The **main task** for this homework assignment is implementing an interpreter for the language *Lexaard* (language for exploring automata and related doodads), outlined below. The description covers the main points but is not exhaustive. Using discussions in class and on the class discussion forum for clarifications and further details is part of the required work, as is proper packaging and submission.

The language consists primarily of newline-terminated statements, with exceptions noted below. Each statement, and each line of the input, consists of whitespace-separated tokens, where whitespace is a nonempty sequence of any mix of spaces and tabs. Whitespace at the beginning and end of a line is permitted but not required. The first token of each statement is a *verb* that determines how the rest of the statement is interpreted. The language uses only an easily printable subset of the 7-bit ASCII character set (letters, digits, punctuation, space, tab, newline) and is case sensitive.

The language has three types of objects: symbols, strings, and automata. Symbols are unquoted strings (sequences of characters) that follow the typical rules for identifiers in a language such as Java or C. Examples: `x`, `m101`, `my_first_automaton`. Strings use the familiar quoted representation. Examples: `"x"`, `"am I a string?"`. Automata are represented as suggested by the following two equivalent representations of the automaton M_1 from page 36 of the textbook.[1] (For clarity, we use ␣ to denote a space character. There is a single newline character terminating each line, and a single blank line that terminates each representation.)

```
fsa
m101
␣␣␣␣␣0␣␣␣␣1
␣q1␣␣q1␣␣␣q2
*q2␣␣q3␣␣␣q2
␣q3␣␣q2␣␣␣q2

```

```
fsa
m101␣a␣rather␣pointless␣comment
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣0␣1
q1␣q1␣q2␣␣␣
*q2␣␣␣␣␣q3␣␣␣␣␣␣␣␣␣q2␣␣
q3␣q2␣␣␣q2

```

An FSA's representation always begins with the literal `fsa` followed by a newline. The first token on the next line (`m101` above) is a descriptive identifier associated with the automaton. Any further tokens on this line are ignored. The next line lists the alphabet of the automaton ($\{\mathtt{0}, \mathtt{1}\}$ above). These lines are followed by one line for each state of the automaton (three lines for states `q1`, `q2`, and `q3` above) in turn followed by a blank line. The state listed first (`q1`) is the start state of the automaton. An accepting state is denoted by adorning its line with a `*` prefixed to the state's name in the leftmost column. The representation suggested above is intentionally very similar to the usual tabular description of an automaton's state-transition table, such as the one on page 36 of the textbook. For example, $\delta(\mathtt{q3}, \mathtt{1}) = \mathtt{q2}$ above. The intuitive formatting illustrated on the left is *required in the output*, but *not required in the input.*

The interpreter must produce output exactly when and as described below for each statement. In particular, it must not produce extraneous output such as prompts and informative

[1] Michael Sipser, *Introduction to the Theory of Computation*, 3rd edition (Cengage Learning, 2013).

feedback unless noted below. The descriptions use `typewriter font` for literal text and *italic font* for meta-variables.

`quit` Terminate the interpreter gracefully (even if there is additional data on standard input). The end of standard input is treated as an implicit quit statement.

`print` ***x*** Print the external representation of the object named *x*. It is not an error if *x* is undefined; print nothing in this case. Automata should be printed in the well-formatted manner illustrated by the first depiction `m101` earlier.

`define` ***x v*** Define the name *x* to be the object represented by *v*.

`run` ***x i*** Run the automaton named *x* on the input string literal *i*. It is an error if *x* is not defined to be an automaton. The output is a single line containing `accept` or `reject` depending on whether the automaton accepts or rejects the input.

`run` ***x n*** As above, except *n* is the name of a previously defined string that is used as input to the automaton.

Blank lines, i.e., lines composed of only whitespace, are ignored, except when they are used in representations of objects. For this submission, you may assume that all test input will be valid; however, you are encouraged to implement at least rudimentary error checking.

```
define␣x␣"01011"
print␣x
define␣x␣"1101011"
print␣x
define␣m1␣fsa
m1
␣␣␣␣␣0␣␣␣1
␣q1␣␣q1␣␣q2
*q2␣␣q1␣␣q2

print␣m1
run␣m1␣"000101010010"
run␣m1␣"0001010100101"
run␣m1␣"0001010100100"
run␣m1␣x
quit
```

```
01011
1101011
m1
␣␣␣␣0␣␣1
␣q1␣q1␣q2
*q2␣q1␣q2

reject
accept
reject
accept
```

Figure 1: Sample input (left) and output (right).

The **submission** consists of an single electronic package that contains the **source** code, following the submission procedure described in class and on the class discussion forum. *Using the* ***discussion forum*** *to clarify details of both the main program and the submission format and procedures is an important part of this homework.* Packaging and documentation of code are worth a very significant portion of the grade. Use the *gzipped tar* (strongly preferred) or *zip* formats to package your submission. Name the electronic submission using the template

`cos451-hw01-`*`lastname`*`-`*`firstname`*`-`*`pqrs`*`.tgz`

where *`lastname`* and *`firstname`* are replaced by the obvious and *`pqrs`* is replaced by a 4-*digit* string of your choosing. (Replace `.tgz` with `.zip` if you use zip instead of tar for packaging.) The submission should be designed so that the command

`tar zxf cos451-hw01-`*`lastname`*`-`*`firstname`*`-`*`pqrs`*`.tgz`

results in the creation of a directory `cos451-hw01-`*`lastname`*`-`*`firstname`*`-`*`pqrs`*. In that directory should be all the source code (organized in further sub-directories as needed) as well as a README file with the usual semantics. *Do not submit any kind of non-source files* (results of compilation, etc.). Running `make` in the above directory should result in the creation of an executable file called `lexaard` that implements the Lexaard interpreter described here.

The interpreter should read from *standard input* and write to *standard output* (and optionally *standard error*). Please be sure to understand what these terms mean (they do not mean "terminal") and to ensure that your programs do not make additional assumptions (such as interactive input/output at a terminal).

You are welcome to use any inanimate **resources** (e.g., books, Web sites, publicly available code) to help you with your work. However, *all such help must be clearly noted* in your submissions. Further, no matter what you use, *you must be able to explain, in detail, how it works.* (You may be called upon to explain your homework individually.) Refer to the class policy for details, and ask for clarifications if you are unsure if something is allowed.

The **README file** should be a *plain text* file (not PDF, .docx, etc.) that includes, at a minimum, the following:

- Class information (University of Maine, COS 451, Fall 2024).
- Author information (your name)
- Primary (`@maine.edu`) email address.
- Date in an unambiguous format
- A brief summary of the contents of the submitted package (file-wise).
- A brief description of what the submitted code does.
- Instructions for compiling the code. (Ideally, just typing 'make' should work, but any special requirements or wrinkles should be noted here.)
- Instructions for running and testing the code.
- Known bugs, limitations, etc.
- Any other information that will help someone understand the submitted material.