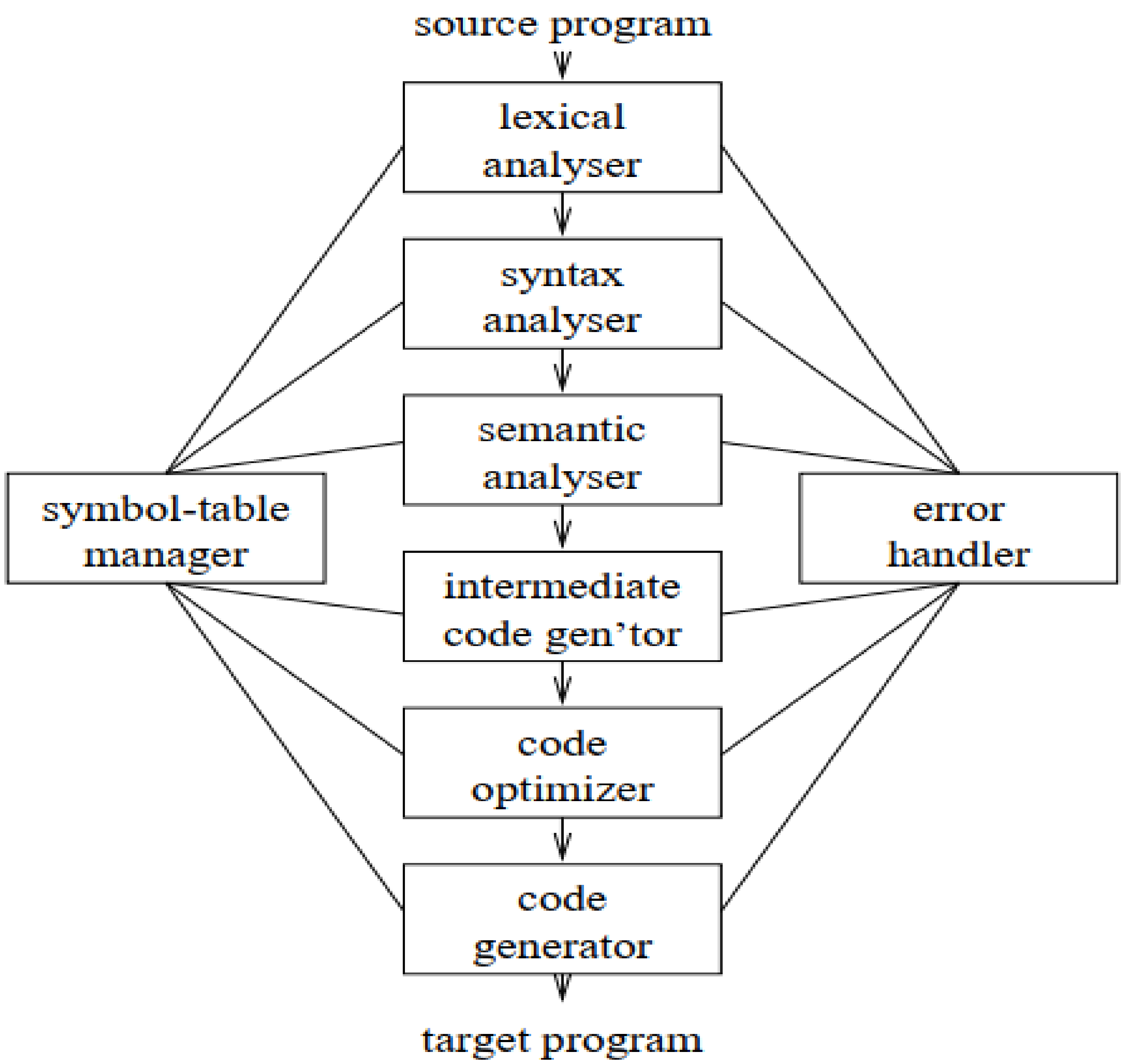
Compiler Design (CoSc3112)

UNIT-2: LECICAL ANALYSIS

2014

Structure or Phases of Compiler



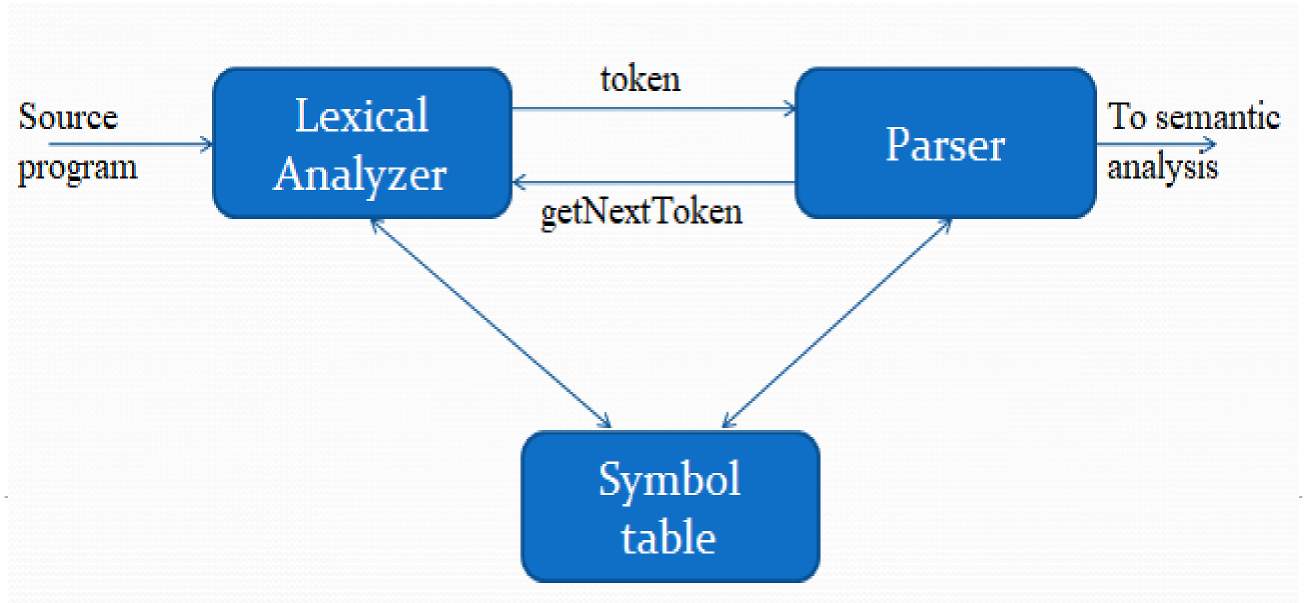
Introduction Lexical analysis

- Lexical analysis is the first phase of a compiler.
- It takes the modified source code from language preprocessors that are written in the form of sentences.
- It breaks the sentences into a series of tokens, by removing any whitespace or comments in the source code.
- If the lexical analyzer finds a **token** as **invalid**, it generates an error.
- The lexical analyzer works closely with the syntax analyzer.

Role of Lexical Analyzer

- The main task of the lexical analyzer is
  - To read the *input characters of the source program*, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis.
  - It is common for the lexical analyzer to interact with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

Cont…

- Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis.
- As in the figure below, upon receiving a “getNextToken” command from the parser the lexical analyzer reads input characters until it can identify the next token.
- 

Cont…

- When lexical analyzer identifies the first token, it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens.
- During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

Separating Lexical Analyzer from parsing

- There are several reasons for separating the analysis phases of the compiling into lexical analysis and parsing:
  1. Simplicity of design
  2. Improved compiler efficiency
  3. Higher portability

TOKENS, PATTERNS AND LEXEMES:

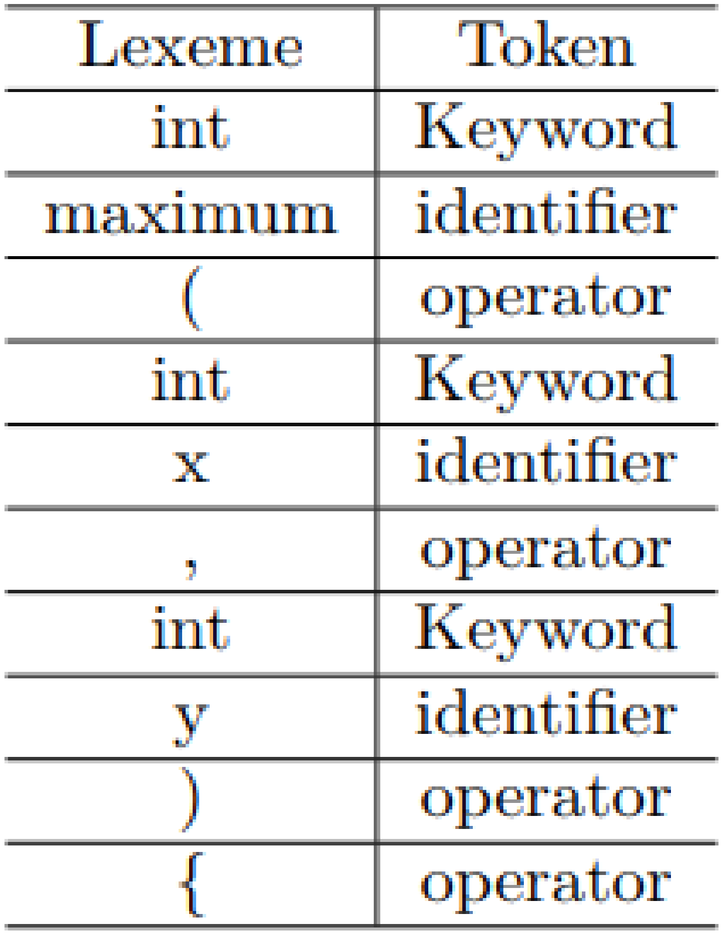
- **Token:**
  - A token is a group of characters having collective meaning.
  - Tokens may be Identifiers, keywords, constants, operators, special symbols, punctuations etc.
- **Lexeme**:
  - A lexeme is an actual character sequence forming a specific instance of a token, such as num.
- **Pattern:**
  - A pattern is a rule expressed as a regular expression and describing how a particular token can be formed.
  - It is the rule describing the lexemes.
  - For example, [A-Za-z][A-Za-z_09] is a rule.

Example of Tokens and Non-Tokens

- Consider the following code that is fed to lexical analyzer and identify the tokens.

-
  ```
  #include <stdio.h>
  int maximum(int x, int y)
  //This program compares two integers
  if (x > y)
       return x;
  else
        return y;
  ```

Example of Tokens



Lexeme	Token
int	Keyword
maximum	identifier
(	operator
int	Keyword
x	identifier
,	operator
int	Keyword
y	identifier
)	operator
{	operator

Example of Non-Tokens

Type	Example
Comment	//This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
White space	/n /b /t

Example

- For example, in C language, the variable declaration line
  int value = 100;
- Contains the tokens:
- **Lexem Tokens**
  1) int --------(keyword)
  2) value ----(identifier)
  3) = ----------(operator)
  4) 100 -------(constant)
  5) ; -----------(symbol)

Lexical Errors

- **Lexical errors** are the errors thrown by the **lexer** when unable to continue. There is no way to recognize a lexeme as a valid token for your lexer.
- A sequence of character which is not possible to scan into any valid token is a lexical error.
- Lexical errors are not very common, but should be managed by a scanner.
- Misspelling of identifiers, operators, keyword are considered as lexical errors.
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Cont…

- The common methods used to recover from lexical error are:
  1. **Delete one character from the remaining input.**
  2. **Insert a missing character in to the remaining input.**
  3. **Replace a character by another character.**
  4. **Transpose two adjacent characters.**

Token specification

- Regular expression is a set of pattern that matches a character or string.
- Regular expressions are an important notation for specifying lexeme patterns.
- Regular expressions cannot express all possible patterns, but they are effective in specifying patterns needed for tokens.

Regular Expressions

- Each regular expression r denotes a language L(r).
- Definition of regular expressions over some alphabet Σ and the languages that those expressions denote:
  1. ε is a regular expression, and L(ε) is { ε }.
  2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}.
  3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,
     a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
     b) (r)(s) is a regular expression denoting the language L(r)L(s).

Cont…

a) (r)* is a regular expression denoting (L(r))*.
b) (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

Recognition of tokens

- Tokens can be recognized by Finite Automata.
- The lexical analyzer uses DFA to determine if the regular expression is valid or not.
- If the regular expression is valid the lexer generates token.
- A Finite automaton is a simple idealized machine used to recognize patterns within input taken from some character set(or Alphabet) C.
- The job of FA is to accept or reject an input depending on whether the pattern defined by the FA occurs in the input.

Formalization of patterns

- A set of strings described by a rule associated with the token is called a pattern.
- Examples of patterns:

  $*$ digit $\rightarrow$ [0-9]

  $*$ lower case letters $\rightarrow$ [a-z]

  $*$ upper case letters $\rightarrow$ [A-Z]

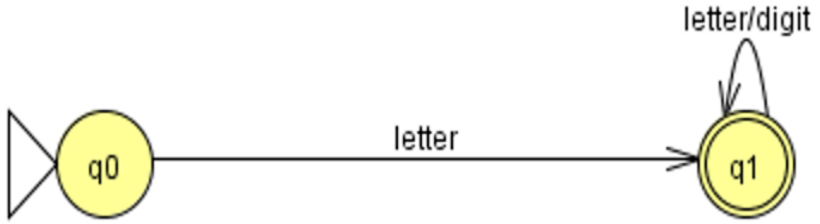
  $*$ If $\rightarrow$ if

  $*$ Else $\rightarrow$ else

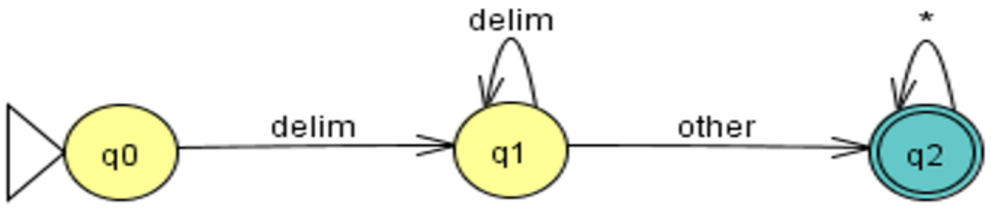
  $*$ Relop $\rightarrow$ < | > | <= | >= | = | <>

Cont…

- Recognizing reserved words and white spaces

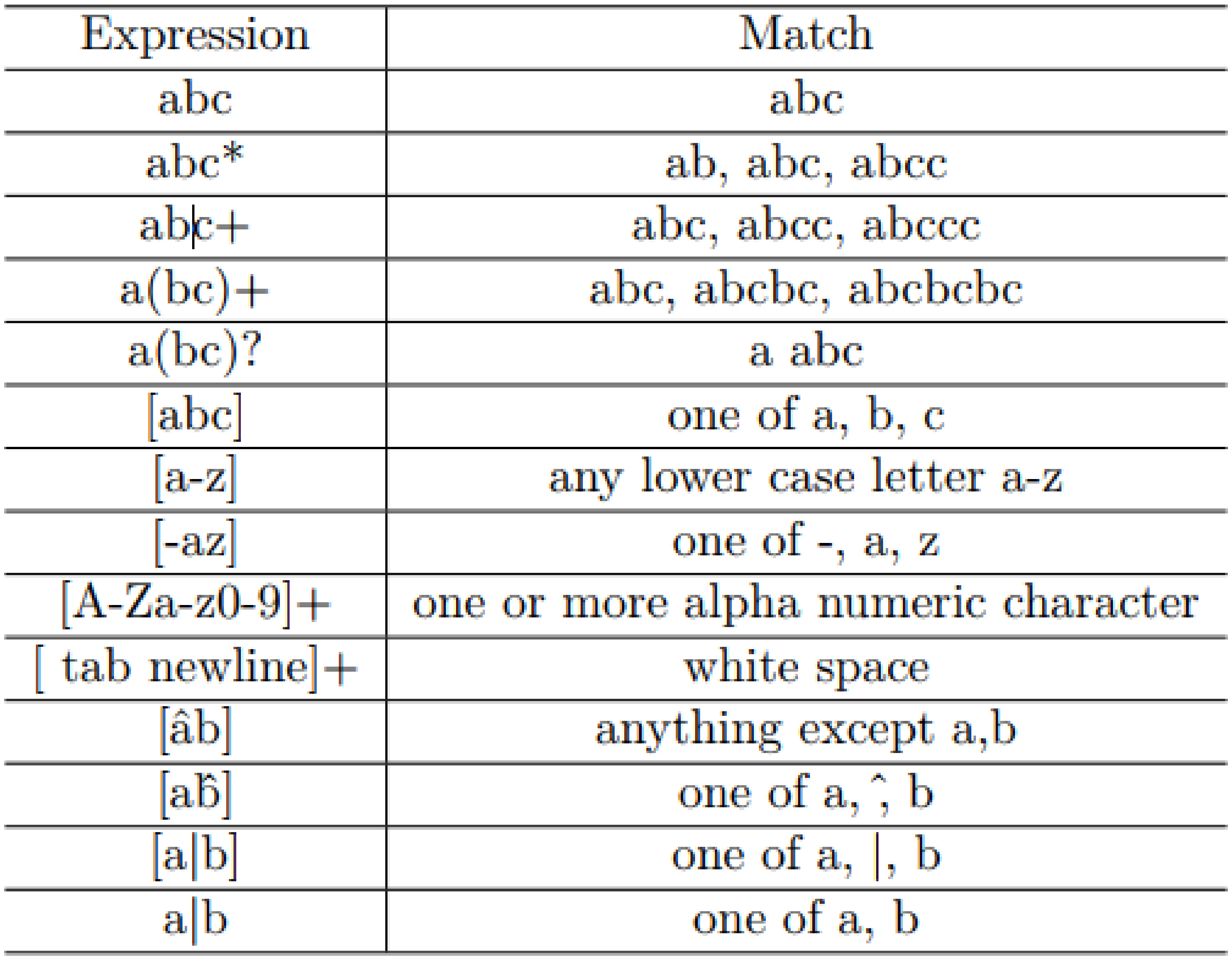


- Figure 2.3: Transition diagram for recognizing reserved words and identifiers



- Figure 2.4: Transition diagram for recognizing white space

Lex specifications using regular expressions

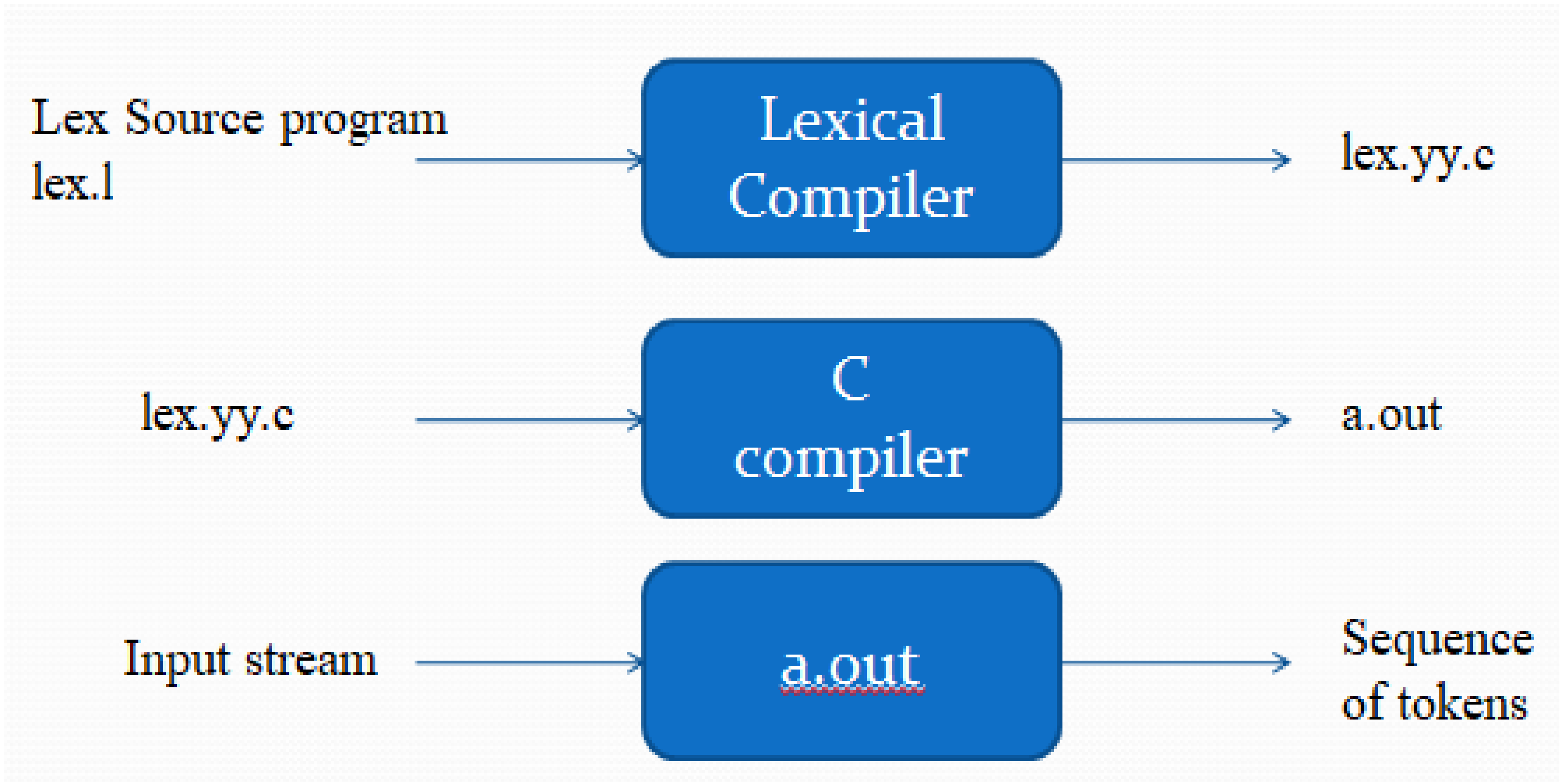


Expression	Match
abc	abc
abc*	ab, abc, abcc
abc+	abc, abcc, abccc
a(bc)+	abc, abcbc, abcbcbc
a(bc)?	a abc
[abc]	one of a, b, c
[a-z]	any lower case letter a-z
[-az]	one of -, a, z
[A-Za-z0-9]+	one or more alpha numeric character
[tab newline]+	white space
[âb]	anything except a,b
[ab̂]	one of a, ˆ, b
[a|b]	one of a, |, b
a|b	one of a, b

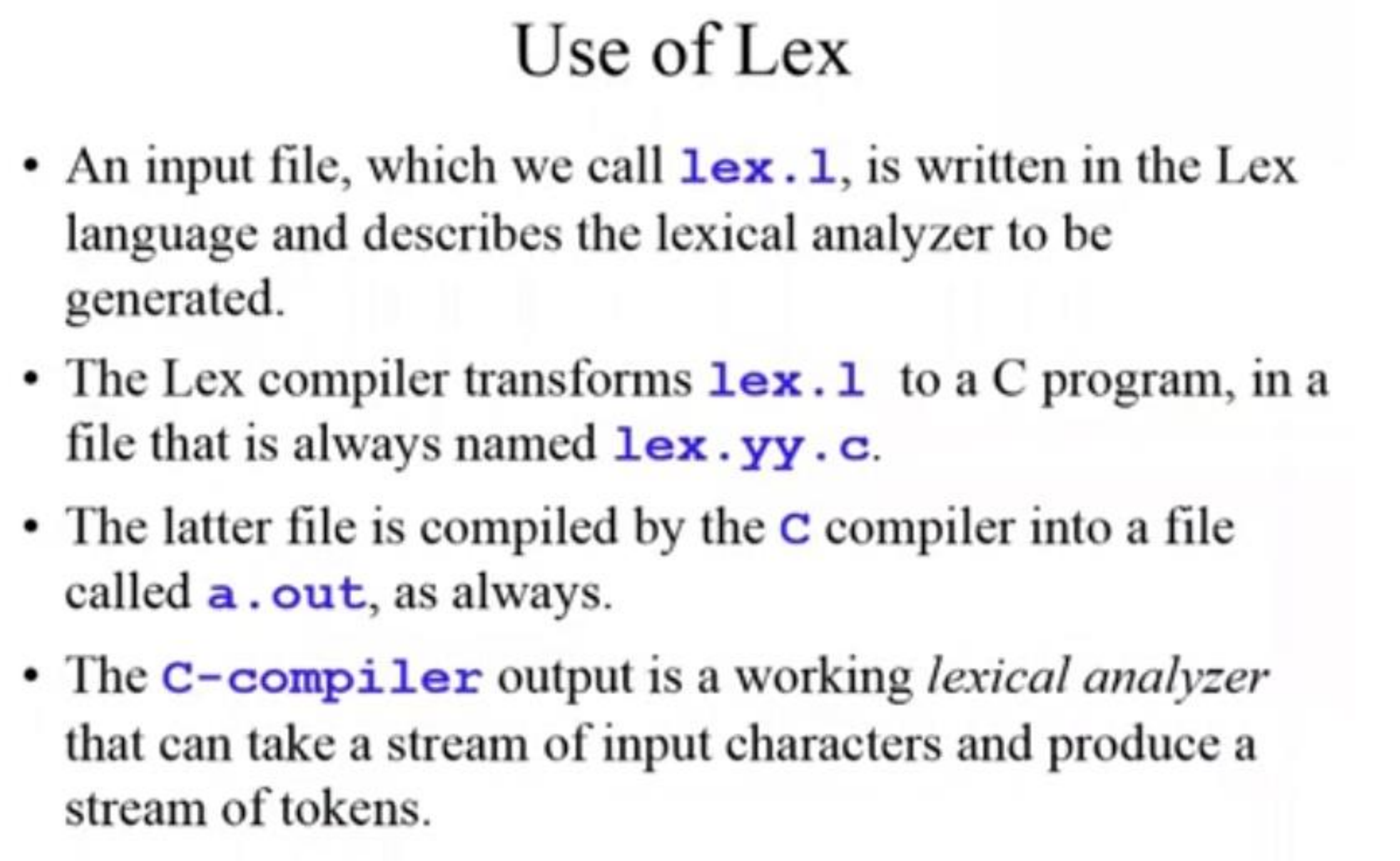
Lexical Analyzer Generator: **Lex**

- **Lex** is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the **Lex language** and the tool itself is the **Lex compiler**.
- The **Lex** compiler transforms the input patterns into a transition diagram and generates code, in a file called **lex .yy .c**,
- **lex .yy .c** is a c program given for C Compiler, gives the **Object code**.

Creating Lexical Analyzer with Lex



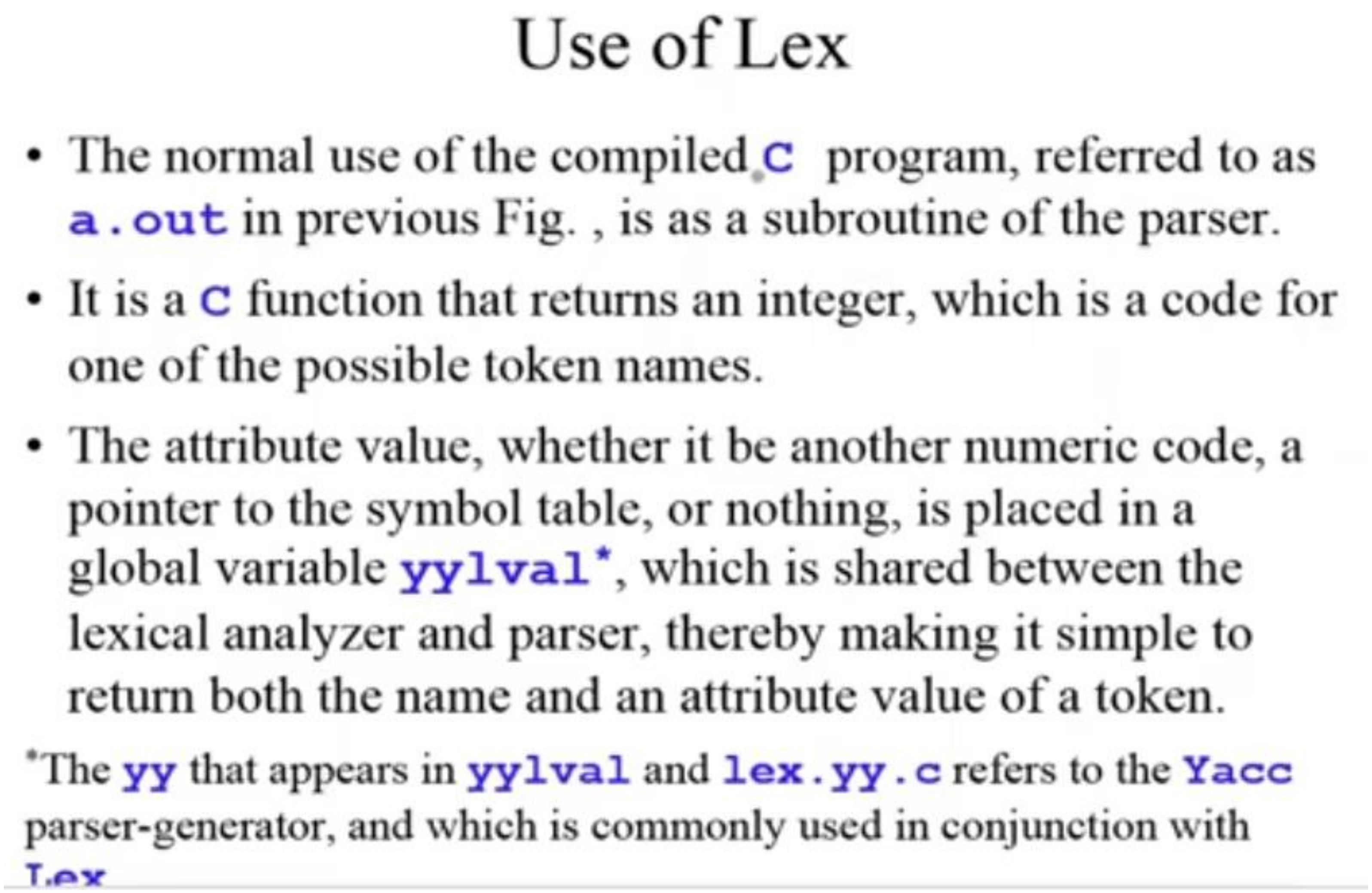
Cont…



Use of Lex

- An input file, which we call `lex.l`, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms `lex.l` 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`, as always.
- The `C-compiler` output is a working *lexical analyzer* that can take a stream of input characters and produce a stream of tokens.

Cont...



Use of Lex

- The normal use of the compiled `C` program, referred to as `a.out` in previous Fig. , is as a subroutine of the parser.
- It is a `C` function that returns an integer, which is a code for one of the possible token names.
- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable `yylval`*, which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

*The `yy` that appears in `yylval` and `lex.yy.c` refers to the `Yacc` parser-generator, and which is commonly used in conjunction with `Lex`

Structure of LEX Program

- A Lex program consists of three parts:

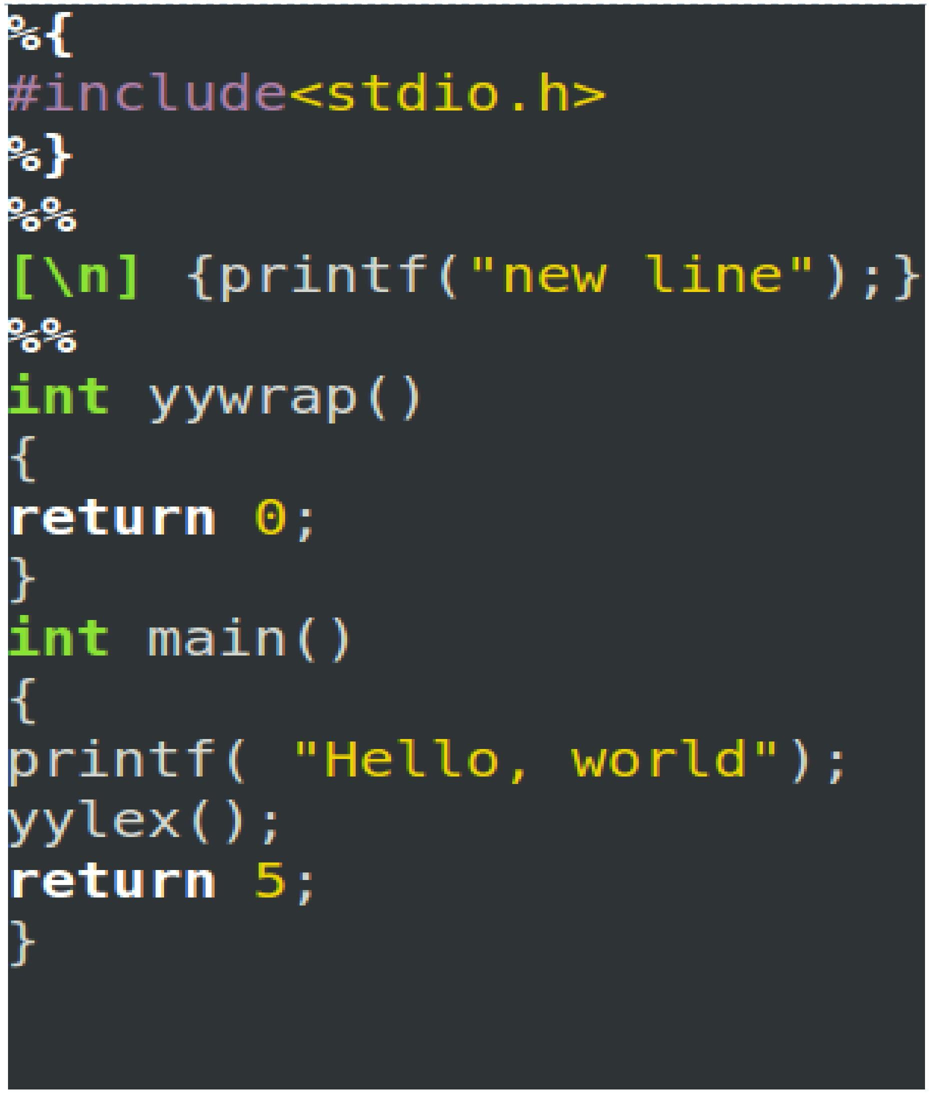
```
declarations
 %%
translation rules
 %%
auxiliary functions definitions
```

Cont…

- **The declarations section:**
  - It includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.
  - It appears between **%{…%}**
- **Translation rules section**:
  - We place Pattern Action pairs where each pair have the form

    Pattern   {Action}
- **The auxiliary function:**
  - It includes the definitions of functions used to install identifiers and numbers in the Symbol table.

Example:



```
%{
#include<stdio.h>
%}
%%
[\n] {printf("new line");}
%%
int yywrap()
{
return 0;
}
int main()
{
printf( "Hello, world");
yylex();
return 5;
}
```

End of chapter