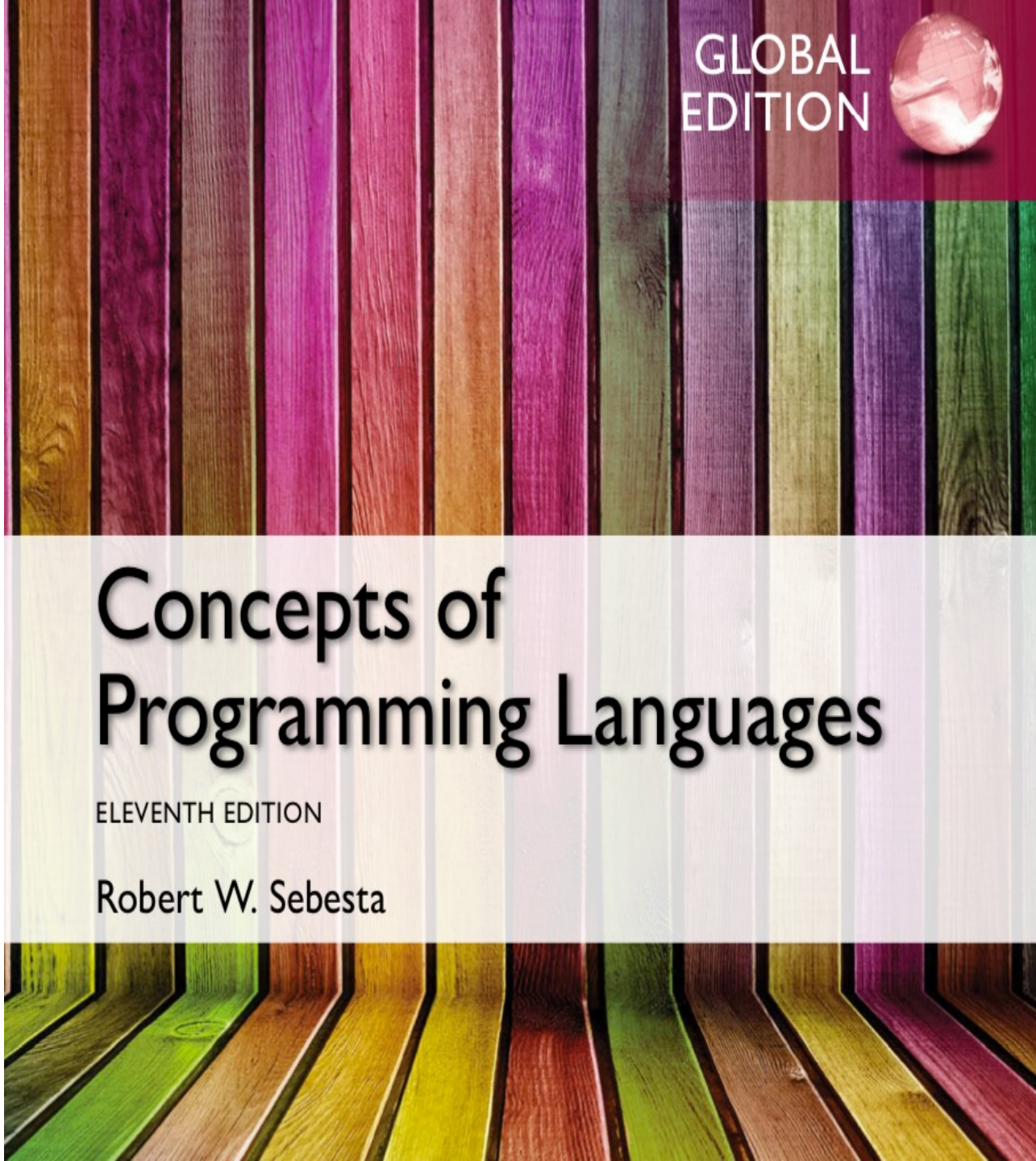
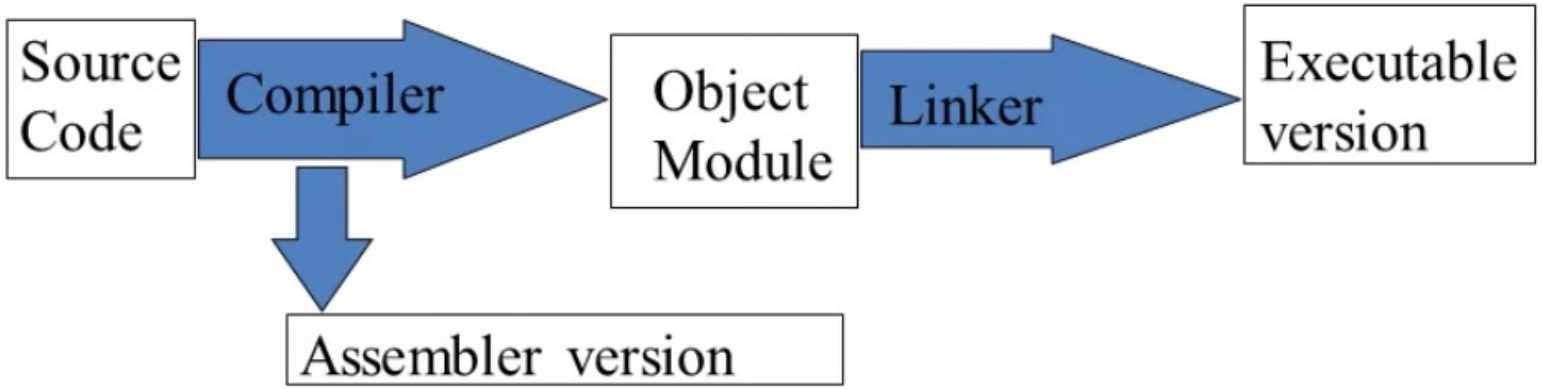
# Chapter 4

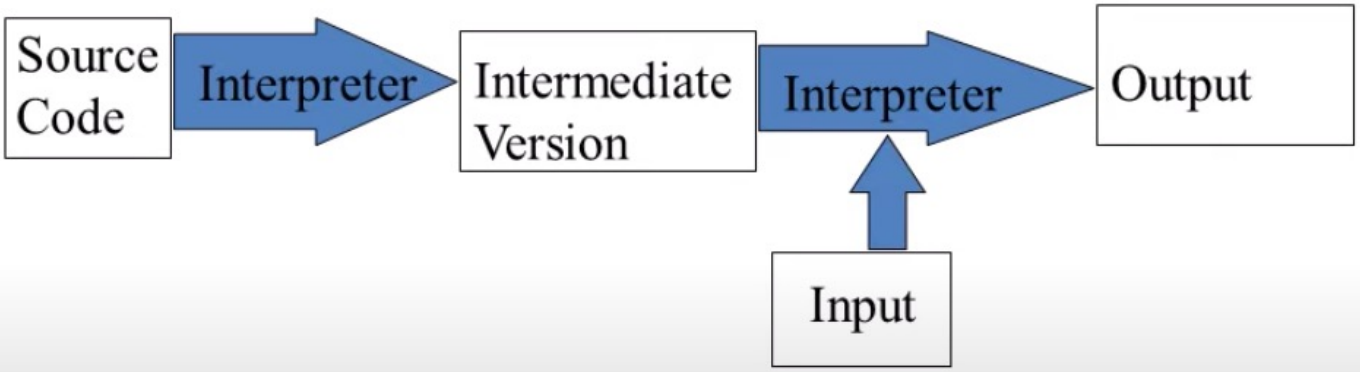
Lexical and Syntax Analysis



# The Compiling Process



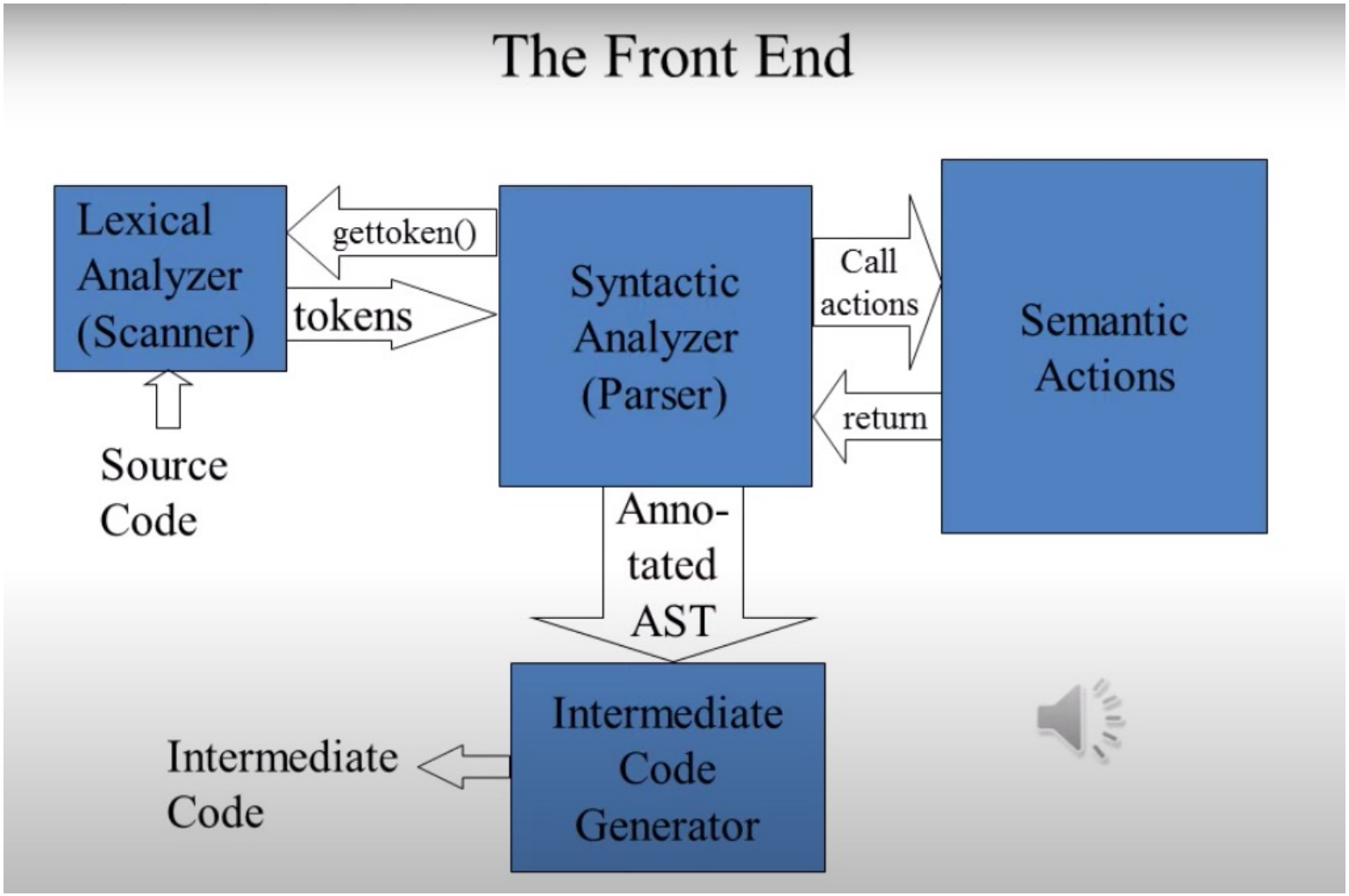
# The Interpretation Process



# Organization of a Complier

- The various component of a complier are organized into a **frontend** and a **backend.**
- The **front end** is designed to produced to some intermediate representation a program written in some source language.
- The backend is designed to produce for a target program from the intermediate representation.

# Front End


The Front End
Lexical Analyzer (Scanner)
gettoken()
tokens
Syntactic Analyzer (Parser)
Call actions
return
Semantic Actions
Source Code
Anno-tated AST
Intermediate Code Generator
Intermediate Code

# Lexical Analysis

- The lexical analyzer (or **scanner**) breakups the stream of text into stream of strings called **lexemes.** (or string tokens).
- The scanner checks one character at a time until it determines that it has found one character that does not belong in lexeme.
- The scanner looks it up in the **symbol table.** And it determines the token associated with that lexeme.

# Syntactic Analysis

- The syntactic analyzer (or **parser**) takes the stream of tokens and determine the syntactic structure of the program.
- The parser create the structure called the parser tree. The parser usually does not store the parse in the memory or the disk, but it does formally recognize program's grammatical structure.

# Semantic Analysis

- Semantic Analysis involve ensuring that the semantic (or meaning) or the program is correct.
- It is quit possible for a program to be syntactically correct but semantically incorrect.
- Semantic Analysis usually means that the data types and the control structures of a program are correct.
- The parsing of generating and intermediate representation (usually and abstract syntax tree) is usually directed by the parsing of the program.

# Symbol Table

- The symbol table tracks all the symbol used in the give program.
- This includes
  - Key words
  - Standard Identifiers
  - Numeric, character and other literals
  - User-defined data types
  - User-defined Variables
- Symbol table must include:
  - Token Classes
  - Lexemes
  - Scope
  - Types
  - Pointers to other symbol table entries.

# Building Lexical Analyzer

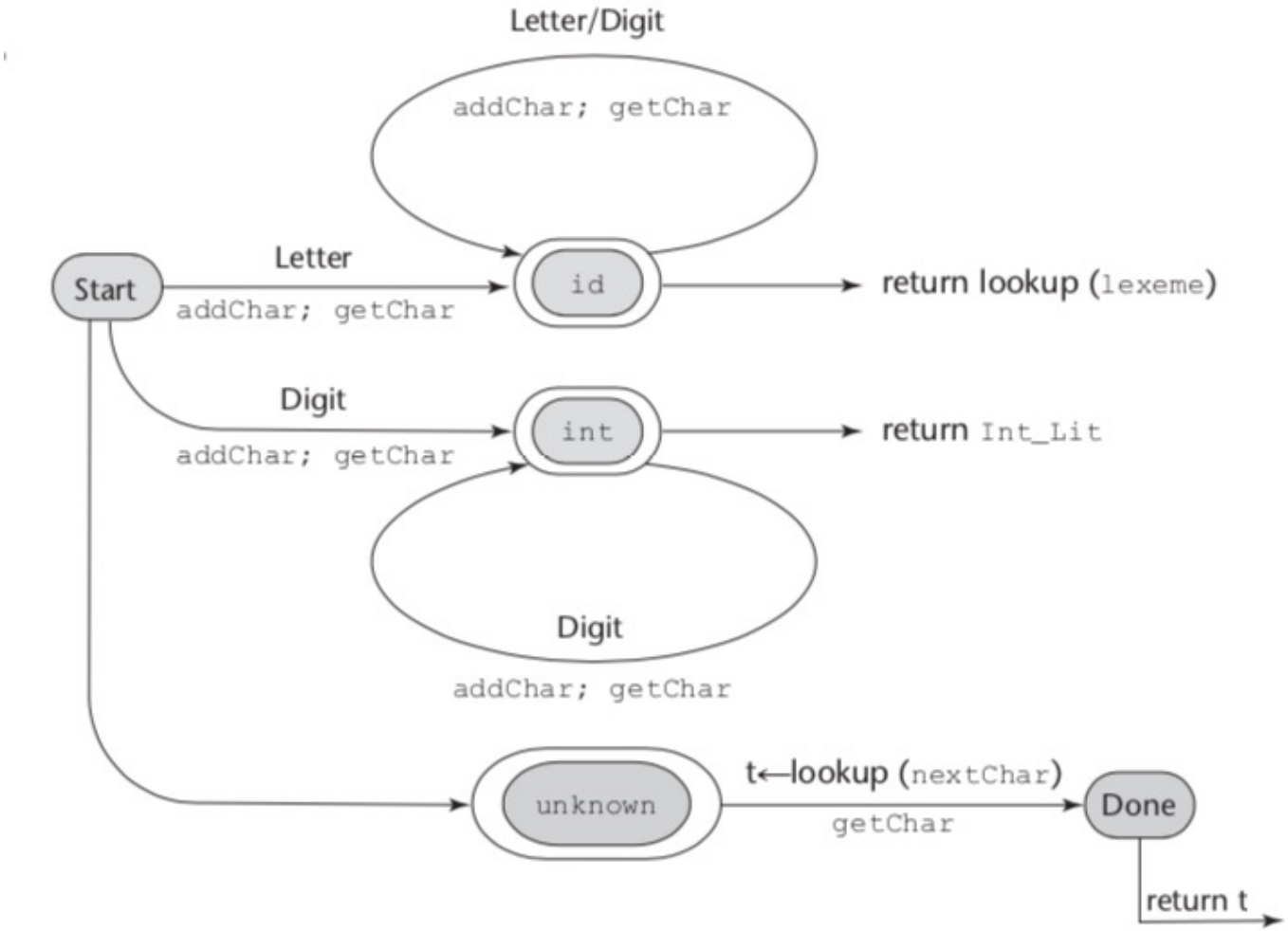
- There are three approaches to build a lexical analyzer
- Write a formal description of the token patterns of the language using a descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a lexical analyzer.
- There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.
- Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
- Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

# State Transition Diagram

- A state transition diagram, or just state diagram, is a directed graph
- The nodes of a state diagram are labeled with state names.
- The arcs are labeled with the input characters that cause the transitions among the states.
- An arc may also include actions the lexical analyzer must perform when the transition is taken.
- A state transition diagram is a special from of fine automata.
- Finite automata can be designed to recognize members of a class of languages called **regular languages**

- Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands.
- We define a character class named **LETTER** for all 52 letters and use a single transition on the first letter of any name.(a|b|c|d|e …….. A|B|C|D|E..).
- We define a character class named **DIGIT** for digits.
- We need a subprogram, which we can name **getChar()**.
- We must determine the character class of the input character and put it in the global variable **charClass.**
- The lexeme being built by the lexical analyzer will be named **lexeme.**

- We implement the process of putting the character in **nextChar** into the string array lexeme in a subprogram named **addChar.**
- A function named **getNonBlank()** is used to skip white space every time the analyzer is called.
- a subprogram named **lookup** is needed to compute the token code for the single-character tokens


Letter/Digit
addChar; getChar
Start
Letter
addChar; getChar
id
return lookup (lexeme)
Digit
addChar; getChar
int
return Int_Lit
Digit
addChar; getChar
unknown
t←lookup (nextChar)
getChar
Done
return t

# The Parsing Process

- The part of the process of analyzing syntax that is referred to as syntax analysis is often called parsing.
- Parsers for programming languages construct parse trees for given programs.
- First, the syntax analyzer must check the input program to determine whether it is syntactically correct.
- When an error is found, the analyzer must produce a diagnostic message and recover.
- The second goal of syntax analysis is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input.
- Parsers are categorized according to the direction in which they build parse trees. i.e. **Top down parser**, **Bottom up parser**.

# Top-down Parsers

- A top-down parser traces or builds a parse tree in preorder.
- A preorder traversal of a parse tree begins with the root.
- Each node is visited before its branches are followed.
- Branches from a particular node are followed in left-to-right order.
- These are easier to understand than bottom up traversals.
- S → xAa
- A → bB
- A → cBb
- A → a
- A **recursive-descent** parser is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language

# Bottom-Up Parsers

- A bottom-up parser constructs a parse tree by beginning at the leaves and progressing toward the root.
- This parse order corresponds to the reverse of a rightmost derivation.
- That is, the sentential forms of the derivation are produced in order of last to first.
- Bottom up parser shift by each token, reducing them into non-terminal as the grammar require.

- S → aAc
- A → aA | b

- a a b c
- a a A c
- a A c
- S

# The LL Grammar class

- Top-down grammars are referred to as LL(k) grammars
  - The Frist L indicate Left to Right scanning
  - The Second L indicate Left to Right Derivation.
  - The k indicate lookahead character
- If we take LL(1) which means we read only one character input
- S → aA | bB  w = aaabd
- A → aB | cB
- B → bC | aC
- C → bD
- D → d

- S → aA → aaB → aaaC → aaabD → aaabd

# The LL Grammar class

- S → abB | aaA  w = abd
- B → d
- A → c | d

- Now this string can not be determined by LL(1). So we will take LL(2). Which means we ll take 2 character input.

- S → abB → abd

- This LL approach is used by recursive decent parser.