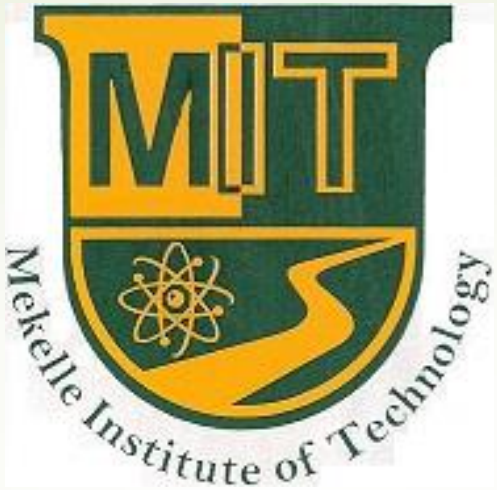


Compiler Design

Ch_1: Introduction to Compilers and lexical analyzer

Widassie Gerezgiher

Department of Computer Science and Engineering (VI 2nd semister)

Contents

Contents

Overview to Compilers

- High-level language fed to series of tools and OS components to get the desired code that can be used by machine➔ Language processing system
- Tools that work closely with compilers: Preprocessor, Assembler, Interpreter, Linker, Loader, Cross-Compiler, source to source Compiler
- **Preprocessor**: produce i/p for compilers
  - deals with macro-processing, augmentation, file inclusion, language extension,
- **Compiler vs Interpreter**

Compiler	Interpreter
High-level language into low-level machine language	High-level language into low-level machine language
Reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program	Reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence
Reads the whole program even if it encounters several errors.	lf an error occurs, an interpreter stops execution and reports it

Overview to Compilers

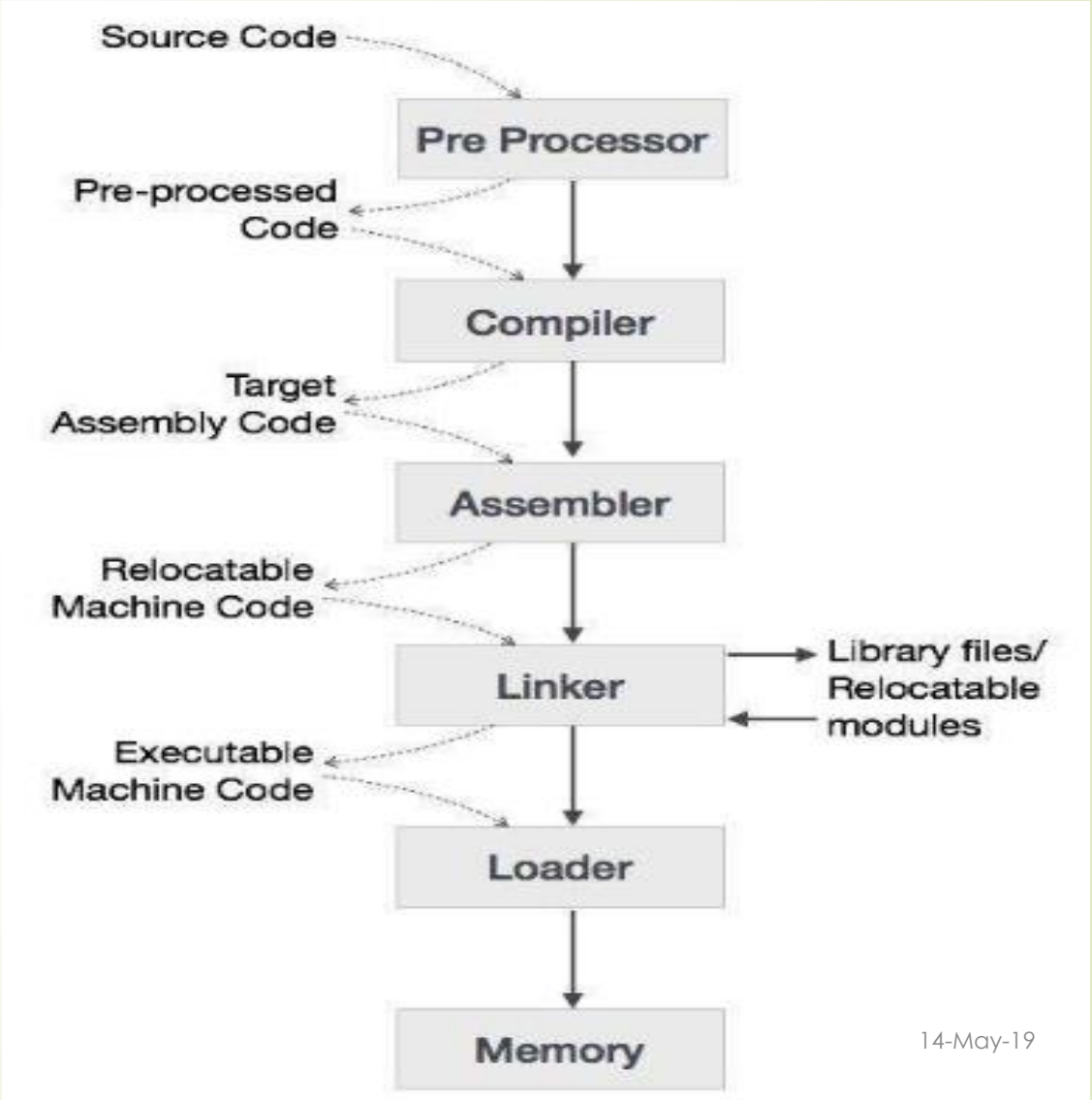
- **Assembler**: AL to MLL
  - o/p: Object file (contains a combination of machine instructions as well as the data required to place these instructions in memory)
- **Linker**: links and merges various object files together in order to make an executable file
  - Object files might have been compiled by separate assemblers
  - Makes the program instruction to have absolute references
- **Loade**r: is a part of operating system and
  - Responsible for loading executable files into memory and execute them
  - It calculates the size of a program (instructions and data) and
  - Creates memory space for it
  - It initializes various registers to initiate execution

Overview to Compilers

- **Cross-compiler**
  - A compiler that runs on platform (A) and is capable of generating executable code for platform (B)
- **Source-to-source Compiler**
  - A compiler that takes the source code of one programming language and translates it into the source code of another programming language

Overview to Compilers

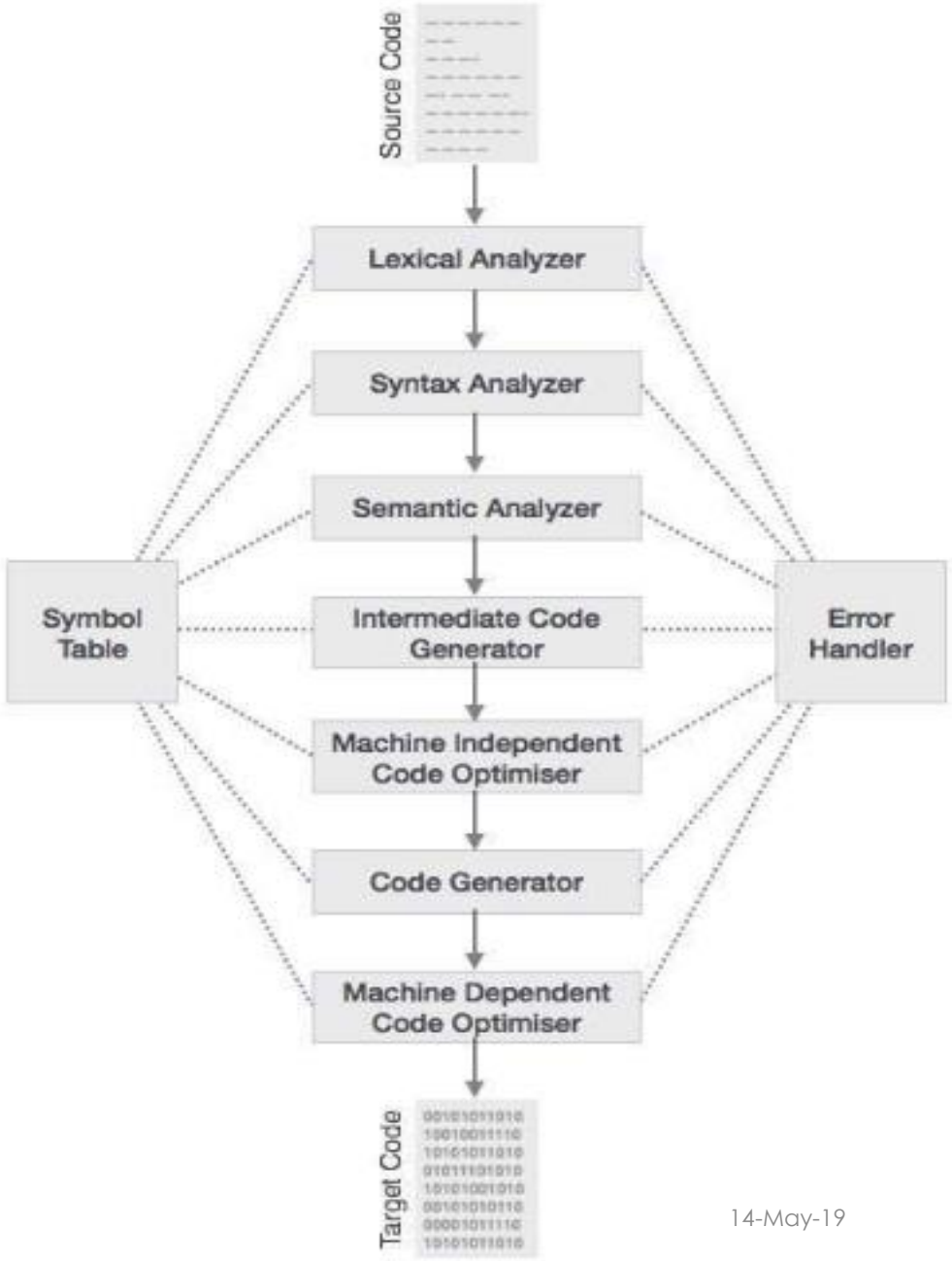
- **Language processing system**



Contents

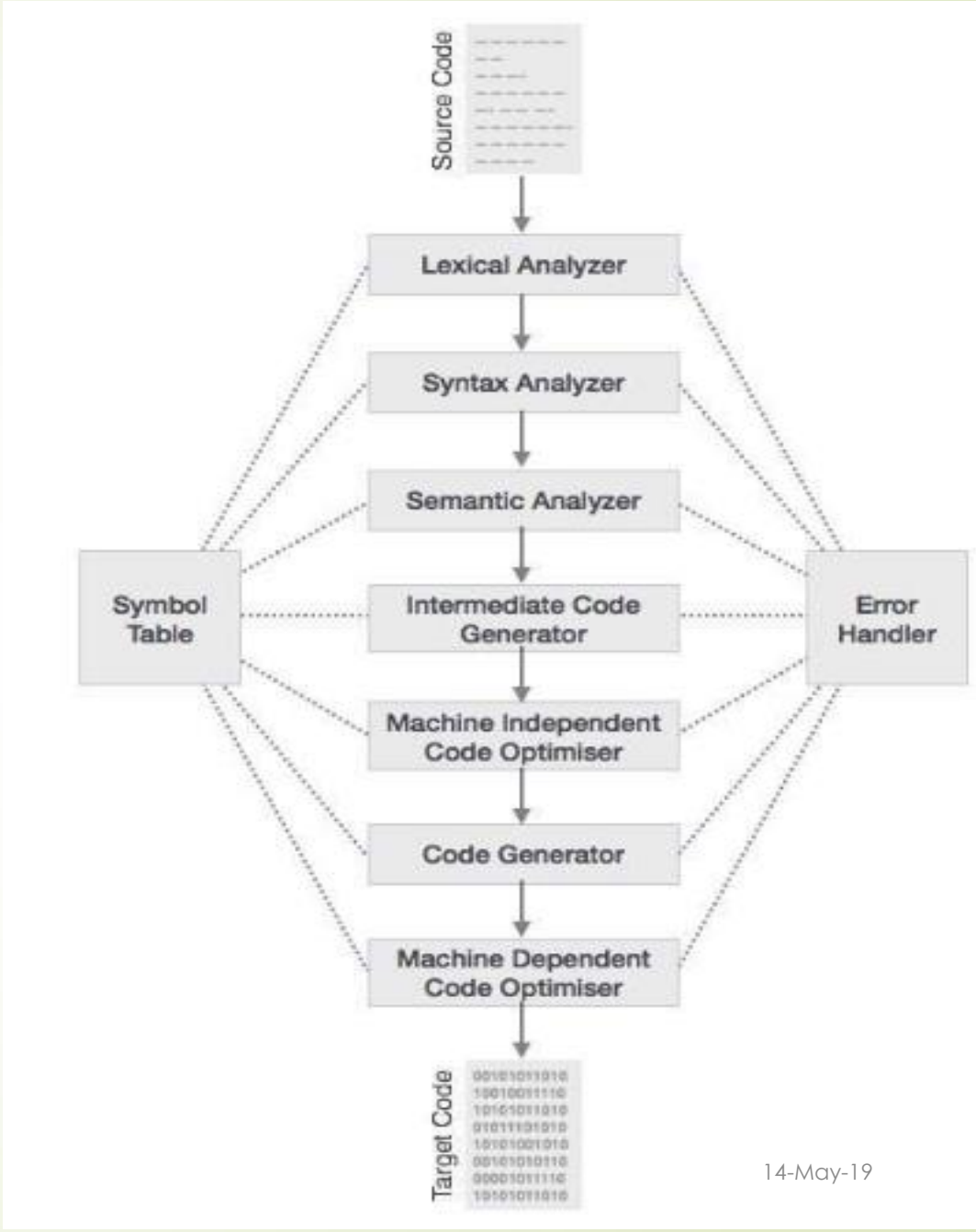
Phases of compiler

- **Symbol Table**: data-structure,
  - stores all identifier names along its types
- **Lexical analysis**: scans source code as stream of characters and converts it to toke
  - <token-name, attribute-value>
- **Syntax Analysis** (parsing):
  - i/p (tokens), o/p (syntax tree or parse tree)
- **Semantic Analysis**:
  - keeps track of identifiers, their types and expressions, if identifiers are declared before use or not etc
  - Annotated syntax tree as an output



Phases of compiler

- **Intermediate Code Generation**
  - Intermediate code is translated in a way that it makes it easier to be translated into the target machine code
- **Code Optimization**
  - Optimization of intermediate code
  - Removes unnecessary code lines, and
  - Arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory)

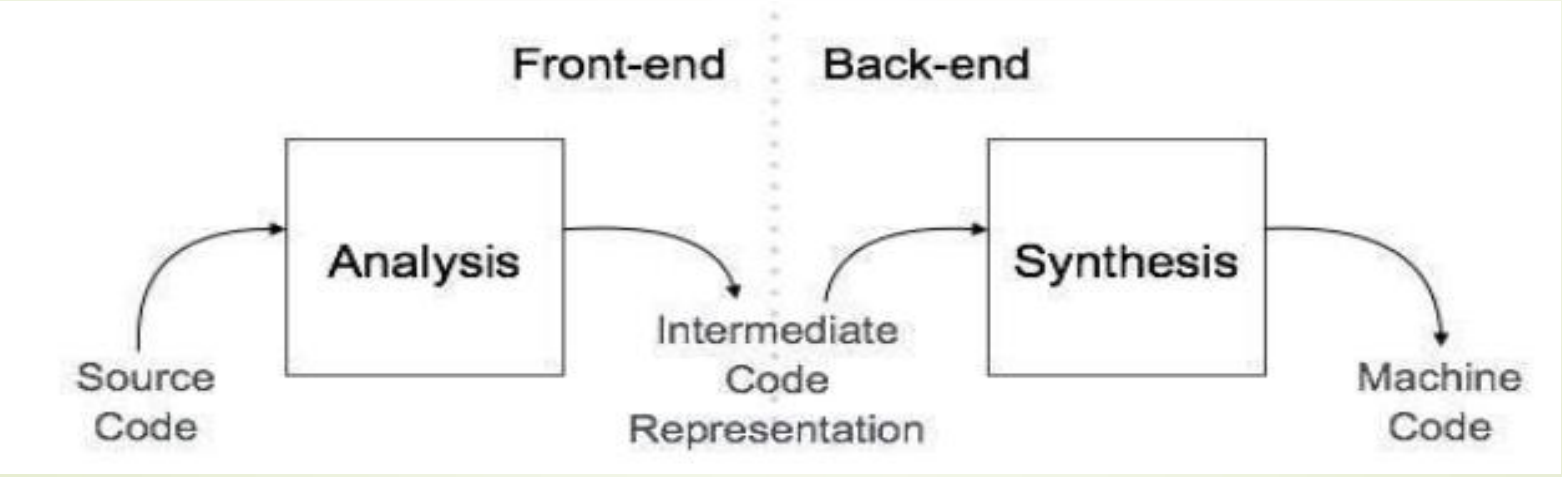


Contents

- Introduction to Compiler
  - Overview to Compilers
  - Phases of a Compiler
  - Grouping of Phases
- Role of Lexical Analyzer
  - Speciation of tokens
  - Recognition of tokens
- Finite Automata:
  - NFA and DFA
  - Constructing NFA from a Regular expression
  - Constructing DFA from Regular expression
  - Minimizing number of states

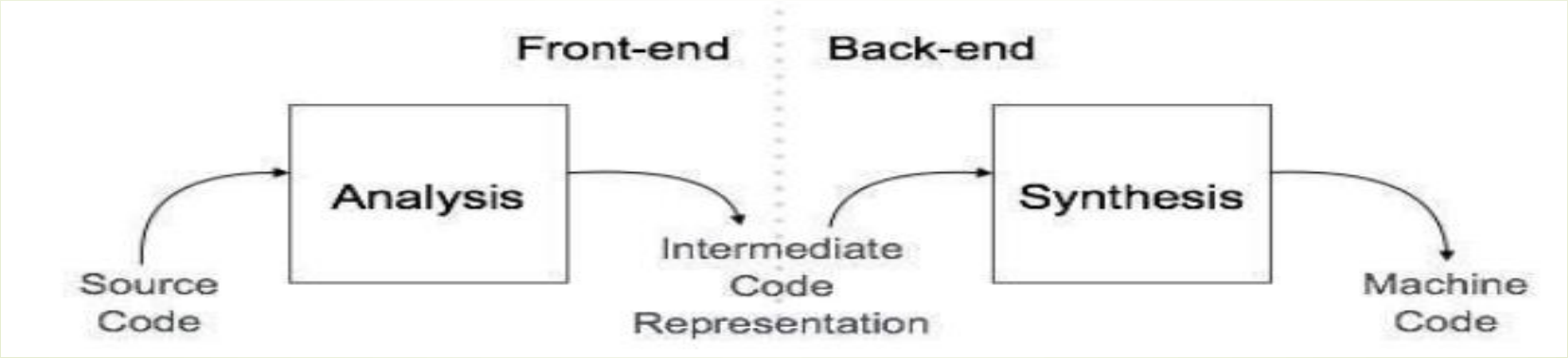
Grouping of Phases

- Grouping of Phases: Analysis, and synthesis phases
  1. **Analysis Phase**
     - Front end of compiler
     - Reads source program divide it to core parts
     - Checks lexical, grammar, and syntax errors
     - Generates IR and symbol table



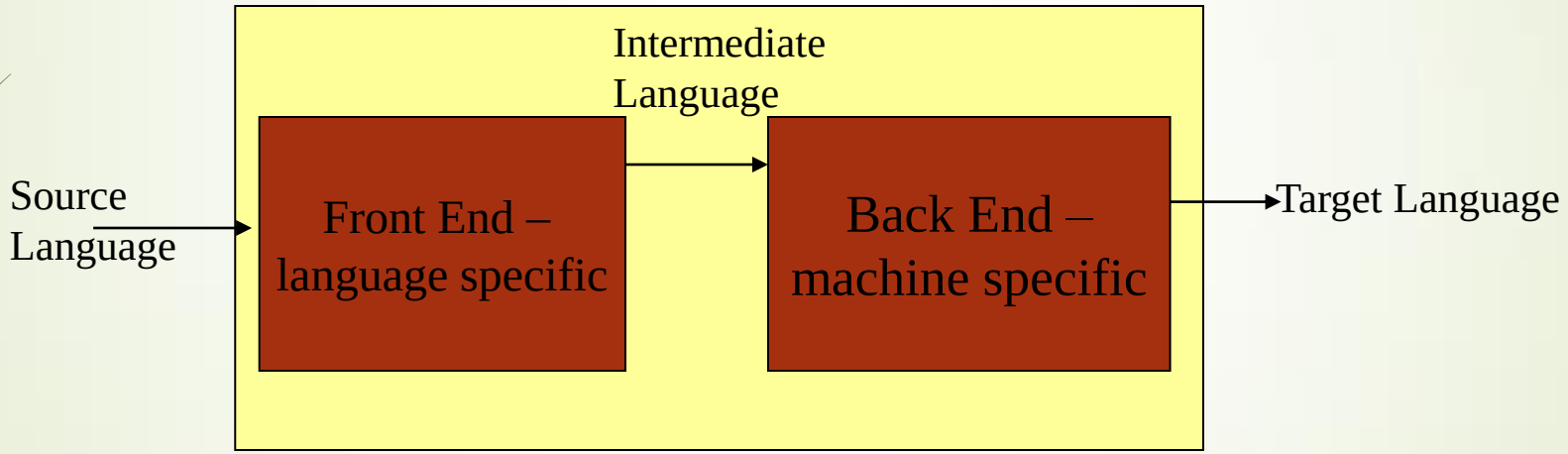
Grouping of Phases

- Grouping of Phases
  2. **Synthesis Phase**
  - Back end of compiler
  - Generates target program with the help of IR and symbol table
- **N.B**: A compiler can have many phases and passes
  - **Pass** : A pass refers to the traversal of a compiler through the entire program
  - **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase

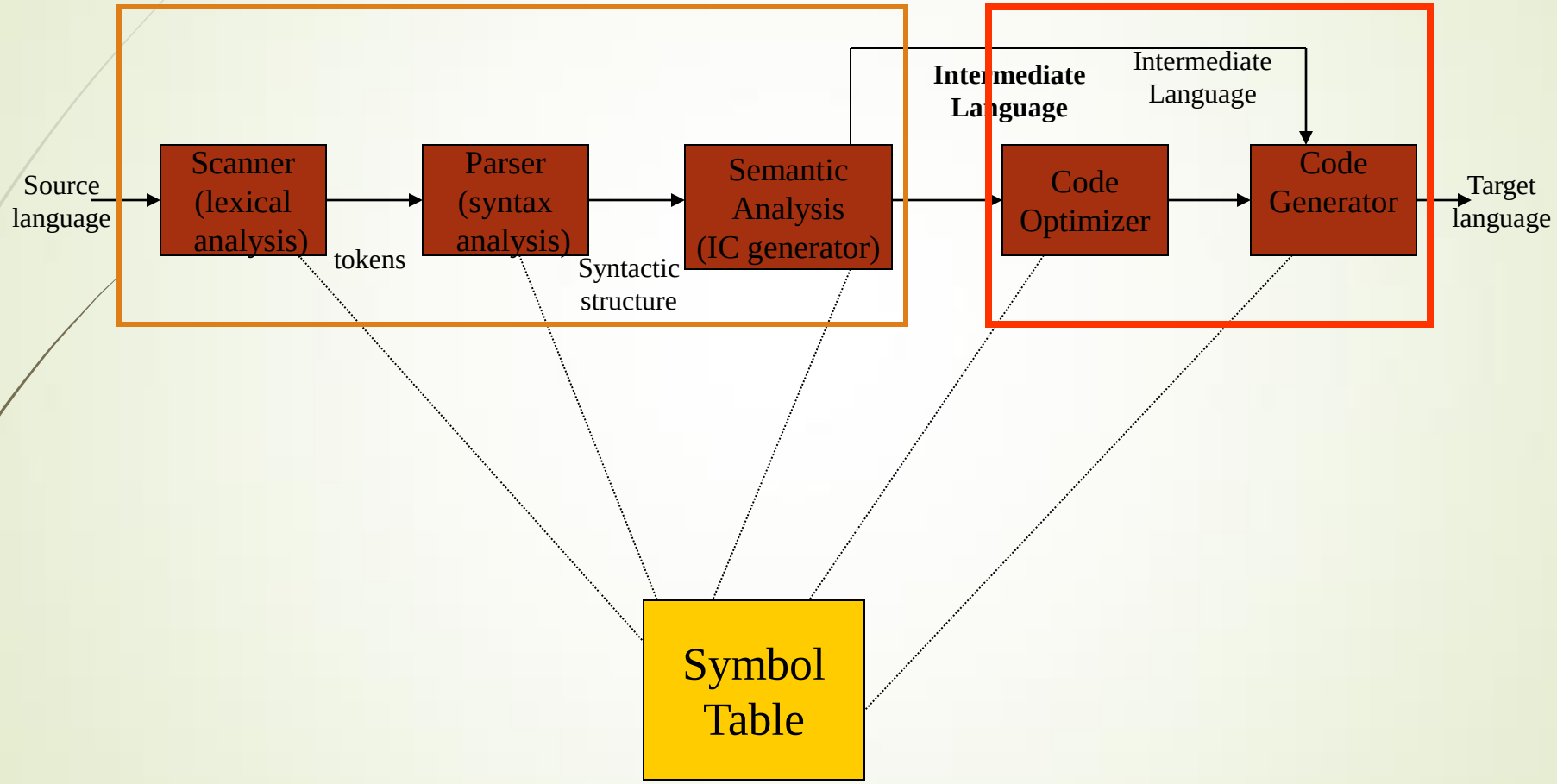


Compiler Architecture

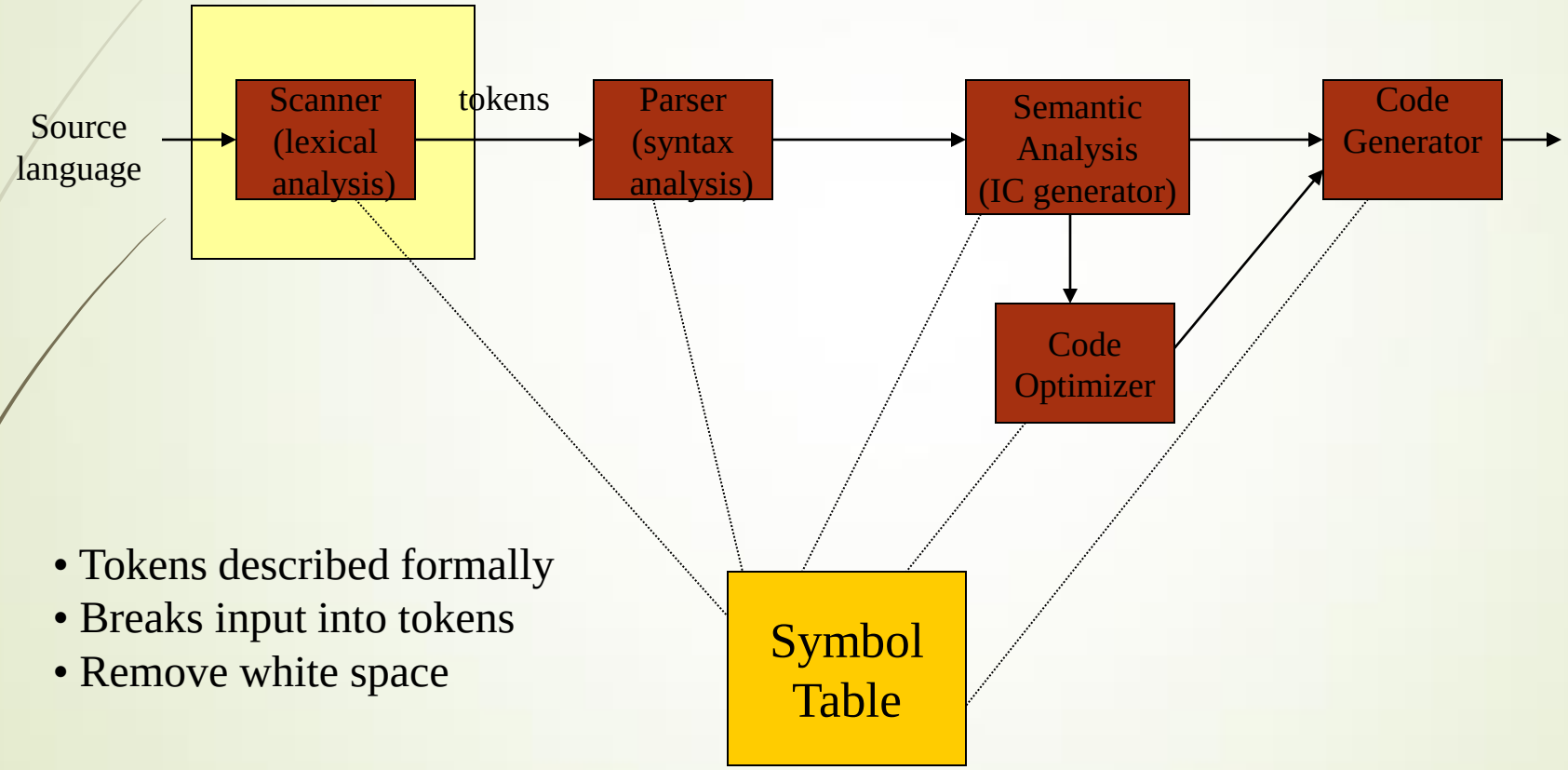
In more detail:



Compiler Architecture



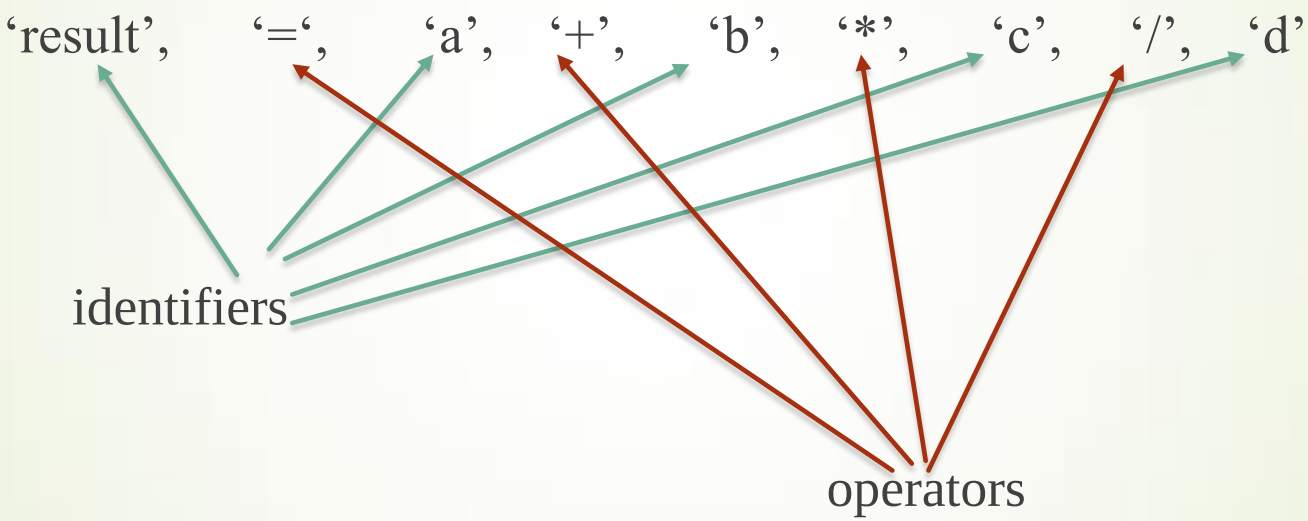
Lexical Analysis - Scanning



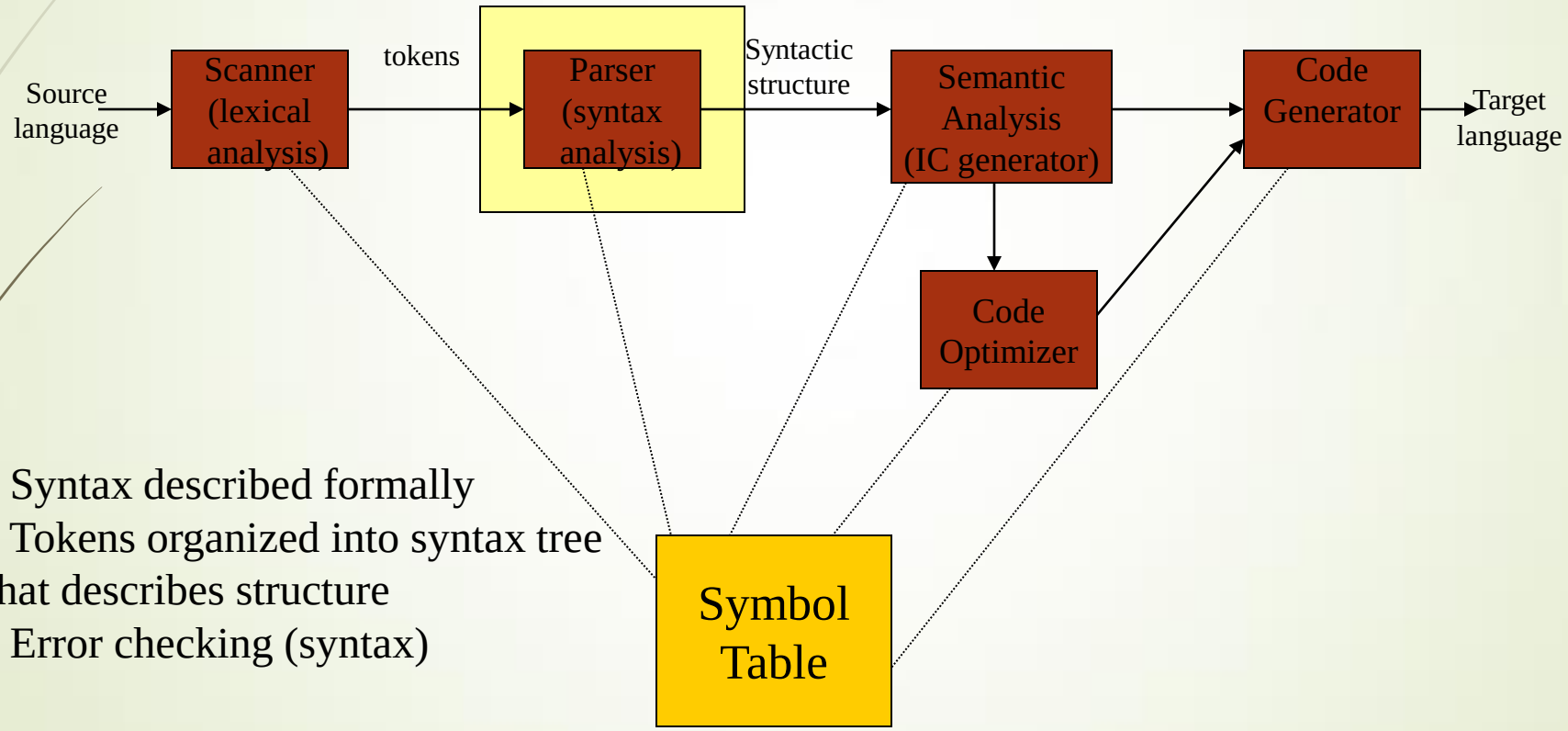
- Tokens described formally
- Breaks input into tokens
- Remove white space

Example:
Input: result = a + b * c / d

- Tokens:



Static Analysis - Parsing

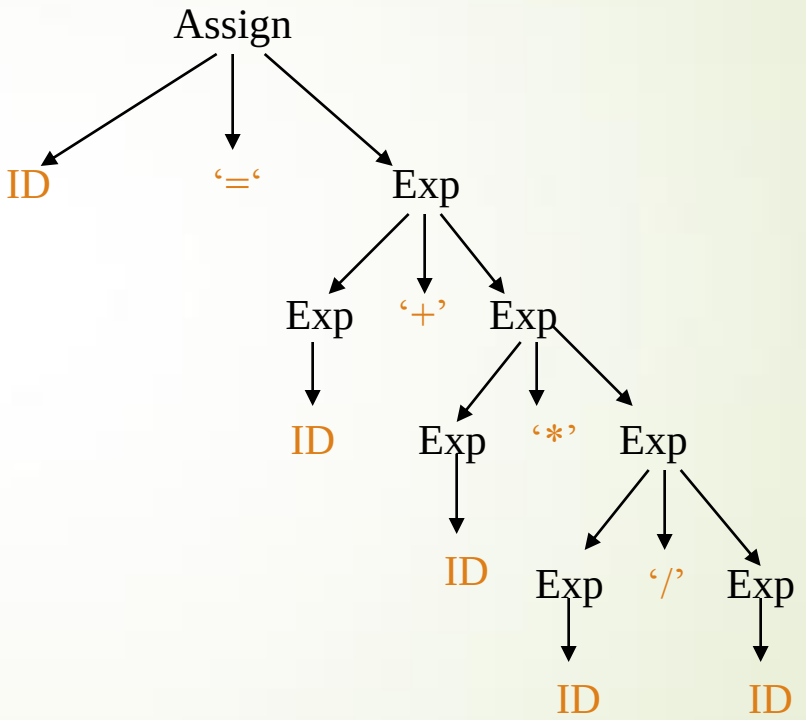


- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

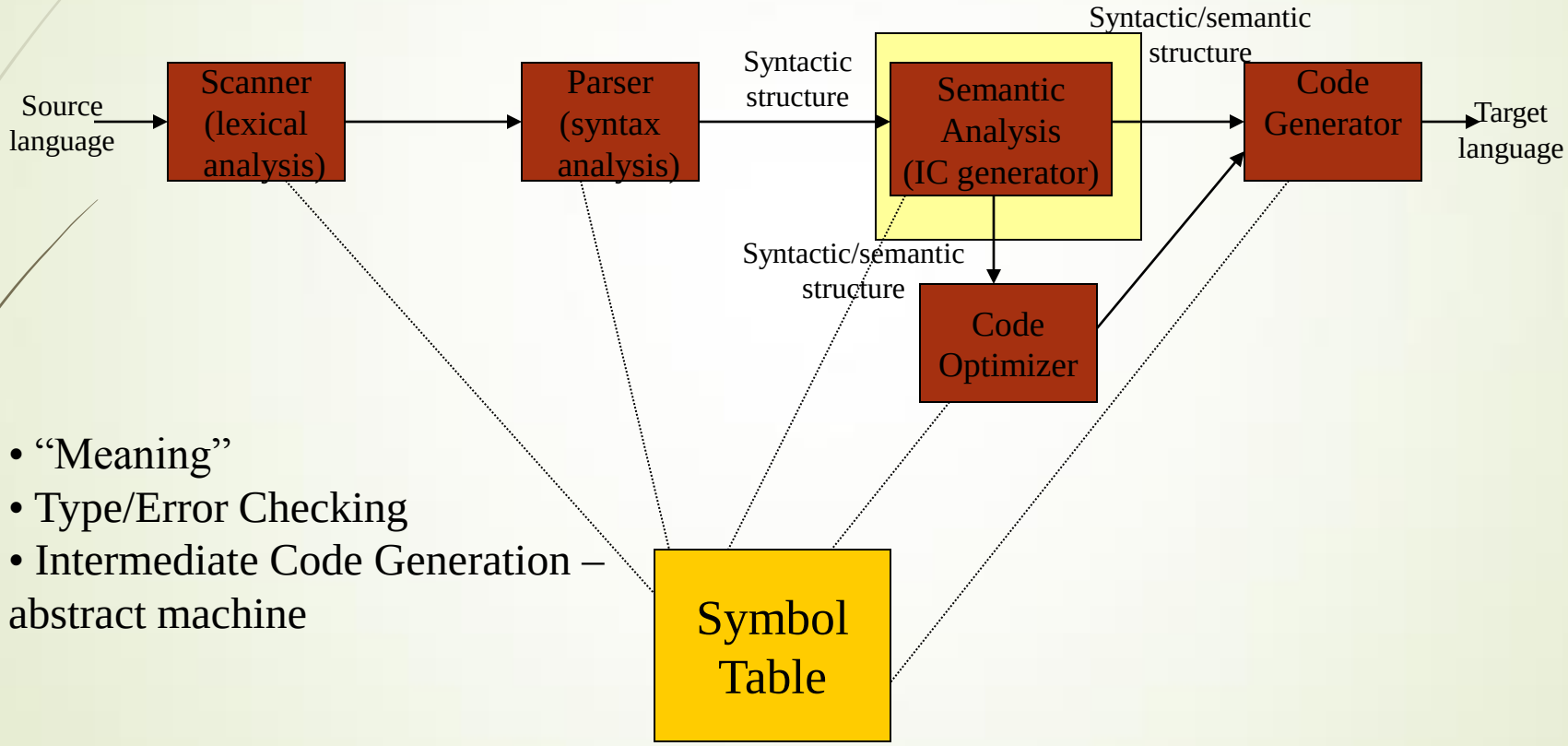
Input: result = a + b * c / d

Exp ::= Exp ‘+’ Exp
| Exp ‘+’ Exp
| Exp ‘*’ Exp
| Exp ‘/’ Exp
| ID

Assign ::= ID ‘=‘ Exp

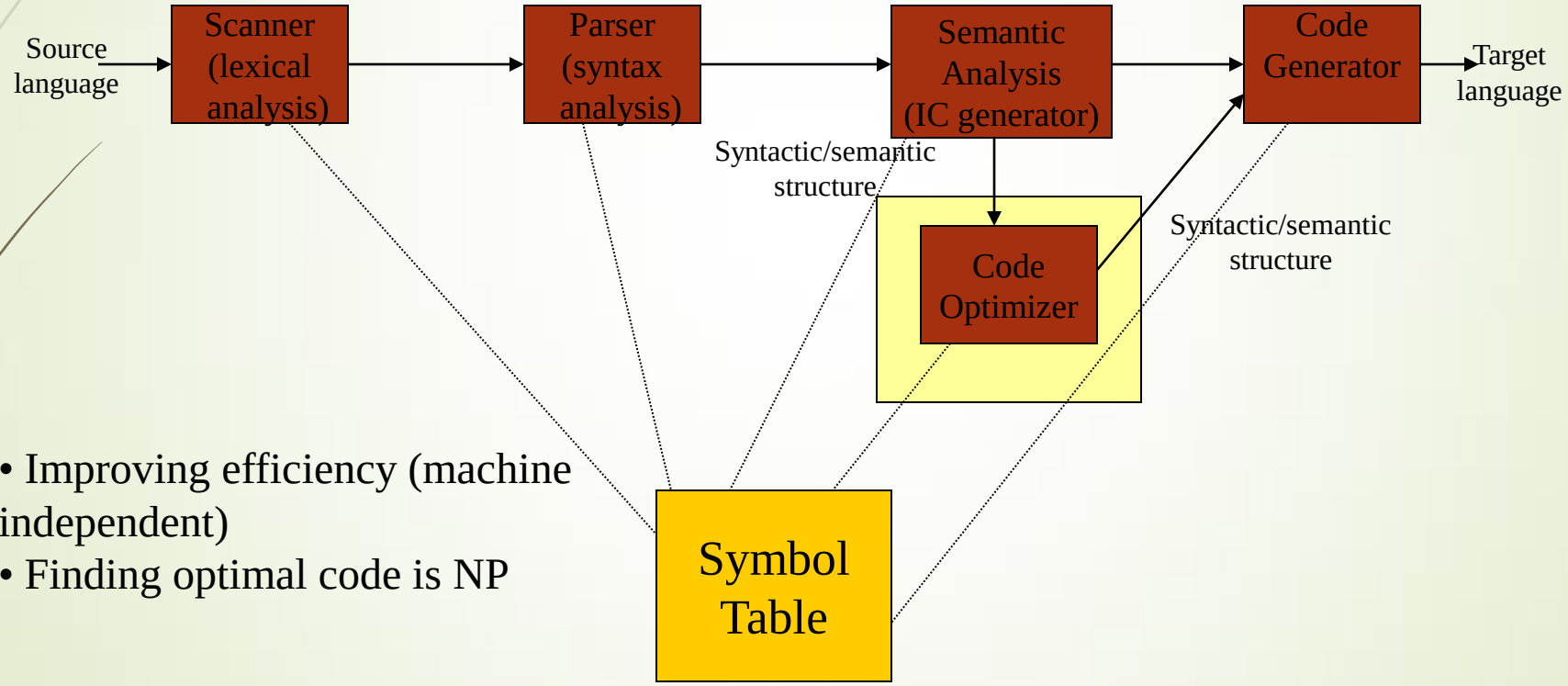


Semantic Analysis



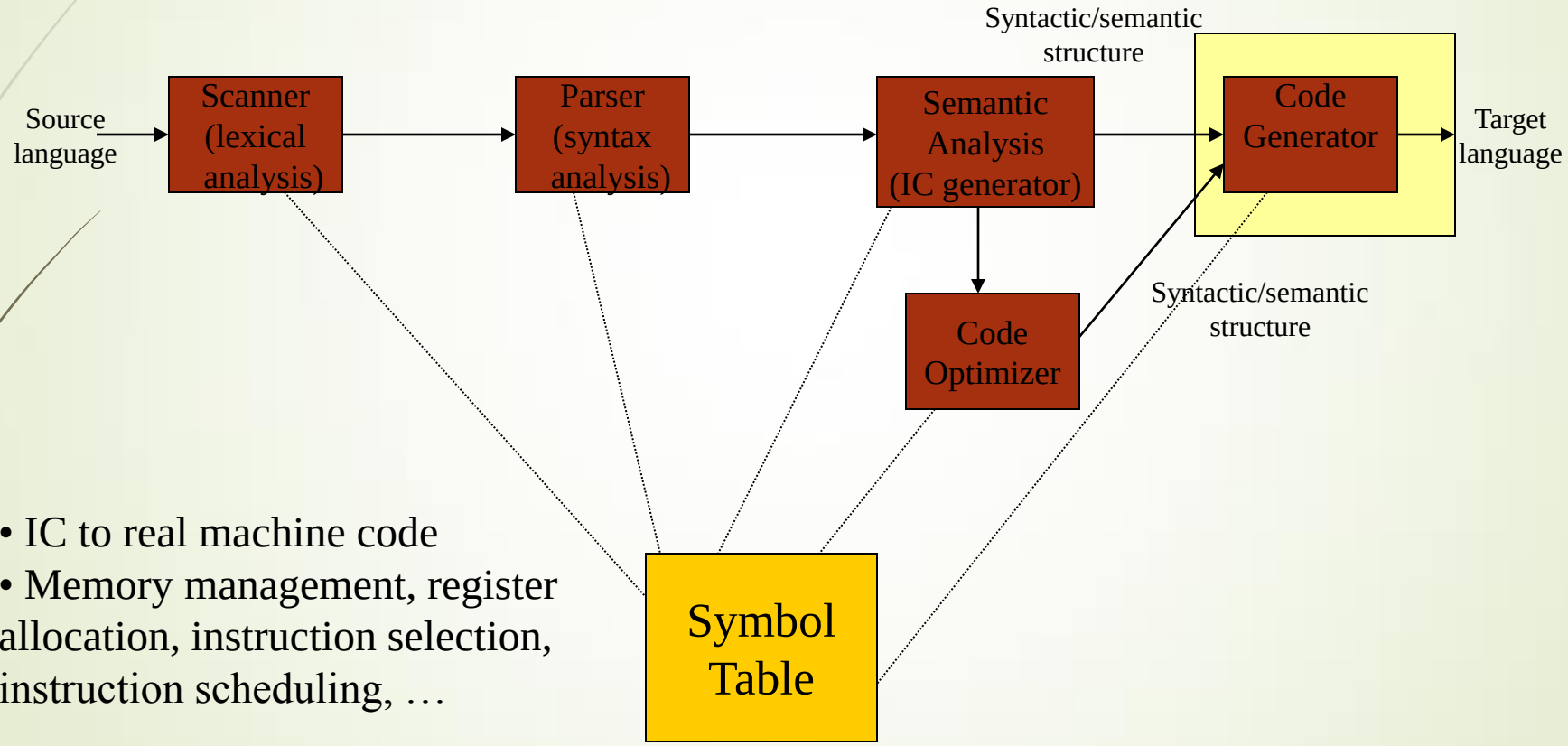
- "Meaning"
- Type/Error Checking
- Intermediate Code Generation – abstract machine

Optimization



- Improving efficiency (machine independent)
- Finding optimal code is NP

Code Generation

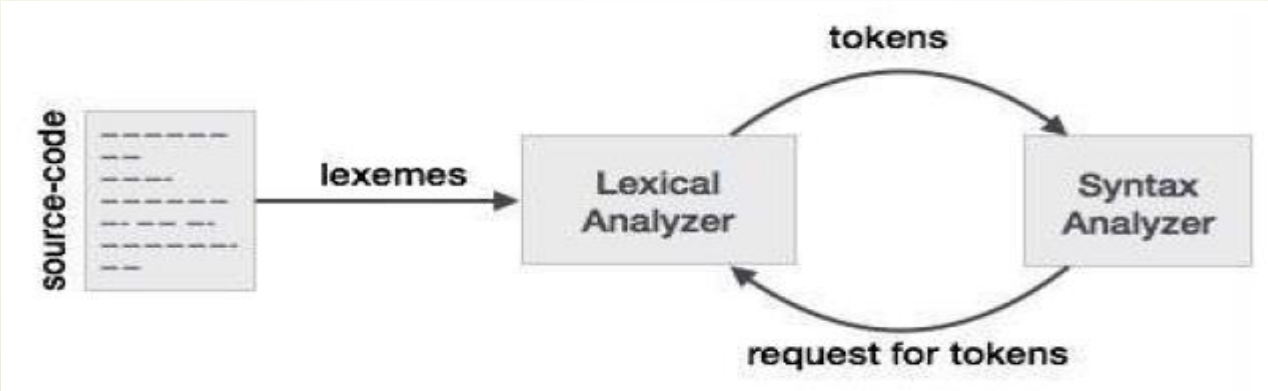


- IC to real machine code
- Memory management, register allocation, instruction selection, instruction scheduling, …

Contents

Role of Lexical Analyzer

- Takes the modified source code from language preprocessors
- Breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code
- Generates an error for invalid token
- Reads character stream checks for legal token & passes it to syntax analyzer.



Contents

- Introduction to Compiler
  - Overview to Compilers
  - Phases of a Compiler
  - Grouping of Phases
- Role of Lexical Analyzer
  - Speciation of tokens
  - Recognition of tokens
- Finite Automata:
  - NFA and DFA
  - Constructing NFA from a Regular expression
  - Constructing DFA from Regular expression
  - Minimizing number of states

Specification of Token

- Lexeme: a sequence of characters alphanumeric in a token
- Grammar rules: predefined rules for every lexeme to be identified as a valid token by a means of pattern
- **Pattern**: defined by regular expressions and explains what can be a token
- In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens
  - E.g. in C language (int value = 100;)
    - Contains the tokens:
      - int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Specification of Token

- Let us understand how the language theory undertakes the following terms:
- Alphabets:

Binary alphabet {0,1}	Hexadecimal alphabet {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}	English alphabets {a-z, A-Z}

- Strings: Any finite sequence of alphabets.
  - A string of zero length is known as an empty string and is denoted by ε epsilon
- Special symbols: typical HLL contains ff. symbols:

Arithmetic Symbols	**Addition + , Subtraction − , Modulo, Multiplication ∗ , Division/**
Punctuation	**Comma, , Semicolon; , Dot. , Arrow− >**
Assignment	**=**
Special Assignment	**+=, /=, *=, -=**
Comparison	**==, !=, <, <=, >, >=**
Preprocessor	**#**
Location Specifier	**&**
Logical	**&, &&, |, ||, !**
Shift Operator	**>>, >>>, <<, <<<**

Contents

Recognition of Tokens

- Language: is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them.
- Finite languages can be described by means of regular expressions.
- Longest Match Rule: When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed
- The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.
- The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error
- 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

Contents

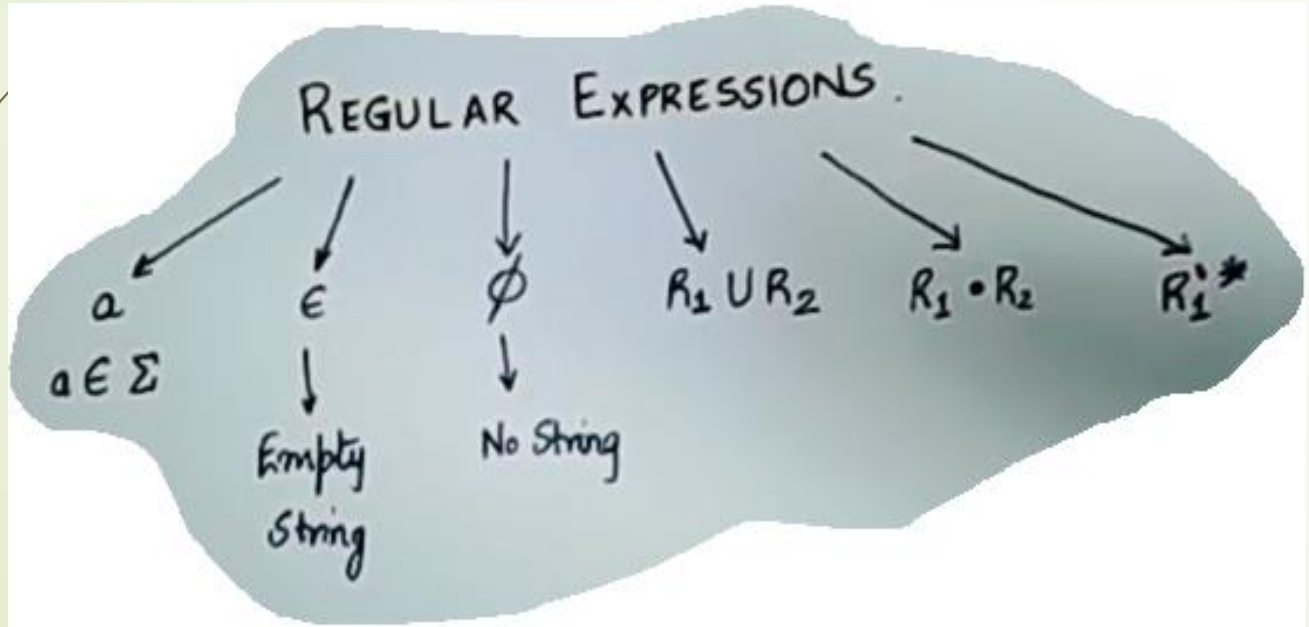
Regular Expression

- Tokens are built from symbols of a finite vocabulary
- We use regular expressions to define structures of tokens

Component of regular expression	
x	the character x
.	any character, usually accept a new line
[x y z]	any of the characters x, y, z, …..
R?	a R or nothing (=optionally as R)
R*	zero or more occurrences…..
R+	one or more occurrences ……
R1R2	an R1 followed by an R2
R1|R1	either an R1 or an R2.

Regular Expression

- A formula that describes a possible set of string



Regular Expression

- L(001) = {001}
- L(0+10*) = { 0, 1, 10, 100, 1000, 10000, … }
- L(0*10*) = {1, 01, 10, 010, 0010, …} i.e. {w | w has exactly
- a single 1}
- L(□ □)* = {w | w is a string of even length}
- L((0(0+1))*) = { ε, 00, 01, 0000, 0001, 0100, 0101, …}
- L((0+ε)(1+ ε)) = {ε, 0, 1, 01}
- L(1Ø) = Ø ; concatenating the empty set to any set yields the empty set.
- Rε = R
- R+Ø = R

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
- Finite Automata is a Recognizer of tokens

Finite Automata

- The token that gets returned to the parser

Lexeme	Token	Attribute value
Any WS (blank/tab/newline)	-	-
if	If	-
then	Then	-
else	else	-
Any id	id	Pointer to table entry
Any number	Number	Pointer to table entry
<, <=, ==, <>	Operator	LT, LE,EQ, NE respectively

Contents

- Introduction to Compiler
  - Overview to Compilers
  - Phases of a Compiler
  - Grouping of Phases
- Role of Lexical Analyzer
  - Speciation of tokens
  - Recognition of tokens
- Finite Automata:
  - NFA and DFA
  - Constructing NFA from a Regular expression
  - Constructing DFA from Regular expression
  - Minimizing number of states

NFA & DFA

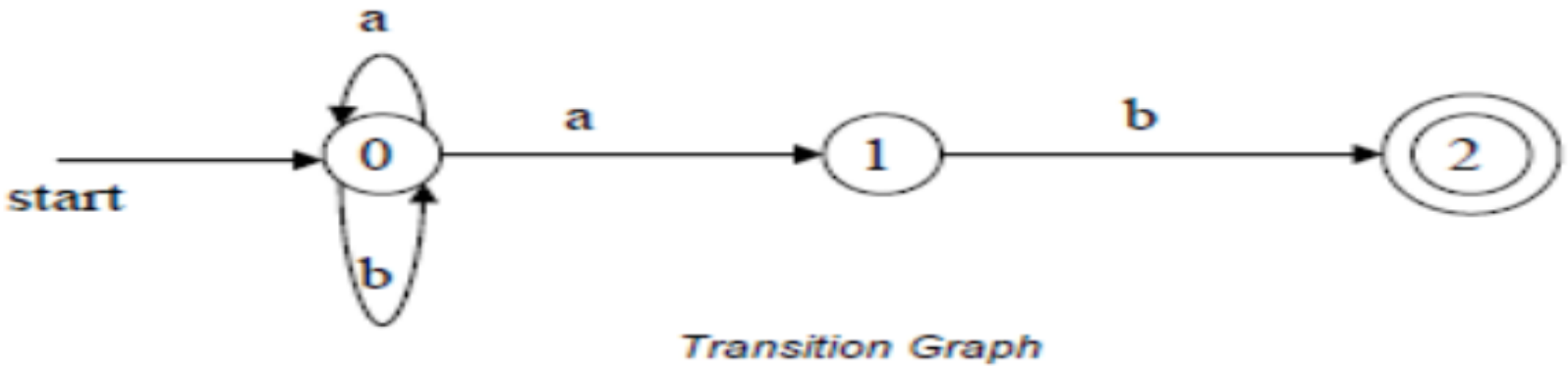
- A finite automaton can be: deterministic (DFA) or non-deterministic (NFA)
  - Can use deterministic or non-deterministic automaton as a lexical analyzer
  - Both recognize regular sets
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but 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.

Non-Deterministic Finite Automata (NFA)

- 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.
  - s0 - a start (initial) state
  - F- a set of accepting states (final states)
- ε- transitions are allowed in NFAs
  - It is possible to 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.

Non-Deterministic Finite Automata (NFA)

- Example: language to be recognized is **(a|b)*ab**



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

s\Σ	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

The language recognized by this NFA is (a|b)*ab

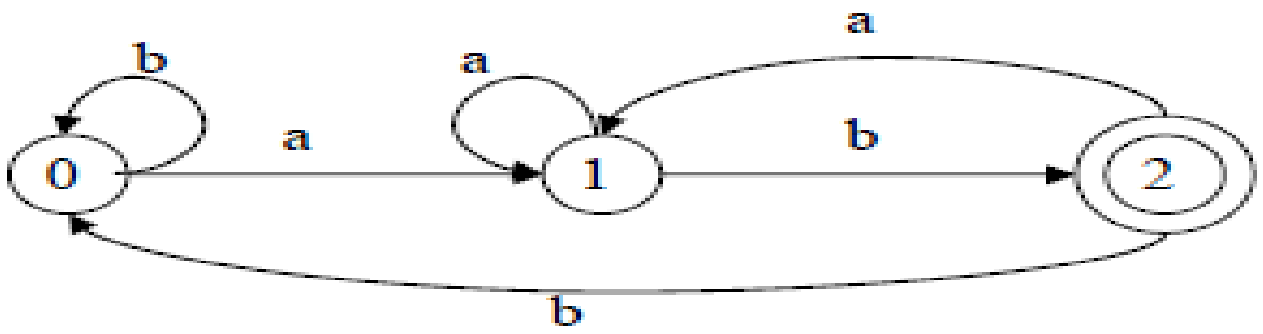
Deterministic Finite Automata (DFA)

- DFA is a special form of a NFA
- No state has ε- transition
- For each symbol **a** and state **s**, there is at most one labeled edge a leaving **s**
  - i.e. transition function is from pair of state-symbol to state (not set of states)

Deterministic Finite Automata (NFA)

Example:

The DFA to recognize the language (a|b)* ab is as follows.



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

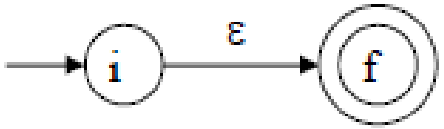
s\Σ	a	b
0	1	0
1	1	2
2	1	0

Note that the entries in this function are single value and not set of values (unlike NFA).

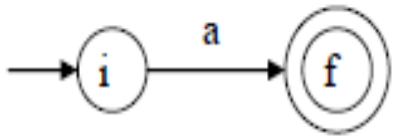
Contents

Constructing NFA from a Regular expression

- Converting RE to NFA
  - 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
- To recognize an empty string ε:

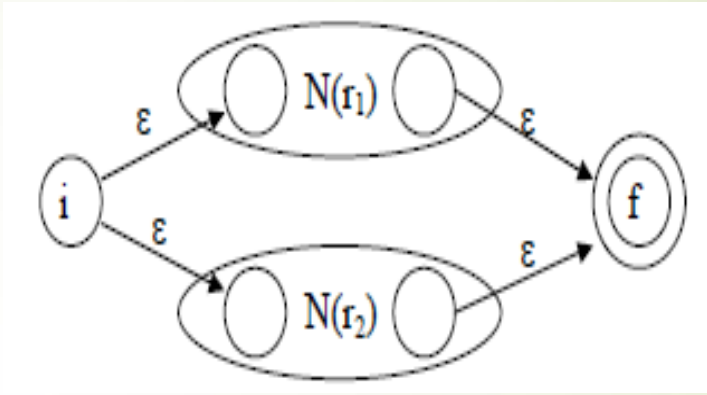


- To recognize a symbol a in the alphabet Σ:



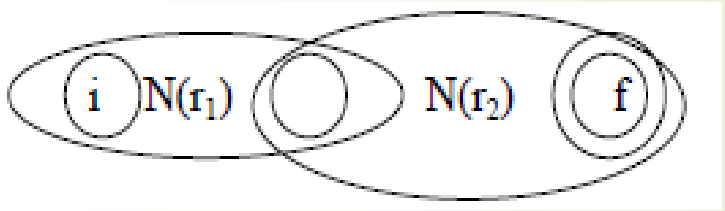
Constructing NFA from a Regular expression

- Combining sub expressions:
  - For regular expression r1 | r2:



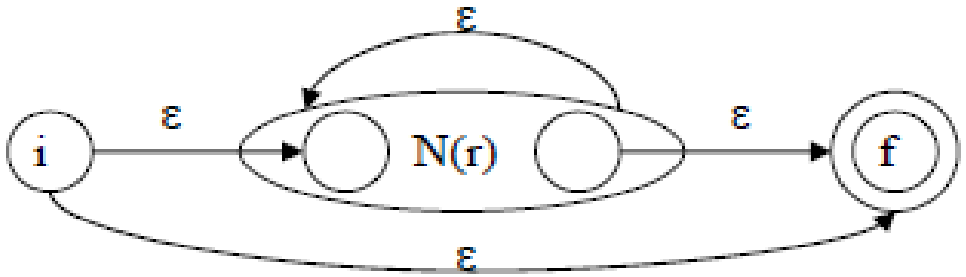
N(r1) and N(r2) are NFAs for regular expressions r1 and r2.

  - For regular expression r1 r2:



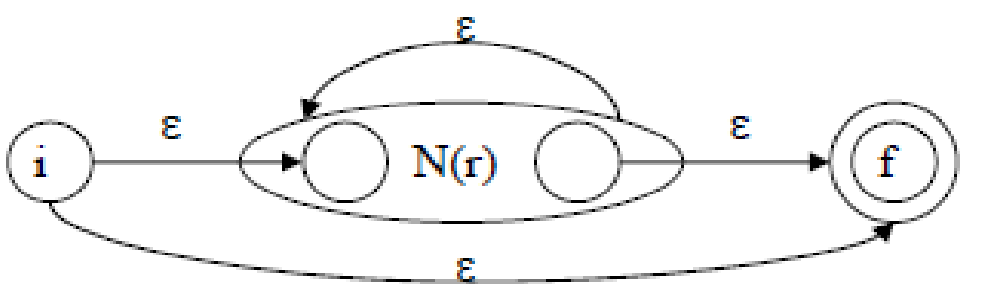
Here, final state of N(r1) becomes the final state of N(r1r2).

  - For regular expression r*:

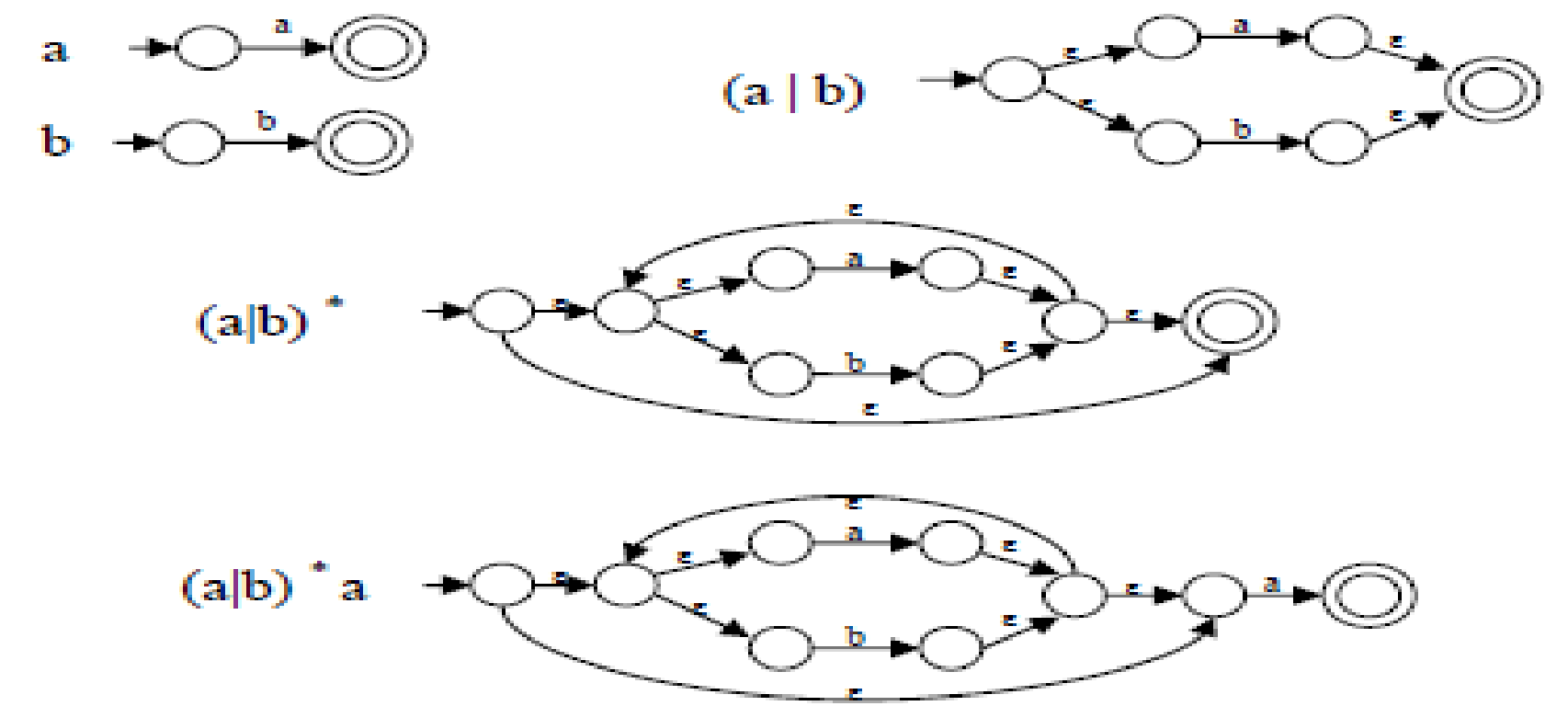


Constructing NFA from a Regular expression

- Combining sub expressions:
  - For regular expression r*:



  - Example: For a RE (a|b) * a, the NFA construction is shown below



Contents

Constructing DFA from a Regular expression

- A DFA is defined by the 5-tuple:{Q Σ q F δ }
- A DFA consists of:
  - Q ==> a finite set of states
  - Σ ==> a finite set of input symbols (alphabet)
  - q0 ==> a start state
  - F ==> set of final states
  - δ ==> a transition function, which is a mapping between Q x Σ ==> Q

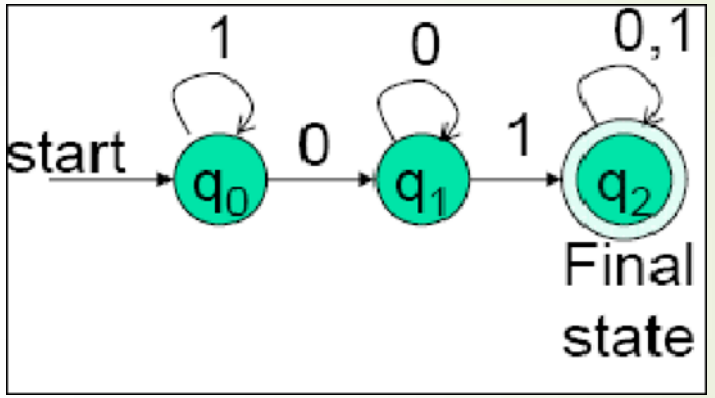
Constructing DFA from a Regular expression

Example:

- Build a DFA for the following language:
- L = {w | w is a binary string that contains 01 as a substring}.
- Steps for building a DFA to recognize L:
  - $\Sigma = \{0,1\}$
  - Decide on the states: Q
  - Designate start state and final state(s)
  - δ: Decide on the transitions:

Constructing DFA from a Regular expression

- Regular expression: (0+1)*01(0+1)*
  - Q= {q0, q1, q2}
  - Σ={0,1}
  - start state = q0
  - F = {q2}
  - DFA



  - Transition table:

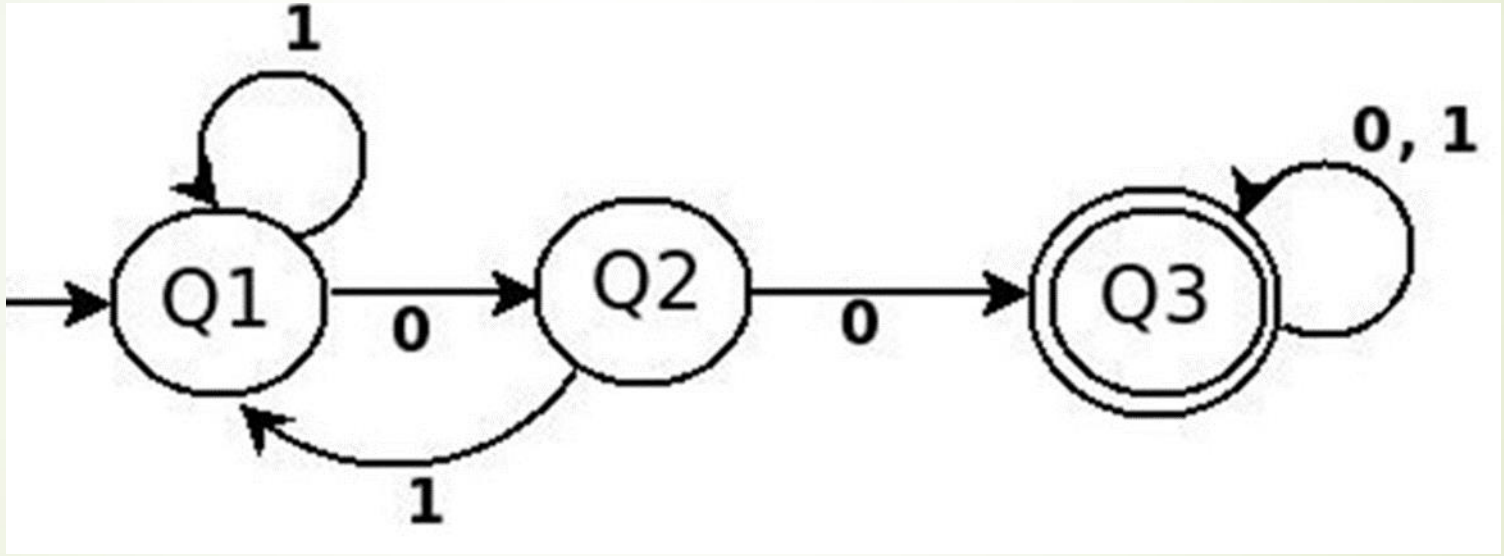
	symbols	
δ	0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

states

Constructing DFA from a Regular expression

Example:

- (0+1)*00(0+1)*



Contents

- Introduction to Compiler
  - Overview to Compilers
  - Phases of a Compiler
  - Grouping of Phases
- Role of Lexical Analyzer
  - Speciation of tokens
  - Recognition of tokens
- Finite Automata:
  - NFA and DFA
  - Constructing NFA from a Regular expression
  - Constructing DFA from Regular expression
  - Minimizing number of states

Reading assignment

Practical session!!!

Lexical analyzer in C

- C program to detect tokens in a C program
  - Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens
  - A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol
  - Example

1) Keywords: for, while, if etc. **2) Identifier: Variable name, function name etc.**	**3) Operators: '+', '++', '-' etc.** **4) Separators: ', ' ';' etc**

  - So a C program to print all the keywords, literals, valid identifiers, invalid identifiers, integer number, real number in a given C program: