LINKÖPINGS UNIVERSITET

TDP007: Konstruktion av datorspråk

Utvecklarblogg
Seminarieuppgift 1

Författare

Daniel Huber, danhu849@student.liu.se
Theodore Nilsson, theni230@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2021
Version 1.0

2021-01-20

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Utvecklarblogg, Seminarie 1 TDP007	200121

Samtliga uppgifter förutom där det specifikt specificerats har whitebox testats. Dvs där testerna skrivits efter och/eller vid sidan av koden.

Avsnitt 1: Iteratorer

Uppgift 1

En förenkling vi hittade på geeksforgeeks hemsidan är att ersätta `for i in (1..n) do` med `n.times do`.

Det känns väldigt underligt att block inte stoppas in som parameter inom parenteser vid funktionsanrop så som vi är vana vid, men vid defintionen av en funktion så representeras blocket som att det ska skrivas inom parentes.

Efter en tid insågs att `n_times` och `each` i klassen `Repeat` gjorde samma sak. Det är trevligt men lite konstigt att inte mer behöver göras för att anropa ett block i ett annat:

```
1  def each(&block)
2    n_times(@num) { block.call() }
3  end
```

Även om det specificerades under föreläsningen att funktionsöverlagringar inte finns i Ruby så känns det motstridigt att kunna omdefiniera medlemsfunktionen `each` i en klass problemfritt.

Uppgift 2

Vi testar inte för 0 då 0 inte tillhör de positiva heltalen.

Vid block är det svårt att veta vad som är vad och vad som blir vad. I till exempel:

```
1 (1..20).inject(1) \{ |product, element| product *= element \}
```

är det oklart att product antar parametervärdet till inject och element antar första värden i rangen 1..20. Fortfarande oklart hur värdena byts.

Metoder kan skrivas som inparameter till inject och då kommer metoden att utöva sin funktion på respektive värde i samlingen:

```
1 (1..20).inject(:*)
```

`:*` är symbolen för multiplikationsmetoden i Ruby. Hämtat från kursboken s.57.

Uppgift 3

Misstaget gjordes i början att jämföra strängarnas värde istället för längd. I kursboken beskrivs att om ingen inparameter skickas med injekt antas automatiskt första objektet i samlingen:

```
1 def longest_string(list)
2   list.inject() { |longest, elem| longest > elem ? longest = longest : longest = elem }
3 end
```

Kom dock efter testning fram till att strängars längd skulle användas och efter ytterligare testning fram till att blocket automatiskt returnerade det längsta ordet:

```ruby
def longest_string(list)
  list.inject() { |longest, elem| longest.length > elem.length ? longest : elem }
end
```

Lite störande att parenteser inte behövs vid funktionsanrop med inparameter. Preliminärt känns det svårt att veta vad som är associerat till vad.

Uppgift 4

Efter att kodskelettet skrivits för uppgiften var det svårt att veta vart logiken skulle påbörjas. Testade först att bara stoppa in en block.call() inom hakparenteser, men fick då felet att nil inte har .length funktion:

```ruby
def find_it(str_array, &block)
  str_array.inject() { block.call() }
end
```

Detta blir fullt förståeligt efter det insetts att det i `block.call()` inte skickas med några parametrar a och b:

```ruby
def find_it(str_array, &block)
  str_array.inject() { |a, b| block.call(a,b) }
end
```

Eftersom blocket returnerar sant eller falskt måste det också skrivas till att a ska returneras vid sant och b vid falskt såsom det är menat från det givna blocket:

```ruby
def find_it(str_array, &block)
  str_array.inject() { |a, b| block.call(a, b) ? a : b}
end
```

Det svåraste var att få till samt förstå hur parametrar kunde anta värden från samlingen och därefter skickas vidare och användas i blocket. Lösningen hittades genom analys av exempel i kursboken samt gedigen trial and error.

Avsnitt 2: Åtkomstfunktioner

Uppgift 5

För att ge åtkomst till medlemsvariablerna i de skapade klasserna har har metoden med attr_accessor använts så de variabler som ska ska kunna kommas åt listas på följande vis:

```ruby
class PersonName
  attr_accessor :name, :surname
[...]
```

Det påminner syntaktiskt om datamedlemsinitieringslistorna som finns i C++ men i själva verket skapar en getter/setter funktion för de givna medlemsvariablerna.

En tredje variabel skulle vara ett *virtuellt attribut* vilket tolkades som att den inte explicit skulle sparas som en medlemsvariabel, utan kunde beräknas av en funktion med hjälp av de existerande medlemsvariablerna. Eftersom medlemsfunktioner och medlemsvariabler använder samma syntax blir det likvärdigt för användaren.

En setter för vårt virtuella attribut kunde enkelt göras med en funktionssignatur på formen

```ruby
  def fullname=(nameString)
```

Detta liknar operatoröverlagring, men fungerar i själva verket därför att det blir tillåtet att skriva till mellanslag och ta port paranteser vid anropet

```
myName.fullname = "Bob"
```

Uppgift 6

I klassen Person skulle alla parametrar vara frivilliga, vilket kunde uppnås genom att ge de standardvärden av rätt datatyp i konstruktorn.

Ett tidsobjekt för den nuvarande tiden ges av Time.now, och för att få ut året läggs .year till:

```
Time.now.year
```

Det går alltså att få åtkomst till attribut i flera led. Detta användes senare för att få åtkomst till name attribut för Person objektet.

Attributet name skulle sparas som ett PersonName objekt vilket gjordes genom att skapa en ny instans av en sådan med de givna parametrarna från Person konstruktorn.

```
    @name = PersonName.new(name, surname)
```

Det var förhållandevis enkelt att hitta information på nätet, och eftersom åtkomst till medlemsvariabler och medlemsfunktioner kunde skrivas på samma sätt i anropet var det hela hyfsat intuitivt.

Att förstå att uttrycket String#split, i själva verket skulle skrivas string.split var till en början otydligt och det är oklart ifall uttrycket *virtuellt attribut* har tolkats korrekt.

Avsnitt 3: Utökning av existerande klasser

Uppgift 7

I början var det svårt att börja då vi inte visste om vi behövde skriva något extra för att markera att vi utökade klassen Integer. På nätet fanns info om moduler, men eftersom vi inte gått in på det alls sköts det åt sidan.

Efter lite testande kom vi dels fram till att om parenteser inte fanns vid definieringen av funktionen så blev det en punktannoteringsfunktion och dels att bara klass namnet behövdes:

```
class Integer
  def fib
    if self ==1
      1
    elsif self == 2
      1
    else (self-1).fib + (self-2).fib
    end
  end
end
```

End behövdes bara i slutet efter else och inte efter både if och elsif.

Uppgift 8

Vid skrivandet av funktionen acronym var det svårt att få till hur inject skulle hantera respektive ord. Först fick vi bara fram ett ord och två bokstäver som resultat.

```
def acronym
    self.split.inject() { |acro, word| acro += word[0].upcase }
end
```

Det var för att vi inte specificerat en tom sträng för injekts startvärde. Vi löste det genom att testa oss fram med exempel från kursboken och såg att inject möjliggör att annat startvärde än det som finns i samlingen väljs.

```
def acronym
    self.split.inject("") { |acro, word| acro += word[0].upcase }
end
```

Avsnitt 4: Reguljära uttryck

Uppgift 10

För att hitta matchningar till reguljära uttryck användes medlemsfunktioner för dessa.

För att komma åt den del i matchningen vi ville ha i funktionen find_user_name(), delade vi upp vårt reguljära uttryck i två grupper och med hjälp av den inbyggda funktionen .captures plockade vi ut matchgruppen som önskades:

```
def find_user_name(string)
  md = /([a-zA-Z]*: )([a-zA-Z]+)/.match(string)
  md.captures[1]
end
```

Det var mycket enklare inkorporerat än i både python och c++ att reguljära uttryck antogs vara egna objekt så vi behövde inte kolla upp något externt material för att det skulle fungera. Att det dessutom var superlätt att läsa in en hel hemsida med funktionen open() är fantastiskt!

Uppgift 11

I funktionen tag_names plockade vi ut de delar vi ville ha genom att först, pythoninspirerat, göra om arrayn till ett set och sedan tillbaka till en array igen. Problemet med detta reguljära uttryck var att varje ord vi önskade behövde plockas ut i en loop:

```
def tag_names(string)
  md = string.scan( /(<)([a-zA-Z]+)(>)/ )
  md.to_set.to_a
end
```

Efter testning och letande på sidan för array i ruby doc hittade vi .uniq funktionen. Vi kände igen problemet från en tidigare python lab och kunde klippa/klistra det reguljära uttrycket därifrån så krav för någon loop eller vidare behandling undveks.

```
def tag_names(string)
  md = string.scan( /(?<=<)([a-zA-Z]+)(?=>)/ )
  md.uniq
end
```

Avslutningsvis gjordes enkla modifikationer på det första uttrycket för att kunna ha understreck och numeriska tecken i ett användarnamn. Dessutom skrevs ett test för att säkerställa att datastrukturen på vårt returvärde i den andra funktionen var som beskrivet i uppgiften då utseendet i terminalen inte stämde överens med exempelkörningen.

```
for i in (0..tags_arr.length-1) do
    assert_equal( tags_arr[i].class, Array )
    assert_equal( tags_arr[i][0].class, String )
end
```