UNIT-I

Preliminary Concepts

Reasons for Studying of Programming Languages

Increased capacity to express ideas:

- People can easily express their ideas clearly in any language only when they have clear understanding of the natural language.
- Similarly, if programmers want to simulate the features of languages in another language, they should have some ideas regarding the concepts in other languages as well.

Improved background for choosing appropriate languages

- Many programmers when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to the project.
- If these programmers were familiar with a wider range of languages, they would be better able to choose the language that includes the features that best address the characteristics of the problem at hand.

Increased ability to learn new languages

- In software development, continuous learning is essential.
- The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only two or more languages.
- Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

Better understanding the significance of implementation

- An understanding of implementation issues leads to an understanding of why languages are designed the way they are.
- This knowledge in turn leads to the ability to use a language more intelligently, as it was designed to use.
- We can become better programmers by understanding the choices among programming language constructs and consequences of those choices.

Better use of languages that are already known

- By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

Overall advancement of computing

- There is a global view of computing that can justify the study of programming language concepts.

- For example, many people believe it would have been better if ALGOL 60 ha displaced Fortran in the early 1960s, because it was more elegant and had much better control statements than Fortran. That it did not is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60.

- If those who choose languages were better informed, perhaps, better languages would eventually squeeze out poorer ones.

Different Programming Domains

- **Scientific applications**
  - Large number of floating point computations. The most common data structures are arrays and matrices; the most common control structures are counting loops and selections
  - The first language for scientific applications was Fortran, ALGOL 60 and most of its descedants
  - Examples of languages best suited: Mathematica and Maple
- **Business applications**
  - Business languages are characterized by facilities for producing reports, precise ways of describing and storing decimal numbers and character data, and ability to specify decimal arithmetic operations.
  - Use decimal numbers and characters
  - COBOL is the first successful high-level language for those applications**.**
- **Artificial intelligence**
  - Symbols rather than numbers are typically manipulated
  - Symbolic computation is more conveniently done with linked lists of data rather than arrays
  - This kind of programming sometimes requires more flexibility than other programming domains
  - First AI language was LISP and is still most widely used

– Alternate languages to LISP are Prolog – Clocksin and Mellish

- **Systems programming**
  - The operating system and all of the programming support tools of a computer system are collectively known as systems software
  - Need for efficiency because of continuous use
  - Low-level features for interfaces to external devices
  - C is extensively used for systems programming. The UNIX OS is written almost

entirely in C

- **Web software**
  - – Markup languages
    - Such as XHTML
  - – Scripting languages
    - A list of commands is placed in a file or in XHTML document for execution
    - Perl, JavaScript, PHP
- **Special-purpose languages**
  - – RPG – business reports
  - – APT – programmable machine tools

– GPSS – simulation

Language Evaluation Criteria

The following factors influences Language evalution criteria

1) Readability 2) Simplicity 3) Orthogonality 4) Writ ability 3) Reliability 4) Cost and 5) Others

Readability

- One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.
- Language constructs were designed more from the point of view of the computer than of computer users
- From 1970s, the S/W life cycle concept was developed; coding was relegated to a much smaller role, and maintenance was recognized as a major part of the cycle, particularly in terms of cost. Because ease of maintenance is determined in large part by readability of programs, readability became an important measure of the quality of programs

Overall simplicity

- Language with too many features is more difficult to learn
- Feature multiplicity is bad. For example: In Java, increment can be performed if four ways as:
- Count= count+1
- Count+=1
- Count++
- ++count

- Next problem is operator overloading, in which single operator symbol has more than one meaning

Orthogonality

- A relatively small set of primitive constructs that can be combined in a relatively small number of ways
- Consistent set of rules for combining constructs (simplicity)
    - Every possible combination is legal
- For example, pointers should be able to point to any type of variable or data structure
- Makes the language easy to learn and read
- Meaning is context independent
- VAX assembly language and Ada are good examples
- Lack of orthogonality leads to exceptions to rules
- C is littered with special cases
    - E.g. - **struct**s can be returned from functions but arrays cannot

Orthogonolity is closely related to simplicity. The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

- Useful control statements
- Ability to define data types and structures
- Syntax considerations
    - Provision for descriptive identifiers
        - BASIC once allowed only identifiers to consist of one character with an optional digit
    - Meaningful reserved words
    - Meaning should flow from appearance

Writability

- Most readability factors also apply to writability
- Simplicity and orthogonality
- Control statements, data types and structures
- Support for abstraction
    - Data abstraction

        - Process abstraction
    - Expressivity
        - It is easy to express program ideas in the language
        - APL is a good example
        - In C, the notation C++ is more convenient and shorter than C = C + 1
        - The inclusion of for statement in Java makes writing counting loops easier than the use of while
- **Reliability**
  A program is said to be reliable if performs to its specifications under all conditions.
    - Type checking
        - Type checking is simply testing for type errors in a given program, either by the compiler or during the program execution
        - Because run time type checking is expensive, compile time type checking is more desirable
        - Famous failure of space shuttle experiment due to **int / float** mix-up in parameter passing
    - Exception handling
        - Ability to intercept run-time errors
    - Aliasing
        - Ability to use different names to reference the same memory
        - A dangerous feature
    - Readability and writability both influence reliability
- **Cost**
    - Training programmers to use language
    - Writing programs in a particular problem domain
    - Compiling programs
    - Executing programs
    - Language implementation system (free?)
    - Reliability
    - Maintaining programs
- Others: portability, generality, well-definedness

Influences on Language Design

Computer Architecture

– Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture

• Programming Methodologies

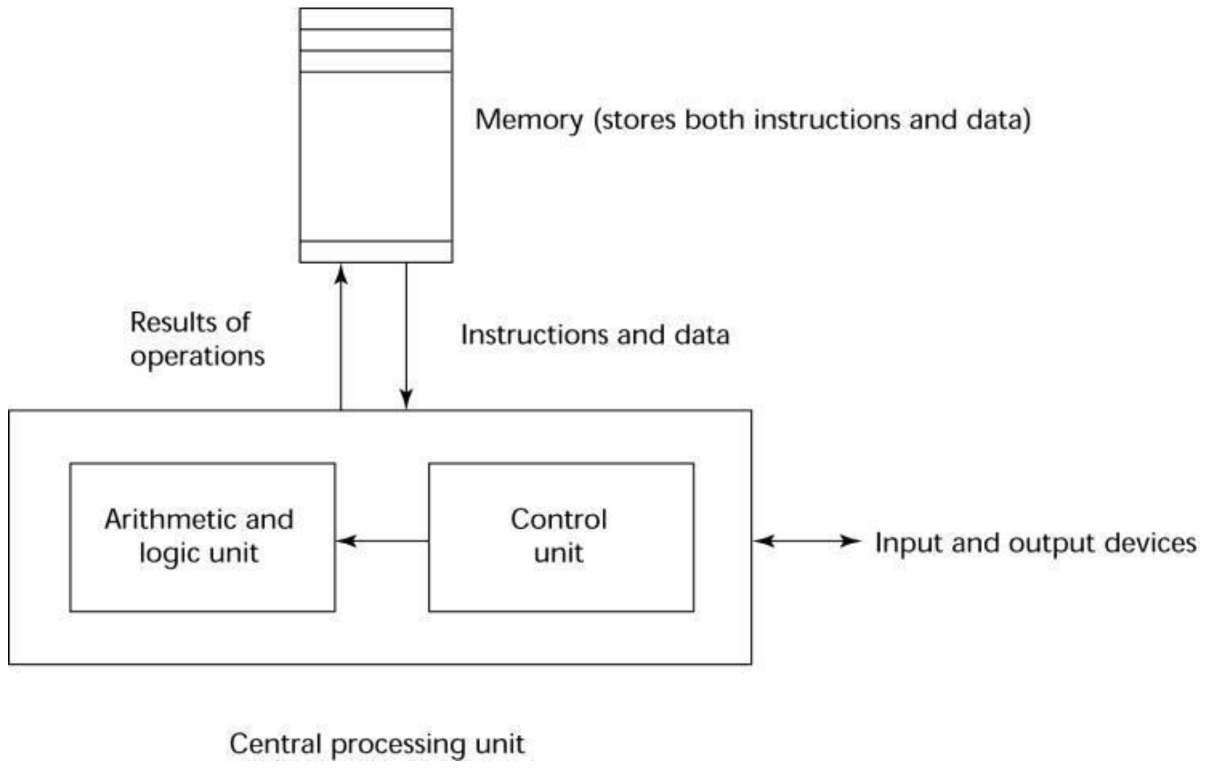
– New software development methodologies (e.g., object oriented software development) led to new programming paradigms and by extension, new programming languages.

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping

- Iteration is efficient

<u>The von Neumann Architecture</u>



<u>Instruction Execution</u>

```
Fetch-execute-cycle (on a von Neumann architecture)
initialize the program counter

repeat forever
fetch the instruction pointed by the
counter
increment the counter decode
the instruction execute the
instruction end repeat
```

Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures

  – structured programming
  – top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  – Data abstraction

- Middle 1980s: Object-oriented programming
  – Data abstraction + inheritance + polymorphism

Different types of programming languages and its design issues

Language Categories
• Imperative
 – Central features are variables, assignment statements, and iteration
 – Examples: C, Pascal

• Functional
 – Main means of making computations is by applying functions to given parameters
 – Examples: LISP, Scheme

• Logic (declarative)
 – Rule-based (rules are specified in no particular order)

 – Example: Prolog

• Object-oriented
 – Data abstraction, inheritance, late binding
 – Examples: Java, C++
• Markup
 – New; not a programming per se, but used to specify the layout of information in Web documents

– Examples: XHTML, XML

Language Design Trade-Offs

Reliability vs. cost of execution

– Conflicting criteria

– Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs

• **Readability vs. writability**

– Another conflicting criteria

– Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.

• **Writability (flexibility) vs. reliability**

– Another conflicting criteria

– Example: C++ pointers are powerful and very flexible but not reliably used

Different types Implementation Methods for programming languages.

We have three different types of implementation methods . They are

- Compilation
  – Programs are translated into machine language
- Pure Interpretation
  – Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters

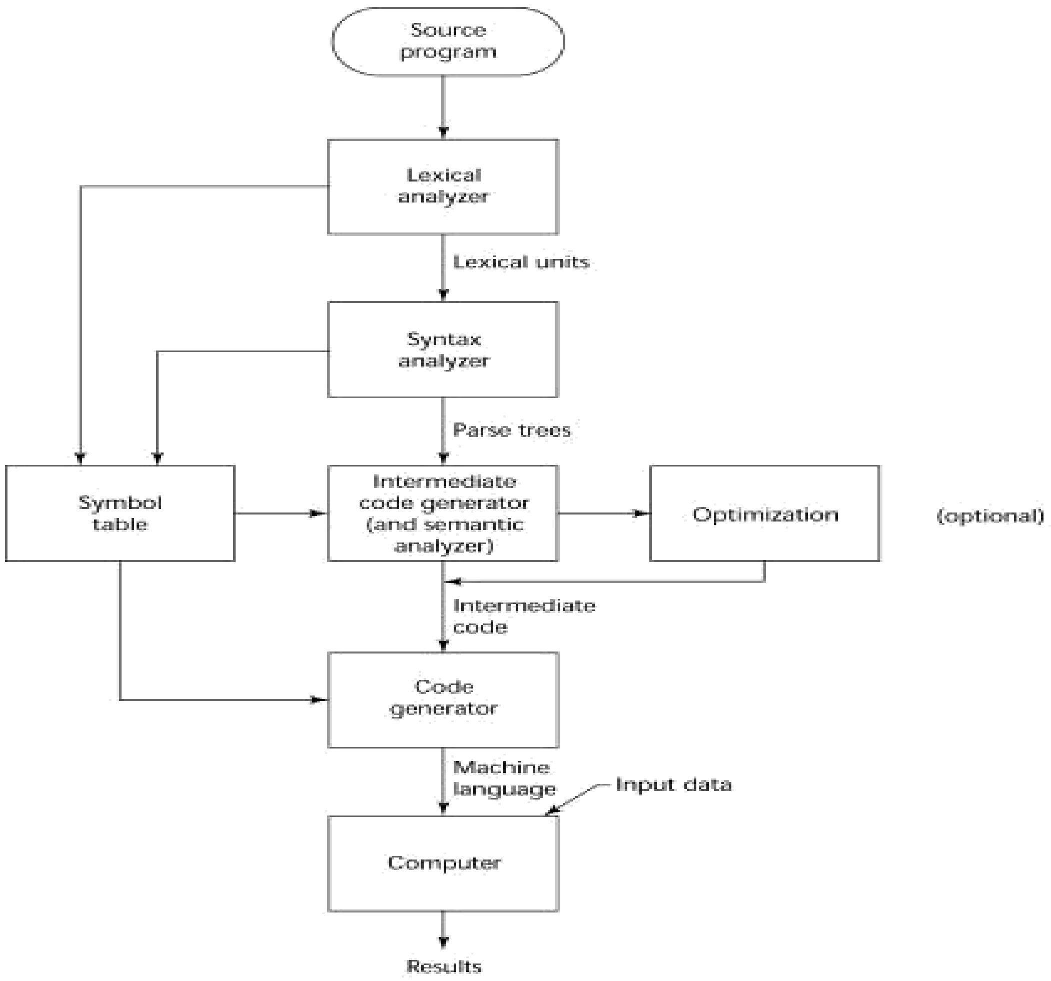
The Compilation Process

Translate high-level program (source language) into machine code (machine language)

• Slow translation, but fast execution.

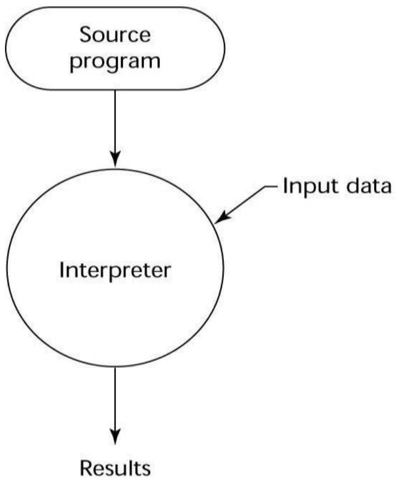
Translates whole source code into object code at a time and generate errors at the end . If no errors ,generates object code. The object code after linking with libraries generates exe code. User input will be given with execution.

Ex: C++ is a compiler



<u>Pure Interpretation Process T</u>translation of source code line by line so that generates errors line by line.If no errors in the code generates object code. While interpretation it selef user input will be given.

• Immediate feedback about errors
• Slower execution
• Often requires more space

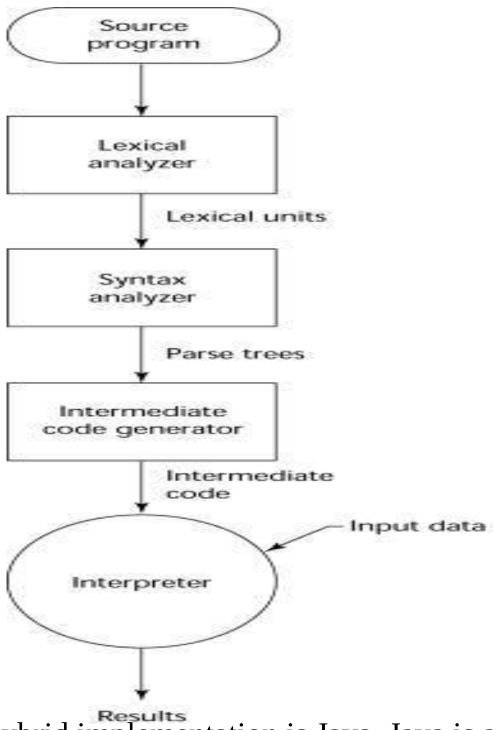


• Used mainly for scripting languages
Example for interpreter is Dbase III plus

Hybrid Implementation Process

A compromise between compilers and pure interpreters

- A high-level language program is is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation



Example for hybrid implementation is Java. Java is a compiled interpreted language.

Just-in-Time Implementation Systems

•Initially translate programs to an intermediate language

• Then compile intermediate language into machine code

• Machine code version is kept for subsequent calls

• JIT systems are widely used for Java programs

• .NET languages are implemented with a JIT system
`

Preprocessors

• Pre-processor macros (instructions) are commonly used to specify that code from another file is to be included
• A pre-processor processes a program immediately before the program is compiled to expand
embedded pre-processor macros
• A well-known example: C pre-processor
– expands #include, #define, and similar macros

Programming Environments'

• The collection of tools used in software development
• The old way used the console and independent tools
  – UNIX/Linux
• vi or emacs for editing
• compiler
• debugger
• Integrated Development Environments provide a graphical interface to most of the necessary tools

Integrated Development Environments
• Eclipse
  – An integrated development environment for Java, written in java
  – Support for other languages is also available
• Borland JBuilder, NetBeans
  – Other integrated development environments for Java
• Microsoft Visual Studio.NET
  – A large, complex visual environment
  – Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

Syntax and Semantics

Describing Syntax and Semantics

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- Pragmatics:
- Syntax and semantics provide a language's definition
  – Users of a language definition
    Other language designers
    Implementers
Programmers (the users of the language)

Terminology

- A metalanguage is a language used to describe another language
- A sentence is a string of characters over some alphabet
- A language is a set of sentences
  - a language is specified by a set of rules
  - A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin)
  - A token is a category of lexemes (e.g., identifier)

Two approaches to Language Definition

- Recognizers

– Read a string and decide whether it follows the rules for the language

– Example: syntax analysis part of a compiler

- Generators

  – A device that generates sentences of a language (BNF)

  – More useful for specifying the language than for checking a string

Syntax

- Backus-Naur Form and Context-Free Grammars

  – Most widely known method for describing programming language syntax

  – Developed as part of the process for specifying ALGOL

  – Define a class of languages called context-free languages

- Extended BNF

  – Improves readability and writability of BNF

BNF Fundamentals

Non-terminals: BNF abstractions used to represent classes of syntactic structures

Terminals: lexemes and tokens
Grammar: a collection of rules

Examples of BNF rules:
<ident_list> → identifier

| identifier, <ident_list>
<if_stmt> → **if** <logic_expr> **then** <stmt>

BNF Rules

A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols

In a context-free grammar, there can only be one symbol on the LHS

A grammar is a finite nonempty set of rules

An abstraction (or nonterminal symbol) can have more than one RHS

Derivations

- BNF is a generative device
  - – Use a grammar to generate sentences that belong to the language the grammar describes
- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Derivation

- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

An Example Grammar

<program> -> <stmts>

<stmts> -> <stmt> | <stmt> ; <stmts> <stmt> -> <var> = <expr>

<var> -> a | b | c | d

<expr> -> <term> + <term> | <term> - <term>

<term> -> <var> | const

An Example Derivation

<program> => <stmts> => <stmt> => <var> = <expr> => a = <expr>

=> a = <term> + <term> => a = <var> + <term> => a = b + <term>

=> a = b + const

Parse Tree

A hierarchical representation of a DERIVATION

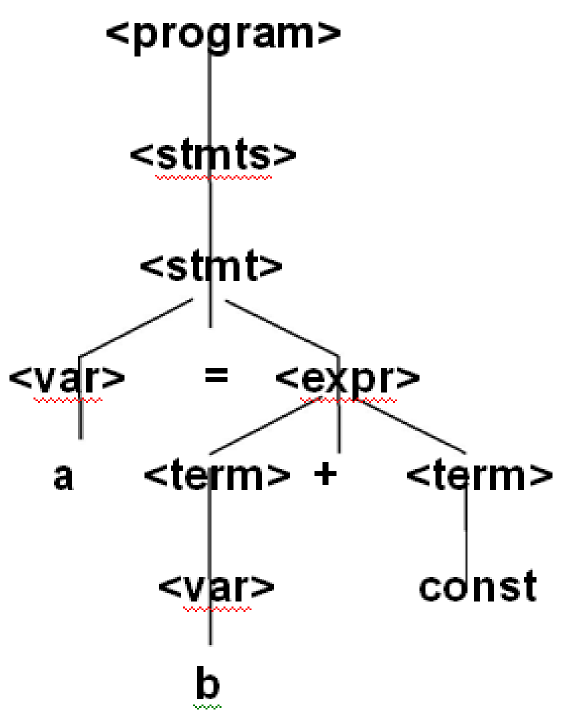
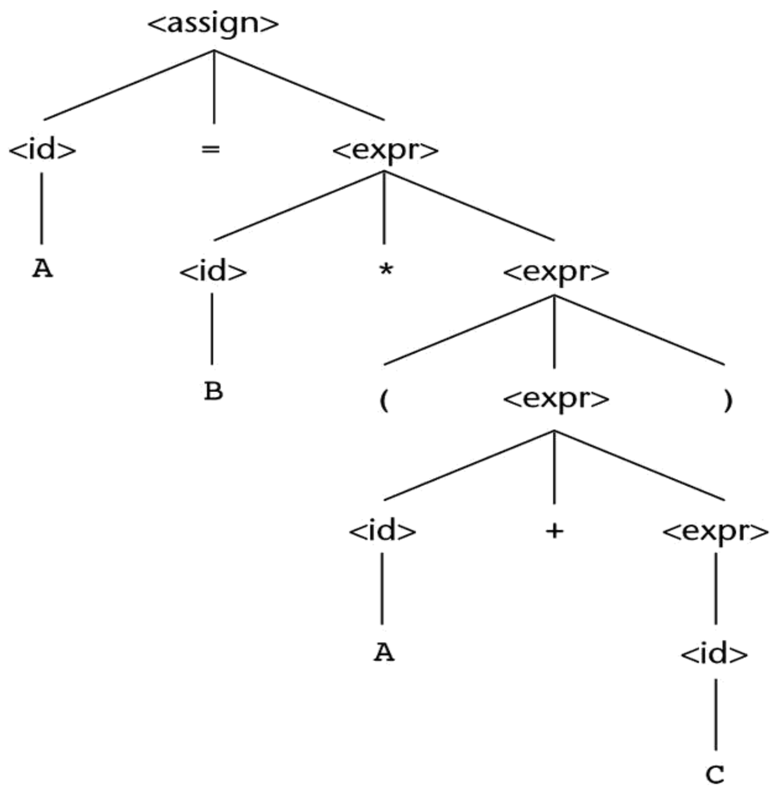


Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`



Associativity of Operators

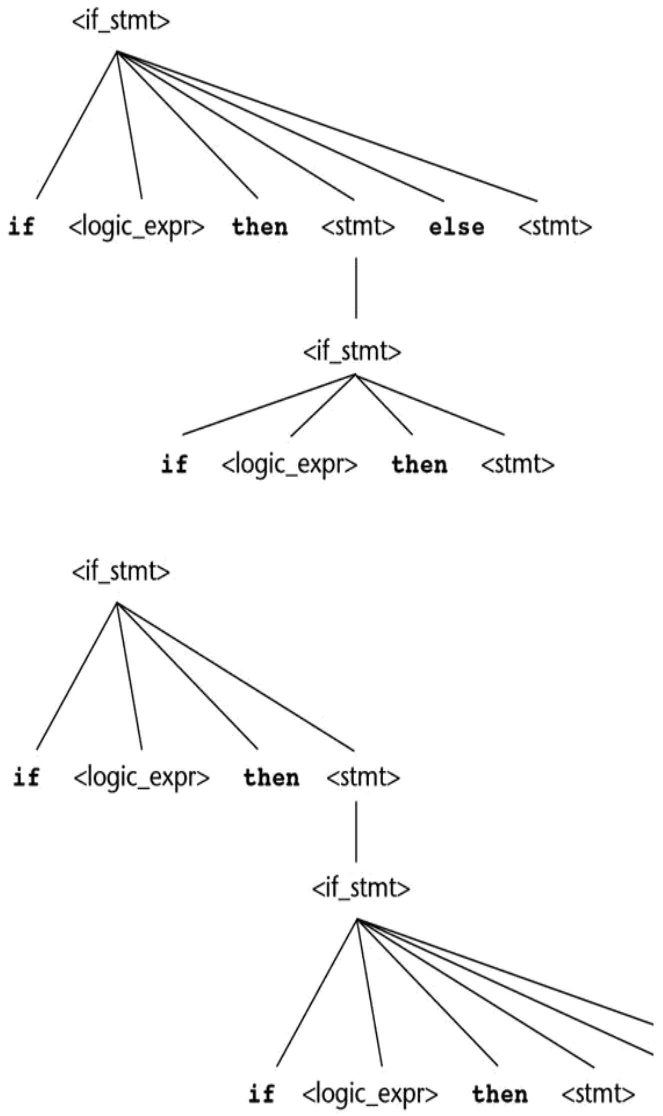
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- For example

  – the requirement that a variable be declared before it can be used is impossible to express in a grammar

– information about variable and expression types could be included in a grammar but only at the cost of great complexity

Figure 3.5

Two distinct parse trees for the same sentenial form



<if_stmt>
if
<logic_expr>
then
<stmt>
else
<stmt>
<if_stmt>
if
<logic_expr>
then
<stmt>
<if_stmt>
if
<logic_expr>
then
<stmt>
<if_stmt>
if
<logic_expr>
then
<stmt>