INGI1131 Practical Exercises
Lab 2: Procedures, Functions and Tail Recursion

By the end of this lab session, you should be able to understand and write sequential declarative programs, and to write efficient recursive procedures using tail-recursion. If you have some spare time, you can continue with the exercises of Lab02extra.pdf

1. **Functions are procedures**.

   a Rewrite the following piece of code just using procedures. Every declaration and application of functions must be turned into declaration and application of procedures. The following function returns the maximum number in a list of integers.

   ```
   fun {Max L}
      fun {MaxLoop L M}
         case L of nil then M
         [] H|T then
            if M > H then {MaxLoop T M}
            else {MaxLoop T H} end
         end
      end
   in
      if L == nil then error
      else {MaxLoop L.2 L.1} end
   end
   ```

   b Write a function that receives an integer `N` and returns a list of increasing factorials, starting from 1! until $N!$. Thus, the invocation of `{Fact 4}` will generate `[1 2 6 24]`, which corresponds to the list of factorials `[1! 2! 3! 4!]`.

2. **Tail recursion**

   a Consider the following implementation of Sum.

   ```
   fun {Sum N}
      if N == 0 then 0
      else N+{Sum N-1}
      end
   end
   ```

- Is it a tail recursive function? Why?
- If not (at this point you realised that it is not tail recursive), then, rewrite the function to make it tail recursive.

b Consider the following implementation of Append.

```
fun {Append L1 L2}
   case L1 of nil then L2
   [] H|T then H|{Append T L2}
   end
end
```

- Is it a tail recursive function? Why?
- If not, then, rewrite the function to make it tail recursive.

c Implement function `Fact` from Exercise 1b as tail recursive.

3. **Arithmetic operations and data structures**

a What is going to happen when the following code gets executed? Is there a relationship with the implementation of `Sum` in the exercise we saw in the previous exercise?

```
local
   X Y
in
   {Browse 'hello nurse'}
   X = 2 + Y
   {Browse X}
   Y = 40
end
```

b Similar analysis here. What is going to happen when the following code is executed? What is the relationship with the implementation of `Append` *(Hint: think in terms of blocking and tail recursion)*

```
local
   X Y
in
   {Browse 'hello nurse'}
   X = sum(2 Y)
   {Browse X}
   Y = 40
end
```

4. **Recursive functions on lists and tress**

a Write a procedure {ForAllTail Xs P} that applies the procedure P to each non-nil tail of Xs, i.e., applying {ForAllTail [X1 ... Xn] P} reduces to the sequence of statements

```
{P [X1 X2 ... Xn]}
{P [X2 ... Xn]}
...
{P [Xn]}
```

Example of use: the following piece of code browses all the possible pairs that can be formed from the elements in [a b c d]. It uses the extra procedure Pairs.

```
proc {Pairs L E}
   case L of nil then skip
   [] H|T then
      {Browse pair(H E)}
      {Pairs T E}
   end
end
{ForAllTail [a b c d] proc {$ Tail}
                         {Pairs Tail.2 Tail.1}
                      end}
```

b The Ministry of Plenty asked us to go green in our lab, so we will work with trees. Assume that binary trees are represented with records of the form: tree(info:Info left:Left right:Right) and that the empty tree is represented by the atom nil. Under this representation, consider the following tree

```
Tree = tree(info:10
            left:tree(info:7
                      left:nil
                      right:tree(info:9
                                 left:nil
                                 right:nil))
            right:tree(info:18
                       left:tree(info:14
                                 left:nil
                                 right:nil)
                       right:nil))
```

Write a function {`GetElementsInOrder Tree`}, that returns a list of the elements of `Tree` in ascending order. To do this, assume that `Tree` is an ordered tree (i.e. the parent is greater than or equal to elements in the left child, and less than elements in the right child)

Example: {`GetElementsInOrder Tree`} `-> [7 9 10 14 18]`

5. The Fibonacci function is defined by the rules

```
fib(n)=1                     if n<2
fib(n)=fib(n-1)+fib(n-2)     if n>=2
```

a Implement that function in the normal way (without using accumulators). Try `Fib 35`

b Implement Fibonacci function with accumulators. Then, try runnning {`Fib 35`} and {`Fib 100`}.

6. Write a function {`Add B1 B2`} that returns the result of adding the binary numbers `B1` and `B2`. A binary number is represented as list of binary digits. The most significant digit is the first element of the list. For instance:

{`Add [1 1 0 1 1 0] [0 1 0 1 1 1]`} `-> [1 0 0 1 1 0 1]`

Note 1: You can assume that the two given lists have the same length.

Note 2: You should start by implementing an adder of two digits:

{`AddDigits D1 D2 CI`} That takes two binary digits and a carry as input, and returns a record containing the addition and the output carry

Example:

{AddDigits 1 0 0} −> output(sum:1 carry:0)
{AddDigits 1 1 0} −> output(sum:0 carry:1)
{AddDigits 1 0 1} −> output(sum:0 carry:1)
{AddDigits 1 1 1} −> output(sum:1 carry:1)

Now use `AddDigits` in your definition of `Add`

High-order functions

7. Write a function {`Filter L F`} that returns a list with all the elements from `L` that pass the filterring function `F`. Where `F` is a function that receives one argument and returns a boolean.

8. Using function `Filter`, write a function that takes a list of integers and returns only the even numbers.

9. Read again question 4a `ForAllTrail` - is it also about higher-order programming?

Extra exercises for curious minds

You have two options now: You can continue with these series of exercises, or, you can first do the Lab02extra.pdf, and then come back to finish this lab.

10. Write a function {`Flatten List`} that returns the result of flattening out List.

    Example: {`Flatten [a [b [c d]] e [[[f]]]]`} `-> [a b c d e f]`

11. The following function returns a list of factorials from `N` to 1. Rewrite it as a procedure.

```
fun {Fact N}
   if N==1 then [N]
   else
      Out={Fact N-1}
   in
      N*Out.1|Out
   end
end
```

    Example of execution: {`Fact 4`} `-> [24 6 2 1]`

12. Let us implement dictionaries with binary ordered trees. So, an empty dictionary would be the atom `leaf`. A non empty dictionary would look like: `dict(key:Key info:Info left:Left right:Right)`

    Write a function {`DictionaryFilter D F`} that, given a dictionary `D` and a Boolean function `F`, returns the list of tuples Key#Info such that {`F Info`} is true. (See example in the next page)

Example: suppose that

```
Class=dict(key:10
           info:person('Christian' 19)
           left:dict(key:7
                     info:person('Denys' 25)
                     left:leaf
                     right:dict(key:9
                                info:person('David' 7)
                                left:leaf
                                right:leaf))
           right:dict(key:18
                      info:person('Rose' 12)
                      left:dict(key:14
                                info:person('Ann' 27)
                                left:leaf
                                right:leaf)
                      right:leaf))
```

And

```
fun {Old Info}
   Info.2 > 20
end
```

Then

```
{DictionaryFilter Class Old} -->
                         [7#person('Denys' 25)
                          14#person('Ann' 27)]
```

13. A grammar for a small language can be defined as follows:

```
<pattern> ::= '[' <list of pattern> ']' | <element>
<list of patterns>::= <pattern> <list of patterns> | <pattern>
<element>::= a | b | c | d | x
```

Patterns in this language can be seen as a template defining either the atoms `a,b,c,d` or a list containing these atoms or sub-lists with the same construction. The special atom `x` doesn't specify itself, it is a wild-card matching any pattern.

Note that the list of valid patterns is defined by the grammar. For instance, you can infer that `[a b e]` is not a pattern since `e` is not a valid `<element>`)

Example 1: The pattern `[a x a]` matches all three-element lists starting with a and ending with a.

Example 2: The pattern `[a [c x] a]` matches all three-elements lists starting with `a` and ending with `a`, having as its middle element a sub-list of two elements, the first of which is `c`.

Write a function, {`Matcher Pattern Value`}, that takes two arguments: a pattern and a value and returns true or false depending on whether the value corresponds to the pattern.

Try Matcher on the following cases and confirm you get the correct result:

{Matcher [a x a] [a b a]} −> **true**
{Matcher [a x a] [a b d]} −> **false**
{Matcher [a x a] [a [a b] a]} −> **true**
{Matcher [a [c x] a] [a [b c] a]} −> **false**
{Matcher [a [c x] a] [a [c b] a]} −> **true**
{Matcher x [a [c b] e]} −> **false**
{Matcher x [a [c b] x]} −> **false**