exercise 3

fredrik robertsen

2025-09-28

task 1

my implementation of the quadratic equation

a)

```
proc {QuadraticEquation A B C ?RealSol ?X1 ?X2} Discriminant in
    Discriminant = B*B - 4.0*A*C
    RealSol = Discriminant >= 0.0
    if RealSol then
        X1 = (~B - {Sqrt Discriminant})/(2.0*A)
        X2 = (~B + {Sqrt Discriminant})/(2.0*A)
    end
end
```

b)

this is the oz emulator output of `System.showing` `RealSol`, `X1`, `X2`

```
[true ~1 0.5]
[false _<optimized> _<optimized>]
```

c)

procedural abstractions (`proc` instead of `fun`) enable side-effects and interacting with the outside world. in this example, it is useful for providing multiple assigned return values, and because of oz's unification, not all return parameters need be assigned. as such, we are able to let X1 and X2 remain unassigned to any value, in case there are no real roots.

d)

task c) already covers some of the differences. a `function` will have a set signature and have some different syntactic parsing. in some languages, functions don't allow sequential expressions, but oz is weird, so they are allowed. you may also print values to stdout in a function, unlike languages like haskell.

the distinction is in a way pedantic, but a useful one to make. one entails mathematical functions and all their bells and whistles, while the other corresponds more closely to how a computer works. this furthers oz as a general purpose multi-paradigm programming language.

task 2

a simple recursive sum implementation

```
fun {Sum List}
    case List of Head|Tail then Head + {Sum Tail} else 0 end
end
```

task 3

a) & b)

the recursive structure of the right fold is the same as that of sum and length. they both simply pop an element from the list, then apply the binary operator on that element and the recursively accumulated value.

folds can be thought of as putting the binary operator infix between each element of the list, then evaluating it. they are also called “reduce” (in languages like APL/BQN/Uiua), because they apply an operation to all the elements of an array such that it *reduces* the rank of the array by one, i.e. making a matrix a vector, or a vector a scalar.

in my mind, the right fold implementation is a single line, because i am quite used to thinking in terms of folds.

```
fun {RightFold List Op U} % U is the 'default' value of the reduction
    case List of Head|Tail                    % grab the first element
        then {Op Head {RightFold Tail Op U}}  % recursively apply operator
        else U                                % use a default value when the list
is empty
    end
end
```

c)

the functions can be defined easily with a fold:

```
fun {RFSum List}
    {RightFold List fun {$ X Y} X + Y end 0}
end

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

d)

because addition is both associative and commutative, the elements in the List passed to Sum or Length can be in any order and also summed or counted in any order. thus, using a left-associative fold would yield the same result.

the following code snippet uses exponentiation as an example of a non-commutative and non-associative binary operator that would yield different results for the same list.

here, using a left-associative fold results in 0, because the base is set to 0 (U=0), while the right-associative fold results in 1, since 0 is the last exponent.

```
fun {NonAssociativeOperator X Y}
    {Pow X Y}
end
% are FoldL and FoldR the same under a non-associative operator?
% note: FoldL is already provided
{System.show {FoldL [1 2 3 4] NonAssociativeOperator 0}  % = 0
    == {FoldR [1 2 3 4] NonAssociativeOperator 0}}       % = 1
% -> false
```

e)

another example is the factorial function. this illustrates that the identity element of multiplication is 1, and is thus suited as the U-element for a fold under multiplication. setting U=0 would always yield 0 under a multiplication-reduction.

```
% same as python's range(1, N+1) -- but in reverse order.
% all natural numbers less than or equal to N.
% name is borrowed from APL.
fun {Iota N}
    if N > 0 then N|{Iota N-1} else nil end
end
{System.show {Iota 5}} % -> [5 4 3 2 1]

% multiplies all elements in a list together
fun {Product List}
    {RightFold List fun {$ X Y} X*Y end 1} % note: U=1
end

% factorial is trivially implemented as a fold
fun {Factorial N}
    {Product {Iota N}}
end
{System.show {Factorial 5}} % -> 5! = 120
```

task 4

trivially curried* function

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

*: Quadratic is of arity 3, but could be thought of as arity 4 if you consider the argument of it's returned function as well. all functions of arity greater than 1 can be thought of as functions returning functions composed with each other: currying. named after Haskell Curry.

task 5

a)
this embedded function simulates lazy function evaluation and represents an infinite list.

```
fun {LazyNumberGenerator StartValue}
    StartValue|(fun {$} {LazyNumberGenerator StartValue+1} end)
end
```

b)
the idea is that the lazy number generator should return both a current number, and a function that knows how to get to the next number, based on that current number. embedding these two into a data structure allows for lazy evaluation, i.e. "getting the thing you want just when you need it, and not a moment earlier".

this can be useful to save on memory usage by deferring calculations until you need the results.

i suppose a limitation with my above implementation is that it uses recursion as its driving force, thus it uses quite a lot of stack memory for each new number. this could in theory be mitigated by using tail recursion (as we will see in the next task).

task 6

a)
the `Sum`-implementation in task 2 is not tail-recursive, because it performs computations on the result of the recursive call – i.e. it adds the head value to the recursive result.

we can make it recursive by doing continuation-passing style:

```
fun {TailRecursiveSum List} Auxiliary in
    fun {Auxiliary List Accumulator}
        case List
            of nil then Accumulator
            [] Head|Tail then {Auxiliary Tail Head+Accumulator}
        end
    end
    {Auxiliary List 0}
end
```

essentially, we are just carrying the results of the recursion in the argument of a helper (auxiliary) function. this correctly implements tail-recursion.

b)
tail recursion allows for compiler optimizations to use less memory because we don't need to keep the stack frames for each recursive call, since we don't care about the results of the recursion (since they are cleverly baked into the arguments now). proper tail recursion should yield performance similar to an iterative implementation with loops.

it seems oz has quite a bit of tail recursion optimization baked into it. i found this, which shows how oz also converts simple recursive functions into their tail recursive variants, especially if the last operation is cons `|`.

c)
tail recursion will not be beneficial if it isn't supported by the compiler. in fact, it may be worse off performance-wise if you do tail recursion instead, like for the `Sum` implementation. this can be seen because you have to create another helper function and use it instead, and the helper function needs to keep track of more data in its arguments, thus it necessitates more memory usage. so, without being able to unroll the recursion via tail recursion optimization, it is not worth it.

i mentioned the continuation-passing style, which is a way to write virtually all recursive functions as a tail recursive function, but this is not always practically feasible, due to growing complexity with nested functions and such. thus it might often be better to opt for more readable solutions than performant ones, if performance is not the be-all/end-all.

an interesting problem with tail recursion optimizations is that when the compiler optimizes away the stack frames, the execution of the program becomes obfuscated and harder to debug. guido van rossum decided against tail recursion optimizations in python for this particular reason. but in such a language, recursion isn't the only tool at your disposal (you have loops, list comprehensions, etc), so it makes sense to just bite the bullet and let the memory usage go up. but in oz; if all you have is a hammer, then you better make that hammer fast.