

Assignment 3

By: Alexander Høyby

October 6, 2021

Task 1

```
declare QuadraticEquation RealSol X1 X2 in

  proc {QuadraticEquation A B C ?RealSol ?X1 ?X2} D Sq in
    D = {Number.pow B 2.} - 4. * A * C
    {Show 'D: ' # D}
    RealSol = D >= 0.
    if RealSol then
      Sq = {Float.sqrt {Number.pow B 2.} - 4. * A * C}
      X1 = (~B - Sq)/ (2. * A)
      X2 = (~B + Sq)/ (2. * A)
    end
  end
end

{QuadraticEquation 42. 73. ~137. RealSol X1 X2}
{Show 'RealSol: ' # RealSol}
{Show 'x1: ' # X1}
{Show 'X2: ' # X2}
```

Q: Why are procedural abstractions useful? Give at least two reasons.

A: Makes code easier to read and understand. Provides the ability to return none, one, or multiple values.

Q: What is the difference between a procedure and a function?

A: A function performs a task and always have one or more return values, whereas a procedure is only performing a task with no explicit return value.

Task 2



```
declare Sum in

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

  {Show {Sum [0 1 2 3 1 7 1]}}
```

Task 3

```

declare RightFold Lenght Sum in

  fun {RightFold List Op U}

    /* Check for- and split list into Head | Tail */
    case List of Head | Tail then

      /* Call on the Op function with Head as the first parameter and the Head of the next recursive call. */
      /* Because of the call stack, this will be performed from right to left */
      {Op Head {RightFold Tail Op U}}

    /* If the case check is nil, then we're at the end of the list. */
    [] nil then

      /* We're returning the 'neutral' U value */
      U

    /* If none of the above something went wrong. */
    else raise "error" end
  end
end

fun {Lenght List}
  {RightFold List fun {$ X Y} 1 + Y end 0}
end

fun {Sum List}
  {RightFold List fun {$ X Y} X + Y end 0}
end

{Show{Sum [0 1 2 3 1 7 1]}}
```


Q: For the Sum and Lenght operations, would left fold (a left-associative fold) and right fold give different results? What about subtractions?

A: Associative law states that the order of grouping the numbers does not matter. This law holds for addition and multiplication but it doesn't hold for subtraction and division. Sum and Lenght does therefore not care if we left or right fold. But subtraction will.

Q: What is a good value for U when using RightFold to implement the product of list elements?

A: 1 would be a good value as it would not alter the result from list and also not make the whole result 0.

Task 4




```
declare Quadratic in

  fun {Quadratic A B C}
    fun {Calculate X}
      A*X*X + B*X + C
    end
  end

  in
    Calculate
end

{Show {{Quadratic 3 2 1} 2}}
```

Task 5



```
declare LazyNumberGenerator in


  fun {LazyNumberGenerator StartValue}
    StartValue | fun {$} {LazyNumberGenerator StartValue + 1} end
  end

  {Show{ {{{{{LazyNumberGenerator 0}.2}.2}.2}.2}.2}.1}}
```

Instead of calling our main function recursively, creating an infinite call where StartValue increases towards infinity. We're calling a function in our recursive call instead. This will suspend our current function to get an answer from our called (anonymous) function which will provide us with just the number of answer we want on demand and returns that value. Ending the infinite call.

A limitation here is that this method only simulates an infinite list. We're not actually returning a complete list, but a list that requires a function call. Hence, we cannot do actions such as `{LazyNumberGenerator 0}.2.2.1.` as we would with a regular list.

Task 6



```
declare Sum in

  fun {Sum List}
    fun {Iterate List Sum}
      case List of Head|Tail then
        {Iterate Tail Head + Sum}
      else
        Sum
      end
    end
  end

  in
    {Iterate List 0}
  end

{Show {Sum [0 1 2 3 1 7 1]}}
```

Q: Is your Sum function from Task 2 tail recursive? If yes, explain why. If not, implement a tail recursive version and explain how your changes made it so.

A: The tail recursive version moves from the back of the tail to the front of the list cumulating the sum along the way.

Q: What is the benefit of tail recursion in Oz?

A: There is no need to retain a stack frame, resulting in less burden on the system, especially for deep stacks.

Q: Do all programming languages that allow recursion benefit from tail recursion? Why/why not?

A: A tail recursive function call allows the compiler to perform a special optimization which it normally can not

with regular recursion. This would therefore be specific to the compiler, and whether or not the compiler has this optimization implemented.