COMP 431 — INTERNET SERVICES & PROTOCOLS

Spring 2020

Homework 1, January 9

Due: January 23

Baby-steps Towards the Construction of a Mail Server – Parsing in Python

Introduction

The goal for the first half of the course is to (partially) build a simple mail server and mail reader that will work with standard Internet mail servers using SMTP (Simple Mail Transfer Protocol) and POP (Post Office Protocol), and mail readers such as Gmail, Apple Mail, and certain versions of Thunderbird or Microsoft Outlook. This assignment is the first (very!) small piece of the mail server: a simple string parser that we will build upon later.

At a high-level a mail server is simply a program that receives SMTP commands (text strings) from clients, processes the commands, and sends the results of the processing back to the client as a response to the command. In this abstract view of a mail server, a server is a program that executes a logically infinite loop wherein it receives a command, processes the command, and then waits for the next command. In this assignment you will develop the portion of the code that will be used by the mail server to process commands it receives.

You will be implementing the protocol validation checks required to assure conformance with the SMTP specification (and interoperability with SMTP implementations developed by others). In a nutshell, you are to write a program to determine if a command (a text string) is a valid SMTP command. An SMTP command is simply a line of text that may look like the following:

```
MAIL FROM:<jasleen@cs.unc.edu>
```

In the SMTP protocol, this command is called the "MAIL" or "MAIL FROM" command. This command is made up of three substrings:

- a command name — the string "MAIL FROM:",
- a "reverse path" — a well-formed "email address" delimited by angle brackets ("<" and ">") that represents the sender of the message being mailed, and
- a "CRLF" line terminator — the command must be terminated by the "carriage return-line feed character sequence" (the Linux "newline" character [which is not visible in the example above since it is a non-printable character]).

The MAIL FROM command is part of the larger SMTP protocol. Protocols such as SMTP are typically specified more formally than the English description above by using a more mathematical specification notation. (These notations are, in essence, a textual form of the syntax diagrams — sometimes called "railroad diagrams" — that are used to specify the formal syntax of a programming language.)

The Assignment — A Python Programming Warmup Exercise

For this assignment you are to write a Python program on Linux to read in lines of characters from standard input (*i.e.*, the keyboard) and determine which lines, if any, are legal SMTP MAIL FROM commands. In computer-science-speak, what you are building is a *parser* to "recognize" strings that conform to the grammar for a MAIL FROM command. (And the process of processing input lines is called "parsing.")

For the purpose of this assignment, you are to implement a parser for the following grammar (formal description of a subset of SMTP commands):[1]

```
<mail-from-cmd> ::= "MAIL" <whitespace> "FROM:" <nullspace> <reverse-path>
                    <nullspace> <CRLF>
<rcpt-to-cmd> ::= "RCPT" <whitespace> "TO:" <nullspace> <forward-path>
                  <nullspace> <CRLF>
<data-cmd> ::= "DATA" <nullspace> <CRLF>

   <whitespace> ::= <SP> | <SP> <whitespace>
           <SP> ::= the space or tab character
    <nullspace> ::= <null> | <whitespace>
         <null> :== no character
 <reverse-path> ::= <path>
<forward-path> ::= <path>
         <path> ::= "<" <mailbox> ">"
      <mailbox> ::= <local-part> "@" <domain>
   <local-part> ::= <string>
       <string> ::= <char> | <char> <string>
         <char> ::= any one of the printable ASCII characters, but not any
                    of <special> or <SP>
       <domain> ::= <element> | <element> "." <domain>
      <element> ::= <letter> | <name>
         <name> ::= <letter> <let-dig-str>
       <letter> ::= any one of the 52 alphabetic characters A through Z
                    in upper case and a through z in lower case
  <let-dig-str> ::= <let-dig> | <let-dig> <let-dig-str>
      <let-dig> ::= <letter> | <digit>
        <digit> ::= any one of the ten digits 0 through 9
         <CRLF> ::= the newline character
      <special> ::= "<" | ">" | "(" | ")" | "[" | "]" | "\" | "."
                  | "," | ";" | ":" | "@" | """
```

In this notation:

- Items appearing (in angle brackets) on the left-hand side of an expression are called *tokens*,
- Anything in quotes is interpreted as a string or character that must appear exactly as written,
- Anything in square brackets ("[" and "]") is optional and is not required to be present,
- The vertical bar "|" is interpreted as a logical 'or' operator and indicates a choice between components that are mutually exclusive.
- Note that in most cases angle brackets (< and >) are used as token delimiters – the exception is that these may be literals in the specification of the forward and reverse path elements.

---

[1] As an aside, this form of notation is a variation of a commonly used notation called Backus-Naur Form (BNF). You will often see the syntax of protocols expressed using BNF and variations on BNF.

In addition, SMTP requires that MAIL FROM, RCPT TO, and DATA commands be received in a particular order. Specifically, SMTP commands must be received in the order:

- a MAIL FROM command must be received first, followed by
- a RCPT TO command, followed by
- a DATA command.

You should implement this grammar (including expected order) *exactly* as it is specified here. Any question you may have about what is or isn't a valid SMTP command can be answered by studying this grammar.

Your program should use your implementation of a parser for the above command grammar as follows. Your program should be structured as a loop that:

- Reads a line of input from standard input (the keyboard in Linux).
- If/when "end-of-file" is reached on standard input (*i.e.*, when *control-D* is typed from the keyboard under Linux), terminate your program. Otherwise…
- Echo the line of input to standard output (*i.e.*, print the line of input exactly as it was input to standard output [*i.e.*, to the Linux window in which you entered the command to execute your program]).
- When well-formed (syntax and order) SMTP commands are read, print responses as described below in section titled "Responses for Well-Formed Commands".
- For invalid (ill-formed) commands, print out the error message as described in the section below titled "Error Processing".

Responses for Well-Formed Commands:

Your program should start in the state of waiting to receive a MAIL FROM command (or end-of-file) and upon receiving this command, transition to the state of waiting for a RCPT TO command. Once a RCPT TO command has been received, upon receipt of a DATA command, your program should transition to the state of processing message data (described below).

- When well-formed MAIL FROM and RCPT TO commands are read, print on the next line the string "250 OK".

  ```
  250 OK
  ```

- When a well-formed DATA command is read, your program should reply with a text message prompting the sender for the message data (the text of the email) as follows:

  ```
  354 Start mail input; end with . on a line by itself
  ```

  The preceding MAIL FROM, RCPT TO, and DATA command lines should be appended to the file named forward/<forward-path> in the current working directory (the directory in which you executed your program), where <forward-path> is the name of the forward path in the RCPT TO command (example provided later). You may assume that the directory *forward* already exists in the current working directory. (Thus, when you are testing your program be sure to (manually) create this directory before you execute your program!) However, if the *forward-path* file does not already exist it will have to be created by your program as part of the process of writing to the file.

  After a DATA command is input, all subsequent lines of input are treated as message data (*i.e.*, the body of an email message). Each line of input should be read in and stored (appended) in the file described above.

Processing of the DATA command ends when the user enters a line containing only a single period (*i.e.*, when the sequence <newline> "." <newline> is read). ~~This last line is not considered to be part of the mail message and hence is not stored with the contents of the mail message.~~

If the DATA command and the subsequent email message text has been successfully entered, your program should again reply with a line of the form:

```
250 OK
```

Processing Multiple Messages

The input of a MAIL FROM command, followed by a RCPT TO command, followed by a DATA command, followed by the text of an email message, corresponds to the sending of a single email message. Once a DATA command has been received and processed (as explained above), another MAIL FROM command can be received. This would correspond to the scenario of an SMTP client sending multiple email messages to a server in a single SMTP session.

Thus, in terms of the SMTP state machine, upon receipt and processing of a DATA command, your program should transition back to the state of waiting to receive either a MAIL FROM command or an end-of-file indication.

In all cases, the SMTP processing continues until end-of-file is reached on the input stream. If errors are encountered on input lines you should simply emit the appropriate error message and begin the parse of the next line of input. If end-of-file is reached in the middle of processing an email message, then no message data for this message should be written to the forward-path file. That is, only complete, well-formed messages should be written to the *forward-file*.

Example Processing of a Well-Formed Sequence:

Consider that the following input is provided to your program:

```
MAIL FROM: <jasleen@cs.unc.edu>
RCPT TO: <smithfd@cs.unc.edu>
DATA
Hey Don, do you really think we should use SMTP as a class
Project in COMP 431 this year? The smart students are going to
Figure out automated ways to use this project to send
Anonymous SPAM to the world!
.
```

Then a valid interactive exchange with your program might look like the following (the blue lines are simply demonstrating the input and should **not** be printed by your program):[2]

```
MAIL FROM: <jasleen@cs.unc.edu>
MAIL FROM: <jasleen@cs.unc.edu>
```

[2] This example assumes you are manually typing your inputs to your program. If you type your commands into your program the above is what you can expect to see in your terminal window. The blue input lines represent what Linux will echo to your terminal/ssh window as you type in commands and the black output lines represent the outputs that your program will generate in response. If you use I/O redirection to read commands from a file (or write the output to a file) you will not see all these lines in your terminal window.

```
250 OK
RCPT TO: <smithfd@cs.unc.edu>
RCPT TO: <smithfd@cs.unc.edu>
250 OK
DATA
DATA
354 Start mail input; end with . on a line by itself
Hey Don, do you really think we should use SMTP as a class
Hey Don, do you really think we should use SMTP as a class
project in COMP 431 this year? The smart students are going to
project in COMP 431 this year? The smart students are going to
figure out automated ways to use this project to send
figure out automated ways to use this project to send
anonymous SPAM to the world!
anonymous SPAM to the world!
.
.
250 OK
```

Note that the DATA command isn't fully "accepted" (the 250 acknowledgement message isn't emitted) until after the body of the mail message is read in till the end of the input sequence. After the DATA command and message data have been read, the next expected line in standard input should be a MAIL FROM: command.

In addition to the above standard I/O, your program will append the following text to the file *forward/<smithfd@cs.unc.edu>*:

```
MAIL FROM: <jasleen@cs.unc.edu>
RCPT TO: <smithfd@cs.unc.edu>
DATA
Hey Don, do you really think we should use SMTP as a class
Project in COMP 431 this year? The smart students are going to
Figure out automated ways to use this project to send
Anonymous SPAM to the world!
.
```

Error Processing:

For ill-formed or unrecognized SMTP commands, your program should output one of the following SMTP error responses:

```
500 Syntax error: command unrecognized
501 Syntax error in parameters or arguments
503 Bad sequence of commands
```

Ill-formed MAIL FROM, RCPT TO, and DATA commands can generate either of these errors. Whenever the first parsed tokens on an input line do not match the tokens for any command name in the grammar, a type 500 error message is generated. Operationally, a 500 error means that your parsing can't recognize which SMTP command it should be parsing.

If the command token(s) are recognized (*i.e.*, your parser "knows" what command it's parsing), but some other error occurs on the line, a type 501 error message is generated. The following would be examples of 500 and 501 errors:

```
MAILFROM: <jasleen@cs.unc.edu>
MAILFROM: <jasleen@cs.unc.edu>
500 Syntax error: command unrecognized
RCPT TO : < ivysat@cs.unc.edu>
RCPT TO : < ivysat@cs.unc.edu>
500 Syntax error: command unrecognized
MAIL   FROM: <jasleen @cs.unc.edu>
MAIL   FROM: <jasleen @cs.unc.edu>
501 Syntax error in parameters or arguments
MAIL FROM: < ivysat@cs.unc.edu
MAIL FROM: < ivysat@cs.unc.edu
501 Syntax error in parameters or arguments
MAILFROM: <jasleen @cs.unc.edu>
MAILFROM: <jasleen @cs.unc.edu>
500 Syntax error: command unrecognized
```

Note that this last command appears to have two errors in it. It has a syntax error in the command name and, it appears to also have a parameter error ("jasleen @"). We report a syntax error and not a parameter error because since a parser would not recognize "MAILFROM," the parser does not "know" what SMTP command it is parsing. And since the parser does not know what SMTP command it is parsing it can't know if what follows (the parameters/arguments) is correct or not. Said another way, syntax errors in the SMPT command name (500 errors) take precedence for reporting over parameter/argument errors (501 errors). That is, in order to have a 501 error, you have to first recognize which SMPT command you are parsing the parameters/arguments for.

Whenever a command is received out of order, a type 503 error is generated.

Syntax errors in the command name (type 500 errors) should take precedence over all other errors. That is, before a command can be considered to be out-of-order, the command being parsed must be recognized. Thus, for example, if a RCPT TO: command is expected but the input DATAX is seen next (*i.e.*, an ill-formed DATA command is seen), this would be considered a type 500 syntax error and not an out-of-order error.

Out-of-order errors (type 503 errors) should take precedence over parameter/argument errors (type 501 errors). That is, if a RCPT TO: command is expected but the input "MAIL FROM: bob" is seen, this would be considered an out-of-order error and not a parameter/argument error even though there is an error in the parameters for the MAIL FROM: command. Said another way, parameter/argument errors can only occur on commands that (1) have a syntactically correct command name, and (2) are received in the correct order.

In all cases the parsing of an input line generates exactly one of the responses 250, 354, or 50x. If one of the 50x responses is output, your state machine should remain in the same state and process the next input line (i.e., continue processing looking for a correct instance of the next expected command).

Homework Requirements

Write a Python program to read in lines of input from standard input and process them as SMTP commands. You should check for syntax errors as well as ordering errors in the sequence in which commands are input

and output error responses as appropriate. All ouput and error messages should be formatted *exactly* as shown above.[3]

Your program should terminate when it reaches the end of the standard input stream (when *control-D* is typed from the keyboard under UNIX or *control-Z* on Windows). Your program must not output any user prompts, debugging information, status messages, *etc*.

Testing

To aid in testing, sample input and output files will be provided on the course web page (and announced on piazza). Please note that these sample tests will *not* be comprehensive (i.e., you should test your program much more thoroughly than these test files) – and grading will certainly rely on many additional tests. These sample files are being provided simply to aid you in initial testing, as well as catching if your program is making basic formatting/syntax mistakes.

Submitting Your Programs for Grading

For this assignment, you will "turn in" your program for grading by placing it in a special directory on a Department of Computer Science UNIX machine and filling out a Google form that will be provided on the course website. To ensure the TAs can grade your assignments in an efficient and timely fashion, please follow the following guidelines *precisely*. In particular, the order of these steps is critical. You should perform the steps below in *exactly* the order listed. WARNING: Failure to follow these steps exactly will result in the TA being unable to read your files. Should this occur, you will receive a grade of "0" for the assignment!

- Log on to the Linux server named comp431-1sp20.cs.unc.edu (use your onyen login id and password).
- In your Linux home directory on the above server, create the directory structure: *comp431/submissions*. (That is, create the directory *comp431* in your home directory and inside this directory, create the directory *submissions*.)
- Do not change any ACL settings for any files in your home directory.
- For each assignment you will create a subdirectory with a name specified in the assignment. You must also name your program as specified in the assignment. For this assignment you should name your final program "SMTPserver.py" and store it in a directory named "hw1" (inside your *~/comp431/submissions* directory).
- When you have completed your assignment you should put your program in the specified subdirectory and fill out the Google form linked from the course web page, indicating that the program is ready for grading.
- Make sure that your program has the correct path by running the command below: "less ~/comp431/submissions/hw1/SMTPserver.py".
- *Do not change any of your files for this assignment after the submission deadline!* The lateness of assignments will be determined by the Linux timestamps on your program files. If the timestamps on the files change after the submission deadline, you will be penalized for turning in a late assignment.
- All programs will be tested under Linux using python3. You should be able to develop your programs in whatever Python development environment you prefer and then upload to Linux. However, it is your responsibility to test and ensure the program works properly in Linux

[3] This is done for ease of grading. The actual SMTP specification only specifies the response numbers and allows arbitrary text to follow the error code/acknowledgement number.

(specifically, on the machine comp431-1sp20.cs.unc.edu). In particular, if your program performs differently on your PC than it does on the Departmental server (*e.g.*, because of some difference in library versions or Python version), your grade will be based on your program's performance on the Departmental server.

- The program should be neatly formatted (*i.e.*, easy to read) and well documented.
- The homework grade will have the following distribution:
    - 20% 500 error processing: "500 Syntax error: command unrecognized"
    - 20% 501 error processing: "501 Syntax error in parameters or arguments"
    - 20% 503 error processing "503 Bad sequence of commands"
    - 40%: Valid Input Processing (complete, well-formed messages should be written to the forward-file).

Grading Notes

~~For this and all other programming assignment you will only get a "pass"/"fail" grade for your assignment. In order to pass this class you must eventually get a passing "grade" on every assignment.~~

This assignment will be graded according to the grading rubric provided above. The grading will be performed by the TAs only after the submission deadline. It is your responsibility to:

1. Test against the test scripts provided by the TAs (which test against only the sample inputs provided).
2. Test extensively against test cases created by you (before the submission deadline). The TAs will certainly be testing against many more test cases, in addition to the sample cases provided to you. Your grade will depend on how you process the test cases used by the TAs for grading (and not how well you handle just the sample cases provided).
3. This assignment does not have a provision for "resubmissions". It is your responsibility to thoroughly test your code before the submission deadline.

To pass this assignment, your program must correctly process a series of lines of input we will provide to your program. Any processing errors in your program must be fixed before you can pass (get completion credit for) HW1.

All programs in this class will be graded by a script that will provide inputs to your program and compare your program's output against a "key." If your program fails a particular test we will provide you with the input that your program did not process properly. However, it will be up to you to determine why your program did not process the input properly (including determining what the correct output for the test should have been).

We will NOT provide the set of test inputs for your program ahead of time. Part of the skill you must develop as a programmer is to learn how to test your programs and how to generate complete test cases for a program's specification.

Important notes:

- All programs will be tested on the Departmental Linux compute servers. You should develop and test your programs on a Departmental Linux server. Should you choose to develop your program on some other platform — something I *strongly* recommend against — and then upload your program to a Departmental server, it is your responsibility to test and ensure the program works properly on the Departmental servers. In particular, if your program performs differently on your development platform than it does on the Departmental server (*e.g.*, because of some difference in library versions or Python

version), whether or not you get passing credit this assignment will be based solely on your program's performance on the Departmental server.

- The program should be neatly formatted (*i.e.*, easy to read) and well documented. The Instructor and the TA reserve the right to not help you with your code if it's not well formatted and well documented.

Honor Code Reminder

All the code you submit for grading must have been written by you. Under no circumstances can you use code written by a third party. Using code written by another person in the class, by another person at (or previously at) UNC, or code obtained from some website on the Internet — even if you copy the code and rewrite it or otherwise adapt it for use in your program — is expressly forbidden and will constitute an Honor Code violation.

Final Editorial Comment

A frequent complaint about the grading in this course is that the grading of the programs is too "picky." To paraphrase an end-of-semester course evaluation comment from a former student: "*too much emphasis is placed on how hard did you try to break your program instead of whether or not you implemented what was intended*." The point this student missed, and the point you should internalize, is that for most protocol implementations there is no notion of "partial correctness." Either a protocol is correctly implemented or it isn't. An SMTP implementation can be 99.99% correct and yet still not be able to send or receive a single email message. For this reason, you have to correctly process all lines of our test inputs in order to pass this assignment.