Course Project 2
Regular Expressions

CSE 30151 Spring 2024

Version of 2023-12-18
Due date 2024-03-01

In this project, you'll write a regular expression matcher similar to `grep`. This has three major steps: first, parse a regular expression into regular operations; second, execute the regular operations to create a NFA; third, run the NFA on input strings. Our linear-time NFA recognition algorithm is much faster than a naive regular expression matcher, and is in general much faster than Python and Perl's standard regular-expression matchers!

You will need a correct solution for CP1 to complete this project. If your CP1 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/regexp-skeleton
git push
```

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

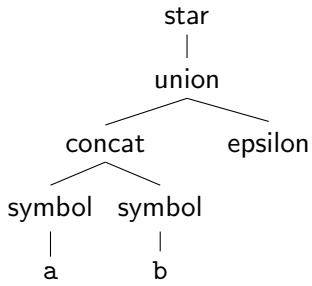
```
bin.{linux,darwin}/
  parse_re
  normalize_tree
  union_nfa
  concat_nfa
  star_nfa
  string_nfa
  re_to_nfa
  agrep
  compare_nfa
tests/
  test-cp2.sh
nonsolutions/
  bgrep.pl
  bgrep.py
cp2/
```

Please place the programs that you write into the `cp2/` subdirectory.

1 Parser

Note: This part and part 2.1 can be done in parallel.

In this first part, we'll write a parser for regular expressions. The goal is to input a regular expression like `(ab|)*` and output a tree like



In plain text, we write this tree as

```
star(union(concat(symbol("a"),symbol("b")),epsilon())).
```

Figure 1a shows a grammar for regular expressions. Let U be the set of all Unicode characters, let the terminal alphabet be $T = U \setminus \{\dashv\}$, and let $\Sigma = T \setminus \{(,), *, |, \backslash\}$. The nonterminal alphabet is $V = \{\mathsf{E}, \mathsf{M}, \mathsf{T}, \mathsf{F}, \mathsf{P}\}$, with start symbol E.

A typical parser for a programming language is based on a PDA equivalent to the CFG. You've seen one way to convert a CFG to a PDA (Lemma 2.21), which is "top-down" in the sense that it starts with the start symbol (E) and tries to rewrite it into the input string.

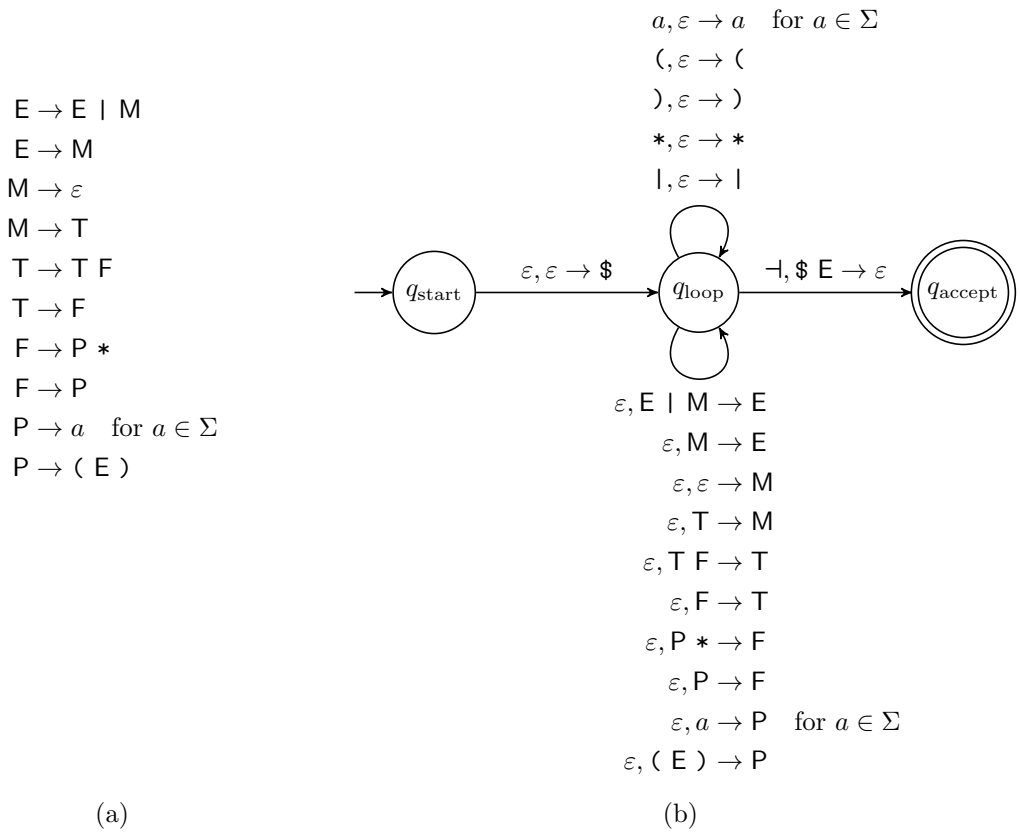
But there's another way to convert a CFG to a PDA, which works "bottom-up" in the sense that it starts with the string and tries to reduce it to E. It converts our grammar to the PDA in Figure 1b. This PDA's input alphabet is $T \cup \{\dashv\}$, where $\dashv$, called the *endmarker*, must be appended to the input string. Its stack alphabet is $\Gamma = V \cup T \cup \{\$\}$. Notice that some transitions pop multiple symbols. This is a shorthand similar to the shorthand in the proof of Lemma 2.21. But here we write stacks with the bottom to the left and the top to the right. For example, the transition $\varepsilon, \mathsf{E}\ \mathsf{|}\ \mathsf{M} \to \mathsf{E}$ means "pop M, pop $\mathsf{|}$, pop E, then push E." Table 2 shows an example run of this PDA.

The PDA is nondeterministic: for example, at step 11, three transitions seem to be possible:

$$\mathsf{|}, \varepsilon \to \mathsf{|}$$
$$\varepsilon, \mathsf{T} \to \mathsf{M}$$
$$\varepsilon, \varepsilon \to \mathsf{M}$$

The good news is that it's always possible to know which transition to follow. To do this, we use Table 3. Each row in this table is a transition from q_{loop} to q_{loop}, but with extra information.

- The below column is a set of stack symbols. The next stack symbol below the pop symbols must belong to this set. If blank, it defaults to Γ.
- The pop column is the stack symbols to be popped, but each nonterminal symbol has a subscript (α or β) that will be explained below.
- The read column is the input symbol that the transition reads.



(a) (b)

Figure 1: (a) A CFG for regular expressions, with start symbol E. Note that | is a terminal symbol; it is not being used for writing two rules on one line. (b) A PDA converted from the CFG.

	state	stack	input	transition
1	q_{start}	ε	`(ab|)*⊣`	$\varepsilon, \varepsilon \to \$$
2	q_{loop}	`$`	`(ab|)*⊣`	$(, \varepsilon \to ($
3	q_{loop}	`$(`	`ab|)*⊣`	$\mathtt{a}, \varepsilon \to \mathtt{a}$
4	q_{loop}	`$(a`	`b|)*⊣`	$\varepsilon, \mathtt{a} \to \mathsf{P}$
5	q_{loop}	`$(P`	`b|)*⊣`	$\varepsilon, \mathsf{P} \to \mathsf{F}$
6	q_{loop}	`$(F`	`b|)*⊣`	$\varepsilon, \mathsf{F} \to \mathsf{T}$
7	q_{loop}	`$(T`	`b|)*⊣`	$\mathtt{b}, \varepsilon \to \mathtt{b}$
8	q_{loop}	`$(Tb`	`|)*⊣`	$\varepsilon, \mathtt{b} \to \mathsf{P}$
9	q_{loop}	`$(TP`	`|)*⊣`	$\varepsilon, \mathsf{P} \to \mathsf{F}$
10	q_{loop}	`$(TF`	`|)*⊣`	$\varepsilon, \mathsf{TF} \to \mathsf{T}$
11	q_{loop}	`$(T`	`|)*⊣`	$\varepsilon, \mathsf{T} \to \mathsf{M}$
12	q_{loop}	`$(M`	`|)*⊣`	$\varepsilon, \mathsf{M} \to \mathsf{E}$
13	q_{loop}	`$(E`	`|)*⊣`	$\mathtt{\|}, \varepsilon \to \mathtt{\|}$
14	q_{loop}	`$(E|`	`)*⊣`	$\varepsilon, \varepsilon \to \mathsf{M}$
15	q_{loop}	`$(E|M`	`)*⊣`	$\varepsilon, \mathsf{E}\mathtt{\|}\mathsf{M} \to \mathsf{E}$
16	q_{loop}	`$(E`	`)*⊣`	$), \varepsilon \to)$
17	q_{loop}	`$(E)`	`*⊣`	$\varepsilon, (\mathsf{E}) \to \mathsf{P}$
18	q_{loop}	`$P`	`*⊣`	$*, \varepsilon \to *$
19	q_{loop}	`$P*`	`⊣`	$\varepsilon, \mathsf{P}* \to \mathsf{F}$
20	q_{loop}	`$F`	`⊣`	$\varepsilon, \mathsf{F} \to \mathsf{T}$
21	q_{loop}	`$T`	`⊣`	$\varepsilon, \mathsf{T} \to \mathsf{M}$
22	q_{loop}	`$M`	`⊣`	$\varepsilon, \mathsf{M} \to \mathsf{E}$
23	q_{loop}	`$E`	`⊣`	$\dashv, \$\mathsf{E} \to \varepsilon$
24	q_{accept}	ε	ε	

Table 2: Example run of the parser. The stack is written with the bottom to the left and the top to the right.

		below	pop	read	next	push
$a, \varepsilon \to a$	for $a \in \Sigma$	{$, (, |, T}	ε	a		a
$(, \varepsilon \to ($		{$, (, |, T}	ε	(		(
$), \varepsilon \to)$		{E}	ε	)		)
$\|, \varepsilon \to \|$		{E}	ε	|		|
$*, \varepsilon \to *$		{P}	ε	*		*
$\varepsilon, \mathsf{E}\,\mathsf{|}\,\mathsf{M} \to \mathsf{E}$			E_α | M_β	ε		$\mathsf{E}_{\texttt{union}(\alpha,\beta)}$
$\varepsilon, \mathsf{M} \to \mathsf{E}$		{$, (}	M_α	ε		E_α
$\varepsilon, \varepsilon \to \mathsf{M}$		{$, (, |}	ε	ε	{|,), ⊣}	$\mathsf{M}_{\texttt{epsilon()}}$
$\varepsilon, \mathsf{T} \to \mathsf{M}$			T_α	ε	{|,), ⊣}	M_α
$\varepsilon, \mathsf{TF} \to \mathsf{T}$			$\mathsf{T}_\alpha\ \mathsf{F}_\beta$	ε		$\mathsf{T}_{\texttt{concat}(\alpha,\beta)}$
$\varepsilon, \mathsf{F} \to \mathsf{T}$		{$, (, |}	F_α	ε		T_α
$\varepsilon, \mathsf{P}* \to \mathsf{F}$			P_α *	ε		$\mathsf{F}_{\texttt{star}(\alpha)}$
$\varepsilon, \mathsf{P} \to \mathsf{F}$			P_α	ε	$T \setminus \{*\}$	F_α
$\varepsilon, a \to \mathsf{P}$	for $a \in \Sigma$		a	ε		$\mathsf{P}_{\texttt{symbol("}a\texttt{")}}$
$\varepsilon, (\mathsf{E} \to \mathsf{P}$			(E_α)	ε		P_α

Table 3: Additional information for transitions from q_{loop} to q_{loop} of the PDA in Figure 1b.

- The next column is a set of input symbols. The next input symbol after the read symbols must belong to this set. If blank, it defaults to $T \cup \{\dashv\}$.
- The push column is the stack symbol to be pushed, but again with a subscript that will be explained below.

For example, at step 11, transition $\mathsf{|}, \varepsilon \to \mathsf{|}$ is not possible because it has below $= \{\mathsf{E}\}$ but the stack symbol below ε is T; similarly, transition $\varepsilon, \varepsilon \to \mathsf{M}$ is not possible because it has below $= \{\$, (, \mathsf{|}\}$. But transition $\varepsilon, \mathsf{T} \to \mathsf{M}$ is possible because it has next $= \{\mathsf{|},), \dashv\}$ and the next input symbol is |. The constraints are designed so that at most one transition is possible.[1] If no transition is possible, the parser must print an error message and quit.

In order to make the parser output a tree (not just an accept/reject decision), we modify the stack so that every element is either a terminal symbol $a \in T$ or of the form X_α, where $X \in V$ is a nonterminal symbol and α is a tree. In Table 3, the subscripts are instructions for how to build the trees. For example, the transition $\mathsf{T}_\alpha\ \mathsf{F}_\beta \to \mathsf{T}_{\texttt{concat}(\alpha,\beta)}$ means, pop an F and let β be its tree, pop a T and let α be its tree, and push T with tree `concat(`α`,`β`)`. Table 4 shows the same example run, but with trees. The transition in question is used in Table 4 at step 10. Just before this step, the second-from-the-top element is a T with tree `symbol("a")`, and the top element is an F with tree `symbol("b")`. So this step pops both and pushes T with tree `concat(symbol("a"),symbol("b"))`.

At the end of the input string, if the stack is $ E, the parser moves to q_{accept} and pops the E with a tree, which is the output parse tree.

[1] If you want to learn how to derive such constraints yourself, we invite you to read Sipser, Section 2.4, or Prof. Thain's *Introduction to Compilers and Language Design*, Chapter 4, about how to build LR parsers. Our parser here is known as a (1,1)-BRC parser, which is less powerful but very easy to implement.

	stack	input	transition
2	$	(ab|)*⊣	(, $\varepsilon \to$ (
3	$(	ab|)*⊣	a, $\varepsilon \to$ a
4	$(a	b|)*⊣	ε, a $\to \mathsf{P}_{\texttt{symbol("a")}}$
5	$($\mathsf{P}_{\texttt{symbol("a")}}$	b|)*⊣	$\varepsilon, \mathsf{P}_\alpha \to \mathsf{F}_\alpha$
6	$($\mathsf{F}_{\texttt{symbol("a")}}$	b|)*⊣	$\varepsilon, \mathsf{F}_\alpha \to \mathsf{T}_\alpha$
7	$($\mathsf{T}_{\texttt{symbol("a")}}$	b|)*⊣	b, $\varepsilon \to$ b
8	$($\mathsf{T}_{\texttt{symbol("a")}}$b	|)*⊣	ε, b $\to \mathsf{P}_{\texttt{symbol("b")}}$
9	$($\mathsf{T}_{\texttt{symbol("a")}}\mathsf{P}_{\texttt{symbol("b")}}$	|)*⊣	$\varepsilon, \mathsf{P}_\alpha \to \mathsf{F}_\alpha$
10	$($\mathsf{T}_{\texttt{symbol("a")}}\mathsf{F}_{\texttt{symbol("b")}}$	|)*⊣	$\varepsilon, \mathsf{T}_\alpha\mathsf{F}_\beta \to \mathsf{T}_{\texttt{concat}(\alpha,\beta)}$
11	$($\mathsf{T}_{\texttt{concat(symbol("a"),symbol("b"))}}$	|)*⊣	$\varepsilon, \mathsf{T}_\alpha \to \mathsf{M}_\alpha$
12	$($\mathsf{M}_{\texttt{concat(symbol("a"),symbol("b"))}}$	|)*⊣	$\varepsilon, \mathsf{M}_\alpha \to \mathsf{E}_\alpha$
13	$($\mathsf{E}_{\texttt{concat(symbol("a"),symbol("b"))}}$	|)*⊣	|, $\varepsilon \to$ |
14	$($\mathsf{E}_{\texttt{concat(symbol("a"),symbol("b"))}}$|	)*⊣	$\varepsilon, \varepsilon \to \mathsf{M}_{\texttt{epsilon()}}$
15	$($\mathsf{E}_{\texttt{concat(symbol("a"),symbol("b"))}}$\|$\mathsf{M}_{\texttt{epsilon()}}$	)*⊣	$\varepsilon, \mathsf{E}_\alpha\mid\mathsf{M}_\beta \to \mathsf{E}_{\texttt{union}(\alpha,\beta)}$
16	$($\mathsf{E}_{\texttt{union(concat(symbol("a"),symbol("b")),epsilon())}}$	)*⊣	), $\varepsilon \to$)
17	$($\mathsf{E}_{\texttt{union(concat(symbol("a"),symbol("b")),epsilon())}}$)	*⊣	$\varepsilon, (\mathsf{E}_\alpha) \to \mathsf{P}_\alpha$
18	$$\mathsf{P}_{\texttt{union(concat(symbol("a"),symbol("b")),epsilon())}}$	*⊣	*, $\varepsilon \to$ *
19	$$\mathsf{P}_{\texttt{union(concat(symbol("a"),symbol("b")),epsilon())}}$*	⊣	$\varepsilon, \mathsf{P}_\alpha* \to \mathsf{F}_{\texttt{star}(\alpha)}$
20	$$\mathsf{F}_{\texttt{star(union(concat(symbol("a"),symbol("b")),epsilon()))}}$	⊣	$\varepsilon, \mathsf{F}_\alpha \to \mathsf{T}_\alpha$
21	$$\mathsf{T}_{\texttt{star(union(concat(symbol("a"),symbol("b")),epsilon()))}}$	⊣	$\varepsilon, \mathsf{T}_\alpha \to \mathsf{M}_\alpha$
22	$$\mathsf{M}_{\texttt{star(union(concat(symbol("a"),symbol("b")),epsilon()))}}$	⊣	$\varepsilon, \mathsf{M}_\alpha \to \mathsf{E}_\alpha$
23	$$\mathsf{E}_{\texttt{star(union(concat(symbol("a"),symbol("b")),epsilon()))}}$	⊣	

Table 4: Example run of the parser, including construction of the parse tree. Only the transitions from q_{loop} to q_{loop} are shown. The stack is written with the bottom to the left and the top to the right.

Write code to run the parser described above on a regular expression and construct a tree from it. You will want to write it in a modular way so that in Part 2.2, you can modify the parser to construct an NFA intsead. To test your parser, write a program with the following command-line usage:

`parse_re` *regexp*

- *regexp*: a regular expression
- Output: string representation of the syntax tree for *regexp*

For example:

```
$ parse_re 'a'
symbol("a")
$ parse_re ''
epsilon()
$ parse_re '(a)'
symbol("a")
$ parse_re '()'
epsilon()
$ parse_re 'a*'
star(symbol("a"))
$ parse_re 'abc'
concat(concat(symbol("a"),symbol("b")),symbol("c"))
$ parse_re 'abc'
union(union(symbol("a"),symbol("b")),symbol("c"))
$ parse_re ''
union(union(epsilon(),epsilon()),epsilon())
```

Test your program by running `tests/test-cp2.sh`.

2 Converter

2.1 Regular operations

Write a function that creates an NFA that accepts exactly one string. To test it, write a program with the following command-line usage:

`string_nfa` w

- w: a string (possibly empty)
- Output: an NFA recognizing the language $\{w\}$

Write functions that perform the three regular operations, using the constructions given in the book, and programs to test them:

`union_nfa` M_1 M_2

- M_1, M_2: NFAs

- Output: NFA recognizing language $L(M_1) \cup L(M_2)$

`concat_nfa` M_1 M_2

- M_1, M_2: NFAs
- Output: NFA recognizing language $L(M_1) \circ L(M_2)$

`star_nfa` M

- M: an NFA
- Output: NFA recognizing language $L(M)^*$

Test all of these programs by running `tests/test-cp2.sh`.

2.2 Building the NFA

Modify your parser from Part 1 so that, instead of constructing a tree, it constructs an NFA. That is, whenever it used to construct a node `epsilon()`, it constructs an NFA that accepts only ε; whenever it used to construct a node `symbol("`a`")`, it constructs an NFA that accepts only a; whenever it used to construct a `union` node, it constructs the union of the two child NFAs, and so on. Beware that if you choose to construct a tree and then convert the tree to an NFA, you may get an error for exceeding the maximum recursion depth. Test your modified parser by writing a program with the following command-line usage:

`re_to_nfa` *regexp*

- *regexp*: Regular expression
- Output: NFA M equivalent to *regexp*

Test your program using `tests/test-cp2.sh`.

3 Putting it together

Finally, combine your regular expression converter with your NFA simulator from CP1 to write a `grep` replacement, called `agrep` (for "automaton-based `grep`"), that has the following command-line usage:

`agrep` *regexp*

- *regexp*: regular expression
- Input: strings (one per line)
- Output: the input strings that match *regexp*

Note that unlike `grep`, the regular expression must match the entire line, not just part of the line. Test your program by running `tests/test-cp2.sh`.

The test script also tests the time complexity of `agrep`. This test is the same as in CP1, but now we can say a bit more about it. For various values of n, it creates the regular expression `(|a)`^{n} and tries to match it against the string `a`^{n}`b`, using your `agrep`. You can also try timing `nonsolutions/bgrep.pl` and `nonsolutions/bgrep.py`, use Perl and Python's standard regular expression engines, by making `cp2/agrep` a symlink to either of them. How do they compare to yours?

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, `cd` into its root directory, run `make -C cp2`, and run `tests/test-cp2.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 `cp2` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use a tag version of the form `cp2-123`, indicating which parts you're submitting.

Rubric

Part 1	
parsing	3
building syntax tree	3
`parse_re`	3
Part 2	
`string_nfa`	3
`union_nfa`	3
`concat_nfa`	3
`star_nfa`	3
`re_to_nfa`	3
Part 3 (`agrep`)	
correctness	3
time complexity	3
Total	30