CS 453/553: (Optional) Assignment 3

Due: May 9, 2025, by 11:00 PM (Central)

Overview

For this **optional** assignment, you will write a program that can simulate single-tape Turing machines. Your grade for this assignment will be determined as follows:

- If you **do not submit anything** for this assignment, then you will receive a score proportional to your average grade for Assignments 1 and 2. (This will result in your overall assignment grade being determined from Assignments 1 and 2 only.)
- If you **do submit materials** for this assignment, then Assignment 3 will be factored into your overall assignment grade (in addition to Assignments 1 and 2).

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "Turing machine in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

Your program should be runnable from the command line, and it should be able to process command-line arguments to dictate behavior. The combinations of command-line arguments that your program should support are listed below:

- `--info <FILENAME>`
  Loads the **Turing machine** specified in the file named `<FILENAME>` and prints some basic information about it.
- `--run <FILENAME> <INPUT> <MOVELIMIT>`
  Loads the Turing machine specified in the file named `<FILENAME>`, simulates it with a move limit of `<MOVELIMIT>` steps on the given `<INPUT>` string, and reports the outcome of the simulation (`ACCEPT`, `REJECT`, or `LIMIT` if the simulation was inconclusive).
- `--language <FILENAME> <LENGTHLIMIT> <MOVELIMIT>`
  Loads the Turing machine specified in the file named `<FILENAME>`, simulates it with a move limit of `<MOVELIMIT>` on each possible string of length at most `<LENGTHLIMIT>`, and prints all strings with an outcome of `ACCEPT`, followed by any strings with an outcome of `LIMIT` in case the associated simulation was inconclusive.

Note that **exactly one** of the above argument combinations will be given to your program (e.g., your program will **not** be run with both the `--info` and `--run` flags).

Filenames may include path information, so do not assume that the files are located in the same directory as your source code. You can assume that your program will only be run with valid combinations of arguments (so you do not need to include error checking, though it may be helpful for your own testing).

More details for these commands are provided later in this document, along with output requirements and examples.

Several example input files of Turing machines are provided in the `a03-data.zip` archive on Canvas. **Some** of these input files use a particular naming convention to make it easier (for humans) to identify the associated automaton and its properties. **DO NOT ASSUME** that every file will follow this or a similar naming convention! (You **can assume** that your program will be run with valid files though.)

File Format for Turing Machines

A Turing machine (TM) will be specified in a text file that stores the transition table in a **flexible format** along with markings for the initial state, accept state, and reject state. An example is shown below:

```
# TM for language { w#w | w \in {0,1}^n }, Sipser TM M1.
# Below is a full transition table version of this TM.
      3 | 0        1        #        x        _
->   q1 | (q2,x,R) (q3,x,R) (q8,#,R) -        -
     q2 | (q2,0,R) (q2,1,R) (q4,#,R) -        -
     q3 | (q3,0,R) (q3,1,R) (q5,#,R) -        -
     q4 | (q6,x,L) -        -        (q4,x,R) -
     q5 | -        (q6,x,L) -        (q5,x,R) -
     q6 | (q6,0,L) (q6,1,L) (q7,#,L) (q6,x,L) -
     q7 | (q7,0,L) (q7,1,L) -        (q1,x,R) -
     q8 | -        -        -        (q8,x,R) (qA,_,R)
   * qA | -        -        -        -        -
   ! qR | -        -        -        -        -
```

Lines that begin with a `#` symbol are comment lines and should be ignored (as should lines that are empty). The first non-empty, non-comment line specifies the "alphabet header" of the transition table. The first non-whitespace character on this line is an **integer**, which we'll denote as k here, that indicates the number of alphabet symbols that belong to the **input alphabet**. This integer is followed by one or more spaces, the `|` symbol, and then one or more additional spaces. After this follows a whitespace-separated list of the symbols in the machine's **tape alphabet**, the first k of which also belong to the machine's **input alphabet**. Each input alphabet symbol is a **single unique letter, digit, or special symbol** from the categories below:

- Uppercase letters: `A`, `B`, ..., `Z`
- Lowercase letters: `a`, `b`, ..., `z`
- Digits: `0`, `1`, ..., `9`
- Special symbols: `#`, `$`, `*`, `.`, `:`, `[`, `]`

Each tape alphabet symbol must be must be one of the following:

- A symbol chosen from one of the input symbol categories;
- The special **blank symbol** represented by `_` (underscore);
- A sequence of one or more symbols from the input symbol categories and/or the underscore character surrounded by angle brackets; e.g., `<A1>`, `<foo.bar>`, `<a_long_symbol>`, `<foo:z>`.

Each input and/or tape symbol **must be unique**, and the last tape symbol in the alphabet header will be `_` which represents the blank symbol.

The following symbols are **not allowed** in either alphabet: `{`, `}`, `(`, `)`, `<`, `>`, `|`, `,`, `-`, `=`.

The remaining non-empty, non-comment lines of the file specify the states and their associated transitions, and they have the form shown below:

```
<MARKERS> stateName | entry1 entry2 ...
```

The `<MARKERS>` block may include the following:

- `->` to indicate that the state is the **initial state**

- `*` to indicate that the state is the **accept state**
- `!` to indicate that the state is the **reject state**

All markers must be separated by whitespace, and there must be at least one space between any markers and the state's name itself. The initial state can also be the accept state or the reject state, but no state can be both the accept state and the reject state. **No specific ordering** should be assumed for the lines specifying states and their associated transitions (e.g., do not assume that the first state specified will always be the initial state, nor that the last state will always be the reject state!).

A state's name is a **unique** sequence of characters consisting of letters (`A`, ..., `Z`, `a`, ..., `z`), numbers (`0`, `1`, ..., `9`), and/or any of the following symbols: `.` (period), `:` (colon), `[`, `]`, `_` (underscore). Whitespace and other characters **not allowed**; in particular, state names **cannot include** any of the symbols that are **not allowed** as alphabet symbols either.

Following the state's name is the `|` symbol (with at least one space separating these pieces). Following the `|` symbol is a list of **entries**, where each entry **must be** separated from the `|` symbol and/or any adjacent entries by **at least one whitespace character**. **No whitespace** is permitted within any entry. The **default format** for entries is shown below on lines 2 and 3:

```
      3 | 0        1        #        x        _
->   q1 | (q2,x,R) (q3,x,R) (q8,#,R) -        -
     q2 | (q2,0,R) (q2,1,R) (q4,#,R) -        -
```

Entries here come in one of the following forms:

- `(q,w,D)` where `q` is a state name, `w` is a tape symbol, and `D` is a direction that is either `L` or `R`
- `-` which indicates a transition to the reject state

Entries are ordered to match the alphabet header ordering. In the example above for state `q1`:

- The first entry `(q2,x,R)` indicates that when the TM is in state `q1` and reading a `0` symbol, it should write a `x` symbol, move the tape head right (`R`), and update its state to `q2`.
- The second entry `(q3,x,R)` indicates that when the TM is in state `q1` and reading a `1` symbol, it should write a `x` symbol, move the tape head right (`R`), and update its state to `q3`.
- The fourth entry `-` indicates that when the TM is in state `q1` and reading a `x` symbol, it should write nothing (equivalently, write a `x`), move the tape head right, and transition to the reject state (where the TM halts).

There are some additional **conveniences** that may be used in the entries themselves:

- An entry of the form `(,w,D)` that is missing a state name should be interpreted as a transition back to the state named at the beginning of the line.
- An entry of the form `(q,,D)` that is missing a tape write symbol should be interpreted as a transition that writes the same tape symbol that it reads (so no change to the tape contents).
- An entry of the form `(,,D)` allows for both of the above implicit specifications.

As a concrete example, the transitions for state `q2` in the above could also be expressed as follows (note that the first entry could be further simplified as well):

```
      3 | 0        1        #        x        _
->   q1 | (q2,x,R) (q3,x,R) (q8,#,R) -        -
     q2 | (,0,R)   (,,R)    (q4,,R)  -        -
```

Adjacency List Entry Format

To provide **further convenience** in specifying Turing machine behavior (though this is perhaps an **inconvenience** for the programmer who writes the TM simulator itself!), the entries associated with a state may alternately be specified in a more compact **adjacency list format** using key-value pairs. To illustrate this, we'll return to the example shown earlier:

```
     3 | 0        1        #        x        _
->   q1 | (q2,x,R) (q3,x,R) (q8,#,R) -        -
     q2 | (q2,0,R) (q2,1,R) (q4,#,R) -        -
```

State `q1` doesn't have explicit transitions when reading `x` and `_`, but the **default entry format** requires us to include `-` for these entries to ensure that the number of entries matches the number of symbols in the tape alphabet.

The **adjacency list format** for entries associated with state `q1` is shown below:

```
     3 | 0        1        #        x        _
->   q1 | 0=(q2,x,R) 1=(q3,x,R) #=(q8,#,R)
     q2 | (q2,0,R) (q2,1,R) (q4,#,R) -        -
```

Here each entry has the form `r=(q,w,D)`, where `r` is the tape symbol that is being read, `q` is the next state, `w` is the tape symbol that is to be written, and `D` is the direction that the tape head should be moved after writing. In this format, we omit entries that correspond to a transition to the reject state. Additionally, entries in the **adjacency list format** can be listed in **any order** (that is, the first entry doesn't need to correspond to the first tape symbol in the alphabet header).

We can add **further convenience** by allowing multiple tape symbols to be specified prior to the `=` symbol. For example, an entry of the form `01=(q,w,D)` indicates that the transition `(q,w,D)` should be followed when reading either a `0` or a `1`. (This exploits the fact that tape symbols must be either a single character or be a sequence of characters enclosed in `< >`, so we can split a string of tape symbols into pieces without needing any explicit separators!)

We can also combine the earlier convenience of omitting a tape write symbol in the entry with multiple tape read symbols. For example, `01=(q,,D)` indicates the transition that should be followed when reading either a `0` or a `1`, with the additional requirement that the symbol that is read should also be what gets written to the tape.

With the adjacency list format and other conveniences, we could specify the Turing machine from the earlier example as follows:

```
# TM for language { w#w | w \in {0,1}^n }, Sipser TM M1 (condensed).
    3 | 0 1 # x _
-> q1 | 0=(q2,x,R) 1=(q3,x,R) #=(q8,,R)
   q2 | 01=(,,R) #=(q4,,R)
   q3 | 01=(,,R) #=(q5,,R)
   q4 | x=(,,R) 0=(q6,x,L)
   q5 | x=(,,R) 1=(q6,x,L)
   q6 | 01x=(,,L) #=(q7,,L)
   q7 | 01=(,,L) x=(q1,,R)
   q8 | x=(,,R) _=(qA,,R)
 * qA |
 ! qR |
```

Your program must support both the **default entry format** and the **adjacency list format**. Any **single state transition line** within the Turing machine description file must follow **exactly one format**, but any file may use **both formats** for different lines.

The above examples have been formatted for easy human readability, but **any** file that meets the specifications outlined above is **valid**. In particular, you should **not assume** that contents will be aligned in a particular way.

Output Details

The behavior and output associated with each of the program commands is specified below, and several examples are given for each command. While your output does not need to match mine exactly, it should be fairly similar aside from potential minor differences in spacing. In particular, you should strive to ensure that the printed description of a Turing machine is reasonably well-formatted for human readability, at least for smaller machines. Printing poorly formatted descriptions may result in a minor reduction of points for the assignment grade.

Lastly, **use space characters** for output alignment **instead of tabs** (`\t`) (see here for a discussion of why). *Hint:* `System.out.printf` and/or `System.out.format` are **very helpful** for this purpose! For example, `System.out.format("%8s", str);` and/or `System.out.format("%-8s", str);` will print strings with additional leading or trailing spaces to ensure a length of at least `8` characters; with a somewhat kludgy adjustment, you can specify this "length" in a variable!

Additional Details: The `--info` Command

The `--info` command should print information regarding a TM in the format shown below:

```
shell$ java Driver --info data/tm-even-ones.txt
States: {q0,q1,qA,qR}
Input alphabet: {0,1}
Tape alphabet: {0,1,_}
Transition Table:
      2 | 0        1        _
->   q0 | (q0,0,R) (q1,1,R) (qA,_,R)
     q1 | (q1,0,R) (q0,1,R) (qR,_,R)
   * qA | -        -        -
   ! qR | -        -        -
Initial State: q0
 Accept State: qA
 Reject State: qR
```

Note that the printed information should use the **default entry format** and it **should include** state names and tape write symbols in every non-`-` entry even if the input file uses an alternate format:

```
shell$ java Driver --info data/sipser-tm-M1.txt
States: {q1,q2,q3,q4,q5,q6,q7,q8,qA,qR}
Input alphabet: {0,1,#}
Tape alphabet: {0,1,#,x,_}
Transition Table:
      3 | 0        1        #        x        _
->   q1 | (q2,x,R) (q3,x,R) (q8,#,R) -        -
     q2 | (q2,0,R) (q2,1,R) (q4,#,R) -        -
     q3 | (q3,0,R) (q3,1,R) (q5,#,R) -        -
     q4 | (q6,x,L) -        -        (q4,x,R) -
     q5 | -        (q6,x,L) -        (q5,x,R) -
     q6 | (q6,0,L) (q6,1,L) (q7,#,L) (q6,x,L) -
     q7 | (q7,0,L) (q7,1,L) -        (q1,x,R) -
     q8 | -        -        -        (q8,x,R) (qA,_,R)
   * qA | -        -        -        -        -
   ! qR | -        -        -        -        -
Initial State: q1
```

```
 Accept State: qA
 Reject State: qR
```

The below example uses some long tape symbols which are enclosed in < >:

```
shell$ $ java Driver --info data/tm-monus-V2.txt
States: {check,check0,check1,resetLeft,init,lookLeft,findR1,eraseL1,erase0s,add0,eraseAll,
    accept,reject}
Input alphabet: {0,1}
Tape alphabet: {0,1,<s0>,<s1>,_}
Transition Table:
             2 | 0               1               <s0>       <s1>       _
->       check | (check1,<s0>,R) (check0,<s1>,R) -          -          -
        check0 | (check1,0,R)    (check0,1,R)    -          -          -
        check1 | -               (check1,1,R)    -          -          (resetLeft,_,L)
     resetLeft | (resetLeft,0,L) (resetLeft,1,L) (init,0,L) (init,1,L) -
          init | (lookLeft,0,L)  (init,1,R)      -          -          -
      lookLeft | (eraseAll,0,R)  (findR1,1,R)    -          -          -
        findR1 | (findR1,0,R)    (eraseL1,0,L)   -          -          (erase0s,_,L)
       eraseL1 | (eraseL1,0,L)   (lookLeft,0,L)  -          -          -
       erase0s | (erase0s,_,L)   (add0,1,R)      -          -          -
          add0 | -               -               -          -          (accept,0,R)
      eraseAll | (eraseAll,_,R)  (eraseAll,_,R)  -          -          (accept,_,R)
   *     accept | -               -               -          -          -
   !     reject | -               -               -          -          -
Initial State: check
 Accept State: accept
 Reject State: reject
```

Additional Details: The `--run` Command

The `--run` command should run the Turing machine (i.e., by simulating its behavior) on the given input string, print the **configurations** of the machine during its computation, and indicate whether it accepts, rejects, or hits the move limit:

```
shell$ java Driver --run data/tm-even-ones.txt 001010 8
Running on input [001010] with limit 8:
Config: <q0>001010
Config: 0<q0>01010
Config: 00<q0>1010
Config: 001<q1>010
Config: 0010<q1>10
Config: 00101<q0>0
Config: 001010<q0>_
Config: 001010_<qA>_
ACCEPT
```

In the above, each configuration shows the tape contents as well as the current state with the state name enclosed in `<` `>` and printed immediately to the left of the tape symbol that is under the tape head. Trailing blanks are only displayed if the tape head has moved over them.

In the below example, the machine is allowed to make 5 moves; after the fifth move, it enters the reject state and halts (so it is not move-limited):

```
shell$ java Driver --run data/tm-even-ones.txt 1011 5
Running on input [1011] with limit 5:
Config: <q0>1011
Config: 1<q1>011
Config: 10<q1>11
Config: 101<q0>1
Config: 1011<q1>_
Config: 1011_<qR>_
REJECT
```

With only 4 moves, the machine is unable to decide:

```
shell$ java Driver --run data/tm-even-ones.txt 1011 4
Running on input [1011] with limit 4:
Config: <q0>1011
Config: 1<q1>011
Config: 10<q1>11
Config: 101<q0>1
Config: 1011<q1>_
LIMIT
```

The below example shows how a Turing machine can be used to compute the monus (proper subtraction) operation, with the answer written on the tape at the end:

```
shell$ java Driver --run data/tm-monus-V1.txt 11101 20
Running on input [11101] with limit 20:
Config: <q0>11101
Config: 1<q0>1101
Config: 11<q0>101
```

```
Config: 111<q0>01
Config: 11<q1>101
Config: 111<q2>01
Config: 1110<q2>1
Config: 111<q3>00
Config: 11<q3>100
Config: 1<q1>1000
Config: 11<q2>000
Config: 110<q2>00
Config: 1100<q2>0
Config: 11000<q2>_
Config: 1100<q4>0_
Config: 110<q4>0__
Config: 11<q4>0___
Config: 1<q4>1____
Config: 11<q5>____
Config: 110<qA>___
ACCEPT
```

You may assume that the input string will be in Σ^*, where Σ is the input alphabet of the Turing machine. To run the machine on the empty string, use a pair of double-quotes at the command line. This and additional fun corner case behavior is illustrated below:

```
shell$ java Driver --run data/tm-all-but-epsilon.txt "" 1
Running on input [] with limit 1:
Config: <q>_
Config: _<r>_
REJECT
```

```
shell$ java Driver --run data/tm-all-but-epsilon.txt "" 0
Running on input [] with limit 0:
Config: <q>_
LIMIT
```

```
shell$ java Driver --run data/tm-empty.txt 010 3
Running on input [010] with limit 3:
Config: <q>010
REJECT
```

```
shell$ java Driver --run data/tm-universal.txt 110 0
Running on input [110] with limit 0:
Config: <q>110
ACCEPT
```

```
shell$ java Driver --run data/tm-all-but-epsilon.txt 10 0
Running on input [10] with limit 0:
Config: <q>10
LIMIT
```

```
shell$ java Driver --run data/tm-all-but-epsilon.txt 10 1
Running on input [10] with limit 1:
Config: <q>10
Config: 1<a>0
ACCEPT
```

Additional Details: The `--language` Command

The `--language` command should print all strings in the **language** of the Turing machine, up to the given length limit, say k. To do this, you'll need to generate all possible strings from $\Sigma^0, \Sigma^1, \ldots, \Sigma^k$ and then simulate the TM on each of them (up to the given move limit) to determine whether the TM accepts or rejects (or is unable to tell within the given limit). The strings that are accepted should be printed in ascending order based on length, with lexicographic order used across all strings of the same length. Following the accepted strings, your program should print a list of all strings for which the simulation was inconclusive (in ascending order based on length, then lexicographically).

```
shell$ java Driver --language data/sipser-tm-M1.txt 5 15
L(M) = {
  #,
  0#0,
  1#1,
  ...
}
Undetermined strings (due to limit):
  00#00
  01#01
  10#10
  11#11
```

```
shell$ java Driver --language data/sipser-tm-M1.txt 5 18
L(M) = {
  #,
  0#0,
  1#1,
  00#00,
  01#01,
  10#10,
  11#11,
  ...
}
```

```
shell$ java Driver --language data/tm-L010.txt 10 50
L(M) = {
  "",
  010,
  001100,
  000111000,
  ...
}
```

```
shell$ java Driver --language data/tm-monus-V2.txt 4 20
L(M) = {
  0,
  01,
  10,
  011,
  101,
  110,
```

```
  0111,
  1110,
  ...
}
Undetermined strings (due to limit):
  1011
  1101
```

```
shell$ java Driver --language data/tm-universal.txt 2 0
L(M) = {
  "",
  0,
  1,
  00,
  01,
  10,
  11,
  ...
}
```

Hints

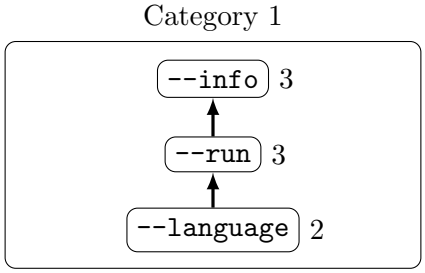
- Data structures are your friends! In particular, **maps** can be very helpful here.
- Use a `Scanner` for reading from the files! A good pattern is to use a primary `Scanner` object to read the file, one line at a time, and then pass each line to the constructor of a secondary `Scanner` object to parse that line's contents.
- Use `System.out.format` and/or `System.out.printf` for controlling the width of printed strings.
- Test as you go! Use **incremental development**. Refactor as needed to improve readability. You will learn a lot by doing things the "wrong way" (or at least in a suboptimal way) the first time, so don't be afraid to redo some work to improve the code itself!
- Words of wisdom from a pivotal figure in computer science:

  > *"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."*
  > — *Donald Knuth, The Art of Computer Programming*

  In summary: "Make it work; make it good; make it fast." (In that order as time allows).

Grading Notes

At a high level, your assignment grade will depend on the functionality associated with the following command names:



Dependencies between components are indicated with arrows. **In order to earn points for a particular component, your program needs to be correct or nearly correct across all components on which that particular component depends**. So, for example, your program needs to be able to process any valid Turing machine description file including the **adjacency list entry format** (tested via the `--info` command) in order to earn any points for the `--run` command.

Given these dependencies, you should focus your efforts on implementing these components according to the **partial ordering** above! Additionally, The numbers next to each component are **approximate point values** and are provided to help you gauge where to allocate your efforts. I reserve the right to make minor adjustments to these point values during grading itself.