Course Project 3
Replacement and Turing machines

CSE 30151 Spring 2020

Version of 2020/01/07
Due 2020/03/03 5:00pm

In this project, you'll reimplement a fragment of another Unix tool, `sed`, whose most common function is using regular expressions to make changes to a file. For example, the command `sed -E -e 's/(a*)(b*)/\1#\2/'` changes `aaabbb` to `aaa#bbb`. The `\1` means "copy what matched the first pair of parentheses." You'll implement `sed`'s `s` command as well as its `b` command for conditional branching. Then, you'll show that this fragment is Turing-complete by implementing a translator from Turing machines to `sed` scripts.

You will need a correct solution for CP2 to complete this project. If your CP2 doesn't work correctly (or you just weren't happy with it), you may use the official solution or another team's solution, as long as you properly cite your source.

Getting started

To make sure your repository is up to date, please have one team member run the commands

```
git pull https://github.com/ND-CSE-30151-SP18/theory-project-skeleton
git push
```

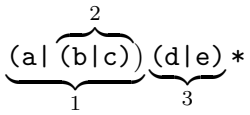
and then other team members should run `git pull`. The project repository should then include the following files (among others):

```
bin/
  re_groups
  msed
  run_tm
examples/
  sipser-m1.tm
  sipser-m2.tm
tests/
  test-cp3.sh
cp3/
```

Please place the programs that you write into the `cp3/` subdirectory.

1 Groups

The subexpression enclosed by a matching pair of parentheses is called a *group*. The groups are numbered starting from 1, based on the order of their *left* parentheses. For example:



Many regular expression libraries let you retrieve the contents of a group after a successful match. For example, when the above regular expression matches string `ade`, group 1 has contents `a`.

It's possible for a group to match no substring or more than one substring. In the above example, group 2 matches no substring, whereas group 3 matches two substrings (`d` and `e`). For this project, you can treat a group that matches no substring as containing ε. A group matching more than one substring is treated as containing the *last* substring (in this case, `e`).

Extend your regular expression matcher to capture contents of groups, and write a program to demonstrate it:

`re_groups regexp string`

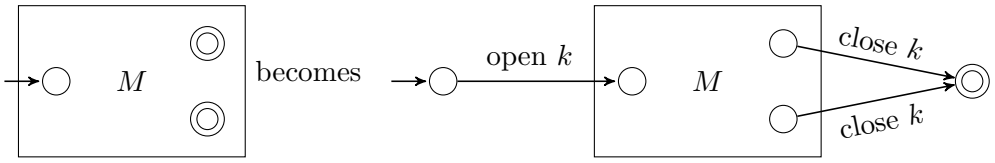
- `regexp`: regular expression
- `string`: input string
- Output:
  - If `regexp` matches `string`, prints `accept` followed by any matching groups, one per line (see below for format)
  - Otherwise, prints `reject`

For example,

```
$ re_groups '(a|(b|c))(d|e)*' 'ade'
accept
1:a
3:e
```

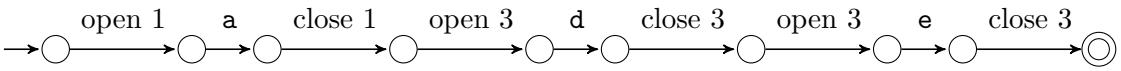
Print just those groups that matched a substring, in numerical order. If there's more than one way for the groups to match, you can choose any of them (but this doesn't happen in any of the supplied tests). Run `test-cp3.sh` to check your implementation.

There's more than one way to do this, but here's our suggestion. In CP2, parentheses didn't trigger any NFA operation. But now, we can make the parentheses around group k cause the following operation to be performed:



where "open k" and "close k" are special transitions that don't read any input. As far as the matcher is concerned, they behave just like epsilon transitions.

The NFA matcher you wrote in CP1 returns an accepting path (if there is one). Now if the accepting path contains open and close transitions, you can use them to reconstruct the contents of each group. For example, if the path is



then by walking this path from left to right, you can determine that group 1 is `a` and group 3 is `e`.

2 A fragment of `sed`

Write a program called `msed` (for mini-`sed`):

```
msed -f command_file              [file ...]
msed -e command [-e command ...] [file ...]
```

- `-f command_file` specifies a file to read commands from
- `-e command` specifies a command; can be used more than once
- `file ...` specifies what file(s) to read strings from; if none, then read from stdin

If called using the first form, it executes the commands from `command_file`, one per line. If called using the second form, it executes the commands given using the `-e` option. If you want, you can allow multiple uses of the `-f` option and/or mixing the `-e` and `-f` options; the real `sed` allows this, but we won't check for it.

The commands are executed in order. There are three kinds of commands:

- The command `:label`, where `label` is any string not containing whitespace, doesn't do anything; it's just a target for the branch command.

- A branch command has the form `/regexp/blabel`. If the current string matches the `regexp`, it jumps to `label` (or quits if there is no such label). Note that unlike the real `sed`, regular expressions must match the entire line.
- A substitution command has the form `s/regexp/replacement/`. If the (entire) current string matches the `regexp`, it replaces the current string with `replacement`. The `replacement` can contain backreferences, which can have two forms:
  - `\`k: expands to the contents of group k, where $k \geq 1$.
  - `\g<`k`>`: expands to the contents of group k, where $k \geq 1$.

  In both forms, k cannot have leading zeros. Note that `\10` expands to the contents of group 10. If you want the contents of group 1 followed by the character `0`, then write `\g<1>0` instead.[1]

For example, the following script reverses strings over {`a`, `b`}:

```
s/((a|b)*)/^\1/
:loop
s/((a|b)*)^(a|b)((a|b)*)/\3\1^\4/
/(a|b)*^(a|b)(a|b)*/bloop
s/((a|b)*)^/\1/
```

Line 1 inserts a `^` marker. Line 3 moves the character after the marker to the beginning of the string. Line 4 checks whether there are any characters left; if so, it goes back to line 2. Finally, line 5 removes the marker.

Test your implementation by running `test-cp3.sh`. The test script also times your implementation to make sure that your modifications haven't changed its time complexity.

3 Turing machines to `sed`

Write a program called `tm_to_sed` that has the following usage:

`tm_to_sed tm_file`

- `tm_file`: specification of a Turing machine (see below)
- Output: an equivalent `msed` script (see below)

[1] Regular expression libraries differ wildly in how they handle double-digit groups; this (weird) syntax follows Python's.

Turing machine file format The `tm_file` should start with a header with six lines:

1. A whitespace-separated list of states, Q.
2. A whitespace-separated list of input symbols, Σ. It should be disjoint from Q, and it should not contain ␣ (blank). Each symbol must be a single character.
3. A whitespace-separated list of tape symbols, $\Gamma \supseteq \Sigma$. It should be disjoint from Q, and it should contain ␣ (blank). Each symbol must be a single character.
4. The start state, $q_0 \in Q$.
5. The accept state, $q_{\text{accept}} \in Q$.
6. The reject state, $q_{\text{reject}} \in Q$.

The header is followed by transitions, one per line. Each one consists of five whitespace-separated fields:

1. The state that the transition goes from.
2. The tape symbol that the transition reads.
3. The state that the transition goes to.
4. The tape symbol that the transition writes.
5. Either `L` or `R` to indicate the direction the head moves.

For every $q \in Q$ except q_{accept} or q_{reject} and for every $a \in \Sigma$, there should be at most one transition from q on symbol a. States q_{accept} and q_{reject} should have no outgoing transitions. See `sipser-m1.tm` and `sipser-m2.tm` for examples, which correspond to the machines in Sipser, page 174 and 173, respectively.

Operation The program should convert the Turing machine into an `msed` script. The resulting `msed` script should read lines from the input, and for each line, simulate the Turing machine on the input. If the machine accepts, the script should output `accept:` followed immediately (without whitespace) by the final contents of the tape. If the machine rejects, the script should output `reject`.

There's more than one way to do this. Our suggestion is to follow Sipser's encoding (p. 168–169) of Turing machine configurations as strings. You can encode the state in the string, like he does, or you could encode the states as labels in the `msed` script. In either case, you will need to use the `s` command to simulate the moves of the Turing machine.

Test your converter using `test-cp3.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, run `make -C cp3`, and then run `tests/test-cp3.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work, please push your repository to Github and then create a new release with tag version `cp3` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use tag version `cp3-1` for part 1, `cp3-2` for part 2, and so on.

Rubric

Part 1	
capturing and reconstructing groups	6
`re_groups`	3
Part 2 (`msed`)	
label/branch	3
substitution	6
time complexity	3
Part 3 (`tm_to_sed`)	
reading TM file	3
correct conversion	6
Total	30