Course Project 2
Regular Expressions

CSE 30151 Spring 2025

Version of 2024-12-31
Due date 2025-02-28

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

The project repository should include the following files (among others):

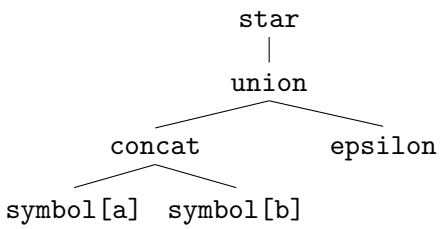
```
bin.{linux,darwin}/
  parse_re
  normalize_tree
  union_nfa
  concat_nfa
  star_nfa
  re_to_nfa
  agrep
  compare_nfa
tests/
  test-cp2.sh
nonsolutions/
  bgrep.pl
  bgrep.py
cp2/
```

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

1 Parser

Note: Part 1 (this part) and part 2.1 can be done in parallel. Part 2.2 depends on both, and may require modifications to part 1.

In this first part, we'll write a parser for regular expressions. A *parser* is an algorithm that inputs a string w, and tests whether w belongs to the language of a context-free grammar. If so, it outputs the *semantics* (meaning) of w. Eventually (in part 2.1) the semantics of a regular expression will be an NFA. But presently, for easier debugging, the semantics of a regular expression is a tree. For example, if $w =$ `(ab|)*`, the tree is



In plain text, we write the tree as

```
star(union(concat(symbol[a],symbol[b]),epsilon))
```

1.1 Background

Define the following alphabets:

$$A = \text{all printable, non-whitespace ASCII characters}$$
$$\Sigma = A \setminus \{\texttt{|}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{\textbackslash}, \texttt{\&}\}$$
$$T = \Sigma \cup \{\texttt{|}, \texttt{*}, \texttt{(}, \texttt{)}, \dashv\}$$

Alphabet Σ is the set of "ordinary" characters; our regular expression matcher will read strings over Σ. Alphabet T is the set of characters that regular expressions can contain; we'll explain $\dashv$ later.

Figure 1a shows a context-free grammar for regular expressions. The terminal alphabet is T, and the nonterminal alphabet is $V = \{\mathsf{E}, \mathsf{E'}, \mathsf{T}, \mathsf{T'}, \mathsf{F}, \mathsf{F'}, \mathsf{P}\}$, with start symbol E. If this grammar looks bigger than the grammar you would have written, it's because this grammar has been designed to avoid *left recursion* (e.g., rules of the form $X \to X\beta$). This is because, for the style of parser we're going to build, left recursion causes infinite loops.[1]

Parsers (for programming languages) are typically based on PDAs. If we convert the grammar to a PDA using the construction in Lemma 2.21, the resulting PDA is shown in Figure 1b. The input alphabet is T, and the stack alphabet is $\Gamma = V \cup T \cup \{\$\}$. The transitions involving a are actually many transitions, one for each $a \in \Sigma$. And $\dashv$ is an *endmarker* that must be appended to the input regular expression. Table 3 shows an example run of this PDA. For now, ignore the symbols `epsilon`, `symbol[`a`]`, `union`, `concat`, `star`, and the "semantic stack" column.

There are two problems with this PDA. First, it's nondeterministic, and we would rather implement a deterministic PDA (one where there's at most one transition that can be taken in any given configuration). Second, it will accept valid regular expressions and reject invalid ones, but we want it to output a tree (or eventually, an NFA).

For an example of nondeterminism, look at Table 3, row 5. The top stack symbol is P, so there are multiple transitions that the PDA could take: $\varepsilon, \mathsf{P} \to \texttt{(}\mathsf{E}\texttt{)}$ or $\varepsilon, \mathsf{P} \to a$. Which transition is the right one? Since the next input symbol is a `(`, you know it has to be $\varepsilon, \mathsf{P} \to \texttt{(}\mathsf{E}\texttt{)}$. For another

[1] For an explanation of how to eliminate left recursion, please see *Compilers and Language Design*, 2nd ed., Section 4.3.1–2.

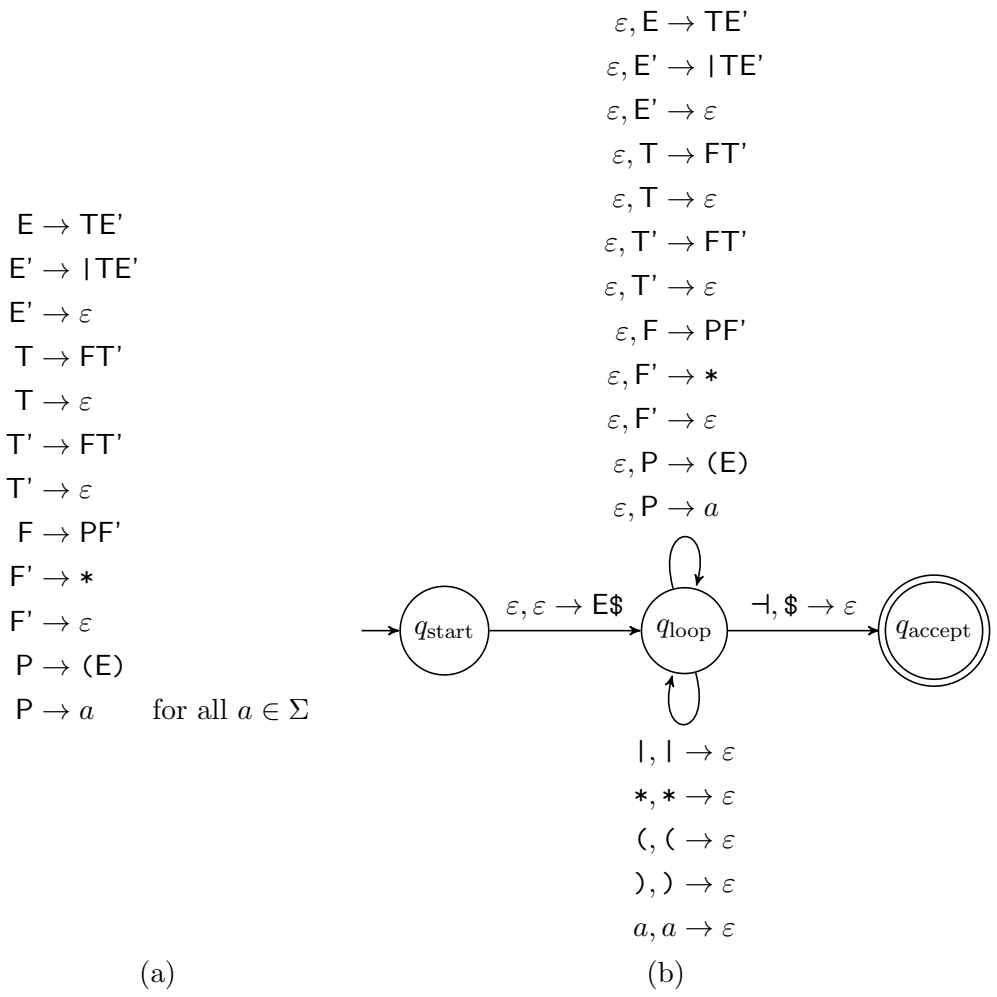


Figure 1: (a) A CFG for regular expressions, with start symbol E. Note that | is a terminal symbol; it is not being used for writing two rules on one line. (b) The equivalent PDA, by the construction in Lemma 2.21.

input		**parse stack**		**semantic stack**	
read	peek	pop	push	pop	push
ε	T	E	TE'	ε	ε
ε	$\{\texttt{|}\}$	E'	`|` T `union` E'	ε	ε
ε	$T \setminus \{\texttt{|}\}$	E'	ε	ε	ε
ε	$T \setminus \{\texttt{|}, \texttt{)}, \dashv\}$	T	FT'	ε	ε
ε	$\{\texttt{|}, \texttt{)}, \dashv\}$	T	`epsilon`	ε	ε
ε	$T \setminus \{\texttt{|}, \texttt{)}, \dashv\}$	T'	F `concat` T'	ε	ε
ε	$\{\texttt{|}, \texttt{)}, \dashv\}$	T'	ε	ε	ε
ε	T	F	PF'	ε	ε
ε	$\{\texttt{*}\}$	F'	`*` `star`	ε	ε
ε	$T \setminus \{\texttt{*}\}$	F'	ε	ε	ε
ε	$\{\texttt{(}\}$	P	`(`E`)`	ε	ε
ε	$\{a\}$	P	a `symbol[`a`]`	ε	ε
`|`		`|`	ε	ε	ε
`*`		`*`	ε	ε	ε
`(`		`(`	ε	ε	ε
`)`		`)`	ε	ε	ε
a		a	ε	ε	ε
ε	T	`epsilon`	ε	ε	`epsilon`
ε	T	`symbol[`a`]`	ε	ε	`symbol[`a`]`
ε	T	`union`	ε	$\beta\alpha$	`union(`α`,`β`)`
ε	T	`concat`	ε	$\beta\alpha$	`concat(`α`,`β`)`
ε	T	`star`	ε	α	`star(`α`)`

Table 2: Parse table. Every row is a transition from q_{loop} to itself. A row mentioning a stands for many transitions, one for each $a \in \Sigma$.

example, look at Table 3, row 3. The top stack symbol is T, so there are two transitions that the PDA could take: $\varepsilon, \mathsf{T} \to \mathsf{FT}'$ or $\varepsilon, \mathsf{T} \to \varepsilon$. Now, it's less obvious which transition is the right one. It turns out (for this grammar, not for all grammars) that you can always decide which transition to use by looking at the next input symbol. Have a look at Table 2. The "read" column and the "pop" and "push" columns under "parse stack" correspond to the three parts of a PDA transition that you're familiar with. The "peek" column tells you how to use the next input symbol to decide which transition to use. In our example, the next input symbol is `(`, so we look in Table 2 for the transition that has `(` in the "peek" column, which is $\varepsilon, \mathsf{T} \to \mathsf{FT}'$. In general, each entry in the "peek" column is a set of symbols; a transition can only be used if the next unread input symbol is in the "peek" set.[2]

To compute semantics, we're going to add a second stack, called the *semantic stack*; we'll call the original stack the *parse stack*. The semantic stack (for now) stores trees. In Table 3 at step 10, the PDA pops a P from the parse stack and pushes an `a` as well as the *action* `symbol[a]` onto the parse stack. When the `symbol[a]` reaches the top of the parse stack (step 12), the relevant row of Table 2 says to pop the `symbol[a]` from the parse stack, as well as to push `symbol[a]` onto the semantic stack. This is the semantics of `a`.

Similarly, at step 15, when the PDA pops T' and pushes FT', it also pushes the action `concat`.[3] When the `concat` reaches the top of the parse stack (step 19), the relevant row of Table 2 says to pop the `concat` from the parse stack, as well as to pop β then α from the semantic stack (where α and β stand for any trees) and push `concat(`α`,`β`)` onto the semantic stack. Accordingly, it pops `symbol[b]` then `symbol[a]` from the semantic stack, and pushes `concat(symbol[a],symbol[b])` onto the semantic stack. This is the semantics of `ab`.

At the end of an accepting run, the semantic stack has exactly one tree; this is the semantics of the input regular expression.

1.2 Implementation

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 instead. 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:

```
$ cp2/parse_re 'a'
symbol[a]
$ cp2/parse_re ''
epsilon
$ cp2/parse_re '(a)'
symbol[a]
```

[2] For an explanation of how to derive the "peek" sets, please see *Compilers and Language Design*, 2nd ed., Section 4.3.3.

[3] Why does `concat` go between F and T'? This has to do with the removal of left recursion. For an explanation, please see Intro to PLs and Compilers, Section 8.

	state	input	parse stack	semantic stack
1	q_{start}	(ab|)*⊣	ε	ε
2	q_{loop}	(ab|)*⊣	E$	ε
3	q_{loop}	(ab|)*⊣	TE'$	ε
4	q_{loop}	(ab|)*⊣	FT'E'$	ε
5	q_{loop}	(ab|)*⊣	PF'T'E'$	ε
6	q_{loop}	(ab|)*⊣	(E)F'T'E'$	ε
7	q_{loop}	ab|)*⊣	E)F'T'E'$	ε
8	q_{loop}	ab|)*⊣	TE')F'T'E'$	ε
9	q_{loop}	ab|)*⊣	FT'E')F'T'E'$	ε
10	q_{loop}	ab|)*⊣	PF'T'E')F'T'E'$	ε
11	q_{loop}	ab|)*⊣	a symbol[a] F'T'E')F'T'E'$	ε
12	q_{loop}	b|)*⊣	symbol[a] F'T'E')F'T'E'$	ε
13	q_{loop}	b|)*⊣	F'T'E')F'T'E'$	symbol[a]
14	q_{loop}	b|)*⊣	T'E')F'T'E'$	symbol[a]
15	q_{loop}	b|)*⊣	F concat T'E')F'T'E'$	symbol[a]
16	q_{loop}	b|)*⊣	PF' concat T'E')F'T'E'$	symbol[a]
17	q_{loop}	b|)*⊣	b symbol[b] F' concat T'E')F'T'E'$	symbol[a]
18	q_{loop}	|)*⊣	symbol[b] F' concat T'E')F'T'E'$	symbol[a]
19	q_{loop}	|)*⊣	F' concat T'E')F'T'E'$	symbol[b] symbol[a]
20	q_{loop}	|)*⊣	concat T'E')F'T'E'$	symbol[b] symbol[a]
21	q_{loop}	|)*⊣	T'E')F'T'E'$	concat(symbol[a],symbol[b])
22	q_{loop}	|)*⊣	E')F'T'E'$	concat(symbol[a],symbol[b])
23	q_{loop}	|)*⊣	|T union E')F'T'E'$	concat(symbol[a],symbol[b])
24	q_{loop}	)*⊣	T union E')F'T'E'$	concat(symbol[a],symbol[b])
25	q_{loop}	)*⊣	epsilon union E')F'T'E'$	concat(symbol[a],symbol[b])
26	q_{loop}	)*⊣	union E')F'T'E'$	epsilon concat(symbol[a],symbol[b])
27	q_{loop}	)*⊣	E')F'T'E'$	union(concat(symbol[a],symbol[b]),epsilon)
28	q_{loop}	)*⊣	)F'T'E'$	union(concat(symbol[a],symbol[b]),epsilon)
29	q_{loop}	*⊣	F'T'E'$	union(concat(symbol[a],symbol[b]),epsilon)
30	q_{loop}	*⊣	* star T'E'$	union(concat(symbol[a],symbol[b]),epsilon)
31	q_{loop}	⊣	star T'E'$	union(concat(symbol[a],symbol[b]),epsilon)
32	q_{loop}	⊣	T'E'$	star(union(concat(symbol[a],symbol[b]),epsilon))
33	q_{loop}	⊣	E'$	star(union(concat(symbol[a],symbol[b]),epsilon))
34	q_{loop}	⊣	$	star(union(concat(symbol[a],symbol[b]),epsilon))
35	q_{accept}	ε	ε	star(union(concat(symbol[a],symbol[b]),epsilon))

Table 3: Example run of the parser.

```
$ cp2/parse_re '()'
epsilon
$ cp2/parse_re 'a*'
star(symbol[a])
$ cp2/parse_re 'abc'
concat(concat(symbol[a],symbol[b]),symbol[c])
$ cp2/parse_re 'a|b|c'
union(union(symbol[a],symbol[b]),symbol[c])
$ cp2/parse_re '||'
union(union(epsilon,epsilon),epsilon)
```

Test your program by running `tests/test-cp2.sh`. You can run `bin/parse_re` to see more examples, and you can use the `-v` option to see a trace similar to Table 3.

2 Converter

2.1 Regular operations

In CP1, you wrote functions to construct NFAs recognizing $\{\varepsilon\}$ and $\{a\}$ for any $a \in \Sigma$. These serve as the base cases for the conversion from a regular expression to an NFA. Now, 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. The semantic stack used to be a stack of trees, but now it is a stack of NFAs. In Table 2, the row that used to push a tree `epsilon` now pushes an NFA recognizing $\{\varepsilon\}$. The row that used to push a tree `symbol[`a`]` now pushes an NFA recognizing $\{a\}$. The row that used to pop trees β and α and push tree `union(`α`,`β`)` now pops NFAs β and α and pushes an NFA recognizing $L(\alpha) \cup L(\beta)$, and so on.

An alternative solution is to still construct a tree and then traverse the tree bottom-up to construct an NFA. This is more modular, but beware that you may get an error for exceeding the maximum recursion depth.

Using this modified parser, write 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, change to its root directory, run `make -C cp2`, and then 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:

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 `cp2` if you're submitting the whole assignment, `cp2-1` if you're submitting part 1, `cp2-2` 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: `cp2`... on publish."
5. Finally, click "Publish Release."

Rubric

Part 1	
parsing	6
building syntax tree	3
`parse_re`	3
Part 2	
`union_nfa`	3
`concat_nfa`	3
`star_nfa`	3
`re_to_nfa`	3
Part 3 (`agrep`)	
correctness	3
time complexity	3
Total	30