

# 15-441: Computer Networks

## Recitation 1

P1 Lead TAs: Mingran Yang, Alex Bainbridge



# Agenda

---

1. Project 1 Checkpoint 1
2. Lex/Yacc
3. Reading the RFC
4. Small group exercise
5. Q&A



# Project 1: Liso Web Server

You will be building a web server that will handle multiple concurrent connections and support a subset of HTTP/1.1!

For 15-641 students, you will also support HTTPS and CGI protocols!



# Project 1: Quick reminder

The Project 1 starter code will be released later today.

CP	Grade	Deadline
1	15% 25%	Sep 6 Sep 10
2	50% 75%	Sep 20 Sep 27
3	35%	Sep 27

15-441  
15-641

**Start early! Do not wait until the last day!**



# Remember building an HTTP/1.0 Proxy in 15-213?

For this project, you will instead:

- Support a subset of HTTP/1.1 (writeup - pg 2, sec 2.2)
- Use **select()**! *This is very important! Do not use threads to handle multiple clients.*
- Parse requests using Lex and Yacc

# Checkpoint 1

---

*Due Sep 6, 2019*

What you need to do (writeup - pg 4, sec 3):

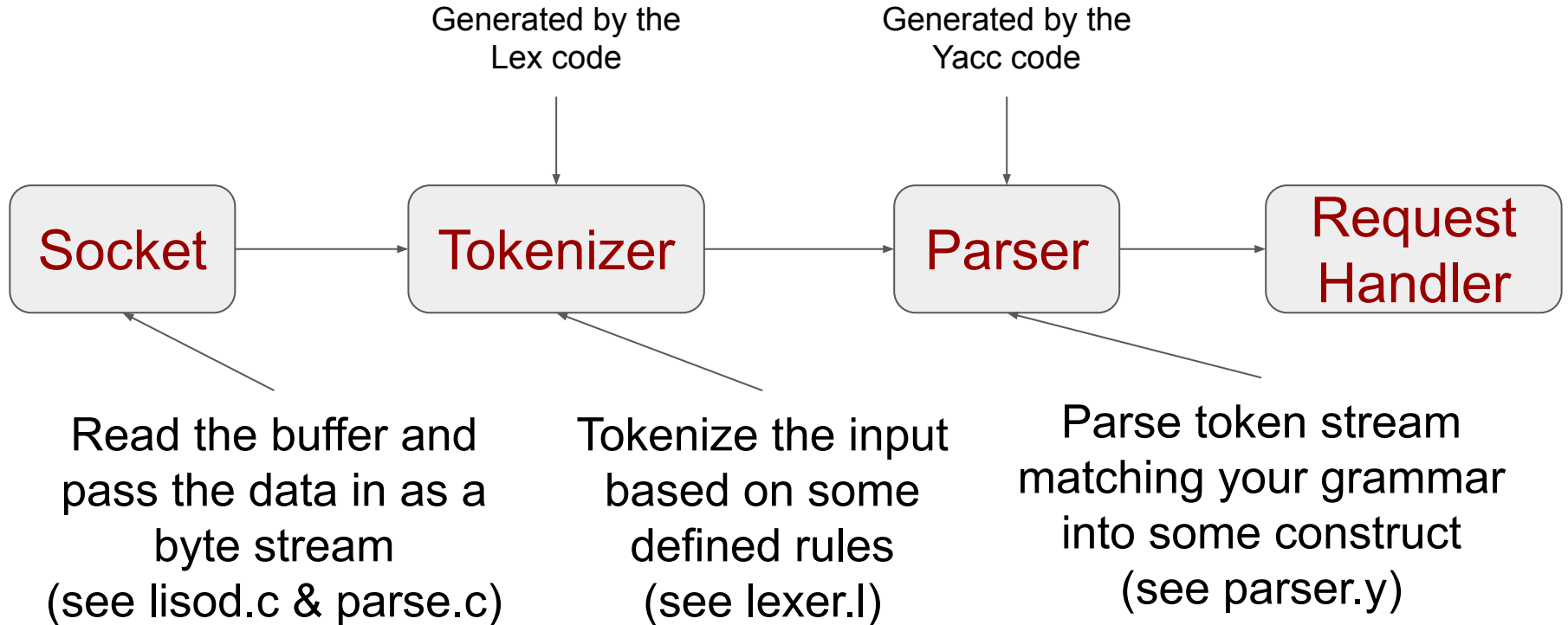
1. Use **select()** to handle up to 1024 concurrent connections. [see ref. 1, 2]
2. Use lex/yacc to parse received requests, and check if they are correctly formed. [see ref. 3]
3. Echo well formed requests back to client, or return 400 error code (BAD\_REQUEST) for incorrectly formed requests

# Select()

1. Utilize the man page [ref. 1]
2. Read the relevant section in CSAPP, the 15-213 textbook, to gain an understanding of how **select()** works.  
*Remember that you must cite everything you take from CSAPP!* [ref. 2]
3. Review section 7.2 in Beej's guide [ref. 3] to see examples. *This is a great resource for most other topics in this class, so use it extensively!*

# Lex and Yacc

The basic flow is this:





# Lex

- It's a program that breaks input into sets of "tokens," roughly analogous to words.
- The general format of a Lex source file is:
  - `{definitions}`                      **Definition of tokens**  
%%
  - `{rules}`                                **Handler for detected token**  
%%
  - `{user subroutines}`            **C code (Process tokens)**
- The absolute minimum Lex program is thus %% (no definitions, no rules) which translates into a program which copies the input to the output unchanged.

# Yacc

- YACC can parse input streams consisting of tokens with certain values.
- YACC has no idea what 'input streams' are, it needs pre-processed tokens.
- A full Yacc specification file looks like:  

<code>{declarations}</code>	Types of each token
<code>%%</code>	
<code>{rules}</code>	Grammar
<code>%%</code>	
<code>{programs}</code>	C code
- The smallest legal Yacc specification is  

<code>%%</code>	
<code>{rules}</code>	

# Quick example of using Lex/Yacc

Let's say we have a thermostat that we want to control using a simple language. A session with the thermostat may look like this:

```
heat on
    Heater on!
heat off
    Heater off!
target temperature 22
    New temperature set!
```

The tokens we need to recognize are: heat, on/off (STATE), target, temperature, NUMBER.

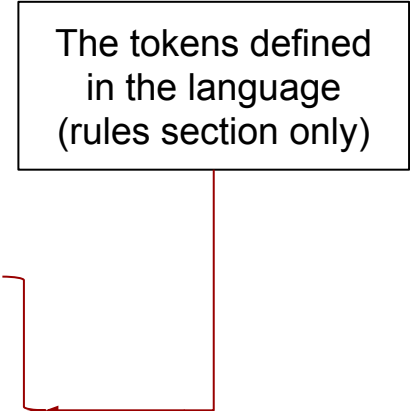
This is taken from the Lex/Yacc reference at the end of the slide deck! [ref. 4]

# Lex file - the tokenizer!

```
%{
#include <stdio.h>
#include "y.tab.h"
}%
%%
[0-9]+
heat
on|off
target
temperature
\n
[ \t]+
%%

return NUMBER;
return TOKHEAT;
return STATE;
return TOKTARGET;
return TOKTEMPERATURE;
/* ignore end of line */;
/* ignore whitespace */;
```

The tokens defined  
in the language  
(rules section only)



# Yacc file - the {rules} section

```
commands: /* empty */
| commands command
;
```

**Tells Yacc to look for commands.** Notice the recursion - this breaks down a series of commands into single commands

```
command:
heat_switch
|
target_set
;
```

**Defines what a command is** - referencing 2 other rules for the heat on/off command or the temp target command

```
heat_switch:
TOKHEAT STATE
{
printf("\tHeat turned on or off\n");
}
;
```

**Defines the heat command** (strictly made up of 2 tokens returned in Lex), and what to do when that token is recognized

```
target_set:
TOKTARGET TOKTEMPERATURE NUMBER
{
printf("\tTemperature set\n");
}
;
```

**Defines the temperature command** (strictly made up of 3 tokens returned in Lex), and what to do when that token is recognized

# Lex and Yacc tips



Read Reference 3 - it continues beyond this to make a more complex grammar

Break down the starter Lex/Yacc code into its sections and start figuring what it already does, so you can add the missing functionality

# RFCs

---



## What is an RFC?

A Request for Comments (RFC) is a formal document from the Internet Engineering Task Force (IETF) that is the result of committee drafting and subsequent review by interested parties. Some RFCs are informational in nature. Others are intended to become Internet standards. A few are even intentionally humoristic (Hyper Text Coffee Pot Control Protocol, RFC 2324)

## Which RFCs should I take a look at for P1?

- HyperText Transport Protocol (HTTP) 1.1      RFC 2616
- Transport Layer Security      RFC 2818
- Common Gateway Interface      RFC 3875

# How to read an RFC

---



RFC 2616 (HTTP/1.1) : 21 sections, 176 pages, .txt doc

- Luckily for us, there is a table of contents!
- Read the RFC selectively:
  - Rapidly skim through the whole RFC at first, to get a sense of the document's structure and the type of information it contains
  - Identify and select which sections are important for what you are trying to build
  - Read the important sections very carefully, the RFCs contain a lot of information.
- You may want to print the RFCs, and mark them up to indicate which parts are important for this project, and which parts are not needed



# RFC 2616, P1CP1

*Which parts of the RFC are the most useful to figure out the requirements of a good HTTP/1.1 request? And the kind of bad requests you should test for?*

The RFC 2616 table of contents contains sections and subsections: below are the names of the sections. Which sections seem important to you?

- |   |  |
|---|--|
| 1. Introduction                               | 9. Method Definitions                            |
| 2. Notational Conventions and Generic Grammar | 10. Status Code Definitions                      |
| 3. Protocol Parameters                        | 11. Access Authentication                        |
| 4. HTTP Message                               | 12. Content Negotiation                          |
| 5. Request                                    | 13. Caching in HTTP                              |
| 6. Response                                   | 14. Header Field Definitions                     |
| 7. Entity                                     | 15. Security Considerations                      |
| 8. Connections                                | 16 - 21. Acknowledgment, Appendices, Index, etc. |

# RFC 2616, P1CP1

*Which parts of the RFC are the most useful to figure out the requirements of a good HTTP/1.1 request? And the kind of bad requests I should test for?*

The RFC 2616 table of contents contains sections and subsections: below are the names of the sections. Which sections seem important to you?

- |   |   |
|---|---|
| 1. Introduction                               |   |
| 2. Notational Conventions and Generic Grammar |   |
| <b>3. Protocol Parameters</b>                 |   |
| <b>4. HTTP Message</b>                        |   |
| <b>5. Request</b>                             |   |
| <b>6. Response</b>                            |   |
| <b>7. Entity</b>                              |   |
| <b>8. Connections</b>                         |   |
|   | <b>9. Method Definitions</b>                        |
|   | <b>10. Status Code Definitions</b>                  |
|   | 11. Access Authentication                           |
|   | 12. Content Negotiation                             |
|   | 13. Caching in HTTP                                 |
|   | <b>14. Header Field Definitions</b>                 |
|   | 15. Security Considerations                         |
|   | 16 - 21. Acknowledgment, Appendices, Index ... etc. |

# Class exercise

---

Read/Skim through the Request section of RFC 2616 (5 pages)

Discuss with your neighbors and try to come up with examples of Requests, both bad and good, you would use to test your CP1 web server



# Possible answers

(Non-exhaustive)

- **Good Requests**

- GET / HTTP/1.1\r\n\r\n
- GET / HTTP/1.1\r\nUser-Agent: 441UserAgent/1.0.0\r\n\r\n #One header
- \r\nGET / HTTP/1.1\r\nUser-Agent: blablabla\r\n blablabla\r\n blablabla\r\n HiIAmANewLine\r\n\r\n\r\n #Multiple line header

- **Bad Requests**

- GET\r / HTTP/1.1\r\nUser-Agent: 441UserAgent/1.0.0\r\n\r\n # Extra CR
- GET / HTTP/1.1\nUser-Agent: 441UserAgent/1.0.0\r\n\r\n # Missing CR
- GET / HTTP/1.1\rUser-Agent: 441UserAgent/1.0.0\r\n\r\n # Missing LF



# Writing tests: some advice

---

- The starter code contains an example of testing script : cp1\_checker.py  
It can be run as follows:

*#Start your server first*

```
./lisod <HTTP port> <HTTPS port> <log file> <lock file> <www folder> <CGI script path>  
<private key file> <certificate file>
```



The arguments in **blue** are useless for CP1 but should still be included  
-> dummy values

*#Start the testing script*

```
python cp1_checker <server ip> <server port> <# trials> <# writes and reads per trial>  
<#connections>
```

# Writing tests: some advice

---



- The provided testing script can be useful for you as a starting point. You will of course need to expand it, and add your own tests. You will also be able to use a real browser later in P1.

NB: Python allows your testing script to be short and simple. *No need to write a ton of code!* You are of course also allowed to use any other coding language.

- Comments on the provided testing script:  
socket library:

```
s = socket(AF_INET, SOCK_STREAM) #creates a new socket (IPv4, TCP)
s.connect((serverHost, serverPort)) #connects to your server
s.send(random_string) #sends data to your server
rdata = s.recv(random_len) #receives random_len bytes or less
                                from your server and puts it into rdata
s.close #closes the connection to your server
```

# Writing tests: other tools

---



Apart from the provided testing script, you can also manually test your server using other tools, like *telnet* and *netcat*.

These are command-line tools that allow you to send data to any web server (including yours!) and can help do quick, manual tests before writing a real one in the testing script. See references 5-7 for information about these tools.

For netcat, a simple command to initiate a connection to your *running* Liso server and send a simple HTTP request:

```
$ nc <server IP> <server HTTP port>
GET / HTTP/1.1\r\nUser-Agent: 441UserAgent/1.0.0\r\n\r\n
<Liso-CP1 server's response (400 or echo)>
```

Q & A





# References

1. <http://man7.org/linux/man-pages/man2/select.2.html>
2. CSAPP section 12.2(15-213 textbook)
3. Beej's Guide: <https://beej.us/guide/bgnet/html/multi/index.html>
4. <http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-4.html>
5. <https://linux.die.net/man/1/telnet>
6. CSAPP pgs. 986-987
7. <https://linux.die.net/man/1/nc>