Exam 2011-04-28
Advanced Functional Programming

Uppsala University
Department of information technology

Sven-Olof Nyström

April 28, 2011

Please read the following instructions carefully:

- The exam is five hours.
- This is a closed book exam. No literature or notes are allowed.
- Use a separate sheet for each question, and write your name on each sheet. Leave a margin. Don't write anything important in the top-left corner.
- Please write clearly. Keep in mind that if I cannot read an answer, I cannot give any points for it.
- If something is unclear, try to make reasonable assumptions and write them down. I plan to come in around 15:00 to answer questions. Make sure that you have read the whole exam then.
- The last pages contain some simple program examples in Erlang, Common Lisp and Haskell.

Maximum points: 100. Normally, 40 points are required for a 3 (passing) grade, 60 for a 4, and 80 for a 5 (very good) grade.

Read the whole exam before you start. If you get stuck on one question, proceed to the next—the questions are not necessarily ordered in level of difficulty.

1. **Erlang: Pick the fastest (15p)**

   Suppose we have implemented three algorithms for computing a mathematical function `f`:

   - `algorithm1:f(..)`
   - `algorithm2:f(..)`
   - `algorithm3:f(..)`

   We don't know which algorithm is the fastest for a certain input. For some inputs the first algorithm may be the fastest, but at other times it may be much slower than the other two.

   (a) Define a module `best` where you use the three modules described above to define your own function `f`. It should create processes that use the three algorithms. When the a result has been returned by one of the algorithms, that result is returned by your function.

   (b) Modify your solution so that the remaining processes are killed when one solution has been found.

2. **Erlang: Binary search tree (20p)**

   Implement a binary search tree where each node in the tree is an Erlang process.

   The following operations should be implemented:

   - `empty()`
     Returns a new empty tree.
   - `insert(Key, Val, Tree)`
     Inserts `Key` with value `Val` into the tree. Since the tree is implemented as a process, and the state of a process may change, there is no need to return a new tree. Instead, the tree is updated.
   - `lookup(Key, Tree)`
     Looks up the key in the tree; returns `{value, Val}` if the key is present, and `none` otherwise.
   - `to_list(Tree)`
     Converts a tree into an ordered list of key-value tuples.

   Test run (the values returned by `insert` do not matter).

```
2> T = tree:new().
<0.42.0>
3> tree:insert(foo, 42, T).
{insert,foo,42}
4> tree:insert(bar, 43, T).
{insert,bar,43}
5> tree:lookup(foo, T).
{value,42}
6> tree:lookup(meta, T).
none
7> tree:insert(meta, 99, T).
{insert,meta,99}
8> tree:lookup(meta, T).
{value,99}
9> tree:to_list(T).
[{bar,43},{foo,42},{meta,99}]
```

Each node in the tree should be implemented as an Erlang process. The process that implement one node should talk to at most three other processes in the tree–its parent and its two children.

Hint: you will probably need two types of processes–one for empty trees and one for trees that have at least one key-value pair.

3. **Logical and in Common Lisp (15p)**

In CL, the logical and operator is written `(and ...)` (it is written `&&` in C and Java).

Example: `(and (< x y) (eq a b) (some-test a x))`

Note that unlike C and Java, CL's and operator may take any number of arguments. Just as in C and Java, if one argument evaluates to false, the rest of the expression is not evaluated.

Your task is to define your own `and` as a macro.

(a) Show how an `and`-expression as the one in the example above can be translated into a conditional Common Lisp expression that uses `if`, `t` and `nil` to control evaluation.

(b) When you define a Lisp function or macro that takes any number of arguments, it will receive its arguments as a list. A macro will receive a list of expressions, of course.

Define a function that takes a list of expressions (as in the example:

`((< x y) (eq a b) (some-test a x))`

and translates it into a conditional expression, as in the previous sub-question.

(c) The definition of your and-macro should start

```
(defmacro my-and (&rest l)
    ...)
```

Finish it!

4. **Lazy Evaluation in Common Lisp (16p)**

Unlike Haskell, Common Lisp does not support lazy evaluation directly. However, infinite sequences can be represented in Common Lisp using functions.

Here is one way to do it:

Represent a potentially infinite sequence with at least one element as a pair of

(a) the first value, and

(b) a function that computes the next value.

The empty sequence is represented by `nil`.

Here are some examples of functions that manipulate infinite sequences.

```
(defun take (n s)
  (cond
    ((or (null s)
         (= 0 n))
     nil)
    (t
     (cons (car s)
           (take (- n 1) (funcall (cdr s)))))))
```

This function takes an integer and a sequence and returns the first `n` elements as an ordinary list.

```
(defun constant (c)
  (cons c (lambda () (constant c))))
```

This function computes an infinite sequence where each element is equal to `c`.

```
(defun smap (f s)
  (cond
    ((null s)
     nil)
    (t
     (cons (funcall f (car s))
           (lambda ()(imap f (funcall (cdr s))))))))
```

The map function, but on infinite sequences.

Example:

```
CL-USER> (take 10 (smap (lambda (x) (* x x)) (intsfrom 0)))
(0 1 4 9 16 25 36 49 64 81)
```

Your task is to implement the following functions:

(a) `insert`

(b) `drop`

(c) `to-sequence`

`(insert x s)`

Takes some value `x` and a sequence `s` and returns a new sequence where the first element is `x` and the rest are the elements of `s`.

Example:

```
CL-USER> (take 10 (insert 42 (intsfrom 0)))
(42 0 1 2 3 4 5 6 7 8)
```

`(drop n s)`

Takes an integer `n` and a sequence `s` and returns a new sequence where the first `n` elements have been removed.

Finally,

`(to-sequence l)`

takes an ordinary list and converts it to the sequence representation.

If the list has at most `n` elements,

`(take n (to-sequence l))`

should return a copy of the list.

5. **All pairs (18p)**

Define, in Haskell, a function that takes two infinite lists $[a_0, a_1, \ldots]$ and $[b_0, b_1, \ldots]$ and computes the list of all pairs (a_i, b_j).

You may assume that both input lists are infinite.

Example:

```
*Main> take 20 (allPairs [1..] [100..])
[(1,100),(2,100),(1,101),(3,100),(1,102),(2,101),(1,103),(4,100),(1,104),
(2,102),(1,105),(3,101),(1,106),(2,103),(1,107),(5,100),(1,108),(2,104),
(1,109),(3,102)]
```

Any possible combination (a_i, b_j), for $i, j \geq 0$ should appear on the list, and each combination should appear exactly once (if the input contains repeated elements, it is acceptable if the output does too).

Also, your solution should be reasonably efficient. Solutions with quadratic complexity should be avoided.

Make sure to state the type of any function you define.

6. **All expressions (16p)**

Suppose we have defined the following Haskell data type for expressions:

```
data Expr = Zero
          | One
          | Plus Expr Expr
```

(a) Define a function that computes the list of all expressions. You may use the function `allPairs` from the previous question, even if you did not answer that question. Example:

```
*Main> take 5 allExp
[Zero,One,Plus Zero Zero,Plus One Zero,Plus Zero One]
```

(b) Define a function that takes an expression and computes its value.

(c) Define a function that takes a non-negative integer and returns an expression that evaluates to the integer, using the functions you defined in the previous two subquestions.

Make sure to state the type of any function you define.

Good Luck!

Sven-Olof

```
-module(counter).

-export([start/0, loop/1, increment/1, value/1, stop/1]).

start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(), value},
    receive
{Counter, Value} ->
    Value
    end.

stop(Counter) ->
    Counter ! stop.

loop(Val) ->
    receive
increment ->
    loop(Val+ 1);
{From, value} ->
    From ! {self(), Val},
    loop(Val);
stop ->
    true;
_ ->
    loop(Val)
    end.
```

```
;; Factorial written in a rather clumsy imperative style.

(defun fac1 (n)
  (let ((p 1))
    (block my-loop
      (loop
         (when (= n 0)
           (return-from my-loop p))
         (setf p (* p n))
         (setf n (- n 1))))))

;; A slightly less clumsy imperative factorial. (dotimes (i n) ...)
;; iterates over the values 0 .. n-1.

(defun fac2 (n)
  (let ((p 1))
    (dotimes (i n)
      (setf p (* p (+ i 1))))
    p))
```

```
;; Append two lists

(defun appendx (x y)
  (cond
    ((null x) y)
    (t (cons (car x) (appendx (cdr x) y)))))

;; Efficient list reversal using a help function.

(defun myrev (l)
  (labels
      ((rev-help (l acc)
         (cond
           ((null l) acc)
           (t (rev-help (cdr l) (cons (car l) acc))))))
    (rev-help l nil)))

;; A simple macro.

(defmacro my-if (condition then else)
  `(cond (,condition ,then)
         (t ,else)))
```

```
fak :: Integer -> Integer
fak 0 = 1
fak n = n * fak (n-1)

quicksort :: Ord a => [a] -> [a]
quicksort []     =  []
quicksort (x:xs) =  quicksort [y | y <- xs, y<x ]
                 ++ [x]
                 ++ quicksort [y | y <- xs, y>=x]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  xs     ys    = []

fib :: [Integer]
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

```
class Weight a where
    weight :: a -> Integer

instance Weight Integer where
    weight x = x

instance Weight Int where
    weight x = toInteger(x)

instance Weight [a] where
    weight l = toInteger(length l)
```