Compiler Design

Dr. Ibrahim Elgendy

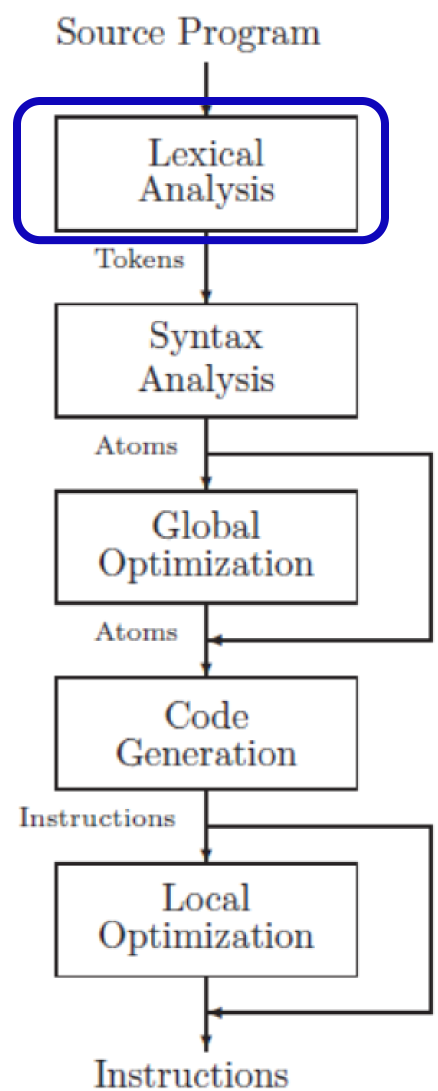
Lecturer at Computer Science Department, Faculty of Computers and Information, Menoufia University

2022-2023

Phases of Compilers

Phases of Compilers

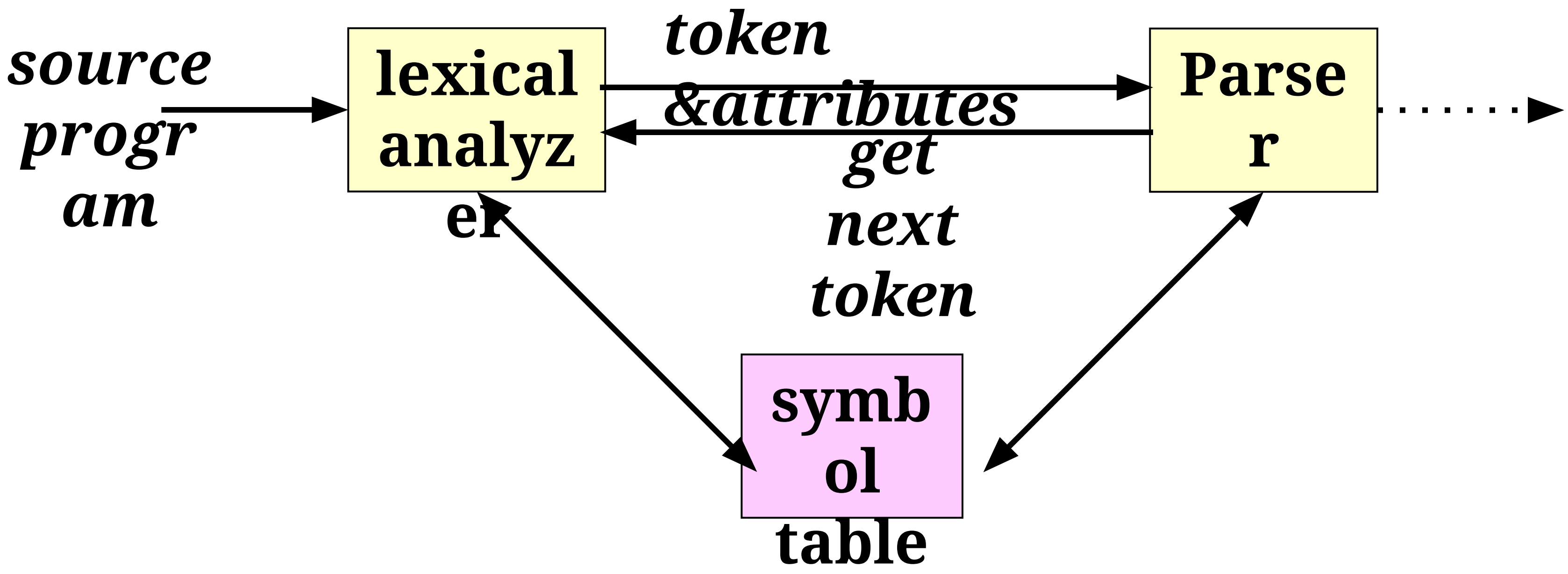
- Lexical Analysis (Scanner)
- Syntax Analysis Phase
- Global Optimization
- Code Generation
- Local Optimization



Outline

- **Finite State Machine.**
- Regular Expression.
- Implementation with Finite State Machines.

The Role of a Lexical Analyzer



The Role of a Lexical Analyzer

What do we want to do? **Example**:

if (i == j)

Z = 0;

else

Z = 1;

- The input is just a string of characters:

\t if (i == j) \n \t \t z = 0;\n \t else \n \t \t z = 1;

- **Goal**: Partition input string into substrings
  - ✔ Where the substrings are tokens

What's a Token?

- In English:
  - ✔ noun, verb, adjective, …
- In a programming language:
  - ✔ Identifier, Keyword, Operator, Special Character

- Before getting into lexical analysis we need to cover the concepts of **finite state machine** and **regular expression** which are critical to the design of the lexical analyzer.

Finite State Machine

Symbol

A Symbol is an abstract entity that has no meaning by itself.

Example:

Letters: **A to Z** (upper case) or **a to z** (lower case).

Digits: **0 to 9**

Special Characters: such as **$ % & * +**

Alphabet

An Alphabet is a non-empty finite set of symbols. It is denoted by the symbol Σ (sigma).

Example:

- **Σ ={a, b, c} is an alphabet which consisting of letters ‘a’, ‘b’ and ‘c’.**
- **Σ ={a, b, …, z} is an alphabet which consisting of lower-case letters.**
- **Σ ={0, 1} is an alphabet which consisting of binary numbers.**

String

A string **or** **word** is defined as finite sequence of symbols over an alphabet (**Σ**).

Example:

1. **Alphabet Σ ={a, b}:**

Strings: w ={a, b, aa, ab, ba, bb, ….}.

2. **Alphabet Σ ={0, 1}:**

Strings: w ={0, 1, 00, 01, 10, 11, ….}.

3. **ϵ** is null strings. ➔ string with zero character.

Set

A **set** is collection of unique objects.

Example:

- S ={Java, C++, C#, Basic, Python}.
- {} or Ø is empty set.

Language

A language is a **set** of **strings** of **symbols** from some one **alphabet** (**Σ**), are **well-formed** according to a **specific set of rules**.

Examples:

The **set of palindromes** over the alphabet **Σ={0, 1}** **is an** infinite language.

The language are: **{ε, 0, 1, 11, 010, 101, 00100, ...}**

What the set of all strings (language) **$L=\{a^nb^n \mid n>0\}$** over an alphabet **Σ={a, b}**?

L={ab, aabb, aaaabbbb,.... }

What is the Finite State Machine?

A Finite State Machine is a machine that represents a **mathematical model** of computation, which we will describe in mathematical terms and its operation should be perfectly clear.

It can be defined as a **5-tuple** denoted by **M**, i.e., $M=(Q, \Sigma, \delta, q_0, F)$, where:

Q : Finite or non-empty set of **States**.

Σ : Input Alphabet.

q_0 **:** **Initial State** or **Start State** and q_0 is in **Q**, i.e., $q_0 \in Q$ (In any Automata initial or start state is only one).

F : Set of **Final** or **Accepting States**, $F \subseteq Q$.

δ : **Transition function** or **mapping function** which determines the next state. It maps from $Q \times \Sigma$ to **Q** i.e.,

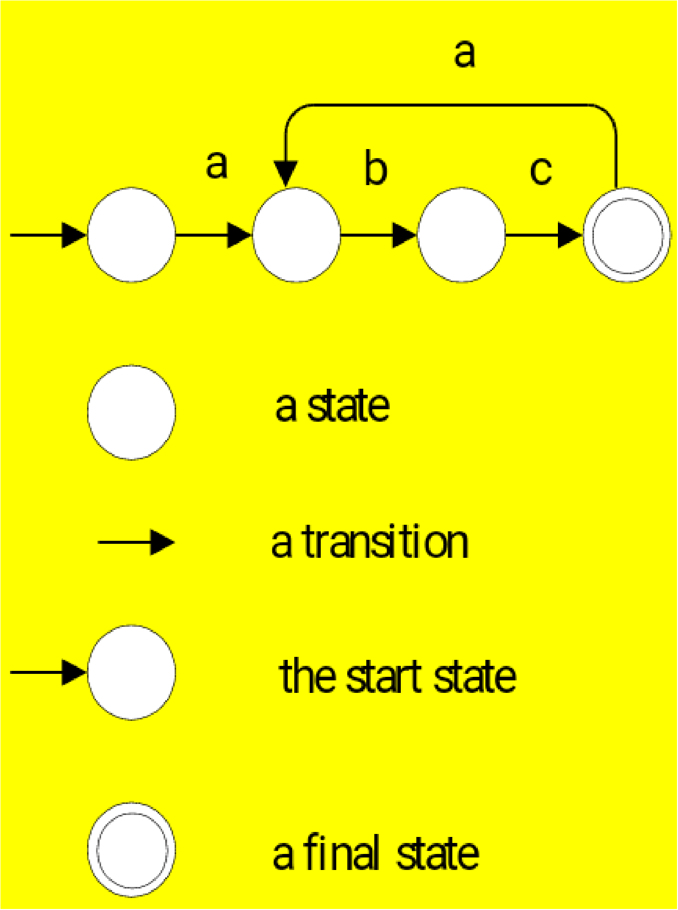
$$\delta : Q \times \Sigma \rightarrow Q$$

Transition diagram

Finite State Machine can be represented by **transition diagram or transition table.**

$$M=(Q, \Sigma, \delta, q_0, F)$$

- In **transition diagram**, each **state** is represented by a **circle**, and the **transition function** is represented by **arcs** labeled by input symbols leading from one state to another.
- The **starting state** is indicated by an **arc** with no state and the **accepting** states are **double circles**.



Finite State Machine

Transition table

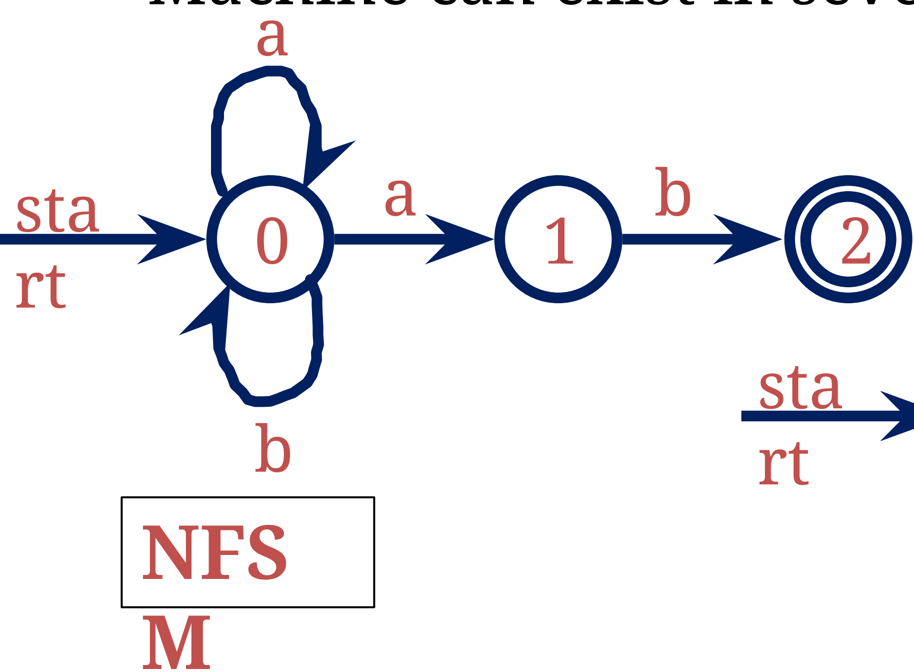
$$M=(Q, \Sigma, \delta, q_0, F)$$

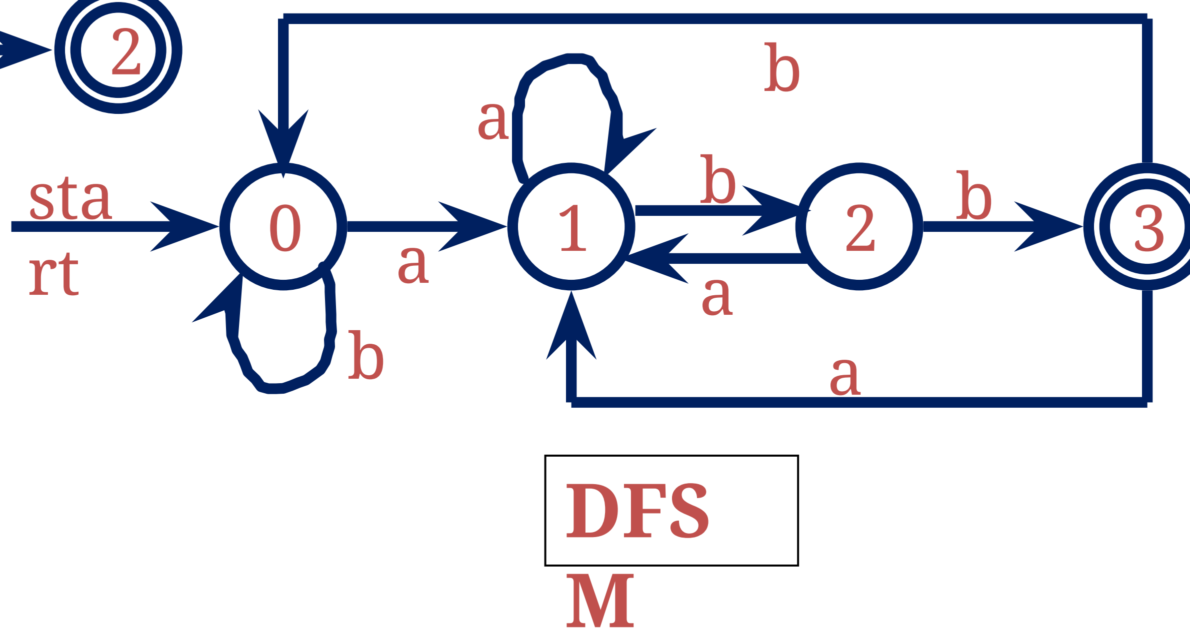
- In **transition table**, each **row** indicates the **states** in finite automate and each **column** is a letter of **input alphabet** (Σ).
- In addition, **start state** is represented by **drawing a prefixed arrow** to that state and **final state** is denoted by **star**.

	0	1
A	D	B
B	C	B
* C	C	B
D	D	D

NFSM and DFSM

- **Deterministic Finite State Machine (DFSM):** Machine can exist in only one state at any given time.
- **Non-deterministic Finite State Machine (NFSM):** Machine can exist in several states at the same time.

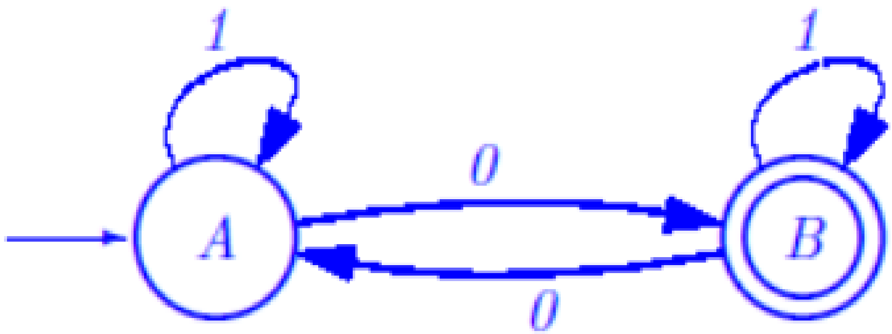




Finite State Machine

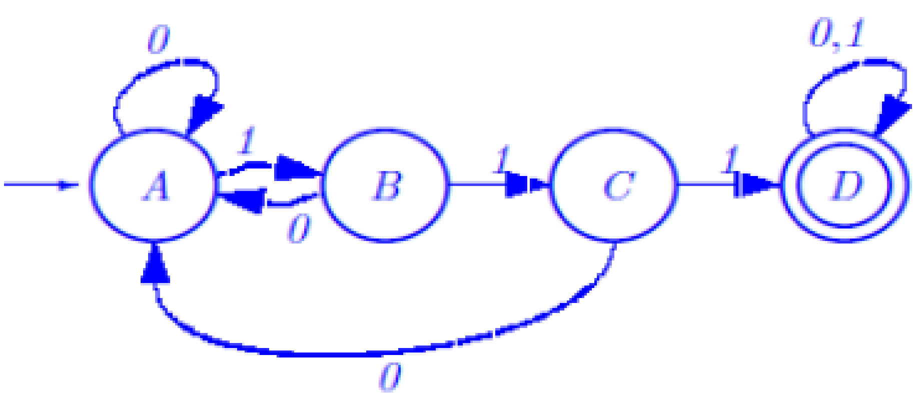
Examples: $\Sigma=\{0, 1\}$

1. Show a finite state machine in either state graph or table form for Strings containing an



	0	1
A	B	A
*B	A	B

2. Show a finite state machine in either state graph or table form for Strings containing .

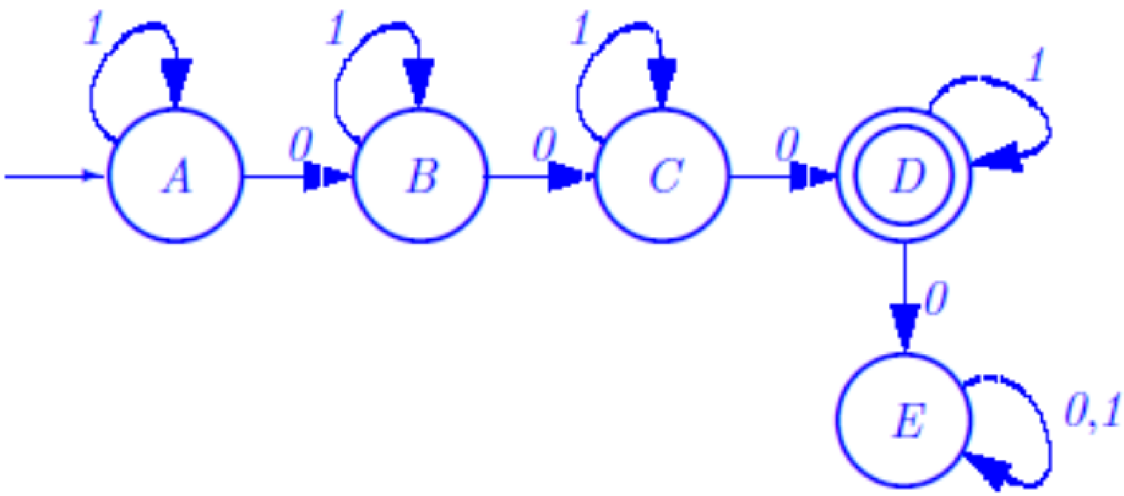


	0	1
A	A	B
B	A	C
C	A	D
*D	D	D

Finite State Machine

Examples: $\Sigma=\{0, 1\}$

3. Show a finite state machine in either state graph or table form for Strings containing **exactly three zeros.**

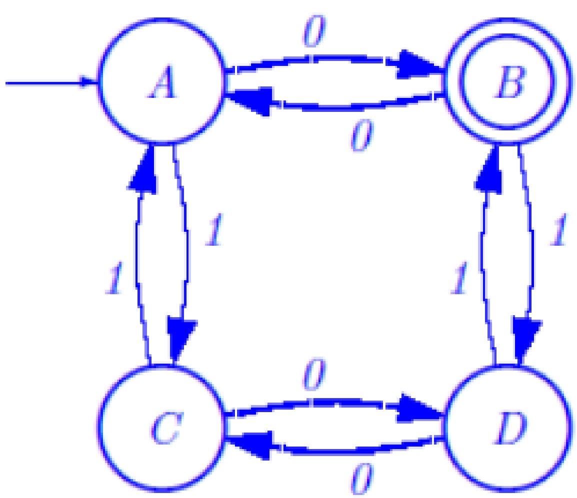


	0	1
A	B	A
B	C	B
C	D	C
*D	E	D
E	E	E

Finite State Machine

Examples: $\Sigma=\{0, 1\}$

4. Show a finite state machine in either state graph or table form for Strings containing an **odd number of zeros and an even number of ones**.

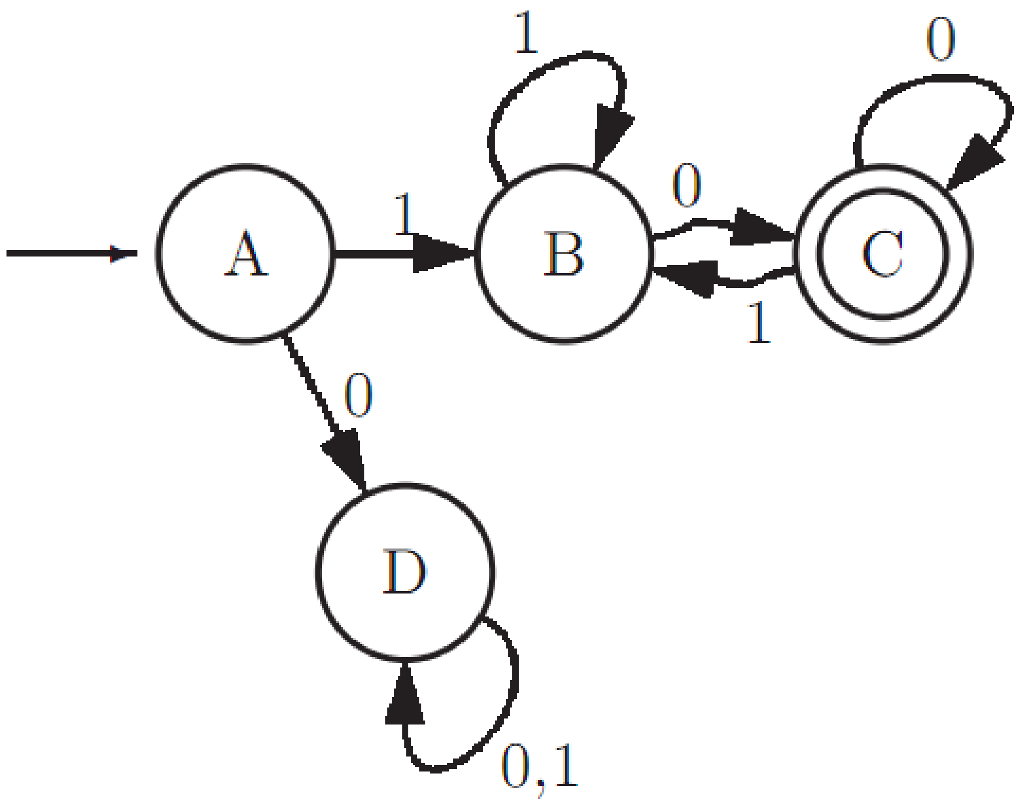


	0	1
A	B	C
*B	A	D
C	D	A
D	C	B

Finite State Machine

Examples: $\Sigma=\{0, 1\}$

5. Show a finite state machine in either state graph or table form for Strings that **begins with a one and ends with zero.**



Regular Expression

Regular Expressions

Regular Expression is another method for specifying languages that use patterns. This pattern can consist of three possible operations on languages – (**union, concatenation**, and **Kleene star**):

- ***Union*** –The union of two sets is that set that contains all the elements in each of the two sets and nothing else. And it is designated with a '+'.

For example: {abc, ab, ba} + {ba, bb} = {abc, ab, ba, bb}

- ***Concatenation*** –concatenating each string in one set with each string in the other set. And it is designated with a '.'

For example, {ab, a, c} . {b} = {ab.b, a.b, c.b} = {abb, ab, cb}

- ***Kleene **** -generates zero or more concatenations of strings from the language to which it is applied. And it is designated with a '*'

Precedence *Kleene* □ Concatenation □ Union*

For example, a* = {ε, a, aa, aaa, aaaa, aaaaa,

Example

Examples: $\Sigma = \{a, b\}$

$\mathbf{r} = \mathbf{a+b} \Leftrightarrow \{a, b\}$

$\mathbf{r} = \mathbf{(a+b)(a+b)} \Leftrightarrow \{aa, ab, ba, bb\}$

$\mathbf{r} = \mathbf{a^*} \Leftrightarrow \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

Example

An example of a regular expression is: **(0+1)***
L={ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, ...}
This is the set of all strings of zeros and ones.

Another example: **1(0+1) *0**
L= {10, 100, 110, 1000, 1010, 1100, 1110, ...}
This is the set of all strings of zeros and ones which begin with a 1 and end with a 0.

Example

For each of the following regular expressions, list six strings which
are in its language.

1. $(a(b+c)^*)^*d$
2. $(a+b)^*(c+d)$
3. $(a^*b^*)^*$

1. d ad abd acd aad abbcbd

2. c d ac abd babc bad

3. ε a b ab ba aa

Example

Give a regular expression for each of the languages:

1. **Strings containing an odd number of zeros.**
2. **Strings containing three sequential ones.**
3. **Strings containing exactly three zeros.**
4. **Strings that begins of 1 and end with zero.**

1. **1*01*(01*01*)***
2. **(0+1)*111(0+1)***
3. **1*01*01*01***
4. **1(0+1)*0**

Example

- Suppose L1 represents the set of all strings from the alphabet **0,1** which contain an **even number of ones** (even parity). Which of the following strings belongs to L1?

a) 0101
b) 110211
c) 000
d) 010011
e) ε

(a, c, e)

Example

- Suppose L2 represents the set of all strings from the alphabet **a,b,c** which contains an **equal number of a's, b's, and c's**. Which of the following strings belong to L2?

a) bca
b) accbab
c) ε
d) aaa
e) aabbcc

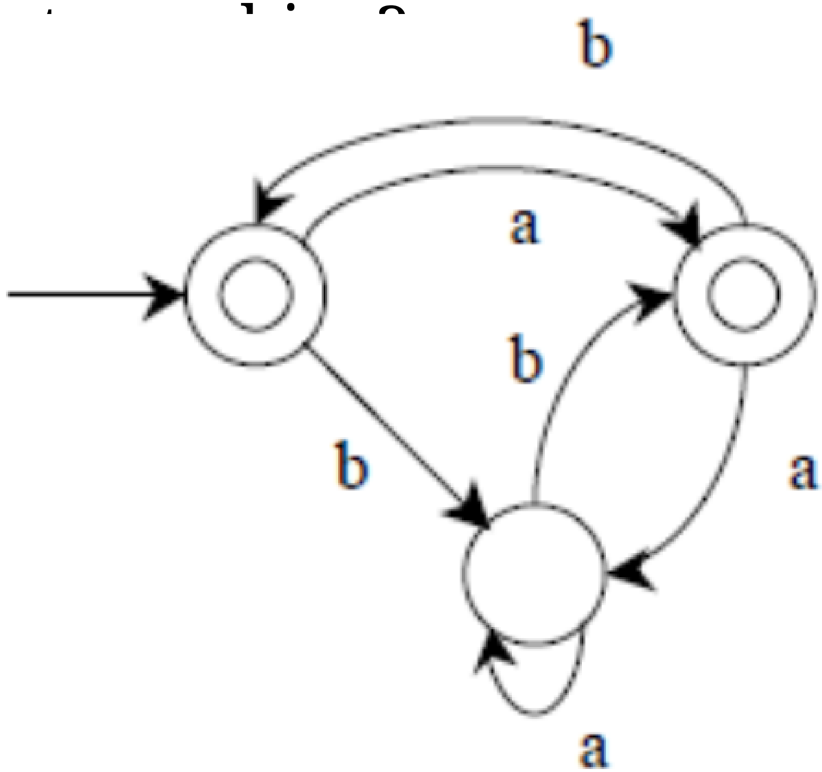
(a, b, c, e)

Regular Expressions

Example

- Which of the following strings are in the language specified by this finit

a) abab
b) bbb
c) aaab
d) Aaa
e) ε



(a, b, c, e)

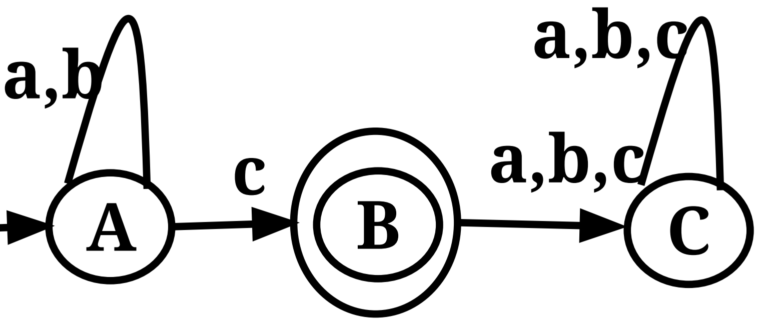
Regular Expressions

Example

- Construct finite state machines which specify the same language as each of the following regular expressions.

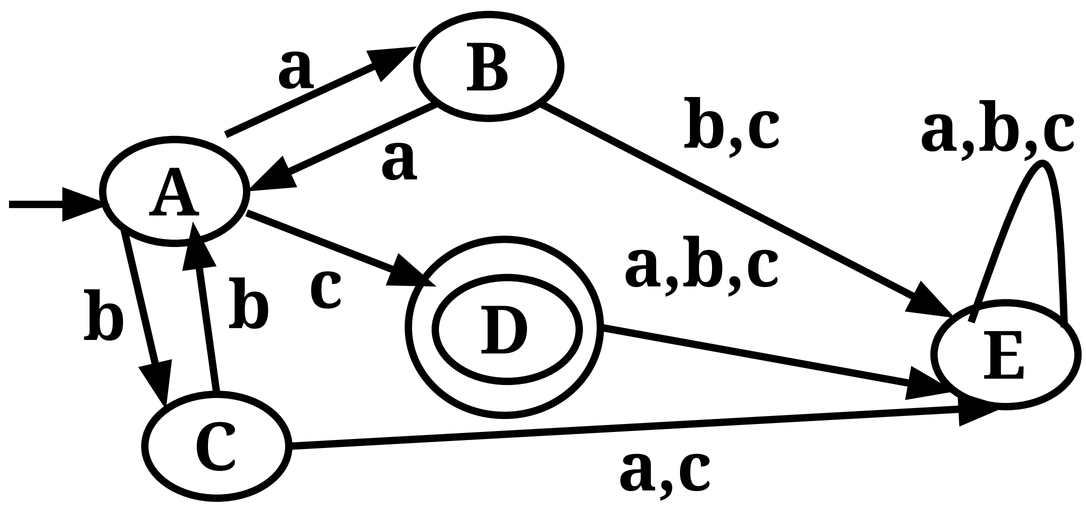
1. (a+b)*c

	a	b	c
A	A	A	B
*B	C	C	C
C	C	C	C



2. (aa)*(bb)*c

	a	b	c
A	B	C	D
B	A	E	E
C	E	A	E
*D	E	E	E
E	E	E	E



Example

An example of Java source input, showing the word boundaries and types is given below:

```
while ( x33 <= 2.5e+33 - total ) calc ( x33 ) ; //!
  1   6  2   3     4     3   2   6  2   6  2  6 6
```

The output of this phase is a **stream of tokens**, one token for each word encountered in the input program.

Each **token** consists of two parts:

1. **class** indicating which kind of token

Token Class	Token Value
1	[code for while]
6	[code for (]
2	[ptr to symbol table entry for x33]
3	[code for <=]
4	[ptr to constant table entry for 2.5e+33]
3	[code for -]
2	[ptr to symbol table entry for total]
6	[code for)]
2	[ptr to symbol table entry for calc]
6	[code for (]
2	[ptr to symbol table entry for x33]
6	[code for)]
6	[code for ;]

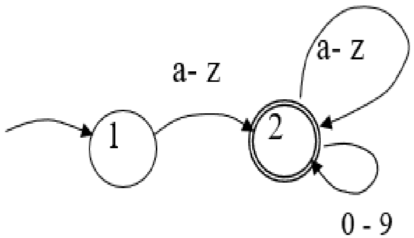
Example

For each of the following Java input strings show the word boundaries and token classes (for those tokens which are not ignored) if /*if* /a)}+whiles

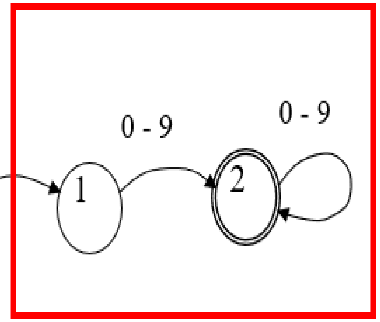
if	/*if*/	a	)	}	+	whiles
1		2	6	6	3	2

Note that the lexical analysis phase does not check for proper syntax.

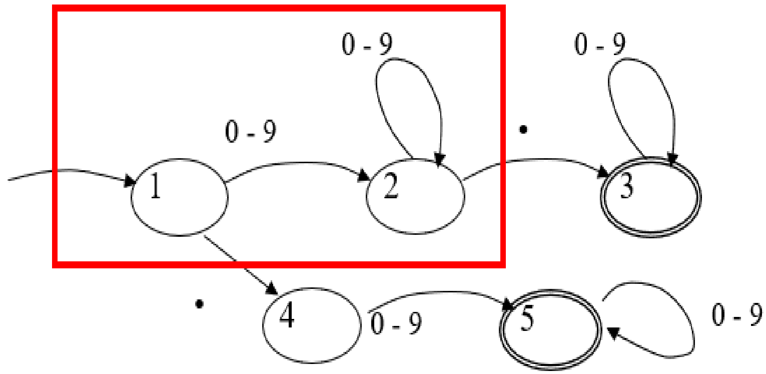
Examples of Finite State Machines for Lexical Analysis



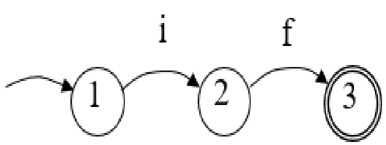
ID



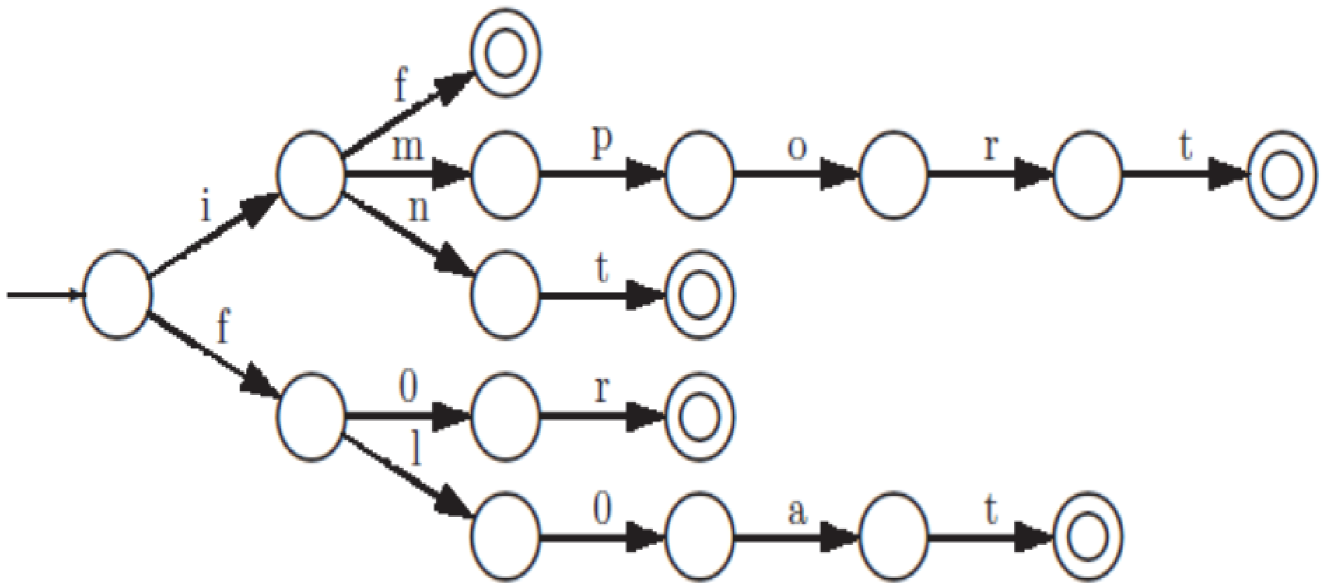
NUM



REAL



IF



This machine accepts the keywords if, int, import, for, float.

Implementation with Finite State Machines

Implementation with Finite State Machines

- A finite state machine can be implemented very simply **by an array in which there is a row for each state of the machine and a column for each possible input symbol.**
- This array will look very much like the table form of the

```
boolean [] accept = new boolean [STATES];
int [][] fsm = new int[STATES][INPUTS];           // state table
// initialize table here...
int inp = 0;                                      // input symbol (0..INPUTS)
int state = 0;                                    // starting state;
try
{  inp = System.in.read();       // character input,
   // convert to int.
   while (inp>=0 && inp<INPUTS)
   {  state = fsm[state][inp];                    // next state
      inp = System.in.read() ;                    // get next input
   }
} catch (IOException ioe)
  {  System.out.println ("IO error " + ioe); }

if (accept[state])
   System.out.println ("Accepted"); System.out.println ("Rejected");
```

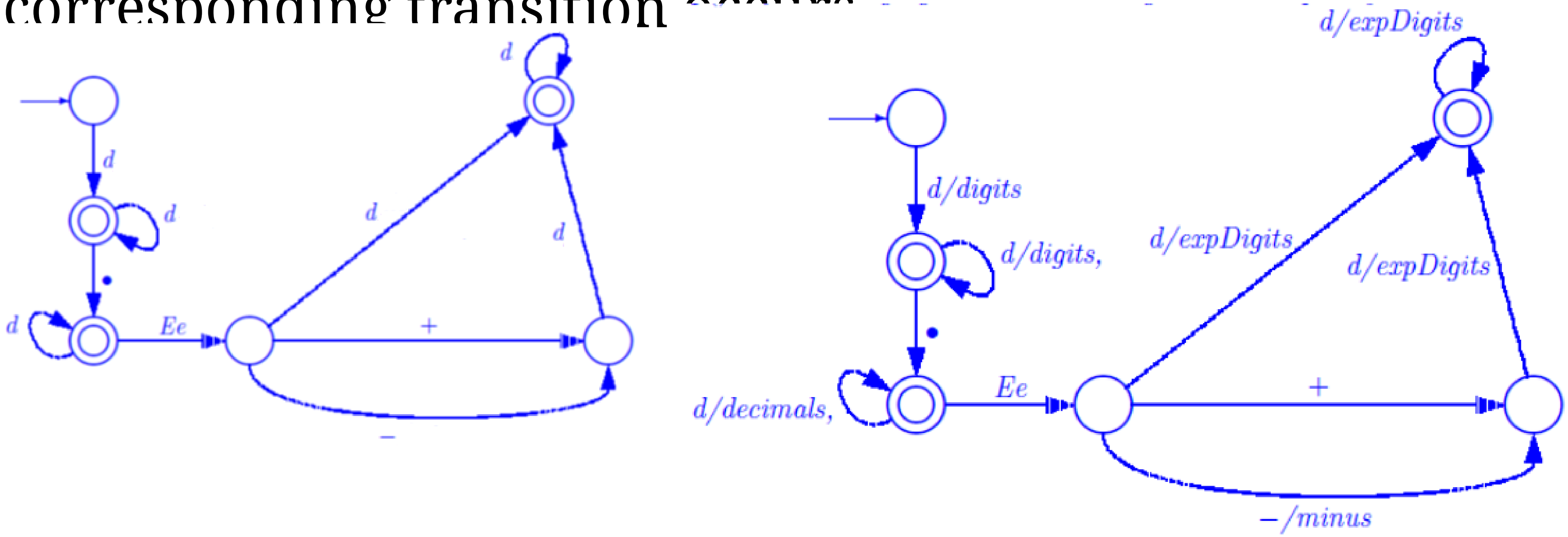
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D

Actions for Finite State Machines

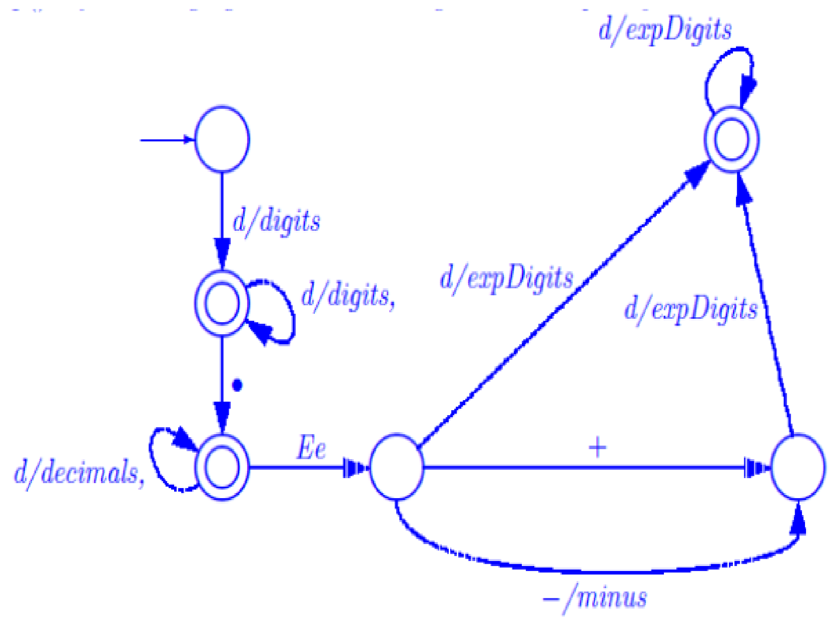
- At this point, we have seen how finite state machines are capable of specifying a language and how they can be used in lexical analysis.
- But lexical analysis involves **more than simply recognizing words**.
- It may involve **building a symbol table**, **converting numeric constants to the appropriate data type**, and **putting out tokens**.
- For this reason, we wish to **associate an action**, or function to be invoked, **with each state transition** in the finite state machine.

Example of FSM with Action

- Design a finite state machine, with actions, to **read numeric strings** and convert them to an **appropriate internal numeric format**, such as floating point.
- In the state diagram shown below we need to include method calls designated **digits(), decimals()**, **minus()**, and **expDigits()** which are to be invoked as the corresponding transition occurs



Example of FSM with Action




```
// instance variables
int d; // A single digit, 0..9
int places=0; // places after the decimal point
int NUM=0; // all the digits in the number
int exp=0; // exponent value
int signExp=+1; // sign of the exponent
```

```
// process digits before the decimal point
void digits()
{ NUM= NUM* 10 + d; }
// process digits after the decimal point
void decimals()
{ digits();
places++; // count places after the decimal point
}
```

```
// Change the sign of the exponent
void minus()
{ signExp = -1; }
// process digits after the E
void expDigits()
{ exp = exp*10 + d; }
```

How to implement Lexical Tables?

- One of the most important functions of the lexical analysis phase is the creation of tables which are used later in the compiler.
- Such tables could include a symbol table for identifiers, a table of numeric constants, string constants, and statement labels.
- The implementation techniques (**Sequential Search, Binary Search Tree, Hash Table**) could apply to any of these tables.

Sequential Search

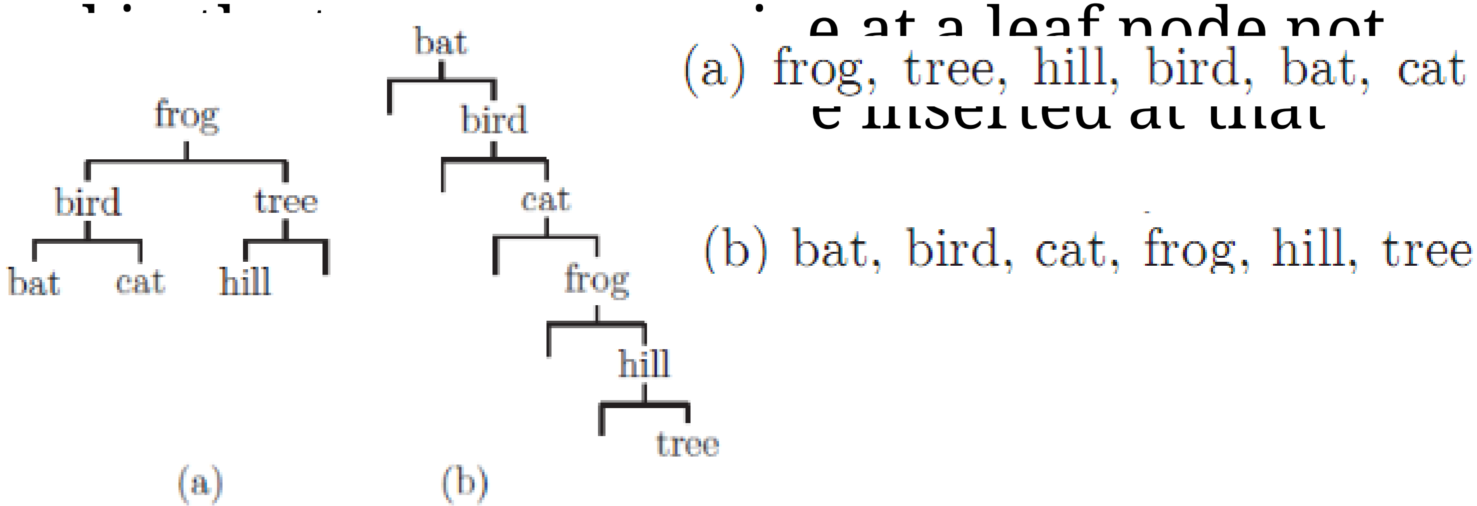
- The table could be organized as an array or linked list. Each time a word is encountered, the list is scanned and if the word is not already in the list, it is added at the end.

- The time required to build a table of **n** words is **$O(n^2)$.**

- This sequential search technique is easy to implement but not very efficient, particularly as the number of words becomes large.

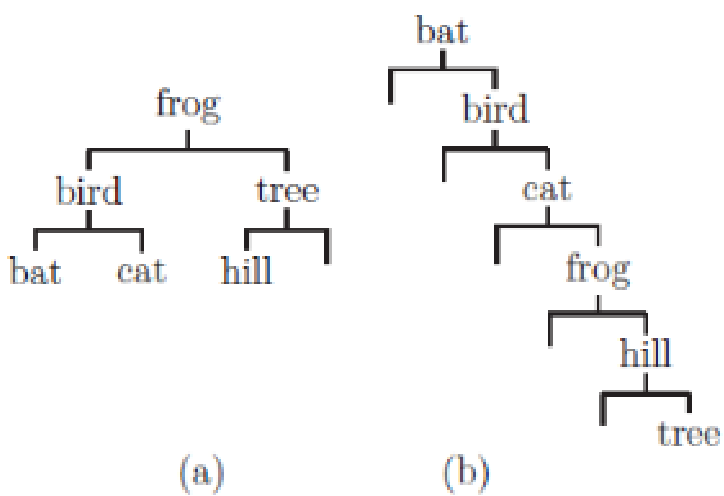
Binary Search Tree

- The table could be organized as a binary tree. Since the tree is initially empty, the first word encountered is placed at the root. Each time a word, w, is encountered the search begins at the root; w is compared with the word at the root.
- If w is smaller, it must be in the left subtree; if it is greater, it must be in the right subtree; and if it is equal, it is already in the tree. This is repeated until w has been fo … e at a leaf node not equal tc … e inserted at that point.



Binary Search Tree

- The time required to build such a table of n words is **O(n log n)** in the best case (**the tree is balanced**), but could be **O(n^2)** in the worst case (**the tree is not balanced**).

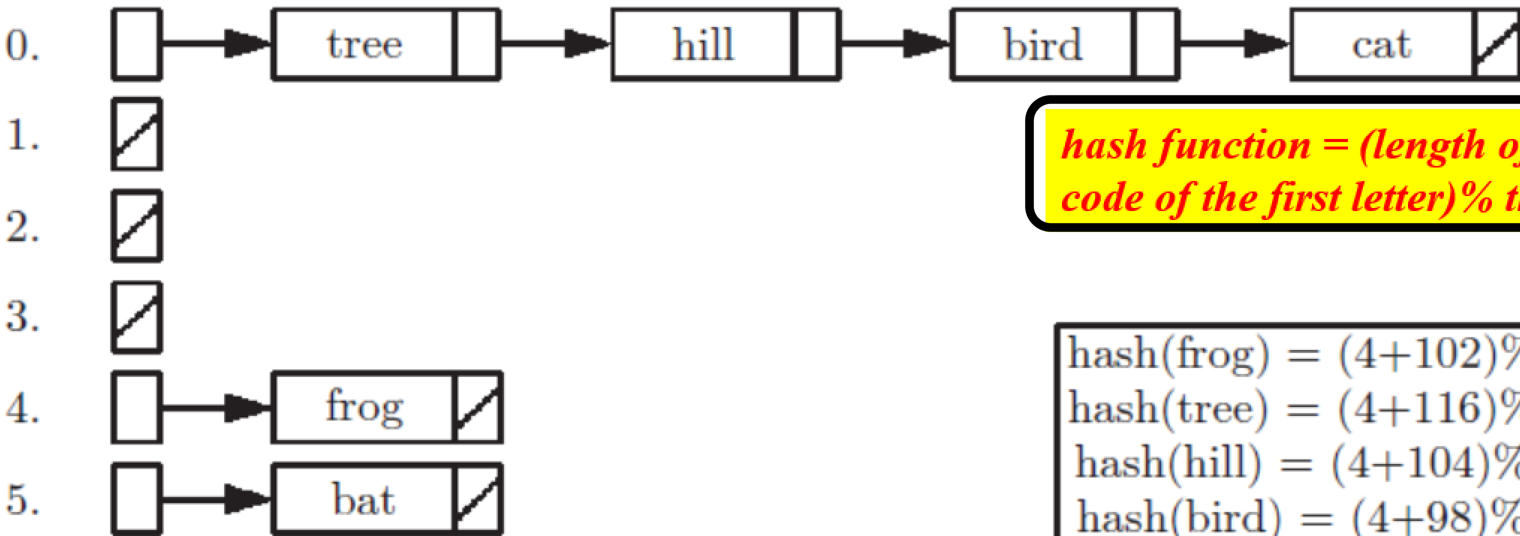


(a) frog, tree, hill, bird, bat, cat

(b) bat, bird, cat, frog, hill, tree.

Hash Table

- It can be organized as an array, or as an array of linked lists.
- A ***hash function*** is used to determine which list the word is to be stored in.



hash function = (length of word + the ascii code of the first letter)% the array size.

hash(frog) = (4+102)%6 = 4
hash(tree) = (4+116)%6 = 0
hash(hill) = (4+104)%6 = 0
hash(bird) = (4+98)%6 = 0
hash(bat) = (3+98)%6 = 5
hash(cat) = (3+99)%6 = 0

The selection of a good hash function is critical to the efficiency of this method.

THANKS

for your attention