Berekening van een polynoom in Elm

In de docentenworkshop van 10 juni 2020 behandeld.

We willen functies zoals $x^2 + 3x + 1$ uitrekenen met Elm. In de wiskunde noemt men zoiets een *polynoom*. De som van een machten van x, evt. vermenigvuldigd met een factor, een zogenaamde *coëfficiënt*.

In de Elm-REPL notebooks is het symbool ^ voor machtsverheffing niet beschikbaar. Dus we definiëren eerst een functie **pow** die machtsverheffen uitvoert.

In [1]:
```
pow a b = \
    if b <= 0 then 1 \
    else a * pow a (b-1)
```

```
<function> : number -> number1 -> number
```

In [2]:
```
pow 10 0
pow 2 10
pow 10 6
```

```
1 : number
1024 : number
1000000 : number
```

Met powerfunctie

Eerste stap is gewoon een functie die bovengenoemde polynoom uitrekent.

In [3]:
```
poly0 x = \
    pow x 2  + 3 * x  +  1
```

```
<function> : number -> number
```

In [4]:
```
poly0 2
```

```
11 : number
```

We willen nu een functie die werkt voor alle polynomen werkt. Een polynoom is bepaald door een lijst van coëfficiënten. Iedere coëfficiënt hoort bij een macht van x. Als een macht er niet in voorkomt, zetten we een 0 in de lijst.

Dus [1, 3, 1] stelt $x^2 + 3x + 1$ voor. En [2, 0, 3, 0, 5] stelt $2x^4 + 3x^2 + 5$ voor. (Let op! straks draaien we de volgorde om.)

In [5]:
```
poly1 coefs x = \
    case coefs of \
    [] -> 0 \
    c :: cs -> c * pow x (List.length cs) + poly1 cs x
```

```
<function> : List number -> number -> number
```

In [6]:
```
poly1 [1,3,1] 2
```

```
11 : number
```

Merk op dat de functie *poly0* gelijk is aan *poly1 [1,3,1]* (laat het argument *x* weg, en je hebt een functie)

Zonder powerfunctie

We zien dat steeds de macht van x moet worden uitgerekend. Eerst x^4, dan x^3, enzovoort. Dat kan handiger. Als je het in omgekeerde volgorde zou doen, hoef je alleen maar steeds de vorige macht te nemen en die met x te vermenigvuldigen. We moeten dan wel die vorige macht onthouden. Dat kan met een extra parameter, lastpowerofx. (Verderop zullen we die weer lozen.)

En verder nemen we aan dat de lijst coëfficiënten nu andersom geordend is en eindigt met de hoogste macht van x.

In [7]:
```
poly2 coefs lastpowerofx x = \
    case coefs of \
    [] -> 0 \
    c::cs -> c * lastpowerofx  +  poly2 cs (lastpowerofx * x) x
```

```
<function> : List number -> number -> number -> number
```

In [8]:
```
poly2 [1,3,1] 1 2
```

```
11 : number
```

Steeds de startwaarde 1 voor lastpowerofx aangeven is wat onhandig. Daarom definiëren we een niet-recursieve functie *poly3* die de recursieve *poly2* aanroept met de goede startwaarde. Dit is een patroon dat je vaak ziet bij recursieve functies.

In [9]:
```
poly3 coefs x = poly2 coefs 1 x
```

```
<function> : List number -> number -> number
```

of (de x aan beide kanten kun je weglaten)

In [10]:
```
poly3 coefs= poly2 coefs 1
```

```
<function> : List number -> number -> number
```

Zonder lastpowerofx

Het kan nog iets strakker. Kijk naar het polynoom $5 + 4x + 3x^2 + 2x^3$. Je kunt dat ook schrijven als $5 + x(4 + x(3 + x(2 + 0)))$ Er komt precies hetzelfde uit, maar je vermenigvuldigt niet met *machten* van x, maar alleen met x. Als we deze berekening gebruiken in onze poly-functie hoeven we dus geen macht van x te bewaren en kunnen we de parameter *lastpowerofx* weer lozen.

Deze manier om een polynoom te berekenen heet het *Hornerschema*, naar de Britse wiskundige William Horner.

In [11]:
```
poly4 coefs x = \
    case coefs of \
    [] -> 0 \
    c::cs -> c + x * (poly4 cs x)
```

```
<function> : List number -> number -> number
```

In [12]:
```
poly4 [1,3,1] 2
```

```
11 : number
```

Pointfree

(Niet in de workshop) We kunnen nog een stap maken, waarbij we een *pointfree* definitie van de functie geven. Het is een stijl in het functioneel programmeren waarbij je zoveel mogelijk functies met behulp van andere functies definieert en niet met losse waarden.

List.foldr

Als je de som van de elementen van een lijst wilt uitrekenen kun je dat als volgt, niet pointfree, doen.

In [13]:
```
som xs = \
    case xs of \
    [] -> 0 \
    y::ys -> y + som ys

som [1,2,3,4]
```

```
<function> : List number -> number
10 : number
```

Je ziet dat in de functie het eerste element van de lijst wordt afgepeld en gebruikt wordt in de optelling. Je zou dat element een punt kunnen noemen en deze functie is daarom niet pointfree. Maar met behulp van de functie *foldr* kunnen we dit anders aanpakken. De functie *foldr* neemt net als map een functie en past die toe op alle elementen van de lijst. Maar map gebruikt een functie met één parameter en het resultaat is weer een lijst. Bij *foldr* gebruik je een functie met twee parameters, bijvoorbeeld optellen, en die wordt steeds toegepast op een element van de lijst en het resultaat dat je tot dan toe had. De lijst wordt als het ware opgevouwen tot één getal, de som in dit geval. Je moet ook nog de startwaarde geven, deze waarde krijg je er ook uit als foldr op een lege lijst werkt. Hieronder zie je hoe je *foldr* toepast. Operaties als *foldr* heten ook wel *reduce*-operaties, omdat ze een lijst terugbrengen tot één waarde.

Als je in Elm de functie van het optellen nodig hebt, schrijf je **(+)**.

In [14]:
```
List.foldr (+) 0 [1,2,3,4]
```

```
10 : number
```

Alle elementen *vermenigvuldigen* gaat net zo makkelijk. Je moet wel een andere startwaarde gebruiken.

In [15]:
```
List.foldr (*) 1 [1,2,3,4]
```

```
24 : number
```

De r in de naam *foldr* staat voor right, omdat de berekening rechts begint. Hij rekent dit uit als 1+(2+(3+(4+0))). Er is ook een variant *foldl* die links begint. Bij optellen komt er hetzelfde uit, maar bij andere functies kunnen *foldr* en *foldl* wel verschillende uitkomsten geven.

Berekening met foldr

We passen nu *foldr* toe om de polynoom-berekening pointfree te maken. We moeten daarvoor de functie bedenken die we aan *foldr* gaan meegeven. (Bij de som-berekening was dat de +.) Kijk daarom naar de recursieve cases van de functies *som* en *poly4*:

y::ys -> y + som ys

y::ys -> y + x * (poly4 x ys)

In de som-functie worden de head van de lijst, y, en de tail met daarop de recursieve call, som ys, gecombineerd met de +-operator. Bij poly4 is het ietsje ingewikkelder: de tail met recursieve call (inclusief de extra parameter x) wordt eerst met x vermenigvuldigd en dan pas opgeteld bij de head. De functie die we nodig hebben in plaats van de plus is dus (bijna): *maaldanplus a b = a + x * b*. Maar omdat x moet kunnen variëren, moet die als paramter meegegeven worden en krijg je de volgende functie.

In [16]:
```
maaldanplus x a b = a + x * b
```

```
<function> : number -> number -> number -> number
```

En de nieuwe polynoomfunctie wordt:

In [17]:
```
poly5 coefs x = List.foldr (maaldanplus x) 0 coefs
poly5 [1,3,1] 2
```

```
<function> : List number -> number -> number
11 : number
```

De functie *maaldanplus* hoeft niet per se apart gedefinieerd te worden. We kunnen haar ook als *lambda-expressie* meegeven. Omdat *x* op die plek bekend is, hebben we die niet als parameter van de lambda-functie nodig.

Zo kunnen we de hele polynoomberekening in één regel definiëren.

In [18]:
```
poly6 coefs x = List.foldr (\a -> \b -> a + x * b) 0 coefs
poly6 [1,3,1] 2
```

```
<function> : List number -> number -> number
11 : number
```

Type *Markdown* and LaTeX: α^2