Exercise: Implement a CLI to extract data from an EML header

1. Objective

Write a program in Go that, given an email in the EML format, extracts information about the sender of the email as indicated in the “From” header. The details of the output format are given below.

The program is to be run from the command line, as follows:

```
$ eml-sender file.eml
```

Any fatal error (for example, the input file doesn’t exist) must be logged on stderr and the process’ exit status must be non-zero. Normal program output must go to stdout.

2. Description of the input

An EML file is a simple text file whose content adheres to RFC 5322. Figure 1 shows an example of an email in the EML format.

```
Received: from AM9PR09MB4721.eurprd09.prod.outlook.com (2608:10f6:20c:283::21)
 by DB8PR09MB4389.eurprd09.prod.outlook.com with
 Microsoft SMTP Server (version=TLS1_2)
Received: from AM9PR09MB4721.eurprd09.prod.outlook.com
 ([fe80::61c2:c511:13ad:d558]) by AM9PR09MB4721.eurprd09.prod.outlook.com
From: "Vogel, Martin (SIG Peitz GmbH)" <martin.vogel@sig.com>
To: "Neumann, Artie" <artie.Neumann@someone.com>
Subject: =?utf-8?B?QW5mcmFnZSBSb2hyX0JFVCBFaW5zcHJpdHprw7xobGVy?=
Date: Fri, 29 Nov 2027 12:17:34 +0100
Message-ID: <03b701d5a6a6$9c9881c0$d5c98540@sender>
MIME-Version: 1.0
Content-Type: text/plain
Content-Transfer-Encoding: 8bit

This is sample email. It only contains this text.
```

Figure 1. An example email in the EML format

Only read the file until you need it: until you find the first “From:” or the header ends; that is, don’t read the body of the email.

The header ends when there is an empty line; you can take that to mean the sequence of bytes <LF><LF> or <CR><LF><CR><LF>, where LF=0x0A, CR=0x0D. Hardcode the sequence that is native to your platform (Linux/macOS, or Windows).

3. Output format

The result written to stdout is a JSON like the following (please do respect the indentation and line formatting):

```
{
   "addr_spec": "martin.vogel@sig.com",
   "display_name": "Vogel, Martin (SIG Peitz GmbH)",
   "error": null
}
```

If the file could be read but it was not possible to find an addr-spec, the “error” field can be used to indicate the issue. Example:

```
{
   "addr_spec": "",
   "display_name": "",
   "error": "no addr-spec found"
}
```

The contents for “display_name” and “addr_spec” are explained below.

4. Display name versus addr-spec

The following are examples of correctly formatted “From” headers:

1. From: “Peter Walters” <**peter@company.com**>
2. From: <**peter@company.com**>
3. From: **peter@company.com**

The part that with a grey background in these examples is called the “addr-spec”. The part with a yellow background, appearing only in the first example, is called the “display name”.

In this exercise we assume a very simplified syntax for email addresses. You can assume that:

- email address can only contain 7-bit ASCII characters (no extended ASCII such as é or ü, no Unicode, etc.).
- the “From” header fits entirely in one line.

But even with these simplifications, correctly extracting the information requires some effort.

All the following examples must be correctly processed by your program. As before, the parts with a grey background are addr-specs.

1. "Peter" (Sally's friend) <**peter@pan.com**>
2. "Peter" <**peter@pan.com**> (Mary's friend)
3. "Peter" <**peter@pan.com**> (met @ school)
4. Peter <**peter@pan.com**>
5. Peter Pan <**peter@pan.com**>
6. Peter **peter@pan.com**

```
7.    Peter Jackson <peter@pan.com> (Mary's friend)
8.    “Peter Pan"   <  info@peter.pan >
9.    “Embedded quote \” here” <peter@pan.com>
10.   “Escaped \\backspace” <peter@pan.com>
11.   (Peter) <peter@pan.com>
12.   (Peter) peter@pan.com
13.   “Bill G <bill@gates.com>" <peter0312@gmail.com>
14.   “Bill Gates” (bill@gates.com) peter0421@gmail.com
15.   “peter@pan.com ” <peter@pan.com>
16.   “Nested comments” (comment (other comment) more) <peter@pan.com>
17.   “Peter” (ex-Harvard) <peter@pan.com> (now corp.com’s CTO)
```

The character \ introduces an “escape”: when \ appears, the following character must be taken literally (at least one character must follow it; otherwise, it is an error).

For simplicity, you can consider the “display name” whatever precedes the addr-spec. If the addr-spec is within angle brackets, those must be removed. Example:

```
Peter Jackson <peter@pan.com> (Mary's friend)
```

display_name is Peter Jackson and addr_spec is **peter@pan.com**

The following are invalid as email addresses; an error must be reported via the “error” field:

#	Faulty email address	Reason
1	`Liese Martin <e14011>`	Missing @domain
2	`<google - Leupoldmwv<chartise.clark@gmail.com>>`	Nested < .. > not allowed as part of addr-spec
3	`"DGA Zierbach"`	No addr-spec
4	`"Knab" <newsreply.knab.@planet.com>`	RFC 5322 forbids the localpart (what comes before the last @ in addr-spec) from ending in a dot
5	`"Peter <peter@pan.com>`	Unterminated quoted part
6	`"Peter\” Pan <peter@pan.com>`	Unterminated quoted part
7	`"Peter” <peter@pan.com> <peter@corp.com>`	More than one addr-spec given
8	The From header is missing or its value is empty	“From” header missing or value is empty

5. Regex for locating potential addr-spec candidates

In case you find it useful, the following regex can help you find potential addr-spec candidates:

```
("(?:[!#-\[\]-~]|\\[\t -~])*"|[!#-'*+\-/-9=?A-Z\^-~](?:\.?[!#-'*+\-/-9=?A-Z\^-~])*)@([!#-'*+\-/-9=?A-Z\^-~](?:\.?[!#-'*+\-/-9=?A-Z\^-~])*|\[[!-Z\^-~]*\])
```

This regex doesn't solve the problem completely. For example, given:

> "Bill G <bill@gates.com>" <peter0312@gmail.com>

it will find two potential addr-specs. You need to determine yourself which one is effectively an addr-spec.

Suggestion: Use an online tool such as https://regex101.com to play around with the regex and see what it does for a given input.

6. Constraints

1. You can use packages from the Go standard library, but you are not allowed to use the **net/mail** package, neither directly nor indirectly
2. You must do this exercise strictly on your own, without any help
3. Using code that you have not designed and written yourself for this exercise is not allowed
4. Using code generation tools (with or without AI) is not allowed
5. Your solution must include unit tests, covering both the positive and negative cases (those that should fail based on the spec, must be proved to fail)
6. Functions must be documented (in the style suggested by/customary in Go).

7. Overall documentation

Write a README.md file containing:

1. a high-level description of your approach to the problem
2. the key insights to understand your solution
3. weaknesses (or failures) you have identified in your approach (design) or in the implementation.
4. any other comment you would like to share.

8. Deliverables

Please submit your complete project as a compressed archive (ZIP file) to our recruiting team, at the address and on or before the time indicated to you when you received this instruction.

Your ZIP file must include all sources, test data (for example, EMLs) and documentation or any other file that are necessary to build or use your solution. Please don't include any binary.

9. Interview preparation

The recruiting team will organize a follow-up interview, where you will have the opportunity to do a comprehensive walkthrough of your code, and explain the rationale behind your key design decisions. Be prepared to discuss your approach to solving the problem, any challenges you encountered, and how you overcame them.

We appreciate the time and thought you'll put into this exercise. Our goal is to get a sense of your ability to learn a new language, your problem-solving approach and communication style, not to test for the "one right answer." Best of luck — we're looking forward to reviewing your work!