# Course Project 2
# Regular Expressions

CSE 30151 Spring 2023

Version of 2022-12-08
Due date 2023-03-10

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. Because we use a linear-time NFA recognition algorithm, our regular expression matcher will actually be much faster than one written using Perl or Python's regular expression engine. (Most implementations of `grep`, as well as Google RE2, are linear like ours.)

**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):

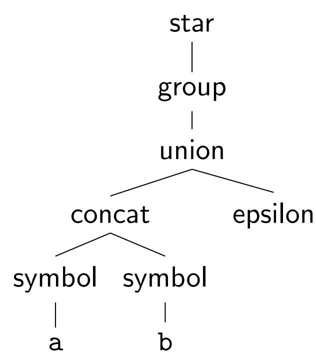
```
bin.{linux,darwin}/
  parse_re
  normalize_tree
  union_nfa
  concat_nfa
  star_nfa
  string_nfa
  re_to_nfa
  agrep
  compare_nfa
perl/
  bgrep.pl
tests/
  test-cp2.sh
cp2/
```

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 (group (union (concat (symbol "a") (symbol "b")) (epsilon)))).
```

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

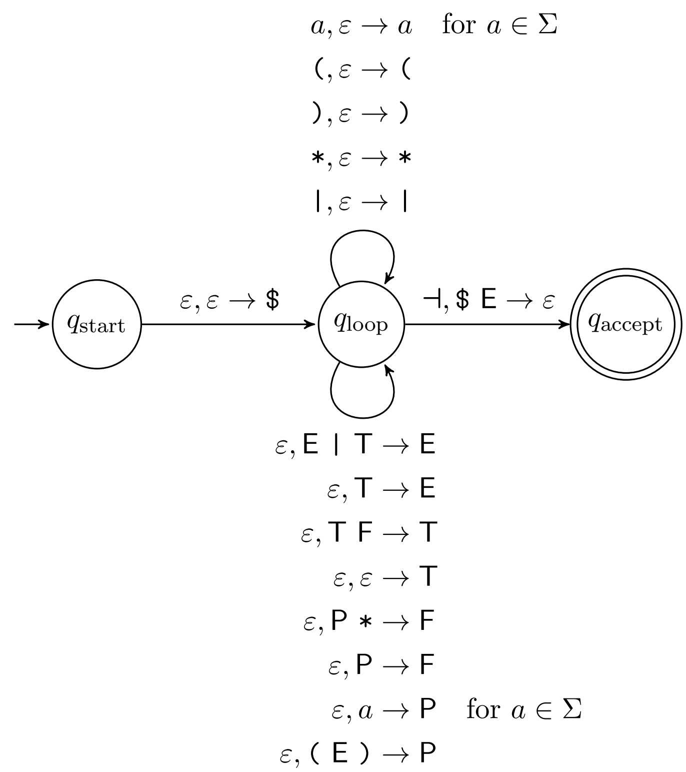
A typical parser for a programming language is based on a PDA equivalent to the CFG. You've seen one way to convert a CFG to a PDA (Lemma 2.21), which is "top-down" in the sense that it starts with the start symbol ($\mathsf{E}$) and tries to rewrite it into the input string. There's another way to convert a CFG to a PDA, which works "bottom-up" in the sense that it starts with the string and tries to reduce it to $\mathsf{E}$. It converts our grammar to the PDA in Figure 1b. Its input alphabet is $T \cup \{\dashv\}$, where $\dashv$, called the *endmarker*, must be appended to the input string. Its stack alphabet is $\Gamma = V \cup T \cup \{\$\}$. We are using a shorthand similar to that in the proof of Lemma 2.21, in which a transition can pop multiple symbols. However, we are writing stacks with the bottom to the left and the top to the right. For example, the transition $\varepsilon, \mathsf{E}\ \texttt{|}\ \mathsf{T} \to \mathsf{E}$ means "pop $\mathsf{T}$, pop `|`, pop $\mathsf{E}$, then push $\mathsf{E}$."

Table 3 shows an example run of this PDA. It starts in state $q_{\text{start}}$, pushes $\$$, and moves to state $q_{\text{loop}}$. Then, at each step, it follows a transition. The PDA is nondeterministic: for example, at step 12, three transitions seem to be possible: $\texttt{|}, \varepsilon \to \texttt{|}$ and $\varepsilon, \mathsf{T} \to \mathsf{E}$ and $\varepsilon, \varepsilon \to \mathsf{T}$. To decide which transition to follow, we need some additional constraints, shown in Table 2. Each row in this table is a transition from $q_{\text{loop}}$ to $q_{\text{loop}}$, and has a $\mathsf{below}$ set (which defaults to $\Gamma$) and a $\mathsf{next}$ set (which defaults to $T \cup \{\dashv\}$). If a transition has label $a, x \to y$, where $a \in T \cup \{\dashv, \varepsilon\}$ and $x \in \Gamma^*$, it can be followed iff the remaining input string is $abv\dashv$ for some $b \in \mathsf{next}$ and $v \in T^*$, and the stack is $uzx$ for some $u \in \Gamma^*$ and $z \in \mathsf{below}$. For example, at step 12, transition $\texttt{|}, \varepsilon \to \texttt{|}$ is not possible because it has $\mathsf{below} = \{\mathsf{E}\}$ but the stack symbol below $\varepsilon$ is $\mathsf{T}$; similarly, transition $\varepsilon, \varepsilon \to \mathsf{T}$ is not possible because it has $\mathsf{below} = \{\$, \texttt{(}, \texttt{|}\}$. But transition $\varepsilon, \mathsf{T} \to \mathsf{E}$ is possible because it has $\mathsf{below} = \{\$, \texttt{(}\}$ and the stack symbol under $\mathsf{T}$ is `(`, and it has $\mathsf{next} = \{\texttt{|}, \texttt{)}, \dashv\}$ 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 should print an error message and quit.

[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.

$\mathsf{E} \rightarrow \mathsf{E} \ | \ \mathsf{T}$
$\mathsf{E} \rightarrow \mathsf{T}$
$\mathsf{T} \rightarrow \mathsf{T} \ \mathsf{F}$
$\mathsf{T} \rightarrow \varepsilon$
$\mathsf{F} \rightarrow \mathsf{P} \ *$
$\mathsf{F} \rightarrow \mathsf{P}$
$\mathsf{P} \rightarrow a \quad \text{for } a \in \Sigma$
$\mathsf{P} \rightarrow ( \ \mathsf{E} \ )$

(a)




(b)

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

| | below | next | tree |
|---|---|---|---|
| $a, \varepsilon \rightarrow a \quad \text{for } a \in \Sigma$ | $\{\mathsf{T}\}$ | | |
| $(, \varepsilon \rightarrow ($ | $\{\mathsf{T}\}$ | | |
| $), \varepsilon \rightarrow )$ | $\{\mathsf{E}\}$ | | |
| $\|, \varepsilon \rightarrow \|$ | $\{\mathsf{E}\}$ | | |
| $*, \varepsilon \rightarrow *$ | $\{\mathsf{P}\}$ | | |
| $\varepsilon, \mathsf{E} \ \| \ \mathsf{T} \rightarrow \mathsf{E}$ | | $\{\|, ), \dashv\}$ | `(union` $\tau_1$ $\tau_2$`)` |
| $\varepsilon, \mathsf{T} \rightarrow \mathsf{E}$ | $\{\$, (\}$ | $\{\|, ), \dashv\}$ | $\tau_1$ |
| $\varepsilon, \mathsf{T} \ \mathsf{F} \rightarrow \mathsf{T}$ | | | `(concat` $\tau_1$ $\tau_2$`)` |
| $\varepsilon, \varepsilon \rightarrow \mathsf{T}$ | $\{\$, (, \|\}$ | | `(epsilon)` |
| $\varepsilon, \mathsf{P} \ * \rightarrow \mathsf{F}$ | | | `(star` $\tau_1$`)` |
| $\varepsilon, \mathsf{P} \rightarrow \mathsf{F}$ | | $\Sigma \cup \{(, ), \|, \dashv\}$ | $\tau_1$ |
| $\varepsilon, a \rightarrow \mathsf{P} \quad \text{for } a \in \Sigma$ | | | `(symbol "`$a$`")` |
| $\varepsilon, ( \ \mathsf{E} \ ) \rightarrow \mathsf{P}$ | | | `(group` $\tau_1$`)` |

Table 2: Additional information for transitions from $q_{\text{loop}}$ to $q_{\text{loop}}$ of the PDA in Figure 1b.

<table>
<tr><th></th><th>stack</th><th>input</th><th>transition</th><th>pushed tree</th></tr>
<tr><td>1</td><td>$</td><td>(ab|)*⊣</td><td>$\varepsilon, \varepsilon \to$ T</td><td><code>(epsilon)</code></td></tr>
<tr><td>2</td><td>$T</td><td>(ab|)*⊣</td><td>(, $\varepsilon \to$ (</td><td></td></tr>
<tr><td>3</td><td>$T(</td><td>ab|)*⊣</td><td>$\varepsilon, \varepsilon \to$ T</td><td><code>(epsilon)</code></td></tr>
<tr><td>4</td><td>$T(T</td><td>ab|)*⊣</td><td>a, $\varepsilon \to$ a</td><td></td></tr>
<tr><td>5</td><td>$T(Ta</td><td>b|)*⊣</td><td>$\varepsilon$, a $\to$ P</td><td><code>(symbol "a")</code></td></tr>
<tr><td>6</td><td>$T(TP</td><td>b|)*⊣</td><td>$\varepsilon$, P $\to$ F</td><td><code>(symbol "a")</code></td></tr>
<tr><td>7</td><td>$T(TF</td><td>b|)*⊣</td><td>$\varepsilon$, TF $\to$ T</td><td><code>(symbol "a")</code></td></tr>
<tr><td>8</td><td>$T(T</td><td>b|)*⊣</td><td>b, $\varepsilon \to$ b</td><td></td></tr>
<tr><td>9</td><td>$T(Tb</td><td>|)*⊣</td><td>$\varepsilon$, b $\to$ P</td><td><code>(symbol "b")</code></td></tr>
<tr><td>10</td><td>$T(TP</td><td>|)*⊣</td><td>$\varepsilon$, P $\to$ F</td><td><code>(symbol "b")</code></td></tr>
<tr><td>11</td><td>$T(TF</td><td>|)*⊣</td><td>$\varepsilon$, TF $\to$ T</td><td><code>(concat (symbol "a") (symbol "b"))</code></td></tr>
<tr><td>12</td><td>$T(T</td><td>|)*⊣</td><td>$\varepsilon$, T $\to$ E</td><td><code>(concat (symbol "a") (symbol "b"))</code></td></tr>
<tr><td>13</td><td>$T(E</td><td>|)*⊣</td><td>|, $\varepsilon \to$ |</td><td></td></tr>
<tr><td>14</td><td>$T(E|</td><td>)*⊣</td><td>$\varepsilon, \varepsilon \to$ T</td><td><code>(epsilon)</code></td></tr>
<tr><td>15</td><td>$T(E|T</td><td>)*⊣</td><td>$\varepsilon$, E|T $\to$ E</td><td><code>(union (concat (symbol "a") (symbol "b")) (epsilon))</code></td></tr>
<tr><td>16</td><td>$T(E</td><td>)*⊣</td><td>), $\varepsilon \to$ )</td><td></td></tr>
<tr><td>17</td><td>$T(E)</td><td>*⊣</td><td>$\varepsilon$, (E) $\to$ P</td><td><code>(group (union (concat (symbol "a") (symbol "b")) (epsilon)))</code></td></tr>
<tr><td>18</td><td>$TP</td><td>*⊣</td><td>*, $\varepsilon \to$ *</td><td></td></tr>
<tr><td>19</td><td>$TP*</td><td>⊣</td><td>$\varepsilon$, P* $\to$ F</td><td><code>(star (group (union (concat (symbol "a") (symbol "b")) (epsilon))))</code></td></tr>
<tr><td>20</td><td>$TF</td><td>⊣</td><td>$\varepsilon$, TF $\to$ T</td><td><code>(star (group (union (concat (symbol "a") (symbol "b")) (epsilon))))</code></td></tr>
<tr><td>21</td><td>$T</td><td>⊣</td><td>$\varepsilon$, T $\to$ E</td><td><code>(star (group (union (concat (symbol "a") (symbol "b")) (epsilon))))</code></td></tr>
<tr><td>22</td><td>$E</td><td>⊣</td><td>⊣, $E $\to \varepsilon$</td><td></td></tr>
</table>

Table 3: Example run of the parser. The stack is written with the bottom to the left and the top to the right.

In order to make the parser output a tree (not just an accept/reject decision), every transition that pushes a nonterminal $Y$ attaches a subtree to $Y$. Table 2 shows, for each such transition, what the tree to be attached to $Y$ is. The variables $\tau_1$ and $\tau_2$ stand for the trees attached to the first and second popped nonterminals. For example, in Table 3, at step 11, since transition $\varepsilon$, T F $\to$ T has `tree` = `(concat` $\tau_1$ $\tau_2$`)`, we push T with a tree that has a root with label `concat` and two children: `(symbol "a")`, attached to the popped T, and `(symbol "b")`, associated with the popped F.

For efficiency's sake (in particular, if your Part 2.1 is recursive and you find that it exceeds the maximum recursion depth), you can apply simplifications like the following:

- `(concat (epsilon)` $\alpha$`)` becomes $\alpha$.
- `(union (union` $\alpha$ $\beta$`)` $\gamma$`)` becomes `(union` $\alpha$ $\beta$ $\gamma$`)`.
- `(concat (concat` $\alpha$ $\beta$`)` $\gamma$`)` becomes `(concat` $\alpha$ $\beta$ $\gamma$`)`.

For example, in Table 3, at step 7, the tree would have been `(concat (epsilon) (symbol "a"))`, but is simplified to just `(symbol "a")`. The automated tester simplifies your trees for you, so a correct parser will pass the tests with or without simplification.

Finally, at the end of the input string, if the stack is $ E, the parser moves to $q_{\text{accept}}$ and outputs the tree associated with the popped E. That's the end! It will probably be helpful at this point to review the entire example run in Table 3.

Write code to run the parser described above on a regular expression and construct a tree from it. To test your parser, write a program with the following 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)'
(group (symbol "a"))
$ parse_re '()'
(group (epsilon))
$ parse_re 'a*'
(star (symbol "a"))
$ parse_re 'abc'
(concat (symbol "a") (symbol "b") (symbol "c"))
$ parse_re 'a|b|c'
(union (symbol "a") (symbol "b") (symbol "c"))
$ parse_re '||'
(union (epsilon) (epsilon) (epsilon))
```

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

# 2 Converter

## 2.1 Regular operations

Write a function that creates an NFA that accepts exactly one string, and a program to test it:

`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 `test-cp2.sh`.

### 2.2 Building the NFA

Write a function that converts (the syntax tree of) a regular expression to an NFA, by walking the tree bottom-up and using the operations implemented in Section 2.1. For now, the `group` operation should do nothing. Then combine your regular expression parser with this function into a test program:

`re_to_nfa` *regexp*

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

Test your program using `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`"):

`agrep` *regexp*

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

Note that unlike `grep`, the regular expression should 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 $(\texttt{(aa|a)(a|aa)})^n\texttt{a}^{4n}$ and tries to match it against the string $\texttt{a}^{4n}$, using our `agrep` and yours. For fun, we've provided a Perl implementation, called `bgrep.pl`, which you can try for comparison. (I ran out of patience and killed it.)

## 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 |