CSCE 355, Fall 2022
Programming Assignment

Due Monday November 21, 2022 at 11:59 pm EDT

For the programming portion of the course (15% of your total grade) you are to write a program to read in the description of a Turing machine (TM) and simulate it on various inputs. Your program will be invoked from the command line (the bash shell on one of the Linux machines in the department's Linux lab). Your program takes a single command-line argument: the name of a file containing the description of a TM M. It then reads (from standard input) a series of strings w, one per line, and for each such w, simulates the computation of M on input w, displaying the state (aka, configuration, aka, instantaneous description) of the computation at each step. Your program ends when encountering end-of-file on an empty line.

Formats for describing TMs, input strings, and machine configurations to output, as well as rules for submitting, are given below. Your program will be run automatically via a script run on one of the Linux machines (e.g., `l-1d49-10.cse.sc.edu`), so it is important that you adhere to these specifications exactly and test your submission on one of those machines. You will lose points if your program fails to run correctly via the script.

1 Details

All I/O will be ASCII text. Output is written to standard output only; if you want to for your own sake, you may send error/debug messages to standard error, and we will ignore them. (These streams may have different names depending on which programming environment you use.) You may write your program in any programming language you want, provided it is implemented on the Linux machines in the CSE Linux lab. We recommend Java or C or C++ or Perl or Python or Ruby or ML or Haskell or Prolog or Scheme or .... Your program (after compiling, if necessary) should be a stand-alone executable that can be run directly from the Linux bash shell, requiring no special user interface to execute (e.g., Eclipse).[1] More about this below.

1.1 An example

Suppose you write your program in C or C++, and the executable is named `tmsim`. You may invoke your program thus ($ is the bash shell prompt):

```
$ ./tmsim some-TM-description-file.txt
```

Here, the argument is the description of a TM. If all is well, the program waits for keyboard input, which might be

[1] It's OK if we need to invoke the JVM by preceding your main class name with "java" on the command line, or the python interpreter by preceding your program with "python3."

```
ababc
101

62
^D
```

After you hit the `enter` key for each line, your program outputs the computation of M on that line: `ababc`, `101`, ϵ (the empty string), and `62`. When you enter Ctrl-D on an empty line, your program quits.

When the grading script runs your program, it will actually do something like this:

```
$ ./tmsim some-TM-description-file.txt < test-input-strings.txt > test-output.txt
```

The "`<`" in the shell command redirects standard input in your program to read from the file `test-input-strings.txt` instead of the keyboard. The "`>`" in the shell command redirects standard output in your program to write to the file `test-output.txt` instead of the screen.

The file `https://cse.sc.edu/~fenner/csce355/prog-proj3/test/TM_sample.txt` contains a description of the TM from Problem 7 on Homework 5, suitable for reading by your program. Another TM description file is `https://cse.sc.edu/~fenner/csce355/prog-proj3/test/TM_add.txt`. `TM_add.txt` describes a TM that performs binary addition. It takes as input a string of the form $w\#x$, where w and x are binary strings, and accepts, leaving the sum of the two numbers represented by w and x on the tape.

An executable solution that you can run on the Linux machines is at `https://cse.sc.edu/~fenner/csce355/prog-proj3/tmsim`. It only allows input strings of length ≤ 127 (longer strings are truncated), and you can assume the same limitation in your own program.

1.2 TM description file syntax

Your program will be tested on the files linked to above, as well as some others. You may of course generate your own files for testing purposes.

To understand the format of these files, we make some simplifying adjustments to the TM definition in the textbook:

- The state set Q will always be $\{0, 1, \ldots, n\}$ for some $n \leq 99$.
- The start state q_0 will always be 0.
- There will always be exactly one accepting state, namely the last state n as above (i.e., $F = \{n\}$). No transitions out of the accepting state will ever be defined.
- The tape alphabet Γ will only consist of plain ASCII characters (in the range 1 to 127) and will never include any characters that do not print normally with width 1 (so no newlines, tabs, form feeds, back tabs, control characters, nulls, etc.). The tape alphabet will also never include parentheses.
- The blank symbol will always be the underscore character (`_`) and will always be explicitly included in the tape alphabet.
- No explicit input alphabet Σ will be given. However, input strings will never include the blank symbol or any characters outside Γ.

The TM description file is a plain ASCII text file. The first line is always

```
Number of nonhalting states: ##
```

where `##` is a number in the range 1 to 99. This is the number n described above and also represents the unique accepting state. The second line is always

```
Alphabet: #######
```

where `#######` is a string of ASCII characters indicating the tape alphabet (without duplicates). Note that these characters need not appear in order of increasing ASCII value.

The rest of the file, starting with the third line, gives all the transitions of the transition function δ, one per line, ordered by input state. Each transition is of the form, "q a / r b D", which means that $\delta(q, a) = (r, b, D)$. Here, q and a are the input state (an unsigned integer ≤ 99) and input symbol, respectively, r is the output state, b is the output symbol (to replace a on the tape), and D is the head movement direction—either "`<`" for moving left or "`>`" for moving right. These symbols are all separated by single space characters. The transition may be preceded by arbitrary whitespace (space characters and or tabs) at the start of the line, which should be ignored. The rest of the line following the transition may consist of arbitrary text, which should also be ignored. This is to allow for an arbitrary 1-line comment after a transition.

All TM description files we use to test your program will adhere to these rules, so you need not check the file for syntax errors. (You may wish to error-check anyway to expose bugs in your own program.)

1.3 Output format

Given a TM M and input string w, the successive configurations of M running on input w should be displayed to standard output one per line, starting with the initial configuration and ending with the halting configuration (if there is one; however, we will only test your code with halting computations). The first two characters give the current state (a decimal number right-justified in a field of width 2), followed by a colon (`:`), followed by a string giving the current contents of the tape. A contiguous segment of the tape should be displayed that includes all nonblank cells and the cell currently being scanned, but nothing more. The scanned symbol should be surrounded by parentheses. On the line following the halting configuration, output "`accept`" or "`reject`," depending on whether M accepts or rejects w.

For example, if current state is 6, the current tape has the string 000110 surrounded by all blanks, and the head is currently scanning the leftmost 1, then you should output

```
 6:000(1)10
```

(Note the initial space character.) For another example, if the current state is 17, the current tape has the string *abc* surrounded by all blanks, and the head is currently scanning the third blank to the right of the *c*, then you should output

```
17:abc__(_)
```

Here is an actual running of the program `tmsim` linked to above, using the TM description file `TM_add.txt` and running on input string `11#1` (input at the keyboard):

```
$ ./tmsim TM_add.txt
11#1
 0:(1)1#1
 0:1(1)#1
 0:11(#)1
 1:1(1)#1
 2:1b(#)1
 2:1b#(1)
 2:1b#1(_)
 3:1b#(1)
 5:1b(#)
 5:1(b)#
 6:(1)0#
 7:b(0)#
 7:b0(#)
 7:b0#(_)
 8:b0(#)
12:b(0)
12:(b)0
13:(_)00
14:(_)100
15:(1)00
16:(_)100
accept
```

At this point, the program waits for another input string. I hit Ctrl-D, so the program ends.

As mentioned, you may assume that when we test your program for grading, the inputs will adhere to their respective formats, i.e., you won't need to error-check the input, either TM description or input strings. You can send anything you want to standard error; the grading program will ignore it.[2]

Your code should be economically written, well-structured, and well-commented, following the common stylistic guidelines of the programming language you use. The code should also be reasonably efficient, but this is a secondary requirement. If your code runs correctly and within the allotted time (11 seconds), we won't really look too closely at the source code. If it does not run correctly or times out, however, the source code style might make some difference.

2 Notes and Hints

You may use any internal data structures you like, provided they are reasonably efficient. You may assume the bounds on the number of states and the lengths of the input strings given above, but you should not assume any bound on the length of tape you need to display at any given time step

[2]We call the three I/O streams open by default on GNU/Linux programs *standard input* (buffered keyboard input by default), *standard output* (unbuffered screen output by default), and *standard error* (unbuffered output sent to the screen by default, even if standard input and output are redirected by the system). Some programming environments may use different names for these streams, e.g., C programs using stdio.h for high-level I/O call these stdin, stdout, and stderr, respectively; C++ programs typically use cin, cout, and cerr for the same purpose.

(except that it can reasonably be stored in main memory). Since the portion of tape you display may expand or contract at either end, I don't recommend maintaining the tape in the initial part of an array. Instead, you might consider using a circular (array) buffer, or a linked list of some sort. If you use a circular buffer, you may need to resize it (if the tape portion to be displayed gets too long).

We will only test your program with TMs that halt on all inputs (no infinite loops!)

Your output should *exactly match* that of the `tmsim` program. The rules are set up so there is no leeway possible in the output. We will compare your output with the solution's output using the Linux `diff` utility. To get full credit, running `diff` should produce no output.

2.1 A debugging pitfall

I don't use debugging utilities; I've never devoted the time to actually use them (maybe I'm just old-school). Instead, if something goes horribly wrong, e.g., a segmentation fault, I start inserting print statements into to my code to locate the exact place where the program crashed. These print statements should always be sent to standard error, not standard output. The reason is that, if standard output is redirected to a file by the shell, then it is *buffered*, meaning that it isn't written right away—only when the buffer gets full or the program exits normally. If your program crashes before the output buffer is flushed, then there may be lots of output that never made it into the file, making you think the crash occurred earlier than it actually did. Standard error, on the other hand, is always *unbuffered*; it is sent immediately to the screen (or redirected to a file) when it is generated and will never be left sitting in a buffer.

2.2 A Windows vs. Linux pitfall

Windows-based text files end each line with the two-character sequence `\r\n` (carriage return, newline), and GNU/linux/Mac OSX and similar systems expect only an `\n` ending each line. We strongly recommend against doing your development on a Windows box, but if you absolutely must, be aware that your code may not work properly with the test script if it is copied over without newline conversion. (This is the cause of many mysterious failures when running the test script.) Our Linux boxes support the `scc` command. Running

```
scc my_text_file.txt
```

converts all `\r\n` sequences to `\n` in `my_text_file.txt`. (NOTE, however, that this command alters the contents of the file and does *not* produce a backup copy!)

3 Testing and Grading

As we mentioned, your project will be graded automatically. To grade your project, we will use the script `project-self-test.pl` (written in Perl) and test files in a test suite directory to test and grade your project. *All these files will be available to you soon from the project homepage,* so that you can see how your code will be tested and even run the test script yourself to see in advance how well you do. Just to be perfectly clear: we will grade your project by running the script `project-self-test.pl` on it with owner privileges using one of the Linux lab machines. We will not run your code personally. The comments produced by that script will determine your grade. This means that you will not get credit for attempting to do something. You will only get

credit for what actually works, as determined by the `project-self-test.pl` script run on a CSE Linux Lab machine.

4 Submission

Submission will be via CSE Departmental Dropbox (Moodle). Upload a single file, either a .zip file or a .tar.gz file, containing

1. all your source code files, which should all be in the same directory, i.e., no subdirectories (and no automatically generated files, please),
2. an optional file `readme.txt` with anything you want to tell us (we will read this with our own eyes), and
3. a "build-run" text file giving Linux (bash) shell commands to compile and/or run your program. Don't include the command line argument (e.g., `TM_sample.txt`); we will supply those separately when we run your program. This file should be named `build-run.txt` and placed in the same directory as your other source files. See below for the contents of this file.

IMPORTANT NOTE: You *must* use either the ZIP format (file extension .zip) or the GZIPPED TAR format (file extension .tar.gz) for your submission file. Your file will be de-archived either with `unzip` or with `gunzip; tar -xf`, depending on your file name's extension. Do not use any other archive format, particularly the RAR format, which is proprietary to Windows (I personally do not have Windows on any machine I use). If you deviate from the allowed formats, you risk getting zero credit for the entire assignment. Keep in mind that Linux file names are case-sensitive.

4.1 Examples of build-run files

Suppose you implement your program in Java, and your main class is called `MyTMSimulator`. Then your `build-run.txt` file would look like this:

```
# Lines like these are comments and will be ignored
Build:
  javac MyTMSimulator.java
Run:
  java MyTMSimulator
# Don't include command line arguments to the run command!
# The indenting is optional.
```

For another example, suppose you implement your program in C as a single compilation unit called `my_TM_simulator.c`. Then your `build-run.txt` file might look something like this:

```
Build:
  gcc my_TM_simulator.c
  mv a.out my_TM_simulator
Run:
  ./my_TM_simulator
# Again, no command line argument, please.  It will be supplied automatically.
```

Note that you can have any number of build commands, and they will be executed in order (in the directory containing your source files) before the run command. Always give the Build commands first before the Run command.

Suppose instead that you have several compilation units for your programs, including shared code, and a complicated build procedure, but you have a single Makefile controlling it all, capable of producing an executable called `tmsim` (this is what I have for my solution). Then the `build-run.txt` file can just look something like this:

```
Build:
  make -B
Run:
  ./tmsim
```

(Use the `-B` option or `--always-make` option with `make`; it will build your entire program from source regardless of any intermediate files.)

As a final example, suppose you implement your program in Python, which is a scripting language that can be run directly without a compilation step. Then your `simulate.txt` file might look like this:

```
Build:
Run:
  python my_TM_simulator.py
# You still need to say "Build:" even though there are no build commands.
```

Finally, be sure your CSE Dropbox account exists and is accessible. Do this early on to avoid last-minute glitches.

5 Do Your Own Work

The code you write and submit must be yours alone. You may discuss the homework with others at the conceptual level (see the next paragraph), but you may not copy code directly from any other source, even if you modify it afterwards. Likewise, you must take all reasonable precautions not to let your code be copied by anyone else, either in this class or in future classes. This includes uploading or developing your code on a web platform—such as SourceForge or GitHub—in a way that can be seen by others. Violating this policy constitutes a violation of the Carolina Honor Code, and will have serious consequences, including, but not limited to, failure of the course.

Discussing the project with others in the class is allowed (even encouraged), but you must include in your `readme.txt` file the names of those with whom you discussed the project.

If you have any questions about what this policy means, please review the relevant section of the course syllabus or ask me.