Derivatan

Kenan Dizdarevic

17 Januari 2023

Inledning

Att derivera en funktion innebär att vi söker efter en ny funktion som anger *förändringshastigheten* hos en känd funktion. Derivatans definition är en matematisk sats som gäller för alla funktioner men den blir väldigt snabbt invecklad att arbeta med. Därav finns det dervieringsregler som bidrar med matematiska förenklingar.

Vi skall i denna uppgift arbeta med funktioner som *symboliska uttryck* samt derivera dem i **Elixir**. En viktig del av denna uppgift är att förstå hur vi kan representera matematiska funktioner med de primitiva datatyper som Elixir har.

Undersökning

Funktioner

Att representera en funktion i Elixir kan göras på följande vis:

```
f = fn(x) -> ln(x) end
```

I detta fall är f en funktion med ett argument som vi kan nyttja. Om vi endast har tillgång till f kan vi inte veta att den endast tar emot ett argument samt att det är en logaritm.

För att vi skall kunna derivera funktioner med avseende på en viss variabel måste vi veta exakt hur funktionen ser ut. Dessa funktioner som finns i Elixir är inget som vi kan ta nytta av, vi måste därför hitta ett annat sätt att representera funktioner på.

Representation av funktioner

Elixir ger oss en mycket kraftfull datatyp som vi skall använda för att representera uttryck, en så kallad *atom*. En atom används för att representera variabler och konstanter som exempelvis x och π. En atom kan också representera enumerationstyper och matematiska operator, där vi skall använda den sistnämnda.

Vi kommer att representera alla tal som tupler på formen:

```
{:num, c}
```

, där **c** är en integer eller ett flyttal. Variabler kan också representeras som tupler på formen:

```
{:var, v}
```

, där **v** är en atom. Detta kommer vara *literaler* i vår kod och de skrivs på följande vis:

```
@type literal() :: {:num, number()} | {:var, atom()}
```

Uttryck

Funktioner består av matematiska uttryck, vi behöver således ett meningsfullt sätt att uttrycka aritmetiska operatorer på. Vi har infört *literaler* i vår kod, dessa skall vi självfallet utnyttja. Dessa tupler kommer att ha en atom som första element, den beskriver operatorn. Några av uttrycken som vi har i vår kod är:

```
@type expr() :: literal()
| {:ln, expr()}
| {:div, expr(), expr()}
| {:sqrt, expr()}
| {:sin, expr()}
```

Detta innebär att vi nu har möjlighet att uttrycka funktioner som vi sedan kan derivera. Funktionen $f(x) = 5\sqrt{x}$ kan representeras i Elixir som:

```
f = {:mul, {:num, 5}, {:sqrt, {:var, x}}}
```

Syntaxen är svårläst men den medför fördelar när vi arbetar med funktioner i Elixir.

Derivator

Vi har en fungerande syntax för funktioner, vi kan således börja implementera reglerna för derivator. De första fyra reglerna som vi skall implementera är *förstagradsfunktioners derivata*, *nolltegradsfunktioners derivata*, *additionsregeln* och *multiplikationsregeln*. De två första fallen kan ses som enkla basfall där vi endast returnerar en konstant. Additionsregeln implementeras rekursivt i Elixir. Vi skall addera derivatan av första termen med derivatan av andra termen. Dessa derivator kan i sin tur bestå av fler derivator. Koden ser ut som följande:

```
def deriv({:add, e1, e2}, v) do
    {:add, deriv(e1, v), deriv(e2, v)}
end
```

Multiplikationsregeln är också enkel att implementera. Matematisk ser den ut som följande:

$$\frac{d}{dx}[f(x) \cdot g(x)] = \frac{df}{dx} \cdot g(x) + f(x) \cdot \frac{dg}{dx}$$

I Elixir implementeras detta som:

```
def deriv({:mul, e1, e2}, v) do
    {:add,
      {:mul, deriv(e1, v), e2},
      {:mul, e1, deriv(e2, v)}
    }
end
```

Det koden gör är att om vi har en multiplikation med två uttryck skall vi addera summan av två olika multiplikationer. I den första multiplikationen deriverar vi första uttrycket med avseende på en viss variabel och multiplicerar med andra uttrycket. I den andra multiplikationen gör vi tvärtemot.

Vidare skall vi implementera möjligheten att derivera fler funktioner, exempelvis trigonometriska och logaritmiska funktioner. När vi konstruerar dessa derivator bör vi ha i åtanke att de skall fungera för det generella fallet. Derivatan för $ln(x)$ är trivial, om vi istället skall derivera $ln(f(x))$ blir det mer omständligt. För att derivera det senare fallet använder vi oss av *kedjeregeln*, denna regel gäller även för det första fallet. Vi implementerar derivatan för logaritmer som följande:

```
def deriv({:ln, e}, v) do
    {:mul,
      deriv(e, v),
      {:div, {:num, 1}, e}
    }
end
```

Derivatan av en logaritm är en multiplikation av den inre derivatan som anropas rekursivt multiplicerat med en division på 1 dividerat med den inre funktionen.

Förenkling

Derivatorna är korrekta men svårlästa, vi bör således implementera funktioner som förenklar uttrycken. Vi skall förenkla uttrycken genom att alla funktioner evalueras. Sedan skall vi bryta ned funktionen i mindre beståndsdelar

och förenkla dem. När vi förenklar dem skall vi exempelvis ta bort alla multiplikationer med 0. Fallet för att ta bort en multiplikation med 0 kommer se ut som följande:

```
def simplify({:mul, e1, e2}) do
    simplify_mul(simplify(e1), simplify(e2))
end
def simplify_mul({:num, 0}, _) do {:num, 0} end
```

Det som sker i koden är att när vi går in i *simplify* med en multiplikation som parameter så kommer vi anropa *simplify_mul*. I denna funktion kommer allt som multipliceras med 0 att sättas till 0. Detta anropas rekursivt vilket innebär att alla multiplikationer med 0 kommer att tas bort från resultatet.

När vi implementerar förenklingen för övriga fall använder vi samma tankesätt. Vilka fall kommer ge oss ett konstant värde som vi kan skriva ut direkt? Ett tydligt fall är när vi dividerar 0 med något tal som är skiljt från 0, det blir alltid 0. Ett annat fall är $\sqrt{1}$, som alltid är ekvivalent med 1.

När vi förenklar uttrycken inser vi att det behövs en konvention för hur vi skall representera resultatet. Vill vi representera ett svar som $((2x)/(3x+2))$ eller $2x/(3x+2)$? Detta är upp till oss själva att bestämma, så länge de matematiska reglerna efterföljs.

Vi har implementerat ytterligare en funktion:

```
def pprint({:mul, e1, e2}) do
    "#{pprint(e1)} * #{pprint(e2)}"
end
```

Funktionen som den har är att ge användaren ett tydligare svar när det skrivs ut.

Slutsats

Sammanfattningsvis ser vi att det inte krävs mycket kod för att kunna derivera funktioner. Allt kommer att anropas rekursivt och användas fler gånger.

Det viktiga är att vara uppmärksam på vad som skall deriveras, samt hur deriveringsreglerna är definierade.

När vi förenklar bör vi analysera alla fall som kan uppstå samt förenkla dem så långt som möjligt, med vår egna representation. Detta leder till ett kort och koncist program som kan derivera matematiska funktioner.