Utvecklarblogg - Seminarium 3

Grupp B6 - guser766 | hadan326 | vinah331

Att lära sig Ruby (nya saker)

Parsning

Att förstå parsning kan vara enkelt eller komplicerat beroende på vilken abstraktionsnivå man lägger sig på. Vi valde i denna uppgift att hålla oss på en relativt hög nivå, där vi bara behövde arbeta med den grammatiska specifikationen. Således berodde vår lösning framförallt på våra kunskaper i regex och hur sofistikerad vår grammatiska spec. var, och vi behövde inte göra en djupdykning i hur parsern fungerade. Detta kommer vi troligtvis behöva göra längre fram men i nuläget känns det som att vi med vår begränsade förståelse skulle kunna skapa rätt komplexa program.

Tolkning av uppgifter och förklaring av lösningar

Uppgift 1

Uppgiften handlar om att skapa ett DSL för personalen i en bilförsäkringsbolag så att de kan med enkla uttryck definiera olika policy som sedan kan beräkna försäkringspremien för en person. Vår lösning går ut på att personalen ska skriva dessa regler i filen "policy.rb" i den följande format:

brand_policy
BMW 5
Citroen 4
Fiat 3

Alla rader kommer att köras i en klass som heter "Policies". Tanken är att när man kallar på evaluate_policy på en person ska dessa policy laddas till en Policies-objekt och beräkna personens försäkringspremien.

Raden "brand_policy" kommer att skicka en signal som säger att alla andra följande rader kommer att vara elementer (nyckel: värde) till hashen "brand" i klassen Policies. Därför alla rader som kommer efter och har ett värde anses vara nycklarna och värdena till hashen.

I Policies klassen har man följande datamedlemmar för varje policy-kategori (alla i typen hash):

@brand = {}
@age = {}
@zip_code = {}
@license = {}
@gender 0 = {}

Vår kod fungerar så att den alltid kallar på metod_missing funktionen eftersom vi inte har några funktioner alls som matchar uttrycken i policy.rb.

Syntaxen för att definiera andra kategorier (av de som finns) är på samma sätt och man kan skriva gender_policy, license_policy, zip_policy och age_policy. Det finns en liten skillnad på hur man definierar nycklarna för till exempel license_policy. Vi har bestämt oss att det ska se ut på följande sätt:
license_policy
range0_to_1 3
range2_to_3 4
Anledningen till detta var att vi stötte på ett problem om vi hade skrivit "0_to_1" eftersom ett funktionsanrop inte kan inledas med siffror. Det som är lite unik här är att nycklarna i det här fallet kommer att tolkas som "rang " i hashen license.
{(0..1) : 3, (2..3): 4}

Vi har tänkt att personalen ska kunna skriva regler som beskrivs i uppgiften på följande sätt:
rule1 0.9
rule2 1.2

Rule1: För män som haft körkort mindre än tre år multipliceras säkerhetspoängen med 0.9.
Rule2: För personer som äger Volvo och som bor i ett postnummerområde som börjar med 58 (större delen av Östergötland) multipliceras säkerhets poängen med 1.2

Med den här implementationen kan man ändra värdet på dem om man vill men själva regler är hårdkodat och inte går att modifiera i DSL.

Till sist när DLS tolkas till relevanta hash i Policies klassen, returnerar vi ett objekt av Policies typen som innehåller alla policy och regler. Med Policies objekten kan vi sedan jämföra personens information med motsvarande policy och regler och beräkna försäkringspremien i evaluate_policy.

Uppgift 2

Uppgiften var att utifrån den givna parsern implementera ett språk som kunde hantera logiska uttryck. I detta antog vi även att man skulle kunna tilldela variabler värden.
Språket hanterar boolska värden och är begränsat till de logiska uttrycken *and, or* och *not*.
Vi utgick från *DiceRoller* och ändrade i *token, start* och *rule* för att passa vårt språk enligt de regler som angetts på kurssidan. Den största skillnaden blev att vi framförallt hanterade ord istället för tecken och heltal.
Konverteringen från *DiceRoller* var problemfri fram till definitionen av assign, eftersom *DiceRoller* inte hade någon variabeltilldelning.
Det vi först hade svårt med var att förstå hur vi skulle definiera en variabel. Vi kom fram till att en Hash som medlemsvariabel i LogicExpr-klassen var det bästa valet, eftersom man då kunde sätta variabelnamnet som nyckel och ge nyckeln ett värde.

Vi hade problem med att få "set" att fungera korrekt. När skulle variabeln läggas i vår Hash och hur skriver vi över en redan tilldelad variabel?
Vi försökte först lösa det genom en if-sats i :var men det hindrade oss från att skriva över variabler. Resonemangen var att :var skulle kunna avgöra om vi sökte en variabels värde eller själva variabeln.
Efter diskussion med assistent insåg vi att denna distinktion kunde ske i steget från :term där nu variabelns värde returnerades. :var returnerar enbart variabeln utan anknytning till Hashen.

Detta var något vi helt missat innan som nu känns uppenbart. När parsern erbjuder oss friheten att hoppa mellan assign, expr, term och var kan fall likt ovan smidigt lösas utan onödiga if-satser.