TDT4165 PROGRAMMING LANGUAGES

Fall 2015

Exercise 01
Introduction to Oz

This exercise will introduce you to Emacs and Mozart. You need to be able to use these these tools to do the rest of the exercises.

Mozart is installed on `selje` and on the computers in Rose. If you want to do this exercise at home you can install Emacs and Mozart on your own computer by following the instructions at

You might want to browse chapter one of the textbook while doing this exercise.

Task 1 Your first Oz-program

Start Mozart by writing `oz` (selje) or by choosing it from the Start menu (Windows). An Emacs-window will appear. Use this window to communicate with Mozart. In Emacs, the open windows are called *buffers*.

When you start Mozart, the top buffer will be where you write the code and the bottom window 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 `eos` 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. There

are several differences between `Show` and `Browse`, as will become clear in the following.

Task 2 Declaring variables I

`declare` is a reserved word in Oz. By using it you can create new internal variables that can later be asign values. Oz uses dynamic typing, which means that you don't have to specify whether the variable will hold an integer, a string og something else. Oz figures it out on its own while running the program. You can declare a variable and assign it a value in one step or separately.

Note that when a variable in Oz has been assigned a value it's no longer possible to assign it another value. For example:

```
declare
X = 'if I have been assigned this value'
X = 'I cannot be assigned this one' % static analysis error
```

It is possible to declare a new X, but this will only hide the old variable, not change its value.

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

```
declare
X = 42 * 23
{Browse X}
```

Run this code:

b)
```
declare
X Y
X = 'this is magic'
{Browse Y}
Y = X
```

How do you think `Browse` can know the value of Y even when it is called before Y is assigned? Why is this behaviour useful?

Task 3 Functions and procedures

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

```
declare
fun {Max X Y}
   if X > Y then
      X
   else
      Y
   end
end
```

Oz doesn't use a special `return` command like Java. The value of the expression on the last line is returned.

To call a function, use the following syntax:

```
{Function-name Arg1 Arg2 .. ArgN}
```

So if we for example want to know which of 4 and 7 is the bigger, we can use:

```
{Browse {Max 4 7}}
```

`proc` is used to make a procedure. These work as functions, except that they don't return a value.

a) Write a function {Min Number1 Number2} that returns the minimum of Number1 and Number2.

b) Write a function {IsBigger Number Threshold} that returns true if the Number parameter is bigger than the Threshold parameter, and false if not.

Task 4 Declaring variables II

If we want only some variables to be visible within specific parts of a program—for example a function—we can use:

```
local
   Variable1 Variable2 .. VariableN
in
   % your code here
end
```

A shorter and equivalent variant utilizing syntactic sugar is:

```
Variable1 Variable2 .. VariableN in
   % your code here
```

The variables declared in `local` statements are local to the scope they were declared in.

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 $C = \pi * D$.

Task 5 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 typcially 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 calculate the factorial of a number:

```
fun {Factorial N}
   if N==0 then
      1
   else
      N * {Factorial N-1}
   end
end
```

a) Write a function {SumTo FirstInteger LastInteger} that calculates the sum of the integers between and including FirstInteger and LastInteger.

Example : The execution of {SumTo 0 2} should result in the SumTo function returning 3. Execution of {SumTo 3 5} should return 12.

You will now make a new function {Max X Y} that like the previous one returns the greatest of the two numbers X and Y. The difference is that the only test you are allowed to perform is to check if a number equals zero or not, in other words `if N==0 then ... else ... end`. In this exercise we assume that X and Y are greater than 0.

You can use the following equations in making the function.

- $max(n, m) = m$, if $n = 0$.
- $max(n, m) = m$, if $m = 0$.
- $max(n, m) = 1 + max(n - 1, m - 1)$, generally.

Task 6 Lists - theory

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 incremen-

tally 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 fom a list, we kan 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 headand the tail of a list we will then write:

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

Task 7 Lists - practice

You will now, in preparation of Exercise 2, write some functions for handling lists. You may not use the built-in functions that perform similar operations. Hint: many of the exercises can be solved by constructing lists through recursion. If you need examples of this you can look in the textbook.

a) {Length Xs} returns the length of the list Xs.

b) {Take Xs N} returns a list of the N first elements of the list Xs. If N is greater than the number of elements in the list, it instead returns the whole list.

c) {Drop Xs N} returns Xs without the first N elements. If N is greater that the number of elements in Xs, it instead returns nil.

d) {Append Xs Ys} returns a list where all the elements from Xs are followed by all the elements from Ys.

e) {Member Xs Y} returns true if Y exists in Xs, and false otherwise.

f) {Position Xs Y} returns the position of the first occurrence of Y in Xs. You may assume that Y exists in Xs. For example, {Position [1 3 5] 3} returns 2.

Task 8 Lexer

A *lexer* (lexical analyzer, tokenizer) is a program which performs lexical analysis. It accepts a list of lexemes and returns a list of tokens.

Implement the function {Tokenize L} which takes a sequence of lexemes and returns a sequence of tokens for a small subset of the declarative sequential kernel language of Oz.

The function should be able to classify the words `local`, `in`, `if`, `then`, `else`, `end`, identifiers, atoms, binary arithmetic operators (`+`, `-`, `*`, `/`), the unification operator `=`, and the comparison operator `==`.

`Tokenize` should accept a list of lexemes as strings and return a list of tokens.

See the Oz documentation for more information about strings:
3.8 Lists
2.1 Strings

For example:

```
{Tokenize ["local" "X" "in" "if" "x" "end"]}
 evaluates to
[key("local") id("X") key("in") key("if") atom("x") key("end")]
```

or

```
{Tokenize ["local" "X" "in" "if" "x" "end"]}
evalutaes to
['local' lex(var:"X") 'in' 'if' lex(atom:"x") 'end']
```

We don't consider (macro-)syntax here. The examples above are good examples, but they are not syntactically correct. Also note that variables begin with an upper case letter, while atoms begin with lower case.

Hint: It can be helpful to use the function List.member (or Member) which tests whether a list contains an object:

```
{Member 1 [1]} % evaluates to true
{Member 2 [1]} % evaluates to false
```

It can also be useful to know that strings in Oz are lists of characters, and that characters are represented by numbers:

```
"abc" == [&a &b &c] ==  [97 98 99]

&a >= &b % false
&a =< &b % true
```

You must specify what happens if an illegal lexeme is found, for example `?` or `&`.