

Lab 4

In this lab, you will write code for a computation server and a client. On a multi-core machine computations are guaranteed to execute in parallel (i.e. 4 parallel computations on a 4-core machine). Note that there are explicit instructions on what you need to submit (only in parts 4 and 5).

Part 1

The computation server will accept a String containing an integer and will reply with the number of prime numbers between 2 and that integer. You can use the following functions (they consume less heap space than some of the implementations I've seen in your solutions, and they should allow you to work with larger numbers – but don't forget to include the type signatures):

```
countPrimes :: Int -> Int
countPrimes x
  | x < 2    = 0
  | isPrime x = 1 + countPrimes (x-1)
  | otherwise = countPrimes (x-1)
```

```
isPrime :: Int -> Bool
isPrime x = if x<2 then False else isPrime0 x 2
```

```
isPrime0 :: Int -> Int -> Bool
isPrime0 x t
  | t >= x    = True
  | x `mod` t == 0 = False
  | otherwise = isPrime0 x (t+1)
```

To listen on a port given as an argument to the program, you need to do the following:

```
root w = class
  env = new posix w
  result action
    Right port = parse (env.argv!1)
    soc <- env.inet.tcp.listen (Port port) myserver
```

where *myserver* is function of type *Socket -> Class Connection*, i.e. the POSIX run-time system will automatically create a new object of this class whenever it receives a new connection on the specified port. Your task is to define the class *myserver* with the following methods:

```
myserver socket = class
  established = action
    {- here you should install another method of this object as a reader method
       (use socket.inFile.installIR) that will be invoked each time there is new
       data on the socket; to write to the socket, use socket.outFile.write -}
  neterror str = action
    {- invoked automatically when there is an I/O error -}
    env.stdout.write ("Network error: "++str++"\n")
```

```

close = request
  {- invoked automatically when the peer closes the connection -}
  env.stdout.write ("Connection closed by peer\n")
result Connection{..}

```

If you need to pass extra arguments to *myserver*, you can do that provided that socket remains the last argument, for example:

```

root w = class
  env = new posix w
  result action
    Right port = parse (env.argv!1)
    soc <- env.inet.tcp.listen (Port port) (myserver env)

```

```

myserver env socket = class

```

...

Start by writing everything you receive on the socket to *env.stdout*, and then change the code so that the received String is parsed into an Int; if parsing fails, the error string is written back to the socket; if parsing succeeds, then the number of prime numbers between 2 and the received number is calculated, converted into a String (use *show* and append '\n') and written to *socket.outFile*. In any case, the server should write to *env.stdout* the computation request you received (before the computation) and the result (after the computation).

You should also write a separate class that will monitor *env.stdin* and if the user enters 'q', it will close the server socket (returned by *env.inet.tcp.listen*) and exit the program. An object of this class should be created in the root, and its method installed as the reader of *env.stdin*.

Part 2

The client (defined in a separate file from the server) should define a class that will be instantiated by the POSIX run-time system when a connection has been created:

```

root w = class
  env = new posix w
  result action
    Right port = parse (env.argv!1)
    env.inet.tcp.connect (Host "localhost") (Port port) (myclient env)

```

```

myclient env socket = class
  established = action
    {- install readers for socket.inFile and env.stdin -}
  neterror str = action
    env.stdout.write ("Network error: "++str++"\n")
    env.exit 1
  close = request
    env.stdout.write "Connection closed by peer\n"
  send action env.exit 0
  result Connection {..}

```

This class should also define two methods that take a String as an argument and should be installed as readers for *env.stdin* (strip the input String from the final '\n' and write it to *socket.outFile*) and for *socket.inFile* (write everything you receive to *env.stdout*).

Now you should be able to test the server and the client together. Start the server and two clients; on a multi-core machine you should see that requests from two different clients are processed in parallel whereas requests from one client are queued. This is because a new object is created for each connection, and any two objects can execute concurrently; however, only one method of each object can be active at any given time. See example output below:

Output from client 1:

```
Connection established
500000
100
Answer: The number of primes from 2 to 500000 is 41538
Answer: The number of primes from 2 to 100 is 25
Connection closed by peer
```

Output from client 2:

```
Connection established
400000
300
Answer: The number of primes from 2 to 400000 is 33860
Answer: The number of primes from 2 to 300 is 62
Connection closed by peer
```

Output from server:

```
Connection established
Connection established
Request: 500000
Request: 400000
Answer: the number of primes from 2 to 400000 is 33860
Request: 300
Answer: the number of primes from 2 to 300 is 62
Request: 100
Answer: the number of primes from 2 to 500000 is 41538
Answer: the number of primes from 2 to 100 is 25
```

Part 3

You should now change the output from the server to *env.stdout* so that it is easy to keep track of the clients (numbered from 1) and their requests (numbered from 1 separately for each client). See example output from the server below:

```
1# Connection established
2# Connection established
1:1# Request: 500000
2:1# Request: 400000
2:2# Request: 300
2:1# Answer: the number of primes from 2 to 400000 is 33860
```

```
2:2# Answer: the number of primes from 2 to 300 is 62
1:2# Request: 100
1:1# Answer: the number of primes from 2 to 500000 is 41538
1:2# Answer: the number of primes from 2 to 100 is 25
```

Part 4

You should now add to the output from the server to *env.stdout* the time it takes to perform the computation. You can do that by using the method *getAbsTime* in POSIX that reads the world clock and returns a value of type *Time*; you can then use the methods *secOf* and *microsecOf* to get the separate second and microsecond count as integers (a microsecond count is the number of microseconds after the last second).

```
socketReader env str = action
```

```
    time1 <- env.getAbsTime
    ... {- "str" is parsed into an integer named "k" -}
    ans = countPrimes k
    time2 <- env.getAbsTime
    time = time2 - time1
```

An example of output from the server is shown below:

```
1# Connection established
1:1# Request: 200000
1:2# Request: 100
1:1# Answer after 9 sec and 569404 microsec: the number of primes from 2 to 200000 is
17984
1:2# Answer after 0 sec and 33 microsec: the number of primes from 2 to 100 is 25
```

Now you should submit the code for the client and the server

Part 5

Save the server file under another name and modify it so that different computation requests from the same client can be processed concurrently. On a multi-core machine output from the client may then look like this:

```
Connection established
100000
21
Answer: The number of primes from 2 to 21 is 8
Answer: The number of primes from 2 to 100000 is 9592
```

Now you should submit the modified code for the server.