SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Compiler Design

Preliminaries Required

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

- Basic knowledge of programming languages.
- Basic knowledge of FSA and CFG.
- Knowledge of a high programming language for the programming assignments.

Textbook:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman,

"*Compilers: Principles, Techniques, and Tools*"

Addison-Wesley, 1986.

Course Outline

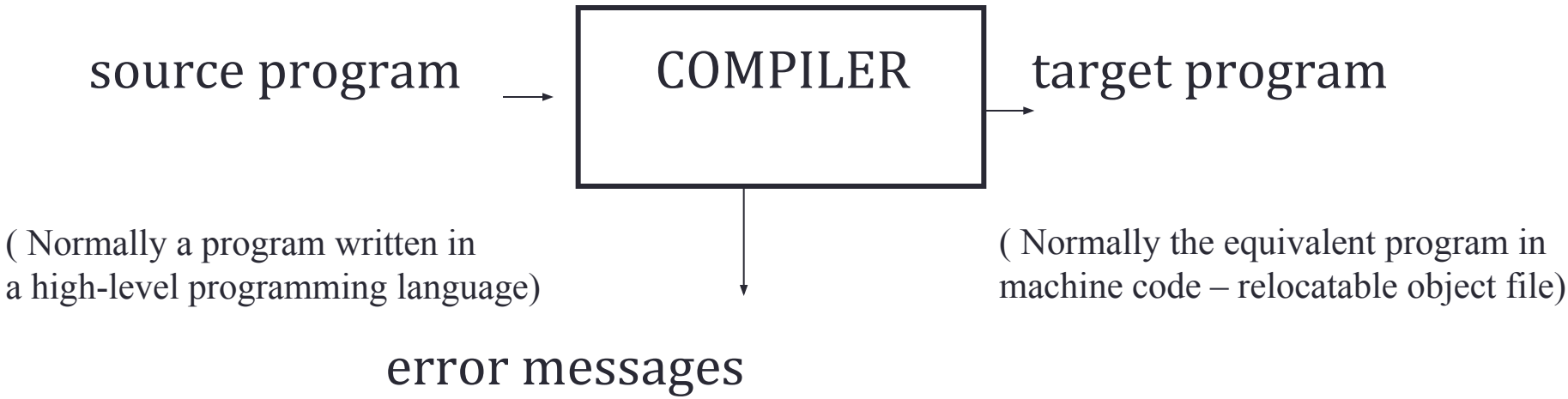
- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation
- Code Optimization
- Code Generation

Compiler - Introduction

- A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language.
- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- Ignore machine-dependent details for programmer

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Compiler vs Interpreter

- An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement

Compiler Applications

- Machine Code Generation

– Convert source language program to machine understandable one

– Takes care of semantics of varied constructs of source language

– Considers limitations and specific features of target machine

– Automata theory helps in syntactic checks

– valid and invalid programs

– Compilation also generate code for syntactically correct programs

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**
- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Structure of a Compiler

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

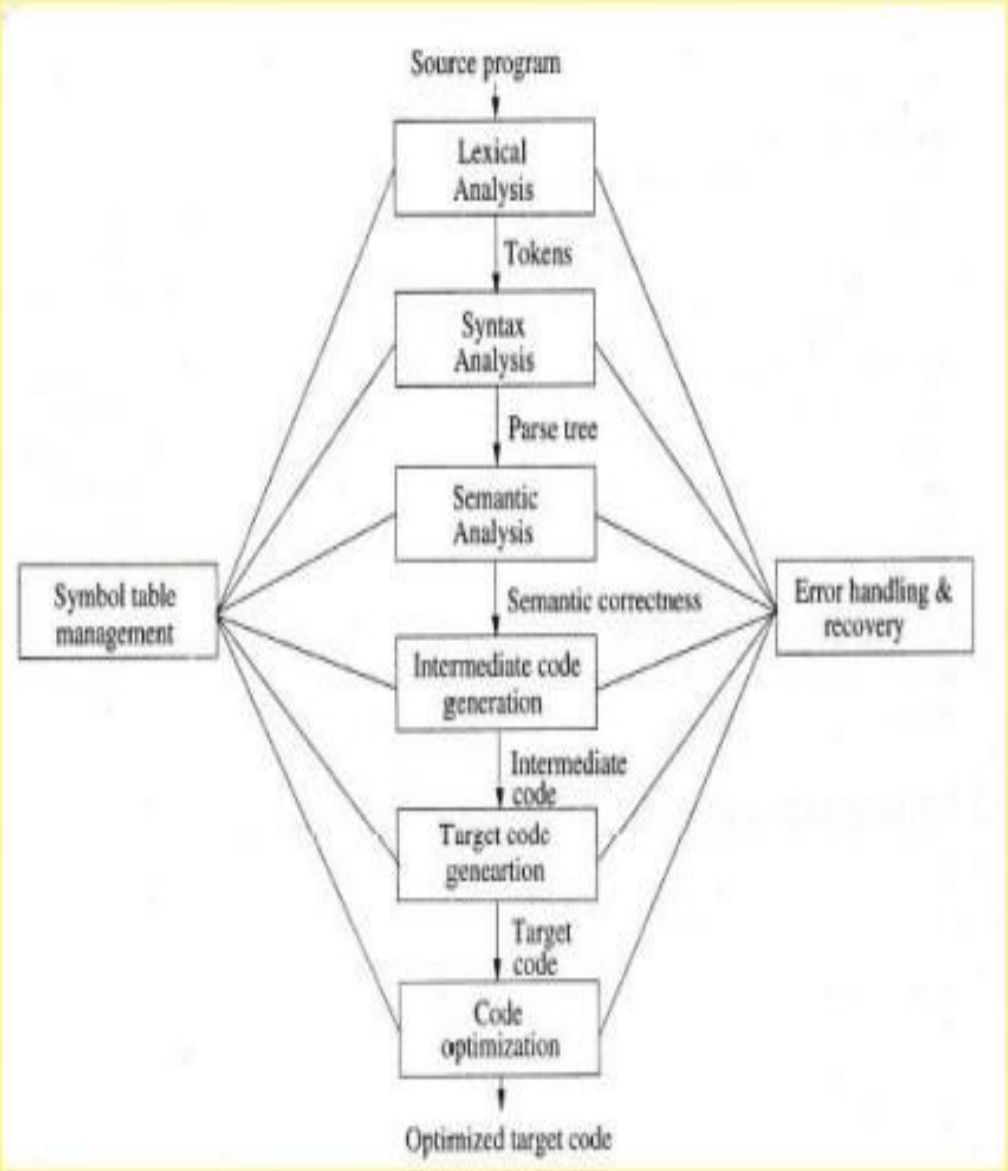
Analysis

- Breaks the source program into pieces and fit into a grammatical structure
- If this part detect any syntactically ill formed or semantically unsound error it is report to the user
- It collect the information about the source program and stored in a data structure – Symbol Table

Synthesis

- Construct the target program from the available symbol table and intermediate representation

Phases of a Compiler



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Phases of A Compiler

- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Lexical Analyzer

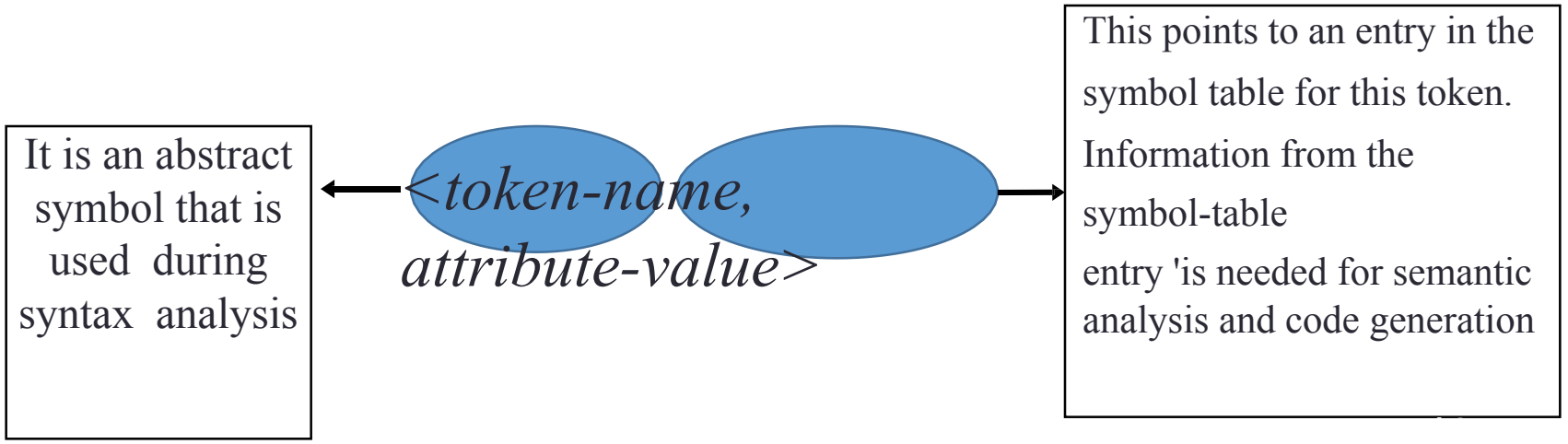
- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

  Ex: newval := oldval + 12 => tokens: newval identifier
  := assignment operator
  oldval identifier
  + add operator
  12 a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Phases of Compiler-Lexical Analysis

- It is also called as scanning
- This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the form
- It passes on to the subsequent phase, syntax analysis.



Lexical Analysis

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

For example consider `position := initial + rate * 60`

- *position* is lexeme mapped to token <id,1>
  - <*Id*> is an abstract symbol
  - <*1*> points to the symbol table entry for position
- The assignment symbol = is a lexeme that is mapped into the token
  <=>
  - We can omit second component as it has no attribute value
- Similarly *initial* is mapped to <id,2>,*rate* <id,3>
- The lexeme + is mapped to <+>, * is mapped to <*> and *60* is mapped to <60> $\langle \mathbf{id}, 1 \rangle \langle = \rangle \langle \mathbf{id}, 2 \rangle \langle + \rangle \langle \mathbf{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Lexical Analysis

- Lexical analysis breaks up a program into tokens
  - Grouping characters into non-separatable units (tokens)
  - Changing a stream to characters to a stream of tokens

```
program gcd (input, output);
var i, j : integer;
begin
  read (i, j);
  while i <> j do
    if i > j then i := i - j else j := j - i;
  writeln (i)
end.
```

program	gcd	(	input	,	output	)	;
var	i	,	j	:	integer	;	begin
read	(	i	,	j	)	;	while
i	<>	j	do	if	i	>	j
then	i	:=	i	-	j	else	j
:=	i	-	i	;	writeln	(	i
)	end	.					

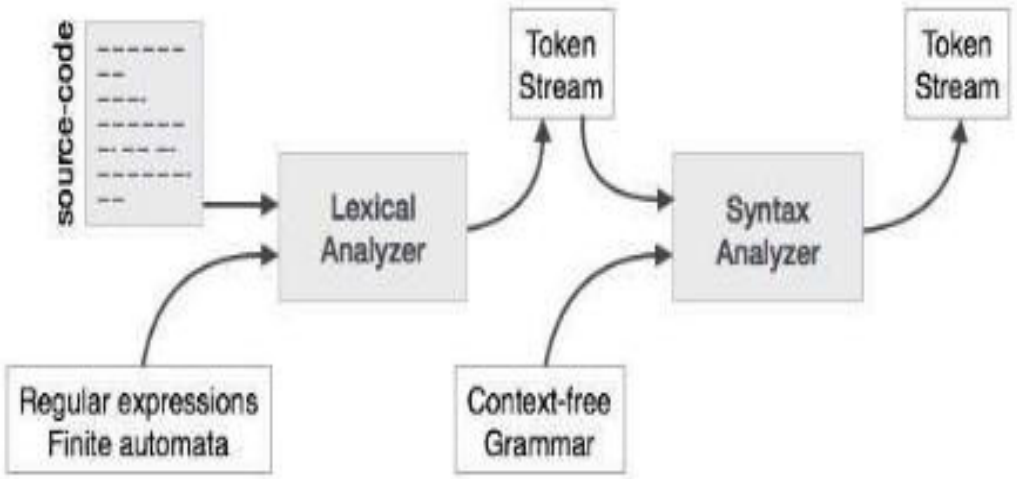
Token , Pattern and Lexeme

- **Token:** Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants
- **Pattern:** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
- **Lexeme:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Token	lexeme	pattern
const	const	const
if	if	if
relation	<,<=,=,<>,>=,>	< or <= or = or <> or >= or letter followed by letters & digit
i	pi	any numeric constant
num	3.14	any character b/w "and "except"
literal	"core"	pattern

Phases of Compiler-Symbol Table Management

- Symbol table is a data structure holding information about all symbols defined in the source program
- Not part of the final code, however used as reference by all phases of a compiler
- Typical information stored there include name, type, size, relative offset of variables
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

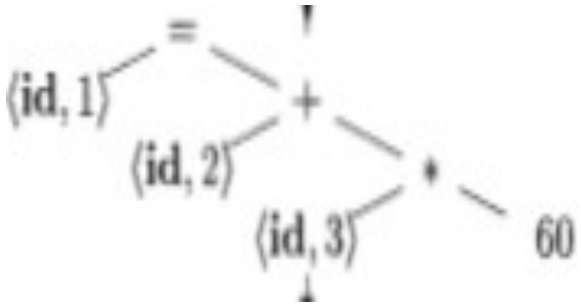
Syntax Analysis

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
A syntax analyzer is also called as a **parser**.
A **parse tree** describes a syntactic structure

- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar

Phases of Compiler-Syntax Analysis

- This is the second phase, it is also called as **parsing**
- It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.



Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.
- Ex: We use BNF (Backus Naur Form) to specify a CFG

  assgstmt -> identifier := expression

  expression -> identifier

  expression -> number

  expression -> expression + expression

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Semantic Analysis

Phases of Compiler-Semantic Analysis

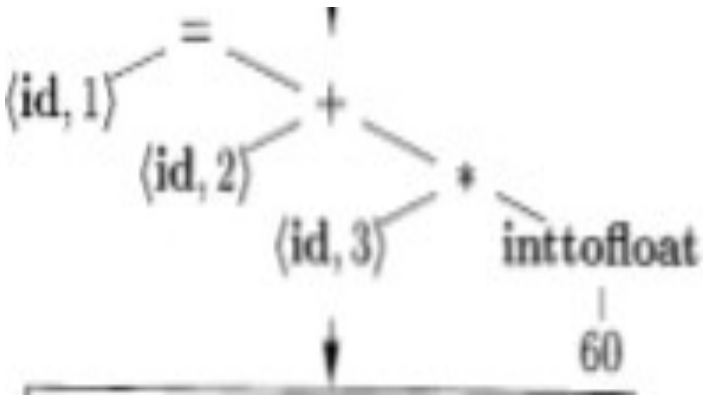
- Semantic analysis checks whether the parse tree constructed follows the rules of language.
- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is **type checking**

Phases of Compiler-Semantic Analysis

- Suppose that position, initial, and rate have been declared to be floating-point numbers and that the lexeme 60 by itself forms an integer.
- The type checker in the semantic analyzer discovers that the operator

* is applied to a floating-point number rate and an integer 60.

- In this case, the integer may be converted into a floating-point number.



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Intermediate Code Generation

Phases of Compiler-Intermediate Code Generation

- After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- It represents a program for some abstract machine.
- It is in between the high-level language and the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes

Phases of Compiler-Intermediate Code Generation

- An intermediate form called three-address code were used
- It consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Code Optimization

Phases of Compiler-Code Optimization

- The next phase does code optimization of the intermediate code.
- Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Code Generation

Phases of Compiler-Code Generation

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

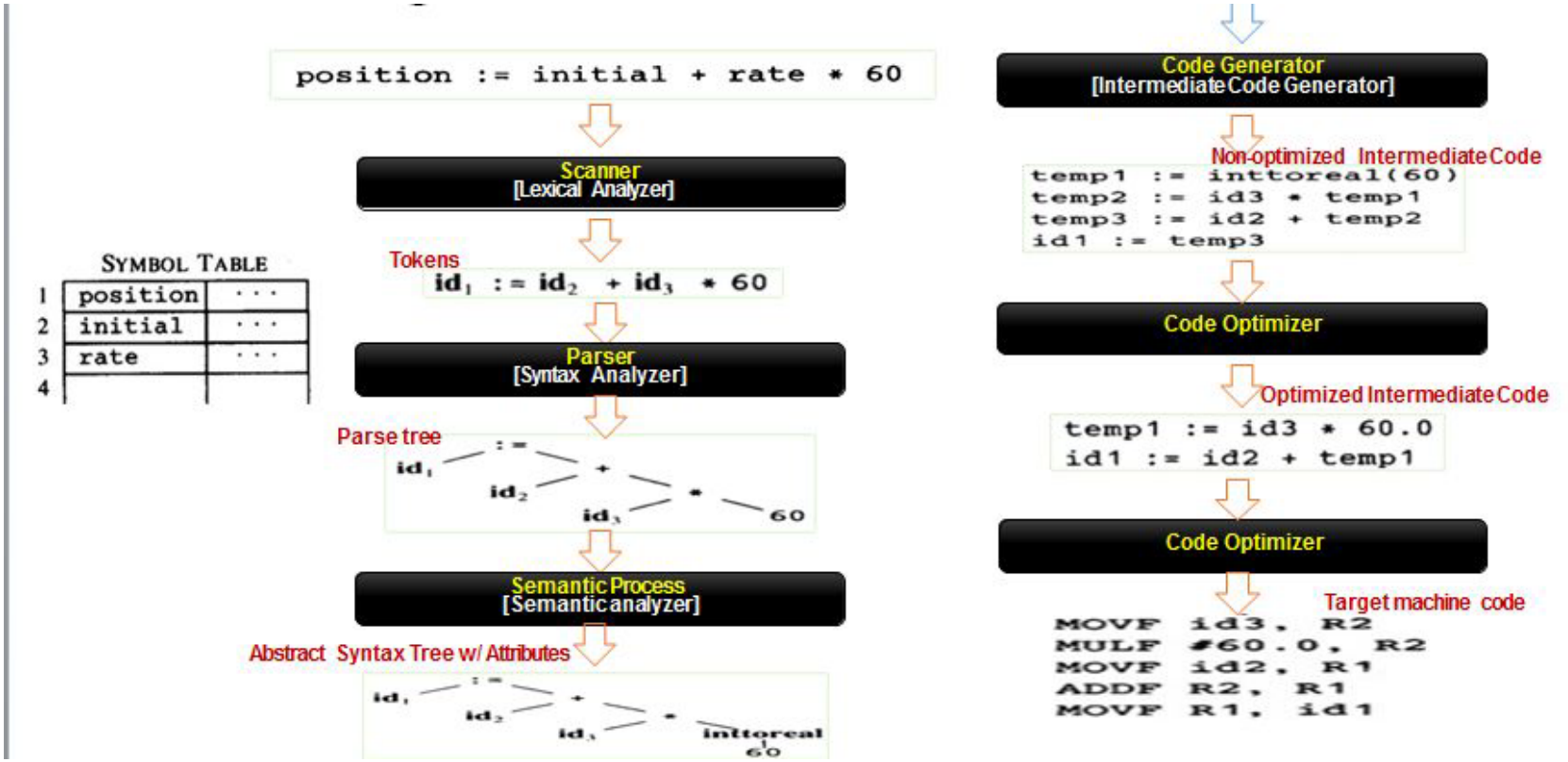
- In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes

Phases of Compiler-Code Generation

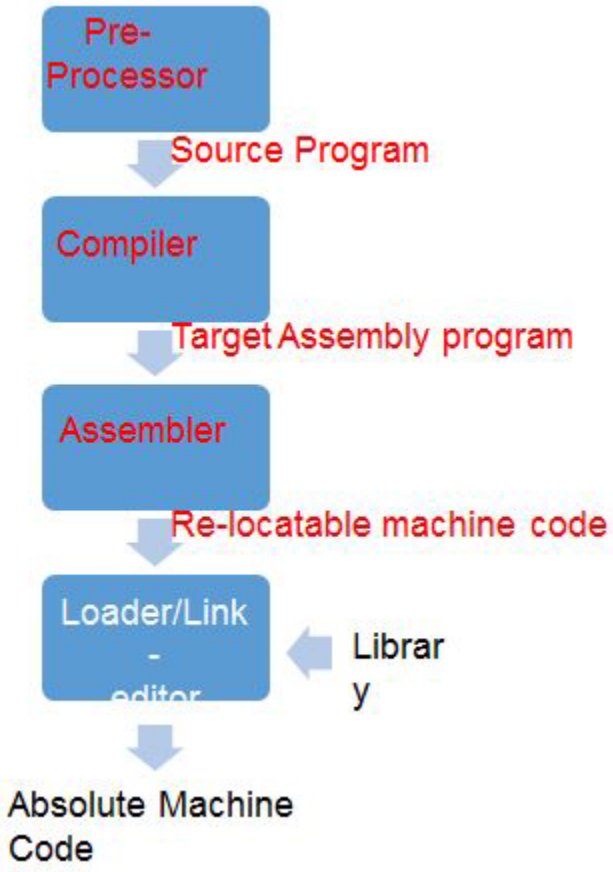
- For example, using registers R1 and R2, the intermediate code might get translated into the machine code
- The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers.

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

Phases of Compiler-Translation of assignment statement



Cousins of Compiler- Language Processing System



Preprocessor

- Pre-processors produce input to compilers
- The functions performed are:
  - **Macro processing** - allows user to define macros
  - **File inclusion** - include header files into the program
  - **Rational pre-processors** - It augment older languages with more modern flow-of-control and data structuring facilities
  - **Language extension** - It attempt to add capabilities to the language by what

amounts to built-in macros. (embed query in C)

Assembler

- Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operation

```
MOV
a,R1
ADD
#2,R1
MOV
R1,b
```

- Some compiler produce assembly code , which will be passed to an assembler for further processing
- Some other compiler perform the job of assembler, producing relocatable machine code which will be passed directly to the loader/link editor

Two-Pass Assembler

- This is the simplest form of assembler
- In First pass, all the identifiers that denote storage location are found and stored in a symbol table.

Let consider b=a+2

Identifier	Address
a	0
b	4

Loader/Link editor

- Loading – It Loads the relocatable machine code to the proper location
- Link editor allows us to make a single program from several files of relocatable machine code

Compiler Construction Tool

Parser generator	• Automatically produce syntax analyzer from a grammatical programming language
Scanner Generator	• It produce lexical analyzer from a regular expression
Syntax directed translation engine	• Generate intermediate code
Code generator	• Intermediate language into machine language
Data flow analysis engines	• It shows how values are transmitted from one part to other part
Compiler construction toolkit	• Integrate set of routines in different phases

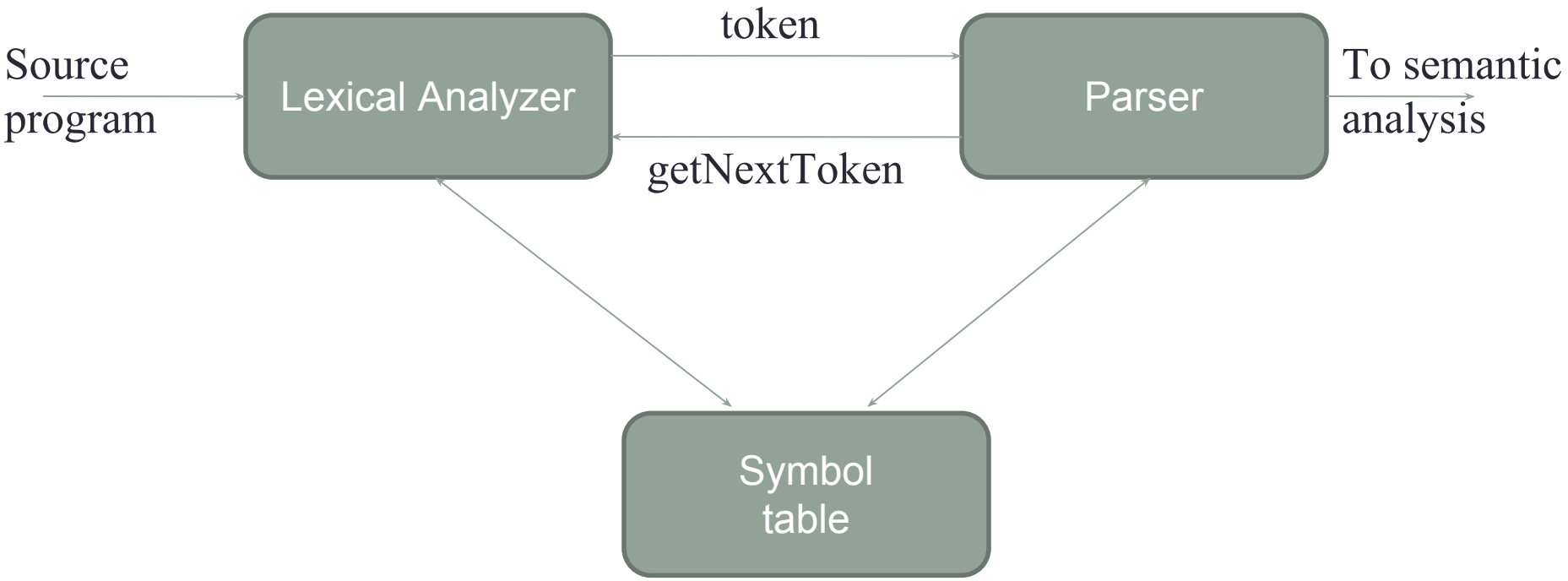
Role of a Lexical Analyzer

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

Why to separate Lexical analysis and parsing

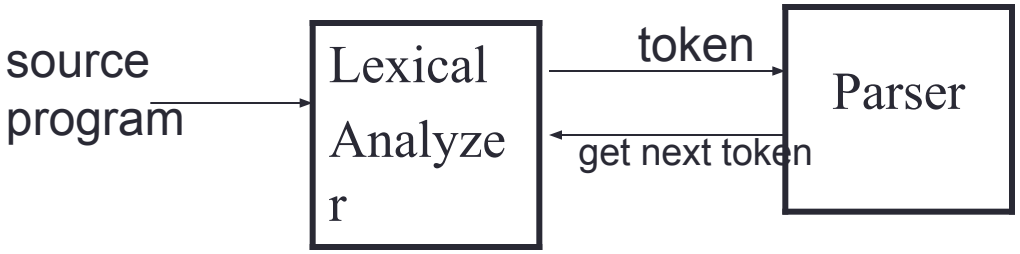
1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

The role of lexical analyzer



Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Lexical errors

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Token

- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits
  - The actual string (newval) is called as *lexeme*.
  - Tokens: identifier, number, addop, delimeter, …
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the *attribute* of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.

Token

- Some attributes:
  - <id,attr> where attr is pointer to the symbol table
  - <assgop,_> no attribute is needed (if there is only one assignment operator)
  - <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identifies a lexeme.
- ***Regular expressions*** are widely used to specify patterns.

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

printf("total = %d\n", score);

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.

Terminology of Languages

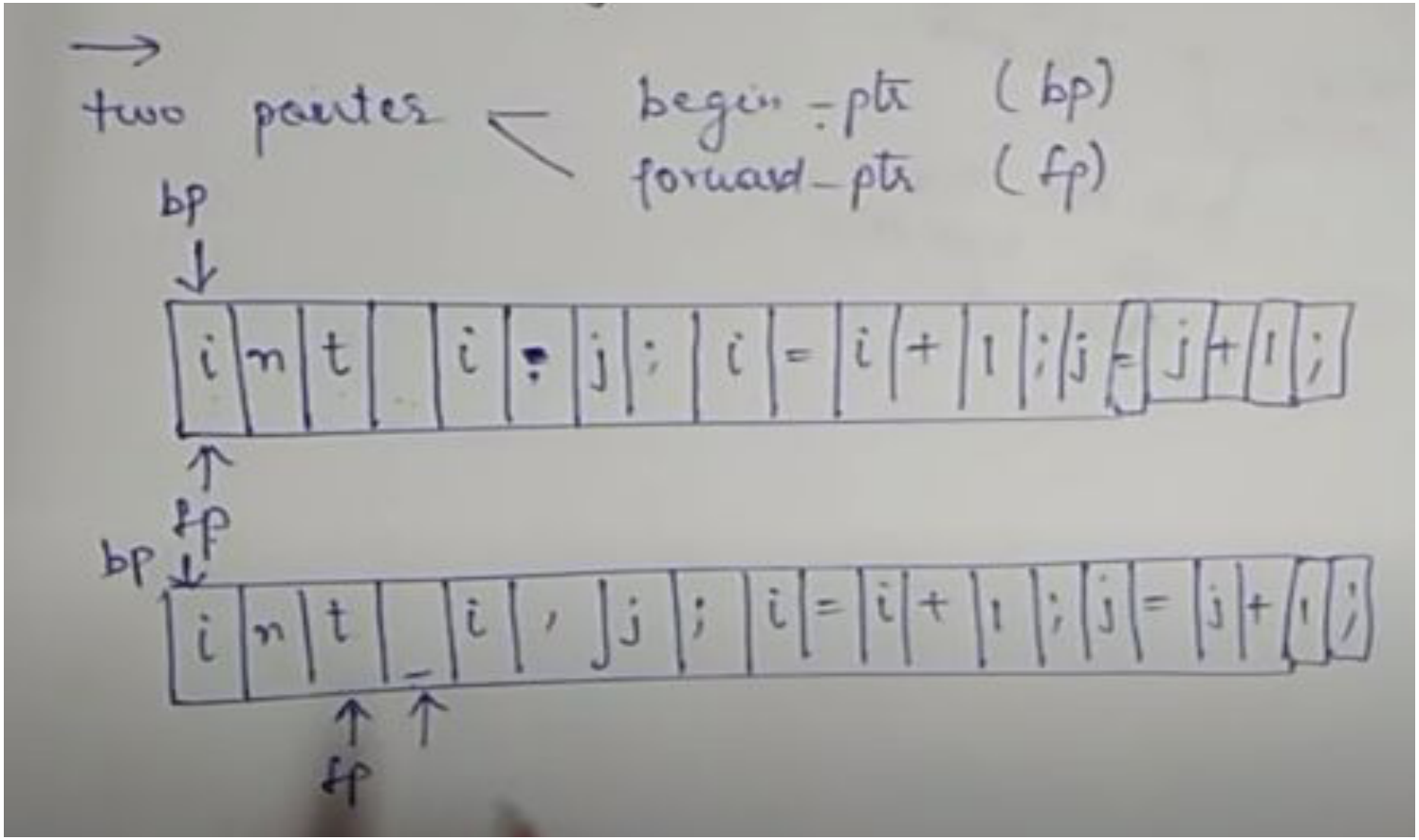
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - s^n = s s s .. s (n times) $s^0 = \varepsilon$

Input buffering

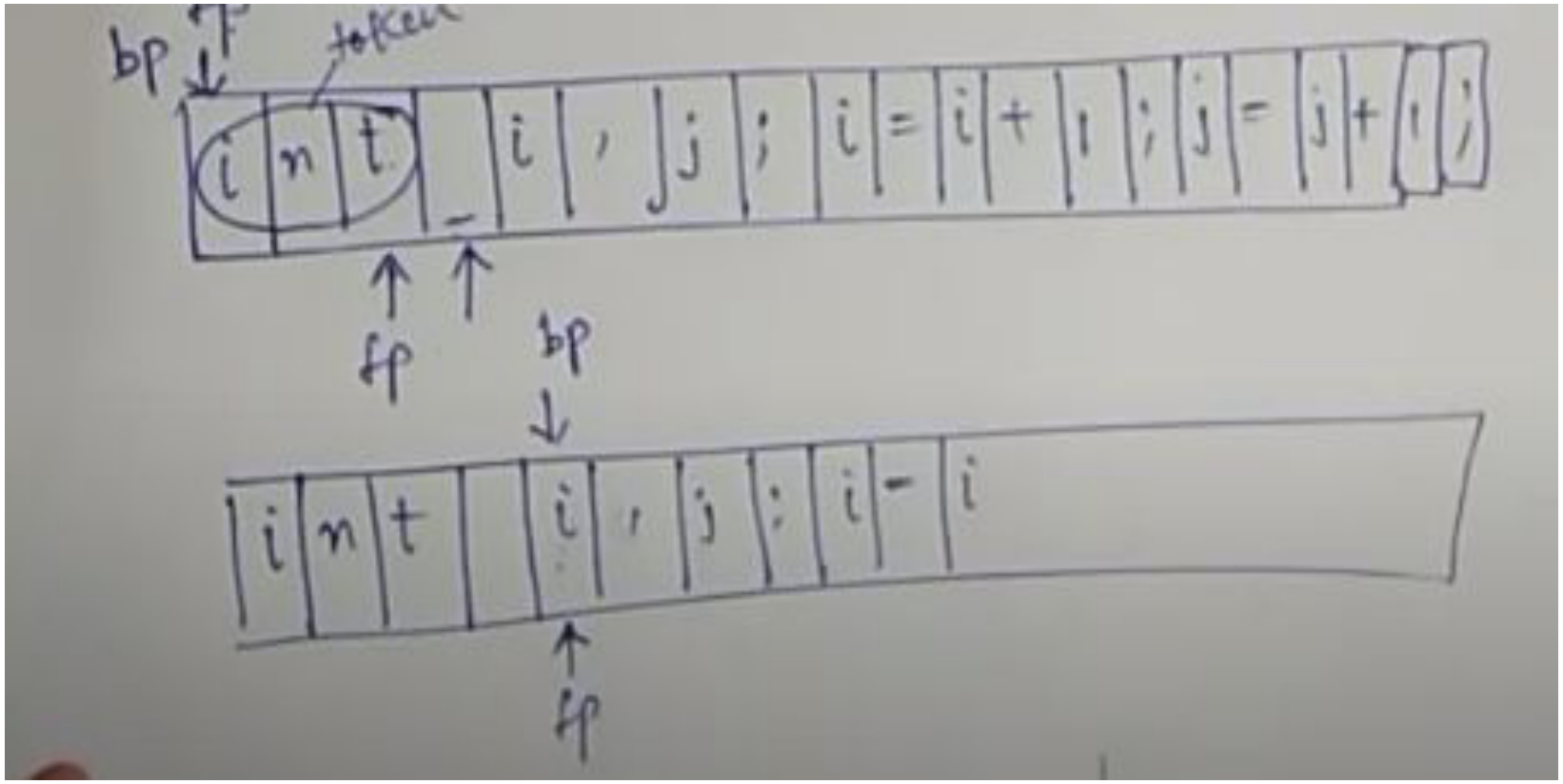
- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

A **buffer** contains data that is stored for a short amount of time, typically in the computer's memory (RAM). The purpose of a **buffer** is to hold data right before it is used.

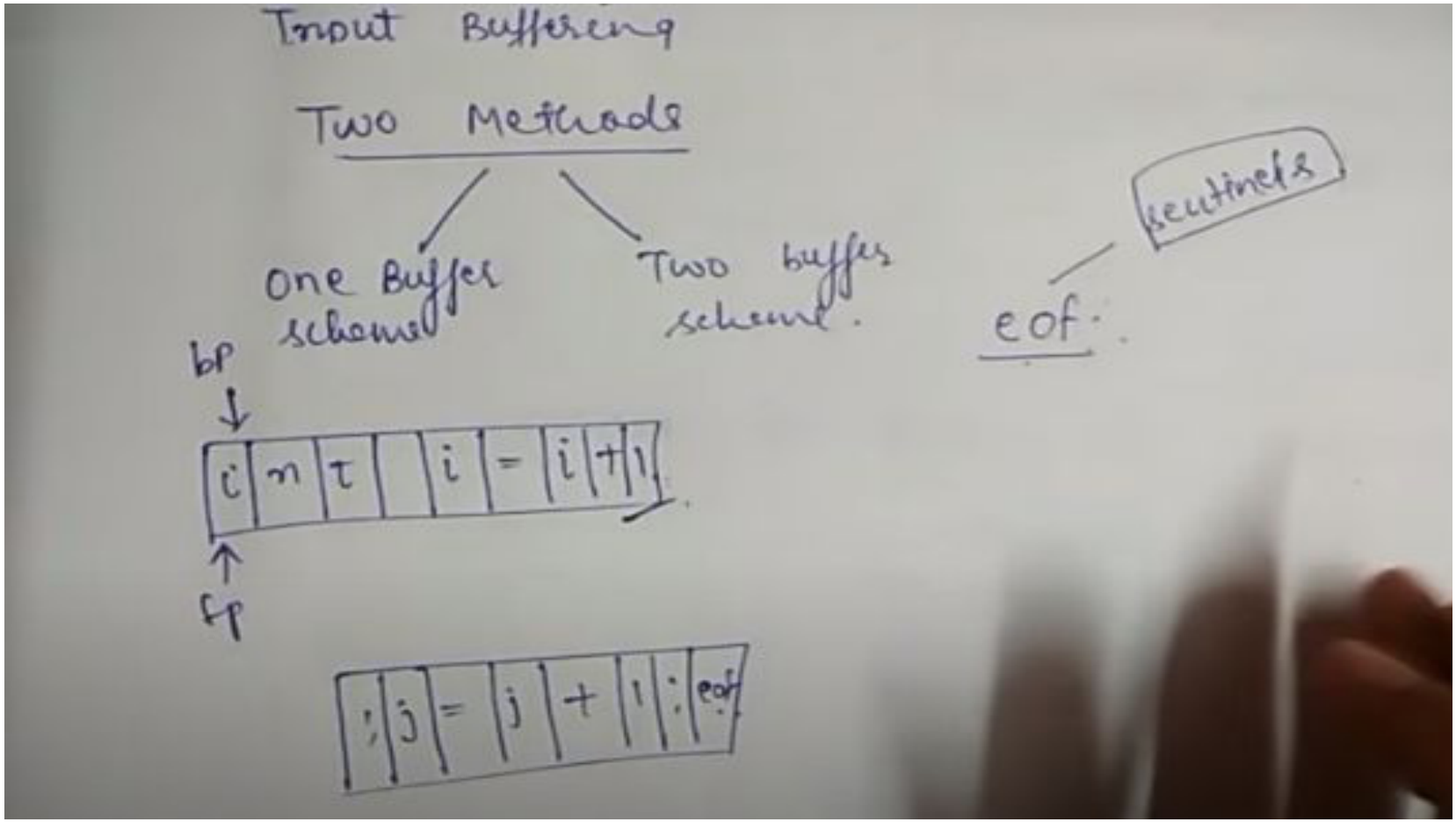
Cont..,



Cont..,



Cont..,


Input Buffering
Two Methods
One Buffer scheme
Two buffer scheme.
sentinels
eof.
bp
fp
i n t i = i + 1
; j = j + 1 ; eof

Sentinels


```
Switch (*forward++) {
   case eof:
      if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
      }
      else if {forward is at end of second buffer) {
            reload first buffer;\
            forward = beginning of first buffer;
      }
      else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

...

dn -> rn

- Example:

letter_ -> A | B | ... | Z | a | b | ... | Z | _

digit -> 0 | 1 | ... | 9

id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero or one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| ε

expr -> term **relop** term

| term

term -> **id**

| **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

  digit -> [0-9]

  Digits -> digit+

  number -> digit(.digits)? (E[+-]? Digit)?

  letter -> [A-Za-z_]

  id -> letter (letter|digit)*

  If -> if

  Then -> then

  Else -> else

  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$
- Kleene Closure
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- Positive Closure
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	{ε}
$a \in \Sigma$	{a}
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\,(r_2)$	$L(r_1)\,L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon,0,00,000,0000,....\}$
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter → A | B | ... | Z | a | b | ... | z

  digit → 0 | 1 | ... | 9

  id → letter (letter | digit) *

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit → 0 | 1 | ... | 9

  digits → digit +

  opt-fraction → (. digits) ?

  opt-exponent → (E (+|-)? digits) ?

  unsigned-num → digits opt-fraction opt-exponent

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

...

dn -> rn

- Example:

letter_ -> A | B | ... | Z | a | b | ... | Z | _

digit -> 0 | 1 | ... | 9

id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero or one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| ε

expr -> term **relop** term

| term

term -> **id**

| **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

  digit -> [0-9]

  Digits -> digit+

  number -> digit(.digits)? (E[+-]? Digit)?

  letter -> [A-Za-z_]

  id -> letter (letter|digit)*

  If -> if

  Then -> then

  Else -> else

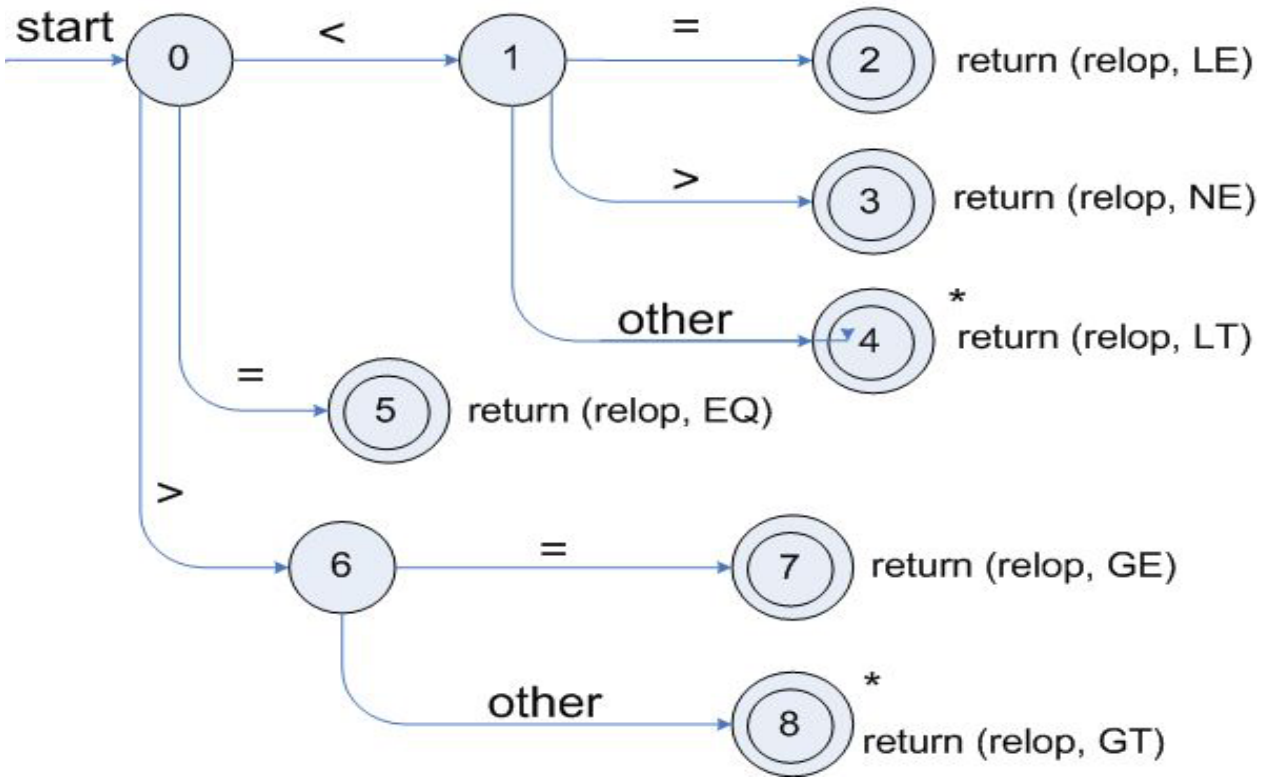
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

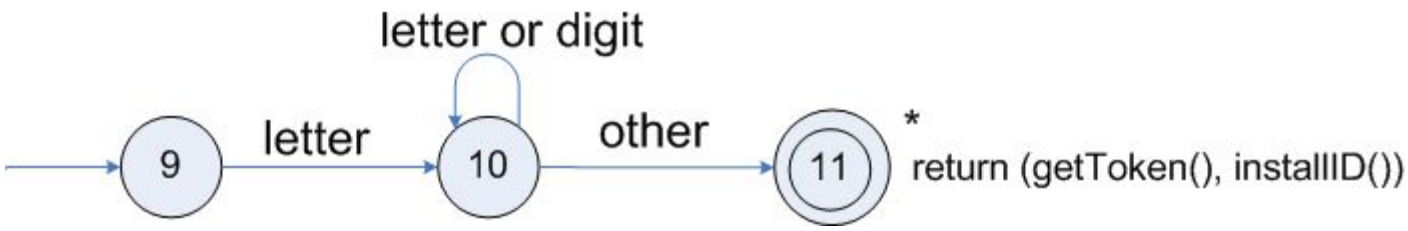
Transition diagrams

- Transition diagram for relop



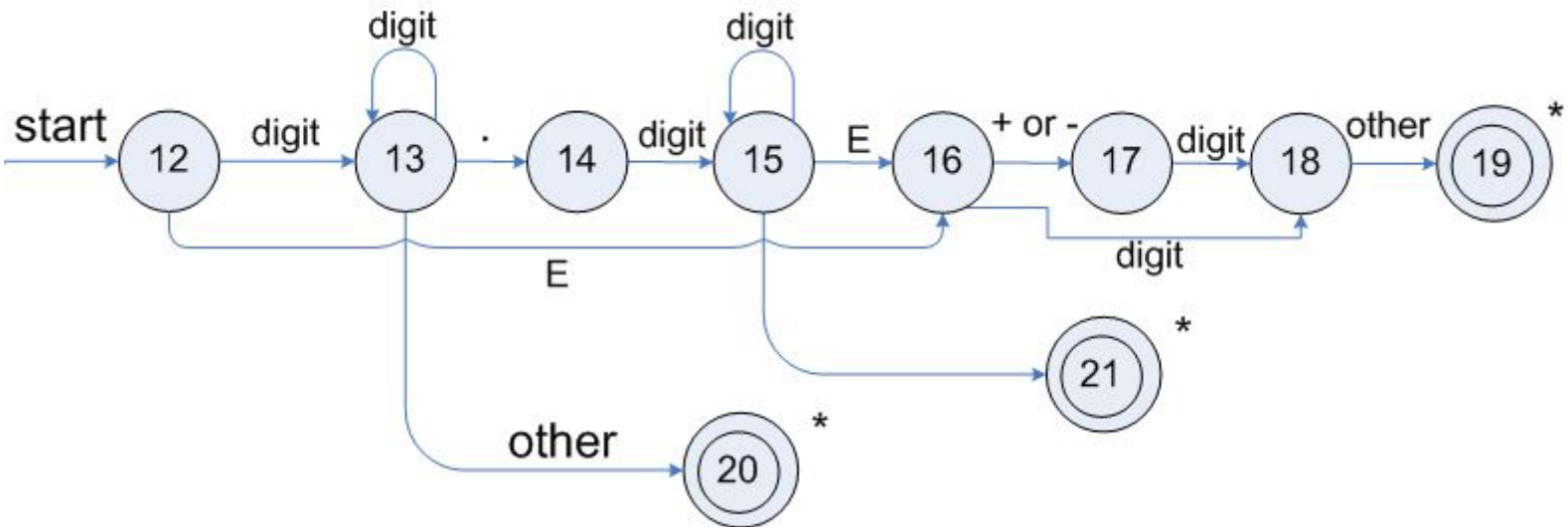
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



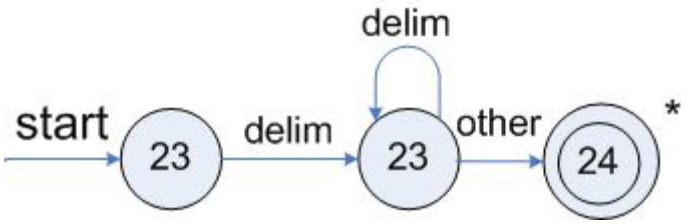
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Design of a Lexical Analyzer (LEX)

Design of a Lexical Analyzer

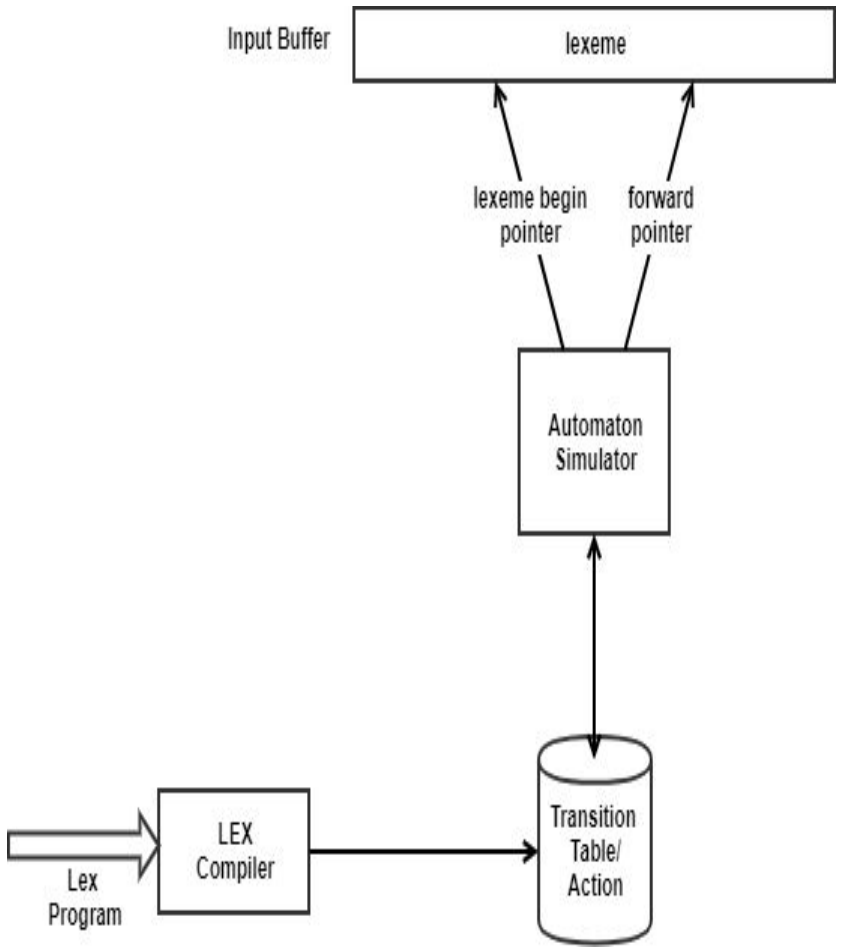
- LEX is a software tool that automatically construct a lexical analyzer from a program
- The Lexical analyzer will be of the form

```
P1    {action 1}
P2    {action 2}
--
--
```

- Each pattern pi is a regular expression and action i is a program fragment that is to be executed whenever a lexeme matched by pi is found in the input
- If two or more patterns that match the longest lexeme, the first listed matching pattern is chosen

Design of a Lexical Analyzer

- Here the Lex compiler constructs a transition table for a finite automaton from the regular expression pattern in the Lex specification
- The lexical analyzer itself consists of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer

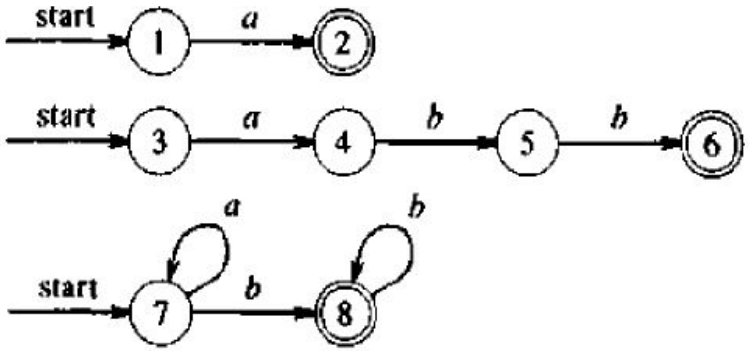


Example

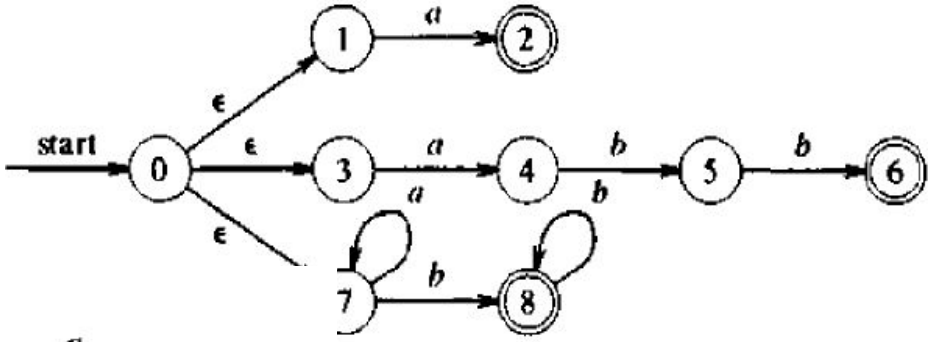
Consider Lexeme

a {action A1 for pattern p1}
abb{action A2 for pattern p2}

a*b* {action A3 for pattern p3}



(a) NFA for a, abb, and a^*b^+.



) Combined NFA.

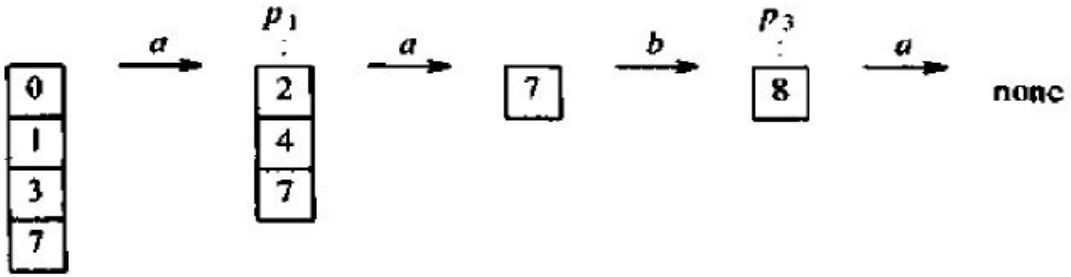


Fig. 3.36. Sequence of sets of states entered in processing input *aaba*.

LEX in use

- An input file, which we call **lex.1**, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms lex. 1 to a C program, in a file that is always named **lex. yy . c.**
- The latter file is compiled by the C compiler into a file called a. out.
- The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

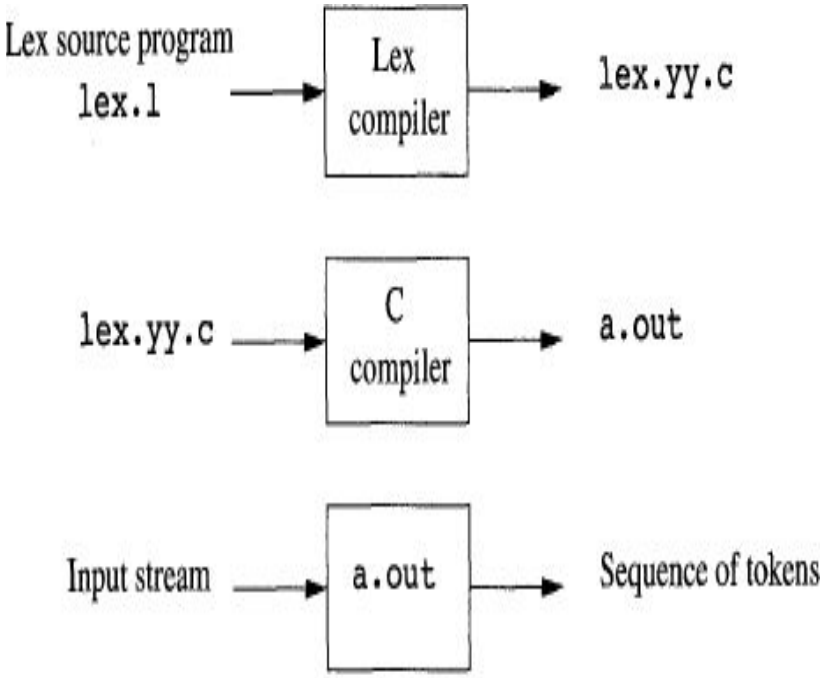


Figure 3.22: Creating a lexical analyzer with Lex

General format

1. The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token)
2. The translation rules each have the form

Pattern { Action)

- Each pattern is a regular expression, which may use the regular definitions of the declaration section.
- The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

3. The third section holds whatever additional functions are used in the actions.

declarations
%%
translation rules
%%
auxiliary functions

Consider the following statement

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

digit	→	[0-9]
digits	→	$digit^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

Figure 3.11: Patterns for tokens of Example 3.8

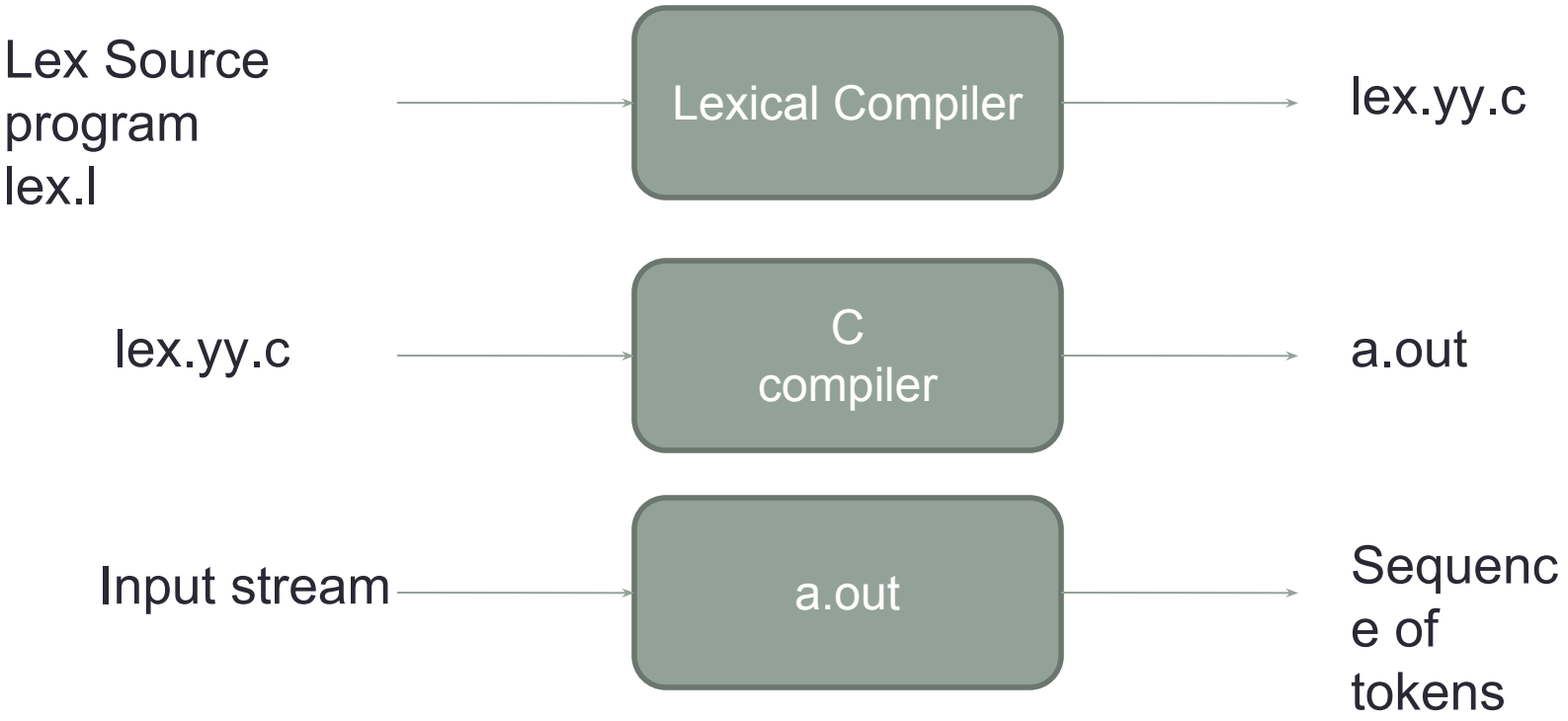
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Figure 3.12: Tokens, their patterns, and attribute values

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
```

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lexical Analyzer Generator - Lex



Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- **Recognizer ---**A recognizer for a language is a program that takes as input a string x answers 'yes' if x is a sentence of the language and 'no' otherwise.

- A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a **finite automaton.**

- Finite Automaton can be
- Deterministic
- Non-deterministic

Finite Automata

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \xrightarrow{\text{input}} \text{state}$

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

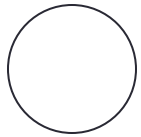
- Is read

In state s_1 on input "a" go to state s_2

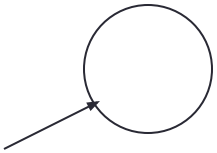
- If end of input
  - If in accepting state => accept, otherwise => reject
- If no transition possible => reject

Finite Automata State Graphs

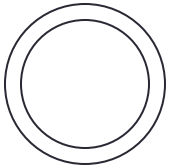
- A state



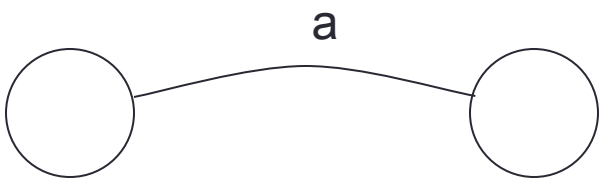
- The start state



- An accepting state



- A transition



Finite Automata

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

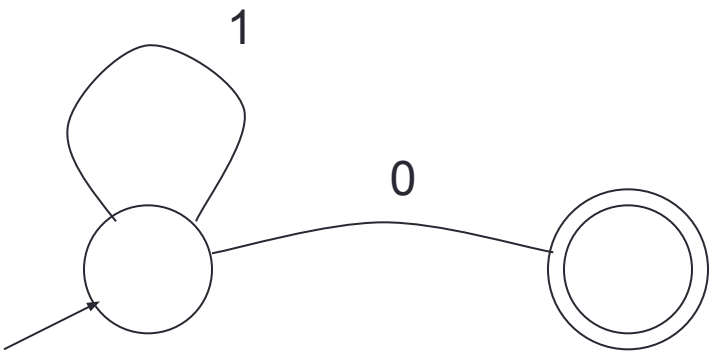
A Simple Example

- A finite automaton that accepts only “1”

- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted.

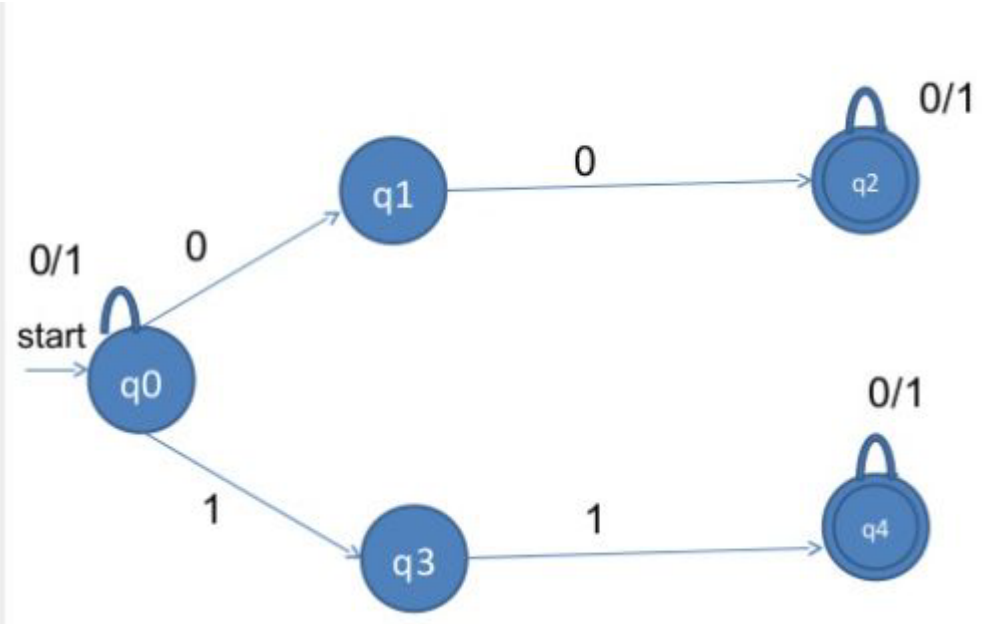
NFA

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Design a NFA for the language L=all strings over {0,1} that have atleast two consecutive 0's or 1's

NFA

Design a NFA for the language L=all strings over {0,1} that have atleast two consecutive 0's or 1's



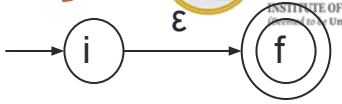
Transition Table

	0	1
→ qo	{qo,q1}	{qo,q3}
q1	q2	?
* q2	q2	q1
q3	?	q4
*q4	Q4	Q4

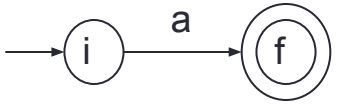
Converting A Regular Expression into A NFA (Thomson's Construction)

- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method. It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Thomson's Construction (cont.)

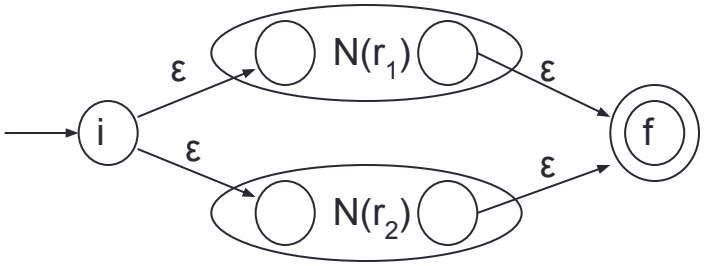


- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ

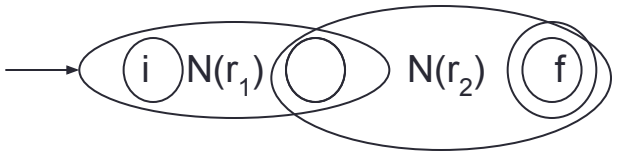
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

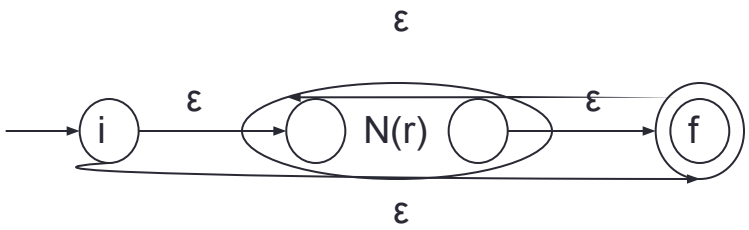
- For regular expression $r_1 r_2$



NFA for $r_1 r_2$

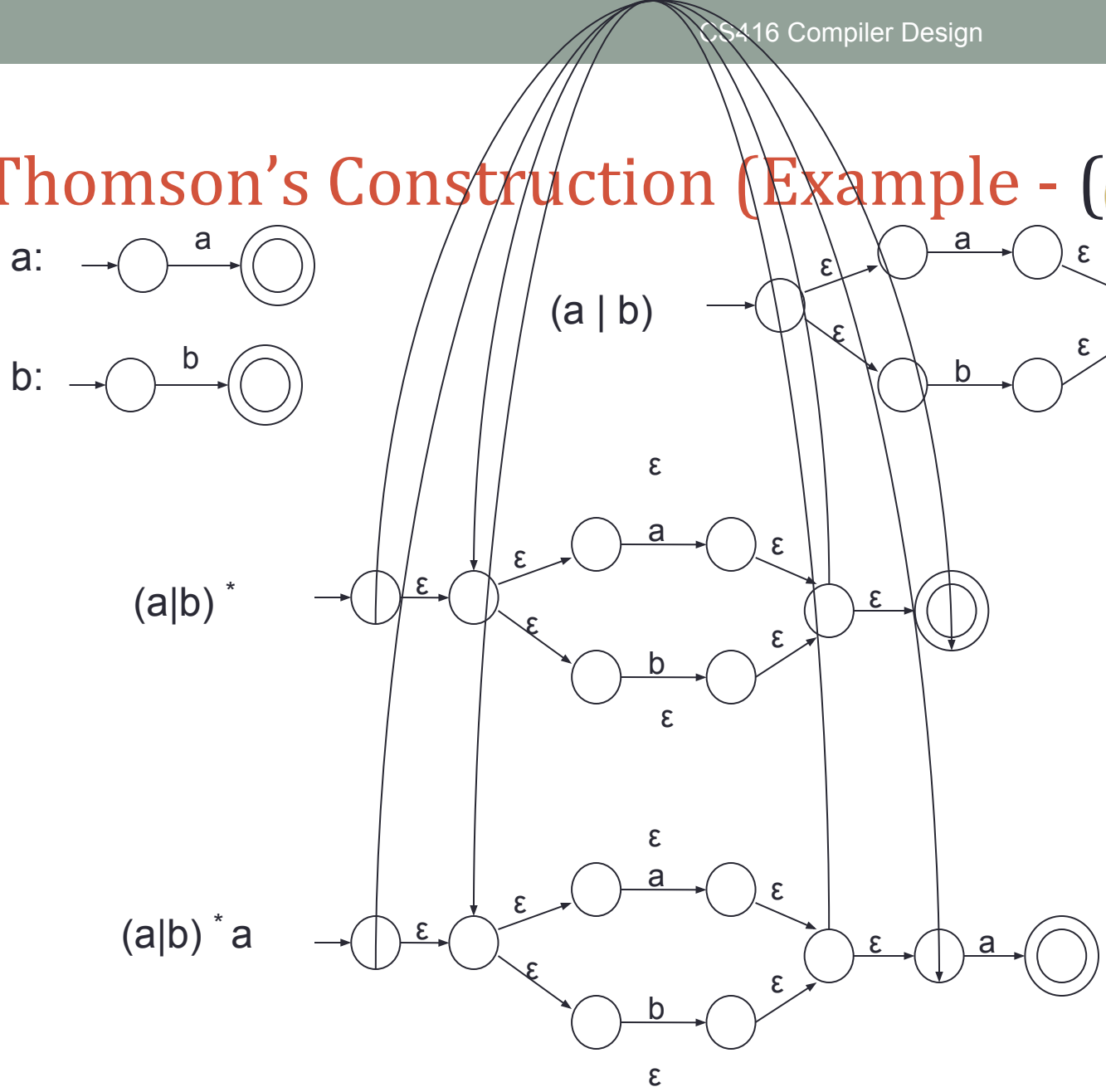
Final state of $N(r_2)$ become final state of $N(r_1 r_2)$

- For regular expression r^*



NFA for r^*

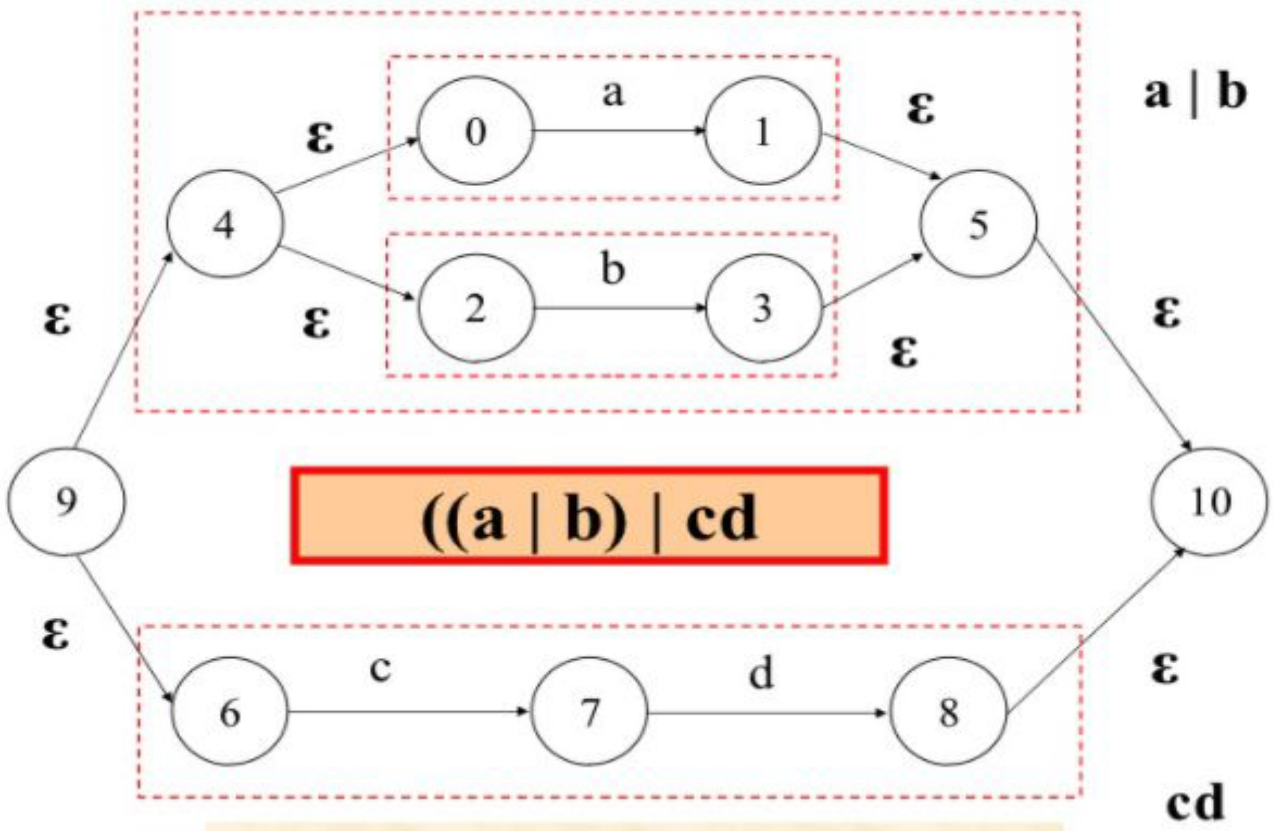
Thomson's Construction (Example - (a|b)*a)



Thompson's Method

Construct an NFA that recognizes

((a | b) | cd)



Converting a NFA into a DFA (subset construction)

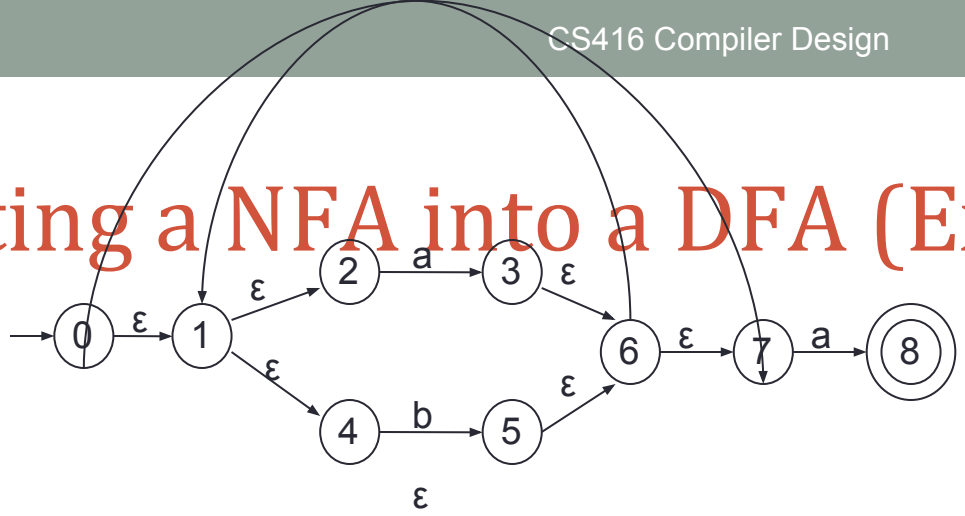
put ε-closure({s_0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S_1 in DS) do
begin
    mark S_1
    for each input symbol a do
        begin
            S_2 ← ε-closure(move(S_1,a))
            if (S_2 is not in DS) then
                add S_2 into DS as an unmarked state
            transfunc[S_1,a] ← S_2
        end
end

ε-closure({s_0}) is the set of all states can b accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure({s_0})

Converting a NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] □ S_1 transfunc[S_0,b] □ S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] □ S_1 transfunc[S_1,b] □ S_2

⇓ mark S_2

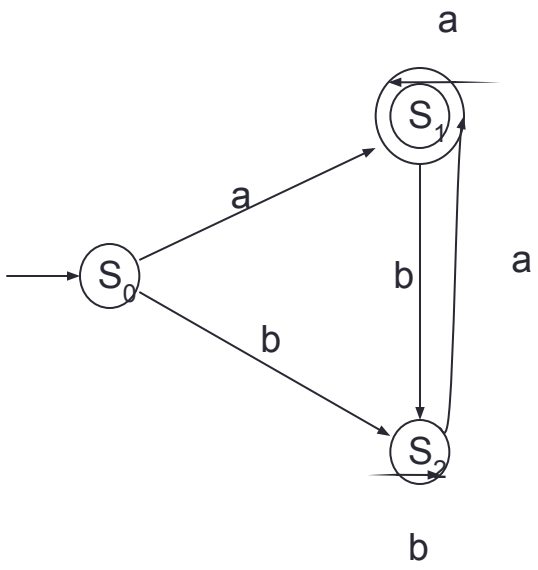
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] □ S_1 transfunc[S_2,b] □ S_2

Converting a NFA into a DFA (Example – cont.)

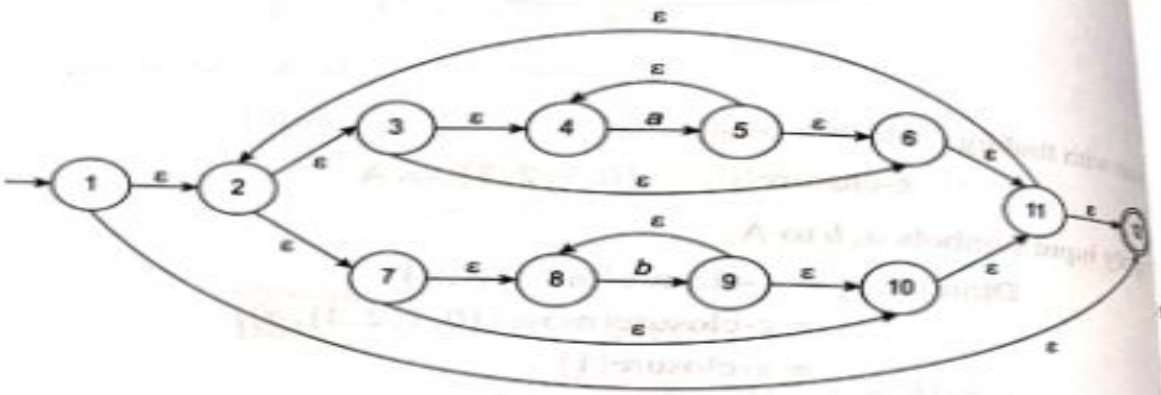
S_0 is the start state of DFA since 0 is a member of S_0={0,1,2,4,7}
S_1 is an accepting state of DFA since 8 is a member of S_1 = {1,2,3,4,6,7,8}



$(a^* \mid b^*)^*$

Answer:

Step 1: Construct NFA



Step 2: Start with finding ϵ-closure of state 1

$$\epsilon\text{-closure}(1) = \{1, 2, 3, 4, 6, 11, 12, 7, 8, 10\}$$
$$= \{1, 2, 3, 4, 6, 7, 8, 10, 11, 12\} \Rightarrow \text{A}$$

Step 3: Apply input symbols a, b to A

$$\text{Dtran}[\text{A}, a] = \epsilon\text{-closure}(\text{move}(\text{A}, a))$$
$$= \epsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12\}, a))$$
$$= \epsilon\text{-closure}(5)$$
$$= \{5, 6, 11, 12, 2, 3, 4, 7, 8, 10\}$$
$$= \{2, 3, 4, 5, 6, 7, 8, 10, 11, 12\} \Rightarrow \text{B}$$
$$\text{Dtran}[\text{A}, a] = \text{B}$$
$$\text{Dtran}[\text{A}, b] = \epsilon\text{-closure}(\text{move}(\text{A}, b))$$
$$= \epsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8, 10, 11, 12\}, b))$$
$$= \epsilon\text{-closure}(9)$$

$$= \{9, 10, 11, 12, 2, 3, 4, 6, 7, 8, 10\}$$
$$= \{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\} \Rightarrow \mathbf{C}$$

$$\text{Dtran}[\text{A}, b] = \mathbf{C}$$

Step 4: Apply input symbols to new state B

$$\text{Dtran}[\text{B}, a] = \varepsilon\text{-closure}(\text{move}(\text{B}, a))$$
$$= \varepsilon\text{-closure}(\text{move}(\{2, 3, 4, 5, 6, 7, 8, 10, 11, 12\}, a))$$
$$= \varepsilon\text{-closure}(5)$$
$$= \{2, 3, 4, 5, 6, 7, 8, 10, 11, 12\} \Rightarrow \mathbf{B}$$

$$\text{Dtran}[\text{B}, a] = \mathbf{B}$$

$$\text{Dtran}[\text{B}, b] = \varepsilon\text{-closure}(\text{move}(\text{B}, b))$$
$$= \varepsilon\text{-closure}(\text{move}(\{2, 3, 4, 5, 6, 7, 8, 10, 11, 12\}, b))$$
$$= \varepsilon\text{-closure}(9)$$
$$= \{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\} \Rightarrow \mathbf{C}$$

$$\text{Dtran}[\text{B}, b] = \mathbf{C}$$

Step 5: Apply input symbols to new state C

$$\text{Dtran}[\text{C}, a] = \varepsilon\text{-closure}(\text{move}(\text{C}, a))$$
$$= \varepsilon\text{-closure}(\text{move}(\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}, a))$$
$$= \varepsilon\text{-closure}(5)$$
$$= \{2, 3, 4, 5, 6, 7, 8, 10, 11, 12\} \Rightarrow \mathbf{B}$$

$$\text{Dtran}[\text{C}, a] = \mathbf{B}$$

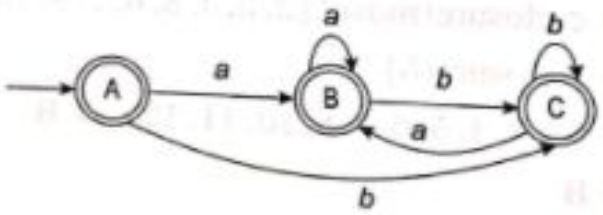
$$\text{Dtran}[\text{C}, b] = \varepsilon\text{-closure}(\text{move}(\text{C}, b))$$
$$= \varepsilon\text{-closure}(\text{move}(\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}, b))$$
$$= \varepsilon\text{-closure}(9)$$
$$= \{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\} \Rightarrow \mathbf{C}$$

$$\text{Dtran}[\text{C}, b] = \mathbf{C}$$

Note:

- ✓ Start state is the ϵ-closure(1), i.e., A.
- ✓ Final state is the state that contains final state of given NFA.

Step 8: Construct transition diagram



On minimization of finite automata, the transition diagram will be reduced as below.