TDT4165 PROGRAMMING LANGUAGES

Assignment 1
Introduction to Oz

Fall 2019

This assignment will introduce you to Oz (the language), Emacs (the beloved extensible lisp-based environment) and the Mozart Oz programming system. You need to be able to use these tools to do the rest of the assignments.
Fetch a binary installation from **https://github.com/mozart/mozart2/releases** or simpler, run the environment remotely on NTNU Program farm (search).

Task 1: Hello World!

Open a terminal and type 'oz'. Hit enter. The Emacs environment should now start. If you are unfamiliar with Emacs, or Aqua Emacs (in MacOS). Look for the tutorial in the environment! Emacs has a steep learning curve, but will become an efficient and powerful instrument in your deft hands!
When you start Mozart, the top buffer will be where you write the code and the bottom buffer is the output from the compiler. Try writing the following code:

```
{Show 'Hello World'}
```

Choose Oz from the menu above the buffers. If you are using Emacs inside a terminal you can press F10 to activate the menu line, followed by shift+o. Here you you will see the different commands you can give to Oz. Choose Feed Buffer to feed in your code. You will notice that not much is happening, so where did the result go? All output from Show comes in another buffer called Oz Emulator. To change back to this you can either choose it from Buffers at the menu line or go back to the Oz menu and choose Show/Hide followed by Emulator.
Another way to output data is by using the command Browse. Change your code to: `{Browse 'Hello World!'}`. If you now again do Feed Buffer, nothing will be printed in the emulator. Instead a window called Oz Browser will pop up and print the text. As such we'll mostly use `Show` and `System.showInfo` for a while. There are important differences in the meaning of `Browse` and `Show`. More on that later.

Task 2: Compiling Oz

Compiling oz allows you to use your favorite editor instead of Emacs. Compiling the oz code creates an executable that you can run using the oz interpreter. We use ozc (short for oz-compiler) to compile our code: Write the following to `main.oz`:

```
functor
import
    Application(exit:Exit)
    System
define
    {System.showInfo 'Hello Compiler!'}
    {Exit 0}
end
```

Now compile and run your code using:

```
ozc -c main.oz
ozengine main.ozf
```

The program will exit automatically due to the {`Exit 0`} statement. If you forget this statement, your program will not exit. If your program hangs, exit using the control+c key combination.

Task 3: Variables

A variable is a name for a binding in your program. You can declare variables in oz by using the following syntax:

```
functor
import
    System
define
    X = 9000
end
```

Functor import, and define-end will not be included in the following code snippets, but is still required for your code to run. The above snippet binds the name X to the atom 9000. You can declare multiple variables.

a) Rewrite the following code so that instead of calculating X directly it creates two other variables, Y and Z, assigns the values to them, and calculates X indirectly from these.

```
X = 300*30
```

b) Run the code

```
X = "This is a string"
thread {System.showInfo Y} end
Y = X
```

Why do you think showInfo can print Y before it is assigned? Why is this behaviour useful? What does the statement $Y = X$ do?
Note that variables must start with an uppercase letter.

Task 4: Functions and procedures

fun is a reserved word that is used to make functions. These resemble functions in Python or MATLAB in that they take a number of arguments and return a value. Their use is exemplified in the following code, which returns the smallest of two arguments.

```
fun {Min X Y}
    if X < Y then
        X
    else
        Y
    end
end
```

Oz does not use a special **return** statement like Python or MATLAB. The value of the last expression is returned instead.
To call a function, use the following syntax:

```
{Name-of-the-function Argument1 Argument2 ... ArgumentN}
```

If we for example want to observe which value is smaller, we can use:

```
{System.showInfo {Min 10 89}}
```

proc is used to make a procedure. These are similar to functions, except that they have no return value.
a) Write a function {Max Number1 Number2} that returns the maximum of Number1 and Number2.
b) Write a procedure {PrintGreater Number1 Number2} that prints the maximum value of the arguments.

Task 5: Variables II

If we want to have variables available in a limited scope, we can use the following:

```
local
    Variable1 Variable2 ... VariableN
in
    % Code here
end
```

Write a procedure {Circle R} that calculates area, diameter, and circumference of a circle with radius r, stores the three results in three variables, and then prints the results. Use the expressions $A = \pi * R^2$, $D = 2R$, and $C = \pi * D$. (Hint: You may want to bind Pi to $\frac{355}{113}$)

Task 6: Recursion

Recursion is a method for calculating an answer by having a function calling directly or indirectly calling itself until the answer is reached. This is typically a direct parallel to mathematical induction. Recursion is one of the most important concepts in declarative programming, and you will use it a lot in later assignments. For example, the following code will increment a number until a satisfactory value is reached:

```
fun {IncUntil Start Stop} A in
    {System.showInfo "Pushing Start: "#Start}
    if Start < Stop then
        A = {IncUntil Start+1 Stop}
    else
        A = Stop
    end
    {System.showInfo "Popping Start: "#Start}
    A
end
{System.showInfo {IncUntil 10 15}}
```

Notice how - as we call the function from within itself - a stack of previous variables is stored.

a) Write a function {Factorial N} that computes the factorial of any natural number (0-inclusive) using recursion.

Task 7: Lists

Lists are one of the most important data structures in Oz. They are used to represent sequences of elements, and are often used with recursion to incrementally build an answer. Lists can be represented in many ways. The following lists are all equivalent:

```
    List = [1 2 3]
```

```
List = 1|2|3|nil

List = '|'(1 '|'(2 '|'(3 nil)))
```

Note that a complete list always has nil as its last element. To retrieve data from a list, we can use the dot notation. The problem with this is that we can only refer to the head and tail of the list, meaning that we can only retrieve the first element or the rest of the list. In the lists above, `List.1` will return 1 while `List.2` will return `[2 3]`. So if we want the third element we must write `List.2.2.1`, which is somewhat cumbersome. A better solution might be to use pattern matching with the case statements. To retrieve the head and the tail of a list we will then write:

```
case List of Head|Tail then
% code that uses Head and/or Tail
else
% otherwise
end
```

It is also possible to do more advanced pattern matching, for example like this:

```
case List of Element1|Element2|Element3|Rest then
% code that uses the elements
[] Head|Tail then
% If the previous pattern did not match
else
% If no pattern matched
end
```

a) Implement {`Length List`}, it returns the element count of List.
b) Implement {`Take List Count`}, it returns a list of the first *Count* elements. If *Count* is bigger than the amount of elements in the list, it instead returns the entire *List*.
c) Implement {`Drop List Count`}, it returns a list without the first *Count* values. If *Count* is greater than the length of the list, return *nil*.
d) Implement {`Append List1 List2`}, it returns a list of all elements in List1 followed by all elements in List2.
e) Implement {`Member List Element`}, it returns true if *Element* is present in *List*, false otherwise.
f) Implement {`Position List Element`}, it returns the position of *Element* in *List*, assume the element is present in the list.
You can store all your functions inside a different file. Create a file `List.oz` and copy the functions you created into it. You can now use your little library from another file:

```
functor
import
    System
define
    \insert List.oz
    {System.showInfo {Length [1 2 3]}}
end
```

The \insert statement pastes the code from the file List.oz directly into your code.
Want more? Look to the **http://strasheela.sourceforge.net/strasheela/doc/Basics-1.html** tutorial!