COMP 431 — INTERNET SERVICES & PROTOCOLS

Kevin Jeffay

Spring 2023

Homework 1, January 10

Due: 12:30 PM, January 19

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

The goal for the first half of the course is to 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 in later assignments.

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

Specifically, you are to write a program to determine if a message (a text string) is a valid SMTP "MAIL FROM" message. This is a message that tells a mail server who (which user) is trying to send an email message.

An SMTP message is simply a line of text that looks like the following:

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

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

- a command/message 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 message 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 message 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.)

For example, the formal description of the MAIL FROM message is:[1]

```
<mail-from-cmd> ::= “MAIL” <SP> “FROM:” <reverse-path> <CRLF>

 <reverse-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
                    <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
           <SP> ::= the space or tab 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.

For example, the sample MAIL FROM message above conforms to the formal description and hence is a valid SMTP message (assuming it is terminated with a newline character — the default line termination “character” for Linux).

The following strings do not conform to the formal description and would be rejected as invalid or illegal requests.

```
MAIL FROM:<jeffay @cs.unc.edu>
MAIL FROM:<jeffay@cs.unc.edu >
mail from:<jeffay@cs.unc.edu>
```

The first string contains an invalid *mailbox* component, and the second string contains an invalid *path* component. The third string is technically invalid because it does not contain the literal strings “MAIL” or “FROM” (in upper case). However, as a practical matter, it turns out that the third string will be recognized by most mail servers as a valid SMTP MAIL FROM message. That is, while the formal SMTP protocol definition requires messages to be in upper case, virtually all implementations of SMTP in servers allow messages to be in either upper or lower case. This illustrates one of the (frustrating!) realities of networking: often protocol implementations take liberties with the formal specifications and these liberties become *de facto* standards.

---

[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.

As another example, notice that in the grammar above, the reverse path token must occur immediately after the literal string "FROM:". That is, the grammar does not allow any space to appear between "FROM:" and the *reverse-path* token. However, it turns out that virtually all SMTP implementations don't implement this rule in the grammar exactly as specified and instead allow any amount of "whitespace" (spaces and tabs) to appear between any of the elements of the MAIL FROM message. For example, technically, the following MAIL FROM messages would be invalid according to the formal specification because they either have "too much" whitespace or whitespace in disallowed locations:

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

Nonetheless, most SMTP servers would consider each of these strings to be valid MAIL FROM messages and treat them as being equivalent to:

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

Thus, the *de facto* standard grammar for the MAIL FROM message (the grammar implemented by most SMTP servers) is

```
<mail-from-cmd> ::= <mail-token> <whitespace> <from-token> ":" <nullspace>
                    <reverse-path> <nullspace> <CRLF>
   <mail-token> ::= "MAIL" | "mail"
   <from-token> ::= "FROM" | "from"
   <whitespace> ::= <SP> | <SP> <whitespace>
    <nullspace> ::= <null> | <whitespace>
```

where "null" means "no character." (Thus, a "nullspace" is zero or more whitespace characters.)

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 messages. In computer-science-speak, what you are building is a *parser* to "recognize" strings that conform to the grammar for a MAIL FROM message. (And the process of processing input lines is called "parsing.")

Specifically, you are to implement a parser for the following grammar:

```
<mail-from-cmd> ::= "MAIL" <whitespace> "FROM:" <nullspace> <reverse-path>
                    <nullspace> <CRLF>
   <whitespace> ::= <SP> | <SP> <whitespace>
           <SP> ::= the space or tab character
    <nullspace> ::= <null> | <whitespace>
         <null> :== no character
 <reverse-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> ::= "<" | ">" | "(" | ")" | "[" | "]" | "\" | "."
                | "," | ";" | ":" | "@" | """
```

You should implement this grammar *exactly* as it is specified here. Any question you may have about what is or isn't a valid MAIL FROM message can be answered by studying this grammar.

Your program should use your implementation of a parser for the MAIL FROM message 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 message to execute your program]).
- For valid MAIL FROM messages, print on the next line the string "Sender ok".
- For invalid messages, print out the error message "ERROR -- *x*" where *x* is the name of the first token or line in the grammar that is missing or ill-formed.

For example, if the following sample requests were read by your program,

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

the output of your program would be:[2]

```
MAIL FROM: <jeffay@cs.unc.edu>
Sender ok
MAIL FROM:<jeffay@cs.unc.edu>
Sender ok
MAIL   FROM: <jeffay@cs.unc.edu>
Sender ok
MAIL FROM: <jeffay @cs.unc.edu>
ERROR -- mailbox
MAIL FROM: <jeffay@cs.unc.edu >
ERROR -- path
```

---

[2] Note that if you are manually typing in MAIL FROM messages to your program from the keyboard, depending on how your program is organized you will likely see each line of input twice of your screen: once because Linux is echoing what you type to the window of your terminal/ssh session, and a second time because your program is required to echo the input lines to standard output (the window of your terminal/ssh session). Thus, for the first example input above, what you are likely to see on your screen is:

```
MAIL FROM: <jeffay@cs.unc.edu>
MAIL FROM: <jeffay@cs.unc.edu>
Sender ok
```

All output should be written to standard output. Your program should format its output *exactly* as shown above.

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

The purpose of this assignment is to get up to speed with the Python programming language and the use of Linux program development tools. Note that while motivated by a protocol processing problem, in the abstract this assignment has nothing to do with networking and is just a simple text parsing problem.

Program Development and Grading Notes

A Linux server, *comp431sp23.cs.unc.edu*, is for available for use by COMP 431 students. You should be able to log in using your ONYEN and your ONYEN password. If you have any problems please contact the instructor immediately.

Once logged in, in your Linux home directory, create the directory HW1 ("`mkdir HW1`"). For this assignment you should name your final program "parse.py" and store it in the HW1 directory. Remember that Linux file names are case sensitive and hence case matters! Please leave all of your HW1 related files in this directory. Throughout your use of the COMP 431 server, please do not change ACLs (access control lists) or file directory permissions on any files or directories that you create.

It is *highly* recommended that you do *all* your program development and testing on *comp431sp23.cs.unc.edu*. Importantly, all programs will be tested on this server and hence the correctness of your program will be assessed by its performance on this server. Should you choose to develop your program on some other platform — something that is *strongly* discouraged — and then upload your program to the class server, it is your responsibility to test and ensure the program works properly on the class server. If your program performs differently on your development platform than it does on the class server (*e.g.*, because of some difference in library versions or Python version) your completion status will depend on the program's performance on *comp431sp23.cs.unc.edu*. That is, *whether or not you get passing credit this assignment will be based solely on your program's performance on the class server*. Make sure your programs work on *comp431sp23.cs.unc.edu*!

All programs, unless otherwise specified, should be written to execute in the current working directory. Generally, unless the homework assignment specifies otherwise, you should execute your program without using any command line arguments or compiler/interpreter options.

Your program should be neatly formatted (*i.e.*, easy to read) and well documented. If your program is not well formatted and well documented, the TA has the right to not provide help with the assignment until the program is made readable.

Make sure you put your name in a header comment in every file you submit. Make sure you also put an Honor pledge in every file you submit for grading.

Submitting Your Program for Grading

The precise details on the process for submitting programs for grading will be distributed later this week. However, in the meantime, note that whether or not your homework submission is "on time" or "late" will be determined solely based on the last file modification date of the file(s) in your HW submission directory. If you are trying to get the extra credit bonus for completing all assignments on time and if your program has a timestamp after 12:30 PM on the due date it will be considered late and you will be ineligible for the "on time" bonus.

Once you have completed your program and submitted it or grading, *do not change any of the files in the HW submission directory until you receive a grading response from the TA*. If you wish to keep fiddling with your program after you submit it, you should make a copy of your program and work on the copy and not modify the original. Once you've received the pass/fail notice from the TA you may modify your program as you like.

Grading

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.

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 HW1 (*i.e.*, before you can 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.

Honor Code Reminder

All the code you submit for grading must have been written by you. Under no circumstances can you use any 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.