TDT4165 — Programming Languages

Assignment 1: Introduction to Oz

Fall 2022

This exercise will introduce you to Emacs and Mozart. You need to be able to use these tools to do the rest of the exercises. Mozart2 is available through the NTNU Software Farm. The easiest way to access it is through the web browser; once inside select the "Developer" section and then "Oz-Mozart". We however recommend to configure proper access through remote desktop, as described here.

It is also possible to install the Mozart software manually from the Mozart website. You can find further information on how to install Mozart2 on Blackboard, in the "Tools configuration" page.

Task 1: Hello World!

When you start Mozart, the will be two text areas, 1) The top buffer where you write the code, and 2) The output from the compiler. Try writing the following code:

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

Choose "Oz" from the menu above the buffers. 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: where did the result go? All output from `Show` is displayed in another buffer called "Oz Emulator". The easiest way to show that buffer is to click on the name of the buffer currently being displayed, normally *Oz Compiler*. You can also choose it from the "Buffers" menu 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.

The documentation for using Oz in Emacs (also called the OPI, Oz Programming Interface) can be found here, and provides some information about shortcuts and how this environment works.

Task 2: Using other text editors

You might want to use a different text editor than Emacs to write your Oz code. You may simply open the file in the OPI selecting the "File" menu and then "Open File…". Note that when using the Software Farm you have access to your home directory at NTNU, and you can find it under "This PC" and then "Hjemmeområdet".

Alternatively, when you wish to use an external file in Emacs, in the "Oz" buffer, you can write:

```
\insert 'path_to_file'
```

and then feed this to the buffer. The `\insert` command simply adds the contents of the file into the buffer before feeding it to Mozart. You can also combine `\insert` commands and plain Oz code, for example to reuse code you have saved in a different file.

Note that when using the `\insert` command, you should use the absolute path to the file. Otherwise, paths are typically intended as relative to the Mozart executable path.

There is also an Oz extension for VS Code, which works quite well and has some code completion features. However, you still need to have the Mozart platform installed, because execution still requires the Oz emulator. Note that execution through VSCode is not performed as it might be the case with other programming languages, but instead you must right-click on the source file and select "Feed File". You still have the two buffers "Oz Compiler" and "Oz Emulator", which show the output of the compiler and of the runtime environment, respectively.

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:

```
local X in
    X = 9000
end
```

Variables need to be declared with the "`local X in S end`" construct, where `S` is the statement(s) which the variable `X` is being used used. You can also declare multiple variables in the same line as such "`local A B C in S end`" Further, you can also declare and bind variables at the same time, e.g., "`local A=1 B=2 C in S end`".

If you are getting a "binding analysis error" from the compiler, it will often be that a variable is not declared properly. Note that variables must start with an uppercase letter.

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` 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?

Task 4: Functions and procedures

`fun` is a reserved word that is used to define 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 in the function is returned instead.

To call a function, use the following syntax:

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

To call the previously defined `Min` function, we can for example write "`{Min 10 89}`". To observe its result we can use:

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

Similarly, the keyword **proc** is used to define a procedure. These are similar to a "*void*" function in Java. That is, 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

To declare variables in a local scope for procedures and functions, we can use the following:

```
fun {Function X Y} A B in
    %The function definition
end
```

In this function, `X` and `Y` are parameters to the function, while `A` and `B` are local variables within the scope of the function.

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 π to $\frac{355}{113}$)

Task 6: Recursion

Recursion is a method for calculating an answer by having a function calling directly or indirectly 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 exercises.

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 using recursion. Note that $0! = 1$ (i.e., the factorial of zero is one).

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 notations are all equivalent:

```
List = [1 2 3]

List = 1|2|3|nil

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

In Oz, internally, a list is always represented as in the third line: a record having the symbol '`|`" as label, and two components. The first component contains the first element of the list, while the second component recursively contains another list, and so on. 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 two components of the record, i.e., the *head* and *tail* of the list. This means that we can only retrieve either 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 not so convenient.

Another solution is 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
```

Note that pattern matching in Oz is not specific to lists, but can instead be used with any kind of record and also with plain variables.

a) Implement `{Length List}`. It should return the element count of `List`.

b) Implement `{Take List Count}`. It should return a list of the first `Count` elements. If `Count` is bigger than the amount of elements in the list, it should return the entire `List`.

c) Implement `{Drop List Count}`. It should return a list without the first `Count` values. If `Count` is greater than the length of the list, the function should return `nil`.

d) Implement `{Append List1 List2}`. It should return a list of all the elements in `List1` followed by all the elements in `List2`.

e) Implement `{Member List Element}`. It should return `true` if `Element` is present in `List`, `false` otherwise.

f) Implement `{Position List Element}` It should return the position of `Element` in `List`. You can in this case assume that the element is present in the list.

Store all your functions inside the same file. Create a file `List.oz` and copy the functions you created into it. You can now use your little library from another file:

```
\insert 'List.oz'
{System.showInfo {Length [1 2 3]}}
```

Recall that when using the `\insert` command, you should use the absolute path to the file. Otherwise, paths are typically intended as relative to the Mozart executable path.

Task 8: Lists II

Lists are very versatile, and they can be used to represent different kinds of data structures. We can for example use a list to implement a simple stack.

a) Implement `{Push List Element}`. It should return an updated version of `List`, in which `Element` has been added in the first position.

b) Implement `{Peek List}`. It should return the element in the first position of `List`, or `nil` if the stack (list) is empty.

c) Implement `{Pop List}`. It should return an updated version of `List`, in which the first element has been removed.