Introductie Erlang

Leo Rutten

15/12/2015

Table of Contents

1. Basis 2
1.1. Inleiding 2
1.2. De shell 3
1.3. Datatypes 4
1.3.1. Integer 4
1.3.2. Floats 5
1.3.3. Bitbewerkingen 5
1.3.4. Binaries 5
1.3.5. Atomen 5
1.3.6. Tuples 6
1.3.7. Lists 7
1.3.8. Strings 7
1.3.9. Pids, Poorten en referenties 7
1.3.10. Vergelijken 8
1.4. Modules 8
1.5. Variabelen en patronen 9
1.6. Functies 10
1.6.1. Eenvoudige functies 10
1.6.2. Functies met meerdere definities 11
1.6.3. Functies met guards 12
1.6.4. Ingebouwde functies 13
1.7. `Case` en `if` 13
1.8. Staartrecursie 15
1.9. Anonieme functies 16
1.10. List comprehensions 18
1.11. Datastructuren 19
1.11.1. Records 19
1.11.2. Maps 20
1.11.3. Proplist 21
1.12. Foutafhandeling 21
1.12.1. Runtime fouten 22
1.12.2. Exits 22
1.12.3. Throws met `try` en `catch` 22
1.13. Processen 23
1.14. Fouten en processen 31
1.14.1. Processen zonder links 31
1.14.2. Processen met links 32
1.14.3. Traps 33
1.14.4. Monitors 33
1.14.5. Processen met naam 34
1.15. ETS tabellen 35
1.16. Gedistribueerde processen 39
1.16.1. Concepten 39
1.16.2. Een connectie tussen nodes 41
1.16.3. Berichtenverkeer tussen gedistribueerde nodes 42
2. Gevorderd Erlang/OTP 44

2.1. Inleiding 44
2.2. Een eerste applicatie 46
2.3. Een applicatie bouwen met `erlang.mk` en `relx` 49
2.4. Cowboy integreren in een applicatie 53
2.5. ErlyDTL bijvoegen in een Cowboy applicatie 57
3. Extra's 61
3.1. Native implemented functions (NIF's) 61
3.2. C port 64
4. Oplossingen 69
4.1. Labo 1 69

1. Basis

1.1. Inleiding

De programmeertaal Erlang is al een hele tijd geleden ontworpen door Ericsson en werd hoofdzakelijk ingezet voor het programmeren van telecomapparatuur. Ondertussen wordt de taal ook met succes ingezet in het Internet. Een voorbeeld hiervan is CouchDB; dit is een NoSQL database die volledig in Erlang is ontwikkeld.

Voor wie al ervaring heeft met objectgeoriënteerde talen zoals C++ en Java is Erlang wel een tegenvaller: de taal is niet objectgeoriënteerd en heeft voor nieuwelingen een bijzonder vervelende eigenschap. Namelijk deze, de variabelen kan je wel initialiseren maar daarna kan je ze niet meer wijzigen. Door deze beperking moet je dan ook een andere programmeerstijl toepassen. Ondanks deze (en ook andere vreemde eigenschappen) is de taal stilaan aan het uitgroeien tot de taal die bijzonder goed geschikt is om complexe gedistribueerde softwaresystemen te ontwerpen.

We overlopen snel de eigenschappen van Erlang.

- Eigen virtuele machine

  Net zoals Java heeft Erlang een eigen virtuele machine. Dit heeft als voordeel dat je Erlang op bijna alle platformen kan draaien. Er zijn ook experimenten geweest om de Erlang virtuele machine rechtstreeks bovenop de hardware te draaien, dus zonder besturingssysteem.

  Deze virtuele machine heeft een eigen implementatie voor lichtgewicht threads of processen. Het kost maar weinig geheugen en weinig tijd om een nieuw proces te creëren. Het wordt dan ook aangemoedigd om het ontwerp van een complex systeem op te splitsen in veel processen.

- Functionele taal

  Erlang is een functionele programmeertaal. Dit betekent dat de functies geen zijeffecten (tenzij je informatie naar een bestand of database schrijft) hebben. Ook wordt zoveel mogelijk in termen van functies geschreven. Dit tekent dat er veel recursie wordt gebruikt. Deze recursie heb je ook nodig om herhalingen te maken. Erlang kent eenvoudige datatypes zoals integer en float en daarnaast ook complexere zoals tuple en lijsten. Je kan functies ook als data beschouwen. Erlang kent dus anonieme functies. Klassen ontbreken omdat Erlang geen objectgeoriënteerde taal is. Wel zijn er records.

- Onwijzigbare variabelen

  Het meest in het oog springende kenmerk is wel dat de Erlang variabelen wel éénmaal geïnitialiseerd kunnen worden en dat je ze daarna nooit meer kan wijzigen. Deze eigenschap zorgt ervoor dat de implementatie van de garbage collector veel eenvoudiger kan zijn. De oplossing voor het dilemma - Erlang overboord gooien en vervangen door C++, Java of toch in Erlang programmeren - is eenvoudig: van zodra een variabele moet wijzigen, bereken je de nieuwe waarde en geef je die door als parameter van een functie die je dan oproept. Hierdoor ontstaat er een nieuw run van de functie met zijn eigen stackframe waarin de nieuwe waarde opgeslagen wordt.

- Actormodel

  In Erlang zijn globale variabelen niet mogelijk. Ook ontbreken de synchronisatiemogelijkheden tussen processen zoals die wel in Java bestaan. Als je informatie tussen processen wil delen, dan moet je berichten sturen. Elk proces heeft een berichtenwachtrij. Het proces kan wachten tot er een bericht ontvangen en dan de passende reactie uitvoeren. Als de afzender mee opgenomen is in het bericht, kan de ontvanger een bericht als antwoord terugsturen.

  Met het actormodel vermijd je de klassieke synchronisatieproblemen.

- Gedistribueerd

  De huidige toepassingen zijn niet alleen ontworpen voor multicore systemen maar kunnen in veel gevallen ook gedistribueerd uitgebaat worden. Dit betekent dat je het gemeenschappelijke geheugen van de multicore hardware niet meer kan gebruiken voor de uitwisseling van informatie. Het is duidelijk dat het berichtenverkeer hier de oplossing is. In Erlang is het verzenden en ontvangen van berichten transparant. Je kan aan de code niet zien of berichten afkomstig zijn van een proces binnen de eigen multicore hardware of van een andere. Deze eigenschap maakt het mogelijk om applicaties te ontwerpen die op een veel grotere schaal werken.

- Patroonherkenning

  In Erlang ontbreken klassen; alleen records, tuples en lijsten kunnen dienen om data te structureren. Om snel bepaalde gegevens uit gestructureerde data te halen kan je patroonherkenning gebruiken. Deze patroonherkenning komt ook van pas om de verschillende ontvangen berichten te onderscheiden.

1.2. De shell

Erlang kan ofwel binair geïnstalleerd worden ofwel ga je de Erlang broncode zelf compileren. Hoe je de installatie doet, wordt niet in deze tekst uitgelegd. Erlang heeft zowel een compiler (`erlc`) en een shell (`erl`). Met de shell kan je zelf interactief een aantal kleine Erlang constructies uittesten. Je kan ook broncodemodules compileren, laden en testen. Voor kleine delen geschreven Erlang is het gebruik van de shell een oplossing. Voor grotere delen of samenstellingen moet je anders tewerk gaan.

Nu leggen we uit hoe je met de shell overweg kan. Die start je met:

```
user@etester:~/test$ erl
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe]

Eshell V7.1  (abort with ^G)
1>
User switch command
 --> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
 -->
```

Met `^g` kan je onderbreken. Bij de prompt die dan verschijnt, kan je user switch commands gebruiken. Met `h` wordt een overzicht gegeven van de commando's.

Bij `^C` kan je ook commando's gebruiken:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

Je kan bij de gewone prompt ook stoppen met `q()` of `init:stop()`.

`erl` verstaat het gebruik van TAB. Type `li` gevolgd door TAB en je ziet dat dit uitgebreid wordt tot `lists:`. Nog een TAB geeft een overzicht van alle methoden in de module `lists`.

Bij de prompt kan je eenvoudige uitdrukkingen ingeven en testen:

```
1> 1+2.
3
```

De bovenstaande uitdrukking berekent de som van 1 en 2. Onmiddellijk wordt het resultaat weergegeven. Je moet de uitdrukking altijd beëindigen met een punt, dit is de Erlang syntax.

```
2> 4/0.
** exception error: an error occurred
   when evaluating an arithmetic expression
     in operator  '/'/2
        called as 4 / 0
```

Als er fouten ontstaan, dan worden die gemeld. Je kan ook functies starten.

```
3> pwd().
/home/lrutten
ok
```

Een functieoproep bestaat altijd uit de naam van de functie gevolgd door ronde haken. Dit is net zoals in C, C++ en Java. Vergeet natuurlijk de punt niet. `pwd()` geeft de huidige directory terug.

```
4> math:cos(0.0).
1.0
```

In een aantal geval moet je de modulenaam meegeven. Hierboven zie je dat de `cos()` functie zich in de module `math` bevindt. De dubbele punt is het scheidingsteken. In de documentatie zal je de notatie `math:cos/1` tegenkomen. Deze schrijfwijze geeft aan dat er één parameter verwacht wordt.

Hoe je zelf functies in modules kan maken, zien we later. Eerst bekijken we welke datatypes Erlang kent.

1.3. Datatypes

1.3.1. Integer

Zoals alle talen kent Erlang gehele getallen. Er wordt een vaste bitgrootte gebruikt voor de opslag. Indien deze grootte overschreden wordt, wordt het getal als een bigint opgeslagen. In dit geval wordt er zoveel geheugen gebruikt als de grootte van het getal dat vraagt.

```
123               gewone integer
1234567899999999  bigint
```

Getallen in een andere basis gaat ook:

```
```

```
2#1010
8#377
```

Het eerste getal is binair, het tweede octaal (basis 8).

Karaktercodes krijg je zo:

```
$9
$A
$\n
```

Intern worden de karakters als integer opgeslagen.

1.3.2. Floats

Worden zo geschreven:

```
1e10
```

Erlang gebruikt 64 bit double precision floats.

De bewerkingen zijn +, -, * en /. Voor de deling kan je / en `div` gebruiken. Het laatste is voor gehele getallen. Voor de rest heb je `rem`.

1.3.3. Bitbewerkingen

Deze zijn:

```
bsl    1 bit naar links schuiven
bsr    1 bit naar rechts schuiven
band   en
bor    of
bxor   exclusieve of
bnot   niet
```

Elke van deze bewerkingen start met de letter b om aan te geven dat het bitbewerkingen zijn.

1.3.4. Binaries

Stellen een aantal unsigned 8 bit waarden voor:

```
<<1, 2, 3 >>
<<"hallo", 78, 98>>
```

Dit type heb je nodig als je low level bewerkingen op berichten wil. Als je zelf binaire berichten wil versturen via TCP of UDP kan je dit nodig hebben.

1.3.5. Atomen

Atomen zijn woorden die een bepaalde betekenis dragen als ze opduiken in uitdrukkingen. Atomen hebben dan ook geen waarde.

Ze hebben altijd als eerste teken een kleine letter. De rest mag groot, underscore, cijfer of @ zijn.

```
punt
```

```
vlak
n66
```

Bij spaties moet je single quotes gebruiken.

```
'Dit is een atom'
```

De maximale lengte is 255.

Deze woorden zijn verboden als atom:

```
after and andalso band begin
bnot bor bsl bsr bxor case catch cond div
end fun if let not of or orelse query
receive rem try when xor
```

Dit zijn sleutelwoorden van Erlang. Merk ook nog op dat het totale aantal atomen in een draaiende Erlang systeem beperkt is tot 1048576.

1.3.6. Tuples

Bij tuples associeer je een aantal waarden met elkaar. De plaats is bepalend voor de betekenis van een waarde binnen de tuple.

```
{4, 5}
{punt, 6, 7}
```

De eerste tuple zou een x en y kunnen voorstellen. De tweede tuple zou dat ook kunnen; hier wordt expliciet aangegeven dat het over een punt gaat. De atoom `punt` zorgt hiervoor. Een tuple varieert nooit in lengte. Als een tuple uit 4 elementen bestaat, dan zal dat gedurende de ganse levenduur van deze tuple zo blijven.

De volgende testen geven aan hoe je waarden uit een tuple kan halen of hoe je een nieuwe tuple maakt vanuit een bestaande tuple.

```
7> T={punt, 3, 4}.
{punt,3,4}
8> element(1, T).
punt
9> element(2, T).
3
10> element(3, T).
4
11> T2=setelement(2, T, 5).
{punt,5,4}
12> tuple_size(T2).
3
```

Commando 7 kent de waarde `{punt, 3, 4}` toe aan de variabele `T`. In Erlang moeten de variabelenamen in grote letters geschreven worden.

Commando's 8, 9 en 10 starten een functieoproep waarbij respectievelijk het eerste, tweede en derde element van de tuple `T` opgevraagd worden.

In commando 11 wordt de waarde van het tweede element vervangen door een 5. Hierbij wordt een nieuwe tuple teruggegeven waarin de nieuwe waarde is opgenomen. De oude tuple is niet gewijzigd

omdat dit in Erlang niet mogelijk is. Eens er een waarde aan een variabele is toegekend kan die variabele niet meer gewijzigd worden. Commando 13 toont een poging om `T2` een nieuwe waarde te gegeven. Aan de foutmelding zie je dat dit mislukt.

```
13> T2={punt, 8, 9}.
** exception error: no match of right hand side value {punt, 8, 9}
```

Omdat je geen variabele kan wijzigen moet je andere strategieën gebruiken om met wisselende gegevens te kunnen omgaan.

1.3.7. Lists

Lijsten kunnen meerdere elementen bevatten. Je gebruikte de rechte haken `[` en `]` om de lijst te vormen. Een lijst kan om het even welke waarde(n) bevatten.

Hier zijn enkele voorbeelden van lijstconstanten:

```
[]
[1, 2, 3]
```

Een element en een lijst samenvoegen doe je zo:

```
[1 | [2,3]] geeft [1, 2, 3]
```

In het bovenstaande voorbeeld wordt de rechte streep `|` gebruikt om het element en de bestaande lijst samen te voegen tot een nieuwe lijst.

Lijsten samenvoegen gaat ook:

```
[1,2] ++ [3,4] geeft [1, 2, 3, 4]
```

Er zijn BIF's (built in functions) die head en tail van een lijst leveren:

```
hd([1,2,3]) geeft 1
tl([1,2,3]) geeft [2,3]
length([1,2,3,4]) geeft 4
```

1.3.8. Strings

Strings worden intern naar lijsten omgezet.

```
"ABC" opgeslagen als [65, 66, 67]
```

1.3.9. Pids, Poorten en referenties

Een pid krijg je terug als je een proces maakt.

```
<0.35.0>
```

Een poort is zoals een pid maar je kan ermee met niet-Erlang software communiceren.

```
#Port<1.6>
```

Referenties zijn unieke labels. Je maakt ze met `make_ref()`.

```
#Ref<0.0.0.39>
```

1.3.10. Vergelijken

Doe je met:

```
==  rekenkundige gelijkheid
/=  rekenkundige ongelijkheid
<   kleiner dan
>   groter dan
=<  kleiner of gelijk aan
>=  groter of gelijk aan
```

Je hebt nog exacte gelijkheid en exacte ongelijkheid:

```
=:=
=/=
```

Hierbij worden de waarden als boom vergeleken.

Je kan ook samenstellingen maken:

```
and
or
not
```

Om te groeperen gebruik je de haken `(` en `)`. Voor short-circuit testen heb je:

```
andalso
orelse
```

Je kan alles met alles vergelijken:

```
number < atom < reference < fun
< port < pid < tuple < list < bit string
```

1.4. Modules

Modules zijn de oplossing in Erlang om functies te groeperen in één broncodebestand. Je moet dat bestand dezelfde naam geven als de modulenaam. De module `geo` zit dus in het bestand `geo.erl`.

Als je een functie wil oproepen die zich in een andere module bevindt dan deze van waaruit de oproep start, dan moet je de modulenaam met een dubbele punt laten voorafgaan aan de functienaam.

Dit is een voorbeeld van een functieoproep met modulenaam:

```
geo:start(5).
```

Dit is een zeer eenvoudige module:

```
%% This is a simple Erlang module
-module(transcendent).
-export([pi/0]).

pi() ->
  3.14.
```

In deze module wordt de naam vastgelegd met `-module()` en er wordt met `-export()` aangegeven welke functies publiek worden; in andere woorden, welke functies kunnen van buiten de module opgeroepen worden. De slash en het getal geven aan hoeveel parameters er met de betrokken functies moeten meegegeven worden.

Dit is nog een module.

```
-module(vb1).
-export([dubbel/1, start/0]).
-vsn([1.0]).
-author("Leo Rutten").

dubbel(X) ->
   2 * X.

start() ->
   G = dubbel(5),
   io:format("getal is ~B~n", [G]).
```

Deze module bevat twee functies `dubbel/1` en `start/0`. In de laatste functie wordt er uitvoer gedaan met `io:format/2`. Deze laatste werkt ongeveer zoals `printf()` in C. Met `~B` wordt een getal weergegeven en `~n` is een nieuwe regel. De benodigde argumenten moet je met een lijst meegeven.

En dit is de `Makefile` om te compileren en uit te voeren. Deze `Makefile` was een poging om vanuit commandolijn een Erlang module te compileren en te starten. In praktijk worden andere middelen gebruikt om dit te realiseren.

```
all: run

vb1.beam: vb1.erl
        erlc vb1.erl

run: vb1.beam
        erl -noshell -s vb1 start

clean:
        rm -vf *.dump
        rm -vf *.beam
        rm -vf *~
```

In elk geval toont de `Makefile` dat je een Erlang module kan compileren met `erlc`. Dit is de Erlang compiler. Het resultaat is een `.beam` bestand. Je ziet dat de werkwijze een beetje is zoals in Java. Om de module te starten wordt `erl` gestart.

1.5. Variabelen en patronen

De namen van variabelen beginnen altijd met een hoofdletter. Een variabele mag zomaar in gebruik genomen worden zonder declaratie. Je kan dus het volgende schrijven:

```
G = 2*H.
```

Dit betekent dat de nieuwe variabele `G` een waarde krijgt die later kan gebruikt worden. Er wordt geen type vermeld bij het eerste gebruik. En als je vergeet om de variabele te gebruiken verderop in het programma, geeft de compiler een fout.

Als je expliciet wilt aangeven dat je een variabele (bijna altijd is dit een parameter) niet wilt gebruiken, dan kan je een dummy variabele gebruiken. Die wordt als een underscore (_) geschreven. Je kan ook een variabele laten starten met een underscore, bijvoorbeeld `_Aantal`, dit geeft ook een dummy variabele.

Het `=` teken kennen we in de meeste programmeertalen als het teken voor de toekenning maar in Erlang werkt het ook als patroonherkenning. In de volgende toekenning wordt dit getoond.

```
1> {A,B,C}={1,2,3}.
{1,2,3}
```

In dit voorbeeld wordt er getracht om de linker- en rechteruitdrukkingen gelijk te stellen aan elkaar. Beide uitdrukkingen zijn maar gelijk aan elkaar als `A 1` is, `B 2` en `C 3`. Deze toekenningen gebeuren ook effectief. Na het uitvoeren van het `=` teken zijn `A`, `B` en `C` gebonden aan een waarde en kunnen ze nooit aan een andere waarde gebonden worden. Men zegt ook kortweg dat `A`, `B` en `C` gebonden zijn.

Met patroonherkenning ga je niet alleen maar na of je een bepaalde structuur hebt in een term maar je haalt er ook de afzonderlijke delen uit. Als er geen overeenkomst is, komt er een foutmelding.

```
2> {A,B,C,D}={1,2,3}.
** exception error: no match of right hand side value {1,2,3}
```

Hier is nog een voorbeeld met een lijst.

```
8> [H|Rest] = [een,twee,drie,vier].
[een,twee,drie,vier]
9> b().
H = een
Rest = [twee,drie,vier]
ok
```

De verticale streep `|` wordt hier gebruikt om een lijst op te splitsen in het eerste element en de rest van de lijst. Deze laatste is ook een lijst.

Patroonherkenning wordt ook toegepast in de definitie van functies (de naam van de functie gevolgd door de parameterlijst tussen ronde haken), bij de `if`, `case`, `try` en `receive`.

1.6. Functies

Functies zijn de elementaire delen van Erlang waarmee je de functionaliteit van een programma kan opbouwen.

1.6.1. Eenvoudige functies

Deze functie hebben we al gezien maar nu zijn er twee acties.

```
dubbel(X) ->
   Y = 2*X,
```

```
   Y.
```

Dit kleine voorbeeld toont hoe je eenvoudige functies moet schrijven. Het lastige hieraan is niet het bedenken wat de functie precies moet doen maar wel de syntax. Erlang heeft een syntax die overgenomen is van de taal Prolog en in deze taal spelen de punten, komma's en puntkomma's een belangrijke rol. Het moeilijke is nu dat je die leestekens correct moet schrijven.

De functie start met de naam `dubbel` gevolgd door ronde haken met daartussen de kommalijst van de parameters. In dit voorbeeld is er maar één parameter, namelijk `X`. De parameter heeft geen type omdat in Erlang variabelen nooit een type krijgen. Daarna volgt de pijl `->` met daarna de body. Deze bestaat uit twee acties of uitdrukkingen: `Y = 2*X` en `Y`. Omdat deze twee uitdrukkingen allebei moeten uitgevoerd worden, moeten ze gescheiden worden door een komma. De komma in Erlang heeft de rol gekregen van * en * operator. De tweede uitdrukking `Y` staat er om te bepalen wat er moet teruggegeven worden door de functie. De waarde van de laatste uitdrukking is altijd de returnwaarde van een functie. Tot slot moet de hele definitie afgesloten worden met een punt.

Het eenvoudige voorbeeld in deze sectie is een functie met slechts één definitie. Je kan ook ook functies maken met meerdere definities.

1.6.2. Functies met meerdere definities

Elke functie kan bestaan uit meerdere definities. Elke definitie heeft een naam met een opsomming van de parameters. Dit wordt gevolgd door `->` waarna de body van de functie volgt. In het volgende voorbeeld wordt dit getoond.

```
-module(figuren).
-export([omtrek/1]).

omtrek({vierkant, Zijde}) ->
   4*Zijde;
omtrek({rechthoek, Breedte, Hoogte}) ->
   2*(Breedte + Hoogte);
omtrek({cirkel, Straal}) ->
   2*math:pi()*Straal.
```

De bovenstaande module `figuren` exporteert één functie `omtrek`. Deze functie heeft een ariteit (* arity *) van 1. Dit wil zeggen dat deze functie één parameter verwacht. De functie heeft drie definities. Elke definitie heeft zijn eigen parameterlijst. Om het onderscheid te kunnen maken in welke soort paremeter er meegegeven wordt, wordt er gebruik gemaakt van patroonherkenning. De drie definities worden gescheiden door twee puntkomma's. In erlang fungeert de puntkomma als een of operator. Dit is ook de logische betekenis van het gebruik van de puntkomma in deze context: kies bij de oproep van de functie voor één van de drie definities. De definitie die overeen komt met het patroon van de doorgegeven parameter wordt gestart.

Dit verklaart waarom na de eerste en tweede definitie een puntkomma wordt geschreven. Na de derde definitie staat zoals verwacht een punt.

Vermits de functie met als zijn definities in een apart tekstbestand staat, kan je het compileren en testen vanuit `erl`. Dat doe je zo:

```
1> c(figuren).
{ok,figuren}
2> figuren:omtrek({cirkel,2}).
12.566370614359172
3> figuren:omtrek({rechthoek,3,2}).
10
4> figuren:omtrek({vierkant,3}).
```

```
12
```

Met `c(figuren).` wordt het bestand `figuren.erl` gecompileerd. Het resultaat is `figuren.beam`. Deze code is na de compilatie automatisch geladen. Je mag bij het oproepen van de functie `figuren:omtrek/1` de modulenaam vergeten.

De werkwijze om een reeks functies in een gezamenlijke module te plaatsen en dan te testen met `erl` is heel handig. Van zodra een project groter wordt en er sprake is van meerdere modules wordt deze methode omslachtig; je moet dan een andere werkwijze kiezen.

1.6.3. Functies met guards

In functies kan je per definitie een beperking (* guard *) instellen. Dit betekent dat naast de patroonherkenning ook de guard als beperking geldt.

```
oud_genoeg(X) when X >= 16 -> true;
oud_genoeg(_) -> false.
```

In het bovenstaande voorbeeld is iedereen boven of gelijk aan 16 oud genoeg. Merk op dat in de tweede definitie de parameter als een _ is geschreven. Dit betekent dat je om het even welke waarde kan meegeven en dat dan deze waarde nooit gebruikt wordt. Dit effect is wel gebaseerd op de gekozen volgorde van de definities. Als de eerste definitie niet geldig is, wordt automatisch de tweede getest. Als de eerste definitie wel geldig, wordt de tweede nooit getest. De definities worden dus getest in de volgorde dat ze geschreven zijn.

Meerdere guards mogen ook:

```
oud_genoeg(X) when X >= 16, X =< 116 -> true;
oud_genoeg(_) -> false.
```

De komma werkt hier als een `andalso` samenstelling. En de puntkomma werkt hier als een `orelse` samenstelling.

```
niet_oud_genoeg(X) when X < 16; X > 116 -> true;
niet_oud_genoeg(_) -> false.
```

Je kan bij de koppeling van guards zowel komma's als puntkomma's gebruiken.

Niet alle uitdrukkingen mogen als guard gebruikt worden. Dit is een overzicht van wat allemaal mag.

- `true`
- constanten en gebonden variabelen gelden als `false`
- vergelijking van termen
- rekenkundige en booleaanse uitdrukkingen
- typetesten zoals `is_atom/1`, `is_constant/1`, `is_integer/1`, `is_float/1`, `is_number/1`, `is_reference/1`, `is_port/1`, `is_pid/1`, `is_function/1`, `is_tuple/1`, `is_record/2`, `is_list/1` en `is_binary/1`
- andere BIF's: `abs(Integer | Float)`, `float(Term)`, `trunc(Integer|Float)`, `round(Integer|Float)`, `size(Tuple, Binary)`, `element(N,Tuple)`, `hd(List)`, `tl(List)`, `length(List)`, `self()`, `node()` en `node(Pid|Ref|Port)`

-module(reken). -export([ggd/2]).

Tot slot is er nog een voorbeeld met een guard. De functie `ggd/2` berekent de grootste gemene deler van twee gehele getallen A en B met het algoritme van Euclides. Dit algoritme is recursief geformuleerd. Dit leidt uiteraard tot recursie in de oplossing. Wel moet A groter zijn dan B. De guard wordt gebruikt om dit te bewaken. De eerste definitie verwisselt de twee getallen indien nodig. Met de guard wordt getest of dit nodig is.

```
ggd(A, B) when B > A ->
   ggd(B, A);
ggd(A, 0) ->
   A;
ggd(A, B) ->
   ggd(B, A rem B).
```

De werking van het algoritme is als volgt:

1. neem twee getallen A en B zodat B kleiner is dan A
2. indien B nul is, is A de ggd
3. indien B niet nul is, vervang het paar A,B door B,A rem B
4. herhaal stap 3 tot B nul wordt

1.6.4. Ingebouwde functies

Deze bevinden zich in de `erlang` module. Met de TAB toets kan je ze zichtbaar maken in `erl`. De onderstaande lijst is niet compleet. Je kan het commando zelf testen om de complete lijst te zien.

```
1> erlang:
'!'/2                             '*'/2
'+'/1                             '+'/2
'++'/2                            '-'/1
'-'/2                             '--'/2
'/'/2                             '/='/2
'<'/2                             '=/='/2
'=:='/2                           '=<'/2
'=='/2                            '>'/2
'>='/2                            'and'/2
'band'/2                          'bnot'/1
'bor'/2                           'bsl'/2
'bsr'/2                           'bxor'/2
'div'/2                           'not'/1
'or'/2                            'rem'/2
'xor'/2                           abs/1
adler32/1                         adler32/2
adler32_combine/3                 append/2
append_element/2                  apply/2
apply/3                           atom_to_binary/2
atom_to_list/1                    await_proc_exit/3
binary_part/2                     binary_part/3
binary_to_atom/2                  binary_to_existing_atom/2
binary_to_list/1                  binary_to_list/3
binary_to_term/1                  binary_to_term/2
...
```

De handleiding van de `erlang` module kan je zo opvragen.

```
erl -man erlang
```

1.7. `Case` en `if`

Bij de `if` hoort een voorwaarde en een actie. Een `else` bestaat niet; je moet een `true` gebruiken. Bij de voorwaarden mag je een komma of puntkomma gebruiken. En de puntkomma scheidt de verschillende alternatieven.

```
doe_test(X) ->
   if X>0  -> io:format("positief~n");
      true -> io:format("niet positief~n")
   end.
```

Dit is een voorbeeld met een `case`.

```
insert(X,[]) ->
   [X];
insert(X,Set) ->
   case lists:member(X,Set) of
      true  -> Set;
      false -> [X|Set]
   end.
```

Het bovenstaande voorbeeld is de functie `insert` die een element aan een lijst toevoegt. De nieuwe lijst wordt als resultaat teruggegeven. De oude lijst wordt nooit gewijzigd. Dit zou in Erlang ook niet gaan.

De functie heeft twee definities. De eerste legt vast dat een element bijvoegen in een lege lijst een lijst oplevert met alleen maar dat ene element. Je ziet hier dat er aan patroonherkenning wordt gedaan met een lege lijst `[]` als parameter.

De tweede definitie gaat na of het nieuw element al in de lijst zit. Indien ja, wordt de oude lijst teruggegeven. Indien niet, wordt een nieuwe lijst samengesteld en teruggegeven, bestaande uit het nieuwe element en de oude lijst. Dit wordt als `[X|Set]` geschreven.

Je kan bij de waarden waarop getest worden binnen de `case` ook guards bijvoegen. Dat wordt in het volgende voorbeeld gedemonstreerd.

```
beach(Temperature) ->
   case Temperature of
      {celsius, N} when N >= 20, N =< 45 ->
         'favorable';
      {kelvin, N} when N >= 293, N =< 318 ->
         'scientifically favorable';
      {fahrenheit, N} when N >= 68, N =< 113 ->
         'favorable in the US';
       _ ->
          'avoid beach'
   end.
```

De `if` en de `case` zijn uitdrukkingen die zelf ook een waarde teruggegeven. Dit past in de filosofie van Erlang waarbij elke uitdrukking een waarde voorstelt. Hierdoor kan je voorwaardelijke toekenningen schrijven met zowel de `if` als de `case`

```
B = if
      A>100 -> 100;
      true  -> A
    end.
```

Dit is een voorbeeld met de `case`.

```
   Zijden =
      case Figuur of
```

```
        vierkant  -> 4;
        rechthoek -> 4;
        driehoek  -> 3;
        zeshoek   -> 6;
        _ -> onbepaald
      end.
```

1.8. Staartrecursie

Het zal nu wel duidelijk zijn dat er in Erlang geen constructies bestaan om herhalingen te maken. Je kan niet anders dan recursie gebruiken. Dit is wel even wennen omdat je dan voor elke herhaling een nieuwe functie moet maken.

Wanneer in programma's die lang draaien herhalingen worden gebruikt waarvan je op voorhand weet dat je eigenlijk nooit stoppen, zal die recursie een probleem zijn. Dit wordt veroorzaakt doordat er bij elk oproep van de functie een nieuw stackframe wordt aangemaakt. Dit stackframe bevat alle parameters, lokale variabelen en terugkeeradres. Bij een oneindige herhaling krijg je dus een oneindig grote stack. En dit kan natuurlijk niet; de stackgrootte is beperkt.

Om deze problemen te vermijden moet je staartrecursie gebruiken. Deze vorm van recursie is zodanig dat er geen code meer voorkomt na de oproep van de eigen functie. We bekijken een aantal voorbeelden zonder en met staartrecursie.

Het eerste voorbeeld gebruikt geen staartrecursie.

```
lengte([]) ->
   0;
lengte([_X | Rest]) ->
   1 + lengte(Rest).
```

De bovenstaande functie heeft geen staartrecursie omdat plusbewerking in de laatste regel pas gebeurt na het verkrijgen van het resultaat van de oproep van de eigen functie. Je moet daarom dit voorbeeld herschrijven zodat het wel staartrecursie toepast.

Dit is het herwerkte voorbeeld.

```
lengtetail(L) ->
   lengtetail(L, 0).

lengtetail([], Acc) ->
   Acc;
lengtetail([_X | Rest], Acc) ->
   L = 1 + Acc,
   lengtetail(Rest, L).
```

Hier wordt gewerkt met een accumulatorvariabele. Deze variabele houdt het voorlopige resultaat bij. `lengtetail/1` roept onmiddellijk `lengtetail/2` op. De tweede parameter is nul; dit is de startwaarde voor de accumulator. `lengtetail/2` heeft twee definities. De eerste verwerkt een lege lijst: in dit geval wordt de accumulator teruggegeven. De tweede definitie telt 1 op bij de accumulator en roept zichzelf met de rest van de lijst en de verhoogde accumulator.

Ter vergelijking is hier een C voorbeeld dat driemaal is uitgewerkt. De eerste versie gebruikt geen staartrecursie en de tweede wel. De derde versie is de versie die verkregen wordt als de compiler de staartrecursie optimaliseert tot een iteratie. Deze derde versie toont hoe uiteindelijk de staartrecursie vervangen kan worden door een iteratie zodat er zeker geen stackoverflow kan gebeuren.

Dit is de versie zonder staartrecursie.

```
int som(int n)
{
   if (n == 0)
   {
      return 0;
   }
   else
   {
      return n + som(n - 1);
   }
}
```

In het bovenstaande voorbeeld is niet `som(n - 1)` de laatste bewerking maar wel is de `+` bewerking de laatste. Daardoor is er geen staartrecursie.

Dit is de versie met staartrecursie.

```
int som_staart(int n, int s)
{
   if (n == 0)
   {
      return s;
   }
   else
   {
      return som_staart(n - 1, s + n);
   }
}
```

Dit is de iteratieve versie.

```
int som_iteratief(int n, int s)
{
start:
   if (n == 0)
   {
      return s;
   }
   else
   {
      s = s + n;
      n = n - 1;
      goto start;
   }
}
```

Het is niet zo dat er altijd staartrecursie moet gebruikt worden maar staartrecursie levert het voordeel dat er kan geoptimaliseerd worden. Bij recursie die dient om oneindige herhalingen te maken, moet je wel altijd staartrecursie gebruiken.

1.9. Anonieme functies

In functionele programmeertalen (en ook in C: pointers naar functies, bijvoorbeeld `int (*pf)(int a)`) bestaat de mogelijkheid om functies ook te beschouwen als data. Hierdoor kan je een functie die opgeslagen is in een variabele doorgeven aan een andere functie. Met deze techniek kan je de werking van een functie algemeen maken door alle specifieke bewerkingen te laten uitvoeren door een functie

die als parameter wordt doorgegeven. In de klassieke OO talen werd deze techniek niet zo dikwijls gebruikt omdat de combinatie van erfenis en virtuele functie een functionele veralgemening mogelijk maakte. Recent zijn in Java en C++ de mogelijkheid van anonieme functies (lambda's) bijgevoegd. Blijkbaar is er recent een invloed van functionele talen op de OO talen.

Een anonieme functie schrijf je zo:

```
4> F = fun(A, B) -> A + B end.
#Fun<erl_eval.12.90072148>
5> F(5,6).
11
```

Met het sleutelwoord `fun` start de definitie van een functie zonder naam. Daarna schrijf je de gebruikelijke syntax van een functie. Je moet afsluiten met `end`. De definitie van deze functie, die geen naam heeft, kan je bijhouden in een variabele. Via deze variabele kan je de functie starten: `F(5,6).`.

Hier is een voorbeeld waarin de anonieme functie als parameter wordt meegegeven.

```
map(_F, []) ->
   [];
map(F, [X|Rest]) ->
   [F(X) | map(F, Rest)].
```

Dit voorbeeld hoef je niet in te geven; ze bestaat al in de module `lists`. De functie `map/2` verwacht als eerste parameter een anonieme functie en als tweede parameter een lijst. De anomieme functie wordt voor elk element van de lijst gestart. Met de resultaten van elke oproep wordt een nieuwe lijst samengesteld.

```
7> lists:map(fun(X) -> 2*X end, [1,2,3]).
[2,4,6]
```

Je ziet aan het resultaat dat de nieuwe lijst bestaat uit elementen die het dubbel zijn van de elementen van de oude lijst.

Dit zijn de functies in de `lists` module die een anonieme functie als parameter verwachten.

```
map/2
filter/2
foldr/3
foldl/3
all/2
any/2
dropwhile/2
takewhile/2
partition/2
flatten/1
flatlength/1
flatmap/2
merge/1
nth/2
nthtail/2
split/2
```

Hier is nog een voorbeeld. `partition/2` splitst een lijst in twee nieuwe lijsten: de eerste lijst bevat alle elementen waarvoor de anonieme functie een true teruggeeft en een tweede lijst die alle overige elementen bevat.

```
3> lists:partition(fun(X)->X rem 2 == 0 end, [1,2,3,4]).
{[2,4],[1,3]}
```

Een uitgebreide uitleg over anonieme functies vind je bij http://www.erlang.org/doc/programming_examples/funs.html

1.10. List comprehensions

Dit is een verkorte schrijfwijze om lijsten op te bouwen. Met deze controlestructuur kan je doorheen de elementen van één of meerdere lijsten lopen. Het resultaat van een list comprehension is altijd een nieuwe lijst. Door het gebruik van de list comprehension hoef je zelf geen recursie te schrijven. Niet elk probleem waarin een herhaling voorkomt kan met een list comprehension opgelost worden. In een aantal gevallen moet je toch zelf een recursieve oplossing ontwerpen.

Hier zijn enkele voorbeelden. Het eerste voorbeeld maakt een lijst waarin elk getal het dubbele is van het overeenkomstige getal in de originele lijst.

```
4>[2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

De list comprehension wordt als een lijst geschreven met daarin de dubbele verticale streep `||` als scheidingsteken. Rechts van de dubbele streep staat in dit voorbeeld de generator `N <- [1,2,3,4]`: deze notatie zorgt ervoor dat de variabele `N` telkens is gebonden met één van de waarden uit de lijst. Voor elke waarde wordt een element toegevoegd aan de nieuwe lijst. De uitdrukking die hiervoor gebruikt wordt, staat links van de dubbele streep. In dit geval is dat `2*N`.

Naast een generator kan je rechts van de dubbele streep ook een filter vermelden. De komma werkt hier als een en-functie. De filter `N rem 2 =:= 0` zorgt ervoor dat er alleen maar getallen toegevoegd worden die even zijn.

```
5>[2*N || N <- [1,2,3,4,5,6,7], N rem 2 =:= 0].
[4,8,12]
```

De elementen van de lijst mogen uiteraard ook andere waarden bevatten dan gehele getallen. In het volgende voorbeeld zijn het tuples. Het resultaat is een lijst van tuples waarbij iedere tuple bestaat uit een getal en zijn deler. De tuples waarvoor de deelbaarheid niet opgaat, zijn eruit gefilterd.

```
6>[{A,B}||{A,B}<-[{7,3}, {6,3}, {8,4}, {9,4}],A rem B =:= 0].
[{6,3},{8,4}]
```

Het volgende voorbeeld toont dat je mee dan één generator mag vermelden. Het totale aantal combinaties krijg je door het product te maken van de afzonderlijke aantallen van de generatoren. In dit voorbeeld worden er 9 tuples gegenereerd.

```
7>[{X,Y}||X<-[1, 2, 3],Y<-[4,5,6]].
[{1,4},{1,5},{1,6},{2,4},{2,5},{2,6},{3,4},{3,5},{3,6}]
```

Tot slot is hier nog een voorbeeld waarin getoond wordt dat je meerdere generatoren en filters kan combineren.

```
11> [{H,T,E} || H<-lists:seq(1,9), T<-lists:seq(0,9),E<-lists:seq(0,9),
                100*H+10*T+E =:= H*H*H+T*T*T+E*E*E].
```

```
[{1,5,3},{3,7,0},{3,7,1},{4,0,7}]
```

Dit voorbeeld doorloopt alle honderdtallen, tientallen en eenheden van de getallen van 100 tot en met 999. Alleen de getallen waarbij de som van de 3de macht van de cijfers gelijk is aan het getal zelf blijven behouden. Dit blijken de getallen 153, 370, 371 en 407 te zijn.

Voor eenvoudige lijstbewerkingen zijn de list comprehensions een goede oplossing. Van zodra we een accumulator willen gebruiken of de herhaling moet bij een zekere voorwaarde stoppen, zijn list comprehensions niet geschikt en moeten we zelf de recursie schrijven.

1.11. Datastructuren

Naast lijsten en tuples kent Erlang ook een aantal oplossingen die geschikt zijn voor een complexere dataopslag. Een aantal van deze oplossingen worden hier besproken. De eerste oplossing is de record. Dit is een elegantere schrijfwijze ter vervanging van de tuple. Het laat je toe om de elementen van een tuple via een naam te bereiken. De tweede oplossing is de map. Dit is een recente toevoeging. Zowel records als maps zijn uitbreidingen van de taal.

Verder zijn er nog een hele reeks oplossingen die als meegeleverde modules kunnen gebruikt worden. Alleen de modules `proplist`, `orddict` en `gb_trees` worden hier kort besproken.

1.11.1. Records

Records bieden de mogelijkheid om de elementen van een tuple via een naam op te halen. Intern worden records als tuples bijgehouden.

Een record moet je vooraf definiëren.

```
-record(punt, {x=0, y=0}).
```

Deze regel zegt dat de naam van de record `punt` is en dat er twee velden met namen `x` en `y` zijn. Wanneer er een nieuwe record gemaakt wordt, zijn de startwaarden 0 als de initialisatie ontbreekt. Dit betekent dat in het volgende fragment

```
P1 = #punt{},
P2 = #punt{x=6, y=7},
```

P1 het punt 0, 0 voorstelt en P2 het punt 6, 7. Je kan op records ook patroonherkenning toepassen.

```
#punt{x=X, y=Y} = P2,
```

Dit patroon lukt enkel als `P2` een `punt` record als waarde heeft. Indien dat zo is, krijgt `X` de x-waarde en `Y` de y-waarde. Uiteraard kan deze patroonherkenning ook toegepast worden in de parameterlijst van een functie.

Hier is een voorbeeld dat met een record werkt.

```
-record(punt, {x=0, y=0}).

doe_test(P = #punt{}) ->
   io:format("P is een punt~n"),
   io:format("   x is ~p~n", [P#punt.x]),
   io:format("   y is ~p~n", [P#punt.y]);
doe_test(P) ->
```

```
   io:format("P is geen punt~n").

P1 = #punt{x=5, y=7},
io:format("P1 ~p~n", [P1]),
doe_test(P1),
doe_test(56),
```

Je ziet dat er twee definities bestaan van `doe_test/1`. De eerste verwacht een `punt` record en de tweede om het even wat. Met de notaties `P#punt.x` en `P#punt.y` kan je respectievelijk de x-waarde en de y-waarde van `P` ophalen.

In een aantal gevallen wordt een recorddefinitie in meerdere modules gebruikt. Je kan de recorddefinitie dan best in een apart bestand met extensie `.hrl` plaatsen en dan in elke module een include plaatsen om de recorddefinitie op te halen. Deze werkwijze is te vergelijken met die van de taal C.

Dit is `punt.hrl`:

```
-record(punt, {x=0, y=0}).
```

En dit is `punten.erl`:

```
-module(punten).
-export([new/0, test/0]).

-include("punt.hrl").

new() ->
   #punt{}.


test() ->
   R  = #punt{x=4,  y=5},
   _R2 = #punt{x=40, y=50},
   #punt{x=X, y=Y} = R,
   io:format("punt is ~p ~p~n", [X, Y]).
```

Door de definitie van de record apart te zetten, kan je die in meerdere modules gebruiken.

1.11.2. Maps

Dit is een recente toevoeging aan Erlang. Door een uitbreiding van de syntax kan je in Erlang key/value associaties opslaan in een map datastructuur.

```
13> M=#{een => 1, twee => 2}.
#{een => 1,twee => 2}
```

Het bovenstaande fragment maak een nieuwe map met key/value paren `een => 1` en `twee => 2`. Als key mag je zeker geen variabelen of resultaten van bewerkingen gebruiken. Voor de value mag dat wel voor zover de variabelen gebonden zijn. De pijl `=>` betekent in deze context het maken van een nieuw key/value paar.

Hier zijn nog een aantal stappen waarbij de map extra key/values krijgt.

```
14> M2 = M#{drie => 3}.
```

```
#{drie => 3,een => 1,twee => 2}
15> M3 = M2#{vier => 4,acht => 8}.
#{acht => 8,drie => 3,een => 1,twee => 2,vier => 4}
```

Je ziet aan het resultaat dat de key/values geordend volgens key worden bijgehouden.

Met patroonherkenning kan je de waarde van een bepaalde key opvragen. In het volgende fragment komt de waarde van key `twee` in de variabele `Value` terecht.

```
16> #{twee := Value} = M4.
#{acht => 8,drie => 3,een => 1,twee => 2,vier => 4}
17> Value.
2
```

Je ziet dat voor dit patroon geen `=>` maar wel `:=` als scheidingsteken wordt gebruikt. Je moet dit teken ook gebruiken als je de waarde van een bestaande key wil updaten.

```
27> M5 = M4#{vier := 44}.
#{acht => 8,drie => 3,een => 1,twee => 2,vier => 44}
```

Uiteraard krijg je bij de update van een bestaande map een nieuwe map.

Er bestaan ook extra functies voor het bewerken van maps. Ze staan in de module `maps`.

```
erl -man maps
```

Dit is de aankondiging van de komst van maps in Erlang:

http://joearms.github.io/2014/02/01/big-changes-to-erlang.html

Deze maps zijn maar mogelijk in Erlang R17. Wie met een oudere versie van Erlang werkt, moet andere oplossingen kiezen.

1.11.3. Proplist

Een `proplist` is een lijst van key/value tuples. Dit is een oudere vorm van key/value opslag. Je kan deze vorm in de toekomst nog gebruiken voor bijvoorbeeld het configureren van programma's.

Qua snelheid zal een proplist niet kunnen concurreren met de nieuwe map. Dit wordt veroorzaakt door de gelinkte lijst implementatie die inherent traag is.

```
L = proplists:delete(2, [{1,"een"}, {2,"twee"}, {3,"drie"}]),
io:format("L is ~p~n", [L]),
io:format("2 in ~p~n", [proplists:is_defined(2, L)]),
io:format("3 in ~p~n", [proplists:is_defined(3, L)]).
```

Verder bestaan er nog `dict`, `gb_trees`, `ordsets`, `sets`, `gb_sets` en `sofs`. Voor grafen bestaan de modules `digraph` en `digraph_utils`. Er zijn ook nog de `queue` en de `array` modules.

1.12. Foutafhandeling

Uiteraard zijn er naast de compilatiefouten ook runtimefouten mogelijk. In een taal zoals Erlang zal het zwaartepunt op de runtimefouten liggen omdat de compiler geen typefouten controleert. De compiler

is immers niet in staat om te controleren of de argumenten die bij een functieoproep meegegeven worden, wel van de juiste soort zijn. Deze types zijn pas bekend van zodra het programma loopt.

De compiler doet geen typecontroles; met het commando `dialyzer` kan je die controles wel uitvoeren maar dat wordt niet uitgelegd in deze cursustekst.

1.12.1. Runtime fouten

Het gevolg van een runtime fout is dat het proces crasht. Dit zijn de mogelijke runtime fouten.

- `function_clause` de functie heeft geen definitie voor de gegeven argumenten.
- `case_clause` de geteste waarde in `case` komt niet als patroon voor.
- `if_clause` er is geen `true` tak gevonden voor een bepaalde waarde in een `if`.
- `badarg` een verkeerd argument bij functieoproep.
- `undef` de op te roepen functie bestaat niet.
- `badarith` verkeerd argument in een rekenkundige uitdrukking.
- `badfun` er is iets mis met de meegegeven anonieme functie.
- `badarity` het aantal meegegeven argumenten klopt niet.

Je kan een proces doen stilleggen met:

```
erlang:error(badarid).
```

Hiermee veroorzaak je zelf een runtime fout.

Eigenlijk behoort een runtime tot de exceptionklasse `error`. Er zijn 3 soorten exceptions: `error`, `throw` en `exit`. De laatste twee klassen noemt men ook gegenereerde exceptions.

1.12.2. Exits

Dit is een expliciete manier om een proces stil te leggen. `exit/1` is een interne exit, dit betekent dat een proces zichzelf stillegt.

```
exit(tehogetemperatuur).
```

en `exit/2` zou je kunnen gebruiken om een ander dan het eigen proces stil te leggen.

1.12.3. Throws met `try` en `catch`

Deze constructie is ongeveer zoals Java en C++. Je schrijft het woord `try` gevolgd door de uitdrukking die onder de controle van de `try` zal lopen. Dan volgt het woord `of` gevolgd door één of meerdere patronen om het resultaat van de uitdrukking op te vangen. Na het woord `catch` staan er patronen om fouten op te vangen.

De volgende module laat toe om `try` en `catch` te testen.

```
-module(fouten).
-export([test/1]).

test(Fu) ->
   try Fu() of
      R -> R
   catch
      error:Fout -> {runtimefout, Fout};
      throw:Fout -> {throwfout, Fout}
```

```
   end.
```

In de `catch` worden mogelijke runtime en throw fouten opgevangen. Dit is de eigenlijke test:

```
1> fouten:test(fun() -> 1/0 end).
{runtimefout,badarith}
2> fouten:test(fun() -> throw(mijnfout) end).
{throwfout,mijnfout}
3> fouten:test(fun()->exit(tewarm) end).
** exception exit: tewarm
     in function  fouten:test/1 (fouten.erl, line 5)
```

In de eerste test wordt een rekenfout veroorzaakt. Dit levert een `badarith` op. In de tweede test wordt zelf een fout veroorzaakt. Deze fout komt in het tweede patroon van de `catch` terecht. En de derde test laat het proces beëindigen. Deze fout kan niet opgevangen worden omdat het proces onmiddellijk bij `exit/1` stopt en het `catch` deel niet meer uitvoert.

Tot slot vermelden we nog dat er een alternatieve manier bestaat om exceptions op te vangen. Je schrijft dan een `catch` zonder `try`. Deze manier wordt echter afgeraden.

1.13. Processen

Vanaf zijn ontstaan kent Erlang processen als onderdeel van de taal. Door processen te beschouwen als een ingebouwd kenmerk van de taal, zal je gauw een programmeerstijl ontwikkelen waarin je veel processen creëert. Processen in Erlang hebben een zeer beperkte toestand die bijgevolg maar weinig geheugen in beslag neemt. Hierdoor is het mogelijk om systemen te bouwen waarin zeer veel processen tegelijkertijd tot leven komen. Verder is het zo dat processen nooit gemeenschappelijk gegevens delen; er is geen shared memory tussen de processen zoals dat bij Java threads wel het geval is. Dat betekent dan ook dat synchronisatie met semaforen en locks compleet ontbreekt in Erlang.

Om nu toch Erlang processen met elkaar te laten samenwerken, worden er berichten tussen de processen verstuurd. Elk proces krijgt vanaf zijn ontstaan een berichtenwachtrij. In deze wachtrij komen berichten aan die door andere processen worden verstuurd. Heel de afhandeling van het berichtenverkeer verloopt asynchroon: een proces kan een bericht versturen en hoeft daarop niet te wachten. Een ander proces ontvangt dit bericht en kiest zelf wanneer de afhandeling van dit bericht wordt uitgevoerd. Een proces kan slechts één ontvangen bericht per keer afhandelen. Hierdoor worden interne consistentieproblemen vermeden.

Berichten kunnen opgebouwd worden met alle datatypes en -structuren die Erlang ter beschikking stelt. Door de patroonherkenning is het heel eenvoudig om ontvangen berichten uit te pakken. Erlang ondersteunt ook berichtenverkeer tussen processen die zich in verschillende nodes in verschillende platformen bevinden. Deze gedistribueerde werking is volledig transparant voor de processen. Je kan aan de broncode van een proces niet zien of een bericht lokaal of gedistribueerd verzonden/ontvangen is. Dit maakt het ontwerp van gedistribueerde systeem heel wat eenvoudiger. Dit kenmerk is ook één van de redenen geweest waarom Erlang ontstaan is.

De Erlang virtuele machine heeft zijn eigen implementatie van multitasking. Dit betekent dat er niet gesteund wordt op proces of thread API's van de betrokken besturingssystemen. Hierdoor is een zeer lichtgewicht procesimplementatie mogelijk. Er zijn al met succes experimenten uitgevoerd om de Erlang virtuele machine rechtstreeks bovenop de hardware te laten draaien. In dat geval valt het besturingssysteem weg. Voor bepaalde vormen van virtualisatie kan dit experiment interessant zijn.

Het eerste voorbeeld toont hoe één of meerdere processen kan starten met een anonieme functie.

```
1> F = fun()-> 2+2 end.
#Fun<erl_eval.20.90072148>
2> spawn(F).
```

```
<0.36.0>
3> spawn(F).
<0.38.0>
4> spawn(F).
<0.40.0>
```

Je maakt eerste een anonieme functie `F` en daarna start je het proces met `spawn/1`. Als resultaat zie je de pid van het nieuwe proces. Telkens als je een nieuw proces maakt, krijg je een nieuwe pid.

Je kan de anonieme functie ook rechtstreeks doorgeven:

```
spawn(fun()->io:format("hallo~n") end).
```

Het volgende fragment toont hoe je met een list comprehension meerdere processen kan maken. Het resultaat van deze bewerking is een lijst met de pid's van de nieuwe processen.

```
1>[spawn(fun()->io:format("hallo ~p~n", [X]) end) || X <- lists:seq(1,10)].
hallo 1
hallo 2
hallo 3
hallo 4
hallo 5
hallo 6
hallo 7
hallo 9
hallo 8
hallo 10
[<0.35.0>,<0.36.0>,<0.37.0>,<0.38.0>,<0.39.0>,<0.40.0>,
 <0.41.0>,<0.42.0>,<0.43.0>,<0.44.0>]
```

Elk proces heeft een pid die dienst doe als unieke identificatie. Met `self/0` kan je de pid van het huidige proces opvragen.

Je kan interactief de werking van `self()` testen.

```
1> self().
<0.33.0>
2> error(mijnfout).
** exception error: mijnfout
3> self().
<0.36.0>
4>
```

De eerste oproep van `self().` geeft de waarde `<0.33.0>` terug. Dit is de pid van de shell die momenteel loopt. Met `error(mijnfout).` wordt een fout in de lopende shell veroorzaakt. Dit heeft tot gevolg dat het proces van de lopende shell stilgelegd wordt. Bovenop de shell draait er een bewakerproces die deze onderbreking ziet en onmiddellijk een nieuw proces als shell start. Met de tweede oproep van `self().` krijg je de pid van dit nieuwe proces en aan de waarde `<0.36.0>` zie je dat het om een ander proces gaat.

De enige manier waarop processen gegevens kunnen uitwisselen is via het versturen van berichten. Het uitroepteken wordt gebruikt om een bericht te versturen naar een proces. Met de pid geef je aan wat het ontvangend proces is. Het volgend commando verstuurt het bericht `hallo` naar het shellproces.

```
4> self() ! hallo.
```

```
hallo
5> flush().
Shell got hallo
ok
```

Hier wordt een bericht naar zichzelf gestuurd. Met `flush()` kan je ze allemaal ophalen. Berichten kunnen uit om het even welke data bestaan. Je kan dus complete lijsten en tuples als bericht versturen. Een bericht dat aankomt bij een proces wordt daar in een wachtrij opgeslagen tot het verder behandeld wordt. Met `receive` kan je ontvangen berichten één per één verwerken.

In het volgende voorbeeld stopt het proces telkens na het ontvangen van een bericht.

```
-module(vb3).
-export([ontvanger/0]).

ontvanger() ->
   receive
      hallo ->
         io:format("hallo~n");
      bericht ->
         io:format("bericht~n");
      _ ->
         io:format("onbekend bericht~n")
   end.
```

In de module `vb3` wordt de `receive  end` constructie gebruikt om ontvangen berichten te verwerken. Om de verschillende soorten berichten te kunnen onderscheiden wordt patroonherkenning toegepast. In dit voorbeeld zijn er 3 patronen: `hallo`, `bericht` en _. Deze laatste is het jokerteken en dient om elk ander soort bericht te kunnen opvangen. De volgende sequentie toont hoe je met deze module een proces kan maken en er berichten naar toe kan sturen.

```
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe]

Eshell V6.1  (abort with ^G)
1> c(vb3).
{ok,vb3}
2> Pid = spawn(vb3,ontvanger,[]).
<0.40.0>
3> Pid ! hallo.
hallo
hallo
4> Pid2 = spawn(vb3,ontvanger,[]).
<0.43.0>
5> Pid2 ! bericht.
bericht
bericht
6> Pid3 = spawn(vb3,ontvanger,[]).
<0.47.0>
7> Pid3 ! xyz.
onbekend bericht
xyz
```

Commando 1 compileert de module. Commando 2 maakt een nieuw proces met `spawn/3`. De eerste parameter is de naam van de module, de tweede de naam van de functie en de derde parameter is de argumentenlijst de meegegeven wordt aan de functie die als proces zal draaien. In dit geval is de lijst leeg. Als resultaat krijg je een pid die in de variabele `Pid` wordt opgeslagen. De waarde van het resultaat, in dit geval een pid, wordt getoond.

Commando 3 verstuurt het bericht `hallo` naar het nieuwe proces. Dit proces staat te wachten met `receive` tot er een bericht ontvangen wordt. De patroonherkenning kiest de gepaste verwerking van het bericht. Na het ontvangen en verwerken van één bericht stopt het proces. Met commando 4 wordt opnieuw een proces gestart. Je moet nu wel een andere variabele `Pid2` gebruiken omdat `Pid` al een waarde had en niet meer kan gewijzigd worden.

De module `vb3` is in staat om als een proces telkens één bericht te verwerken. Om te vermijden dat een proces stopt na een bericht moet je er een herhaling inbouwen. En herhalingen in Erlang wordt als recursie geschreven.

Door recursie toe te passen blijft de ontvanger draaien na het ontvangen van berichten. Je moet zeker staartrecursie gebruiken om te vermijden dat de stack groeit.

```
-module(vb3b).
-export([ontvanger/0]).

ontvanger() ->
   receive
      hallo ->
         io:format("hallo~n"),
         ontvanger();
      bericht ->
         io:format("bericht~n"),
         ontvanger();
      _ ->
         io:format("onbekend bericht~n"),
         ontvanger()
   end.
```

In de bovenstaande module is op het einde van elke verwerking van een binnenkomend bericht een oproep `ontvanger()` geplaatst. Hierdoor ontstaat de staartrecursie.

Het volgende voorbeeld in de module `koelkast` is met een aantal concepten verder uitgewerkt tot een complete en bruikbare module. We overlopen even deze concepten:

- De module `koelkast` heeft een eigen functie `start/1` waarmee het proces gestart kan worden.
- De module heeft ook een `stop/1` functie om het proces om een fatsoenlijke manier stil te leggen
- Het is mogelijk om de toestand van de koelkast bij te houden. Dit is een lijst van objecten die zich in de koelkast bevinden. Deze toestand kan gewijzigd worden. Hiervoor wordt staartrecursie gebruikt.
- Er zijn de functies `plaats/2` en `neem/2` waarmee je respectievelijk iets in de koelkast kan plaatsen en iets eruit kan halen. Deze functies stellen zelf de berichten op die naar de koelkast verstuurd worden. Je moet dus zelf geen berichten meer samenstellen.

Hier volgt de broncode van `koelkast.erl`.

```
-module(koelkast).
-export([start/1, init/1, stop/1, plaats/2, neem/2]).

% Publieke functies

start(Voedsellijst) ->
   spawn(koelkast, init, [Voedsellijst]).

stop(Pid) ->
   Pid ! stop.

plaats(Pid, Voedsel) ->
```

```
   Pid ! {self(), {plaats, Voedsel}},
   receive
      A -> io:format("antwoord ~p~n", [A])
   after
      2000 -> io:format("geen antwoord~n")
   end.

neem(Pid, Voedsel) ->
   Pid ! {self(), {neem, Voedsel}},
   receive
      A -> io:format("antwoord ~p~n", [A])
   after
      2000 -> io:format("geen antwoord~n")
   end.

% Interne functies

init(Voedsellijst) ->
   koelkast(Voedsellijst).

koelkast(Voedsellijst) ->
   io:format("koelkast start ~p~n", [Voedsellijst]),
   receive
      {Van, {plaats, Voedsel}} ->
         io:format("plaats ~p~n", [Voedsel]),
         Van ! {self(), ok},
         koelkast([Voedsel | Voedsellijst]);

      {Van, {neem, Voedsel}} ->
         io:format("neem ~p~n", [Voedsel]),
         case lists:member(Voedsel, Voedsellijst) of
            true ->
               Van ! {self(), {ok, Voedsel}},
               koelkast(lists:delete(Voedsel, Voedsellijst));
            false ->
               Van ! {self(), niet_gevonden},
               koelkast(Voedsellijst)
         end;

      stop ->
         io:format("stop~n"),
         ok;

      _ ->
         io:format("onbekend~n"),
         ok
   end.
```

De `start/1` functie krijgt als parameter de lijst van objecten die als toestand worden bijgehouden in het te creëren proces. Via de `init/1` functie wordt de `koelkast` functie gestart. Deze functie wacht met `receive` op binnenkomende berichten. Je kan 3 soorten berichten sturen:

```
{Van, {plaats, Voedsel}}
{Van, {neem, Voedsel}}
stop
```

Er worden atomen gebruikt om het onderscheid tussen de verschillende berichten mogelijk te maken. Het eerste bericht meldt aan de koelkast dat er een object `Voedsel` in de koelkast moet geplaatst

worden. De `Van` variabele bevat de pid van de afzender. Je moet die meesturen anders weet de koelkast niet van wie de aanvraag komt. Het tweede bericht meldt aan de koelkast dat er een object `Voedsel` moet uitgenomen worden.

Dit is de verwerking van het plaatsen van een object.

```
{Van, {plaats, Voedsel}} ->
   io:format("plaats ~p~n", [Voedsel]),
   Van ! {self(), ok},
   koelkast([Voedsel | Voedsellijst]);
```

Eerst wordt een bericht `{Pid, ok}` teruggestuurd aan de afzender. Daarna wordt de koelkast functie recursief opgeroepen met als parameter de samenstelling van het nieuwe object en de bestaande lijst. Dit wordt als `[Voedsel | Voedsellijst]` geschreven. De verticale streep is het bewerkingsteken voor de samenstelling van een element met een bestaande lijst. Het is door de recursie dat een nieuwe toestand ontstaat. Door de staartrecursie wordt er geen nieuwe stackframe gemaakt en kan het proces eindeloos verder draaien zonder stackoverflow.

Dit is de verwerking van het nemen van een object. Deze verwerking hangt af van het feit of het object al dan niet voorkomt in de lijst. Dit wordt getest met `lists:member(Voedsel, Voedsellijst)`. Bij `true` wordt eerst een ok-antwoord teruggestuurd en wordt het object uit de lijst verwijderd met `lists:delete(Voedsel, Voedsellijst)`. Deze bewerking levert een nieuwe lijst op die recursief aan `koelkast/1` wordt doorgegeven. Opnieuw is de toestand van het proces gewijzigd.

Bij een `false` wordt een `niet_gevonden` teruggestuurd naar de afzender en blijft de toestand ongewijzigd. Ook hiervoor is recursie nodig. Denk eraan dat de recursie noodzakelijk is om een herhaling van `receive` mogelijk te maken.

```
{Van, {neem, Voedsel}} ->
   io:format("neem ~p~n", [Voedsel]),
   case lists:member(Voedsel, Voedsellijst) of
      true ->
         Van ! {self(), {ok, Voedsel}},
         koelkast(lists:delete(Voedsel, Voedsellijst));
      false ->
         Van ! {self(), niet_gevonden},
         koelkast(Voedsellijst)
     end;
```

Naast de publieke functie `start/1` zijn er ook nog `stop/1`, `plaats/2` en `neem/2`. Door als gebruiker van de module `koelkast` deze functies te gebruiken, hoef je zelf geen berichten samen te stellen.

```
stop(Pid) ->
   Pid ! stop.

plaats(Pid, Voedsel) ->
   Pid ! {self(), {plaats, Voedsel}},
   receive
      A -> io:format("antwoord ~p~n", [A])
   after
      2000 -> io:format("geen antwoord~n")
   end.

neem(Pid, Voedsel) ->
```

```
   Pid ! {self(), {neem, Voedsel}},
   receive
      A -> io:format("antwoord ~p~n", [A])
   after
      2000 -> io:format("geen antwoord~n")
   end.
```

De `stop` functie verstuurt het bericht `stop` zodat het koelkastproces stopt. De functies `plaats/2` en `neem/2` versturen elk een bericht en wachten dat op een antwoord. Omdat je niet altijd zeker bent of het verstuurde bericht wel aangekomen is, wordt het ontvangen van het antwoord bewaakt met een timeout. Hiervoor wordt het woord `after` gebruikt. Als patroon bij dit woord noteer je een tijd in ms. Indien er binnen deze tijd geen bericht ontvangen wordt, wordt de bijbehorende actie uitgevoerd.

Tot slot is er voor dit voorbeeld nog een sequentie van commando's waarmee de module getest kan worden.

```
1> c(koelkast).
{ok,koelkast}
2> Pid = koelkast:start([appel,peer]).
koelkast start [appel,peer]
<0.40.0>
3> koelkast:plaats(Pid, banaan).
plaats banaan
koelkast start [banaan,appel,peer]
antwoord {<0.40.0>,ok}
ok
4> koelkast:plaats(Pid, kiwi).
plaats kiwi
koelkast start [kiwi,banaan,appel,peer]
antwoord {<0.40.0>,ok}
ok
5> koelkast:neem(Pid, appel).
neem appel
koelkast start [kiwi,banaan,peer]
antwoord {<0.40.0>,{ok,appel}}
ok
6> koelkast:neem(Pid, kers).
neem kers
koelkast start [kiwi,banaan,peer]
antwoord {<0.40.0>,niet_gevonden}
ok
7> koelkast:stop(Pid).
stop
stop
8> koelkast:neem(Pid, peer).
geen antwoord
ok
```

Merk op dat er bij commando 8 een timeout ontstaat. Commando 7 heeft immers het koelkastproces stilgelegd. Hierdoor zal het door commando 8 verstuurde bericht nergens aankomen. Dit wordt terecht bewaakt met een timeout.

Hier is nog een voorbeeld waarin getoond wordt dat je processen maken zodat het rekenwerk kan verdeeld worden over de beschikbare cores. Dit voorbeeld rekent de som van de `1/i` reeks uit. Voor elke term in deze reeks wordt er een apart proces gestart.

```
-module(reeks).
```

```
-export([start/1]).

start(N) ->
   io:format("start~n"),
   C = self(),

   % start de processen
   [spawn(
      fun() ->
         io:format("reken met ~p~n", [I]),
         C ! 1/I
      end
    ) || I <- lists:seq(1,N)],

   % wacht op de antwoorden
   L = [receive T -> T end || _I <- lists:seq(1,N)],
   lists:foldl(fun(X, Som) -> X + Som end, 0, L).
```

Er wordt gebruik gemaakt van een list comprehension om de processen te starten. Dit is deze list comprehension:

```
   [ ... || I <- lists:seq(1,N)],
```

Hiermee laten we `I` variëren van `1` tot en met `N`. Voor het starten van de processen wordt `spawn/1` gebruikt. Deze versie verwacht een anonieme functie:

```
    spawn(
      fun() ->
         io:format("reken met ~p~n", [I]),
         C ! 1/I
      end)
```

Deze functie rekent de waarde van de term uit. Merk op dat deze functie gebruik maakt van de variabele `I`. Het is namelijk zo dat een anonieme functie naast de lokale variabelen en parameters ook kan gebruik maken van variabelen die buiten de definitie van de anonieme functie hun waarde hebben gekregen. De waarde die de anonieme functie ziet is de waarde die bevroren wordt op het moment van de definitie van de anonieme functie en niet de uitvoering. Door dit mechanisme is het mogelijk dat deze anonieme functie telkens in een ander proces draait met telkens een andere waarde van `I`. De `io:format/2` staat er op dit te verifiëren.

Voor de variabele C geldt dezelfde redenering. Deze variabele is nodig om de juiste PID te verkrijgen. Als je `self() ! 1/I` zou schrijven, gaan de processen de term aan zichzelf studeren en dit is niet de bedoeling.

De voorlaatste stap is het verzamelen van de termen die door de afzonderlijke processen worden verstuurd. De volgende list comprehension maakt een lijst met alle termen.

```
   L = [receive T -> T end || _I <- lists:seq(1,N)],
```

Tot slot worden alle termen in de lijst opgeteld met `foldl/3`. De anonimie functie doet ht werk. `foldl/3` zorgt alleen maar voor de herhaling.

```
   lists:foldl(fun(X, Som) -> X + Som end, 0, L).
```

Door het feit dat de berekening van de termen in aparte processen gebeurt, kan dit rekenwerk verdeeld worden over de verschillende cores.

1.14. Fouten en processen

In dit deel van deze cursustekst gaan we na wat er exact gebeurt als processen door een fout afgebroken worden. Ook bekijken we wat we kunnen doen om hierop gepast te reageren.

Het principe in Erlang is dat we taken van een programma verdelen over verschillende processen. Erlang ontwerpers zullen dus veelvuldig gebruik maken van processen. Processen hebben het voordeel dat ze zelfstandig en zonder invloed van andere processen hun werk kunnen doen. Processen in Erlang hebben immers geen gemeenschappelijke data. Als nu een proces crasht door één of andere fout, zullen de overige processen daar niets van opvangen. Dat kan voor onverwachte problemen zorgen omdat alle andere processen berichten blijven versturen naar het gecrashte proces en bijgevolg geen antwoord zullen ontvangen. In Erlang weet je nooit of een bericht is aangekomen tenzij je de afzender expliciet een bericht laat terugsturen.

In veel gevallen wil een proces weten of een buurproces nog in leven is. Je zou dit kunnen organiseren door telkens een pingbericht naar het buurproces te sturen en die dan met een pongbericht te laten antwoorden. Deze werkwijze is echter zeer omslachtig en de bewaking kan anders opgelost worden. Erlang heeft een aantal ingebouwde mechanismen die enkel in actie treden wanneer het nodig is. Zo kan Erlang jouw proces op de hoogte brengen dat een buurproces gecrasht is. Hoe we van dit soort interacties tussen processen kunnen gebruik maken, bekijken in de volgende secties.

In de volgende secties doen we een aantal testen met processen die we vrijwillig laten crashen. De module die we daarvoor gebruiken ziet er zo uit:

```
-module(vb4).
-export([loop/0]).

loop() ->
   receive
      X ->
         io:format("de helft is ~p~n",[X/2]),
         loop()
   end.
```

Met de `loop/0` functie kan een proces permanent wachten op een bericht en bij ontvangst van een bericht een actie uitvoeren. Als actie wordt het doorgestuurde getal gehalveerd. Als het bericht geen getal is, is er een probleem bij de deling en zal het proces crashen. Dit is hetgeen we willen bereiken. We willen nagaan wat er gebeurt als een proces crasht.

1.14.1. Processen zonder links

In de eerste test wordt het proces zonder meer gestart en versturen we getallen naar het proces. Commando 3 maakt het proces en commando's 4 en 5 versturen een getal. Met commando 6 versturen we een atoom en dan loopt het mis. Omdat het proces vanuit de shell is gestart zie je de foutmelding. Wanneer een Erlang programma in de achtergrond wordt gestart, zal deze foutmelding niet zichtbaar zijn en zullen de overige processen niet weten dat een ander proces gecrasht is.

```
1> c(vb4).
{ok,vb4}
2> self().
<0.33.0>
3> Pid=spawn(vb4,loop,[]).
<0.41.0>
4> Pid!44.
de helft is 22.0
44
5> Pid!88.
```

```
de helft is 44.0
88
6> Pid!acht.
acht
7>
=ERROR REPORT==== 5-Sep-2014::09:51:18 ===
Error in process <0.41.0> with exit value: {badarith,[{vb4,loop,0,[{file,"vb4.e


7> self().
<0.33.0>
8>
```

Met de functie `self()` hebben we éénmaal vóór de crash en éénmaal na de crash de pid van de shell opgevraagd. We zien telkens dezelfde pid. Dit betekent dat het shellproces zelf is blijven draaien ook al is er een ander proces gecrasht. Dit gedrag kan je wijzigen door links tussen processen vast te leggen.

1.14.2. Processen met links

Een proces dat crasht is een geïsoleerd fenomeen dat niet door de andere processen waargenomen wordt. Dit verandert als je links tussen processen gaat leggen. Een proces dat crasht verstuurt een signaal naar alle processen waarmee het een link heeft. Deze laatste processen crashen dan ook. Je krijgt dus een kettingreactie die uiteindelijk een hele wolk van gelinkte processen doet crashen. In praktijk is dit wel niet wat we wensen maar in een testfase is het wel nuttig om te zien dat door een minste fout met een crash als gevolg een heel systeem stilgelegd wordt.

Opnieuw starten we een test.

```
1> c(vb4).
{ok,vb4}
2> self().
<0.33.0>
3> Pid=spawn(vb4,loop,[]).
<0.41.0>
4> link(Pid).
true
5> Pid!88.
de helft is 44.0
88
6> Pid!66.
de helft is 33.0
66
7> Pid!zeven.

=ERROR REPORT==== 5-Sep-2014::10:27:59 ===
Error in process <0.41.0> with exit value: {badarith,[{vb4,loop,0,[{file,"vb4.e

** exception exit: badarith
     in function  vb4:loop/0 (vb4.erl, line 9)
8> self().
<0.46.0>
```

Het verschil met de vorige test is dat we nu met `link(Pid)` een link maken tussen de shell en het `vb4` proces. Dit heeft voor gevolg dat als het `vb4` proces crasht er een signaal naar het shellproces wordt gestuurd. Het shellproces crasht dan ook. Dat zie je aan de tweede `self()` van commando 8. Daar wordt een andere pid getoond dan die van commando 2. Er is door de crash van het shellproces een nieuw shellprocess gestart. Ook zie je tweemaal dezelfde foutmelding omdat er twee processen om dezelfde rden zijn gecrasht.

1.14.3. Traps

Je kan vermijden dat een proces door een link met aan ander proces in een kettingreactie crasht. Dan moet je instellen dat je het ontvangen exit signaal wil laten omvormen door een bericht. Dat doe je met

```
process_flag(trap_exit,true).
```

En we starten de test opnieuw. Deze keer wordt het proces gestart met `spawn_link/3`. Dit is de variant die eerst het proces start en dan een link legt tussen de twee processen.

```
1> c(vb4).
{ok,vb4}
2> self().
<0.33.0>
3> process_flag(trap_exit, true).
false
4> Pid=spawn_link(vb4, loop, []).
<0.42.0>
5> Pid!44.
de helft is 22.0
44
6> Pid!66.
de helft is 33.0
66
7> Pid!tien.
tien
8>
=ERROR REPORT==== 5-Sep-2014::10:49:49 ===
Error in process <0.42.0> with exit value: {badarith,[{vb4,loop,0,[{file,"vb4.e


8> self().
<0.33.0>
9> flush().
Shell got {'EXIT',<0.42.0>,
                  {badarith,[{vb4,loop,0,[{file,"vb4.erl"},{line,9}]}]}}
ok
```

We zien dat het shellproces nooit verandert. We zien bij de `self()` steeds dezelfde pid. Wanneer het `vb4` proces crasht, is er maar één foutmelding en met `flush()` zien we dat er een bericht ontvangen is om te melden dat er een proces gecrasht is. Op deze manier kan één proces een ander proces bewaken en ingrijpen met de nodige acties als het proces in bewaking. Het proces dat bewaakt, wordt in de Erlang terminologie een supervisor genoemd.

1.14.4. Monitors

Er is een tweede techniek om processen te bewaken. Hierbij wordt met `monitor/2` een proces in bewaking geplaatst. De oproep van deze functie geeft een referentie terug die later in de berichten opduikt. Er wordt geen link tussen de processen gelegd.

```
1> c(vb4).
{ok,vb4}
2> self().
<0.33.0>
3> Pid=spawn(vb4, loop, []).
<0.41.0>
```

```
4> monitor(process, Pid).
#Ref<0.0.0.79>
5> Pid!88.
de helft is 44.0
88
6> Pid!55.
de helft is 27.5
55
7> Pid!zes.
zes
8>
=ERROR REPORT==== 5-Sep-2014::11:00:47 ===
Error in process <0.41.0> with exit value: {badarith,[{vb4,loop,0,[{file,"vb4.e


8> self().
<0.33.0>
9> flush().
Shell got {'DOWN',#Ref<0.0.0.79>,process,<0.41.0>,
                 {badarith,[{vb4,loop,0,[{file,"vb4.erl"},{line,9}]}]}}
ok
```

Als een proces dat met een monitor bewaakt wordt crasht, ontvangt de bewaker een bericht met daarin alle informatie over de crash. Ook de referentie en de pid van het proces komen hierin voor.

Er is een verschil tussen links en monitors. Een link is bidirectioneel en werkt intern met signalen (is te vergelijken met C signals) en monitors zijn unidirectioneel en werken met berichten. Monitors zullen eerder gebruikt worden voor een tijdelijk bewaking en links geven een hiërarchie tussen een permanente supervisor en ondergeschikte processen.

1.14.5. Processen met naam

Met `register/2` kan je een pid van een proces onder een naam registreren. Als naam moet je een atoom gebruiken. Je kan dan de naam ter vervanging van de pid gebruiken.

```
1> c(vb4).
{ok,vb4}
2> Pid=spawn(vb4, loop, []).
<0.40.0>
3> register(deel, Pid).
true
4> deel ! 44.
de helft is 22.0
44
5> deel ! 66.
de helft is 33.0
66
6> whereis(deel).
<0.40.0>
7> unregister(deel).
true
8> deel ! 22.
** exception error: bad argument
     in operator  !/2
        called as deel ! 22
```

Met `unregister/1` kan je de registratie ongedaan maken. Deze registratie van processen heeft maar zin voor een beperkt aantal processen omdat het maximale aantal dat kan geregistreerd worden,

beperkt is. Ook is het zo dat je de registratie moet herzien als het betrokken proces crasht en vervangen wordt door een nieuwe proces.

Er bestaat een `gproc` module die veel minder beperkingen heeft.

1.15. ETS tabellen

Met ETS heeft Erlang een implementatie van een database waarbij de gegevens enkel in het geheugen opgeslagen worden. Je kan in ETS één of meerdere tabellen aanmaken waarin records worden opgeslagen. Elke record heeft een vast aantal kolommen. Dit werkt precies zoals een relationele database. In ETS heb je wel de vrijheid om als waarde voor de kolommen om het even welk datatype te gebruiken. Je kan getallen, atomen, tuples en lijsten of combinaties van de voorgaande types als waarde voor de kolommen van een ETS record gebruiken. Ook zoals relationele databases kan één van de kolommen dienst doen als unieke sleutel. De waarde die je als sleutel gebruikt, mag van om het even welk type zijn.

ETS kan meerdere tabellen bijhouden. Elke tabel krijgt een atoom als naam. Per node is het aantal tabellen beperkt tot 1400. Anders dan we gewoon zijn bij de variabelen van Erlang kan je de waarde van de kolommen van een record wijzigen. Met een ETS tabel heb je dus een opslagmiddel waarin wijzigingen mogelijk zijn. Het is dan ook aanlokkelijk om een aantal Erlang programma's te herschrijven zodat de belangrijkste toestandsvariabelen in ETS tabellen worden bijgehouden. Zeker voor testen is ETS handig maar voor algemeen gebruik is ETS niet altijd nodig.

De tabellen die ETS bijhoudt, bevinden zich in het geheugen. Een tabel is ook altijd geassocieerd met een proces. Als het proces dat eigenaar is van een ETS tabel crasht, dan verdwijnt ook deze tabel. Wanneer je ETS vervangt door DETS dan worden de tabellen in bestanden bijgehouden. Er is tenslotte ook nog Mnesia. Dit is een databasesysteem waarbij de bestanden waarin de tabellen bijgehouden worden, verspreid zijn over meerdere nodes. Dit laat redundantie toe zodat bij uitval van een node de informatie toch niet verloren gaat. In deze sectie wordt alleen maar ETS besproken.

De ETS tabellen kunnen aangemaakt worden volgens verschillende principes. Dit zijn de atomen die de verschillende mogelijkheden vastleggen:

- `set`

  Voor elke sleutel heb je maar één record. Als je de waarde van de record voor een bepaalde sleutel opnieuw schrijft, wordt de oude waarde vervangen.

- `ordered_set`

  Dit is hetzelfde als `set` maar dan met een geordende sleutel. Deze ordening is handig als je queries met bereiken wil doen. De ordening wordt bepaald door het vergelijken van Erlang termen.

- `bag`

  Hier zijn meerdere records voor een zelfde sleutel mogelijk. De waarden in deze records moeten verschillend zijn.

- `duplicate_bag`

  Dit is zoals een `bag` maar er mogen meerdere identieke records zijn.

Hier zijn enkele voorbeelden:

- `{a,1} {b,2} {c,3}` is een `set`.
- `{a,1} {a,2} {b,2}` is een `bag`.
- `{a,1} {a,1}{a,2} {b,3}` is een `duplicate_bag`.

Een tabel in ETS kan op de volgende manier aangemaakt worden. Het atoom `lessen` legt de naam van de tabel vast. De lijst die erop volgt legt een aantal kenmerken van de nieuwe tabel vast.

```
LesTabel = ets:new(lessen, [named_table, {keypos, #les.vak}]).
```

`name_table` geeft aan dat de tabel een naam krijgt. Deze naam mag dan als verwijzing naar de tabel gebruikt worden in de overige functies van de `ets` module. Met de tuple `{keypos, Nummer}` wordt aangegeven op welk element er geordend wordt. In dit voorbeeld wordt een record gebruikt als opslag en wordt de notatie `#les.vak` gebruikt om het rangnummer van `vak` binnen de record `les` te verkrijgen.

Je kan de rangnummers zo opvragen:

```
7> c(tabel).
{ok,tabel}
8> rr(tabel).
[les]
9> #les.vak.
2
10> #les.lokaal.
3
11> #les.docent.
4
```

Je ziet dat de rangnummers voor de velden `vak`, `lokaal` en `docent` respectievelijk 2, 3 en 4 zijn. Rangnummer 1 is gereserveerd voor het atoom dat het type van de record beschrijft. Het is niet verplicht om met records te werken als je gegevens in een ETS tabel wil opslaan. Tuples gaan ook. Eigenlijk kan je alleen maar tuples gebruiken. Het is immers zo dat records intern als tuples worden voorgesteld.

Als je niet aangeeft of een tabel een `set`, `ordered_set`, `bag` of `duplicate_bag` moet zijn, wordt default voor `set` gekozen. Je kan het atoom `ordered_set` vermelden om een gesorteerde tabel te verkrijgen. Dat wordt dan zo geschreven:

```
LesTabel = ets:new(lessen, [ordered_set, named_table, {keypos, #les.vak}]).
```

Je kan zo gegevens in de tabel plaatsten.

```
ets:insert(LesTabel, #les{vak="RTOS",  lokaal="B103", docent="RuLe"}),
ets:insert(LesTabel, #les{vak="INTAP", lokaal="B102", docent="RuLe"}),
ets:insert(LesTabel, #les{vak="JAVA",  lokaal="B102", docent="RuLe"}),
ets:insert(LesTabel, #les{vak="GWEBA", lokaal="E101", docent="RuLe"}),
```

Je kan evengoed een tuple in plaats van een record in een ETS tabel bijvoegen. De tuples hoeven zelfs niet evenveel elementen te hebben.

```
ets:insert(LesTabel, {test, 2, 3}),
```

Merk op dat het mogelijk is om voor een bepaalde waarde van de sleutel een gewijzigde record in de tabel op te slaan. In ETS kan je records overschrijven.

```
ets:insert(Tab, #les{vak="RTOS",lokaal="B106",docent="RuLe"}).
```

Door deze `insert/2` oproep is de record met als sleutel `RTOS` gewijzigd. Het lokaal is nu `B106`. Samengevat kan je stellen dat met `insert/2` een nieuwe record wordt bijgevoegd als die nog niet bestaat en anders wordt de bestaande record vervangen.

Met `info/1` kan je de volgende informatie over een tabel opvragen.

```
12>ets:info(lessen).
[{compressed,false},
 {memory,444},
 {owner,<0.50.0>},
 {heir,none},
 {name,lessen},
 {size,5},
 {node,nonode@nohost},
 {named_table,true},
 {type,set},
 {keypos,2},
 {protection,protected}]
```

Met `lookup/2` kan je de record met een bepaalde sleutel opvragen.

```
ets:lookup(lessen, "JAVA").
```

Met `delete/2` kan je de record met een bepaalde sleutel verwijderen.

```
ets:delete(lessen, "GWEBA").
```

Met `foldl/3` kan je de records van een tabel doorlopen. De eerste parameter is een anonieme functie die voor elke record opgeroepen wordt. De tweede parameter is de startwaarde van de accumulator en de derde parameter is de verwijzing naar de tabel. De anonieme functie heeft als eerste parameter de waarde van de record en als tweede de accumulator. Als returnwaarde geeft de anonieme functie de nieuwe waarde van de accumulator terug. In dit voorbeeld wordt een lijst met daarin alle records opgebouwd. Naast `foldl3` bestaat ook `foldr/3`; deze werkt in de omgekeerde volgorde.

```
ets:foldl(fun(T, Acc)-> [T | Acc] end, [], lessen).
```

Het volgende fragment toont hoe je `foldr/3` kan gebruiken om bepaalde records te selecteren. De anomieme functie heeft nu 2 definities (clauses): de eerste selecteert een record met het gewenste lokaal en de tweede definitie is voor de overige records. De records met het gewenste lokaal worden wel in de accumulatorlijst geplaatst, de overige niet. In dat geval blijft de doorgegeven accumulator behouden.

```
lokaal(LT, Lokaal) ->
   F = fun
          (T, Acc) when T#les.lokaal==Lokaal ->
             [T | Acc];
          (_, Acc) ->
             Acc
       end,
   ets:foldr(F, [], LT).
```

Een andere manier om doorheen de tabel te lopen is gebruik maken van `first/1` en `next/2`. Met `first/1` kan je de eerste sleutel van een tabel opvragen. Met `next/2` ga je van de huidige sleutel naar de volgende. Indien een sleutel niet bestaat wordt `'$end_of_table'` teruggegeven.

```
alles(LT) ->
   stap(LT, ets:first(LT)).
```

```
stap(_LT, '$end_of_table') ->
   einde;
stap(LT, Key) ->
   io:format("key  ~p~n", [Key]),
   stap(LT, ets:next(LT, Key)).
```

Dit is de module `tabel` waarmee ETS kan getest worden.

```

-module(tabel).
-export([info/1, lookup/2, delete/2, foldl/1, foldr/1, lokaal/2, alles/1, new/0

-record(les, {vak, lokaal, docent}). % les record

info(LT) ->
   ets:info(LT).

lookup(LT, Key) ->
   ets:lookup(LT, Key).

delete(LT, Key) ->
   ets:delete(LT, Key).

foldl(LT) ->
   ets:foldl(fun(T, Acc)-> [T | Acc] end, [], LT).

foldr(LT) ->
   ets:foldr(fun(T, Acc)-> [T | Acc] end, [], LT).

lokaal(LT, Lokaal) ->
   F = fun
          (T, Acc) when T#les.lokaal==Lokaal ->
             [T | Acc];
          (_, Acc) ->
             Acc
       end,
   ets:foldr(F, [], LT).

alles(LT) ->
   stap(LT, ets:first(LT)).

stap(_LT, '$end_of_table') ->
   einde;
stap(LT, Key) ->
   io:format("key  ~p~n", [Key]),
   stap(LT, ets:next(LT, Key)).

new() ->
   LesTabel = ets:new(lessen, [named_table, {keypos, #les.vak}]),
   ets:insert(LesTabel, #les{vak="RTOS",  lokaal="B103", docent="RuLe"}),
   ets:insert(LesTabel, #les{vak="INTAP", lokaal="B102", docent="RuLe"}),
   ets:insert(LesTabel, #les{vak="JAVA",  lokaal="B102", docent="RuLe"}),
   ets:insert(LesTabel, #les{vak="GWEBA", lokaal="E101", docent="RuLe"}),
   LesTabel.
```

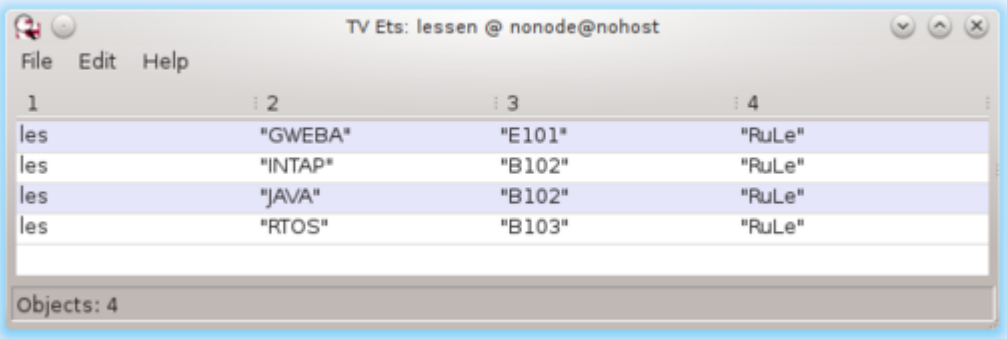
Je kan een tabel nakijken met de observer. Die start je zo:

```

8>observer:start().
```

Deze applicatie toont een venster met verschillende tabs. Eén van de tabs is table viewer. Daarin kan je een tabel aanklikken en dan wordt de inhoud van de tabel getoond.

Figure 1. table viewer



Tot slot is er nog een techniek om ervoor te zorgen dat een tabel het eigenaarsproces overleeft. Het volgende fragment maakt een extra proces dat mede-eigenaar wordt van de tabel. Dit doe je met `give_away/3`. Als de eerste eigenaar crasht, dan blijft de tabel nog bestaan omdat er een mede-eigenaar is.

```
loop() ->
   io:format("loop~n", []),
   receive
      X ->
         io:format("receive ~p~n", [X]),
         loop()
   end.

maak_tabel() ->
   Pid = spawn(tabel,loop, []),
   Tab = new(),
   ets:give_away(Tab, Pid, []),
   Tab.
```

De functie `loop/0` is het proces dat als mede-eigenaar dienst doet. Dit proces stopt nooit en laat zo de tabel verder bestaan.

1.16. Gedistribueerde processen

1.16.1. Concepten

Je kan de Erlang virtuele machine `erl` op verschillende knooppunten (node) starten en dan processen in de verschillende knooppunten met elkaar laten communiceren. Dit kenmerk van Erlang is oorspronkelijk één van de belangrijkste redenen geweest om Erlang te ontwerpen. Deze communicatie tussen de processen is zodanig gemaakt dat de processen zelf niet kunnen weten of ze communiceren met processen uit het eigen knooppunt of uit afgelegen knooppunten. Hierdoor is het mogelijk om in Erlang softwaresystemen te ontwerpen waarbij alle processen op eenzelfde knooppunt draaien en later deze processen te verspreiden over meerdere knooppunten. Je hebt in Erlang de vrijheid om processen onder te brengen in één of meerdere knooppunten naar keuze. Bij de start van een ontwerp draai je alle processen op hetzelfde knooppunt. Naarmate het ontwerp vordert, kan je de processen verspreiden over verschillende knooppunten zonder dat dit ingrijpende wijzigingen vraagt.

Voor alle duidelijkheid is hier een overzicht van de concepten bij de gedistribueerde werking van Erlang:

- platform

  Met platform wordt een PC, server of single board computer bedoeld. Op dit platform draait een besturingssysteem en is Erlang/OTP geïnstalleerd. Elk platform heeft tenminste één IP adres waaraan een domeinnaam is gekoppeld. Het IP adres kan dynamisch of static toegekend zijn. Het moet niet statisch zijn, dynamisch werkt ook zolang er maar een domeinnaam aangekoppeld is. Bij een statisch adres moet je ervoor zorgen dat het platform een domein in `/etc/hosts` krijgt. Bij een dynamisch adres moet je een andere oplossing kiezen, een Rendez-Vous implementatie zoals avahi.org is een mogelijke oplossing. De domeinnaam moet in orde zijn omdat de platformen elkaar contacteren via de domeinnaam.

- knooppunt of node

  Op elk platform kunnen meerdere nodes gestart worden. Elke opstart van het commando `erl` is een nieuwe node die tot leven komt. Je kan meerdere nodes op één platform starten, maar je kan ook per platform slechts één node starten. Dit zal eerder het geval zijn op een SBC. Elke node heeft een samengestelde naam bestaande uit de eigenlijke naam van de node en de DNS naam van het platform. Er zijn korte en lange nodenamen. Dit verschil wordt later nog uitgelegd.

- gewone node of gedistribueerde node

  Een node die je met `erl` start, is een gewone node. Dit type node kan nooit met andere nodes communiceren. Een gedistribueerde node kan dat wel. Deze moet gestart worden met `erl -name naam` of `-sname naam`. Het extra argument geeft de node een naam zodat die door een andere node kan gecontacteerd worden.

- `-name` of `-sname`

  Binnen een cluster moet je ofwel voor een korte naam ofwel voor een lange naam kiezen. Je kan de twee niet combineren omdat een node met een korte naam nooit kan communiceren met een node met een lange naam. En heel belangrijk, of je nu kiest voor korte of lange namen, deze namen moeten altijd via DNS bereikbaar zijn. Vanuit node `'een@geel'` moet het shell commando `ping oranje` werken als je voor korte namen kiest. Bij lange namen moet `ping oranje.local` werken. Dit betekent dat de organisatie van DNS zal bepalen of je met korte of lange namen kan werken.

  Met `-name` geef je de node een lange samengestelde naam en met `-sname` wordt het een korte naam. De keuze tussen lang en kort heeft betrekking op de domeinnaam van het platform waarop de node draait.

  Met `-name twee` krijg je een lange naam:

```
2> node().
'twee@oranje.local'
```

Met `-sname twee` krijg je een korte naam:

```
2> node().
'twee@oranje'
```

- proces

  Binnen een node kunnen er meerdere processen draaien. Elk proces heeft een pid die als een reeks getallen zoals `<0.40.0>` wordt voorgesteld. Als je een procespid liever als een naam voorstelt, dan moet je die naam registeren met `register/2`. De tabel met alle geregistreerde namen is lokaal voor de node. Dit wil zeggen dat een proces alleen maar met die naam bekend is binnen de node.

- cluster

  Nodes die elkaar tenminste éénmaal gecontacteerd hebben, vormen een cluster. Elke nieuwe node die een node van de cluster contacteert, is dan meteen bekend bij alle nodes van de cluster. Een cluster wordt beveiligd met een cookie. Alleen de nodes die de cookie kennen, kunnen toetreden tot de cluster.

1.16.2. Een connectie tussen nodes

We maken nu een cluster met twee nodes. Elke node draait op zijn eigen platform. Op het platform `oranje` starten we `erl` met de volgende opties:

```
root@oranje:/etc# erl -sname twee -setcookie test
Erlang R15B01 (erts-5.9.1) [source] [smp:2:2] [async-threads:0] [kernel-poll:fa

Eshell V5.9.1  (abort with ^G)
(twee@oranje)1>
```

Met `-sname twee` wordt de nodenaam aangegeven. Elke `erl` instantie die gestart wordt, is een aparte node. Er kunnen meerdere nodes op dezelfde machine draaien; ze moeten we een verschillende nodenaam hebben. Door het meegeven van de optie `-sname` wordt een node meteen een gedistribueerde node. Dit betekent dat deze node in staat is om met andere nodes op hetzelfde of andere platform te communiceren. Op een gedistribueerde node draait ook altijd de daemon `epmd` in de achtergrond. Deze daemon houdt een lijst van namen met het bijbehorende IP-adres bij.

Je kan ook kiezen voor de optie `-name een`, in dit geval moet een volledig gekwalificeerde domeinnaam gebruikt worden. Bij de optie `-sname` kan je een verkorte naam gebruiken; dit is de hostname zonder domein.

Naast de nodenaam moet nog een cookie meegegeven worden. Alle nodes die willen samenwerken in een cluster moeten gestart worden met eenzelfde cookie. Door met een geheime gemeenschappelijke cookie te werken kan je ongewenste nodes buiten de cluster houden.

We starten nog een node, deze keer met nodenaam `een` op het platform `geel`. Dit laatste is de hostname.

```
user@geel:~$ erl -sname een -setcookie test
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe]

Eshell V6.1  (abort with ^G)
(een@geel)1> net_adm:ping('twee@oranje').
pong
```

Op één van beide nodes moet je een ping starten met `net_adm:ping/1`. Als parameter geef je de specificatie van de node mee, dat wordt dan `'twee@oranje'`. Merk op dat hier de verkorte naam voor de host gebruikt wordt. Het resultaat van deze ping is het atoom `pong`. Dit betekent dat de ene node met succes gekoppeld is aan de andere node. In beide nodes kan je dat verifiëren met `nodes()`. Dit geeft een lijst van alle andere nodes in dezelfde cluster.

```
(twee@oranje)1> nodes().
[een@geel]
```

Als de `ping/1` geen `pong` geeft maar een `pang`, is er een probleem. Ofwel is er een DNS probleem of een probleem met de cookie of is er een combinatie gemaakt van een node met `-sname` en een node met `-name`. In elk geval moet je eerste het `ping` shell commando gebruiken om DNS te verifiëren: of je nu `-sname` of `-name` gebruikt, de node moet via `ping` bereikt kunnen worden.

Als je een bericht wil sturen van één node naar de andere, is het handig om de bestemmeling met een naam te registreren.

```
(twee@oranje)2> register(bordje, self()).
true
(twee@oranje)3> whereis(bordje).
<0.38.0>
```

Een bericht stuur je dan zo:

```
(een@geel)2> {bordje,'twee@oranje'}!hallo.
hallo
```

Als bestemmelingsadres moet je een tuple maken bestaande uit de geregistreerde naam en de nodenaam. De nodenaam kan je per node opvragen met `node()`.

Vermits we de `erl` shell als ontvanger hebben geregistreerd, kan je `flush/0` gebruiken om na te gaan of er een bericht is binnengekomen.

```
(twee@oranje)4> flush().
Shell got hallo
ok
```

Merk op dat de registratielijst waar we processen registreren, lokaal is per node. Vanuit een remote node zie je deze atomen niet.

Er zijn geen beperkingen voor wat betreft de communicatie tussen processen binnen eenzelfde node of tussen processen op verschillende gedistribueerde nodes. Hou er wel rekening mee dat als de communicatie over het netwerk gaat, alles wel trager gaat. Ook is het zo dat programmacode niet of nauwelijks hoeft aangepast te worden om gedistribueerd te kunnen werken. Als je er een gewoonte van maakt om de pids van de processen waarmee je communiceert in variabelen bij te houden, dan kan deze variabele zowel een Pid of een tuple bijhouden. Door met deze variabelen te werken wordt het al dan niet gedistribueerd communiceren transparant.

1.16.3. Berichtenverkeer tussen gedistribueerde nodes

Deze keer doen we een test waarbij automatisch een aantal berichten worden verstuurd tussen twee gedistribueerde nodes op de platformen `geel.local` en `oranje.local`.

De volgende module wordt voor deze test gebruikt. Met `start_ontvanger/0` wordt eerst het ontvangerproces gestart. Dit proces wordt geregistreerd met de naam `ontvanger`. Dit proces wacht op binnenkomende bericht en stuurt bij een ontvangst een antwoord terug. Met `loopo/0` wordt de herhaling gestart. Met `start_zender/2` wordt de zender gestart. De eerste parameter `N` is het te versturen aantal berichten en de parameter `Node` is de naam van node waar het ontvangerproces zich bevindt. De zender maakt eerst een proces voor het verwerken van de antwoorden en zendt dan met een list comprehension alle berichten. Het bestemmelingsadres wordt als `{ontvanger, Node}` geschreven. Hieraan zie je het gaat over de identificatie van een proces in een andere node.

```
-module(distrib_test).
-export([start_ontvanger/0,start_zender/2,loopo/0,loopz/0]).

start_ontvanger() ->
   Pid = spawn(distrib_test, loopo, []),
   register(ontvanger, Pid),
   Pid.
```

```
loopo() ->
   receive
      {bericht, I, Afzender}  ->
         io:format("bericht ~p ontvangen van ~p~n", [I, Afzender]),
         Afzender ! {antwoord, I, self()},
         loopo();
      X ->
         io:format("onbekend bericht ~p~n",[X]),
         loopo()
   end.

start_zender(N, Node) ->
   Pid = spawn(distrib_test, loopz, []),
   [ {ontvanger,Node} ! {bericht, I, Pid}|| I <- lists:seq(1,N)].

loopz() ->
   receive
      {antwoord, I, Afzender}  ->
         io:format("antwoord ~p ontvangen van ~p~n", [I, Afzender]),
         loopz();
      X ->
         io:format("onbekend antwoord ~p~n",[X]),
         loopz()
   end.
```

Hier wordt de ontvanger gestart. Er wordt voor de node een lange naam gebruikt.

```
root@oranje:~# erl -name twee -setcookie test
Erlang R15B01 (erts-5.9.1) [source] [smp:2:2] [async-threads:0] [kernel-poll:fa

Eshell V5.9.1  (abort with ^G)
(twee@oranje.local)1> nodes().
['een@geel.local']
(twee@oranje.local)2> c(distrib_test).
{ok,distrib_test}
(twee@oranje.local)3> distrib_test:start_ontvanger().
<0.48.0>
bericht 1 ontvangen van <5793.52.0>
bericht 2 ontvangen van <5793.52.0>
bericht 3 ontvangen van <5793.52.0>
bericht 4 ontvangen van <5793.52.0>
bericht 5 ontvangen van <5793.52.0>
(twee@oranje.local)4>
```

En hier zie je de werking van de zender. Nadat de ontvanger gestart is, moet je in de zender een `net_adm:ping/1` starten om ervoor te zorgen dat de nodes elkaar herkennen. Met `nodes/0` kan je verifiëren of de nodes elkaar kennen. Een ping vanaf één van de twee zijden volstaat.

```
lrutten@geel:~/test/erlang/distrib$ erl -name een -setcookie test
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe]

Eshell V6.1  (abort with ^G)
(een@geel.local)1> net_adm:ping('twee@oranje.local').
pong
(een@geel.local)2> c(distrib_test).
{ok,distrib_test}
(een@geel.local)3> distrib_test:start_zender(5, 'twee@oranje.local').
```

```
[{bericht,1,<0.52.0>},
 {bericht,2,<0.52.0>},
 {bericht,3,<0.52.0>},
 {bericht,4,<0.52.0>},
 {bericht,5,<0.52.0>}]
antwoord 1 ontvangen van <6250.48.0>
antwoord 2 ontvangen van <6250.48.0>
antwoord 3 ontvangen van <6250.48.0>
antwoord 4 ontvangen van <6250.48.0>
antwoord 5 ontvangen van <6250.48.0>
(een@geel.local)4>
```

Met `start_zender/2` start de eigenlijke test. In de ontvangen antwoorden zie je pid van het proces dat het antwoord teruggestuurd heeft. Deze pid wordt als `<6250.48.0>` weergegeven. Bij de ontvanger kan je dezelfde soort pid zien. Dit betekent dat een node die een afzenderpid verkrijgt, niet kan weten of het proces in andere of eigen node draait. Hierdoor is de gedistribueerde werking transparant voor de processen.

Wanneer je een complete Erlang oplossing gedistribueerd maakt, is alleen de fase van het opstarten van de processen anders dan bij een niet-gedistribueerde oplossing. Hieruit volgt dat een bestaande oplossing relatief gemakkelijk kan omgevormd worden van niet-gedistribueerd naar gedistribueerd. Uiteraard zijn er problemen waarbij er vanaf de start van het ontwerp moet gekozen worden voor gedistribueerde nodes. Indien er SBC's gebruikt worden, die elk een eigen reeks IO poorten hebben, kan je niet anders dan een gedistribueerde oplossing maken.

2. Gevorderd Erlang/OTP

In dit deel van de cursustekst worden een aantal gevorderde concepten van Erlang uitgelegd. Tot nu toe hebben we de Erlang voorbeelden via de `erl`shell getest. Deze werkwijze is handig om interactief een aantal kleine voorbeelden te testen en te verifiëren. Voor grotere programma's die uit meerderen modules en applicaties bestaan, is deze werkwijze omslachtig en zullen we anders te werk gaan.

Daarom wordt in dit deel uitgelegd hoe je complete applicaties kan samenstellen en hoe je een bundeling van meerdere applicaties kan bundelen tot een zelfstartend geheel. Voor de uitbating van Erlang software is dit een pluspunt.

Ook wordt uitgelegd hoe je de Cowboy webserver kan integreren in een bestaande applicatie.

Al deze concepten worden uitgelegd aan de hand van vier voorbeelden.

- `simpeleteller`

  Dit is een minimaal voorbeeld van een applicatie.

- `teller`

  Dit is hetzelfde als het vorige voorbeeld, maar nu gebouwd met `erlang.mk` en `relx`

- `teller-cowboy`

  Het vorige voorbeeld is uitgebreid met de Cowboy webserver.

- `teller-cowboy-template`

  Het vorige voorbeeld is uitgebreid met de ErlyDTL.

2.1. Inleiding

Kort samengevat is Erlang/OTP de combinatie van Erlang als taal met OTP als omgeving met een aantal faciliteiten die de uitbating van Erlang applicaties vergemakkelijken. Door meerdere modules

te groeperen tot een applicatie is het gemakkelijker om deze modules als een geheel te behandelen. Je kan in een node meerdere applicaties starten en beheren. Ook de afhankelijkheid tussen de applicaties kan Erlang/OTP voor zijn rekening nemen.

Om applicaties te kunnen bouwen moet je een aantal ontwerppatronen volgen. Om deze patronen gemakkelijker in te voeren heeft men in Erlang behaviours ingevoerd. Een behaviour is te vergelijken met een Java `interface`. Wanneer je een module voorziet van een behaviour ben je verplicht om een aantal functies te implementeren. Een behaviour werkt zoals een contract: een module die een contract aangaat, belooft een bepaalde functionaliteit te voorzien. Deze behaviours zijn er gekomen omdat de Erlang ontwerpers een aantal veel voorkomende ontwerppatronen op een formele manier wilden ondersteunen. Het bestaan van behaviours is een voordeel voor Erlang/OTP, vooral voor het OTP deel. OTP kan er dan vanuit gaan dat bepaalde modules een bepaald contract volgen.

Dit is een overzicht van de meest voorkomende behaviours:

- `gen_server`

  Dit is een contract voor een algemene server. Deze server kan allerlei soorten berichten verwerken en is in staat om door middel van een proces zijn eigen toestand bij te houden. Als je dit contract volgt, hoef je zelf geen loop meer te schrijven.

- `supervisor`

  Dit is een opzichterproces dat als enige taak de bewaking van een ander proces uitvoert. Je hoeft voor dit type contract geen Erlang te schrijven. Het volstaat om de `supervisor` te configuren. Zo moet je onder andere vastleggen wat er moet gebeuren als het te bewaken proces crasht: herstarten of niet. Een supervisor is nuttig in complexe systemen waarin mogelijk nog processen door fouten kunnen crashen.

- `application`

  Dit is een bundeling van een aantal modules tot een applicatie. Erlang/OTP is in staat om applicaties te beheren. Om te weten hoe een applicatie moet gestart en gestopt kunnen worden, moet je dit contract volgen.

Het gebruik van behaviours is niet verplicht. In dit deel van de cursustekst wordt alleen maar de `appliction` behaviour uitgelegd en toegepast in een voorbeeld. De andere behaviours komen niet aan bod. Applicaties kunnen gebouwd worden zonder de `gen_server` en `supervisor` behaviours maar de `application` behaviour heb je wel altijd nodig.

Om een beter zicht te krijgen hoe software in Erlang/OTP in verschillende delen (modularisering) kan opgedeeld worden, is hier een overzicht van de faciliteiten die Erlang/OTP hiervoor aanbiedt.

- functie

  Een functie is een klein deel van de functionaliteit met een eigen naam.

- `module` (Erlang):

  Een module is een Erlang broncodebestand met `-module()`. Dit kan apart gecompileerd en geladen worden.

- `application` (Erlang/OTP)

  Dit is een bundeling van meerdere modules, met tenminste de `application` behaviour. Overige behaviours zijn niet verplicht. De `application` behaviour is dat wel. Een application kan apart gestart en gestopt worden. Alle modules die deel uit maken van een application worden dan samen gestart en gestopt.

- `release` (Erlang/OTP)

> Dit is een bundeling van een aantal applications samen met de nodige Erlang/OTP runtime tot een release. Een release kan als een aparte node gestart worden. Alle applications erin worden automatisch gestart. En er kan altijd een nieuwe versie van een release gestart worden. Een release kan ook gestart worden op een platform waar geen Erlang/OTP geïnstalleerd is.

Het is duidelijk dat een release het verst gaat. Het is zelfs mogelijk in Erlang/OTP om een nieuwe release te starten die de werking van een oude release overneemt. Dit betekent dat je een nieuwe versie van een softwaresysteem in gebruik kan nemen zonder dat het systeem eerst buiten dienst genomen wordt. Uiteraard is dit maar mogelijk indien de betrokken processen in staat zijn om deze overgang te maken. Door de structuur van Erlang/OTP systemen, processen met een lokale toestand, is dit niet zo moeilijk.

Het bouwen van een release werd oorspronkelijk gedaan met tools die als functies binnen Erlang/OTP gestart moeten worden. Deze werkwijze is omslachtig en wordt door bijna niemand meer toepast. Er zijn ondertussen nieuwe tools zoals `rebar` en `erlang.mk`/`relx` ontstaan die het bouwen van releases sterk vereenvoudigd hebben.

Om een zicht te krijgen op de structuur van applicaties starten we een heel eenvoudig voorbeeld.

2.2. Een eerste applicatie

Hier wordt de `simpeleteller` applicatie uitgelegd. Deze applicatie is echt minimaal groot zodat het snel duidelijk wordt, hoe je een Erlang/OTP applicatie in elkaar steekt. Er wordt in dit voorbeeld alleen maar gebruik gemaakt van de `application` behaviour. Hier blijft de applicatie overzichtelijk.

Deze `application` behaviour is wel verplicht maar in dit voorbeeld is de implementatie ervan minimaal en overzichtelijk. De applicatie heeft zijn eigen loop die om de seconde automatisch een teller verhoogt.

Om de applicatie te bouwen maken we gebruik van het commando met `erl -make`. De broncode van een applicatie moet volgens een bepaalde directorystructuur georganiseerd worden. Dit is tegenwoordig bij de meeste ontwerpsystemen in de verschillende programmeertalen en webframeworks het geval. In de Java wereld wordt Maven gebruikt als buildtool en daar werk je ook met een vaste directorystructuur.

Meer uitleg over Erlang/OTP applicaties vind je hier:

http://www.erlang.org/doc/design_principles/applications.html

Deze directories moeten bestaan:

```
src
ebin
priv
include
```

In `src` zet je de broncodebestanden, dat zijn de bestanden en extensie `.erl`. `priv` en `include` zijn optioneel. En in `ebin` komt het resultaat van de compilatie terecht. Dit zijn de `.beam` bestanden terecht. Dit zijn de binaire bestanden die ontstaan bij de compilatie.

In het voorbeeld zijn er enkel de directories `ebin` en `src`.

```
simpeleteller
|-- Emakefile
|-- ebin
|   `-- simpeleteller.app
```

```
`-- src
    |-- simpeleteller.erl
    `-- simpeleteller_app.erl
```

De `Emakefile` legt de configuratie voor de compilatie vast.

Dit is `./simpeleteller/Emakefile`.

```
{"src/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
```

Dit bestand heeft het formaat van een Erlang term. Dit is de gangbare werkwijze voor configuratiebestanden voor Erlang/OTP.

De applicatie wordt beschreven in `./simpeleteller/ebin/simpeleteller.app`. Aan de hand van dit configuratiebestand is Erlang/OTP in staat om de applicatie te beheren, namelijk het laden, starten en stoppen.

```
{application,simpeleteller,mod
  [
    {description,"Simpele teller"},
    {vsn,"0.0.1"},
    {modules,[simpeleteller, simpeleteller_app]},
    {applications,[kernel,stdlib]},
    {mod,{simpeleteller_app,[]}}
  ]
}.
```

De tuple `{application, ...}` beschrijft de applicatie. Na het atoom `application`, dat er altijd zo moet staan, volgt de naam van de applicatie. Hier is dat `simpeleteller`. Je kiest hier best een naam zonder spaties. Er volgt dan een lijst met extra opties. Met `description` wordt een beschrijving vast gelegd. En `vsn` legt een versienummer vast. Dit kan nuttig zijn om later afhankelijkheden tussen applicaties vast te leggen. `modules` bepaalt welke modules deel uitmaken van deze applicatie. Hier zijn dat `simpeleteller` en `simpeleteller_app`. Met `applications` wordt een lijst van applicaties vastgelegd die zeker moeten gestart worden voordat deze applicatie mag starten. In deze lijst kan je extra applicaties vermelden zoals de `observer`. Denk eraan dat je dan ook alle afhankelijkheden van `observer` moet vermelden. De enige overige applicaties zijn hier `kernel` en `stdlib`. Tot slot wordt er met `mod` vastgelegd welke de module is die het `application` behaviour heeft. Hier is dat de module `simpeleteller_app`. Het is via deze module dat de applicatie gestart wordt.

Nu volgt de module met `application` behaviour. Hierin wordt het start- en stoppunt vastgelegd. Het is de module ./simpeleteller/src/simpeleteller_app.erl

```
-module(simpeleteller_app).
-behaviour(application).

-export([start/2]).
-export([stop/1]).

start(_Type, _Args) ->
    simpeleteller:start().

stop(_State) ->
    ok.
```

Deze module heeft een `start/2` en een `stop/1` functie. Met deze functies wordt de applicatie gestart en gestopt. In `start/2` wordt de `start/0` van de andere module gestart.

In `./simpeleteller/src/simpeleteller.erl` staat de module die de teller bijhoudt. Deze module heeft een eigen proces. Het proces kan twee soorten berichten verwerken: `get` en `stop`, respectievelijk voor het ophalen van de waarde van de teller en het stopzetten van het proces.

```
-module(simpeleteller).

-export([start/0, stop/0]).

-export([init/0, loop/1, get/0]).

start() ->
    Pid = spawn(?MODULE, init, []),
    register(simpeleteller, Pid),
    {ok, Pid}.

stop() ->
    simpeleteller ! {stop}.

init() ->
    loop(0),
    {ok, []}.

get() ->
    simpeleteller ! {get, self()},
    receive
       Result -> Result
    end.

loop(Teller) ->
   receive
      {get, From} ->
         From ! Teller,
         loop(Teller);
      {stop} ->
         ok;
      _ ->
         loop(Teller)
   after
      1000 ->
         loop(Teller + 1)
   end.
```

Bij het starten wordt de pid van het proces geregistreerd onder de naam `simpeleteller`. Dit heeft het voordeel dat de pid niet als parameter moet meegegeven worden bij de functies `start/0`, `stop/0` en `get/0`. De `start/0` en `stop/0` functies worden vanuit de `application` module opgeroepen. Deze functies hoef je nooit zelf te starten. De functie `get/0` is wel bedoeld om van buiten de applicatie opgeroepen te worden.

De `loop` functie is verantwoordelijk voor de verwerking van de berichten. Er is een timeout voorzien van 1000 ms. Als er binnen deze tijd bericht binnenkomt, wordt de `after` verwerking gestart. Dit is de recursieve oproep `loop(Teller + 1)`. Hierdoor verandert de toestand. Alleen de teller wordt als toestand bijgehouden.

Nu is het tijd om de applicatie te testen. Je moet wel eerst de applicatie compileren. Dat doe je zo:

De applicatie wordt manueel geladen en gestart.

```
simpeleteller$ erl -make
Recompile: src/simpeleteller_app
Recompile: src/simpeleteller
```

De eigenlijke test verloopt zo:

```
simpeleteller$ erl
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe]

Eshell V6.1  (abort with ^G)
1> code:add_path("ebin").
true
2> application:which_applications().
[{stdlib,"ERTS  CXC 138 10","2.1"},
 {kernel,"ERTS  CXC 138 10","3.0.1"}]
3> application:load(simpeleteller).
ok
4> application:start(simpeleteller).
ok
5> application:which_applications().
[{simpeleteller,"Simpele teller","0.0.1"},
 {stdlib,"ERTS  CXC 138 10","2.1"},
 {kernel,"ERTS  CXC 138 10","3.0.1"}]
6> whereis(simpeleteller).
<0.41.0>
7> simpeleteller:get().
39
8> simpeleteller:get().
42
9> simpeleteller:get().
43
10>
```

Met `code:add_path("ebin").` wordt ingesteld dat de beam bestanden zich in de `ebin` directory bevinden. Met `application:which_applications().` kan je opvragen welke applications er momenteel draaien. Zoals gewenst zijn dat `kernel` en `stdlib`. Met `application:load(simpeleteller).` wordt de eigen `simpeleteller` applicatie geladen en met `application:start(simpeleteller).` wordt die gestart. Met een nieuwe oproep van `application:which_applications().` kan je verifiëren of dat zo is. Je kan dan nagaan of het proces met de geregistreerde naam `simpeleteller` wel bestaat. Dat doe je met `whereis(simpeleteller).`. En tenslotte kan je met `simpeleteller:get().` de waarde van de teller opvragen.

Het maken van een applicatie is niet zo moeilijk. In praktijk worden er aangepaste tools gebruikt om applicaties te bouwen. In het volgende voorbeeld wordt dit getoond.

2.3. Een applicatie bouwen met `erlang.mk` en `relx`

Voor het compileren van een applicatie en het bouwen van een bijbehorende release zijn de juiste tools noodzakelijk. De compilatie wordt geleid door `erlang.mk` en `relx` wordt gebruikt als buildtool. Een andere veel gebruikte tool is `rebar` maar wordt in deze cursustekst niet uitgelegd.

`erlang.mk` is een include bestand voor een `Makefile` en `relx` is een programma waarmee de release van de applicatie kan gebouwd worden.

Dit is de structuur van de applicatie:

```
teller
|-- Makefile
|-- rel
|   `-- vm.args
|-- erlang.mk
|-- relx.config
`-- src
    |-- teller.app.src
    |-- teller.erl
    `-- teller_app.erl
```

Een verschil met de structuur van het vorige voorbeeld is dat het bestand met de beschrijving nu `teller.app.src` heet (met een dubbele extensie) en dat het zich bevindt in de directory `src` en niet in `ebin`. Het is wel zo dat `erlang.mk` het bestand kopieert van `src` naar `ebin` waarbij de lijst met modules aangevuld wordt.

Om te kunnen compileren moet je zelf een korte `Makefile` maken. Hierin moet je o.a. een include opnemen. Dit `erlang.mk` include bestand moet je afhalen.

```
$ wget https://raw.github.com/extend/erlang.mk/master/erlang.mk
$ ls
src/
erlang.mk
```

Dit is `./teller/Makefile`.

```
PROJECT = teller

include erlang.mk
```

Het enige dat in deze `Makefile` staat, is de naam van het project en de include van `erlang.mk`. Door deze include blijft de `Makefile` heel eenvoudig en hoef je zeker geen moeite te doen om te achterhalen hoe de compilatie precies verloopt.

Nu kan je compileren met `make`.

```
$ make
```

Nadat `make` gelopen heeft, zie je dat de directories `ebin` en `_rel` erbij gekomen zijn. In `ebin` staan de `.beam` bestanden en in `_rel` staat de volledige release. Dit zijn de bestanden van de applicatie samen met alle bestanden die nodig zijn om Erlang op een platform te draaien. Je kan ze zo overlopen:

```
$ tree _rel
```

Heel de release bevindt zich in `_rel`; je kan deze bestanden eventueel kopiëren van het ontwikkelingsplatform naar het doelplatform.

We overlopen nu de afzonderlijke bestanden van de applicatie. Dit is `./teller/src/teller_app.erl`. Dit is het bestand met de `application` behaviour.

```
-module(teller_app).
-behaviour(application).

-export([start/2]).
```

```
-export([stop/1]).

start(_Type, _Args) ->
    teller:start().

stop(_State) ->
    ok.
```

Dit is `./teller/src/teller.erl` die het proces start waarin de teller beheerd wordt.

```

-module(teller).

-export([start/0, stop/0]).

-export([init/0, loop/1, get/0]).

start() ->
    Pid = spawn(?MODULE, init, []),
    register(teller, Pid),
    {ok, Pid}.

stop() ->
    teller ! {stop}.

init() ->
    loop(0),
    {ok, []}.

get() ->
    teller ! {get, self()},
    receive
       Result -> Result
    end.

loop(Teller) ->
   receive
      {get, From} ->
         From ! Teller,
         loop(Teller);
      {stop} ->
         ok;
      _ ->
         loop(Teller)
   after
      1000 ->
         loop(Teller + 1)
   end.
```

Dit is `./teller/src/teller.app.src`. Dit bestand staat nu in `src` en bevat een lege modulelijst. Deze lijst wordt automatisch aangevuld bij het maken van de release.

```

{application, teller, [
    {description, "Teller applicatie!"},
    {vsn, "0.1.0"},
    {modules, []},
    {registered, [teller]},
    {applications, [
```

```
        kernel,
        stdlib,
        runtime_tools,
        wx,
        observer
    ]},
    {mod, {teller_app, []}},
    {env, []}
]}.
```

Je ziet ook dat in de lijst van benodigde applicaties ook de applicaties `runtime_tools`, `wx` en `observer` zijn opgenomen. Dit is nodig om de `observer` te kunnen starten. Denk eraan dat hier een release wordt gebouwd die je naar een ander platform zonder Erlang installatie kan kopiëren. Daarom moet de `observer` expliciet opgenomen worden in de release.

Dit is `./teller/rel/vm.args`. Dit bestand is optioneel maar is wel nodig om gedistribueerd te kunnen werken. In dit bestand plaats je de naam van de node en de cookie van de cluster.

```
## Name of the node
-name teller@127.0.0.1

## Cookie for distributed erlang
-setcookie teller

## Heartbeat management; auto-restarts VM if it dies or becomes unresponsive
## (Disabled by default..use with caution!)
##-heart

## Enable kernel poll and a few async threads
##+K true
##+A 5

## Increase number of concurrent ports/sockets
##-env ERL_MAX_PORTS 4096

## Tweak GC to run more often
##-env ERL_FULLSWEEP_AFTER 10
```

Dit is `./teller/relx.config`. Hiermee wordt het bouwen van de release geconfigureerd.

```
{release, {teller, "1"}, [teller]}.
{extended_start_script, true}.
{vm_args, "./rel/vm.args"}.
```

De laatste regel met `vm_args` kan je eventueel weglaten als je geen gedistribueerde node wil maken. Na het bouwen met `make` kan je de release starten.

```
$ make
$ _rel/teller/bin/teller console
```

`teller` is hier een script voor het starten van de release. Met het argument `console` geef je aan dat je een `erl` prompt wil hebben. Met `teller help` kom je te weten welke andere argumenten dan `console` kunnen gebruikt worden om de release te starten, stoppen of beheren.

```
$ _rel/teller/bin/teller start
```

Nu is de release gestart en krijg je de bekende `erl` prompt. Je kan zo nagaan of de applicatie draait:

```
(teller@127.0.0.1)1> application:which_applications().
[{teller,"Teller applicatie!","0.1.0"},
 {observer,"OBSERVER version 1","2.0.1"},
 {wx,"Yet another graphics system","1.3"},
 {runtime_tools,"RUNTIME_TOOLS","1.8.14"},
 {stdlib,"ERTS  CXC 138 10","2.1"},
 {kernel,"ERTS  CXC 138 10","3.0.1"}]
```

Je ziet dat alle gewenste applicaties draaien. Je kan ook nagaan of het `teller` proces draait.

```
<teller@127.0.0.1)2> whereis(teller).
<0.47.0>
```

En je kan de waarde van de teller opvragen door `teller:get/0` op te roepen.

```
(teller@127.0.0.1)3> teller:get().
368
(teller@127.0.0.1)4> teller:get().
369
```

Je kan de release ook starten zonder console.

```
$ _rel/teller/bin/teller start
```

Dit is interessant voor platformen die bij een boot automatisch de release starten. En dan is dit de manier om de release stil te leggen

```
$ _rel/teller/bin/teller stop
```

Je kan applicaties met verschillende tools bouwen en van alle tools blijkt de combinatie `erlang.mk` en `make` het handigst te zijn. Eén van de redenen om deze combinatie te gebruiken is dat hiermee het inbouwen van Cowboy in een eigen applicatie relatief eenvoudig is. Dit tonen we met het volgende voorbeeld.

2.4. Cowboy integreren in een applicatie

In deze sectie wordt het voorgaand voorbeeld uitgebreid met de Cowboy webserver. Cowboy is een webserver die speciaal ontworpen is om geïntegreerd te worden in een bestaande Erlang applicatie. Omdat Cowboy een extra afhankelijkheid is voor de teller applicatie, kan je niet anders dan een tool zoals `erlang.mk` en `relx` te gebruiken. Deze tool neemt het oplossen van de afhankelijkheid voor zijn rekening.

In dit voorbeeld hebben we nog steeds het tellerproces dat om de seconde de teller verhoogt. Aan deze applicatie wordt een handler toegevoegd die opgeroepen wordt als er HTTP aanvragen via Cowboy binnenkomen. De taak van de handler is een HTML pagina samenstellen waarin de huidige waarde van de teller wordt verwerkt. Dit scenario is een typisch voorbeeld van hoe je een applicatie kan uitbreiden met een webinterface. In dit voorbeeld wordt Cowboy dus gebruikt om de waarde van de teller via een webinterface weer te geven.

Dit startervoorbeeld staat hier gedocumenteerd:

http://ninenines.eu/docs/en/cowboy/HEAD/guide/getting_started/

Dit is de structuur van de applicatie:

```
teller-cowboy
|-- Makefile
|-- erlang.mk
|-- relx.config
`-- src
    |-- teller.app.src
    |-- teller.erl
    |-- teller_app.erl
    `-- teller_handler.erl
```

Opnieuw zien we dat de applicatie een standaardstructuur heeft. Vermits dit een webapplicatie wordt, is er een afhankelijkheid van Cowboy. Je hoeft deze laatste niet apart te installeren. De gebruikte buildtools `erlang.mk` en `relx` zijn in staat om Cowboy automatisch te integreren in deze nieuwe applicatie. Het enige nieuwe bestand in deze structuur is `teller_handler.erl`. Dit is de handler voor de verwerking van de binnenkomende HTTP aanvragen.

`./teller-cowboy/Makefile` is de `Makefile` van het voorbeeld. Je ziet dat er een afhankelijkheid is van Cowboy.

```
PROJECT = teller

DEPS = cowboy

include erlang.mk
```

De regel die start met DEPS vermeldt alle afhankelijkheden. In het geval van Cowboy hoef je enkel maar de naam te vermelden. In `erlang.mk` zit de nodige logica om de broncode af te halen en dan apart te compileren. Als je een afhankelijkheid naar een voor `erlang.mk` onbekend project wil vermelden moet je ook de git URL vermelden. `erlang.mk` zal dan zelf met `git` de broncode afhalen en compileren.

Dit is `hello_erlang.app.src`, de beschrijving van de applicatie. Na compilatie duikt dit bestand op in `ebin/`. Dat hebben we in het vorige voorbeeld al gezien.

Dit is ./teller-cowboy/src/teller.app.src

```
{application, teller, [
    {description, "Teller applicatie!"},
    {vsn, "0.1.0"},
    {modules, []},
    {registered, [teller]},
    {applications, [
        kernel,
        stdlib,
        cowboy
    ]},
    {mod, {teller_app, []}},
    {env, []}
]}.
```

Je ziet dat `cowboy` opgenomen in de lijst van applicaties die samen met `teller` opgestart worden. De applicaties `kernel` en `stdlib` heb je voor bijna elke applicatie nodig. Je ziet ook dat de lijst van modules (`{modules, []}`) leeg mag blijven. Deze lijst wordt automatisch aangevuld door de buildtools. Met `{registered, [teller]}` wordt aangegeven welke module een procesid laat

registreren. Met `{mod, {teller_app, []}}` wordt aangegeven dat de module `teller_app` als applicationmodule gestart wordt; dit is de module met het `application` behaviour.

Dit is `./teller-cowboy/relx.config`.

```
{release, {teller, "1"}, [teller]}.
{extended_start_script, true}.
```

Dit is `./teller-cowboy/src/teller.erl` Hier is niets aan gewijzigd.

```
-module(teller).

-export([start/0, stop/0]).

-export([init/0, loop/1, get/0]).

start() ->
    Pid = spawn(?MODULE, init, []),
    register(teller, Pid),
    {ok, Pid}.

stop() ->
    teller ! {stop}.

init() ->
    loop(0),
    {ok, []}.

get() ->
    teller ! {get, self()},
    receive
       Result -> Result
    end.

loop(Teller) ->
   receive
      {get, From} ->
         From ! Teller,
         loop(Teller);
      {stop} ->
         ok;
      _ ->
         loop(Teller)
   after
      1000 ->
         loop(Teller + 1)
   end.
```

Cowboy vraagt een HTTP handler. Deze verwerkt de binnenkomende HTTP aanvragen. Deze handler staat in `./teller-cowboy/src/teller_handler.erl`.

```
-module(teller_handler).
-behaviour(cowboy_http_handler).

-export([init/3]).
-export([handle/2]).
```

```
-export([terminate/3]).

init(_Type, Req, _Opts) ->
   {ok, Req, undefined_state}.

handle(Req, State) ->
   Teller = teller:get(),
   Body = [<<"<html>
<head>
        <meta charset=\"utf-8\">
        <title>Teller!</title>
</head>
<body>
  <h1>Teller</h1>
  <p>De tellerwaarde is ">>,  io_lib:format("~p",[Teller]), <<"</p>
  <p>Herlaad om de nieuwe waarde van de teller te zien.</p>
</body>
</html>">>],
    {ok, Req2} = cowboy_req:reply(200,
      [
        {<<"content-type">>, <<"text/html">>}
      ], Body, Req),
    {ok, Req2, State}.

terminate(_Reason, _Req, _State) ->
    ok.
```

Deze module gebruikt de `cowboy_http_handler` behaviour. Dit is verplicht voor een HTTP handler. De `handle` functie wordt opgeroepen wanneer er een HTTP aanvraag binnenkomt. Alle informatie over de aanvraag zit in de `Req` variabele. Hiermee kan je het antwoord opbouwen. Dit staat dan in `Req2` dat samen met de ongewijzigde `State` als resultaat wordt teruggegeven. Het antwoord wordt samengesteld met de hulpfunctie `cowboy_req:reply/4`. De waarde 200 is een HTTP antwoordcode.

In de HTTP handler wordt een HTML pagina opgesteld en teruggegeven. De variabele `Body` krijgt een toekenning van de binaire string die de volledige HTML bevat. In deze HTML wordt de waarde van de teller opgenomen. In de string zijn newlines opgenomen. Dat betekent dat deze newlines mee naar de browser worden verstuurd. Met `io_lib:format/1` wordt de string met de teller opgebouwd. De waarde van de teller wordt opgevraagd met `` `teller:get/0 ``.

Dit volgende bestand is `teller_app.erl`, de module met het `application` behaviour. Hierin wordt nu ook de HTTP server gestart. Deze module moet de functies `start/2` en `stop/1` bevatten. Dit is opgelegd door de `application` behaviour.

Dit is ./teller-cowboy/src/teller_app.erl

```
-module(teller_app).
-behaviour(application).

-export([start/2]).
-export([stop/1]).

start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        %% {URIHost, list({URIPath, Handler, Opts})}
        {'_', [{'_', teller_handler, []}]}
    ]),
    %% Name, NbAcceptors, TransOpts, ProtoOpts
```

```
    cowboy:start_http(my_http_listener, 100,
        [{port, 8080}],
        [{env, [{dispatch, Dispatch}]}]
    ),
    teller:start().

stop(_State) ->
    ok.
```

`src/teller_app.erl` is de module met de `application` behaviour. Deze module is verantwoordelijk voor het opstarten van de `teller` module. Dat zie je als laatste actie in de `start/2` functie: de `teller` module wordt gestart met `teller:start()`.

Vooraf wordt Cowboy gestart. Eerst wordt de routinglijst gecompileerd. Dit is de lijst van URL's die door de webserver kunnen verwerkt worden. Deze lijst gebruikt zijn eigen syntax voor de patroonherkenning en daarom moet die gecompileerd worden. De underscore doet er dienst als jokerteken.

Per patroon wordt ook telkens de naam van de module vermeld die voor de verwerking zorgt. De modulenaam alleen volstaat omdat deze module het `cowboy_http_handler` behaviour volgt.

Het compileren van de URL patronen wordt met `cowbow_router:compile/1` uitgevoerd. Het resultaat van deze compilatie is een dispatcher. Deze dispatcher verzorgt de routing. Hiermee wordt bedoeld op welke wijze een bepaalde HTTP aanvraag wordt geleid naar een bepaalde handler. Dit mechanisme voor de doorverwijzingen noemt men routing. Met de notatie `{'_', [{'_', teller_handler, []}]}` wordt aangegeven dat een aanvraag bij om het even welke host (eerste `'_'` wildcard) met om even welke URL (tweede `'_'` wildcard) door de handler `teller_handler` verwerkt wordt. Je kan meerdere patronen met elk een eigen handler voorzien maar in dit eenvoudige voorbeeld is dat niet gedaan.

Meer informatie over routing vind je bij http://ninenines.eu/docs/en/cowboy/HEAD/guide/routing/. De handleiding van Cowboy staat op http://ninenines.eu/docs/en/cowboy/HEAD/guide/.

Cowboy kan gemakkelijk geïntegreerd worden in een bestaande applicatie. Het aanmaken van de HTML is wel omslachtig omdat de HTML code geïntegreerd moet zijn in de Erlang broncode. Bij moderne webframeworks is er een scheiding tussen de broncode van de gebruikte programmeertaal en de HTML. In Cowboy is dit ook mogelijk als je gebruik maakt van ErlyDTL.

2.5. ErlyDTL bijvoegen in een Cowboy applicatie

In deze stap is het voorgaande voorbeeld herwerkt zodat de gegenereerde HTML niet meer vanuit een Erlang functie wordt gemaakt maar met een ErlyDTL template. Deze stap is een echte vereenvoudiging.

Het genereren van de HTML is herzien. Dit gebeurt nu niet meer vanuit een Erlang functie maar er wordt een sjablonensysteem ErlyDTL in gebruik genomen. Dit systeem heeft overeenkomsten met sjablonensystemen die in andere frameworks zoals Play en Django bestaan.

ErlyDTL (Erlang implementation of the Django Template Language) is speciaal voor Erlang ontworpen en maakt gebruik van de sjablonentaal van Django. Deze laatste is een webframework voor Python.

Het bijvoegen van ErlyDTL is de enige wijziging ten opzicht van het vorige voorbeeld.

Dit is de structuur van de applicatie:

```
teller-cowboy-template/
|-- Makefile
```

```
|-- erlang.mk
|-- relx.config
|-- src
|   |-- teller.app.src
|   |-- teller.erl
|   |-- teller_app.erl
|   `-- teller_handler.erl
`-- templates
    `-- index.dtl
```

Je ziet dat de map `templates` erbij gekomen is. Hierin staat de pagina die getoond zal worden. De extensie `.dtl` wijst op het formaat van ErlyDTL. Een ErlyDTL sjabloon wordt bij de compilatie automatisch gecompileerd naar Erlang bytecode.

De `Makefile` is licht gewijzigd. De afhankelijkheid van ErlyDTL is bijgevoegd. Dit betekent dat bij de build ErlyDTL automatisch afgehaald wordt en na compilatie bijgevoegd wordt in het project. Waar precies de broncode van ErlyDTL broncode afgehaald moet worden, hoef je niet te vermelden.

Dit is `./teller-cowboy-template/Makefile`.

```
PROJECT = teller

DEPS = cowboy erlydtl

include erlang.mk
```

Ook in de releaseconfiguratie `relx.config` wordt `erlydtl` bijgevoegd.

Dit is `/teller-cowboy-template/src/teller.app.src`.

```
{application, teller, [
    {description, "Teller applicatie!"},
    {vsn, "0.1.0"},
    {modules, []},
    {registered, [teller]},
    {applications, [
        kernel,
        stdlib,
        cowboy,
        erlydtl
    ]},
    {mod, {teller_app, []}},
    {env, []}
]}.
```

Dit is `./teller-cowboy-template/relx.config`.

```
{release, {teller, "1"}, [teller]}.
{extended_start_script, true}.
```

De implementatie van de teller in `./teller-cowboy-template/src/teller.erl` is ongewijzigd maar wordt toch vermeld voor de volledigheid.

```
-module(teller).
```

```
-export([start/0, stop/0]).

-export([init/0, loop/1, get/0]).

start() ->
    Pid = spawn(?MODULE, init, []),
    register(teller, Pid),
    {ok, Pid}.

stop() ->
    teller ! {stop}.

init() ->
    loop(0),
    {ok, []}.

get() ->
    teller ! {get, self()},
    receive
       Result -> Result
    end.

loop(Teller) ->
   receive
      {get, From} ->
         From ! Teller,
         loop(Teller);
      {stop} ->
         ok;
      _ ->
         loop(Teller)
   after
      1000 ->
         loop(Teller + 1)
   end.
```

Ook het opstarten met `src/teller_app.erl` is ongewijzigd. Het gebruik van ErlyDTL kan je hier niet zien.

Dit is ./teller-cowboy-template/src/teller_app.erl

```

-module(teller_app).
-behaviour(application).

-export([start/2]).
-export([stop/1]).

start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        %% {URIHost, list({URIPath, Handler, Opts})}
        {'_', [{'_', teller_handler, []}]}
    ]),
    %% Name, NbAcceptors, TransOpts, ProtoOpts
    cowboy:start_http(my_http_listener, 100,
        [{port, 8080}],
        [{env, [{dispatch, Dispatch}]}]
    ),
    teller:start().
```

```
stop(_State) ->
    ok.
```

In de handler `./teller-cowboy-template/src/teller_handler.erl` is er wel een verschil. De HTML code is verdwenen en is vervangen door een oproep van een functie `render/1` van de door ErlDTL gegenereerde module. Dat ziet er zo uit:

```
{ok, Body} = index_dtl:render([{waarde, Teller}]),
```

Door de compilatie van de `index.dtl` pagina krijg je automatisch een gegenereerde `index_dtl` module. Deze module heeft een `render/1` functie waarmee je de HTML kan ophalen. Aan deze functie moet je een lijst van waarden doorgeven die in de sjabloon kunnen gebruikt worden. De waarde van de teller wordt doorgegeven met `{waarde, Teller}`. Hierdoor kan je in de sjabloon de schrijfwijze `{{ waarde }}` gebruiken. Deze notatie zal vervangen worden door de waarde die meegegeven wordt.

```
-module(teller_handler).
-behaviour(cowboy_http_handler).

-export([init/3]).
-export([handle/2]).
-export([terminate/3]).

init(_Type, Req, _Opts) ->
   {ok, Req, undefined_state}.

handle(Req, State) ->
   Teller = teller:get(),
   {ok, Body} = index_dtl:render([{waarde, Teller}]),
   Headers = [{<<"content-type">>, <<"text/html">>}],
   {ok, Req2} = cowboy_req:reply(200, Headers, Body, Req),
   {ok, Req2, State}.

terminate(_Reason, _Req, _State) ->
    ok.
```

Tot slot is dit de HTML pagina `templates/index.dtl`. Hierin zie je de notatie `{{ waarde }}`. Hiermee geef je aan dat een dynamisch bepaalde waarde in HTML moet opgenomen worden.

Dit is `./teller-cowboy-template/templates/index.dtl`.

```
<html>
  <head>
    <title>Teller!</title>
  </head>
  <body>
    <p>De teller is nu {{ waarde }}.</p>
  </body>
</html>
```

Meer uitleg over de syntax van ErlyDTL vind je bij https://docs.djangoproject.com/en/dev/ref/templates/builtins/.

Door ErlyDTL in te zetten ontstaat er een scheiding van belangen. De HTML die getoond wordt, staat nu los van de Erlang logica. Dit maakt de webapplicatie beter onderhoudbaar en zeker sneller

geschreven. Om toch de HTML dynamisch aanpasbaar te maken, kan je vanuit Erlang de waarden doorgeven door ErlyDTL variabelen vast te leggen.

De uitbreiding van Erlang applicaties met Cowboy en ErlyDTL is relatief voor de hand liggend en heeft als voordeel dat je complexe Erlang applicaties gemakkelijk kan voorzien van een webinterface. Deze webinterface kan dienen voor de bediening van de applicatie maar het is ook een goede strategie om de een webinterface als testtool te gebruiken. Op deze wijze kan je de interne toestand van een draaiend systeem zichtbaar maken.

3. Extra's

Hier zijn nog enkele extra's die los staan van de thema's die in de vorige secties gepresenteerd werden.

3.1. Native implemented functions (NIF's)

De eerste extra legt uit hoe je C functies kan integreren in een Erlang programma. Soms kan je niet anders dan een bepaalde functionaliteit in C te ontwerpen. In het volgende voorbeeld, dat uit de Erlang handleiding komt, wordt getoond hoe je een C functie vanuit Erlang kan starten en hoe je het resultaat kan verwerken.

Meer uitleg over dit onderwerp vind je op deze pagina: http://www.erlang.org/doc/tutorial/nif.html.

Het voorbeeld bestaat uit meerdere broncode bestanden. Het eerste bestand is `complex6.erl`. Dit is het Erlang deel dat toelaat om de C functies te bereiken. Door deze module kan je de C functies oproepen vanuit Erlang met `complex6:foo/1` en `complex6:bar/1`. Deze module heeft een `-on_load` aanduiding. Dit zorgt ervoor dat de C functie dynamisch geladen wordt. Als het laden lukt, worden de dummy implementaties die in deze module staan, vervangen door de echte implementaties. Als het laden mislukt, blijven de dummies actief en krijg je altijd het atoom `nif_library_not_loaded` als resultaat

```
-module(complex6).
-export([foo/1, bar/1]).
-on_load(init/0).

init() ->
    ok = erlang:load_nif("./complex6_nif", 0).

foo(_X) ->
    exit(nif_library_not_loaded).
bar(_Y) ->
    exit(nif_library_not_loaded).
```

Dit is `complex.c`. Deze C module bevat de implementaties van de twee functies. Merk op dat er een zijeffect is door het bestaan van de globale variabele `glob`. Dit is een gedrag dat je in C niet kan uitsluiten. Het woord `static` zorgt ervoor dat de variabele `glob` niet vanuit een andere C module kan bereikt worden. Dit is een afscherming op C module niveau.

Beide C functies zijn eenvoudig en keren onmiddellijk terug met hun resultaat. `foo()` plaatst een waarde in de globale variabele en `bar()` haalt die waarde op.

```
static int glob=0;

int foo(int x)
{
   glob = x;
```

```
   return x + 1;
}

int bar(int y)
{
   return glob;
}
```

Dit is `complex6_nif.c`. Deze C module bevat de logica die de koppeling tussen Erlang en de voorgaande module verzorgt. Doordat de interne werking van de Erlang wereld totaal verschillend is van die van de C wereld, is de aanpassing van Erlang en C nogal complex.

```
#include <erl_nif.h>

extern int foo(int x);
extern int bar(int y);

static ERL_NIF_TERM foo_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]
{
   int x, ret;

   if (!enif_get_int(env, argv[0], &x))
   {
      return enif_make_badarg(env);
   }
   ret = foo(x);
   return enif_make_int(env, ret);
}

static ERL_NIF_TERM bar_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]
{
   int y, ret;

   if (!enif_get_int(env, argv[0], &y))
   {
      return enif_make_badarg(env);
   }
   ret = bar(y);
   return enif_make_int(env, ret);
}

static ErlNifFunc nif_funcs[] =
{
    {"foo", 1, foo_nif},
    {"bar", 1, bar_nif}
};

ERL_NIF_INIT(complex6, nif_funcs, NULL, NULL, NULL, NULL)
```

We zullen de elementen van deze module stap voor stap overlopen.

Eerst wordt de include geschreven die ervoor zorgt dat we de ERL_NIF C macro's kunnen gebruiken.

```
#include <erl_nif.h>
```

Daarna vermelden we de `extern` aanduidingen. Met `extern` geef je aan dat je een functie uit een andere C module wil gebruiken. De linker zorgt ervoor dat uiteindelijk de juiste koppeling ontstaat.

```
extern int foo(int x);
extern int bar(int y);
```

Dit is het prototype van de `foo_nif` functie. Deze functie is `static`; dit betekent dat ze niet bekend is buiten de module. Als terugkeertype wordt `ERL_NIF_TERM` vermeld. Dit is een macro dat een type voorstelt waarmee je een Erlang term kan bijhouden. De parameters zijn:

- `ErlNifEnv* env`: een pointer naar een omgeving waarin extra informatie over de oproep opgeslagen is.
- `argc`: het aantal parameters dat bij de oproep van `foo()` doorgegeven wordt.
- `ERL_NIF_TERM argv[]`: een array met daarin zoveel parameters als `argv` aangeeft.

Deze complexe manier van parameteroverdracht laat een variabel aantal parameters toe.

```
static ERL_NIF_TERM foo_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]
```

Het eerste wat er in de functie gebeurt, is nagaan of de Erlang term die als eerste parameter meegegeven wordt wel een geheel getal is. Dat wordt zo getest:

```
   if (!enif_get_int(env, argv[0], &x))
   {
      return enif_make_badarg(env);
   }
```

Indien dat zo is, dan wordt de `int` waarde in de variabele `x` opgeslagen. Met deze waarde kan dan `foo()` opgeroepen worden:

```
ret = foo(x);
```

Met de returnwaarde van `foo()` wordt een Erlang term aangemaakt die een geheel getal voorstelt. Dit doen we met `enif_make_int()`.

```
   return enif_make_int(env, ret);
```

Dan volgt er een tabel met telkens de naam van de functie, het aantal parameters en het adres van de functie. Denk eraan dat in C de naam van een functie dienst doet als het adres van die functie.

```
static ErlNifFunc nif_funcs[] =
{
    {"foo", 1, foo_nif},
    {"bar", 1, bar_nif}
};
```

Deze tabel wordt op de volgende wijze doorgegeven aan de Erlang runtime.

```
ERL_NIF_INIT(complex6, nif_funcs, NULL, NULL, NULL, NULL)
```

Met de volgende `Makefile` kan alles gecompileerd worden. Het C bestand wordt gecompileerd tot `.so`. Dit is het formaat in Linux voor dynamisch laadbare modules.

```
all: complex6_nif.so complex6.beam
```

```
complex6_nif.so: complex.c complex6_nif.c
    gcc -o complex6_nif.so -fpic -shared -I/usr/lib64/erlang/erts-6.1/include/ c

complex6.beam: complex6.erl
    erlc complex6.erl

clean:
    rm -vf *.so
    rm -vf *.beam
```

Door `make` te starten, wordt alles gecompileerd.

```
$ make
gcc -o complex6_nif.so -fpic -shared -I/usr/lib64/erlang/erts-6.1/include/ compl
erlc complex6.erl
```

En zo kan je de functies testen.

```
1> c(complex6).
{ok,complex6}
2> complex6:foo(7).
8
3> complex6:bar(12).
7
```

Helaas zijn er wel gevaren aan deze techniek om C functies in Erlang systemen te integreren. Het eerste gevaar is dat als de C functie een runtime fout begaat, de hele node zal crashen. De Erlang runtime kan geen fouten in C functies onderscheiden en crasht daarom in zijn geheel. Daarom kan deze techniek fataal zijn voor de werking van een node. Je moet echt zeker zijn dat er geen fouten in het C gedeelte zijn. Voor eenvoudige C functies is het niet zo moeilijk om alle fouten eruit te krijgen maar als het C gedeelte complexer is, is de kans op fouten veel groter en daardoor ook de kans op een nodecrash.

Een bijkomend probleem is dat de C functie een korte uitvoeringstijd moet hebben. Erlang processen worden door de Erlang scheduler beheerd maar C functies niet. Een C functie die te lang duurt kan de werking van lopende Erlang processen te lang onderbreken. Het gedrag van de Erlang node wordt dan onvoorspelbaar. Je mag dus vanuit C geen API functies oproepen die blokkeren. Als je zoiets van plan bent, moet je het ontwerp aanpassen. Functionaliteit die onvoorspelbaar lang moet wachten, hoort niet thuis in C maar wel in Erlang. En als je die functionaliteit niet in Erlang kan schrijven, dan moet je de Erlang node op een of andere manier laten communiceren met een dienst die dan niet in Erlang geschreven is. Een mogelijke oplossing is een C node. Dit is een node die volledig in C geschreven is en die in staat is om mee te dialogeren met andere Erlang nodes.

3.2. C port

Een tussenvorm tussen NIF en C node is de C port. Bij deze oplossing draait de Erlang node in één of meerdere Linux processen en draait de C port in een eigen Linux proces. Het voordeel dat je zo verschillende crashing zones krijgt. Dat is hetzelfde als twee bergbeklimmers die niet met een touw verbonden zijn. Als één bergbeklimmer crasht, zal de andere nog overeind blijven.

Het volgende voorbeeld is eigenlijk een voorbeeld dat voor de Raspberry Pi is ontworpen. In dit voorbeeld ontbreekt het C gedeelte dat verantwoordelijk is voor de koppeling met de IO pinnen van de Raspberry Pi. Met het voorbeeld kan je vanuit Erlang een led in- en uitschakelen.

Vanuit de volgende C module wordt via de RTDM bibliotheek de led aangesproken. Hoe dat precies gebeurt, wordt niet uitgelegd. Er wordt wel uitgelegd hoe deze C module communiceert met Erlang. Er wordt daarvoor gebruik gemaakt van de zogenaamde `erl_interface`. Dit is een bibliotheek

die in losstaande C programma's kan gebruikt worden om de data die van en naar een Erlang node verstuurd wordt te decoderen en te coderen.

Dit is `gpio-port-led7.c`.

```
//#define RT


#include <stdlib.h>
#include <stdio.h>

#include <rtdm/rtdm.h>

#include <erl_interface.h>


typedef unsigned char byte;


#define DEVICE_NAME     "led-device"


// erl_interface hulpfuncties

int read_cmd(byte *buf)
{
   int len;

   if (read_exact(buf, 2) != 2)
      return(-1);
   len = (buf[0] << 8) | buf[1];
   return read_exact(buf, len);
}

write_cmd(byte *buf, int len)
{
   byte li;

   li = (len >> 8) & 0xff;
   write_exact(&li, 1);

   li = len & 0xff;
   write_exact(&li, 1);

   return write_exact(buf, len);
}

int read_exact(byte *buf, int len)
{
   int i, got=0;

   do
   {
      if ((i = read(0, buf+got, len-got)) <= 0)
         return(i);
      got += i;
   } while (got<len);
```

```
   return(len);
}

int write_exact(byte *buf, int len)
{
   int i, wrote = 0;

   do
   {
      if ((i = write(1, buf+wrote, len-wrote)) <= 0)
         return (i);
      wrote += i;
   }  while (wrote<len);

   return (len);
}




unsigned int led_wr(long long periode)
{
   char         buf[1024];
   ssize_t      size;
   int          device;
   int          ret;
   unsigned int teller;

   // open het device
   device = rt_dev_open(DEVICE_NAME, 0);
   if (device < 0)
   {
      printf("ERROR : can't open device %s (%s)\n",
      DEVICE_NAME, strerror(-device));
      exit(1);
   }

   // printf("periode %ld\n", periode);
   size = rt_dev_write (device, (const void *) &periode, sizeof(long long));
   // printf("Write to device %s\t: %d bytes\n", DEVICE_NAME, size);

   size = rt_dev_read (device, (void *) &teller, sizeof(unsigned int));
   //printf("Read in device %s\t: size %d, teller %ld\n", DEVICE_NAME, size, te

   // sluit het device
   ret = rt_dev_close(device);
   if (ret < 0)
   {
      printf("ERROR : can't close device %s (%s)\n",
         DEVICE_NAME, strerror(-ret));
      exit(1);
   }

   return teller;
}

int main()
```

```
{
   ETERM *tuplep;
   ETERM *intp;
   ETERM *fnp;
   ETERM *argp;
   unsigned int teller;
   byte buf[100];
   long allocated, freed;

   erl_init(NULL, 0);

   while (read_cmd(buf) > 0)
   {
      tuplep = erl_decode(buf);
      fnp    = erl_element(1, tuplep);
      argp   = erl_element(2, tuplep);

      if (strncmp(ERL_ATOM_PTR(fnp), "periode", 7) == 0)
      {
         int periode = ERL_INT_VALUE(argp);
         teller = led_wr(periode);
      }

      intp = erl_mk_uint(teller);
      erl_encode(intp, buf);
      write_cmd(buf, erl_term_len(intp));

      erl_free_compound(tuplep);
      erl_free_term(fnp);
      erl_free_term(argp);
      erl_free_term(intp);
   }
}
```

De bovenstaande C module wordt maar gedeeltelijk uitgelegd. De functies `read_cmd()`, `write_cmd()`, `read_exact()` en `write_exact()` zijn hulpfuncties die je zo uit andere voorbeelden kan overnemen. De werking kan je zelf ontcijferen. De functie `led_wr()` schrijft een knipperperiode naar de led en wordt ook niet uitgelegd.

De `main()` functie wordt wel uitgelegd. Je ziet dat er voor elke Erlang term een `ETERM *` pointer wordt gemaakt. In `buf` komen de bytes van de ontvangen term terecht.

Het volgende fragment verloopt in drie stappen.

```
tuplep = erl_decode(buf);
fnp    = erl_element(1, tuplep);
argp   = erl_element(2, tuplep);
```

De eerste stap decodeert de buffer naar een tuple. De tweede stap haalt het eerste element van de tuple en de derde stap haalt het tweede element uit de tuple. Het formaat van de doorgestuurde term is `{periode, Periode}`. Het eerste element is een atoom en het tweede element moet een integer zijn.

Met `ERL_ATOM_PTR` kan je de `char *` pointer opvragen die naar de string van het atoom wijst. Er wordt vergeleken met de string constante `"periode"`. Indien dit overeenkomt, wordt de integer waarde gedecodeerd met `ERL_INT_VALUE`. Deze periode wordt dan doorgegeven aan de led.

```
if (strncmp(ERL_ATOM_PTR(fnp), "periode", 7) == 0)
```

```
{
   int periode = ERL_INT_VALUE(argp);
   teller = led_wr(periode);
}
```

De teller die je dan terugkrijgt van de led wordt dan naar een integer term gecodeerd met het volgende fragment:

```
intp = erl_mk_uint(teller);
erl_encode(intp, buf);
write_cmd(buf, erl_term_len(intp));
```

In de eerste stap wordt een nieuwe integer term aangemaakt. Je krijgt de pointer naar de term terug. In de tweede stap wordt term omgevormd tot een reeks bytes in een buffer. En de derde stap stuurt de buffer door naar de Erlang node. De communicatie tussen de Erlang node en de C port verloopt via een pipe. Dit is een communicatiemiddel tussen twee Linux processen. Hier is een voorbeeld van twee processen die met een pipe gekoppeld zijn:

```
ls -l | less
```

De rechte streep wordt het pipeteken genoemd.

Deze module is `led_client.erl` en kan je gebruiken om het geheel te testen.

```
-module(led_client).
-export([setPeriode/1]).

setPeriode(Tijd) ->
   call_port({periode, Tijd}).

call_port(Msg) ->
   {led, ledserver@raspberrypi14} ! {call, self(), Msg},
   io:format("receive start~n"),
   receive
      {teller, Result} ->
         io:format("receive ok~n"),
         Result
   end.
```

Doe vanuit de `erl` prompt een oproep van `led_client:setPeriode/1` en `call_port/1` zal het bericht doorsturen naar een andere Erlang node. Als je alles op dezelfde node wil testen, moet je `{led, ledserver@raspberrypi14}` door `led` vervangen. Na het verzenden van het bericht wordt er gewacht op het antwoord.

Dit is `led.erl`. Deze module is verantwoordelijk voor het opstarten van de C port, die zoals eerder vermeld, in een eigen Linux proces draait.

```
-module(led).
-export([start/0, init/1, setPeriode/1]).

start() ->
   spawn(?MODULE, init, ["./gpio-port-led7"]).

init(ExtPrg) ->
   register(led, self()),
   process_flag(trap_exit, true),
   Port = open_port({spawn, ExtPrg}, [{packet, 2}, binary]),
```

```
   loop(Port).

setPeriode(Tijd) ->
   call_port({periode, Tijd}).

call_port(Msg) ->
   led ! {call, self(), Msg},
   receive
      {teller, Result} ->
         Result
   end.

loop(Port) ->
   receive
      {call, Caller, Msg} ->
         Port ! {self(), {command, term_to_binary(Msg)}},
         receive
            {Port, {data, Data}} ->
               Caller ! {teller, binary_to_term(Data)}
         end,
         loop(Port)
   end.
```

De C port wordt gestart met deze oproep:

```
Port = open_port({spawn, ExtPrg}, [{packet, 2}, binary]),
```

De variabele `ExtPrg` heeft op dat moment de waarde `"./gpio-port-led7"`. Dit is de naam van het programma dat als C port draait. Dit programma wordt automatisch gestart zodat de integratie van de C port binnen het Erlang gedeelte probleemloos verloopt. De `loop/1` functie ontvangt berichten van klanten en stuurt die door naar de C port. Je ziet dat het doorsturen van berichten naar de C port op dezelfde wijze verloopt als het doorsturen van berichten naar gewone Erlang processen. De gewone uitroeptekennotatie wordt gebruikt:

```
Port ! {self(), {command, term_to_binary(Msg)}},
```

De `command` tuple is wel verplicht en het door te sturen bericht, hier `Msg`, moet omgevormd worden van term naar binair. Alleen dan kan het als een reeks bytes doorgestuurd worden naar de C port. Deze omvorming met `term_to_binary/1` is vergelijkbaar met de serialisatie van objecten in Java.

Dit voorbeeld toont een complexere oplossing die de koppeling tussen Erlang en C mogelijk maakt. Eventueel kan je overschakelen naar een volledige C node. Deze node moet je dan zelf starten. Voor de C node moet je extra functionaliteit bijvoegen voor het ontvangen en zenden van berichten.

4. Oplossingen

4.1. Labo 1

```
-module(faculteit).
-export([fac/1, factail/1, fibonacci/1,
         elementLijst/2, kopie/1,
         verwijder/2]).

% Faculteit zonder staartrecursie
fac(0) ->
```

```
    1;
fac(X) ->
    X*fac(X-1).

% Faculteit met staartrecursie
factail(X) ->
    factail(X, 1).

factail(0, Acc) ->
    Acc;
factail(X, Acc) ->
    Acc2 = X * Acc,
    factail(X-1, Acc2).

% Bepaal het getal uit de reeks van Fibonacci
% op plaats X
fibonacci(X) ->
    fibonacci(X, 0, 1).

fibonacci(0, _V, Acc) ->
    Acc;
fibonacci(N, V, Acc) ->
    fibonacci(N-1, Acc, V+Acc).

% Geef het Nde element van een lijst terug
% element(N, Lijst)
elementLijst(_N,[]) ->
    fout;
elementLijst(0, [E|_Rest]) ->
    E;
elementLijst(N, [_E|Rest]) ->
    elementLijst(N-1, Rest).


% Maak een kopie van een lijst
%   met optie: verdubbel elke element
kopie(Lijst) ->
    L = kopie(Lijst, []),
    lists:reverse(L).

kopie([], Acc) ->
    Acc;
kopie([E|Rest], Acc) ->
    kopie(Rest, [2*E|Acc]).


% Verwijder het Nde element van een lijst
%    verwijder(2, [aa,bb,cc,dd])
% geeft
%    [aa, bb, dd]
% Deze versie heeft nog geen staartrecursie
verwijder(_N,[]) ->
    fout;
verwijder(0, [_E|Rest]) ->
    Rest;
verwijder(N, [E|Rest]) ->
    L = verwijder(N-1, Rest),
    [E|L].
```