CSci 148, Spring 2002, Exam 2

Name (please print): _____ ID #: _____
First Last

- This is a 75-minute closed-book exam. There are **six questions worth a total of 95 points. Everybody gets** 5 free points, so 100 points are possible. Budget your **time so you get to all the questions. If a question seems** unclear, write a short note about what assumption(s) you are making, and proceed as best as you can.

1.	____	5.	____
2.	____	6.	____
3.	____		+5
4.	____	Total.	____

(for grader use only)

Question 1. (12 points) Give a value of type `int event` that, when synchronized, does the following: (a) reads an integer value from channel `ch1`, then (b) reads an integer value from channel `ch2`, then (c) writes the sum of the two integers to channel `ch3`, and then finally (d) produces the (integer) sum. This event should be constructed in such a way that it has communication (b) as its commit point.

Question 2. (10 points) The book defines a function with signature

```
val mkEvent :  'a event * string * string -> 'a event
```

such that synchronizing on the event `mkEvent(e, ackMsg, nackMsg)` first synchronizes on `e`, and then if `e` is chosen, prints `ackMsg`, whereas if `e` is not chosen, prints `nackMsg`. For example, the synchronization

```
select [
    mkEvent(recvEvt ch1, "ch1\n", "not ch1\n"),
    mkEvent(recvEvt ch2, "ch2\n", "not ch2\n")
]
```

will either print the messages "`ch1`" and "`not ch2`" (in some order) or print the messages "`ch2`" and "`not ch1`" (in some order), depending on whether the communication on `ch1` or `ch2` is enabled first. Give a definition of `mkEvent`. (Hints: Use `withNack`, spawning a separate process in the "guard" function to wait for the negative acknowledgement. Printing can be achieved using `TextIO.print` of type `string -> unit`.)

Question 3. (23 = 16 + 6 + 3 points) Suppose that there are two global channels defined:

```
val reqCh : (int * int ivar) chan    (* request channel *)
val updCh : int chan                 (* update channel  *) .
```

For this question, you will define three functions with signatures

```
val addServer : int -> unit   (* RPC server constructor *)
val addn      : int -> int    (* client call #1         *)
val update    : int -> unit   (* client call #2         *)
```

and with the following behavior:

- The call `addServer` n creates an "add n" server, with local state n, that accepts requests on the two global channels. An *add* request of the form (i, iv) on `reqCh` stores the value of $i+n$ into the I-variable iv and leaves the server state unchanged. An *update* request of the form m on `updCh` turns the server into an "add m" server (i.e., its local state becomes m).
- The call `addn` i computes and returns $i + n$ by making an add request on `reqCh` to the server (n is the local state of the server)
- The call `update` m makes an update request on `updCh` to the server, turning it into an "add m" server.

You may assume that the `CML` and `SynchVar` structures have been opened, and you are **not** allowed to use the `MakeRPC` structure.

Question 4. ($14 = 7 \times 2$ points) For each of the following CML primitives, indicate its type by writing it next to its name:

`CML.sendEvt`

`CML.recvEvt`

`CML.wrap`

`CML.guard`

`CML.withNack`

`CML.choose`

`CML.timeOutEvt`

Question 5. **($21 = 7 \times (1 + 2)$ points)** For each of the following CML primitives, indicate **both** its type and whether or not it can block when called (circle Y for "can block" or N for "does not block"):

Y N `CML.spawn`

Y N `CML.send`

Y N `CML.recv`

Y N `CML.sendPoll`

Y N `CML.recvPoll`

Y N `CML.synch`

Y N `SynchVar.iPut`

Y N `SynchVar.iGet`

Y N `SynchVar.mTake`

Y N `SynchVar.mSwap`

Y N `Mailbox.send`

Y N `Mailbox.recv`

Y N `Multicast.multicast`

Y N `Multicast.recv`

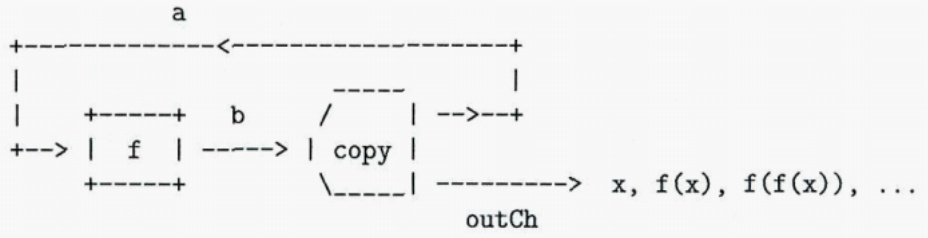
Question 6. (15 points) Recall the process network `copy` combinator

```
val copy : 'a chan * 'a chan * 'a chan -> unit ,
```

where, for channels c_1, c_2, and c_3, `copy`(c_1, c_2, c_3) creates a process that repeatedly reads values from channel c_1 and writes them to both c_2 and c_3. For this question, you will use this `copy` combinator to define a function `iterate` with signature

```
val iterate : 'a -> ('a -> 'a) -> 'a chan
```

The call `iterate` x f builds the following process network from f and `copy` and initializes it with the value x, returning the output channel, `outCh`:



Thus, the return value from the call `iterate` x f is a *stream* whose values are x, $f(x)$, $f(f(x))$, and so on. Note that your `iterate` function, besides creating channels for communication and "hooking together" the processes into the network shown and initializing it, will also need to create the *process* labelled "f" in the diagram above from the given *function* f.