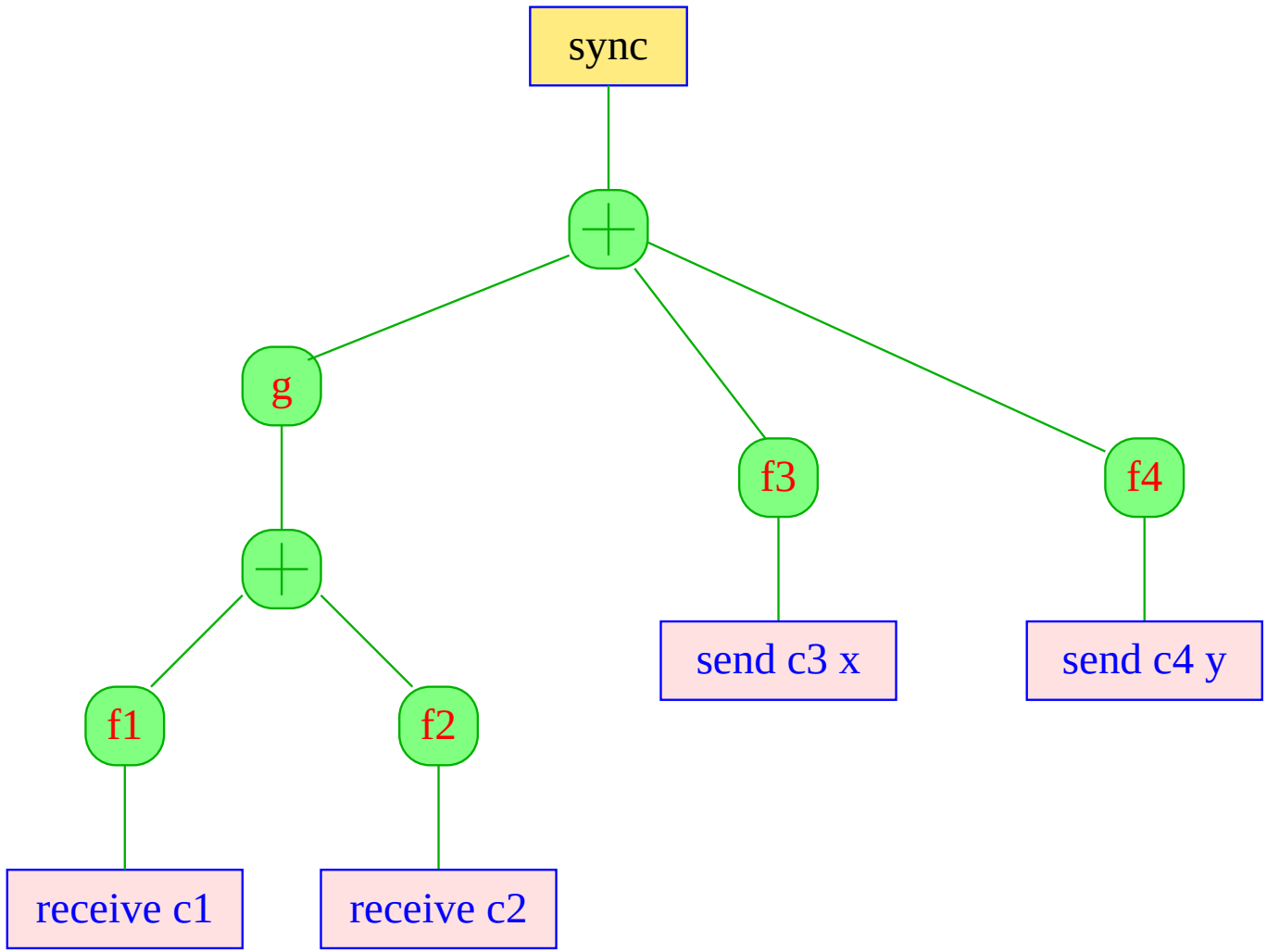
```
...
let get (get_chan,_) = sync (receive get_chan)
let put (_,put_chan) x = sync (send put_chan x)
let new_cell x = let get_chan = new_channel ()
                 in let put_chan = new_channel ()
                 in let serve x = select [
          wrap (send get_chan x) (fun () -> serve x);
          wrap (receive put_chan) serve
      ]
      in
         create serve x;
         (get_chan, put_chan)
```

Im allgemeinen kann ein Event-Baum vorliegen:



- → Die Blätter sind Basis-Events.
- → Auf ein Event kann eine Wrapper-Funktion angewandt werden.
- → Mehrere Ereignisse des gleichen Typs können mit einem Auswahl-Knoten zusammengefasst werden.
- → Synchronisierung auf einen Event-Baum aktiviert ein Blatt-Event.

  Als Ergebnis wird der Wert geliefert, den die Komposition der Wrapper-Funktionen auf dem Weg zur Wurzel liefert.

Beispiel: Ein Swap-Channel

Ein Swap-Channel soll beim Rendezvous die Werte der beiden beteiligten Threads austauschen. Die Signatur lautet:

```
module type Swap = sig
   type 'a swap
   val new_swap : unit -> 'a swap
   val swap : 'a swap -> 'a -> 'a event
end
```

Jeder beteiligte Thread muss in einer Implementierung mit normalen Kanälen sowohl die Möglichkeit zu empfangen wie zu senden anbieten.

Sobald ein Thread sein Senden erfolgreich beendete (d.h. der andere auf ein `receive`-Event synchronisierte), muss noch der zweite Wert übermittelt werden.

Mit dem ersten Wert übertragen wir deshalb einen Kanal für den zweiten Wert:

```
module Swap =
struct open Thread open Event
   type 'a swap = ('a * 'a channel) channel
   let new_swap () = new_channel ()
   ...
```

```
...
let swap ch x = let c = new_channel ()
    in choose [
       wrap (receive ch) (fun (y,c) ->
            sync (send c x); y);
       wrap (send ch (x,c)) (fun () ->
             sync (receive c))
       ]
```

Einen konkreten Austausch können wir implementieren, indem wir `choose` in `select` umwandeln.

Timeouts

Oft dauert unsere Geduld nur eine Weile :-)

Dann wollen wir ein angefangenes Sende- oder Empfangswarten beenden ...

```
module type Timer = sig
   set_timer : float -> unit event
   timed_receive : 'a channel -> float -> 'a option event
   timed_send    : 'a channel -> 'a -> float -> unit option event
end
```

```
module Timer = stuct open Thread Event
   let set_timer t = let ack = new_channel ()
                     in let serve () = delay t;
                                       sync (receive ack)
                     in create serve (); send ack ()

   let timed_receive ch time = choose [
       wrap (receive ch) (fun a -> Some a);
       wrap (set_timer time) (fun () -> None)
   ]

   let timed_send ch x time = choose [
       wrap (send ch x) (fun a -> Some ());
       wrap (set_timer time) (fun () -> None)
   ]
end
```

11.3 Threads und Exceptions

Eine Exception muss immer innerhalb des Threads behandelt werden, in der sie erzeugt wurde.

```
module Blow = struct open Thread
let thread x = (x / 0);
       print_string "thread terminated regularly ...\n"
let main = create thread 0; delay 1.0;
           print_string "main terminated regularly ...\n"
end
```

... liefert:

```
> /.a.out
Thread 1 killed on uncaught exception Division_by_zero
main terminated regularly ...
```

Der Thread wurde gekillt, das Ocaml-Programm terminierte trotzdem.

Auch ungefangene Exceptions innerhalb einer Wrapper-Funktion beenden den laufenden Thread:

```
module BlowWrap = struct open Thread open Event open Timer
let main = try sync (wrap (set_timer 1.0) (fun () -> 1 / 0))
           with _ -> 0;
           print_string "... this is the end!\n"
end
```

Dann liefert:

```
> ./a.out
Fatal error: exception Division_by_zero
```

Achtung:

Exceptions können nur im Rumpf der Wrapper-Funktion selbst, nicht aber hinter dem `sync` abgefangen werden !

11.4 Gepufferte Kommunikation

Ein Kanal für gepufferte Kommunikation erlaubt nicht blockierendes Senden. Empfangen dagegen blockiert, sofern keine Nachrichten vorhanden ist. Für solche Känals implementieren wir die Struktur `Mailbox` zur Verfügung:

```
module type Mailbox = sig
   type 'a mbox
   val new_mailbox : unit -> 'a mbox
   val send :    'a mbox -> 'a -> unit
   val receive : 'a mbox -> 'a event
end
```

Zur Implementierung benutzen wir einen Server, der eine Queue der gesendeten, aber noch nicht erhaltenen Nachrichten verwaltet.

Damit implementieren wir:

```
module Mailbox =
struct open Thread open Queue open Event
   type 'a mbox = 'a channel * 'a channel
   let send (in_chan,_) x     = sync (send in_chan x)
   let receive (_,out_chan)   = receive out_chan
   let new_mailbox () = let in_chan  = new_channel ()
                        and out_chan = new_channel ()
...
```

```
    ...
        in let rec serve q = if (is_empty q) then
                             serve (enqueue (
                             sync (Event.receive in_chan)) q)
           else select [
                  wrap (Event.receive in_chan)
                       (fun y -> serve (enqueue y q));
                  wrap (Event.send out_chan (first q))
                       (fun () -> let (_,q) = dequeue q
                                  in serve q)
                ]
        in create serve (new_queue ());
           (in_chan, out_chan)
end
```

... wobei `first : 'a queue -> 'a` das erste Element einer Schlange liefert, ohne es zu entfernen :-)

11.5 Multicasts

Für das Versenden einer Nachricht an viele Empfänger stellen wir die Struktur `Multicast` zur Verfügung, die die Signatur `Multicast` implementiert:

```
   module type Multicast = sig
      type 'a mchannel and 'a port
      val new_mchannel : unit -> 'a mchannel
      val new_port : 'a mchannel -> 'a port
      val multicast : 'a mchannel -> 'a -> unit
      val receive : 'a port -> 'a event
end
```

Die Operation `new_port` erzeugt einen neuen Port, an dem eine Nachricht empfangen werden kann.

Die Operation `multicast` sendet (nicht-blockierend) an sämtliche registrierten Ports.

```
module Multicast = struct open Thread open Event
module M = Mailbox
type 'a port = 'a M.mbox
type 'a mchannel = 'a channel * unit channel
                                * 'a port channel
let new_port (_, req, port)   = sync (send req ());
                                sync (receive port)
let multicast (send_ch,_,_) x = sync (send send_ch x)
let receive mbox              = M.receive mbox
    ...
```

Die Operation `multicast` sendet die Nachricht auf dem Kanal `send_ch`. Die Operation `receive` liest aus der Mailbox des Ports.

Der Multicast-Kanal selbst wird von einem Server-Thread überwacht, der eine Liste der zu bedienenden Ports verwaltet:

```
let new_mchannel () = let send_ch = new_channel ()
                      in let req = new_channel ()
                      in let port = new_channel ()
                      in let send_port x mbox = M.send mbox x
    ...
```

```
    ...
 in let rec serve ports = select [
        wrap (Event.receive req) (fun () ->
              let p = M.new_mailbox ()
              in sync (send port p);
                 serve (p :: ports));
        wrap (Event.receive send_ch) (fun x ->
              create (iter (send_port x)) ports;
              serve ports)
        ]
    in create serve [];
       (send_ch, req, port)
...
```

Beachte, dass der Server-Thread sowohl auf Port-Requests auf dem Kanal `req` wie auf Sende-Aufträge auf dem Kanal `send_ch` gefasst sein muss.

Achtung:

Unsere Implementierung gestattet zwar das Hinzufügen, nicht aber das Entfernen nicht mehr benötigter Ports.

Zum Ausprobieren benutzen wir einen Test-Ausdruck `main`:

```
    ...
  let main  = let mc = new_mchannel ()
      in let thread i = let p = new_port mc
         in while true do let x = sync (receive p)
                          in print_int i; print_string ": ";
                             print_string (x^"\n")
                       done
      in  create thread 1; create thread 2;
          create thread 3; delay 1.0;
          multicast mc "Hallo!";
          multicast mc "World!";
          multicast mc "... the end.";
          delay 10.0
  end
end
```

Dann haben wir:

```
-  ./a.out
3: Hallo!
2: Hallo!
1: Hallo!
3: World!
2: World!
1: World!
3: ... the end.
2: ... the end.
1: ... the end.
```

Fazit:

- Die Programmiersprache Ocaml bietet komfortable Möglichkeiten an, nebenläufige Programme zu schreiben :-)
- Kanäle mit synchroner Kommunikation können Konzepte der Nebenläufigkeit wie asynchrone Kommunikation, globale Variablen, Locks für wechselseitigen Ausschluss und Semaphore simulieren ;-)
- Nebenläufige funktionale Programme können deshalb genauso undurchsichtig und schwer zu verstehen sein wie nebenläufige Java-Programme :-(
- Es werden Methoden benötigt, um systematisch die Korrektheit solcher Programme zu verifizieren ...

Abschluss:

- Jenseits der hier besprochenen Sprachkonzepte gibt es in Ocaml einige weitere Konzepte, die insbesondere objekt-orientierte Programmierung ermöglichen.
- Darüberhinaus bietet Ocaml elegante Möglichkeiten, Betriebssystemsfunktionalität auszunutzen, graphische Bibliotheken anzusteuern, mit anderen Rechnern zu kommunizieren ...

$\Longrightarrow$ Ocaml ist eine interessante Alternative zu Java.