The Basic Principle

Given a regular expression, parse the expression and represent it as an NFA. The constructed NFA can be used to determine whether or not a string is accepted by the regular expression.

Program Usage

Running the program will show the prompt below. Enter a number to select an option.

```
Current alphabet: ab
Current expression: none
Enter a digit to choose an option:
1. Change alphabet
2. Change expression
3. Process string
4. Help
5. Exit
Enter selection:
```

Prompt for changing the alphabet, which accepts alphanumeric characters. This will clear the current expression. Defaults to ‘ab’. Spaces are ignored.

```
Enter selection: 1
Type all characters in alphabet: abcdefg
```

Prompt for entering a regular expression. An expression is specified with (), + for union, * for star, _ for the empty string (ε), and $ for any symbol in the alphabet (Σ). Spaces are ignored.

```
Enter selection: 2
Enter expression: (a + b)*d(ef + _)g$
```

Prompt for processing a string. The empty string is specified by _. Spaces are ignored.

```
Enter selection: 3
Enter string: baab dgc
Accepted
```

```
Enter string: _
Rejected
```

Help displays information about the available commands.

```
Enter selection: 4
   Spaces are ignored for all inputs
1. The alphabet can consist of alphanumeric characters. Changing the alphabet will clear the current expression
2. Specify a regular expression using (), + for union, * for star, _ for epsilon, and $ for sigma
   The expression parser will inform you of any invalid syntax in your expression string
3. Processes a string. The empty string is specified by _
4. Help displays this message
5. Exists the program
```

Exiting terminates the program

```
Enter selection: 5
```

Parsing a Regular Expression

Summary of section

- Extract a list of operators from regular expression, complexity O(n)
- Divide and conquer list to generate parse tree, complexity O(nlogn) assuming favorable division
- Overall parsing complexity of O(n + nlogn) → O(nlogn)

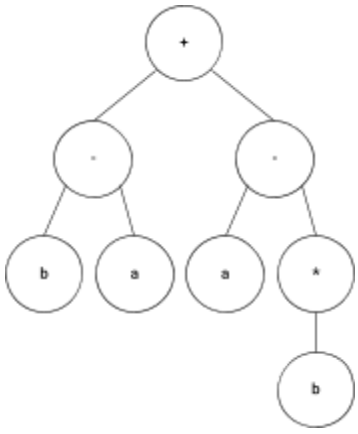
For this program, I used a slightly modified definition of a regular expression than the one provided in the textbook. This is to simplify the implementation.

> **R** is a regular expression if **R** is
>
> 1. **a** for some **a** in the alphabet Σ_ε,
> 2. $\mathbf{R}_1$ **+** $\mathbf{R}_2$, where $\mathbf{R}_1$ and $\mathbf{R}_2$ are regular expressions,
> 3. $\mathbf{R}_1\mathbf{R}_2$, where $\mathbf{R}_1$ and $\mathbf{R}_2$ are regular expressions, or
> 4. $\mathbf{R}_1$*****, where $\mathbf{R}_1$ is a regular expression.

The key difference here is that the empty language cannot be represented. This is not a problem though, as it is trivial to determine if a string is accepted by the empty language or not.

In a regular expression, operators are evaluated in the following order: star, concatenation, union. Parentheses can be used to change the order in which these are applied. To implement parsing, I approached the problem from a top down perspective.

Consider the expression **ba + ab***. In this expression, **b*** is evaluated first, followed by **ba** and **ab***. Lastly, the union is applied. The parse tree for the expression is shown on the right. Notice how the operators appear in reverse precedence, with union occuring at the root of the tree, concatenation occurring after union, and star after concatenation. This is the case for operators that occur within the same 'level' – that is, operators in the same set of parentheses. Using this fact, the parse tree for a regular expression can be constructed if one can get a list of all operators in the expression and their respective level, which describes how many sets of parentheses enclose the operator.



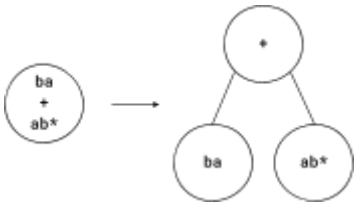
To extract the operators, I did a single pass over the input expression. This has a time complexity of O(n), in which n is the number of characters in the expression. At each character, the possible operator implied by the current character is compared to the previously implied operator. Through this, the operators are extracted and any errors in the expression will be caught.

The result of the operator extraction is a list of operators that are described by a level, an index corresponding to its position in the expression string, and the type of operation. For example, the expression string **"ba + ab*"** produces the following operator list:

```
Op[Level=0, Index=1, Type=Concatenation],
Op[Level=0, Index=2, Type=Union],
Op[Level=0, Index=4, Type=Concatenation],
Op[Level=0, Index=5, Type=Star]
```

The next step is to find the operator at the lowest level with the lowest precedence. This can be done in O(n) time, where n is the number of operators. Once the lowest operator is found, the expression can be split in two centered at the lowest operator, where the left side of the expression is R_1 and the right is R_2. Continuing the above example, **`Op[Level=0, Index=2, Type=Union]`** is the lowest. Splitting the expression yields two lists:

```
Op[Level=0, Index=1, Type=Concatenation]
```
and
```
Op[Level=0, Index=4, Type=Concatenation],
Op[Level=0, Index=5, Type=Star].
```



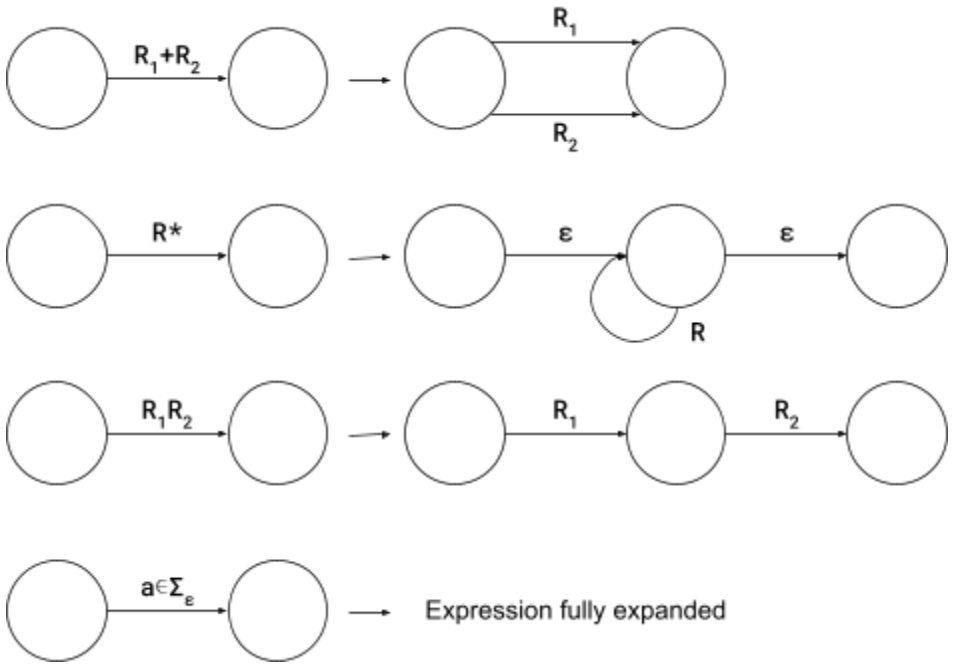
This splitting operation can now be applied to the two new sublists, each with O(n) complexity, where n is the length of each sublist. Performing this recursively until no operations are left will generate a complete parse tree for the expression. This algorithm is a divide and conquer type algorithm with a time complexity of O(nlogn), assuming the splitting operation is performed near the center of the list each time.

Constructing the NFA

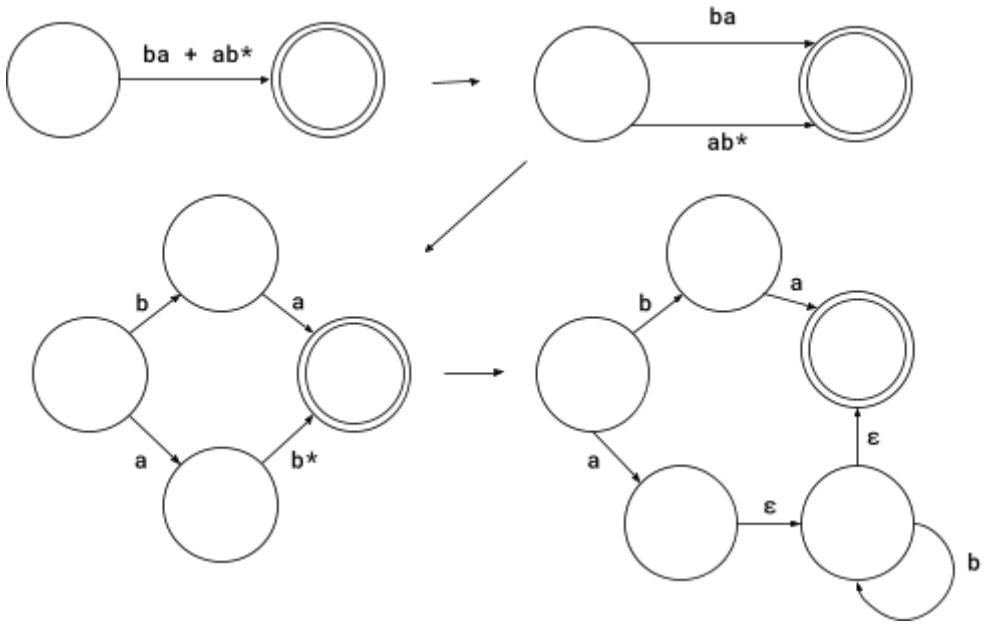
Summary of section

- Create NFA with one transition, specified by the regular expression parse tree
- Expand NFA until only terminals transitions remain
- Remove epsilon transitions
- Hard to determine meaningful time complexity

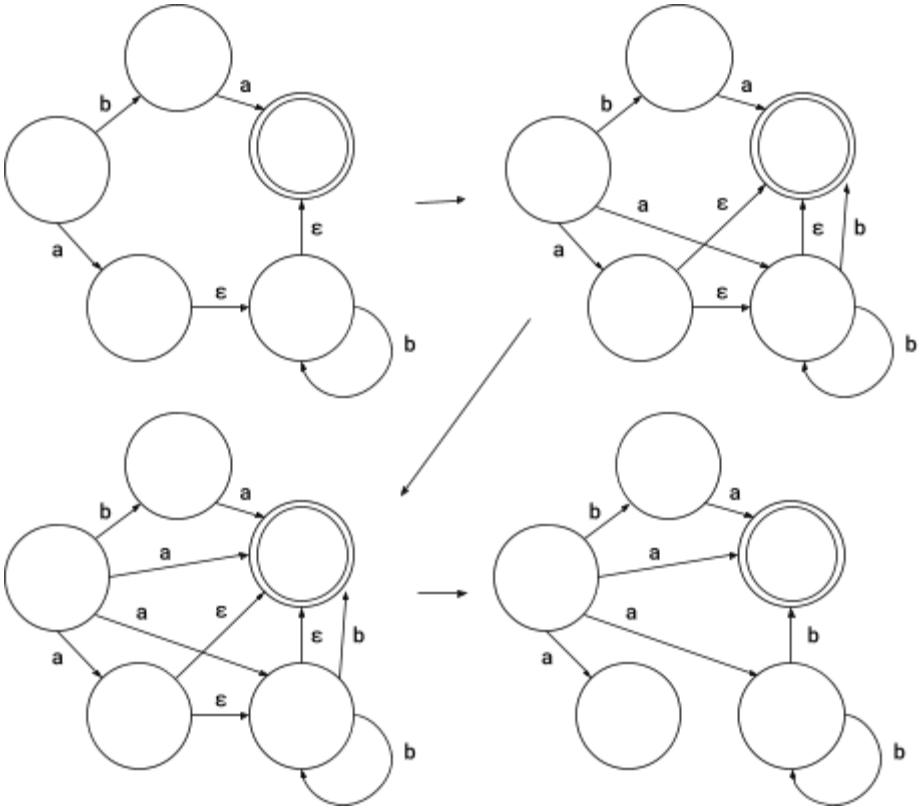
To construct an NFA from the regular expression, I relied on a simple expansion procedure. First, the regular expression is represented by a two state NFA with a single arrow from start to end, with the transition being the entire regular expression. The rules illustrated below are then applied until all regular expressions are expanded into symbols from the alphabet or epsilon.



For the expression **ba + ab***, the procedure is as follows



After expansion is done, the epsilon transitions are shortened. This is done by checking where each transition leads to. If a transition leads to a node with an epsilon, the destination of the epsilon is added to the previous node, essentially shortening the path. This is done until no new transitions are added, at which point all epsilon transitions are removed from all states except for the start state. This is the final form of the NFA used in this algorithm. An example is shown on the next page.



Internally, the NFA is stored as a list of states, each with a map that maps regular expressions to the indices of other states in the list. Because of this, it is tricky to remove useless states, so they are left alone. This is the case in the above example. The performance of the NFA construction is highly dependent on the structure of the parse tree. As a result, it is hard to calculate a meaningful time complexity.

(Why I'm Not) Constructing the DFA

Originally, the plan was to convert the NFA into a DFA, but the algorithm for constructing the delta function is quite inefficient. This is because the algorithm creates a new state for each possible combination of states in the NFA. In total, that means an NFA with n nodes would result in a DFA with 2^n nodes. The growth of this function is far too quick for this implementation to be practical.

Passing a String Through the NFA

The core idea of the DFA conversion algorithm is still being used to process the string. The only difference is, instead of precomputing the transitions for every combination of states, the combination of possible states will be calculated as the string is consumed. This allows for better performance, both in time and memory. This part has a complexity of about O(n), where n is the length of the input string.