Ein paa Erlang Beispielprogramme, von Ulrich Grude.

1 Ein ausführlich kommentiertes Hallo-Programm

```
 1 % Datei hallo01.erl     % Erlang-Quelldatei
 2 -module (hallo01).      % Der Modulname muss zum Dateinamen passen
 3 -compile(export_all).   % Pauschaler Export aller Funktionen dieses Moduls
 4 % ---------------------------------------------------------------------
 5 sagHallo() ->
 6    io:format("Hallo Welt!~n").
 7 % ---------------------------------------------------------------------
 8 sagHallo(Name) ->
 9    gruss(),
10    io:format("~s! Wie geht's?~n", [Name]).
11 % ---------------------------------------------------------------------
12 gruss() ->
13    io:format("Hallo ").
14 % ---------------------------------------------------------------------
15 % Diese Datei von einer Kommandozeile aus (DOS-Prompt, bash-Shell, ...)
16 % compilieren und die Funktionen sagHallo starten. Voraussetzungen:
17 % 1. Diese Datei liegt im aktuellen Arbeitsverzeichnis
18 % 2. Die Programme erlc.exe und erl.exe sind ueber die Umgebunsvariable
19 %    PATH direkt erreichbar.
20 % Das Promptzeichen der Kommandozeile wird hier durch > angedeutet.
21 %
22 % > erlc hallo01.erl
23 % > erl -noshell -s hallo01 sagHallo -s init stop
24 % > erl -noshell -s hallo01 sagHallo richard -s init stop
25 % ---------------------------------------------------------------------
26 % Eine Erlang-Shell starten und von der aus diese Datei
27 % compilieren und die Funktion sagHallo starten. Voraussetzung:
28 % 1. Wir haben eine Kommandozeile gestartet und diese Datei liegt im
29 %    aktuellen Arbeitsverzeichnis dieser Kommandozeile.
30 % 2. Die Programme erlc.exe und werl.exe (oder erl.exe) sind ueber die
31 %    Umgebunsvariable PATH direkt erreichbar.
32 % Das Promptzeichen der Kommandozeile wird hier durch > angedeutet.
33 % 1>, 2>, 3>, ... sind die Promptzeichen der Erlang-Shell.
34 %
35 % > werl
36 %
37 % 1> 3+4.                        % Kleiner Test der Erlang-Shell
38 % 7
39 % 2> c(hallo01).                 % Die Datei hallo01.erl compilieren
40 % {ok,hallo01}
41 % 3> hallo01:sagHallo().         % sagHallo mit 0 Param aufrufen
42 % Hallo Welt!
43 % ok
44 %                                % sagHallo mit 1 Param aufrufen
45 % 4> hallo01:sagHallo(richard).  % Achtung: richard, nicht Richard!
46 % Hallo richard! Wie geht's?
47 % ok
48 % 5> hallo01:sagHallo("Richard"). % Hier geht auch Richard
49 % Hallo Richard! Wie geht's?
50 % 6> q().                        % Erlang-Shell beenden (quit)
51 % ---------------------------------------------------------------------
```

1. Ein *Kommentar* beginnt mit `%` und endet mit der betreffenden Zeile

2. Funktionen mit unerschieldich vielen Parametern dürfen *gleiche Namen* haben (wie z.B. `sagHallo/0` und `sagHallo/1`).

3. Die Funktion `io:format` hat (entfernt) Ähnlichkeit mit `printf` in C und Java, aber mit ~ statt %. Als Parameter erwartet sie einen *Formatstring* (muss) und eine *Liste* beliebiger Terme (kann).

2 Zeilen einlesen von der Standardeingabe

```
 1 % Datei hallo02.erl    % Erlang-Quelldatei
 2 -module(hallo02).      % Der Modulname muss zum Dateinamen passen
 3 -export([sagHallo/0]). % Gezielter Export bestimmter Funktionen
 4 % ---------------------------------------------------------------------
 5 sagHallo() ->
 6    Nam1 = io:get_line("Wie heissen Sie? "),
 7    Nam2 = string:strip(Nam1,right,$\n),        % Alle \n am Ende entfernen
 8    io:format("Hallo ~s! Wie geht's?~n", [Nam2]).
 9 % ---------------------------------------------------------------------
10 % Ein Dialog mit diesem Programm, nachdem man es compiliert und
11 % eine Erlang-Shell gestartet hat. Erlaeuterungen dazu:
12 %
13 % 37>      ist der Prompt der Erlang-Shell
14 % Richard  ist eine Eingabe des Benutzers,
15 % ok       ist der Wert des Ausdrucks io:format("Hallo...2]) und somit
16 %          das Ergebnis der Funktion sagHallo):
17 %
18 % 37> hallo02:sagHallo().
19 % Wie heissen Sie? Richard
20 % Hallo Richard! Wie geht's?
21 % ok
22 % ---------------------------------------------------------------------
```

1. Zeichenketten, die mit einem *Großbuchstaben* beginnen, werden als *Namen* von (unveränderbaren) Variablen interpretiert, z.B. `Nam1`, `Nam2`, `ALFRED`, `A_123_xyz` etc.
Zeichenketten, die mit einem *Kleinbuchstaben* beginnen, werden als *Atome* interpretiert, z.B. `right`, `ok`, `die_Funktion_ist_jetzt_fertig`, `true`, `false` etc.
Zeichenketten, die in *doppelte Anführungszeichen* eingeschlossen sind, werden als *Strings* interpretiert, z.B. `"Wie heissen Sie?"`, `"Hallo ~s!"` etc.

2. Der Befehl in Zeile 7 (`Nam1 = io:get_line(...),`) gibt den angegebenen Prompt-String zur Standardausgabe aus, liest dann eine Zeile von der Standardeingabe ein und erzeugt eine Variable `Nam1` mit dieser Zeile als Wert. Dieser Wert endet mit einem Zeilenwechsel `\n`.

3. Um den Zeilenwechsel zu entfernen müssen wir eine weitere Variable (`Nam2`) vereinbaren, da Variablen grundsätzlich unveränderbar sind (man darf ihnen nur einmal einen Wert zuordnen).

4. Der Umwandlungsbefehl `~s` im Formatstring des `io:format`-Befehls in Zeile 9 ist besonders gut geeignet, String-Werte auszugeben.

5. Das Ergebnis einer Funktion ist der Wert des letzten Ausdrucks, der ausgeführt wird. Z.B. ist das Ergebnis der Funktion `sagHallo` der Wert des Ausdrucks in Zeile 9:
`io:format("Hallo ~s! Wie geht's?~n", [Nam2])`
Normalerweise ist das der Wert `ok`.

6. Wenn man in einer Erlang-Shell eine Funktion aufruft (wie z.B. in Zeile 19), wird das Ergebnis dieser Funktion ausgegeben (siehe das `ok` in Zeile 22).

3 Terme einlesen und das case-Konstrukt

```
 1 % Datei case01.erl     % Erlang-Quelldatei
 2 -module(case01).       % Der Modulname muss zum Dateinamen passen
 3 -export([liesTerm/0]). % Exportiert nur eine Funktion
 4 % ---------------------------------------------------------------------
 5 % Ein case-Befehl unterscheidet Faelle anhand von Musterabgleichen
 6 % ---------------------------------------------------------------------
 7 liesTerm() ->
 8    Eingabe = io:read("Ein Erlang-Term: "),
 9    case Eingabe of
10       {ok, T} ->   % Entspricht der Wert von Eingabe dem Muster {ok, T}?
11          io:format("~p ist ein guter Term!~n",  [T]);
12       {error,{Znr,Von,Meld}}   ->
13          io:format("Die Funktion ~p ",          [Von] ),
14          io:format("meldet fuer Zeile ~p :~n",  [Znr] ),
15          io:format("~p~n",                      [Meld]);
16       Sonst ->     % Ein Musterabgleich mit einer Variablen gelingt immer!
17          io:format("Unerwartetes Ergebnis von io:read:~n~p~n",[Sonst])
18    end.
19 % ---------------------------------------------------------------------
20 % Beispiele fuer korrekte Eingaben (nach dem Prompt "Ein Erlang-Tern:"):
21 %
22 % hallo.
23 % "Hallo".
24 % {a,2,c}.
25 % [1,b,3].
26 % {[1,2,3],{a,b,c}}.
27 % [a,{2,b},[c,3],d].
28 %
29 % Man kann die Eingabe ueber mehrere Zeilen verteilen (indem man
30 % zwischendurch  auf Return drueckt). Erst wenn man einen Punkt .
31 % und Return eingibt wird das Einlesen beendet.
32 % ---------------------------------------------------------------------
33 % Ein Dialog mit diesem Programm (nachdem man es compiliert und
34 % eine Erlang-Shell gestartet hat ( 21> ist der Prompt der Erlang-Shell.
35 % Eingaben des Benutzers sind fett hervorgehoben:
36 %
37 % 21> case01:liesTerm().
38 % Ein Erlang-Term: {10,20,
39 % Ein Erlang-Term: 30,
40 % Ein Erlang-Term: 40}.
41 % {10,20,30,40} ist ein guter Term!
42 % ok
43 % ---------------------------------------------------------------------
44 % Ein Dialog mit Eingabefehler ( 22> ist der Prompt der Erlang-Shell):
45 %
46 % 22> case01:liesTerm().
47 % Ein Erlang-Term: {10,20,
48 % Ein Erlang-Term: 30;              % Falsch: Semikolon statt Komma
49 % Ein Erlang-Term: 40}.
50 % Die Funktion erl_parse meldet fuer Zeile 2 :
51 % ["syntax error before: ",["';'"]]
52 % ok
53 % ---------------------------------------------------------------------
```

1. Der Befehl in Zeile 8 gibt den angegebenen Prompt-String aus, versucht dann, einen *Erlang-Term* einzulesen und erzeugt eine Variable `Eingabe` mit diesem Term als Wert. Wenn die Eingabe keinen Term enthält, wird eine Ausnahme geworfen.

2. Das `case`-Konstrukt von Zeile 9 bis 18 untersucht den Term `Eingabe` mit Hilfe von 3 *Musterabgleichen* (in Zeile 10, 12 und 16). Die Ausdrücke hinter dem ersten gelingenden Musterabgleich werden ausgeführt. Der Musterabgleich mit der Variablen `Sonst` (Zeile 16) gelingt immer. Der Musteraabgleich mit dem 2-Tupel `{ok, T}` (Zeile 10) gelingt, wenn die Variable `Eingabe` ebenfalls ein 2-Tupel `{X,Y}` enthält, bei dem `X` gleich dem Atom `ok` und `Y` ein beliebiger Term ist.

4 Ein case-Konstrukt mit Musterabgleichen und Wächtern (guards)

```
 1 anaTerm() ->
 2    Eingabe = io:read("Ein Erlang-Term (q. zum Anhalten): "),
 3    case Eingabe of
 4       {ok,q} ->
 5          io:format("anaTerm() beendet sich!~n");
 6       {ok,N} when is_number(N) andalso N > 0 ->
 7          io:format("Einzelne positive Zahl!~n"),
 8          anaTerm();
 9       {ok,N} when is_number(N), N < 0 ->
10          io:format("Einzelne negative Zahl!~n"),
11          anaTerm();
12       {ok,{N1,N2}} when
13          is_number(N1), (N1==2 orelse N1==3 orelse N1==5) ,
14          is_number(N2), (N2==2 orelse N2==3 orelse N2==5) ->
15          io:format("2-Tupel kleiner Primzahlen!~n"),
16          anaTerm();
17       {ok,Sonst} ->
18          io:format("Sonstiger Term: ~p~n",[Sonst]),
19          anaTerm();
20       {error,Sonst} ->
21          io:format("Das war kein Term: ~p~n",[Sonst]),
22          anaTerm()
23    end.
```

1. Der `io:read`-Befehl in Zeile 1 liefert als Ergebnis ein 2-Tupel, entweder eines der Form `{ok,T}` wobei `T` ein Term ist, oder eines der Form `{error,M}` wobei `M` eine Fehlermeldung ist.

2. Der in Zeile 6 beginnende Fall beginnt mit einem Musterabgleich (zwischen dem Wert der `Eingabe` und dem Muster `{ok,N}`) und einem Wächter `when is_number(N) andalso N > 0`. Dieser Fall wird nur gewählt, wenn der Musterabgleich gelingt und der Wächter den Wert `true` ergibt. Der Operator `andalso` wertet seinen rechten Operanden *nicht* aus, wenn der linke den Wert `false` hat.

3. Der in Zeile 12 beginnende Fall hat einen komplexen Wächter. Darin sind die 3 Kommas alternative Notationen für den Operator `andalso`. Der Operator `orelse` wertet seinen rechten Operanden *nicht* aus, wenn der linke den Wert `true` hat.

Es folgt hier ein Dialog mit der Funktion `anaTerm`. 12> ist der Prompt der Erlang-Shell, die Eingaben des Benutzers sind fett hervorgehoben:

```
12> case02:anaTerm().
Ein Erlang-Term (q. zum Anhalten): 17.
Einzelne positive Zahl!
Ein Erlang-Term (q. zum Anhalten): -5.
Einzelne negative Zahl!
Ein Erlang-Term (q. zum Anhalten): {3,4}.
Sonstiger Term: {3,4}
Ein Erlang-Term (q. zum Anhalten): {3,5}.
2-Tupel kleiner Primzahlen!
Ein Erlang-Term (q. zum Anhalten): }}.
Das war kein Term: {1,erl_parse,["syntax error before: ",["'}'"]]}
Ein Erlang-Term (q. zum Anhalten): q.
anaTerm() beendet sich!
ok
```

5 Zwei Prozesse spielen Ping-Pong

```
 1 % Datei pingpong.erl       % Erlang-Quelldatei
 2 -module(pingpong).         % Der Modulname muss zum Dateinamen passen
 3 -export([los/1, pingPong/2]).
 4 % ---------------------------------------------------------------------
 5 % Die Funktion los(Anz) erzeugt zwei Prozesse, die beide die Funktion
 6 % pingPong ausfuehren und Anz viele Nachrichten austauschen.
 7 % ---------------------------------------------------------------------
 8 los(Anz) ->
 9    % Diese Funktion startet zwei Prozesse, die beide die Funktion pingPong
10    % ausfuehren und "wirft einen Ball ins Spiel", indem sie dem einen
11    % Prozess eine ping-Nachricht mit der Pid des anderen als Absender
12    % schickt.
13
14    % Falls Anz keine Ganzzahl oder nicht groesser als 0 ist,
15    % wird 3 als Ersatz genommen.
16    if
17       is_integer(Anz) andalso Anz > 0 ->
18          AnzP = Anz;
19       true ->
20          AnzP = 3
21    end,
22
23    % ?MODULE bezeichnet den aktuellen Modul (d.h. pingpong)
24   Pid1 = spawn(?MODULE, pingPong, [AnzP, spielerA]),
25   Pid2 = spawn(?MODULE, pingPong, [AnzP, spielerB]),
26
27   Pid1 ! {ping, Pid2},
28   los_hat_2_pingPong_Prozesse_gestartet.        % Ein Atom als Ergebnis
29 % ---------------------------------------------------------------------
30 pingPong(Anz, Spieler) ->
31    % Wartet Anz Mal auf eine   ping- oder eine  pong-Nachricht und
32    % beantwortet sie mit einer pong- bzw. einer ping-Nachricht.
33    % Spieler dient zum Unterscheiden mehrerer pingPong-Prozesse.
34
35    if
36       Anz > 0 ->
37          receive
38             {ping, Pid} ->
39                io:format("~p: ping-~w erhalten!~n", [Spieler,Anz]),
40                Pid ! {pong, self()};
41             {pong, Pid} ->
42                io:format("~p: pong-~w erhalten!~n", [Spieler,Anz]),
43                Pid ! {ping, self()};
44             Sonst ->
45                io:format("~p: Unerwartete Meldung: ~p~n",
46                   [Spieler,Sonst])
47          end,
48          pingPong(Anz-1, Spieler);                        % Endrekursion
49       true ->
50          io:format("~p  beendet sich***!~n", [Spieler])
51    end.
52 % ---------------------------------------------------------------------
53 % Ein Dialog mit diesem Programm ( 17> ist der Prompt der Erlang-Shell):
54 %
55 % 17> pingpong:los(3).
56 % spielerA: ping-3 erhalten!
57 % los_hat_2_pingPong_Prozesse_gestartet
58 % spielerB: pong-3 erhalten!
59 % spielerA: ping-2 erhalten!
60 % spielerB: pong-2 erhalten!
61 % spielerA: ping-1 erhalten!
62 % spielerA  beendet sich***!
63 % spielerB: pong-1 erhalten!
64 % spielerB  beendet sich***!
65 % ---------------------------------------------------------------------
```

1. In Zeile 24 wird ein Prozess gestartet, der die Funktion `pingPong` im aktuellen Modul `?Module` ausführt. Der Funktion werden die Parameter `AnzP` (eine Variable) und `spielerA` (ein Atom) übergeben. In Zeile 25 passiert etwas ganz ähnliches, nur der zweite Parameter ist `spielerB`. Die beiden Prozesse werden hier als Spieler-Prozesse bezeichnet.

2. Wenn `AnzP` (die Anzahl der Schläge, die die beiden Spieler ausführen sollen) größer `0` ist, warten die beiden Spieler-Prozesse erstmal in Zeile 37 auf einen Ball (d.h. auf eine Nachricht). Erst wenn die `los`-Funktion in Zeile 27 dem einen Spieler-Prozess eine Nachricht schickt, geht das Spiel richtig los.

Die `los`-Funktion hat 4 Möglichkeiten, das Spiel in Gang zu bringen:

Dem Spieler-Prozess `Pid1` eine ping-Nachricht schicken
Dem Spieler-Prozess `Pid1` eine pong-Nachricht schicken
Dem Spieler-Prozess `Pid2` eine ping-Nachricht schicken
Dem Spieler-Prozess `Pid2` eine pong-Nachricht schicken

Hier realisiert die `los`-Funktion (willkürlich) die erste dieser Möglichkeiten. Als Absender der ping-Nachricht gib sie aber nicht ihre eigene Pid an, sondern `Pid2` (damit die weiteren Nachrichten zwischen den Spieler-Prozessen `Pid1` und `Pid2` ausgetauscht werden, und nicht zwischen `Pid1` und dem Prozess der `los`-Funktion).

6 Ein Serverprozess versorgt 2 Klienten mit Zufallszahlen

```
 1 % Datei proz01.erl       % Erlang-Quelldatei
 2 -module(proz01).         % Der Modulname muss zum Dateinamen passen
 3 -export([los/1, intServer/1, intKlient/2]).
 4 % ---------------------------------------------------------------------
 5 % Die Funktion los(Anz) startet 3 Prozesse: Einen Server und zwei Klienten.
 6 % Der Server beantwortet Meldungen der Klienten mit Zufallszahlen. Die
 7 % Klienten geben diese Zufallszahlen zum Bildschirm aus.
 8 % ---------------------------------------------------------------------
 9 los(Anz) ->
10    % Startet einen Sever-Prozess und registriert ihn als int_server.
11    % Startet dann zwei Klienten-Prozesse, die vom int_server Zufalls-
12    % zahlen anfordern und ausgeben.
13
14    % Falls Anz keine Ganzzahl und groesser als 0 ist,
15    % wird 3 als Ersatz genommen.
16    if
17       is_integer(Anz) andalso Anz > 0 ->
18          AnzK = Anz;
19       true ->
20          AnzK = 3
21    end,
22
23    % Der intServer muss doppelt so viele Ganzzahlen erzeugen wie
24    % jeder der beiden intKlient-Prozesse verbraucht:
25    AnzS = 2 * AnzK,
26
27    ServerPid = spawn(?MODULE, intServer, [AnzS]),
28    register(int_server, ServerPid),
29
30    spawn(?MODULE, intKlient, [AnzK, a]),
31    spawn(?MODULE, intKlient, [AnzK, b]),
32
33    timer:sleep(1000), % Zur Verschoenerung des Druckbildes
34    los_ist_fertig.
35 % ---------------------------------------------------------------------
```

```
36 intServer(AnzS) ->
37    % Erwartet AnzS viele Nachrichten und beantwortet jede mit einer
38    % zufaelligen Ganzzahl aus dem Bereich 1 bis 1000 (einschliesslich)
39    if
40       AnzS > 0 ->
41          receive
42             {int, Pid} ->
43                Pid ! random:uniform(1000),
44                intServer(AnzS-1)                          % Endrekursion
45          end;
46       true ->
47          timer:sleep(1000), % Zur Verschoenerung des Druckbildes
48          io:format("Server haelt an!~n")
49    end.
50 % ---------------------------------------------------------------------
51 intKlient(AnzK, Kennung) ->
52    % Schickt dem int_server AnzK viele Nachrichten und gibt die
53    % Zufallszahlen, die er als Anwort erhaelt, zum Bildschirm aus.
54    % Die Kennung dient dazu, mehrere Klienten zu unterscheiden
55    % (besonders empfehlenswert sind kurze Kennungen wie a, b, c etc.)
56    if
57       AnzK > 0 ->
58          int_server ! {int, self()},
59          receive
60             R -> ok
61          end,
62          io:format("~w~-4w", [Kennung, R]),
63 %        timer:sleep(R),
64          intKlient(AnzK-1, Kennung);                       % Endrekursion
65       true ->
66          timer:sleep(1000), % Zur Verschoenerung des Druckbildes
67          io:format("Klient ~w haelt an!~n", [Kennung])
68    end.
69 % ---------------------------------------------------------------------
70 % Dialog mit diesem Programm ( 12> ist der Prompt der Erlang-Shell):
71 %
72 % 12> proz01:los(30).
73 % a93  b444 a724 b946 a502 b312 a598 b916 a667 b478 a597 b143 a210 b698 a160
74 % b559 a215 b458 a422 b6   a563 b476 a401 b310 a59  b579 a990 b331 a184 b203
75 % a34  b890 a837 b829 a329 b254 a810 b19  a26  b49  a55  b988 a562 b610 a389
76 % b667 a877 b893 a795 b583 a295 b720 a401 b794 a75  b394 a598 b677 a645 b75
77 % los_ist_fertig
78 % Klient b haelt an!
79 % Klient a haelt an!
80 % Server haelt an!
81 % ---------------------------------------------------------------------
```

1. In diesem Beispiel startet die `los`-Funktion 3 Prozesse: Einen, der die Funktion `intServer` ausführt (Zeile 27) und zwei, die beide die Funktion `intKlient` ausführen (Zeilen 30 und 31).

2. In Zeile 28 wird der Server-Prozess unter dem Namen `int_server` registriert. Danach kann man beim Senden einer Nachricht an diesen Prozess seine Pid angeben (falls man sie zur Hand hat) oder diesen Namen. Die Klienten-Prozesse benutzen den Namen, weil sie die Pid nicht kennen (siehe Zeile 58).

3. Beim Starten des Servers wird ihm mitgeteilt, dass er `AnzS` viele Nachrichten empfangen und beantworten soll, bevor er sich beendet. Jedem der Klienten wird mitgeteilt, dass er `AnzK` viele Nachrichten an den Server schicken (und die Antwort ausgeben) soll. In Zeile 25 sieht man, wie `AnzS` aus `AnzK` berechnet wird. Es gibt flexiblere Muster, nach denen man Prozesse im richtigen Moment anhalten kann, aber die sind etwas aufwendiger zu programmieren.

7 Verschiedene Arten von Seiteneffekten in Erlang-Programmen

Wie alle guten funktionalen Sprachen umfasst auch Erlang eine Reihe von prozeduralen Konstrukten. Die folgende Liste ist keineswegs vollständig.

7.1 Ausgabebefehle

Jeder Ausgabebefehl hat einen Seiteneffekt auf den Bildschirm (oder eine Datei etc.), z.B.

```
 1 io:format("Hallo"),
```

7.2 Seiteneffekt auf den Stapel eines Prozesses

Jeder Prozess, der die folgende Funktion ausführt, ist eine Art globale (Ganzzahl-) Variable N, deren Wert von anderen Prozessen *verändert* und *gelesen* werden kann, indem sie dem N-Prozess bestimmte Meldungen schicken:

```
 2 next(N) ->
 3    receive
 4       {'N?', Absender} ->
 5          Absender ! N,
 6          next(N);             % Endrekursion
 7       'incN!' ->
 8          next(N+1);           % Endrekursion
 9       halt ->
10          ok
11    end.
```

Hier wird vorausgesetzt, dass die Funktion `next` (aus dem Modul `effekte01`) beim Start des Prozesses mit einer *Ganzzahl* als Parameter aufgerufen wird, z.B. so:

```
12 Pid = spawn(effekte01, next, [0])
```

Ein anderer Prozess kann den momentanen Wert des N-Prozesses mit folgenden Befehlen lesen:

```
13    Pid ! {'N?' ! self()},
14    receive
15       Wert -> % mach irgendetwas mit dem Wert
16    end,
```

und mit dem folgenden Befehl kann er den Wert der Variablen um 1 erhoehen:

```
17    Pid ! 'incN!'
```

Anstelle der Atome `'N?'` (zum Lesen) und `'incN!'` (zum Erhöhen) hätte man auch andere Atome verwenden können, z.B. `lies` und `erhoehe` oder `karlHeinz` und `annaMaria`.

7.3 Seiteneffekte auf die Prozess-Abbildung (engl. process dictionary)

Jeder Erlang-Prozess besitzt (automatisch) eine *Abbildung* (engl. map oder dictionary). Dabei ist mit *Abbildung* eine Sammlung von Einträgen gemeint, von denen jeder aus einem *Schlüssel* und einem *Wert* besteht (engl. key and value). Als Schlüssel und Werte sind beliebige Erlang-Terme erlaubt. In einer Abbildung kann jeder Schlüssel nur mit *einem* Wert verbunden sein, aber ein Wert kann mit *verschiedenen* Schlüsseln verbunden sein. Diese Abbildung kann man auch als eine Sammlung von veränderbaren Variablen betrachten.

Konzeptuell ist in einer leeren Abbildung *jeder mögliche Schlüssel* mit dem Wert `undefined` verbunden. Die Abbildung eines Prozesses kann man mit folgenden Befehlen bearbeiten:

Befehl	Wirkung
`put(Schluessel,Wert)`	Fügt einen neuen Eintrag `{Schluessel,Wert}` in die Abbildung ein. Liefert den Wert, mit dem der Schluessel vorher verbunden war (z.B. `undefined`)
`get(Schluessel)` `get()`	Liefert den aktuellen Wert zum `Schluessel` (z.B. `undefined`) Liefert eine Liste aller Einträge, deren Wert ungleich `undefined` ist.

`get_keys(Wert)`
 	
`get_keys(undefined)`
	Liefert eine Liste aller Schlüssel, die zur Zeit mit dem Wert `Wert` verbunden sind.
	Liefert (ein bisschen inkonsequent :-) nicht eine Liste mit fast allen möglichen Schlüsseln darin, sondern eine *leere Liste* `[]`.

`erase(Schluessel)`
 	
`erase()`
	Liefert wie `get(Schluessel)` und verbindet `Schluessel` mit `undefined`.
	Liefert wie `get()` und verbindet alle Schluessel mit `undefined`.

Die folgende Funktion bearbeitet die *Prozess-Abbildung* (engl. the process dictionary):

```
 1 globVar() ->
 2    % Ein paar Eintraege in die Prozess-Abbildung einfuegen:
 3    io:format("-----------------------------------------------~n"),
 4    io:format("put(z1, 17)        : ~w~n", [put(z1, 17)]),
 5    io:format("put(z2, 21)        : ~w~n", [put(z2, 21)]),
 6    io:format("put(z3, 42)        : ~w~n", [put(z3, 42)]),
 7
 8    % Pruefen, ob die Eintraege in der Abbildung drin sind:
 9    io:format("-----------------------------------------------~n"),
10    io:format("get(z1)            : ~w~n", [get(z1)]),
11    io:format("get(z2)            : ~w~n", [get(z2)]),
12    io:format("get(z3)            : ~w~n", [get(z3)]),
13    io:format("get(z4)            : ~w~n", [get(z4)]),
14
15    % Die mit den Namen z1 und z2 assoziierten Werte veraendern:
16    io:format("-----------------------------------------------~n"),
17    io:format("put(z1, 42)        : ~w~n", [put(z1, 42)]),
18    io:format("put(z2, 22)        : ~w~n", [put(z2, 22)]),
19
20    % Pruefen, welche Schluessel-Wert-Paare jetzt in der Abbildung stehen:
21    io:format("-----------------------------------------------~n"),
22    io:format("get(z1)            : ~w~n", [get(z1)]),
23    io:format("get(z2)            : ~w~n", [get(z2)]),
24    io:format("get(z3)            : ~w~n", [get(z3)]),
25    io:format("get(z4)            : ~w~n", [get(z4)]),
26    io:format("-----------------------------------------------~n"),
27    io:format("get()              : ~p~n", [get()]),
28    io:format("get_keys(42)       : ~p~n", [get_keys(42)]),
29    io:format("get_keys(undefined): ~p~n", [get_keys(undefined)]),
30    io:format("-----------------------------------------------~n"),
31    io:format("erase(z3)          : ~p~n", [erase(z3)]),
32    io:format("erase()            : ~p~n", [erase()  ]),
33    io:format("erase()            : ~p~n", [erase()  ]),
34    io:format("-----------------------------------------------~n").
```

Diese Funktion gibt folgende Zeilen zum Bildschirm aus:

```
-----------------------------------------------
put(z1, 17)        : undefined
put(z2, 21)        : undefined
put(z3, 42)        : undefined
-----------------------------------------------
get(z1)            : 17
get(z2)            : 21
get(z3)            : 42
get(z4)            : undefined
-----------------------------------------------
put(z1, 42)        : 17
put(z2, 22)        : 21
-----------------------------------------------
get(z1)            : 42
get(z2)            : 22
get(z3)            : 42
get(z4)            : undefined
-----------------------------------------------
```

```
get()              : [{z3,42},{z2,22},{z1,42}]
get_keys(42)       : [z3,z1]
get_keys(undefined): []
-----------------------------------------
erase(z3)          : 42
erase()            : [{z2,22},{z1,42}]
erase()            : []
-----------------------------------------
ok
```

8 Erlang-Server und -Klienten kann man auch in Java programmieren

Ein Erlang-Server-Prozess, in Java programmiert

```
 1 // Datei Plus3Server.java
 2 /* ---------------------------------------------------------------------------
 3 Dieses Java Programm Plus3Server kann mit Erlang-Prozessen interagieren.
 4 Diesem Server kann man Meldungen der Form {ABSENDER, ZAHL} schicken.
 5 Dabei sollte ABSENDER die Erlang-Pid des Absenders und ZAHL eine long-Zahl
 6 sein (z.B. 123 oder 7654321). Der Server antwortet dann mit einer Meldung
 7 der Form {ok, ERG}, wobei ERG gleich ZAHL plus 3 ist.
 8
 9 Fuer diesen Server gibt es zwei Klienten-Programme:
10 plus3Clinet.erl   // In Erlang geschrieben
11 Plus3Client.java  // In Java   geschrieben
12 ---------------------------------------------------------------------------
13 Dies Java-Programm erzeugt einen Erlang-Knoten namens k01@HOSTNAME, startet
14 darin einen Prozess und registriert ihn unter dem Namen "plus3Server"
15 (diesen Namen benoetigt man, um dem Prozess eine Meldung zu schicken).
16 Dabei steht HOSTNAME fuer den Inhalt der Umgebungsvariablen HOSTNAME (der
17 z.B. dann angezeigt wird, wenn man den folgenden Erlang-Befehl in einem
18 Erlang-Fenster eingibt:
19 17> net_adm:localhost().
20
21 Bedienungsanleitung fuer dieses Serverprogramm:
22 S1: Diese Java-Datei compilieren. Dabei muss die Datei Erlang.jar im
23     CLASSPATH eingetragen sein.
24 S2: Eine Kommandozeile starten und das Verzeichnis, in dem die
25     Erlang-Quelle plus3Client.erl liegt, zum aktuellen Arbeitsverzeichnis
26     machen. Dann einen verteilten Erlang-Knoten mit beliebigem Namen
27     (z.B. xxx) und dem Cookie abc (muss abc sein!) starten, etwa so:
28     > werl -name xxx -setcookie abc
29     Hinweis: Durch diesen Schritt wird unter anderem das Programm epmd.exe
30     (der Port Mapper Daemon, PMD) gestartet, der unbedingt erforderlich ist!
31 S3: Ob der PMD laeuft, kann man mit folgendem Kommando pruefen:
32     > ERLANG_INSTALL/erts-5.7.1/bin/epmd -names
33     Zwei Zeilen wie die folgenden sollten erschreinen:
34     epmd: up and running on port 4369 with data:
35     name xxx at port 1201
36 S4: Im Erlang-Fenster (das durch S2 geöffnet wurde) den Erlang-Klienten
37     compilieren, etwa so:
38     (xxx@PCGrude)1> c(plus3Client).
39     Der Erlang-Compiler sollte mit  {ok,plus3Client}  antworten
40 S5: Das Java-Programm Plus3Server starten, etwa mit folgendem Kommando:
41     > java Plus3Server
42     Es sollte folgende Zeilen zu seiner Standardausgabe ausgeben:
43     Plus3Server: Jetzt geht es los!
44     Plus3Server: Warte auf Nachricht!
45 S6: Wiederholt man jetzt S3 , sollten etwa folgende 3 Zeilen erscheinen:
46     epmd: up and running on port 4369 with data:
47     name k01 at port 1226
48     name xxx at port 1201
49 S7: Im Erlang-Fenster (das durch S2 geoeffent wurde) die Funktion p3
50     im Modul plus3Client aufrufen, etwa so:
51     (xxx@PCGrude)2> plus3Client:p3(17).
52     Der Klient sollte die 17 zum Server schicken und als Antwort 20
53     erhalten. Im Erlang-Fenster sollten folgende Zeilen erscheinen:
```

```
 54     plus3client:p3(17) wurde aufgerufen!
 55     p3(17) ist gleich 20
 56     ok
 57     Falls die Funktion p3 keine Verbindung zum Server herstellen kann,
 58     beendet sie sich nach 2 Sekunden mit der Medlung:
 59     plus3client:p3(17) wurde aufgerufen!
 60     Timeout in p3(17)
 61     ok
 62 S8: Wenn der Klient dem Server die Zahl 999 schickt, etwa so:
 63     (xxx@PCGrude)3> plus3Client:p3(999).
 64     beendet sich der Server (nachdem er als Antwort 1002 geschickt hat).
 65 ---------------------------------------------------------------------- */
 66 import com.ericsson.otp.erlang.OtpNode;
 67 import com.ericsson.otp.erlang.OtpMbox;
 68 import com.ericsson.otp.erlang.OtpErlangObject;
 69 import com.ericsson.otp.erlang.OtpErlangTuple;
 70 import com.ericsson.otp.erlang.OtpErlangPid;
 71 import com.ericsson.otp.erlang.OtpErlangAtom;
 72 import com.ericsson.otp.erlang.OtpErlangLong;
 73 import com.ericsson.otp.erlang.OtpErlangExit;
 74
 75 class Plus3Server {
 76    // ----------------------------------------------------------------------
 77    // Die Funktion, die dieser Server fuer seine Klinenten ausfuehrt:
 78    static Long plus3(Long n) {
 79       return n + 3;
 80    }
 81    // ----------------------------------------------------------------------
 82    static public void main(String[] _) throws Exception {
 83       printf("Plus3Server: Jetzt geht es los!%n");
 84
 85       // Einen verteilten Erlang-Knoten mit Cookie abc erstellen und
 86       // darauf einen Prozess starten:
 87       String  serverKnotenName  = "k01"; // Hier ohne @HOST-Teil
 88       String  serverProzessName = "plus3Server";
 89       String  cookie            = "abc";
 90
 91       OtpNode serverKnoten      = new OtpNode(serverKnotenName, cookie);
 92       OtpMbox serverProzess     = serverKnoten.createMbox(serverProzessName);
 93
 94       // Im Englischen werden Erlang-Prozesse in einem Java-Programm auch
 95       // als mail boxes bezeichnet (weil jeder Prozess eine mail box hat).
 96
 97       // Fragen empfangen und beantworten, bis die Zahl 999 als Frage kommt:
 98       while (true) try {
 99          // Eine Frage empfangen:
100          printf("Plus3Server: Warte auf Nachricht!%n");
101          OtpErlangObject frageObj = serverProzess.receive();
102          printf("Plus3Server: Nachricht erhalten: %s%n", frageObj);
103          OtpErlangTuple  frageOtp = (OtpErlangTuple) frageObj;
104          OtpErlangPid    absender = (OtpErlangPid) (frageOtp.elementAt(0));
105          OtpErlangLong   zahlOtp  = (OtpErlangLong)(frageOtp.elementAt(1));
106          Long            zahl     = zahlOtp.longValue();
107
108          // Eine Antwort erstellen und zurueckschicken:
109          OtpErlangObject[] antwortR = new OtpErlangObject[2];
110          antwortR[0]               = new OtpErlangAtom("ok");
111          antwortR[1]               = new OtpErlangLong(plus3(zahl));
112          OtpErlangTuple    antwortT = new OtpErlangTuple(antwortR);
113          serverProzess.send(absender, antwortT);
114
115          // Diesen Server evtl. beenden:
116          if (zahl == 999) break; // Damit der Client den Server beenden kann
117
118       } catch(OtpErlangExit e) {
119          break;
120       }
```

```
121
122       printf("Plus3Server: Das war's erstmal!%n");
123    } // main
124    // ---------------------------------------------------------------------
125    // Eine Methode mit einem kurzen Namen:
126    static void printf(String f, Object... v) {System.out.printf(f, v);}
127    // ---------------------------------------------------------------------
128 } // class Plus3Server
```

Ein Erlang-Klient für obigen Server, in Java programmiert

```
  1 // Datei Plus3Client.java
  2 /* ---------------------------------------------------------------------------
  3 Dieses Programm ist ein Java-Klient fuer den Java-Plus3Server.
  4 Dieser Klient kommuniziert mit dem Server ueber Erlang-Knoten und
  5 Erlang-Prozesse.
  6
  7 Wenn man dieses Klient-Programm startet, sollte man auf der Kommandozeile
  8 eine Ganzzahl als Parameter angeben, etwa so:
  9 > java Plus3Client 17
 10
 11 Der Parameter (im Beispiel: 17) wird zum Server geschickt. Der sollte
 12 eine um 3 groessere Zahl zurueckschicken. Der Klient gibt die Antwort
 13 des Servers zur Standardausgabe aus.
 14
 15 Wenn der Klient dem Server die Zahl 999 schickt dann beendet der sich
 16 (nachdem er die Antwort 1002 zurueckgeschickt hat).
 17 ---------------------------------------------------------------------------
 18 Bevor man dieses Klient-Programm startet, sollte man den Server starten
 19 (siehe dazu die Bedienungsanleitung in der Datei Plus3Server.java).
 20
 21 Ausser diesem Java-Klienten gibt es auch einen Erlang-Klienten (siehe
 22 die Datei plus3Client.erl), der sich ganz aehnlich verhaelt.
 23 --------------------------------------------------------------------------- */
 24 import com.ericsson.otp.erlang.OtpNode;
 25 import com.ericsson.otp.erlang.OtpMbox;
 26 import com.ericsson.otp.erlang.OtpErlangObject;
 27 import com.ericsson.otp.erlang.OtpErlangTuple;
 28 import com.ericsson.otp.erlang.OtpErlangLong;
 29
 30 class Plus3Client {
 31    // ---------------------------------------------------------------------
 32    static public void main(String[] sonja) throws Exception {
 33       printf("Plus3Client: Jetzt geht es los!%n");
 34       printf("-----------------------------%n");
 35
 36       // Wurde genau 1 Kommandozeilen-Parameter angegeben?
 37       if (sonja.length != 1) {
 38          printf("Beispiel fuer eine Aufruf dieses Programms:%n");
 39          printf("> Plus3Clinet 17%n%n");
 40          return;
 41       }
 42
 43       // Wenn der Kommandozeilen-Parameter sich nicht in einen long-Wert
 44       // umwandlen laesst, wird die Zahl 17 als Ersatz genommen:
 45       long fZahl = 17; // Die Frage-Zahl
 46       try {
 47          fZahl = Long.decode(sonja[0]);
 48          printf("%d ist eine sehr gute Eingabe!%n", fZahl);
 49       } catch (NumberFormatException ex) {
 50          printf("Falsche Eingabe: %s%n", ex.getMessage());
 51          printf("Als Ersatz wird die Zahl %d genommen!%n", fZahl);
 52       }
 53
 54       // Die Namen von zwei Erlang-Knoten und zwei Erlang-Prozessen:
 55       String localHost         = System.getenv("HOSTNAME");
 56       String serverKnotenName  = "k01@" +  localHost; // Mit @HOST-Teil
```

```
57       String serverProzessName = "plus3Server";
58       String clientKnotenName  = "k02";         // Hier ohne @HOST-Teil
59       String clientProzessName = "plus3Clinet";
60       String cookie            = "abc";
61
62       // Einen verteilten Erlang-Knoten mit Cookie abc erstellen und
63       // darauf einen Prozess starten:
64       OtpNode clientKnoten     = new OtpNode(clientKnotenName, cookie);
65       OtpMbox clientProzess    = clientKnoten.createMbox(clientProzessName);
66
67       // Pruefen (mit ping), ob der Server erreichbar ist:
68       if (clientKnoten.ping(serverKnotenName, 2000)) {
69          printf("%s hat geantwortet!%n", serverKnotenName);
70       } else {
71          printf("%s antwortet nicht!%n", serverKnotenName);
72          return;                 // Diesen Klienten beenden
73       }
74
75       // Eine Frage erstellen und an den Server schicken:
76       OtpErlangLong     fZahlOtp = new OtpErlangLong(fZahl);
77       OtpErlangObject[] frageR   = new OtpErlangObject[2];
78       frageR[0]                  = clientProzess.self();
79       frageR[1]                  = fZahlOtp;
80       OtpErlangTuple    frageT   = new OtpErlangTuple(frageR);
81
82       clientProzess.send(serverProzessName, serverKnotenName, frageT);
83       printf("send(%s, %s, %s)%n",
84          serverProzessName, serverKnotenName, frageT);
85
86       // Maximal 2 sec auf eine Antwort warten:
87       OtpErlangObject antwortObj = clientProzess.receive(2000);
88
89       printf("Antwort: %s%n", antwortObj);
90
91       // Aus der Erlang-Antwort die Java-long-Zahl extrahieren und ausgeben:
92       if (antwortObj != null) {
93          OtpErlangTuple antwortT = (OtpErlangTuple) antwortObj;
94          OtpErlangLong  aZahlOtp = (OtpErlangLong) antwortT.elementAt(1);
95          Long           aZahl    = aZahlOtp.longValue();
96          printf("aZahl: %,d%n", aZahl);
97       }
98       printf("--------------------------------%n");
99       printf("Plus3Client: Das war's erstmal!%n");
100   } // main
101   // ---------------------------------------------------------------------
102   // Eine Methode mit einem kurzen Namen:
103   static void printf(String f, Object... v) {System.out.printf(f, v);}
104   // ---------------------------------------------------------------------
105 } // class Plus3Client
```

Man kann den Erlang-Server-Prozess auch in Erlang programmieren

```
 1 % Datei plus3Server.erl
 2
 3 -module(plus3Server).
 4 -export([start/0, funktion01/0]).
 5 % ---------------------------------------------------------------------
 6 start() ->
 7    io:format("AA plus3Server:start wird ausgefuehrt!~n"),
 8
 9    Hst = net_adm:localhost(),
10    Arg = '-setcookie abc',
11    {ok, Node} = slave:start(Hst, 'k01', Arg),
12
13    spawn(Node, plus3Server, funktion01, []).
14 % ---------------------------------------------------------------------
15
```

```
16 funktion01() ->
17    io:format("C plus3Server-Prozess auf knoten ~p wird gestartet~n", [node()]),
18    register('plus3Server', self()),
19    naechsteFrage().
20 % ---------------------------------------------------------------------
21 naechsteFrage() ->
22    receive
23       {Pid, 999} ->
24          Pid ! {ok, 999+3};
25       {Pid, N} ->
26          Pid ! {ok, N+3},
27          naechsteFrage();
28       Sonst ->
29          naechsteFrage()
30    end.
31 % ---------------------------------------------------------------------
```

Und den Erlang-Klienten kann man auch in Erlang programmieren

```
 1 % Datei plus3Client.erl
 2 % ---------------------------------------------------------------------
 3 % Dieses Programm ist ein Erlang-Klient fuer den Java-Plus3Server.
 4 % Dieser Klient kommuniziert mit dem Server ueber Erlang-Knoten und
 5 % Erlang-Prozesse.
 6 %
 7 % In einem Erlang-Fenster kann man diesen Klienten z.B. so aufrufen:
 8 % 8> plus3Client(17).
 9 %
10 % Der Parameter (im Beispiel: 17) wird zum Server geschickt. Der sollte
11 % eine um 3 groessere Zahl zurueckschicken. Der Klient gibt die Antwort
12 % des Servers zum Erlang-Fenster aus.
13 %
14 % Wenn der Klient dem Server die Zahl 999 schickt dann beendet der sich
15 % (nachdem er die Antwort 1002 zurueckgeschickt hat).
16 % ---------------------------------------------------------------------
17 % Bevor man dieses Klient-Programm startet, sollte man den Server starten
18 % (siehe dazu die Bedienungsanleitung in der Datei Plus3Server.java).
19 %
20 % Ausser diesem Java-Klienten gibt es auch einen Java-Klienten (siehe
21 % die Datei Plus3Client.java), der sich ganz aehnlich verhaelt.
22 % ---------------------------------------------------------------------
23 -module(plus3Client).
24 -export([p3/1]).
25
26 p3(N) ->
27    io:format("plus3client:p3(~p) wurde aufgerufen!~n", [N]),
28    NodeT = "k01@" ++ net_adm:localhost(),
29    NodeA = list_to_atom(NodeT),
30    io:format("plus3client:p3(~p) NodeA: ~p!~n", [N, NodeA]),
31    {plus3Server, NodeA} ! {self(), N},
32    receive
33       {ok, Erg} -> io:format("p3(~p) ist gleich ~p~n", [N, Erg])
34    after 2000   -> io:format("Timeout in p3(~p)~n", [N])
35    end.
36 % ---------------------------------------------------------------------
```