Compiler Design

CS VI SEM

Prepared by
Arun Kumar Dewangan

Unit - 1

Introduction

Finite Automata and Lexical Analysis

Introduction

What is Compilers & Translators

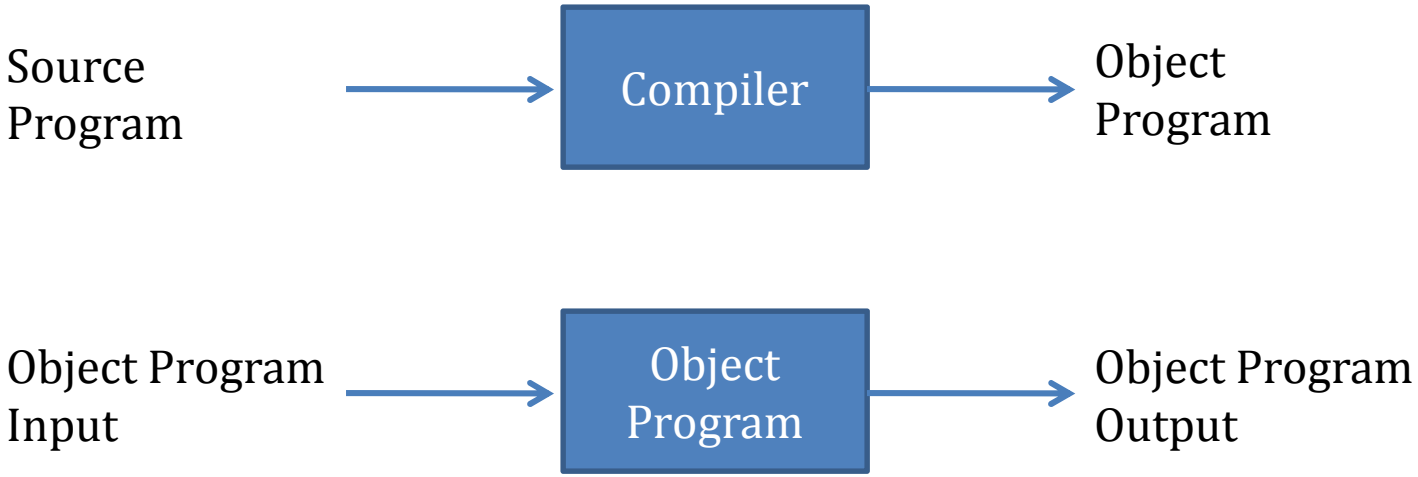
A ***translator*** is a program that takes as input a program written in one programming language (called source language) and produces as output a program in another language (called object or target language).

if the source language is high level language (e.g. FORTRAN, COBOL) and the object language is low level language (e.g. assembly language, machine language), then such a translator is called a ***compiler***.

Compilation and Execution

Executing a program written in a high level programming language is basically a two step process.

1. The source program must first be compiled, i.e. translated into object program.
2. Then the resulting object program is loaded into memory and executed.



Other Translators

Interpreter:

There are other translators too, which transform a programming language into a simplified language, called ***intermediate code***, which can be directly executed using a program called an ***interpreter***.

Interpreters are often smaller than compilers.

Main disadvantage of interpreters is the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

Assembler:

If the source language is assembly language and the target language is machine language, then the translator is called an ***assembler***.

Preprocessor:

Translators that take programs in one high level language and translate into equivalent programs in another high level language are called ***preprocessor***.

Compiler Vs Interpreter

- Both are written in some high level programming language and they are translated into machine code.
- Interpreter source program is machine independent as it does not generate machine code.
- Interpreter is slower than compiler because it processes and interprets each statement in program as many time get executed.

Need of Translators

- With machine language we have to communicate directly in terms of bits, registers and very primitive machine operations.
- Machine language is nothing more than a sequence of 0's and 1's.
- Programming an algorithm in such a language is tedious task and there may be opportunities of mistakes.
- Another disadvantage of machine language coding is that all operations and operands must be specified in a numeric code.

Symbolic Assembly Language

Because of difficulties with machine language programming, higher level language have been invented to enable the programmer to code in a way that looks like his own thought processes.

The most immediate step away from machine language is symbolic assembly language.

In this language programmer uses mnemonic names for both operation codes and data addresses.

To add values of X and Y, in assembly language programmer could write

ADD X, Y

that may be

0110 001110 010101 in machine language

where 0110 is machine operation code for "add" and 001110 and 010101 are the address of X and Y resp.

Computer cannot execute a program written in assembly language. The program has to be first translated to machine language, which computer can understand. This task is performed by ***assembler***.

Macros

Many programming languages provide a ***macro*** facility where a macro statement will translate into a sequence of assembly language statements and other statements before being translated into machine code.

Two aspects of macro : definition and use

Consider a situation if machine does not have ADD X, Y as a single statement (add the contents of one memory address to another)

Suppose machine has an instruction

- LOAD (moves datum from memory to a register)
- ADD (adds the contents of memory address to that of register)
- STORE (moves data from a register to memory)

Using these instructions, we can create a macro as

```
MACRO
        ADD2    X, Y
        LOAD    Y
        ADD     X
        STORE   Y
ENDMACRO
```

First statement gives name ADD2 to the macro with X and Y as dummy arguments (formal parameters)
If ADD2 A, B encounters somewhere after the definition of ADD2, then we have a macro use, with A and B as actual parameters.
The macro processor substitute ADD2 A,B the three statements with actual parameters A and B for X and Y respectively.
Hence Add2 A, B is translated to

```
LOAD   B
ADD    A
STORE  B
```

Drawback

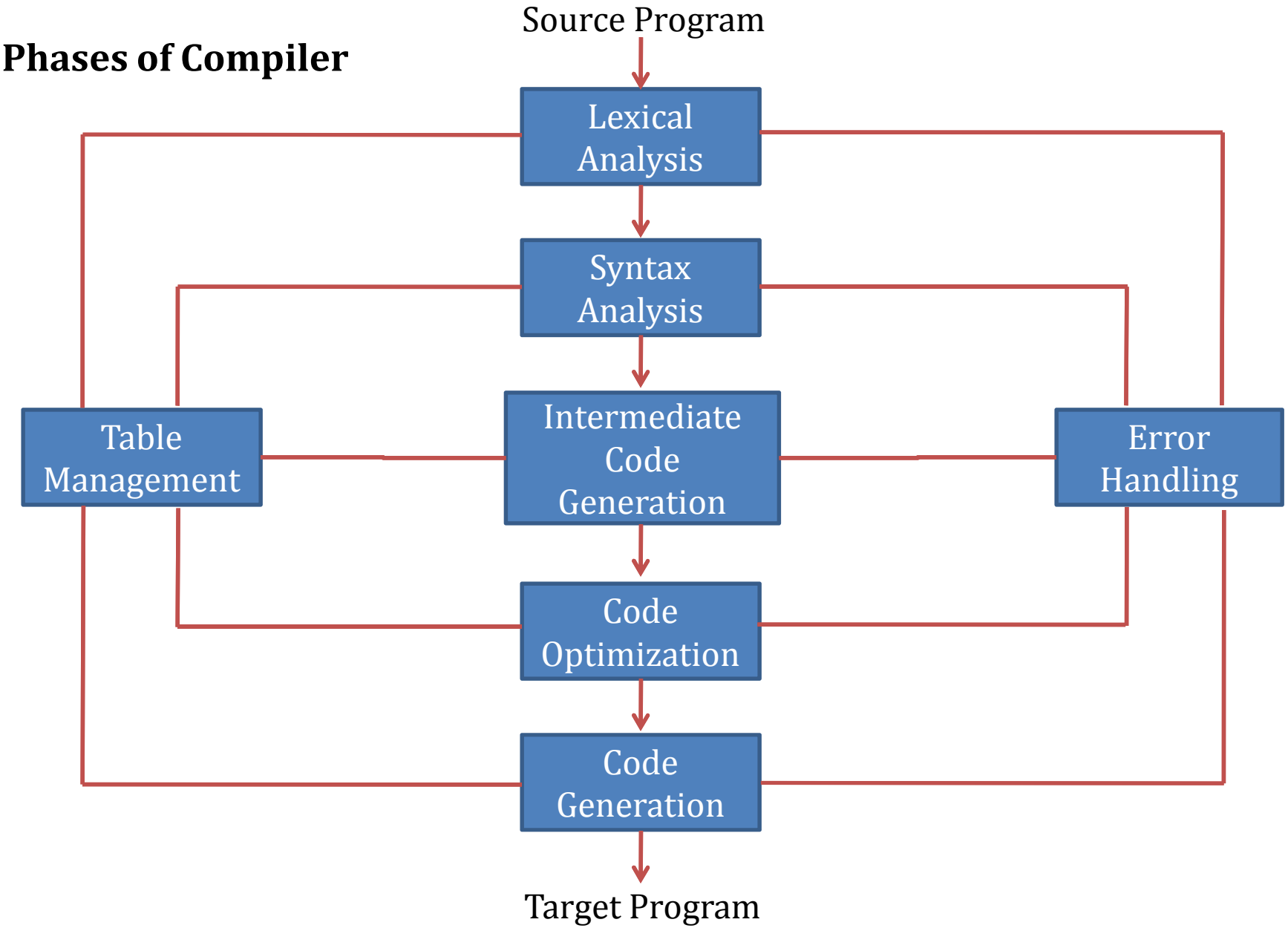
- Programmer must know the details of how a specific computer operates.
- Complex task to sequence low level operation which includes only primitive data types that machine language provides .
- Difficult to concern with how and where data is represented within the machine.
- Detailed knowledge is required for efficiency (leads to wasting of time)

High Level Languages

- To avoid problems with assembly language, high level programming languages are developed.
- Allows programmers to write expression in natural way like A + B instead of ADD A, B.
- We need a program to translate high level language into language the machine can understand.
- Some compilers makes use of assembler as an appendage.
- Compiler producing assembly code is assembled and loaded before being executed in machine language form.

COBOL, FORTRAN, PASCAL, LISP and C are some high level languages

Phases of Compiler



Phases of Compiler

A ***phase*** is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation. A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions.

1. Lexical Analyzer

- This is first phase of compiler.
- Also called ***scanner***.
- Separates characters of source language into groups that logically belong together.
- Groups are called tokens (DO or IF, identifiers, operator symbols like <= or +, punctuation symbols like parentheses or commas).
- Output of lexical analyzer is stream of tokens.
- These tokens are passed to the next phase.
- The tokens are represented by codes (e.g. DO might by 1, + by 2, identifier by 3 etc.).

2. Syntax Analyzer

- This is second phase of compiler.
- Also called ***parser***.
- Groups tokens together into syntactic structure called ***expression***.
- Expressions might be combined to form statements.
- Syntactic structure can be regarded as a tree whose leaves are tokens.
- The interior nodes of the tree represent strings of tokens that logically belong together.

3. Intermediate Code Generator

- This is third phase of compiler.
- Uses the structure produced by syntax analyzer to create stream of simple instruction.
- There may be many styles of intermediate code.
- Most common style is instruction with one operator and small no. of operands.
- Instructions are like macros.
- Intermediate code need not specify the registers to be used for each operation.

4. Code Optimization

- This is fourth and optional phase of compiler.
- Designed to improve the intermediate code.
- Ultimate object program runs faster, takes less space.
- Its output is another intermediate code program doing same job as original.
- Saves time or space.

5. Code Generation

- This is last phase of compiler.
- Produces object code by deciding
  - ✓ Memory locations for data.
  - ✓ Selecting code to access each datum.
  - ✓ Selecting the registers in which each computation is to be done.
- One of the difficult part of compiler.

Apart from these phases

Routines that interact with all phases of compiler are

Table Management

- Also called ***book keeping***.
- Compiler keeps track of the names used by the program.
- Records essential information about each (such as integer, real, etc.).
- The data structure used to record these information is called ***symbol table***.

Error Handler

- Invoked when a flaw in source program is detected.
- Must warn the programmer by issuing diagnostic information.
- Compilation be completed on flawed programs, at least through the syntax analysis phase, so that as many errors can be detected in one compilation.

Passes

In a compiler, portions of one or more phases are combined into a module called a ***pass***.

- A pass reads source program or output of previous pass.
- Makes the transformations specified by its phases.
- Writes output into an intermediate file, which may be read by a subsequent pass.
- A multi-pass compiler is slower than a single pass compiler because each pass reads and writes an intermediate file.
- Compiler running on small memory computer would use several passes.
- Computer with a large RAM, a fewer passes would be possible.

Backpatching

If output of a phase cannot be determined without looking at the remainder of the phase's input, the phase can generate output with slots which can be filled in later, after more of the input is read.

Consider an assembler might have statement **GOTO L**
which precedes a statement with label L.

- Two pass assembler uses its first pass to enter into its symbol table a list of all identifiers together with machine address.
- Then second pass replaces mnemonic operation codes, such as GOTO, by their machine language equivalent and replaces uses of identifiers by their machine addresses.
- Append the machine address for this instruction to a list of instructions to be backpatched once the machine address for L is determined.

  L: ADD X
- It scans list of statements referring to L and places the machine address for statement *L: ADD X* in the address field of each such instruction.
- The distance over which backpatching occurs must remain accessible until backpatching is complete.

Lexical Analysis

- It is the interface between source program and compiler.
- It reads source program one character at a time.
- Divides source program into sequence of atomic units called ***tokens***.
- Token represents sequence of characters treated as a single logical entity.
- Identifiers, keywords, constants, operators and punctuation symbols (comma and parenthesis) are typical tokens.

Example (Fortran statement)

IF (5 .EQ. MAX) GOTO 100

Tokens are : IF, (, 5, .EQ., MAX,), GOTO, 100

Types of Tokens:

- Specific Strings: IF or semicolon
- Classes of Strings: Identifiers, constants, or labels

- Token consists of two parts *token type* and *token value*.
- Specific strings such as semicolons are treated as having a type but no value.
- Token such as identifier MAX has a type "identifier" and a value consisting of string MAX.
- The lexical analyzer and syntax analyzer are grouped together into same pass.
- Lexical analyzer operates either under control of parser or as a co-routine with parser.
- Parser asks the lexical analyzer for next token whenever parser needs one.
- Lexical analyzer returns a code, for the token that it found, to the parser.
- If token is identifier or another token with a value, the value is also passed to parser.
- Method of providing this information by lexical analyzer is called *bookkeeping routine*.
- This routine puts the actual value in symbol table if it is not already present.
- Lexical analyzer passes two components of token:
  - Code for the token type (identifier)
  - A pointer to the place in the symbol table

Finding Tokens

❖The lexical analyzer examines successive characters in the source program, starting from the first character not yet grouped into a token.

❖Lexical analyzer may require to search many characters beyond the next token in order to determine what the next token actually is.

Example (Fortran statement)

IF(5.EQ.MAX)GOTO100

Blanks have been removed

❖Consider the situation that strings to the left of bracket are already broken up into tokens.

❖When the parser asks for next token, lexical analyzer reads all the characters between 5 and Q including . (dot) to determine that next token is constant 5

❖Reason to read up to Q is, until it sees Q, it is not sure that it has seen complete constant. It could be working on floating point constant such as ***5.E-10***.

- After determining that next token is constant 5, lexical analyzer repositions its input pointer at first . (dot).
- The lexical analyzer return token type "constant" to the parser and value associated with this constant could be numerical value 5 or a pointer to the string 5.
- After processing the statement, the token stream look like

if ([**const**, 341] **eq** [**id**, 729]) **goto** [**label**, 554]

- The relevant entries of symbol tables are

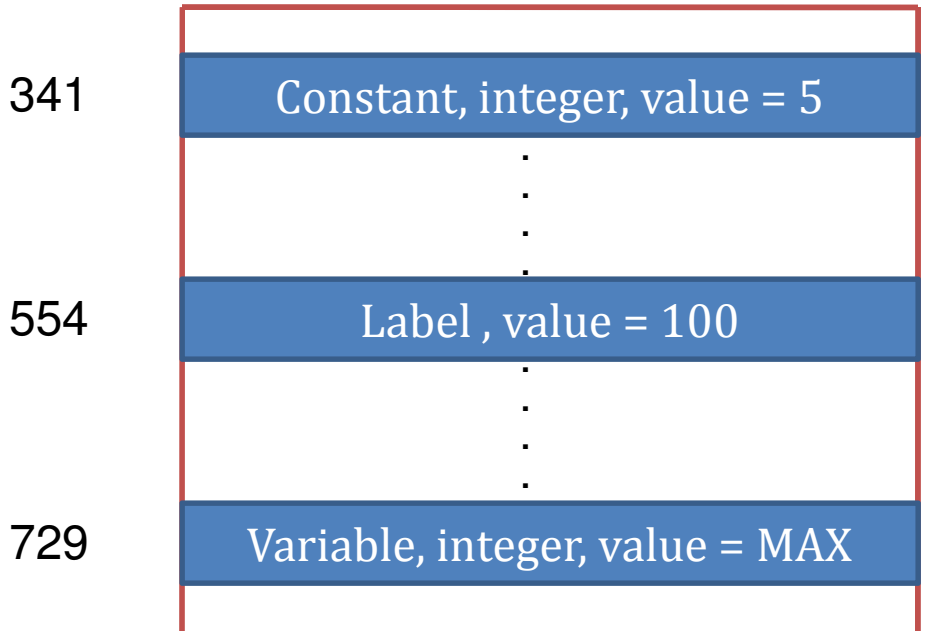


Fig: Symbol Table

Syntax Analysis

It has two functions

- It checks that tokens appearing in its input are in proper pattern as per specification of source language.
- Also imposes on tokens a tree like structure used by subsequent phases of compiler

Consider expression

A + / B

After lexical analysis, this expression appears for syntax analyzer as

id + / **id**

On seeing the /, parser should detect an error since presence of these two adjacent binary operators violates the formation rules

Parser makes hierarchical structure of incoming token stream by identifying which parts of token stream should be grouped together.

A / B * C

has two possible interpretation

- Divide A by B then multiply by C
- Multiply B by C then use result to divide A

- The interpretation can be represented in terms of ***parse tree***.
- Parse tree exhibits the syntactic structure of the expression.
- The language specification must tell us which interpretation is to be used.
- Context free grammars are helpful in specifying syntactic structure of a language.

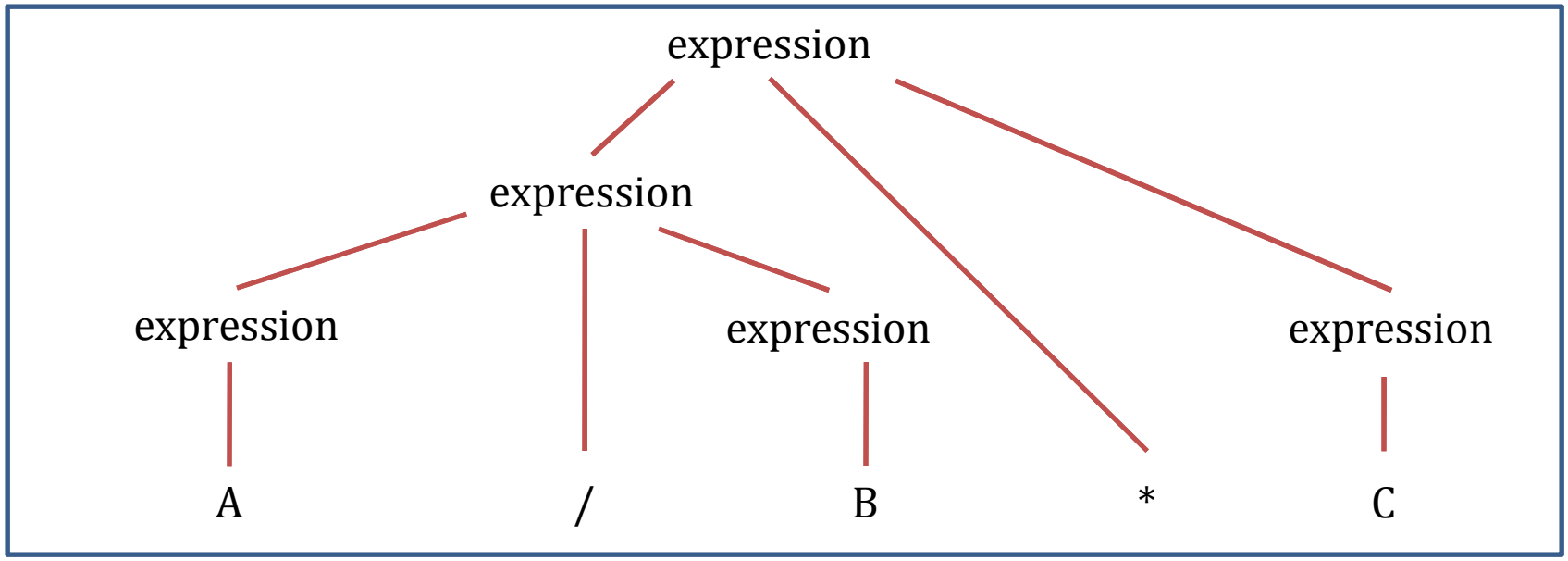
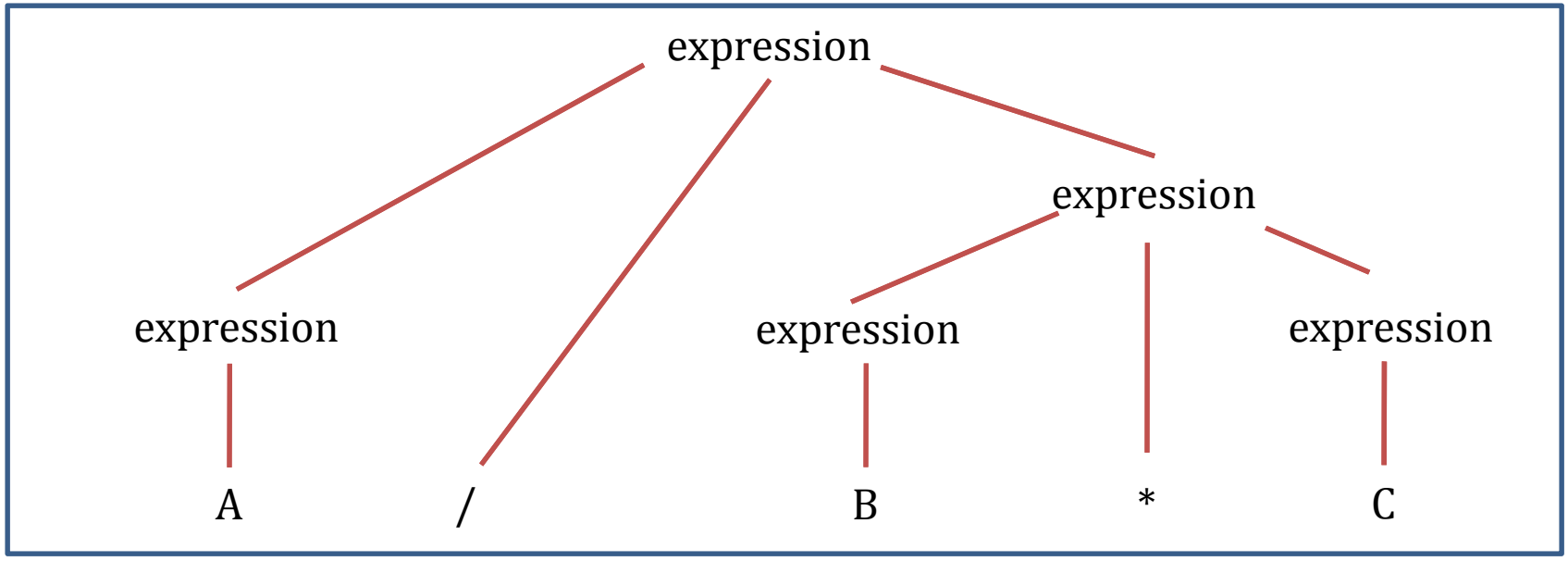


Fig: Possible parse tree for A / B * C



Semantic Analyzer

- It gathers type information and checks tree produced by syntax analyzer for semantic errors.
- Term semantic analysis is applied to determination of the type of intermediate result (check that arguments are of types that are legal).
- It is required to collect information concerning ***type and scope*** of variables.
- Semantic analysis can be done during syntax analysis, intermediate code generation or final code generation phase.

Intermediate Code Generation

❖Transforms parse tree into an intermediate language representation of the source program.

❖Popular type of intermediate language is ***three address code***.

Example A:= B **op** C

where A,B,C are operands and **op** is a binary operator.

The parse tree for **A / B * C** might be converted into three address code sequence

```
T1 := A / B
T2 := T1 * C
      or
T1 := B * C
T2 := A / T1
```

where T1 and T2 are names of temporary variables created by intermediate code generator.

Optimization

- Object programs that are frequently executed should be fast and small.
- Compilers may have phase in itself to transform output of intermediate code generator to another version of intermediate language which is faster and smaller object language program.
- This phase is called ***optimization phase***.
- There is no particular algorithmic way to produce best target program.
- Optimizing compilers just attempt to produce a better target program compared to no optimization.

Local Optimization

- Local transformation: Consider a statement

```
        if  A  >  B  goto  L2
        goto  L3
L2:
```

This sequence can be replaced by single statement

```
        if  A  <=  B  goto  L3
```

- Elimination of common subexpressions: consider statements

```
        A := B + C + D
        E := B + C + F      might be evaluated as
        T1 := B + C
        A :=  T1 + D
        E :=  T1 + F
```

Loop Optimization

Consider a for loop

```
for ( int i = 0 ; i  <  2000 ;  i + + )
{
        x := y + 3 ;
        q := a *  i ;
}
```

- In this loop the statement *x := y + 3 ;* has no effect in any part of loop and executed unnecessarily 2000 times.
- object code can be generated such that machine code for this line is moved outside for's body.
- Hence instruction is executed only once, instead of 2000 times. This is called ***loop optimization***.

Code Generation

- It converts intermediate code into sequence of machine instructions.
- Example: consider statement A := B + C
- Machine code sequence

```
LOAD   B
ADD    C
STORE  A
```

- The machine code contains many redundant load and store and utilizes resources of target machine inefficiently.
- To avoid redundant load and store, code generator keep track of run time contents of registers and can generate load and store only when necessary.
- Good code generator utilize registers efficiently as many computers have only few high speed registers.

Bookkeeping

- A compiler needs to collect information about all data objects appear in source program.
- Example: compiler need to know
  - Datatype of a variable
  - Size of an array
  - Number of arguments in a function etc.
- The information about data objects is collected by early phase of compiler (lexical and syntactic analysis).
- Information about data objects are entered into symbol table.

Use of Bookkeeping

❖Example: consider an expression

A + B

where A is of type integer and B is of real.

❖If language permits an integer value to be added to a real value then computer code must be generated to convert A from integer type to real before addition.

❖If expression having such nature is forbidden by the language, then compiler must issue an error message.

Error Handling

- Main function is detection and reporting of errors in source program.
- Error message should allow programmer to determine exact location of error.
- Error can be encountered by all phases.
- Example:
  - Lexical analyzer is unable to process next token due to misspelled.
  - Syntax analyzer is unable to determine a structure for input due to missing parenthesis.
  - Intermediate code generator detect an operator having operands of incompatible type, etc.
- In case of error, compiler must report error to error handler for issuing appropriate diagnostic message.

Compiler Writing Tools

- Tools developed to help construct compilers.
- Tools range from scanner, parser generator to compiler compilers, compiler generators, translator writing system.
- These tools produce a compiler from some form of specification of source language and target machine.
- Input specification may contain:
  - ✓Description of lexical and syntactic structure of source language.
  - ✓Description of output to be generated for each source language construct.
  - ✓Description of target machine.
- Instead of writing a program to perform syntax analysis, user writes a context free grammar and compiler-compiler automatically converts that grammar into program for syntax analysis.

Existing Compiler-Compiler provides

- Scanner generator.
- Parser generator.
- Facilities for code generation.

Bootstrapping

- A compiler is characterized by three languages:
  - ✓Source language.
  - ✓Object language.
  - ✓Language in which compiler is written.
- These languages may be quite different.
- *Cross Compiler*: a compiler run on one machine and produce object code for another machine. Such compiler is called cross compiler.

A compiler may be written in its own language. HOW??

Suppose we have new language L, to which we want to run on machine A and B.

- For machine A, write small compiler C_A^{SA} that translate subset S of language L ($S \subseteq L$) into machine/ assembly code of A. This compiler first be written in a language already available on A.
- Write a compiler C_S^{LA} in language S, run through C_A^{SA}, becomes C_A^{LA}, compiler for complete language L, running on machine A, and producing object code for A.

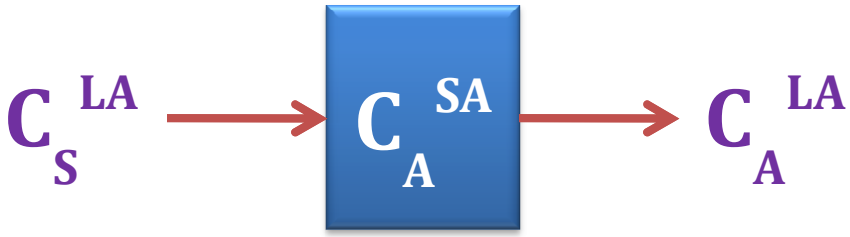


Fig: Bootstrapping a compiler

***Note**: C_Z^{XY} means - compiler for language X, written in language Z, producing object code in language Y.*

Finite Automata and Lexical Analysis

Role of Lexical Analyzer

- Works as an interface between source program and parser.
- It examines input text character by character and separates the source program into pieces called tokens.
- Acts as a subroutine which is called by the parser for new token.
- It returns to parser a representation for token it found.
- Representation is an ***integer code*** if token is simple construct (parenthesis, comma, colon, etc.).
- Representation is pair consisting of ***integer code and a pointer to a table*** if token is complex element (identifier, constant, etc.).
- Integer code gives token type, pointer points to the value of that token.

also

- Remove the content free characters (comment, white spaces, etc.).
- Insert line number during analysis.
- Evaluating constants.
- Detect lexical errors such as numeric literals are too long, identifiers are too long.
- Output of lexical analysis is input to syntax analysis.

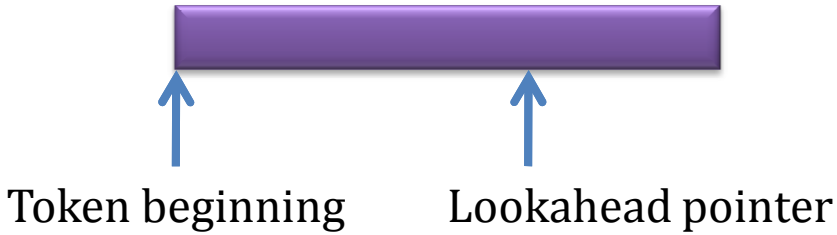
Input Buffering

- Lexical analyzer scans one character at a time of source program to discover tokens.
- Many characters beyond token may have to be examined to determine the token itself.
- So lexical analyzer reads its input from input buffer.

Example

Consider two pointer

- One pointer marks beginning of the token being discovered.
- A lookahead pointer scans ahead of the beginning point, until token is discovered.



Tokens

- A program can be partitioned at lowest level into a sequence of substrings called tokens.
- Each token is a sequence of characters whose significance is possessed collectively rather than individually.
- Example of tokens:
  - Constants (1, 2.3, 4.5E6)
  - Identifiers (A, H2035B, SPEED)
  - Operator symbols (+, -, **, :=, .EQ.)
  - Keywords (IF, GOTO, SUBROUTINE)
  - Punctuation symbols (parenthesis, brackets, comma and semicolon)

Design of Lexical Analyzer

- Behavior of any program can be represented by flowchart.
- We represent behavior of lexical analyzer by ***transition diagram***.
- In transition diagram, boxes of flowchart are drawn as circles called *states*.
- States are connected by arrows called *edges*.
- Labels on the edges leaving the state indicate the input characters that can appear after that state.

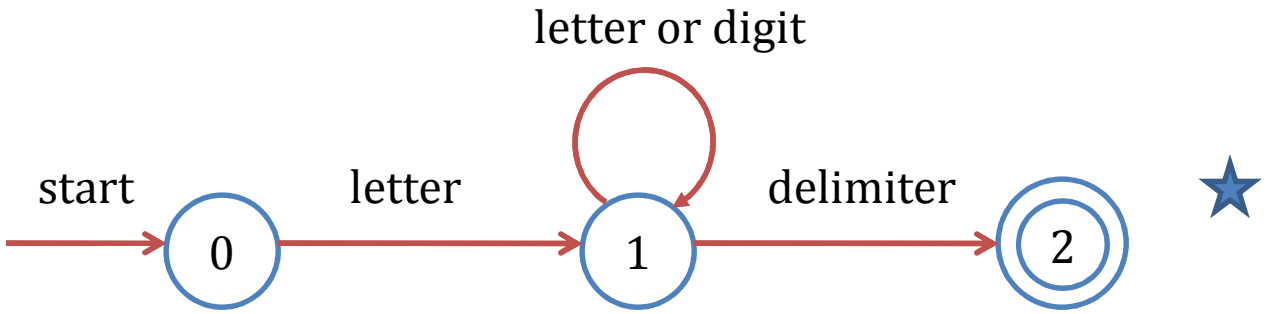


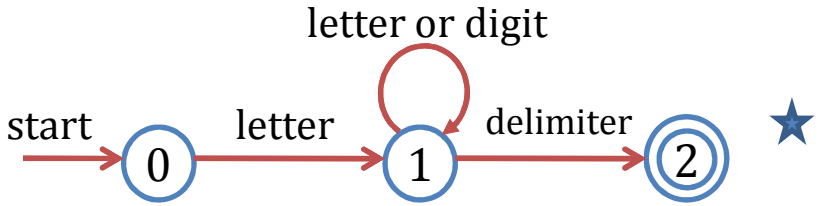
Fig: Transition diagram for identifier

Description of transition diagram

- Figure shows transition diagram for an identifier.
- Identifier is letter followed by any number of letters or digits.
- Starting state is 0.
- Edge from 0 indicates that first character must be a letter (enter state 1).
- Look for next input character, if letter or digit, reenter state 1.
- Continue reading letters or digits and making transition from state 1 to itself.
- If input character is a delimiter (not a letter or digit) for an identifier, enter state 2.

Construction of code for transition diagram

- Construct segment of code for each state.
- First step is: obtain next character from input buffer (using **GETCHAR** function).
- GETCHAR function returns the next character and advancing the lookahead pointer at each call.
- Determine edge, out of state is labeled by a character or class of character.
- If such edge is found, control is transferred to state pointed to by that edge.
- If no such edge is found, and state is not one which indicates that token has been found (final state), we have failed to find this token.
- The lookahead pointer must be **retracted** to where the beginning pointer is, and search for another token.



Code for state 0 might be:

```
state 0:  C := GETCHAR ();
          if LETTER (C) then goto state 1
          else FAIL ()
```

- Here LETTER is a procedure, returns true iff C is a letter.
- FAIL is routine which retracts lookahead pointer and starts up next transition diagram.

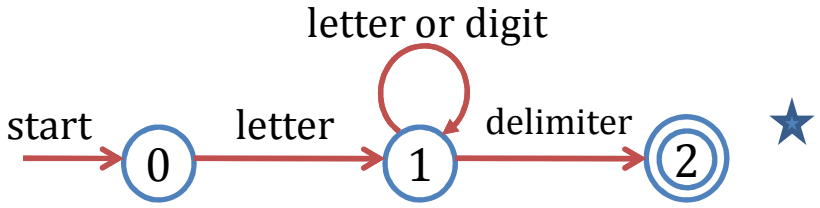
Code for state 1 might be:

```
state 1:        C := GETCHAR ();
          if LETTER (C) or DIGIT (C) then goto state 1
          else if DELIMITER (C) then goto state 2
          else FAIL ()
```

- Here DIGIT is a procedure, returns true iff C is one of the digits (0,1,…,9).
- DELIMITER is procedure which returns true whenever C is character that could follow an identifier.
- DELIMITER is defined to be any character that is not a letter or digit.

Code for state 2 might be:

```
state 2:  RETRACT ();
          return (id, INSTALL ())
```



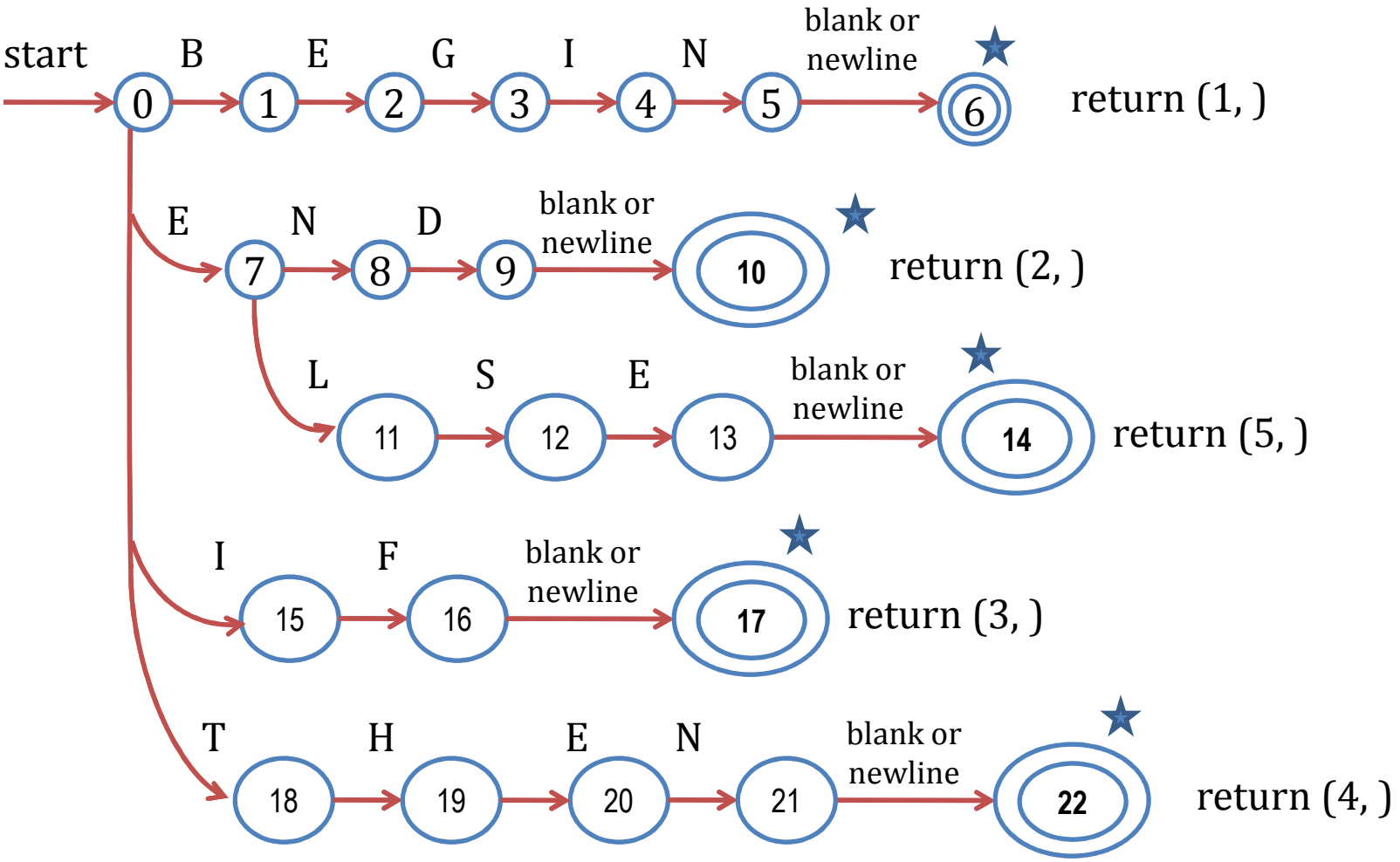
- State 2 indicates that an identifier is found.
- Delimiter is not part of identifier, retract lookahead pointer one character (using RETRACT procedure).
- Use * to indicate state on which input retraction take place.
- Install newly found identifier in symbol table if it is not already there (using procedure INSTALL).
- Return to the parser a pair consisting of integer code (id) for an identifier, and value that is a pointer to symbol table returned by INSTALL.

Token	Code	Value
begin	1	---
end	2	---
if	3	---
then	4	---
else	5	---
identifier	6	Pointer to symbol table
constant	7	Pointer to symbol table
<	8	1
< =	8	2
=	8	3
< >	8	4
>	8	5
> =	8	6

Fig: Token recognized

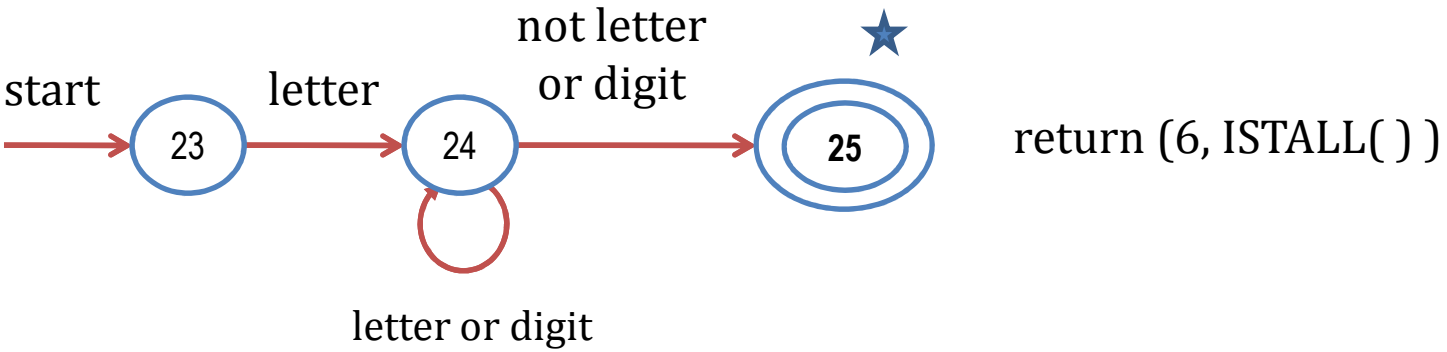
Implementation of Transition Diagram

- Keywords:



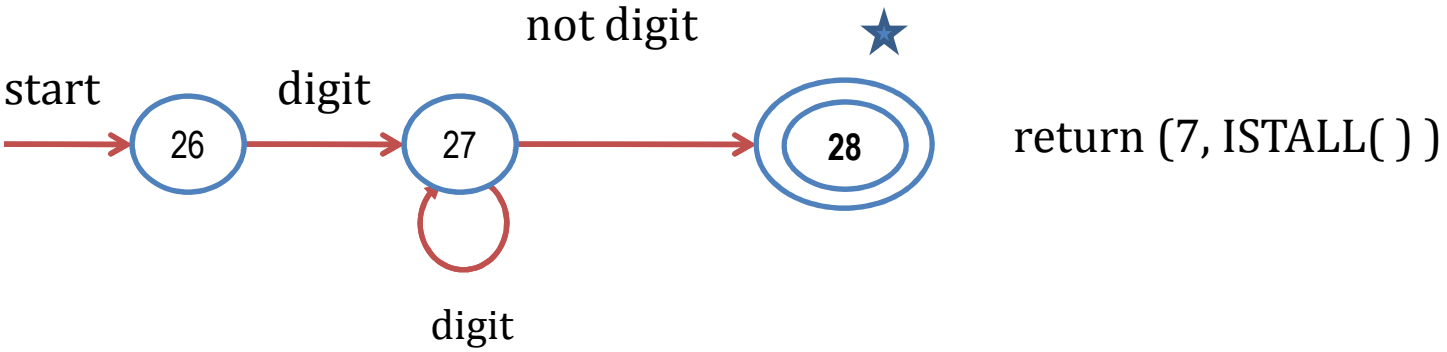
- ✓ To implement state, generate code that calls GETCHAR.
- ✓ Converts resulting character to an integer.
- ✓ Produce code that uses that integer to index into an array of labels.
- ✓ Each label marks the location of piece of program for next state.
- ✓ In last figure, label indexed by B would point to code for state 1.
- ✓ Label for E would point to program for state 7 and so on.
- ✓ Labels for characters that did not begin a keyword would point to code which implemented the transition diagram for identifier.

❖Identifier:



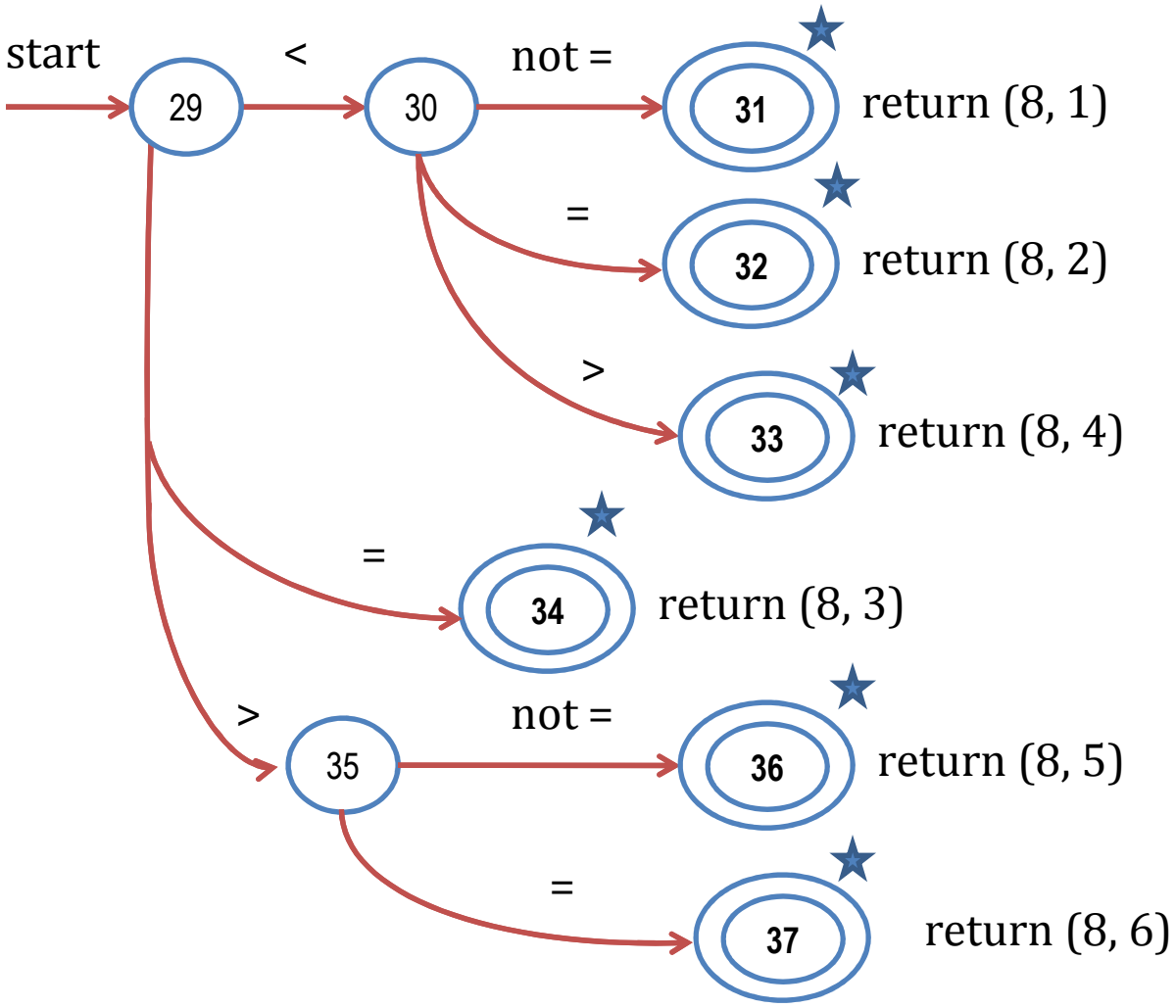
return (6, ISTALL( ))

❖Constant:



return (7, ISTALL( ))

- Relops:


start
29
<
30
not =
31
return (8, 1)
=
32
return (8, 2)
>
33
return (8, 4)
=
34
return (8, 3)
>
35
not =
36
return (8, 5)
=
37
return (8, 6)

String:

- ✓ String is a finite sequence of symbols such as 001.
- ✓ *Sentence* and *words* are synonyms for string.
- ✓ Length of string, |x|, is total number of symbols in x.
- ✓ 01101 is a string of length 5.
- ✓ Special string called *empty string*, ϵ, is of length zero. We take x^0 to be ϵ.
- ✓ If x and y are strings, then concatenation of x and y, x.y or xy is the string formed by placing symbols of y after symbols of x.
- ✓ ϵ x = x ϵ = x.
- ✓ x^i is the string x repeated i times.
- ✓ If x is some string, then string formed by discarding zero or more trailing symbols of x is called a *prefix* of x.
- ✓ A *suffix* of x is string formed by deleting zero or more of the leading symbols x.
- ✓ abc is prefix of abcde, and cde is suffix of abcde.
- ✓ A *substring* of x is any string obtained by deleting prefix and suffix from x, but a substring need not be a prefix or suffix.

Language:

✓Any set of strings formed from some specific alphabet.

✓Ø (empty set), having no member, or {ϵ}, the set containing only empty string, is also a language.

✓If L and M are languages then L.M or LM is the language consisting of all strings xy which can be formed by selecting string x from L and y from M.

LM = { xy | x is in L and y is in M }

LM is concatenation of L and M.

❖Example : Let L be { 0, 01, 110 } and M be { 10, 110 }. Then

LM = { 010, 0110, 0110, 01110, 11010, 110110 }

❖{ϵ} L = L {ϵ} = L

❖L ∪ M = { x | x is in L or x is in M }

❖Ø ∪ L = L ∪ Ø = L

❖Ø L = L Ø = L

❖$L^* = \bigcup_{i=0}^{\infty} L^i$

Regular Expressions:

❖Regular expression are useful for describing tokens.

Definition

Set of regular expression is defined by following rules:

1. null string ϵ is a regular expression denoting {ϵ}.
2. For each a in ∑ is a regular expression {a}.
3. If R and S are regular expressions denoting languages L_R and L_S resp. then
   i. (R) | (S) is a regular expression denoting $L_R \cup L_S$.
   ii. (R) . (S) is a regular expression denoting $L_R . L_S$.
   iii. (R)* is a regular expression denoting L_R*.

Example:

Regular expression for identifier can be given as

identifier = letter (letter | digit)*

- Identifier is defined to be a letter followed by zero or more letters or digits.
- | means “or” that is union.

Finite Automata:

❖These are machines that represents a computer by providing a medium, which executes a finite no. of instructions sequentially as per

- algorithm
- accepting valid input
- and producing an output if input is accepted

❖ These machines have

❖ input tape

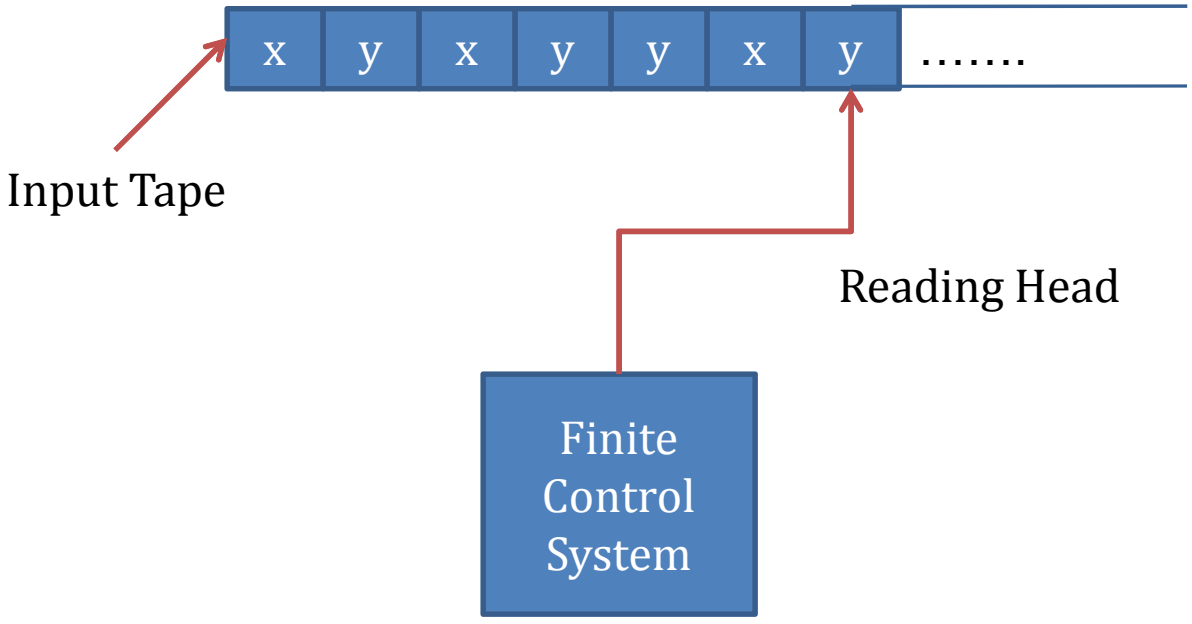
❖ reading head

❖ register recording the state for machine

❖ a set of instructions that govern the transition from one state to another

Finite Automata:

- x and y are input symbols.
- At regular time the automaton reads one symbol from input tape.
- Then enters in a new state that depends on the current state and input symbol just read.
- After reading symbol, reading head moves one square to the right.

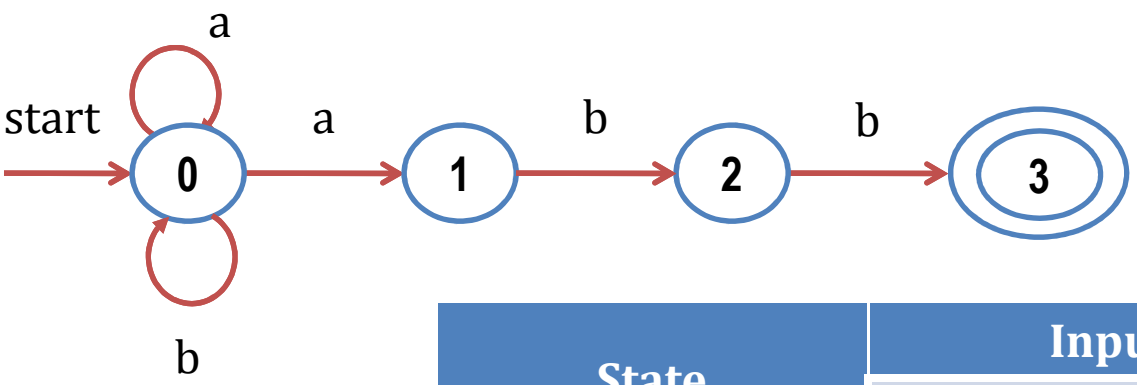


Recognizer:

- A recognizer for a language L is a program that takes a string x
- and answers "yes" if x is a sentence of L and "no" otherwise.
- Recognizer is the part of lexical analyzer that identifies the presence of a token on input.

Nondeterministic Finite Automata (NFA):

- It is a labeled directed graph.
- Nodes are called states.
- Labeled edges are called transitions.
- It looks like a transition diagram.
- Edges can be labeled by ϵ as well as characters.
- Same character can label two or more transitions out of one state.
- Accepting state is indicated by double circle.
- Transitions are represented in tabular form by means of transition table.



State	Input Symbol	
	a	b
0	{0,1}	{0}
1	---	{2}
2	---	{3}
3	---	---

Fig: Transition Table

❖The NFA accepts input string x if and only if there is a path from start state to some accepting state.

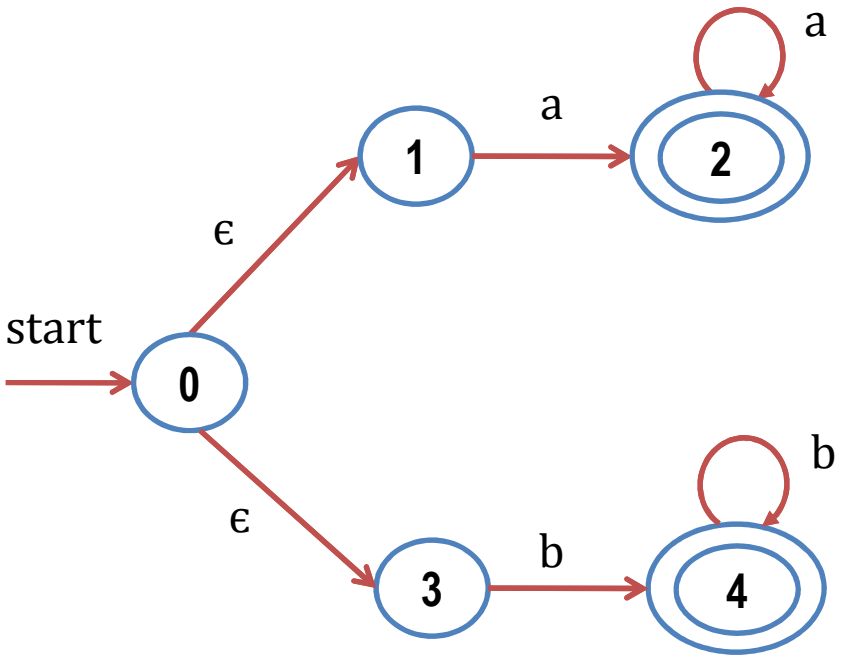


Fig: NFA accepting **aa* | bb***

Deterministic Finite Automata (DFA):

- It has no transitions on input ϵ.
- For each state s and input symbol a, there is at most one edge labeled a leaving s.

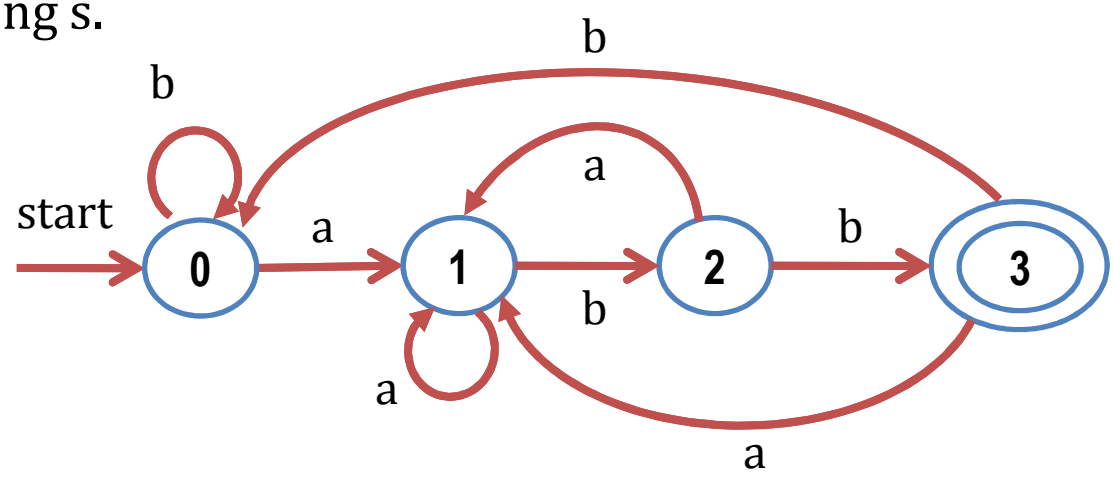


Fig: DFA accepting **(a | b)* abb**

Algorithm: Constructing a DFA from NFA

Do yourself

From Regular Expression to Finite Automata:

Algorithm: Constructing an NFA from a regular expression

Do yourself

Minimizing Number of States of a DFA:

Algorithm: Minimizing the number of states of a DFA

Do yourself

Language for Specifying Lexical Analyzers:

- LEX is a tool for constructing a lexical analysis program.
- LEX source program is a specification of lexical analyzer.
- It consists of set of regular expression.
- Code written in LEX source program is executed whenever a token specified by corresponding regular expression is recognized.
- Action will pass an indication of token found to the parser with making an entry in symbol table.

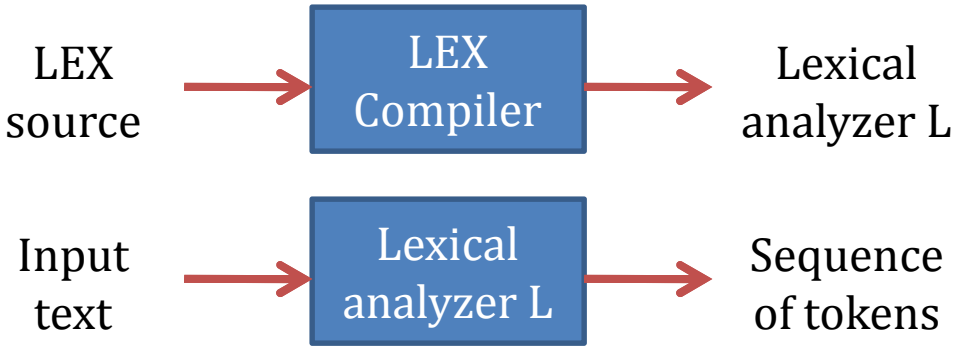


Fig: The role of LEX

Auxiliary Definition:

- LEX source program consists of two parts:
  - A sequence of auxiliary definitions followed by
  - A sequence of translation rules
- Auxiliary definitions are statements of the form

$$D_1 = R_1$$
$$D_2 = R_2$$
$$\vdots$$
$$D_n = R_n$$

- D_i is a distinct name
- R_i is a regular expression whose symbols are chosen from $\sum \cup \{ D_1, D_2, \ldots, D_{i-1}\}$
- D_i's are short names for regular expressions.
- $\sum$ is input symbol alphabet.

Translation Rules:

- These are statements of the form

$$P_1 \quad \{A_1\}$$
$$P_2 \quad \{A_2\}$$
$$\vdots$$
$$P_m \quad \{A_m\}$$

- P_i is a regular expression called pattern over alphabet consisting of $\sum$ and auxiliary definition names.
- Patterns describe the form of tokens.
- Each A_i is a program fragment describing what action lexical analyzer should take when token P_i found.

Working of Lexical Analyzer L:

- Let lexical analyzer L is generated by LEX.
- L reads its input one character at a time until it found longest prefix of input which matches one of regular expression P_i.
- Once L found that prefix, L removes it from its input and places it in a buffer called TOKEN.
  - TOKEN may be pair of pointers to the beginning and end of the matched string in input buffer itself.
- L then executes an action A_i.
- After completing A_i, L returns control to the parser.
- L repeats this series of actions on remaining input.
- If none of regular expressions denoting tokens matches any prefix of input, an error occurred and L transfers control to some error handling routine.

Implementation of Lexical Analyzer:

- Create NFA for each translation rule.
- Convert all NFA's into one NFA.
- Convert NFA into DFA.

Example

Auxiliary Definitions

(none)

Translation Rules

a

abb

$a^* b^+$

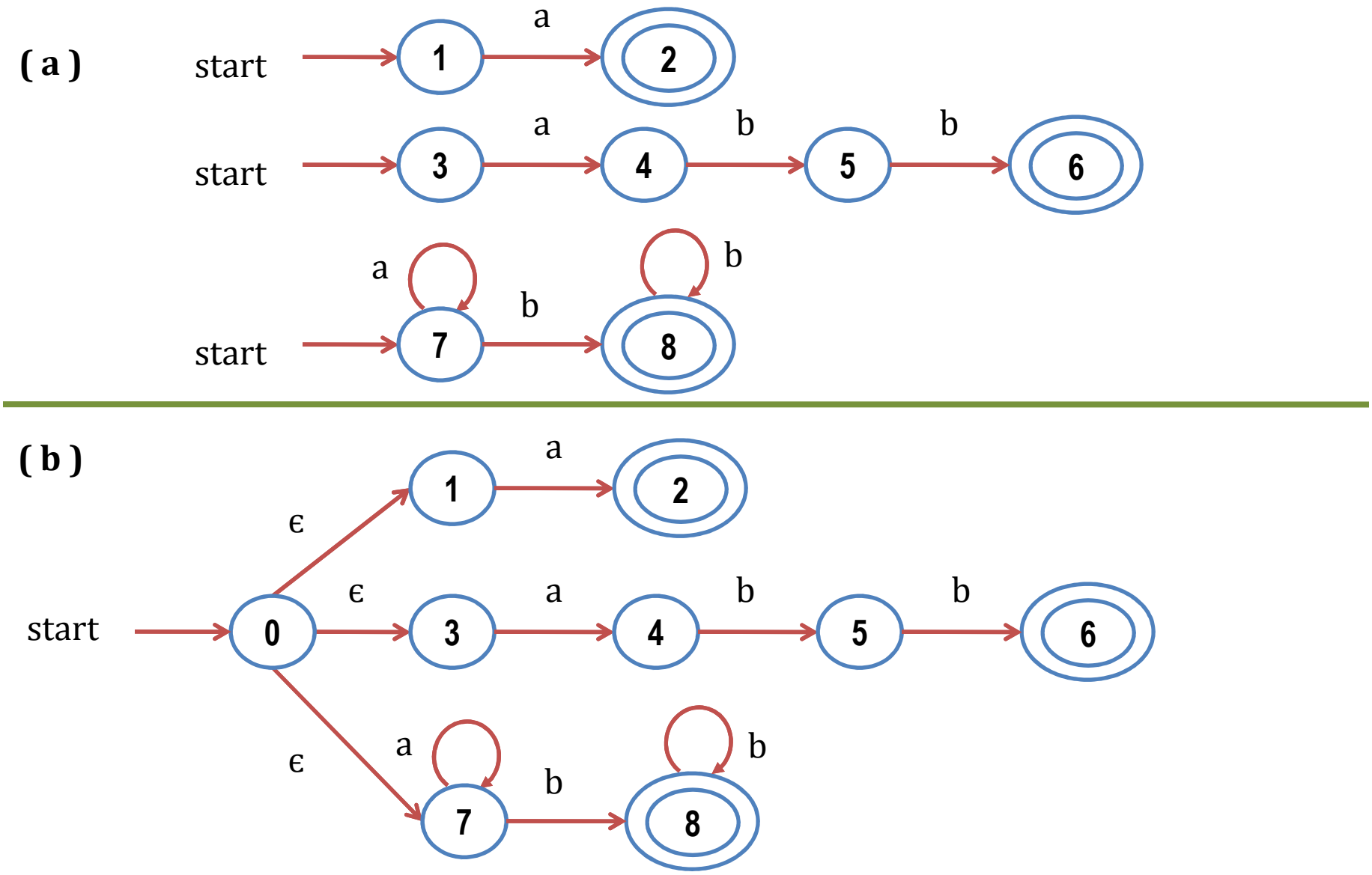


Fig: NFA recognizing three different tokens

State	a	b	Token Found
0137	247	8	none
247	7	58	a
8	---	8	a^*b^+
7	7	8	none
58	---	68	a^*b^+
68	---	8	abb

Fig: Transition table for DFA

End of Unit 1