# TDP007 - Dugga 2

2026-03-04

## Regler

- All kod liverättas under denna examination
- Inga elektroniska hjälpmedel får medtas. Mobiltelefon ska vara avstängd och ligga i jacka eller väska.
- Inga ytterkläder eller väskor vid skrivplatsen.
- Student får lämna salen tidigast en timme efter start.
- 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 vakt i sal genom att räcka upp handen.
- Endast uppgifter inskickade före tentamenstidens slut rättas.

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

# Information

## Betygssättning vid tentamen

Tentamen består av 2 delar. Del 1 är uppgifter på grund-nivå som relaterar till tekniker från kursens seminarier. Del 3 består av mer avancerade uppgifter, där varje uppgift relaterar till någon teknik från kursens seminarier. Alla uppgifter bedöms som G/U. Betygsättning vid tentamen görs enligt följande tabell (för alla betyg 3, 4, 5 krävs att alla uppgifter på grund-nivå är avklarade):

| **Avklarade uppgifter del 2** | **Betyg** |
|---|---|
| 0 | 3 |
| 1 | 4 |
| 2 | 5 |

## Duggor

Varje dugga motsvarar halva tentan. Dugga 1 tar upp koncept från kursens första 2 seminarier. Dugga 2 tar upp koncept från de sista 2 seminarierna. För varje avklarad del under en dugga tillgodoräknas motsvarande uppgifter på första ordinarie tentamen efter kursen. Exempel: (1) Som student har jag klarat endast klarat av del 1 av dugga 2 i kursen. Om jag skriver ordinarie tentamen behöver jag då inte göra motsvarande hälften av del 1 på tentamen. (2) Som student har jag klarat av del 1 och del 2 på både dugga 1 och dugga 2. Jag har nu betyg 5 på tentamen och det finns ingen anledning för mig att skriva tentan.

## Inloggning

Logga in på datorn i labbsalen med ditt liu-id och lösenord. Detta är samma inloggningsuppgifter som du använder i Lisam.

## Skrivbordsmiljön

När du kommit in i tentasystemet har du en normal skrivbordsmiljö (Mate-session i Ubuntu). Efter en stund kommer även din kommunikationsklient att dyka upp automatiskt. Startmenyn är nedskalad till enbart det som examinator bedömt relevant. Andra program kan fortfarande finnas tillgängliga genom att starta dem från en terminal. Observera att en del program som använder nätverkstjänster inte fungerar normalt, eftersom nätverket inte är åtkomligt.

*När du är inloggad är det viktigt att du har tentaklienten igång hela tiden. Om den inte dykt upp fem minuter efter inloggning och inte heller när du startar den manuellt från menyn (fisken) tar du kontakt med assistent eller vakt i sal.*

## Terminalkommandon

`ruby` används för att köra en ruby-fil.
`irb` används för att starta ruby i interaktivt läge.
`ri` används för att komma åt den inbyggda dokumentationen.

### Webresurser

På tentan har du tillgång till referenssidorna, rubular och en version av kursboken. Klicka på web access på skrivbordet.

### Givna filer

Eventuella givna filer finns i katalogen `given_files`. Denna underkatalog är skrivskyddad, så det är ingen risk du råkar ändra på dessa filer. Skrivskyddet gör dock att du måste kopiera in de givna filer du vill använda till tentakontots hemkatalog. Hur du listar och kopierar filer ska du kunna. Hemkatalogen står du i från början, och du kommer alltid tillbaka till den genom att bara exekvera kommandot `cd` i terminalen.

### Avslutning

När du är nöjd med din insats (inte tänker arbeta vidare) kan du avsluta.

Avsluta alla öppna program och logga ut ur datorn. Lämna inte datorn innan du ser att du är utloggad.

## Del 1: Uppgift 1 - rdparse

I den givna filen **uppgift1.rb** hittar du rdparse som vi arbetat med i föreläsningsserien. Funktionen **roll** är dock modifierad så att den heter parse och för att ta emot en sträng som parameter för att förenkla testningen vid rättning. Din uppgift är att modifiera den givna lexern och parsern för att implementera `stringlang` ett häfting nytt språk som låter användaren mata in ord och duplicera samt slå ihop orden. Ett ord är en sekvens av bokstäver a-z (stora eller små). Duplicering av ord görs med symbolen @ och sammanslagning av två ord görs med operatorn +. Operatorn @ har högre prioritet än operatorn +.

Alla exempel i de funktionella kraven ska fungera som förväntat.

**Funktionella krav:**

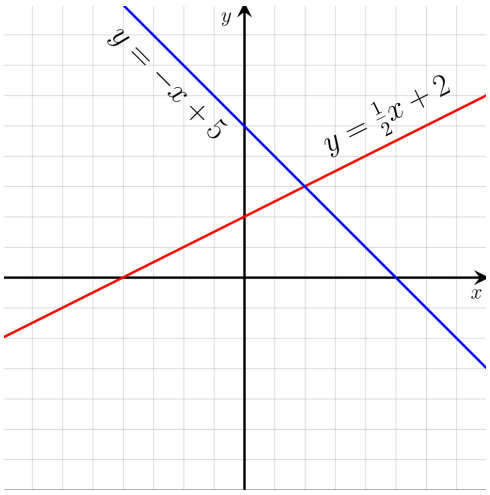
```
[stringlang] Pontus => Pontus
[stringlang] Pontus@ => PontusPontus
[stringlang] Pontus@@ => PontusPontusPontusPontus
[stringlang] Pontus + Haglund => PontusHaglund
[stringlang] Pontus@ + Haglund => PontusPontusHaglund
[stringlang] Pon+Ha@+Glund => PonHaHaGlund
[stringlang] Pon+Ha@@+Glund => PonHaHaHaHaGlund
```

## Del 1: Uppgift 2 - DSL

I den givna filen **uppgift2.rb** hittar du den sedvanliga grunden för en dsl-läsare och i filen **dsl.rb** hittar du ett litet domänspecifikt språk där användaren kan spara information om enheter (units) från spelet Warhammer. Ditt jobb är att modifiera **uppgift2.rb** så att denna kan läsa språket i **dsl.rb**. **unit** är ett reserverat ord i språket och förekommer alltid när man vill skriva information till en enhet och kan hanteras separat med en if-sats eller separat medlemsfunktion. Allting som följer efter en sådan rad tillskriver den enheten mer information. Din läsare behöver hantera att användaren tillskriver vilken information som helst, men du kan förutsätta att parameterlistan alltid är en eller flera primitiver/strängar. Observera att enheten **Riptide** *öppnas* för skrivning två gånger. Detta ska fungera som i de funktionella kraven nedan.

**Funktionella krav (exakt formattering spelar ingen roll, datan ska stämma (du kan ha symboler eller strängar som nycklar):**

```
{:Riptide=>
  {:weapons=>"Ion accelerator",
   :abilities=>
    ["Nova Charge", "Battlesuit Support Systems"],
   :wounds=>14},
 :Victrix_Honour_Guard=>
  {:number_of_models=>3,
   :weapons=>"Blades of honour",
   :wounds=>3,
   :attached=>["Captain", "Chapter Master"]}}
```




**Figur 1:** Räta linjens ekvation, public domain, wikipedia

## Del 1: Uppgift 3 - Constraints

Constraints kan användas för att representera ekvationer. I den givna filen **uppgift3.rb** finns constraintnätverken vi arbetat med i labbserien (buggen från labbserien är löst i denna fil). Ditt jobb är att implementera funktionen **linear_equation(k, m)**. Denna funktion ska sätta upp ett constraint nätverk som representerar linjära ekvationer (räta linjens ekvation, se Figur 1). Pontus förutsätter inte att du har denna memorerad från gymnasiet så här är den:

$y = kx + m$.

Funktionen sätter lutningen och var linjen skär origo (**k** respektive **m**) med parametrarna som skickas till funktionen. Funktionen returnerar representationen av **x** och **y**. Användaren ska sedan kunna sätta värdet på **x** eller **y** i huvudprogrammet för att få ut värdet på den andra (se given kod och funktionella kraven).

**Funktionella krav (Logg-utskrifter kan vara kvar i din output):**

```
Value of y after x is assigned 3.0 => 3.5
Value of x after forget and y is assigned 4.5 => 5.0
Value of y after x is assigned 5.0 => 0.0
```

## Del 2: Uppgift 4 - Parser och AST

I **uppgift4.rb** finns parsern vi arbetat med i labbserien. Parsern är modifierad för att vara enkel att testa på samma sätt som i uppgift 1. Ditt jobb är att implementera ett logikspråk. Följande tokens kan skrivas i språket: true, false, not, xor, or, and.

Beskrivning av semantiken för varje token, dessa förekommer i sin prioritetsordning (högst prioritet högst upp i listan).

**true** är en av två primitiver i språket och representerar ett sant värde

**false** är en av två primitiver i språket och representerar ett falskt värde

**not** är en unär operator som står till vänster om sin operand. Denna returnerar inversen av operandens sanningsvärde.

**xor** är en binär operator som returnerar sant om en av operanderna är sann och den andra är falsk, i alla andra fall returnerar operatorn falskt. Detta är alltså en exclusive-or operator.

**or/and** är binära operatorer, fungerar som förväntat (logisk or respektive logiskt and)

**Icke-funktionella krav:** Parsern ska skapa ett AST (Abstrakt Syntax Träd) och returnera rotnoden av trädet. Du ska sedan utvärdera trädet för att få ut resultatet. Utvärderingen av trädet kan göras i funktionen **parse**.

**Funktionella krav:**

```
[MyLang] true => true
[MyLang] false => false
[MyLang] not true => false
[MyLang] not false => true
[MyLang] not not true => true
[MyLang] not not not false => true
[MyLang] true and false => false
[MyLang] true and true => true
[MyLang] true and false or true => true
[MyLang] true xor true => false
[MyLang] true xor false => true
[MyLang] false xor false => false
[MyLang] not not true and not false xor not true => true
```