Principles of Programming Language

UNIT 1

Role of programming languages:

we didn‘t have the idea how the thing [FORTRAN language and compiler] would work out in detail. …We struck out simply to optimize the object program, the running time, because most people at that time believed you couldn‘t do that kind of thing. They believed that machined-coded programs would be so inefficient that it would be impractical for many applications.

Unexpected successes are common – the browser is another example of an unexpected success .

Reasons for Studying Concepts of Programming Languages:

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing

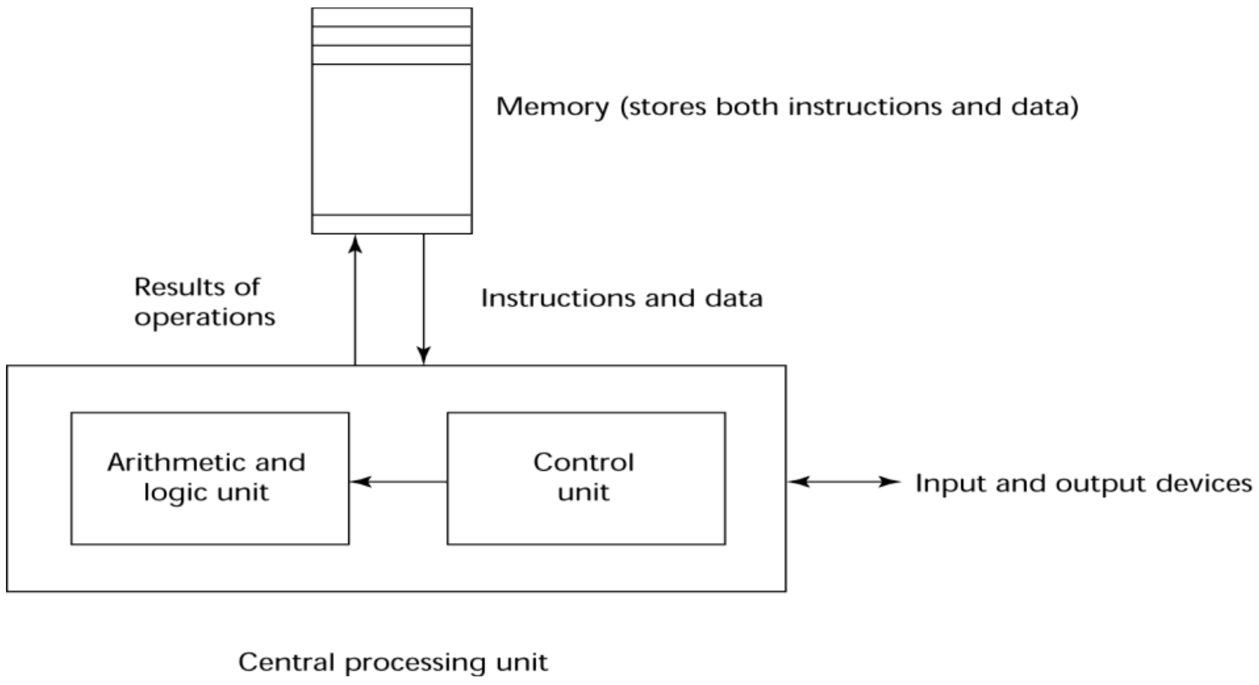
Programming Domains:

• Scientific applications:
  – Large number of floating point computations
  – Fortran
• Business applications :
  – Produce reports, use decimal numbers and characters
  – COBOL :
• Artificial intelligence
  – Symbols rather than numbers manipulated
  - LISP :
• Systems programming
  – Need efficiency because of continuous use
  – C
• Web Software
  – Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

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

The von Neumann Architecture



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

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
  - 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, 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

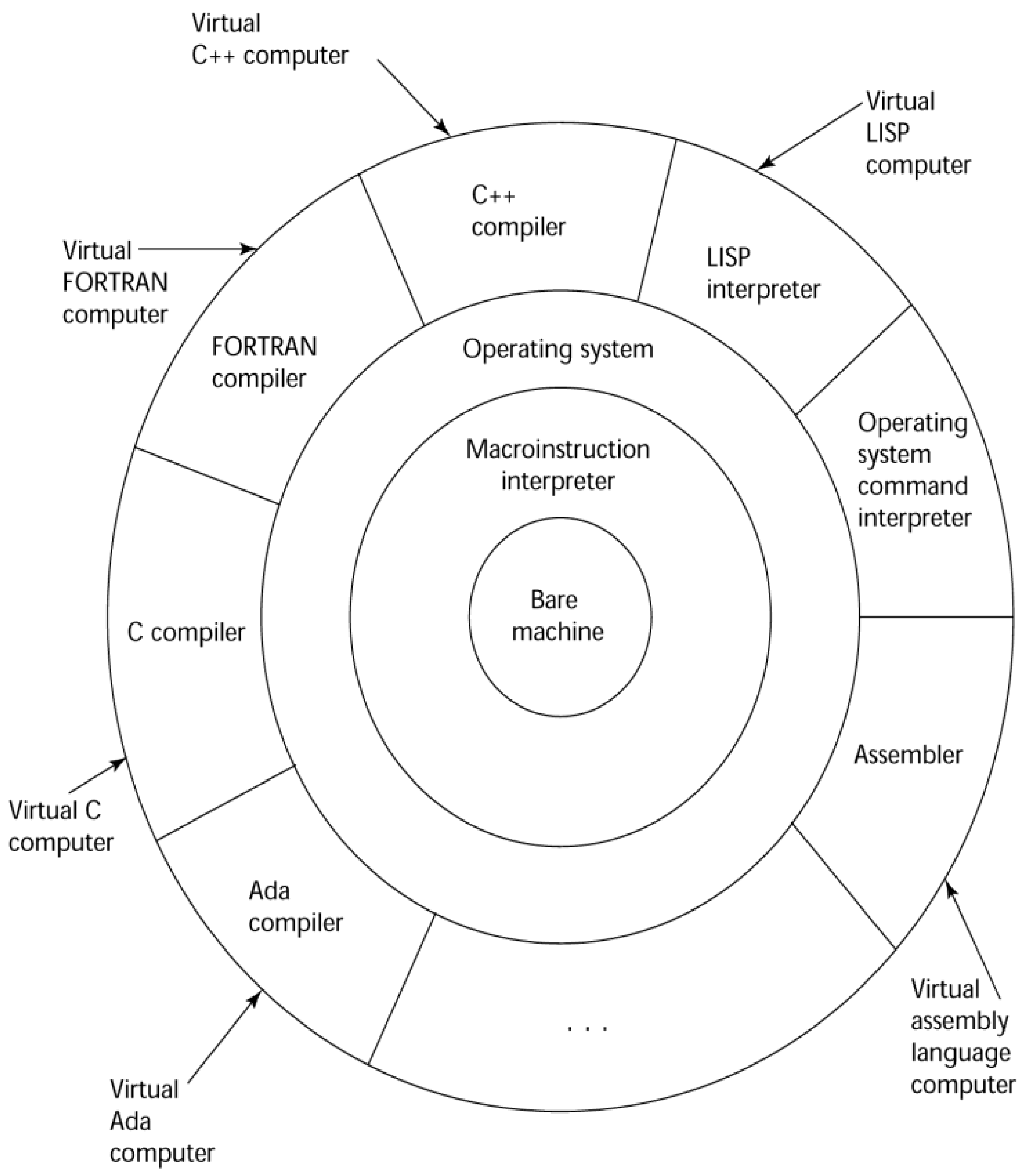
• Writability (flexibility) vs. reliability
  – Another conflicting criteria
  – Example: C++ pointers are powerful and very flexible but not reliably used

Implementation Methods :

• 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

Layered View of Computer :

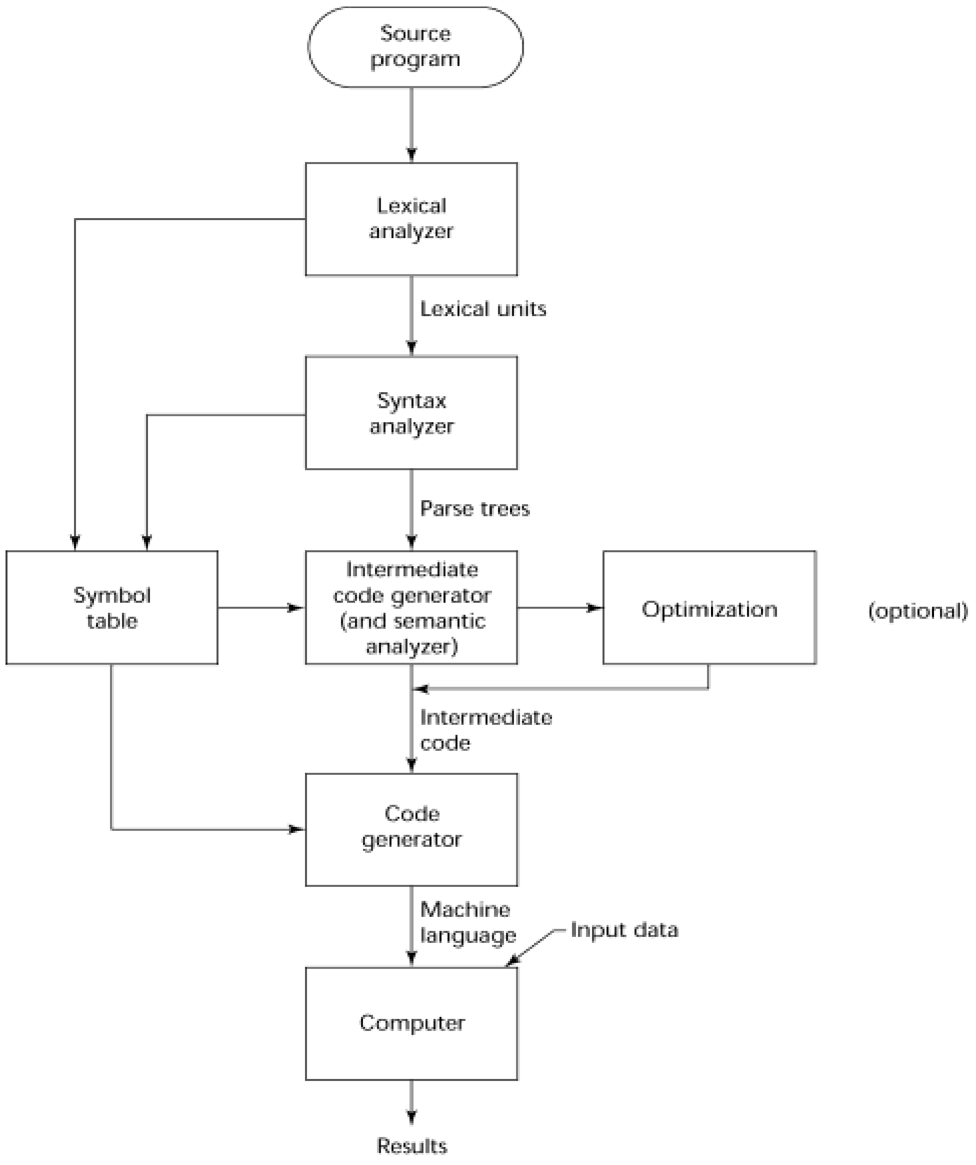
The operating system and language implementation are layered over Machine interface of a computer


Virtual C++ computer
Virtual LISP computer
Virtual FORTRAN computer
C++ compiler
LISP interpreter
FORTRAN compiler
Operating system
Macroinstruction interpreter
Operating system command interpreter
Bare machine
C compiler
Assembler
Virtual C computer
Ada compiler
. . .
Virtual assembly language computer
Virtual Ada computer

Compilation :

• Translate high-level program (source language) into machine code (machine language)
• Slow translation, fast execution
• Compilation process has several phases:
  – lexical analysis: converts characters in the source program into lexical units
  – syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  – Semantics analysis: generate intermediate code
  – code generation: machine code is generated

The Compilation Process:



Additional Compilation Terminologies

• Load module (executable image): the user and system code together
• Linking and loading: the process of collecting system program and linking them to user program

Execution of Machine Code :

• Fetch-execute-cycle (on a von Neumann architecture)

1. initialize the program counter
2. repeat forever
3. fetch the instruction pointed by the counter
4. increment the counter
5. decode the instruction
6. execute the instruction
7. end repeat

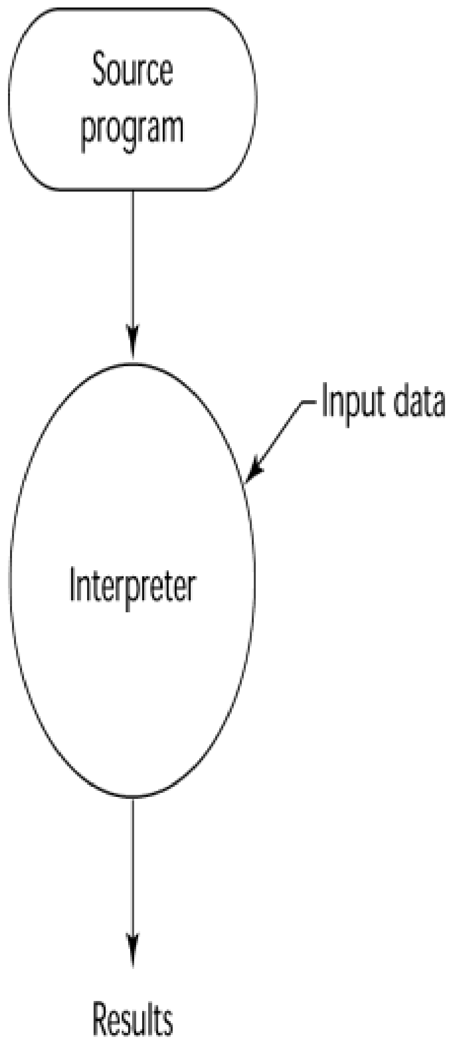
Von Neumann Bottleneck :

• Connection speed between a computer‘s memory and its processor determines the speed of a computer
• Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a *bottleneck*
• Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

Pure Interpretation :

• No translation
• Easier implementation of programs (run-time errors can easily and immediately displayed)
• Slower execution (10 to 100 times slower than compiled programs)
• Often requires more space
• Becoming rare on high-level languages

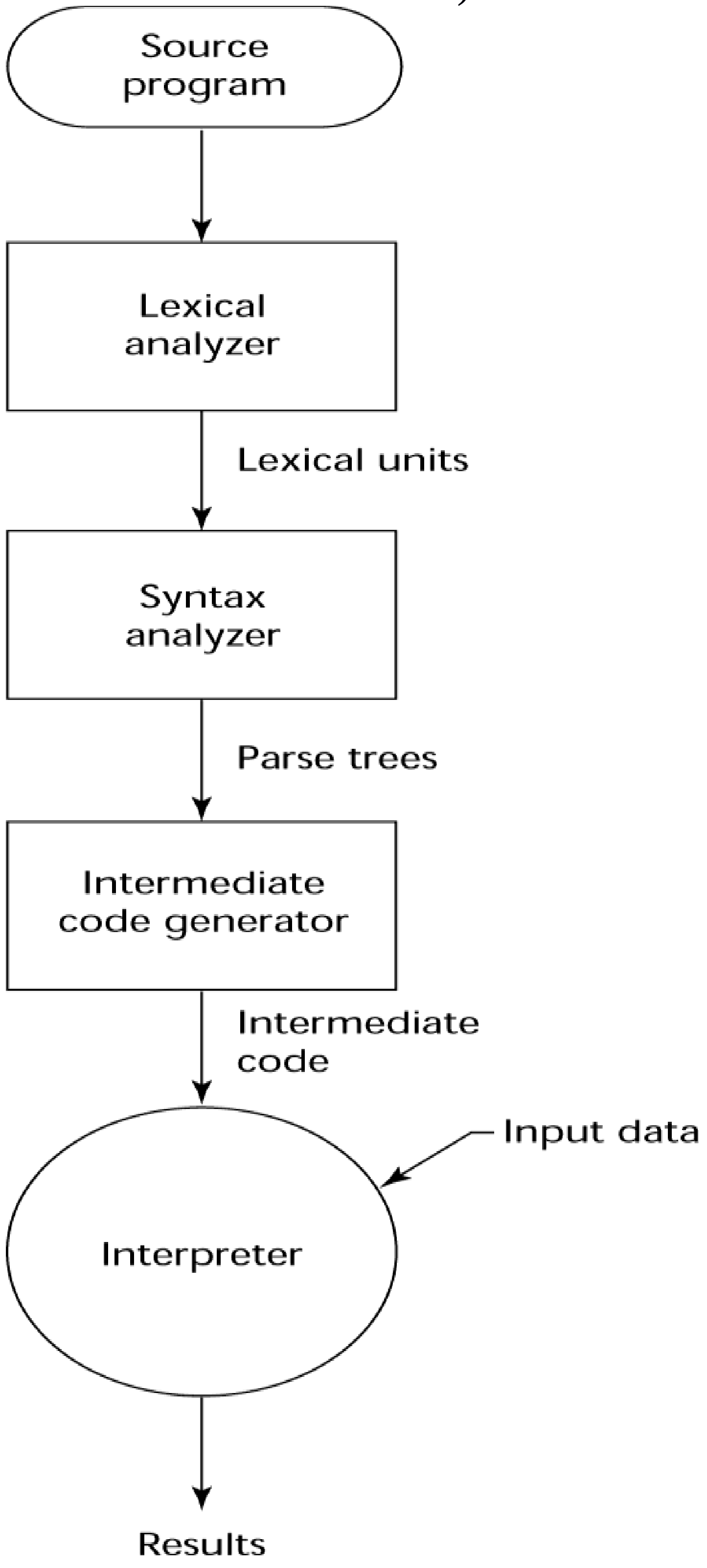
Significant comeback with some Web scripting languages (e.g., JavaScript)



Hybrid Implementation Systems:

• A compromise between compilers and pure interpreters
• A high-level language program is translated to an intermediate language that allows easy interpretation
• Faster than pure interpretation
• Examples
    – Perl programs are partially compiled to detect errors before interpretation
    – Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte

code interpreter and a run-time system (together, these are called *Java Virtual Machine*)



Binding:

A binding is an association, such as between an attribute and an entity, or between an operation and a symbol.

Types of Binding:

1. **Static Binding:**

   A binding is static if it first occurs before run time and remains unchanged throughout program execution.

2. **Dynamic Binding:**

   A binding is dynamic if it first occurs during execution or can change during execution of the program.

Binding Time:

Binding time is the time at which a binding takes place.

Possible binding times:

- Language design time
  e.g., bind operator symbols to operations
- Language implementation time
  e.g., bind fl. pt. type to a representation
- Compile time
  e.g., bind a variable to a type in C or Java
- Load time
  e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
- Runtime

e.g., bind a non-static local variable to a memory cell

Storage Binding:

Allocation: getting a cell from some pool of available cells

De-allocation: putting a cell back into the pool

Data Types:

Let's discuss about a very simple but very important concept available in almost all the programming languages which is called **data types**. As its name indicates, a data type represents a type of the data which you can process using your computer program. It can be numeric, alphanumeric, decimal, etc.

Let’s keep Computer Programming aside for a while and take an easy example of adding two whole numbers 10 & 20, which can be done simply as follows −

10 + 20

Let's take another problem where we want to add two decimal numbers 10.50 & 20.50, which will be written as follows −

10.50 + 20.50

The two examples are straightforward. Now let's take another example where we want to record student information in a notebook. Here we would like to record the following information −

Name:
Class:
Section:
Age:
Sex:

Now, let's put one student record as per the given requirement −

Name: Zara Ali
Class: 6th
Section: J
Age: 13
Sex: F

The first example dealt with whole numbers, the second example added two decimal numbers, whereas the third example is dealing with a mix of different data. Let's put it as follows −

- Student name "Zara Ali" is a sequence of characters which is also called a string.
- Student class "6th" has been represented by a mix of whole number and a string of two characters. Such a mix is called alphanumeric.
- Student section has been represented by a single character which is 'J'.
- Student age has been represented by a whole number which is 13.
- Student sex has been represented by a single character which is 'F'.

This way, we realized that in our day-to-day life, we deal with different types of data such as strings, characters, whole numbers (integers), and decimal numbers (floating point numbers).

Similarly, when we write a computer program to process different types of data, we need to specify its type clearly; otherwise the computer does not understand how different operations can be performed on that given data. Different programming languages use different keywords to specify different data types. For example, C and Java programming languages use **int** to specify integer data, whereas **char** specifies a character data type.

Subsequent chapters will show you how to use different data types in different situations. For now, let's check the important data types available in C, Java, and Python and the keywords we will use to specify those data types.

• Data stored in memory is a string of bits (0 or 1).
• What does 1000010 mean?
66?
'B'?
9.2E-44?
• How the computer interprets the string of bits depends on the context.
• In Java, we must make the context explicit by specifying the *type* of the data.

Primitive Data Types:

• Java has two categories of data:
• primitive data (e.g., number, character)
• object data (programmer created types)
• There are 8 primitive data types:
byte, short, int, long, float, double,
char, boolean
• Primitive data are only single values; they have no special capabilities.

Primitive Data Types

•Almost all programming languages provide a set of *primitive data types*
•Primitive data types: Those not defined in terms of other data types
•Some primitive data types are merely reflections of the hardware
•Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

•Almost always an exact reflection of the hardware so the mapping is trivial
•There may be as many as eight different integer types in a language
•Java‘s signed integer sizes: byte, short, int, long

Primitive Data Types: Floating Point

•Model real numbers, but only as approximations
•Languages for scientific use support at least two floating-point types (e.g., float and double; sometimes more
•Usually exactly like the hardware, but not always
•IEEE Floating-Point
Standard 754

Primitive Data Types: Complex

•Some languages support a complex type, e.g., Fortran and Python
•Each value consists of two floats, the real part and the imaginary part
•Literal form (in Python):
(7 + 3j), where 7 is the real part and 3 is the imaginary part

Primitive Data Types: Decimal

•For business applications (money)
–Essential to COBOL
–C# offers a decimal data type
•Store a fixed number of decimal digits, in coded form (BCD)
•*Advantage*: accuracy
•*Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all

•Range of values: two elements, one for ―true‖ and one for ―false‖
•Could be implemented as bits, but often as bytes
–Advantage: readability

Primitive Data Types: Character

•Stored as numeric codings
•Most commonly used coding: ASCII
•An alternative, 16-bit coding: Unicode
–Includes characters from most natural languages
–Originally used in Java
–C# and JavaScript also support Unicode

User-Defined Ordinal Types

•An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
•Examples of primitive ordinal types in Java
–integer
–char
–boolean

Names

Design issues for names:
Maximum length?
Are connector characters allowed?

Are names case sensitive?
Are special words reserved words or keywords?

Length

If too short, they cannot be connotative
Language examples:
FORTRAN I: maximum 6
COBOL: maximum 30
FORTRAN 90 and ANSI C: maximum 31
Ada and Java: no limit, and all are significant
C++: no limit, but implementors often impose one

Connectors

Pascal, Modula-2, and FORTRAN 77 don't allow
Others do
Case sensitivity
Disadvantage: readability (names that look alike are different)
worse in C++ and Java because predefined names are mixed case (e.g. **IndexOutOfBoundsException**)
C, C++, and Java names are case sensitive
The names in other languages are not

Special words

An aid to readability; used to delimit or separate statement clauses
Def: A keyword is a word that is special only in certain contexts i.e. in Fortran: Real VarName (*Real is data type followed with a name, therefore Real is a keyword)*
Real = 3.4 (*Real is a variable)*
Disadvantage: poor readability

Def: A reserved word is a special word that cannot be used as a user-defined name

Variables

A variable is an abstraction of a memory cell
Variables can be characterized as a sextuple of attributes:
(name, address, value, type, lifetime, and scope)
Name - not all variables have them (anonymous)
Address - the memory address with which it is associated (also called *l*-value)
A variable may have different addresses at different times during execution
A variable may have different addresses at different places in a program
If two variable names can be used to access the same memory location, they are called aliases
Aliases are harmful to readability (program readers must remember all of them)

How aliases can be created:

Pointers, reference variables, C and C++ unions, (and through parameters
Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN
Replace them with dynamic allocation
Type - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
Value - the contents of the location with which the variable is associated

Abstract memory cell - the physical cell or collection of cells associated with a variable.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type. This automatic conversion is called a coercion.
- A type error is the application of an operator to an operand of an inappropriate type

If all type bindings are static, nearly all type checking can be static
If type bindings are dynamic, type checking must be dynamic

- Def: A programming language is strongly typed if type errors are always detected

Strong Typing

- **Advantage of strong typing:**
  allows the detection of the misuses of variables that result in type errors
- Language examples:
- FORTRAN 77 is not: parameters, **EQUIVALENCE**
- Pascal is not: variant records
- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (**UNCHECKED CONVERSION** is loophole)
- (Java is similar)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
- Subranges of integer types are not compatible with integer types
- Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Consider the problem of two structured types:
- Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different?
- (e.g. [1..10] and [0..9])
- Are two enumeration types compatible if their components are spelled differently?
- With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Language examples:

- Pascal: usually structure, but in some cases name is used (formal parameters)
- C: structure, except for records
- Ada: restricted form of name
- Derived types allow types with the same structure to be different
- Anonymous types are all unique, even in:
- **A, B : array (1..10) of INTEGER**:

Scope

- The scope of a variable is the range of statements over which it is visible
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables

Static scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
- In Ada: **unit.name**
- In C++: **class_name::name**

Blocks

- A method of creating static scopes inside program units--from ALGOL 60

Examples:

C and C++:

```
for (...)
{
int index;
...
}
```

Ada:

```
 declare LCL : FLOAT;
begin
...
end
```

Evaluation of Static Scoping

Consider the example:

- Assume MAIN calls A and B
- A calls C and D

- B calls A and E
  Static Scope Example Static Scope Example

Static Scope

- Suppose the spec is changed so that D must now access some data in B

Solutions:

- Put D in B (but then C can no longer call it and D cannot access A's variables)
- Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

MAIN
- declaration of x
SUB1
- declaration of x -
...
call SUB2
...

```
SUB2
...
- reference to x -
...
...
call SUB1
…
```

Static scoping

Reference to x is to MAIN's x

Dynamic scoping

Reference to x is to SUB1's x

Evaluation of Dynamic Scoping:

- Advantage: convenience
- Disadvantage: poor readability

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **static** variable in a C or C++ function

Referencing Enviornment:

- Def: The referencing environment of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of
- the visible variables in all of the enclosing scopes

- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Named Constants

- Def: A named constant is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifest constants) or dynamic

Languages:

- Pascal: literals only
- FORTRAN 90: constant-valued expressions
- Ada, C++, and Java: expressions of any kind

Variable Initialization:

- Def: The binding of a variable to a value at the time it is bound to storage is called initialization
- Initialization is often done on the declaration statement

e.g., Java

int sum = 0;