FPV Endterm Exam 1617

Funktionale Programmierung (Technische Universität München)

Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann, J. Brunner 20.02.2017

Klausur

Vorname	Nachname
Matrikelnummer	Unterschrift

- Füllen Sie die oben angegebenen Felder aus.
- Schreiben Sie nur mit einem dokumentenechten Stift in schwarzer oder blauer Farbe.
- Verwenden Sie kein "Tipp-Ex" oder ähnliches.
- Die Arbeitszeit beträgt **120** Minuten.
- Prüfen Sie, ob Sie **11** Seiten erhalten haben.
- Sie können maximal **150** Punkte erreichen. Erreichen Sie mindestens **60** Punkte, bestehen Sie die Klausur.
- Als Hilfsmittel ist nur ein beidseitig handbeschriebenes A4-Blatt zugelassen.
- Alle Funktionen aus der Ocaml-Referenz dürfen ohne Angabe von Modulnamen verwendet werden.

vorzeitige Abgabe um ..... Hörsaal verlassen von ..... bis .....

1	2	3	4	5	6	7	Σ

..... Erstkorrektor

..... Zweitkorrektor

Aufgabe 1 Multiple-Choice [16 Punkte]

Kreuzen Sie in den folgenden Teilaufgaben jeweils die richtigen Antworten an. Es können pro Teilaufgabe keine, einige oder alle Antworten richtig sein. Die Teilaufgaben werden isoliert bewertet, Fehler in einer Teilaufgabe wirken sich also nicht auf die erreichbare Punktzahl bei anderen Teilaufgaben aus.

1. Betrachten Sie folgendes MiniJava-Programm:

```
1 while(x < 15) {
2   x = 2*x;
3 }
4 write(x);
```

   Welche der folgenden Bedingungen am Programmanfang (vor der Schleife) sind hinreichend, sodass am Programmende die Zusicherung $x = 16$ gilt?

   - [x] $x = 8$
   - [x] $x = 4$
   - [ ] $x = 0$
   - [ ] $x \geq 0$
   - [ ] $x \leq 0$
   - [ ] $x < 0$
   - [x] $x > 0$
   - [ ] $x = -42$
   - [ ] `false`
   - [ ] `true`

2. Ordnen Sie die folgenden Zusicherungen für $i \in \mathbb{Z}$ mittels Implikation und Äquivalenz. Beispiel: $(a) \Rightarrow (b) \equiv (d)$...

   (a) $i^2 > 0$

   (b) $i > 0$

   (c) $2 * i > 0$ $i > 0$

   (d) $i - 1 > 0$ $i > 1$

   (e) $i + 1 > 1$ $i > 0$

   (f) $i \in [131; \infty] \wedge i > 100$

   (g) $|i| \geq 0$ $a \Rightarrow f$

   Antwort: $a \Rightarrow g$ $g \equiv b \equiv c \equiv e$ $a \Rightarrow d$

3. Eine gültige Zusicherung am Ende eines MiniJava-Programms

   - [ ] gilt nur unter der Annahme von Terminierung.
   - [x] impliziert Terminierung.
   - [ ] gilt immer nur ohne Terminierung.

- [ ] gilt nur an verkaufsoffenen Sonntagen.
- [ ] gilt unabhängig von der Terminierung.

4. Welchen Wert berechnet der Ausdruck `List.fold_left ( * ) 0 [6; 2; 1; 3]`?

- [ ] `0`
- [x] `36`
- [ ] `[0; 0; 0; 0]`
- [ ] `[36; 4; 1; 9]`

5. Welchen Wert berechnet der Ausdruck `List.map (fun x -> [x :: []]) [(2, 8)]`?

- [ ] `[2; 8]`
- [ ] `[[2]; [8]]`
- [x] `[[(2, 8)]]`
- [ ] `[[[(2, 8)]]]`

6. Welchen Typ hat der Ausdruck `let f a b = a + a in f`?

- [ ] `int`
- [ ] `int -> int`
- [x] `int -> 'a -> int`
- [ ] `int -> int -> int`

7. Der Ausdruck `(17, (fun i -> [2 * i - 7]) 3) :: (0, [6])`

- [ ] berechnet den Wert `[(17, []); (0, [6])]`.
- [x] hat den Typ `(int * int) list`.
- [ ] verwendet partielle Funktionsapplikation (Currying).
- [ ] enthält einen Typfehler.

8. Der Ausdruck `(fun a b -> a (a b)) (fun a -> a + 1) 7`

- [x] berechnet den Wert 9.
- [ ] hat den Typ `int -> (int -> int) -> int`.
- [ ] enthält eine Funktion höherer Ordnung.
- [ ] enthält einen Typfehler.

Aufgabe 2 Endrekursion [20 Punkte]

Bearbeiten Sie die folgenden Teilaufgaben. Sie können jeweils in Stichpunkten antworten.

1. Wann ist eine Funktion endrekursiv?
2. Welchen Vorteil bringen endrekursive Funktionen gegenüber Funktionen, die nicht endrekursiv sind?
3. Warum ist die Optimierung von endrekursiven Funktionen gerade bei funktionalen Programmiersprachen von besonderer Bedeutung?
4. Geben Sie für die folgenden Funktionen an, ob sie endrekursiv sind oder nicht:

   (a)
   ```
   let rec g x =
     if x < 0 then g (0 - x)
     else if x = 0 then 1
     else -g (x-1)
   ```

   ☐ Endrekursiv
   ☐ Nicht endrekursiv

   (b)
   ```
   let rec f a b =
     if a < 7 then f (a+1) [a] @ b
     else if a > 7 then f (a-1) b @ [a]
     else a :: b
   ```

   ☐ Endrekursiv
   ☐ Nicht endrekursiv

   (c)
   ```
   let rec a m n = match (m,n) with
     | (0, _) -> n + 1
     | (_, 0) -> a (m-1) 1
     | _      -> a (m-1) (a m (n-1))
   ```

   ☐ Endrekursiv
   ☐ Nicht endrekursiv

5. Gegeben sei die folgende Funktion:

```
1 let rec f a b c =
2   if c = 0 then 1 else a * b * f a b (c-1)
```

   Implementieren Sie die Funktion endrekursiv. Verändern Sie den Typ der Funktion dabei nicht.

Aufgabe 3 Kleine Kamele [22 Punkte]

Gegeben seien folgende MiniOcaml-Funktionen:

```
let rec rev1 x y = match x with
    [] -> y
  | x::xs -> rev1 xs (x::y)

let rec fold_left f acc l = match l with
    [] -> acc
  | x::xs -> fold_left f (f acc x) xs
```

Bearbeiten Sie folgende Teilaufgaben.

1. Zeigen Sie mittels der Big-Step-Semantik, dass der Ausdruck

   ```
   fold_left (fun acc x -> x::x::acc) 3::4::[] 1::[]
   ```

   in MiniOcaml zu `1::1::3::4::[]` ausgewertet wird. Sie dürfen Setzungen vornehmen. Es ist außerdem erlaubt, Regelanwendungen auszulassen, sofern der ausgewertete Ausdruck aus einem einzigen Wert besteht („$v \Rightarrow v$“).

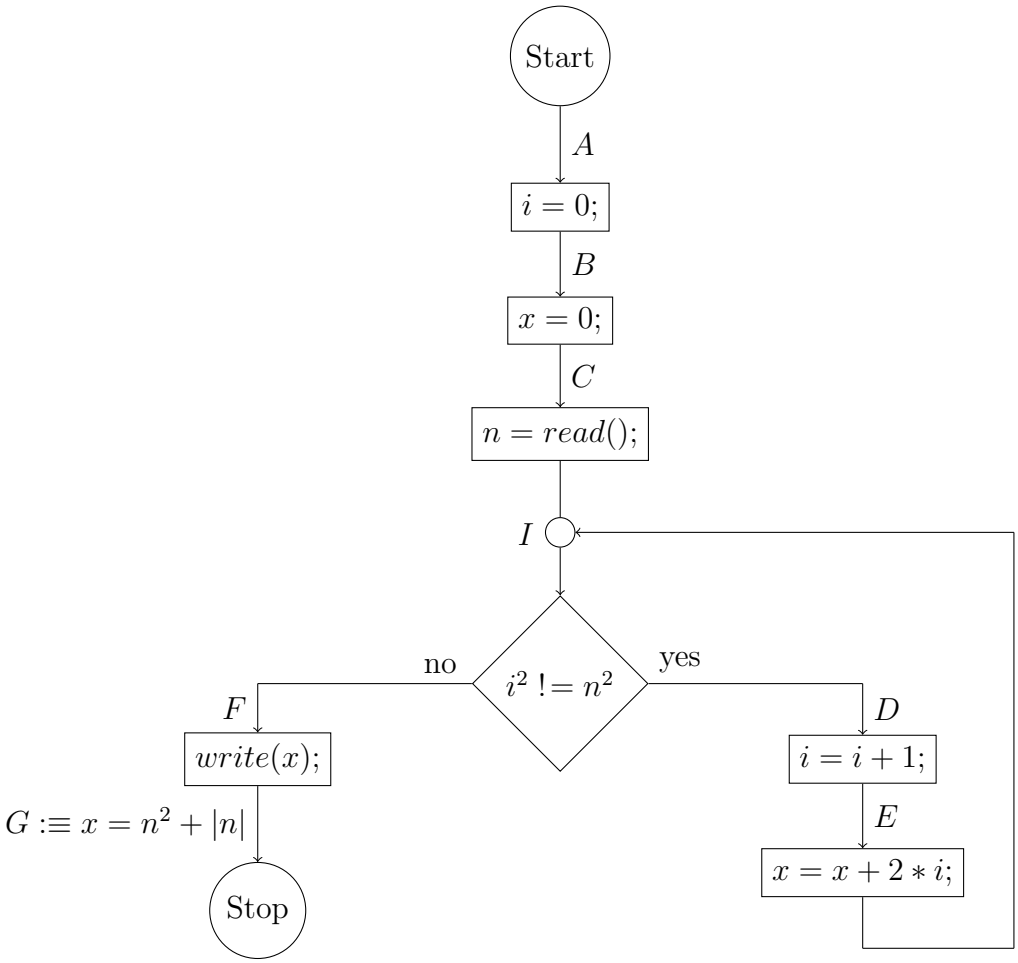
2. Zeigen Sie durch einen Induktionsbeweis, dass

   ```
   fold_left (fun acc x -> x::acc) [] x = rev1 x []
   ```

   gilt.

Aufgabe 4 Petra von Gauß [22 Punkte]

Es sei der folgende Kontrollflussgraph gegeben:



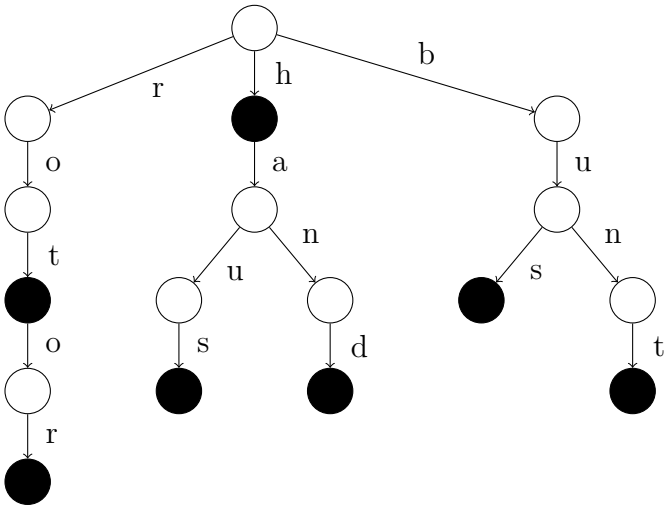
Zeigen Sie, dass am Programmpunkt G die Zusicherung $x = n^2 + |n|$ gilt.

Hinweis:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

Aufgabe 5 Eins, Zwei, Trie [23 Punkte]

Ein Trie (auch Präfixbaum) ist ein Suchbaum zur effizienten Speicherung von Wörtern. Jeder Knoten im Baum repräsentiert dazu ein bestimmtes Wort. Außerdem sind die Kanten, die von einem Knoten zu dessen Kindern führen, mit verschiedenen Buchstaben versehen.



Möchte man nun durch den Baum navigieren, um zum Beispiel das Wort "rot" zu finden, beginnt man bei der Wurzel, welche das leere Wort repräsentiert. Von dort aus verfolgt man die Kanten, die mit den Buchstaben 'r', 'o' und 't' (in dieser Reihenfolge) versehen sind, um zum Knoten für das gesuchte Wort zu gelangen. Ein Trie enthält damit für jedes gespeicherte Wort auch Knoten für alle Präfixe dieses Wortes (z.B. "", "r" und "ro"). Deshalb erfordert die Speicherung des Wortes "rotor" lediglich zwei zusätzliche Knoten. Um explizit gespeicherte Wörter von denjenigen Wörtern zu unterscheiden, die lediglich Präfix eines gespeicherten Wortes sind, enthält jeder Knoten ein Boolsches Flag, welches für erstere auf *true* (schwarze Knoten) und letztere auf $false$ (weiße Knoten) gesetzt wird.

Zur Repräsentation eines Trie sei der folgende OCaml-Datentyp gegeben:

```
type trie = Node of bool * (char * trie) list
```

1. Implementieren Sie die Funktion

   ```
   val insert : string -> trie -> trie,
   ```

   welche ein Wort in einen Trie einfügt. Verwenden Sie die Funktion `val explode : string -> char list`, um das Wort in eine Liste der Buchstaben zu zerlegen.

2. Implementieren Sie die Funktion

   ```
   val merge : trie -> trie -> trie,
   ```

   die zwei Tries t_1 und t_2 so verschmilzt, dass ein neuer Trie t entsteht, der ein Wort w genau dann enthält, wenn w entweder in t_1, t_2 oder in beiden Tries enthalten ist.

Hinweis: Da Knoten ihre Kinder in einer (assoziativen) Liste speichern, können einige Funktionen aus dem OCaml `List` Modul von großem Nutzen sein (siehe Anhang).
Hinweis: Sie dürfen entscheiden, ob die Kindlisten der Knoten sortiert sind; Ihre Implementierung muss aber über alle Teilaufgaben konsistent sein!

Aufgabe 6 Tiefenentspannung mittels Funktor [22 Punkte]

In dieser Aufgabe soll die Tiefensuche auf Bäumen und (sonstigen) Graphen mittels eines Funktors implementiert werden. Zur Erinnerung sei zunächst die Definition der Tiefensuche gegeben.

Tiefensuche: Die Tiefensuche besucht die Knoten eines Graphen ausgehend von einem Startknoten. Dazu folgt sie Kanten zu den Nachfolgern des Startknotens. Sie besucht dabei jeweils zunächst den Startknoten und anschließend rekursiv die Knoten, die vom ersten Nachfolger des Startknoten aus erreichbar sind. Anschließend fährt sie mit dem zweiten Nachfolger des Startknotens fort usw. Um Terminierung zu sichern, besucht die Tiefensuche jeden Knoten höchstens einmal.

Die Signatur eines Graphen, der Integer-Zahlen (ohne Duplikate) in seinen Knoten hält, habe hier die folgende Signatur:

```
1  module type Graph = sig
2    type node
3    type graph
4
5    (* Gibt die Liste direkter Nachfolger des gegebenen Knotens zurück
     ↪  *)
6    val successors : graph -> node -> node list
7  end
```

Bearbeiten Sie nun folgende Teilaufgaben.

1. Implementieren Sie ein Modul `BinaryTree`, welches die obige Signatur umsetzt (Sie müssen nur die von der Signatur geforderten Typen und Funktionen implementieren). Das Modul soll als Graphstruktur einen binären Baum verwenden, wobei Teilbäume jeweils in den Knoten gespeichert werden.

2. Implementieren Sie ein Modul `GraphImpl`, welches die obige Signatur umsetzt (Sie müssen nur die von der Signatur geforderten Typen und Funktionen implementieren). Das Modul soll einen gerichteten Graphen implementieren, der Kanten zwischen beliebigen Knoten erlaubt. Nutzen Sie Listen, um Kanten bzw. Knoten zu speichern.

3. Implementieren Sie den Funktor `MakeGraphSearch`, der, gegeben ein Modul `G` der Signatur `Graph`, die Tiefensuche implementiert. Dazu soll der Funktor die Funktion `val dfs : G.graph -> G.node -> ('a -> G.node -> 'a) -> 'a -> 'a` anbieten, die über einen Graphen (1. Parameter), ausgehend von einem Startknoten (2. Parameter) eine Funktion `f` (3. Parameter) mittels einer Tiefensuche faltet. Der Startwert der Faltung ist dabei der letzte Parameter. Die Funktion `f` soll die Schlüssel der Knoten in der Reihenfolge ihrer Entdeckung bearbeiten. **Hinweis:** Speichern Sie bereits bekannte Knoten in einer Liste, um mehrfaches Besuchen auszuschließen!

Aufgabe 7 Paralleles Map + Reduce [25 Punkte]

In dieser Aufgabe soll eine Funktion `val map_reduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a list -> 'b` parallel mittels Threads implementiert werden. Die Funktion bildet dabei zunächst die Elemente einer nicht leeren[1] Liste mittels einer Funktion `f` ab (*map*-Phase). Über die Ergebnisliste wird anschließend eine zweite Funktion `g` gefaltet (*reduce*-Phase). Insgesamt kann `map_reduce` unter Nutzung nur eines Threads also wie folgt implementiert werden:

```
1 let map_reduce f g l =
2   let l' = List.map f l in
3   List.fold_left g (List.hd l) (List.tl l')
```

Um diese Funktion effizient parallelisieren zu können, setzen wir zusätzlich voraus, dass `g` assoziativ und kommutativ ist; typische Beispiele für derartige Funktionen sind die Addition und die Multiplikation.

Gehen Sie entsprechend der folgenden Teilaufgaben vor.

1. Implementieren Sie die Funktion `val pmap : ('a -> 'b) -> 'a list -> 'b event list`, die parallel eine Funktion `f` auf jedes Element einer Liste anwendet. Jede Anwendung von `f` soll dabei in einem eigenen Thread geschehen. Das Ergebnis ist eine Liste von Ereignissen. Jedes Ereignis entspricht dem Ergebnis der Anwendung von `f` auf das jeweilige Listenelement. **Hinweis**: `pmap` soll nicht auf die Fertigstellung der asynchronen Berechnung warten!

2. Implementieren Sie die Funktion `val preduce : ('a -> 'a -> 'c) -> 'a -> 'a -> 'c event`, die zwei Elemente mittels einer Funktion `g` parallel vereint. Die Vereinigung soll dabei in einem eigenen Thread geschehen. Das Ergebnis ist ein Ereignis, dessen Eintreten der Fertigstellung der Anwendung von `g` entspricht. **Hinweis**: `preduce` soll nicht auf die Fertigstellung der asynchronen Berechnung warten!

3. Implementieren Sie die Funktion `val reduce_list : ('a -> 'a -> 'a) -> 'a event list -> 'a`, die als Parameter eine Funktion `g` sowie eine Liste von Ereignissen (Rückgabe von `pmap`) erwartet. Die Funktion wartet wiederholt auf das Eintreten zweier beliebiger Ereignisse aus der Ereignisliste. Die beiden Ereignisse liefern zwei Werte, aus denen mittels `preduce` ein neues Ereignis erzeugt wird; das neue Ereignis wird dann in die Liste eingefügt. Die Menge der ausstehenden Ereignisse schrumpft so sukzessive, bis nur mehr ein Ereignis übrig bleibt, dessen Ergebnis als Rückgabewert der Funktion dient.

4. Implementieren Sie die Funktion `val map_reduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a list -> 'b` auf Basis obiger Funktionen.

[1] Sie müssen Fehler nicht behandeln.

Anhang

Big-Step Semantik

Axiome: $v \Rightarrow v$ für jeden Wert v

Tupel: $$\text{T}\ \frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen: $$\text{L}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen: $$\text{GD}\ \frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

Lokale Definitionen: $$\text{LD}\ \frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_0 \Rightarrow v_0}$$

Funktionsaufrufe: $$\textsc{App}\ \frac{e_1 \Rightarrow \texttt{fun}\ x\ \texttt{->}\ e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1\ e_2 \Rightarrow v_0}$$

Funktionsaufrufe mit mehreren Argumenten: $$\textsc{App}',\ \frac{e_0 \Rightarrow \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e \quad e_1 \Rightarrow v_1\ \ldots\ e_k \Rightarrow v_k \quad e[v_1/x_1\ \ldots\ v_k/x_k] \Rightarrow v}{e_0\ e_1\ \ldots\ e_k \Rightarrow v}$$

Pattern Matching: $$\text{PM}\ \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \ldots, v_k/x_k] \quad e_i[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

— sofern v' auf keines der Muster $p_1, \ldots, p_{i-1}$ passt

Eingebaute Operatoren: $$\textsc{Op}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1\ \texttt{op}\ v_2 \Rightarrow v}{e_1\ \texttt{op}\ e_2 \Rightarrow v}$$

— Unäre Operatoren werden analog behandelt.

Substitutionslemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

Ocaml Referenz

Modul List

Signatur	Erklärung
`val map : ('a -> 'b) -> 'a list -> 'b list`	`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.
`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`	`fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
`val filter : ('a -> bool) -> 'a list -> 'a list`	`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.
`val exists : ('a -> bool) -> 'a list -> bool`	`exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.
`val assoc : 'a -> ('a * 'b) list -> 'b`	`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with a in the list `l`.
`val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`	`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any.
`val mem_assoc : 'a -> ('a * 'b) list -> bool`	Same as `assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.
`val partition : ('a -> bool) -> 'a list -> 'a list * 'a list`	`partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.

Modul Batteries.String

`val explode : string -> char list`	`explode s` returns the list of characters in the string `s`.
`val implode : char list -> string`	`implode cs` returns a string resulting from concatenating the characters in the list `cs`.

Modul Thread und Event

`val create : ('a -> 'b) -> 'a -> t`	`create funct arg` creates a new thread of control, in which the function application `funct arg` is executed concurrently with the other threads of the program.
`val send : 'a channel -> 'a -> unit event`	`send c x` sends a value `x` over the channel `c`. It returns an event that occurs as soon as value is received.
`val receive : 'a channel -> 'a event`	`receive c x` returns an event that occurs as soon as a value is received from the channel.
`val sync : 'a event -> 'a`	`sync e` waites for a single event `e` to occur.
`val select : 'a event list -> 'a`	`select l` waites for any event in `l` to occur. The list may contain events that already occurred.
`val new_channel : unit -> 'a channel`	`new_channel ()` creates a new channel.