A] SOFTWARE DEVELOPMENT PROCESS:

1. 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.
2. The software is said to have a life cycle 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 a fragment of specification, a test plan or a user's manual.
3. In the traditional waterfall model of the software life cycle, the development process is a sequential combination of phases, each having well-identified starting and ending points, with clearly identifiable deliverables to the next phase.
4. Each step may identify deficiencies in the previous one, which then must be repeated.
5. **A sample software development process based on the waterfall model may be comprised of the following phases:**
6. **Requirement analysis and specification:** - The purpose of this phase is to identify and document the exact requirements for the system. These requirements are developed jointly by users and software developers. The success of a system is measured by how well the software mirrors these stated requirements, how well the requirements mirror the users' perceived needs, and how well the users' perceived needs reflect the real needs. The result of this phase is a requirements document stating what the system should do, along with users' manuals, feasibility and cost studies, performance requirements, and so on. The requirements document does not specify how the system is going to meet its requirements.

7. **Software design and specification: -** Starting with the requirements document, software designers design the software system. The result of this phase is a system design specification document identifying all of the modules comprising the system and their interfaces. Separating requirements analysis from design is an instance of a fundamental“what/how” dichotomy that we encounter quite often in computer science. The general principle involves making a clear distinction between what the problem is and how to solve the problem. In this case, the requirements phase attempts to specify what the problem is. There are usually many ways that the requirements can be met. The purpose of the design phase is to specify a particular software architecture that will meet the stated requirements. The design method followed in this step can have a great impact on the quality of the resulting application; in particular, its understandability and modifiability. It can also affect the choice of the programming language to be used in system implementation.
8. **Implementation (coding):-** The system is implemented to meet the design specified in the previous phase. The design specification, in this case, states the “what”; 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. The result is a fully implemented and documented system.
9. **Verification and validation: -** This phase assesses the quality of the implemented system, which is then delivered to the user. Note that this phase should not be concentrated at the end of the implementation step, but should occur in every phase of software development to check that intermediate deliverables of the process satisfy their objectives. For example, one should check that the design specification document is consistent with the requirements

which, in turn, should match the user's needs. These checks are accomplished by answering the following two questions:

"Are we building the product right?"

"Are we building the right product?"

Two specific kinds of assessment performed during implementation are module testing and integration testing. Module testing is done by each programmer on the module he or she is working on to ensure that it meets its interface specifications. Integration testing is done on a partial aggregation of modules; it is basically aimed at uncovering intermodule inconsistencies.

10. **Maintenance:** - 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 (e.g., the operating system of the target machine). These changes are referred to as maintenance.The importance of this phase can be seen in the fact that maintenance costs are typicallyat least as large as those of all the other steps combined.

Programming languages are used only in some phases of the developmentprocess. They are obviously used in the implementation phase, when algorithmsand data structures are defined and coded for the modules that form theentire application.

B] LANGUAGE AND SOFTWARE DEVELOPMENT ENVIRONMENTS:

1. 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.
2. These tools have evolved gradually, as the need for automation has been recognized. Nowadays, one can normally use an interactive editor to create a program and the file system to store it for future use. When needed, several

previously created and (possibly) compiled programs may be linked to produce an executableprogram. A debugger is commonly used to locate faults in a program and eliminate them.

3. These computer-aided program development tools have increased programming productivity by reducing the chances of errors. Yet, as we have seen, software development involves much more than programming.
4. In order to increase the productivity of software development, computer support is needed for all of its phases. By a software development environment we mean an integrated set of tools and techniques that aids in the development of software.
5. The environment is used in all phases of software development: requirements, design, implementation, verification and validation, and maintenance.
6. 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.
7. The environment keeps track of the requirements as they are being developed and updated, and guards against incompleteness or inconsistency.
8. It also provides facilities to validate requirements against the customer's expectations, for example by providing ways to simulate or animate them. The environment ensures the currency of the documentation as changes are being made to the requirements.
9. Following the completion of the requirements, system designers, interacting with the environment, develop an initial system design and gradually refine it, that is, they specify the needed modules and the module interfaces.
10. Test data may also be produced at this stage. The implementers then undertake to implement the system based on the design.

11. The environment provides support for these phases by automating some development steps, by suggesting reuse of existing design and implementation components taken from a library, by recording the relationships among all of the artifacts, so that one can trace the effect of a change in–say–the requirements document to changes in the design document and in the code.
12. 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.
13. For this ideal scenario to work, all of the tools must be compatible and integrated with tools used in the other phases. For example, the programming language must be compatible with the design methods supported by the environment at the design stage and with the design notations used to document designs.
14. As other examples, the editor used to enter programs might be sensitive to the syntax of the language, so that syntax errors can be caught before they are even entered rather than later at compile time. A facility for test data generation might also be available for programs written in the language.

C] LANGUAGE AND SOFTWARE DESIGN METHODS:

1. The relationship between software design methods and programming languages is an important one. Some languages provide better support for some design methods than others.
2. Older languages, such as FORTRAN, were not designed to support specific design methods. For example, the absence of suitable high-level control structures in early FORTRAN makes it difficult to systematically design algorithms in a top-down fashion.

3. Conversely, Pascal was designed with the explicit goal of supporting top down program development and structured programming.
4. In both languages, the lack of constructs to define modules other than routines makes it difficult to decompose a software system into abstract data types.
5. To understand the relationship between a programming language and a design method, it is important to realize that programming languages may enforce a certain programing style, often called a programming paradigm.
6. For example, as we will see, Smalltalk and Eiffel are object-oriented languages. They enforce the development of programs based on object classes as the unit of modularization. Similarly, FORTRAN and Pascal, as originally defined, are procedural languages. They enforce the development of programs based on routines as the unit of modularization.
7. Languages enforcing a specific programming paradigm can be called paradigm-oriented. In general, there need not be a one-to-one relationship between paradigms and programming languages. Some languages, in fact, are paradigm-neutral and support different paradigms.
8. Design methods, in turn, guide software designers in a system's decomposition into logical components which, eventually, must be coded in a language. Different design methods have been proposed to guide software designers.
9. For example, a procedural design method guides designers in decomposing a system into modules that realize abstract operations that may be activated by other procedural modules.
10. An object-oriented method guides in decomposing a system into classes of objects. If the design method and the language paradigm are the same, or the language is paradigm-neutral, then the design abstractions can be directly mapped into program components.

11. Otherwise, if the two clash, the programming effort increases. As an example, an object-oriented design method followed by implementation in FORTRAN increases the programming effort.
12. In general, we can state that the design method and the paradigm supported by the language should be the same. If this is the case, there is a continuum between design and implementation. Most modern high-level programming languages, in fact, can even be used as design notations.
13. For example, a language like Ada or Eiffel can be used to document a system's decomposition into modules even at the stage where the implementation details internal to the module are still to be defined.

D] LANGUAGES AND COMPUTER ARCHITECTURE:

1. Design methods influence programming languages in the sense of establishing requirements for the language to meet in order to better support software development.
2. Computer architecture has exerted influence from the opposite direction in the sense of restraining language designs to what can be implemented efficiently on current machines.
3. Accordingly, languages have been constrained by the ideas of Von Neumann, because most current computers are similar to theoriginal Von Neumann architecture (Figure 1).

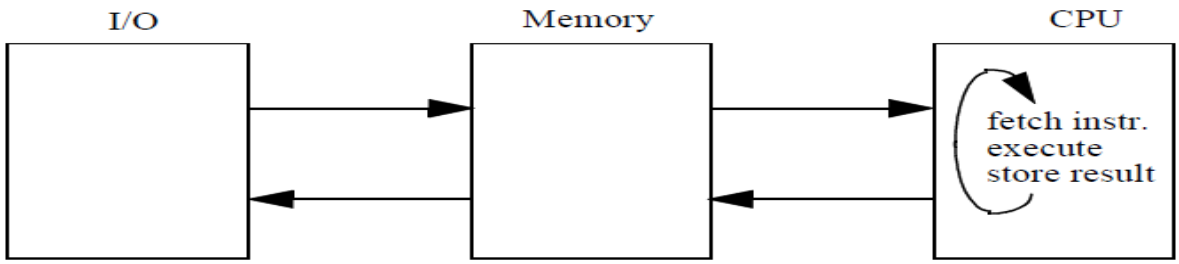

FIGURE 1. A Von Neumann computer architectur

4. 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.
5. 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.
6. 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. For this reason, they are called Von Neumann languages. An abstraction of a phenomenon is a model which Ignores irrelevant details and highlights the relevant aspects.
7. Conventional programming languages keep their computation model from the underlying Von Neumann architecture, but abstract away from the details of the indivual steps of execution.
8. 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.
9. Also, the variables of conventional programming languages, which can be modified by assignment statements, reflect the behavior of the memory cells of the computer architecture. Conventional languages based on the Von Neumann computation model are often called imperative languages.
10. Other common terms are state-based languages, or statement-based languages, or simply Von Neumann languages. The historical developments of imperative languages have gone through increasingly higher levels of abstractions.

11. Many kinds of abstractions were later invented by language designers, such as procedures and functions, data types, exception handlers, classes, concurrencyFeatures, etc.
12. 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.

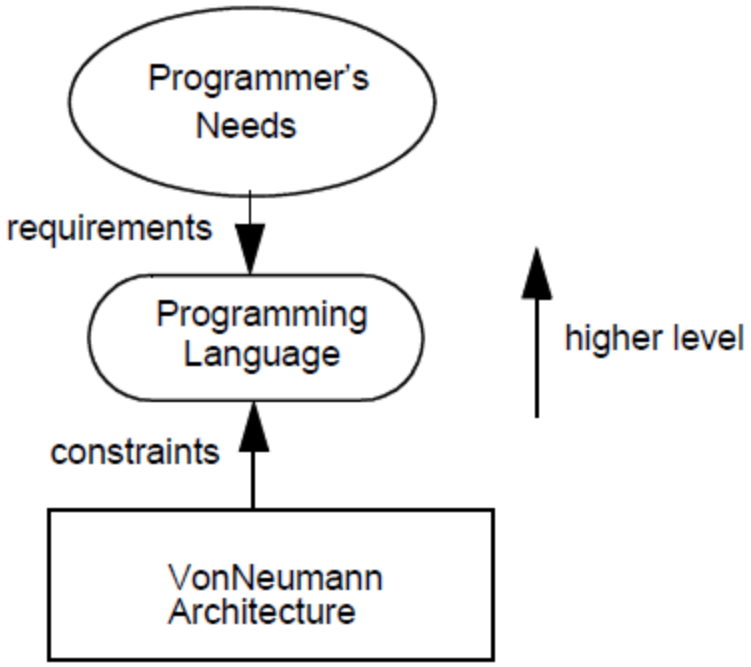

FIGURE 2. Requirements and constraints on a language

13. Some programming languages, namely, functional and logic languages have abandoned the Von Neumann computation model.
14. Both paradigms are basedon mathematical foundations rather than on the technology of the underlyinghardware: the theory of recursive functions and mathematical logic, respectively.
15. The conceptual integrity of these languages, however, is in conflictwith the goal of an efficient implementation. This is not unexpected, sinceconcerns of the underlying architecture did not permeate the design of suchlanguages in

the first place. To improve efficiency, some imperative featureshave been introduced in most existing unconventional languages.

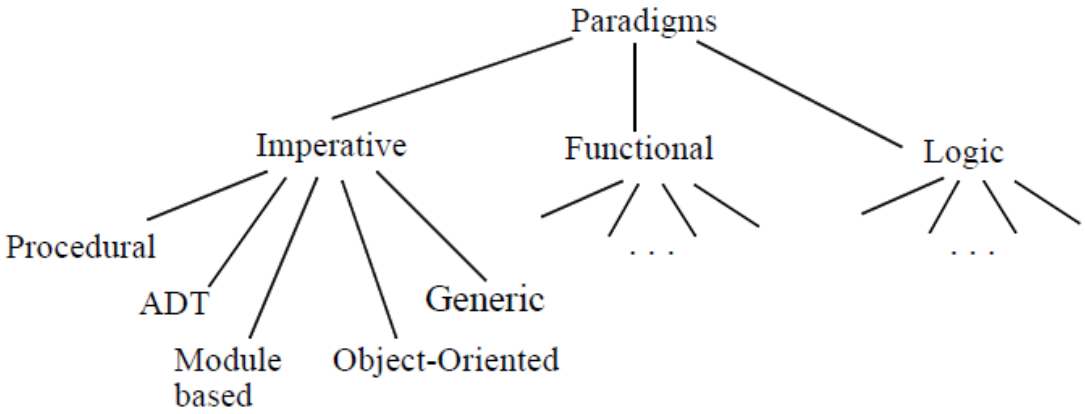

FIGURE 3. Hierarchy of paradigms

E] PROGRAMMING LANGUAGE QUALITIES:

1. In order to understand that, we should keep in mind that a programming languageis a tool for the development of software. Thus, ultimately, 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 presence of infrequent or undesirable events such as hardware or software failures. The reliability requirement has gained importance as software has been called upon to accomplish increasingly complicated and often critical tasks.
   - ***Software must be maintainable.***Again, as software costs have risen and increasingly complex software systems have been developed, economic considerations have reduced the possibility of throwing away existing software and developing similar applications from scratch. 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 efficiently.***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-dayarchitectures) and the choice of algorithms to be used. Althoughthe cost of hardware continues to drop as its performancecontinues to increase (in terms of both speed and space), the needfor efficient execution remains because computers are being appliedin increasingly more demanding applications.

F] LANGUAGES AND RELIABILITY:

1. The reliability goal is promoted by several programming language qualities. A non-exhaustive list is provided hereafter. Most of them, unfortunately, are based on subjective evaluation, and are difficult to state in a precise–let alone, quantitative–way. In addition, they are not independent concepts: in some cases they are overlapping, in others they are conflicting.

- ***Writability*.** It refers to the possibility of expressing a program in a way that is natural for the problem. The programmer should not be distracted by details and tricks of the language from the more important activity of problem solving. Even though 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). For example, an assembly language programmer is often distracted by the addressing mechanisms needed to access certain data, such as the positioning of index registers, and so on. 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. The provision of specific constructs to define new operations (via routines) and new data types, which keep the definition of such concepts separate from the rest of the program that may use them, greatly enhance readability.
- ***Simplicity*.** A simple language is easy to master and allows algorithms to be expressed easily, in a way that makes the programmer self-confident. Simplicity can obviously conflict with power of the language. For example, Pascal is simpler, but less powerful thanC++.
- **Safety.** The language should not provide features that make it possible to write harmful programs. For example, a language that does not provide goto statements nor pointer variables eliminates two well-known sources of danger in a program. Such features may cause subtle errors that are difficult to track during program development, and may manifest themselves unexpectedly in the delivered software. Again, features that decrease the dangers may also reduce power and flexibility.
- **Robustness.** The language supports robustness whenever it provides the ability to deal with undesired events (arithmetic overflows, invalid input, and so on). That is, such events can be trapped and a suitable response can be programmed to respond to their occurrence. In this way, the behavior of the system becomes predictable even in anomalous situations.

G] LANGUAGES AND MAINTAINABILITY:

1. 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 (especially if we give a meaningful name to subprograms) and more easily modifiable (a change to the fragment is localized to the routine's body). As another example, several programming languages allow constants to be given symbolic names. Choosing an appropriate name for a constant promotes readability of the program (e.g., we may use pi instead of 3.14). Moreover, a futureneed to change the value would necessitate a change only in the definition of the constant, rather than in every use of the constant.
   - ***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 operationsthat manipulate the data structure are invoked in the sameway. Factoring and locality are strongly related concepts. In fact, factoring promotes locality, in that changes may apply only to the factored portion. Consider for example the case in which we wish to change the

number of digits used to represent pi in order to improve accuracy of a geometrical computation.

H] LANGUAGES AND EFFICIENCY:

1. 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.
2. For example, FORTRAN originally was designed for a specific machine (the IBM 704). Many of FORTRAN's restrictions, such as the number of array dimensions or the form of expressions used as array indices, were based directly on what could be implemented efficiently on the IBM 704.
3. The issue of efficiency has changed considerably, however. Efficiency is no longer measured only by the execution speed and space. The effort required producing a program or system initially and the effort required in maintenance can also be viewed as components of the efficiency measure.
4. In other words, in some cases one may be more concerned with productivity of the software development process than the performance of the resulting products. Moreover, productivity concerns can span over several developments than just one.
5. That is, 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, even if an ad hoc optimized version for each machine would be faster. Efficiency is often a combined quality of both the language and its implementation.
6. The language adversely affects efficiency if it disallows certain optimizations to be applied by the compiler. The implementation adversely

affects efficiency if it does not take all opportunities into account in order to save space and improve speed.

7. For example, we will see that in general a statement likeas different example, the language can affect efficiency by allowing multithreaded concurrent computations. An implementation adversely affects efficiency if–say–it does not reuse memory space after it is released by the program.
8. Finally, a language that allows visibility of the underlying size of memory or access to–say–the way floating-point numbers are stored would impede portability, and thus saving in the effort of moving software to different platforms.

I] A BRIEF HISTORICAL PERSPECTIVE AND EARLY HIGH LEVEL LANGUAGES:

1. 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.
2. 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.
3. A programming language, therefore, only needed to support one programmer, who was programming what would be by today's standards an extremely simple application.
4. 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.

5. 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.
6. Because so much effort and money was being spent on the development 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.
7. Also, program maintenance now became an important issue. System reliability is another issue that has gained importance gradually, because of two major factors.
8. 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.
9. 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.
10. The first attempts towards definition of high-level languages date back to the 1950s. Language design was viewed as a challenging compromise betweenthe users' needs for expressiveness and the machine's limited power. However, hardware was very expensive and execution efficiency concerns were the dominant design constraint.
11. The most important products of this historical phase were FORTRAN, ALGOL 60, and COBOL. FORTRAN and ALGOL 60 were defined as tools for solving numerical scientific problems, that is, problems involving complex computations on relatively few and simple data.

12. COBOL was defined asa tool for solving business data-processing problems, that is, problemsinvolving simple computations on large amounts of data (e.g., a payroll application).
13. These languages are among the major achievements in the whole history ofcomputer science, because they were able to prove that the idea of a higher-levellanguage was technically sound and economically viable.
14. Besides that,each of these languages has brought up a number of important concepts. Forexample, FORTRAN introduced modularity via separately developed andcompiled subprograms and possible sharing of data among modules via a global(COMMON) environment.
15. ALGOL 60 introduced the notion of blockstructure and recursive procedures. COBOL introduced files and data descriptions,and a very preliminary notion of programming in quasi-natural language.
16. An even more convincing proof of the validity of these languages is that,apart from ALGOL 60 which did not survive but spawned into their languages,they are still among the most widely used languages in practice.

J] A BIRD'S EYE VIEW OF PROGRAMMING LANGUAGE CONCEPTS:

Using a simple C/C++program as an example, we look at the kinds of facilities that a programminglanguage must support and the different ways that languages go about providingthese facilities.

A simple program:

1. We are interested in the kinds of things one can do with programminglanguages, rather than the specifics of a given program. What are

theinherent capabilities and shortcomings of different programming languages? What makes one language fundamentally different from another and whatmakes one language similar to another, despite apparent differences?

2. Below Example We have divided the program into three parts, separated from each other by single blank lines.
3. The first section consists of two “#include” statements; the second part consists of three “declaration” statements; and, finally, the third part is the actual code of a function called main that supposedly “does the work”.
4. We can say that the first part is used to organize the structure of the program, in this case in terms of the various files that constitute the program.
5. The second part defines the environment in which the program will work by declaring some entities that will be used by the program in this file. These declarations may import entities defined in other files.
6. For example, the line extern phone_list pb; indicates that the variable pb of type phone_list is being used in this program but has been created elsewhere. The third part deals with the actual computation.
7. This is the part we most often associate with a program. It contains the program’s data and algorithms. Some of the data and processing in this part may use the entities defined in the environment established in the second part.
8. For example, in Figure 4 the routines insert and lookup are used in the main program. Another example is the output statement: cout << “Enter 1 to insert, 2 to lookup: \n”; which uses cout, the standard output device defined in the standard input-output library iostream.h included in the first line of the program.
9. Even in this short, simple program, we see that a programming language providesmany different kinds of facilities.

```
#include <iostream.h>
#include "phone.h"

extern phone_list pb;
void insert();
number lookup ();

main()
{
    int request;

    cout << "Enter 1 to insert, 2 to lookup: \n";
    cin >> request;
    if (request == 1)
        insert();
    else if (request == 2)
        cout << lookup();
    else
        {cout << "invalid request.\n";
        exit (1);
        }
}
```

10.\ **FIGURE 4.** A phone-list program

K] SYNTAX AND SEMANTICS:

1. Any programming language specifies a set of rules for the form of valid programs in that language.
2. For example, in the program of Figure 4, we see that many lines are terminated by a semicolon. We see that there are some special characters used, such as { and }.
3. We see that every if is followed by a parenthesized expression. The syntax rules of the language state how to form expressions, statements, and programs that look right.

4. The semantic rules of the language tell us how to build meaningful expressions, statements, and programs. For example, they might tell us that before using the variable request in the if-statement, we must declare that variable.
5. They also tell us that, the declaration of a variable such as request causes storage to be reserved for the variable. On the other hand, the presence of the extern in the declaration of the variable pb indicates that the storage is reserved by some other module and not this one.
6. Characters are the ultimate syntactic building blocks. Every program is Formed by placing characters together in some well-defined order. The syntacticrules for orming programs are rather straightforward. The semantic building blocks and rules, on the other hand, are more intricate. Indeed, most of the deep differences among the various programming languages stem from their different semantic underpinnings.

L] LANGUAGE DEFINITION:

1. 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 to translate the program? Any programming language must be defined in enough detail toenable these kinds of issues to be resolved.
2. More specifically, a language definitionshould enable a person or a computer program to determine (1)whether a purported program is in fact valid, and (2) if the program is valid,what its meaning or effect is. In general, two aspects of a language-programmingor natural language-must be defined: syntax and semantics.

M] SYNTAX:

1. Syntax is described by a set of rules that define the form of a language: theydefine how sentences may be formed as sequences of basic constituentscalled words.
2. 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 thesentence–the semantic rules tell us that.
3. As an example, C keywords (such aswhile, do, if, else...), identifiers, numbers, operators ... are words of the language.The C syntax tells us how to combine such words to construct well-formedstatements and programs.
4. Words are not elementary. They are constructed out of characters belongingto an alphabet. Thus the syntax of a language is defined by two sets of rules:lexical rules and syntactic rules.
5. Lexical rules specify the set of charactersthat constitute the alphabet of the language and the way such characters canbe 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 lexicalrules, “Memory” and “memory” refer to the same variable in Pascal, butto distinct variables in C and Ada.
6. The lexical rules also tell us that <> (or ¦) isa valid operator in Pascal but not in C, where the same operator is representedby! =. Ada differs from both, since “not equal” is represented as /=; delimiter<> (called “box”) stands for an undefined range of an array index.
7. The distinction between syntactic and lexical rules is somewhat arbitrary.They both contribute to the “external” appearance of the language. Often, wewill use the terms “syntax” and “syntactic rules” in a wider sense thatincludes lexical components as well.
8. How does one define the syntax of a language? Because there are an infinitenumber of legal and illegal programs in any useful language, we clearly

cannotenumerate them all. We need a way to define an infinite set using a finitedescription.

9. FORTRAN was defined by simply stating some rules in English.ALGOL 60 was defined with a context-free grammar developed by JohnBackus.
10. This method has become known as BNF or Backus Naur form (PeterNaur was the editor of the ALGOL 60 report.) BNF provides a compact andclear definition for the syntax of programming languages.

N] ABSTRACT SYNTAX, CONCRETE SYNTAX AND PRAGMATICS:

1. Some language constructs in different programming languages have the sameconceptual structure but differ in their appearance at the lexical level. For example, the C fragmentcan both be described by simple lexical changes in the EBNF rules of Figure 5

```
while (x!= y)
{
. . .
};
```

and the Pascal fragment

```
while x <> y do
begin
. . .
End
```

(a) Syntax rules

<program>::={ <statement>* }
<statement>::=<assignment> | <conditional> | <loop>
<assignment>::=<identifier> = <expr> ;
<conditional>::=**if** <expr> { <statement> + } |
if <expr> { <statement> + } **else** { <statement> + }
<loop>::=**while** <expr> { <statement> + }
<expr> ::=<identifier> | <number> | (<expr>) | <expr> <operator> <expr>

(b) Lexical rules

<operator>::= + | - | * | / | = | ¦ | < | > | ≤ | ≥
<identifier>::= <letter> <ld>*
<ld>::= <letter> | <digit>
<number>::= <digit>+
<letter>::= **a** | **b** | **c** | . . . | **z**

FIGURE 5. EBNF definition of a simple programming language (a) syntax rules, (b) lexical rules

2. They differ from the simple programming language of Figure 5 only in theway statements are bracketed (begin ... end vs. {...}), the “not equal” operator(<> vs.! =), and the fact that the loop expression in C must be enclosed withinparentheses.
3. When two constructs differ only at the lexical level, we say thatthey follow the same abstract syntax, but differ at the concrete syntax level.
4. That is, they have the same abstract structure and differ only in lower-leveldetails.Although, conceptually, concrete syntax may be irrelevant, pragmatically itmay affect usability of the language and readability of programs.
5. For example,symbol ¦ is obviously more readable than! =. As another example, thesimple language of Figure 5 requires the body of while statements and thebranches of conditionals to be bracketed by {and}.

6. Other languages, such asC or Pascal, allow brackets to be omitted in the case of single statements. Forexample, one may write:

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

7. Pragmatically, however, this may be error prone. If one more statement needsto insert in the loop body, one should not forget to add brackets to group thestatements constituting the body.
8. Modula-2 adopts a good concrete syntaxsolution, by using the "end" keyword to terminate both loop and conditionalstatements. A similar solution is adopted by Ada. The following are Modula 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 asequence of statements separated by a semicolon.

O] 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 elementsmay 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 syntaxrules tell us how to form this statement–for example, where to put a ";"–andthe semantic rules tell us what the effect of the statement is.

The semantics of the language might require such expressions to deliver atruth value (TRUE or FALSE, not–say–an integer value). In many cases, suchrules that further constrain syntactically correct programs can be verifiedbefore a program's execution: they constitute static semantics, as opposed todynamic semantics, which describes the effect of executing the different constructsof the programming language. In such cases, programs can be executedonly if they are correct both with respect to syntax and to staticsemantics. In this section, we concentrate on the latter; i.e., any reference tothe term "semantics" implicitly refers to "dynamic semantics".**Axiomatic semantics** views a program as a state machine. Programming languageconstructs are described by describing how their execution causes astate change.

A state is described by a first-order logic predicate whichdefines the property of the values of program variables in that state. Thus themeaning of each construct is defined by a rule that relates the two statesbefore and after the execution of that construct.

Denotational semantics associates each language statement with a functiondsem from the state of the program before the execution to the state after execution.

The state (i.e., the values stored in the memory) is represented by afunction mem from the set of program identifiers ID to values. Thus Denotational semantics differs from axiomatic semantics in the way states aredescribed (functions vs. predicates). For simplicity, we assume that valuescan only belong to type integer.

P] LANGUAGE PROCESSING

1. Although in theory it is possible to build special-purpose computers to execute directly programs written in any particular language, present-day

computers directly execute only a very low-level language, the machine language.

2. 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.
3. A basic problem, then, is how a higherlevel language eventually can be executed on a computer whose machine language is very different and at a much lower level.
4. There are generally two extreme alternatives for an implementation: interpretation and translation.

➢ **Interpretation:**

1. In this solution, the actions implied by the constructs of the language are executed directly (see Figure 7). 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.

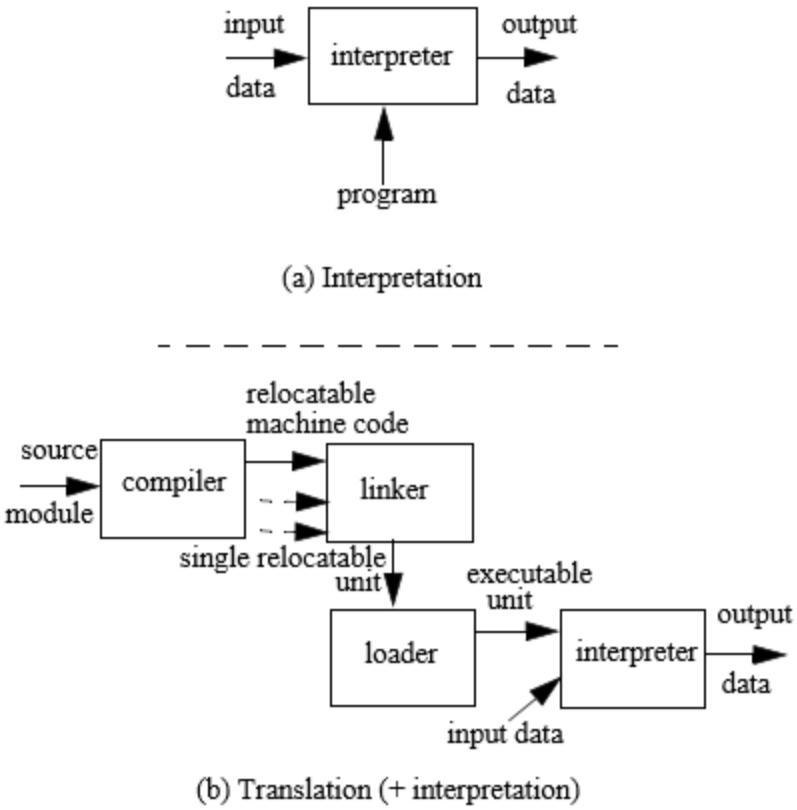

FIGURE 7. Language processing by interpretation (a) and translation (b)

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:**

1. In this solution, programs written in a high-level language are translated into an equivalent machine-language version before being executed.
2. This translation is often performed in several steps (see Figure 7). 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.
3. The translators used in each of these steps have specialized names: compiler, linker (or linkage editor), and loader, respectively.
4. 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).
5. This kind of translation is called cross-translation. Crosstranslators offer the only viable solution when the target machine is a specialpurpose processor rather han a general-purpose one that can support a translator.
6. 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.
7. 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.

Alternatively, the intermediate code might be the (low-level) machine code for a virtual machine that is to be later interpreted by software.

8. This solution, which relies more heavily on translation, can be adopted for generating portable code, that is, code that is more easily transferable to different machines than machine language code.
9. For example, for portability purposes, one of the best known initial implementations of a Pascal compiler was written in Pascal and generated an intermediate code, called Pcode. The availability of a portable implementation of the language contributed to the rapid diffusion of Pascal in many educational environments.
10. More recently, with the widespread use of Internet, code portability became a primary concern for network application developers. A number of language efforts have recently been undertaken with the goal of supporting code mobility over a network.
11. 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.
12. In a purely interpretive solution, executing a statement may require a fairly complicated decoding process to determine the operations to be executed and their operands.
13. Compilers and interpreters differ in the way they can report on run-time errors. Typically, with compilation, any reference to the source code is lost in the generated object code.
14. If an error is generated at run-time, it may be impossible to relate it to the source language construct being executed. This is why run-time error messages are often obscure and almost meaningless to the programmer.
15. On the opposite, the interpreter processes source statements, and can relate a run-time error to the source statement being executed. For these

reasons, certain programming environments contain both an interpreter and a compiler for a given programming language.

16. The interpreter is used while the program is being developed, because of its improved diagnostic facilities.
17. The compiler is then used to generate efficient code, after the program has been fully validated. Macro processing is a special kind of translation that may occur as the first step in the translation of a program.
18. 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.
19. In C, one can write macros, handled by a preprocessor, 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++)
{
    sum += a [index];
}
```

➢ **The concept of binding:**

1. Programs deal with entities, such as variables, routines, statements, and so on. Program entities have certain properties called attributes.
2. 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.

3. Binding is a central concept in the definition of programming language semantics.
4. 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).
5. A binding that cannot be modified is called static. A modifiable binding is called dynamic.
6. 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).
7. The following is a (nonexhaustive) 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.

8. 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).
9. Conversely, a binding established at run time is usually modifiable during execution. The fourth example illustrates this case.
10. This kind of binding regime is often called dynamic. There are cases, however, where the binding is established at run time, and cannot be changed after being established.
11. An example is a language providing (read only) constant variables that are initialized with an expression to be evaluated at run time.
12. The concepts of binding, binding time, and stability help clarify many semantic aspects of programming languages. In the next section we will use these concepts to illustrate the notion of a variable.

Q] Variables:

1. 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 is an encoded representation of a value.
2. 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.

3. With a few exceptions, programming languages can be viewed as abstractions, at different levels, of the behavior of such conventional computers.
4. In particular, they introduce the notion of variables as an abstraction of the notion of memory cells. the variable name as an abstraction of the address. and the notion of assignment statements as an abstraction of the destructive modification of a cell.
5. 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 :**

1. 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.
2. 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.
3. We also say that a variable is visible under its name within its scope, and invisible outside it. Different programming languages adopt different rules for binding variable names to their scope.

For example, consider the following example of a C program:

```
# include <stdio.h>
main ( )
{
    int x, y;
    scanf ("%d %d", &x, &y);
        /*two decimal values are read and stored in the l_values of x and y */
    {
        /*this is a block used to swap x and y*/
        int temp;
        temp = x;
        x = y;
        y = temp;
    }
    printf ("%d %d", x, y);
}
```

4. 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.
5. 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.
6. In the example, if the inner block declares a new local variable named x, the outer variable named x would not be visible in it. The inner declaration masks the outer variable. The outer variable, however, continues to exist even though it is invisible. It becomes visible again when control exits the inner block.
7. 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.
8. Static scope binding rules are adopted by most programming languages, such as C, as we saw in the previous example.

9. 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. APL, LISP (as originally defined), and SNOBOL4 are examples of languages with dynamic
scope rules.

As an example, consider the following program fragment written in a C-like notation.

```
{
    /* block A */
    int x;
    . . .
}
. . .
{
    /* block B */
    int x;
    . . .
}
. . .
{
    /* block C */
    . . .
    x = ...;
    . . .
}
```

10. 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.
11. 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.

12. Dynamic scope rules look quite simple and are rather easy to implement, but they have disadvantages in terms of programming discipline and efficiency of implementation.
13. Programs are hard to read because the identity of the particular declaration to which a given variable is bound depends on the particular point of execution, and so cannot be determined statically.

- **Type:**

1. 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.
2. 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.
3. 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.
4. The latter binding may also restrict the set of values that can be represented, based on the storage capacity of the target machine.
5. In some languages, the programmer can define new types by means of type declarations.
6. For example, in C one can write typedef int vector [10];
7. 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).
8. 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.

9. 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.
10. The declaration of the new type has the following general form, expressed in C-like syntax:
11. typedef new_type_name
12. { data structure representing objects of type new_type_name; routines to be invoked for manipulating data objects of type new_type_name; }
13. 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.
14. 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.
15. 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.

```
class stack_of_char{
    int size;
    char* top;
    char* s;
public:
    stack_of_char (int sz) {
        top = s = new char [size =sz];
    }
    ~stack_of_char ( ) {delete [ ] s;}
    void push (char c) {*top++ = c;}
    char pop ( ) {return *--top;}
    int length ( ) {return top - s;}
};
```

FIGURE 8. User-defined type in C++

16. Traditional languages, such as FORTRAN, COBOL, Pascal, C, C++, Modula-2, and Ada bind variables to their type at compile time, and the binding cannot be changed during execution.
17. This solution is called static typing. In these languages, the binding between a variable and its type is specified by a variable declaration. For example, in C one can write:int x, y; char c; By declaring variables to belong to a given type, variables are automatically protected from the application of illegal (or nonsensical) operations.
18. For example, in Ada the compiler can detect the application of the illegal assignment I:= not A, if I is declared to be an integer and A is a boolean. Through this check, the compiler watches for violations to static semantics concerning variables and their types. The ability to perform checks before the program is executed (static type checking) contributes to early error detection and enhances program reliability.
19. Assembly languages, LISP, APL, SNOBOL4, and Smalltalk are languages that establish a (modifiable) run-time binding between variables and their type.
20. This binding strategy is called dynamic typing. Dynamically typed variables are also called polymorphic variables (literally, from ancient Greek, “multiple shape”) variables.
21. In most assembly languages, variables are dynamically typed. This reflects the behavior of the underlying hardware, where memory cells and registers can contain bit strings that are interpreted as values of any type.
22. For example, the bist string stored in a cell may be added to the bit string stored in a register using integer addition. In such a case, the bit strings are interpreted as integer values.

23. In other languages, the type of a variable depends on the value that is dynamically associated with it. For example, having assigned an integer value to a variable, such value cannot be treated as if it were–say–a string of characters.
24. That is, the binding is still dynamic, but once a value is bound to a variable, an implicit binding with a type is also established, and the binding remains in place until a new value is assigned.
25. As another example, in LISP, variables are not explicitly declared; their type is implicitly determined by the value they currently hold during execution. The LISP function CAR applied to a list L yields the first element of L, which may be an atom (say, an integer) or a list of–say–strings, if L is a list of lists.
26. If the element returned by CAR is bound to a variable, the type of such variable would be an integer in the former case, a string list in the latter. If such value is added to an integer value, the operation would be correct in the former case, but would be illegal in the latter.
27. Moreover, suppose that the value of the variable is to be printed. The effect of the print operation depends on the type that is dynamically associated with the variable.
28. It prints an integer in the former case; a list of strings in the latter. Such a print routine, which is applicable to arguments of more than one type, is called a polymorphic routine.

➢ **l_value :**

1. The l_value of a variable is the storage area bound to the variable during execution.
2. 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>.

3. The action that acquires a storage area for a variable–and thus establishes the binding–is called memory allocation.
4. The lifetime extends from the point of allocation to the point in which the allocated storage is reclaimed (memory deallocation).
5. 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.
6. In most cases, the lifetime of a program variable is a fraction of the program's execution time. It is also possible, however, to have persistent objects.
7. A persistent object exists in the environment in which a program is executed and its lifetime has no a-priori relation with any given program's execution time. Files are an example of persistent objects.
8. Once they are created, they can be used by different program activations, and different activations of the same program, until they are deleted through a specific command to the operating system. Similarly, persistent objects can be stored in a database, and made visible to a programming language through a specific interface.

➢ **r_value:**

1. The r_value of a variable is the encoded value stored in the location associated with the variable (i.e., its l_value).
2. 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.

3. l_values and r_values are the main concepts related to program execution. Program instructions access variables through their l_value and possibly modify their r_value.
4. 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).
5. 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.
6. 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.
7. 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. This, however, is true only for conventional imperative languages, like FORTRAN, C, Pascal, Ada, and C++.
8. Functional and logic programming languages treat variables as their mathematical counterpart: they can be bound to a value by the evaluation process, but once the binding is established it cannot be changed during the variable's lifetime.
9. 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
10. const float pi = 3.1415; and then use pi in expressions such as

11. circumference= 2 * pi * radius;
12. 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.
13. Pascal and C differ in the time of binding between the const variable and its value, although binding stability is the same for both languages. In Pascal the value is provided by an expression that must be evaluated at compile time; i.e., binding time is compile time.
14. The compiler can legally substitute the value of the constant for its symbolic name in the program. In C and Ada the value can be given as an expression involving other variables and constants: consequently, binding can only be established at run time, when the variable is created.

R] References and unnamed variables:

1. Some languages allow unnamed variables that can be accessed through the r_value of another variable. Such a r_value is called a reference (or a pointer) to the variable.
2. In turn, the reference can be the r_value of a named variable, or it can be the r_value of a referenced variable. Thus, in general, an object can be made accessible via a chain of references (called access path) of arbitrary length.
3. If A = <A_name, A_scope, A_type, A_l_value, A_r_value> is a named variable, object <A_l_value, A_r_value> is said to be directly accessible through name A_name in A_scope, with an access path of length 0.
4. If B= <--, --, --, B_l_value, B_r_value>, where -- stands for the “don’t care value”, is a variable and B_l_value = A_r_value, object <B_l_value, B_r_value> is said to be accessible through name A_name in A_scope indirectly, with an access path of length 1.

5. Similarly, one can define the concept of an object indirectly accessible through a named variable, with an access path of length i, i>1.
6. For example, in Pascal we can declare type PI (pointer to an integer):
7. type PI = ^ integer; We can then allocate an unnamed integer variable and have it pointed by a variable pxi of type PI: new (pxi);
8. In order to access the unnamed object referenced by pxi, it is necessary to use the dereferencing operator ^, which can be applied to a pointer variable to obtain its r_value, i.e., the l_value of the referenced object. For example, the value of the unnamed variable can be set to zero by writing:

```
pxi^ := 0;
```

The unnamed variable can also be made accessible indirectly through a "pointer to a pointer to an integer", as sketched below:

```
type PPI = ^PI;
var ppxi: PPI;
. . .
new (ppxi);
^ppxi := pxi;
```

Here the r_value of variable ppxi is the l_value of an unnamed variable, whose r_value is the l_value of variable x.

Other languages, like C, C++, and Ada, allow pointers to refer to named variables. For example, the following C fragment:

```
int x = 5;
int* px;
px = &x;
```

generates an integer object whose r_value is 5, directly accessible through a variable named x and indirectly accessible (with an access path of length 1) through px, declared as a pointer to integer. This is achieved by assigning to px the value of the address of x (i.e., &x). Indirect access to x is then made possible by dereferencing px. In C (and C++) the dereferencing operator is denoted by *. For example, the following C instruction

```
int y = *px;
```

assigns to y the r_value of the variable pointed at by px (i.e., 5).

Two variables are said to *share* an object if each has an access path to the object. A shared object modified via a certain access path makes the modification visible through all possible access paths. Sharing of objects is used to improve efficiency, but it can lead to programs that are hard to read, because the value of a variable can be modified even when its name is not used. In the previous C example, if one writes:

```
*px = 0;
```

the contents of the location pointed at by px becomes 0 and, because of sharing, the value of x becomes 0 too.

S] Routines:

1. Programming languages allow a program to be composed of a number of units, called routines. The neutral term “routine” is used in this chapter in order to provide a general treatment that enlightens the important principles that are common to most programming languages, without committing to any specific feature offered by individual languages.
2. Routines can be developed in a more or less independent fashion and can sometimes be translated separately and combined after translation. Assembly language subprograms, FORTRAN subroutines, Pascal and Ada procedures and functions, C functions are well-known examples of routines.
3. In this chapter we will review the main syntactic and semantic features of routines, and in particular the mechanisms that control the flow of execution among routines and the bindings established when a routine is executed.
4. Other, more general kinds of units, such as Ada packages, Modula-2 modules, and C++ classes will be described elsewhere.
5. In the existing programming language world, routines usually come in two forms: procedures and functions.
6. 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.

```
/* sum is a function which computes the sum
of the first n positive integers, 1 + 2 + ... + n;
parameter n is assumed to be positive */
int sum (int n)
{
    int i, s;
    s = 0;
    for (i = 1; i <= n ; ++i)
        s+= i;
    return s:
}
```

FIGURE 9. A C function definition

7. Like variables, routines have a name, scope, type, l_value, and r_value.
8. 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.
9. For example, in C a function declaration extends the scope of the function till the end of the file in which the declaration occurs. Routine activation is achieved through a routine call, which names the routine and specifies the parameters on which the routine operates. Since a routine is activated by call, the call statement must be in the routine's scope.
10. Besides having their own scope, routines also define a scope for the declarations that are nested in them. Such local declarations are only visible within the routine.
11. Depending on the scope rules of the language, routines can also refer to nonlocal items (e.g., variables) other than those declared locally. Nonlocal items that are potentially referenced by every unit in the program are called global items.

12. 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 type can be precisely defined by the concept of signature.
13. The signature specifies the types of parameters and the return type. A routine fun which behaves like a function, with input parameters of types T1, T2, . . ., Tn and returning a value of type R, can be defined by the following signature:

- **Generic routines:**

1. 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.
2. Often, however, similar routines must be written several times, because they differ in some detail aspects that cannot be factored through parameters.
3. 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.
4. Generic routines, as offered by some programming languages, provide a solution to this problem. In this section we provide a view of generic routines as they appear in languages like C++ or Ada.
5. A generic routine can be made parametric with respect to a type. In the previous example, the routine would be generic with respect to the type of the array elements.
6. Type parameters in a generic routine, however, differ from conventional parameters, and require a different implementation scheme.

7. A generic routine is a template from which the specific routine is generated through Instantiation, an operation that binds generic parameters to actual parameters at compile time.
8. Such binding can be obtained via macro processing, which generates a new instance (i.e., an actual routine) for each type parameter. Other implementation schemes, however, are also possible.

```
template <class T>  void swap (T& a , T& b)
/* the function does not return any value; it is generic with respect to type T;
a and b refer to the the same locations as the actual parameters;
swap interchanges the two values*/
{
    T temp = a;
    a = b;
    b = temp;
}
```

FIGURE 10. A generic routine in C++

T] More on scopes: aliasing and overloading:

1. As our discussion so far emphasized, a central issue of programming language semantics has to do with the conventions adopted for naming. In programs, names are used to denote variables and routines.
2. 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.
3. 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.
4. For example, in C one can write the following fragment:

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
```

5. 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.
6. Although the name is the same for the 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.
7. We can generalize the previous example by introducing the concept of overloading.
8. 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.
9. 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).
10. 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.
11. 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.

12. 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.

U] An abstract semantic processor:

1. To describe the operational semantics of programming languages, we introduce a simple abstract processor, called SIMPLESEM, and we show how language constructs can be executed by sequences of operations of the abstract processor.
2. In this section, we provide the main features of SIMPLESEM; additional details will be introduced incrementally, as additional language features are introduced.
3. In its basic form, SIMPLESEM consists of an instruction pointer (the reference to the instruction currently being executed), a memory, and a processor.
4. 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.
5. 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 instruction set, 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.

6. Input/output in SIMPLESEM is achieved quite simply by using the set instruction and referring to the special registers read and write, which provide for communication of the SIMPLESEM machine with the outside world.
7. For example, set 15, read means that the value read from the input device is to be stored at location 15; set write, D[50] means that the value stored at location 50 is to be transferred to the output device.
8. We are quite liberal in the way we allow values to be combined in expressions; for example, D[15]+D[33]*D[41] would be a an acceptable expression, and set 99, D[15]+D[33]*D[41] would be an acceptable instruction to modify the contents of location 99.
9. As we mentioned, ip is SIMPLESEM's instruction pointer, which is initialized to zero at each new execution and automatically updated as each instruction is executed.
10. The machine, in fact, operates by executing the following steps repeatedly, until it encounters a special halt instruction:
    - ✓ Get the current instruction to be executed (i.e., C[ip]);
    - ✓ 2. Increment ip;
    - ✓ 3. Execute the current instruction.
11. Notice, however, that certain programming language instructions might modify the normal sequential control flow, and this must be reflected by SIMPLESEM. In particular, we introduce the following two instructions: jump and jumpt.
12. The former represents an unconditional jump to a certain instruction. For example, jump 47 forces the instruction stored at address 47 of C to be the next instruction to be executed; that is, it sets ip to 47.

13. The latter represents a conditional jump, which occurs if an expression evaluates to true. For example, in:
    - ✓ jumpt 47, D[3] > D[8] the jump occurs only if the value stored in cell 3 is greater than the value stored in cell 8.
    - ✓ SIMPLESEM allows indirect addressing. For example:
    - ✓ set D[10], D[20] assigns the value stored at location 20 into the cell whose address is the value stored at location 10. Thus, if value 30 is stored at location 10, the instruction modifies the contents of location 30. Indirection is also possible for jumps. For example:
    - ✓ jump D[13] jumps to the instruction stored at location 88 of C, if 88 is the value stored at location 13.
14. SIMPLESEM, which is sketched in Figure 11, is quite simple. It is easy to understand how it works and what the effects of executing its instructions are.
15. In other terms, we can assume that its semantics is intuitively known; it does not require further explanations that refer to other, more basic concepts. The semantics of programming languages can therefore be described by rules that specify how each construct of the language is translated into a sequence of SIMPLESEM instructions.
16. Since SIMPLESEM is perfectly understood, the semantics of newly defined constructs becomes also known. As we will see, however, SIMPLESEM will also be enriched as new programming language concepts are introduced. This will be done in this book incrementally, as we address the semantics of new concepts.

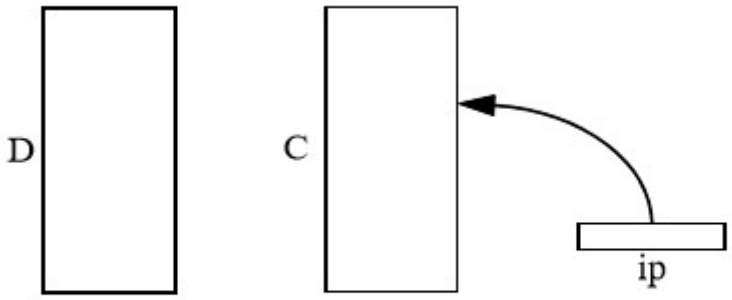

FIGURE 11. The SIMPLESEM machine

V] Run-time structure:

1. In this section we discuss how the most important concepts related to the execution-time processing of programming languages may be explained using SIMPLESEM.
2. We will proceed gradually, from the most basic concepts to more complex structures that reflect what is provided by modern general-purpose programming languages. We will move through a hierarchy of languages that are based on variants of the C programing language. They are named C1 through C5.
3. Our discussion will show that languages can be classified in several categories, according to their execution-time structure.
4. 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.
5. 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.

6. (As we will see later, the implementation is not required to perform the memory allocation statically. The semantics of the language, however, give the implementer the freedom to make that choice.)
7. 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.
8. It is therefore possible to manage SIMPLESEM's D store as a stack to model the executiontime 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.
9. Fully dynamic languages These languages have un unpredictable memory usage; i.e, data are dynamically allocated only when they are needed during execution.
10. The problem then becomes how to manage memory efficiently. In particular, how can unused memory be recognized and reallocated, if needed. To indicate that store D is not handled according to a predefined policy (like a FIFO policy for a stack memory), the term "heap" is traditionally used.