TDT4165 Programming Languages Assignment 3

Ola Lømo Ellingsen

October 3, 2024

Task 1:

a) The code is shown in Figure 1.

b)

A = 2, B = 1 and C = -1: X1 = 1, X2 = -3
A = 2, B = 1 and C = 2: false

c)

Procedural Abstractions are useful because they allow program statements to be "packed inside" a procedure, which can be called somewhere else in the program. The statement will not be executed right away, instead a procedure value is created. The procedure can have arguments which allows us to alter their behaviour when called later in the program, and we can therefore reuse blocks of code to make the program easier to read and modify. Procedures can help *encapsulate* different complex operations and allow the programmer to organize the code into smaller well defined sections, thus it underlies high-order programming and object orientation.

d)

Functions and procedures have different behavior and flexibility. A function always returns exactly one output, and is often used to calculate mathematical functions or computations where a single result is expected. On the other hand, a procedure is more versatile beceaus it can have any number of inputs and outputs, and it is not constrained to return a value. This flexibility makes procedures better suited for defining abstractions like objects or components, which may have different behaviour than mathematical functions. Procedures are therefore ideal for managing operations or taking actions that not necessarily results in a single output.

```
% Task 1)
% a)
declare proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}
    Discr = B*B - (4.0 * A * C)
in
    if (Discr >= 0.0) then
        RealSol = true
        X1 = (~B + {Sqrt Discr})/(2.0 * A) % using '~' instead of '-'
        X2 = (~B - {Sqrt Discr})/(2.0 * A)
    else
        RealSol = false
        X1 = nil
        X2 = nil
    end
end

% procedure to print the results
declare proc {QuadraticEquation_print A B C}
    local RealSol X1 X2 in
        {QuadraticEquation A B C RealSol X1 X2}
        if RealSol then
            {System.showInfo "X1 = " #X1}
            {System.showInfo "X2 = " #X2}
        else
            {System.show RealSol}
        end
    end
end

```

Figure 1: Screenshot of the code in Task 1a)

Task 2:

The code is shown in Figure 2.

```
36  % Task 2)
37
38  declare fun {Sum List}
39      case List of nil then
40          0
41      [] Head|Tail then
42          Head + {Sum Tail} % calling the function recursively
43      end
44  end
45
46  {System.show {Sum [4 3 2 1]}} % 10
47  {System.show {Sum [2 3 2 5]}} % 12
```

Figure 2: Screenshot of the code in Task 2

Task 3:

a) See Figure 3.

b)

We define the function RightFold which takes in *List, Op* and *U* as arguments. Op is the function that is being applied to the elements in the List. In Op, the first argument is the Head and the second arguments is the result of the recursive call on the List's Tail, this is how we recursively move from left to right in the list. If the list is empty (else statement), the "base case" U is returned and the recursion is done.

c) See Figure 3.

d)

LeftFold and **RightFold** will produce the same result for operations like Sum and Length, because the order of applying the operation does not matter (2+3 == 3+2). However, for operations like subtraction or division, the results will differ because LeftFold applies the operation from left to right, while RightFold applies it from right to left (2-3 != 3-2).

e)

An appropriate value for the base case U in RightFold (or LeftFold), is 1. This is because multiplying any value with 1 will return the same value, thus when the recursion is done we just return the current value * 1.

```
51  % Task 3)
52  % a)
53  declare fun {RightFold List Op U}
54      case List of Head|Tail then
55          {Op Head {RightFold Tail Op U}}
56      else
57          U
58      end
59  end
60
61
62  % c)
63  declare fun {Length_ List}
64      {RightFold List fun {$ X _} X+1 end 0}
65  end
66
67  declare fun {Sum_ List}
68      {RightFold List fun {$ X Y} X+Y end 0}
69  end
70
```

Figure 3: Screenshot of the code in Task 3

Task 4:

See Figure 4.

```
% Task 4)

% f(x) = Ax^2 + Bx + C

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

{System.show {{Quadratic 3 2 1} 2}} % 17
```

Figure 4: Screenshot of the code in Task 4

Task 5:

a) See Figure 5.

b)

The function **LazyNumberGenerator** takes a start value and creates an infinite list of incrementing integers. It returns a structure where each element is paired with a function to generate the next value on demand, and not building all at once. This approach allows us to potentially represent infinite lists by only compute elements as needed. However, accessing deeper elements will probably be inefficient due to repeated recursive calls, and may take up a lot of memory. Accessing the nth element will require n recursive calls, thus this solution has limitations when we want to access a specific element or random access.

```
% Task 5)

% a)
fun {LazyNumberGenerator StartValue}
    StartValue|fun {$}
        {LazyNumberGenerator StartValue + 1}
    end
end

{System.show {LazyNumberGenerator 0}.1} % 0
{System.show {{LazyNumberGenerator 0}.2}.1} % 1
{System.show {{{{{LazyNumberGenerator 0}.2}.2}.2}.2}.2}.1} % 5
```

Figure 5: Screenshot of the code in Task 5

```
% Task 6)

% a)
declare fun {Sum_tail_rec List}
    fun {Helper_func List Acc}
        case List of nil then
            Acc
        [] Head|Tail then
            {Helper_func Tail Head + Acc}
        end
    end
in
    {Helper_func List 0}
end

{System.show {Sum_tail_rec [4 3 2 1]}} % 10
{System.show {Sum_tail_rec [2 3 2 5]}} % 12
```

Figure 6: Screenshot of the code in Task 6

Task 6:

a)

The function in Task 2 is not tail recursive because the recursive call is not the last operation in the function. Instead, the function adds up the sum by adding Head to the result of Sum Tail, and the plus operation is performed after the recursive call returns. Thus, each recursive call needs to maintain its own stack frame, which leads to potential stack overflow for large lists.

In order for the function to be tail recursive, we need to ensure no operation is being executed after the recursive call. As we traverse thru the list recursively, we store the temporarily sum and use it as a parameter in the next recursive call. When we reach our base case (nil), the current sum is returned as the final result. See the code in Figure 6.

b)

In Oz, tail recursion is beneficial because it enables the compiler to optimize memory allocation. This is done by using the current function's stack frame instead of creating a new stack frame for each recursive call. Thus the tail-recursive functions can run in constant memory and this improves performance and reduces the risk of stack overflow errors.

c)

Not all programming languages that support recursion benefit from tail recursion, this is because tail call optimization is handled by the compiler and not the language itself. For tail recursion to be efficient, the compiler must be able to optimize tail calls. Although the **C programming language** allows recursion, it does not benefit from tail recursion since most C compilers don't perform this optimization.

On the other hand, **Java** does not gain any advantage from tail recursion because the Java Virtual Machine (JVM) always preserves the stack for each recursive call and doesn't optimize tail calls. **Kotlin** on the other hand, which also runs on the JVM, can benefit from tail recursion because the Kotlin compiler transforms tail-recursive functions into iterative loops, making them more efficient.