Course Project 3
Replacement and Turing machines

CSE 30151 Spring 2024

Version of 2024-10-29
Due date 2024-04-06

In this project, you'll reimplement a fragment of another Unix tool, `sed`, whose most common application 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

The project repository should 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:

$$\underbrace{\texttt{(a|}\overbrace{\texttt{(b|c)}}^{2}\texttt{)}}_{1}\underbrace{\texttt{(d|e)}}_{3}\texttt{*}$$

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`).

1.1 Parser

Extend your regular expression parser so that groups appear in the abstract syntax tree. That is, your `parse_re` must work like this:

```
$ cp3/parse_re '(a|(b|c))(d|e)*'
concat(group(1,union(symbol("a"),group(2,union(symbol("b"),symbol("c"))))),star(group(3,union(symbol("d"),symbol("e")))))
$ cp3/parse_re '(())'
group(1,group(2,epsilon()))
```

(If you want `bin/parse_re` to output groups, pass it the `-g` option.)

To do this, modify the parse table as follows:

	input		parse stack		semantic stack	
	read	peek	pop	push	pop	push
delete row:	ε	{(}	`P`	`(E)`	ε	ε
add row:	ε	{(}	`P`	`(E)` `group(`k`)`	ε	ε
add row:	ε	T	`group(`k`)`	ε	α	`group(`k`,`α`)`

This requires some extra explanation. When the parser sees a `P` on the parse stack and a `(` in the input, it pops the `P` and pushes `(E)` `group(`k`)`, where k is the next group number. Later, when the parser sees `group(`k`)` on the parse stack, then on the semantic stack, it replaces the top tree α with `group(`k`,`α`)`. See Table 1 for an example run. Test your modified parser by running `tests/test-cp3.sh`.

1.2 Capturing

Next, 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,

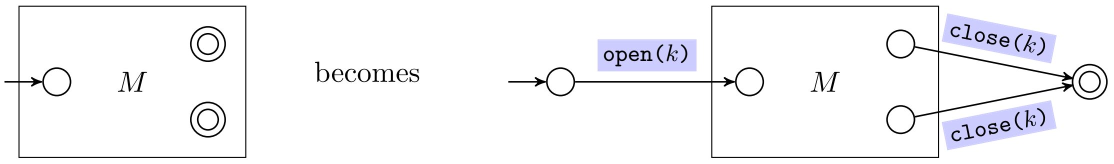
	state	input	parse stack	semantic stack
1	q_{start}	`(())⊣`	ε	
2	q_{loop}	`(())⊣`	E\$	
3	q_{loop}	`(())⊣`	TE′\$	
4	q_{loop}	`(())⊣`	FT′E′\$	
5	q_{loop}	`(())⊣`	PF′T′E′\$	
6	q_{loop}	`(())⊣`	(E) `group(1)` F′T′E′\$	
7	q_{loop}	`())⊣`	E) `group(1)` F′T′E′\$	
8	q_{loop}	`())⊣`	TE′) `group(1)` F′T′E′\$	
9	q_{loop}	`())⊣`	FT′E′) `group(1)` F′T′E′\$	
10	q_{loop}	`())⊣`	PF′T′E′) `group(1)` F′T′E′\$	
11	q_{loop}	`())⊣`	(E) `group(2)` F′T′E′) `group(1)` F′T′E′\$	
12	q_{loop}	`))⊣`	E) `group(2)` F′T′E′) `group(1)` F′T′E′\$	
13	q_{loop}	`))⊣`	TE′) `group(2)` F′T′E′) `group(1)` F′T′E′\$	
14	q_{loop}	`))⊣`	E′) `group(2)` F′T′E′) `group(1)` F′T′E′\$	`epsilon()`
15	q_{loop}	`))⊣`	) `group(2)` F′T′E′) `group(1)` F′T′E′\$	`epsilon()`
16	q_{loop}	`)⊣`	`group(2)` F′T′E′) `group(1)` F′T′E′\$	`epsilon()`
17	q_{loop}	`)⊣`	F′T′E′) `group(1)` F′T′E′\$	`group(2,epsilon())`
18	q_{loop}	`)⊣`	T′E′) `group(1)` F′T′E′\$	`group(2,epsilon())`
19	q_{loop}	`)⊣`	E′) `group(1)` F′T′E′\$	`group(2,epsilon())`
20	q_{loop}	`)⊣`	) `group(1)` F′T′E′\$	`group(2,epsilon())`
21	q_{loop}	`⊣`	`group(1)` F′T′E′\$	`group(2,epsilon())`
22	q_{loop}	`⊣`	F′T′E′\$	`group(1,group(2,epsilon()))`
23	q_{loop}	`⊣`	T′E′\$	`group(1,group(2,epsilon()))`
24	q_{loop}	`⊣`	E′\$	`group(1,group(2,epsilon()))`
25	q_{loop}	`⊣`	\$	`group(1,group(2,epsilon()))`
26	q_{accept}	ε	ε	`group(1,group(2,epsilon()))`

Table 1: Example run of the parser on the regular expression `(())`.

```
$ cp3/re_groups '(a|(b|c))(d|e)*' 'ade'
accept
1:a
3:e
$ cp3/re_groups '(a(b))' 'ab'
accept
1:ab
2:b
```

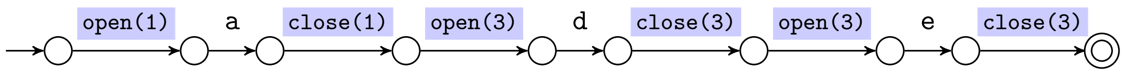
Print just those groups that matched a substring, in numerical order. If there's more than one way for the groups to match, you can choose any of them (but this doesn't happen in any of the supplied tests). Run `tests/test-cp3.sh` to check your implementation.

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`), which can be run in two ways:

```
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. It is an error for the same label to appear twice.
- A branch command has the form `/`*`regexp`*`/b`*`label`*. If the current string matches the *`regexp`*, it jumps to *`label`*. Note that unlike the real `sed`, regular expressions must match the entire line. It is an error for *`label`* to be undefined.
- 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 is a single digit ($1 \leq k \leq 9$).
  - \\g<k>: expands to the contents of group k, where $k \geq 1$.

For example, the following script reverses strings over $\{\texttt{a}, \texttt{b}\}$:

```
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. The provided `bin/msed` has a `-v` option to print out the steps:

```
$ bin/msed -v -f examples/reverse.sed
abab
1. subst abab -> ^abab
2. subst ^abab -> a^bab
3. branch 2
2. subst a^bab -> ba^ab
3. branch 2
2. subst ba^ab -> aba^b
3. branch 2
2. subst aba^b -> baba^
4. subst baba^ -> baba
baba
```

Test your implementation by running `tests/test-cp3.sh`.

3 Turing machines to `sed`

In this part, you will demosntrate that `msed` is Turing-complete by implementing a translation from Turing machines to `msed`. 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 `msed` script that is equivalent to *`tm_file`* (see below)

Note that `tm_to_sed` does *not* read in any input strings, and it does *not* attempt to run the Turing machine; it only translates the Turing machine into an equivalent `msed` script, as detailed below.

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

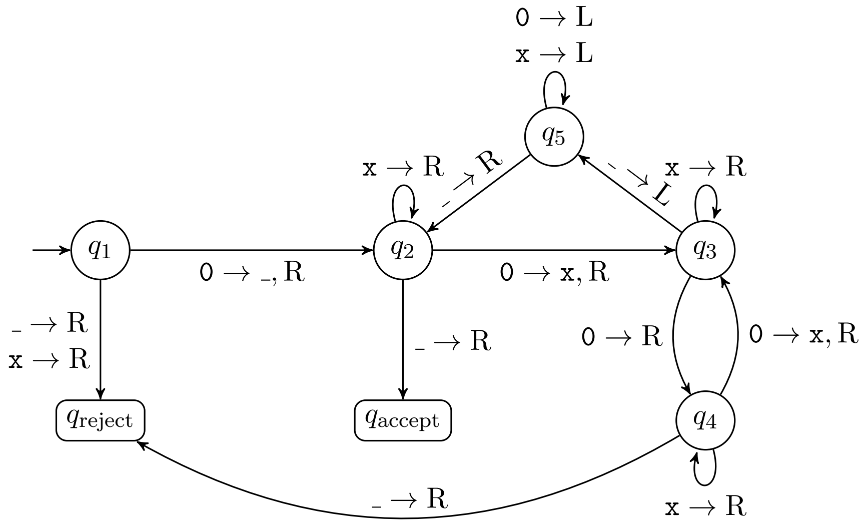
1. A whitespace-separated list of states, Q.
2. A whitespace-separated list of input symbols, Σ. It must be disjoint from Q, and it must not contain _ (blank). Each symbol must be a single character.
3. A whitespace-separated list of tape symbols, $\Gamma \supseteq \Sigma$. It must be disjoint from Q, it must contain _ (blank), and it should not include any of the special characters {`^`, `[`, `]`}. 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 must be at most one transition from q on symbol a. States q_{accept} and q_{reject} must have no outgoing transitions.

For example, the following Turing machine (M_2 in the book, page 173)



is specified by the file (`examples/sipser-m2.tm`):

```
q1 q2 q3 q4 q5 qaccept qreject
0
0 x _
q1
qaccept
qreject
q1 0 q2 _ R
q1 x qreject x R
q1 _ qreject _ R
q2 0 q3 x R
q2 x q2 x R
q2 _ qaccept _ R
q3 0 q4 0 R
q3 x q3 x R
q3 _ q5 _ L
q4 0 q3 x R
q4 x q4 x R
q4 _ qreject _ R
q5 0 q5 0 L
q5 x q5 x L
q5 _ q2 _ R
```

See `sipser-m1.tm` for another example, which corresponds to the machine M_1 in Sipser, page 174.

Operation The output of `tm_to_sed` (on stdout) must be an `msed` script that reads inputs strings, one per line, and for each string, it simulates the Turing machine described in ***`tm_file`***. If the machine accepts, the script must output `accept:` followed (without whitespace) by the final contents of the tape. If the machine rejects, the script must output `reject` by itself.

For example:

```
$ cp3/tm_to_sed examples/sipser-m2.tm > sipser-m2.sed
$ cp3/msed -f sipser-m2.sed
0
accept:___
00
accept:_x__
000
reject
0000
accept:_xxx__
```

The simulated Turing machine must follow the definition in Sipser. If the head is on the first cell and moves left, it stays on the first cell. If a transition is missing, reject.

There's more than one way to use `msed` to simulate a Turing machine. 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.

Since our Turing machine format does not allow the characters `^ [ ]` as tape symbols, you are free to use them in your encoding of configurations.

The test script `tests/test-cp3.sh` compares your `tm_to_msed` plus `msed` against `run_tm`, a direct Turing machine simulator. In other words, the following two commands must produce the same output, except perhaps for different numbers of trailing blanks:

```
cp3/tm_to_sed tm_file > sed_file; echo string | cp3/msed -f sed_file
echo string | bin/run_tm tm_file
```

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, change to its root directory, 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:

1. Push your repository to GitHub.
2. In GitHub, create a new release by clicking on "Releases," then "Draft a new release."
3. Fill in "Release title" with `cp3` if you're submitting the whole assignment, `cp3-1` if you're submitting part 1, `cp3-2` if you're submitting part 2, etc.
4. Click on "Choose a tag," then type the same name you used for the release title, then "Create new tag: `cp3`... on publish."
5. Finally, click "Publish Release."

Rubric

Part 1	
`parse_re`	3
capturing groups	3
`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