**Principles of Programming Language  
Lecture Notes of Unit -1**

**1-Introduction:-**

We sometimes use the term ***programming linguistics*** to mean the study of programming languages. This is by analogy with the older discipline of *linguistics*, which is the study of natural languages. Both programming languages and natural languages have *syntax* (form) and *semantics* (meaning). However, we cannot take the analogy too far. Natural languages are far broader, more expressive, and subtler than programming languages. A natural language is just what a human population speaks and writes, so linguists are restricted to analyzing existing (and dead) natural languages. On the other hand, programming linguists can not only analyze existing programming languages; they can also design and specify new programming languages, and they can implement these languages on computers.

A programming language must be ***universal***. That is to say, every problem must have a solution that can be programmed in the language, if that problem can be solved at all by a computer.

A programming language must also be ***implementable*** on a computer. That is to say, it must be possible to execute every well-formed program in the language. Mathematical notation (in its full generality) is not implementable, because in this notation it is possible to formulate problems that cannot be solved by any computer. Natural languages also are not implementable, because they are imprecise and ambiguous.

the ***concepts*** that underlie the design of programming languages: *data* and *types, variables* and *storage, bindings* and *scope, procedural abstraction, data abstraction, generic abstraction, type systems, control*, and *concurrency*.

**2-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.

**1.3 Characteristics and attributes of programming Language**

1. **Clarity, Simplicity and Unity-**

Language with too many features is more difficult to learn

Feature multiplicity is bad. For example: In C, increment can be performed if four ways as:

• Count= count+1

• Count+=1

• Count++

• ++count

1. **Orthogonality**

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

1. **Naturalness**

A Language needs a syntax that allows the program structure to reflect that underlying logical structure of a program

1. **Support for Abstraction**

Abstraction hides the complexity. It keeps only essential information details working of any entity is not required.

1. **Program Verifications**

The technique for verifying that a program correctly perform its required function.

1. **Programming environment**

the technical structure of a programming language is only one aspect affecting its utility. The presence of an appropriate programming environment may make a weak language easier to work then a stronger language that has little external support.

1. **Portability of Program**

Language should be portable. If we run one program to first machine and if this program can be run at different kind of machine without changing the program then language is called portable.

1. **Cost of use**

– 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

1. **Robustness**

A programming language should have good exception handling technique. Exception means run time error.

1. **Reliable:**

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 / floatmix-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

**1.4 Historical Development**

****

**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

– not a programming, but used to specify the layout of information in Web documents

– Examples: XHTML, XML

**1.5 Language Categories**

1. **Compiled vs. Interpreted**
   1. The compiler versus interpreter implementation is often fuzzy
      1. One can view an interpreter as a virtual machine
      2. A processor (CPU) can be viewed as an implementation in hardware of a virtual machine
   2. Some languages cannot be purely compiled into machine code
      1. Some languages allow programs to rewrite/add code
   3. In general, compilers try to be as smart as possible to fix decisions that can be taken at compile time to avoid to generate code that makes a decision at run time
   4. Compilation leads to better performance in general
      1. Allocation of variables without variable lookup at run time
      2. Aggressive code optimization to exploit hardware features
   5. Interpretation leads to better diagnostics of a programming problem
      1. Procedures can be invoked from command line by a user
      2. Variable values can be inspected and modified by a user
2. **High-level vs. Low level**

1. HLL are strongly abstracted from the details of a computer. It may use natural   
   elements, English like words, be easier to use, making the program easier to   
   understand and more simpler as compared to LLL.  
2. HLL uses arrays, complex arithmetic or boolean expressions, subroutines and     
    functions, loops, threads, locks, and other abstract computer science concepts     
    rather then dealing with registers, memory addresses and call stacks as   
   compared to LLL.   
3. HLL focus is usability whereas the focus of LLL is efficiency.  
4. Some HLL's are Python, Lisp

5. A low-level language is a programming language that provides little or no abstraction of programming concepts, and is very close to writing actual machine [instructions](http://www.computerhope.com/jargon/c/compinst.htm). Two good examples of low-level languages are assembly and machine code.

1. **General purpose vs. Special purpose**

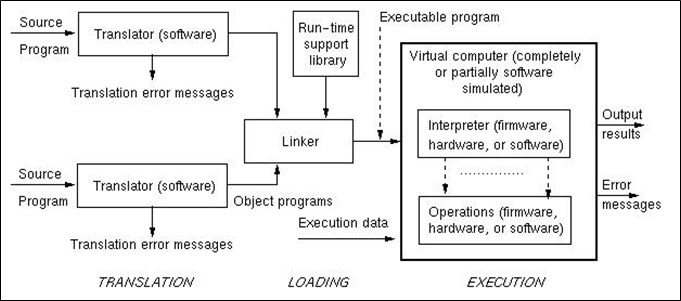
* A general-purpose programming language is a programming language designed to be used for writing software in a wide variety of application domains (a general-purpose language).
* There are languages that were developed with a special purpose in mind (also called domain specific languages) Usually, special purpose languages require more intelligence in compiler/interpreter and run-time system and one reason why some of these languages never were successful among the target audience were bad compilers/interpreters and run-time systems that essentially threw away any advantages of the languages

**1.6 Language Translation**

1. **Assembler**

Usually, special purpose languages require more intelligence in compiler/interpreter and run-time system and one reason why some of these languages never were successful among the target audience were bad compilers/interpreters and run-time systems that essentially threw away any advantages of the languages

1. **Compiler**
   1. **Structure of a typical language implementation**



* 1. **Stages of translation**

**Lexical Analysis** The purpose of this assignment is to design and implement the lexical analyzer: a program that will examine each token in the token table built by the tokenizer, and will collect information about the tokens necessary for the syntax and semantics analysis.

The tasks of the lexical analyzer are:

1. Build a table of token types

2. Build symbol tables:

a. Store all variable names (without repetition) in an array

b. Store all label names (without repetition) in an array. For each label, record in a matching array the number of the statement to which the label is attached.

3. Create an address table that provides references to the variables, label, keywords, and has the numeric values of the numbers used in the source program

4. Report errors:

a. two or more labels with the same name preceding statements,

b. dangling labels – used after ‘goto’ but not attached to a statement

At this phase, the students have to make several design and implementation choices.

First of all, they have to decide how to represent the syntactic categories of the tokens. Obviously, if they use the names from the BNF description, their table will use too much memory. The obvious solution is to use names of one letter only, e.g. V for , N 6 for . By necessity they introduce a category K for ‘keyword’, which however is not used in the BNF notation. Thus, they can appreciate the importance of having adequate documentation, since only in the documentation they will write about why they have used ‘K’ in the table of token types to represent any keyword.

Second, some of the students realize that it does not seem right to store in one table names of variables that may be of different type. They also realize that so far they have not thought about how to declare the type of the variables. Students start asking questions in class and thus everybody becomes aware of the ‘type’ issue. This helps understand the usage of the descriptors, studied at the same time. Generally, the majority of students choose to have only one type – integer, which simplifies the implementation, but they also realize that the decision significantly reduces the potential of the language. Students have two weeks to accomplish this assignment, because it is relatively more difficult than the other assignments constituting the project.

**Syntactic Analysis** The tasks of the syntactic analysis are:

• Determine the syntax structure of each statement

• Report errors if any

• Record in an array the index of the syntax pattern for each statement (to be used later by the semantic analysis and the synthesis of the executable table)

The syntactic analyzer is based on the BNF description of the language. The assignment however does not require the development of a full parser. This decision is based on the following considerations:

a. Building a parser that assigns parse trees is a task that exceeds the scope of this class

b. The language is simple enough so that the parse trees of each type of statements can be flattened, i.e. students can use syntactic patterns derived from the BNF description. The syntactic patterns are derived on paper and then embodied in the syntax module as an array of type string.

Examples of such patterns are:

rules[0] = "V=V+N"; // a = a+1, a = a-1

rules[1] = "V=V-N" ; // a = a-1

rules[2] = "V=V"; // a = b

rules[3] = "V=N"; // a = 5

rules[5] = "KV=NKL"; // if a = 0 goto L

……

For a statement to be syntactically correct it must match some pattern and it must meet certain conditions specific for each pattern. The second requirement accounts for the context-sensitive features of the language that cannot be represented in the context-free syntax patterns. For example a statement with syntax pattern "KV=NKL" is valid only if the first keyword is “if” and the second keyword is “goto”.

With this assignment the students understand the difference between context-free and context-sensitive language description.

**Semantic Analysis** In this project, semantic analysis is based on the operational semantics of each statement. The operational semantics is defined by a set of rules specifying how the state of the memory changes while executing the statement. The semantics of the program is defined as a composition of the semantics of each statement. The program memory is simulated by an array that serves as a storage for the values of the program variables.

The semantic analysis in this project consists in appropriate encoding of the semantics of each statement, which will be decoded when running the program.

The result of the semantic analysis is a table that contains the codes of the operations specified in the statements and references to their operands (if any). The codes are designed by the students. Depending on the language design, statements have two or three operands at most. For example, if addition and subtraction are used only in statements of the type “a = a + N”, the number of the operands is two. If the language allows statements of the type “a = b + N”, the operands will be three. The operands are represented by their ‘addresses’ – indexes in the memory array if the operands are variables, or as literals if they are numbers.

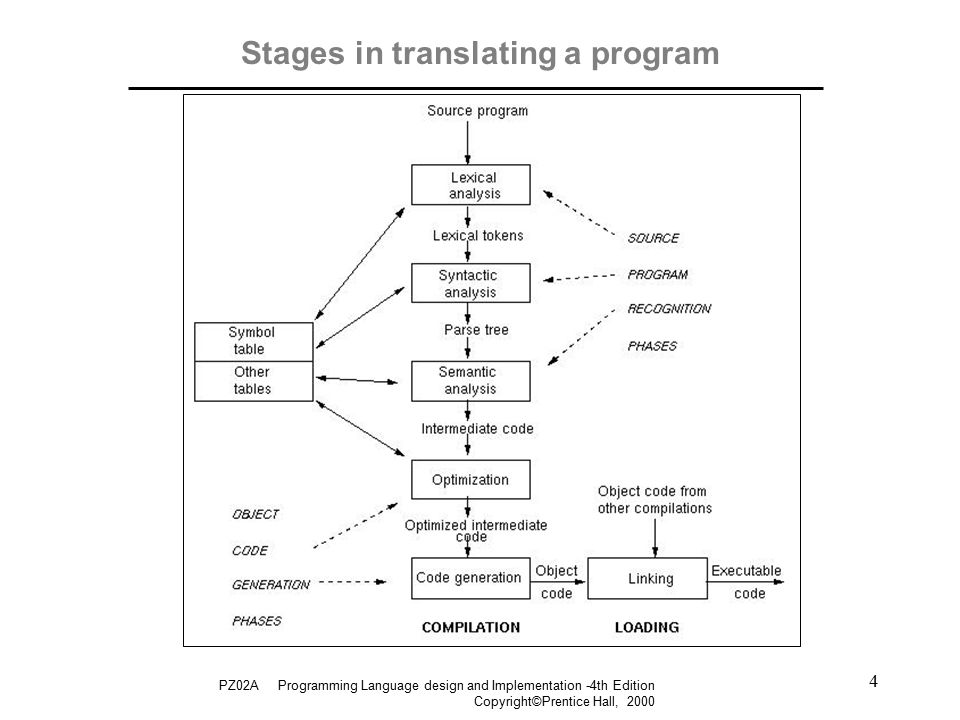
Conceptually, the resulting table can be viewed as containing machine-like instructions with their operands in a two- or three-operand machine. This representation is very convenient for decoding at the final step of the interpretation – executing the program.

With this assignment, students gain practical understanding of the operational semantics

1. **Code Generation and Optimisation**

Code generation and optimisation exists to generally transform the annotated syntax tree into some form of intermediate code, typically three-address code or code for a virtual machine (e.g., p-code). Some level of optimisation can be applied here. This intermediate code can then be transformed into instructions for the target mahcine and optimised further.

Optimisation is really code improvement, e.g., constant folding (e.g., replace x = 4\*4 with x = 16), and also at a target level (e.g., multiplications by powers of 2 replaced by shift level instructions).

****

**1.7 Binding and Binding Times**

* A *binding* is an association between a name and the thing that is named
* *Binding time* is the time at which an *implementation decision* is made to create a binding
  1. *Language design time:* the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  2. *Language implementation time:* fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc.
  3. *Program writing time:* the programmer's choice of algorithms and data structures
  4. *Compile time:* the time of translation of high-level constructs to machine code and choice of memory layout for objects
  5. *Link time:* the time at which multiple object codes (machine code files) and libraries are combined into one executable
  6. *Load time:* the time at which the operating system loads the executable in memory
  7. *Run time:* the time during which a program executes (runs)

**Types of Binding**

1. **Static Binding:** the binding performed at compile time is known as static binding. In this type associated with the variable is determined at compile time. Once the type of variable is declared it can not be changed.

May be specified through explicit or an implicit declaration

***Explicit declaration*** is a program statement used for declaring the types of variables

Ex: int a

***Implicit declaration*** is a default mechanism for specifying types of variables (first appearance of variable in program)

* 1. FORTRAN, PL/I, and BASIC provide implicit declarations
     1. Ex: Fortran: vars starting with I-N are integers; others are reals
  2. Advantage: writability (fewer lines of code to write)
  3. Disadvantage: reliability
     1. implicit declaration prevents compilation process from detecting typographical and programmer errors
     2. In FORTRAN, variables that are accidentally left undeclared are given default types and unexpected attributes
     3. less trouble with Perl: uses names beginning with special char ($ for scalar; @ for arrays, % for hash structure)

1. **Dynamic Binding** the binding performed at run time is known as dynamic binding. In this type associated with the variable is determined at run time. A single variable could have many different types at different point in the program.

Dynamic Type Binding (APL, JavaScript, SNOBOL)

* 1. Type is not specified by a declaration statement, nor can it be determined by the spelling of its name
  2. Type is specified through an assignment statement   
     e.g. in JavaScript:
     1. list = [2, 4.33, 6, 8]; (1-dim array)
     2. list = 17.3; (scalar) % using same var name
  3. Advantage: flexibility (generic program units)
     1. Ex: program to sort data; at run time, can sort integers, reals, characters, etc.
  4. Disadvantages:
     1. High cost (dynamic type checking; can only be implemented using interpreters)
     2. Type error detection by the compiler is difficult