CS350 2012 HW4

Due: November 4, 2012

1. Semantics of Trigger Creation for Lazy Execution. Extend your program in HW2 to handle the following statement.

   `[byneed [subr p] ident(x)]`

   with semantics equivalent to {`ByNeed P X`} described in the class. You need not program trigger activation semantics.

   [25 points]

2. Why does the trigger activation semantics require concurrency? Construct an example program where if trigger activation was done sequentially, you would break declarative concurrency.

   [10 points]

3. Write a function for the non-deterministic choose operation

   $$\{\texttt{Choose}[X_1\#S_1...X_n\#S_n]\}$$

   with the following behaviour. Here $X_1, \ldots, X_n$ are boolean-valued variables and $S_1, \ldots, S_n$ are statements. You can assume that the list is non-empty.

   If $X^{(1)}, \ldots, X^{(k)}$ is the subsequence of variables bound to `true`, then nondeterministically select one among the corresponding statements $S^{(1)}, \ldots, S^{(k)}$ and execute it. If every variable is bound to `false`, then raise a `missingClause` exception. If all variables are unbound, then the statement blocks until at least one variable is bound to either true or false.

   Use the message-passing model in Oz to program this.

   [10 points]

4. Write a lazy version of the Append function, which takes two lists as arguments, and returns their concatenation, in the lazy execution model - that is, using {`ByNeed`} rather than using the `lazy` keyword.

[10 points]

5. (Textbook, Exercise 7 from Chapter 4) Programmed Triggers using higher-order programming.

   Consider the following demand-driven producer-consumer code.

```
% Code from Page 262 of the textbook
% Producer
proc {Gen N Xs}
    case Xs
    of X|Xr then
        X=N
        {Gen N Xs}
    end
end


% Consumer
fun {SumList ?Xs Accumulator Limit}
    if Limit > 0
    then
        Xs=X|Xr
    in
        {SumList Xr Accumulator+X Limit}
    else
        Accumulator
    end
end


% Usage
local
    Xs S
in
    thread {Gen 0 Xs} end                    % Producer thread
    thread S={SumList Xs 0 14000} end        % Consumer thread
    {Browse S}
end
```

   In the above code, the consumer demands a new item from the producer when it needs it, by giving unbound dataflow variables, which are bound by the producer.

Instead of using the stream Xs and unbound variables, rewrite the above code using higher-order programming. Here, the producer returns a 0-argument function `F`, which, when called by the consumer, returns a "pair" `X#F2` where X is the next value produced, and F2 is a function which has identical behaviour to `F`.

[10 points]

6. Using the Thread module for defining control structures. Oz has a generalized `break` operation that can exit any level of nesting of loops. Use a the Thread module (refer to the documentation online) to define the **simple case** of the operation `break`, which exits from the innermost loop. The intended use is illustrated with an example.

```
for(i=0; i<8; j++){
    if(foo(i)!=0){
        break; /* exit from loop by one level */
    }else{
        /* do something */
    }
}
```

The usual `break` operation exits the current loop.

For a start, here's how you could implement a simple for loop.

```
proc {For CurrentValue Limit LoopBodyProcedure}
     if CurrentValue < Limit
     then
         {LoopBodyProcedure}
         {For CurrentValue+1 Limit LoopBodyProcedure}
     end
end
```

This has the same behaviour as the following threaded code.

```
declare ForT
proc {ForT CurrentValue Limit LoopBodyProcedure}
     proc {RunInThread Proc ParentId}
         thread
              {Proc CurrentValue}
              {Thread.resume ParentId}
         end
     end
in
     if CurrentValue < Limit
     then
        {RunInThread
         LoopBodyProcedure
         {Thread.this}}   % Run loop body in a thread, passing the
                          % parent's id

        {Thread.suspend {Thread.this}} % wait for Loop body to terminate
        {ForT CurrentValue+1 Limit LoopBodyProcedure}
     end
end

%====
% Example Usage
%====
{ForT 0 10 proc{$ X} {Browse X} end}
```

Modify the above example to implement a loop that would break when the current value satisfies some breaking condition, specified as a boolean function. You may need to change the number of arguments in all the functions given above.

[10 points]