# G. PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

# Accredited by NAAC with ‘A’ Grade of UGC, Approved by AICTE, New Delhi

# Permanently Affiliated to JNTUA, Ananthapuramu

# (Recognized by UGC under 2(f) and 12(B) & ISO 9001:2008 Certified Institution)

# Nandikotkur Road, Venkayapalli, Kurnool – 518452

## Department of Computer Science and Engineering

 **Lecture Notes**

 **On**

 **Principles of Programming Language**

**For III YEAR – 1 Sem B.Tech CSE (R15 Regulations)**

#  UNIT-I

# SOFTWARE DEVELOPMENT PROCESS

**INTRODUCTION**:-

* Programming language are tools for writing software.
* To know about programming languages, must begin with the discussion of the software development process and the role of programming languages in this process.
* Software is a part of computer system.
* Software is nothing but collection of computer programmes and related documents that are intended to provide desired features, functionalities and better performances.
* Software products may be:-

i) Generic:- That means to be sold to a range of different customers.

ii) Custom:- That means developed for a single customer according to

their specification.

**SOFTWARE DEVELOPMENT PROCESS:-**

* From the inception of an idea for a software system until it is implemented and delivered to a customer, and even after that,the software undergoes gradual development and evolution.
* The software is said to have a life cycle of composed of several phases.
* Each of these phases results in the development of either a part of the system or something associated with the system, such as fragment of specification.
* A sample software development process based on waterfall model may be comprised of the following phases.

**1) Requirement analysis and specification:-**

i) The purpose of this phase is to identify and document the exact requirements for the system.

ii) These requirements are developed jointly by users and software developers.

iii) The success of a system is measured by:

 a) How well the software mirrors these stated requirements.

 b) How well the users' perceived needs reflect the real needs.

 c) The result of their phase is a requirementsdocumentstating what the system should do, along with user’s manuals, feasibility and cost studies, performance requirements, and so on.

**2) Software Design Specification:-**

i) Starting with the requirements document, software designers design the software system.

ii) The result of this phase is a system design specification document identifying all of the modules comprising the system and their interfaces.

iii) The purpose of the design phase is to specify a particular software architecture that will meet the stated requirements.

**3) Implementation:-**

i) Implementation is nothing but coding part.

ii) The system is implemented to meet the design specified in the previous phase.

iii) The goal of the implementation step is to choose how, among the many possible ways, the system shall be coded to meet the design specification.

iv) The result is a fully implemented and documented system.

**4) Verification and Validation:-**

i) Verification and validation are nothing but testing.

ii) This phase assesses the quality of the implemented system, which is then delivered to the user.

iii) This phase checks are accomplished by answering the following two questions:

 ---- “Are we building the product right?”----

 ---- “Are we building the right product?”----

iv) Two specific kinds of assessment performed during implementation are module testing and integration testing.

v) Module testing is done by each programmer on the module.

vi) Integration testing is done on a partial aggregation of modules; it is basically aimed at uncovering inter module inconsistencies.

**5) Maintenance:-**

i) Following delivery of the system, changes to the system may become necessary either because of detected malfunctions, or a desire to add new capabilities or to improve old ones, or changes that occurred in operational environment.

 Eg:- The operating system of the target machine.

ii) These changes are referred as maintence

Programming languages are used only in some phases of the development process.

* They are obviously used in the implementation phase, when algorithms and data structures are defined and coded for the modules that form the entire application.
* Software development environment is the set of process and programming tools used to create the program or software product.

**LANGUAGES AND SOFTWARE DEVELOPMENT ENVIRONMENTS:-**

* The work in any of the phases of software development may be supported by computer-aided tools.
* The phase currently supported best is the coding phase, with such tools as text editors, compilers, linkers, and libraries.
* A debugger is commonly used to locate faults in a program and eliminate them.
* These computer-aided program development tools have increased programming productivity by reducing the chances of errors.
* Software development involves much more than programming. In order to increase the productivity software development, computer support is needed for all of its.
* By a software development environment we mean an integrated set of tools and techniques that aids in the development of software.
* The environment is used in all phases of software development: requirements, design, implementation, verification and validation, and maintenance.

 An idealized scenario for the use of such an environment would be the following. A team of application and computer specialists interacting with the environment develops the system requirements. The environment keeps track of the requirements as they are being developed and updated, and guards against incompleteness or inconsistency.

 It also provides facilities to validate requirements against the customer’s expectations, following the completion of the requirements, system designers, interacting with the environment, develop an initial system design and gradually refine it, i.e; they specify the needed modules and the module interfaces. Test data may also be produced at this stage. The implementers then undertake to implement the system based on the design. The tools provided by the software development environment to support implementation are the most familiar. They include programming language processors, such as editors, compilers, simulators, interpreters, linkers, debuggers, and others.

**Languages and software design methods:-**

* The relationship between software design methods and programming languages is an important one. Some languages provide better support for some design methods than others.
* Older languages, such as FORTRAN, were not designed to support specific design methods.
* Conversely, Pascal was designed with the explicit goal of supporting topdown program development and structured programming.
* To understand the relationship between a programming language and a design method, it is important to realize that programming languages may enforce a certain programming style, often called a *programming paradigm*.
* The most prominent programming languages paradigms are
1. **Procedural Programming:-**

This is conventional programming style, where programs are decomposed into computation steps that perform complex operations. Procedures and functions are used as modularization units to define such computation steps.

1. **Functional Programming:-**

The functional style of programming is rooted in the theory of mathematics functions. It emphasizes the use of expressions and functions. The functions are the primary building blocks of the program. They may be passed freely as parameters and may be constructed and returned as result parameters of other functions.

**c) Abstract Datatype Programming:-**

Abstract datatype (ADT) programming recognises abstract datatypes as the unit of program modularity. CLU was the first language designed specially to support this style of programming.

**d)Module-Based Programming:-**

Rather than emphasizing abstract-data type, module-based programming emphasizes modularization units that are groupings of entities such as variables procedures, functions, types, etc. A program is composed of a set of such modules.

**e)Object-Oriented Programming:-**

The object Oriented programming style emphasizes the definition of classes and objects. Instances of classes are created by the program as needed during program execution. This style is based on the definition of hierarchies of classes and run-time selection of units to execute. C++, smalltalk and Eiffel are representative languages of this class.

**f)Generic Programming:-**

This style emphasize the definition of generic modules that may be instantiated, either at compile time on runtime ,to create the entities,datastructures,functions,and procedures needed to form the program. The approach to programming encourages the development of highlevel, generic, abstractions as units of modularity.

**g)Declarative Programming:-**

This style emphasizes the declarative description of a problem, rather than the decomposition of the problem into an algorithmic implementation.

**Language and Computer Architecture:-**

* Design methods influence programming languages in the sense of establishing requirements for the language to meet in order to better support software development.
* Programming languages have been constrained by the ideas of Von Neumann , because most current computers are similar to the original Von Neumann architecture (figure 1)



* The Von Neumann architecture, sketched in Figure 1, is based on the idea of a memory that contains data and instructions, a CPU, and an I/O unit.
* The CPU is responsible for taking instructions out of memory, one at a time.
* Machine instructions are very low-level. They require the data to be taken out of memory, manipulated via arithmetic or logic operations in the CPU, and the results copied back to some memory cells.
* Thus, as an instruction is executed, the *state* of the machine changes.
* Conventional programming languages can be viewed as abstractions of an underlying Von Neumann architecture.
* Conventional programming languages keep their computation model from the underlying Von Neumann architecture, but abstract away from the details of the individual steps of execution.
* Such a model consists of a sequential step-by-step execution of instructions which change the state of a computation by modifying a repository of values.
* Sequential step-by-step execution of language instructions reflects the sequential fetch and execution of machine instructions performed by hardware.
* Also, the variables of conventional programming languages, which can be modified by assignment statements, reflect the behaviour of the memory cells of the computer architecture.
* Conventional languages based on the Von Neumann computation model are often called *imperative languages*.
* Other common terms are *state-based languages*, or *statement-based languages*, or simply *Von Neumann languages*.
* Many kinds of abstractions were later invented by language designers, such as procedures and functions, data types, exception handlers, classes, concurrency features, etc.



* As suggested by Figure 2, language developers tried to make the level of programming languages higher, to make languages easier to use by humans, but still based the concepts of the language on those of the underlying Von Neumann architecture.
* Some programming languages, namely, *functional* and *logic languages*, have abandoned the Von Neumann computation model.
* Both paradigms are based on mathematical foundations rather than on the technology of the underlying hardware: the theory of recursive functions and mathematical logic, respectively.
* Hierarchy of paradigms are shown in fig3.



* Imperative, functional, and logic paradigms reflect the different underlying computation model of the language.
* The next level paradigms reflect the different organizational principles for program structuring supported by the language.

**Programming language qualities:-**

* Programming language is a tool for the development of software.
* The quality of the language must be related to the quality of the software.

**Software must be Reliable:-**

* Users should be able to rely on the software, i.e., the chance of failures due to faults in the program should be low.
* As far as possible, the system should be fault-tolerant; i.e., it should continue to provide support to the user even in the presenceof infrequent or undesirable events such as hardware or software failures.

**Software must be Maintainable:-**

* Existing software must be modified to meet new requirements. Also, because it is almost impossible to get the real requirements right in the first place, for such complex systems one can only hope to gradually evolve a system into the desired one.

**Software must execute Efficiency:-**

* Efficiency has always been a goal of any software system. This goal affects both the programming language (features that can be efficiently implemented on present day architectures) and the choice of algorithms to be used.
* These three requirements–reliability, maintainability, and efficiency–can be achieved by adopting suitable methods during software development, appropriate tools in the software development environment, and by certain characteristics of the programming language.
* Language issues that directly support these goals.

**Languages and Reliability:-**

* The reliability goal is promoted by several programming language qualities.

**Writability:-**

* It is a subjective criterion; we can agree that higher-level languages are more writable than lower-level languages (e.g., assembly or machine languages).
* The easier it is to concentrate on the problem-solving activity, the less error prone is program writing and the higher is productivity.

**Readability:-**

* It should be possible to follow the logic of the program and to discover the presence of errors by examining the program. Readability is also a subjective criterion that depends a great deal on matters of taste and style.

**Simplicity:-**

* A simple language is easy to master and allows algorithms to be expressed easily, in a way that makes the programmer self-confident.
* For example, Pascal is simpler, but less powerful than c++.

**Safety:-**

* The language should not provide features that make it possible towrite harmful programs. For example, a language that does notprovide goto statements or pointer variables eliminates two well-knownsources of danger in a program. Such features may causesubtle errors that are difficult to track during program development.

 **Robustness:-**

* The language supports robustness whenever it provides the ability to deal with undesired events (arithmetic overflows, invalid input, and so on).

**Languages and maintainability**

* Programming languages should allow programs to be easily modifiable. Readability and simplicity are obviously important in this context too. Two main features that languages can provide to support modification are factoring and locality.

**Factoring:-**

* This means that the language should allow programmers to factor related features into one single point. As a very simple example, if an identical operation is repeated in several points of the program, it should be possible to factor it in a routine and replace it by a routine call. In doing so, the program becomes more readable.

**Locality:-**

* This means that the effect of a language feature is restricted to a small, local portion of the entire program. Otherwise, if it extends to most of the program, the task of making the change can be exceedingly complex. For example, in abstract data type programming, the change to a data structure defined inside a class is guaranteed not affect the rest of the program as long as the operations that manipulate the data structure are invoked in the same way.

**Languages and Efficiency:-**

* The need for efficiency has guided language design from the beginning. Many languages have had efficiency as a main design goal, either implicitly or explicitly. For example, FORTRAN originally was designed for a specific machine (the IBM 704).
* The issue of efficiency has changed considerably, however. Efficiency is no longer measured only by the execution speed and space. The effort required to produce a program or system initially and the effort required in maintenance can also be viewed as components of the efficiency measure.
* One might be interested in developing software components that might be *reusable* in future similar applications. Or one might be interested in developing *portable* software (i.e., software that can be moved to different machines) to make it quickly available to different users.
* The implementation adverselyaffects efficiency if it does not take all opportunities into account in order tosave space and improve speed. For example, we will see that in general astatement like

x = fun (y) + z + fun (y);

in C cannot be optimized as

x = 2\* fun (y) + z

Which would cause just one call to function fun.

**A Brief historic perspective:-**

* The developments in language design by following the evolution of ideas and concepts from a historical perspective.
* The software development process originally consisted only of the implementation phase. In the early days of computing, the computer was used mainly in scientific applications. An application was programmed by one person.
* The problem to be solved (e.g., a differential equation) was well-understood. As a result, there was not much need for requirements analysis or design specification or even maintenance.
* A programming language, therefore, only needed to support one programmer, who was programming what would be by today's standards an extremely simple application.
* The desire to apply the computer in more and more applications led to its being used in increasingly less understood and more sophisticated environments.
* This, in turn, led to the need for “teams” of programmers and more disciplined approaches.
* The requirements and design phases, which up to then essentially were performed in one programmer's head, now required a team, with the results being communicated toother people.
* Because so much effort and money was being spent on thedevelopment of systems, old systems could not simply be thrown away when a new system was needed.
* Economic considerations forced people to enhance an existing system to meet the newly recognized needs. Also, program maintenance now became an important issue.
* System reliability is another issue that has gained importance gradually, because of two major factors.
* One factor is that systems are being developed for users with little or no computer background; these users are not as tolerant of system failures as the system developers.
* The second factor is that systems are now being applied in critical areas such as chemical or nuclear plants and patient monitoring, where system failures can be disastrous
* In order to ensure reliability, verification and validation became vital.

**Chapter 2**

**Syntax and semantics**

**Introduction:**

* A programming language is a formal notation for describing algorithms for execution by computer. Like all formal notations, a programming language has two major components: syntax and semantics
* Syntax rules describe the form of any legal program.
* It is a set of formal rules that specify the composition of programs from letters, digits, and other characters.
* For example, the syntax rules may specify that each open parenthesis must match a closed parenthesis in arithmetic expressions, and that any two statements must be separated by a semicolon.
* The semantic rules specify “the meaning” of any syntactically valid program written in the language.
* Such meaning can be expressed by mapping each language construct into a domain whose semantics is known.
* For example, one way of describing the semantics of a language is by giving a description of each language construct in English.
* Such a description, of course, suffers from the informality, ambiguity, and wordiness of natural language, but it can give a reasonably intuitive view of the language.
* Informal, description of semantics by specifying the behavior of an abstract processor that executes programs written in the language.
* This kind of semantic characterization of a language is called operational semantics.

**Language Definition:-**

* When you read a program, how do you know if it is well formed? How do you know what it means? How does a compiler know how totranslate the program? Any programming language must be defined in enough detail to enable these kinds of issues to be resolved.
* In general, two aspects of a language-programming or natural language-must are defined: syntax and semantics.

**Syntax:-**

* Syntax is described by a set of rules that define the form of a language: they define how sentences may be formed as sequences of basic constituents called words.
* Using these rules we can tell whether a sentence is legal or not.
* The syntax does not tell us anything about the content (or meaning) of the sentence–the semantic rules tell us that.
* As an example, C keywords (such as while, do, if, else...), identifiers, numbers, operators, are words of the language.
* The C syntax tells us how to combine such words to construct well-formed statements and programs.
* Words are not elementary. They are constructed out of characters belonging to an alphabet.
* Thus the syntax of a language is defined by two sets of rules: lexical rules and syntactic rules.
* Lexical rules specify the set of characters that constitute the alphabet of the language and the way such characters can be combined to form valid words.
* For example, Pascal considers lowercase and uppercase characters to be identical, but C and Ada consider them to be distinct.
* Thus, according to the lexical rules, “Memory” and “memory” refer to the same variable in Pascal, but to distinct variables in C and Ada.
* The lexical rules also tell us that <> (or ¦) is a valid operator in Pascal but not in C, where the same operator is represented by! =.
* Ada differs from both, since “not equal” is represented as /=; delimiter <> (called “box”) stands for an undefined range of an array index.
* ALGOL 60 was defined with a context-free grammar developed by John Backus.
* This method has become known as BNF or Backus Naur form (Peter Naur was the editor of the ALGOL 60 report.)
* BNF provides a compact and clear definition for the syntax of programming languages.

**Sidebar-start-1:-**

* EBNF is a meta-language.
* A meta-language is a language that is used to describe other languages. We describe EBNF first, and then we show how it can be used to describe the syntax of a simple programming language (Figure 5(a)).
* The symbols ::=, <, >, \*, +, (, ), and | are symbols of the meta language: they are meta symbols.
* A language is described in EBNF through a set of rules.
* For example, <program >:={<statement>\* } is a rule.
* The symbol "::=" stands for “is defined as”.
* The symbol “\*” stands for “an arbitrary sequence of the previous element”.
* Thus, the rule states that a <program> is defined as an arbitrary sequence of <statement> within brackets “{” and “}”.
* The entities inside the meta language brackets “<”, and “>” are called non terminals; an entity such as the “}” above is called a terminal.
* Terminals are what we have previously called words of the language being defined, whereas non terminals are linguistic entities tha­­­­­­­­­t are defined by other EBNF rules.
* In order to distinguish between Meta symbols and terminals, Figure 5 uses the convention that terminals are written in bold. To complete our description of EBNF, the Meta symbol “+” denotes one or more repetitions of the previous element (i.e., at least one element must be present, as opposed to “\*”).
* The Meta symbol “|” denotes a choice.
* For example, a <statement> is described in Figure 5(a) as either an<assignment>, a <conditional>, or a <loop>.



* The lexical rules, which describe how identifiers, numbers, and operators look like in our simple language are also described in EBNF, and shown in Figure 5(b).
* To do so, <operator>, <identifier>, and <number>, which are words of the language being defined, are detailed in terms of elementary symbols of the alphabet.
* Figure 6 shows the syntax diagrams for the simple programming language whose EBNF has been discussed above.
* Non terminals are represented by circles and terminals by boxes.
* The nonterminal symbol is defined with a transition diagram having one entry and one exit edge.



In conclusion, the syntactic description of a language has two primary uses:

* It helps the programmer know how to write a syntactically correct program
* It can be used to determine whether a program is syntactically correct. This is exactly what a compiler does.

**Abstract syntax, concrete syntax and pragmatics:**

* Some language constructs in different programming languages have the same conceptual structure but differ in their appearance at the lexical level.

 For example, the C fragment

While (x != y)

 {

 . . .

};

And the Pascal fragment

While x <> y do

Begin

 . . .

 End

 Can both be described by simple lexical changes in the EBNF rules of Figure 5?

* They differ from the simple programming language of Figure 5 only in the way statements are bracketed (begin ... end vs. {...}), the “not equal” operator (<> vs.!=), and the fact that the loop expression in C must be enclosed within parentheses.
* When two constructs differ only at the lexical level, we say that they follow the same abstract syntax, but differ at the concrete syntax level.

That is, they have the same abstract structure and differ only in lower-level details.

* Other languages, such as C or Pascal, allow brackets to be omitted in the case of single statements.
* For example, one may write:

While (x! = y) do x = y + 1;

* Pragmatically, however, this may be error prone.
* If one more statement needs to insert in the loop body, one should not forget to add brackets to group the statements constituting the body.
* Modula-2 adopts a good concrete syntax solution, by using the “end” keyword to terminate both loop and conditional statements.
* A similar solution is adopted by Ada.
* The following are Modula2 examples:

If x = y then if x = y then while x = y do

 . . . . . . . . .

End else end

 . . .

 End

* In all three fragments, the “...” part can be either a single statement or a sequence of statements separated by a semicolon.

**Semantics**

* Syntax defines well-formed programs of a language. Semantics defines the meaning of syntactically correct programs in that language.
* For example, the semantics of C help us determine that the declaration

int vector [10];

* Causes ten integer elements to be reserved for a variable named vector.
* The first element of the vector may be referenced by vector [0]; all other elements may be referenced by an index i, 0 ≤ i ≤ 9.
* As another example, the semantics of C states that the instruction

If (a > b) max = a; else max = b;

Means that the expression a > b must be evaluated, and depending on its value, one of the two given assignment statements is executed.

* Note that the syntax rules tell us how to form this statement–for example, where to put a “;”–and the semantic rules tell us what the effect of the statement is.

**Language processing:-**

* Machine languages are designed on the basis of speed of execution cost of realization, and flexibility in building new software layers upon them.
* On the other hand, programming languages often are designed on the basis of the ease and reliability of programming.
* A basic problem, then, is how a higher level language eventually can be executed on a computer whose machine language is very different and at a much lower level.
* There are generally two extreme alternatives for an implementation:

I)Interpreter II)Transmiter

**Interpretation:-**

In this solution, the actions implied by the constructs of the language are executed directly. Usually, for each possible action there exists a subprogram–written in machine language–to execute the action. Thus, interpretation of a program is accomplished by calling subprograms in the appropriate sequence





* More precisely, an interpreter is a program that repeatedly executes the following sequence.

Get the next statement;

Determine the actions to be executed;

Perform the actions;

* This sequence is very similar to the pattern of actions carried out by a traditional computer, that is:

-Fetch the next instruction (i.e., the instruction whose address is specified by the instruction pointer).

-Advance the instruction pointer (i.e., set the address of the instruction to be fetched next).

-Decode the fetched instruction.

-Execute the instruction.

* This similarity shows that interpretation can be viewed as a simulation, on a host computer, of a special-purpose machine whose machine language is the higher level language.

**Translation:**

* In this solution, programs written in a high-level language are translated into an equivalent machine-language version before being executed.
* This translation is often performed in several steps.
* Program modules might first be separately translated into relocatable machine code; modules of relocatable code are linked together into a single relocatable unit; finally, the entire program is loaded into the computer’s memory as executable machine code.
* The translators used in each of these steps have specialized names: compiler, linker (or linkage editor), and loader, respectively.
* In some cases, the machine on which the translation is performed (the host machine) is different from the machine that is to run the translated code (the target machine).
* This kind of translation is called cross-translation.
* Pure interpretation and pure translation are two ends of a continuous spectrum.
* In practice, many languages are implemented by a combination of the two techniques.
* A program may be translated into an intermediate code that is then interpreted.
* The intermediate code might be simply a formatted representation of the original program, with irrelevant information (e.g., comments and spaces) removed and the components of each statement stored in a fixed format to simplify the subsequent decoding of instructions.
* In this case, the solution is basically interpretive.
* Language Java is perhaps the best known and most promising example.
* Java is first translated to an intermediate code, called Java bytecode, which is interpreted in the client machine.
* Macro processing is a special kind of translation that may occur as the first step in the translation of a program.
* A macro is a named source text fragment, called the macro body.
* Through macro processing, macro names in a text are replaced by the corresponding bodies.
* In C, one can write macros, handled by a pre-processor, which generates source C code through macro expansion.
* For example, one can use a macro to provide a symbolic name for a constant value, as in the following fragment.

 #define upper\_limit 100

Sum = 0;

For (index = 0; index < upper\_limit; index++)

{

 += a [index];

 }

**The Concept of Binding:-**

* Programs deal with entities, such as variables, routines, statements, and so on.
* Program entities have certain properties called attributes.
* For example, a variable has a name, a type, a storage area where its value is stored; a routine has a name, formal parameters of a certain type, certain parameter-passing conventions; a statement has associated actions.
* Attributes must be specified before an entity is elaborated.
* Specifying the exact nature of an attribute is known as binding.
* For each entity, attribute information is contained in a repository called a descriptor.
* Binding is a central concept in the definition of programming language semantics.
* Programming languages differ in the number of entities with which they can deal, in the number of attributes to be bound to entities, in the time at which such bindings occur (binding time), and in the stability of the binding (i.e., whether an established binding is fixed or modifiable).
* A binding that cannot be modified is called static.
* A modifiable binding is called dynamic.
* Some attributes may be bound at language definition time, others at program translation time (or compile time), and others at program execution time (or run time).

The following is a (non-exhaustive) list of binding examples:

* **Language definition time binding:** In most languages (including FORTRAN, Ada, C, and C++) the type "integer" is bound at language definition time to its well-known mathematical counterpart, i.e., to a set of algebraic operations that produce and manipulate integers;
* **Language implementation time binding:** In most languages (including FORTRAN, Ada, C, and C++) a set of values is bound to the integer type at language implementation time.That is, the language definition states that type "integer" must be supported and the language implementation binds it to a memory representation, which–in turn–determines the set of values that are contained in the type.
* **Compile time (or translation time) binding:** Pascal provides a predefined definition of type integer, but allows the programmer to redefine it. Thus type integer is bound a representation at language implementation time, but the binding can be modified at translation time.
* **Execution time (or run time) binding:** In most programming languages variables are bound to a value at execution time, and the binding can be modified repeatedly during execution.
* In the first two examples, the binding is established before run time and cannot be changed thereafter.
* This kind of binding regime is often called static.
* The term static denotes both the binding time (which occurs before the program is executed) and the stability (the binding is fixed).
* Conversely, a binding established at run time is usually modifiable during execution. The fourth example illustrates this case. This kind of binding regime is often called dynamic.
* The concepts of binding, binding time, and stability help clarify many semantic aspects of programming languages.

**Variables:**

* Conventional computers are based on the notion of a main memory consisting of elementary cells, each of which is identified by an address. The contents of a cell are an encoded representation of a value. A value is a mathematical abstraction; its encoded representation in a memory cell can be read and (usually) modified during execution. Modification implies replacing one encoding with a new encoding.
* Formally, a variable is a 5-tuple <name, scope, type, l\_value, r\_value>, where
* Name is a string of characters used by program statements to denote the variable;
* Scope is the range of program instructions over which the name is known;
* Type is the variable’s type;
* L\_value is the memory location associated with the variable;
* R\_value is the encoded value stored in the variable’s location.

**Name and scope:**

* A variable’s name is usually introduced by a special statement, called declaration and, normally, the variable’s scope extends from that point until some later closing point, specified by the language.
* The scope of a variable is the range of program instructions over which the name is known.
* Program instructions can manipulate a variable through its name within its scope. We also say that a variable is visible under its name within its scope, and invisible outside it.



* The declaration int x, y; makes variables named x and y visible throughout program main. The program contains an internal block, which groups a declaration and statements.
* The declaration int temp; appearing in the block makes a variable named temp visible within the inner block, and invisible outside.
* Thus, it would be impossible to insert temp as an argument of operation printf.
* Variables can be bound to a scope either statically or dynamically.
* Static scope binding defines the scope in terms of the lexical structure of a program, that is, each reference to a variable can be statically bound to a particular (implicit or explicit) variable declaration by examining the program text, without executing it.
* Static scope binding rules are adopted by most programming languages, such as C, as we saw in the previous example.
* Dynamic scope binding defines the scope of a variable's name in terms of program execution.
* Typically, each variable declaration extends its effect over all the instructions executed thereafter, until a new declaration for a variable with the same name is encountered during execution.



* If the language follows dynamic scoping, an execution of block A followed by block C would make variable x in the assignment in block C to refer to x declared in block A.
* Instead, an execution of block B followed by block C would make variable x in the assignment in block C refer to x declared in block B.
* Thus, name x in block C refers either to the x declared in A or the one declared in B, depending on the flow of control followed during execution.
* We define the type of a variable as a specification of the set of values that can be associated with the variable, together with the operations that can be legally used to create, access, and modify such values. A variable of a given type is said to be an instance of the type.
* When the language is defined, certain type names are bound to certain classes of values and sets of operations. For example, type integer and its associated operators are bound to their mathematical counterpart. Values and operations are bound to a certain machine representation when the language is implemented. The latter binding may also restrict the set of values that can be represented, based on the storage capacity of the target machine.
* In some languages, the programmer can define new types by means of type declarations. For example, in C one can writeTypedef int vector [10];
* This declaration establishes a binding–at translation time–between the type name vector and its implementation (i.e., an array of 10 integers, each accessible via an index in the subrange 0. .9).
* As a consequence of this binding, type vector inherits all the operations of the representation data structure (the array); thus, it is possible to read and modify each component of an object of type vector by indexing within the array.
* There are languages that support the implementation of user-defined types (usually called abstract data types) by associating the new type with the set of operations that can be used on its instances; the operations are described as a set of routines in the declaration of the new type.
* The declaration of the new type has the following general form, expressed in C-like syntax:

Typedef new\_type\_name

{

Data structure representing objects of type new\_type\_name;

Routines to be invoked for manipulating data objects of type new\_type\_name;

}

* To provide a preview of concepts and constructs that will be discussed at length in this text, Figure 8 illustrates an example of an abstract data type (a stack of characters) implemented as a C++ class1.
* The class defines the hidden data structure (a pointer s to the first element of the stack, a pointer top to the most recently inserted character, and an integer denoting the maximum size) and five routines to be used for manipulating stack objects.
* Routines stack\_of\_char and ~stack\_of\_char are used to construct and destruct objects of type stack\_of\_char, respectively.
* Routine push is used to insert a new element on top of a stack object. Routine pop is used to extract an element from a stack object. Routine length yields the current size of a stack object.



**l\_value:**

* The l\_value of a variable is the storage area bound to the variable during execution. The lifetime, or extent, of a variable is the period of time in which such binding exists.
* The storage area is used to hold the r\_value of the variable. We will use the term data object (or simply, object) to denote the pair <l\_value, r\_value>.
* The action that acquires a storage area for a variable–and thus establishes the binding–is called memory allocation. The lifetime extends from the point of allocation to the point in which the allocated storage is reclaimed (memory deallocation).
* In some languages, for some kinds of variables, allocation is performed before run time and storage is only reclaimed upon termination (static allocation).
* In other languages, it is performed at run time (dynamic allocation), either upon explicit request from the programmer via a creation statement or automatically, when the variable's declaration is encountered and reclaimed during execution.

**r\_value:**

* The r\_value of a variable is the encoded value stored in the location associated with the variable (i.e., l\_value). The encoded representation is interpreted according to the variable's type. For example, a certain sequence of bits stored at a certain location would be interpreted as an integer number if the variable’s type is int; it would be interpreted as a string if the type is an array of char.
* L\_values and r\_value are the main concepts related to program execution. Program instructions access variables through their l\_value and possibly modify their r\_value. The terms l\_value and r\_value derive from the conventional form of assignment statements, such as x = y; in C.
* The variable appearing at the left-hand side of the assignment denotes a location (i.e., its l\_value is meant).
* The variable appearing at the right-hand side of the assignment denotes the contents of a location (i.e., its r\_value is meant).
* Whenever no ambiguity arises, we use the simple term “value” of a variable to denote its r\_value.
* The binding between a variable and the value held in its storage area is usually dynamic; the value can be modified by an assignment operation.
* An assignment such as b = a; causes a's r\_value to be copied into the storage area referred to by b’s l\_value. That is, b’s r\_value changes.
* Some conventional languages, however, allow the binding between a variable and its value to be frozen once it is established.
* The resulting entity is, in every respect, a user-defined symbolic constant. For example, in C one can write

Const float pi = 3.1415;

And then use pi in expressions such as

Circumference= 2 \* pi \* radius;

Variable pi is bound to value 3.1416 and its value cannot be changed; that is, the translator reports an error if there is an assignment to pi. A similar effect can be achieved in Pascal.

**Routines:-**

* Programming languages allow a program to be composed of a number of units, called routines.
* Assembly language subprograms, FORTRAN subroutines, Pascal and Ada procedures and functions, C functions are well-known examples of routines.
* In the existing programming language world, routines usually come in two forms: procedures and functions.
* Functions return a value; procedures do not. Some languages, e.g., C and C++, only provide functions, but procedures are easily obtained as functions returning the null value void. Figure 9 shows the example of a C function definition.



* Like variables, routines have a name, scope, type, l\_value, and r\_value. A routine name is introduced in a program by a routine declaration.
* Usually the scope of such name extends from the declaration point on to some closing construct, statically or dynamically determined, depending on the language.
* For example, in C a function declaration extends the scope of the function till the end of the file in which the declaration occurs.
* The header of the routine defines the routine’s name, its parameter types, and the type of the returned value (if any).
* In brief, the routine’s header defines the routine type. In the example of Figure 9, the routine’s type is: routine with one int parameter and returning int
* A routine call is type correct if it conforms to the corresponding routine type. For example, the call

i = sum (10); /\* i is declared as an int \*/ would be correct with respect to the function definition of Figure 9. Instead, the call

i = sum (5.3); would be incorrect.

* Some languages support the notion of a “pointer to a routine” and provide a way of getting a routine l\_values, which can be assigned (as a r\_value) to a pointer.
* For example, the following Cstatement declares a pointer PS to a function with an int parameter and returning an int:

int (\*ps) (int);

The following assignment

ps = & sum;

Makes ps point to the l\_values of the previously defined routine sum. A call may then be issued via ps as in the following example:

int i = (\*ps) (5);

 /\* this invokes the r\_value of the routine that is currently referred to by ps \*/

* Some languages (like Pascal, Ada, C, and C++) distinguish between declaration and definition of a routine. A routine declaration introduces the routine’s header, without specifying the body.
* The name is visible from the declaration point on, up to the scope end. The definition specifies both the header and the body.
* The distinction between declaration and definition is necessary to allow routines to call themselves in a mutual recursion scheme, as illustrated by the following fragment.



**Generic routines:-**

* Routines factor a code fragment that is executed at different points of the program in a single place and assign it a name.
* The fragment is then executed through invocation, and customized through parameters.
* Often, however, similar routines must be written several times, because they differ in some detail aspects that cannot be factored through parameters.
* For example, if a program needs both a routine to sort arrays of integers and arrays of strings, two different routines must be written, one for each parameter type, even if the abstract algorithm chosen for the implementation of the sort operation is the same in both cases.
* Figure 10 shows an example of a generic swap routine in C++. Generic C++ units are called templates.
* More on generics, their effect on reusability of program components, and the features offered by C++ in support of these concepts.



**Aliasing and Overloading:**

* In programs, names are used to denote variables and routines. The language uses special names (denoted by operators), such as + or \* to denote certain predefined operations. So far, we implicitly assumed that at each point in a program a name denotes exactly one entity, based on the scope rules of the language.
* Since names are used to identify the corresponding entity, the assumption of unique binding between a name and an entity would make the identification unambiguous. This restriction, however, is almost never true for existing programming languages.

For example, in C one can write the following fragment:

int i, j, k;

float a, b, c;

 ...

i = j + k;

a = b + c;

* In the example, operator + in the two instructions of the program denotes two different entities. In the first expression, it denotes integer addition; in the second, it denotes floating-point addition.
* Although the name is the same forthe operator in the two expressions, the binding between the operator and the corresponding operation is different in the two cases, and the exact binding can be established at compile time, since the types of the operands allow for the disambiguation.
* We can generalize the previous example by introducing the concept of overloading. A name is said to be overloaded if more than one entity is bound to the name at a given point of a program and yet the specific occurrence of the name provides enough information to allow the binding to be uniquely established. In the previous example, the types of the operands to which + is applied allows for the disambiguation.
* As another example, if the second instruction of the previous fragment would be changed to

a = b + c + b ( );

the two occurrences of name b would (unambiguously) denote, respectively, variable b and routine b with no parameters and returning a float value (assuming that such routine is visible by the assignment instruction). Similarly, if another routine named b, with one int parameter and returning a float value is visible, instruction

a = b ( ) + c + b (i);

would unambiguously denote two calls to the two different routines.

* Aliasing is exactly the opposite of overloading. Two names are aliases if they denote the same entity at the same program point. This concept is especially relevant in the case of variables. Two alias variables share the same data object in the same referencing environment. Thus modification of the object under one name would make the effect visible, maybe unexpectedly, under the other.
* Although examples of aliasing are quite common, one should be careful since this feature may lead to error prone and difficult to read programs. An example of aliasing is shown by the following C fragment:

int i;

int fun (int& a);

{ . . .

 a = a + 1;

printf ("%d", i);

 . . .

}

main ( )

 {

 . . .

 x = fun (i);

. . .

}

When function f is executed, names i and a in fun denote the same data object. Thus an assignment to a would cause the value of i printed by fun to differ from the value held at the point of call.

Aliasing can easily be achieved through pointers and array elements. For example, the following assignments in C

int x = 0;

int\* i = &x;

int\* j = &x;

would make \*i, \*j, and x aliases.

* In its basic form, SIMPLESEM consists of an instruction pointer (the reference to the instruction currently being executed), a memory, and a processor. The memory is where the instructions to be executed and the data to be manipulated are stored.
* For simplicity, we will assume that these two parts are stored into two separate memory sections: the code memory (C) and the data memory (D). Both C's and D's initial address is 0 (zero), and both programs and data are assumed to be stored from the initial address. The instruction pointer (ip) is always used to point to a location in C; it is initialized to 0.
* We use the notation D[X] and C[X] to denote the values stored in the X-th cell of D and C, respectively.
* Thus X is an l\_value and D[X] is the corresponding r\_value. Modification of the value stored in a cell is performed by instructionset, with two parameters: the address of the cell whose contents is to be set, and the expression evaluating the new value. For example, the effect on the data memory of instruction

set 10, D[20]

 is to assign the value stored at location 20 into location 10.



 **Execution-time structure:**

* Our discussion will show that languages can be classified in several categories, according to their execution-time structure.

**Static languages:**

* Exemplified by the early versions of FORTRAN and COBOL, these languages guarantee that the memory requirements for any program can be evaluated before program execution begins.
* Therefore, all the needed memory can be allocated before program execution. Clearly, these languages cannot allow recursion, because recursion would require an arbitrary number of unit instances, and thus memory requirements could not be determined before execution.

**Stack-based languages:**

* Historically headed by ALGOL 60 and exemplified by the family of so-called Algol-like languages, this class is more demanding in terms of memory requirements, which cannot be computed at compile time.
* However, their memory usage is predictable and follows a last-in-first-out discipline: the latest allocated activation record is the next one to be deallocated.
* It is therefore possible to manage SIMPLESEM’s D store as a stack to model the execution time behavior of this class of languages.
* Notice that an implementation of these languages need not use a stack (although, most likely, it will): deallocation of discarded activation records can be avoided if store can be viewed as unbounded.
* In other terms, the stack is part of the semantic model we provide for the language; strictly speaking, it is not part of the semantics of the language.

**Fully dynamic languages:**

* These languages have an unpredictable memory usage; i.e, data are dynamically allocated only when they are needed during execution.
* The problem then becomes how to manage memory efficiently. In particular, how can unused memory be recognized and reallocated, if needed.

**C1: A language with only simple statements**

* Let us consider a very simple programming language, called C1, which can be seen as a lexical variant of a subset of C, where we only have simple types and simple statements (there are no functions).
* Let us assume that the only data manipulated by the language are those whose memory requirements are known statically, such as integer and floating point values, fixed-size arrays, and structures.
* The entire program consists of a main routine (main ( )), which encloses a set of data declarations and a set of statements that manipulate these data.
* For simplicity, input/output is performed by invoking the operations get and print to read and write values, respectively.



A C1 program is shown in Figure 12 and its straightforward SIMPLESEM representation before the execution starts is shown in Figure 12. The D portion shows the activation record of the main program, which contains space for all variables that appear in the program. The C portion shows the SIMPLESEM code.

