Utvecklarblogg Seminarie 3

Uppgift 1

Tolkning av uppgiften:

- Bilmärken ger ett specifikt värde
- Postnummer anger intervall. 0 till lägsta värdet är 0, ifrån högsta och uppåt är högsta angivna värdet.
- Antal år man haft körkort anges i intervall enligt ovan
- Ålder anges i intervall också enligt ovan.
- Intervall anges i policy.rb i stigande ordning

Förklaring funktioner:

def evaluate_policy(file_name)
Här kallar vi på instance_eval för att gå igenom policy.rb och fylla hashen @policy_cost med värden. Vi använder sedan funktionen modifier som då räknar ut och returnerar @sum som är det slutgiltiga värdet vi vill åt.

def modifier
vi använder oss av .map på hashen @policy_costs värden för att kalla på blocket på alla värden och summera dem i @sum. Vi gör sedan checkar för de regler som angavs i uppgiften och modifierar @sum om kriterierna uppfyllts.

method_missing(name, *args)
Är den funktionen som kallas när instance_eval inte hittar det som angivits i policy.rb som en funktion i ruby. Vi tar in en parameter, "name", som anger vilken typ av värden det är på den raden och sedan en array med de värden vi vill göra något med. Vi sparar alla värden i hashen @policy_cost.

Reflektion:

I allmänhet har det kännts som vi fastnat mycket mer på de här uppgifterna än vi har på tidigare seminarier. Det här kändes som något helt nytt och det var lite konceptuellt svårt att sätta sig in i ifrån början och inte särskilt enkelt att hitta information om när man googlade så det har slutat i väldigt mycket att man bara testar olika saker, kompilerar och hoppas på det bästa.

Vi har inte en särskilt elegant eller effektiv lösning heller då man tex skriver över värden flera gånger och vi alltid kollar mot ett första ord vad som ska göras. Vi har funderat på om man kan lösa det utifrån att bara skriva policy.rb i formatet:
"BMW", 5
58647, 3
0, 3

"K", 1
18, 2.5

Vi har dock inte kommit på något bra sätt man kunde särskilja de olika typerna av värden ifrån varandra på något bra sätt. Man hade kanske kunnat ha en titel inför varje del i policy.rb:
Bilmärke
"BMW", 5
Postnummer
58647, 3

När man då stötte på tex Bilmärke så kunde man pusha en state som hanterade bilmärken till en state-stack och då få korrekt hantering men man hade då också behövt lösa hur man skulle byta state på ett bra sätt utan att kolla varje gång så det kändes inte rimligt att ge sig in på för denna uppgiften.

Om det här hade varit en situation där man implementerat det här på riktigt i ett företag så vet jag inte hur lätt det hade varit för någon att utöka funktionaliteten i policy.rb om de inte hade en programmeringsbakgrund och faktiskt kom åt koden i seminarie3.rb. Det hade varit intressant om man kunde implementera allting direkt i policy.rb på något sätt.

Testning:

Testar personens angivna data på så sätt att:

- Sum inte modifieras
- sum modifieras av kriterie 1 (*0,9)
- sum modifieras av kriterie 2 (*1.2)
- sum modifieras av kriterie 1 & 2 (*0.9*1.2)

Vi testar även om det blir korrekt värde om personens angivna data är högre eller lägre än intervallet som är angivet i policy.rb

Vi testar inte negativa nummer på zip/erfarenhet/ålder för det känns konceptuellt konstigt att någon anger det.

Uppgift 2

Tolkning av uppgiften:

- Vi ska använda oss av rdparse.rb och endast implementera klassen som hanterar grammatiken, i stil med DiceRoller klassen.
- Vi har valt att frångå "ASSIGN ::= '(' 'set' VAR EXPR')' " regeln något då vi inte har VAR utan en separat regel för att undvika en oändlig loop, se reflektion för mer info.

Förklaring funktioner:

Initialize är egentligen den enda som är intressant då de andra är kopierade direkt från DiceRoller klassen och eval_test endast är där för att det ska vara smidigt att enhetstesta klassen.

def initialize
Vi valde att separera texten vi läser in på så sätt att vi gör oss av med whitespaces, läser in "(", ")", bokstäver och siffror som separata tokens. Vi ville implementera tokens som läste in true/false och returnerade som bools direkt men det gick inte, se reflektion för mer info.

i mångt och mycket tycker vi reglerna är självförklarliga då de tex använder rubys interna logik för or/and/not och returnerar en bool när det är angivet som en sträng.

Det som kan nämnas är att i vår :assign regel så använder vi inte en :var utan en :key som har en egen regel för att undvika ett problem med en oändlig loop då vi har en regel i :var där vi hämtar värdet ifrån en variabel. Våra variabler som anges när någon använder vår :assign regel sparas i hashen @variables och det är även därifrån värdet hämtas när någon använder :var regeln med korrekt variabelnamn.

Reflektion:

Vi ville göra tokens som direkt returnerade bools, tex "token(/true/) {true}" men rdparse.rb bråkade med den idén. Det fungerade konstigt nog för true men kraschade när vi gjorde det med false och vi fick ett error på rad 179 i rdparse.rb där det stod att den förväntade sig true och en Integer. Vi förstod aldrig riktigt varför det inte fungerade och fick kolla efter strängarna “true”/”false” i :term istället och returnera boolen där.

Då vi sparar våra variabler i en hash och våra :var regel har ett uttryck där vi kan hämta ut värdet ifrån hashen så hade vi ett intressant problem där man kunde ange tex (set banan 5) och (set 5 banan) och sedan bara skrev antingen “banan” eller 5 så hamnade man i en oändlig loop. Detta för att i vår assign regel så angavs keyn som en :var som då alltid hämtade det andra värdet ifrån hashen när den returnerade värdet ifrån hashen för den

tolkade det värdet som en :var. Vi har gjort en ny regel som är specifikt för keyn som anges i :assign regeln som här specar att keys bara får vara strängar för att lösa det problemet.

Även här uppfattade vi det som att det var svårt att hitta information om hur man korrekt gjorde denna typ av uppgift. Det är svårt att söka på BNF grammatik då det kan användas för så himla många olika saker inte bara denna typ av implementation. Det således mycket att man testade något, kompilera, och sen försöka lista ut varför kompilatorn blev arg.

Testning:

Vårt test för set kollar så att reglerna för :assign fungerar korrekt. Att keys kan anges som strängar men att värden kan ha olika typer. Vi testar även att ha en expression nestlad i en expression. Här testas också implicit :key regeln och att :expr kan utvärderas ned till :var för att bli en String/Integer

Vi testar sedan or/and/not i olika test. Vi testar de olika kombinationerna som finns för true/false så att de ger korrekt värden. Här testas även implicit att en :expr kan utvärderas till en :term som returnerar en bool när strängen är antingen "true" eller "false".

Till sist testas att en :expr kan utvärderas ned till att bli en :var som är antingen en Integer eller en String, med eller utan parenteser.