Seminarie 3

Insurance

Kod

Uppgiften var relativt enkel att lösa. Vi använde vanlig ruby-syntax för att specificera specialfallen i slutet av policy-filen. Vi övervägde först att göra en egen funktion för det, men kunde inte se hur det skulle förenkla upplevelsen för användaren. För att slippa skriva redundanta funktioner överlagrade vi method_missing och fick således ett generellt beteende för funktioner som inte finns.

Testning

Testningen bestod helt enkelt utav att skapa en par exempelpersoner, räkna ut vilken `score` dessa skall ha för hand och jämföra detta med utdatan.

rdparse

Kod

Rdparse var en svårare uppgift att lösa. Vi valde att konvertera indata till syntaxträd där noderna representerar olika konstrukt i språket som alla kan utvärderas till ett booleskt värde med eval-metoden. Anropas rot-nodens eval-metod ges då värdet på hela uttrycket som trädet representerar.

Utöver trädstrukturen har vi även en hashtabell för att lagra variabler. Då variabler måste behållas mellan satser uppstår ett problem om de lagras lokalt i syntaxträden. Vår lösning är att lagra variablers namn och värde som nyckel-värde par. Variabelnoder i syntaxträd utvärderas till motsvarande värde i hashtabellen. Värt att notera är att odefinerade variabler alltid utvärderas till `false`.

Vår hashtabell `variables` är globalt definerad. Det är sällan populärt att göra så, då det lätt orsakar problem. Ett exempel som kan återskapas i vår kod är följande:

```
a = LogExp.new.logexParser
b = LogExp.new.logexParser

a.parse("(set var true)").eval  # sätt var till true i parsern *a*
puts(b.parse("var").eval)       # returnerar true trots att odefinerade
                                # variabler utvärderas till false
```

Detta beteende är antagligen inte förväntat och skulle kunna orsaka problem för omedvetna. Vi funderade en del på snygga lösningar till detta, men hittade ingen som inte kändes överkonstruerad i förhållande till uppgiftsstorleken.

Namngivningen av variabler är väldigt fri. De får ha i princip vilket namn som helst som är en följd av ordtecken (a-z, A-Z, 0-9 och _) som inte är `true` eller `false`.

Testning

Vi tycker att det är svårt vara helt uttömmande i testning av ett programmeringsspråk, även när det har relativt enkel grammatik. Vi testar grundläggande logiska operationer som negation, konjunktion och disjunktion uttömmande då möjliga kombinationer är få. Vi testar ett par kombinationer av uttryck för att se att de också fungerar, men grundtanken är att om alla beståndsdelar fungerar borde även alla kombinationer av dessa också fungera.