Nondeterministic Finite Automata (NFA):

Nondeterministic Finite Automata (NFA) states of an automaton of this kind may or may not have a transition for each symbol in the alphabet, or can even have multiple transitions for a symbol. The automaton accepts a word if there exists at least one path from q0 to a state in F labeled with the input word. If a transition is undefined, so that the automaton does not know how to keep on reading the input, the word is rejected.

Formulas:

An *NFA* is represented formally by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, consisting of

- a finite set of states Q
- a finite set of input symbols Σ
- a transition function $\Delta : Q \times \Sigma \rightarrow P(Q)$.
- an *initial* (or *start*) state $q_0 \in Q$
- a set of states F distinguished as *accepting* (or *final*) *states* $F \subseteq Q$.

Here, $P(Q)$ denotes the power set of Q. Let $w = a_1a_2 \ldots a_n$ be a word over the alphabet Σ. The automaton M accepts the word w if a sequence of states, $r_0, r_1, \ldots, r_n$, exists in Q with the following conditions:

1. $r_0 = q_0$
2. $r_{i+1} \in \Delta(r_i, a_{i+1})$, for $i = 0, \ldots, n-1$
3. $r_n \in F$

In words, the first condition says that the machine starts in the start state q_0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function Δ.

The last condition says that the machine accepts w if the last input of w causes the machine to halt in one of the accepting states. In order for w being accepted by M it is not required that every state sequence ends in an accepting state, it is sufficient if one does. Otherwise, i.e. if it is impossible at all to get from q_0 to a state from F by following w, it is said that the automaton *rejects* the string. The set of strings M accepts is the language *recognized* by M and this language is denoted by $L(M)$.

We can also define $L(M)$ in terms of $\Delta^*: Q \times \Sigma^* \rightarrow P(Q)$ such that:

1. $\Delta^*(r, \varepsilon) = \{r\}$ where ε is the empty string, and
2. If $x \in \Sigma^*$, $a \in \Sigma$, and $\Delta^*(r, x) = \{r_1, r_2, \ldots, r_k\}$ then $\Delta^*(r, xa) = \Delta(r_1, a) \cup \ldots \cup \Delta(r_k, a)$.

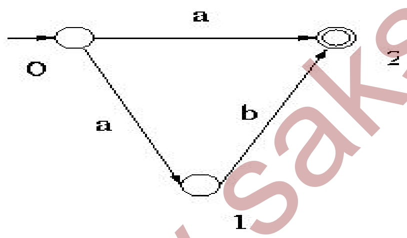
Now $L(M) = \{w \mid \Delta^*(q_0, w) \cap F \neq \emptyset\}$.

Note that there is a *single initial state*, which is not necessary. Sometimes, NFAs are defined with a set of initial states. There is an easy construction that translates a NFA with multiple initial states to a NFA with single initial state, which provides a convenient notation. For a more elementary introduction of the formal definition see automata theory

Q = { 0, 1, 2 }, Σ = { a, b }, A = { 2 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	A	{ 1 , 2 }
0	B	$\emptyset$
1	A	$\emptyset$
1	B	{ 2 }
2	A	$\emptyset$
2	B	$\emptyset$

Note that for each state there are two rows in the table for δ corresponding to the symbols a and b. A state transition diagram for this finite automaton is given below.



Regular Expression to NFA:

It is proven (Kleene's Theorem) that RE and FA are equivalent language definition methods. Based on this theoretical result practical algorithms have been developed enabling us actually to construct FA's from RE's and simulate the FA with a computer program using Transition Tables. In following this progression an NFA is constructed first from a regular expression, then the NFA is reconstructed to a DFA, and finally a Transition Table is built.

The Thompson's Construction Algorithm is one of the algorithms that can be used to build a Nondeterministic Finite Automaton (NFA) from RE, and Subset construction Algorithm can be applied to convert the NFA into a Deterministic Finite Automaton (DFA).

The last step is to generate a transition table. We need a finite state machine that is a deterministic finite automaton (DFA) so that each state has one unique edge for an input alphabet element. So that for code generation there is no ambiguity. But a nondeterministic finite automaton (NFA) with more than one edge for an input alphabet element is easier to construct using a general algorithm - Thompson's construction. Then following a standard procedure, we convert the NFA to a DFA for coding.

Regular expression

Consider the regular expression

r = (a|b)*abb, that matches {abb, aabb, babb, aaabb, bbabb, ababb, aababb,……}

To construct a NFA from this, use Thompson's construction. This method constructs a regular expression from its components using ε-transitions. The ε transitions act as "glue or mortar" for the subcomponent NFA's. An ε-transition adds nothing since concatenation with the empty string leaves a regular expression unchanged (concatenation with ε is the identity operation).

Parse the regular expression into its subexpressions involving alphabet symbols a and b and ε: ε, a, b, a|b, ()*, ab, abb

These describe

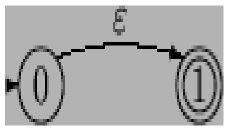
- a regular expression for single characters ε, a, b
- alternation between a and b representing the union of the sets: L(a) U L(b)
- Kleene star ()*
- concatenation of a and b: ab, and also abb

Subexpressions of these kinds have their own Nondeterministic Finite Automata from which the overall NFA is constructed. Each component NFA has its own start and end accepting states. A Nondeterministic Finite Automata (NFA) has a transition diagram with possibly more than one edge for a symbol (character of the alphabet) that has a start state and an accepting state. The NFA definitely provides an accepting state for the symbol.

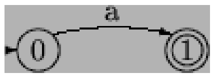
Converting a regular expression to a NFA - Thompson's Algorithm

We will use the rules which defined a regular expression as a basis for the construction:

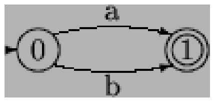
1. The NFA representing the empty string is:



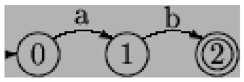
2. If the regular expression is just a character, eg. a, then the corresponding NFA is :



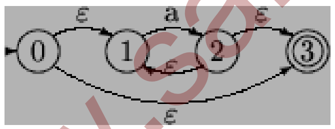
3. The union operator is represented by a choice of transitions from a node; thus a|b can be represented as:



4. Concatenation simply involves connecting one NFA to the other; eg. ab is:



5. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus a* looks like:



Converting NFA to DFA:

This section specifically describes how one may transform any nondeterministic finite automaton (NFA) into a deterministic automaton (DFA) by using the tools under the “Convert → Convert to DFA” menu option. For any string x, there may exist none or more than one path from initial state and associated with x.

The Von Neumann principle for the design and operation of computers requires that a program has to be primary memory resident to execute. Also, a user requires to revisit his programs often during its evolution. However, due to the fact that primary memory is volatile, a user needs to store his program in some non-volatile store. All computers provide a non-volatile secondary memory available as an online storage.

Programs and files may be disk resident and downloaded whenever their execution is required. Therefore, some form of memory management is needed at both primary and secondary memory levels. Secondary memory may store program scripts, executable process images and data files. It may store applications, as well as, system programs.

In fact, a good part of all OS, the system programs which provide services (the utilities for instance) are stored in the secondary memory. These are requisitioned as needed. The main motivation for management of main memory comes from the support for multiprogramming.

Several executables processes reside in main memory at any given time. In other words, there are several programs using the main memory as their address space.
Also, programs move into, and out of, the main memory as they terminate, or get suspended for some IO, or new executables are required to be loaded in main memory. So, the OS has to have some strategy for main memory management. In this chapter we shall discuss the management issues and strategies for both main memory and secondary memory.

Main Memory Management

Let us begin by examining the issues that prompt the main memory management.

Allocation: First of all the processes that are scheduled to run must be resident in the memory. These processes must be allocated space in main memory.

Swapping, **fragmentation and compaction**: If a program is moved out or terminates, it creates a hole, (i.e. a contiguous unused area) in main memory. When a new process is to be moved in, it may be allocated one of the available holes. It is quite possible that main memory has far too many small holes at a certain time. In such a situation none of these holes is really large enough to be allocated to a new process that may be moving in. The main memory is too Operating Systems/Memory management Lecture Notes PCP Bhatt/IISc, Bangalore M4/V1/June 04/2 fragmented. It is, therefore, essential to attempt compaction. Compaction means OS re-allocates the existing programs in contiguous regions and creates a large enough free area for allocation to a new process.

Garbage collection: Some programs use dynamic data structures. These programs dynamically use and discard memory space. Technically, the deleted data items (from a dynamic data structure) release memory locations. However, in practice the OS does not collect such free space immediately for allocation. This is because that affects performance. Such areas, therefore, are called garbage. When such garbage exceeds a certain threshold, the OS would not have enough memory available for any further allocation. This entails compaction (or garbage collection), without severely affecting performance.

Protection: With many programs residing in main memory it can happen that due to a programming error (or with malice) some process writes into data or instruction area of some other process. The OS ensures that each process accesses only to its own allocated area, i.e. each process is protected from other processes.

Virtual memory: Often a processor sees a large logical storage space (a virtual storage space) though the actual main memory may not be that large. So some facility needs to be provided to translate a logical address available to a processor into a physical address to access the desired data or instruction.

IO support: Most of the block-oriented devices are recognized as specialized files. Their buffers need to be managed within main memory alongside the other processes. The considerations stated above motivate the study of main memory management. One of the important considerations in locating an executable program is that it should be possible to relocate it any where in the main memory. We shall dwell upon the concept of relocation next.

Code Generation:

In computing, code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine. The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three-address code.

Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

Target language: The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

IR Type: Intermediate representation has various forms. It can be in Abstract Syntax Tree AST structure, Reverse Polish Notation, or 3-address code. Selection of instruction : The code generator takes Intermediate Representation as input and converts maps it into target machine's instruction set. One representation can have many ways instructions to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

Register allocation: A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values. Ordering of instructions : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors: The code generator has to track both the registers foravailability and addresses locationofvalues while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor:** Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

- **Address descriptor:** Values of the names identifiers used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations. Code generator keeps both the descriptor updated in real-time.

For a load statement, LD R1, x, the code generator: updates the Register Descriptor R1 that has value of x and updates the Address Descriptor x to show that one instance of x is in R1.

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input. Note : If the value of a name is found at more than one place register, cache, ormemory, the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg: Code generator uses getReg function to determine the status of available registers and the location of name values.

getReg works as follows: If variable Y is already in register R, it uses that register. Else if some register R is available, it uses that register. Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions. For an instruction x = y OP z, the code generator may perform the following actions.

- Let us assume that L is the location preferablyregister where the output of y OP z is to be saved: Call function getReg, to decide the location of L.
- Determine the present location registerormemory of y by consulting the Address Descriptor of y. If y is not presently in register L, then generate the following instruction to copy the value of y to L: MOV y', L where y' represents the copied value of y.
- Determine the present location of z using the same method used in step 2 for y and generate the following instruction: OP z', L where z' represents the copied value of z.
- Now L contains the value of y OP z, that is intended to be assigned to x. So, if L is a register, update its descriptor to indicate that it contains the value of x.
- Update the descriptor of x to indicate that it is stored at location L. If y and z has no further use, they can be given back to the system.

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machineor Deterministic Finite Automaton.

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q_0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of a DFA

A DFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

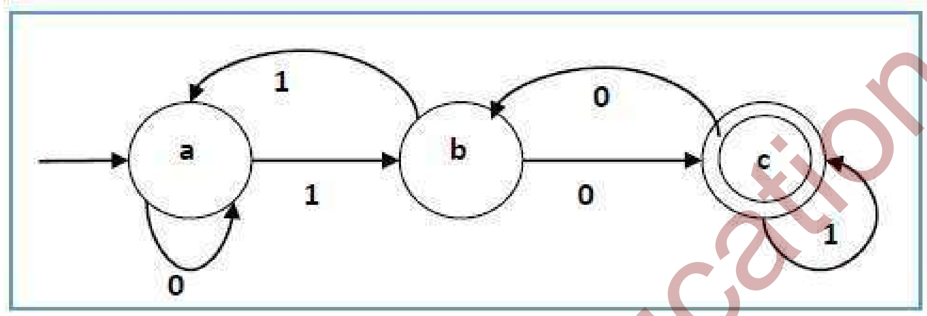
Example

Let a deterministic finite automaton be →

- Q = {a, b, c},
- ∑ = {0, 1},
- q_0={a},
- F={c}, and
- Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
A	A	b
B	C	a
C	B	c

Its graphical representation would be as follows −



Problem Statement

Let X = (Q_x, ∑, δ_x, q_0, F_x) be an NDFA which accepts the language L(X). We have to design an equivalent DFA Y = (Q_y, ∑, δ_y, q_0, F_y) such that L(Y) = L(X). The following procedure converts the NDFA to its equivalent DFA −

Algorithm

Input: An NDFA

Output: equivalent DFA

Step 1 Create state table from the given NDFA.

Step 2 Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 Mark the start state of the DFA by q_0 (Same as the NDFA).

Step 4 Find out the combination of States {Q_0, Q_1,... , Q_n} for each possible input alphabet.

Step 5 Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

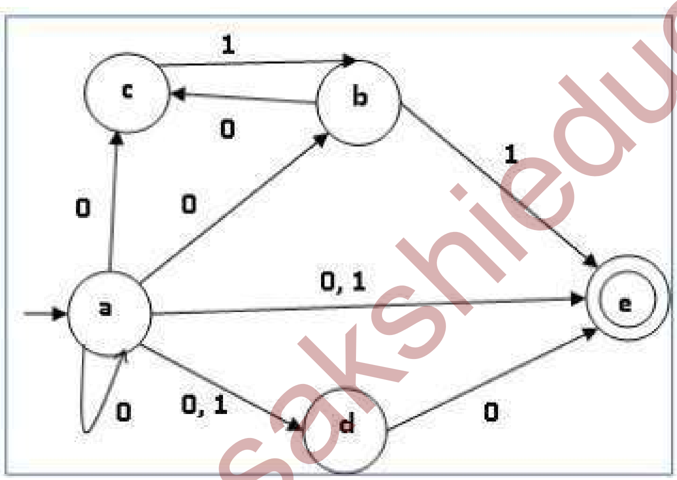
Step 6 The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Example

The NDFA table is as follows −

q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

Let us consider the NDFA shown in the figure below.

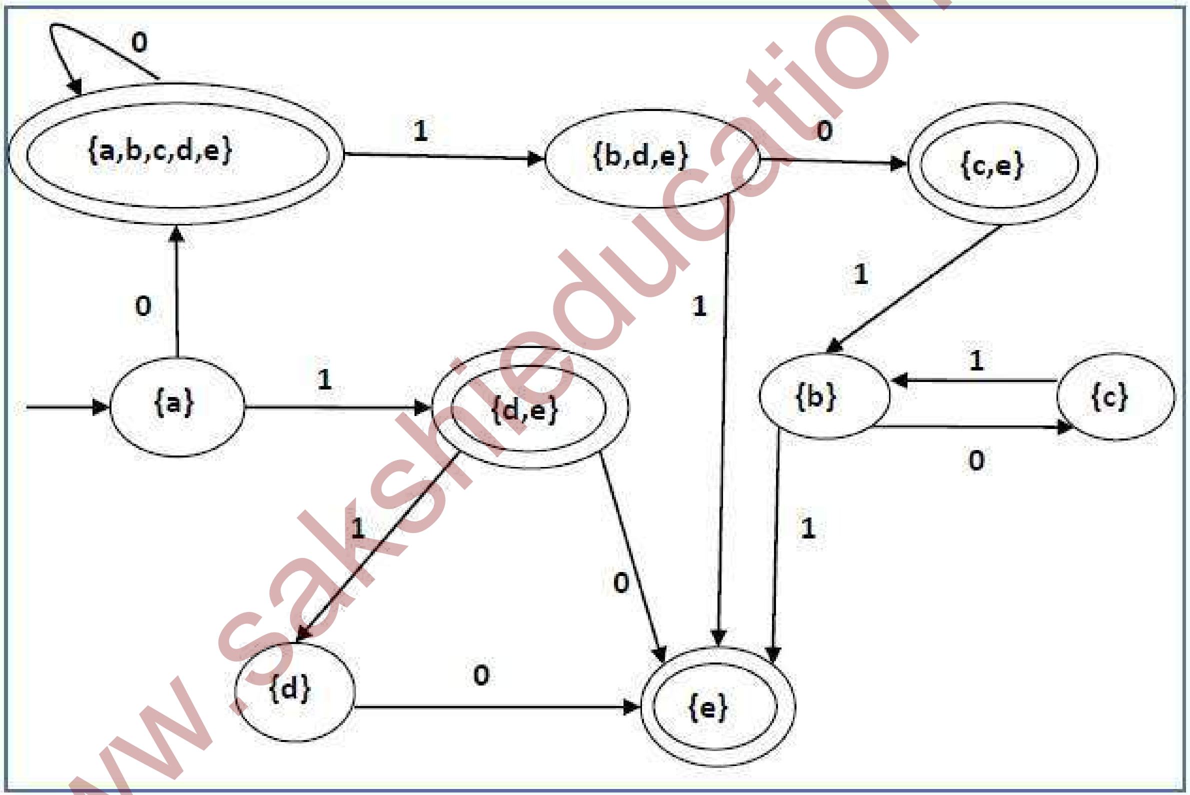


Using above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

Q	δ(q,0)	δ(q,1)
A	{a,b,c,d,e}	{d,e}
{a,b,c,d,e}	{a,b,c,d,e}	{b,d,e}
{d,e}	E	D
{b,d,e}	{c,e}	E

E	∅	∅
D	E	∅
{c,e}	∅	B
B	C	E
C	∅	B

The state diagram of the DFA is as follows −



Runtime Environment:

- Before code generation static source text of a program needs to be related to the actions that must occur at runtime to implement the program.
- As execution proceeds same name in the source text can denote different data objects in the target machine.
- The allocation and de allocation of data objects is managed by the runtime support package.

- Each execution of a function is referred to as an activation of the procedure/function
- If the function is recursive several of its activations may be alive at the same time
- A procedure is *activated* when called
- The *lifetime* of an activation of a procedure is the sequence of steps between the first and last steps in the execution of the procedure body
- A procedure is *recursive* if a new activation can begin before an earlier activation of the same procedure has ended