Im Beispiel spaltet `main` einen Thread ab. Dann sendet sie diesem einen String und wartet auf Antwort. Entsprechend wartet der Thread auf Übertragung eines `string`-Werts auf dem Kanal. Sobald er ihn erhalten hat, sendet er auf dem selben Kanal eine Antwort.

Achtung!

Ist die Abfolge von `send` und `receive` nicht sorgfältig designt, können Threads leicht blockiert werden ...

Die Ausführung des Programms liefert:

```
> ./a.out
main is sending ... Greetings!
He got it!
>
```

Beispiel: Eine globale Speicherzelle

Eine globale Speicherzelle, insbesondere in Anwesenheit mehrerer Threads sollte die Signatur `Cell` implementieren:

```
module type Cell = sig
   type 'a cell
   val new_cell : 'a -> 'a cell
   val get : 'a cell -> 'a
   val put : 'a cell -> 'a -> unit
end
```

Dabei muss sichergestellt werden, die verschiedenen `get`- und `put`-Aufrufe sequentialisiert ausgeführt werden.

Diese Aufgabe erfüllt ein Server-Thread, mit dem `get` und `put` kommunizieren:

```
type 'a req  = Get of 'a channel | Put of 'a
type 'a cell = 'a req channel
```

Der Kanal transportiert Requests an die Speicherzelle, welche entweder den zu setzenden Wert oder den Rückkanal enthalten ...

```
let get req = let reply = new_channel ()
              in  sync (send req (Get reply));
                  sync (receive reply)
```

Die Funktion `get` sendet einen neuen Rückkanal auf `req`. Ist dieser angekommen, wartet sie auf die Antwort.

```
let put req x = sync (send req (Put x))
```

Die Funktion `put` sendet ein `Put`-Element, das den neuen Wert der Speicherzelle enthält.

Spannend ist jetzt nur noch die Implementierung der Zelle selbst:

```
let new_cell x = let req = new_channel ()
          in let rec serve x = match sync (receive req)
             with Get reply -> sync (send reply x);
                               serve x
                | Put y     -> serve y
          in
             create serve x;
             req
```

Beim Anlegen der Zelle mit dem Wert `x` wird ein Server-Thread abgespalten, der den Aufruf `serve x` auswertet.

Achtung:

der Server-Thread ist potentiell nicht-terminierend!

Nur deshalb kann er beliebig viele Requests bearbeiten :-)

Nur weil er end-rekursiv programmiert ist, schreibt er dabei nicht sukzessive den gesamten Speicher voll ...

```
let main = let x = new_cell 1
        in print_int (get x); print_string "\n";
           put x 2;
           print_int (get x); print_string "\n"
```

Dann liefert die Ausführung:

```
> ./a.out
1
2
>
```

Anstelle von `get` und `put` könnte man auch kompliziertere Anfrage- bzw. Update-Funktionen vom `cell`-Server ausführen lassen ...

Beispiel: Locks

Oft soll von mehreren möglicherweise aktiven Threads nur ein einziger auf eine bestimmte Resource zugreifen. Um solchen wechselseitigen Ausschluss zu implementieren, kann man Locks verwenden:

```
module type Lock = sig
   type lock
   type ack
   val new_lock : unit -> lock
   val acquire : lock -> ack
   val release : ack -> unit
end
```

Ausführen der Operation `acquire` liefert ein Element des Typs `ack`, dessen Rückgabe den Lock wieder frei gibt:

```
type ack = unit channel
type lock = ack channel
```

Der Einfachkeit halber wählen wir `ack` einfach als den Kanal, über den die Freigabe des Locks erfolgen soll :-)

```
let acquire lock = let ack = new_channel ()
                   in  sync (send lock ack);
                       ack
```

Der Freigabe-Kanal wird von `acquire` hergestellt :-)

```
let release ack = sync (send ack ())
```

... und von der Operation `release` benutzt.

```
let new_lock () = let lock = new_channel ()
                  in let rec acq_server () =
                                 rel_server (sync (receive lock))
                             and rel_server ack =
                                 sync (receive ack);
                                 acq_server ()
                  in create acq_server ();
                     lock
```

Herzstück der Implementierung sind die beiden wechselseitig rekursiven Funktionen `acq_server` und `rel_server`.

`acq_server` erwartet ein `ack`-Element, d.h. einen Kanal, und ruft nach Erhalt `rel_server` auf.

`rel_server` erwartet über den erhaltenen Kanal ein Signal, dass die Resource auch freigegeben wurde ...

Jetzt sind wir in der Lage, einen anständigen Deadlock zu programmieren:

```
let dead  = let l1 = new_lock ()
          in let l2 = new_lock ()
  ...
```

```
in let th (l1,l2) = let a1 = acquire l1
                    in let _ = delay 1.0
                    in let a2 = acquire l2
                    in release a2; release a1;
                       print_int (id (self ()));
                       print_string " finished\n"
in let t1 = create th (l1,l2)
in let t2 = create th (l2,l1)
in join t1
```

Das Ergebnis ist:

```
> ./a.out
```

Ocaml wartet und wartet :-)

Beispiel: Semaphore

Gelegentlich gibt es mehr als ein Exemplar einer Resource. Dann sind Semaphore ein geeignetes Hilfsmittel ...

```
module type Sema = sig
type sema
new_sema : int -> sema
up   : sema -> unit
down : sema -> unit
end
```

Idee:

Wir implementieren wieder einen Server, der in einem akkumulierenden Parameter die Anzahl der noch zur Verfügung stehenden Resourcen bzw. eine Schlange der wartenden Threads enthält ...

```
module Sema = struct open Thread Event
type sema = unit channel option channel
let up sema = sync (send sema None)
let down sema = let ack = (new_channel : unit channel)
                in sync (send sema (Some ack));
                   sync (receive ack)
...
```

```
...
let new_sema n = let sema = new_channel ()
    in let rec serve (n,q) =
       match sync (receive sema)
        with None -> (match dequeue q
             with  (None,q) -> serve (n+1,q)
             | (Some ack,q) -> sync (send ack ());
                               serve (n,q))
        | Some ack -> if n>0 then sync (send ack ());
                                  serve (n-1,q)
                      else serve (n,enqueue ack q)
    in create serve (n,new_queue()); sema
end
```

Offensichtlich verwalten wir in der Schlange nicht die Threads selbst, sondern ihre jeweiligen Rückantwort-Kanäle :-)

7.2 Selektive Kommunikation

Häufig weiß ein Thread nicht, welche von mehreren möglichen Kommunikations-Rendezvous zuerst oder überhaupt eintrifft.
Nötig ist eine nicht-deterministische Auswahl zwischen mehreren Aktionen ...

Beispiel: Die Funktion

```
add : int channel * int channel * int channel -> unit
```

soll Integers aus zwei Kanälen lesen und die Summe auf dem dritten senden.

1. Versuch:

```
   let forever f init = let
       let rec loop x = loop (f x)
       in create loop init
in let add1 (in1, in2, out) = forever (fun () ->
       sync (send out (sync (receive in1) +
                           sync (receive in2))
       )) ()
```

Nachteil:

Kommt am zweiten Input-Kanal bereits früher ein Wert an, muss der Thread trotzdem warten.

2. Versuch:

```
let add (in1, in2, out) = forever (fun () ->
    let (a,b) = select [
        wrap (receive in1) (fun a -> (a, sync (receive in2)));
        wrap (receive in2) (fun b -> (sync (receive in1), b))
        ]
    in sync (send out (a+b))
    ) ()
```

Dieses Programm müssen wir uns langsam auf der Zunge zergehen lassen :-)

Idee:

→ Initiierung von Input- wie Output-Operationen erzeugen Events.

→ Events sind Daten-Objekte des Typs: `'a event`.

→ Die Funktion:

```
wrap : 'a event -> ('a -> 'b) -> 'b event
```

wendet eine Funktion nachträglich auf den Wert eines Events – sollte er denn eintreffen – an.

Die Liste enthält damit `(int,int)`-Events.

Die Funktionen:

```
choose : 'a event list -> 'a event
select : 'a event list -> 'a
```

wählen nicht-deterministisch ein Event einer Event-Liste aus.

`select` synchronisiert anschließend auf den ausgewählten Event, d.h. stellt die nötige Kommunikation her und liefert den Wert zurück:

```
let select = comp sync choose
```

Typischerweise wird dabei dasjenige Event ausgewählt, das zuerst einen Partner findet :-)

Weitere Beispiele:

Die Funktion

```
copy : 'a channel * 'a channel * 'a channel -> unit
```

soll ein gelesenes Element auf zwei Kanäle kopieren: