TDT4165 - Programming Languages

Assignment 1: Introduction to Oz

Benjamin Zubača

August 30, 2024

Task 3: Variables

a) The code mentioned in the task description can be written as follows:

```
1 local X Y Z in
2     Y = 300
3     Z = 30
4     X = Y * Z
5 end
```

b) When the thread in the second line starts executing `{System.showInfo Y}`, it encounters `Y`, which is currently unbound. Instead of failing, the thread waits or blocks until `Y` gets bound to a value.

Once the main thread executes `Y = X`, the variable `Y` becomes bound to the value of `X`, which is `"This is a string"`. The waiting thread is then unblocked, allowing `{System.showInfo Y}` to print the value of `Y`.

Task 4: Functions and Procedures

a) The `{Max Number1 Number2}` function is implemented as shown below:

```
fun {Max Number1 Number2}
   if Number1 >= Number2 then Number1
   else Number2 end
end
```

b) The procedure `{PrintGreater Number1 Number2}` below calls the previously defined `Max` function with `Number1` and `Number2` as arguments, and passes the result to the `{System.showInfo}` procedure, displaying its output:

```
proc {PrintGreater Number1 Number2}
   {System.showInfo {Max Number1 Number2}}
end
```

Task 5: Variables II

```
proc {Circle R}
   PI = 355.0 / 113.0
   D = 2.0 * R
   A = PI * R * R
   C = PI * D
in
   {Show D}
   {Show A}
   {Show C}
end
```

With the procedure above, we assume that the parameter `R` is of the `Float` type.

Task 6: Recursion

The listing below presents an implementation of the `{Factorial N}` function. While a simple `if-else` statement could have been used, here we use an explicit pattern matching `case` statement to control the function's execution flow:

```
fun {Factorial N}
  case N of 0 then 1
  else N * {Factorial N-1} end
end
```

An alternative approach to solving this task is by using tail call recursion:

```
fun {Factorial N}
   {TailFactorial N 1}
end

fun {TailFactorial N Acc}
   case N of 0 then Acc
   else {TailFactorial N-1 Acc * N} end
end
```

Here, the function `{Factorial N}` invokes another function `{TailFactorial N Acc}`, which uses an accumulator `Acc` to track the current state throughout the recursive calls until the final result is returned in the base case.

Task 7: Lists

a)

```
fun {Length List}
  case List
  of nil then 0
  [] Head|Tail then 1 + {Length Tail}
  end
end
```

b)

```
local Take in
   fun {Take List Count}
      if Count == 0 then nil
      else if List == nil then nil
           else List.1|{Take List.2 Count - 1} end
      end
   end
end
```

c)

```
local Drop in
   fun {Drop List Count}
      if Count == 0 then List
      else if List == nil then nil
           else {Drop List.2 Count - 1} end
      end
   end
end
```

d)

```
local Member in
   fun {Member List Element}
      if List == nil then false
      else if List.1 == Element then true
           else {Member List.2 Element} end
      end
   end
end
```

e)

```
local Append in
   fun {Append List1 List2}
      if List1 == nil then List2
      else List1.1 | {Append List1.2 List2} end
   end
end
```

f)

```
local Position FindPosition in
   fun {Position List Element}
      {FindPosition List Element 0}
   end

   fun {FindPosition List Element Acc}
      if List == nil then ~1
      else if List.1 == Element then Acc
         else {FindPosition List.2 Element Acc + 1} end
      end
   end
end
```

Task 8: Lists II

a)

```
local Push in
   fun {Push List Element}
      Element|List
   end
end
```

b)

```
local Peek in
   fun {Peek List}
      case List of Head|Tail then Head
      else nil end
   end
end
```

c)

```
local Pop in
   fun {Pop List}
      case List of Head|Tail then Tail
      else nil end
   end
end
```