Chapter 3

Practical assignment 3

3.1 Background

The HTTP protocol is used to transfer packets between a web server and a web client (or browser, such as *Chrome*, *FireFox* or *Safari*). Normally these packets contain data encoded in HTML.

The web client marks all the links in a web page by underlining it and/or displaying it in a different colour. When one clicks on such a link, an HTTP (or HTTPS) request is sent to the server concerned — usually to fetch the page identified in the HTML. The format of such an HTTP request to fetch, for example, `/index.htm` from `www.cs.up.ac.za`, is as follows:

```
GET /index.htm HTTP/1.1
Host: www.cs.up.ac.za
```

Each line is followed by a CR character (ASCII 13_{10}).

When the server receives a `GET` request it simply sends the required data to the client as a stream of characters. Normally the stream of data will contain HTML encoded data that will be interpreted by the web client.

Note that the server may also receive requests other than `GET`; technically it is supposed to respond at least to the `GET` and `HEAD` requests. (The latter request is not considered further in this assignment.)

The description of HTTP given above has obviously been oversimplified, but it is sufficient for this exercise. Those who require more information may consult RFC 2616. (It consists of 176 pages in the text version, so do not procrastinate.)

Web pages are constructed by marking content up using HTML. To create a link, one uses the `<a>` tag:
`<a href=...> ... </a>`
The text between the `<a>` and `</a>` tags is displayed as the link text. The portion

that follows `href=` is used to construct the `GET` message that the browser will send to the web server. As noted, it will often be the name of a file that should be fetched. It may also be the name of a CGI program that is to be executed.

However, it is possible to construct a special-purpose web server that interprets this parameter very differently. Let us use an example to illustrate this idea. Suppose one builds a server that accepts 'normal' HTTP requests that look as follows:

```
GET <string> HTTP/1.1
...
...
<blank line>
```

When it receives the request, it may check that it starts with the string `GET`. Then it extracts the `string`; let's call that string s. It may interpret and/or ignore anything that follows, up to the blank line, which demarcates the end of the header (and therefore also the end of the `GET` request).

It now considers the string s. If it happens to be '`/`' (typically meaning the root), it 'prints' the following to the standard output:

```
<some appropriate header line>
<some further appropriate header lines>
<blank line>
<html>
<body>
<p>X</p><br>
<p>
<a href="L">Left</a>
<a href="R">Right</a>
<a href="U">Up</a>
<a href="D">Down</a>
<a href="/">Home</a>
</p>
</body>
```

The receiving web client will display an 'X' in its upper left corner, and the links *Left*, *Right*, *Up*, *Down* and *Home* in the next line.

Say the user clicks on *Down*, then the server will get the request
`GET D HTTP/1.1`
followed by other header lines. Our special-purpose web server interprets this as meaning that the 'X' should move down one line. It therefore returns the same HTML response as it did above, but this time a new line consisting of `<br>` is

inserted before the `<p>X</p><br>` line. In fact, every time the server sees a `GET D ...` it adds another `<br>` line to its output. Hence, every time the user clicks on *Down*, the 'X' will move down one line.

The actions when a user clicks *Up* are the opposite: The server removes one of these `<br>` lines, until none of these `<br>` lines remains, in which case the server just produces the same output it previously did.

Similarly, whenever the user clicks *Right*, another space is inserted before the 'X', and when the user clicks *Left* one space preceding the 'X' (if any) is removed from the output produced by the server.

In essence, this (silly) server just isolates the character s, and produces output that differs from its previous output in the appropriate manner.

The example illustrates that the content of a `GET` request may be interpreted in a non-standard manner to build a special-purpose HTTP server, that can be used via an ordinary browser.

There are some caveats that should be noted from the example above. A server must, at least, support `GET` and `HEAD` requests. How does our server deal with `HEAD`? What should our server do when it receives a `GET` request that is not followed by one of the five characters that have some meaning for it? Should it return a 404 error? (Note that the typical browser will attempt to send a `GET /favicon.ico HTTP/1.1` request in addition to the request it is expected to send. Note that the addition of spaces in front of the 'X' in our example may be interpreted as whitespace by the browser, and the 'X' may not move to the right, as one would hope in this example. And finally, the browser may need to be coaxed to clear its content and replace it with the new content. Despite these issues, the example hopefully serves its intended purpose.

3.2 Your assignment

Write a program that repeats the functionality of assignment 1: It should again display Fibonacci triples. However, rather than implementing the functionality using an existing server and CGI, the software that you write for this assignment should be a dedicated 'Fibonacci server'. Hence it would open a server socket (say 55555) and wait for connections. Once a browser (such as Firefox) connects, your program should interpret the `GET` request that the browser sends. The response that your server sends should not only be the HTML that your scripts sent in assignment 1, but also the appropriate HTTP headers, such that the browser is able to interpret the HTTP message correctly (and render the HTML correctly).

Implement all the commands that an HTTP server *must* implement.

3.3 Assessment

A working program will be awarded 8 out of 10. To earn a higher mark your program has to do more than just the basics - in particular should it demonstrate that you understand something of the HTTP RFC.