Programmering og Problemløsning, 2019

Træstrukturer — Part III

Martin Elsman

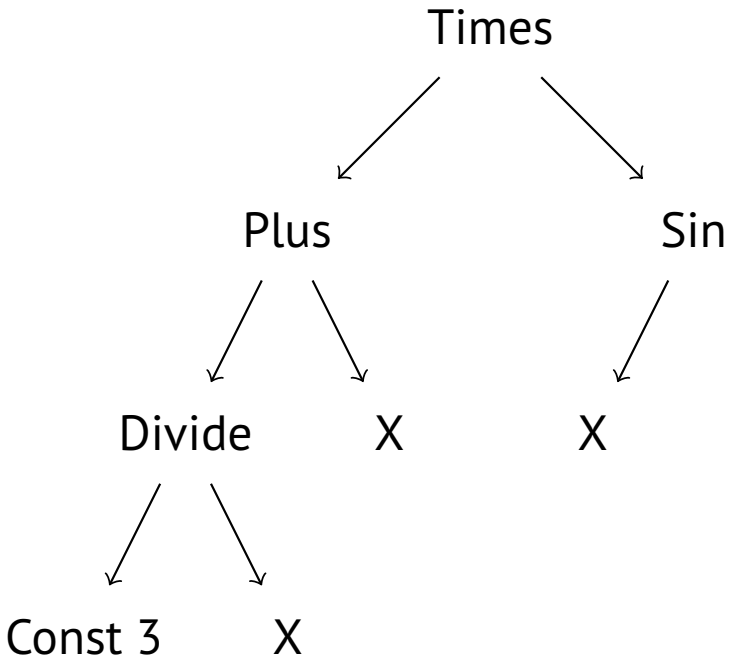
Datalogisk Institut
Københavns Universitet
DIKU

31. oktober, 2019

Udtrykstræer og Symbolsk Differentiering

Emner for i dag:

1. **Udtrykstræer.**
   Sum-type definition (funktioner af en variabel)
2. **Evaluering af udtryk.**
3. **Simpel pretty-printing.**
   Pretty bad...
4. **Symbolsk differentiering.**
   Vi implementerer gymnasiereglerne for differentiering...
5. **Generering af LaTeX kode.**
   Vi prøver at undgå unødige parenteser...
6. **Simplificering.**
   Vi implementerer forskellige regneregler til simplificering...



$$(\frac{3}{x} + x) \cdot \sin x$$

Regneudtryk i én variabel

Et udtryk kan (rekursivt) være på en af følgende former:

1. Variablen x.
2. En konstant $c \in \mathbb{R}$.
3. To udtryk adskilt af $+$.
4. To udtryk adskilt af $-$.
5. To udtryk adskilt af $\cdot$ (gange).
6. To udtryk adskilt af en brøkstreg.
7. Et udtryk opløftet til en potens p.
8. Funktionen $\sin$ anvendt på et udtryk.
9. Funktionen $\cos$ anvendt på et udtryk.
10. Funktionen $\log$ anvendt på et udtryk.
11. Konstanten e opløftet til en potens angivet med et udtryk.
12. Et udtryk omgivet af parenteser.

Eksempel: $\frac{\sin x}{\cos x} + (x + 1)^2$

Sum-Type til Udtrykstræer

```
type expr = X                          // The variable x
          | Const of float             // Constants
          | Plus of expr * expr        // Addition
          | Minus of expr * expr
          | Times of expr * expr
          | Divide of expr * expr
          | Power of expr * float      // a^d, e.g., (x + 2)^2.3
          | Sin of expr
          | Cos of expr
          | Log of expr
          | Exp of expr                // e^a, e.g., e^(2+x)
```

Bemærk

- Vi understøtter kun funktioner af én variabel.
- Parenteser er implicitte; træet indkoder parenteser direkte.
- Dvs: Parsning (at konvertere en streng til et udtrykstræ) er en separat problemstilling vi vil se på en anden gang...

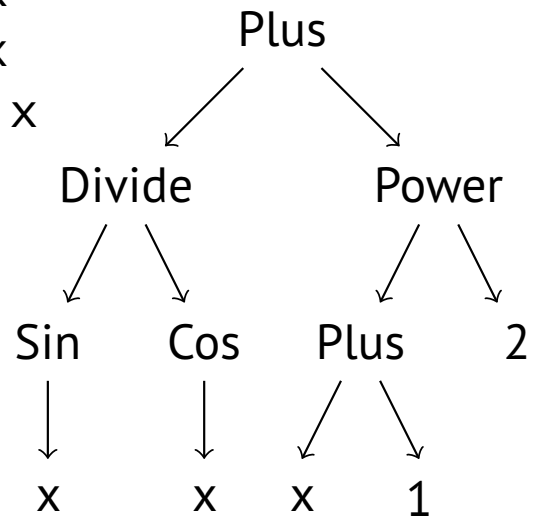
Evaluering af Udtryk (fortolkning)

Et udtrykstræ repræsenterer kroppen på en funktion $f(x)$.

```
let rec eval e x =    // assume a value for the variable X
  match e with
    | X -> x
    | Const c -> c
    | Plus (e1, e2) -> eval e1 x + eval e2 x
    | Minus (e1, e2) -> eval e1 x - eval e2 x
    | Times (e1, e2) -> eval e1 x * eval e2 x
    | Divide (e1, e2) -> eval e1 x / eval e2 x
    | Power (e, p) -> (eval e x) ** p
    | Sin e -> sin (eval e x)
    | Cos e -> cos (eval e x)
    | Log e -> log (eval e x)
    | Exp e -> exp (eval e x)

let ee = Plus (Divide (Sin X, Cos X),
               Power (Plus (X, Const 1.0), 2.0))

let v = eval ee 3.0   // evaluates to 15.85745346
```



Simpel Pretty-Printing (pretty bad)

```
let par s = "(" + s + ")"
let rec pp e : string =
  match e with
    | X -> "x"
    | Const c -> sprintf "%g" c
    | Plus (e1, e2) -> par(pp e1 + "+" + pp e2)
    | Minus (e1, e2) -> par(pp e1 + "-" + pp e2)
    | Times (e1, e2) -> par(pp e1 + "*" + pp e2)
    | Divide (e1, e2) -> par(pp e1 + "/" + pp e2)
    | Power (e, p) -> par(pp e + "^" + sprintf "%g" p)
    | Sin e -> "sin " + pp e
    | Cos e -> "cos " + pp e
    | Log e -> "log " + pp e
    | Exp e -> "exp " + pp e
```

Hvad er der galt ved denne pretty-printer?

1. _____
2. _____

Differentieringsregler

$h(x)$	$h'(x)$
x	1
c	0
$f(x) + g(x)$	$f'(x) + g'(x)$
$f(x) - g(x)$	$f'(x) - g'(x)$
$f(x) \cdot g(x)$	$f'(x) \cdot g(x) + f(x) \cdot g'(x)$
$\frac{f(x)}{g(x)}$	$\frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{(g(x))^2}$
x^n	$n \cdot x^{n-1}$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\log x$	$\frac{1}{x}$
e^x	e^x
$f(g(x))$	$f'(g(x)) \cdot g'(x)$

Symbolsk Differentiering i F#

```
let rec ddx e =
  match e with
  | X -> Const 1.0
  | Const c -> Const 0.0
  | Plus (e1, e2) -> Plus(ddx e1, ddx e2)
  | Minus (e1, e2) -> Minus (ddx e1, ddx e2)
  | Times (e1, e2) -> Plus (Times (ddx e1, e2),
                            Times (e1, ddx e2))
  | Divide (e1, e2) ->
      Divide(Minus (Times (ddx e1, e2), Times (e1, ddx e2)),
             Power (e2, 2.0))
  | Power (e, p) -> Times (Times (Const p, Power (e, p-1.0)),
                           ddx e)
  | Sin e -> Times (Cos e, ddx e)
  | Cos e -> Times (Minus (Const 0.0, Sin e), ddx e)
  | Log e -> Times (Divide (Const 1.0, e), ddx e)
  | Exp e -> Times (Exp e, ddx e)
```

Symbolsk Differentiering i F#

```
> ddx ee;;
val it : expr =
  Plus
    (Divide
       (Minus
          (Times (Times (Cos X,Const 1.0),Cos X),
           Times (Sin X,Times (Minus (Const 0.0,Sin X),
                               Const 1.0))),
        Power (Cos X,2.0)),
     Times
       (Times (Const 2.0,Power (Plus (X,Const 1.0),1.0)),
        Plus (Const 1.0,Const 0.0)))

> pp (ddx ee);;
val it : string =
   "(((((cos x*1)*cos x)-(sin x*((0-sin x)*1)))/  ...
    (cos x^2))+((2*((x+1)^1))*(1+0)))"
```

Generering af LaTeX kode

I stedet for at generere tekst vi kan generere LaTeX kode.

Strategien er at lade vores pretty-printer returnere **et par** af to værdier:

1. En streng der repræsenterer den genererede LaTeX kode.
2. Et tal (præcedens) der siger hvor stærkt det underliggende udtryk binder.

Eksempler

- `pp (Plus(X, Const 1.0))` $\rightsquigarrow$ `("x+1.0",3)`
- `pp (Times(Const 2.0, Const 3.0))` $\rightsquigarrow$ `("2.0\\cdot 3.0",4)`

Betragt udtrykket `Times(X,Plus(X,Const 1.0))`:

Når vi skal pretty-printe det højre argument til `Times` skal vi sætte paranteser rundt om udtrykket `"x+1.0"`, da præcedens-tallet 3 er mindre end (eller lig med) præcedens for konstruktøren `Times`:

- `pp (Times(X, Plus(X, Const 1.0)))` $\rightsquigarrow$ `("x\\cdot (x+1.0)",4)`

Generering af LaTeX kode—del 0

Nogle hjælpfunktioner samt præcedens-konstanter:

```
let p_plus   = 3      // precedence
let p_times  = 4
let p_divide = 0      // division with horizontal bar
let p_unop   = 5
let p_max    = 1000

// [par p s] adds parentheses around s if the relative
// precedence p (p_parent - p_child) is positive or 0
let par p s =
  if p < 0 then s else "(" + s + ")"
```

Igen:

- Ingen parenteser er nødvendige omkring et barn hvis dets udtryk binder stærkere end forældre-udtrykket.

Generering af LaTeX kode—del 1

```
let toLaTeX e =
  let rec pp e : string * int =
    match e with
    | X -> ("x", p_max)
    | Const c -> (sprintf "%g" c, p_max)
    | Plus (e1, e2) -> pp_binop ("+",p_plus) e1 e2
    | Minus (e1, e2) -> pp_binop ("-",p_plus) e1 e2
    | Times (e1, e2) -> pp_binop ("\\cdot ",p_times) e1 e2
    | Divide (e1, e2) ->
      let (s1,s2,_) = pp_bin p_divide e1 e2
      in ("\\frac{" + s1 + "}{" + s2 + "}", p_max)
    | Power (e, k) -> let (s1,s2,p) = pp_bin p_unop e (Const k)
                      in (s1 + "^{" + s2 + "}", p)
    | Sin e -> pp_unop "sin " e
    | Cos e -> pp_unop "cos " e
    | Log e -> pp_unop "log " e
    | Exp e -> let (s,a) = pp e
               in ("e^{" + par (p_unop-a) s + "}", p_unop)
  and ...
```

Generering af LaTeX kode—del 2

```
...
and pp_bin p e1 e2 =
  let (s1,p1) = pp e1
  let (s2,p2) = pp e2
  in (par (p-p1) s1, par (p-p2) s2, p)
and pp_binop (op,p) e1 e2 =
  let (s1,s2,p) = pp_bin p e1 e2
  in (s1 + op + s2, p)
and pp_unop op e =
  let (s,p) = pp e
  in ("\\"+op + par (p_unop-p) s, p_unop)
in fst(pp e)
```

Eksempel: `toLaTeX (ddx ee)` $\rightsquigarrow$

```
"\frac{(\cos x\cdot 1)\cdot \cos x-\sin x\cdot ((0-\sin x)...
 \cdot 1)}{(\cos x)^{2}}+(2\cdot (x+1)^{1})\cdot (1+0)"
```

$$\frac{(\cos x \cdot 1) \cdot \cos x - \sin x \cdot ((0 - \sin x) \cdot 1)}{(\cos x)^2} + (2 \cdot (x+1)^1) \cdot (1+0)$$

Simplificering af Symbolske Udtryk

Der er tilsyneladende en række konstruktioner der kan simplificeres...

```
ddx ee ⇝
  Plus
    (Divide
       (Minus
          (Times (Times (Cos X,Const 1.0),Cos X),
           Times (Sin X,Times (Minus (Const 0.0,Sin X),
                               Const 1.0))),
        Power (Cos X,2.0)),
     Times
       (Times (Const 2.0,Power (Plus (X,Const 1.0),1.0)),
        Plus (Const 1.0,Const 0.0)))
```

Eksempler:

- `Times (Cos X,Const 1.0)` $\Longrightarrow$ `Cos X`
- `Plus (Const 1.0,Const 0.0)` $\Longrightarrow$ `Const 1.0`

En Simplificeringsfunktion i F# — del 1

```
let rec simplify e =
  match e with
   | Plus (e1, Const 0.0) -> simplify e1
   | Plus (Const 0.0, e2) -> simplify e2
   | Plus (Const a, Const b) -> Const (a + b)
   | Plus (e1, e2) -> Plus (simplify e1, simplify e2)
   | Minus (e1, Const 0.0) -> simplify e1
   | Minus (Const a, Const b) -> Const (a - b)
   | Minus (e1, Minus(Const 0.0, e2)) ->
      Plus (simplify e1, simplify e2)
   | Minus (e1, e2) -> Minus (simplify e1, simplify e2)
   | Divide (Const 0.0, e2) -> Const 0.0
   | Divide (e1, Const 1.0) -> simplify e1
   | Divide (e1, e2) -> Divide (simplify e1, simplify e2)
   | Power (e, 1.0) -> simplify e
   | Power (e, c) -> Power (simplify e,c)
   ...
```

En Simplificeringsfunktion i F# — del 2

```
let rec simplify e =
  match e with
  ...
  | Times (e1, Const 0.0) -> Const 0.0
  | Times (Const 0.0, e2) -> Const 0.0
  | Times (e1, Const 1.0) -> simplify e1
  | Times (Const 1.0, e2) -> simplify e2
  | Times (Const a, Const b) -> Const (a * b)
  | Times (e1,Plus(e2,e3)) ->
    simplify (Plus(Times(e1,e2),Times(e1,e3)))
  | Times (e1,Minus(e2,e3)) ->
    simplify (Minus(Times(e1,e2),Times(e1,e3)))
  | Times (e1, e2) ->
    if e1 = e2 then simplify (Power (e1,2.0))
    else Times (simplify e1, simplify e2)
  | Sin e -> Sin (simplify e)
  | Cos e -> Cos (simplify e)
  | Log e -> Log (simplify e)
  | Exp e -> Exp (simplify e)
  | _ -> e
```

Det tager flere steps før "simplifyeren" stabiliseres:

`toLaTeX(ddx ee)` $\rightsquigarrow$

$$\frac{(\cos x \cdot 1) \cdot \cos x - \sin x \cdot ((0 - \sin x) \cdot 1)}{(\cos x)^2} + (2 \cdot (x + 1)^1) \cdot (1 + 0)$$

`toLaTeX(simplify(ddx ee))` $\rightsquigarrow$

$$\frac{\cos x \cdot \cos x - \sin x \cdot (0 - \sin x)}{(\cos x)^2} + (2 \cdot (x + 1) + 0)$$

`toLaTeX(simplify(simplify(ddx ee)))` $\rightsquigarrow$

$$\frac{(\cos x)^2 - (0 - (\sin x)^2)}{(\cos x)^2} + (2 \cdot x + 2)$$

`toLaTeX(simplify(simplify(simplify(ddx ee))))` $\rightsquigarrow$

$$\frac{(\cos x)^2 + (\sin x)^2}{(\cos x)^2} + (2 \cdot x + 2)$$

Komplet Simplificering

Med generisk lighed og rekursion kan vi let skrive en "simplifyer" der kalder `simplify` igen og igen indtil udtrykket ikke længere simplificeres:

```
let rec simplifyMax e =
  let se = simplify e
  in if se = e then e else simplifyMax se
```

Komplet Simplificering i Aktion:

```
toLaTeX(simplifyMax(ddx ee)) ⇝
  "\\frac{(\\cos x)^{2}+(\\sin x)^{2}}{(\\cos x)^{2}}+(2\\cdot
    x+2)"
```

LaTeX Fortolkning:

$$\frac{(\cos x)^2 + (\sin x)^2}{(\cos x)^2} + (2 \cdot x + 2)$$