assignment 4

fredrik robertsen

2025-10-27

task 1

(a)

the code quite consistently outputs

```
30
3000
10
20
200
100
```

(b)

the result is as such, because 30 gets printed first, since statements are sequential. then 3000 gets printed, which is the last statement. then, all main statements have been executed, so the threads will start executing, starting with the first thread to have been declared. thus it outputs 10. but then while the thread is waiting for 100 ms, the next thread gets to run and 20 is printed. then the last thread finishes first and prints 200, then lastly 100 from the first thread.

it is indeed possible for it to output a different sequence. i managed to have it print out 100 before 200, which is what i would've expected it to print out initially, since the first thread would have been waiting while the last thread got to printing, but for some reason most cases show that the last thread prints before the first.

(c)

the code consistently outputs

```
2
20
22
```

(d)

the code initially comes to `C = A + B` and halts, letting threads execute. the first thread sets `A = 2` and prints this, then the second thread sets `B = A * 10` such that 20 is printed. then lastly we print
22.

this code will always produce the correct values (no data races), but it may rarely print a different order. this is because C waits for A and B, and B waits for A. thus A has to be handled first, then B, then C. however, since the second thread isn't atomic, it may be slow to show, thus the main thread might print first and we'd end up with

```
2
22
20
```

task 2

the following code is my implementation of `{Enumerate Start End}`, which should asynchronously generate a stream of numbers between both endpoints, inclusive.

```
fun {Enumerate Start End} Inner in
  fun {Inner Start End}
    if End == Start then
      Start|nil
    else
      Start|{Inner (Start+1) End}
    end
  end
  thread {Inner Start End} end
end
```

using the previous function, i can simply filter this list on modulo `2` to see if it is divisible by `2` or not, i.e. if it is odd or even.

```
fun {GenerateOdd Start End}
  thread
    {Filter
      {Enumerate Start End}
      fun {$ N}
        N mod 2 \= 0
      end
    }
  end
end
```

displaying these functions using `Show` yields `_<optimized>` as an output, due to my wrapping of the output in a `thread ... end` block.

using `Browse` correctly shows the expected numbers.

my understanding is that if the function returns a thread that performs some computation that doesn't spawn more threads (which is why we use an `Inner` function in `Enumerate`), then it will perform that computation asynchronously whenever that function is called.

thus, if we use `Browse`, it will perform the computation and display the output in the `Browse`-window as soon as it is completed. but `Show` will immediately attempt to print out the value, which doesn't give the cpu any time to perform the computation, thus there are no values to show yet, and it is only shown as the mysterious `_<optimized>`.

task 3

we can similarly to `GenerateOdd` consume `Enumerate` in various ways to obtain different lists of numbers. this is actually one of the main modes of computation in array programming, where programs are often compositions of functions that ultimately act on an array of numbers, such as that of an iota-sequence, which is the same as `{Enumerate 0 N-1}` or `{Enumerate 1 N}`, depending on indexing.

anyway, we can find the divisors of a number by checking for divisibility (remainder/modulus is 0) of each number up to that number. this can be done as follows

```
fun {ListDivisorsOf Number}
  thread
    {Filter
      {Enumerate 1 Number}
      fun {$ N}
        Number mod N == 0
      end
    }
  end
end
```

we can then simply count how many elements are in the list of divisors of a given number to find if it is prime or not. a prime number has only itself and 1 as divisors, so the length of the list must be 2. the following code filters numbers satisfying this.

```
fun {ListPrimesUntil Number}
  thread
    {Filter
      {Enumerate 2 Number}
      fun {$ N}
        {List.length {ListDivisorsOf N}} == 2
      end
    }
  end
end
```

note that we start our enumerate on 2, since we already know 1 isn't prime. we could make further optimizations, such as avoiding the expensive `ListDivisorsOf` call for even numbers greater than 2, essentially halving our computational load. but i digress

task 4

the following code defines a lazy infinite list counting from 1 and up in increments of 1.

i couldn't find many examples of code using the `lazy` keyword, but i found one, which was precisely a counting example. so i could just use that as a base to fix the argument to 1.

```
fun lazy {Enumerate}
  % stolen from
  % https://github.com/alhassy/OzCheatSheet?tab=readme-ov-file#lazy-evaluation
  fun lazy {Ints N} N|{Ints N+1} end
  in {Ints 1}
end
```

the `Ints` function simply builds an infinite recursion on itself, but since it is specified as lazy, oz knows to only compute the next step when it needs to.

we can filter this infinite list to obtain an infinite list of prime numbers. we use the same logic to check for primality, i.e. counting the number of divisors of the given number.

```
fun lazy {Primes}
  {Filter {EnumerateLazy} fun {$ N} {List.length {ListDivisorsOf N}} == 2 end}
end
```

this code produces a list that contains all prime numbers. i.e.

```
{Browse {Primes}.1}  % -> 2
{Browse {Primes}.2.1}  % -> 3
{Browse {Primes}.2.2.1}  % -> 5
{Browse {Primes}.2.2.2.1}  % -> 7
...
```