UNIT 1

PRELIMINARY CONCEPTS, SYNTAX AND SEMANTICS

PART-A
SHORT QUESTIONS WITH SOLUTIONS

Q1. What are the reasons for studying the concepts of programming languages?

Answer :

The reasons for studying the concepts of programming languages are,

1. To increase the capacity of expressing ideas.
2. To improve the knowledge of selecting an appropriate language.
3. To increase the learning ability of a programmer to learn new languages.
4. To have a better understanding of the importance of implementation.
5. To increase the designing ability of a programmer for designing new languages.
6. To achieve an overall advancement in computing.

Q2. What is programming domain? List various types of it.

Answer :

Model Paper-II, Q1(a)

Programming Domains

The appropriate language to use often depends on the application domain for the problem to be solved. This is because different languages are defined for taking care of various computer applications.

Types of Programming Domains

The various types of programming domains are,

1. Scientific applications
2. Business applications
3. Artificial intelligence
4. System programming
5. Web software.

Q3. List out language categories.

Answer :

(Model Paper-I, Q1(a) | Nov./Dec.-16(R13), Q1(b))

Programming languages are categorized into the following four categories,

1. Imperative programming
2. Functional programming
3. Logic programming
4. Object-oriented programming.

1. **Imperative Programming**

   Imperative programming also known as procedural programming is a programming paradigm based on procedural call. It describes the steps to be followed by a program to acquired its desired state. Imperative programming can be written and maintained easily.

2. **Functional Programming**

   Functional programming is a programming paradigm that performs computations as the evaluation of mathematical functions.

3. **Logic Programming**

   Logic programming considers mathematical logic for computer programming. It is a rule-based language wherein the rules are defined without any order. The implementation of the language must select an execution order that generates the desired output. Prolog language is an example of logic programming.

4. **Object-Oriented Programming**

   Object-oriented programming is a programming paradigm that designs computer programs and applications using objects and interactions between them. Features like encapsulation, polymorphism, inheritance and modularity are included in object-oriented programming languages.

Q4. List the principle phases of compilation.

Answer : (Model Paper-III, Q1(a) | Nov./Dec.-17(R15), Q1(a))

In compilation, programs written by the programmers are translated into machine language that can be executed directly on the computer. This implementation is very fast. Most of the popular languages such as C, FORTRAN, COBOL and ADA uses compiler implementation.

The principle phases involved in compilation are as follows,

1. Source program
2. Lexical Analyzer
3. Syntax Analyzer
4. Intermediate Code Generator
5. Semantic Analyzer
6. Code Optimizer
7. Code Generator
8. Symbol Table.

Q5. What is programming environment? List the types of programming environments.

Answer :

Programming Environment

It refers to the collection of tools used for developing software. The collection may include only a file system, a linker, a text editor and a compiler or it may comprise of a group of integrated tools which can be accessed through a uniform user interface.

Types of Programming Environments

There are several programming environments, such as,

1. UNIX
2. Borland JBuilder
3. Microsoft Visual Studio
4. NetBeans.

Q6. Write six features of Java.

Answer :

The features of Java are given as follows,

Java is smaller, simpler and more reliable.

1. It contains both classes and primitive types.
2. In Java, arrays are instances of predefined class.
3. It does not support pointers, instead references are used to point class instances. Some times references behave like ordinary scalar variable. Because, they implicitly dereferenced whenever required.
4. It consists of primitive boolean type named Boolean, which are used in control expressions of its control structures.
5. In Java, subprograms are known as methods and are defined in classes and methods which are called through class.
6. Standalone subprograms are not supported in Java.

Q7. Define syntax and semantics.

(Model Paper-II, Q1(b) | Nov./Dec.-16(R13), Q1(a) | Nov.-15(R13), Q1(a))

Answer :

Syntax

The syntax of a language is a set of rules that defines the form of a language. They define low expressions, sentences, statements and program units are formed by the fundamental units called words/lexemes.

Semantics

Semantics of a programming language refers to the meaning designated to various syntactic constructs such as expressions, statements and program units. The syntax of if-statement in C is,

if(<expression>) <statement>

The semantics of the above statement specifies that "If the expression results to true then the statements which follows are executed".

The syntax and semantics of a programming language are closely related to each other i.e., the languages semantics should follow from syntax.

Q8. Define derivation and a parse tree.

(Model Paper-I, Q1(b) | Nov.-15(R13), Q1(b))

Answer :

Derivation

The process of generating sentences is called a derivation. The grammar for a complete programming language consists of a start symbol called < program> which represents the complete program.

Parse Tree

The sentences in a language can be hierarchically described by using 'parse trees'.

For a grammar $G = (V, \Sigma, P, S)$, the parse tree representation is as follows.

(i) The root node of a parse-tree is marked with the start symbol 'S' of a grammar.

(ii) Each internal node is marked by a variable in V.

(iii) Each leaf (terminal node) is marked by a terminal or a variable or $\in$. If $\in$ is used to mark a leaf then it specifies that it is the only child of its parent

(iv) If an internal node is marked as 'x' and its children are labelled from left as $x_1, x_2, x_3, \ldots x_n$ then $x \rightarrow x_1, x_2, \ldots, x_n$ is a production in P.

Each subtree of a parse tree denotes an abstraction instance in a statement.

Q9. Define ambiguous grammar with an example.

Answer :

Ambiguity

A grammar that results in a sentence for which there are two or more parse trees, is said to be 'ambiguous'. Such grammars allows the parse trees to grow in both left and right directions.

Example

Consider the following ambiguous grammar.

<assign> → <id> = <expr>

<ids> → P|Q|R|S

<expr> → <expr> + <expr>|<expr>*<expr>|(<expr>)|<id>

This grammar is ambiguous because, any sentence of the form $P = Q + R * S$ results in two distinct parse trees, shown in figure (a) and figure (b).

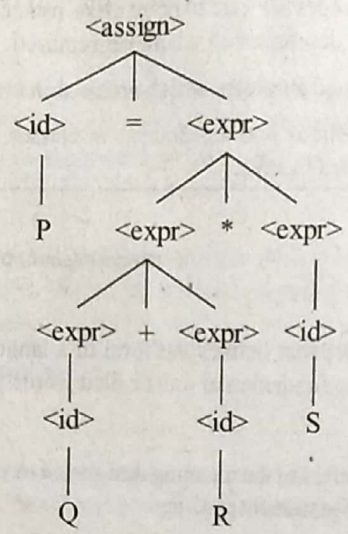


Figure (a): Parse Tree-1

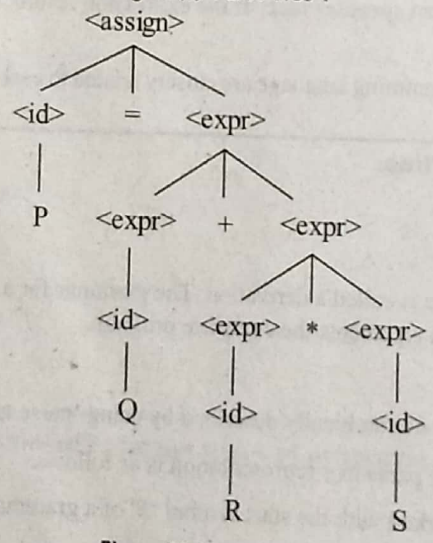


Figure (b): Parse Tree-2

If a language structure consists of many parse trees, for a single statement, then the meaning of a structure cannot be predicted uniquely.

Q10. Explain the features of denotational semantics.

Answer :

(Model Paper-III, Q1(b) | Nov./Dec.-17(R15), Q1(b))

Denotational semantics describes the meaning of a program as a mathematical object. It is a function that takes the systems state as its input and produces an updated state as its output after executing the program.

Recursive function theory form the basis for denotational semantics. It loosely corresponds to 'compilation'.

1. Denotational semantics can be used to prove the program correctness.
2. It acts as an aid in language designing.
3. It provides a clear and thorough way of thinking about programs.
4. It finds its applications in compiler generation systems.

PART-B

ESSAY QUESTIONS WITH SOLUTIONS

1.1 PRELIMINARY CONCEPTS

1.1.1 Reasons for Studying Concepts of Programming Languages

Q11. What are the reasons for studying the concepts of programming languages.

Model Paper-I, Q2(a)

Answer :

The reasons for studying programming languages are given below,

1. **To Increase the Capacity of Expressing Ideas**

Programmers while developing software feel difficult to idealize structures either verbally or in writing. The language used by a programmers while developing software, imposes restrictions on some control structures, data structures and abstractions. Hence, the algorithms constructed are also restricted. These restrictions or limitations can be minimized by understanding the features of various programming languages and by learning new constructs in the programming languages.

The features of one language can be simulated in other languages, without affecting the importance of designing languages. However, it is a best practice to use a feature whose design is integrated into a language rather than using a simulation of that feature (as it makes the language less elegant and unmanageable).

2. **To Improve the knowledge of Selecting an Appropriate Language**

Programmers with knowledge in few languages cannot judge a well suited programming language for the assigned project and hence they often use the language with which they are well known or acquainted, even if it is not appropriate for the new project.

If the programmer is familiar with the other available languages and particularly some features of these languages, then programmer can make a better choice in selecting an appropriate language.

3. **To Increase the Learning Ability of a Programmer to Learn New Languages**

Now-a-days, programming languages are in a state of continuous evolution. Learning a new programming language is a lengthier and a difficult process, especially for those who are familiar with a few languages and who never assessed the concepts of programming language.

After understanding the basic concepts of a programming language it would be easier to analyze how these (concepts or) features are integrated in the new language.

4. **To Have a Better Understanding of the Importance of Implementation**

The implementation issues that affects a programming language must be considered while learning it. This increases the ability of a programmer to intelligently use a language. Certain kinds of bugs encountered during program execution can be easily found by the programmer who is aware of the related implementation details.

It also allows to conceptualize how various constructs of a programming language are executed by a computer. For example, a programmer who is unaware of the implementation of recursion doesn't know that a recursive algorithm is very much slower than its equivalent iterative version.

5. **To Increase the Designing Ability of a Programmer for Designing New Languages**

Most professional programmers, sometimes design languages of one kind or the other. For example, most software systems required only small amount of data and commands to be entered by a user in order to obtain the desired output.

Designing a user interface is a complex design problem in some systems (like word processor) in which a user needs to traverse several levels of menus and enter a variety of commands.

Hence, a thorough analysis of a programming language is needed in order to design a language for some complex systems, as it helps the users to assess and evaluate such products.

6. **To Achieve an Overall Advancement in Computing**

The popularity of a programming language can be easily determined but the most popular languages are not always the best available languages.

Sometimes, a programming language is used widely because the programmers are not well aware of the concepts of programming languages in order to choose the best language for programming.

1.1.2 Programming Domains

Q12. Discuss about various programming domains.

Answer :

Programming Domains

The appropriate language to use often depends on the application domain for the problem to be solved. This is because different languages are defined for taking care of various computer applications.

1. Scientific Applications

Scientific programming is primarily concerned with making complex calculations very fast and accurate. The calculations are defined by mathematical models that represent scientific phenomena. They are primarily implemented using the imperative programming paradigm. FORTRAN was the first higher level language used for scientific applications. The syntax of FORTRAN is similar to Mathematics and the scientists find it easy to use.

2. Business Applications

Business applications refer to the large processing applications such as order-entry programs, inventory control and payroll. The language used to develop these applications is COBOL. It is still used in data processing applications, but after the invention of 4G languages, the database systems and spread sheets are most widely used.

3. Artificial Intelligence

The artificial intelligence programming community which is responsible for developing programs that can model human intelligent behavior and logical deductions has been under working for a long period of time. The activities such as symbol manipulations, functional expressions and the logical proof systems design are the essential requirements for artificial intelligence applications. Functional and logic programming results from artificial intelligence applications. The first functional programming language developed for AI applications was LISP and then the PROLOG logic programming was invented.

4. System Programming

System programming refers to the activity of designing and maintaining the basic software that runs the components of operating system, network software, language compilers and debuggers, virtual machines and interpreters, etc. The people involved in this activity are called system programmers. These programs can either be written in an assembly language or in some specific language of system programming. 'C' is the most widely used system programming language.

5. Web Software

Various collection of languages support WWW ranging from markup languages such as HTML, which is not a programming language to general purpose such as Java. Due to need of dynamic web content, some computing capabilities are included in content presentation technology. This functionality can be provided by embedding programming code in an HTML document which is in the form of scripting languages like JavaScript and PHP.

6. Special-purpose Languages

These languages are designed for a specific problem domain and have narrow applicability. These languages are ranging from RPG(A language used to general business reports) to APT(A language used for instructing programmable machine tools) and to GPSS (A language used for systems simulation). The problem with these languages is that they are difficult to compare with other programming languages.

PROLOG is the most widely used special purpose language which was designed to serve the narrow range of natural language processing

1.1.3 Language Evaluation Criteria

Q13. Briefly discuss the list of criteria for languages.

Answer :

Language Evaluation Criteria

A programming language possess many characteristics. These characteristics can be evaluated based on the language evaluation criteria.

1. Readability

For answer refer Unit-I, Q14.

2. Writability

For answer refer Unit-I, Q15.

3. Reliability

For answer refer Unit-I, Q16.

4. Cost

For answer refer Unit-I, Q17.

Q14. Describe the characteristics of readability.

Answer :

Readability

It is the most important criteria for assessing a language i.e., how well a language can be read and understood. The fundamental characteristics of programming languages are its efficiency and machine readability.

Characteristics of Readability

The following characteristics are also included in readability,

(i) Simplicity and orthogonality

(ii) Control statements

(iii) Data types and structures.

(i) Simplicity and Orthogonality

The languages readability is greatly influenced by its simplicity. A language that contains a large number of fundamental components is difficult to learn than the language containing small number of fundamental components. Hence, programmers who uses a language must learn only a subset of the language, thereby neglecting all other characteristics.

(ii) Control Statements

Unnecessary use of GOTO statements greatly reduces program readability. Reading and understanding a program from top to bottom is easier than jumping from one statement to the other nonadjacent statement. Hence the use of GOTO statement must be restricted in order to make a program more readable. It can be restricted in the following ways,

- The GOTO statement must always occur before their targets except in the case when they are used to form loops.
- The number of GOTO statements must be limited.
- The targets of GOTO statements must be nearer.

BASIC and FORTRAN in early 1970s doesn't have control statements. Most of the programming languages today have sufficient control statements. Hence, the need for the GOTO statement is eliminated.

(iii) Data Types and Structures

Language's readability is also affected by the data types and data structures. A language which doesn't support boolean data type uses a numeric type to represent a flag as follows.

For a language that doesn't support 'bool' data type.

```
sum_is_small = 1
```

For a language that supports 'bool' data type.

```
sum_is_small = true
```

Similarly, 'record' data types makes a language more readable than the collection of similar arrays.

Syntax Design

The two syntactic design choices that have a significant affect on readability are as follows,

1. Special Words

Program's appearance and readability can be affected by the special words used in a language especially the method of making compound statements, statement groups etc.

Example

PASCAL uses begin_end pairs for forming the groups of all control constructs except **repeat** statement. C uses opening and closing braces for the same purpose.

When the special words are used as names of program variables, they make it very confusing.

2. Form and Meaning

The program's readability can be enhanced by forming the statements in such a way that their appearance specifies their purpose. Language semantics must follow its syntax but, this rule is violated when two constructs have similar appearance with different meanings.

Example

The keyword **static** in C language serves different meanings in different contexts. If it is associated with a variable inside a function, it indicates that the variable is created at the compile time else if it is used for a variable defined outside all functions then it indicates that the variable can be seen only within the file in which it is defined.

Q15. Describe the important factors influencing the writability of a language.

Answer : Nov./Dec.-17(R15), Q3(b)

Writability

Writability refers to the ease with which a language can be used to create programs. The characteristics that affect readability also affect writability. The process of writing a program involves reading the part of the program that is already written. The writability of two languages can't be compared in the realm of a particular application when one of them was designed for that application and the other was not.

The following characteristics are included in writability,

(i) Simplicity and orthogonality

(ii) Support for abstraction

(iii) Expressivity.

(i) Simplicity and Orthogonality

If a programming language consists of a large number of constructs then some of the programmers may not be well aware of all these constructs. This may lead to the misuse of some of the language features and many of the other features are not used which may be efficient or elegant. Having a small number of primitive constructs with a set of rules is more efficient than having large primitives. A complex problem can be solved by using a simple set of primitive constructs.

Too much orthogonality also affects writability.

(ii) Support for Abstraction

Abstraction refers to the process of defining and using complicated structures or operations in such a way that many of the implementation details are ignored. Hence, the degree of abstraction of its expression greatly affects writability. An example of abstraction is the repeated use of a subprogram for implementing a sort algorithm.

(iii) Expressivity

Language expressivity refers to many different characteristics. In APL, it means the presence of many powerful operators that deals with large number of computations that can be achieved with a very small program. In C, the representation count ++ is more convenient and shorter than count = count +1. All these increases the writability of a language.

Q16. Discuss the factors influencing the reliability of a lanaguage.

Answer :

Reliability

A program is said to be reliable if it is working according to its specifications under all conditions. A programming language must be designed in such a way that the syntactic and logical errors can be detected easily. A language that is reliable can easily handle run-time errors.

In the realm of programming languages, reliability is a mechanism that upgrades writing, maintaining and debugging of correct programs and handles the exceptions while executing the programs.

Reliability can be affected by certain features which are described below,

(i) Type Checking

It is a process of checking a program for type errors occurred during compilation and program execution. It has a significant effect on reliability. Compile time type checking is desirable because run time type checking is expensive.

(ii) Readability and Writability

❖ **Readability**

For answer refer Unit-I, Q14, Topic: Readability.

❖ **Writability**

For answer refer Unit-I, Q15, Topic: Writability.

(iii) Exception Handling

Exception handling refers to the ability of a program to detect unusual conditions and run-time errors. This characteristic affects reliability. The languages such as Ada, C++ and Java have extensive exception handling capabilities.

(iv) Aliasing

Aliasing is a technique in which a single memory cell is defined by two or more discrete referencing methods or names. It is a dangerous feature in a programming language. Use of aliasing helps in eliminating and controlling the limitations in the data abstraction facilities of a language. Reliability of a language can be increased by restricting the aliasing.

Q17. Explain the characteristics to determine the cost of a programming language.

Answer :

The total cost of a programming language can be determined from many characteristics which are as follows,

(i) The cost of training programmers about how to use the language is a function of language's simplicity, orthogonality and the experience of the programmers.

(ii) The cost of writing programs in a language is a function of the language's writability. The high-level languages must be designed in such a way that it reduces the cost of creating the software.

(iii) The cost of compiling programs in a language.

(iv) The cost of executing the programs in a language. This cost is greatly affected by the language's design. A language which involves many run-time type checks will prevent the fast execution of code irrespective of the compiler's quality. Hence, optimization in which a collection of methods are used by the compilers to reduce the size and increase the execution speed of the code.

(v) The cost of language implementation system. A language whose implementation system is costly or runs only on expensive hardware will not be widely used.

(vi) The cost of poor reliability, failure of the software in critical systems results in huge costs.

(vii) The cost of maintaining the programs. This cost involves both corrections and modifications to add new capabilities. It mainly depends on readability.

Of all the above mentioned cost contributors, the program development cost, maintenance cost and reliability are most important. Hence, the total cost can be summarized as follows.

Total Cost = Cost of training programmers + Cost of writing programs + Cost of compiling programs + Cost of executing programs + Poor reliability cost + Cost of maintaining software + Cost of language implementation system.

Q18. The levels of acceptance of any language depend on the language description. Comment on this.

Nov.-15(R13), Q3(b)

Answer :

The success of any programming language depends on the concise and understandable way of describing it. ALGOL 60 and ALGOL 68 were initially developed by using the precise formal descriptions but are not easily understandable because of the new notations employed by both of them. Hence, both ALGOL 60 and ALGOL 68 suffered from this problem. However, there are some other languages that can be understood easily, but provides imprecise definitions.

Another problem associated with any language description is that, it must be easily understandable by a variety of people belonging to different classes. Hence, most of the new programming languages must be scrutinized by potential users before their designs are completed. The success of this step is based on the clarity of the language's description. The implementor of the programming languages must identify how the expressions, statements and other program units are formed and how they effect when executed.

Also, the language users must determine the process of encoding software systems by making a reference to a language reference manual.

Syntax and semantics are the two important aspects of studying the programming languages. Though they are different, they have a close relationship between them.

1.1.4 Influences on Language Design

Q19. What are the major influences on language design?

Model Paper-II, Q2(a)

Answer :

For answer refer Unit-I, Q13.

In addition to the above factors, the basic design of programming languages are influenced by the following factors.

1. Computer architecture
2. Programming methodologies.

1. Computer Architecture

The language design can be greatly influenced by the basic computer architecture. Von-Neumann computer architecture is the most commonly used architecture. The languages which are designed according to Von-Neumann architecture are called imperative languages.

In Von-Neumann architecture, same memory area is shared between both data and programs. The CPU whose function is to execute instructions, is separated from the memory. Hence, instructions and data must be transmitted from memory to CPU and the result obtained after computation must be sent back to memory. All computers since 1940 are based on Von-Neumann architecture. The layout of a Von-Neumann computer is shown in the following figure.

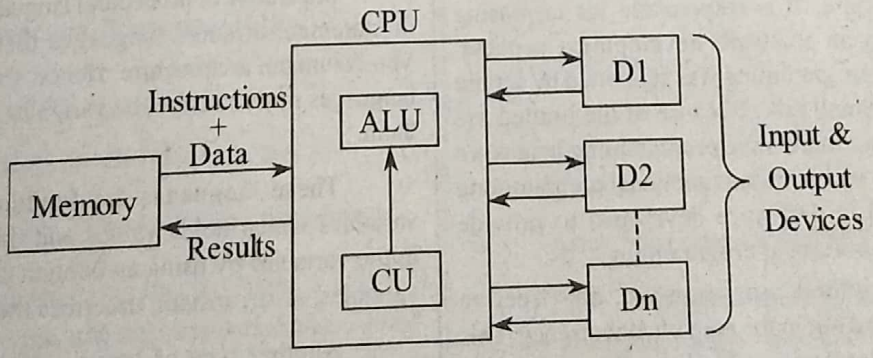


Figure: The Von-Neumann Computer Architecture

Due to the main characteristics of Von Neumann architecture which made imperative languages as variables that helps in modelling assignment statements, memory cells depending on the piping operation. It also helps in modelling the efficient form of repetition. Expression operands are transmitted from memory to the CPU and the result is sent back to the memory cell expressed at the left side of the assignment statement.

Advantage

- The process of iteration is fast on this architecture as the instructions are stored in neighboring memory cells hence, eliminating the use of recursion.

2. Programming Methodologies

In late 1960s to early 1970s, several programming methods came into existence in order to overcome the deficiencies and to enhance the features of programming languages.

In the early 1960s, computers were used to solve simple problems such as solving a set of equations to simulate satellite tracks. Later, in the late 1960s and early 1970s, a detailed analysis on structured programming related to both software development process and programming language design were conducted. This was done because of the increase in programming costs and decrease in hardware costs. Further, more difficult problems, such as providing-worldwide airline reservation systems and controlling petroleum refining facilities were solved by the computers.

During the analysis of 1970s, new programming methodologies called *top-down design* and *step wise refinement* were developed. Several deficiencies such as incompleteness of type checking and inadequacy of control statements were identified.

Later, in the late 1970's, data-oriented programming methodology came into existence. These methods focused mainly on the usage of data abstraction and data design. A language that is data-oriented must support the concept of data-abstraction. SIMULA 67 was the first language which gave support to data abstraction till its development, the advantages of data abstraction were not identified.

In the early 1980s, object-oriented methodology came into existence which is the extension of data-oriented methodology. A language that is object-oriented must support three features namely data abstraction, inheritance and dynamic method binding.

(a) Data Abstraction

It encapsulates data objects and controls access to data.

(b) Inheritance

In order to achieve code reusability, inheritance is an effective feature. It is responsible for increasing the productivity in software development process. Object-oriented programming was supported by certain languages like, Small talk. Because of the limited use of small talk, other imperative programming languages like Ada 95, C++ and Java and functional programming language named CLOS were developed to provide support for object-oriented programming.

In procedure-oriented programming, an effective analysis has carried out in the area of concurrency. This leads to the need for language facilities for constructing and controlling concurrent program units. These capabilities are supported by some languages namely Ada and Java.

(c) Dynamic Method Binding

Dynamic binding is the binding which occurs at run-time. It is used to build dynamic type hierarchies and to form 'abstract data types'. It is also used to provide more functional and efficient implementation to be selected out run-time of the derived classes.

Q20. Discuss the role of programming languages to support particular computer architecture.

Answer : Nov./Dec.-18(R15), Q2

Programming Languages

A programming language should be consistent and possess many characteristics to support architecture of computer system such as,

1. Readability
2. Writability
3. Reliability
4. Cost.

1. Readability

For answer refer Unit-I, Q14, Topic: Readability.

2. Writability

For answer refer Unit-I, Q15, Topic: Writability.

3. Reliability

For answer refer Unit-I, Q16, Topic: Reliability.

4. Cost

For answer refer Unit-I, Q17.

1.1.5 Language Categories

Q21. Discuss various language categories.

Answer : Model Paper-III, Q2(a)

Languages are categorized into different types. They are,

1. Imperative or Procedural Languages

Imperative or procedural languages are command driven or statement-oriented languages that are directly based on Von-Neumann architecture. Hence, the programmers of these languages must manage the variables and assign the values to them.

These languages are fundamentally dependent on variables which holds values and these values are assigned to the variables by using an explicit assignment operator. The variables, at any instant, describes the state of computation.

Another type of language, visual language which is the subcategory of the imperative languages. .NET languages are considered as the most popular visual languages. These languages are capable of generating code segments drag and drop. Such type of languages were referred to as fourth generation languages. These visual languages provide an easy way of generating programming graphical user interface. For example, using visual studio to develop software in the .NET languages, a single keystroke can be used to create a form control display code such as button or text box. These capabilities are available in all .NET languages.

Few Authors consider scripting languages as a separate programming language. In this category, languages are linked together by their method of implementation, particular or complete interpretation, than by a common languages design. The languages which are referred to Scripting languages such as perl, JavaScript and Ruby are also considered as imperative languages in every sense.

Syntax

Statement 1;

Statement 2;

Examples

FORTRAN, PASCAL, C, ALGOL, COBOL, etc.

2. Applicative or Functional Languages

A functional or applicative language is one in which the primary means of making computation is by applying functions to given parameters. Programming in functional languages can be done without using any kind of variables, assignment statements and iterations. Thus, the functional languages, instead of focusing on the changes in state, emphasizes on the function that must be applied to the initial machine state. In these languages, the programmers need not be concerned with the variables because the memory cells are not abstracted into the language. The development of a program proceeds by generating the functions from previously created functions to obtain more complex functions that can manipulate the initial data to obtain the final result.

Syntax

function_n(..... function_2(function_1(data)).....)

Examples

LISP(List processor) & ML(Meta language)

3. Rule-based or Logic Programming Languages

In rule-based languages, no specific order is followed in specifying rules. Thus, the language implementation system must select some order of execution that leads to the appropriate result. These languages can execute an action based on the satisfaction of certain enabling conditions. PROLOG is the most common rule-based (logic) programming language which consists of a class of predicate logic expressions as its enabling conditions. Execution of a rule based language is similar to an imperative language except that the statements are not sequential. Enabling conditions determine the order of execution.

Syntax

enabling condition_1 → Action_1

enabling condition_2 → Action_2

enabling condition_3 → Action_3

⋮

enabling condition_n → Action_n.

Example

PROLOG.

4. Object Oriented Programming

In Object Oriented Programming, everything is considered as objects. In this case, complex data objects are created and set of functions that can operate on those created objects are designed. Complex objects can inherit properties of the simpler objects. By creating concrete data objects, the object-oriented programs can gain efficiency over imperative languages.

Syntax

class class_name

{

access_specifier-1:

member 1;

access_specifier-2:

member 2;

...

}

object_names;

Examples

C++, Java, Ada and Smalltalk.

5. Special-purpose Languages

These languages are designed for a specific problem domain and have narrow applicability. These languages are ranging from RPG(A language used to general business reports) to APT(A language used for instructing programmable machine tools) and to GPSS (A language used for systems simulation). The problem with these languages is that they are difficult to compare with other programming languages.

PROLOG is the most widely used special purpose language which was designed to serve the narrow range of natural language processing.

1.1.6 Language Design Trade-offs

Q22. Explain in detail about language design trade-offs.

Answer :

Some of the language design trade-offs are as follows,

1. Readability Vs Cost of Execution

In Java language, it is very important to ensure that all the references to array elements lies in the prescribed index range which will gradually increase the cost of execution as each program comprising of references to array elements needs to be checked on the other hand, 'C' language does not require index checking that leads to faster execution of programs. Thus designers compromised execution efficiency for reliability.

2. Writability Vs Readability

APL involved large number of powerful operators for array operands thus making it necessary to involve new set of symbols in order to signify them. APL operators can also be used to express single long complex expressions resulting into two different outcomes. One is it helps applications to be developed by writing a small program instead of writing a big program involving various operators. Another result is a small program involving operators that makes it really difficult to understand for any other person except the writer himself. Thus the designer traded readability for writability.

1.1.7 Implementation Methods

Q23. What are the three general methods of implementing a programming language?

Answer : (Model Paper-I, Q3(a) | Nov.-15(R13), Q3(a))

Programming languages can be implemented by any one of the following methods.

(a) Compiler implementation

(b) Pure interpretation

(c) Hybrid implementation.

(a) Compilation Implementation

In compilation programs written by the programmers are translated into machine language that can be executed directly on the computer. This implementation is very fast. Most of the popular languages such as C, FORTRAN, COBOL and ADA uses compiler implementation.

The phases involved in the process of compilation are shown in figure (a).

(i) Source Program

The program that is translated by a compiler is called a 'source program'.

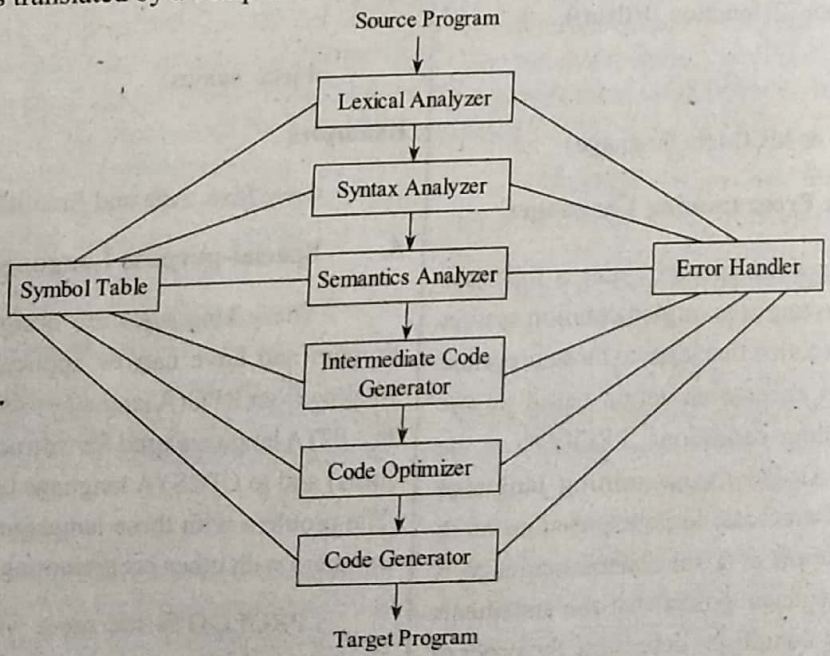


Figure (a): The Compilation Process

(ii) **Lexical Analyzer**

The lexical analyzer accumulates the characters to form lexical units which are identifiers, operators, special words and punctuation symbols. The comments in a source program are ignored by a lexical analyzer.

(iii) **Syntax Analyzer**

The syntax analyzer uses the lexical units, generated by a lexical analyzer and constructs the hierarchical structures called 'parse trees' which represents the syntactic structure of a program.

(iv) **Intermediate Code Generator**

At an intermediate stage between the source program and the machine language program, a program in a different language is generated by an intermediate code generator.

(v) **Semantic Analyzer**

It is an integral part of the intermediate code generator which checks the validity and meaning of the parse tree i.e., it checks for the errors that are difficult to find during syntax analysis such as type errors.

(vi) **Code Optimizer**

It is a discretionary part of compilation which improves programs by making them smaller, faster or both. This type of compiler is often used when the compilation speed is more important than the execution speed.

(vii) **Code Generator**

It is responsible for translating the optimized intermediate code form of a program into its equivalent machine language program.

(viii) **Symbol Table**

It acts as a database for the compilation process. The type and attribute information associated with each user-defined name are stored in symbol table by the lexical and syntax analyzers. This information is used by the semantic analyzer and the code generator.

Though the machine language generated by a compiler can be directly run on the hardware, but many programs require the previously executed programs that are stored in standard libraries. These programs can be linked with the user programs by the linker and the process is called 'linking'. After linking the machine code is ready for execution.

(b) **Pure Interpretation**

Programs written by the programmers are interpreted by a program called an 'interpreter' which serves as a software simulation of a machine. The fetch-execute cycle of an interpreter deals with the high-level program statements but not with the machine instructions. The software simulation acts as a virtual machine for the language. This method is called the 'pure interpretation'.

Figure (b) shows the process of pure interpretation in which an interpreter takes the source program along with the other data as input to produce the desired output. It reads and executes the input statements one at a time.

Comparison between Compilation and Interpretation

1. **Interpretation can be More Flexible than Compilation**

Interpretation has greater flexibility and better error detection capability than compilation. An interpreter directly runs the source program making it possible to make changes in a program there by adding new features with more error correction ability.

In a compiler, all the translation is completed before running the target program which prevents it from being readily adapted as it runs.

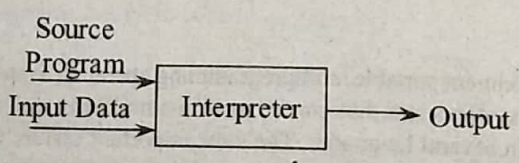


Figure (b): Pure Interpretation

2. Compilation can be more Efficient than Interpretation

The performance of compilation is better than the interpretation. A decision made at the compile time can't be necessarily made at the run-time. For example, if the compiler can guarantee that the variable '*a*', will always lie at a location 2006, it can generate machine language instructions that can access this location, whenever the source program refers to '*a*'. On a contrary, an interpreter may need to look up in a table every time, when '*a*' is accessed in order to determine its location. Since the program is compiled only once but can be executed many times, the saving in time can be substantial.

The disadvantage of interpretation is that it requires more space. This is due to the fact that it requires source program along with the symbol table for interpretation. The source program must be stored in such a way that it can provide easy access and modification.

The disadvantage of compilation is that it requires machine time to compile a source program into target code.

(c) Hybrid Implementation

Hybrid implementation translates the programs written in high-level language to an intermediate language for easy interpretation. It performs faster implementations compared to pure interpretation as source language statements are decoded only one time. Hybrid implementation as shown in the figure below interprets the intermediate code rather than translating the intermediate code to machine code.

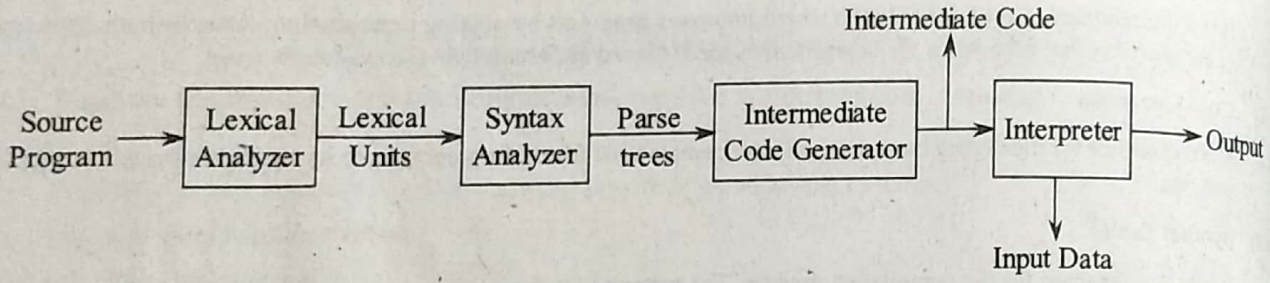


Figure (c): Hybrid Implementation

The language 'perl' is implemented with a hybrid system. Initially all the java implementation were in hybrid form with its byte code providing portability to any machine that has a byte-code interpreter and a run-time system (called the Java-virtual Machine).

An implementor may sometimes offers both compiled and interpreted implementation for a language. In such situations, an interpreter generates and debugs the program. Finally, they are compiled to increase their execution speed.

1.1.8 Programming Environments

Q24. Explain about programming environments.

Model Paper-II, Q3(a)

Answer :

Programming Environment

It refers to the collection of tools used for developing software. The collection may include only a file system, a linker, a text editor and a compiler or it may comprise of a group of integrated tools which can be accessed through a uniform user interface.

There are several programming environments,

1. UNIX
2. Borland JBuilder
3. MicrosoftVisualStudio
4. NetBeans.

1. UNIX

It is an older programming environment portable multiprogramming operating system distributed during early 1970's. Unix was specially designed for mainframe systems and thus can be used in a multiuser environment. It provides tools that support software production and maintenance in several languages. The most important feature that UNIX does not possess in the past is uniform interface among its tools. However, presently UNIX is used through GUI (Graphical User Interface). Some examples of UNIX GUI's are GNOME, Solaris CDE (Common Desktop Environment) and KDE.

2. Borland JBuilder

It is a programming environment used for the development of java, it provides debugger, editor, integrated compiler and file system, these are all accessed through a graphical user interface. It is a complex system and most powerful for creating software for java.

3. Microsoft Visual Studio .NET

It is a recent innovation in the field of software development environments. It consists of huge collection of software tools used through a windowed interface. The five .NET languages allow to develop software by using any one of the language such as C#, .NET, Jscript, managed C++ and F#.

4. NetBeans

NetBeans programming environment is mainly used for java application development. It also support JavaScript, Ruby and PHP. Visual studio and Netbeans are not considered as programming environment, but it is considered as framework, which provides common parts of the application code.

1.1.9 Evolution of Major Programming Languages

Q25. Discuss about Zuse's Plankalkul.

Answer :

Plankalkul was developed in 1945 but came into existence in 1972.

History of Zuse's Plankalkul

Konrad Zuse, a German scientist developed a set of complex and dynamic computers in between 1936 and 1945. But due to the allied bombing, all of his models were destroyed except Z4. All the group members separated so he alone started working and tried to build a computing language for the Z4 as a proposal for phd from a village. This language is named as "PlanKalkul" which is known as program calculus. The algorithms in this language were described by zuse to solve huge set of problems.

Language Overview

Plankalkul consists of advanced features of data structures. The data type of it is single bit and from this the integer and floating point numeric types were developed. The floating-point type have used two's complement notation along with "hiddenbit" scheme in order to avoid the storage of most significant bit of normalized fraction part. It also includes arrays and records that can have nested records. This language includes an iterative statement and selection statement but there is no explicit goto or an else clause.

Mathematical expressions is the main feature of Zuses' programs and it represents the relationships between program variables. These mathematical expressions are similar to that of an axiomatic semantics and assertions of Java.

Zuse's manuscript programs were complex than earlier programs such as sorting arrays of numbers, graph connectivity testing, floating-point and integer operations and 49 pages of algorithm. For playing chess plankalkul description contains two or three lines of code for every statement. The first line was similar to the statement of present languages. The second line represents the subscripts of the array references in first line. This method was also used by Charles Babbage in 19th Century to indicate subscripts in his program for Analytical Engine. The last line represents the names of variables defined in the first line.

Consider an example of assignment.

| A + 1 ⇒ A	
V | 4	5
S | 1.n	1.n

The above example represents notation A[4] + 1 to A[5] expression. The row 'V' represents subscripts and 'S' represents data types and 1.n is integer of 'n' bits.

Q26. Explain briefly about Pseudocodes.

Answer :

Pseudocode

Pseudocode term is used in a different sense from the actual meaning. The earlier languages are named as pseudocodes but they are actually not pseudocodes. Earlier in the later 1940's and 1950's computers had lack of supporting, software, slow, unreliable and very small memories. The programming was done in machine code because there were no languages like higher level and assembly language. This task was tedious and error prone. The other problem is usage of numericcodes to specify instructions. There were so many problems with machine languages and they led to the invention of assemblers and assembly language.

As most of the programming problems earlier time were numerical and thus require floating point arithmetic operations and indexing to allow arrays to be used conveniently. Because of these deficiencies, higher-level languages were developed.

Short Code

John Mauchly developed the first new language called Short code for the BINAC in 1949. It was later transferred to a UNIVAC I computer and it was considered as one of the primary means of programming such type of machines.

UNIVAC I's memory words ranges upto 72 bits that are grouped as 12 six-bit bytes. The short code contained coded version of mathematical terms that are to be evaluated. The codes were of byte-pair values and multiple equations can be coded in a word. The operation codes which are included are as follows,

01	–	06	abs value	1n	(n + 2)nd power
02	)	07	+	2n	(n + 2)nd root
03	=	08	pausc	4n	if < = n
04	/	09	(	58	print and tab

Consider an example,

a = b (c + a)

Here a, b, c are values and are named with byte codes such as,

$X_1 = X_2(X_3 + X_1)$

It is coded as $X_1 = X_2(X_3 + X_1)$

00 X_1 03 X_2 09 X_3 07 X_1 02

Initially, 00 should be used as padding to compensate the word. There is no short code for multiplication, it is simply represented by placing the two operands next to each other short code. It was implemented with pure interpreter rather than being translated into machine code known as automatic programming. This short code interpretation is 50 times slower than machine code.

Speed Coding

Speed coding system is developed by JohnBackus for the IBM 701. It is a type of interpretive system that extends the machine language to add floating point operations. It converts the 701 to a virtual three-address floating point calculator. It consists of pseudo instructions for nearly four arithmetic operations on floating point data, including the operations like square root, sine, arc tangent, exponents and logarithms. It also consists of a new facility of automatic increment of address registers. Only in UNIVAC 1107 computers of 1962., this facility was found with the help of such feature like matrix multiplication is possible in speed coding instructions. According to backup the program and problems can be programmed within few hours through speed coding where as it takes two weeks to program in machine code.

The UNIVAC "Compiling" System

Grace Hopper and his team developed a series of "Compiling" systems called A – 0, A – 1 and A – 2 at UNIVAC. It expanded a pseudocode into machine code subprograms as macros are expanded into assembly language. Pseudocode source is considered as primitive and an improvement over machine code for these compilers because it has made source programs shorter.

Q27. Describe in brief about the IBM 704 and Fortran.

Answer :

IBM 704

IBM 704 provides advancement in computing. It is developed in 1954 and even prompted for the development of Fortran. It has both the foresight and the resources useful for these developments.

History

The interpretive systems earlier in late 1940's to mid 1950's were slow due to lack of floating point hardware in the computers. Embedding float point operations in the software is a time consuming process. Since, much processor time was spent in the processing of floating point software, the inefficiency of interpretation and indexing simulation is not important and if the software had to perform floating point, the cost of interpretation was acceptable. Most of the programmers earlier preferred hand coded machine language efficiently rather than interpretive systems. IBM 704 had introduced new features like indexing and floating point instructions in hardware which led to the end of interpretive systems.

Fortran

Fortran is the first compiled high level language Alick E. Glennie developed Autocode compiler for the Manchestar Mark I computer at Fort Halstead in England. It came into use in september 1952. But according to John Backus, Glennie compiler is low level and machine oriented rather it may not be to considered as a compiler system. The Laning and Zeierler was first algebraic translation system which translates the expressions that are used to compute transcendental functions along with arrays. This system was implemented as prototype model in 1952 on MIT whirlwind computer and used in May 1953. This translator generates a subroutine call to code every formula in the program. This language is readable and provides machine instructions for branching. Rather than these works, fortran was the first accepted compiled high level language.

Design Process

In November 1954, John Backus and his group introduced a report named "The IBM mathematical FORMULA Tran slating system "FORTRAN". It contained description about the first version of Fortran 0. This document stated that the language would provide the efficiency of hand coded programs and ease of programming of interpretive pseudocode system. It also stated that Fortran removes errors that are present in coding and dubugging process. Due to this, there is less checking of syntax errors is in the first Fortran compiler.

Fortran was developed based on the following environments,

1. Computers were very slow and unreliable with small memories.
2. Computers were used for scientific computations.
3. Effective and efficient methods to program computers are not available.
4. As the cost of computers is higher than cost of programmers the speed of generated object code is considered as the major primary goal of first Fortran compilers.

FORTRAN-I

In 1955, Fortran 0 was changed while implementation and continued till the release of first compiler in 1957. Fortran-I includes input/output formatting, name of variables with length upto six characters, do loop statement, if selection statement, user defined subroutines, which cannot be compiled separately.

The control statements of Fortran-I depends on 704 instructions. It was not known whether the developers of 704 described about the design of control statement or designers of Fortran-I described about these instructions to 704 designers. In Fortran I, there are no datatyping statements. Variables which starts with I, J, K, L, M and N are of implicit integer type and the remaining are floating point.

Fortran development group claimed that the code produced by the compiler is half efficient than the code produced manually. After the development, Fortran achieved the goal with efficiency. The success Fortran is exposed in the survey conducted in Apirl 1958.

FORTRAN-II

In 1958, the Fortran-II compiler was developed. It has many new features and fixed many problems (or) errors in the Fortran-I compilation system. The important feature is that compilation of subroutines can be performed independently. Without this feature, if there are any modifications done to the program, then there should be recompilation of entire program. As fortran I does not possess this feature and even become unreliable, it had restrictions over the length of the programs (300 to 400) of the occurrence of machine failure. Addition of precompiled machine language subprogram versions had decreased the compilation process.

Fortrans IV, 77, 90 95, 2003 and 2008

Fortran IV was developed in between 1960 and 1962. It is used mostly and standardized as Fortran 66. Fortran IV is the enhancement of Fortran II. It included many features such as explicit type declarations for variables, a logical if construct and passing subprograms as parameters to other subprograms etc.

Fortran-77

Fortran 77 replaced Fortran IV and became the naive standard in 1978. Most of the features of fortran IV are present in fortran 77. It also includes extra features such as character string handling, logical loop control statements and If with an optional Else clause.

Fortran 90

Fortran 90 has many different features it includes dynamic arrays, records, pointers, a multiple selection statement, modules and the subprograms that can be called recursively. It had two syntactic changes which changed the structure of programs as well as the literature of the language. First by the format of code that needs the usage of specific character positions for some parts of statements is removed. The second was the name FORTRAN changed to Fortran i.e., only the first letter of keywords and identifiers would be uppercase apart from that all the letters will be in lower case.

Fortran 95

Fortran 95 had less changes such as a new iteration construct, for all, it was added that simplifies Fortran programs parallelizing.

Fortran 2003

It supports object-oriented programming, parameterized derived types, procedure pointers and interoperability.

Fortran 2008

Fortran 2008 is the latest version of Fortran. It support blocks to define local scopes, co-arrays and the DO CONCURRENT construct which specify loops without interdependencies.

Q28. Write about functional programming (LISP).

Answer :

LISP

LISP is the first functional programming language that provides feature for list processing. It is favoured in the area of artificial intelligence.

The Beginning of Artificial Intelligence and List Processing

The interest of AI has appeared in many areas such as in psychology and mathematics. In psychology, psychologists were interested in modelling human information storage, retrieval and basic brain process. Mathematicians were interested to build a smart process such as theorem proving. All these investigations generated same result. Any method must be developed to allow computation on symbolic data in linked lists, because computation was done on numeric data in arrays earlier.

List processing was developed by Allen Newell, J.C Shaw and Herbet Simon at the RAND corporation. It was first published in a paper which described the logic theorist one of the first AI programs and in a language in which it is implemented. The versions of IPL are IPL I, IPL II among which IPL I never published and IPL II was implemented on RAND Johnniac computer. Evolution of IPL was continued till 1960. The reasons for which the IPL languages prevent their widespread use of the language was implemented with an interpreter in which instructions of list processing were included. An other reason is it was implemented on the obscure Johnniac machine.

The list processing is considered as teasible and useful. IBM is exposed in AI in 1950's and decided to demonstrate theorem proving. Fortran project was still on process due to high cost Fortran I compiler. IBM decided to attach list processing in designing a new language rather than to Fortran. Due to this reason Fortran List Processing Language (FLPL) was developed and published as an extension to Fortran. FLPL was used to construct a theorem prover for plane geometry also implemented in the mechanism of theorem proving.

Design Process of LISP

The design process of LISP was started by JohnMC Carthy. In 1958, the took charge to investigate Symbolic computations. at IBM information research department. There is a need of requirements to develop such computations. On the study of pilot example problem, an algebraic expressions were chosen, a list of language requirements were discovered. The requirement are as follows,

1. Control flow methods of mathematical functions i.e., recursion and conditional expressions.
2. Dynamically allocated linked lists and some kind of implicit deallocation of abandoned lists.

But MC Carthy would not allow to place explicit deallocation statements in an algorithm for differentiation because FLPL does not provide support for recursion, conditional expressions, dynamic storage allocation, or implicit deallocation. So, Mc Carthy had a clear idea to develop new language.

Atlast Mc Carthy and Marvin Minsky started a project to introduce a software system for list processing. This was used to implement a program called Advice Taker. This application helped in the development process of LISP. The first version of LISP is considered as Purely functional language which is sometimes called "Pure LISP".

Q29. Discuss the Language Overview and Evaluation of LISP.

Answer : Model Paper-III, Q3(a)

Data Structures

LISP consists of two types of data structures. They are atoms and lists.

Atoms

Atoms can be either symbols, that are in identifier form or numeric type. The linked lists stores the symbolic information and are used in IPL-II. These data structures allow operations like insert and delete, which are required for list processing. And LISP programs need these operations.

Lists

Lists are represented with parenthesis to delimit the elements. Simple list elements are restricted by atoms. The following statement represents the form.

(A B C D)

Nested list structures are also represented with parentheses by delimiting the elements.

Example

The following statement represents the example of nested list.

(A (B C) D (E (F G)))

The above list consists of four elements. 'A' is the first atom, (B C) is the second sublist, 'D' is the third atom, (E (F G)) is the fourth sublist, and internally it has sublist (F G).

Internally, the lists are stored as single linked list with two points together representing a list element. The first pointer points to the atom which is symbol or numeric value or a pointer to a sublist. There is also a node for a sublist, in which first pointer points to the first node of sublist and the second pointer points to the next element in the list. A reference to the list can be made by maintaining a pointer to its first element.

Processes in Functional Programming

LISP is developed as functional programming language. In functional programming language, computation can be done by using functions in arguments. The assignment statements and variables are required in imperative languages programs, but they are not required in functional language programs.

The recursive function calls and loops specify the repetitive process which are mostly not required. These concepts make functional programming different from other imperative programming languages.

The horizontal representation of the nested list structure.

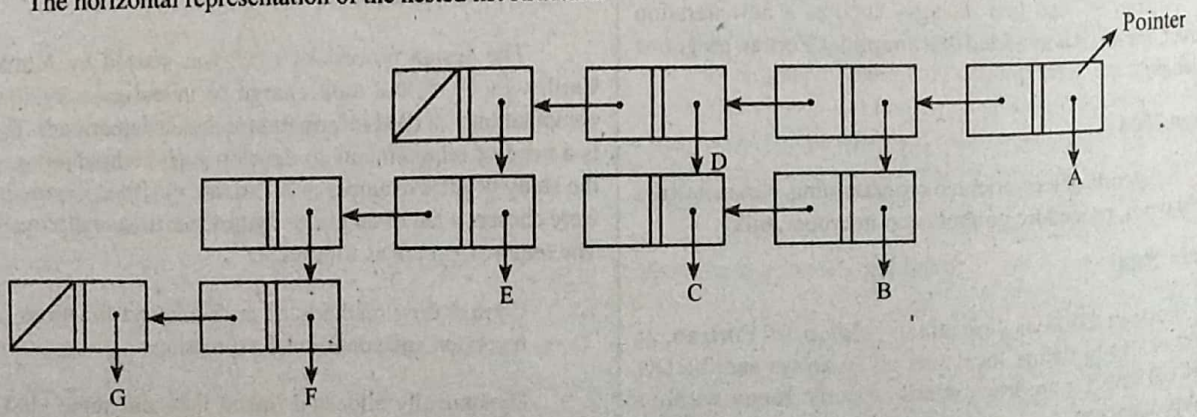


Figure: Nested List Structure

Syntax of LISP

LISP is different from imperative language since it is functional programming and its appearance is also different from other language like C++ or Java. In Java, syntax will be combination of English and algebra whereas in the lisp's syntax, program code and data posses the same form such as parenthesized lists.

Consider the list,

(A B C D)

When data is interpreted, it contains four elements in the list but in the code, it is represented as function name 'A' with three parameters B, C, and D.

Conclusion

LISP dominated the AI applications. LISP is considered as highly efficient and the features leading to it are removed. Due to all these reason, LISP is developed as functional programming and is considered as better way for software development than procedural programming using imperative languages.

LISP Program for Finding the Factorial of 'n'

The factorial of a given number 'n' in LISP can be determined using the 'DEFUN' keyword.

The function definition is as follows,

```
(DEFUN factorial (n)
   (if (= n  1)
      1
   (* n (factorial (–n  1)))))
```

The above factorial function returns '1' if the entered number 'n' is 1 else it makes a recursive call to the factorial function to compute the factorial of a given number.

Q30. Discuss about the two descendants of LISP and related languages.

Answer :

There are two kinds of LISP dialects that are commonly used. They are,

1. Scheme
2. COMMON LISP.

1. Scheme

The scheme language has emerged in the mid-1970's from MIT lab. It has the properties like small size, exclusively uses static scoping and treats functions as a first-class entities.

Since scheme functions are considered as first-class entities, they can be used as values of expressions and list elements assigned to variables, passed as parameters and returned as the value of function applications. All these capabilities were not available in the early versions of LISP. And also early versions of LISP even did not make use of static scoping.

Scheme is a small language having simple syntax and semantics. Because of these features it can be employed in educational applications such as, functional programming courses and the course of general introduction to programming.

2. COMMON LISP

A large number of LISP dialects have been introduced during the year 1970's and early 1980's. All of them have the problem of portability. To rectify this problem Graham created a LISP dialect called COMMON LISP in the year 1996. COMMON LISP has been designed by combining all the features of different LISP dialects of early 1980's including scheme.

However, the basis of COMMON LISP is pure LISP, therefore most of its syntax, primitive, functions and fundamental nature have been obtained from LISP. It is a large and complex language.

COMMON LISP supports both dynamic and static scoping thus providing flexibility and simplicity to the language. Each variable in COMMON LISP has static scope by default. But it can be changed to dynamic scope by declaring that variable as, $\frac{\text{Special}}{\text{B/I}}$.

COMMON LISP provides large number of data types and structures. Such as records, arrays, complex numbers and character strings. Beside these, it even provides packages that are used to modularize collection of functions and data for access control.

Related Languages

Meta language was developed by Robin Milner at the University of Edinburgh. It was used as a Meta language for a program verification system named logic for computable functions (LCF). It is considered as a functional language but it provides support for imperative programming also variable type and expression type can be determined at the compile time. It relates types to objects, but not to their names. It does not use parenthesized functional syntax which is derived with lambda expressions. The syntax is closely related to imperative languages like Java, C++.

Miranda was designed in 1980's by David Turner at University of kent in Canterbury, England. It depends on the ML, SASL and KRC. Haskell is a pure functional language. It does not have variables and assignment statements. It has a distinct feature called lazy evaluation it means expression evaluation is not done till its value is needed.

Caml and its dialect supports object-oriented programming. Ocaml is derived from ML and Haskell. Finally F# is based on Ocaml. It is a .NET programming language that supports both functional and procedural programming.

Q31. Discuss in brief about ALGOL 60.

Answer :

History of ALGOL 60

ALGOL 60 was developed for the purpose of scientific applications. Earlier in 1954, the laning zierlier algebraic system were undergone operation for over a year and published on fortran. Several other languages were developed and Fortran came into existence in 1957. Among the languages IT was developed by ALAN Perlis and MATHMATIC and UNICODE were developed for UNIVAC computers. In 1957 on May 10, scientific user groups such as SHARE AND USE requested ACM to create a committee to study and take an action to create computing machinery for independent scientific languages because of wide spread use of machine dependent languages.

Earlier in 1955, GAMM had created sub committee to develop a machine independent programming language and in 1957 to collaborate with ACM a invitation letter was sent by fritzBauer to them. After the formal agreement, the two groups decided to start a joint language design project.

Early Design Process

The first design meeting was held in zurich from May 27 to June 1, 1958 to discuss the goals of the new language.

The goals of the new language are given as follows,

1. The language syntax must be similar to standard mathematical notation and programs which are written must be easy to read.
2. The language should be used to describe algorithms in printed publications.
3. The new language programs must be converted into the language of machine mechanically.

Every goal of new language has a purpose like first goal to indicate the new language that is used for scientific programming, the second goal is related to computing business and the last one is required for any programming language.

Overview of ALGOL 58

ALGOL 58 was earlier known as International Algorithmic Languages (IAL) which was developed at the Zurich meeting. The committee decided to change it as ALGOL which represents as ALGOrithmic Language but as it is not reflecting international scope of committee, it was later changed to ALGOL 58.

ALGOL 58 implementation is based on Fortran. Many features, concepts and new constructs were added and simplified many of Fortran's features. It introduced the concept of data type, concept of compound statements. The features of Fortran which were simplified are as follows,

1. Identifiers were allowed to have any size, Unlike Fortran I.
2. Array dimensions are allowed to any numbers with no limitation. Fortran I's limitation is not more than '3'.
3. The programmer could specify the lower array bound.
4. Nested selection statements were allowed.

Earlier, the assignment statement in plankalkul are in the below form

expression ⇒ value

Later due to arguments about character set limitations, the greater than symbol is changed to colon i.e.,

variable: = expression

And the ALGOL 58 developed the assignment operator in another way.

ALGOL 60 Design Process

In 1959, there was a debate regarding ALGOL 58. Several modifications and additions were represented in the European ALGOL Bulletin and in communications of the ACM. And the other event in the 1959 was zurich committee presented work regarding Backus BNF notation which describes the syntax of programming languages to the International conference on information processing.

In January 1960, the meeting was held regarding the suggestions to be debated in ALGOL language. Peter Naur of Denmark contributed in the development of ALGOL though he is not a member of zurich group. He created and released ALGOL Bulletin and also made research on Backus's paper which describes BNF and stated that BNF should be used to describe the results of the 1960 meeting. He made changes for BNF and suggested a new language in BNF.

Overview of ALGOL 60

The meeting was held for six days. The following are the new developments made to ALGOL 58.

1. The concept of block structure is introduced which allows the user to divide the programs and introduce new data environments.
2. Two parameter passing techniques are allowed. They are pass by value and pass by name.
3. Procedures should be recursive. The ALGOL 58 was not sure about this issue. As this recursion concept is new to the imperative languages, but already LISP provided the concept of recursive functions in 1959.
4. Stack dynamic arrays can be implemented. In which subscript range or ranges can be specified by variables so that array size is allocated at the time storage.

The ALGOL 60 report was released in May 1960. There is a confusion regarding the language description. So the third meeting was held in Rome 1962 to discuss about the problems. Addition of features to the language were not discussed but only the problems were discussed. The output generated after this meeting is released as "Revised Report on the Algorithmic Language ALGOL 60".

Evaluation

ALGOL60 is partially succeeded and in some other ways it considered as dismal failure. The succeeded part is that it is a language which provides the immediate formal means to communicate with algorithms in computer literature. Most of programming languages are descendants of ALGOL 60 such as PL/I, SIMULA 67, ALGOL 68.

Many characteristics are included in the design effort of ALGOL 58/ALGOL 60. They are,

1. ALGOL 60 is first programming language whose design was attempted by a international group.
2. Its syntax was described formally.
3. Its language was designed as to be machine independent.
4. It uses BNF form
5. Finally, Machine architecture is affected by the structure of ALGOL 60.

Here the failure part is that ALGOL 60 was never used widely in the united states, also in Europe. There are many reasons for which it was not used are,

1. It is considered as failure because the language is not efficient to implement and is difficult to understand.
2. There is lack of input and output statements.
3. BNF was also a factor for its failure as it was strange and complex earlier.

The advanced features of Fortran and it was not supported by IBM were considered as two major reasons for the failure of widespread use of ALGOL 60.

Example

The Example program of ALGOL 60

```
begin
integer length, i, j, k, res;
i: = 0;
res: = 0;
read int (length);
if(length > 0) ∧ (lenght < 50) then
begin
for j: = 1 step 1 until length do
begin
read int (int arr[i]);
j: = j + int arr[i];
end;
k: = j/length;
for i: = 1 step 1  until length do
if int arr[i] > k
then res: = res + 1;
print string("values > k is: ");
print int(res)
end
else
Print string ("length of list is not allowed");
end
```

Q32. Explain about computerizing business records (COBOL).

Answer :

COBOL

COBOL stands for common business-oriented language. It is mostly used language than any other programming languages. This language is developed for the purpose of business applications. COBOL had little impact on the design of subsequent languages except for PL/I. Reason for this is some of then have attempted to design a language for business purpose and other reason is in small business. There has been lot of growth in business computing for past 30 years.

Historical Background of COBOL

The evolution of COBOL is similar to ALGOL 60, as it was developed by a committee of people met for a short period of time. Earlier in the 1959 when Fortran was being designed, the business computation and scientific computation were similar. In 1957, FLOW MATIC language was implemented for business application which is owned by manufacturer and then UNIVAC also developed for company's computers purpose. The other language AIMACO, which is similar but exists with a small variation of FLOW-MATIC is used by U.S. Air force. COMTRAN is a language developed by IBM for business applications but it was not implemented and there is plan of several other projects for language design.

FLOW MATIC

FLOW MATIC is the primary inheritor of COBOL. In December 1953, Grace Hopper wrote a proposal at Remington Rand UNIC regarding (or) suggesting that "mathematical programs should be written in mathematical notation, programs for data processing should be written in English statements". In 1953, non programmers were not convinced that a computer could understand Englishwords. Later a prototype model was implemented by UNIVAC management, which involved compiling and running small program by using English words, french keywords and German keywords.

Design Process of COBOL

Department of Defence had sponsored a meeting to discuss on a subject of common language for business applications on May 28 and 29, 1959 at the pentagon. The general opinion of the group was the language called CBL (Common Business Languages). It had the following characteristics.

1. English must be used in the language and few of them argued for mathematical notation.
2. The language should be user friendly.

   It was assumed that use of English words makes the managers to read programs.

The short range committee was formed to make a quick study of existing languages. So that a universal language can be developed and used widely.

Earlier it was decided as statements of language should be divided into two categories i.e., data description and executable operations and also statements of these categories should present in two different parts of the program. The short committee range was also proposed to include subscripts, but the other members of committee did not accept because, it would be too complex in data processing and thought it will sync with mathematical notation. Also many arguments happened regarding arithmetic expressions to be included or not. At last, in December 1959, short range committee made a final report to describe a language and it is named as COBOL 60.

The language specifications for COBOL 60 were defined as 'initial' and it was published by the Government printing office in April 1960. Earlier versions were published in 1961 and 1962. ANSI group provided the standards to the languages in 1968. The other three revisions were also provided with standards in 1974, 1985 and 2202.

Conclusion

The COBOL language has introduced several novel concepts from for which few appeared in other languages too. For instance the DEFINE verb of COBOL 60 is first high-level language construct for macros. The hierarchical data structures of plankalkul were first implemented in COBOL. The COBOL allows both long names (up to 30 characters) and word-connector characters (hyphens).

The data division of COBOL's design is a strong part of it and procedure division is a weak part. In the data division, every variable is defined including the number of decimal digits and decimal point location. File records are described in this level are sent it as output to a printer, and this makes COBOL ideal for printing accounting reports. Procedure division is relatively weak because it has lack of functions. COBOL versions, which were introduced before 1974 standard did not allow subprograms with parameters.

COBOL is the first programming language which was mandated by the Department of Defense. It is not specifically designed for the purpose of DoD, but survived with the mandate given by DoD. Earlier the compilers were poorly designed, this made the language costly to be used. The compilers were designed more efficiently and computers become faster, cheaper and were designed with larger memories. All these factors forced to use COBOL inside and outside the DoD and it also led to electronic mechanization of accounting.

Example

The following is an example of COBOL

```
IDENTIFICATION DIVISION
PROGRAM-ID FILES
ENVIRONMENT DIVISION
INPUT-OUTPUT SECTION
FILE-CONTROL
SELECT TRANSACTIONS ASSIGN TO USER
DATA DIVISION
FILE SECTION
FD TRANSACTIONS
01 TRANSACTION-STRUCT
02 UID NO PIC 9(5)
02 FILE DESC PIC × (20)
02 FILE DETAILS
03 BALANCE PIC 9(6) ∨ 9(2)
```

Q33. Write about the beginnings of time sharing (BASIC).

Answer :

BASIC

BASIC is a programming language, which is very popular on microcomputers in the late 1970's and early 1980's. This language was easy for beginners and those who were not science oriented. With the evolution of other programming languages and increased capabilities of microcomputers, the usage of BASIC programming language was decreased. Later, it was used again with the presence of visual Basic in the early 1990's.

Design Process of BASIC

The two mathematicians named John kemeny and Thomas kurtz developed BASIC at dartmouth college in New Hampshire. They developed compiler for a variety of Fortran and ALGOL 60. Dartmouth is an institution of liberal arts, as only 25% of students are of science and engineering. It was decided to design a new language, which would use terminals as the computer access method for liberal arts students. The system objectives are as follows,

1. Learning and using must be easy for nonscience students.
2. It must be "pleasant and friendly".
3. For homework, it must provide fast turnaround.
4. It should provide access freely and privately.
5. User time should be considered more than computer time.

Language Overview

BASIC's original version was considered as very small, oddly enough and not interactive. As there was no way for an executing program to obtain user input data, programs were typed in, compiled and run in a kind of batch-oriented way. There are only 14 different types of statements and a single data type floating point. These types were defined as "numbers". Finally, it is considered as limited language but easy to learn.

Evaluation

The most important factor of the original BASIC was broadly used by language and used through terminals linked to a remote computer. Earlier most of the programs were entered through punched cards or paper tape into computers. At that time, terminals came into existence.

The design of the BASIC mostly came from Fortran, with the syntax of ALGOL 60 having some minor influence. It evolved in number of ways but no standards are available. A minimal BASIC was issued by American National Standard Institute (ANSI) but this was only the bare minimum of language features. And also, the original BASIC was very similar to minimal BASIC.

Digital Equipment corporation used an elaborated version of BASIC known as BASIC-PLUS to write major portions of their largest operating system for the PDP-II minicomputers, RSTS in the 1970's.

BASIC criticized many factors such as,

1. Poor structure of programs.
2. Evaluation criteria specifically readability, reliability were considered very poor.

Earlier versions of the language were not used for serious programs. For such tasks, later versions are better suited.

BASICS emergence in the 1990's was driven by visual BASIC (VB) performance. VB was broadly used because it provides on easy way to build graphical user interface. Visual Basic .Net or VB .Net is one of the Microsoft's .NET languages. The major difference between VB and VB.Net is that VB .NET fully supports object oriented programming.

Example Program

```
DIM int list(50)
output = 0
add = 0
INPUT length
IF length > 0 AND length < 100 THEN
FOR ctr = 1 TO length
INPUT int list (ctr)
add = add + int list(ctr)
NEXT ctr
avg = add/length
FOR ctr = 1 To length
IF int list(ctr) > avg
THEN output = output + 1
NEXT ctr
PRINT "The values that are greater than average are:",
output
ELSE
PRINT "length of the list is not valid"
END IF
END
```

Q34. Discuss about PL/I in detail.

Answer :

PL/I

PL/I is the first programming language designed to be used in various application areas. The other languages were designed only for specific application area, such as science, artificial intelligence or business.

History

PL/I was designed as an IBM product like Fortran. The clients of computers industry in the mid of 1960's were settled into two separate parts such as scientific users and business users. From the IBM perspective, either the large scale 7090 or the small scale 1620. IBM computers could be used by scientific users. These users made extensive use of floating-point data and arrays. These users maintain their own user group called SHARE and maintain little communication with others who works on business applications.

Business users for business applications have used the large 7080 or the small 1401 IBM computers. These users require decimal and character string data types along with efficient input and output facilities.

Business users also had their own user group named as GUIDE and often maintained contact with scientific users.

The different computer user groups moved towards each other in ways to cause issues. Scientific users started file data processing which needed efficient input and output facilities. The Business users started using regression analysis for management of information system which needs floating-point data and arrays.

Due to these perceptions, a decision was made to introduce single universal computer for both business and scientific applications. Because of this, the IBM system 360 line of computers came into existence. Along with this, a new programming language is designed for both business and scientific applications to replace Fortran, COBOL, lisp and the system applications of assembly language.

Design Process

The design process of the language was started by IBM and SHARE, which formed an Advanced language development committee of SHARE Fortran project. Its sub-committee called 3 × 3 committee. It is named because there are 3 members from IBM and three members from SHARE.

The first version PL/I, is named as Fortran VI which was supposed to be completed by December but it was extended to January and then to late February of 1964.

Earlier, the decision made that the new language would be an extension of Fortran IV, but later it was changed for Fortran VI. Until 1965, the language was known as NPL (New Programming Language). In 1964, a complete description was published by compiler group at IBN hursley laboratory in England which was further proceed to implement and finally, in 1965 the name was changed to PL/I to avoid confusion.

Language Overview

PL/I includes the collection of new constructs which were considered as the best parts of other languages like ALGOL 60, FORTRAN IV and COBOL 60.

PL/I was the first programming which provides the following facilities,

1. Programs are allowed to execute subprograms simultaneously, but it was developed in PL/I poorly.
2. It is possible to detect and handle 23 different types of exceptions at a time.
3. Subprogram can be used recursively, but the efficiency could be disabled allowing more linkage for non recursive subprograms.
4. Pointers were treated as a datatype.
5. Referencing can be done in cross sections of arrays. For example, the third row of matrix can be referenced as if it was a single-dimensional array.

Evaluation

Evaluation of PL/I is done by recognizing the ambitiousness of the design effort. PL/I was combined with many constructs, which were useful and implemented. But there was an issue like how a programmer could understand and to use these constructs and features. PL/I is complex because of its large size, poorly designed constructs such as pointers, exception handling and concurrency.

As per the usage, PL/I is succeeded partially. In 1970's, it was used in both business and scientific applications, as an instructional vehicle in colleges and also in several subset forms such as PL/C (comell, 1977) and PL/CS (conway and constable 1976).

Example

Example of PL/I program

```
INPUT:
OUTPUT: A PROGRAM TO OUTPUT HELLO WORLD
HELLO: PROCEDURE1 OPTIONS (MAN);
FLAG1 = 0;
LOOP1: DO WHILE (FLAG1 = 0);
/*PRINT THE RESULT*/
PUTSKIP DATA C ('HELLO WORLD!!');
END LOOP;
END HELLO;
```

Q35. Write short notes on,

(a) APL

(b) SNOBOL.

Answer :

(a) Origin and Characteristics of APL

APL was created by kenneth E. Iverson at IBM around 1960. It was not designed to be implemented programming language originally, but designed a vehicle to describe computer architecture. APL was first described in a book from which it got its name 'A programming language'. And later in the mid-1960's, IBM developed the first implementation of APL.

APL consists of large number of useful operators specified with large number of symbols, which create a problem for implementors. APL was initially used through IBM printing terminals. And these terminals posses special print balls, which provide the language's odd character set. The reason that APL has multiple operators is because, it provides a large number of unit operations on arrays. For example, a single operator is used to transpose any matrix.

Huge set of operators makes high expressivity, but make the APL programs complex to be read. Thus, people consider APL as a best language which is used for "throw-away" programming. Program can be written quickly, these programs must be discorded after usage because these are hard to maintain.

Nearly 50 years, APL has been around is still being used today, but not widely. It has not changed much in its lifetime.

(b) Origin and Characteristics of SNOBOL

SNOBOL was designed by three people in the early 1960's at Bell laboratories D.J. Farber, R.E. Griswold and I.P. Polonsky (1964, Farber et al.). This was mainly designed for text processing. SNOBOL's heart consists of set of powerful string pattern matching operations. Earlier, SNOBOL was used to write text editors. It has dynamic nature, which makes it slower than alternative languages and it is not used for such programs any more. But SNOBOL is still considered as live and supportive language, used in several different application for a variety of text processing tasks.

Q36. Explain about,

(a) The Beginnings of Data Abstraction (SIMULA 67)

(b) Orthogonal design (ALGOL 68).

Answer :

(a) The Beginnings of Data Abstraction (SIMULA 67)

Design Process

SIMULA 67 was developed in between 1962 and 1944 by two Norwegians, named Kristen Nyagaard and Ole Johan Dahl at Norwegian Computing Center (NCC) in Oslo. It was developed for system simulation purpose in computers and implemented initially in the 1964 on a UNIVAC 1107 computer. After the implementation of SIMULA1, Nygaard and Dahl started working together to extend the language by adding new features and changing some existing constructs to make the language useful for general purpose applications. This extension resulted in SIMULA 67, was first exposed in March 1967. Some of the features of SIMULA 67 are also present in SIMULA 1.

Language Overview

SIMULA 67 is an extension of ALGOL 60 with features such as block structure and control statements. The main drawback of ALGOL 60 was the design of its subprograms for simulation application. Simulation involves (or) needs the subprograms which can restart at place where it was stopped earlier. This type of control in subprograms is defined as co-routines because the caller and so-called subprograms maintain the relationship with each other equally.

The class construct was developed to support coroutines in SIMULA 67. With this development, the concept of data abstraction came into existence and it provides the basis for object oriented programming. Till 1972 when Hoare recognized the connection concept of data abstraction was not developed and attributed to class construct.

(b) Orthogonal Design (ALGOL 68)

ALGOL 68 is an imperative programming language which is considered as source for new ideas to design the language.

Design Process

The development of ALGOL 68 resulted almost after the design of ALGOL. The ALGOL 68 is different from its earlier versions of languages. The main design criteria of ALGOL is Orthogonality. With the use of orthogonality, there are many innovative features of ALGOL 68.

Language Overview

ALGOL 68 includes user defined data types, that lead to Orthogonality. The other earlier languages like Fortran, included only basic data structures and PL/I included many data structures. Which made it complex to learn and implement. The purpose of ALGOL 68 is to provide data structures that support primitive types and structures and even enable the user to combine primitives into a huge set of different structures. User defined data types are valuable because they enable the user to design data abstractions to fix the problems.

ALGOL 68 introduced the concept of dynamic arrays named as implicit heap-dynamic. In ALGOL 68, the dynamic arrays are called flex arrays in which subscript specification is not required.

Conclusion

ALGOL 68 is a programming language which includes many significant features which were not included in previous languages. The drawback of ALGOL 60 also repeated in ALGOL 68. The language was described by using unknown metalanguage but it is concise and elegant. To read a language describing document, a meta language called van wijngaarden grammars should be learnt (which is complex than BNF). For instance, keywords are called indicants, substring extraction is called trimming and the process of executing a subprogram is called a coercion of deproceduring. There is a difference between the design of PL/I and ALGOL 68. ALGOL 68 achieved writability with the use of orthogonality and PL/I achieved writability by including a number of fixed constructs. ALGOL 68 extended the ALGOL 60 features and PL/I combined the features of various languages to provide a unified tool for wide range of problems where as ALGOL provide solutions for scientific applications.

Q37. Write about,

(a) Programming based on logic (PROLOG)

(b) Object oriented programming (SMALLTALK).

Answer :

(a) Programming Based on Logic (Prolog)

Logic programming is the type of programming which uses the formal logic as the notation to create communication between computational process and a computer. But Now, in the current logic programming languages, predicate calculus is used as notation.

Logic programming languages were considered as non-procedural. In this type of languages, programs do not mention how the result is calculated but it provides the characteristics of result. To provide this, logic programming needs a relevant information and inferencing technique to compute desired results as a means to supply the computer. Here the predicate calculus provides communication to the computer, proof method, named resolution and inferencing technique.

Design Process

Alain colmerauer, phillippee Roussel, and Robert Kowalski of the Department of Artificial intelligence group in the early 1970's developed the basic design of prolog. The main components of prolog are methods to specify predicate calculus propositions and to implement a restricted form of resolution. The first prolog interpreter was developed at marseille in 1972. The name prolog is derived from the Programming Logic.

Language Overview

Prolog programs consists of a set of statements. There are some kind of statements, which are complex. Prolog can be used as a type of intelligent database. It consists of two types of statements. They are facts and rules.

Examples of facts are as follows,

1. employee(X, Sia)
2. Student(Y, JNTU).

The above examples represents that X is the employee of Sia and Y is the student of JNTU.

Example of rule statement is,

grandparent (A, C): parent(A, B), parent(B, C)

The above example states that,

It can be deduced that A is a grand parent of C, it is said as true if A is the parent of B and B is the parent of C satisfies specific values for A, B, C variables.

Prolog system uses resolution process to determine the truth of the statements presented by a query or goal. It represents 'true' if it satisfies the goal or else it displays 'false'.

Evaluation

Earlier in 1980's, it was believed that logic programming had reduced complexity created developed by imperative languages and problems produced by reliable software.

Logic programming was not used widely because of two major reasons. They are as follows,

1. Logic language programs are considered as inefficient for non imperative languages compared to imperative languages.
2. It is considered as an effective approach only in few application areas such as AI and kinds of DBMS.

Prolog ++ supports object-oriented programming, which is a dialect of prolog.

(b) Object-oriented Programming (Smalltalk)

Smalltalk is considered as the first language that completely support object oriented language.

Design Process

The principles that led to smalltalk's development came into existence in Alan kay's ph.D work at the university of Utah in late 1960. As kay had predicted the future availability of desktop computers. Earlier in the mid of 1970's, the first micro computer systems are not available in market but remotely connected to machines, these systems executed a million instructions per second or more and contains large memory i.e., in megabytes. Those type of machines were available broadly in the early 1980's kay assumed that desktop computers should possess very powerful human-interfacing capabilities to be used by non programmers. In 1960's batch oriented systems were used by professional programmers and scientists. But for the non-programmers, computer should be highly interactive and possess sophisticated graphics in its user interface.

Kay has released a system called Dynabook which is based on the part of flex language, and the flex is based on SIMULA 67. Dynabook is the general information processor that uses the paradigm of the typical desk. The display of it represents many sheets of paper with the help of screen windows. The user interacts with display by using keystrokes and touch pad. kay received ph.d after the preliminary design of Dynabook, achieved his goal by seeing machine constructed system.

As Kay presented his ideas of Dynabook at Xerox Palo Alto research center, simultaneously a Learning Research group was created at Xerox. The group's main aim was to design a language which supports kay's programming paradigm and implements it on the personal computer "Interim" Dynabook, was result of this design and it consists of Xerox Alto workstation and smalltalk-72 software. A research tool is also developed and many number of research projects were conducted like explaining programming to children. The further development of this languages was ended with smalltalk-80.

Language Overview

The smalltalk consists of objects, integer constants and large complex software systems. The computing process in smalltalk is done by a technique i.e., sender sends a message to an object for method invoking and reply will also be in the form of an object that returns the requested information or sends a notification that the requested processing has completed. The main difference between a message and a subprogram call is, A message is sent to a method defined for the data object. Then, it is executed and changed the data of an object and a subprogram call which is a message to the subprogram code. The data processed by the subprogram is sent in the form of a parameter.

In smalltalk, object abstraction is considered as classes, which are similar to SIMULA 67 classes. Instances of classes are called objects of program.

Smalltalk syntax is different from other programming languages, as it uses messages more than arithmetic, logic expressions and conventional control statements.

Conclusion

Smalltalk has contributed in promoting the two important aspects of the computing such as graphical user interface and object oriented programming. The windowing system was developed from smalltalk, which is used for the method of user interfaces to software system. Software design methodologies and object oriented programming languages were fully developed and used in smalltalk. Some of the ideas of object-oriented was also taken from SIMULA 67. At last, smalltalk has created great impact on the computing world.

Example

Example program for smalltalk is as follows,

Class name	Rectangle
Superclass	obj
instance variable names	ourTool

```
sides
length
new
super new get Tool
get Tool
our Tool ← Tool new default : 4
draw
sides times Repeat : [ourTool go:length turn : 360 //sides]
ll : l
Length ← l
sides : number
sides ← number.
```

Q38. Describe in detail about Ada.

Answer :

History of Ada

Ada was developed for the purpose of Department of Defense. In DoD, most of the computer applications were embedded systems. They are the systems in which computer hardware is incorporated to control or to provide services. Due to the increase in complexity of systems, the cost of software was increased. Many programming languages were used for DoD projects, but not standardized by DoD. Because of rapid increase of languages, application software was rarely reused. The software development tools were not developed. There are huge number of languages, but none of them were suitable for embedded systems. For the purpose of embedded system, Army, Navy and Airforce came forward to develop a single high level language in embedded systems in 1974.

Design Process of Ada

The design process was started in 1975, the director of defense research and engineering created a group called HOLWG (High Order Language Working Group). Earlier it was headed by Willian Whitaker of the Air force. The group had representatives from all the military services and liaisons with Great Britain, France and West Germany.

The primary things that should be done by the group are,

(i) To identify the requirements of new high level language for new DoD.

(ii) Analyzing the existing languages to check that they are viable candidate or not.

(iii) Suggest a minimal set of programming language to be adopted (or) implemented.

In April 1975, HOLWG released the document for a new language, which was later distributed to military branches, federal agencies and university representatives and for interested parties in Europe.

After the passing of strawman document to many DoD, finally the Cii Honeywell/Bull design proposal was accepted from the four finalists. The design team of Cil Honeywell/Bull is led by Jean khbiah, which is the only foreign competitor among the four finalists. Jackcooper proposed the name for new language as Ada resembling the Augusta Ada Byron, who is a mathematician, and world's first programmer. She worked with Charles Babbages on the mechanical computers to write several numerical processes. As the design of Ada was released by ACM in its SIGPLAN Notices and readership was given to 10, 000 people. Evaluation and test conference was conducted in Boston in October 1979 with representatives of more than 100 organizations. The Evaluation reports suggested few modifications. Based on these reports the requirements of the next version was released as the stoneman document in feb 1980.

The revised version for the language design is released in July 1980 and accepted as MIL STD 1815, the number 1815 was taken as the birth year of Augusta Ada. In 1982, another version for Ada language Reference manual was released in July 1982 and in 1983 Ada was standardised by ANSI.

Language Overview

The major contributions of the Ada language are as follows,

1. In Ada language, packages allows to encapsulate data objects, data type specification and procedures which again provide the usage of data abstraction in program design.
2. It includes many facilities for exception handling, it provides control to the user when the variety of exceptions occurred.
3. In Ada, program units can be generic.
4. It also provides concurrent execution of special program units, name tasks, with the help of rendezvous method (which is a method of inter task communication and synchronization).

Conclusion

The important aspects of Ada language design are as follows,

1. Participation is not restricted since the design was competitive.
2. The language Ada consists of the software engineering concepts and language design of late 1970's, these features are considered as valuable.
3. The development of a Ada compiler was a difficult task and no one estimated it. After the completion of language design, the usage of Ada compilers has began.

The disadvantage of Ada is that, it is considered as large and complex, and it was told by Hoare that it cannot be used for any application where there is high reliability. But on the other side many of them said that it is a ideal language design at that time.

Example

The example program for Ada

```
-- Input: Takes two matrices as input
Output: Print product of two matrices
with Ada.Text_Io; Ada.Numeric.Real arrays
use Ada.Text_Io; Ada.Numeric.Real arrays
Procedure Matrix_Mul is
Procedure Put (X:Real_Matrix) is
type Fixed is delta 0.01 range – 100.0...100.0;
begin
for I in X Range(1) loop
for J in X Range(2) loop
Put (Fixed 'Image (Fixed (X (I, J))));
end loop
New_line
end..loop
end put;
A : constant Real_matrix: =
    ((1  1  1  1)
    (2  4  8  16)
    ( 3  6  9  12)
    (4  8  12 16)
    );
B : Constant Real_matrix: =
    ((1  1  1  1)
    (1  1  1  1)
    ( 1  0  0  1)
    (1  2  3  4)
    );
begin
put(A * B)
end Matrix_Mul;
```

Ada 95 and Ada 2005

Ada 95 is a standard language which is described in ARM type derivation of Ada 83 that is extended in Ada 95 to allow inheritance and to add new components to a class derived from a base class. Dynamic binding of a subprogram calls to subprogram definitions is achieved by subprogram dispatching. It is based on tag values of derived types and it provides polymorphism.

The rendezvous method provides means of sharing data among concurrent processes which is inefficient. A new task is implemented for access control of shared data. This data is moved in a syntactic structure which controls the data access by rendezvous method or by subprogram call.

The usage of Ada 95 has decreased in military software, DoD has stopped using it and it became less popular. There are also many factors which made its less popular.

Ada 2005, had new features such as interfaces, scheduling algorithms control and synchronized interfaces.

Finally, Ada was used mostly in commercial and defense avionics, air traffic control and rail transportation and in many other areas.

Q39. Explain the combination of imperative and object oriented features (C++).

Answer :

C++ is a programming language, which is evolved from C with few modifications, improved features and add constructs to support object-oriented programming.

Design Process of C++

The evolution of C++ from C is started in 1980 by Bjarne stroustrup at Bell Laboratories. Modifications made in 'C' are addition of function parameter type checking, conversion, classes which are related to SIMULA 67 and smalltalk, derived class, constructors and destructor methods and friend classes. In 1981, there are some other modifications such as inline functions, default parameters and assignment operator overloading with these modifications 'C' is considered as new language named as C with classes.

There are some goals to be considered in C with classes.

1. It should provide organized programs as they are in SIMULA 67. Such as with classes and inheritance.
2. It should not allow performance penalty with relative to C.
3. It should be used for every application where C can also be used so that removal of C features is not required.

In 1984, it is again extended with new features such as virtual methods. These methods provide dynamic binding of method calls to method definitions, method name and operator overloading. The language with these extended features is called C++.

In 1985, C front is a system, which converts C++ programs to C programs. The combined version of C front and C++ implementation known as Release 1.0.

In between 1985 and 1989, C++ evolved and next version called Release 2.0 came into existence. The features included in 2.0 are added support for multiple inheritance and abstract classes.

The version 3.0 came into existence between 1989 and 1990. It has template which provides parameter type and exception handling.

Microsoft has released .NET computing platform which has a new version of C++ called Managed C++ or MC++. MC++ is an extension of C++ and it provides accessibility for the .NET framework. The new version includes properties, delegates, interfaces and a reference type for garbage collected objects.

Overview of C++

C++ has following features.

1. C++ consists of functions and methods, it supports both procedural and object oriented programming.
2. It supports operator overloading, method overloading.
3. In C++ virtual methods provide dynamic binding and define type-dependent operations with the help of overloaded methods in the inherited classes.
4. It supports multiple inheritance and exception handling.
5. Methods and classes can be implemented with templates to have many types of parameters.

Conclusion

C++ is used widely because of several reasons such as, there is availability of efficient compilers which costs less. It is compatible with 'C' language and program code is related to C languages. It makes case of use for the users to learn C++. It was the only language which supported object oriented programming and used widely for the large commercial software projects.

But, C++ also has drawbacks such as it is considered as very large and complex language. And it possess drawbacks similar to the PL/I. It is said to be unsafe language than Ada and Java because of insecurities taken by C.

There are some related languages for C++ such as objective C, Delphi and Go.

Objective C

Objective 'C' was developed by Brad cox and Tom Love in the early 1980's. It consists of classes of C plus plus and message passing of Smalltalk. It uses smalltalk syntax to support object oriented programming. Objective 'C' is used to implement the Next computer system software, which was found by Steve Jobs. But after the failure NeXT project, Apple used objective 'C' for writing MAC OS X. It is used language for the iphone software.

Objective C has one of smalltalk characteristic i.e., dynamic binding of messages to objects.

The version 2.0 named objective C2.0 is released by Apple in 2006 with added features like garbage collection and a syntax to declare properties. But iphone run time system does not support garbage collection. Objective C is considered as superset of C, so the insecurities which are included in C language are also included in objective C.

Delphi

Delphi is developed by Anders hejlsberg, who already developed Turbo pascal system and C# by inserting object oriented support in an existing imperative languages. It is similar to C++ and objective C. Delphi is a derived language from the predecessor languages. It is considered as safe and less complex than C++, since it does not allow user defined operator overloading, generic subprograms and parameterized classes etc.

Delphi allows to create GUI (Graphical User Interface) interfaces to applications and it provides GUI to the user.

Go

The Go language was developed by Rob pike, Ken Thompson and Robert Griesemer at Google. Initially, the design of language started in 2007 and its first implementation was released in 2009.

The Go language was developed to increase the compilation speed of C++ large programs. The first compiler for Go is fast and it took few constructs and syntax from C. The new features of Go are as follows,

1. Syntax of data declaration are opposite to other C base languages.
2. The type name is preceded by the variables.
3. Variable declaration can be type inferenced if it is initiated.
4. Multiple values are returned by functions.

The other features are it does not support traditional object oriented programming and it includes control statements. They are similar to the other C-based languages, goto statement, pointers, associative arrays, interfaces and supports concurrency with the help of coroutines.

Q40. Explain in detail about imperative based object oriented language (Java).

Answer :

Design Process

Java language was introduced by James Gosling, Patrick Naughton, Chris Wath, Mike Sheridan and EdFrank at Sun Microsystems. Java was invented in 1991. Initially, Java language was referred as 'Oak' and later renamed as Java in 1995.

The primary motive of Java language was to prepare a software that can be embedded in different consumer applications such as microwave ovens and remote controls. However, in early days, computer languages were designed for a particular type of CPU and compiled in machine code. Consider that, C++ language executes program in a full compiler which is selected for that particular CPU. The reason behind this compilation is, C++ programs are compiled into machine instructions which inturn executed by CPU. However, the set of machine instructions vary from one CPU to the other. In addition to this, the design of compilers is expensive and time consuming.

Inorder to overcome the above problems, Gosling and the others started working on portable and cross-platform language which results a software that can be executed on different CPU's under various environments. This leads to the invention of Java.

At the initial stages of Java, an important factor, which would play a vital role in the future of Java is World Wide Web (WWW). However, WWW has not taken its shape during the implementation of Java. The emergence of WWW made the way for Java to forefront the design of computer language. This is due to presence of several types of computers with different types of CPU's and operating systems in the Internet.

By the end of 1993, the focus of Java was shifted from consumer electronics to Internet programming due to the occurrence of portability problems during the creation of code for the Internet.

Java can be inherited from the syntax of C and object oriented concepts are inherited from C++. Therefore, it is easy to learn Java when one is familiar with C or C++. Eventhough, Java is adapted from C++, it cannot be an enhanced version of C++. Simply, Java was implemented for Internet programming using C++ concepts.

Language Overview

Features of Java

1. Java is smaller, simpler and more reliable.
2. Java contains both classes and primitive types.
3. In Java, arrays are instances of predefined class.
4. Java does not support pointers, instead references are used to point class instances. Some times references behave like ordinary scalar variables, because they implicitly dereference whenever required.
5. Java consists of primitive boolean type named Boolean, which are used in control expressions of its control structures.
6. In Java subprograms are known as methods and are defined in classes and methods are called through stand alone subprograms which are not supported in Java.
7. Java only supports object-oriented programming.
8. The difference between C++ and Java is, C++ supports multiple inheritance in its class definitions but Java supports single inheritance of classes. However, some of benefits can be gained by using its interface construct.
9. Java possesses simple form of concurrency control through its synchronized modifier. It is available on methods and blocks. In Java, it is easy to create concurrent processes called threads.
10. Java uses garbage collection which help the user to deallocate the storage for objects implicitly. Sometimes, if the languages do not have garbage collection there is a chance of memory leakage.
11. Implicit Type conversions are only allowed in Java.

Conclusion

Java has removed many unsafe features to make the language highly reliable and safer. Accessing the Index range checking of array makes the language safer. The scope of application can be increased by adding concurrency. It supports portability i.e., programs written on one system can be run on any other platform. This kind of portability implies cost of interpretation. The first version of Java interpreter is called Java Virtual Machine (JVM), which is 10 times slower than compiled C programs. With the help of Just-In-Time (JIT), Java programs are converted to machine code before execution. It makes the Java programs more efficient than C++ programs.

The usage of Java has been increased than other programming languages because of its value in programming dynamic web documents. Design of language and compiler/interpreter for Java can be obtained on web freely and easily. It is used in many application areas.

The recent version of Java i.e., Java 7 was released in 2017, which includes features like enumeration class, generics and a new iteration construct. Example of Java program is as follows,

//Java Example program

//Input: To check whether a given number is palindrome or not.

//Output: String is displayed as palindrome if it satisfies the condition.

Program

```
import java.io.*;
class Palindrome
{
    public static void main(String args[ ])
    {
        StringBuffer S1 = new StringBuffer
                          ("KATAK");
        StringBuffer S2;
        System.out.println("String="+ S1);
        S2 = S1.reverse( );
        if(S1.equals(S2) == true)
        System.out.println("String is a Palindrome");
        else
        System.out.println("String is not a
        Palindrome");
    }
}
```

Output

String=KATAK

String is a Palindrome

Q41. Discuss about scripting languages in detail.

Model Paper-I, Q2(b)

Answer :

Scripting Languages

Scripting languages were developed before 25 years. Earlier, these languages are used by inserting a sequence of commands in a file interpretation called script. It was also named as 'sh' (for shell), as it is a small collection of commands, which interprets a call to system subprograms to perform utility functions such as file management and simple file filtering. The widely used language is 'ksh' which was developed by David korn at Bell laboratories.

Awk language is scripting language which is developed by Al Aho, Brian kernighan and Peter Weinberger at Bell laboratories. It was started as a report generation language and then became a general purpose.

The various type of scripting languages are Java script PHP, Perl, Python, Ruby and Lua.

Origin and Characteristics of Perl

Larrywall developed the language called 'Perl' which is a combination of sh and awk. perl is considered as primitive programming language, powerful language typical imperative language and often a scripting language. Perl has all constructs, that can be applied to many computational problems.

Characteristics

Perl has many characteristics which are illustrated as follows,

1. In perl, variables are declared implicitly, There are three distinct namespaces for variables, They are,
   (i) Scalar variable name should begin with dollar sign ($).
   (ii) Array name must begin with @.
   (iii) Hash names must begin with %.
   These classification make variable names readable and easy to identify.
2. It includes implicit variables, which are used to store perl parameter, like newline character (or) characters used in implementation. These are also used as default parameters in built-in functions and default operands in some operators. It has different names such as $! and @. User defined names include three namespaces such as %, @, $.
3. In perl, array has two characteristics which differentiate it from other programming languages.
   (i) The arrays which have dynamic length can be added (incremented) and removed (decremented) as per requirement during execution.
   (ii) Arrays have gaps between the elements and the advantage is, these gaps does not occupy space in memory. The for each statements are used to iterate over the missing elements.
4. Perl also consists of associative arrays, which are also called hashes. The perl system provides hash function and if necessary it increases the size of the structure.

In Perl, scalar type stores both string and number, that are usually stored in a floating point form the numbers can be coerced to strings and viceversa depending on the context. Once if the string is used in numeric context, then it cannot be converted to number and zero can be used but the user does not receive any error message. So, this can lead to errors that cannot be detected by compiler or run time system.

Earlier, Perl is used as UNIX utility for text processing and also used as UNIX system administration tool. Perl is used in many application areas such as computational biology and artificial intelligence.

Example

The following program illustrates the use of arrays,

```
{
#Initialize an array with name countries
@countries = ("India", "Pakistan");
print "These are the contents of array \n";
print "@countries";
print "<br/>";
#Lets add some content to already present array
push(@countries, "Srilanka");
unshift(@countries, "Bangladesh");
print "These are the new contests of array \n");
print "@countries";
}
```

Origin and Characteristics of JavaScript

Java script was developed by Brendan Eic at netscape Earlier it is named as Mocha and then later renamed as Live script. In 1995, Livescript was handled by Netscape and Sun Microsystems and changed the name as JavaScript. JavaScript has extended from version 1.0 to version 1.5 with several features and capabilities. ECMA-262 is considered as language standard for JavaScript which was developed by the European Computers Manufactures Association (ECMA) in the late 1990's. It has approved by ISO as ISO-16262. The microsoft's version of JavaScript is named as JScript .NET.

JavaScript code is embedded in HTML documents and these documents displayed are interpreted by the browsers. The purpose of JavaScript in web programming is to check the input data form and create HTML document. It is also used with Rails web development framework. JavaScript is related to Java because it has similar syntax. There are differences in between Java and JavaScript. They are,

(i) Java is strongly typed, whereas JavaScript is dynamically typed.

(ii) JavaScripts arrays have dynamic length due to which it is not checked for validity but in java arrays are checked.

(iii) Java supports object-oriented programming whereas JavaScript does not support inheritance and dynamic binding.

JavaScript is mainly used to create and modify HTML documents dynamically. JavaScript defines an object hierarchy, which matches with a hierarchical model of an HTML document defined by the Document object Model. These objects are used to access the elements of an HTML document and provide the basis for dynamic control of elements in documents.

Example

JavaScript Program to Determine Whether an Integer is an Even or Odd Number

```
<html>
<head>
<title>JavaScript program </title>
</head>
<body>
<script language = "JAVASCRIPT">
var n;
n=parseInt(window.prompt("enter the value"));
if (n % 2 = = 0)
{
    document.writeln("<h2> It is an even number
    </h2>");
}
else
{
    document.writeln("<h2> It is an odd number
    </h2>");
}
</script>
</body>
</html>
```

Origin and Characteristics of PHP

PHP was created by Rasmus Lerdorf, in 1994. Originally, it was named as Personal Home Page (PHP). Later, it was changed by its user community to Hypertext PreProcessor, which is the recursive name for PHP.

PHP is an XHTML-embedded server-side scripting language. It means that the PHP code, can be embedded within the XHTML document to generate dynamic web pages. Since, PHP is a scripting language, it uses the PHP processor, to process the PHP code. PHP takes most of its syntax from C, Java and Perl. It is an open source which means that, its code is freely available and can be changed. It is written in C language and can run on most of the operating systems with almost all web servers.

PHP is related to the client-side of the JavaScript. If the XHTML document contains the embedded JavaScript code then the browser calls the JavaScript interpreter to interpret the script. When the browser requests web server for the XHTML document containing the PHP script, then the web server calls the PHP processor to process PHP code. A PHP filename has extension .php, .php3 or .phtml

PHP processor receives a PHP document file, as input and produces an XHTML document file, as output. It operates in two modes. In 'copy' mode, it copies the XHTML code, (which may contain embedded client-side script) to the output file, as it is. It works in 'interpret mode' if the input file contains the PHP script.

In this mode, it interprets the PHP scripts and sends the generated script output to the output file. Thus, the output from a PHP script is always an XHTML file that may include embedded client-side script. The output file is sent to the browser that requested the file. The client at browser can see only the XHTML code but not the PHP script. Even the **view source** option does not show the PHP script, because, the browser receives only the XHTML code from the server. Like JavaScript, PHP is purely interpreted language.

Like the JavaScript, PHP also uses dynamic typing. In PHP, variables are not declared with types. The type of a variable is determined, when a value is assigned to it. Arrays in PHP are dynamic. Arrays in PHP are the combination of arrays in other common programming languages and associative arrays in Perl. PHP supports procedural programming as well as object-oriented programming.

PHP's library includes, a large collection of functions, which makes it flexible and powerful tool for developing server-side software. It includes many functions that allow to develop interface for other software systems.

Origin and Characteristics of Python

Python is an object oriented interpreted scripting language, which was initially designed by Goido van Rossum in early 1990's, The current versions of Python are being developed in early 1990's at python software foundation. As this language is similar to Perl language, it can be used in application areas where perl is used. Some of these areas include system administration, CGI programming. Apart from this, Python language is basically used in implementing well-known web-search engine 'GOOGLE' and in different application areas that range from science fiction to real science.

Characteristics of Python

1. Python is an open source language and is accessible by common computing platforms.
2. Python scripts are brief but are readable and highly expressive.
3. Python is a compact language that depends on rich, set of library modules for performing high-level operations like string matching services.
4. Pythons is a dynamically-type language due to which the scripts of python doses not contain type in formation.
5. Python consists of three kinds of data structures i.e., lists, tuples (immutable lists) and dictionaries (hashes)
7. Python supports pattern matching capabilities, exception handling and garbage collection.
8. Python supports CGI programming and form processing through CGI module
9. Python consists of other modules that supports networking, database access, cookies.
10. Python provides support for concurrency through its thread and network programming with its sockets. Also it has more support for functional programming rather than non functional programming languages.
11. Python enables the user to extend sits functionality (Such as adding functions, variables and object types) through modules, which are written in any compiled language.

Origin and Characteristics of Ruby

Ruby was developed by Yukihiro matsumoto in the early 1990's but published in 1996. The reason for the development of Ruby language is to produce a pure object-oriented language. Perl and python support object oriented programming. They are not considered as pure object-oriented language.

Characteristics

1. The main feature of Ruby is, it is a pure object language same as smalltalk.
2. Every data value is considered as an object and the operations are done through method calls. Operators are used to specify method calls to perform operations, because methods can be redefined and classes can be divided into subclasses.
3. In Ruby, classes and objects are dynamic so that methods can be added to any of them. It implies that there will be different sets of methods during execution. A class can contain methods, data and constants.
4. The syntax of Ruby is similar to Eiffel and Ada. Here, variable declaration is not required because dynamic typing is used. Variable scope is mentioned in its name. A variable, which begins with a letter has localscope. Instance variable that begins with @ and variable which starts with '$' has global scope. Some of the features of perl are similar to Ruby, which include implicit variables.

Ruby can be extended (or) modified by any user. It is a first programming language developed in Japan, which is used widely in United States.

Origin and Characteristics of Lua

Lua was developed by Roberto Ierusalimschy, waldemar celes and Luis Henrique in the early 1990's which supports procedural functional programming. The languages influenced its design are scheme, icon and python.

There are similarities between Javascript and Lua. Both the languages does not support object-oriented programming. The objects in both the languages can act as classes and objects. It has prototype inheritance rather than class inheritance. But Lua can be extended to support object-oriented programming. Luas functions are first class values and it support closures, so that it can be used for functional programming with these abilities. It possess only a single data structure i.e., the table. It extends PHP's associate arrays, because functions are treated as first class values and stored in tables which are further used as namespaces.

It also uses garbage collection for the heap allocated objects and dynamic typing. It is considered as small and simple language with 21 keywords. The Lua language was designed to implement it in a simple ways in many application areas.

Lua can be used as an extension of scripting languages to other languages. Similar to Implementation of Java, Lua is converted to intermediate code and interpreted. It can be incorporated in other system which is 150 k bytes in size.

Lua was widely used in gaming industry during 2006 and 2007. Many scripting language, which are developed over the past 20 years were used widely and Lua is one among them.

Q42. Explain about,

(a) The Flagship .Net language (C#)

(b) Markup/programming hybrid languages.

Answer :

(a) The Flagship .Net Language (C#)

C# is a .NET language developed by Microsoft in 2000.

Design process of C#

C# depends on C++ and JAVA, and also includes capabilities from Delphi and Visual BASIC.

The purpose of C# is to provide support for .NET frame and to develop component based software for a language. The .NET languages such as C#, visual Basic .NET, managed C++, F# and JScript .NET use the Common Type System (CTS) that provides a common class library. The languages of .NET inherited from a single class root i.e., system.object. The compilers which meet the CTS specification can create objects and they are combined into software systems. The languages of .NET compiled in the form of intermediate language. As the IL is not interpreted, a JIT compiler is used to convert IL into machine code before execution.

Language Overview

Many will assume that the most of the features of C++ were excluded by Java. For instance, C++ supports multiple inheritance, pointers, structs, enum types operator overloading and goto statement. In Java, all of these were not included. As the designers of C# disagreed with the removal of features because all these features except multiple inheritance are added to the new language.

In many cases, C# version features are better than C++ features such as enum types of C# are safer than C++, because they never allowed to be converted to integers implicitly and in C#, the struct type was changed to a useful construct, but in C++ it serves for nothing.

As C++ includes function pointers, that had lack of security which is inherent in C++'pointers to variables. C# includes a new type, delegates which are considered as both object-oriented and type safe method references to subprograms. Delegates can be used to implement event handlers, to control the execution of threads and callbacks. But in java callbacks are implemented with interfaces and in C++, method pointers are used.

In C#, methods can take any number of parameters of same type. It is specified by the use of a formal param of array type which is followed by the keyword params. C# also include rectangular arrays, in which these are not supported by other programming languages and foreach statement is considered as iterator for arrays and collection objects. It is also found in perl, PHP and Java 5.0. C# also includes properties, these are specified with get and set methods as data members. This is called implicitly when references and assignments are assigned to data members.

C# was developed and evolved from 2002. The recent version is C# 2010. It has features like dynamic typing, implicit type and anonymous types.

Conclusion

C# is considered as general purpose programming language, which is improved better than C++ and Java. The features of C# are older but it includes some constructs which move beyond its predecessors.

Example Program

```
Using system;
name space Loop
{
class Example
{
public static void main( )
{
char[ ] students = { 'g', 'b', 'g', 'g', 'b', 'b', 'b'};
int girl = 0,
boy = 0;
for each (char s in students)
{
if(s = = 'g')
girl ++ ;
else if (s = = 'b')
boy ++ ;
}
console.writeline ("Number of girls = {0}", girl);
console.writeline ("Number of boys = {0}", boy);
}
}
}
```

(b) Markup/Programming Hybrid Language

Markup/programming hybrid language can be defined as a markup language in which programming actions such as control flow and computation are contained. The two hybrid languages are as follows,

XSLT

Extensible markup language is considered as metamarkup language, which is used to define markup languages. XML-derived markup languages are used to describe data documents. These are processed by computers, but it will be in the human readable format. Transformations of data documents take place to display or print effectively. It can be transformed to HTML, which is displayed through a web browser. In some cases, the data in the document is processed with other forms. XSLT (Extensible Style sheet Language Transformations) is an another markup language which transforms the XML documents to HTML documents. It can specify programming like operations and is defined by world wide consortium (W3C) in the late 1990's.

The transformations in XSLT are performed by using an XSLT processor, which takes an XML data document and XSLT document as input and while processing, the XML data document is converted to another XML document with the transformations defined by XSLT documents as per the templates. XML input file is associated with each template in which transformation instructions are given to transform the matching data before generating resultant document. Here, when the XSLT processor looks for pattern matching in XML document the templates acts as subprograms. At lower level, XSLT has programming constructs. These lower level constructs are represented with XSLT tags such as <for-each>.

JSP

The main part of Java Server pages standard Tag Library (JSTL) is an another markup/programming hybrid language. Before knowing JSTL, the concept of servelets and JSP is necessary to learn. A servlet can be defined as instance of Java, which depends on and executed on a web server system. The markup document requests for execution of a servlet in a web browser. After execution, the generated result will be in the form of an HTML document and returns to the requesting web browser. A servlet container is a program, which runs in the web server process and controls the execution of servlets. These are used for processing purpose and for accessing database.

JSP is defined as technology, which creates dynamic web documents and also provides other processing needs of web document. JSP contains HTML and Java document types, which are requested by web browser. The JSP processor program present on web server system converts the input document to servlet. The plain HTML is converted to java statement which is an output. To run Javaserver pages, a servlet container JSTL is a set of XML action elements, which also controls the processing of JSP document on the web server. These JSTL action elements are similar to HTML and XML. In JSTL control action elements 'if' is commonly used to specify a Boolean expression as an attribute. The if element description contains HTML code, and it is represented in output document if the expression satisfies the condition.

It is related to C/C++ of preprocessor command. The processing of servlet container is similar to the C/C++ processing. The preprocessor commands are used to instruct preprocessor to generate the out put file from input file. In the similar way, JSTL control action elements specify instructions to the JSP processor to build a XML output file from XML input file.

The 'if' element is also used to validate a form data given by browser user. JSP processor access the form data and tests the data with the help of 'if' element to check whether the data is sensible or not. If it is not a sensible data it gives an error message to user in the output document. JSTL consists of choose, when and otherwise elements for multiple selection control. It also includes for each element which is iterated over collections. The for each consists of begin, end and step attributes for controlling iterations.

1.2 SYNTAX AND SEMANTICS

1.2.1 General Problem of Describing Syntax

Q43. Discuss the general problem of describing syntax and write short notes on syntax, lexemes, language generators recognizers and language generators.

OR

Discuss about language recognizers and language generators.

(Refer Only Topic: Language Recognizer, Language Generator)

Answer : **(Model Paper-II, Q2(b) | Nov./Dec.-16(R13), Q3(a))**

Syntax

The syntax of a language is a set of rules that defines the form of a language. They define low expressions, sentences, statements and program units are formed by the fundamental units called words/lexemes.

Lexemes

The lexemes of a programming language refers to its identifiers, literals, operators and special words.

A token refers to the class of language lexemes. An identifier can be a token consisting of lexemes or instances such as "sum" and "total". Sometimes a token possess only a single lexeme.

Example

Consider the following statement in C.

$$i = 3 * c + 15;$$

The lexemes and tokens of the above C statement are,

Lexemes	Tokens
i	Identifier
=	Equal sign
3	Integer – Literal
*	Multiply – Operator
c	Identifier
+	Plus – Operator
15	Integer – Literal
;	Semicolon

The description of most of the programming languages can be made simple by using lexeme descriptions.

Language Recognizer

A language recognizer is a part of the compiler, responsible for syntax analysis of the translating language.

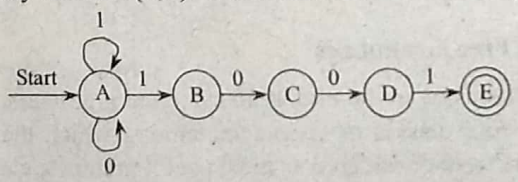
It doesnot test all the strings of characters resulted from a set of strings to find out whether each available string is in the language. It only checks whether the given programs are in the language or not. Finally, all the syntactic errors are determined and corrected by the syntax analyzer.

A language recognizer takes the string '*i*' and results to 'true' only if '*i*' is available in that language, otherwise it returns 'false'.

Example

Let '*L*' be a language that consists of a string of characters. To define a language using the recognition method, a mechanism called 'automata' is constructed. An automata is a recognition device responsible for reading the strings of characters from '*L*'. The automata is designed in such a way that it can read the strings and determines whether a particular string is a part of the language or not. The automata then accepts or rejects the string based on the availability of an input string in the language.

Let the language '*L*' consist of a string of bits '0' and '1' as 1001. The set of states are represented as {*A*, *B*, *C*, *D*, *E*} and the input symbols as {1,0}. This is shown in the following figure.



Figure

From the above figure, it is known that '*E*' is an accepted state and is recognized by the automata.

Language Generator

A language generator is a device responsible for recognizing the sentences of a language. They have the following advantages over language recognizers,

1. They are easy to read and understand.
2. They are more useful than the language recognizers because they are not only used in trial and error mode but also in other modes.
3. The correctness of the statement's syntax in language recognizers can be determined by submitting an expected statement to check whether it is accepted by a compiler or not but for a language generator, the correctness of the statements syntax can be determined by simply comparing the statement with the structure of the generator.

The language generators are also called the grammars which are used for describing the languages syntax. We have two types of grammars. They are,

(i) Regular grammar and

(ii) Context-free grammar.

The regular grammar describes the tokens of a programming language whereas, the context-free grammar describes the syntax of programming languages with little exceptions.

BNF is a notation used to specify the syntax of the programming languages. It is similar to context-free grammars.

1.2.2 Formal Methods of Describing Syntax

Q44. Explain the formal methods of describing syntax.

Answer :

Formal language generation mechanisms, which are called grammars are used to define the syntax of the programming languages. Some of the grammars are as follows,

1. Backnus-Naur form and context free grammars
2. Extended BNF
3. Grammars and recognizers.

1. Backnus-Naur Form and Context Free Grammars

Noam Chomsky and John Backus, in the mid of late 1950's developed the similar syntax description format which has been considered widely used method for programming syntax.

Context Free Grammars

Chomsky, in the mid 1950's defined grammars, which describe four classes of languages, among which these two grammars were considered as most useful grammar classes to describe the programming languages syntax. They are,

(i) Regular grammar

(ii) Context free grammar.

Regular grammars describes the tokens of programming language and context free grammar describes the syntax of programming languages with few exceptions.

Origin of Backus-Naur Form

After the completion of Chomsky's research on language classes, John Backus, a well known number of ACM GAMM group presented paper at an international conference in 1959 to describe ALGOL 58. It is a new formal notation to specify syntax of programming language. Later it was slightly changed by peternaur to describe ALGOL 60. This modified method to describe syntax is known as Backus-Naur Form (or) BNF.

BNF is similar to context free grammar. BNF notation is used to specify the syntax of programming.

Fundamentals

BNF is considered as meta language for programming languages, which is used to describe other programming languages.

For remaining answer refer Unit-I, Q46(a).

Describing Lists

For answer refer Unit-I, Q46(b).

Grammar

For answer refer Unit-I, Q47, Topic: Grammar.

Derivation

For answer refer Unit-I, Q47, Topic: Derivation.

Parse Tree

For answer refer Unit-I, Q47, Topic: Parse Tree.

Ambiguity

For answer refer Unit-I, Q49, Topic: Ambiguity.

Operator Precedence

For answer refer Unit-I, Q49, Topic: Operator Precedence.

Associativity of Operators

For answer refer Unit-I, Q50, Topic: Associativity of Operators.

An Unambiguous Grammar for if-then-else

For answer refer Unit-I, Q51.

2. Extended BNF

For answer refer Unit-I, Q52, Topic: Extended BNF (EBNF).

3. Grammars and Recognizers

For answer refer Unit-I, Q52, Topic: Grammars and Recognizers.

Q45. Contrast syntax analysis of a particular programming language. Describe how syntax for routine is specified.

Nov./Dec.-18(R15), Q3

Answer :

Syntax Analysis

Syntax analysis is considered as second phase of compiler. It is also called as Hierarchical analysis. This analysis combines source program tokens into grammatical production.

Parse trees are used in syntax analysis to show sentence structure, but it contains redundant information because of implicit definitions.

As the lexical analysis uses regular expressions and pattern rules to identify tokens. It cannot check the syntax of a given sentences, due to limitations of regular expressions. So, syntax analysis phase uses context free grammar, CFG is considered as useful tool to describe programming language syntax.

Syntax

```
datatype routine name (parameters)
{
statements
}
```

The above syntax represents that data type is the return type of data, routine name is the name of a routine and parameters represents list of arguments.

Example

```
int sum (int a)
{
int i, b;
b = 0;
for (i = 1; i < = n; i + +)
b + = i
return b
}
```

The above example represents routine invocation which provides routine activation.

- Routine name, parameter type and the return types are present in the Routine header.
- Here parameter is of type int and return type is int.

The Routine activation gives identity to the routine and describes the parameters upon which the routine function is performed. The call statement should be with in the scope of the routine once the routine gets activated by a call. Thus the routines hold their own scope. Besides this, the routine also specifies the scope for the declaration present within them and this is referred to as local declaration which gets visible in the routine itself. Interestingly the routines can point to non local items based upon the scope rules of the language. The global items are those which are visible to every unit with in the program.

Q46. Write a short notes on,

(a) Fundamentals

(b) Describing lists.

Answer :

(a) Fundamentals

BNF is a notation used for describing the syntax of a programming language. It is a context-free grammar developed by John Backus and Naur Chomsky.

Syntax

The syntax rules or BNF notation of a '*C*' program is as follows:

<program>::={<statement>*}

<statement>::=<assignment>|<conditional>|<loop>

<assignment> ::=<identifier>=<expression>

<conditional>::= if<expr> {<statement>*}|

if<expr>{statement>+}else

{<statement>+}

<loop>::= while<expr> {<statement>+}

<expr>:: = <identifier>|<number>|(<expr>)|

<expr> <operator> <expr>

The left hand side of the above statements are the abstractions being defined. They are called non-terminal symbols or non-terminals.

The right hand side of the above statements can be the definitions, rules, productions, tokens or lexemes. They are also called as 'terminal symbols' or 'terminals'. A grammar refers to the collection of rules.

The lexical rules of '*C*' language are,

<operator>::= + | − | * | / | = | ! = | < | > | <= | >=

<identifier>::= <letter> <id>*

<id> ::=<letter>|<digit>

<number> ::=<digit>*

<letter>::= a | b | c | d |....| z

<digit>::= 0 | 1 | 2 | 3 | 4 |....| 9.

(b) Describing Lists

BNF does not uses ellipsis for describing the variable-length lists, instead it employs a process called 'recursion' i.e., a rule is said to be 'recursive' if its L.H.S appears in its R.H.S.

Example

<ident_list> → identifier|identifier, <ident_list>

Q47. Define grammars, derivation and a parse tree.

Answer : (Model Paper-III, Q2(b) | Nov./Dec.-16(R13), Q2(b))

Grammar

It is a generative device used for language definitions. In any language, the sentences are produced by applying a series of rules starting with a special nonterminal symbol called the start-symbol.

Derivation

The process of generating sentences is called a derivation.

The grammar for a complete programming language consists of a start symbol called <program> which represents the complete program.

Example

Consider the following grammar,

<program> → **begin** <stt_list> **end**

<stt_list> → <stt> | <stt> ; <stt_list>

<stt> → <var> = <expr>

<var> → X | Y | Z

<expr> → <var> + <var> | <var> – <var> | <var>

A derivation for the above grammar is as follows,

<program> ⇒ **begin** <stt_list> **end**

⇒ **begin** <stt> ; <stt_list> **end**

⇒ **begin** <var> = <expr> ; <stt_list> **end**

⇒ **begin** X = <expr> ; <stt_list> **end**

⇒ **begin** X = <var> + <var>; <stt_list> **end**

⇒ **begin** X = Y + <var>; <stt_list> **end**

⇒ **begin** X = Y + Z ; <stt_list> **end**

⇒ **begin** X = Y + Z ; <stt> **end**

⇒ **begin** X = Y + Z ; <var> = <expr> **end**

⇒ **begin** X = Y + Z ; Y = <expr> **end**

⇒ **begin** X = Y + Z ; Y = < var> **end**

⇒ **begin** X = Y + Z ; Y = Z **end**

The above derivation begins with the start symbol called <program> and the symbol '⇒' is read as '*derives*'. Each string in succession is then derived from the previous string by substituting one of the nonterminal's definitions in place of nonterminal symbol. Each string used in the derivation is called a *sentential form.*

The derivation in which always the leftmost nonterminal symbol is substituted is called a *leftmost-derivation.* This process of derivation proceeds until no nonterminals are present in the sentential form. The derivation can also be rightmost or in any order, because it doesn't have any effect on the language generated by a grammar.

Parse Tree

The sentences in a language can be hierarchically described by using 'parse trees'.

For a grammar $G = (V, \Sigma, P, S)$, the parse tree representation is as follows,

(i) The root node of a parse-tree is marked with the start symbol 'S' of a grammar.

(ii) Each internal node is marked by a variable in V.

(iii) Each leaf (terminal node) is marked by a terminal or a variable or $\in$. If $\in$ is used to mark a leaf then it specifies that it is the only child of its parent

(iv) If an internal node is marked as 'x' and its children are labelled from left as $x_1, x_2, x_3, \ldots x_n$ then $x \to x_1, x_2, \ldots, x_n$ is a production in P.

Each subtree of a parse tree denotes an abstraction instance in a statement.

Example

The parse tree for an expression $P := Q * (P + R)$ is shown in the following figure,

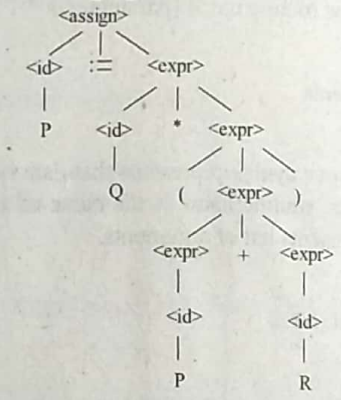


Figure: A Parse Tree

Q48. Construct the parse tree for the simple statement.

A : = B* (A + C).

Answer : Nov./Dec.-17(R15), Q2(b)

The parse tree for a simple statement A : B = * (A + C) is as follows,

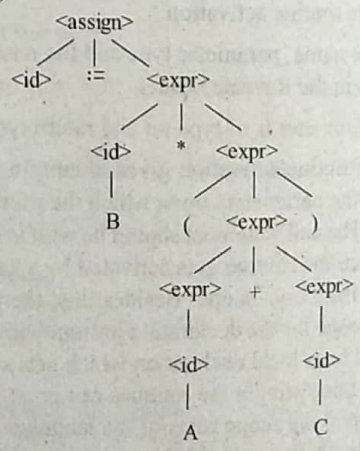


Figure: Parse Tree for A := B * (A + C)

Q49. Define ambiguity, operator precedence and associativity of operators.

Answer :

Ambiguity

A grammar that results in a sentence for which there are two or more parse trees, is said to be 'ambiguous'. Such grammars allows the parse trees to grow in both left and right directions.

Example

Consider the following ambiguous grammar.

<assign> → <id> = <expr>

<ids> → P|Q|R|S

<expr> → <expr>+<expr>|<expr>*<expr>|(<expr>)|<id>

This grammar is ambiguous because, any sentence of the form $P = Q + R * S$ results in two distinct parse trees, shown in figure (a) and figure (b).

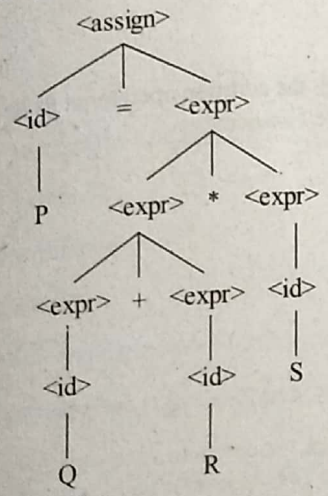


Figure (a): Parse Tree-1

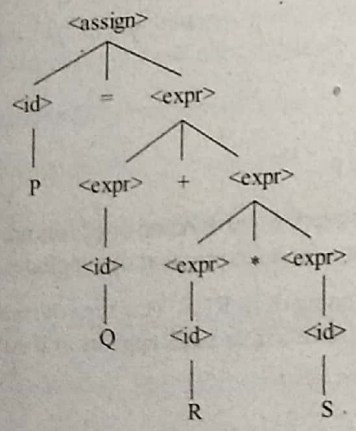


Figure (b): Parse Tree-2

If a language structure consists of many parse trees, for a single statement, then the meaning of a structure cannot be predicted uniquely.

Operator Precedence

The meaning of the syntactic structure of a grammar can be determined from its parse tree. An operator which is shown at the lower level in the parse tree has a higher precedence than the operators at the higher levels in a tree. Consider the parse tree shown in figure (c), in which the multiplication operator, having the higher precedence, is shown at the lower level to indicate that it has higher precedence over the addition-operator.

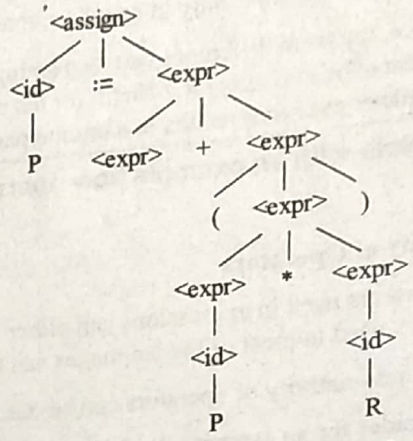


Figure (c): Parse Tree for an Expression P = Q + (P * R)

For an unambiguous grammar, the parse tree of an expression irrespective of any number of multiple operators, places the rightmost operator in the expression at the lowest level in the tree, while keeping all the subsequent operators at the higher levels in the tree. This is shown in figure (d).

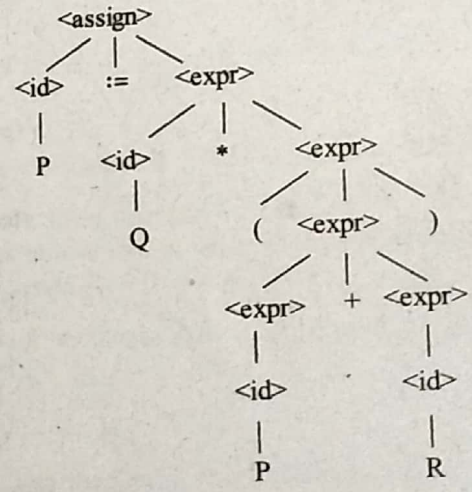


Figure (d): Parse Tree for an Expression P = Q * (P + R)

In the above parse tree, the addition operator is the rightmost operator in an expression and hence shown at the lowest level in a tree giving it a higher precedence than the multiplication operator at the left.

The purpose of writing grammar is to make a separation between the addition and multiplication operators in such a way that the higher-to-lower ordering is maintained in the parse tree irrespective of the order in which the operators appear in an expression.

The correct operator ordering is specified by using separate abstractions for the operands. The parse trees and their derivations are linked with each other, one can be constructed from the other. Each derivation of an unambiguous grammar possess distinct parse trees though each parse tree is expressed by different derivations.

Associatively of Operators

For answer refer Unit-I, Q50.

Unambiguous Grammar

A grammar that results in a sentence for which there is only one parse tree is said to be unambiguous. Such grammar allows a parse tree to grow only in one direction (either left or right).

Hence, the main difference between ambiguous and unambiguous grammar is that the ambiguous grammar results in large number of parse trees making it difficult for the compiler to determine the meaning of the language structure uniquely, whereas, an unambiguous grammar results in a unique parse tree which clearly depicts the meaning of the language structure.

Q50. Explain with an example how operator associativity can be incorporated in grammars.

Answer : Nov.-15(R13), Q2(b)

Associativity of Operators

Operators used in expressions can either be left associative or right associative. Associativity of operators specifies that the operators used in most of the languages can be grouped from left-to-right.

The associativity of operators can be described using parse trees by showing the operators in proper hierarchical order.

Consider for an example the following assignment statement.

$$P = Q + R + P$$

The parse tree for this statement is shown in the following figure in which the addition operator at its left is shown at the lower level than the right addition operator. This order is correct, if addition is left associative.

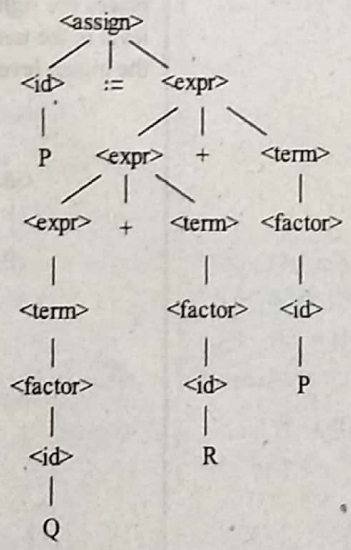


Figure: Parse Tree for P = Q + R + P

However, the associativity of addition in computer is immaterial. The subtraction and division operators are not associative in both mathematics and computers. Hence, the accurate association is necessary, for an expression that includes either of them.

A grammar rule is said to be left-recursive, if its LHS appears at the beginning of its RHS. This type of recursion indicates left associativity. On the other hand, a grammar rule is said to be 'right associative', if its LHS appears at the right end of the RHS. This type of recursion indicates right associativity.

Examples of Left-associative Operators

Addition (+) and multiplication (*) are the left-associative operators.

Example of Right-associative Operators

Exponentiation (↑) is a right-associative operator.

Q51. Write an unambiguous grammar for it-then-else.

Answer :

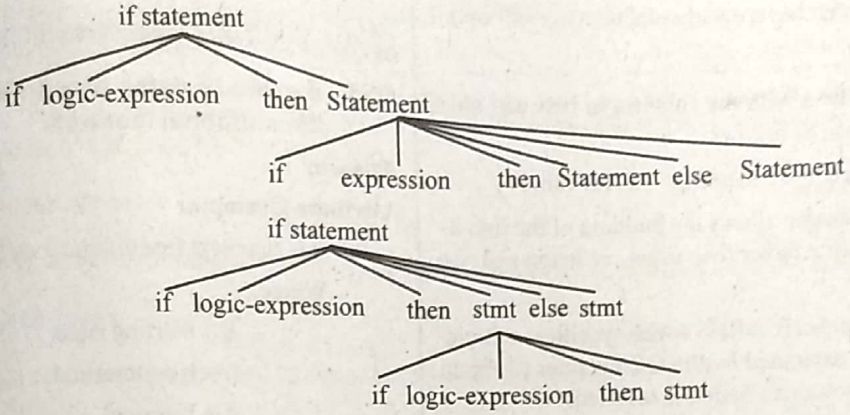
A grammar is said to be unambiguous, if it generates only one parse tree for a sentence.

BNF rules for an Ada if-then-else statements are as follows,

<if-stmt> if <logic-expr>then<stmt>

if <logic-expr>then<stmt>else<stmt>

The two possible parse trees are as follows,



Example

Consider an example of construct,

if done = = true

then if denominator = = 0

then quotient = 0;

else quotient = numerator/denominator;

The second parse tree is used as basis for translation the else clause is executed only if it does not satisfy the condition.

In the above parse trees, if statement cannot be used without an else between then and its matching else. To solve this situation, statements should be represented as that of matched and which are unmatched. In unmatched statements else-less if and other statements are matched. The Earlier grammar represents all the statements which are matched.

To represent different categories of statements, different abstractions or non terminals should be used. The ambiguous grammar to represent these ideas are as follows.

Unambiguous grammar for if-then-else:

<stmt> → <matched> | <unmatched> | <matched> → if <expr> then <matched> else <matched> lany non-if statement

<unmatched> → if <expr> then <stmt> | if<expr> then <matched> else <unmatched>

For above grammar, there is a possibility of only one parse. The sentential form is as follows,

if<expr> then if <expr> then <stmt> else <stmt>.

Q52. Define Extended BNF, Grammars and Recognizers.

Answer :

Extended BNF (EBNF)

EBNF is a technique used to eliminate some of the inconveniences that occur in BNF. EBNF enhances the readability and curitability of BNF without increasing its descriptive power.

The extensions that have been included in EBNF are,

1. The optional part in RHS is delimited by the brackets.

Example

The selection statement in C can be described as follows.

<selection_stt> → if(<expr>) <statement> [else <statement>]

If brackets are not used then the syntactic description of the above statement requires two rules.

2. The use of braces in RHS which specifies that the enclosed part can be repeated indefinitely or left out.

Example

Lists of identifiers with the commas in between them can be described as follows.

<ident_list> → <identifier>, {<identifier>}

The above extension allows the building of the lists by using a single rule rather than using recursion and two rules.

3. The use of parenthesis in RHS which specifies multiple-choice options separated by the OR operator (|). From the multiple-choices an option is selected.

Example

<term> → <term> (*|/|%) <factor>

The brackets, braces and parenthesis used in EBNF extensions are called 'meta symbols' i.e., they are 'notational tools', but not the 'terminal symbols'.

Example for BNF and EBNF Versions

The BNF and EBNF versions of an expression grammar are shown below.

BNF Version

<expr> → <expr> + <term>
|<expr>–<term>
|<term>

<term> → <term> * <factor>
|<term>/<factor>
|<factor>

<factor> → <expression> ** <factor>
|<expression>

<expression> → (<expr>)
|id

EBNF Version

<expr> →<term> {(+|–) <term>}

<term> →<factor> {(*|/)<factor>}

<factor> → <expression> {**<expression>}

<expression> → (<expr>)
|<id>

Grammars and Recognizers

For a context-free grammar, a recognizer for the language, produced by the grammar can be constructed algorithmically. A large number of systems have been developed for this purpose.

Syntax analyzers, also called the 'parsers' are responsible for constructing parse trees for the given programs. Two types of parsers exist based on the manner in which they are constructing the parse trees. They are top-down parsers, which build a tree from root to leaves and bottom-up parsers, which build a tree from leaves to the root.

1.2.3 Attribute Grammars

Q53. Explain in detail about attribute grammar and its additional features.

Answer : Model Paper-I, Q3(b)

Attribute Grammar

It is a context free grammar of the form $G = (S, N, T, P)$.

Where,

S = Starting state

N = Non-terminal

T = Terminal

P = Production.

Attribute grammar is a formal approach for describing and checking the correctness of static semantics of a program.

Static Semantics

The static semantics of a program refers to the process of analyzing the program specifications at the compile time. These type of rules are not directly related to the meaning of an executing program, but is usually associated with the programming language rather than its semantics. These syntax of the rules can also define the type constraints related to the language. These are called 'static' because the specifications are checked at compile time.

There exists some characteristics related to the structure of programming languages that are difficult otherwise impossible, to describe using BNF. An example of such a characteristic is type compatibility rules. In java, value of floating type can't be assigned to an integer type variable, but the opposite is possible, this limitation can be specified in BNF with the use of extra nonterminal symbols and rules. On specifying all the typing rules in BNF, the grammar will become large and complex.

An example of a language rule that can't be described using BNF is that, declaration of all the variables must be done prior to referencing them. Another example is that, if the *end* of an Ada subprogram is followed by the name, then this name must coincide with the subprogram name.

Hence, the two categories of the problems stated above, can be solved using static semantics rules.

Additional approaches such as attributes, attribute computation function and predicate functions have been added to an attribute grammar.

Attributes

Attributes are similar to variables and are related to grammar symbols.

Attribute Computation Functions

It is also called as semantic function. These functions are associated with grammar rules. They represent how attribute values are computed.

Predicate Functions

They are associated with grammar rules and specify the syntax and static semantic rules of the language.

Consistency among the attributes are also verified by the predicate functions.

Defining Attribute Grammar

1. For each symbol Y, a set of attributes $A(Y)$, consisting of two disjoint sets $S(Y)$ and $I(Y)$, are associated. These disjoint sets are called 'Synthesized' and 'Inherited' attribute sets.

   (a) Synthesized Attributes

   They are used to move the semantic information up in a parse tree.

   (b) Inherited Attributes

   They are used to move the semantic information down the parse tree.

2. A grammar where in a set of semantic functions and a null set of predicate functions are associated with each grammar rule is attribute grammar. For a given rule,

$$Y_0 \rightarrow Y_1\ Y_2\ Y_3\ .....\ Y_n.$$

The synthesized attributes of Y_0 can be computed as follows,

$$S(Y_0) = f(A(Y_1), A(Y_2), A(Y_3), ..., A(Y_n)).$$

The inherited attributes of Y_n can be computed as follows,

$$I(Y_n) = f(A(Y_0), A(Y_1), A(Y_2), ..., A(Y_n)).$$

A predicate function takes the form of a boolean expression on the attribute set $\{A(Y_0), A(Y_1), A(Y_2), ..., A(Y_n)\}$.

The only derivations allowed in attribute grammar are those in which every predicate associated with every non-terminal is true. If it results in a false value, then the syntax or semantic rules of a language is violated.

Intrinsic Attributes

These are the synthesized attributes of terminal nodes whose values can be deduced outside the parse tree.

Example

The type of a variable instance can be determined by using certain declarative statements from the symbol table, where the variable names along with their types are stored. Initially, it is assumed that an unattributed parse tree is generated and the determination of attribute values are required. The only attributes available, (along with their values) are the intrinsic attributes of the terminal nodes. With the given intrinsic attribute values, the semantic functions can determine other remaining attribute values.

Computing Attribute Values

For answer refer Unit-I, Q56(a).

Evolution of Attribute Values

For answer refer Unit-I, Q56(b).

Q54. Distinguish between ambiguous grammar and attribute grammar with an example.

Answer : Nov./Dec.-17(R15), Q2(a)

	Ambiguous grammar		Attribute grammar
1.	A grammar is said to be 'ambiguous' if it results in two or more parse trees. Such grammars allow the parse trees to grow in both left and right directions.	1.	A grammar is a context free grammar and is used to define and determine the correctness of static semantics of a program.
2.	There are several characteristics of a grammar and it is said to be 'ambiguous' i.e., if grammar generates a sentence with more than one left most derivation and one right most derivation.	2.	There are several features of attribute grammar such as attributes, attribute computation functions and predicate functions.
3.	Few passing algorithms mass depend on the ambiguous grammars and use nongrammatical information provided by the developer to build the concise parse tree.	3.	A parse tree of an attribute grammar is based on its BNF grammar with an empty set of values attached to each node. If all the values of attributes were calculated in a parse tree, it is said that the tree is fully attributed.
4.	Ambiguous grammar can be defined as G = (V, T, P, S) where 'V' is finite set of variable, 'T' is a finite set of terminals, 'P' is finite set of production and 'S' is the start symbol.	4.	There are syntax rules and semantic rules for attribute grammar.
5.	This grammar has more than one syntactic structure. So, meaning of structure cannot be determined uniquely.	5.	The basic concepts of attribute grammars are used in every compiler atleast informally.
6.	There are some languages, which only admit ambiguous grammars, those languages are called as inherently ambiguous language.	6.	Attribute grammar is classified into two types (i) Synthesized attributes (ii) Inherited attributes.
7.	Example for ambiguous grammar <assign> → <id> = <expr> <id> → P | Q | R | S <expr> → <expr> + <expr> | <expr> * <expr> | (<expr>) | <id> The above grammar is ambiguous, because the form P = Q + R * S generates two parse trees.	7.	Example for attribute grammars for simple expressions $E \rightarrow E_1 + T$ $T \rightarrow T * F$ $E \rightarrow T$ **Syntax** — **Semantic Rule** $E_1 \rightarrow E_1 + T$ — E.val : = E_1.val + T.val $T \rightarrow T * F$ — T.val : = T.val * F.val $E \rightarrow T$ — E.val : = T.val

Q55. Write the syntax and semantic rule of an attribute grammar for simple assignment statements.

Answer : Nov./Dec.-17(R15), Q4(b)

A parse tree of an attribute grammar comprises of an empty set of attribute values associated with each node. Once, all the attribute values in a parse tree are computed, the tree is said to be fully attributed.

Consider the attribute grammar for an assignment statement. Let x, y, z be the int or real type variables. The variables type on the left side of the assignment statement must match with the type at the right side.

Assignment Statement

Let the assignment statement be $x = x + y$.

The syntax and semantics of an attribute grammar can be represented as,

1. **Syntax Rule**

<assign> ⇒ <var> := <expr>

Semantic Rule

<expr>.expected_type ⇐ <var>.actual_type

2. **Syntax Rule**

<expr> ⇒ <var> [2] + <var> [3]

Semantic Rule

<expr>.actual_type ⇐ if(<var> [2].actual_type=int) and (<var> [3] actual_type=int)

then int

else real

end if

Predicate

<expr>.actual_type = <expr>.expected_type

3. **Syntax Rule**

<expr> ⇒ <var>

Semantic Rule

<expr>.actual_type ⇐ <var>.actual_type

Predicate

<expr>.actual_type = <expr>.expected_type

4. **Syntax Rule**

<var> ⇒ $x|y|z$

Semantic Rule

<var>.actual_type ⇐ look_up(<var>.string)

The look_up function is used to find a given variable name in the symbol table and returns the type of a variable, when found. In the above grammar, bracketed numbers are added after the repeated node labels in the tree in order to reference them unambiguously. A synthesized attribute is used to store the actual type, int or real of a variable or expression.

An inherited attribute is associated with the non-terminal <expr> and is used to store the expected type either int or real for an expression.

Q56. Write short notes on,

(a) Computation of attribute values

(b) Evaluation of attribute grammar.

Answer :

(a) Computing Attribute Values

Attribute values in a parse tree can be computed by a process called "parse tree annotation or decorating the parse tree". There are two ways in which parse trees can be decorated.

Top Down Order

In this process, the attributes follows a top down order i.e., they are evaluated from the root to the leaves if and only if all the attributes are inherited.

Bottom up Order

In this process, the attributes follows a bottom up approach i.e., the attributes are evaluated from leaves to the root if and only if they are synthesized.

As the grammar contains both synthesized and inherited attributes, the evaluation process must proceed in both the directions.

One possible evaluation order is shown below.

(i) <var>.defined_type ← look_up (*A*)

(ii) <expr>.resultant_type ← <var>.defined_type

(iii) <var>[2].defined_type ← look_up (*A*)

(iv) <var> [3].defined_type ← look_up (*B*)

(v) <expr>.defined_type ← float|real

(vi) <expr>.resultant_type = <expr>.defined_type which is TRUE|FALSE.

Consider an expression $P = P + Q$, the parse tree for this expression is shown in figure (a) and the flow of attribute values is shown in figure (b).

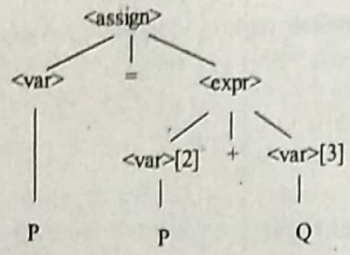


Figure (a): Parse Tree for P = P + Q

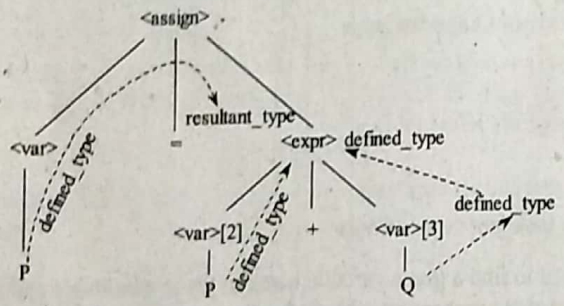


Figure (b): Flow of Attribute Values

In figure (b), the solid lines are used for the parse tree and the dashed lines are used for showing the flow of attribute values in a tree.

Figure (c) shows the final attribute values on the nodes, such as 'float' for the variable '*P*' and 'int' for variable '*Q*'. Finding the evaluation order for the attributes is a complex task and involves the construction of a dependency graph to depict all the dependencies that exists among the attributes.

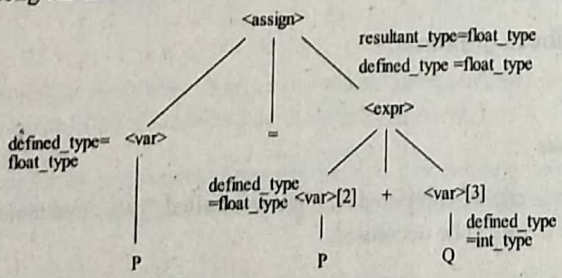


Figure (c): A Fully Attributed Parse Tree

(b) Evaluation of Attribute Grammar

Evaluating the static semantic rules is a key component of all compilers. If a compiler writer is unaware of an attribute grammar, then writer must use their basic ideas to design the compiler regarding the checks of static semantic rules. One of the main challenge in using a grammar attribute is to define all the real programming languages syntax and static semantics is the size and complexity of the attribute grammar. Huge number of attributes and semantic rules are needed for a complete programming language to make it difficult to write and read such grammar. In fact, the values of the attributes on a large parse tree are expensive to evaluate. On the other side, less formal attribute grammars are efficient and widely used tool for compiler writers, those concerned with compiler process than in formalism.

1.2.4 Describing the Meanings of Programs

Q57. Describe the meaning of programs using various approaches. Model Paper-II, Q3(b)

Answer :

Dynamic Semantics

Dynamic semantic can be defined as meaning of the expressions, statements and program units of a programing language.

The need of methodology and notation to describe semantics depends on many different reasons such as,

(i) Programmers must know about language statements usage before, so that they can use them in programs effectively.

(ii) If a programming language had a precise semantic specification, so that it can be proven correct without testing.

(iii) To avoid ambiguity and inconsistency in the designs for languages designers, who would develop the semantic descriptions of their languages.

(iv) A tool can use complete specification of the syntax and semantics of a programming language to generate a compiler for the language.

Some of the approaches which are suitable for imperative language to describe the meaning of programs are as follows,

1. Operational semantics
2. Denotational semantics
3. Axiomatic semantics.

1. Operational Semantics

Operational semantics explains the meaning of a program by executing its statements on a real or a simulated machine.

The variations that arises in the machine's state specifies the meaning of that statement. For example, describing the meaning of a program by specifying each computation step.

It loosely corresponds to interpretation.

2. Denotational Semantics

Denotational semantics describes the meaning of a program as a mathematical object. It is a function that takes the systems state as its input and produces an updated state as its output, after executing the program.

Recursive function theory forms the basis for denotational semantics. It loosely corresponds to 'compilation'.

The process of creating a denotational specification for a language involves the following steps:

(i) Defining a mathematical object for each entity in a language.

(ii) Defining a function that can map language entities, instances to mathematical objects instances.

For example, denotational semantics of functional languages translate it to domain theory.

3. Axiomatic Semantics

This method was developed to prove the correctness of programs. It describes the logical characteristics of programs i.e., if it signifies that some property is valid before starting the program execution then, some other property will replace it after executing the program. For example, the semantics of a program to find the minimum of the two numbers '*a*' and '*b*' states that if '*a*' is less than '*b*', then after program execution, a variable 'min', which used to store the minimum of the two values contains *a*'s value.

It is based on formal logic and makes no distinction between a phrase's meaning and the logical formulas that describes it.

Q58. Describe the meaning of program using operational semantics. Explain with suitable example.

Answer :

Description of Program Meaning Using Operational Semantics

For answer refer Unit-I, Q57, Topic: Operational Semantics.

The use of operational semantics involves various levels. These levels are categorized as,

(i) **Highest Level**

In this level, the operational semantic is used to determine the final result for the complete program execution. This level of use is called natural operational semantics.

(ii) **Lowest Level**

In this level, it is used to obtain the specific meaning for a single statement. This level of use is called structural operational semantics.

Basic Process

An operational semantics description of a language can be created by designing an appropriate intermediate language wherein the language clarity should be considered as its basic characteristics. Each constructs of the intermediate language must contain clear and definite meaning. Here, the language is at the intermediate level. It is due to the low-level machine language which is easy to understand and also due to the other higher-level language which is clearly unsuitable. The use of semantic description in natural operational semantic requires to construct a virtual machine (like interpreter) for the intermediate language. This machine is then utilized for executing a single statement, code segment, or the complete programs. But, in case of structural operational semantics, the semantic description does not require any virtual machine.

In operational semantic, the basic process involved is usual. However, its concepts are commonly used in the reference manuals of programming languages. For an instance, in C language, the semantics of **for** construct can be clearly explained in other simple statements as follows,

Statements of C	**Operational Semantics**
	exp1;
for (exp1; exp2; exp3)	Loop : **if** exp 2 = = 0
{	**goto** out
.....	.....
}	exp3;
	goto loop
	out: .....

In case of low level language used for operational semantics, for example consider the list of statements given below,

iden = var

iden = iden + 1

iden = iden – 1

goto label

if var relop var **goto** label

Here,

relop = one of the relational operators taken from { =, < >, >, <, > = , < = }.

iden = identifier

var = an identifier or a constant

The above statements are simple that can be easily understood and implemented. If the above three assignment statements are slightly generalized, then they will allow to describe more common assignment statements and arithmetic expressions. Therefore, the changed statements are,

iden = var bina_ope var

iden = una_ope var

Where,

bina_ope = Binary arithmetic operator

una_ope = Unary operator

However, this generalization might become complicated due to multiple arithmetic data types and automatic type conversions. But, an addition of some relatively simple instructions will allow to describe the semantics of arrays, records, pointers and sub-programs.

Evaluation of Operational Semantics

1. Operational semantics provides an efficient mechanism for describing the semantics for language users and implementors when used in an informal manner.
2. It becomes extremely complex when used formally.
3. It is basically an interpreter defined.
4. Operational semantics is based on low-level programming languages i.e., the statements of one programming language can be best described by using the statements of lower-level programming languages.

Q59. Explain in detail about denotational semantics.

Answer :

Denotational Semantics

For answer refer Unit-I, Q57, Topic: Denotational Semantics.

The method is called as denotational because the mathematical objects denotes the meaning of their respective syntactic entities.

The mapping function of a denotational semantic programming language consists of domain and a range. The domain is defined as the collection of values that are legitimate parameters for the function and the range can be defined as collection of objects to which the parameters are mapped.

In denotational semantics, the domain is considered as syntactic domain because, it is the mapping of syntactic structure and the range is considered as semantic domain.

Examples

Consider an example, which represents character string of binary numbers to implement denotational method. The syntax of binary numbers is defined by grammar rules, it is given as follows,

<bin_num> → '0'
| '1'
| <bin_num> → '0'
| <bin_num> → '1'

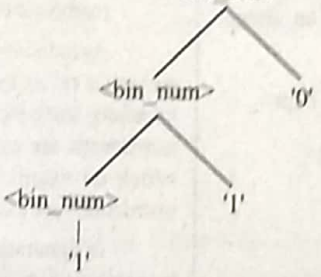


Figure: Parse Tree for Binary Number 110

The above figure shows a parse tree for binary number 110, place apostrophes around the syntactic digits to represent that they are not math digits.

Syntactic domain of the mapping function for binary numbers is the set of all character string representations of binary numbers. The semantic domain, which is symbolized by N is the collection of nonnegative decimal numbers.

Actual meaning is associated with each rule that has a single terminal symbol as its RHS. So, in the above example, the first two grammar rules must be associated with decimal numbers and the other two grammar rules are computational rules because, they combine a terminal symbol that is associated with an object with a nonterminal that can be expected to represent some construct.

In a syntax rule, a nonterminal as its RHS need a function that calculates the meaning of LHS, representing the meaning of entire RHS.

The semantic function called M_{bin}, which maps the syntactic objects to the object in N, the set of non negative decimal numbers. The M_{bin} function is defined as,

M_{bin} ('0') = 0

M_{bin} ('1') = 1

M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)

M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num> + 1)

State of a Program

The denotational semantics of a program can be described in terms of the variations that occurs in the state of an ideal computes i.e., it is defined in terms of the values of all the variables present in a program. These state changes can be defined by thorough mathematical functions.

If 'S' denotes the state of a program then, it can be expressed as a set of ordered pairs as,

$$\{<a_1, val_1>, <a_2, val_2>, ..., <a_n, val_n>\}$$

Where,

$a_1, a_2, ..., a_n$ are the variable names and $val_1, val_2, ..., val_n$ are the values of these variables.

Many of the variables values are represented as **undef** which specifies that the variable is currently undefined, let 'VARMAP' be a function which takes two parameters of which the first is the variables name and the second is the program state.

Expressions

Expressions are the most basic units in programming languages. In denotational semantics, they do not have any side effects.

Let us take simple expressions consisting of only two operators + and *, each expression has at most one operator, the scalar variables and integer literals as its operands. No parenthesis is present and the expression results in an integer value.

<expr> →<decimal_num>| <var> |<binary_expr>

<binary_expr> → <expr> <operator> <expr>

<expr> → <decimal_num> | <var>

<operator> → + | *

An error occurs in this expression which specifies that a variable contains an undefined value.

Assignment Statements

An assignment statement refers to the evaluation of an expression plus assignment of the resultant value to the left-side variable.

Logical Pretest Loops

The meaning of the loop refers to the value of the program variables that results after executing the loop statements, a predefined number of times without making any errors. An iterative loop is converted to a recursive loop, which is mathematically controlled by the recursive state mapping functions.

Evaluation

1. Denotational semantics can be used to prove the program correctness.
2. It acts as an aid in language designing.
3. It provides a clear and thorough way of thinking about programs.
4. It finds its applications in compiler generation systems.

Q60. In what fundamental way do operational semantics and denotational semantics differ.

Answer : Nov.-15(R13), Q2(a)

1. Operational semantics describes the meaning of a program in language 'L' by specifying how the machine state is affected by the statements. This effect can be simulated or actual. Denotational semantics describes the meaning of a program in terms of mathematical functions applied to programs and program components.
2. The meaning of a statement in operational semantics can be inferred from the changes occurred in the machine's state when a given statement is executed. The state of a machine refers to the values stored in memory, registers, stack or heap.

   The meaning of a statement in denotational semantics can also be inferred from the changes that occur in the state of an ideal computer.
3. The main difference between operational and denotational semantics is that the state changes in operational semantics are defined by the coded algorithms whereas, in denotational semantics they are defined by precise and accurate mathematical functions.

As mentioned earlier the state of a computer in operational semantics refers to the values stored in all its registers, memory locations, stack etc. If this state of a machine is recorded and the instructions are executed then a new machine state is reached, which on examining represents the state changes when the instructions are executed.

In denotational semantics, the state 's' of a program can be expressed as a set of ordered pairs, such as

$$\{<a_1, v_1>, <a_2, v_2>, ..., <a_n, v_n>\}$$

Where,

$a_1, a_2, ..., a_n$ are the variable names and $v_1, v_2, ..., v_n$ are the current values associated with the variables.

Let VMAP be a function that returns the current value of the variables, when a variable name and a state are given i.e.,

$$VMAP\ (a_n, s) = v_n$$

Where,

'v_n' is the value associated with 'a_n' in state s. Mapping between the states can be done by many semantics mapping functions and program constructs. Hence, the meaning of a program and program constructs can be defined by the changes in the system's state.

Q61. Explain the basic concept of axiomatic semantics.

Answer : Nov./Dec.-16(R13), Q3(b)

Axiomatic Semantics

For answer refer Unit-I, Q57, Topic: Axiomatic Semantics.

Predicate calculus is used to describe the constraints particularly the language of axiomatic semantics.

Assertions

Mathematical logic forms the basis for axiomatic semantics. The logical expressions are called 'assertions' or 'predicates'. The assertions that precedes the program statements, defines the constraints imposed on program variables at any instant. Such assertions are called 'preconditions'. The assertions that follows the program statements, defines the new constraints on the variables after executing the statements. Such assertions are called 'post conditions'. These postconditions and preconditions are shown in braces to distinguish them from program statements.

Example

$a = 2 * b + 1 \ \{a > 1\}$.

In the above expression,

$\{a > 1\}$ is the postcondition, one possible precondition for this statement is $\{b > 10\}$.

Weakest Preconditions

These are the least restrictive preconditions that assures the validity of the corresponding postconditions. For the above statement and postcondition, some of the valid preconditions are $\{b > 10\}$, $\{b > 50\}$ and $\{b > 1000\}$ but the weakest precondition for this expression is $\{b > 0\}$.

The correctness proofs can be constructed for the programs in case, if the weakest precondition can be determined from the given postconditions for each statement. The desired result of the program execution is used as a postcondition of the last statement in finding the correctness proof of the program. Then the weakest precondition for the last statement in a program can be computed by using this postcondition. This process is repeated till the beginning of the program is encountered. For some programs, the computation of the weakest precondition is simple and can be done by using an axiom, whereas in other cases it is determined by using an 'inference rule'.

The general form of an inference rule is as follows,

$$\frac{S_1, S_2, \ldots S_n}{S_2}$$

The above rule represents that if $S_1, S_2, \ldots, S_n$ is true, then it is possible to infer the truth of 'S'. The numerator part of an inference rule is called as its antecedent and denominator part is called its consequent.

Axiom is a logical assumption that it is true. Thus, an axiom is an inference rule without an antecedent.

Assignment Statements

Let the general assignment statement, postcondition and precondition be $y = A$, Q and P respectively. These can be defined by an axiom,

$$\boxed{P = Q_{y \to A}}$$

Which implies that the precondition 'P' can be computed as 'Q' along with all the instances of y replaced by A.

Example

If the given assignment statement and postcondition are,

$$\boxed{x = 2 + y - 5 \ \{x < 10\}}$$

The weakest precondition can be computed by substituting $2 + y - 5$ in the assertion $\{x < 10\}$, as follows,

$$2 + y - 5 < 10$$

$$\boxed{y < 13}$$

Therefore, the weakest precondition is $\{y < 13\}$. This assignment axiom is valid, only if no side effects are present.

The axiomatic semantics of a given statement is usually expressed as,

$$\boxed{\{P\}S\{Q\}}$$

Where,

P = Precondition,

Q = Postcondition and

S = Statement

An assignment statement whose precondition and postcondition are given, is regarded as a theorem, which can be proved as follows.

When the assignment axiom is applied to the given assignment statement and postcondition and if it results in the given precondition then the theorem is said to be proved.

Consider the following logical statement with the given precondition and postcondition.

$$\boxed{\{y > 6\}y = y - 2\{y > 0\}}$$

In the given expression, the precondition $\{y > 6\}$ is not similar to the resultant assertion produced by the axiom bit, it is obvious that $\{y > 6\}$ implies $\{y > 5\}$. Hence, an inference rule called the 'rule of consequence' is used, whose general form is given as,

$$\frac{S_1, S_2, S_3, \ldots, S_n}{S} \text{ (or) } \frac{\{P\}S\{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q\}}$$

Which specifies that if $S_1, S_2, S_3, \ldots, S_n$ are true, then the truth value of 'S' can be inferred. When this rule is applied to the given logical statement, it results in,

$$\frac{\{y > 5\}y = y - 2\{y > 0\}, (y > 6) \Rightarrow (y > 5), (y > 0) \Rightarrow (y > 0)}{\{y > 6\}y = y - 2\{y > 0\}}$$

This proves the above theorem.

Sequences

The weakest precondition for a sequence of statements can't be explained by an axiom because the precondition is based on some statements in the sequence.

Let $S1$ and $S2$ be the two statements in a sequence with the preconditions $P1$ and $P2$ and postconditions $P2$ and $P3$ respectively then the inference rule can be expressed as,

$$\frac{\{P1\}S1\{P2\}, \{P2\}S2\{P3\}}{\{P1\}S1; S2\{P3\}}$$

Example

Consider the following sequence and postcondition.

$b = 3 * a + 1;$

$a = b + 5 \quad \{a < 12\}$

The precondition for the last assignment statement is,

$b + 5 < 12$

$$\boxed{b < 7}$$

The precondition for the first assignment statement can be computed as follows.

$3 * a + 1 < 7$

$3 * a < 6$

$$\boxed{a < 2}$$

Selection

The inference rule for selection statement containing 'else' clauses is given as,

$$\frac{\{A \text{ and } P\}S1\{Q\}, \{(\text{not } A) \text{ and } P\}S2\{Q\}}{\{P\}\text{if } A \text{ then } S1 \text{ else } S2\{Q\}}$$

The first logical statement $\{A \text{ and } P\}S1\{Q\}$ is the 'then' clause and the statement $\{(\text{not } A) \text{ and } P\}\ S2\ \{Q\}$ is the 'else' clause.

Logical Pretest Loops

The process of finding the weakest precondition for a 'while' loop is more difficult than for a sequence because of the indefinite number of iterations. If the number of iterations are known, then the loop can be regarded as a sequence. This process is similar to the problem of theorem proving about all positive integers. For this, an assertion called a 'loop invariant' must be determined.

The axiomatic description of a 'while' loop is given as,

$\{P\}$ while A do S end $\{Q\}$.

Evaluation

1. Developing axioms or inference rules for all the statements in a language is difficult. One way to solve this problem is to design the language using axiomatic method in such a way that only those statements for which axioms or inference rules are required to be written are included.
2. It is a good tool for correctness proofs and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers.

Q62. Explain about postconditions and preconditions of a given statement mean in axiomatic semantics.

Answer : (Model Paper-III, Q3(b) | Nov./Dec.-17(R15), Q3(a))

Axiomatic semantics is a method developed to prove the correctness of programs. If the correctness proof, when constructed, specifies that a program computation is according to its specification. In these proofs, a logical expression which represents the constraints on program variables precedes and follows each program statement, specifies the meaning of the statement. The predicate calculus is used to describe the language of axiomatic semantics. The logical expressions are known as 'predicates' or 'assertions'. The program statement follows the assertion which specifies, the constraints on the variables at that instant in a program. Assertions are of two types.

1. **Postcondition**

An assertion which immediately succeeds a program statement is called postcondition.

2. **Precondition**

An assertion which immediately precedes a program statement is called precondition.

Preconditions for statements are computed from postconditions and vice versa. The axiomatic description of a program requires both the precondition and postcondition for every statement of the program.

Consider an example.

$a = b * 2 + 1\ (a > 1)$

The condition in the above assignment statement is a postcondition. In this statement, assertions are analyzed in such a way that precondition for the given statement can be computed from the given postcondition. The variables 'a' and 'b' in the given statement are of integer types.

For the given post condition to be true, the precondition must be $\{b > 10\}$. The preconditions and postconditions are represented braces in order to distinguish them from program statements.

The precondition that is, the least restrictive and assures the validity of the postcondition is called the weakest precondition. The valid possible preconditions $\{b > 10\}, \{b > 50\}$ and $\{b > 100\}$ whereas the weakest precondition is $\{b > 0\}$.

The correctness proofs can be generated for the programs, if the weakest precondition can be determined from the postcondition for each program statement which uses the required result of the program execution as the postcondition of last statement of the program. This postcondition and the last statement is used to find the weakest precondition for the last statement. The weakest precondition thus computed is used as the postcondition for the second last statement, continuing this way, working backwards through the program, the weakest precondition for each statement is computed till the beginning of the program is reached.

At this stage, the first precondition signifies the conditions under which the program will produces the desired result. The weakest pre and post conditions can be specified by an axiom but inference rule is most commonly used.

Axiom

It is a logical statement which is assumed to be true.

Inference Rule

It is a method by which the truth value of one assertion can be inferred from the other.

Example

Consider an expression,

$$p = 3 * q - 4 > 23$$

$$3 * q > 23 + 4$$

$$3 * q > 27$$

$$q > \frac{27}{3}$$

$$q > 9$$

$\therefore \{q > 9\}$ is the weakest precondition for the above assignment statement and postcondition.