Middle East Technical University | Department of Computer Engineering

CENG 444

Language Processors

Spring 2023-2024

Project I - Part I

Due date: 22 March 2024, Friday, 23.59

1 Problem Definition

In this assignment, you are expected to write a program that uses a lexer to tokenize and process some UTF-8 encoded text files of a given format. You are required to use C++ and `flex` to write your program. Your program will process a given input file to solve three problems running in parallel:

1.1 Number Sequence Processing

Your processor must be sensitive to two kinds of lines for number sequence processing: number and `reset`. Your processor is always processing a sequence (i.e. the current sequence). When a sequence ends with `reset`, another sequence starts. Your processor should enumerate the detected sequences starting from one, increasing by one. Hence, you must automatically start the sequence number one when the program starts. A number gets detected on the conditions that:

- it is placed in a line solely,
- it conforms to the following micro-syntax:
  - A number can have an optional sign character that can be either '+' or '-'
  - A number can be composed of the valid combinations of whole part (W), decimal separator (.), and fractional part (F). Whole part can be either zero, or a sequence of digits starting with a non-zero digit. Fractional part is a nonempty sequence of digits, not ending with zero. Following are valid examples of W,".", and F combinations:

W	.	F	Example number
1	1	1	1.25
1	1	1	0.25432
0	1	1	.25
0	1	1	.0134
1	0	0	123
1	0	0	0

Table 1: Valid number combinations

  - A number can have an optional exponential part prefixed by 'e' or 'E'. When supplied, an optional sign that can be either '+' or '-' may follow. Then, a sequence of digits must be given. The first digit must be non-zero.

When your processor detects the reserved word `reset` or the file content was consumed, it must generate a report in a text file named `sequences.txt`. The file must contain human-readable feedback that answers the following questions regarding descriptive statistics:

- What is the minimum value?
- What is the maximum value?
- What is the number of entries in the list?
- What is the average?
- What is the standard deviation of the values in the list?
- Is the sequence monotonic? If yes, is it increasing or decreasing?

For example, the following is an acceptable format:

```
Sequence Number: <The one-based number of the sequence>
Minimum : <your calculated value>
Maximum: <your calculated value>
Number of entries: <your calculated value>
Average: <Your calculated value>
Standard Deviation: <Your calculated value>
Is monotonic: "Yes" / "No, increasing" / "No, decreasing"
<A blank line for ease of reading>
```

Beware of empty sequences. Your processor should not be suffering division by zero or similar exceptions. You are expected to handle such cases and report accordingly. In `sequences.txt`, report such cases as follows:

```
Sequence Number: <The one-based number of the sequence>
Empty!
<A blank line for ease of reading>
```

1.2 Subnet Test

Your processor must be sensitive to three kinds of lines: `gateway`, `mask`, and `address`. Your processor must maintain a simple network configuration comprised of `gateway` and `mask`, which are ipv4 network addresses that can be updated with `gateway` or `mask` lines. When an `address` line is detected, your processor must perform a test to detect whether the specified address is in the *subnet* described by the configuration. The configuration items are unbound at the beginning, and any test requested by an address line while any of the configuration items are unbound is invalid. The formats of the lines are as follows:

- Gateway line format: `gateway <ipv4 address>`
- Mask line format: `mask <ipv4 mask>`
- Address line format: `address <ipv4 address>`

A valid ipv4 address has four components separated with dots. Each component is either zero or positive integer using decimal digits. Each component must be in the range [0 - 255]. It is a common and recommended practice to convert each ipv4 address to an unsigned 32-bit integer where the first component is placed as the highest-order byte. A component cannot have a lower byte order than its subsequent component.

A valid ipv4 mask syntactically conforms to the ipv4 address. But its semantics are different. Since the main subject is lexical analysis, limited information about a subnet mask will be sufficient to meet the targets of this assignment. When placed into a 32-bit unsigned integer, the ipv4 mask must have the most significant bit as one. Scanning from the highest order bit to the lowest, if the series of ones is broken by a zero bit, the remaining bits must be zero. Any other bit pattern falling outside this definition of ipv4 mask is illegal.

When a `gateway` line is detected, your processor must set the current `gateway` address if the `gateway` address found in the line is a valid ipv4 address. Note that the line must not contain extra data, which invalidates the whole line.

When a `mask` line is detected, your processor must set the current `mask` if the `mask` found in the line is a valid `mask`. Note that the line must not contain extra data, which invalidates the whole line.

When an `address` line is detected, your processor must perform a test that can be formulated as $(M\&G) == (M\&A)$, where M is the current mask, G is the current gateway, A is the address in question. M, G, and A are 32-bit unsigned values and the notation of the formulation is conformant to C expressions. If the result is true, then the address is identified as `in`, otherwise it is identified as `out`. Note that it is illegal if an address line is detected before setting the current `gateway` and `mask`.

Your processor must create a line in the report file named `nettests.txt` for each successful test. The line must adhere to the format noted below, where the test result will be either `in` or `out` in parallel to the aforementioned test:

```
M: <ipv4 address> G: <ipv4 address> A: <ipv4 address> => <Test result>
```

1.3 Identifier Test

Your processor must be sensitive to the lines that contain a single identifier. An identifier is a word starting with a letter or underscore. Subsequent characters can be a letter, underscore, or a digit. Note that a letter can be English, Greek, and Turkish characters. Your processor should report the number of the valid identifiers detected, `N`, in a file named `identifiers.txt` as follows:

```
The number of the identifiers detected is <N>.
```

2 Specifications & Hints

- You do not need to look for Byte Order Mark in the given input files. You can safely assume the input is UTF-8 encoded.
- You are expected to process the input file line by line. Each line in the input (except the last line) ends with a newline character. The last line is not guaranteed to have a newline character, make sure you handle both cases carefully.
- There can be empty lines in the input text. You should not generate any errors for them.
- When calculating the standard deviation of a sequence, use the whole population-based standard deviation calculation model, not the sample-based model.
- Use `atof` while converting the number lexemes but stick to the micro-syntax given, which is strict. `atof` may be lenient while interpreting its input!
- For each line that does not conform to the rules noted (some of them were referred to as invalid), place a line into the text output file named `exceptions.txt`. Each line must have the following syntax:

```
<input line number>: <error description>
```

- Use `flex` to generate C++, not C file. C implementation will **not** be accepted.
- Name your `.l` file as `project01lex.l`.
- Never modify the automatically generated `lex.yy.cc` file. In the evaluation phase, `flex` will be used as follows to regenerate the lexer: `flex -+ project01lex.l`
- Subclass the generated lexer class to implement the assignment project. Do not rely on globals to implement the state. The only global can be the lexer instance.
- In case you use Eclipse IDE, take care following the directions below, which are easy as these are compliant with defaults:
  - Place all of your source files (.l, .cpp, .cc) in the project folder, not in any subfolder.
  - Develop your solution in a way that input and output text files (.txt) will be placed in the project folder.
  - Do not make any changes to project options that designate executable target folder other than Debug. The evaluation process will use Debug build.
  - Make sure that make utility succeeds building when run (`make clean` followed by `make all`) in the Debug folder of the project.
  - Submit your project folder (not the workspace) as a .zip file in compliance with the naming conventions noted above. The project folder is the folder containing the .project folder.
- If you breach from the advice of using Eclipse C/C++ complying with the directions provided, your presence and participation in the evaluation may be required. In this case, please describe precisely the steps to build and run the program. Additionally, provide shell scripts for building and running with the support of a README file that contains directions for building and running, you may also provide a makefile if needed.
- The assignment material contains two simple text files (`lettersutf8.txt, lettersutf8table.txt`) for your convenience. These files contain non-ANSI letters with their respective UTF-8 codes.

3 Regulations

- **Implementation:** You should use `flex` with C++ to write your program. Make sure that your program will accept the name of the input file. In case the program is run without a parameter or with more than one parameter your program must terminate immediately with an appropriate error message sent to the standard output.
- **Testing:** Your programs will be tested on an Ubuntu (22.04) machine. It is strongly advised that you set up a virtual machine if you do not have access to a machine with Ubuntu or any other Linux distro.
- **Debugging:** You are advised to use Eclipse IDE for C/C++ for debugging and tracing your program. This will come in handy, especially for the later stages of the project.
- **Evaluation:** The evaluation will be based on the correctness of the four output items listed below:
  - `sequences.txt` - 2 points
  - `nettest.txt` - 2 points
  - `identifiers.txt` - 2 points
  - `exceptions.txt` - 1 point
- **Submission:** You need to submit all relevant files you have implemented (.cpp, .h, .l, etc.) as well as a README file with instructions on how to run, a shell script and a makefile to run in a single .zip file named `<studentID.zip>`. If you are using Eclipse IDE, please submit your project folder as a .zip file named `<studentID.zip>`
- This project is expected to be an individual effort, but discussions are always welcome. Please do not hesitate to use the mail group for your questions.