Module 28 Procedures

Onderwerp	Het schrijven van procedures.
Voorkennis	Module 3, 7, 8, 25, 26,27.
Expressies	`proc, end proc, local, global, description, tracelast, showstat, :-, return, print, evaln, nargs, args, interface, verboseproc`
Zie ook	Module 29, 30

28.1  Inleiding

De meeste Maple-commando's hebben de vorm van een procedure-aanroep. Als men bijvoorbeeld het commando `int(2*x, x=0..2);` geeft, past men de *procedure* met de naam `int` toe op de *argumenten* `2*x` en `x=0..2`. Het resultaat 4 wordt toegekend aan de ditto-operator `%`. Als de procedure niet op de juiste manier wordt aangeroepen geeft Maple een foutmelding. Na `int( 2*x );` reageert het bijvoorbeeld met

```
Error, (in int) integration range or variable must be
                        provided.
```

In deze module gaan we nader in op de belangrijkste aspecten van het zelf schrijven van procedures. De behandeling hiervan is niet uitputtend, maar we zullen er in de meeste gevallen toch goed mee uit de voeten kunnen. Verdere details volgen in Module 29 en 30. Bovendien worden enkele termen die we in de literatuur over programmeren veelvuldig tegenkomen aan de lezer bekendgemaakt. Hierdoor zal het raadplegen van bijvoorbeeld `?procedures` (en daarin vooral: 'See Also') minder problemen opleveren.

Aan de hand van een aantal voorbeelden komen in deze module de volgende onderwerpen aan bod:

(1) De basisopzet van een procedure
(2) Lokale en globale variabelen
(3) Recursie
(4) Invoer- en uitvoerparameters

Hiermee kunnen we in principe procedures maken die geschikt zijn voor eigen, eenmalig gebruik. Een aantal andere zaken, zoals testen op de juiste invoerparameters en het genereren van foutmeldingen, die

vooral van belang zijn bij het maken van procedures voor algemener gebruik, komen aan de orde in de volgende module.
Raadpleeg ook nog eens Module 7 voor uitleg van veelgebruikte termen als heading, body, formele en actuele parameters enzovoort.

28.2 Basisopzet van procedures

In Module 7 hebben we al een procedure gemaakt voor het bepalen van de tweedegraadspolynoom door de punten $(1, y_1)$, $(2, y_2)$ en $(3, y_3)$, zie de voorbeeldsessie op blz. 93. We nemen de daar gepresenteerde procedure hier (bijna) ongewijzigd over.

Voorbeeldsessie

```
> pol := proc(y1,y2,y3)
    local p, x, a, b, c, stelsel, s;
  description "Maakt een tweedegraadsfunctie door de punten (1,y1),
  (2,y2) en (3,y3).";
    p := x -> a*x^2 + b*x + c;
  # Los a,b en c op en substitueer in p(x):
    stelsel := {p(1)=y1, p(2)=y2, p(3)=y3};
    s := solve( stelsel, {a,b,c} );
    subs( s, p(x) );
    unapply( %, x )
  end proc:
```

```
> p := pol(4,7,14);
```

$$p := x \to 2\,x^2 - 3\,x + 5$$

```
> print(pol);
```

proc(*y1*, *y2*, *y3*)
local *p*, *x*, *a*, *b*, *c*, *stelsel*, *s*;
description "Maakt een tweedegraadsfunctie door de punten (1,y1), (2,y2) en (3,y3).";
$p := x \to a * x^2 + b * x + c$;
$stelsel := \{p(1) = y1, p(2) = y2, p(3) = y3\}$;
$s :=$ solve($stelsel$, $\{a, b, c\}$);
subs(s, $p(x)$);
unapply(%, x)
end proc

Toelichting

We hebben in deze versie van de procedure `pol` twee soorten commentaar toegevoegd. De '`#`-vorm' kennen we al; deze kan uiteraard in procedures ook gewoon op elke willekeurige plaats gebruikt worden.

`description` Nieuw is het commentaar achter het woord `description`. Dit kan direct achter de declaratie van lokale (en globale) variabelen worden gegeven; het moet tussen 'string-quotes' worden gezet. Het verschil met `#`-commentaar is dat zo'n `description` óók wordt getoond als we Maple vragen de inhoud van de procedure te tonen. ◇

Een procedure-definitie heeft in het algemeen de volgende structuur:

`proc`	`procnaam :=` `  proc(args)`	naam van de procedure; `args` is een *sequence* van de formele parameters
	`local` ...	declaratie van de lokale variabelen
	`global` ...	declaratie van de globale variabelen, zie verderop, §28.4
	statements waarmee het resultaat wordt berekend	
	⋮	
	laatste statement	resultaat van de berekening: de waarde die `procnaam(args)` moet krijgen
`end proc`	`end proc:`	Afsluiting van de proceduredefinitie; de Maple-woorden `proc` en `end proc` horen bij elkaar als een linker- en rechterhaakje.

Let op dat er *geen* puntkomma of dubbele punt ná `proc( )` komt. Ook het statement vóór `end proc` hoeft niet met een puntkomma of dubbele punt te worden afgesloten (mag wel). Verder maakt het binnen een procedure niet uit of dubbele punten of puntkomma's worden gebruikt.

De procedure `pol` levert de gevraagde functie af, omdat dit het resultaat is van het laatste statement dat is uitgevoerd in de body. We noemen dat het *uitvoerstatement*[63]. Via de heading van de procedure `pol` worden de waarden van `y1`, `y2` en `y3` aan de body 'bekendgemaakt'. We noemen dit het *doorgeven van parameters*, of, in het Engels, *parameter passing*.

`local` Door het statement `local p, x, a, b, c, stelsel, s;` is de variabele `y` *gedeclareerd* als *lokale variabele*. Lokale variabelen zijn alleen bekend binnen de body van de procedure waarin ze zijn gedeclareerd.

[63] Het uitvoerstatement hoeft niet per se het laatste statement vóór het woord `end proc` te zijn. Zie §28.5.

Zodra de verwerking van de statements in de body is beëindigd, worden de waarden die de lokale variabelen op dat moment hebben, vergeten. Dit is prettig, want we wilden dat `pol` alleen de gevraagde functie zou afleveren. Alle ‘hulpvariabelen’ die `pol` hierbij heeft gebruikt zijn na aflevering van deze oplossing niet meer van belang. In §28.4 gaan we hier verder op in.

28.3  Parameters en lokale variabelen

Dat het van het grootste belang is om een goed onderscheid te maken tussen lokale variabelen en invoerparameters blijkt uit de volgende Maple-sessie. Hierin proberen we van de ‘interval-halveringsalgoritme’ van het voorbeeld in §27.4 (zie blz. 425) een procedure te maken. We kiezen de functie f, de beginschattingen x_l en x_r en de tolerantie ϵ als formele (invoer)parameters. De benadering van de x-waarde van een nulpunt is de uitvoer.

Voorbeeldsessie

```
> nulp1 := proc(f,xl,xr,eps)
    local xm;
    xm := 0.5*(xr+xl):
    while abs( f(xm) ) > eps do
    if f(xm)*f(xl) < 0
      then xr := xm
      else xl := xm
    end if:
    xm := 0.5*(xr+xl)
    end do;
    xm
  end proc:

> f := x -> 9*x^3 + 5*x^2 + 8*x + 6:

> nulp1( f, -1, 1, 0.01 );

Error, (in nulp1) illegal use of a formal parameter

> tracelast;

 nulp1 called with arguments: f, -1, 1, .1e-1
 #(nulp1,4): xr := xm

Error, (in nulp1) illegal use of a formal parameter

 locals defined as: xm = 0.

> showstat(nulp1);
```

```
nulp1 := proc(f, xl, xr, eps)
local xm;
   1   xm := .5*(xr+xl);
   3     if f(xm)*f(xl) < 0 then
   4       xr := xm
         else
   5       xl := xm
         end if;
   6     xm := .5*(xr+xl)
       end do;
   7   xm
end proc
```

Toelichting

We hebben de procedure gemaakt door het voorbeeld van §27.4 aan de bovenkant aan te vullen met een heading en de declaratie van de lokale variabele en aan de onderkant met een uitvoerstatement (hier eenvoudig alleen "`xm`").
Helaas gaat het niet goed; Maple geeft een foutmelding als we de procedure aanroepen met de actuele parameters `f, -1, 1, 0.01`. Om te achterhalen wáár het in de afhandeling van de statements van de body precies vastgelopen is, kunnen we met het commando

`tracelast`

`tracelast` nadere informatie opvragen. Maple antwoordt dan dat de fout is geconstateerd toen het het vierde statement van `nulp1` wilde

`showstat`

uitvoeren: `xr := xm`. Met `showstat` hebben we een nette afdruk gekregen van onze procedure, met de statementnummers er bij.
Blijkbaar is het verboden om een *invoerparameter* (in dit geval `xr`) een (nieuwe) waarde toe te kennen (in dit geval 0, de waarde van `xm` op dat moment). Zie echter ook §28.7 (*uitvoerparameters*). ◇

We maken een verbeterde versie van `nulp1` door van de formele parameters direct lokale kopieën te maken.

Voorbeeldsessie

```
>  nulp1 := proc(f,links,rechts,eps)
     local xl, xr, xm;
     xl := links; xr := rechts;
     xm := 0.5*(xr+xl):
     while abs( f(xm) ) > eps do
     if f(xm)*f(xl) < 0
       then xr := xm
       else xl := xm
     end if:
     xm := 0.5*(xr+xl)
     end do;
     xm
   end proc:
```

```
> f := x -> 9*x^3 + 5*x^2 + 8*x + 6:
> antw := nulp1(f,-1,1,0.01);
```

$$antw := -0.68359375$$

```
> f(antw);
```

$$-0.007240713$$

Toelichting

We hebben de formele parameters maar andere namen gegeven, zodat we `xl` en `xr` als lokale variabelen kunnen declareren. Velen maken er een gewoonte van om *altijd* lokale kopieën van de formele parameters te maken.
Nu gaat het wel goed; `antw` wordt een x-waarde waarvoor de functiewaarde minder dan 0.01 van 0 verschilt. ◇

> **!** Maak scherp onderscheid tussen *formele* (of invoer-) *parameters* en (lokale en globale) *variabelen*. Parameters kunnen normaliter door het aanroepen van een procedure géén andere waarde krijgen, variabelen uiteraard wél.

28.4 Lokale en globale variabelen

Naast lokale variabelen zijn er ook *globale variabelen*. Globale variabelen zijn variabelen waarvan de waarde in de gehele sessie (zie §1.7) bekend is (dus ook binnen procedures). Alle variabelen die *buiten* enige procedure een waarde hebben gekregen, dus ooit links van de `:=`-operator zijn opgetreden, zijn globaal. Dat is handig, want dat betekent dat we een constante zoals `Pi` ook binnen een procedure rustig kunnen gebruiken. Hetzelfde geldt voor namen van procedures zoals `simplify`, `solve` enzovoort. Wanneer we *binnen een procedure* de waarde van een globale variabele willen wijzigen, moeten we deze declareren in een `global`-statement, analoog aan de declaratie van lokale variabelen. Een eventueel `global`-statement komt ná een eventueel `local`-statement.

`global`

We demonstreren de verschillende mogelijkheden aan de hand van een paar eenvoudige voorbeelden.

Voorbeeldsessie

Voor `p` is `x` een globale variabele:

```
> p := proc(y) x+y end proc:
> x := 10:  p(a);
```

$$10 + a$$

```
> x := 11:  p(a);
```

$$11 + a$$

Voor `q` is `x` een (impliciet) lokale variabele:

```
> q := proc(y) x := 10; x+y end proc:

Warning, `x` is implicitly declared local to procedure `q`
> x := 234:  q(a);
```

$$10 + a$$

```
> x;
```

$$234$$

Voor `r` is `x` een (expliciet) globale variabele:

```
> r := proc(y) global x;
  x := 10; x+y
  end proc:
> x := 893:  r(a);
```

$$10 + a$$

```
> x;
```

$$10$$

Lokale en globale variabele met dezelfde naam

```
> s := proc(y)
  local x;
  x := 2*y;
  :-x + x  # globale x + lokale x
  end proc:
> x := 10:  s(a);
```

$$10 + 2\,a$$

```
> x := 27:  s(a);
```

$$27 + 2\,a$$

Toelichting

In de eerste procedure `p` wordt aan `x` geen waarde toegekend. Een waarde die aan `x` eventueel *buiten* de procedure is toegekend, is ook *binnen* de procedure bekend.
In de tweede procedure `q` krijgt `x` een waarde. Hierop reageert Maple direct door er een *lokale* variabele van te maken en te waarschuwen dat het dat gedaan heeft. Deze waarschuwing hadden we dus kunnen voorkomen door `local x;` in de procedure op te nemen. De waarde van deze lokale `x` heeft niets te maken met de `x` die *buiten* de procedure `q` in gebruik is.
In de derde procedure `r` de `x` als globale variabele gedeclareerd. Dat betekent dat we binnen én buiten de procedure kunnen beschikken

over één gemeenschappelijke variabele met de naam `x`. Dat deze waarde door de procedureaanroep `r(a)` *verandert*, wordt wel een *neveneffect* (Engels: *side-effect*) van de procedure(aanroep) genoemd. De vierde procedure tenslotte is een beetje een rariteit. In de procedure `s` gebruiken we tegelijkertijd een *lokale* én een *globale* `x`. Uiteraard kunnen we `x` niet als `local` én als `global` declareren. De uitweg is, dat de *globale* `x` *binnen* de procedure de naam `:-x` krijgt, en we zo binnen de procedure over de twee versies van de variabele `x` kunnen beschikken. ◇

`:-`

Het gebruik van globale variabelen binnen procedures moet met de nodige voorzichtigheid gebeuren. In het algemeen is het het beste om procedures uitsluitend via de parameters en het uitvoerstatement 'met de buitenwereld te laten communiceren'.[64]

In het voorbeeld op blz. 433 zijn in `nulp1` de variabelen `xl`, `xr` en `xm` gebruikt als *lokale* variabelen. Dat betekent dat de waarde die ze *in* de procedure hebben gekregen *buiten* de procedure weer is vergeten. Dat geldt óók voor `xm`. De waarde die deze variabele uiteindelijk gekregen heeft, wordt door het uitvoerstatement toegekend aan de procedure*aanroep*, dus in het voorbeeld aan de variabele `antw`. Andersom, als `xm` buiten de procedure een waarde had gekregen, dan verandert deze niet door de aanroep van `nulp1(f,-1,1,0.01)`. Door het statement `local xm` wordt een geheel nieuw exemplaar van een variabele met de naam `xm` aangemaakt.

Wanneer een variabele *in de body* van een procedure niet als `global` of `local` gedeclareerd is, wordt aangenomen dat zij

(1) lokaal is als zij links van een `:=` teken of als parameter in een for-loop voorkomt; Maple geeft dan een waarschuwing;
(2) globaal is als dat niet zo is.

Om verwarring te voorkomen is het verstandig elke variabele die in een procedure voorkomt te declareren.

Lokale variabelen zijn dus uitsluitend bedoeld voor gebruik binnen procedures. Indien een lokale variabele 'naar buiten lekt' kunnen er merkwaardige dingen gebeuren, zoals het volgende voorbeeld laat zien.

[64] Een uitzondering op deze vuistregel wordt gevormd door namen van procedures. Als binnen een procedure bijvoorbeeld het statement `q := simplify(p)` voorkomt, dan hoeft de naam `simplify` niet als `global` te worden gedeclareerd. Dit geldt ook voor zelfgemaakte procedures. Zie §30.2.

Voorbeeldsessie

```
> x_nieuw := proc() local x:  x end proc:

> y := x_nieuw(); z := x_nieuw();
```

$$y := x$$

$$z := x$$

```
> simplify(x+y+z);
```

$$x + x + x$$

Toelichting

Het *aantal* formele parameters (dat is dus het aantal argumenten waarmee een procedure wordt aangeroepen) is geheel vrij. In `x_nieuw` zijn dat er dus nul. Deze procedure doet niets anders dan het 'naar buiten brengen' van de lokale variabele met de naam `x`. Uit dit voorbeeld blijkt dat door elke aanroep van `x_nieuw` een *nieuwe* variabele met de naam `x` wordt gecreëerd die verschilt van de andere. Dat daardoor `x + y + z` niet vereenvoudigt tot $3x$ is daarvan een gevolg dat in de praktijk altijd ongewenst is. ◇

> **!** Zorg er dus voor dat het laatste statement in de body van de procedure (het uitvoerstatement) altijd evalueert tot een waarde die uitsluitend van de argumenten (de invoerparameters) afhangt – en eventueel van globale variabelen.

Opmerking Op het eerste gezicht lijkt het alsof in het uitvoerstatement van de procedure `pol` in de voorbeeldsessie op blz. 430 de lokale variabele `x` voorkomt. Echter, door het `unapply`-commando is er voor gezorgd dat deze `x` een *dummy* is geworden.

28.5 Recursie

Een statement in de body van een procedure mag ook een aanroep van *diezelfde* procedure bevatten. In zo'n geval spreken we van *recursie*. Het volgende voorbeeld is een zeer eenvoudige toepassing van een recursieve procedure-aanroep.

Voorbeeldsessie

```
> iterate := proc(f,n,a)
    if n=0 then a
    else iterate(f, n-1, f(a))
    end if
  end proc:
> iterate(g,4,b);
```

$$g(g(g(g(b))))$$

```
> iterate( x->exp(x), 3, c );
```

$$e^{e^{e^c}}$$

Toelichting

De procedure `iterate` past de functie f n keer toe op het argument a (en doet dus precies hetzelfde als `(f@@n)(a)`). Kenmerkend voor een recursieve procedure is het *stopcriterium*. In dit geval is dat `if n=0 then a`: de aanroep `iterate(f,0,a)` geeft `a` zelf terug. In alle andere gevallen wordt f $n-1$ keer op $f(a)$ toegepast, daarna $n-2$ keer op $f(f(a))$ enzovoort. Merk op dat er een probleem is als n geen natuurlijk getal is. Het stopcriterium $n = 0$ wordt dan nooit bereikt en de procedure zou eindeloos blijven doorgaan zichzelf aan te roepen. Maple zal het niet zover laten komen en reageert in zo'n geval met de foutmelding "too many levels of recursion". ◇

Ook de iteratie-algoritme voor nulpuntsbepaling door interval-halvering kan op een natuurlijke manier met een recursie geprogrammeerd worden.

Voorbeeldsessie

```
> nulp2 := proc(f,xl,xr,eps)
    local xm;
    xm := 0.5*(xr+xl):
    if abs( f(xm) ) < eps then return xm
    elif f(xm)*f(xl) < 0 then nulp2(f,xl,xm,eps)
    else nulp2(f,xm,xr,eps)
    end if
  end proc:
> f := x -> 9*x^3 + 5*x^2 + 8*x + 6:
> nulp2(f,-1,1,0.01);
```

$$-0.68359375$$

Toelichting

Na de initialisatie $x_m = \frac{1}{2}(x_l + x_r)$ wordt onderzocht of aan het

stopcriterium $|f(x_m)| < \epsilon$ voldaan wordt. Zo ja, dan zijn we klaar;

return `return xm` is een *uitvoerstatement*. Dat wil zeggen dat de procedure met de waarde van `xm` wordt beëindigd.[65] Als *niet* aan het stopcriterium wordt voldaan, wordt de procedure `nulp2` opnieuw aangeroepen, hetzij met het interval $[x_l, x_m]$, hetzij met $[x_m, x_r]$. ◇

28.6 Invoerparameters: de variabelen `args` en `nargs`

Als de heading van een procedure van de vorm

```
naam := proc(a,b,c)
```

is, dan betekent dat dat de procedure moet worden aangeroepen met (in dit geval) *minstens* drie actuele parameters. Dus `naam(1,2,3,4)` gaat goed, maar `naam(1,2)` geeft een foutmelding. Zo'n aanroep heeft natuurlijk alleen zin als we in de body van de procedure ook kunnen beschikken over de vierde, vijfde enzovoort (formele) parameter, óók als we er in de heading maar drie gebruikt hebben bij de definitie van de procedure.

args Hiervoor is *binnen* een procedure de lokale variabele `args` bekend; dat is de expressierij van actuele parameters. Daarnaast is er de lokale

nargs variabele `nargs`, het *aantal* actuele variabelen waarmee de procedure is aangeroepen. We demonstreren het gebruik ervan met een voorbeeld.

Voorbeeldopgave

Gegeven een rij punten $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Gevraagd de $(n-1)$-de graadspolynoom p die voldoet aan

$$p(x_1) = y_1, \ \ldots, \ p(x_n) = y_n \, .$$

De eerste keus die we moeten maken is de *vorm* waarin de rij punten aan de procedure zal worden aangeboden. Daarvoor zijn diverse mogelijkheden, bijvoorbeeld `x1,y1,x2,y2,...` of

```
x1,x2,...xn, y1,y2,...yn
```

Voor de overzichtelijkheid kiezen we voor

```
[x1,y1], [x2,y2], ..., [xn,yn]
```

[65] Strikt genomen zou hier (net als in de procedure `iterate` van het vorige voorbeeld) het woord `return` weggelaten kunnen worden omdat het statement `xm` altijd het *laatste* uitgevoerde statement is. Om te benadrukken dat de if-claus ervóór het stopcriterium is, laten we het graag volgen door een `return`-commando.

dus voor een expressierij (van onbekende lengte), bestaande uit lijsten die de x- en y-coördinaat bevatten.
We kijken eerst hoe we het probleem stap voor stap kunnen oplossen.

Voorbeeldsessie

De invoer komt in de volgende vorm:

```
> S := [1,2],[2,4],[3,5]; #enzovoort
```

$$S := [1, 2], [2, 4], [3, 5]$$

We maken er lijsten van x- en y-waarden van

```
> X := map( x->x[1], [S] );
```

$$X := [1, 2, 3]$$

```
> Y := map( x->x[2], [S] );
```

$$Y := [2, 4, 5]$$

We maken een polynoom $a_0 + a_1\,x + a_2\,x^2 + ...$ (met vooralsnog onbekende coëfficiënten) als functie van x:

```
> n := nops([S]):
> p := unapply( add( a[i]*x^i, i=0..n-1 ), x );
```

$$p := x \rightarrow a_0 + a_1\,x + a_2\,x^2$$

en het stelsel vergelijkingen wordt dan

```
> stelsel := {seq( p(X[i])=Y[i], i=1..n )};
```

$$\mathit{stelsel} := \{a_0 + a_1 + a_2 = 2,\ a_0 + 2\,a_1 + 4\,a_2 = 4,\ a_0 + 3\,a_1 + 9\,a_2 = 5\}$$

Er zijn precies even veel vergelijkingen als onbekenden.

Dat betekent dat we niet de verzameling van op te lossen onbekenden hoeven op te geven.

```
> s := solve(stelsel);
```

$$s := \left\{a_2 = -\frac{1}{2},\ a_1 = \frac{7}{2},\ a_0 = -1\right\}$$

Dit staat in de juiste vorm om te substitueren

```
> unapply( subs( s, p(x) ), x );
```

$$x \rightarrow -1 + \frac{7}{2}\,x - \frac{1}{2}\,x^2$$

Toelichting

We hebben een voorbeeldrij `S` van parameters gemaakt. In de uitwerking maken we geen gebruik van het feit dat deze rij drie elementen bevat. Dat betekent dat we nu niet, zoals in §28.2, een polynoom met onbekende coëfficiënten a, b, c kunnen maken. Daarom maken we voor de onbekende coëfficiënten in feite een *tabel* met de naam `a`, waardoor we de beschikking hebben over de variabelen `a[0]` tot en met (in dit geval) `a[2]`. ◇

De gevraagde procedure kunnen we nu maken op basis van de bovenstaande voorbeeldsessie. Wat daar `S` was, is in de procedure `args`, en in plaats van `nops([S])` kunnen we `nargs` gebruiken.

Voorbeeldsessie

```
> polynoom := proc()
    local X,Y,n,a,p,x,i,stelsel, s;
    X := map( c->c[1], [args] );
    Y := map( c->c[2], [args] );
    n := nargs-1;
    p := unapply( add( a[i]*x^i, i=0..n ), x );
    stelsel := {seq( p(X[i])=Y[i], i=1..nargs )};
    s := solve( stelsel );
    unapply( subs( s, p(x) ), x );
  end proc:
```

```
> q := polynoom( [1,2],[2,4],[3,3],[4,-2],[0,3] );
```

$$q := x \to 3 - \frac{23}{4}\,x + \frac{163}{24}\,x^2 - \frac{9}{4}\,x^3 + \frac{5}{24}\,x^4$$

```
> r := polynoom( [0,4],[2,-1] );
```

$$r := x \to 4 - \frac{5}{2}\,x$$

Toelichting

Door `polynoom := proc()` moet de procedure worden aangeroepen met minimaal nul, dus een willekeurig aantal parameters. ◇

Zie §29.2 voor een uitvoeriger behandeling van het aantal formele en actuele parameters.

28.7 Uitvoerparameters

We bekijken nog eens de procedure voor het benaderen van een nulpunt met behulp van het interval-halverings-algoritme uit §28.3. Als we na het aanroepen van `nulp1`, zie de voorbeeldsessie op blz. 433, zouden willen weten in hoeveel stappen de benadering is berekend, dan kunnen we dat op verschillende manieren te weten komen.

`print` De eerste manier is het invoegen van een `print`-opdracht in de procedure. De opdracht `print(xm)`, direct na het statement waarin `xm` (opnieuw) wordt berekend, heeft tot gevolg dat bij elke herhaling van dat statement de waarde van `xm` wordt 'afgedrukt', dat wil zeggen: op het scherm getoond. We krijgen dan een uitvoer zoals in §27.4, een heel rijtje van `xm`-waarden. Als je deze telt dan weet je

hoe vaak de loop is herhaald. Mooier wordt het nog met de opdracht `print('xm=',xm)`, vooral nuttig als ook de waarden van andere variabelen moeten worden afgedrukt (zie verder §29.3).
Men zou ook een 'teller' kunnen bijhouden, en aan het eind van de procedure de opdracht `print(teller)` kunnen geven (direct vóór het uitvoerstatement).
Dit is vooral een nuttig hulpmiddel bij het testen van een procedure. Zolang de procedure nog niet precies doet wat de bedoeling is, kan men op strategische plaatsen tijdelijk `print`-opdrachten tussenvoegen om de werking van de procedure op de voet te kunnen volgen. Het nadeel van deze methode is dat het resultaat van een `print`-opdracht niet aan een variabele kan worden toegewezen.[66]

De tweede manier, die dit nadeel niet heeft, is het opnemen van een formele parameter in de heading waaraan een waarde kan worden toegekend. We demonstreren dat met een variant van de procedure van §28.3.

Voorbeeldsessie

```
> nulp3 := proc(f,links,rechts,eps,n::evaln)
    local xl, xr, xm, teller;
    xl := links; xr := rechts;
    xm := 0.5*(xr+xl):
    teller := 0;
    while abs( f(xm) ) > eps do
    if f(xm)*f(xl) < 0
      then xr := xm
      else xl := xm
    end if:
    xm := 0.5*(xr+xl);
    teller := teller+1
    end do;
    n := teller;
    xm
  end proc:

> f := x -> 9*x^3 + 5*x^2 + 8*x + 6:

> aantal := 37:

> antw := nulp3(f,-1,1,0.001,aantal);
```

$$antw := -0.6831054690$$

```
> aantal;
```

11

Toelichting

In §28.3 (voorbeeldsessie op blz. 432) hebben we gezien dat het niet

[66] De nul-operator `%` heeft na een `print`-opdracht de waarde `NULL`.

altijd mogelijk is om een waarde toe te kennen aan een van de argumenten (actuele parameters) waarmee de procedure wordt aangeroepen. Dat is natuurlijk zo als het betreffende argument een *waarde* heeft; de toekenning in de body zou dan zoiets als "`2 := 4`" worden. Indien de actuele parameter géén waarde heeft, dat wil zeggen tot een *naam* evalueert, kan het wél en kan hij als *uitvoerparameter* dienen.

`evaln` Door in de heading de formele parameter als `n::evaln` op te nemen wordt ervoor gezorgd dat de (vijfde) actuele parameter waarmee de procedure wordt aangeroepen altijd éérst van haar eventuele waarde wordt ontdaan ('tot een naam wordt geëvalueerd') vóórdat de statements van de body worden uitgevoerd.
Bij de aanroep van `nulp3` in het voorbeeld is het daardoor alsof deze wordt voorafgegaan door het statement `aantal := 'aantal';`. ◇

We zeggen dat een parameter die tot zijn eigen naam evalueert wordt doorgegeven via het *call by name*-principe, en een parameter die tot een waarde evalueert volgens het *call by value*-principe. Zie verder §30.3.

28.8  Procedures of expressies als invoer en uitvoer

In veel gevallen zal een procedure 'iets doen' met een functie (in de wiskundige betekenis) als invoer en/of een functie als uitvoer hebben. Bij de meeste tot nu toe behandelde voorbeelden is dat het geval. Zie bijvoorbeeld de nulpuntsbepaling door interval-halvering van een functie f, waarbij we in §28.3 en §28.5 hebben gekozen om een *procedure* `f := x -> ...` als input te gebruiken, en de constructie van een interpolatiepolynoom, waarbij we in §28.2 en §28.6 hebben gekozen om een procedure als output te gebruiken.
Dat hoeft niet, en het is in de praktijk soms handiger om voor *expressies* als invoer of uitvoer te kiezen. Vergelijk de bestaande Maple-procedure `D` die een *procedure* als invoer en als uitvoer heeft, en de procedure `diff` die een *expressie* als invoer en uitvoer heeft.
We geven een voorbeeld om de verschillende mogelijkheden te demonstreren.

Voorbeeldopgave

Maak een procedure die van een functie f de primitieve F, met $F(a) = b$ bepaalt. Dus de input wordt gevormd door f, a en b;

de uitvoer moet de primitieve F zijn die aan de bovenstaande eis voldoet.

Voorbeeldsessie

Het eerste argument is een expressie:

```
> prim := proc(f,a,Fa)
    local F;
    F := int(f,x);
    simplify(Fa - subs( x=a, F ) + F)
  end proc:
> prim( ln(x), 1, 4 );
```

$$5 + x\ln(x) - x$$

```
> prim( ln(y), 1, 4 );
```

$$4 - \ln(y) + \ln(y)\,x$$

```
> x := 1:  prim( ln(x), 1, 4 );

Error, (in int) integration range or variable must be
    specified in the second argument, got 1
```

Verbeterde versie:

```
> restart;
> prim := proc(f,x,a,Fa)
    local F;
    F := int(f,x);
    simplify(Fa - subs( x=a, F ) + F)
  end proc:
> prim( ln(y), y, 1, 4 );
```

$$5 + y\ln(y) - y$$

Het eerste argument is een functie:

```
> restart;
> prim := proc(f,a,Fa)
    local x, Fx;
    Fx := Fa + int(f(t), t=a..x);
    unapply(Fx, x)
  end proc:
> F := prim( x->ln(x), 1, 4);
```

$$F := x \rightarrow 5 + x\ln(x) - x$$

Toelichting

In de eerste versie is er 'stilzwijgend van uitgegaan' dat de naam van de variabele in f wel x zal zijn; in feite is `x` als een *globale* variabele gebruikt. Dat levert dus een probleem op als de expressie een andere variabele bevat. Kortom, bij deze opzet – dus met de te primitiveren functie als *expressie* ingevoerd – is het nodig om expliciet in de parameterlijst mee te geven wat de variabele is waar naar moet worden geïntegreerd. Dit is in de tweede (verbeterde) versie gebeurd.

Ook dan moeten we nog oppassen dat deze variabele geen waarde mag hebben bij aanroep van `prim`.
In de laatste versie is `f` een *procedure*. In dat geval moet de variabele `x` als `local` worden gedeclareerd. Het is nu consequent om ervoor te zorgen dat ook de *uitvoer* weer een procedure is. We gebruiken daarvoor `unapply`. In deze versie van de procedure `prim` wordt de *lokale* `x` in het uitvoerstatement gebruikt. Maar `unapply(Fx, x)` zorgt er juist voor dat de resulterende functiedefinitie *on*afhankelijk is van de naam van de parameter (vandaar ook de naam *unapply*). Dit is ook de reden waarom je nóóit zoiets als `x -> Fx` als uitvoerstatement moet gebruiken. Omdat `Fx` in dit statement *niet* wordt geëvalueerd (zie §7.3), kan dit betekenen dat er lokale variabelen in de uitvoer terechtkomen (zie §28.4). ◇

Bij het werken met lokale variabelen treden in sommige gevallen nog andere complicaties op. De behandeling van deze complicaties stellen we uit tot §30.5.

28.9  Plaatjes en procedures

Binnen procedures kunnen ook plotopdrachten voorkomen. We geven twee manieren hoe een procedure een plaatje kan tekenen én 'tegelijkertijd' een nieuwe procedure aflevert.

Voorbeeldopgave

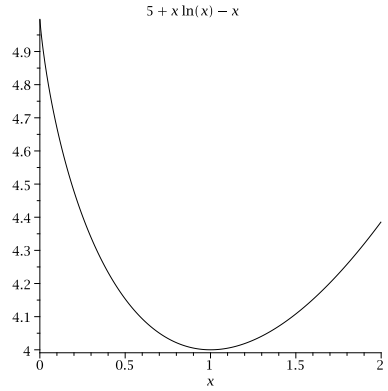
Maak een procedure die van een gegeven functie f de primitieve F berekent waarvoor geldt dat $F(a) = b$. De procedure moet er bovendien voor zorgen dat een plaatje van de grafiek van F wordt getekend.

Methode 1: Met een printopdracht. Het `print`-commando kan ook worden gebruikt om een plaatje te tekenen.

Voorbeeldsessie

```
> primplot := proc(f,a,Fa)
    local x, Fx;
    Fx := Fa + int(f(t), t=a..x);
    print( plot(Fx, x=a-1..a+1, title=typeset( Fx )) );
    unapply(Fx, x);
  end proc:

> F := primplot( t->ln(t), 1, 4);
```



$$F := x \to 5 + x\ln(x) - x$$

```
> F(t);
```

$$5 + t\ln(t) - t$$

Toelichting

Met `typeset(Fx)` (zie §9.3) kunnen we de *formule* Fx in de titel van het plaatje opnemen.
In dit geval krijgt de aanroep `primplot( t->ln(t), 1, 4)` de waarde $x \to 5 + x\ln(x) - x$. Het vertoonde plaatje is een *neveneffect* van deze aanroep. Dat betekent dat dit plaatje niet meer kan worden hergebruikt om het bijvoorbeeld met een `display`-commando met andere plaatjes te combineren. ◇

Methode 2: Plot als uitvoer; *F* als uitvoervariabele. Nu maken we van het `plot`-commando het uitvoerstatement.

Voorbeeldsessie

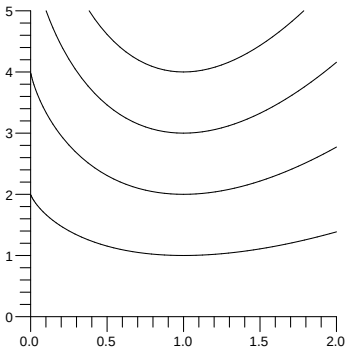
```
> primplot2 := proc(f,a,Fa,F::evaln)
    local x, Fx;
    Fx := Fa + int(f(t), t=a..x);
    F := unapply(Fx, x);
    plot(F, a-1..a+1);
  end proc:

> S := seq( primplot2( t->i*ln(t), 1, i, G[i]), i=1..4 ):

> G[3](t);
```

$$6 + 3\,t\ln(t) - 3\,t$$

```
> plots:-display({S}, view=[0..2,0..5]);
```



Toelichting

Het resultaat van een aanroep van `primplot2` is nu zélf een plaatje. Dat betekent dat `primplot2` als het ware zelf een plotopdracht is geworden, waarvan het resultaat aan een variabele kan worden toegekend en via een `display`-opdracht met andere plaatjes kan worden gecombineerd.
De gevraagde primitieve is in een uitvoervariabele geplaatst. ◇

28.10 Praktische werkwijze

In deze laatste paragraaf geven we een aantal aanbevelingen voor een werkwijze voor het opzetten van een procedure.

Stap 1. Kies vóórdat u iets gaat intypen eerst wát de procedure precies zal moeten gaan opleveren. Welke 'waarde' zal `antw` hebben na het commando `antw := procedurenaam(argumenten)`? Vervolgens moet u vaststellen wat precies de input zal zijn, dat wil zeggen: op welke manier zullen de benodigde gegevens aan de procedure worden meegedeeld? Dat betekent in feite dat de heading het eerst bedacht moet worden.

Stap 2. Kies nu representatieve waarden voor de argumenten en geef deze een naam, liefst de naam die de formele parameters zullen krijgen. Bereken nu stap voor stap datgene wat uiteindelijk door de procedure berekend zal moeten worden. Zie bijvoorbeeld de procedure van §28.6.

Stap 3. Maak nu de eerste versie van de procedure klaar. Denk eraan dat alle statements in één 'execution group' komen. Zet een puntkomma achter `end proc` en druk een keer op [Enter]. Maple meldt dan de veronderstelde typefouten zoals vergeten puntkomma's, vergeten `end do`'s en dergelijke. Kijk waar de cursor staat na zo'n foutmelding, want dat is de plaats waarop Maple heeft geconstateerd dat er iets niet in orde is. Vanaf dat punt kunt u terug lezen.
Als Maple geen typefouten meer kan vinden, komt er output: de tekst van de procedure in een min of meer nette opmaak. Kijk even of dat de bedoeling is en verander de puntkomma na `end proc` in een dubbele punt.

Stap 4. Maak nu een procedureaanroep met zodanige waarden van de parameters dat u in één oogopslag kunt zien of het resultaat goed is. Als Maple nu een foutmelding geeft, en u ziet niet direct wat er aan de hand is, geef dan direct daarna het commando `tracelast;` voor meer informatie. Als u dan nog niet ziet wat de fout is, voeg dan tijdelijk een of meer `print`-commando's in uw procedure toe. U kunt dan zien of bepaalde variabelen de waarde hebben gekregen die ze moeten krijgen. Let vooral ook op de *vorm* van die waarde: is het een naam, een getal, een expressierij, een lijst, ...?
Als er nog fouten overblijven, moet u er misschien aan gaan denken om de procedure op een iets andere manier op te zetten. Wellicht ben u op een subtiliteit gestuit die niet in deze module, maar in Module 30 wordt besproken.

Stap 5. Test nu de procedure met andere inputs. Ga na wat nog wel goed gaat en wat niet meer, en of u bepaalde uitzonderlijke gevallen waar het niet meer werkt in uw procedure wilt opvangen of niet.

Vele Maple-commando's zijn gewone Maple-procedures, waarvan de body zichtbaar gemaakt kan worden (en dus zelfs desgewenst veranderd). Na het commando

`verboseproc`

```
interface(verboseproc=2);
```

kunt u bijvoorbeeld de werking van de procedure `surd` bestuderen met `print(surd);`. Nog gemakkelijker is wellicht: `showstat(surd)`. U hoeft dan `verboseproc` niet te veranderen, maar u krijgt de procedure met de statement-nummers er bij.

Opgave 28.1

Maak een procedure `sublijst`, zodat de aanroep `sublijst(L,a)` de plaatsnummers en de waarden oplevert van de lijst `L` met absolute waarde groter dan `a`.
De procedure moet als volgt werken (voorbeeld):

```
> L := [-10, 0, 4.5, 11, -17.25, -1, 25/2, 1]:
> sublijst(L,10);
```

$$[[4, 11], [5, -17.25], [7, \frac{25}{2}]]$$

Opgave 28.2

Gegeven een differentieerbare functie f op het interval $[x_1, x_2]$, met $-1 < x_1 < x_2 < 1$. Maak een differentieerbare uitbreiding g van f tot het interval $[-1, 1]$ door middel van tweedegraadsfuncties op $[-1, x_1]$ en $[x_2, 1]$, zó dat $g(-1) = 0$ en $g(1) = 0$.
Opdracht: Maak een procedure met als invoervariabelen: een functie (Maple-procedure) f en de waarden van x_1 en x_2, en als uitvoer: de gevraagde functie g. Gebruik `piecewise` (zie §3.4).
Pas uw procedure bijvoorbeeld toe op de functie $x \mapsto 1 + \sin(\pi x^2)$ met $x_1 = -\frac{2}{5}$ en $x_2 = \frac{4}{5}$.
Teken de grafieken van g en van g'.

Opgave 28.3

Maak een procedure `primitieve` (vergelijk §28.8) die als volgt moet werken:

```
> C := 12:  F := primitieve(x -> 2*x);
```

$$F := x \to x^2 + C$$

```
> C;
```

$$C$$

```
> C := 13:  F(x);
```

$$x^2 + 13$$

Aanwijzing: Declareer `C` als globale variabele.

Opgave 28.4

Maak een procedure met de naam `koorden`, zodat na het commando

```
koorden(f, a, b, n);
```

van de functie $f(x)$ de koordenfiguur van de grafiek van f wordt getekend tussen $x = a$ en $x = b$, dat is de gebroken lijn die de punten $(t_k, f(t_k)),\ k = 0, \ldots, n$ met elkaar verbindt. Verdeel het interval $[a, b]$ in gelijke delen, met uiteraard $t_0 = a$ en $t_n = b$.

Opgave 28.5

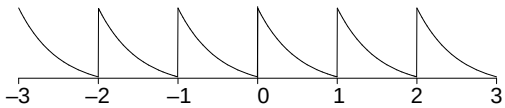
Recursie wordt vaak gebruikt om recurrente betrekkingen mee te programmeren. Bijvoorbeeld

```
> fib := proc(n::integer)
    if n=0 or n=1 then 1
    else fib(n-1) + fib(n-2)
    end if
  end proc;
```

berekent het n^e Fibonacci-getal.
Schrijf zelf een recursieve procedure `fac` die voor een integer n, de waarde $n!$ aflevert.
Zie ook opgave 30.4 waar wordt behandeld hoe dit soort recursieve procedures veel efficiënter gemaakt kan worden.

Opgave 28.6

Maak voor de periodieke functie $f : \mathbb{R} \to \mathbb{R}$ een procedure `f`, zodat $f(x) = e^{-2x}$ als $0 < x \leq 1$; voor de overige waarden van $x \in \mathbb{R}$ herhaalt dit patroon zich.
De aanroep `f(x)` moet dus voor alle mogelijke reële waarden van `x` een antwoord geven en de grafiek van f op het interval $[-3, 3]$ (het resultaat van `plot( f, -3..3 );`) moet er dus uitzien als in figuur 69.



FIGUUR 69. Grafiek van f (opgave 28.6)

Aanwijzing: Maak een recursieve procedure. Gaat het ook met `piecewise`?

Opgave 28.7

Picard-iteratie. Gegeven het beginwaardeprobleem

$$\frac{dy}{dx} = f(x,y), \quad y(x_0) = y_0 \tag{28.1}$$

met $f : \mathbb{R}^2 \to \mathbb{R}$ een continue functie.
Er geldt nu dat de rij functies (y_n), gedefinieerd door

$$\begin{cases} y_0(x) = y_0, & \text{voor alle } x \\ y_n(x) = y_0 + \int_{x_0}^{x} f(s,\, y_{n-1}(s))\, ds\,, & n \geq 1 \end{cases} \tag{28.2}$$

uniform convergeert naar de oplossing $y(x)$ van het beginwaardeprobleem (28.1).

Maak een procedure `Picard`, zodat de aanroep `Picard(x0,y0,f,n)` de n^{de} Picard-iteratie, dat wil zeggen: de functie y_n uit (28.2) oplevert.
Test uw procedure aan de hand van het beginwaardeprobleem $\frac{dy}{dx} = xy$, $y(0) = 1$.

Opgave 28.8

Maak een procedure `verwissel(a, i,j)` die het i-de en j-de element van `a` verwisselt als

(1) `a` een Array is;
(2) `a` een lijst is.

Eis: het statement `b := verwissel(a, 2,4)` moet een `b` van hetzelfde type als `a` afleveren, met uiteraard het tweede en vierde element verwisseld.
Lukt het om de inhoud van `a` zélf te veranderen als `a` een lijst, respectievelijk een Array is? Zie verder de opmerking in §30.3 (blz. 476). Vindt u dat wenselijk?
Aanwijzing: Maak zo nodig een lokale kopie van `a`.