersidan kan vara godtycklig datatyp från ruby eller en text utan citattecken. Alla alternativ skall sparas internt i en `Hash` och returneras med operatorn eller gettern `config` (se körexemplet).

Körexempel:

```
require 'pp'
cp = ConfigParser.parse("dsl.rb")
pp cp.config
=>{:window_w=>[1920],
=> :window_h=>[1080],
=> :fullscreen=>[false],
=> :icon=>["temp.png"],
=> :name=>["fungame"]}
```

- Datatypen på nycklarna i `Hash`en spelar ingen roll. Datatypen på värdena spelar roll.