15 Regular Expressions and Finite Automata

$\boxed{15.1}$ The Pattern Matching Problem

Problem: Recognizing a Word Pattern

Solution: Define a "Pattern" and a "Pattern Matching" Algorithm.

Here our patterns will be <u>regular expressions</u>. Examples:

`abc`	matches	`abc`
`a?c`	matches	`aac, abc, acc, ...`
`ab*c`	matches	`ac, abc, abbc, abbbc, ...`
`a+c`	matches	`abc, abbc, abbbc, ...`
`a(b|d)c`	matches	`abc, adc`
`(1|2)+`	matches	`1, 2, 11, 12, 21, 22, 111, ...`

15.2 Regular Expression (RE) Syntax and Examples

Regular Factors (RFs):	Examples
character-representative:	`a   b   c   0   1`
?	`?`
regular-factor*	`a*   (a|b)*   ?*`
regular-factor+	`a+   (a|b)+   ?+`
(regular-expression)	`(a|b)   (abc)   (a*b)`
Regular Terms (RTs):	
regular-factor regular-factor*	`abc   a(b|c)?   a*b+`
Regular Expressions (REs):	
regular-term { | regular-term }*	`a|b|c    a(b|c)?|a*b+`

? denotes 'any character'

* denotes 'zero or more'

| denotes 'either or'

\+ denotes 'one or more'

15.3 Regular Expressions (REs)

regular-expression ::= regular-term { | regular-term }* [either or]

regular-term ::= regular-factor regular-factor* [followed by]

regular-factor ::=
- character-representative
- | ? [any 1 character]
- | regular-factor* [0 or more]
- | regular-factor+ [1 or more]
- | (regular-expression)

character-representative ::=
<ASCII graphic character> except special-character

special-character ::= | | ? | * | + | (|) | #

RE ::= regular-expression

Comments:

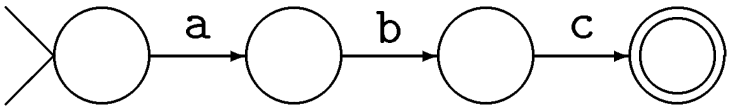
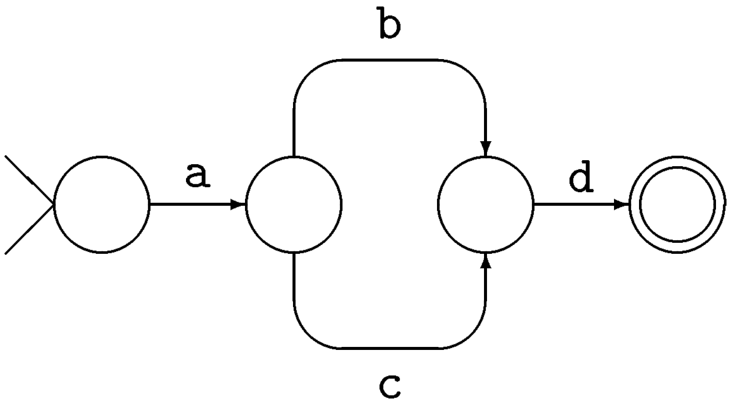
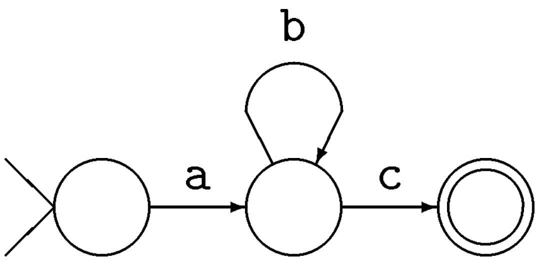
1. We do not permit special characters in the input string to be matched by an RE. The `#` character will be given a special use later.

2. Regular expressions are used by editor search commands. The `vi` editor uses them for example. However, `vi` has a somewhat different syntax for regular expressions, with main differences being that it uses `.` instead of `?`, and does not permit |, `+`, or parentheses. Also, one can write the equivalent of our RE `(a|b|c)` in `vi` notation as `[abc]`.

3. Regular expressions in the style of `vi` are also used by the UNIX `grep` command to search for lines in files that contain character sequences that match a pattern. The UNIX `egrep` program does the same but permits fancier regular expressions: in particular, |, `+`, and parentheses are permitted (but `.` is still used in place of `?`).

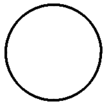
4. The UNIX shell uses a very limited form of regular expression to match file names. `?` is used to match any character, and `*` is used to match any string of zero or more characters: the equivalent of `?*` in our RE notation. As with `vi`, the UNIX shell permits the notation `[abc]` for what would be `(a|b|c)` in our RE notation.

15.4 Deterministic Finite Automata (DFA) Examples

RE	**DFA**
`abc`	
`a(b|c)d`	
`ab*c`	

$\boxed{15.5}$ DFA Definition

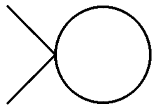
A DFA is a directed graph with:

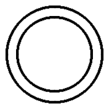
Nodes called ‘<u>states</u>’: 

Arrows called ‘<u>transitions</u>’: $\xrightarrow{\mathtt{x}}$

Arrows are labeled with single input characters.

No two arrows have same source state <u>and</u> label.

One state is called the ‘<u>initial state</u>’: 

One or more states are ‘<u>final states</u>’: 

$\boxed{15.6}$ DFA Execution Pseudo-Code

DFA-state ::= (graph, input-string,
pointer-to-current-graph-state)
input-string ::= input-character*

1. Start at the initial state.
2. Find the transition from the current graph state whose transition label is the first input character.

   If no such transition (or no input left), go to 4.
3. Change the current state to the target of the found transition, and delete the first input character.

   Then go back to 2.
4. If no input left and current state is a final state: announce SUCCESS; else announce FAILURE.

Comments:

5. A DFA is a machine, or simple computer, that runs an algorithm which matches (accepts, recognizes) character strings.

   The word automaton is a synonym for a simple machine that is self contained, and operates without human intervention.

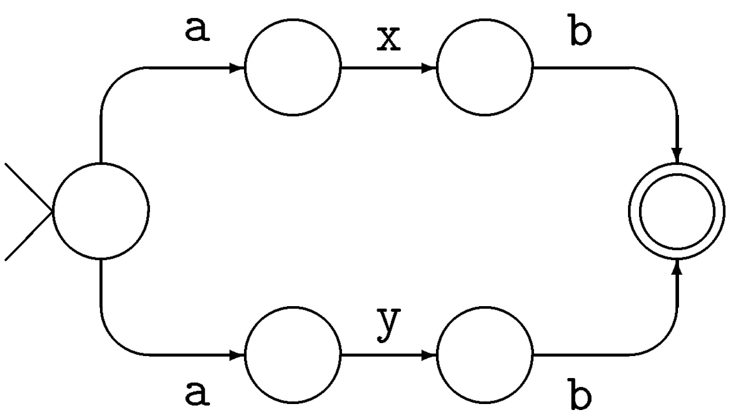
   The character strings are treated as input to the DFA. We assume there are no special characters (like `?`) in the input.

   It is also convenient when programming DFA's to use the NUL character, `'\0'`, as an end-of-string marker.

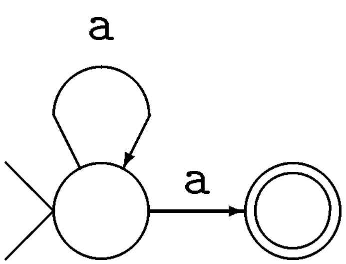
6. A DFA executes an instruction loop, in which the transitions behave like instructions that get executed. Each transition execution is called an execution step.

7. If two transitions had both the same label and the same source state, then an execution would be ambiguous if it arrived at the source state and the next input character was the label. Therefore, DFAs may not have two transitions with both the same label and the same source state.

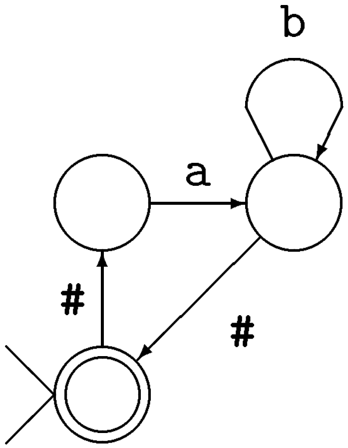
15.7 Where DFAs Are Not Adequate



(axb|ayb)



a*a



(ab*)*

# denotes the empty string,
which is also often denoted by ε.

$\boxed{15.8}$ NDFA Definition

A Non-Deterministic Finite Automaton (NDFA) is just like a DFA except:

> Two transitions can have the same source state and same label.
>
> A transition may be labeled with # to mean that when the transition is followed no character is to be removed from the input string.

Comments:

8. DFAs cannot easily implement some REs.

9. The first two examples of NDFAs are <u>not</u> DFAs because they have two transitions with the same source state and the same label.

   The third example has transitions labeled by the empty string.

   If you tried to make a DFA for `(ab*)*` by taking the DFA for `ab*` and merging its initial and final states, you get instead a DFA for `(a|b)*`, which permits strings beginning with `b`, that are not allowed by `(ab*)*`.

   Merging the initial and final states of the `ab*` DFA is like connecting these two states with `#` transitions going <u>both</u> ways, and this is a mistake. What is needed is to have a `#` transition going only one way, and for this one needs `#` transitions and not state mergings.

10. We need a more powerful machine (automaton) that permits two transitions to have the same label and source state, and that permits transitions labeled with the empty string.

    Such an automaton is called a <u>non-deterministic finite automaton</u>, or <u>NDFA</u>.

15.9 NDFA Execution Pseudo-Code

1. Start at the initial state.
2. Non-deterministically choose a transition whose source is the current graph state and whose transition label is either the first input character or **#**.
3. If no such transition, go to 5.
   If no input left, non-deterministically choose whether to go to 4 or 5.
4. Change the current state to the target of the chosen transition, and delete the first input character if it equals the transition label. Then go back to 2.
5. If no input left and current state is a final state: announce SUCCESS; else announce FAILURE.

$\boxed{15.10}$ Non-Deterministic Choice

A non-deterministic choice is a choice made by an omniscient oracle who is trying to SUCCEED if that is possible, and to FAIL as soon as possible otherwise.

One way to program this is to try all possible choices: i.e. to execute a search of the choice tree.

A better way to program this is to execute a kind of parallel search that computes for each initial segment of the input string the set of states that can be reached after deleting that segment from the input.

Comments:

11. We have previously studied breadth first search, depth first search, and random search.

    Now we have a new kind of search, <u>oracle search</u>, in which the choices are made by an all knowing oracle that leads us through the search tree by the shortest path.

12. Many algorithms can be written that make use of non-deterministic choices. All these algorithms can be recast as search algorithms.

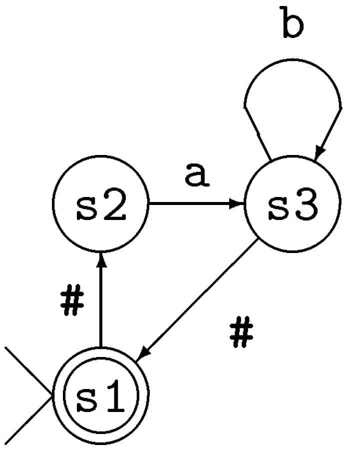
    You can think of an algorithm with non-deterministic choices as an abstraction of an algorithm with some deterministic rule for making these choices. The rule for making choices is ‘unspecified’, or ‘to be determined later’.

$\boxed{15.11}$ Parallel Search

NDFA-state ::= (graph, input-string, current-set-of-states)

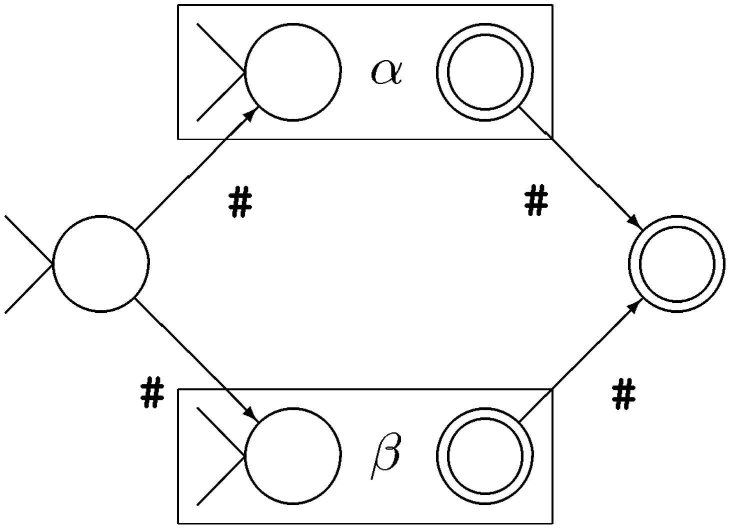
1. Set the current state set to contain just the initial state.
2. Add to the current state set all states reachable from states already in the current state set by transitions labeled with **#**.
3. If no input left or the current state set is empty, announce SUCCESS if current state set holds any final state, and FAIL otherwise.
4. Replace the current state set by the set of all states that can be reached by starting at a state in the current set and following a transition whose label is the first input character. Then delete the first input character. Lastly go back to 2.

15.12 Parallel Search Example

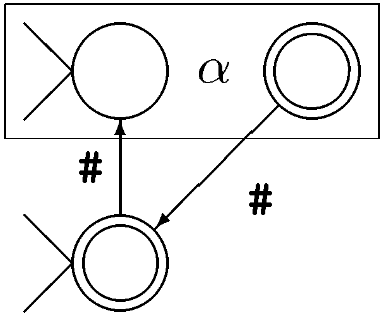


	Current State Set At		
input string	start of step 2	end of step 2	end of step 4
aba	{ s1 }	{ s1, s2 }	{ s3 }
ba	{ s3 }	{ s1, s2, s3 }	{ s3 }
a	{ s3 }	{ s1, s2, s3 }	{ s3 }
	{ s3 }	{ s1, s2, s3 }	
		s1 is final: SUCCESS	

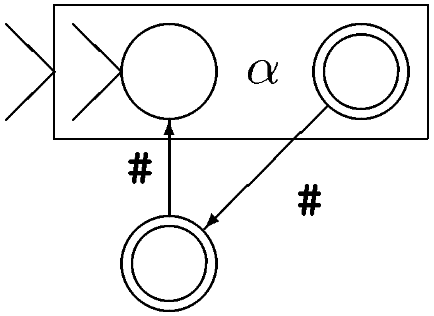
$\boxed{15.13}$ Building NDFAs From REs



$\alpha \,|\, \beta$



α*



α+

There are <u>never</u> more than two transitions leaving a state.
There is <u>at most one</u> non-# transition leaving any state.
There is <u>never</u> more than one transition leaving a final state.

Comments:

13. Given an RE you can recursively build an NDFA for it using the rules given in the viewgraph.

14. For a character, just build the DFA to recognize that character.

15. For |, *, and + add new states and #-transitions as indicated in the viewgraph.

16. At each stage of the building process the following rules will be true:

    1. There are no more than 2 transitions out of each state.

       Suppose we identify these as the first and second transition from the state.

    2. The second transition from a state, if it exists, is always a # transition.

    3. Each NDFA has a single final state.

    4. The second transition from the final state is unused.

17. You can verify these rules by checking them recursively: check that they are true for an RE consisting of a single character; then check for an RE of the form $\alpha|\beta$, assuming they are true for the REs α and β; and lastly check for REs of the forms $\alpha*$ and $\alpha+$ assuming the rules are true for the RE α.

18. These rules make it easier to code the data structures required to program NDFAs built as we have described.