

# TDT4165 Programming Languages Assignment 3

Martin Hegnum Johannessen

October 1, 2024

## Task: 1 Quadratic equation solver

**Task: 1.1 Implementation of** `proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}`

The procedure solves the quadratic equation  $Ax^2 + Bx + C = 0$  using the quadratic formula:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

It first computes the discriminant  $\Delta = B^2 - 4AC$ . If  $\Delta$  is negative, it sets `RealSol` to `false`, indicating no real solutions. Otherwise, it calculates the two real roots `X1` and `X2`. The implementation is presented in Figure 1.

### Task: 1.2 Test Cases

- For  $A = 2, B = 1, C = -1$ :  
Discriminant:  $\Delta = 9$   
Roots:  $X1 = -1.0, X2 = 0.5$   
`RealSol` is `true`.
- For  $A = 2, B = 1, C = 2$ :  
Discriminant:  $\Delta = -15$   
No real solutions. `RealSol` is `false`.

### Task: 1.3 Why are procedural abstractions useful?

Procedural abstractions allow encapsulation of functionality into reusable code blocks, making programs modular, easier to maintain, and more readable.

### Task: 1.4 Difference between a procedure and a function

- A **procedure** performs actions and may have side effects but doesn't return a value.
- A **function** computes and returns a value as its main goal.

## Task: 2 Sum Function

The `Sum` function recursively calculates the sum of a list by adding the head element to the result of summing the tail. If the list is empty, it returns 0. Figure 2 presents an example call of the function.

```

emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Complete In/Out Signals Help

declare QuadraticEquation in

proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}
  % Check for no real solutions
  if B*B-4.0*A*C<0.0 then
    RealSol = false
  % Bind solutions to X1 and X2
  else
    X1 = (0.0-B-{Sqrt B*B-4.0*A*C})/(2.0*A)
    X2 = (0.0-B+{Sqrt B*B-4.0*A*C})/(2.0*A)
    RealSol = true
  end
end

% Test on equation with solutions
local X1 X2 RealSol in
  {QuadraticEquation 2.0 1.0 ~1.0 RealSol X1 X2}
  {Show RealSol} % Expected: true
  {Show X1} % Expected: -1.0
  {Show X2} % Expected: 0.5
end

% Test on equation with no solutions
local __ RealSol in
  {QuadraticEquation 2.0 1.0 2.0 RealSol __}
  {Show RealSol} % Expected: false
end

1\--- quadratic_equation_solver.oz All L10 (Oz)
true
~1
0.5
false

```

Figure 1: quadratic\_equation\_solver.oz

```

emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Complete In/Out Signals

declare Sum in

fun {Sum List}
  % Add the numbers of the list recursively
  case List of Head|Tail then
    Head + {Sum Tail}
  else
    0
  end
end

% Test for Sum
{Show {Sum [1 2 3 4]}} % Expected: 10

1\*- Oz All L11 (Oz)
10

```

Figure 2: sum\_list.oz

## Task: 3 RightFold, Sum, and Length using RightFold

### Task: 3.1 Implementation of fun {RightFold List Op U}

RightFold recursively applies the operation `Op` to each element of the list, starting with the neutral element `U`. Please see Figure 3 for the code implementation.

### Task: 3.2 Explanation of RightFold Code

- Base case: If the list is empty, return the neutral element `U`.
- Recursive case: Apply the operation `Op` to the head of the list and the result of folding the tail.

### Task: 3.3 Re-implementation of Sum and Length

Using RightFold:

- `Sum2` calculates the sum of a list by folding it with `Op = +` and neutral element `0`.
- `Length2` calculates the length of a list by folding it with `Op = 1+Y` and neutral element `0`.

### Task: 3.4 Difference between LeftFold and RightFold

For commutative operations like addition, LeftFold and RightFold yield the same result. For non-commutative operations like subtraction, they produce different results.

### Task: 3.5 Appropriate value for U in product operation

The neutral element for the product operation is `1`.

## Task: 4 Quadratic Function

The `Quadratic` function returns another function that calculates the value of the quadratic expression  $f(x) = Ax^2 + Bx + C$  for a given  $x$ . The code implementation and given example `{System.show {{Quadratic 3 2 1} 2}}` is displayed in Figure 4.

```

emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Complete In/Out Signals Help

declare RightFold Sum2 Length2 in

fun {RightFold List Op U}
  % Pattern match to get head
  case List of Head|Tail then
    % Return the result of doing OP on Head
    % and the result from the Tail of the list
    { Op Head {RightFold Tail Op U} }
  else
    % Base case
    U
  end
end

% Sum2 using RightFold
fun {Sum2 List}
  {RightFold List fun {$ X Y} X + Y end 0}
end

% Test for Sum2
{Show {Sum2 [1 2 3 4]}} % Expected: 10

% Length2 using RightFold
fun {Length2 List}
  {RightFold List fun {$ _ Y} 1 + Y end 0}
end

% Test for Length2
{Show {Length2 [1 2 3 4]}} % Expected: 4

1\*- Oz All L20 (Oz)
10
4

```

Figure 3: rightfold.oz

```

emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Complete In/Out Signals Help

declare Quadratic in

fun {Quadratic A B C}
  % Return a function that takes a single argument,
  % using the environment (A, B, C) to calculate
  % the quadratic function value
  fun {$ X }
    A * X * X + B * X + C
  end
end

% Test for Quadratic
{Show {{Quadratic 3 2 1} 2}} % Expected: 17

1\*- Oz All L13 (Oz)
17

```

Figure 4: quadratic.oz

## **Task: 5 Lazy Number Generator**

### **Task: 5.1 Implementation of fun {LazyNumberGenerator StartValue}**

`LazyNumberGenerator` returns a pair where the first element is the current number and the second element is a function that produces the next number in the sequence. The code implementation and given examples are presented in Figure 5.

### **Task: 5.2 High-level description and limitations**

This function generates a sequence of numbers lazily, meaning that the next number is computed only when requested. However, without native support for lazy evaluation, deeply nested recursive calls can lead to stack overflow.

## **Task: 6 Tail Recursion**

### **Task: 6.1 Is the Sum function tail recursive?**

No, the `Sum` function is not tail-recursive because the addition happens after the recursive call.

### **Task: 6.2 Benefit of tail recursion**

Tail recursion is optimized by reusing the current function frame, thus improving memory usage and preventing stack overflow.

### **Task: 6.3 Do all languages benefit from tail recursion?**

No, only languages that support tail call optimization benefit from tail recursion. Some languages do not optimize tail calls and still accumulate stack frames.

```
emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Oz Help
[Icons]
declare LazyNumberGenerator in

fun {LazyNumberGenerator N} F in
  % F is a function that takes no arguments
  % and returns the value of this function with N=N+1
  F = fun {$} {LazyNumberGenerator N+1} end
  % Return a tuple of the current "count" and
  % the function F that generates the next count
  laz(1:N 2:F)
end

% Test LazyNumberGenerator
{Show {LazyNumberGenerator 0}.1} % Expected: 0
{Show {{LazyNumberGenerator 0}.2}.1} % Expected: 1
{Show {{{{{{LazyNumberGenerator 0}.2}.2}.2}.2}.2}.1} % Expected: 5

1\*- Oz All L12 (Oz)
0
1
5
```

Figure 5: lazy\_number\_generator.oz