Module 30 Procedures (vervolg)

Onderwerp	Procedures: Bereik van lokale variabelen, call by evaluated name, level-1-evaluatie van lokale variabelen, remember-tables.
Voorkennis	Module 3, 7, 8, 25, 26, 27, 28, 29
Expressies	`procname, option, piecewise, remember, trace, forget, time`

30.1  Inleiding

In deze module komt een aantal zaken aan de orde waar u bij het maken van procedures waarschijnlijk niet direct mee te maken zult krijgen. Mochten er echter problemen optreden waar u met de kennis van Module 28 en 29 niet uitkomt, dan bestaat er een gerede kans dat u op een subtiliteit gestuit bent die in deze module wordt behandeld. Verder behandelen we de remember-tables van procedures (in §30.6), die onder andere gebruikt worden om niet verschillende keren hetzelfde te hoeven uitrekenen.

30.2  Nesting en Scope

In §29.3 is in de procedure `Polynoom` gebruikgemaakt van de twee hulp-procedures `verwijderfoutepunten` en `polynoom`. In de voorbeeldsessie op blz. 458 bestaan er dus eigenlijk drie procedures naast elkaar, waarvan we er maar één (namelijk: `Polynoom`) zullen aanroepen in de rest van de sessie. Er is dus één 'hoofdprocedure' met twee 'hulp-procedures'.
Dat kan natuurlijk geen kwaad. Het is echter ook mogelijk om de hulp-procedures als lokale variabelen in de hoofdprocedure op te nemen. Ook procedures zijn namelijk gewoon Maple-objecten met een naam. Ze kunnen dus als lokale variabelen worden gedeclareerd en in de body worden gedefinieerd. Voor het polynoom-voorbeeld is dat gedaan in de volgende sessie. Hierdoor zijn de procedures `verwijderfoutepunten` en `polynoom` alléén binnen de procedure `Polynoom` bekend. De naam `polynoom` kan dan (buiten `Polynoom`) voor iets anders worden gebruikt.

Voorbeeldsessie

```
> Polynoom := proc()
    local polynoom, verwijderfoutepunten, argumenten, antwoord;
  #-------------------------------------------------------------
    polynoom := proc()
      local A,X,Y,n,a,p,x,i,stelsel, s;
      A := [op({args})];
      X := map( c->c[1], A ); Y := map( c->c[2], A );
      n := nops(A)-1; a := array(0..n);
      p := unapply( add( a[i]*x^i, i=0..n ), x );
      stelsel := {seq( p(X[i])=Y[i], i=1..nops(A) )};
      s := solve( stelsel );
      if s=NULL then
         error "Een waarde van x met verschillende y-waarden" end if;
      unapply( subs( s, p(x) ), x );
    end proc:    #(einde van de lokale procedure polynoom)
  #-------------------------------------------------------------
    verwijderfoutepunten := proc()
      local i, A, M:
      A := {args}; i := 1;
      while i<=nops(A) do
        M := select( c->c[1]=A[i][1], A );
        if nops(M)>1 then A := A minus M end if;
        i := i+1
      end do;
      op(A)
    end proc:  #(einde van de lokale procedure verwijderfoutepunten)
  #-------------------------------------------------------------
    argumenten := args;
    try
      antwoord := polynoom(argumenten)
    catch "Een waarde van x met verschillende y-waarden":
      WARNING("De volgende punten zijn verwijderd:  %1",
        {argumenten} minus {verwijderfoutepunten(argumenten)});
      antwoord := polynoom(verwijderfoutepunten(argumenten))
    end try;
    eval(antwoord)
  end proc:
> Polynoom( [1,2],[2,4],[3,3],[4,-2],[1,3] );

Warning, De volgende punten zijn verwijderd: {[1, 2], [1, 3]}
```

$$x \to -6 + 9\,x - 2\,x^2$$

```
> Polynoom( [1,2],[2,4],[3,3],[4,-2] );
```

$$x \to -2 + \frac{14}{3}\,x - \frac{1}{2}\,x^2 - \frac{1}{6}\,x^3$$

Toelichting

Nu zijn `verwijderfoutepunten` en `polynoom` lokale procedures. Deze kunnen alleen worden gebruikt tijdens de verwerking van de statements in de body van `Polynoom`, omdat `verwijderfoutepunten` en `polynoom` als lokale variabelen in `Polynoom` zijn gedeclareerd. Dit

gebruik van procedures als lokale variabelen noemen we *nesting* van procedures. Vooral bij grotere programma's kan nesting een goed hulpmiddel zijn om de programma's overzichtelijk te houden.
De beide lokale procedures in `Polynoom` bevatten zelf weer onder andere `A` en `i` als lokale variabelen. Deze variabelen zijn alleen te gebruiken gedurende de afwerking van de rij statements in de body van `verwijderfoutepunten` en `polynoom`. Gedurende deze afwerking is een eventuele waarde van de lokale `A` gedeclareerd in `Polynoom` niet beschikbaar: verwijzingen naar `A` in `verwijderfoutepunten` hebben betrekking op de variabele `A` die gedeclareerd is in het statement `local i,A,M` uit `verwijderfoutepunten`.
Wanneer in de procedure `verwijderfoutepunten` een verwijzing naar de variabele `antwoord` zou voorkomen, wordt *niet* de lokale variabele `antwoord` uit `Polynoom` bedoeld, maar de *globale* variabele met de naam `antwoord`. ◇

In het algemeen geldt de regel:[68]

Bij een verwijzing *naar een niet-gedeclareerde variabele met naam "name" wordt gezocht naar deze variabele tussen de impliciete en expliciete lokale en globale variabele in de omvattende (geneste) procedures, van binnen naar buiten.*
Als de naam "name" voorkomt als parameter, of als local of global gedeclareerd is, wordt deze in de verwijzing gebruikt.
Als in de omvattende procedures de betreffende naam niet *is gevonden, is zij globaal, tenzij zij voorkomt links van de toekenningsoperator* `:=`*, of een dummy-variabele is in een loop (for of while). In deze laatste twee gevallen is de variabele met de naam "name" lokaal.*

Wanneer op de plek van een statement de eventuele waarde van een variabele kan worden *veranderd*, zeggen we dat het statement behoort tot het *bereik* van die variabele. (Engels: *scope* van die variabele.)

Het *bereik* van een variabele is *altijd* óf de gehele sessie, óf alleen de procedure waarin hij als lokale variabele is gedeclareerd. Daarbij is het bij geneste procedures ook mogelijk om variabelen te hebben die in de buitenste procedure als lokale variabelen zijn gedeclareerd en waarvan de *waarde* in de binnenste procedure *bekend* is. Zie de volgende voorbeeldsessie:

[68] Deze regel is verschillend van de regels die gelden voor lokale variabelen in talen als Pascal en C.

Voorbeeldsessie

Een globale variabele in een lokale procedure; de lokale `y` krijgt de waarde van de globale `x`:

```
> buiten1 := proc()
    local x, binnen;
    binnen := proc()
      local y; global x;
      y := x;
      printf( "%A %A\n", "Is y=4?  ", evalb(y=4) );
     printf( "%A %A", "Waarde van de lokale y:  ", y)
    end proc;
    x := 4;
    binnen()
  end proc:

> x := 372:

> buiten1();

Is y=4?  false
Waarde van de lokale y:  372


> x;
```

$$372$$

Een globale variabele krijgt in een lokale procedure een waarde;

```
> buiten2 := proc()
    local x, binnen;
    binnen := proc()
      global x; x := 80
    end proc;
    x := 4;
    binnen()
  end proc:

> x;
```

$$372$$

```
> buiten2();
```

$$80$$

De waarde van `x` is door de aanroep van `buiten2` veranderd:

```
> x;
```

$$80$$

```
> x := 'x':  y := 'y':
```

Nu is `x` een lokale variabele in `buiten3` en in `binnen` niet gedeclareerd:

```
> buiten3 := proc()
    local binnen;
    binnen := proc()
      local y;
      printf( "%A %A\n", "Wat is x?  ", x );
      x := 4;
      y := x;
      printf( "%A %A\n", "Is y=4?  ", evalb(y=4) );
      printf( "%A %A\n", "Wat is x nu?  ", x )
    end proc;
    x := 4;
    binnen();
    printf( "%A %A", "x in buiten3 =", x )
  end proc:
```

```
Warning, `x` is implicitly declared local to procedure `buiten3`
```

De impliciet globale `x` krijgt eerst een waarde, voordat we `buiten3` aanroepen:

```
> x := 3:
> buiten3();
```

```
Wat is x?  4
Is y=4?  true
Wat is x nu?  4
x in buiten3 = 4
```

```
> x;
```

$$3$$

```
> binnen();
```

$$\text{binnen}()$$

Toelichting

In de eerste twee versies van de procedure `buiten` is een *lokale* `x` in de *buitenste* en een *globale* `x` in de *binnenste* procedure *expliciet* gedeclareerd. De lokale versie van `x` blijkt in de binnenste procedure niet bereikbaar te zijn. Het lukt niet om de lokale `y` in de binnenste procedure gelijk te maken aan de `x` die in de buitenste procedure de waarde 4 gekregen heeft. Uit de tweede versie blijkt dat expliciete declaratie van `x` als globale variabele betekent dat dit de `x` is die *buiten* de procedure `buiten2` bekend is.
In de derde versie wordt `x` (als globale variabele, namelijk buiten elke procedure) gelijk gemaakt aan 3. Als `buiten3` wordt aangeroepen, wordt `x` eerst gelijk gemaakt aan 5. In de procedure `buiten3` is `x` niet door ons gedeclareerd. Maple kan ook geen declaratie van `x` vinden in een procedure die `buiten3` omvat. Volgens de regels over lokale en globale variabelen wordt `x` dan een variabele die lokaal is in `buiten3`: hij komt namelijk links van een toekenningsoperator in de procedure `buiten3` voor en is daarmee impliciet een lokale variabele binnen `buiten3`. Merk op dat Maple een waarschuwing geeft.

Na deze toekenning wordt in `buiten3` de procedure `binnen` aangeroepen. In `binnen` komt `x` links van de toekenningsoperator voor. De variabele `x` is niet gedeclareerd als lokaal in `binnen`, dus wordt éérst in de omvattende procedure naar `x` gezocht. Die is te vinden in de procedure `buiten3`, waarmee door het statement `x:=4` de waarde van de `x` die lokaal is in `buiten3` wordt veranderd. De globale `x` is onveranderd.
We merken ook op dat de procedure `binnen` uitsluitend binnen de procedure `buiten` gebruikt kan worden. ◇

Als er behoefte is aan variabelen met een bereik binnen een procedure, inclusief de daarin geneste lokale procedures, dan moet men gebruikmaken van een *module*, zie §29.6. De betreffende variabelen worden dan binnen de module `local` gedeclareerd.

30.3  Het 'call by (evaluated) name'-principe

Dit is een nuancering van wat in §28.7 is besproken over het call by value- en het call by name-principe.
In principe evalueert Maple *alle* parameters in een procedure-aanroep volgens de normale evaluatieregels, behalve als bij de proceduredefinitie in de heading uitdrukkelijk is vermeld (door `::evaln`) dat dat *niet* moet gebeuren. We zeggen dan dat zo'n parameter wordt doorgegeven via het *call by evaluated name*-principe. In de sessie (zie de voorbeeldsessie op blz. 442)

```
eps := 0.01:  antw := nulp3(f,-1,1,eps,aantal);
```

worden de *invoer*parameters `f` en `eps` volgens het call by value-principe doorgegeven. De *uitvoer*parameter `aantal` daarentegen wordt volgens het call by evaluated name-principe doorgegeven.

Elke actuele parameter die tot een naam evalueert, kan als uitvoerparameter dienen, dat wil zeggen dat er in de procedure een waarde aan kan worden toegekend, óók als dat in de heading niet uitdrukkelijk door `::evaln` zou zijn aangegeven. In alle andere gevallen zal een poging tot waarde-toekenning aan een actuele parameter resulteren in de foutmelding 'illegal use of a formal parameter'.
We demonstreren een en ander aan de hand van een voorbeeld van een procedure die niets anders doet dan een waarde toe te kennen aan de parameter waarmee hij wordt aangeroepen.

Voorbeeldsessie

```
> p1 := proc(a) a := 10 end proc:
```

De actuele parameter `b` heeft nog geen waarde en evalueert dus tot een naam:

```
> p1(b): b;
```

$$10$$

Door de aanroep `p1(b)` krijgt `b` een waarde.

Nu heeft `b` wél een waarde

```
> b := 11: p1(b);
```

```
Error, (in p1) illegal use of a formal parameter
```

De actuele parameter `'b'` evalueert wél tot een naam:

```
> b := 11: p1('b'): b;
```

$$10$$

Nu evalueert `b` wel tot een naam, maar niet z'n eigen naam:

```
> b := c: p1(b): c;
```

$$10$$

Nu definiëren we de formele parameter in de heading uitdrukkelijk als uitvoerparameter:

```
> p2 := proc(a::evaln) a := 10 end proc:
> b := 11: p2(b): b;
```

$$10$$

Hierdoor evalueert `b` tot z'n *eigen* naam:

```
> c := 'c': b := c: p2(b): b,c;
```

$$10, c$$

Toelichting

In de procedure `p1` is de parameter niet uitdrukkelijk als uitvoerparameter aangegeven. Voor de *eerste* aanroep heeft `b` nog geen waarde, evalueert dus tot een naam, zodat de aanroep `p1(b)` resulteert in `b := 10`.
Bij de *tweede* aanroep evalueert `b` niet tot een naam, maar tot de waarde 10, en dat resulteert voor de body van `p1` in de toekenning `11 := 10` en dat geeft een foutmelding.
Bij de *derde* aanroep van `p1` evalueert het argument `b` tot de *naam* `c`. Daardoor komt `p1(b)` neer op `c := 10`.

In de procedure `p2` wordt de formele parameter uitdrukkelijk als uitvoerparameter gespecificeerd. Dat betekent dat de aanroep `p2(b)` wordt 'vertaald' als `b := 'b': b := 10;` Dit verklaart ook waarom na de laatste aanroep `c` *niet* van waarde is veranderd. ◇

De algemene regel luidt:

> Bij de aanroep van een procedure worden de actuele parameters eerst volledig geëvalueerd voordat de statements van de body worden uitgevoerd.

Dat betekent dat *toekenningen* aan formele parameters in een procedure alléén mogelijk zijn als de actuele parameters evalueren tot een naam.

De uitzondering op deze regel wordt gevormd door tables en procedures die altijd tot een naam evalueren, óók als ze als actuele parameter van een procedure optreden (zie §25.3, blz. 395), én voor de *entries* van een Array (en dus ook van een vector of matrix), die *altijd* veranderd kunnen worden. Dat betekent dat bijvoorbeeld een Array die in de parameterlijst van een procedure voorkomt wél van inhoud, maar niet van afmetingen kan veranderen. Een tabel wordt via het call by evaluated name-mechanisme doorgegeven, en kan dus in de procedure zowel van inhoud als van omvang veranderen. Aan een tabel kunnen dus ook entries worden toegevoegd en er kunnen entries worden verwijderd. Zie ook opgave 28.8.

30.4  Uitgestelde evaluatie van de actuele parameters

In sommige gevallen – vooral wanneer een procedure-aanroep optreedt als actuele parameter in een andere procedure – zou men willen dat deze pas wordt geëvalueerd wanneer de (numerieke) waarde van de argumenten bekend is. Bijvoorbeeld: Als we de functie f definiëren als

```
f := proc(x) if x<1 then 1 else -1 end if end proc:
```

dan zal de aanroep `f(x)` zolang `x` nog geen waarde heeft resulteren in een foutmelding omdat Maple niet kan uitmaken of de uitdrukking `x<1` waar of onwaar is. Het volgende voorbeeld laat zien dat het soms nuttig is ervoor te zorgen dat de aanroep `f(x)` in het geval `x` niet van het type `numeric` is, `f(x)` in ongeëvalueerde vorm retourneert.

Voorbeeldsessie

```
> f := proc(x) if x<1 then 1 else -1 end if end proc:
> f(2);
```

$$-1$$

```
> f(x);

Error, (in f) cannot determine if this expression is true or
false: x < 1
> g := proc(x)
    if not type(x,numeric) then return 'procname(args)'
    elif x<1 then 1
    else -1
    end if
  end proc:
> g(x);
```

$$g(x)$$

```
> Sum(g(n), n=-2..4):  % = value(%);
```

$$\sum_{n=-2}^{4} g(n) = -1$$

```
> plot( g(x), x=-4..5 );
```

(Dit gaat goed)

De integraal echter

```
> Int( g(t), t=-4..5 ):  % = value(%);
```

$$\int_{-4}^{5} g(t)\,dt = \int_{-4}^{5} g(t)\,dt$$

gaat dus niet goed.

Een goed alternatief is hier natuurlijk

```
> h := x -> piecewise( x<1, 1, -1 ):  h(x);
```

$$\begin{cases} 1 & x < 1 \\ -1 & otherwise \end{cases}$$

```
> int(h(t), t=-4..5);
```

$$1$$

Toelichting

`procname`

In een procedure zijn de variabelen `procname` en `args` bekend; `procname` is uiteraard de naam van de procedure zelf en `args` is de expressierij van actuele parameters in ongeëvalueerde vorm.
Bij het gebruik van `g(n)` als actuele parameter in `sum` gebeurt er het volgende. Bij de aanroep van `sum` wordt `g(n)` direct geëvalueerd. Nu heeft `n` nog geen waarde, dus het resultaat is: `g(n)`. Vervolgens gaat `sum` aan het werk en moet $g(-2)+g(-1)+\cdots+g(4)$ uitrekenen. Hierbij moeten `g(-2)` enzovoort worden geëvalueerd en dat levert het gewenste resultaat. Bij `plot` gaat het net zo. Zie ook opgave 27.1.

`piecewise`

Voor het berekenen van de integraal werkt de truc niet omdat de integraal niet wordt berekend door het invullen van een aantal punten in `g`. De procedure `piecewise` is speciaal ontworpen om deze moeilijkheid te omzeilen. ◇

30.5 Evaluatie van lokale variabelen

We behandelen nog één voetangel die verborgen zit in het werken met lokale variabelen. We stuiten daarop als we de 'Regula Falsi'-algoritme van §27.4 (zie de voorbeeldsessie op blz. 426) in een procedure proberen onder te brengen.
We nemen de formule van het snijpunt `s`, die we in §27.4 hebben berekend, over in de procedure en we maken lokale kopieën van de relevante invoerparameters. Uiteraard voegen we ook een uitvoerstatement toe.

Voorbeeldsessie

```
> rf := proc( f, links,rechts, eps )
    local s, xl, yl, xr, yr;
    s := (-xl*yr+yl*xr)/(yl-yr);
    xl := links; xr := rechts;
    yl := evalf(f(xl)):  yr := evalf(f(xr)):
    while abs(f(s)) > eps do
      if f(s)*f(xl) < 0
      then xr := s else xl := s
      end if:
      yl := evalf(f(xl)):  yr := evalf(f(xr))
    end do;
    s
  end proc:
> f := x -> 9*x^3 + 5*x^2 + 8*x + 6:
> rf( f, -1,1, 0.01 );

Error, (in rf) cannot determine if this expression is true or
false: 0. < abs(9*(-xl*yr+yl*xr)^3/(yl-yr)^3+
5*(-xl*yr+yl*xr)^2/(yl-yr)^2+8*(-xl*yr+yl*xr)/(yl-yr)+5.99)
> tracelast;

 rf called with arguments: f, -1, 1, .1e-1
 #(rf,6): while eps < abs(f(s)) do ... end do;

Error, (in rf) cannot determine if this expression is true or
false: 0. < abs(9*(-xl*yr+yl*xr)^3/(yl-yr)^3+
5*(-xl*yr+yl*xr)^2/(yl-yr)^2+8*(-xl*yr+yl*xr)/(yl-yr)+5.99)

  locals defined as: s = (-xl*yr+yl*xr)/(yl-yr), xl = -1,
yl = -6., xr = 1, yr = 28.
```

Toelichting

Wat gaat hier mis? De opdracht `tracelast` lokaliseert het optreden van de fout in regel 6 waar de while-loop begint: de boolean

`abs(f(s)) > eps` kan niet worden geëvalueerd. Daaronder zien we dat `xl`, `yl`, `xr` en `yr` wel degelijk een waarde hebben, maar dat deze *niet* in `s` worden ingevuld.
De reden hiervoor is dat lokale variabelen maar één niveau worden geëvalueerd, in tegenstelling tot globale variabelen en invoerparameters (zie Module 25). ◇

Met het proceduretje `demo` wordt dat nog eens gedemonstreerd:

Voorbeeldsessie

```
> demo := proc()
    local x;
    '''x'''
  end proc:
> demo();
```

$$''x''$$

```
> eval('''x''',1);
```

$$''x''$$

Toelichting

`demo()` haalt precies één paar quotes van de lokale `x` af. ◇

We kunnen het probleem dat de locale variabele `s` in de Regula Falsi-procedure niet wordt geëvalueerd op twee manieren oplossen:

(1) Door van `s` een (lokale) procedure te maken:
`s := unapply((-xl*yr+yl*xr)/(yl-yr), xl,yl,xr,yr)`
en in de hele procedure elke `s` vervangen door de aanroep `s(xl,yl,xr,yr)`;

(2) Van elke `s` *volledige* evaluatie afdwingen met een `eval`-commando.

In de verbeterde versie hebben we de tweede mogelijkheid gekozen.

Voorbeeldsessie

```
> rf := proc( f, links,rechts, eps)
    local s, xl, yl, xr, yr;
    s := (-xl*yr+yl*xr)/(yl-yr);
    xl := links; xr := rechts;
    yl := evalf(f(xl)):  yr := evalf(f(xr)):
    while eval( abs(f(s)) ) > eps do
      if eval( f(s)*f(xl) ) < 0
      then xr := eval(s) else xl := eval(s)
      end if:
      yl := evalf(f(xl)):  yr := evalf(f(xr))
    end do;
    eval(s)
  end proc:
```

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

$$-0.6828656635$$

30.6 Options; Remember-tables

option

In de definitie van een procedure mag, direct na eventuele `local`- en `global`-statements, een `option` statement voorkomen. Zo'n statement heeft de vorm

```
option optie1, optie2, ...
```

Een optie in een `option` statement heeft tot doel de procedure wat bijzondere eigenschappen te geven.
We zullen ons hier beperken tot een bespreking van de optie `remember`. In de opgaven komt nog aan de orde de optie `trace`. Raadpleeg zo nodig `?options`.

option remember

Bij elke Maple-procedure hoort een *remember-table*. Elementen uit deze tabel (zie §7.4) hebben als index de waarden van de argumenten waarmee een procedure is aangeroepen. De bij die index behorende waarde is het resultaat dat de procedure met de betreffende argumenten heeft opgeleverd. Voordat Maple met de verwerking van de statements in de body van de procedure begint, kijkt zij eerst of de argumenten waarmee de procedure is aangeroepen al voorkomen als index van de remember-table. Zo ja, dan wordt meteen de bijbehorende waarde afgeleverd; zo nee, dan worden de statements in de body uitgevoerd. Het resultaat hiervan wordt toegevoegd aan de remember-table, echter alleen dan als bij de definitie van de procedure direct na eventuele `local`- en `global`-statements het statement `option remember` voorkomt.
De remember-table van een procedure is opvraagbaar als 4^{de} operande van die procedure.

Voorbeeldsessie

```
> g := proc(x::integer) option remember; x^2 end;
```

$$g := \mathbf{proc}(x::integer)\,\mathbf{option}\,remember;\ x^2\,\mathbf{end\ proc}$$

```
> g(2) := 0;
```

$$g(2) := 0$$

```
> [ g(1), g(2), g(3) ];
```

$$[1,\ 0,\ 9]$$

```
> op(4, eval(g));
```

$$\text{table}([1 = 1,\ 2 = 0,\ 3 = 9])$$

Toelichting

Door de expliciete toekenning `g(2) := 0` plaatsen we de functiewaarde 0 voor het argument 2 in de remember-table van `g`. Als we daarna `g(2)` opvragen, dan berekent `g` deze waarde niet, maar haalt hem uit de remember-table. ◇

We kunnen door een directe toekenning ook elementen in een remember-table van een (nog) niet gedefinieerde functie zetten. Het statement

```
g(2) := 4;
```

maakt dat `g` een procedure wordt (met onbekende body), met een remember-table. Dit is ook feitelijk wat er gebeurt in een statement als

```
g(x) := x^2;
```

en is de reden waarom `g(t)` dan niet t^2 oplevert. Zie ook de voorbeeldsessie in §3.2.

Voorbeeldsessie

```
> g := 'g':  g(2) := 4;
```

$$\text{g}(2) := 4$$

```
> op(4, eval(g));
```

$$\text{table}([2 = 4])$$

```
> g(x) := x^2;
```

$$\text{g}(x) := x^2$$

```
> op(4, eval(g));
```

$$\text{table}([2 = 4,\ x = x^2])$$

```
> g(t);
```

$$\text{g}(t)$$

(t staat nog niet in de remember-table van g)

```
> g := proc(y) option remember; y^3 end proc;
```

$$g := \mathbf{proc}(y)\,\mathbf{option}\,\mathit{remember};\ y^3\ \mathbf{end\ proc}$$

```
> tg := op(4,eval(g));
```

$$tg :=$$

```
> g(2);
```

Toelichting

Hier maken we eerst een remember-table van de verder ongedefinieerde functie `g`. Als we dan daarna de functie zouden willen definiëren voor 'de overige' argumenten, dan kan dat niet. De toekenning `g := ...` maakt een *nieuwe* `g`, met lege remember-table. ◇

`forget`

Soms is het nodig elementen te verwijderen uit remember-tables. Dit kan met de procedure `forget`. Als `g` een procedure is, zorgt

```
forget(g)
```

ervoor dat de gehele remember-table van `g` wordt leeggemaakt;

```
forget(g,x)
```

maakt dat alleen de waarde van de tabel die bij `x` hoort wordt verwijderd.

Opgave 30.1

Een nuttig hulpmiddel om het rekenwerk in (recursieve) procedures in de hand te houden is het statement `option remember`:

```
> fib2 := proc(n::integer)
    option remember;
    if n=0 or n=1 then 1
    else fib2(n-1) + fib2(n-2)
    end if
  end proc;
```

Bereken

```
st := time():  fib(25); t1 := time() - st;
```

en

```
st := time():  fib2(25); t2 := time() - st;
```

Hierbij is `fib` de procedure uit opgave 28.5.

`time`

Het commando `time()` geeft de rekentijd (in seconden) die sinds het begin van de sessie is verlopen; `t1` en `t2` zijn dus de voor het berekenen van `fib(25)`, resp. `fib2(25)` benodigde rekentijd.
Welk van de twee kost het minste tijd? Verklaar het verschil. Hoeveel aanroepen van `fib` en `fib2` zijn er nodig geweest?

Opgave 30.2

Voer de volgende sessie uit:

```
> f := proc(x::float)
    option remember;  evalf(sqrt(x))
  end proc;
> f(2.1);
> Digits := 20;
> f(2.1);
```

Verklaar de resultaten! Wanneer moeten we dus oppassen met de `remember`-optie?

Opgave 30.3

Voer de volgende sessie uit:

```
g := proc(x) options remember; x^2; end proc:
g(2) := 0:  g(3):
h := proc(x) x^2; end proc:  h(2) := 0:  h(3):
```

en bekijk vervolgens de remember-tabellen van `g` en `h`. Wat gebeurt er kennelijk na het statement `h(2) := 0`?

Opgave 30.4

`trace`

Een ander hulpmiddel dat soms handig kan zijn om de werking van een procedure precies na te gaan is het statement `option trace`. Bekijk het effect van dit statement aan de hand van de procedure `fac` die u in opgave 28.5 hebt gemaakt:

- Neem het statement `option trace;` op in de procedure `fac`.
- Maak een procedure `fac2` met het statement

  `option remember, trace;`

Bereken achtereenvolgens `fac(4)` en `fac(5)`, en daarna `fac2(4)` en `fac2(5)`.