TDT4165 PROGRAMMING LANGUAGES

Assignment 3

Higher-Order Programming

Fall 2023

Preliminaries

This exercise is about higher-order programming and tail recursion. There are four basic principles that underline the techniques of higher-order programming: *procedural abstraction*, *genericity*, *instantiation*, and *embedding*.

Relevant reading: Chapters 1–3 in CTMCP, especially 2.5 and 3.6.

Delivered code must be in the form:

```
declare Function Procedure in

fun {Function}
    % Function implementation
end

...

proc {Procedure}
    % Procedure implementation
end

...
```

Please deliver the code as a single `.oz` file. The delivery should also include a `.pdf` file containing a section for each task. For each task, the PDF should describe the implementation, or include a screenshot of the code, as well as answer any theoretical questions. You can use the template found on BlackBoard, under "Coursework" / "Latex template for PDFs", to generate your PDF file.

Evaluation

This assignment is graded as Approved/Not approved.

The requirements to get this exercise approved are as follows:

- Complete Task 1 (implement *a* and *b*; answer *c* and *d*)
- Complete Task 2
- Complete Task 3 (implement *a* and *c*; answer *b*, *c*, and *d*)

- Attempt Task 4
- Attempt Task 5
- Complete Task 6

Make sure that the implemented tasks are able to be run in the standard Emacs environment (Mozart OPI). If the code does not run for tasks 1, 2 and 3, unless there is a trivial error, or a thorough explanation for why the code does not work is provided, the assignment will be failed. Non-running code for task 4 and 5 is fine, as long as the idea behind the provided code is explained.

Procedural abstraction

The ability to convert any statement into a procedure value.

Task 1

(a) Implement `proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}`. `X1` and `X2` should bind to the real solution(s) to the quadratic equation. `RealSol` binds to `true` if there exists a real solution, `false` otherwise. See the appendix for more information about the quadratic equation. When there are no real solution, you may ignore complex solutions, and just set `RealSol` to `false`.

(b) Use your implementation of `QuadraticEquation` and `System.show` to answer the following questions:

What are the values of `X1`, `X2`, and `RealSol`, when `A = 2`, `B = 1` and `C = -1`?

What are the values of `X1`, `X2`, and `RealSol`, when `A = 2`, `B = 1` and `C = 2`?

(c) Why are procedural abstractions useful?

(d) What is the difference between a procedure and a function?

Genericity

The ability to pass procedure values as argumens to a procedure call.

Task 2

In Assignment 1 you implemented `fun {Length List}`, which goes through the list recursively and returns the number of elements in the list. Implement `fun {Sum List}`, which returns the sum of the values of elements in the list.

Task 3

Your implementations of `Length` and `Sum` are probably very similar. Wouldn't it be interesting to have a more general version of these two functions?

In fact, `Length` and `Sum` both reduce a list to a single value by using some combining operation on its elements. A *fold* is an abstraction that captures this pattern.

Write a function `fun {RightFold List Op U}`, which goes through a list recursively and, through the use of a combining operation `Op`, accumulates and returns a result. `U` is the neutral elements for the operation. The function should be right-associative, meaning it should perform the calculation from right to left. For

example, if the operation `Op` is addition and the list is `[1 2 3 4]`, the `RightFold` function should calculate `(1 + (2 + (3 + 4)))` and return `10`. If the operation is length, the function should calculate `(1 + (1 + (1 + 1)))` and return `4`. Re-implement `Length` and `Sum` using `RightFold` and test your solution using {`Length [1 2 3 4]`} and {`Sum [1 2 3 4]`}.

Tip: The `Op` parameter can be provided as an anonymous function, e.g.:

```
fun {$ X Y} <functionbody> end
```

(a) Implement `fun {RightFold List Op U}`

(b) Explain each line of code in `RightFold` in your own words.

(c) Implement `fun {Length List}` and {`Sum List`} using `RightFold`.

(d) For the `Sum` and `Lenght` operations, would `LeftFold` (a left-associative fold) and `RightFold` give different results? Can you provide an example of an operation for which the two folds do not produce the same result?

(e) What is an appropriate value for `U` when using `RightFold` to implement the product of list elements?

Instantiation

The ability to return procedure values as results from a procedure call.

Task 4

We have already seen that functions can be used as input to other functions. Similarly, functions can also return other functions. Implement `fun {Quadratic A B C}` that returns a function which can be used to calculate the value of a quadratic polynomial: $f(x) = Ax^2 + Bx + C$

{`System.show {{Quadratic 3 2 1} 2}`} should display `17`.

Embedding

The ability to put procedure values in data structures.

Task 5

Functions can be embedded in data structures. One application of this technique is to implement lazy (delayed) evaluation. The idea here is to build a data structure on demand, instead of building it all at once. This makes it possible to implement infinite lists.

(a) Implement `fun {LazyNumberGenerator StartValue}` that generates an infinite list of incrementing integers on demand, using higher-order programming. Do not use the built-in `Lazy` keyword neither take advantage of threads or dataflow variables to implement the laziness.

"{`LazyNumberGenerator 0`}`.1`" should return `0`.

"{{`LazyNumberGenerator 0`}`.2`}`.1`" should return `1`.

"{{{{{{`LazyNumberGenerator 0`}`.2`}`.2`}`.2`}`.2`}`.1`" should return `5`.

(b) Give a high-level desciption of your solution and point out any limitations you find relevant.

Tail Recursion

In this assignment you have used recursion to implement most of your functions. Tail recursion is a special case of recusion.

Task 6

(a) Is your `Sum` function from Task 2 tail recursive? If yes, explain why. If not, implement a tail-recursive version and explain which changes you needed to introduce to make it tail recursive.

(b) What is the benefit of tail recursion in Oz?

(c) Do all programming languages that allow recursion benefit from tail recursion? Why/why not?

Appendix

The Quadratic Equation

A quadratic equation is any equation having the form $ax^2 + bx + c = 0$, where x is an unknown number and a, b, and c are known constants.

The *solutions* of such equation are any values of x that satisfy the equation. Their value is given by the result(s) of the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The number of solutions in the domain of real numbers (i.e., such that $x \in \mathbb{R}$) are determined by the value of the so-called discriminant: $\Delta = b^2 - 4ac$.

If $\Delta > 0$ there are two real solutions; if $\Delta = 0$ there is exactly one solution (or two identical solutions if you will); finally, if $\Delta < 0$ there are no real solutions.