TDT4165 PROGRAMMING LANGUAGES

Assignment 4

Declarative Concurrency

Autumn 2025

Preliminaries

This exercise is about streams, threads, lazy programming and declarative concurrency.

Relevant reading: Chapter 4.1–4.5 in CTMCP.

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:

- Answer the questions in Task 1.
- Implement Task 2, Task 3, and Task 4.

Make sure that the implemented tasks are able to be run in the standard Emacs environment (Mozart OPI).

Threads

Task 1

In the sequential model of computation, statements are executed sequentially, in a single computation. The `thread ... end` statement extends the model with concurrency.

(a) Execute the following code in Mozart and observe the results. What sequence of numbers gets printed as output of the Oz environment?

```
local A=10 B=20 C=30 in
    {System.show C}

    thread
        {System.show A}
        {Delay 100}
        {System.show A * 10}
    end
    thread
        {System.show B}
        {Delay 100}
        {System.show B * 10}
    end

    {System.show C * 100}
end
```

(b) Explain with your own words how execution proceeds and why the result is as such. Would it be possible to have a different sequence printed as output? Explain your answer.

(c) Execute the following code in Mozart and observe the results. What sequence of numbers gets printed as output of the Oz environment?

```
local A B C in
    thread
        A = 2
        {System.show A}
    end
    thread
        B = A * 10
        {System.show B}
    end

    C = A + B
    {System.show C}
end
```

(d) Explain with your own words how execution proceeds and why the result is as such. Would it be possible to have a different sequence printed as output? Explain your answer.

Streams

A stream is a list that is created incrementally by leaving the tail as an unbound dataflow variable: it is extended by binding that variable to the next value, and then appending a new unbound tail. Streams are potentially infinite, and have various applications in the processing of sequence of data of unspecified length.

By combining streams and threads it is possible to implement a producer/consumer model in a straightforward way, thanks to declarative concurrency.

Task 2

(a) Implement a function `fun {Enumerate Start End}` that generates, asynchronously, a stream of numbers from `Start` until `End`.

`{Browse {Enumerate 1 5}}` should display `[1 2 3 4 5]`

Hint: You can use the `thread ... end` statement inside the definition of the function, to wrap the iterative process that generates the numbers.

(b) Implement a function `fun {GenerateOdd Start End}` that generates, asynchronously, a stream of odd numbers from `Start` to `End`. The `GenerateOdd` function **must** be implemented as a consumer of `Enumerate`. That is, it must read the stream generated by `Enumerate` and filter it as appropriate.

`{Browse {GenerateOdd 1 5}}` should display `[1 3 5]`

`{Browse {GenerateOdd 4 4}}` should display `nil`

(c) Try to display the output of `Enumerate` and `GenerateOdd` using `Show`, such as:

`{Show {Enumerate 1 5}}`

`{Show {GenerateOdd 1 5}}`

If you have completed the task correctly (that is, if the stream is generated asynchronously), you should get `_<optimized>` as output. How can you explain this behavior?

Task 3

In this task we will implement a generator of prime numbers, exploiting Oz streams and concurrency.

(a) Implement the function `fun {ListDivisorsOf Number}`, which produces a stream of all the divisors of the integer number `Number`. A number $d \in \mathbb{N}$ is a divisor of $n \in \mathbb{N}$ if the rest of the integer division n/d is zero. The *modulo* operation (i.e., rest of integer division) is denoted with the keyword `mod` in Oz. `ListDivisorOf` **must** be implemented as a consumer of `Enumerate`.

(b) Implement the function `fun {ListPrimesUntil N}`, which produces a stream of all the prime numbers up to the number `N`. A number n is *prime* if its only divisors are 1 and n itself. `ListDivisorOf` **must** be implemented as a consumer of `Enumerate`.

Hint: You can chain multiple streams, and also consume multiple streams in the implementation of a function. In particular, you should also consume the stream produced by `ListDivisorsOf`.

Lazy Evaluation

Task 4

The `lazy` keyword can be applied to functions to specify that they will be *evaluated lazily*, meaning that the values would be computed only when needed. This is particularly useful for working with (potentially) infinite streams.

We can rewrite our generator of prime numbers as a lazy function, using the `lazy` annotation.

(a) Implement a function `fun {Enumerate}` as a `lazy` function that generates an infinite stream of numbers, starting from `1`.

(b) Implement a function `fun {Primes}` as a `lazy` function that generates an infinite stream of prime numbers, starting from `2`. You **must** implement `Primes` as a consumer of the stream produced by `Enumerate`, and any other streams you find useful.

Appendix

Visualizing Streams

Using the normal printing procedures (e.g., `Browse` or `System.show`) it might be difficult to visualize the content of streams, especially when they are actually infinite or they require long computation time to be built.

We can solve this problem by again using threads. The following procedures uses threads to visualize a stream as it is being build. The following function uses the `Browse` procedure, but a similar approach can be used with the `System.show` procedure.

```
proc {ShowStream List}
    case List of _|Tail then
        {Browse List.1}
        thread {ShowStream Tail} end
    else
        skip
    end
end
```

Interrupting a computation in Oz

It may happen that you actually start an infinite computation by mistake, or perhaps by purpose when solving Task 4.

To halt a computation in Oz, you should:

- Select "Oz" and then "Halt Oz" if you are using the Emacs-based environment, or
- Press `CTRL+SHIFT+P` and then select "oz.spawn: Halt Oz" if you are using the VS Code environment.

You also have the option to *kill* the `ozemulator` process from the task manager.