I.B.TECH III-ISEM(CSE)

Principles of Programming Language (15A05504)

UNIT I

Unit 1: Introduction: Software Development Process, Language and Software Development Environments, Language and Software Design Models, Language and Computer Architecture, Programming Language Qualities, A brief Historical Perspective.
Syntax and Semantics: Language Definition, Language Processing, Variables, Routines, Aliasing and Overloading, Run-time Structure.

Unit 2: Structuring the data: Built-in types and primitive types, Data aggregates and type constructors, User-defined types and abstract data types, Type Systems, The type Structure of representative languages, Implementation Models

Unit 3: Structuring the Computation: Expressions and Statements, Conditional Execution and Iteration, Routines, Exceptions, Pattern Matching, Non determinism and Backtracking, Event-driven computations, Concurrent Computations
Structuring the Program: Software Design Methods, Concepts in Support of Modularity, Language Features for Programming in the Large, Generic Units

Unit 4: Object-Oriented Languages: Concepts of Object-oriented Programming, Inheritances and the type system, Object-oriented features in programming languages

Unit 5: Functional Programming Languages: Characteristics of imperative languages, Mathematical and programming functions, Principles of Functional Programming, Representative Functional Languages, Functional Programming in C++
Logic and Rule-based Languages: —What versus —how: Specification versus implementation, Principles of Logic Programming, PROLOG, Functional Programming versus Logic Programming, Rule-based Languages.

TEXT BOOKS:

1. Programming Language Concepts‖, Carlo Ghezzi, Mehdi Jazayeri, WILEY Publications. Third Edition, 2014

REFERENCE BOOKS

1. Concepts of Programming Languages, Tenth Edition, Robert W. Sebesta, Pearson Education.
2. Programming Languages Principles and Paradigms, Second Edition, Allen B. Tucker, Robert E. Noonan, McGraw Hill Education.
3. Introduction to Programming Languages, Aravind Kumar Bansal, CRC Press

1Q. Explain about Features and Applications of SDLC?

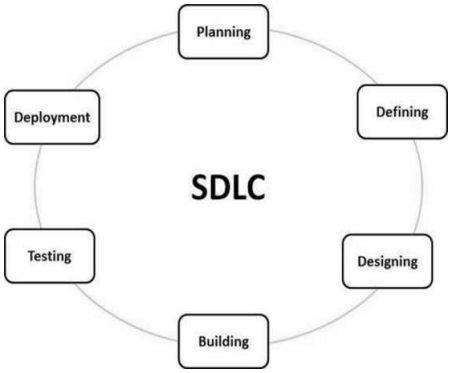
Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality software's. The SDLC aims to produce a high-quality software that meets customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as Software Development Process.
- SDLC is a framework defining tasks performed at each step in the software development process.
- ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

What is SDLC?

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much difficulty.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages in SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS. This can be accomplished by a concept of “Are we building the product right?” “Are we building the right product?”

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

SDLC Models

There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development Process Models". Each process model follows a Series of steps to ensure success in the process of software development.

Following are the most important and popular SDLC models followed in the industry −

- Waterfall Model
- Iterative Model
- Spiral Model
- V-Model
- Big Bang Model

Other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

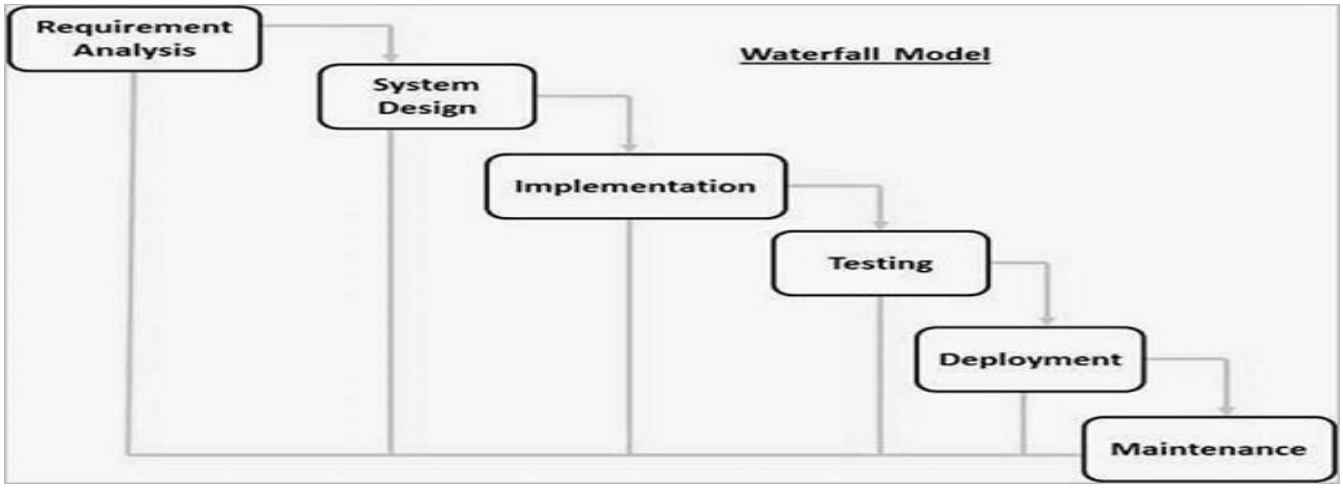
2Q. EXPLAIN DETIAL ABOUT SOFTWAE DESIGN MODELS?

1. WATERFALL MODEL:

The Waterfall Model was the first Process Model. It is also referred to as a **linear-sequential life cycle model**. The Waterfall model is the earliest SDLC approach that was used for software development. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

Waterfall Model - Design

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Waterfall Model - Application

Every software developed is different and requires a suitable SDLC approach based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Waterfall Model - Advantages

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows −

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

Waterfall Model - Disadvantages

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

2. Iterative Model

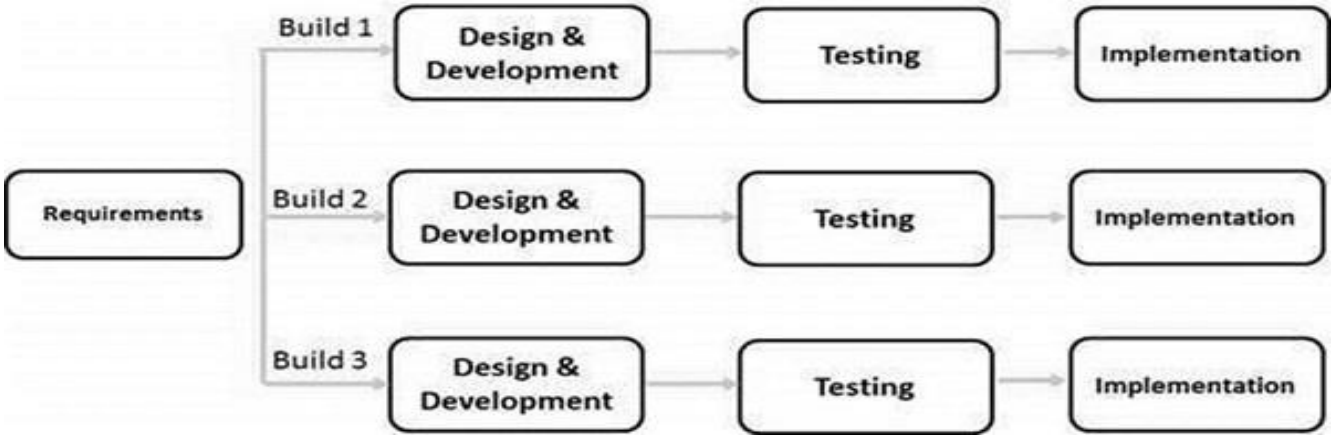
In the Iterative model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to USE.

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which is then reviewed to identify further requirements. This process is then repeated, producing a new version of the software at the end of each iteration of the model.

Iterative Model - Design

Iterative process starts with a simple implementation, At each iteration, design modifications are made and new functional capabilities are added. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

The following illustration is a representation of the Iterative and Incremental model −



Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach."

In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

Iterative Model - Application

This model is most often used in the following scenarios −

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill sets are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

Iterative Model - Pros and Cons

The advantage of this model is that there is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

The advantages of the Iterative and Incremental SDLC Model are as follows −

- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging during smaller iteration is easy.
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone.
- Easier to manage risk - High risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment.
- Risk analysis is better.

- It supports changing requirements.
- Initial Operating time is less.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early which facilitates customer evaluation and feedback.

The disadvantages of the Iterative and Incremental SDLC Model are as follows –

- More resources may be required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- More management attention is required.
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.
- Not suitable for smaller projects.
- Management complexity is more.
- End of project may not be known which is a risk.
- Highly skilled resources are required for risk analysis.
- Projects progress is highly dependent upon the risk analysis phase.

3. Spiral Model

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

Spiral Model - Design

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.

Identification

This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.

This phase also includes understanding the system requirements. At the end of the spiral, the product is deployed in the identified market.

Design

The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

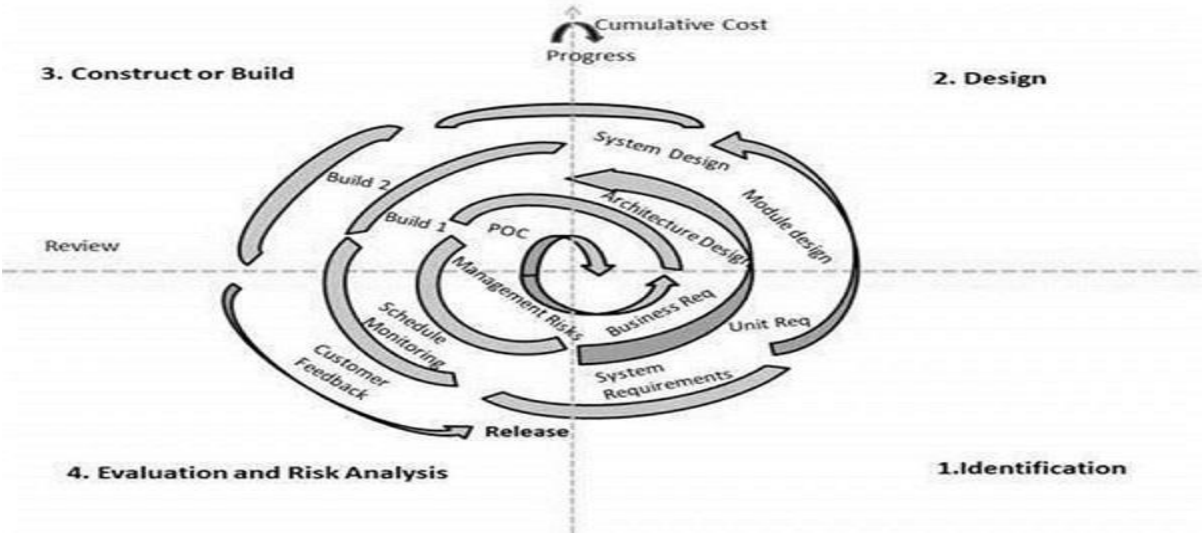
Construct or Build

The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.

Evaluation and Risk Analysis

Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.

The following illustration is a representation of the Spiral Model, listing the activities in each phase.



Based on the customer evaluation, the software development process enters the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iterations along the spiral continues throughout the life of the software.

Spiral Model Application

The following pointers explain the typical uses of a Spiral Model −

- When there is a budget constraint and risk evaluation is important.
- For medium to high-risk projects.
- Long-term project commitment because of potential changes to economic priorities as the requirements change with time.
- Customer is not sure of their requirements which is usually the case.
- Requirements are complex and need evaluation to get clarity.
- New product line which should be released in phases to get enough customer feedback.
- Significant changes are expected in the product during the development cycle.

Spiral Model - Pros and Cons

The advantage of spiral lifecycle model is that it allows elements of the product to be added in, when they become available or known. This assures that there is no conflict with previous requirements and design.

This method is consistent with approaches that have multiple software builds and releases which allows making an orderly transition to a maintenance activity. Another positive aspect of this method is that the spiral model forces an early user involvement in the system development effort.

On the other side, it takes a very strict management to complete such products and there is a risk of running the spiral in an indefinite loop. So, the discipline of change and the

extent of taking change requests is very important to develop and deploy the product successfully.

The advantages of the Spiral SDLC Model are as follows −

- Changing requirements can be accommodated.
- Allows extensive use of prototypes.
- Requirements can be captured more accurately.
- Users see the system early.
- Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.

The disadvantages of the Spiral SDLC Model are as follows −

- Management is more complex.
- End of the project may not be known early.
- Not suitable for small or low risk projects and could be expensive for small projects.
- Process is complex
- Spiral may go on indefinitely.
- Large number of intermediate stages requires excessive documentation.

4. V-model

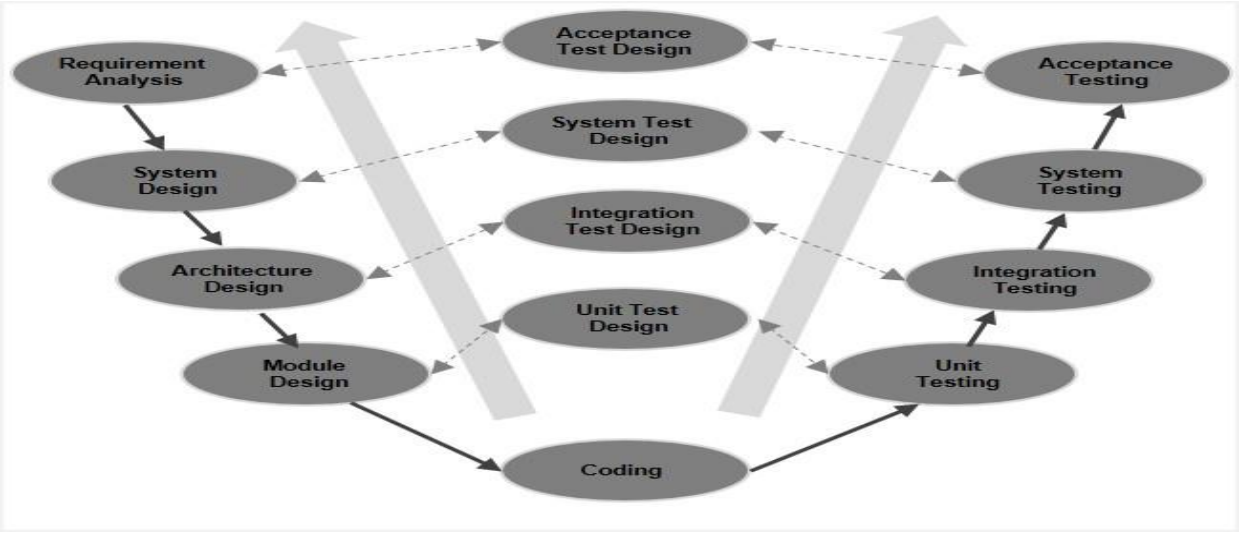
The V-model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as **Verification and Validation model**.

The V-Model is an extension of the waterfall model and is based on the association of a testing phase. This means that for every single phase in the development cycle, there is a directly associated testing phase. This is a highly-disciplined model and the next phase starts only after completion of the previous phase.

V-Model - Design

Under the V-Model, the corresponding testing phase of the development phase is planned in parallel. So, there are Verification phases on one side of the 'V' and Validation phases on the other side. The Coding Phase joins the two sides of the V-Model.

The following illustration depicts the different phases in a V-Model of the SDLC.



V-Model - Verification Phases

There are several Verification phases in the V-Model, each of these are explained in detail below.

Business Requirement Analysis

This is the first phase in the development cycle where the product requirements are understood from the customer's perspective. This phase involves detailed communication with the customer to understand his expectations and exact requirement.

System Design

Once you have the clear and detailed product requirements, it is time to design the complete system. The system design will have the understanding and detailing the complete hardware and communication setup for the product under development. The system test plan is developed based on the system design.

Architectural Design

Architectural specifications are understood and designed in this phase. Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken. The system design is broken down further into modules taking up different functionality. This is also referred to as **High Level Design (HLD)**.

Module Design

In this phase, the detailed internal design for all the system modules is specified, referred to as **Low Level Design (LLD)**. It is important that the design is compatible with the other modules in the system architecture and the other external systems.

Coding Phase

The actual coding of the system modules designed in the design phase is taken up in the Coding phase. The best suitable programming language is decided based on the system and architectural requirements.The coding is performed based on the coding guidelines and standards.

Validation Phases

The different Validation Phases in a V-Model are explained in detail below.

Unit Testing

Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage, though all defects cannot be uncovered by unit testing.

Integration Testing

Integration testing is associated with the architectural design phase. Integration tests are performed to test the coexistence and communication of the internal modules within the system.

System Testing

System testing is directly associated with the system design phase. System tests check the entire system functionality and the communication of the system under development with external systems. Most of the software and hardware compatibility issues can be uncovered during this system test execution.

Acceptance Testing

Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. Acceptance tests uncover the compatibility issues with the other systems available in the user environment. It also discovers the non-functional issues such as load and performance defects in the actual user environment.

V- Model – Application

V- Model application is almost the same as the waterfall model, as both the models are of sequential type. Requirements have to be very clear before the project starts, because it is usually expensive to go back and make changes. This model is used in the medical development field, as it is strictly a disciplined domain.

The following pointers are some of the most suitable scenarios to use the V-Model application.

- Requirements are well defined, clearly documented and fixed.
- Product definition is stable.
- Technology is not dynamic and is well understood by the project team.
- There are no ambiguous or undefined requirements.
- The project is short.

V-Model - Pros and Cons

The advantage of the V-Model method is that it is very easy to understand and apply. The simplicity of this model also makes it easier to manage. The disadvantage is that the model is not flexible to changes and just in case there is a requirement change, which is very common in today's dynamic world, it becomes very expensive to make the change.

The advantages of the V-Model method are as follows –

- This is a highly-disciplined model and Phases are completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Simple and easy to understand and use.
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.

The disadvantages of the V-Model method are as follows –

- High risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- Once an application is in the testing stage, it is difficult to go back and change a functionality.
- No working software is produced until late during the life cycle.

5. Big Bang Model

The Big Bang model is an SDLC model where we do not follow any specific process. The development just starts with the required money and efforts as the input, and the output is the software developed which may or may not be as per customer requirement. This Big Bang Model does not follow a process/procedure and there is a very little planning required. Usually this model is followed for small projects where the development teams are very small.

Big Bang Model – Design and Application

The Big Bang Model comprises of focusing all the possible resources in the software development and coding, with very little or no planning. The requirements are understood and implemented as they come. Any changes required may or may not need to revamp the complete software.

This model is ideal for small projects with one or two developers working together and is also useful for academic or practice projects. It is an ideal model for the product where requirements are not well understood and the final release date is not given.

Big Bang Model - Pros and Cons

The advantage of this Big Bang Model is that it is very simple and requires very little or no planning. Easy to manage and no formal procedure are required.

However, the Big Bang Model is a very high risk model and changes in the requirements or misunderstood requirements may even lead to complete reversal or scraping of the project. It is ideal for repetitive or small projects with minimum risks.

The advantages of the Big Bang Model are as follows −

- This is a very simple model
- Little or no planning required
- Easy to manage
- Very few resources required
- Gives flexibility to developers
- It is a good learning aid for new comers or students.

The disadvantages of the Big Bang Model are as follows −

- Very High risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Can turn out to be very expensive if requirements are misunderstood.

6. Agile SDLC model

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds. These builds are provided in iterations. Each iteration various areas like −

- Planning
- Requirements Analysis
- Design

- Coding
- Unit Testing and
- Acceptance Testing.

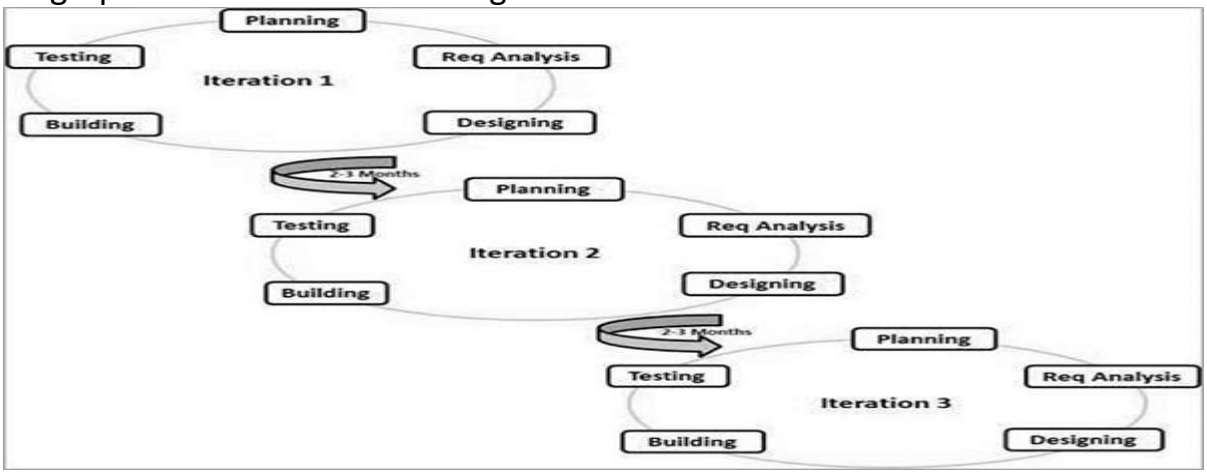
At the end of the iteration, a working product is displayed to the customer and important stakeholders.

What is Agile?

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.

Here is a graphical illustration of the Agile Model −



The Agile thought process had started early in the software development and started becoming popular with time due to its flexibility and adaptability.

Following are the Agile Manifesto principles −

- **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
- **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
- **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

Agile Vs Traditional SDLC Models

Agile is based on the **adaptive software development methods**, whereas the traditional SDLC models like the waterfall model is based on a predictive approach. Predictive teams in the traditional SDLC models usually work with detailed planning and have a complete forecast of the exact tasks and features to be delivered in the next few months or during the product life cycle.

Predictive methods entirely depend on the **requirement analysis and planning** done in the beginning of cycle. Any changes to be incorporated go through a strict change control management and prioritization.

Agile uses an **adaptive approach** where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. There is feature driven development and the team adapts to the changing product requirements dynamically. The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future.

Customer Interaction is the backbone of this Agile methodology, and open communication with minimum documentation are the typical features of Agile development environment. The agile teams work in close collaboration with each other and are most often located in the same geographical location.

Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products.

The advantages of the Agile Model are as follows –

- Is a very realistic approach to software development.
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements
- Delivers early partial working solutions.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.
- Little or no planning required.
- Easy to manage.
- Gives flexibility to developers.

The disadvantages of the Agile Model are as follows −

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.
- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.

6. RAD (Rapid Application Development) model

The **RAD (Rapid Application Development)** model is based on prototyping and iterative development with no specific planning involved. Rapid Application Development focuses on gathering customer requirements through workshops or focus groups, early testing of the prototypes by the customer using iterative concept, reuse of the existing prototypes (components), continuous integration and rapid delivery.

Rapid application development is a software development methodology that uses minimal planning in favor of rapid prototyping. A prototype is a working model that is functionally equivalent to a component of the product.

In the RAD model, the functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. Since there is no detailed preplanning, it makes it easier to incorporate the changes within the development process.

RAD projects follow iterative and incremental model and have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.

The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

RAD Model Design

RAD model distributes the analysis, design, build and test phases into a series of short, iterative development cycles.

Following are the various phases of the RAD Model −

Business Modelling

The business model for the product under development is designed in terms of flow of information and the distribution of information between various business channels. A complete business analysis is performed to find the vital information for business, how it can be obtained, how and when is the information processed and what are the factors driving successful flow of information.

Data Modelling

The information gathered in the Business Modelling phase is reviewed and analyzed to form sets of data objects vital for the business. The attributes of all data sets is identified and defined. The relation between these data objects are established and defined in detail in relevance to the business model.

Process Modelling

The data object sets defined in the Data Modelling phase are converted to establish the business information flow needed to achieve specific business objectives as per the business model. The process model for any changes or enhancements to the data object sets is defined in this phase. Process descriptions for adding, deleting, retrieving or modifying a data object are given.

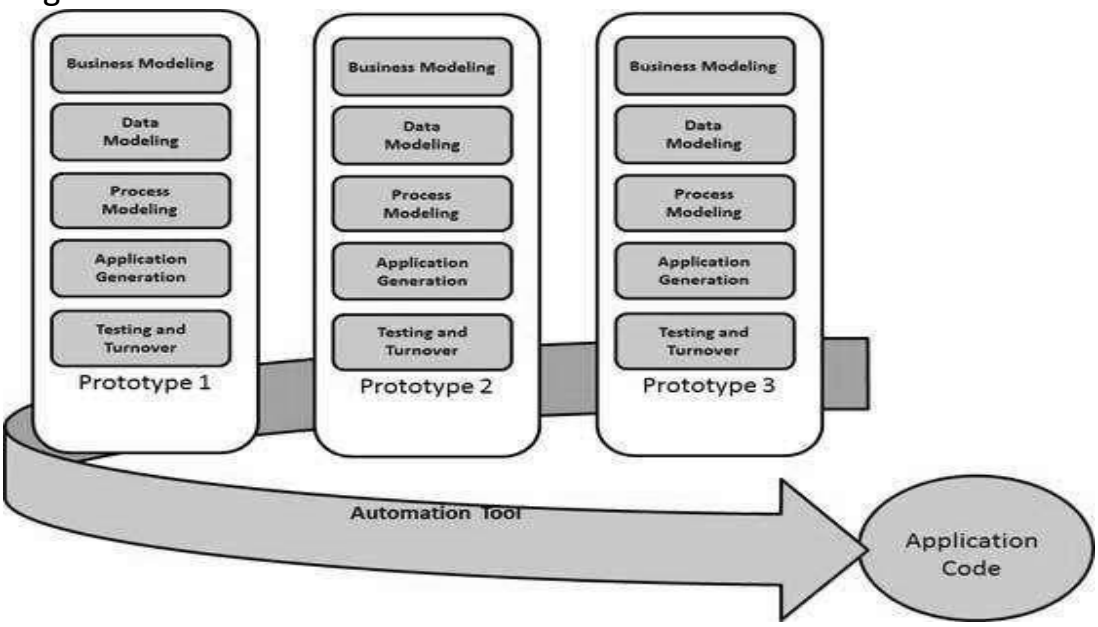
Application Generation

The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.

Testing and Turnover

The overall testing time is reduced in the RAD model as the prototypes are independently tested during every iteration.

The following illustration describes the RAD Model in detail.



RAD Model - Application

RAD model can be applied successfully to the projects in which clear modularization is possible. If the project cannot be broken into modules, RAD may fail.

The following pointers describe the typical scenarios where RAD can be used −

- RAD should be used only when a system can be modularized to be delivered in an incremental manner.
- It should be used if there is a high availability of designers for Modelling.
- It should be used only if the budget permits use of automated code generating tools.
- RAD SDLC model should be chosen only if domain experts are available with relevant business knowledge.
- Should be used where the requirements change during the project and working prototypes are to be presented to customer in small iterations of 2-3 months.

RAD Model - Pros and Cons

RAD model enables rapid delivery as it reduces the overall development time due to the reusability of the components and parallel development. RAD works well only if high skilled engineers are available and the customer is also committed to achieve the targeted prototype in the given time frame. If there is commitment lacking on either side the model may fail.

The advantages of the RAD Model are as follows −

- Changing requirements can be accommodated.
- Progress can be measured.
- Iteration time can be short with use of powerful RAD tools.
- Productivity with fewer people in a short time.
- Reduced development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

The disadvantages of the RAD Model are as follows −

- Dependency on technically strong team members for identifying business requirements.
- Only system that can be modularized can be built using RAD.
- Requires highly skilled developers/designers.
- High dependency on Modelling skills.
- Inapplicable to cheaper projects as cost of Modelling and automated code generation is very high.
- Management complexity is more.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for project requiring shorter development times.

3Q. Write short notes on Software Development Environment?

Software Development Environment (SDE)

A software development environment (SDE) is an environment that automates or augments the routines involved in a software development cycle. 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. This includes programming-in-the-many tasks like project management as well as programming. An SDE supports the large-scale and long-term maintenance of software as well.

With the advancement in technology and increased user expectations. The collection of software tools for developers has been enhanced significantly since the 1990s.

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. The environment is used in all phases of software development: requirements, design, implementation, verification and validation, and maintenance.

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.

The following are four categories represents trends that have a considerable influence over environments:

Language-oriented environments: These environment types are developed around one language, thereby offering a tool set suitable for that particular language. They are very interactive and provide restricted support for programming-in-the-large. Interlisp for Lisp and Smalltalk for Smalltalk are some common examples of language-centered environments.

- **Structure-oriented environments**: These environment types include techniques that let users directly manipulate the structures. These techniques are language independent, which triggered the concept of generators for environments.

- **Toolkit environments**: These environment types offer a set of tools that incorporate language-independent support for programming-in-the-large tasks, which include version control and configuration management.

- **Method-based environments**: These environment types include support for a wide variety of routines involved in the software development process. This includes tasks

like team and project management. They also feature tools for certain specification and design techniques.

Q4. Write short notes on Languages and software design methods?

Some languages provide better support for some design methods than others. Older languages, such as FORTRAN, were not designed to support specific design methods.

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*. 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.

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

Functional programming. The functional style of programming is rooted in the theory of mathematical 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.

Abstract data type programming. Abstract-data type (ADT) programming recognizes abstract data types as the unit of program modularity. CLU was the first language designed specifically to support this style of programming.

Module-based programming. Rather than emphasizing abstract-data types, 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. Modules can be used to define precisely which services are exported to the outside world by the module. In principle, any kind of service can be provided by a module, not just the ability to generate and use abstract data. Modula-2 and Ada support this style of programming.

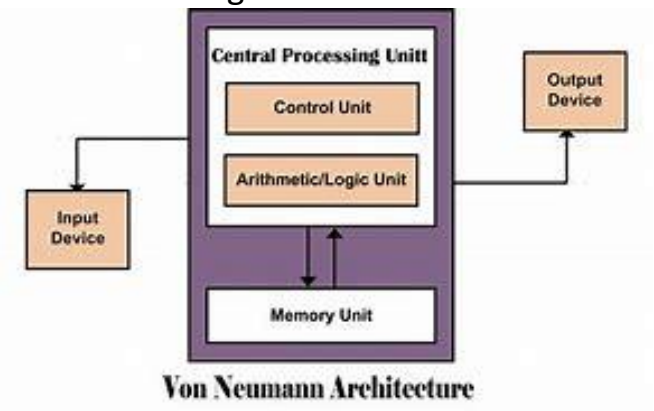
Object-oriented programming. The object-oriented programming style emphasizes the definition of classes of objects. Instances of classes are created by the program as needed during program execution. This style is based on the definition of hierarchies of classes and run-time selection of units to execute. Smalltalk and Eiffel are representative languages of this class. C++ and Ada 95 also support the paradigm.

Generic programming. This style emphasize the definition of generic modules that may be instantiated, either at compile-time or runtime, to create the entities—data structures, functions, and procedures— needed to form the program. This approach to programming encourages the development of high-level, generic, abstractions as units of modularity. The generic programming paradigm does not exist in isolation. It can exist jointly with object-oriented programing, as in Eiffel, with functional programming, as in ML. It also exists in languages that provide more than one paradigm, like Ada and C++.

Declarative programming. This style emphasizes the declarative description of a problem, rather than the decomposition of the problem into an algorithmic implementation. As such, programs are close to a specification. Logic languages, like PROLOG, and rule-based languages, like OPS5 and KEE, are representative of this class of languages.

Q5. EXPLAIN VON-NEUMANN COMPUTER ARCHITECTURE?

Accordingly, languages have been constrained by the ideas of Von Neumann, because most current computers are similar to the original Von Neumann architecture.

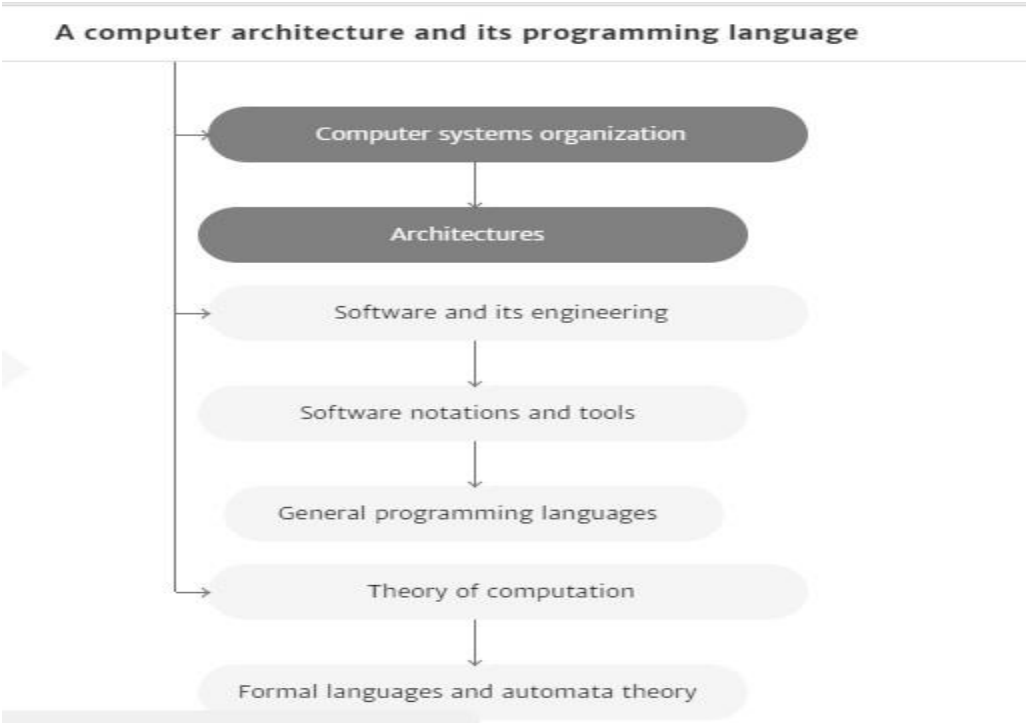


Von Neumann Architecture

The Von Neumann architecture, shown in the figure 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.

Sequential step-by-step execution of language instructions reflects the sequential fetch and execution of machine instructions performed by hardware.

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.



Computer architectures and programming languages are traditionally developed independently. Through suitable computer architecture, for instance, to speed up the processing of a stream of data and instructions, leaving to the software the burden of preparing these streams. Through a suitable programming language, one aims at efficiently describing many classes of problems, in a phrase-structure form that is machine independent. This approach has led to the ever-increasing application of computers, but it has also brought about a growing complexity in the software systems. As a consequence of the latter, there is a new trend toward extending hardware implementation to replace the software ones, especially in view of the new technology of large scale integration.

Hierarchy of Computer language –

Most Popular Programming Languages –

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Q6.Write short notes on Programming language qualities or What are the Good Qualities consist for a Good Programming language?

Characteristics of a programming Language –

- A programming language must be simple, easy to learn and use, have good readability and human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for development, debugging, testing, maintenance of a program must be provided by a programming language.
- A programming language should provide single environment known as Integrated Development Environment(IDE).
- A programming language must be consistent in terms of syntax and semantics.

Characteristic of A Good Programming Language:

There are various factors, why the programmers prefer one language over the another. Some of very good characteristics of a good programming language are,

1) Clarity, Simplicity And Unity: A Programming language provides both a conceptual framework for Algorithm planning and means of expressing them. It should provide a clear, simple and unified set of concepts that can be used as primitives in developing algorithms.

It should have

• It has minimum number of different concepts

•- with Rules for their combina-tion being

•-simple and regular.

This attribute is called **conceptual integrity.**

2) Orthogonality: It is one of the most important feature of PL orthogonality is the property that means **" Changing A does not change B".**

If I take Real world example of an orthogonal system Would be a radio, where changing the station does not change the volume and vice versa.

When the features of a language are orthogonal, language is **easier to learn** and **programs are easier to write** because only few exceptions and special cases to be remembered.

3) Support for Abstraction:- There is always found that a substantial gap remaining between the abstract data structure and operations that characterize the solution to a problem and their particular data structure and operations built into a language.

4) Programming Environment: An appropriate programming environment adds an extra utility and make language to be implemented easily like

The availability of- Reliable- Efficient - Well documentation

Speeding up creation and testing by-special Editors- testing packages
Facility- Maintaining and Modifying- Multi Version of program software product.

5) Ease of program verification:- Reusability:

The reusability of program written in a language is always a central concern. A program is checked by various testing technique like

Formal verification method Desk checking Input output test checking.

We verify the program by many more techniques. A language that makes program verification difficult maybe far more troublesome to use. **Simplicity of semantic and syntactic structure is a primary aspect that tends to simplify program verification.**

6) portability of programs: Programming language should be portable means it should be easy to transfer a program from which they are developed to the other computer.

A program whose definition is independent of features of a Particular machine forms can only support **Portability. Example: Ada, FORTRAN, C, c++, Java.**

7Q. Write short notes on historical developments of programming languages?

A brief history of Programming Language: Programming Languages

What is Programming language ?

Programming language is a notation for describing Algorithm and data.

What is a Program :

A sentence of a programming language.

Let start from year "1954"

Fortran (FORmula TRANslator)

- It is Created in 1954 by John Backus.
- It is First high level language.
- It was developed by using the first compiler ever developed
- It is Machine Independent Language.

FORTRAN 2nd :

- In 1958 introduces subroutines, **functions, loops and primitive for loop**

IAL(International Algebraic Logic)

- It started as a Project later renamed ALGOL58
- The theoretical definition of the language is published
- No compiler is required in this

LISP(LISt Processing) :

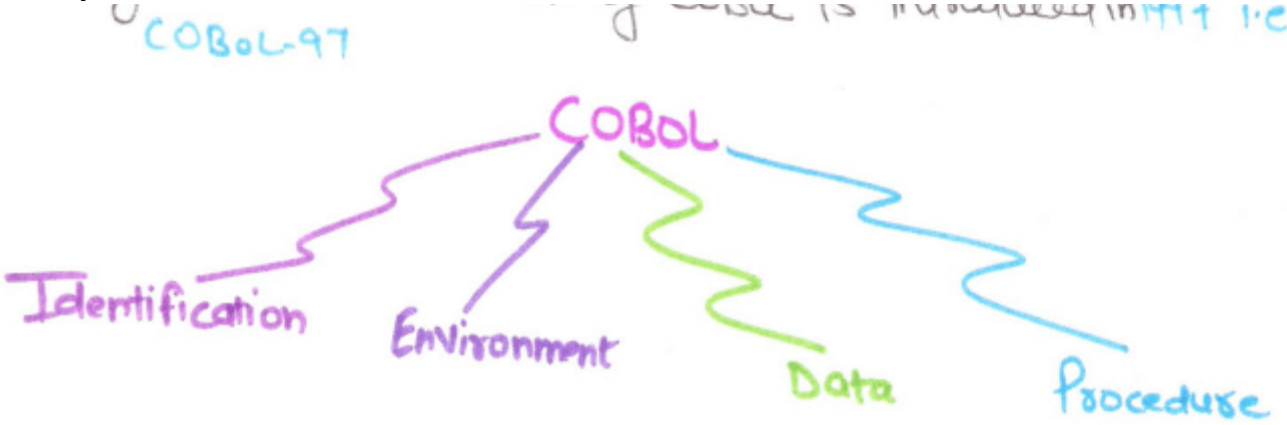
- It is created in 1958 and released in 1960 by John Mccarthy of MIT.
- LISP was intended for writing artificial intelligence programs.

Features

- It uses **Atoms and List** as data structure
- **Functional programming style-** all component computation is performed by applying functions to arguments variable declaration are rarely used.
- **A Reliance on recursion-** a strong Reliance on recursion has allowed LISP to be successful in many areas including artificial intelligence.
- **Garbage collection:** LISP has built in garbage collection so programmers do not need to explicitly free dynamically allocated memory.

COBOL(Common Business Oriented Language)

- It was rated in May 1959 by the ShortRange committee of the us department of DOD.
- The CODASYL(COnference on Data SYstems Languages) Worked from May 1959 to April 1960.
- ANSI standard included(cobol-68(1968), Cobol-74(1974),COBOL-85(1985), and COBOL-2002(2002)
- Object Oriented version of COBOL is introduced in 1997 i.e COBOL-97



- Introduced the Record Data Structure.

ALGOL(ALGOrithmic Language)

- It was released in "1960"and its major releases were in "1960" and "1968".
- It Was first **"Block Structured Language."**
- It was Considered to be the first second generation Computer Language
- It is Machine Independent language
- **It introduced concepts like :**
- 1) Block structure code(Marked by BEGIN and END)
- 2) Scope of variables(Scope of local variables inside blocks)

- 3) BNF (Backus Naur Form)
- 4) Notation for defining syntax
- 5) Dynamic Arrays
- 6) Reserved words
- **7) IF THEN ELSE, FOR, WHILE loops**

8. SWITCH with GOTO

User defined data type.

SNOBOL(StroNg Oriented symBOlic Language)

- It was created in 1964-
- Inted for "Strings"
- First Language to use Associative arrays , indexed by and type of key.
- Had feature of pattern matching, concatenation, and alternation.
- It allowed running code stored in strings
- Data types:- integer, real, array, pattern , and user defined types.

BASIC(Beginner's All-purpose Symbolic Instruction Code)

- Designed as a teaching language in 1963 by John george kemeny and Thomas Eugene Kurtz of Dartmouth college.
- Intended to make it easy to learn programming.

PL/I (Programming language One)

- It was Created in 1964
- Intended to combine the features of fortran with Cobol, plus additional facilities for system programming.
- Also borrows from ALOL 60.
- Originally called NPL (New programming Language).
- Introduces storage classes(automatic, Static, controlled and based), exception processing(on condition)
- It uses Select when otherwise conditional structure and several variation of the DO Loop.
- It uses Numerous data types.

Pascal-(Named for French religious fanatic and mathematician Blaise Pascal)

- It was Created in 1970 .
- It was intended to replace BASIC for teaching language.
- It was quickly developed as a general purpose language.
- It was Programs compiled to a platform-independent intermediate p-code.

- compiler for Pascal was written in Pascal.

C language:

- It was Developed from "1969 - 1972" by "Dennis Ritchie."
- It was used in system programming for UNIX.

ANSI C: The American national standards Institute(ANSI), formed a technical sub committee, X3Jll, to create a standard for the C language and it runtime libraries. **Ada:**

- It was Released in "1983 (ADA 83)", with major released in "1995- (ADA (ADA 95)" and "2005 1005 (ADA 2005)".
- It was created by us Department of Defence(DOD).
- It was intended for embedded systems and later intended for all military computing purposes. **Perl (Practical Extracting and Report Language)**
- It was created by Larrywall ball in 1987.
- It was intended to replace the Unixshell, Sed, AWK.

Python:

- It was Created in 1991 by GuidoVan Rossum.
- A scripting language with dynamic type, intended to replace perl.

Chapter 2:

Q8. EXPLAIN ABOUT SYNTAX AND SEMANTICS OF A PROGRAMMING LANGUAGE?

LANGUAGE DEFINITION: A language definition should 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-programming or natural language-must be defined: syntax and semantics.

Introduction to Syntax and Semantics :

Syntax:

Any programming language specifies a set of rules for the *form* of valid programs in that language. Like ordinary language English, programming languages have syntax**.** The Syntax of a (programming) language is a set of rules that define what sequences of symbols are considered to be valid expression (programs) in the language. Or The Syntax of a programming language is what the program looks like.

The *syntax* rules of the language state how to form expressions, statements, and programs that *look* right. The semantic rules of the language tell us how to build *meaningful* expressions, statements, and programs.

Characters are the ultimate syntactic building blocks. Every program is formed by placing characters together in some well-defined order. The syntactic rules for forming programs are rather straightforward.

Syntax provides significant information needed for understanding a program and provides much needed information towards the translation of the source program into the object program.

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.

A valid representation of syntax

X=Y+Z

Invalid representation maybe

XY+-

2+3*4 text will be interpreted this expression as having value 14 and not 20. That is, expression is interpreted as if written (2+3)*4.

We can specify either interpretation, if we wish, by syntax and hence guide the translator into generating the correct operations for evaluating this expression.

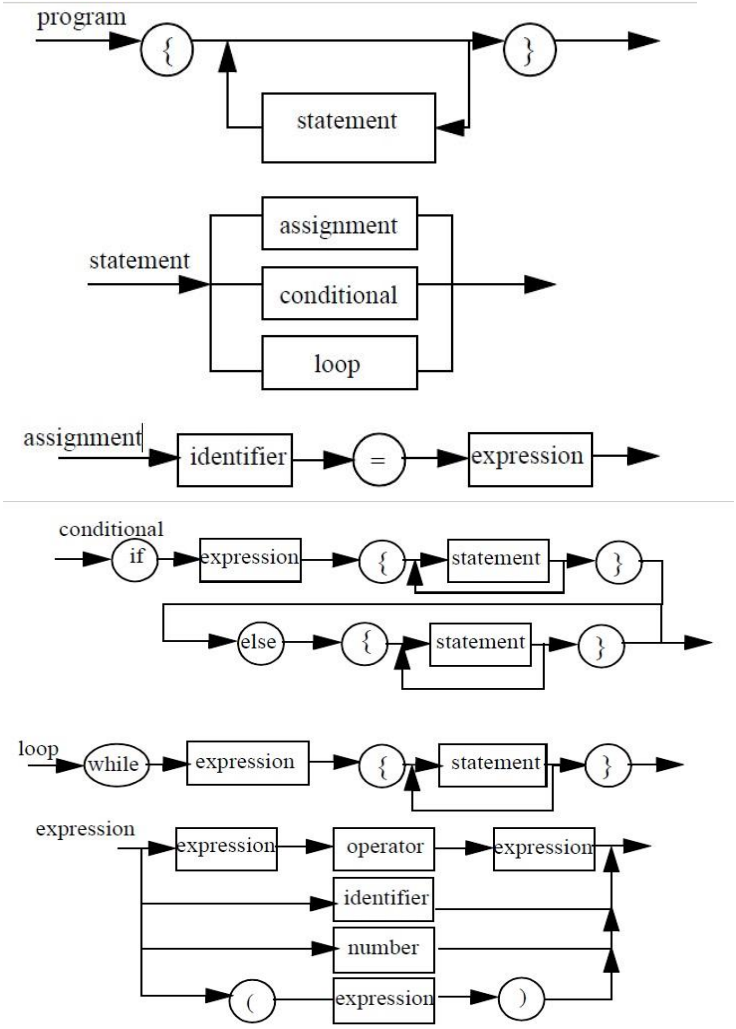
In a Statement

X=2.82 + 3.68
syntax cannot tell the type of x on which result is depended.
If x is Real then output will be 6.50 and
if x is integer then output will be 6.

To completely describe the syntactic structure of programming language we need something else which can tell us the meaning of expression, statement and program units.

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 metalanguage: they are *metasymbols*. 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 “}”.



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

(a) It helps the programmer know how to write a syntactically correct program. For example, if one is unsure about the syntax of **if** statements, a look at the EBNF or syntax diagrams can quickly settle any doubts.
(b) It can be used to determine whether a program is syntactically correct. This is exactly what a compiler does.

Semantics:

Syntax defines well-formed programs of a language. Semantics defines the meaning of syntactically correct programs in that language. Semantics is the meaning of an expression (program) in a programming language. 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.

In C to declare a 10 elements vector V of Integer has declaration

int v:{10};

In Pascal

v:array[0.....9] of integer

Although both creates similar data objects at run time , their syntax is very different. To Understand the meaning of declaration we need to know the semantics of both pascal and c for such array declaration.

Another example

While (< boolean_exp>)< statement>

The semantics of the language might require such expressions to deliver a truth value (TRUE or FALSE, not–say–an integer value). The semantics of this statement form is that when the current value of the Boolean exp. Is true, the embedded statement is true.
NOTE: Not all syntactically correct programs have a meaning. Thus, semantics also separates meaningful programs from syntactically correct ones.

Operational Semantics - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement • To use operational semantics for a high-level language, a virtual machine in needed • A hardware pure interpreter would be too expensive • A software pure interpreter also has problems:

Axiomatic Semantics - Based on formal logic (first order predicate calculus) - Original purpose: formal program verification - Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)

Denotational Semantics - Based on recursive function theory - The most abstract semantics description method - Originally developed by Scott and Strachey - The process of building a

denotational spec for a language: 1. Define a mathematical object for each language entity 2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

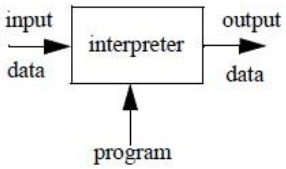
Q9. WRITE SHORTNOTES HOW PROGRAMMING LANGUGE CAN BE PROCESSED?

Language processing:

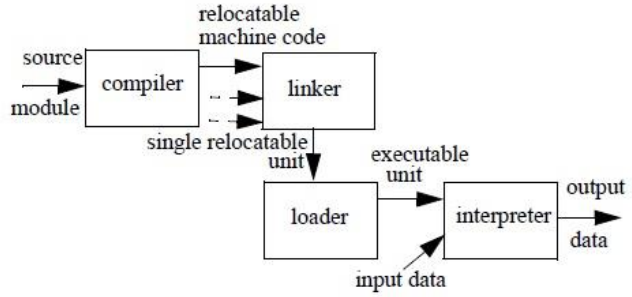
Computers to execute directly programs written in any particular language, present-day computers directly execute only a very low-level language, the machine language. Machine languages are designed on the basis of speed of execution, cost of realization, and flexibility in building new software layers upon them.

There are generally two extreme alternatives for an implementation:

interpretation and translation(Compilation)



(a) Interpretation



(b) Translation (+ interpretation)

Programming Language Translators:

Introduction to translators:

To **Execute** a computer program written in high level language must be translated into machine understandable language that is machine language or machine code .

A **Translator** is basically computer program that performs the translation of a program written in a given programming language into functionality equivalent program in a different computer language, without losing the functional or logical structure of the original code.

There are three types of Programming Language Translators

- **Assembler**
- **Compilers**
- **Interpreters**

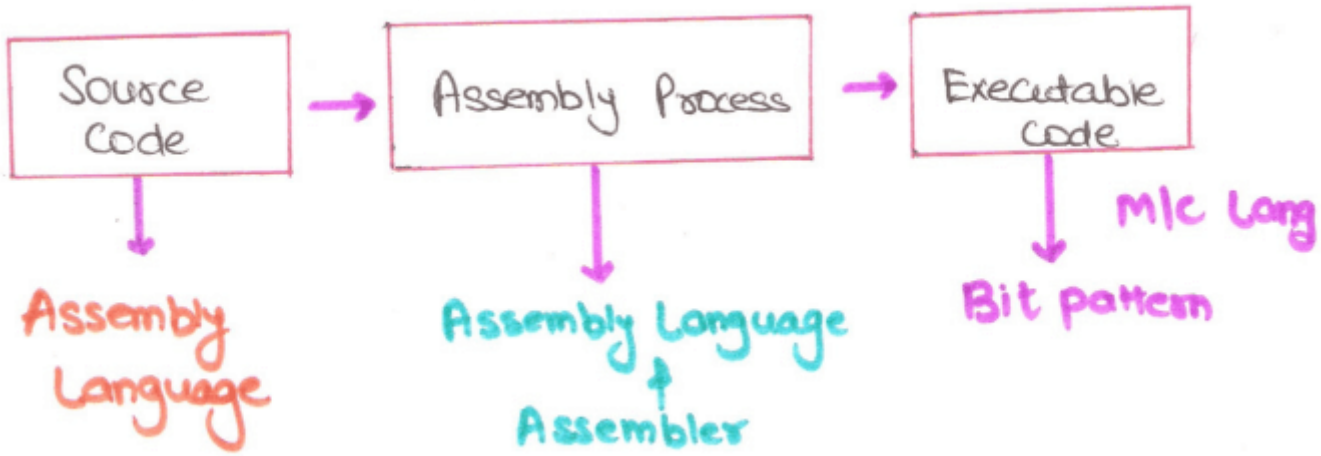
Source Code: It is the input to translator.

Executable code: Is the code that is output from Translator.

Assembler:Programming Language Translator

An Assembly Language which is basically **MNEMONICS** like **GO ,HALT, JUMP, NOT** code which is translated to **Machine Language** by Programming Language Translator i.e **Assembler.**

So **Assembler** is a program that takes assembly and converts them into pattern of bits that the computer processor can use to perform its basic operations. **This pattern of bits is basically Machine Language.**



Examples of Assembler are -NASM and -MASM

Compiler

Programming Language Translator

Compiler is a program that translates a high-level language into machine code.

For Example:

The Turbo Pascal compiler translate a program written in Pascal into machine code that can be run on a PC.

Advantages of a compiler:

- **1) It is fast in execution**
- 2) The object / executable code produced by compiler can be distributed or executed without having to have compiler present.
- 3) The object program can be used whenever required without the need of re-compilation.

Disadvantages of a compiler:

- 1) In this Debugging a program is much harder than other.
- **2) when error is found, the whole program has to be re-compiled.**

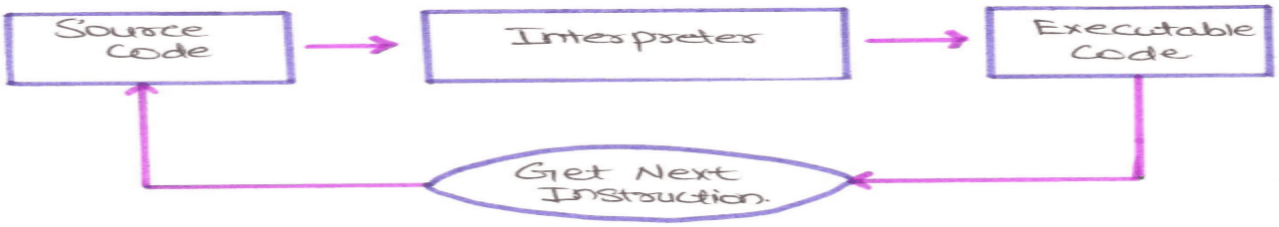
Interpreter: - Programming language Translator:

An**Interpreter** is also a program that translates high level source code into **Executable Code.**

However the

Difference between a Compiler and an Interpreter

is that an interpreter translates one line at a time and then executes it. No object code is produced, and so the program has to be interpreted each time it is to be run.



Advantages of an Interpreter

- 1) Good at locating errors in the program
- 2) Debugging is easier since the interpreter stops when it encounters an error.
- 3) if an error is deducted there is an no need to re-translate the whole program.
- 4) It uses less memory as only few lines has to be into the memory because no object code.

Disadvantages of an interpreter:

- 1) Rather Slow: Speed is biggest disadvantage
- 2) no object code is produced, so translation has to be done every time the program is running.
- 3) For the program to run, the interpreter must be present.

Interpreter has to analyse each line of code (byte code) into machine code before it can be executed.

Programming Language:

What is Data Object:

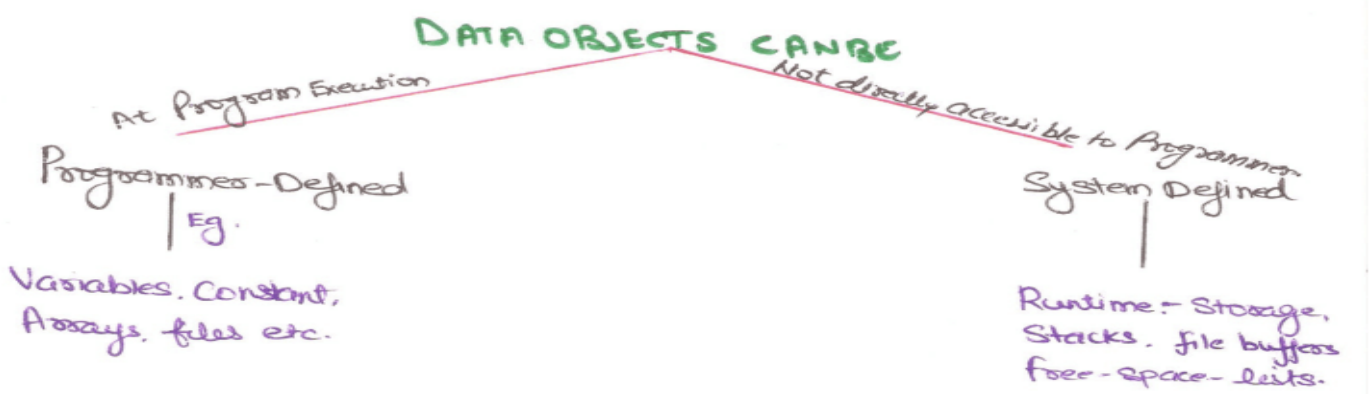
Definition: A Data object represents a container for data values, a place where data values may be stored and later retrieved.

Definition: A runtime grouping of one or more pieces of data in a virtual computer.

Definition: A Location in memory with an assignment name in the actual computer.

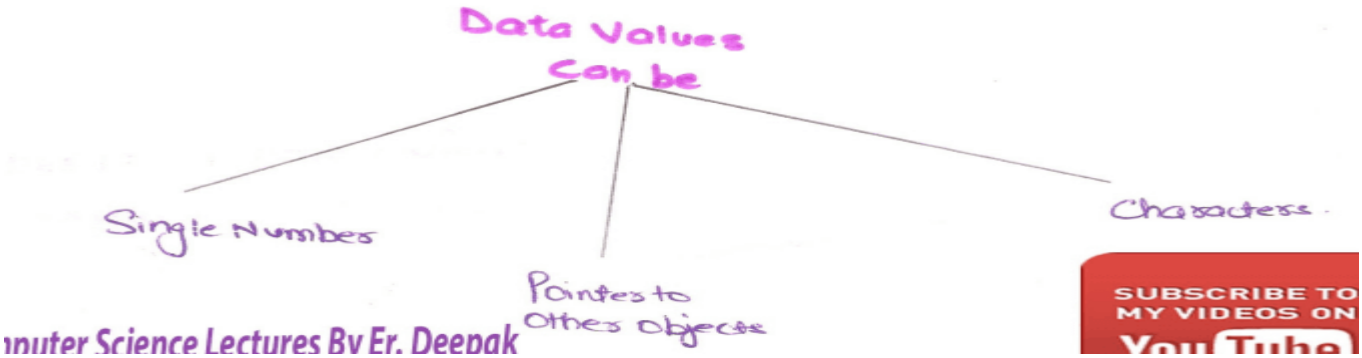
Data objects can be:

- 1) at program execution- Programmer-defined -(example variables, constant, arrays, files etc)
- 2) not directly accessible to programmer- system defined- run time storage, stacks, file buffers, free space lists.

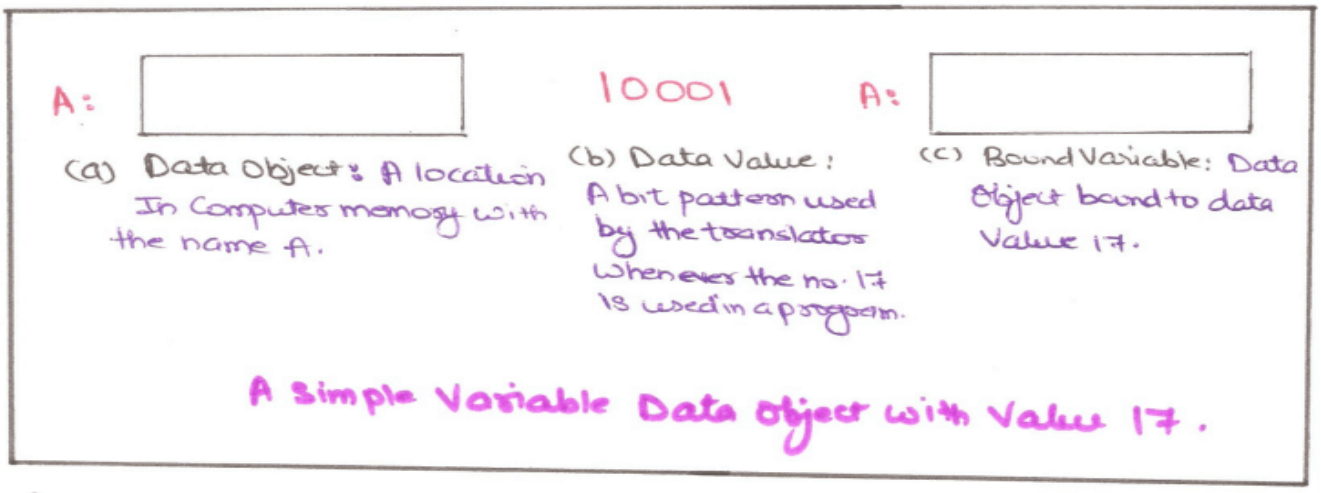


Data values can be:

- Single number
- Pointer to other objects and characters.



Data object is usually represented as storage in Computer memory and a data value is represented by a pattern of bits. **So we can represent the relation between Data Object and Data value.**



A Data Object is elementary if it contains a data value that is always manipulated as a unit.

A Data Object is an **Data Structure** if it is an aggregate of the data object

Binding and Attributes of Data Object:

Binding is an association of data values and entity.

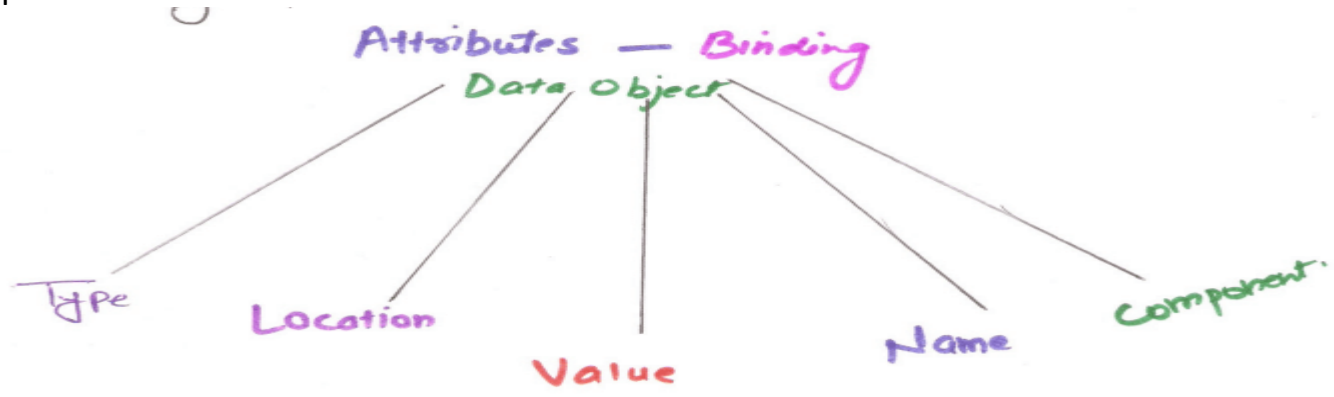
1) Type: This associates the data object with the set of data values that the object may take.

2) location: This associate the binding of a storage location in memory where the data object is represented. Only storage management routines can only change add data object in the virtual computer.

3) Value: This binding is usually the result of an assignment operation.

4) Name: The binding to one or more names by which the object maybe referenced during program execution is usually set up by declaration and modified by subprogram calls the returns.

5) Component: The binding of a data object to one or more data object of which it is a component is often represented by a pointer value. And may be modified by a change in pointer.



Q11. WRITE SHORT NOTES ON VARIABLE S AND CONSTANTS?

Variables and Constants

Variables: A variable is it Quadruple which is composed of a name,a set of attributes, a reference and a value.

A simple variable is an elementary data object with a name and binding of data object to value may change during its lifetime. This data objects are basically defined and named by programmer explicitly.

A variable corresponds to a region of memory which is used to hold *values* that are manipulated by the program. We refer to a variable by its *name*. The syntactic rules specify how variables may be named.

But there are many semantic issues associated with variables. A *declaration* introduces a variable by giving it a name and stating some of its semantic properties. Among the important semantic roperties are:

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. Different programming languages adopt different rules for binding

variable names to their scope.

• S*cope*: what part of the program has access to the variable? That is, the variable may be referred to in any part of the program from the declaration of the variable to the end of the function main.

•T*ype*: what kinds of values may be stored in the variable and what operations may beperformed on the variable? Usually, there are a number of fundamental types defined by the language and there are some facilities for the user to define new types. Languages differ both in terms of the fundamental types and in the facilities for type definition. The fundamental types of traditional languages are dictated by the types that are supported by the hardware. Typically, as in C++, the fundamental types are integer, real, character. Pascal also has boolean types.

•L*ifetime*: when is the variable created and when is it discarded? a variable represents some region of memory which is capable of holding a value. The question is when is a memory area reserved, or *allocated*, for the variable? Some possibilities are: when the program starts, when the declaration is encountered at execution time, when the unit in which the declaration occurs is entered, or there could be a statement that explicitly requests the allocation of storage for the variable.

Indeed, C++ has all of these kinds of variables: automatic variables are allocated when the unit in which they are declared is entered and deallocated when the unit terminates; static variables live throughout the execution of the program; some variables may be created and destroyed explicitly by the programmer using the operators new and delete.

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);
}
```

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.

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.

Constant:

A data object with a name that is bound to a value (or values) permanently during its lifetime. The constant value can only be a number, string or identifier which denotes constant.

A constant definition in Pascal introduces an identifier as a synonym for the constant value.

Pascal uses the reserved word const to begin a constant declaration.

```
const PI=3.1415;
```

In ALGOL68 we can define constant by

```
real root2=1.4142135;
```

That was much acceptable that time.

In Ada, provides a uniform notation for setting constants to initial values and for initializing variables.

```
X: Constant INTEGER:=17;
```

In C language: const is used to initialise the constant value

```
const int MAX=80;
```

The constant MAX is a programmer defined constant because the programmer explicitly defines the name for the value 30.
In C, there is micro definition which is used for control the execution of program and can be used for declaring constant.

Example #define MAX30

It is a compile-time operation that causes all references to MAX in program to be changed to the constant 30.

In this 30 has two names, the programmer defined MAX and literal name 30. Both of which may be used to refer to a data object containing the value 30.

 # define MAX 30 is a command, which the translator used to equate **"MAX"** with the value **"30"**, where as the const attribute in C is a translator directive starting that MAX will always contain the value 30.

Q12. EXPLAIN ABOUT ROLL OF ROUTINES AND WHAT IS THE USE OF ROUTINES?

Routines

Programming languages allow a program to be composed of a number of units, called *routines*.

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.

In 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.

In computer programming, routine and subroutine are general and nearly synonymous terms for any sequence of code that is intended to be called and used repeatedly during the executable of a program. This makes the program shorter and easier to write (and also to read when necessary). The main sequence of logic in a program can branch off to a common routine when necessary.

A routine may also be useful in more than one program and save other programmers from having to write code than can be shared. In higher-level computer languages, many commonly-needed routines are prepackaged as function , which are routines with specified programming interfaces. Some functions can be compiled in line with other code. Other functions are compiled in as stub that make dynamic calls for system services during program execution. Functions are sometimes called *library routines* . The compiler and a set of library routines usually come as part of a related software development package.

Subroutine example

```
&hello;
```

```
sub hello {
    print "Hello World!\n";
}
```

In the example above, a user could call the **hello** subroutine (in this example, by typing *&hello;*) and have the program print *Hello World!*. After the subroutine has completed, the program goes to the return address, which is directly after *&hello;* and runs any additional code.

Like variables, routines have a name, scope, type, l_value, and r_value. 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.

A routine's l_value is a reference to the memory area which stores the routine *body* (i.e., the routine's executable statements) and execution of the routine body, which constitutes the r_value.

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.

A routine instance is composed of a code segment and an activation record. The *code segment*, whose contents are fixed, contains the instructions of the unit. The contents of the *activation record* (also called *frame)* are changeable. The activation record contains all the information necessary to execute the
routine, including, among other things, the data objects associated with the local variables of a particular routine instance.

--- ****----

Aliasing And Overloading:

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. The language uses special names (denoted by operators), such as + or * to denote certain predefined operations.

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.

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.

Execution-time structure(Run time Structure):
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.

UNIT-II

Structuring the data:

Computer programs can be viewed as functions that are applied to values of certain input domains to produce results in some other domains. In conventional programming languages, this function is evaluated through a sequence of steps that produce intermediate data that are stored in program variables. Languages do so by providing features to describe data, the flow of computation, and the overall program organization.

Programming languages organize data through the concept of *type*. Types are used as a way to classify data according to different categories. They are more, however, than pure sets of data. Data belonging to a type also share certain semantic behaviors. A type is thus more properly defined as a set of values and a set of operations that can be used to manipulate them. For example, the type BOOLEAN of languages like Ada and Pascal consists of the values TRUE and FALSE; Boolean algebra defines operators NOT, AND, and OR for BOOLEANs. BOOLEAN values may be created, for example, as a result of the application of relational operators (<, ð, >, Š, +, ¦) among INTEGER expressions.

Programming languages usually provide a fixed, built-in set of data types, and mechanisms for structuring more complex data types starting from the elementary ones.

Q1. EXPLAIN ABOUT NEED OF DATATYPES IN PROGRAMMING LANGUEGES? OR WRITE ABOUT DIFFRENCES BETWEEN BUILT-IN AND PRIMARY DATA TYPES?

Data Types In Programming Languages

Data type is a set of object and a set of operation on those object which create, Build up, destroy, modify and pick apart instances of the objects.

or

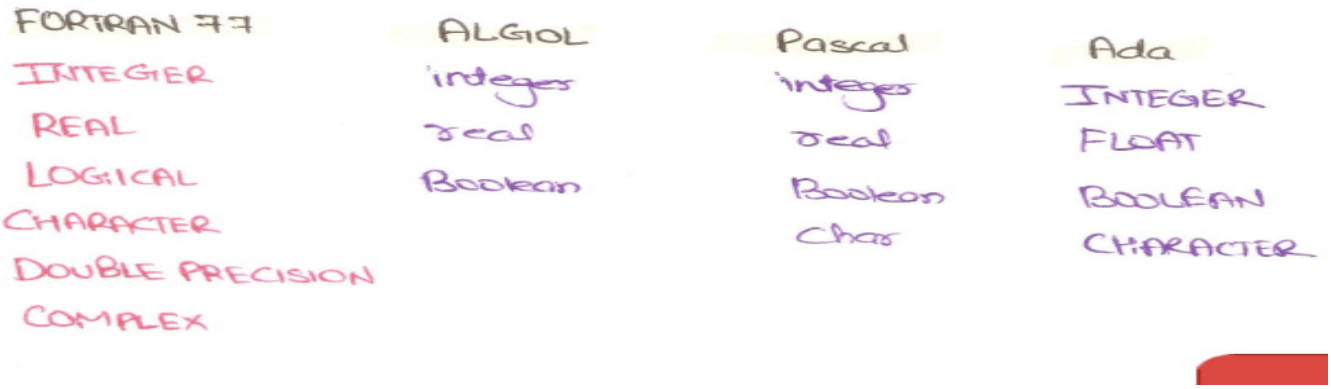
A data object is a class of data objects together with a set of operations for creating and manipulating them.

Any programming language is equipped with a finite set of *built-in types* (or *predefined*) types, which normally reflect the behavior of the underlying hardware.

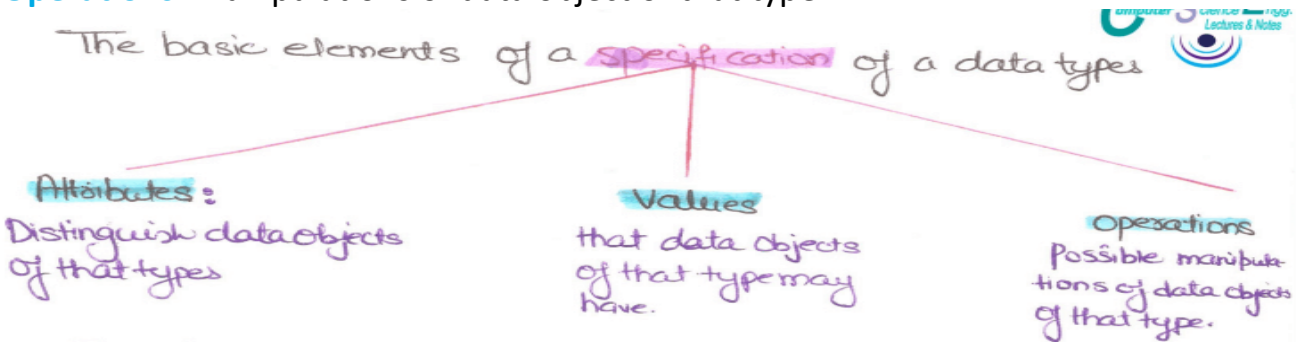
A programming languages necessary deals more commonly with data types such as the class of arrays, integers, or file and the operations provided for manipulating, arrays, integers, or files.

Example:

In LISP major data type is the binary tree(called an S- expression) and basic operations are **CAR,CDR and CONS.**



Fortran 77- integer, real, logical, character, double precision, complex
ALGOL- integer, real, boolean
PASCAL- integer, real, boolean, char
Ada- integer, float,Boolean, character
The basic Elements of a Specification of a data types
Attributes- distinguish data object of that types .
Values- that data object of that type mein have.
Operations- manipulations of data object of that type.



Example: Array data type
Attribute:-
- Numbers of dimensions
- The subscript range for each dimension and
- The data type of components.

Value:
- It would be sets of numbers that form valid values for array components.

Operations-
- It may include subscripting to select individual array components
- create arrays.
- change their shape
- performing arithmetic on pairs of arrays.

The basic elements of the implementation of a Data Types:
1) Storage Representation: It is used to represent the data objects of the data type in the storage of the computer during program execution.
2) Algorithm or Procedures: The manner in which the operations defined for the data type are represented in terms of particular algorithms or procedures that manipulate the chosen storage representation of the data objects.

Q2. EXPLAIN HOW ELEMENTARY DATA TYPES ARE SPECIFIED AND IMPLEMENTED?

Specification of Elementary Data Types

An elementary data object contains a single data values and class of such data objects over which various operations are defined is termed as elementary data type.

Some elementary data types: Integer, real, character, Boolean, enumeration and pointer and specification may differ significantly between two languages.

Attributes: Basic attributes of any data object, such as data type and name are usually invariant during its lifetime.

Some attributes may be stored in a descriptor as a part of the data object daily program execution. Others may be used only to determine the storage representation of the data object.

The value of an attribute of a data object is different from the value that the data object contains.

Values: The type of a data object determines the set of possible values that it may contain.

For Example: C defines the following four classes of integer types

int, short, long and char

because most hardware implements multiple Precision integer arithmetic(example 16 bit and 32 bit integers or 32 bit and 64 integers) We can use' short' for shortest value of the integer word length.

long uses the longest value implemented by the hardware.

int uses the most efficient value that the hardware implements.

In C, Characters are stored as 8 bit integers in the type char, which is subtype of integer.

Operations:- The set of operations Defined by language is basically refers that how data object of that data type may be manipulated.

If the operations are primitive operation, means specified as part of language. Programmer defined operations, in the form of subprograms or method declarations as part of class definitions.

Example

Integer* integer-> **integer**

a) integer addition is an operation that take to integer data objects as an arguments and produces an integer data object as a result.

b) SQRT: real-> real

A Square-root operation, SQRT,on real number data object is specified.

(Port of operation)

An algorithm that specifies how to compute the results for any given set of arguments is a common method for specifying the action of an operation.

In C, we have concept a function prototype which signature of an operation, the number, order and data types of the arguments in the domain of an operation are given as well as the order and the data type of the resulting range.

Binary operation: Two arguments with single result

Monolic operation: Single argument with single result.

Implementation of Elementary Data Types:"

Implementation of Elementary data type consists of

- **Storage representation for data objects**
- **Values of that type**
- **Set of algorithms or procedures that define the operations of the type in terms of manipulations of the storage representation.**

Storage representation of Elementary data type:

1) Hardware Influence: Computer hardware influence the storage of elementary data type.

In this case computer hardware executes the program. If the hardware storage representation are used, then the basic operations on data of that data type not implemented using hardware provided operations.

2) Software influenced: If we do not use hardware storage representation, then the operation must be software simulated and some operation will execute much less efficient.

Two methods to treat Attributes:

It has to be determined by the compiler and not stored in discriptors during execution or not stored in runtime storage representation. It is usually a method in C language.

It is stored in a descriptor as part of the data object at runtime in LISP, Prolog language .

The storage representation is usually described in terms of

- **Size of the block of memory required(the number of memory words bytes, ot bits needed)**
- **Layout of attributes and data values within the block.**

Implementation of operations:

Each operation defined for data objects of a given type may be implemented in one of three main ways:-

1) Directly as a hardware operation: If simple data types are stored using the hardware representation, when the primitive operations are implemented using the arithmetic operations built in to hardware.

2) As a Subprogram or procedure: A square root for an example, this operation is not provided directly as a hardware operation. So it is software simulated implemented as a procedure or function.

3) as an inline code sequence: It is software implementation of the code and its operation. Instead of using a subprogram, operation in the subprogram are copied into the program at the point where the subprogram would otherwise have been invoked.

For Example:

The absolute value of function on numbers

abs(x)= if x<0 then -x else x

is usually implemented as an inline code sequence.

a) fetch value of x from memory
b) if x >0, skip the next instruction
c) set x=-x
d) store new value of x in memory

Here each line is implemented by a single hardware operation.

Declarations>

Declarations provide information about the name and type of data objects needed during program execution.

Two types of declaration:

- implicit declaration
- explicit declaration

Implicit declaration or default declaration:

They are those declaration which is done by compiler when no explicit declaration or user defined declaration is mentioned.

Example

$abc='astring';
$abc=7;

In 'perl' compiler implicitly understand that

$abc ='astring' is a string variable and
$abc=7; is an integer variable.

Explicit declaration of data object:

Float A,B;

It is an example of Float A,B, of c language. In explicit we or user explicitly defined the variable type. In this example it specifies that it is of float type of variable which has name A & B.

A "Declaration" basically serves to indicate the desired lifetime of data objects.

Declarations of operations:

- compiler need the signature of a prototype of a subprogram Or function so it can determine the type of argument is being used and what will be the result type.

*** Before the calling of subprogram, Translator need to know all these information. ***

Example in C language

Float sub(int z, float y)

It declares sub to have the signature

Sub: int xfloat-> float

Purpose of Declarations:

1) Choice of storage representation: AS Translator determine the best storage representation of data types that why it needs to know primarily the information of data type and attribute of a data object.

2) Storage Management: It make to us to use best storage management for data object by providing its information and these information as tells the lifetime of a data object.

For Example:-

In C language we have many options for declaration for elementary data type.

1) Simple Declaration: Like float A,B;

It tells lifetime is only at the end of execution **as lifetime of every data objects can be maximum to end of execution time.**

But simple declaration tells the single block of memory will be allocated.

2) Runtime Declaration: C language and many more language provide us the feature of dynamic memory allocation by keywords "Malloc and Calloc."

So in this special block of memory is allocated in memory and their lifetime is also different.

3) Polymorphic operations: In most language, some special symbol like + to designate any one of the several different operation which depends on type of data or argument is provided.

In this operation has some name like as we discussed + in this case operation symbol is said to be overloaded because it does not designate one specific operation.

Ada: allows programmer to overload subprograms.

ML: Expands this concept with full polymorphism where function has one name but variety of implementation depending on the types of arguments.

4) Type checking:- Declaration is basically for static type checking rather than dynamic.

Q3. EXPLAIN HOW VALUES ARE ASSIGN TO DATA TYPES?

Assignment of Data Types

Assignment: A basic operation for changing the binding of a value to the data object. **Languages like C, LISP and many more**

Assignment also returns a value, which is the data object containing a copy of the value assigned.

In Pascal: dfont>

Assignment(:=): integer1*integer2 -> void

Value of integer2 is copied in integer1 In C Language:

Assignment(=):integer1*integer2 ->integer3

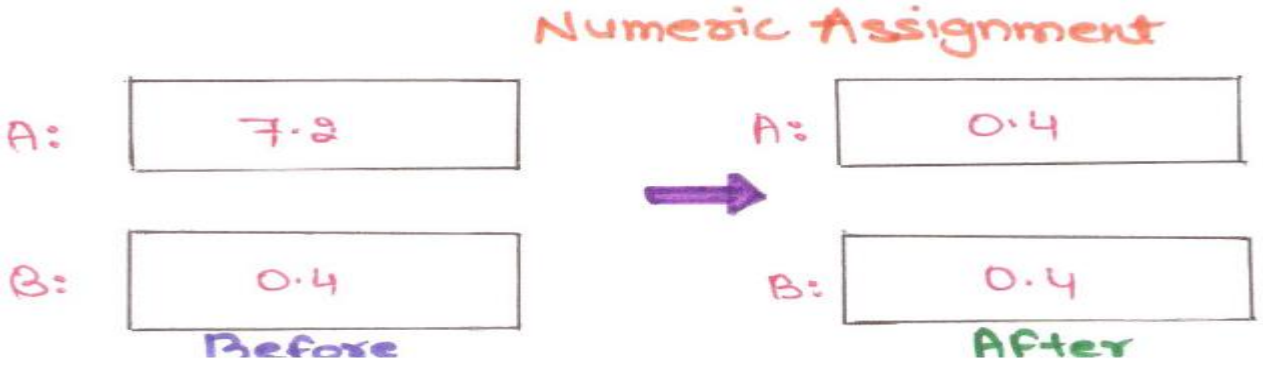
With this action: Set the value contained in data object integer1 to be a copy of the value contained in the data object integer2 and also create and return a new data object integer3, containing a copy of a new value of integer2.

Two Concepts through which we can define assignment

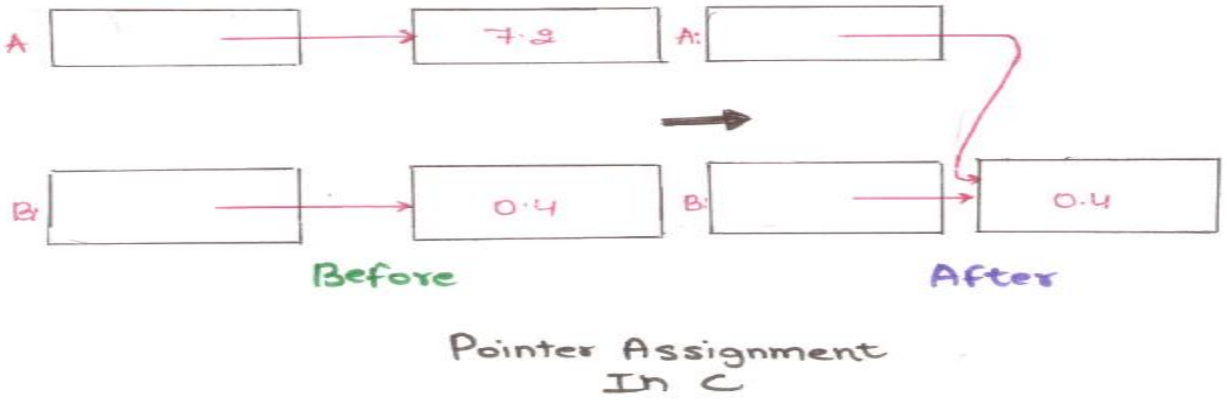
L-Value: Location for an object

R-Value: Content at that location

Using L-Value and R-Value gives a more concise way to describe expression semantics.
Example in case of Integer:
A=B
In this copying the value of variable B to variable A. i.e assign to the value L-Value of A the R-Value of B.
In case of pointer
A=B:
In this A & B are pointers variables. If B is a pointer then B's R-value is the L- value of some other data object. This assignment then means,
"Make the r-value of A refers to the same data object as the r-value of B"
Thus, the assignment A=B means " Assign a copy of the pointer stored in variable B variable A".



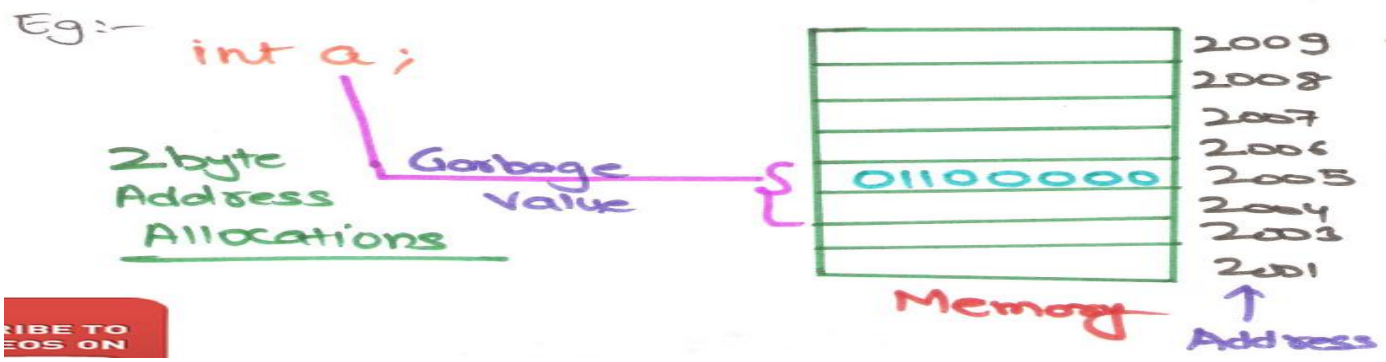
Copy value:(pascal)
A:=B



Pointer Assignment In C

Two views of assignment:
Copy pointer: (SNOBOL)
A=B(ptr to value of variable B assigned to variable A)

Initialization of Data Types
Initialization is basically a step in which we just specify the name of variable and data object but not yet assigned a value **(i.e , an L-Value with no corresponding R-value).**
- in this only block for storage is allocated.
- that block automatically may take some value in form of binary code.
- it is serious programming error to create an uninitialised error as it becomes difficult to distinguish between original value or automatically filled value both of them are of bit patterns.



Scalar Data Type Enumeration

An Enumerated data type is a data type whose domain values are given in a list or ordered list and who's only operations are equality and assignment.

or

An Enumeration is an ordered list of distinct values.

or

An Enumeration is a complete ordered listing of all items in a collection.

Pascal was first language which introduced enumeration. To make enumeration facility useful, a programming language must provide a mechanism for declaring and defining the new data type and for declaring variables whose value will come from the element of type.

It is assumed that this literals are distinct and does equality can be directly defined.

"Before an era of enumeration what we had ?"

For example: A variable student class might have only 4 possible values representing fresher, sophomore, junior and senior. Similarly, a variable StudentSex might have only two values representing Male and Female.

Before the contact of enumeration the language like

Fortran or Cobol such variables is declared as integer type and distinct values are assigned. like

fresher=1 , sophomore=2, and so on

and male=0, female =1

Then translator manipulate values as integers.

That creates big problem like

Sophomore =1 and female=1

As both have some values can we apply integer based operation on it. As a point of view of programmer it should not be but according to translator it can apply as they are of integer types.

Then languages such as C, article Pascal and Ada includes an Enumeration data type that allows the programmer to define and manipulate such variables directly.

Specification of Enumeration

The programmers defined both the literal name to be used for the values and their ordering using a declaration such as in pascal.

Type months=(jan,feb, mar, apr, june, jul, aug, sep, oct,nov,dec);

In C

enum studentclass{ fresh, soph, junior, senior};

enum studentsex {male, female};

In Pascal, C example can be written as

type class =(fresh, soph, junior, senior};

Followed by declaration for variables such as

Studentclass: Class;

Studentsex class: Class;

Here type definition introduces the type name class, which may be used wherever the primitive type name such as integer might be used time.

It also introduces the **literals of fresh, soph, junior, senior** which may be used wherever a language- defined literal such as "27" might be used. Thus we can write.

if studentclass= junior then.....

Instead of the less understandable

if studentclass= 3 then .....

Which would be required if integer variables were used. Static compiler can find error such as

if student class= Male then

As Male is part of student class. Operations which we can perform-

- Relational operations(equal, less-than, greater-than,etc)
- Assignment
- Successor and Predecessor

Implementation of Enumeration

- Each value in the enumeration sequence is represented at run-time by one of the integers 0,1,2,..... as only a small set of values is involved and the values are never negative.
- In this integer representation is often shortened to omit the sign bit and use only enough bits for the range of values required, as with the sub-range values.
- Only and maximum 2 bits are required to represent the senior=3 in memory because 3=11(binary)/ 2 bits only

In C, the programmer main override default and set any values desired for enumeration values for example.

Enum class{ fresh=74, soph=89, junio=7, senior=28}

With this storage representation for enumeration types. Relational operations such as **=,>, and <** may be implemented.

Scalar Data Types: Booleans

The Boolean data type is a data type, having two values(usually denoted true or false), intended to represent the truth values of logic and Boolean algebra.

Specification: In Pascal and Ada, the Boolean data type is considered simply a language - defined enumeration, viz;

type Boolean=(false, true);

Which both defines the names true and false for the values of the types and define ordering false<true

Common Operations in boolean are

and : Boolean*Boolean->Boolean(conjunction)

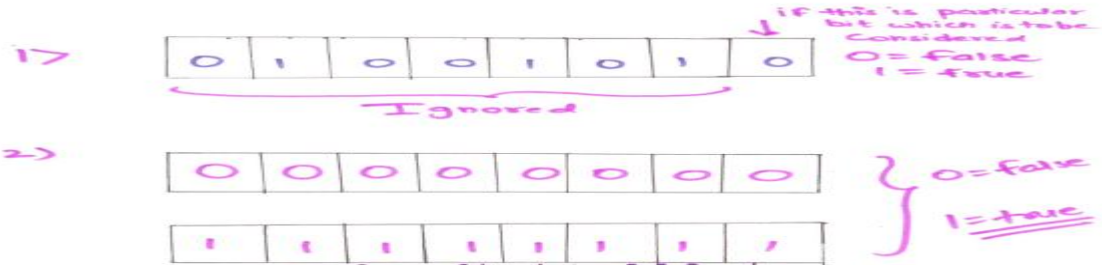
or : Boolean*Boolean->Boolean(inclusive disjunction)

not : Boolean ->Boolean(negative or complement)

Implementation of boolean data type:

Single bit of storage is provided, no descriptor designated the data type is needed. Because single bit may not be separately addressable in memory which often takes a byte or word to represent it if extended. Then the value true and false might be represented in two ways within the storage unit:

- 1) a particular bit is used for the value(often the sign bit of the number representation), with **'0= false', '1=true'**, and the rest of the byte or word ignored, or
- 2) A zero value in the entire storage unit represent False, and any other non zero value represents **true**



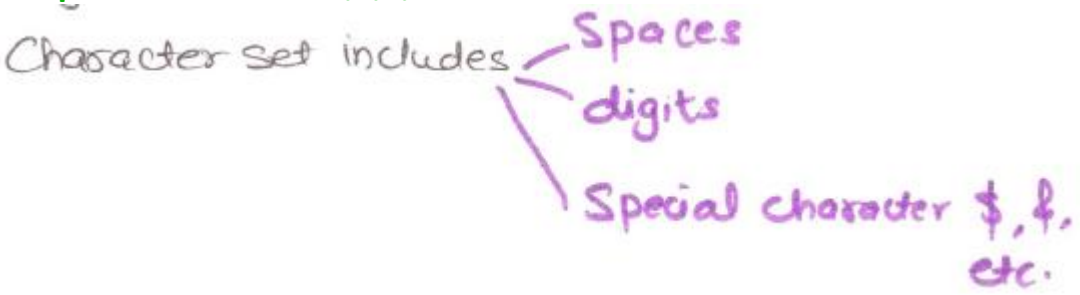
Scalar Data Types: Characters

Specification of Characters

A character data type provides data objects that have a single character as their value. **Set of values in character data type depends upon "Hardware and Operating System"** like ASCII character set. and ordering of the characters in this character set is called **Collecting Sequence** And ordering given by the **'Relational Operations'.**

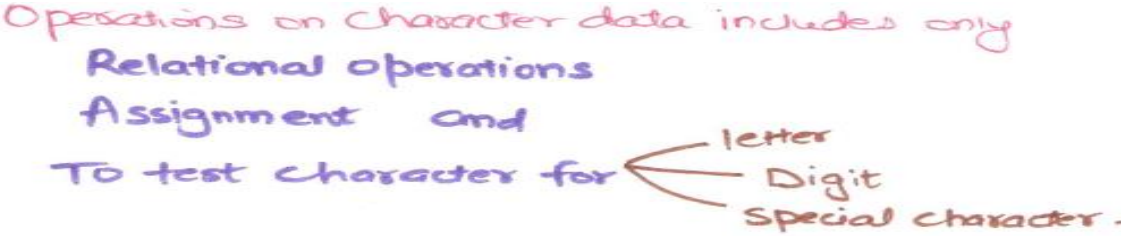
Character set includes

- **Spaces**
- **Digits**
- **Special character @,#,$,& etc.**



Operations on character data include only

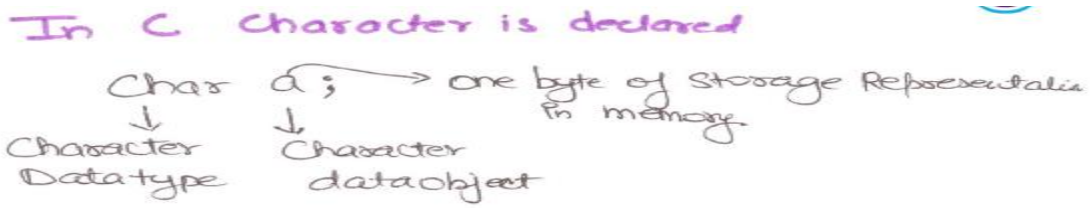
- **Relational operations**
- **Assignment and**
- **To test character for- Letter, Digit, Special Character.**



Implementation of Character Data Type

Character data values are almost directly supported by the underlying hardware and operating system because their use in input-output.

In C Character is declared



ASCII value of character data type in C are

0 to 9 is 48 to 57/ ASCII value

A to z is 65 to 90

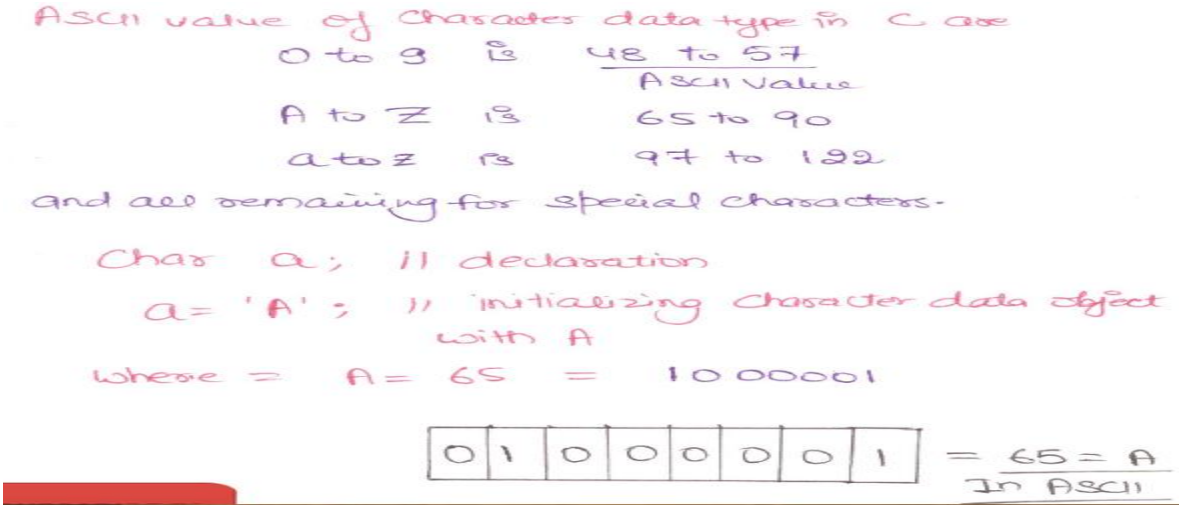
a to z is 97 to 122

And all remaining for special characters.

char a ; //declaration

a='A'; // initializing character data object with A

Where= A=65=1000001



Q4. WRITE SHORT NOTES ON DATA AGGREGATES AND TYPE CONSTRUCTORS?

Data Aggregates and Type Constructors:

Programming languages allow the programmer to specify aggregations of elementary data objects and, recursively, aggregations of aggregates. This is done by providing a number of *constructors*. The resulting objects are called *compound objects*. A well-known example is the array constructor.

An aggregate object has a unique name. Older programming languages, such as FORTRAN and COBOL, provided only a limited number of constructors. For example, FORTRAN only provided the array constructor; COBOL only provided the record constructor. In addition, through constructors, they simply provided a way to define a new *single* aggregate object, not a type. Later languages, such as Pascal, allowed
new compound types to be defined by specifying them as aggregates of simpler types.

- PLs provide programmers with ***constructors*** for specifying ***aggregates*** of elementary data
  objects. Such constructed objects are called ***compound objects***.
- Example: the array constructor which constructs aggregates of homogeneous-type elements.

- These aggregate objects have unique names.

- In most cases operations can be performed on single elementary components
  - A selector is used to specify a particular element

- FORTRAN - array constructor

- COBOL - record constructor

- Pascal- allowed new compound types to be defined by specifying them as aggregates of simpler types

- Modern PLs have various constructors used to define ***compound data types*** by specifying them as aggregates of simpler types.

Data Constructors
Cartesian Product
The Cartesian product of two sets *A* and *B* (also called the product set, set direct product, or cross product) is defined to be the set of all points (*a, b*) where ***a***□□ and ***b***□□. It is denoted and is called the Cartesian product since it originated in Descartes' formulation of analytic geometry. In the Cartesian view, points in the plane are specified by their vertical and horizontal coordinates, with points on a line being specified by just one coordinate. The

main examples of direct products are Euclidean three-space (, where are the real numbers), and the plane ( ).
The Cartesian product of n sets A_1, A_2, A_3, ..., A_n ,denoted as A_1 A_2 A_3 ... A_n , is a set whose elements are ordered n-tuples (a_1, a_2 ,a_3..., a_n) where a_k A_k.

Example: A regular polygon may be defined by the number of edges (an integer) and the length of each side (a real). Thus, a regular polygon would be an element in the Cartesian product **integer real**

Finite Mapping
A mathematical function is a rule for associating (or mapping) values from one set called the ***domain*** of the function, to values in another set called the ***range*** of the function.

A finite mapping is a function from a finite set of values of a domain type DT onto values of a range type RT. Such function may be defined in programming languages through the use of the mechanisms provided to define routines.

A function f, which maps from the set of integers to the set of reals, may be denoted as: **f:integer real**
A ***finite mapping*** is a function from a finite set of values of a domain type DT onto values of a range type RT

- The array constructor can be used to specify a finite mapping as a data aggregate
  - The array associates (maps) each array index to the value stored at that index.
  - The index range is the domain
  - The values stored in the array are the range
  - Such a definition is called extensional since all values of the function are explicitly enumerated
  - Example in C
    - **char digits[10];** defines a mapping from the subset of integers 0..9 to the set of characters.
    - **for (i=0; i < 10; ++i) digits[i] = ' ';** defines a specific such mapping

Q5. WRITE SHORT NOTES ON USER-DEFINED DATA TYPES AND ABSTRCT DATA TYPES?

User-defined types and abstract data types:

Modern programming languages provide many ways of defining new types, starting from built-in types.
Complex data structures to be composed out of the built-in types of the language. Modern languages also allow aggregates built through composition of built-in types to be named as

new types. Having given a type name to an aggregate data structure, one can declare as many variables of that type asnecessary by simple declarations.

For example, after the C declaration which introduces a new type name complex
struct complex {
float real_part, imaginary_part;
}
any number of instance variables may be defined to hold complex values:

Construct that allows both a data structure and operations to be specified for user defined types.
An *abstract data type* is a new type for which we can define the operations to be used for manipulating instances, while the data structure that implements the type is hidden to the users. In what follows we briefly review the constructs provided by C++ and by Eiffel to define abstract data types.

A *constructor* is an operation that has the same name of the new type being defined (in the example, point). A constructor is automatically invoked when an object of the class is allocated. In the case of points p1 and p2, this is done automatically when the scope in which they are declared is entered. In the case of the dynamically allocated point referenced by p3, this is done when the new instruction is executed. Invocation of the constructor allocates the data structure defined by the class and initializes its value according to the
constructor's code.

A special type of constructor is a *copy constructor*. The constructor we have seen for point builds a point out of two int values. A copy constructor is able to build a point out of an existing point. The signature of the copy constructor would be:
point (point&)

The copy constructor is fundamentally a different kind of constructor because it allows us to build a new object from an existing object without knowing the components that constitute the object. That is what our first constructor does. When a parameter is passed by value to a procedure, copy construction is used to build the formal parameter from the argument. Copy construction is almost similar to assignment with the difference that on assignment, both objects exist whereas on copy construction, a new object must be created first and then a value assigned to it.

It is also possible to define *generic abstract data types*, i.e., data types that are parametric with respect to the type of components.

Abstract data types in C++

Abstract data types can be defined in C++ through the class construct. A class encloses the definition of a new type and explicitly provides the operations that can be invoked for correct use of instances of the type.

Type systems:

Types are a fundamental semantic concept of programming languages. Moreover programming languages differ in the way types are defined and behave, and typing issues are often quite subtle. Having discussed type concepts informally in different languages so far, we now review the foundations for a theory of types.

The goal is to help the reader understand the *type system* adopted by a language, defined as the set of rules used by the language to structure and organize its collection of types. Understanding the type system
adopted by a language is perhaps the major step in understanding the language's semantics.

A type is defined as a set of values and a set of operations that can be applied to such values. As usual, since values in our context are stored somewhere in the memory of a computer, we use the term *object* (or *data object*) to denote both the storage and the stored value. The operations defined for a type are the only way of manipulating its instance objects: they protect data objects from any illegal uses. Any attempt to manipulate objects with illegal operations is a *type error*. A program is said to be *type safe* (or *type secure*) if all operations in the program are guaranteed to always apply to data of the correct type, i.e., no type errors will ever occur.

Q6. EXPLAIN HOW DATA TYPE IS CHEKING. WRITE DIFFERENCES BETWEEN STATIC VERSUS DYNAMIC CHECKING?

Static versus dynamic program checking

Errors can be classified in two categories: language errors and application errors. *Language errors* are syntactic and semantic errors in the use of the programming language. *Application errors* are deviations of the program behavior with respect to specifications (assuming specifications capture the required behavior correctly). The programming language should facilitate both kinds of errors to be identified and removed. Ideally, it should help prevent them from being introduced in the program.

Error checking can be accomplished in different ways, that can be classified in two broad categories: static and dynamic. *Dynamic checking* requires the program to be executed on sample input data. *Static checking* does not. Ingeneral, if a check can be performed statically, it is preferable to do so instead of delaying the check to run-time for two main

reasons. First, potential errors are detected at run time only if one can provide input data that cause the error to be revealed.

Static checking is often called compile-time (or translation-time) checking. Actually, the term “compile-time checking” may not be an accurate synonym of “static checking”, since programs may be subject to separate compilation and some static checks might occur at link time. For example, the possible mismatch between a routine called by one module and defined in another might be checked at link time.

Strong typing and type checking

The type system of a language was defined as the set of rules to be followed to define and manipulate program data. Such rules constrain the set of legal programs that can be written in a language. The goal of a type system is to prevent the writing of type unsafe programs as much as possible. A type system is said to be *strong* if it guarantees type safety; i.e., programs written by following the restrictions of the type system are guaranteed not to generate type errors.

A language with a strong type system is said to be a *strongly typed language*. If a language is strongly typed, the absence of type errors from programs can be guaranteed by the compiler. A type system is said to be
weak if it is not strong. Similarly, a *weakly typed language* is a language that is not strongly typed.

Example of a static type system can be achieved by requiring that
1. only built-in types can be used;
2. all variables are declared with an associated type;
3. all operations are specified by stating the types of the required operands and the type of the result.

A statically typed language is a strongly typed language, but there are strongly typed languages that are not statically typed.

Q7. WRITE SHORT NOTES ON TYPE COMPATIBILITY?

Type compatibility

A strict type system might require operations that expect an operand of a type T to be invoked legally only with a parameter of type T. Languages, however, often allow more flexibility, by defining when an operand of another type– say Q–is also acceptable without violating type safety.

Type compatibility is also sometimes called *conformance* or *equivalence*. When compatibility is defined precisely by the type system, a type checking procedure can verify that all operations are always invoked correctly, i.e., the types of the operands are

compatible with the types expected by the operation. Thus a language defining a notion of type compatibility can still have a strong type system.

Structural compatibility is another possible conformance rule that languages may adopt. Type T1 is structurally compatible with type T2 if they have the same structure. This can be defined recursively as follows:

• T1 is name compatible with T2; or

• T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

According to structural equivalence, instructions (1), (2), and (3) are type correct. Instruction (4) contains a type error, since type s3 is not compatible with int. Note that the definition we gave does not clearly state what happens with the field names of Cartesian products (i.e., whether they are ignored in the check or they are required to coincide and whether structurally compatible fields are required to occur in the same order or not). For simplicity, we assume that they are required to coincide and to occur in the same order. In such a case, if we rename the fields of s2 as y1 and w1, or permute their occurrence, s2 would no longer be compatible with s1.

Name compatibility is easier to implement than structural compatibility, which requires a recursive traversal of a data structure. Name compatibility is also much stronger than structural compatibility. Actually, structural compatibility goes to the extreme where type names are totally ignored in the check.

Structural compatibility makes the classification of data objects implied by types exceedingly coarse.

For example, having defined the following two types:

```
struct complex {
float a;
float b;
};
struct point {
float a;
float b;
};
```

The programmer can instantiate variables to represent–say–points on a plane and values of a.c. voltage. The type system allows to use them interchangeably, although most likely the programmer has chosen two different type names in order to keep the different sets of objects separate. In conclusion, name compatibility is often preferable. It prevents two types to be considered compatible just because their structure happens to be identical by coincidence.

Type conversions
For some languages any required conversions are applied automatically by the compiler. Following the Algol 68 terminology, we will call such automatic conversions *coercions*.

Types and subtypes
If a type is defined as a set of values with an associated set of operations, a subtype can be defined to be a subset of those values (and, for simplicity, the same operations)

Pascal was the first programming language to introduce the concept of a subtype, as a subrange of any discrete ordinal type (i.e., integers, boolean, character, enumerations, or a subrange thereof). For example, in Pascal one may define natural numbers and digits as follows:
type natural = 0. .maxint;
digit =0. .9;
small = -9. .9;
where maxint is the maximum integer value representable by an implementation.

A subtype of an array type can constrain its index; a subtype of a variant record type can freeze the variant; a subtype of a discrete ordinal type is a finite subset of contiguous values.

Generic types: As we mentioned, modern languages allow parameterized (generic) abstract data types to be defined.

A typical example is a stack of elements of a parameter type T, whose operations have the following signatures:
push: stack (T) x T -> stack (T) --pushes an element on top of the stack
pop: stack (T) -> stack (T) x T --extracts the topmost element from the stack
length: stack (T) -> int --compute the length of the stack

Q1. EXPLAIN ABOUT EXPRESSIONS AND STATEMENTS. HOW EXPRESSIONS ARE SPECIFIED IN PROGRAMMING LANGUAGE?

In programming language terminology, an "expression" is a combination of values and functions that are combined and interpreted by the compiler to create a new value, as opposed to a "statement" which is just a standalone unit of execution and doesn't return anything.

A series of statements always implies a specific order of evaluation, which means that a statement cannot be understood without looking at prior statements. But with pure expressions, the sub expressions do not have any implied order of execution or dependencies.

So in the expression a+b, if both the 'a' and 'b' parts are pure, then the 'a' part can be isolated, understood, tested and evaluated on its own, as can the 'b' part.

Expressions and statements

Expressions define how a value can be obtained by combining other values through operators. The values from which expresions are evaluated are either denoted by a literal, as in the case of the real value 57.73, or they are the r_value of a variable.

Operators appearing in an expression denote mathematical functions. They are characterized by their *aritiy* (i.e., number of operands) and are invoked using the function's signature. A unary operator is applied to only one operand. A binary operator is applied to two operands. In general, a n-ary operator is applied to n operands. For example, '-' can be used as a unary operator to transform–say–the value of a positive expression into a negative value. In general, however, it is used as a binary operator to subtract the value of one expression from the value of another expression. Functional routine invocations can be viewed as n-ary operators, where n is the number of parameters.

Regarding the operator's notation, one can distinguish between infix, prefix, and postfix. *Infix notation* is the most common notation for binary operators: the operator is written between its two operands, as in x + y. Postfix and prefix notations are common especially for non-binary operators. In *prefix notation*, the operator appears first, and then the operands follow. In *postfix notation* the operands are followed by the Corresponding operator.

For example,
the infix expression
a * (b + c)
can be written in prefix form as
* a + b c
and in postfix form as

a b c + *

In C, the increment and decrement unary operators ++ and -- can be written both in prefix and in postfix notation. Infix notation is the most natural one to use for binary operators, since it allows programs to be written as conventional mathematical expressions.

In C, Ex: In equation a = b < c, operator "less than" (<) has higher precedence than "equal" (==). Thus,
first b < c is evaluated. Some languages, like C++ and Ada, allow operators to be programmer defined.

Some programming languages support the ability of writing conditional expressions, i.e., expressions that are composed of sub expressions, of which only one is to be evaluated, depending on the value of a condition.

Functional programming languages are based heavily on expressions. In such languages, a program is itself an expression, defined by a function applied to operands, which may themselves be defined by functions applied to operands.

Statements
A statement is an instruction that the interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script
print 1
x = 2
print x
produces the output
1
2
Again, the assignment statement produces no output.

Evaluating expressions
An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:
>>> 1 + 1
2

Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

Confusingly, evaluating an expression is not quite the same thing as printing a value.

```
>>> message = 'Hello, World!'
>>> message
'Hello, World!'
>>> print message
Hello, World!
```

When the interpreter displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string without the quotation marks.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
'Hello, World!'
1 + 1
```

Conditional execution and iteration

Conditional execution of different statements can be specified in most languages by the if statement.

Choosing among more than two alternatives using only if-then-else statements may lead to awkward constructions, such as

```
if a
then S1
else
if b
then S2
else
if c
then S3
else S4
end
end
end
```

To solve this syntactic inconvenience, Modula-2 has an else-if construct that also serves as an end bracket for the previous if. Thus the above fragment may be written as

if a
then S1
else if b
then S2
else if c
then S3
else S4
end

some examples using a C-like language

```
public void IfThenElseStatement(bool aBool)
{
  int result;    //what is the value of result before it is used?    if (aBool)
  {
    result = 42; //what is the result in the 'else' case?    }
  Console.WriteLine("result={0}", result);
}
```

Because the “if-then” is a statement, the result variable must be defined *outside* the statement and but assigned to *inside* the statement, which leads to some issues:

- The result variable has to be set up outside the statement itself. What initial value should it be set to?
- What if I forget to assign to the result variable in the if statement? The purpose of the “if “statement is purely to have side effects (the assignment to the variables). This means that the statements are potentially buggy, because it would be easy to forget to do an assignment in one branch. And because the assignment was just a side effect, the compiler could not offer any warning. Since the result variable has already been defined in scope, I could easily use it, unaware that it was invalid.
- What is the value of the result variable in the “else” case? In this case, I haven’t specified a value. Did I forget? Is this a potential bug?
- Finally, the reliance on side-effects to get things done means that the statements are not easily usable in another context (for example, extracted for refactoring, or parallelizing) because they have a dependency on a variable that is not part of the statement itself.

Here is a similar example of statements vs. expressions using a loop statement

```
public void LoopStatement()
{
    int i;   //what is the value of i before it is used?    int length;
```

```
    var array = new int[] { 1, 2, 3 };
    int sum;  //what is the value of sum if the array is empty?
    length = array.Length;   //what if I forget to assign to length?    for (i = 0; i < length;
i++)
    {
       sum += array[i];
    }

    Console.WriteLine("sum={0}", sum);
}
```

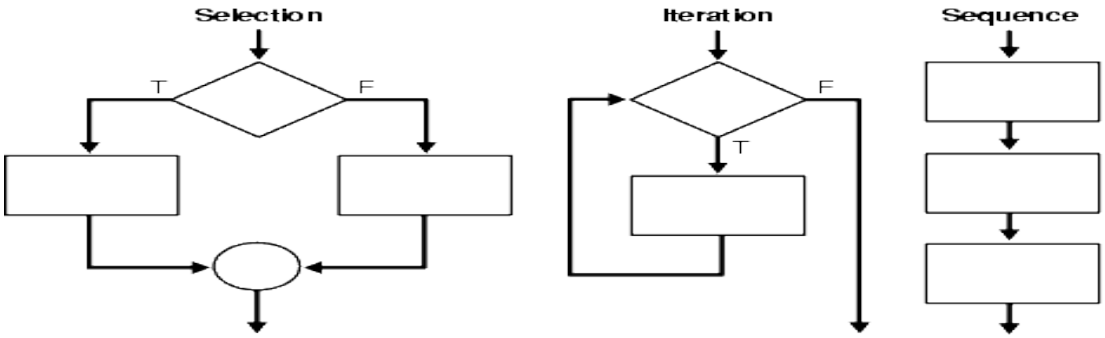
Most languages also provide an ad-hoc construct to express multiple-choice selection. For example, C++ provides the switch construct, illustrated by the following fragment:

```
switch (operator) {
case '+':
result = operand1 + operand2;
break;
case '*':
result = operand1 * operand2;
break;
case '-':
result = operand1 - operand2;
break;
case '/':
result = operand1 / operand2;
break;
default:
break; --do nothing
};
```

Each branch is labelled by one (or more) constant values. Based on the value of the switch expression, the branch labelled by the same value is selected. If the value of the switch expression does not match any of the labels, the (optional) default branch is executed. If the default branch is not present, no action takes place.

Any computer program can be written using the basic control structures shown in Figure 3-1. They can be combined in any way necessary to deal with a given problem.

Figure 3-1 Control Structures



The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A *condition* is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

IF-THEN

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
  sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
  compute_bonus(empid);
  UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

You might want to place brief IF statements on a single line, as in

```
IF x > y THEN high := x; END IF;
```

IF-THEN-ELSE

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
```

```
  sequence_of_statements1
ELSE
  sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
  UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
  IF new_balance >= minimum_balance THEN
    UPDATE accounts SET balance = balance - debit WHERE ...
  ELSE
    RAISE insufficient_funds;
  END IF;
END IF;
```

IF-THEN-ELSIF

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If

all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example:

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

Iterative Control: LOOP and EXIT Statements

Iteration allows a number of actions to be executed repeatedly. Most programming languages provide different kinds of *loop constructs* to define iteration of actions (called the *loop body*). Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met. The former kind of loop is usually called a *for loop*; the latter is often called the *while loop*.

For-loops are named after a common statement provided by languages of the Algol family. For statements define a *control variable* which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
  sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT;  -- illegal
  END IF;
END;
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement. For more information, see "Using the RETURN Statement".

EXIT-WHEN

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

```
LOOP
  FETCH c1 INTO ...
  EXIT WHEN c1%NOTFOUND;  -- exit loop if condition is true
  ...
END LOOP;
CLOSE c1;
```

Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.

The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:

```
IF count > 100 THEN     |     EXIT WHEN count > 100;
  EXIT;                 |
END IF;
```

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. An example follows:

```
WHILE total <= 25000 LOOP
  ...
  SELECT sal INTO salary FROM emp WHERE ...
  total := total + salary;
END LOOP;
```

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed.

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. (Cursor FOR loops, which iterate over the result set of a cursor, are discussed in Chapter 5.) The range is part of

an *iteration scheme*, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
  sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated.

As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP  -- assign the value 3 to i
  sequence_of_statements  -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

```
FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

A programming language might provide support for user-defined control structures, in much the same way as it provides support for user-defined types and operations. For example, having defined a set, the programmer might need to sequence through all elements in the set. User-defined control structures which sequence through elements of user-defined collections are sometimes called *iterators*.

Languages providing constructs for the implementation of abstract data types easily allow iterators to be defined. For example, in C++ let the generic "collection of elements of type T" be defined by a template. To define an iterator, we can design three operations that are

exported by the template: start ( ), which initializes the loop by positioning a cursor on the first element of the collection (if any), more ( ), which yields true if there are elements left to examine in the collection, and next ( ), which yields the current element and positions the cursor on the next element of the collection (if any).

Routines:

Routines are a program decomposition mechanism which allows programs to be broken into several units. Routine calls are control structures that govern the flow of control among program units. The relationships among routines defined by calls are asymmetric: the caller transfers control to the callee by naming it explicitly. The callee transfers control back to the caller without naming it. The unit to which control is transfered when a routine R terminates is always the one that was executing immediately before R. Routines are used to define abstract operations.

Most modern languages allow such abstract operations to be defined recursively. Moreover, many such languages allow generic operations to be defined.

Most languages distinguish between two kinds of routines: procedures and functions. A *procedure* does not return a value: it is an abstract command which is called to cause some desired state change.

A *function* corresponds to its mathematical counterpart: its activation is supposed to return a value, which depends on the value of the transmitted parameters.

Pascal provides both procedures and functions. It allows formal parameters to be either by value or by reference. It also allows procedures and functions to be parameters, as shown by the following example of a procedure header:

procedure example (**var** x: T; y: Q; **function** f (z: R): integer);
In the example, x is a by-reference parameter of type T; y is a by-value parameter of type Q; f is a function parameter which takes one by-value parameter z of type R and returns an integer.

Ada provides both procedures and functions. Parameter passing mode is specified in the header of an Ada routine as either in, out, or in out.

In C all routines are functional, i.e., they return a value, unless the return type is void, which states explicitly that no value is returned. Parameters can only be passed by value. It is possible, however, to achive the effect of call by reference through the use of pointers. For example, the following routine

```
void proc (int* x, int y);
{
*x = *x + y;
```

}
increments the object referenced by x by the value of y. If we call proc as follows
proc (&a, b); /* &a means the address of a */
x is initialized to point to a, and the routine increments a by the value of b.

Exceptions:

Programmers often write programs under the optimistic assumption that nothing will go wrong when the program executes. Unfortunately, however, there are many reasons which may invalidate this assumption. For example, it may happen that under certain conditions an array is indexed with a value which exceeds the declared bounds. An arithmetic expression may cause a division by zero, or the square root operation may be executed with a negative argument.

A request for new memory allocation issued by the run-time system might exceed the amount of storage available for the program execution. Or, finally, an embedded application might receive a message from the field which overrides a previously received message, before this message has been handled by the program.

Often programs fail unexpectedly, maybe simply displaying some obscure message, as an erroneous program state is entered. This behavior, however, is unacceptable in many cases. To improve reliability, it is necessary that such erroneous conditions can be recognized by the program, and certain actionsare executed in response to the error.

To do so, however, the conventional control structures we have discussed so far are simply inadequate. For example, to check that an index never exceeds the array bounds, one would need to explicitly test the value of the index before any indexing takes place, and insert appropriate response code in case the bounds are violated.

Alternatively, one would like the run-time machine to be able to trap such anomalous condition, and let the response to it be programmable in the language. This would be more efficient under the assumption that bound violations are the
exceptional case.

To cope with this problem, programming languages provide features for exception handling. According to the standard terminology, an *exception* denotes an undesirable, anomalous behavior which supposedly occurs rarely. The language can provide facilities to define exceptions, recognize them, and specify the response code that must be executed when the exception is raised (*exception handler*).

Exceptions have a wider meaning than merely computation errors. They refer to any kind of anomalous behavior that, intuitively and informally, corresponds to a deviation from the expected course of actions, as envisioned by the programmer. The concept of "deviation"

cannot be stated in absolute and rigorous terms. It represents a design decision taken by the programmer, who decides that certain states are "normal", and "expected", while others are "anomalous". Thus, an exception does not necessarily mean that we are in the presence of a catastrophic error. It simply means that the unit being executed is unable to proceed in a manner that leads to its normal termination as specified by the programmer.

Earlier programming languages (except PL/I) offered no special help in properly handling exceptional conditions. Most modern languages, however, provide systematic exception-handling features. With these features, the concern for anomalies may be moved out of the main line of program flow, so as not to obscure the basic algorithm.

To define exception handling, the following main decisions must be taken by a programming language designer:
4.What are the exceptions that can be handled? How can they be defined?
5.What units can raise an exception and how?
6. How and where can a handler be defined?
7. How does control flow after an exception is raised in order to reach its handler?
8.Where does control flow after an exception has been handled?

The solutions provided to such questions, which can differ from language to language, affect the semantics of exception handling, its usability, and its ease of implementation

Exception handling:
Ada provides a set of four predefined exceptions that can be automatically trapped and raised by the underlying run-time machine:
• Constraint_Error: failure of a run-time check on a constraint, such as array index out of bounds, zero right operand of a division, etc.;
• Program_Error: failure of a run-time check on a language rule. For example, a function is required to complete normally by executing a return statement which transmits a result back to the caller. If this does not happen, the exception is raised;
• Storage_Error: failure of a run-time check on memory avaliability; for example, it may be raised by invocation of new;
• Tasking_Error: failure of a run-time check on the task system

begin --this is a block with exception handlers
... statements ...
exception when Help =>*handler for exception Help*
when Constraint_Error => *handler for exception*
Constraint_Error, which might be raised by a
division by zero
when others => *handler for any other exception that is not Help*
nor Constraint_Error
end;

In the example, a list of handlers is attached to the block. The list is prefixed by the keyword **exception**, and each handler is prefixed by the keyword **when**.

PL/I was the first language to introduce exception handling. Exceptions are called CONDITIONS in PL/I. Exception handlers are declared by ON statements:
ON CONDITION (exception_name) exception_handler
where exception_handler can be a simple statement or a block. An exception is explicitly raised by the statement
SIGNAL CONDITION (exception_name);
The language also defines a number of built-in exceptions and provides system- defined handlers for them. Built-in exceptions are automatically raised by the execution of some statements (e.g., ZERODIVIDE is raised when a divide by zero is attempted). The action performed by a system-provided handler is specified by the language.

When an exception is raised (either automatically or by a SIGNAL statement), the handler currently bound to the exception is executed as if it were a subprogram invoked explicitly at that point. Therefore, unless otherwise specified by the handler, control subsequently will return to the point that issued the SIGNAL.

PL/I does not allow the programmer to pass any information from the point raising the exception to the exception handler. If this is necessary, the programmer must resort to global variables, which can be an unsafe programming practice.

PL/I exception-handling mechanisms can be complicated further by explicitly enabling and disabling built-in exceptions; user-defined exceptions cannot be disabled, because they must be explicitly signaled anyway.

Most built-in exceptions are enabled by default and bound to the standard system-provided
error handler. Enabling a previously disabled exception can be specified by prefixing a statement, block, or
procedure with the exception name.
For Example:
(ZERODIVIDE) : BEGIN
. . .
END;
The scope of the prefix is static; it is the statement, block, or procedure to which it is attached. An enabled exception can be explicitly disabled by prefixing a statement, block, or procedure with NO exception_name.

Pattern matching:

Pattern matching is a high level way of stating conditions, based on which, different actions are specified to occur. Pattern matching is the most important control structure of the string manipulation programming language SNOBOL4.

Pattern matching is also provided by most modern functional programming languages, like ML, Miranda, SASL, and is also provided by the logical language PROLOG and by rule-based systems.

Let us start by discussing the following simple definitions of a data type and a function:
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun day_off (Sun) = true
|day_off (Sat)= true
|day_off (_) = false

In the example, function day_off is defined by a number of cases. Depending on the value of the parameter with which the function will be invoked, the appropriate case will be selected. Cases are checked sequentially, from the first one on. If the first two cases are not matched by the value of the parameter
with which the function is called, the third alternative will be selected, since the so-called wild card "_" matches any argument.

As the example shows, pattern matching has a twofold effect. On the one hand, it chooses the course of action based on the argument; on the other, since the pattern can be an expression with variables, it binds the variables in the pattern (if any) with the values that match. The same bound variables can then be used in the expression that defines the value of the function. Pattern matching can thus be viewed as a generalization of conventional parameter passing. The value of actual parameters is used to match the pattern appearing in the formal parameter part. Thus the case selected by pattern matching can vary from call to call.

Non-determinism and backtracking:
Problem solutions can often be described via and-or decompositions into subproblems. For example, to solve problem A, one needs to solve either B, C, orD; to solve–say–C, one needs to solve E, F, and G. This can be represented asan *and/or tree*

Node A, which has no incoming arcs, is called
a *root node*; nodes B, D, E, F, and G, which have no exiting arcs, are called *leaf nodes*. And/or decompositions can also be described–in the hypothetical syntax of some programming language–as

A if B or
C or
D;
C if E and
F and
G;

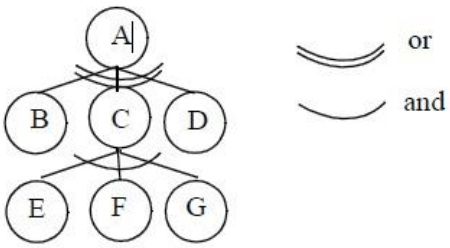


FIGURE 52.And/or tree

The solution of A is described as a *disjunction* of subproblems; the solution of C is described as a *conjunction* of subproblems. We can further assume B, D, E, F, and G to be problem solving routines, which can terminate in either a success or a failure state.

If the order in which subproblems are solved is unspecified (and irrelevant as far as the problem statement is concerned), we say that the program is *nondeterministic*.

An and/or problem decomposition can be viewed as a high-level design of a problem solution, which is then implemented in any progrmming language using the conventional constructs it provides. However, there are programming languages (like logic languages of the Prolog family or the string manipulation language Icon) which support this way of decomposing problems directly.

Since features of this kind are very high level, a programming language incorporating them is extremely powerful. As one can imagine, however, these features are hard to implement efficiently.

If a subproblem solution fails, backtracking implies that another possible sub problem solution be tried. This ensures that the overall problem solution fails only if there is no way of solving the problem without failure. Thus, through backtracking, one needs to guarantee completeness of the search.

Event-driven computations:

In some cases, programs are structured conveniently as *reactive* systems, i.e., systems where certain *events* occurring in the environment cause certain program fragments to be executed. An example is provided by modern user interfaces where a number of small graphical devices (called *widgets*) are often displayed to mediate human-computer

interaction. By operating on such widgets (e.g., by clicking the mouse on a push-button) the user generates an event. The event, in turn causes a certain application fragment to be executed.

Execution of such a fragment may cause further screen layouts to be generated with a new context of available widgets on it. The events that can be generated in any given state are defined by the context.

The entire application can be viewed as a system which reacts to events by dispatching them to the appropriate piece of code that is responsible for handling the event. As a consequence, the application is structured as a set of fragments that are responsible for handling specific events.

This conceptual view of an application can be viewed as a way of structuring its high-level design, which would then need to be detailed by a conventional implementation. There are languages, however, that directly support this conceptual view, by providing the necessary abstractions. For example, languages like Visual Basic, Visual C++, or Tcl/Tk allow one to separately design the widgets and the code fragments, and to bind events on a widget to the fragments which respond to the event.

Another common event-driven control paradigm is the one based on so-called *triggers*. Triggers became popular in recent years, in conjunction with new developments in the field of so-called active data bases. Since there is no precise and universal definition of a trigger, we will give examples based on ahypothetical language syntax.

An *active data base* consists of a conventional underlying (passive) data base and a set of *active rules* (or *triggers*) of the following form
on event
when condition
do action
When the event associated with the rule occurs, we say that the rule is *triggered*.

A triggered rule is then checked to see if the condition holds. If this is the case, the rule can be executed.

As an example, the following trigger specifies that the total number of employees should be updated as a new employee record is inserted in the data base.
on insert in EMPLOYEE
when TRUE
do emp_number ++
As another example, in a database application, triggers may be used to specify some constraints that must be verified as new elements are inserted or existing elements are updated or deleted from the database. For example, a constraint might be that no

employee can have a salary that is more than the average salary of managers. A trigger might watch that no insertion, update, or deletion violates the constraint; if that happens, some appropriate action would be undertaken.

A trigger-based problem solution can be viewed as a high-level design, which is then implemented in any programming language using the conventional constructs it provides. However, there are languages where trigggers are directly available as a built-in language construct; i.e., they are implemented by the underlying run-time machine. As an example, the forthcoming SQL standard includes triggers as one of its features.

Concurrent computations:

Sometimes it is convenient to structure our software as a set of concurrent units which execute in parallel. This can occur when the program is executed on a computer with multiple CPU's (multiprocessor). In such a case, if the number of processors coincides with the number of concurrent units, we say that underlying machine that executes the program provides for *physical* parallelism:

each unit is in fact executed by its dedicated processor. Parallelism, however, may be simply *logical*. For example, if the underlying machine is a uniprocessor, the logical view of parallel execution may be provided by switching the CPU from one unit to another in such a way that all units appear to progress simultaneously. The switching of the execution of the uniprocessor among the various units can be performed by a software layer, implemented on top of the physical machine, which provides the programmer with a view of an abstract parallel machine where all units are executed simultaneously.

Once such abstract machine is in place, one can in fact abstract away from the physical architecture of the underlying hardware, where components are actually executed. The hardware structure might be a multiprocessor, with each processor dedicated to a single unit, or it might be a multiprogrammed uniprocessor.

Allowing for the possibility of different machines means that the correctness of a concurrent system cannot be based on an assumption of the speed of execution of the units. Indeed, the speed can differ greatly if every unit is executed by a dedicated processor, or if a single processor is shared by several units.

Moreover, even if the architecture is known, it is difficult to design a system in such a way that its correctness depends upon the speed of execution of the units. We will return to these points in the discussion of implementation models for concurrency.

Concurrency is an important area of computer science, which is often studied in different context: machine architectures, operating systems, distributed systems, databases, etc. In this section we give an overview of how programming languages support concurrency.

Concurrent programs support concurrency by allowing a number of units (called *processes*) to execute in parallel.

If the abstract machine that executes the program does not support concurrency, it is possible to simulate it by transferring control explicitly from one unit to another. This low-level approach is supported by *co-routines.*

Coroutines are a low-level construct for describing pseudo-concurrent units. They can be used to simulate parallelism on a uniprocessor by explicitly interleaving the execution of a set of units. Therefore, they do not describe a set of concurrent units, but a particular way of sharing the processor to simulate concurrency.

As an example, consider the the two coroutines client and give_me_next shown in Figure 53, written in a hypothetical, self-explaining programing language. Unit client repeatedly activates unit give_me_next to get the next value of a variable. Each reactivation of unit give_me_next produces a new value, which depends on the previously generated value. The two units resume one another. There is a global variable i, which is shared by client and give_me_next. Unit main, which is activated initially, resumes client.

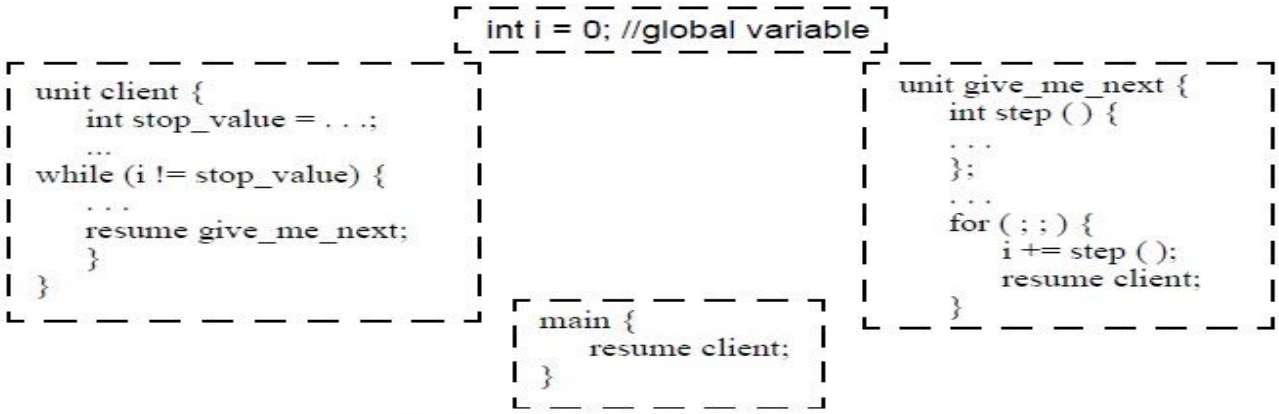


FIGURE 53. An example of coroutines

UNIT3: CHAPTER:2

Structuring the program

Software design methods:

To combat the complexities of programming in the large, we need a systematic design method that guides us in composing a large program out of smaller units—which we call *modules*. A good design is composed of modules that interact with one another in well-defined and controlled ways. Consequently, each module can be designed, understood, and validated independently of the other modules.

Once we have achieved such a design, we need programming language facilities that help us in implementing these independent modules, their relationships, and their interactions.

The goal of software design is to find an appropriate modular decomposition of the desired system. Indeed, even though the boundaries between programming in the large and programming in the small cannot be stated rigorously, we may say that programming in the large addresses the problem of modular system decomposition, and programming in the small refers to the production of individual modules.

A good modular decomposition is one that is based on modules that are as independent from each other as possible. There are many methods for achieving such modularity. A well-known approach is *information hiding* which uses the distribution of "secrets" as the basis for modular decomposition. Each module hides a particular design decision as its secret.

The idea is that if design decisions have to be changed, only the module that "knows" the secret design decision needs to be modified and the other modules remain unaffected.

If a design is composed of highly independent modules, it supports the requirements of large programs:

- Independent modules form the basis of work assignment to individual team members. The more independent the modules are, the more independently the team members can proceed in their work.
- The correctness of the entire system may be based on the correctness of the individual modules. The more independent the modules are, the more easily the correctness of the individual modules may be established.
- Defects in the system may be repaired and, in general, the system may be enhanced more easily because modifications may be isolated to individual modules.

Concepts in support of modularity

The key to software design is modularization. A good module represents a useful abstraction; it interacts with other modules in well-defined and regular ways; it may be

understood,designed, implemented, compiled, and enhanced with access to only the specification (not the implementation secrets) of other modules. Programming languages provide facilities for building programs in terms of constituent modules.

Procedures and functions are an effective way of breaking a program into two modules: one which provides a service and another which uses the service. We may say that the procedure is a server or service provider and the caller is a client. Even at this level we can see some of the differences between different types of modularization units.

Procedures and functions are units for structuring small programs, perhaps limited to a single file. Sometimes, we may want to organize a set of related functions and procedures together as a unit.

Encapsulation

A program unit provides a service that may be used by other parts of the program, called the *clients* of the service. The unit is said to *encapsulate* the service.

The purpose of encapsulation is to group together the program components that combine to provide a service and to make only the relevant aspects visible to clients. *Information hiding* is a design method that emphasizes the importance of concealing information as the basis for modularization.

Encapsulation mechanisms are linguistic constructs that support the implementation of information hiding modules. Through encapsulation, a module is clearly described by two parts: the specification and the implementation.

The specification describes how the services provided by the module can be accessed by clients. The implementation describes the module's internal secrets that provide the specified services.

Different languages provide different encapsulation facilities. For example, in C, a file is the unit of encapsulation. Typically, the entities declared at the head of a file are visible to the functions in that file and are also made available to functions in other files if those functions choose to declare them.

Variables declared in a C function are local and known only to that function. Variables declared outside of functions are assumed to be available to other units, if they declare them using the extern specifier. But a unit may decide to hide such variables from other units by declaring them as static.

Interface and implementation:

A module encapsulates a set of entities and provides access to some of those entities. The available entities are said to be *exported* by the module. Each of the exported entities is

available through an interface. The collection of the interfaces of the exported entities form the *module interface*. Clients request the services provided by a module using the module's *interface*, which describes the module's specification.

The interface specifies the syntax of service requests. Some languages also support or require the specification of the interface's semantic requirements. The idea is that the interface is all that the client needs to know about the provider's unit. The implementation of the unit is hidden from the client. The separation of the interface from the implementation contributes to the independence of the client and the server from one another.

A service provider *exports* a set of entities to its clients. A client module *imports* those entities to be able to use the services of the provider module. The exported entities comprise the service provided by the module. Some languages have implicit and others explicit mechanisms for import and export of entities. Languages also differ with respect to the kinds of entities they allow to be exported. For example, some languages allow a type to be exported and others do not.

The idea of modularity is to enable the construction of large programs out of smaller parts that are developed independently. At the implementation level, independent development of modules implies that they may be compiled and tested individually, independently of the rest of the program. This is referred to as *independent* compilation. The term *separate* compilation is used to refer to the ability to compile units individually but subject to certain ordering constraints.

For example, C supports independent compilation and Ada supports separate compilation. In Ada, as we will see later, some units may not be compiled until other units have been compiled. The ordering is imposed to allow checking of interunit references. With independent compilation, normally there is no static checking of entities imported by a module.

Language features for programming in the large:

All programming languages provide features for decomposing programs into smaller and largely autonomous units. We refer to such units as *physical* modules; we will use the term *logical* module to denote a module identified at the design stage. A logical module represents an abstraction identified at the design stage by the designer. A logical module may be implemented by one or more physical modules. The closer the relationship between the physical modules and logical modules is, the better the physical program organization reflects the logical design structure.

• Module encapsulation: What is the unit of modularity and encapsulation supported by the

language, and how well does it support different programming paradigms?
• Separation of interface from implementation: What is the relationship among modules that form a program? What entities may be exported and imported by a module?
• Program organization and module groupings: How independently can physical modules be implemented and compiled? What are the visibility and access control mechanisms supported by the language?

Pascal, C, C++, Ada, and ML. Pascal and C are viewed here as a representative of the class of traditional, minimalist, procedural languages. FORTRAN. C++ is a representative of classbased languages. Ada is a representative of module-based languages. ML is reviewed as a representative of functional languages.

Pascal:
The only features provided by Pascal for decomposing a program into modules are procedures and functions, which can be used to implement procedural abstractions. The language thus only supports procedural programming. Some later versions of the language have modified the original version of Pascal extensively by adding object-oriented programming features.

A Pascal program has the following structure.

Program program_name (files);
declarations of constants, types, variables, procedures and functions;
begin
statements (no declarations)
end.

A program consists of declarations and operations. The operations are either the built-in ones provided by the language or those declared as functions and procedures. A procedure or function itself may contain the declaration of constants, types, variables, and other procedures and functions. The organization of a Pascal program is thus a tree structure of modules. The tree structure represents the textual nesting of lower-level modules. Nesting is used to control the scope of names declared within modules, according to the static binding rule presented.

C
C provides functions to decompose a program into procedural abstractions. In addition, it relies on a minimum of language features and a number of conventions to support programming in the large. These conventions are well recognized by C programmers and are even reflected in tools that have been developed to support the language. Indeed, a major portion of the programming in the large support is provided by the file-inclusion commands of the C preprocessor. Thus, even though the compiler does not provide any

explicit support or checking for inter-module interaction, the combination of conventions and the preprocessor has proven in practice to be an adequate and popular way to support programming in the large.

The C unit of physical modularity is a file. A logical module is implemented in C by two physical modules (files) which we may roughly call the module's interface and its implementation. The interface, called a "header" or an "include" file, declares all symbols exported by the module and thus available to the clients of the module. The header file contains the information necessary to satisfy the type system when the client modules are compiled. The implementation file of the module contains the private part of the module and implements the exported services. A client module needing to use the functionality of another module "includes" the header file of the provider module.

A header file may declare constants, type definitions, variables, and functions. Only the prototype of the function—its signature—is given by the declaration; the function definition appears in the implementation file. Functions may not be nested. Any names defined in a file are known throughout that file and may also be known outside of that file.

The header files are used to resolve inter-module references at compile-time. At link-time, all implementation files are searched to resolve inter-module (i.e. inter-file) references. The header file is usually named with a .h extension and the implementation file is named with a .c extension. These conventions have largely overcome the lack of any explicit support for program organization.

The general structure of a C file is shown in Figure 72. All files have similar structure except that one of the files (only) must contain a function named main, which is called to start the program.


```
#include ...various files...
global declarations
function definitions
void main (parameters)
{
...one main function needed
...in a program
}
```

FIGURE 72. Structure of a C module

Any names defined in the outer level of a file are implicitly known globally. These include the names of all the functions defined in the file and any other entities defined outside of those functions. There are two ways to control such indiscriminate dispersion of names.

• A module wanting to use an entity that is defined externally must declare such entities as being externally defined.

• A module wanting to limit the scope of one of its defined entities to be local to itself only may declare such an entity to be static.

C++

C++ is based on C and it shares C's reliance on conventions and unit of physical modularity as the file. As C, C++ provides functions as a decomposition construct to implement abstract operations. Nevertheless, C++'s most important enhancements to C are in the area of programming in the large. In particular, the class construct of C++ provides a unit of logical modularity that supports the implementation of information hiding modules and abstract data types. Combined with templates, classes may be used to implement generic abstract data types. The class provides encapsulation and control over interfaces.

The unit of logical modularity in C++ is the class. A class serves several purposes including:

• A class defines a new (user-defined) data type.

• A class defines an encapsulated unit.

Entities defined by a class are either *public*—exported to clients—or *private*—hidden from clients.

Since a class defines a user-defined type, to use the services offered by a class, the client must create an instance of the class, called an *object*, and use that object. C++ supports the style of programming in which programmers write applications by extending the types of the language with user-defined types. Class derivation is a mechanism that supports the definition of new types based on existing types.

Classes may be nested. But as we saw in the case of Pascal, nesting may be used only for programming in the small and is of limited utility for programming in the large.

Program organization

Classes define the abstractions from which the program is to be composed. The main program or a client creates instances of the classes and calls on them to perform the desired task. We saw the definition of a C++ template module implementing a generic abstract data type stack.

C++ supports the development of independent modules (but does not enforce it):

1.A class's interface and implementation may be separated and even compiled separately from each other. The implementation must include the interface definition and therefore must be compiled after the interface file exists.

2. Client modules may be compiled with access to only the interface modules of the service providers and not their implementation modules.

3. Any names defined in a class are local to the class unless explicitly declared to be public.

Even so, client modules must use the class name to gain access to the names internal to the class.

Namespaces. In C and in C++, the unit of global naming is a file. Any names defined at the outer level of a file are known globally by default. For example, the names of all classes defined in a library are known to any client that includes that file. What if two libraries provide two classes with the same name? How can a client use both of those classes? How can a library provider add a new service to its library and be sure that the new name of the service does not conflict with any existing uses of the clients? Since names are created by independent people, a single global name space is a serious problem in the development of large programs.

The solution of C++ is to partition the global name space into a smaller groups; each group is called a *namespace*. The names defined in a namespace are independent from those in any other namespace and may be referenced by supplying the name of the namespace.

This mechanisms enables library providers to provide their libraries in their own namespaces with a guarantee of independence from other library providers. Of course, it is necessary for the names of the namespaces themselves to be unique.

Ada

Ada was designed specifically to support programming in the large. It has elaborate facilities for the support of modules, encapsulation, and interfaces. Rather than relying on convention as in C and C++, Ada makes an explicit distinction between specification and implementation of a module. A file may be compiled if the specifications of the modules it uses are available. Thus, Ada naturally supports a software development process in which module specifications are developed first and implementation of those modules may proceed independently. Ada also requires the existence of a compile-time library in which module specifications are compiled. A module may be compiled if all the module specifications it needs are already in the library. This library supports the checking of inter-module references at compile time.

Encapsulation in Ada

The package is Ada's unit of modularity. An Ada module encapsulates a group of entities and thus supports module-based programming. We have already seen that the language's explicit distinction between module specification and module body forces the programmer to separate what is expected by the module from what is hidden within the module. Additionally, Ada supports concurrent modules or tasks.

Program organization

An Ada program is a linear collection of modules that can be either subprograms or packages. These modules are called units. One particular unit that implements a subprogram is the main program in the usual sense. Module declarations may be nested.

Consequently, a unit can be organized as a tree structure of modules. Any abuse of nesting within a unit causes the same problems discussed for Pascal. These problems can be mitigated by the use of the subunit facility offered by the language. This facility permits the body of a module embedded in the declarative part of a unit (or subunit) to be written separately from the enclosing unit (or subunit). Instead of the entire module, only a *stub* need appear in the declarative part of the enclosing unit.

The following example illustrates the concept of the subunit.

```
procedure X ( ... ) is --unit specification
W: INTEGER;
package Y is --inner unit specification
A: INTEGER;
function B (C: INTEGER) return INTEGER;
end Y;
package body Y is separate; --this is a stub
begin -- uses of package Y and variable W
...
...
...
end X;
----------------------------------next file----------
separate (X)
package body Y is
procedure Z (...) is separate; --this is a stub
function B (C: INTEGER) return INTEGER is
begin --use procedure Z
...
...
...
end B;
end Y;
----------------------------------next file----------
separate (X.Y)
procedure Z (...) is
begin
...
end Z;
```

Interface and implementation

Ada strictly separates the specification and body of a package. In the previous section, we have seen how the **use** and **with** clauses are used to import services from packages. These facilities are used also to support separate compilation. Recall that separate compilation, as opposed to independent compilation, places a partial ordering on compilation units.

The set of units and subunits comprising a program can be compiled in one or more separate compilations. Each compilation translates one or more units and/or subunits.

Grouping of units

Ada has many features for supporting programming in the large. Two clauses, **use** and **with,** are used to import services from other packages. Child library units are used to group packages together in hierarchical organizations. These facilities are defined to enable safe separate compilation.

ML

Modularity is not only the province of imperative languages. The notion of module is important in any language that is to be used for programming in the large. For example, ML is a functional programming language with extensive support for modularity and abstraction.

we will see ML's support for defining new types and abstract data types, which also help in programming in the large. In this section we give a brief overview of ML's support for modules.

Encapsulation in ML

A module is a separately compilable unit. A unit may contain structures, signatures, and functors. Structures are the main building blocks; signatures are used to define interfaces for structures; functors are used to build a new structure out of an existing structure.

Here, we only examine the structure construct as a packaging unit. The ML *structure* is somewhat like the Ada package, used to group together a set of entities.

```
structure Dictionary =
struct
    exception NotFound;

    val root = nil;  (*create an empty dictionary*)

 (* insert (c, i, D) inserts pair <c,i> in dictionary D*)
    fun insert (c:string, i:int, nil) = [(c,i)]
    |   insert (c, i, (cc, ii)::cs) =
        if c=cc then (c,i)::cs
        else (cc, ii)::insert(c,i,cs);

 (* lookup (c, D) finds the value i such that pair <c,i> is in dictionary D *)
    fun lookup(c:string, nil) = raise NotFound
    |   lookup (c, (cc,ii:int)::cs) =
        if c = cc then ii
        else lookup(c,cs);
end;
```

Interface and implementation

The signature of a structure definition consists of the signatures and types of all the entities defined in the structure. ML also provides a construct to define a signature independently of any structure. A signature may be viewed as a specification for a module. For example, Figure 76 gives the signature of a module that exports an exception called NotFound and a function called lookup.

A signature may be used as a specification for a structure

```
signature DictLookupSig = sig
   exception NotFound;

   val lookup : string * (string * int) list -> int
end
```

FIGURE 76. A signature defintion for specialized dictionary

ML also supports the concept of *generic* modules or structures. The signature facility may be combined with generic structures to instantiate a structure for particular types. For example, the dictionaries that we have defined so far, Both in Ada and in ML have been specific to <string, integer> pairs.

Generic units:

One important approach to developing large programs is to build them from existing modules. Traditional software libraries offer examples of how such existing modules may be packaged and used. One of the criteria for evaluating the suitability of a language for programming in the large is whether it provides language mechanisms that enable the construction of independent components, the packaging together of related components, the use and combination of multiple libraries by clients, etc. We have seen the namespaces of C++ and the libraries and child libraries of Ada 95 as language mechanisms explicitly developed for the support of such packaging of related and independent software components.

Generic data structures:

Early typed languages such as Pascal and C require the designer to define one structure for each data type to be supported. This is an unsatisfactory solution for two reasons: one is that the solution only works for only the types the library designer knows about and not for any types to be defined by the user of the library; the second is that the solution forces the library designer towards code duplication.

C++ templates and Ada generics allow us to avoid such code duplication and define a template data structure that is independent of the type of the element to be stored in the structure. For example, we can define a generic *pair* data structure in C++:

```
template <class T1, class T2>
class pair {
```

```
public:
T1 first;
T2 second;
pair (T1 x, T2 y) : first(x), second(y) { }
};
```

The template parameters T1 and T2 stand for any type. We may "instantiate" a particular pair by supplying concrete types for T1 and T2. For example, we may create a pair of integers or a string, integer pair or a pair of employees:

```
pair<int, int> intint(2, 1456);
pair<string, int> stringint("Mehdi", 46);
pair<employee_t, employee_t> (jack, jill); /*pair of user-defined type employee_t*/
```

We may refer to pair as a parameterized or generic type. The template of C++ allows us to define such a parameterized type which may later be used to create concrete types such as pair<int, int>. C++'s template facility is particularly general because it uses classes as parameters and classes represent types uniformly:

We may instantiate a template with either user-defined or primitive types. Eiffel supports a similar scheme for generic classes, with which, for example, we can define a class stack [T] and then instantiate an intstack from stack[integer]. In Chapter 3 we saw examples of generic stacks both in C++ and Eiffel.

Generic algorithms:

Templates may also be used to define generic algorithms. We saw the following generic function swap which interchanges the values of its two parameters:

```
template <class T>
void swap(T& a, T& b)
{
T temp = a;
a = b;
b = temp;
}
```

This function may be used for any two parameters of the same type that support the "=" operation. Therefore, we can use it to swap integers, reals, and even user-defined types such as pairs. This is quite a useful facility because it gives us the possibility to write higher-level generic functions such as sort if they only use generic functions. The ability to write such generic functions is helped in C++ by the fact that generic functions do not have to be instantiated to be used.

Generic modules:
Collections of data and algorithms may also be packaged together and collectively made to depend on some generic type parameter. Both C++ classes and Ada packages may be defined as generic in the types they use.

The ML support for generic modules is particularly interesting because of the separation of structures and signatures.

By applying the signature, we are restricting the view of the structure. Applying a signature to a polymorphic structure is similar to package instantiation in Ada.

```
structure Dictionary =
struct
exception NotFound;
val root = nil; (*create an empty dictionary*)
(* insert (c, i, D) inserts pair <c,i> in dictionary D*)
fun insert (c, i, nil) = [(c,i)]
| insert (c, i, (cc, ii)::cs) =
if c=cc then (c,i)::cs
else (cc, ii)::insert(c,i,cs);
(* lookup (c, D) finds the value i such that pair <c,i> is in dictionary D *)
fun lookup(c, nil) = raise NotFound
| lookup (c, (cc,ii:int)::cs) =
if c = cc then ii
else lookup(c,cs);
end
```

Higher levels of genericity
We have seen that we may define a generic algorithm that works on any type of object passed to it. For example, the max algorithm may be applied to any ordered type. This facility allows us to write one algorithm for n different data types rather than n different algorithms. It leads to great savings for writers of libraries. But consider a higher level of generality. Suppose we want to write an algorithm that works on different types of data *structures*, not just different data types. For example, we may want to write one algorithm to do a linear search in any "linear" data structure. Of course, we have to capture the notion of linearity somehow but intuitively, we want to be able to find an element in a collection regardless of whether the collection is implemented as an array, a list. The goal of the generic programming paradigm is to develop exactly these kinds of units.

For example, consider the following
function find:

```
template<class Iter, class T>
Iter find ( Iter f, Iter l, T x)
{
```

```
while (*f != last && *f != x)
++f;
return f;
}
```

UNIT-4
Object-Oriented Languages

Modularity and abstraction are the two most important principles for building large programs. In the "pure" object-oriented style of programming, the unit of modularity is an abstract data type. We have seen that classes may be used to define abstract data types. Another important principle of object-oriented programming is that you may define new classes of objects by extending or refining existing classes.

Some programming languages have been designed expressly to support this style of programming. These languages, namely Smalltalk and Eiffel, are called object-oriented programming languages. Other languages, such as C++ and Ada 95, while not exclusively object-oriented, support the paradigm through features that enable the programming of extensible abstractions. All object-oriented languages trace their roots to the language Simula 67 which introduced the concept of class and subclass in 1967.

The class construct of C++ directly supports the definition of abstract data types. We may design classes Chair and Table if our application deals with such entities and then create as many instances of the specific Chairs and Tables that we need. Next comes the notion of *inheritance*.

Object-oriented programming is an attractive methodology because it promises the ability to package a whole set of related concepts tied together through their inheritance relationships. It aims to enable the production of libraries of related components that are

- easy to understand by users because of the relationships among the components
- easy to extend by the use of inheritance

Many languages have been or are being extended to support object-oriented programming. Object-orientation has affected not only the implementation phase of the software process but most other phases as well.

Q1. WRITE SHORT NOTES ON OOP CONCEPTS?

Concepts of object-oriented programming:
There are several definitions of what object-oriented programming is. Many people refer to an object-oriented program as any program that deals with entities that may be informally called "objects." In contrast to traditional procedural languages in which the programs consist of procedures and data structures, objects are entities that encapsulate data and related operations. For example, given a stack data structure s, in a procedural language we would call a push operation to add an element as in:
push (s, x);
Dealing with objects, as we have seen in C++, we tell the stack object to push an element onto itself, as in:

s.push(x);

We have seen that in C++ and Eiffel we can use classes to define and create objects. We call a programming language that supports the definition and use of objects *object-based*. *Object-oriented* programming languages support additional features. In particular, object-oriented programming languages are characterized by their support of four facilities:

- abstract data type definitions,
- inheritance,
- inclusion polymorphism, and
- dynamic binding of function calls.

We have already discussed abstract data types extensively. They are used in object-oriented programming to define the properties of classes of objects. Inheritance is a mechanism that allows us to define one abstract data type by *deriving* it from an existing abstract data type. The newly defined type "inherits" the properties of the parent type. Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes.

Dynamic binding supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable. The pure terminology of object-oriented languages refers to *objects* that are *instances* of *classes*. An object contains a number of *instance variables* and supports a number of *methods*. A *message* is sent to an object to request the invocation of one of its methods.

Other languages, such as Eiffel and C++, allow polymorphism but restrict it to enable static type checking. Thus, such languages combine polymorphism with strong typing.

Classes of objects:

The first requirement for object-oriented programming is to be able to define abstract data types. We have already seen that this can be done using the class construct.

```
class stack{
public:
void push(int) {elements[top++] = i;};
int pop() {return elements[--top];};
private:
int elements[100];
int top=0;
};
```

This class is a particular implementation of a fixed-size stack abstraction. We may use objects of this class in many different applications. the class construct enables us to

encapsulate the representation used for the data structure and export useful operations to be used by clients.
A client may create as many objects of the stack class as desired:

```
stack s1, s2;
s1.push(3);
s1.push(4);
s2.push(3);
if (s1.pop() == s2.pop) {...}
```

Clients may create objects of this class just as they may create variables of language-defined types. In fact, classes are user-defined types. They share most properties of language-defined types, including storage properties. For example, the above fragment creates stacks s1 and s2 as automatic variables.

We may also create stacks in the free store:

```
stack* sp = new stack;
```

To access member functions (e.g., pop) the following notations denote equivalent expressions:

```
(*sp).pop();
```

and (more commonly used)

```
sp -> pop();
```

While useful, the class facility only addresses the question of how to encapsulate useful abstractions. What we have seen so far does not address the need to create new abstractions based on existing ones. Inheritance is the mechanism used for this purpose.

Inheritance:
Inheritance is a linguistic mechanism that allows us to do just that by defining a new class which "inherits" the properties of a parent class. We may then add new properties to the child class or redefine inherited properties. The terminology in C++ is to *derive* a class from a *base* class.

We want to derive a counting_stack from stack as shown below:

```
class counting_stack: public stack {
public:
int size(); //return number of elements on the stack
};
```

This new class simply inherits all the functions of the class stack1. All public member functions of stack become public member functions of counting_stack (that's what the public before stack specifies). We have also specified that there will be a new public function size() which is intended to return the number of elements stored in the stack. The class stack is called a base class or the parent class of counting_stack. The class

counting_stack is said to be derived from its base class. The terms *subclass* and *superclass* are also used to refer to derived and base classes respectively.

Inheritance is a fundamental concept for supporting programming in the large in that it enables us to develop modules based on existing ones without any modifications to the existing modules. This is the property that makes object-oriented programming an attractive paradigm for software engineering.

Polymorphism:

The next feature of object-oriented programming languages is the support of polymorphism. All classes derived from the same base class may be viewed informally as specialized versions of that base class. Object-oriented languages provide polymorphic variables that may refer to objects of different classes. Object-oriented languages that adopt a strong type system limit the polymorphism of such variables: usually, a variable of class T is allowed to refer to objects of type T or any classes derived from T.

In the case of our example stack and counting_stack in the previous section, this means that a variable of type stack may also refer to an object of type counting_stack. In purely object-oriented languages such as Eiffel and Smalltalk, *all* objects are referred to through references and references may be polymorphic. In C++, only pointer, reference variables, and by reference parameters may be polymorphic. That is, a stack pointer may also point to a counting_stack object.

Dynamic binding of calls to member functions:

A derived class may not only add to the functionality of its base class, it may also add new private data and *redefine* or *override* some of the operations provided in the base class. For example, in counting_stack we may decide to provide a new implementation of the push function because we may want to keep track of the number of times push has been called. Now, if sp is a reference to a stack variable and csp is a reference to a counting_stack variable, we expect that csp->push() will call the push of a counting_stack object but what about sp->push()? Since sp is a polymorphic variable, it may be pointing at a counting_stack object or a stack object. This raises an issue of proper binding of operations. Consider the following example:

```
stack* sp = new stack;
counting_stack* csp = new counting_stack;
...
sp.push(); // stack::push
csp.push(); // counting_stack::push
...
sp = csp; //assignment is okay
...
sp.push(); //which push?
```

So called purely object-oriented languages, such as Smalltalk and Eiffel, bind the choice dynamically based on the type of the object. In fact, as stated, dynamic binding (often called *dynamic dispatching* in object-oriented terminology) is one of the tenets for object-oriented programming languages. C++, however, not being a purely object-oriented language, provides features for both static and dynamic binding.

Dynamic binding combined with inheritance is a powerful notion. For example, we may define a class polygon and derive various specialized versions of polygons such as square and rectangle. Suppose that polygon defines a function perimeter to compute the perimeter of a general polygon. Some of the derived classes may define their own special perimeter functions because they are presumably more efficient.

Clearly, dynamic binding is more flexible than static binding. In languages that do not support dynamic binding, we may have to use case statements (as in Pascal) or use function pointers (as in C) to achieve the same result but with code that is more verbose and harder to maintain. For example, in Pascal we might implement polygon as a variant record and explicitly call the right perimeter function based on the tag of the variant record. In C, each object could contain a pointer to its perimeter function and the call would have to be made indirectly through this pointer.

Inheritance and the type system: The interaction between inheritance and type consistency rules of the language raises a number of interesting issues.

Q2. WRITE SHORT NOTES ON SUB CLASSES VERSUS SUB TPES.?

Subclasses versus subtypes:

we saw the concept of subtype with which we defined a new type as a subrange of an existing type. For example, we defined week_day as a subrange of day. Subtyping introduces a relationship among objects of the subtype and objects of the the parent type such that objects of a subtype may also be viewed as objects of the parent type. For example, a week_day is also a day. This relationship is referred to as the *is-a* relationship: week_day *is-a* day.

The subtype relationship is generalizable to user-defind types such as those defined by classes. For example, a counting_stack *is-a* stack but not vice versa.

But not all subclasses create subtypes. If a derived class only adds member variables and functions or redefines existing functions in a *compatible* way, then the derived class defines a subtype. If it hides some of the parent's member variables and functions or modifies them in an incompatible way, then it does not create a subtype. Therefore, whether a derived class defines a subtype depends on the definition of the derived class and is not guaranteed by the language.

Strong typing and polymorphism: we defined a strong type system as one which guarantees type safety. Strong type systems have the advantage of enabling type errors to be caught at compile-time. Statically typed languages provide a strong type system.

Let us assume that we have a base class base and a derived class derived and two objects derived from them:

```
class base {...};
class derived: public base {...};
...
base* b;
derived* d;
```

We have seen that we may assign d to b but not b to d. The question is under what circumstances can we guarantee that an assignment

```
b = d;
```

will not lead to a type violation at runtime? We may ask this question in terms of *substitutability*: can an object of type derived be substituted for an object of class base in every context? Or in general, can an object of a derived type be substituted for an object of its parent type in every context, and is such a kind of polymorphism compatible with strong typing? If substitutability is ensured, the derived type can be viewed as a subtype of the parent type. By imposing some restrictions on the use of inheritance, the language can ensure substitutability.

Type extension:

If the derived class is only allowed to extend the type of the parent class, then substitutability is guaranteed. That is, if derived does not modify any member functions of base and does not hide any of them, then it is guaranteed that any call b.f(...) will be valid whether b is holding a base object or a derived object.

The compiler may do type-checking of any uses of base variables solely based on the knowledge that they are of type base. Therefore static type checking can guarantee the lack of runtime violations. Type extension is one of the mechanisms adopted in Ada 95.

Q3. WRITE ABOUT HOW OVERRIDING OF MEMBER FUNCTION CAN HAPPENED IN DIFFERENT PROGRAMMING LANGUAGES?

Overriding of member functions

Restricting derived classes to only extend the parent type is a severe limitation on an object-oriented language. In fact, it rules out dynamic dispatch completely. Many languages allow a derived class to redefine an inherited member function. For example, as in our previous example, we may derive a square class from a polygon class. The square class may redefine the general perimeter function from the base class by a more efficient version of a perimeter function.

In C++, the base class must specify the function perimeter as a virtual function, giving derived classes the opportunity to override its definition. That is,

```
class polygon {
public:
polygon (...) {...} //constructor
virtual float perimeter () {...};
...
};
class square: public polygon {
public:
...
float perimeter() {...}; //overrides the definition of perimeter in polygon
};
```

Q4. WRITE SHORT NOTES ON INHERITANCE HIERARCHIES?

Inheritance hierarchies:

The inheritance relationship imposes a hierarchy and provides a mechanism for the development of a hierarchically organized families of classes. In this we discuss several issues raised by inheritance hierarchies.

Single and multiple inheritance:

In Simula 67, Ada, and Smalltalk, a new class definition is restricted to have only one base class: a class has at most one parent class. These languages have a *single-inheritance* model.

C++ and Eiffel have extended the notion of inheritance to allow a child to be derived from more than one base class. This is called *multiple inheritance*. For example, if we have a class displayable and a class polygon, we might inherit from both to define a displayable rectangle:

```
class rectangle: public displayable, public polygon {
...
}
```

The introduction of multiple inheritance into a language raises several issues. The successful use of multiple inheritance requires not only well-designed inheritance hierarchies but also orthogonally designed classes that may be combined without clashing. In practice, the use of multiple inheritance requires much care. Whether its benefits outweigh its complexity is an open question. Java, which adopts many features of C++, uses only single inheritance but introduce separate interfaces and supports the idea of inheriting from multiple interfaces.

Implementation and interface inheritance:

One of the promises of object-oriented programming is that new software components may be constructed from existing software components. This would be a significant

contribution to programming in the large issues. To what extent does inheritance support a methodology for such incremental building of components?

Inheritance complicates the issue of encapsulation because the derived classes of a class are a different type of client for the class. On the one hand, they may want to extend the facilities of a parent class and may be able to do so solely by using the public interfaces of the parent class; on the other hand, the facilities they provide to their clients may often be implemented more efficiently if they access the internal representations of their parent classes. C++ introduces protected members and friend classes exactly for these special clients of a base class. Eiffel, on the other hand, allows derived classes to access all features defined in the parent class.

From a software engineering view, interface inheritance is the right methodology but to rely only on interface inheritance requires both a well-designed base class and efficient language implementations. A well-designed inheritance hierarchy is a requirement for the successful use of object-oriented programming in any case. Any hierarchy implies that the nodes closer to the root of the hierarchy affect a larger number of the leaf nodes of the hierarchy. If a node close to the root needs to be modified, all of its children are affected. As a result, even though inheritance supports the incremental creation of software components, it also creates a tightly-dependent set of components. Modifications of base classes may have far reaching impact.

Q5. EXPLAIN OOP FEATURES IN DIFFERENT PROG. LANGUAGES?

Object-oriented programming Features in programming Languages:

The way different languages support object-oriented programming is related to the philosophy of the language and more specifically to the language's object and encapsulation models. In C++, the class construct defines a user defined type. Object-oriented features have been added to the language to allow programmers who want to use object-oriented programming to do so.

In Eiffel, the class construct defines an abstract data type. The language has been designed to support the object-oriented programming style exclusively. In Ada 95, the package is simply an encapsulation mechanism for packaging a set of related entities. It is neither necessarily a type, nor an abstract data type. It may be used to support those notions, however. Ada 95 has some object-orientation features but the language remains a module-oriented language.

C++

C++ supports object-oriented programming by providing classes for abstract data types, derived classes for inheritance, and virtual functions for dynamic binding. This support is provided with static type checking.

a C++ class defines a user-defined type. Indeed the programmer can create first-class types because the language allows the definition of initialization, assignment, and equality test for the type being defined.

As a result, objects of user-defined types may behave quite like objects of language-defined types: they may be created on the stack or in free store, passed as parameters, and returned by functions as results.

The language supports both inheritance and multiple inheritance for defining new classes.

Classes:

C++ classes as a definition mechanism for abstract data types. C++ provides the programmer with control over the creation, initialization, and cleanup of objects of such abstract data types. In particular, one or more constructors may be defined for a class. Such constructors are invoked automatically to initialize objects of the class at creation time. A constructor has the same name as the class. By analogy to a constructor, a *destructor* is invoked automatically when the object is destroyed either explicily through a call to delete or implicitly when the object goes out of scope. The ability to control what happens when an object is destroyed is critical for complex data types that may have allocated substructures in the heap.

For example, simply deleting a pointer to the head of a list may leave the entire list inaccessible in the free store. A destructor gives the programmer the possibility to clean up after the object properly based on the requirements of the object. The name of a destructor is the same as the class name preceded by ~ (i.e. the complement of the constructor).

Besides construction and destruction, programmer control is important over two other operations for user-defined types: assignment and equality comparison. These two operations are related semantically. In general, we expect that after assigning an object a to object b, the two objects are equal. C++ by default uses member-wise copy and member-wise comparison for assignment and comparison of class objects. This is often inadequate if the structure of the object involves heap-allocated components. In these cases, the programmer may define class-specific assignment and equality operations.

There are no special C++ features for this: as any other operators, the programmer may overload = and ==. Most languages treat these operations in a special way. For examle, we have seen that Ada lets the programmer designate a type as **private** to indicat that the language-defined assignement and equality apply to the type; **limited private** means that they do not.

Use of virtual functions for specification

Virtual functions may be used to define abstract classes. For example, we may specify that a shape must have three functions named draw, move, and hide, without providing an implementation for these functions. A virtual function that does not have an implementation is called a *pure virtual function*. To write a pure virtual function, its body is written as = 0;. If one of the functions of a class are pure virtual, the class is called *abstract*. We may not create objects of an abstract class. In the example, objects of type shape cannot be created because such a class does not have an implementation. The pure virtual designation for a function says that a derived class based on shape must define such functions concretely.

Ex: shape and a class rectangle derived from it.

```
class shape{
public:
void draw() = 0; // this and the others are pure virtual function
void move() = 0;
void hide() = 0;
point center;
};

class rectangle: public shape{
float length, width; //specific data for rectangle
public:
void draw() {...}; //implementation for the derived pure virtual function
void move() {...};
void hide() {...};
};
```

We may view abstract classes as a specification for a set of derived classes. The abstract class specifies the interface and the derived classes must provide the implementation.

Thus the general form of a C++ class is shown here:

```
class C {
private:
// accessible to members and friends only
protected:
// accessible to members and friends and
// to members and friends of derived classes only
public:
// accessible to the general public
};
```

The C++ language provides three levels of protection. Entities defined in a class may be private (default case), in which case they are only accessible inside the class itself; they may be defined as protected, in which case they are accessible inside the class and inside any classes derived from it; or they may be defined as public, in which case they are accessible generally.

The public entities define the services provided by the class and constitute its interface. The private entities deal with the internal details of the class such as the representation of data structures. The protected entities are not of interest to users of the class but they are of interest to any classes that are derived from this class and need to provide services based on the services already provided by this class.

Overloading, polymorphism, and genericity:

Since the object must belong to a subclass of the class to which b is declared to point, this is a case of inclusion polymorphism. If the function is not declared to be virtual in the base class, then the two functions in the base and derived classes are treated as simply overloading the same function name: the proper function to call is selected at compile-time. Finally, we have seen that C++ also supports generic functions. If a function f(a,b) is generic, the types of a and b are used at compile time to instantiate a function that matches the types of the parameters a and b. There is no dynamic dispatch in this case.

Ada 95:

The original version of the Ada language, introduced in 1983, was an object based language. The package construct may be used to create objects that encapsulate both data, possibly private, and related operations. This enabled object-based programming. Since the introduction of Ada, however, the concepts of object-oriented programming have become better understood. As a result, a number of features were added to Ada 95 to support object-oriented programming techniques. In particular, **tagged types** support derivation of a new type from an existing type, and a technique called "classwide programming" combined with tagged types supports dynamic dispatch.

Another feature of Ada 95 is the ability to define abstract types which in turn enable the association of multiple implementations with the same interface. Such flexibility is usually associated with object-oriented programming.

Tagged types:

With Ada 95, a type declaration may be designated as *tagged*, in which case it is possible to derive new types from it by extending it. This facility allows us to define a hierarchy of related types.

For example, the following program defines a tagged type named Planar_Object as having certain set of basic properties such as X and Y coordinates of its center and three

functions: one to compute its Distance from the origin, another to Move it to a new position, and another to Draw it as a predefined icon on the screen.

```
package Planar_Objects is
type Planar_Object is tagged
record
X: Float := 0.0; --default initial value of the center's x coordinate
Y: Float := 0.0; --default initial value of the center's y coordinate
end record;
function Distance (O: Planar_Object) return Float;
procedure Move (O: inout Planar_Object; X1, X2: Float);
procedure Draw (O: Planar_Object);
end Planar_Objects;
```

We will assume that the body (implementation) of the package Planar_Objects is given elsewhere. With this definition we may declare objects of type Planar_Object and apply Distance, Move and Draw operations to them. Next we can define new types Point, Rectangle, and Circle that inherit the properties of the type Planar_Object. For a Point, the X and Y coordinates are enough, thus the data representation of Planar_Object does not need to be extended. But for a Circle, we will add a Radius field and for a Rectangle, we will add the sizes of the two edges.

Dynamic dispatch through classwide programming

The tagged types of Ada 95 are also used to support dynamic binding of function calls. The tag is an implicit field of an object and is used at runtime to identify the object's type. For example, suppose that we want to write a procedure Process_Shapes that will process a collection of objects that may be Points, Rectangles, or Circles. We need to declare this procedure as accepting a polymorphic type that includes all these types. The 'Class attribute of Ada 95 constructs exactly such a class. That is, the expression T'Class applied to a tagged type T is the union of the type T and all the types derived from T. It is called a *class-wide* type.

we can write our procedure as:

```
procedure Process_Shapes (O: Planar_Object'Class) is
...
begin
...
... Draw (O) ...; --dispatch to appropriate Area procedure
...
end Process_Shapes;
```

Since it is often useful to access objects through pointers, it is also possible to declare polymorphic pointers

Abstract types and routines:

As in C++ and Eiffel, Ada 95 supports the notion of top-down design by allowing tagged types and routines (called subprograms in Ada) to be declared abstractly as a specification to be implemented by derived types. For example, we might have declared our Planar_Object type before as an abstract type

```
package Planar_Objects is
type Planar_Object is abstract tagged null record;
function Distance (O: Planar_Object'Class) return Float is abstract;
procedure Move (O: inout Planar_Object'Class; X, Y: Float) is abstract;
procedure Draw (O: Planar_Object'Class) is abstract;
end Objects;
```

This package will not have a body. It is only a specification. By applying derivation to the type Planar_Object, we can build more concrete types. As before, we may derive a tree of related types. Once concrete entities (records and subprograms) have been defined for all the abstract entities, we have defined objects that may be instantiated.

Smalltalk

Smalltalk was the first purely object-oriented language, developed for a special purpose machine, and devoted to the devopelopment of applications in a highly interactive single-user personal workstation environment. It is a highly dynamic language, with each object carrying its type at runtime. The type of of a variable is determined by the type of the object it refers to at runtime.

Even the syntax of Smalltalk reflects its object orientation. All objects are derived from the predefined class object. A subclass inherits the instance variables and methods of its parent class (called its superclass) and may add instance varaiables and methods or redefine existing methods. A call to an object is bound dynamically by first searching for the method in the subclass for the method. If not found, the search continues in the super class and so on up the chain until either the method is found or the superclass object is reached and no method is found. An error is reported in this case. A class defines both *instance* variables and methods, and *class* variables and methods. There is a single instance of the class variables and methods available to all objects of the class. In contrast, a copy of each instance variable and method is created for each instance of an object of the class.

Q6. WRITE SHORT NOTES ON OO ANALYSIS AND DESIGN ?

Object-oriented analysis and design:

The object-oriented approach to software development has grown to encompass not just programming but most other phases of software development. Object-oriented analysis tries to analyze the application domain in terms of objects, their associated operations, and relationships among objects. Object-oriented design tries to design a system that consists of objects. Such designs are implemented more easily in object-oriented languages.

Indeed, the use of object-oriented languages is only effective if the design is object-oriented. It is at the design stage that component objects and their relationships are identified. Constructs such as abstract classes that we have seen in Ada, Eiffel, and C++ may be used to document object-oriented designs that can then be implemented in programming languages.

The substitutability and proper inheritance properties that we have discussed for programming languages are treated in terms of *is-a* relationship at the analysis and design stages. In our example, a counting_stack *is-a* stack and therefore may be substituted anywhere a stack is needed (for example passed to a procedure that expects a stack). But stack is *not* a counting_stack and therefore a stack may not be substituted for a counting_stack. A good design rule is to use inheritance to derive a new class when derived_class *is-a* base_class. The C++ rule on assignments among derived and base classes may also be defined using the *is-a* relationship. The assignment a = b is allowed if b *is-a* a. While this rule is intuitive and simple to state, it is not always easy to determine whether two objects are related with the *is-a* relation.

UNIT-5
FUNCTIONAL PROGRAMMING LANGUAGES

Q1. WRITE ABOUT CHARACTERISITCS OF FUNCTINAL PROGRAMMING LANGUAGES?

Functional programming languages take as their basis not the underlying computing engine, but rather the theory of mathematical functions. Rather than efficient execution, these languages are motivated by the questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

We have seen that procedural languages use procedures as the unit of program decomposition. Procedures generally use side effects on global data structures to communicate with other procedures. Abstract data types attempt to modularize a program by packaging data structures and operations together in order to limit the scope of side-effects. Functional programs reduce the impact of side-effects further, or even eliminate them entirely, by relying on mathematical functions, which operate on *values* and produce *values* and have no side-effects.

We start in the next section by describing the main elements of imperative programming. These elements help illustrate the main differences with functional programming. To contrast these differences further, we will then compare mathematical functions with programming langauge functions.

We then look at ML and LISP as examples of functional programming languages. Early functional languages, starting from LISP, were dynamically typed and scoped. Scheme is a dialect of |LISP that introduces static scoping into the language. Later functional languages, such as ML, not only include static scoping but also static typing. Many functional languages, include both Scheme and ML, have also added a module construct to address programming in the large.

Characteristics of imperative languages:

Imperative languages are characterized by three concepts: variables, assignment, and sequencing. The state of an imperative program is maintained in program variables. These variables are associated with memory locations and hold values and have addresses. We may access the value of a variable either through its name (directly) or through its address (indirectly). The value of a variable is modified using an assignment statement. The assignment statement introduces an order-dependency into the program: the value of a variable is different before and after an assignment statement.

Therefore, the meaning (effect) of a program depends on the order in which the statements are written and executed. While this is natural if we think of a program being executed by a computer with a program counter, it is quite unnatural if we think of mathematical functions. In mathematics, variables are bound to values and once bound, they do not change value. Therefore, the value of a function does not depend on

the order of execution. Indeed, a mathematical function defines a mapping from a value domain to a value range. It can be viewed as a set of ordered pairs which relate each element in the domain uniquely with a corresponding element in the range. Imperative programming language functions, on the other hand, are described as algorithms which specify how to compute the range value from a domain value with a prescribed series of steps.

One final characteristic of imperative languages is that repetition—loops— are used extensively to compute desired values. Loops are used to scan through a sequence of memory locations such as arrays, or to accumulate a value in a given variable. In contrast, in mathematical functions, values are computed using function application. Recursion is used in place of iteration. Function composition is used to build more powerful functions.

Because of their characteristics, imperative languages have been given labels such as state-based and assignment-oriented. In contrast, functional languages have been called value-based and applicative.

Q2. WRITE ABOUT MATHEMATICAL AND PROGRMMING FUNCTIONS?

Mathematical and programming functions

A function is a rule for mapping (or associating) members of one set (the domain set) to those of another (the range set). For example, the function "square" might map elements of the set of integer numbers to the set of integer numbers. A function definition specifies the domain, the range, and the mapping rule for the function.

For example, the function definition square(x) □□x*x, x is an integer number defines the function named "square" as the mapping from integer numbers to integer numbers. We use the symbol "□" for "is equivalent to." In this definition, *x* is a *parameter*. It stands for *any* member of the domain set.

Once a function has been defined, it can be *applied* to a particular element of the domain set: the application yields (or *results* in, or *returns*) the associated element in the range set. At application time, a particular element of the domain set is specified. This element, called the *argument*, replaces the parameter in the definition. The replacement is purely textual. If the definition contains any applications, they are applied in the same way until we are left with an expression that can be evaluated to yield the result of the original application.

New functions may be created by combining other functions. The most common form of combining functions in mathematics is function composition. If a function F is defined as the composition of two functions G and H, written as
F=G o H,
applying F is defined to be equivalent to applying H and then applying G to the result.

In conventional programming languages, a function is defined proceduraly:
The rule for mapping a value of the domain set to the range set is stated in terms of a number of steps that need to be "executed" in certain order specified by the control structure. Mathematical functions, on the other hand, are defined applicatively—the mapping rule is defined in terms of combinations or applications of other functions.

Many mathematical functions are defined recursively, that is, the definition of the function contains an application of the function itself. For example, the standard mathematical definition of factorial is:
n! ⇐ **if** n = 0 **then** 1 **else** n * (n - 1)!

As another example, we may formulate a (recursive) function to determine if a number is a prime:
prime (n) ⇐ **if** n = 2 **then** true **else** p (n, n **div** 2)
where function p is defined as:
p (n, i) ⇐ **if** (n **mod** i) = 0 **then** false
else if i = 2 **then** true
else p (n, i - 1)
Notice how the recursive call to p(n, i-1) takes the place of the next iteration of a loop in an imperative program. Recursion is a powerful problem-solving technique. It is a used heavily when programming with functions.

Q3. WRITE ABOUT PRINCIPLES OF FUNCTINAL PROGRAMMING?

Principles of functional programming:

A functional programming language has three primary components:
1.A set of data objects. Traditionally, functional programming languages have provided a single high level data structuring mechanisms such as a list or an array.
2. A set of built-in functions. Typically, there are a number of functions for manipulating the basic data objects. For example, LISP and ML provide a number of functions for building and accessing lists.
3. A set of functional forms (also called high-order functions) for building new functions. A common example is function composition. Another common example is function reduction. *Reduce* applies a binary function across successive elements of a sequence.

For example, reducing + over an array yields the sum of the elements of the array and reducing * over the elements of an array yields the product of the elements of the array. In APL, / is the reduction functional form (called operator in APL) and it takes one operation as argument. The plus reduction can be accomplished by /+ and the multiplication reduction by /*. The use of functional forms is what distinguishes a functional program.

Functional forms support the combination of functions without the use of control structures such asses iteration conditional statements.

The execution of functional programs is based on two fundamental mechanisms: binding and application. *Binding* is used to associate values with names. Both data and functions may be used as values. Function application is used to compute new values.

Values, bindings, and functions:

Functional programs deal primarily with values, rather than variables. Indeed, variables denote values. For example 3, and "a" are two constant values. A and B are two variables that may be *bound* to some values. In ML we may bind values to variables using the binding operator =. For example

```
val A = 3;
val B = "a";
```

The ML system maintains an environment that contains all the bindings that the program creates. A new binding for a variable may hide a previous binding but does not replace it. Function calls also create new bindings because the value of the actual parameter is bound to the name of the formal parameter.

Values need not be just simple data values as in traditional languages. We may also define values that are functions and bind such values to names:

```
val sq = fn(x:int) => x*x;
sq 3;
```

will first bind the variable sq to a function value and then apply it to 3 and print 9. We may define functions also in the more traditional way:

```
fun square (n:int) = n * n;
```

The role of iteration in imperative languages is played by recursion in functional languages.

```
int fact(int n)
{ int i=1;
assert (n>0);
{for (int j=n; j>1; ++j)
i= i*n;
return i;
}
fun fact(n) =
if n = 0 then 1
else n*fact(n-1);
```

Functions in ML may also be written using patterns and case analysis. The factorial program in the figure may be written as composed of two cases, when the argument is 0 and when it is not:

```
fun fact(n) =
fact(0) = 1
| n*fact(n-1);
```

In addition to function definition, functional languages provide functional forms to build new functions from existing functions. Functional programming languages provide both built-in higher order functions and allow the programmer to define new ones. Most languages provide function composition and reduction as built-in functional forms.

Lambda *expressions* represent values in the lambda calculus. There are only three kinds of expressions:
1. An expression may be a single identifier such as x.
2. An expression may be a function definition. Such and expression has the form x.e which stands for the expression e with x designated as a *bound* variable. The expression e represents the body of the function and x the paramter. The expression e may contain any of the three forms of lambda expressions. Our familiar square function may be written as x.x*x.
3. An expression may be a function application. A function application has the form e1 e2 which stands for expression e1 applied to expression e2. For example, our square function may be applied to the value 2 in this way: (x.x*x) 2. Informally, the result of an application can be derived by replacing the parameter and evaluating the resulting expression.
(x.x*x) 2=
2*2 = 4

In a function definition, the parameters following the " " and before the "." are called *bound* variables. When the lambda expression is applied, the occurrences of these variables in the expression following the "." are replaced by the arguments. Variables in the definition that are not bound are called *free* variables. Bound variables are like local variables, and free variables are like nonlocal variables that will be bound at an outer level.

Lambda calculus capture the behavior of functions with a set of rules for rewriting lambda expressions. The rewriting of an expression models a step in the computation of a function. To apply a function to an argument, we rewrite the function definition, replacing occurrences of the bound variable by the argument to which the function is being applied.

Q3. WRITE SHORT NOTES ON REPRESENTATION OF FUNCTIONAL PROGRAMING LANGUAGES?

Representative functional languages:

LISP was the first functional programming language. The LISP family of languages is large and popular. LISP is a highly dynamic language, adopting dynamic scoping and dynamic typing, and promoting the use of dynamic data structures. Indeed garbage collection was invented to deal with LISP's heavy demands on dynamic memory allocation. One of the most popular descendants of LISP is Scheme, which adopts static scope rules.

APL in an expression-oriented language. Because of the value-orientation of expressions, it has many functional features. As opposed to LISP's lists, the APL data structuring mechanism is the multidimensional array.

ML is one of the recent members of the family of functional programming languages that attempt to introduce a strong type system into functional programming. We will examine ML in more detail because of its interesting type structure. In the next section, we look at C++ to see how the facilities of a conventional programming language may be used to implement functional programming techniques.

Most functional programming languages are *interactive*: they are supported by an interactive programming system. The system supports the immediate execution of user commands. This is in line with the value-orientation of these languages. That is, the user types in a command and the system immediately responds with the resulting value of the command.

ML

ML starts with a functional programming foundation but adds a number of features found to be useful in the more conventional languages. In particular, it adopts polymorphism to support the writing of generic components; it adopts strong typing to promote more reliable programs; it uses type inference to free the programmer from having to make type declarations; it adds a module facility to support programming in the large. The most notable contribution of ML has been in the area of type systems.

The combination of polymorphism and strong typing is achieved by a "type inference" mechanism used by the ML interpreter to infer the static type of each value from its context.

Bindings, values, and types:

Establishing a binding between a name and a value is an essential concept in functional programming.

Every value in ML has an associated type. For example, the value 3 has type int and the value fn(x:int) =>x*x has type int->int which is the signature of the functional value being defined.

We may also establish new scoping levels and establish local bindings within these scoping levels. These bindings are established using let expressions:

```
let x = 5
in 2*x*x;
```

evaluates to 50. The name x is bound only in the expression in the let expression.

Functions in ML

In ML, we can define a function without giving it a name just as a lambda expression. For example, as we have seen:

fn(x, y):int => x*y

is a value that is a function that multiplies its two arguments. It is the same as the lambda expression x,y.x*y. We may pass this value to another function as argument, or assign it to a name: val intmultiply = fn(x, y):int => x*y;

The type of this function is fn:int*int->int.

We may also use functions as values of arguments. For example, we may define a higher-order function compose for function composition:

fun compose (f, g)(x) = f(g(x));

The type of compose is ('a->'b * 'c->'a)->('c->'a). ML provides some built-in functional forms as well. The classic one is map which takes two arguments, a function and a list. It applies the function to each element of the list and forms the results of the applications into a list. For example, the result of:

val x= map (length,[[], [1,2,3],[3]); is [0,3,1].

Type system

Unlike LISP and APL, ML adopts a strong type system. Indeed, it has an innovative and interesting type system. It starts with a conventional set of built-in primitive types: bool, int, real, and string. Strings are finite sequences of characters. There is a special type called unit which has a single value denoted as (). It can be used to indicate the type of a function that takes no arguments.

There are several built-in type constructors: lists, records, tuples, and functions.

A list is used to build a finite sequence of values of a single type. The type of a list of T values is written as T list. For example, [1,2,3] is an int list and ["a","b","cdef"] is a string list. An empty list is written as nil or []. The type of an empty list is 't list. 't is a *type variable* which stands for any type. The empty list is a polymorphic object because it is not specifically an empty int list or an empty bool list. The expression 't list is an example of a polymorphic type expression (called *polytype* in ML).

Tuples are used to build Cartesian products of values of different types. For example, (5, 6) is of type int*int and (true, "fact", 67) is of type bool*string*int. We can of course use lists as elements of tuples: (true, []) is of type bool* (t' list).

Records are similar to Pascal records: they are constructed from named fields. For example, {name="Darius", id=56789} is of type {name: string, id: int}. Tuples are special cases of records in which fields are labeled by integers starting with 1. The equality operation is defined for records based on comparing corresponding fields, that is, the fields with the same names.

ML combines type inference with extensive support for polymorphism. Consider a simple polymorphic identity function which returns its argument as its result. This function does not care what the type of its argument is. If we were to write such a function in C or Pascal, we would have to write a function for each type that we expect to use. In ML, we only need one function:

```
fun id(x) = x;
```

We could write a similar function in C++ using templates, where the type of the parameter is a template parameter.

Modules:

ML modules are separately compilable units. There are three major building blocks:

1.A *structure* is the encapsulation unit. A structure contains a collection of definitions of types, datatypes, functions, and exceptions.

2. A *signature* is a type for a structure. A signature is a collection of type information about some of the elements of a structure: those we wish to export. We may associate more than one signature with a structure.

3.A functor is an operation that combines one or more structures to form a new structure.

EX:

```
structure Stack = struct
exception Empty;
val create = [];
fun push(x, stack xs) = stack (x::xs);
fun pop (stack nil) = raise Empty;
| pop (stack [e]) = []
| pop (stack [x::xs]) = stack [xs];
fun top(stack nil) = raise Empty;
| top(stack [x::xs]) = x;
fun lenght(stack []) = 0
| length (stack [x::xs]) = length (stack [xs]) + 1;
end;
```

LISP:

The original LISP introduced by John McCarthy in 1960, known as pure LISP, is a completely functional language. It introduced many new programming language concepts, including the uniform treatment of programs as data, conditional expressions, garbage collection, and interactive program execution. LISP used both dynamic typing and dynamic scoping. Later versions of LISP, including Scheme, have decided in favor of static scoping.

Common Lisp is an attempt to merge the many different dialects of LISP into a single language.

Data objects IN LISP:

LISP was invented for artificial intelligence applications. It is referred to as a language for symbolic processing. It deals with symbols. Values are represented by symbolic expressions (called *S-expressions*). An expression is either an *atom* or a *list*. An atom is a string of characters (letters, digits, and others).

A symbol (as atom) is either a number or a name. A number represents a value directly. A name represents a value bound to the name.

There are different ways to bind a value to a name: SET binds a value globally and LET binds it locally. (SET X (A B C)) binds A to the list value (A B C).

Functions

There are very few primitive functions provided in pure LISP. Existing LISP systems provide many functions in libraries. It is not unusual Such libraries may contain as many as 1000 functions.

QUOTE is the identity function. It returns its (single) argument as its value.
This function is needed because a name represents a value stored in a location. To refer to the value, we use the name itself; to refer to the name, we use the identity function. Many versions of LISP use 'A instead of the verbose QUOTE A. The QUOTE function allows its argument to be treated as a constant. Thus, 'A in LISP is analogous to "A" in conventional languages.

Examples
(QUOTE A) = 'A = A
(QUOTE (A B C)) = '(A B C) = (A B C)

There are several useful functions for list manipulations: CAR and CDR are selection operations, and CONS is a structuring operation. CAR returns the first element of a list (like hd in ML); CDR returns a list containing all elements of a list except the first (like tl in ML); CONS adds an element as the first element of a list (like :: in ML).

For example
(CAR '(A B C)) = A

APL

APL was designed by Kenneth Iverson at Harvard University during the late 1950s and early 1960s. Even though APL relies heavily on the assignment operation, its expressions are highly applicative.

Objects

The objects supported by APL are scalars, which can be numeric or character, and arrays of any dimension. An array is written as sequence of space-separated elements of the array. Numeric 0 and 1 may be interpreted as Boolean values. APL provides a rich set of functions and a few higher-order functions for defining new functions.

The assignment operation (¬) is used to bind values to variables. On assignment, the variable takes on the type of the value being assigned to it. For example, in the following, the variable X takes on an integer, a character, and an array, in successive statements:

X ¬ 123;
X ¬ 'b';
X ¬ 5 6 7 8 9;

The assignment is an operation that produces a value. Therefore, as in C, it may be used in expressions:

X ¬ (Y ¬ 5 6 7 8 9) ´ (Z ¬ 9 9 7 6 5);
W¬ Y - Z;

will set the value of Y to 5 6 7 8 9, Z to 9 9 7 6 5, and W to -4 -3 0 2 4.

Functions

In contrast to pure LISP, APL provides a large number of primitive functions (called *operations* in APL terminology). An operation is either monadic (taking one parameter) or dyadic (taking two parameters).

All operations that are applicable to scalars also distribute over arrays. Thus, A x B results in multiplying A and B. If A and B are both scalars, then the result is a scalar. If they are both arrays and of the same size, it is element-by-element multiplication. If one is a scalar and the other an array, the result is the multiplication of every element of the array by the scalar. Anything else is undefined.

The usual arithmetic operations, +, -, ´, ¸, | (residue), and the usual Boolean and relational operation,€ Ù, Ú, ~, <, £, =, >, ³, ¹, are provided. APL uses a number of arithmetic symbols and requires a special keyboard.

There are a number of useful operations for manipulating arrays. The operation “i” is a “generator” (or constructor, using ML terminology) and can be used to produce a vector of integers. For example, i5 produces 1 2 3 4 5

The operation “;” concatenates two arrays. So i4; i5 results in

1 2 3 4 1 2 3 4 5

The operation “r” uses its left operands as dimensions to form an array from the data given as its right operands. For example:

and

The *compress* operation “/” takes two arguments of the same dimensions and

2 2 r 1 2 3 4
1 2
3 4

Q5. WRITE SHORT NOTES ON FUNCTIONAL PROGRAMMING IN C++?

Functional programming in C++:

It turns out that the combination of classes, operator overloading, and templates in C++ provides a surprisingly powerful and flexible support for programming with functions.

Functions as objects

A C++ class encapsulates an object with a set of operations. We may even overload existing operators to support the newly defined object. One of the operator we can overload is the application operator, i.e. parentheses. This can be done by a function definition of the form: operator()(parameters...){...}. We can use this facility to define an object that may be applied, that is, an object that behaves like a function. The class Translate whose outline is shown such an object. We call such objects function or functional object.

They are defined as objects but they behave as functions.

```
...definitions of types word and dictionary
class Translate {
private: ...;
public:
word operator()(dictionary& dict, word w)
{
// look up word w in dictionary dict
// and return result
}
}
```

We may declare and use the object Translate in this way:

```
Translate Translator(); //construct a Translate object
cout << Translate(EnglishGermanDict, “university”);
```

which would presumably print “universitaet”, if the dictionary is correct.

The ability to define such objects means that we have already achieved the major element of functional programming: we can construct values of type function in such a way that we can assign them to variables, pass them as arguments, and return them as result.

Functional forms

Another major element of functional programming is the ability to define functions by composing other functions. The use of such high-order functions is severely limited in conventional languages and they are indeed one of the distinguishing characteristics of functional languages.

With the use of templates in C++, we can simulate high-order functions to a high degree. First, we can use function objects as closures and bind some of their parameters.

The 1995 ANSI proposal for the C++ standard library contains a number of function objects and associated templates to support a functional style of programming. For example, it includes a predicate function object greater which takes two arguments and returns a boolean value indicating whether the first argument is greater than the second. We can use such function objects, for example, as a parameter to a sort routine to control the sorting order. We construct the function object in this way: greater<int>() or less<int>().

Type inference

The template facility of C++ provides a surprising amount of type inference. For example, consider the polymorphic max function given in Figure 90. First, the type of the arguments is simply stated to be of some class T. The template <class T>

```
T max (T x, T y)
{if (x>y) return x;
else return y;
}
```

Modern functional languages have adopted a number of features such as strong typing and modularity that have been found useful in conventional languages. In turn, conventional languages such as C++ and Ada have adopted some functional programming ideas that make it easier to treat functions as objects.

CHAPTER 2:

Logic and rule-based languages

Q1. EXPLAIN WHAT VERSUS HOW IN LOGIC AND RULE BASED LANGUAGE?

Logic and rule based languages. Such languages are different from procedural and functional languages not only in their conceptual foundations, but also in the programming style (or paradigm) they support. Programmers are more involved in describing the problem in a declarative fashion, then in defining details of algorithms to provide a solution. Thus, programs are more similar to specifications than to implementations in any conventional programming language. It is not surprising, as a consequence, that such languages are more demanding of computational resources than conventional languages.

The"what" versus "how" dilemma: specification versus Implementation

A software development process can be viewed abstractly as a sequence of phases through which system descriptions progressively become more and more detailed. Starting from a software requirements specification, which emphasizes *what* the the system is supposed to do, the description is progressively refined into a procedural and executable description, which describes *how* the problem actually is solved mechanically. Intermediate steps are often standardized within software development organizations, and suitable notations are used to describe their outcomes (software artifacts).

Typically, a design phase is specified to occur after requirements specification and before implementation, and suitable software design notations are provided to document the resulting software architecture. Thus the "what" stated in the requirements is transformed into the "how" stated in the design document, i.e., the design specification can be viewed as an abstract implementation of the requirements specification. In turn, this can be viewed as the specification for the subsequent implementation step, which takes the design specification and turns it into a running program.

Programming languages have become increasingly higher level. For example, a language like Ada, Eiffel, and C++ can be used in the design stage as a design specification language to describe the modular structure of the software and module interfaces in a precise and unambiguous way, even though the internals of the module (i.e., private data structures and algorithms) are yet to be defined. Such languages, in fact, allow the module specification (its interface) to be given and even compiled separately from the module implementation. The specification describes "what" the module does by describing the resources that it makes visible externally to other modules; the implementation describes "how" the internally declared data strucures and algorithms accomplish the specified tasks.

All of the stated steps of the process that lead from the initial requirements specification down to an implementation can be guided by suitable systematic methods. They cannot be done automatically, however: they require engineering skills and creativity by the programmer, whose responsibility is to map–translate–requirements into executable (usually, procedural) descriptions.

This mapping process is time-consuming, expensive, and error-prone activities.

In logic programming, problem descriptions are given in a logical formalism, based on first-order predicate calculus. The theories that can be used to describe and analyze logic languages formally are thus naturally rooted into mathematical logic. Our presentation, however, will avoid delving into deep mathematical concepts, and will mostly remain at the same level in which more conventional languages were studied.

The above informal introduction and motivations point out why logic programming is often said to support a *declarative* programming paradigm. As we will show, however, existing logic languages, such as PROLOG, match this description only partially. To make the efficiency of the program execution acceptable, a number of compromises are made which dilute the purity of the declarative approach.

Efficiency issues affect the way programs are written; that is, the programmer is concerned with more than just the specification of what the program is supposed to do. In addition, nondeclarative language features are also provided, which may be viewed as directions provided by the programmer to the inference engine. These features in general reduce the clarity of program descriptions.

To distinguish between specification and implementation, and to introduce logic programming, let us specify the effect of searching for an element x in a list L of elements. We introduce a predicate is_in (x, L) which is true whenever x is in the list L.

The predicate is described using a self-explaining hypothetical logic language, where operator "•" denotes the concatenation of two lists and operator [ ] transforms an element into a list containing it and "iff" is the conventional abbreviation for "if and only if.". for all elements x and lists L: is_in (x, L) iff L = [x]
or
L = L1 • L2 and
(is_in (x, L1) or is_in (x, L2))

The above specification describes a binary search in a declarative fashion. The element is in the list if the list consists exactly of that element. Otherwise, we can consider the list as decomposed into a left sublist and a right sublist, whose concatenation yields the original list. The element is in the list, if it is in either sublist.

Q2. WRITE SHORT NOTES ON PRINCIPLE FACTS OF LOGIC PROGRAMMING?

Principles of logic programming:

To understand exactly how logic programs can be formulated and how they can be executed, we need to define a possible reference syntax, and then base on it a precise specification of semantics. This would allow some of the concepts we used informally in (such as "procedural interpretation") to be stated rigorously.

This is the intended purpose provides the necessay background definitions and properties that are needed to understand how an interpreter of logic programs works. The interpreter provides a rigorous definition the program's "procedural interpretation". This is analogous to SIMPLESEM for imperative programs.

Preliminaries: facts, rules, queries, and deductions

Although there are many syntactic ways of using logic for problem descriptions, the field of logic programming has converged on PROLOG, which is based on a simple subset of the language of first-order logic. Hereafter we will gradually introduce the notation used by PROLOG.

The basic syntactic constituent of a PROLOG program is a *term*. A term is a constant, a variable, or a compound term. A compound term is written as a *functor symbol* followed by one or more arguments, which are themselves terms. A *ground term* is a term that does not contain variables. Constants are written as lower-case letter strings, representing atomic objects, or strings of digits (representing numbers). Variables are written as strings starting with an upper-case letter.

PROLOG programs are written as a sequence of *clauses*. A clause is expressed as either a single predicate, called *fact*, or as a *rule* (called *Horn clause*) of the form
conclusion :- condition
where :- stands for "if", conclusion is a single predicate, and condition is a conjunction of predicates, that is, a sequence of predicates separated by a comma, which stands for the logical and. Facts can be viewed as rules without a condition part (i.e., the condition is always true). Thus the term "rule" will be used to indicate both facts and rules, unless a distinction will be explicitly made. A rule's conclusion is also called the rule's *head*. Clauses are implicitly quantified universally.

A PROLOG rule
conclusion :- condition
containing variable X1, X2, . . ., Xn would be represented in the standard notation of mathematical logic as "X1, X2, . . ., Xn (condition É conclusion)
where É is the logical implication operator. In a procedural program, it would be represented as **if** condition **then** conclusion;

For example, the following program
length ([ ], 0). *--this is a fact*

length ([X | Y], N) :- length (Y, M), N = M + 1. *--this is a rule*
says that
• the length of the null string is zero,
• for all X, Y, N, M, if Mis the length of list Y and N is M + 1, then the length of a nonnull string with head X and tail Y is one more than the length of Y.

PROLOG

PROLOG is the most popular example of a logic language. Its basic syntactic features were introduced informally in Section 8.2. As we anticipated, PROLOG solves problems by performing a depth-first traversal of the search tree.

Whenever a goal is to be solved, the list of clauses that constitutes a program is searched from the top to the bottom. If unification succeeds, the subgoals corresponding to the terms in the righthand side of the selected rule (if any) are solved next, in a predined left-to-right order. This particular way of operating makes the behavior of the interpreter quite sensitive to the way programs are written. In particular, the ordering of clauses and subgoals can influence the way the interpreter works, although from a conceptual viewpoint clauses are connected by the logical operator "or" and subgoals are connected by "and". These connectives do not exhibit the expected commutativity property.

PROLOG provides several extra-logical features, which cause its departure from a pure logical language. The first fundamental departure is represented by the *cut* primitive, written as "!", which can appear as a predicate in the condition part of rules. The effect of the cut is to prune the search space by forbidding certain backtracking actions from occurring. Its motivation, of course, is to improve efficiency of the search by reducing the search space. It is the programmer's responsibility to ensure that such a reduction does not affect the result of the search.

The cut can be viewed as a goal that never fails and cannot be resatisfied. That is, if during backtracking one tries to resatisfy it, the goal that was unified with the lefthand side of the rule fails.

In order to illustrate how the cut works, consider the following rule:
A :- B, C, !, D, E
Suppose that, after a match between the rule's head and a goal A', subgoals B, C, and D (with suitably applied substitutions) have been solved successfully. If subgoal E fails, the PROLOG interpreter backtracks and tries to solve D by matching it to the next available rule's head, if any, found in scanning the program clauses from the top down. If no successful match can be found, the PROLOG interpreter would normally backtrack further, trying to find a new solution for C, and then B. Eventually, if all these fail, the match of A' with the rule would fail and another rule or fact would be tried. The presence of the cut, however, forbids the backtracking procedure from retrying C, then

B, and then a further alternative for the match with A': the current goal A' would fail right away.

In other terms, the cut, viewed as a predicate, always succeeds, and commits the PROLOG interpreter to all the choices made since the goal A' was unified with the head of the rule in which the cut occurs.

Q3. WRITE THE DIFFERENCES BETWEEN FUNCTIONAL PROGRAMMING AND LOGIC PROGRAMMING?

Functional programming versus logic programming:

The most striking difference between functional and logic programming is that programs in a pure functional programming language define functions, whereas in pure logic programming they define relations. In a sense, logic programming generalizes the approach taken by relational databases and their languages, like SQL.

For example, consider the simple PROLOG program shown in Figure, consisting of a sequence of facts. Indeed, a program of this kind can be viewed as defining a relational table; in the example, a mini database of classical music composers, which lists the composer's name, year of birth, and year of death.

In a function there is a clear distinction between the domain and the range. Executing a program consists of providing a value in the domain, whose corresponding value in the range is then evaluated. In a relation, there is no predefined notion of which is the input domain. In fact, all of these possible queries can be submitted for the program of Figure:

?- composer (mozart, 1756, 2001).
?- composer (mozart, X, Y).
?- composer X, Y, 1901).
?- composer (X, Y, Z).

In the first case, a complete tuple is provided, and a check is performed that the tuple exists in the database. In the second case, the name of the composer is provided as the input information, and the birth and death years are evaluated by the program. In the second case, we only provide the year of death, and ask the program to evaluate the name and year of birth of a composer whose year of death is given as input value. In the fourth case, we ask the system to provide the name, year of birth, and year of death of a composer listed in the database.

composer (monteverdi, 1567, 1643).
composer (bach, 1685, 1750).
composer (vivaldi, 1678, 1741).
composer (mozart, 1756, 1791).
composer (haydn, 1732, 1809).
composer (beethoven, 1770, 1827).

composer (schubert, 1797, 1828).
composer (schumann, 1810, 1856).
composer (brahms, 1833, 1897).
composer (verdi, 1813, 1901).
composer (debussy, 1862, 1918).

As most functional languages are not purely functional, PROLOG is not a pure logic language. Consequently, it is not fully relational in the above sense. In particular, the choice of the input domains of a query is not always free. This may happen if the program contains relational predicates, assignment predicates, or other extralogical features. For example the factorial program
?- fact (X, 6).
to find the integer whose factorial is 6. The query would in fact fail, because the extralogical predicate is fails. Similarly, the following query
?- max (X,99, 99).
for the program fragment of Figure 97 does not yield a value less than or equal to 99, as the logical reading might suggest. It fails, since one of the arguments in the invocation of ð is not bound to a value.

Q3. WRITE SHORT NOTES ON RULE BASED LANGUAGES?

Rule-based languages:

Rule-based languages are common tools for developing expert systems. Intuitively, an *expert system* is a program that behaves like an expert in some restricted application domain. Such a program is usually structured as a *knowledge base (KB)*, which comprises the knowledge that is specific to the application domain, and an *inference engine*. Given the description of the *current situation (CS)*, often called the database, expressed as a set of facts, the inference engine tries to match CS against the knowledge base to find the rules that can be *fired* to derive new information to be stored in the database, or to perform some action.

An important class of expert system languages (called *rule-based languages*, or *production systems*) uses the so-called *production rules*. Production rules are syntactically similar to PROLOG rules.

Typical forms are:
if condition then action
For example, the MYCIN system for medical consultation allows rules of this kind to be written:
if
description of symptom 1, and
description of symptom 2, and
. . .
description of symptom n

then there is suggestive evidence (0.7) that the identity of the bacterium is . . .

The example shows that one can state the "degree of certainty" of the conclusion of a rule. In general, the action part of a production rule can express any action that can be described in the language, such as updating CS or sending messages.

– *forward chaining*, and
– *backward chaining*.
Different rule-based languages provide either one of these methods or both. In order to understand forward and backward chaining, let us introduce a simple example described via production rules. The knowledge base provides a model of a supervisory system that can be in two different danger states, characterized by levels 0 and 1, indicated by the state of several switches and lights:

if switch_1_on and switch_2_on
then notify danger_level_0.
if switch_1_on and switch_3_on
then assert problem_1.
if light_red or alarm_on
then assert problem_2.
if problem_1 and problem_2
then notify danger_level_1.
An equivalent representation for the set of production rules is described by the and-or tree representation of Figure 100, which uses the convention introduced.

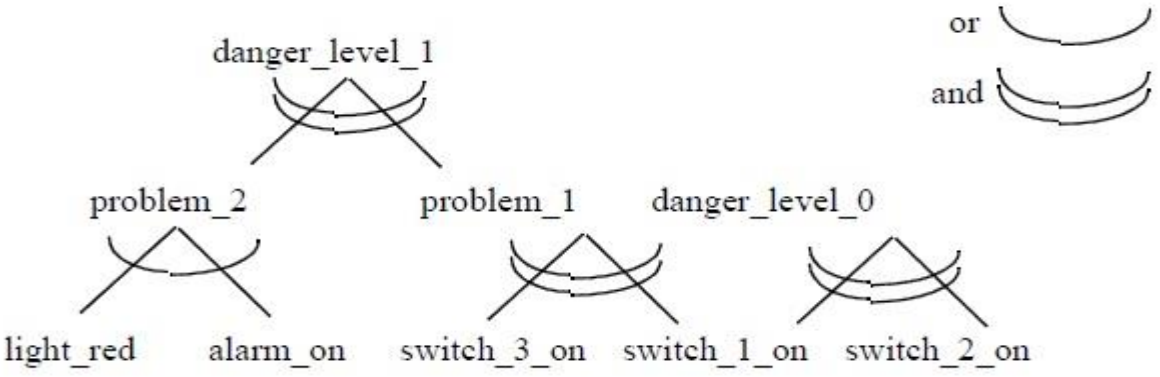


FIGURE 100. An and-or tree representation of production rules

The main difference between logic and rule-based languages is that logic languages are firmly based on the formal foundations of mathematical logic, while rule-based languages are not. Although they have a similar external appearance, being based on rules of the form "if *condition* then *action*", in most cases rule-based languages allow any kind of state-changing actions to be specified to occur in the *action* part.

■ ALL THE BEST