Q.1 Functional Programming :-

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e. −

Pure Functional Languages − These types of functional languages support only the functional paradigms. For example − Haskell.

Impure Functional Languages − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Function(--------function(function,(data))-------)

ie. Sum sqr(x, y)

{

(x (\*xx)(\*xy))

}

Functional Programming – Advantages

Functional programming offers the following advantages −

Bugs-Free Code − Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

Efficient Parallel Programming − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.

Efficiency − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.

Supports Nested Functions − Functional programming supports Nested Functions.

Lazy Evaluation − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc

Q.1 Rules Based language:-

Rule-based programs can be written several ways. Some languages, such as OPS-5, are intended for this purpose. There are also expert systems shells of many kinds, prices, and capabilities. Either approach can offer a great deal of power in rule-based programming and should be considered for an expert system or any program that will have a great many rules. However, in some cases these approaches sacrifice the capability of the procedural programming on such prosaic areas as I/O and screen control.

In any case, you are required to learn new ways to do old things. For a program with just a few rules or for experimenting, you might consider writing a rule-based program in C.

In a rule-based program, the function is embodied in a set of rules written in something resembling English. A rule has a condition or set of conditions called the IF part and a result or set of results called the THEN part. The THEN part of the rule usually consists of actions to be performed. If the IF part of the rule is satisfied the rule can fire. If the rule fires, actions in the THEN part are performed

1

2

3

IF   there is rain

AND   you must walk

THEN    umbrella is necessary

In this case the THEN portion doesn't look like an action. It could also be written umbrella = necessary , and in either case could have the effect of assigning the string necessary to the variable umbrella or giving the Boolean umbrella a value of TRUE or whatever your program requires. The IF part could also be rewritten as IF rain is true, which would look more like a normal program.

Enabling condition1-- action1

Enabling condition2-- action2

-

-

Enabling condition-n-- action-n

ie. PROLOG (rule based language)

Q.1 Von Neumann Architecture

Von Neumann Architecture

Von Neumann architecture was first published by John von Neumann in 1945.

His computer architecture design consists of a Control Unit, Arithmetic and Logic Unit (ALU), Memory Unit,Registers and Inputs/Outputs.

Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory.  This design is still used in most computers produced today.

Central Processing Unit (CPU)

The Central Processing Unit (CPU) is the electronic circuit responsible for executing the instructions of a computer program.

It is sometimes referred to as the microprocessor or processor.The CPU contains the ALU, CU and a variety of registers.

Registers

Registers are high speed storage areas in the CPU.  All data must be stored in a register before it can be processed.

MAR

Memory Address Register

Holds the memory location of data that needs to be accessed

MDR

Memory Data Register

Holds data that is being transferred to or from memory

AC

Accumulator

Where intermediate arithmetic and logic results are stored

PC

Program Counter

Contains the address of the next instruction to be executed

CIR

Current Instruction Register

Contains the current instruction during processing

Arithmetic and Logic Unit (ALU)

The ALU allows arithmetic (add, subtract etc) and logic (AND, OR, NOT etc) operations to be carried out.

Control Unit (CU)

The control unit controls the operation of the computer’s ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit.  The control unit also provides the timing and control signals required by other computer components.

Q.2 Lexical

Lexical analysis is the extraction of individual words or lexemes from an input stream of symbols and passing corresponding tokens back to the parser.

 C=a + b \* 60

Lexical Analysis

Scanner

C, a, b = Identifiers

=, +, \* operation

60 constant value

The tokenisation process takes input and then passes this input through a keyword recogniser, an identifier recogniser, a numeric constant recogniser and a string constant recogniser, each being put in to their own output based on disambiguating rules.

The lexical analysis process starts with a definition of what it means to be a token in the language with regular expressions or grammars, then this is translated to an abstract computational model for recognising tokens (a non-deterministic finite state automaton)

Syntactic Analysis

Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this. For syntactic analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing.

In syntactic analysis, parse trees are used to show the structure of the sentence, but they often contain redundant information due to implicit definitions (e.g., an assignment always has an assignment operator in it, so we can imply that), so syntax trees, which are compact representations are used instead. Trees are recursive structures, which complement CFGs nicely, as these are also recursive (unlike regular expressions).

Parse Trees

Parse trees over a grammar G is a labelled tree with a root node labelled with the start symbol (S), and then internal nodes labelled with non-terminals. Leaf nodes are labelled with terminals or ε. If an internal node is labelled with a non-terminal A, and has n children with labels X1, ..., Xn (terminals or non-terminals), then we can say that there is a grammar rule of the form A → X1...Xn. Parse trees also have optional node numbers.

The above parse tree corresponds to a leftmost derivation.

Traversing the tree can be done by three different forms of traversal. In preorder traversal, you visit the root and then do a preorder traversal of each of the children, in in-order traversal, an in-order traversal is done of the left sub-tree, the root is visited, and then an in-order traversal is done of the remaining subtrees. Finally, with postorder traversal, a postorder traversal is done of each of the children and the root visited.

Syntax Trees

Parse trees are often converted into a simplified form known as a syntax tree that eliminates wasteful information from the parse tree.

At this stage, treatment of errors is more difficult than in the scanner (tokeniser), as the scanner may pass problems to the parser (an error token). Error recovery typically isolates the error and continues parsing, and repair can be possible in simple cases. Generating meaningful error messages is important, however this can be difficult as the actual error may be far behind the current input token.

Q.2 Brief history of programming language

Even though a huge number of computer languages exist, the earliest computers were programmed in binary so the set of instructions was just a series of 0 and 1. Computer languages are a fairly new field, since the first high-level languages were written in the 1950s, around the time computers were invented.

 Computer programming languages were predated in turn by paper tapes and punch cards which held complicated weaving patterns for use in mechanized looms. Later, Charles Babbage started building a computer known as the Analytical Machine. In the 20th century, Herman Hollerith invented the Tabulating Machine. His machine tabulators were used to speed up the counting and sorting of punch cards. In the early 1940s J. Presper Eckert and John W. Mauchly started building the ENIAC (Electronic Numerical Integrator and Calculator), which was completed by 1946.

|  |  |
| --- | --- |
| * 1951 – [Regional Assembly Language](https://en.wikipedia.org/w/index.php?title=Regional_Assembly_Language&action=edit&redlink=1)
* 1952 – [Autocode](https://en.wikipedia.org/wiki/Autocode)
* 1954 – [IPL](https://en.wikipedia.org/wiki/Information_Processing_Language) (forerunner to LISP)
* 1955 – [FLOW-MATIC](https://en.wikipedia.org/wiki/FLOW-MATIC) (led to COBOL)
* 1957 – [FORTRAN](https://en.wikipedia.org/wiki/Fortran) (First compiler)
* 1957 – [COMTRAN](https://en.wikipedia.org/wiki/COMTRAN) (precursor to COBOL)
* 1958 – [LISP](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29)
* 1958 – [ALGOL 58](https://en.wikipedia.org/wiki/ALGOL_58)
* 1959 – [FACT](https://en.wikipedia.org/wiki/FACT_computer_language) (forerunner to COBOL)
* 1959 – [COBOL](https://en.wikipedia.org/wiki/COBOL)
 | * 1959 – [RPG](https://en.wikipedia.org/wiki/IBM_RPG)
* 1962 – [APL](https://en.wikipedia.org/wiki/APL_%28programming_language%29)
* 1962 – [Simula](https://en.wikipedia.org/wiki/Simula)
* 1962 – [SNOBOL](https://en.wikipedia.org/wiki/SNOBOL)
* 1963 – [CPL](https://en.wikipedia.org/wiki/Combined_Programming_Language) (forerunner to C)
* 1964 – [Speakeasy (computational environment)](https://en.wikipedia.org/wiki/Speakeasy_%28computational_environment%29)
* 1964 – [BASIC](https://en.wikipedia.org/wiki/BASIC)
* 1964 – [PL/I](https://en.wikipedia.org/wiki/PL/I)
* 1966 – [JOSS](https://en.wikipedia.org/wiki/JOSS)
* 1967 – [BCPL](https://en.wikipedia.org/wiki/BCPL) (forerunner to C)
 |

|  |  |
| --- | --- |
| * 1968 – [Logo](https://en.wikipedia.org/wiki/Logo_%28programming_language%29)
* 1969 – [B](https://en.wikipedia.org/wiki/B_%28programming_language%29) (forerunner to C)
* 1970 – [Pascal](https://en.wikipedia.org/wiki/Pascal_%28programming_language%29)
* 1970 – [Forth](https://en.wikipedia.org/wiki/Forth_%28programming_language%29)
* 1972 – [C](https://en.wikipedia.org/wiki/C_%28programming_language%29)
 | * 1972 – [Smalltalk](https://en.wikipedia.org/wiki/Smalltalk)
* 1972 – [Prolog](https://en.wikipedia.org/wiki/Prolog)
* 1973 – [ML](https://en.wikipedia.org/wiki/ML_%28programming_language%29)
* 1975 – [Scheme](https://en.wikipedia.org/wiki/Scheme_%28programming_language%29)
* 1978 – [SQL](https://en.wikipedia.org/wiki/SQL) (a query language, later extended)
 |
| * 1980 – [C++](https://en.wikipedia.org/wiki/C%2B%2B) (as [C with classes](https://en.wikipedia.org/wiki/C_with_classes), renamed in 1983)
* 1983 – [Ada](https://en.wikipedia.org/wiki/Ada_%28programming_language%29)
* 1984 – [Common Lisp](https://en.wikipedia.org/wiki/Common_Lisp)
* 1984 – [MATLAB](https://en.wikipedia.org/wiki/MATLAB)
* 1984 - dBase III, dBase III Plus (Clipper and [FoxPro](https://en.wikipedia.org/wiki/FoxPro) as [FoxBASE](https://en.wikipedia.org/wiki/FoxBASE), later developing into [Visual FoxPro](https://en.wikipedia.org/wiki/Visual_FoxPro)
* 1985 – [Eiffel](https://en.wikipedia.org/wiki/Eiffel_%28programming_language%29)
* 1986 – [Objective-C](https://en.wikipedia.org/wiki/Objective-C)
 | * 1986 – [LabVIEW](https://en.wikipedia.org/wiki/LabVIEW) (Visual Programming Language)
* 1986 – [Erlang](https://en.wikipedia.org/wiki/Erlang_%28programming_language%29)
* 1987 – [Perl](https://en.wikipedia.org/wiki/Perl)
* 1988 – [Tcl](https://en.wikipedia.org/wiki/Tcl)
* 1988 – [Wolfram Language](https://en.wikipedia.org/wiki/Wolfram_Language) (as part of [Mathematica](https://en.wikipedia.org/wiki/Mathematica), only got a separate name in June 2013)
* 1989 – [FL](https://en.wikipedia.org/wiki/FL_%28programming_language%29) (Backus)
 |
| * 1990 – [Haskell](https://en.wikipedia.org/wiki/Haskell_%28programming_language%29)
* 1991 – [Python](https://en.wikipedia.org/wiki/Python_%28programming_language%29)
* 1991 – [Visual Basic](https://en.wikipedia.org/wiki/Visual_Basic)
* 1993 – [Lua](https://en.wikipedia.org/wiki/Lua_%28programming_language%29)
* 1993 – [R](https://en.wikipedia.org/wiki/R_%28programming_language%29)
* 1994 – [CLOS](https://en.wikipedia.org/wiki/CLOS) (part of ANSI [Common Lisp](https://en.wikipedia.org/wiki/Common_Lisp))
 | * 1995 – [Ruby](https://en.wikipedia.org/wiki/Ruby_%28programming_language%29)
* 1995 – [Ada 95](https://en.wikipedia.org/wiki/Ada_95)
* 1995 – [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29)
* 1995 – [Delphi (Object Pascal)](https://en.wikipedia.org/wiki/Embarcadero_Delphi)
* 1995 – [JavaScript](https://en.wikipedia.org/wiki/JavaScript)
* 1995 – [PHP](https://en.wikipedia.org/wiki/PHP)
* 1997 – [Rebol](https://en.wikipedia.org/wiki/REBOL)
 |

Q.3 Language translator

A **translator** or **programming language** processor is a computer program that performs the **translation** of a program written in a given **programming language**into a functionally equivalent program in another computer **language** (the target **language**), without losing the functional or logical structure of the original code.

Machine language

Binary numbers

 Translator

Compiler, Assembler

High level language

C, C++, Java

**Compilers**Compilers convert high-level language code to machine (object) code in one session. Compilers can take a while, because they have to translate high-level code to lower-level machine language all at once and then save the executable object code to memory. A compiler creates machine code that runs on a processor with a specific Instruction Set Architecture (ISA), which is processor-dependent. For example, you cannot compile code for an x86 and run it on a MIPS architecture without a special compiler. Compilers are also platform-dependent. That is, a compiler can convert C++, for example, to machine code that’s targeted at a platform that is running the Linux OS. A cross-compiler, however, can generate code for a platform other than the one it runs on itself. A cross-compiler running on a Windows machine, for instance, could generate code that runs on a specific Windows operating system or a Linux (operating system) platform. Source-to-source compilers translate one program, or code, to another of a different language (e.g., from Java to C). Choosing a compiler then, means that first you need to know the ISA, operating system, and the programming language that you plan to use. Compilers often come as a package with other tools, and each processor manufacturer will have at least one compiler or a package of software development tools (that includes a compiler). Often the software tools (including compiler) are free; after all, a CPU is completely useless without software to run on it. Compilers will report errors after compiling has finished.

**Interpreters**another way to get code to run on your processor is to use an interpreter, which is not the same as a compiler. An interpreter translates code like a compiler but reads the code and immediately executes on that code, and therefore is initially faster than a compiler. Thus, interpreters are often used in software development tools as debugging tools, as they can execute a single in of code at a time. Compilers translate code all at once and the processor then executes upon the machine language that the compiler produced. If changes are made to the code after compilation, the changed code will need to be compiled and added to the compiled code (or perhaps the entire program will need to be re-compiled.) But an interpreter, although skipping the step of compilation of the entire program to start, is much slower to execute than the same program that’s been completely compiled. Interpreters, however, have usefulness in areas where speed doesn’t matter (e.g., debugging and training) and it is possible to take the entire interpreter and use it on another ISA, which makes it more portable than a compiler when working between hardware architectures. There are several types of interpreters: the syntax-directed interpreter (i.e., the Abstract Syntax Tree (AST) interpreter), bytecode interpreter, and threaded interpreter (not to be confused with concurrent processing threads), Just-in-Time (a kind of hybrid interpreter/compiler), and a few others. Instructions on how to build an interpreter can be found on the web.[[i]](https://www.microcontrollertips.com/compilers-translators-interpreters-assemblers/%22%20%5Cl%20%22_edn1) Some examples of programming languages that use interpreters are Python, Ruby, Perl, and PHP.

**Assemblers**An assembler translates a program written in assembly language into machine language and is effectively a compiler for the assembly language, but can also be used interactively like an interpreter. Assembly language is a low-level programming language. Low-level programming languages are less like human language in that they are more difficult to understand at a glance; you have to study assembly code carefully in order to follow the intent of execution and in most cases, assembly code has many more lines of code to represent the same functions being executed as a higher-level language. An assembler converts assembly language code into machine code (also known as object code), an even lower-level language that the processor can directly understand. Assembly language code is more often used with 8-bit processors and becomes increasingly unwieldy as the processor’s instruction set path becomes wider (e.g., 16-bit, 32-bit, and 64-bit). It is not impossible for people to read machine code, the strings of ones and zeros that digital devices (including processors) use to communicate, but it’s likely only read by people in cases of computer forensics or brute-force hacking. Assembly language is the next level up from machine code, and is quite useful in extreme cases of debugging code to determine exactly what’s going on in a problematic execution, for instance. Sometimes compilers will “optimize” code in unforeseen ways that affect outcomes to the bafflement of the developer or programmer such that it’s necessary to carefully follow the step-by-step action of the processor in assembly code, much like a hunter tracking prey or a detective following clues.

Q.3 **.**Differentiate between variable and constant

The difference between variable and constant is explained here.

|  |  |
| --- | --- |
| **Constant****The symbol which has a fixed value is called as constant****Example : 2, 3, 2/3, 0.57, -2, -3** | **Variable****The symbol which has no fixed value is called as variable.****Example : x, y, z** |

Let us consider some problems on simple equations to have better understanding on "Variables".

**Problem 1 :**

Solve for "x" :  3x + 6 = 18

3x + 6 = 18 -------> 3x = 12 -------> x = 4

**Problem 2 :**

Solve for "x" :  x - 2 = 0

x - 2 = 0  -------> x = 2

**Problem 3 :**

Solve for "x" :  9x = 27

9x = 27 -------> x = 3

Problem 4 :

Solve for "x" :  x + 3 = -2

x + 3 = -2 -------> x = -2 -3 -------> x = -5

In all the above problems, we have got value for the same alphabet "x".

Even though we have the same alphabet "x", we don't get the same value for "x".

We get different value for "x". So the value of "x" is being changed.

Hence, "x" is considered to be variable.

Some real life examples

Let us consider some real life example on "Difference between constant and variable"

|  |  |
| --- | --- |
| **Constant****Value of $100 currency** | **Variable****Cost of a garment** |

**Constant :**

Let us consider $100 currency.

What is the value of $100 currency when i have it in Newyork?

Its value is value $100

What is the value of $100 currency when i have it in Sancfrancisco?

Its value is value $100

What is the value of $100 currency when i have it in Chicago?

Its value is value $100

From the above questions and answers, one thing is very clear. That is,. in whichever part of America i go, the value of the currency $100 is same.  It is not changed in different parts of the country. So $100 is constant.

**Variable :**

Let us consider a garment whose cost is  "x" dollars.

The manufacturer of the garment sells it to the wholesaler at the cost of $40.

So, x = $40

The wholesaler sells it to the retailer at the cost of $44.

So x = $44

The retailer sells it to the customer at the cost of $50.

So x = $50

The garment is same. But, when it is in manufacturer, wholesaler and retailer, its cost is different.

Hence, the cost of the garment "x" is variable.

Arbitrary constant

Some English alphabets will have fixed values, but the values will be unknown.

**For  example**, Let "k" be arbitrary constant.

Then, "k" will have only one fixed value and it is unknown.

Q.4 Operation of Computer in programming language

 Basic **Computer Operations**. Input: Information and programs are entered into the**computer** through Input devices such as the keyboard, disks, or through other**computers** via network connections or modems connected to the Internet. The input device also retrieves information off disks.

**Input**: Information and programs are entered into the computer through Input devices such as the keyboard, disks, or through other computers via network connections or modems connected to the Internet. The input device also retrieves information off disks.

**Output**: Output Devices displays information on the screen (monitor) or the printer and sends information to other computers. They also display messages about what errors may have occurred and brings up message or dialog box asking for more information to be input. The output device also saves information on the disk for future use.

**Processing**: The CPU or central processing unit is sometimes called the Control Unit and directs the operation of the input and output devices. The Coprocessor or the Arithmetic-Logic Unit does arithmetic and comparisons. The memory or RAM temporarily stores information (files and programs) while you are using or working on them. The BIOS or basic input/output system controls the dialogue between the various devices. Keyboard Layout and Data Entry.



Storage: Computers would not be as useful as they are if they were unable to remember anything. The fourth basic operation, storage, allows the computer to recall previously entered data and store information. This includes documents, music, log files, software, and the operating system (OS).

The most common storage medium is the hard drive. Memory cards, DVD-ROMs, and floppy disks are other examples.

### Control Unit

 ALU dose does not know what should be done with the data likewise, output unit dose not know when the result should be displayed. By selecting, interning and seeing to the execution of the program the CU is able to maintain order and direct the operations of the entire system. CU doesn't perform any actual processing on data yet it is known as a central nervous system for the comforts of the computer. It manages and coordinates the entire system. Hope this discussion helps you and makes you select correct answers in your exam. Following questions are probable from this topic. Following are 10 MCQ Questions from Basic Operations of a Computer. Answer them in the comment section below.