Evaluera ett uttryck

Kenan Dizdarevic

30 Januari 2023

Inledning

Syftet med denna uppgift är att evaluera ett matematiskt uttryck bestående av variabler. I föregående uppgifter har vi implementerat deriveringsregler och en miljö. Dessa delar skall kombineras vilket i sin tur möjliggör för oss att evaluera ett uttryck.

Undersökning

Uttryck

Aritmetiska operatorer representeras som tupler, vilket vi visade i *Derivatan*.

```
{:aritmetiskOperator, argument1, argument2}
```

Vi kommer återigen behöva använda oss av literaler. I *Derivatan* hade vi två literaler, tal och variabler. Vi behöver dock utöka vår kod och implementera en literal för rationella tal. Rationella talen kommer att representeras som {:kvot, täljare, nämnare}, literalen implementeras som följande.

```
@type literal() :: {:q, number(), number()}
```

Vi kan nu representera ett uttryck på samma vis som vi gjorde i *Derivatan*. Uttrycket $f(x) = 1/3 + 6x$ representeras i Elixir som:

```
f = {:add, {:q, 1, 3}, {:mul, {:num, 6}, {:var, x}}}
```

Evaluering

Vi skall nu implementera en funktion som har ett uttryck och en miljö som argument, den skall evaluera uttrycket till en literal. I denna uppgift kommer miljön från Elixirs egna implementation *Map()*. Värt att tillägga är att vi redan antar att variablerna i funktionen är associerade till ett värde i miljön.

Ett exempel på vår funktion när vi skall evaluera en multiplikation ser ut som följande:

```
def eval({:mul, e1, e2}, env) do mul(eval(e1, env), eval(e2, env)) end
```

Det enda vi gör är att vi anropar funktionen *mul/2* samtidigt som vi rekursivt anropar funktionen *eval/2* i *mul/2*. Det rekursiva anropet sker eftersom det är stor sannolikhet att det finns fler uttryck innanför det vi kollar på. Funktionerna för övriga operatorer är identiska, vi anropar endast en annan funktion som har möjligheten att evaluera den specifika operatorn.

De två delarna av funktionen som skiljer sig åt från de andra är:

```
def eval({:num, n}, _) do n end
def eval({:var, v}, env) do Map.get(env, v) end
```

Den första funktionen evaluerar ett tal, vi behöver inte bry oss om miljön då talen redan är definierade. I det andra fallet evaluerar vi en variabel, således måste vi använda miljön. I Elixirs egna implementation av *Map()* kan vi enkelt hitta ett värde som associeras med en variabel, detta görs med hjälp av funktionen *get/2*.

Funktionen för att evaluera ett uttryck är klar, vi behöver dock implementera funktionerna som faktiskt evaluerar uttrycket. Vi behöver dock ha i åtanke att vi kan stöta på rationella tal, vi måste således implementera en lösning för dem. Lösningen för att evaluera en multiplikation är följande:

```
def mul({:q, n, m}, {:q, x, y}) do {:q, n*x, m*y} end
def mul({:q, n, m}, a) do {:q, n*a, m} end
def mul(a, {:q, n, m}) do {:q, n*a, m} end
def mul(a, b) do a * b end
```

Vi har tre fall där vi behandlar multiplikation med rationella tal. I det första fallet multiplicerar vi två rationella tal med varandra. I andra och tredje fallet multiplicerar vi ett rationellt tal med ett heltal. Sista fallet är multiplikation med två heltal.

Vi behöver dock implementera ytterligare en funktion som förenklar vår beräkning. Om vi exempelvis har 2/4 så kan vi förenkla det till 1/2. Vi vill alltså hitta den minsta gemensamma nämnaren mellan talen och dividera bort den för att förenkla bråket. Koden för implementeringen ser ut som följande:

```
def gcd(a, 0) do a end
def gcd(a, b) do gcd(b, rem(a, b)) end
```

Denna implementering är i själva verket *Euklides algoritm*. Värt att ha i åtanke är att nämnaren alltid kommer vara det större talet. Algoritmen fungerar på så vis att man först kollar vad resten blir mellan talen. Sedan fortsätter man dividera nämnaren tills man inte har någon rest kvar. Det sista talet som man dividerade med är den gemensamma nämnaren. Det hanteras i det första fallet, den returnerar således den största gemensamma

nämnaren. För att få resten vid en division använder vi oss av Elixirs egna funktion *rem/2* som ger oss resten vid en division mellan två tal.

För att säkerställa att vår implementering fungerar provar vi att evaluera funktionen:

```
{:add, {:add, {:mul, {:num, 2}, {:var, :a}}, {:num, 3}}, {:q, 6, 8}}
```

Matematiskt är funktionen $2a + 3 + 6/8$, där a representeras med tupeln `{:a, 1}` vilket finns i vår miljö. Svaret är 23/16, vilket är korrekt. Vi ser även att svaret är förkortat så långt det går.

Slutsats

Slutligen ser vi att det är relativt enkelt att evaluera ett uttryck. Vi behöver dock ha förkunskap kring hur en miljö fungerar samt hur vi representerar uttryck.

Funktionen *eval/2* är inte komplicerad att implementera. Det svåra i denna uppgift är att beakta alla fall som de rationella talen medför. Vi måste veta hur vi utför matematiska operationer med dem samt hur vi förkortar dem på ett effektivt vis.