Module 29 Procedures voor hergebruik

Onderwerp	Procedures: Foutmelding en -afhandeling, waarschuwingen en andere boodschappen, definitie van types, modules.
Voorkennis	Module 3, 7, 8, 25, 26,27, 28
Expressies	`error, try..catch, WARNING, %n, printf, ::, AddType, module, export, :-`
Bibliotheken	`TypeTools`
Zie ook	Module 30

In deze module gaan we verder in op een aantal bijzondere mogelijkheden bij het gebruik van procedures. In Module 28 ging het hoofdzakelijk om het schrijven van procedures voor eenmalig 'eigen' gebruik. Als men procedures wil bewaren om ze later nog eens te gebruiken, of om ze ook aan anderen beschikbaar te stellen, moeten er hogere eisen aan worden gesteld. In de eerste plaats moet een nauwkeurige beschrijving van invoer- en uitvoervariabelen, alsmede van de werking van de procedure worden gegeven. Daarnaast is het belangrijk dat zo'n procedure kan controleren of hij op de juiste manier wordt aangeroepen en een informatieve foutmelding geeft als dat niet zo is, of als een van de opdrachten in de procedure niet correct kan worden uitgevoerd.
We gaan zeer kort in op de mogelijkheden om zelf een bibliotheek met procedures te maken, inclusief wat primitieve help-faciliteiten.

In deze module komen achtereenvolgens aan de orde:

(1) Genereren en afhandelen van foutmeldingen;
(2) Type-checking;
(3) Zelf definiëren van typen;
(4) Inleiding procedurebibliotheken (modules).

29.1 Foutmeldingen

We bekijken weer de procedure `polynoom` uit §28.6 om een polynoom door een gegeven aantal punten te berekenen. We zullen de procedure aanroepen voor een rij punten waarvan we zeker weten dat er geen oplossing is. In de eerste plaats willen we dat in dergelijke gevallen een foutmelding wordt gegenereerd. Dit doen we met de Maple-procedure `error`.

`error`

In de tweede plaats willen we dat eventuele 'dubbele punten' in de input worden verwijderd. Dat is nodig omdat de procedure `polynoom` het *aantal* punten in de invoer telt om de graad van de uitvoerpolynoom vast te stellen – dit moeten dan natuurlijk wel *verschillende* punten zijn.
We krijgen dan:

Voorbeeldsessie

Hier is `polynoom` de procedure uit §28.6:

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

$$q := x \to a_0 + a_1\,x + a_2\,x^2 + a_3\,x^3 + a_4\,x^4$$

Er bestaat geen polynoom met q(1) = 2 én q(1) = 3.

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

$$q := x \to -\frac{10}{3} - 8\,a_3 + (7 + 14\,a_3)\,x + \left(-\frac{5}{3} - 7\,a_3\right)\,x^2 + a_3\,x^3$$

Het punt (1, 2) komt dubbel voor. De waarde a_3 is willekeurig, we hadden zelfs met een tweedegraadspolynoom kunnen volstaan.

Nieuwe versie:

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

$$q := x \to -\frac{10}{3} + 7\,x - \frac{5}{3}\,x^2$$

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

Error, (in polynoom) Een waarde van x met verschillende
   y-waarden

> eval(q);
```

$$x \to -\frac{10}{3} + 7\,x - \frac{5}{3}\,x^2$$

Toelichting

We hebben eerst laten zien wat er gebeurt als we de oude procedure aanroepen met argumenten waarmee we in §28.6 geen rekening hebben gehouden.

Eerste geval: 'dubbele punten'. Er wordt een polynoom gemaakt met te hoge graad; er komt dan uiteraard nog een 'vrij te kiezen' (lokale!) variabele in de output mee naar buiten.
Tweede geval: wat gebeurt er als we de oude procedure `polynoom` met een 'onmogelijke' input aanroepen? Het stelsel vergelijkingen is dan strijdig en heeft dus geen oplossing. Dat betekent dat er in het `subs`-commando in het uitvoerstatement niets gebeurt, en dat de `p` met onbepaalde (lokale!) coëfficiënten $a_0, \ldots, a_4$ de uitvoer wordt.
In de nieuwe versie van `polynoom` hebben we een lijst `A` van de argumenten gemaakt. Om het 'eerste geval' te vermijden hebben we daarbij de omweg gemaakt er eerst door {`args`} een *verzameling* van te maken (waardoor de eventuele 'dubbele' argumenten automatisch worden verwijderd) en vervolgens de elementen van deze verzameling, `op({args})`, in een lijst te zetten.
Het 'tweede geval' is niet te repareren, met dergelijke input kan er geen zinvolle polynoom worden gemaakt. Daarom wordt na het `solve`-commando een foutmelding gegenereerd in het geval dat de oplossing `s` van het `stelsel` leeg (dat wil zeggen: `NULL`) is.
Het `error`-commando lijkt op het eerste gezicht een uitvoerstatement. We zien echter dat de waarde van `q` niet is veranderd. Dat betekent dat `error "boodschap"` *niet* hetzelfde effect heeft als `return "boodschap"` of `return print("boodschap")`. Dan zou namelijk de waarde van `q` gelijk moeten worden aan *"boodschap"*, respectievelijk `NULL`. Blijkbaar zorgt `error` er dus voor dat de procedure beëindigd wordt zonder dat er verder iets gebeurt, dus dat het statement `q := ...` *niet* wordt uitgevoerd. ◇

`try..catch`

Het is ook mogelijk om Maple een gepaste actie te laten ondernemen na het optreden van een bepaalde foutmelding. Daarvoor is het `try..catch` statement bedoeld:

```
try
  antwoord := polynoom(argumenten)
catch "Een waarde van x met verschillende y-waarden":
  antwoord := wat_anders
end try
```

Er gebeurt dan het volgende. Als de uitvoering van het commando `polynoom(argumenten)` *niet* in een foutmelding resulteert, dan wordt het resultaat gewoon aan `antwoord` toegekend. Als er wél een foutmelding is, dan wordt gekeken of deze foutmelding wellicht luidt: "`Een waarde van x met verschillende y-waarden`". Zo ja, dan krijgt `antwoord` de waarde `wat_anders`, en zo nee, dan wordt de foutmelding gegeven. Zie §29.3 voor een voorbeeld.

29.2 Procedures met een variabel aantal argumenten

We moeten de uitspraak aan het begin van §28.6 iets nuanceren. We schreven daar: "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." Dit is alléén waar als er in de body van de procedure naar de betreffende parameters wordt verwezen. We laten dat zien met een eenvoudig voorbeeld.

Voorbeeldsessie

```
> p := proc(a,b,c) a+b end proc:
> p(12);
```

```
Error, invalid input: p uses a 2nd argument, b, which is missing
```

```
> p(12,13);
```

$$25$$

```
> p(12,13,14);
```

$$25$$

```
> p(12,13,14,15);
```

$$25$$

Toelichting

De procedure `p` is met drie formele parameters gedefinieerd, waarvan er evenwel maar twee in de body worden gebruikt. Als we de procedure met één actuele parameter aanroepen, krijgen we inderdaad de verwachte foutmelding. Met twee parameters gaat het goed, omdat de derde helemaal niet wordt gebruikt; méér dan drie parameters vormen ook geen probleem.
Wat zou er trouwens gebeuren als de body "`a+c`" zou zijn geweest, in plaats van "`a+b`"? ◇

Een praktische toepassing hiervan is het volgende. We beschouwen nogmaals de procedure `nulp3` van §28.7 (zie blz. 442). Als we niet geïnteresseerd zijn in het aantal benodigde iteratiestappen en de aanroep

```
nulp3(f,-1,1,0.01);
```

zouden doen, dan krijgen we de foutmelding:

```
Error, invalid input:  nulp3 uses a 5th argument, n (of
                type evaln), which is missing
```

We moeten er dus voor zorgen dat het statement `n := teller` zeker niet wordt uitgevoerd als de procedure met vier in plaats van vijf argumenten wordt aangeroepen. Dat doen we door deze toekenning op te nemen in een if-statement waarin op het aantal argumenten wordt getest.

Voorbeeldsessie

```
> nulp4 := 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;
    if nargs > 4 then n := teller end if;
    xm
  end proc:

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

Aanroep met vier argumenten:

```
> nulp4(f,-1,1,0.001);
```

$$-0.6831054690$$

Aanroep met vijf argumenten:

```
> nulp4(f,-1,1,0.001,aantal_stappen);
```

$$-0.6831054690$$

```
> aantal_stappen;
```

$$11$$

Toelichting

De toewijzing `n := teller` zal niet worden uitgevoerd als `nulp4` met 4 argumenten wordt aangeroepen. ◇

Een andere mogelijke toepassing van een test op het aantal argumenten is het genereren van een eenvoudige 'help'-boodschap voor het geval men niet meer precies zou weten hoe een procedure precies aangeroepen moet worden. Als men direct na de `local`-declaratie in de procedure zou opnemen

```
if nargs=0 then print( <korte handleiding> )
```

en de overige statements van de body dan tussen de `else` en de `end if` plaatst, dan wordt de *korte handleiding* afgedrukt als de procedure wordt aangeroepen als `nulp4()`, dus met nul argumenten. Overigens kan men zo'n iets uitgebreidere boodschap beter met een `printf` afdrukken, zie §29.3.

29.3 Boodschappen en waarschuwingen

Als een fout via een `try..catch`-statement wordt afgehandeld zal men dat in sommige gevallen aan de gebruiker bekend willen maken. Uiteraard is daarvoor een `print`-commando te gebruiken. Dat is echter tamelijk primitief; de procedure `WARNING` biedt meer mogelijkheden.

`WARNING`

Als voorbeeld zullen we de procedure `polynoom` uitbreiden. Als de fout van de voorbeeldsessie op blz. 454 optreedt, dan moeten de 'foute' punten uit de argumenten worden verwijderd. Uiteraard willen we dat hiervan melding wordt gemaakt, én dat wordt aangegeven wélke punten zijn verwijderd.
In de volgende sessie is `polynoom` de procedure uit §29.1.

Voorbeeldsessie

```
> 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;
    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:
```

Een procedure die de punten met dubbele x-waarden verwijdert:

```
> verwijderfoutepunten := proc()
    local i, L, M:
    L := {args}; i := 1;
    while i<=nops(L) do
      M := select( c->c[1]=L[i][1], L );
      if nops(M)>1 then L := L minus M end if;
      i := i+1
    end do;
    op(L)
  end proc:
```

Test:

```
> verwijderfoutepunten( [1,2],[2,4],[3,3],[4,-2],[1,3],[2,6],[1,5] );
```

$$[3,\,3],\,[4,\,-2]$$

De procedure `Polynoom` vangt de fouten van `polynoom` af.

```
> Polynoom := proc()
    local argumenten, antwoord;
    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:
> p := Polynoom( [1,2],[2,4],[3,3],[4,-2],[1,3] );
```

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

$$p := x \to -6 + 9\,x - 2\,x^2$$

```
> 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

We hebben eerst een hulpprocedure gemaakt die uit de rij argumenten alle elementen verwijdert die een gelijke x-coördinaat hebben.[67] Het is hier handig om van `args` een verzameling te maken, omdat we dan de verzamelingsoperator `minus` kunnen gebruiken (zie §8.2). Bovendien lopen we dan niet het gevaar dat we eventuele punten waar de x- én de y-coördinaat *beide* aan elkaar gelijk zijn, ten onrechte verwijderen.
We gebruiken deze procedure in de procedure `Polynoom` (met hoofdletter). Als `polynoom` (met kleine letter) een foutboodschap geeft, dan geven we eerst een waarschuwing en vervolgens berekenen we `polynoom` opnieuw met een gekuiste rij argumenten.
Als er geen argumenten hoeven worden weggelaten, krijgen we ook geen waarschuwing. ◇

Opmerking. De aanroep van `verwijderfoutepunten` zou natuurlijk ook direct in `polynoom` (kleine letter) kunnen worden opgenomen. Wij hebben hier voor deze wat omslachtige methode gekozen om het gebruik van `try..catch` te kunnen demonstreren.

Een `WARNING`-commando heeft de volgende vorm:

[67] Zie voor het gebruik van hulp-procedures ook §30.2.

```
WARNING("boodschap met %1 en %2 als variabelen", x, y )
```

Dit geeft als uitvoer: "Warning, boodschap met x en y als variabelen", waarbij voor x en y de eventuele waarde van deze variabelen is ingevuld.
In het voorbeeld hebben we maar één parameter (%1) in de boodschap, namelijk de verzameling van weggelaten argumenten.
Een dergelijke constructie met variabelen in de boodschap is ook bij een `error`-statement mogelijk.

Overzicht. In de volgende sessie geven we een overzicht van de verschillende mogelijkheden.

Voorbeeldsessie

Boodschap als uitvoer:

```
> p1 := proc()
    local x,y;
    x := 10:
    "Er geldt:  x=", x, "en y=", y
  end proc:
```

```
> q := 1:  q := p1();
```

$q :=$ "Er geldt: x=", 10, "en y=", y

```
> q;
```

"Er geldt: x=", 10, "en y=", y

Boodschap in printopdracht:

```
> p2 := proc()
    local x,y;
    x := 10:
    print("Er geldt:  x=", x, "en y=", y);
  end proc:
```

```
> q := 1:  q := p2();
```

"Er geldt: x=", 10, "en y=", y

$q :=$

```
> q;
```

(geen uitvoer)

Printopdracht met formattering:

```
> p3 := proc()
    local x,y;
    x := 10:
    printf("%A %A %A %A","Er geldt:  x =", x, "en y =", y)
  end proc:
```

```
> q := 1:  q := p3();
```

```
Er geldt: x = 10 en y = y
```

$q :=$

Als foutboodschap:

```
> p4 := proc()
    local x,y;
    x := 10:
    error "Er geldt:  x = %1 en y = %2", x, y
  end proc:

> q := 1:  q := p4();

Error, (in p4) Er geldt: x = 10 en y = y

> q;
```

$$1$$

Als waarschuwing:

```
> p5 := proc()
    local x,y;
    x := 10:
    WARNING( "Er geldt:  x = %1 en y = %2", x, y )
  end proc:

> q := 1:  q := p5();

Warning, Er geldt: x = 10 en y = y
```

$$q := $$

Toelichting

In `p1` is het uitvoerstatement van de procedure een expressierij, die in dit geval bestaat uit: een string, een getal, een string en een symbool. Deze wordt door de aanroep `q := p1()` aan de variabele `q` toegekend. In `p2` is het uitvoerstatement een `print`-opdracht met de bovengenoemde expressierij als argumenten. Zo'n printopdracht heeft als waarde `NULL`. De boodschap is leesbaar, maar niet mooi. Dat maakt hem geschikt om tijdelijk in de procedure op te nemen om de werking ervan te kunnen volgen.

`printf` De procedure `printf` biedt de mogelijkheid om de tekst te 'formatteren'. Het eerste argument (tussen string-quotes) bevat de formatspecificaties voor elk van de volgende argumenten. Er zijn vele mogelijkheden; raadpleeg daarvoor `?printf`.

Ten slotte hebben we de `error`- en `WARNING`-opdracht. Hierbij kunnen

`%`n we met behulp van de *labels* `%1`, `%2` enzovoort gemakkelijk stukken tekst en Maple-expressies combineren.

Merk op dat er een verschil is tussen een `WARNING`-opdracht en een `error`-opdracht. De `error`-opdracht heeft tot gevolg dat de procedure bij aanroep in feite *niet* wordt uitgevoerd. Het effect van een `WARNING`-opdracht is hetzelfde als dat van een `print`-opdracht. ◇

29.4 Type-checking

We beschouwen nogmaals de procedure `nulp4` (zie blz. 457). In feite zouden we, voordat wordt begonnen met de verwerking van de statements in de body van de procedure, eerst willen nagaan of de invoerparameters waarmee ze wordt aangeroepen wel in orde zijn. Een aanroep als `nulp4( f, -1, 1, -0.001, n );` levert weliswaar geen foutmelding, maar Maple blijft dan wel erg lang rekenen. Dus zoiets zouden we toch willen verbieden. Dit kan door een zogenaamde *type-check* in te bouwen bij de definitie van `nulp4`. We moeten dan eerst nagaan welke typen we voor elke parameter willen toestaan. In ons geval kiezen we ervoor dat `f` een procedure is, dat `links` en `rechts` reële getallen zijn en dat `eps` positief is.
Er zijn twee manieren om dit te controleren.

Eerste manier: in de heading. Hierbij wordt in de procedure-definitie bij de formele parameters direct aangegeven van welk type deze moeten zijn. Dit gebeurt eenvoudig door het vereiste type er na een `::` direct achter te zetten.

::

Voorbeeldsessie

```
> nulp5 := proc( f::procedure,
            links::realcons, rechts::realcons,
            eps::positive, 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;
    if nargs > 4 then n := teller end if;
    xm
  end proc:
```

Nu krijgen we een automatische foutmelding bij een aanroep met parameters van een verkeerd type:

```
> nulp5( 9*x^3 + 5*x^2 + 8*x + 6, -1,1, -0.001, n );

Error, invalid input: nulp5 expects its 1st argument, f,
  to be of type procedure, but received 9*x^3+5*x^2+8*x+6

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

> nulp5( f, -1,1, -0.001, n );
```

```
Error, invalid input: nulp5 expects its 4th argument, eps,
  to be of type positive, but received -.1e-2
```

Tweede manier: in de body. Deze manier is vooral nuttig wanneer het de bedoeling is dat een procedure moet kunnen worden aangeroepen met een variabel aantal argumenten. We passen dit toe op de procedure `polynoom` uit §29.1. Zie ook opgave 29.1.

Voorbeeldsessie

```
> polynoom := proc()
    local A,X,Y,n,a,p,x,i,stelsel,s;
    for i from 1 to nargs do
      if not (type( args[i], list(realcons) ) and nops(args[i])=2)
      then error "%1e punt ( %2 ) niet goed", i, args[i] end if
    end do;
    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:
> polynoom( [1,2],[2,4],{3,4},[4,-2],[0,3] );

Error, (in polynoom) 3e punt ( {3, 4} ) niet goed
> polynoom( [1,2],[2,4,5],[3,4],[4,-2],[0,3] );

Error, (in polynoom) 2e punt ( [2, 4, 5] ) niet goed
> polynoom( [1,2],[2,4],[3,4],[4,q], rommel );

Error, (in polynoom) 4e punt ( [4, q] ) niet goed
```

Toelichting

In de for-loop wordt voor elk van de argumenten gecontroleerd of het een *lijst* van *reële* getallen is die *bovendien* precies twee elementen bevat. Zo nee, dan wordt de procedure direct met een foutboodschap verlaten; er wordt niet meer naar de rest van de input gekeken. ◇

29.5 Definitie van nieuwe typen

In sommige gevallen, vooral bij wat ingewikkelder mogelijkheden voor de actuele parameters van een procedure, is het gewenst om eigen

`AddType`

typen te definiëren. Hiervoor kan het commando `AddType` uit de bibliotheek `TypeTools` worden gebruikt. Als men bijvoorbeeld wil toestaan dat een actuele parameter een verzameling of een lijst van gehele getallen is, kan men het type `set_or_list` definiëren door

```
AddType( set_or_list, {set(integer), list(integer)};
```

Een andere mogelijkheid is de definitie van een type met een procedure die de waarde `true` of `false` teruggeeft. We geven daarvan een voorbeeld.

Voorbeeldopgave

Bepaal de primitieve F van een functie f, met $F(a) = b$. De procedure moet kunnen worden aangeroepen als `prim(f, F(a)=b);`

Voorbeeldsessie

```
> T := proc(x)
    if type(x,'=') then
      if type(lhs(x),function) and
         type(eval(op(0,lhs(x))),name)
      then if nops(lhs(x))=1 and
              type(op(1,lhs(x)),{name,numeric})
              and type(rhs(x),algebraic)
           then true
           else false
           end if
      else false
      end if
    else false
    end if
  end proc:
> TypeTools:-AddType( voorwaarde, T );
> prim := proc(f::procedure, vw::voorwaarde)
    local x, a, b;
    a := op(1,lhs(vw));
    b := rhs(vw);
    unapply( b + int(f(t),t=a..x), x )
  end proc:
> prim( y->ln(y), F(a)=g(a)+k );
```

$$x \to \mathrm{g}(a) + k - a\ln(a) + a + x\ln(x) - x$$

```
> prim(y->ln(y), g(a)+k=F(a));

Error, invalid input: prim expects its 2nd argument, vw,
  to be of type voorwaarde, but received g(a)+k = F(a)
```

Toelichting

Eerst wordt getest of het argument wel een 'vergelijking' is; als dat niet zo is, zijn we meteen klaar. Het linkerlid moet van het type

`function` zijn, dat wil zeggen van de vorm `F(a)`, met `F` ongedefiniëerd (dus niet zoiets als `sin`). Er wordt dus getest of de nulde operande (zie §26.6) van het linkerlid tot een naam evalueert.
Vervolgens wordt gekeken of er wel `F(a)` staat, en niet zoiets als `F(a,b,c)`. Het rechterlid mag ten slotte elke formule zijn (maar niet zoiets als `[p,q,r]`). ◇

29.6 Modules

Het is ook mogelijk een hele *bibliotheek* van procedures te maken. Dat is vooral nuttig als deze procedures gebruikmaken van 'hulpprocedures', of een aantal constanten gezamenlijk gebruiken. Dat kan door ze in een `module` te plaatsen.

`module`

We behandelen dit niet uitputtend, maar laten de mogelijkheden aan de hand van een voorbeeld zien. Daarin maken we een 'bibliotheek' met de naam `eurocalculator` waarin allerlei procedures zouden kunnen worden opgenomen om guldens, dollars, roepia's enzovoort naar euro's omgerekend kunnen worden en andersom, en waarbij de juiste afrondingen op hele centen gemaakt worden. Daarbij zijn vele omrekeningsfactoren nodig, waarvoor we geen (voor de hele Maple-sessie) globale constanten willen gebruiken.
In het voorbeeld zullen we alleen guldens naar euro's omrekenen en andersom. U kunt gemakkelijk zelf verzinnen hoe de module uitgebreid zou kunnen worden.

Voorbeeldsessie

```
> eurocalculator := module()
    export f_naar_E, E_naar_f;
    local factor, centen;
    option package;
    factor := 2.20371;
    centen := x -> round(100*frac(x));
    f_naar_E := proc(x::nonnegative)
      local ruw;
      ruw := evalf(x/factor):
      floor(ruw) + 0.01*centen(ruw)
    end proc:
    E_naar_f := proc(x::nonnegative)
      local ruw;
      ruw := evalf(x*factor):
      floor(ruw) + 0.01*centen(ruw)
    end proc:
  end module:
```

Aanroep van een procedure uit de module:

```
> eurocalculator :- f_naar_E(150.25);
```

$$68.1800000000$$

We willen al die overbodige nullen niet zien:

```
> interface(displayprecision = 2):
> eurocalculator :- f_naar_E(150.25);
```

$$68.18$$

Laden van de gehele module:

```
> with(eurocalculator);
```

$$[E\_naar\_f, f\_naar\_E]$$

```
> f_naar_E(233.37);
```

$$105.90$$

```
> E_naar_f(105.90);
```

$$233.37$$

De procedure **centen** is buiten de module niet bekend.

```
> centen(%);
```

$$centen\,(233.37)$$

Het aantal getoonde cijfers achter de komma weer terug naar wat het was (wordt namelijk niet veranderd met **restart**):

```
> interface(displayprecision = Digits):
```

Toelichting

De module `eurocalculator` is een soort bibliotheek die twee procedures beschikbaar maakt. Deze worden in de moduledefinitie door de declaratie `export` aangegeven.

`export`

De constante `factor` en de procedure `centen` zijn lokaal binnen de module. Zij kunnen uitsluitend door de andere procedures in de module worden gebruikt.

De procedures komen beschikbaar met de voor bibliotheken gebruikelijke commando's `modulenaam:-procedurenaam` (één procedure tegelijk) of `with(modulenaam)` (alle procedures).

`:-`

`interface`

Met het `interface`-commando wordt hetzelfde bereikt als wanneer via het menu Tools → Options... → Precision het aantal op het scherm getoonde cijfers achter de komma wordt veranderd. ◇

Uiteraard wil men zo'n module in de vorm van een echte bibliotheek in het computergeheugen opslaan. Hoe dat gaat kunt u ontdekken door `?repository` te raadplegen.

Met `save` en `read` (zie §2.5) komt u trouwens ook al een heel eind.

Opgave 29.1

Als we in de procedure `Polynoom` (hoofdletter), zie blz. 458, voor `polynoom` (kleine letter) de versie met de type-controle uit §29.4 zouden nemen, dan gaat er iets mis als de invoer niet uit lijsten van twee getallen bestaat. Ga na waarom.
Verander de procedure `Polynoom`, zo dat beide soorten fouten in `polynoom` correct worden afgehandeld.

Opgave 29.2

Schrijf een procedure `maxint` die van een aantal integers het verschil tussen het grootste en het kleinste getal bepaalt. Zorg ervoor dat van elk argument wordt nagegaan of het wel een integer is.

Opgave 29.3

Breid de procedure `nulp5` (§29.4) verder uit:

(a) Zorg dat er een korte beschrijving wordt gegeven als de procedure als `nulp5()` wordt aangeroepen;
(b) Test of [*links*, *rechts*] een interval is;
(c) Test of f continu is op het gegeven interval (zie `?iscont`), en of de functiewaarden in de eindpunten een verschillend teken hebben.

Kies zelf of er bij de verschillende situaties een foutmelding wordt gegeven, dan wel een waarschuwing met gepaste actie.