**What is a *programming language*?**

**M**any new hundreds of languages are being designed and developed, now a day, which are used for a period of Time and then allowed to fall into disuse.

There are many reasons behind success or failure of a language, sometimes the reason may be external to the Language itself. **For Example:** use of *COBOL or Ada* in US was forced in certain areas of Programming development industries by Government mandate. So extensive Usage of a particular language by a large amount of people leads to Greater Success and most of the developments are done in the same Language preferably. Likewise there is reason for success of *FORTRAN*, because most of the computer manufacturers strongly supported this language. Many more languages are there which are successful due to many external reasons. Though many of them are failures in history, and to indicate their failure level most of us don’t know the name of languages which have failed to make them self on the long list!!!

A programming language is the collection of syntactic rules, keywords, naming structures, data structures, expression and control structures that is intended for the expression of computer programs and that is capable of expressing any computer program. Programming Language is a formal method that:

* Describe a solution to a problem
* Organize a solution to a problem
* Reason about a solution to a problem
* Interface between user and machine

Programming languages trade-off:

* Ease of use - high level
* Efficiency - low level

“A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program.”

**Importance of the study of Programming languages:**

Programming languages are important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming. As a result, the progress of computer science can be traced in the progress of programming languages, and many issues of computer science manifest themselves as programming language issues. The reason is simple: Programming languages remain the central tool for problem solving in computer science.

**-**Important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming.

- The progress of computer science can be traced in the progress of programming languages, and many issues of computer science can be traced in the progress of programming language issues.

-Programming language remains the central tool for problem solving in computer science.

Some importance of knowing evolution of programming language is highlighted below:

**1) Influence of languages on problem solving:**

The Sapir-Whorf hypothesis (still controversial): The structure of language defines the boundaries of thought. Although no programming language can prevent us from finding certain solution to a problem, a given language can influence the class of solution we are likely to see and the frame of mind with which we approach programming, thus influencing the quality of our programs.

**2) Benefits for all computer scientists**

The study of programing language is important to all who uses them. The reason is that from this study you will learn the motivation for and the use of the most important facilities found in modern programming languages. This will provide you with a basis for evaluating languages, which will aid you in choosing the best language for your application.

**3) Benefits for language designers**

All engineering design is a cumulative process; we learn from the success and failures of the designs of the past.

**4) Benefits for language implementers**

If you are interested in language implementations, you will gain insight into the motivations for various facilities, thus allowing you to make reasonable implementation trade-offs.

**5) Benefits for hardware architects**

By understanding the requirements of programming language implementation, hardware architects will gain insight into the ways machines may better support languages.

**6) Benefits for system designers**

Designers of all sorts of s/w system (e.g. OS & DBMS) will learn principles and techniques applicable to all human interfaces. Many s/w tolls including OS command languages, DBMS, editors, text formatters and debuggers, have many of the characteristics of programming languages, and so the principles will be applicable to much of your future s/w design.

**7) Benefits for software managers**

Finally if you manage s/w development efforts, then you will benefit in several ways from the study of programming languages. You will be better able to make the decisions if you know the costs of designing or extending a language, the costs of implementing a language, and the benefits of various language facilities.

**Attributes of a Good Programming Language**

**1. Clarity ,Simplicity and Unity**

A programming Language provides a medium to conceptual thinking of new algorithms and also a medium to Execute your thought Process into real Coding Statements. For algorithms to be implemented on a Language it’s a basic need is that the language is quite *clear, simple and Unified* in structure. Such that the Primitives of language can be utilized to develop algorithms. It is desirable to have a minimum number of different concepts, so that combining multiple concepts won’t be that complex in nature. It should be simple and regular as possible. This attribute of a language is known as conceptual Integrity.

The main concern of a language now a days is its readability. The syntax of language effects the ease with which programs are written, tested and later used for knowledge or research purpose. A complex syntax language may be easy to write program in, but it proves to be difficult to read and debug the code for later sessions. **For example** APL programs are so complicated that even the own developers find it difficult to understand after 1-2 months. The language should be simple enough to understand or point out errors.

**2. Orthogonality**

The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. Language design must follow orthogonality principle i.e. independent functions should be controlled by independent mechanisms **For example,** suppose a language provides with an expression let’s say an arithmetical calculation operator .Taking Another Expression facilitated by the language like conditional Statement, which has 2 outputs either 0 or 1 (in some cases TRUE or FALSE). Now the language should support combination of these two expressions. So that new statement can be formed, and this orthogonality helps to develop many new algorithms.

**3. Naturalness for the application**

A language needs a Syntax that, when applied properly, allows the program Structure to reflect the logical structure what a programmer wanted it to. *Arithmetic Algorithms ,concurrent algorithms , logic Algorithms* and other type of statement have differing natural structures , that can be represented by the Program in that language. The language should provide appropriate *data structures, operations, control structures and a natural syntax* for the problem to be solved.

**For Example:** Consider a real life condition of plates being placed above each plate, this structure is known as Stack. This Stack can be implemented into programming world also. This is used as a data Structure in most of the Languages.

**4. Support for Abstraction**

Many times languages fail to implement many real life problems into Programs. There is always a gap between abstract data structures and operations. Even most natural Programming language fails to bridge the gap. **For Example:** Consider a situation where a scheduling is to be done for college *student for attending a lecture in a class section, teacher.* Suppose the requirement is to assign a student a section lecture and teacher to attend, which are common task for natural application, but are not provided by C.

The need of point is to design an appropriate abstraction for the problems solution and then implementing these abstraction using most primitive features of a language. Ideally, the language should provide the data structures, data types and operations to maintain such abstractions .C++ is one of the most used language, that provide such facilities.

**5. Ease of Program Verification**

The reliability of a programming Language written in a language is always a central Concern. There are many techniques which can be used to keep track of correct functionality of a language. Sometimes testing the Program with random values of the inputs and obtaining corresponding outputs. Program verification should be provided by languages to check and minimize the errors.

**6. Programming Environment**

The environment also plays a vital role in success of a Language. The environment which is technically weak, may get a bad response of Programmer, rather than a language that has less facility than the former but its environment is Technically Good. Some of the Good featured of an environment are Special editors and testing packages tailored to the language may greatly speed up the creation and testing of Programs.

**7. Portability of Programs**

The important criterion for many programming projects is the Transportability of the resulting program from one computer to another systems. A language which is widely available and do not support different features on different computer System, which may have different hardware, is considered a good language. **For Example** *C, C++* and most of the language now days are Portable in nature.

**8. Cost of Use**

The trickiest point that always matter a lot in any system that uses resources. It’s a major element to decide the Evaluation of any programming language, but cost means many different things.

**(a) Cost of Program Execution**

Program Execution cost is total amount which has been used to implement the program. The research work on design, optimizing compilers, data allocation registers etc. These are the basic things which come under the cost of Program Execution.

**(b) Cost of Program Translation**

The next concern is program compilation. The program is compiled many times than it is being executed. In such case, it is important to have a speed and efficient compiler to handle this Job.

**(c) Cost of Program Creation, Testing and Use**

Another aspect of Cost management. This includes the cost which a programmer charges for his work of creating Project with the Specified features, the cost involving the Testing issues.

**(d) Cost of Program Maintenance**

After a program is being installed in a System, then after certain intervals it needs maintenance to run smoothly. The maintenance includes the rectification of Error propagated in real time, the updating of Program as need of time.

**History and Motivation**

**Why Study a Primitive Language?**

A primitive programming language is so primitive, in fact that you would never want to program in it. Furthermore, it’s not even a “real” language. We may benefit by beginning the study of language design in a context in which the issues are salient. Having honed our skills here, we will be better able to apply them to more modern, sophisticated, and complex language.

* **Programming is difficult**

Almost as soon as the first computers were built, it become obvious that programming was very difficult; this fact has not changed. Much of the difficulty of programming stems from complexity, the necessity of dealing with many different details at one time. Programs contain hundreds of thousands, or even millions, of lines of code. Considering these lines(together with the operators and operands that make them up) as parts that must be assembled individually to make a program work correctly leads to the conclusion that programs are some of the most complicated objects ever constructed.

* **Programming in early computers was especially difficult**

Although the problems addressed on early computers were smaller than many of those now addressed, programming was still very difficult. Part of the reason was that early computers had very little storage, a few thousand word were considered a large memory. Thus, compact code was a necessity.

* **Many program design notations were developed**

The complexity of programming led to the development of program design notations, the precursors of programming languages. One of the earliest of these was Von Neumann and Goldstine’s flow diagrams, which developed into the flowcharts that are still often used during program design. Throughout the world, many different notations and languages were developed to try to conquer the complexity of programming. Also mnemonics for the machine operations, muck like an assembly language.

* **Floating point and indexing were simulated**

The earliest computers did not have built-in floating-point operations; these did not appear until the IBM 704 in 1953.On the other hand, the primary application of many of these machines was in scientific and numerical computations, which require numbers of a wide range of magnitude. This necessitated manual scaling, a technique in which numbers were multiplied by scale factors in order to keep them within the range of the integer arithmetic facilities of the computer. This was a very complicated process, which required a detailed analysis of the algorithm. The difficulty of manual scaling led to the development of floating point subroutines, that is, of subroutines for performing basic floating-point operations (addition, subtraction, multiplication, division, square root, etc).

Another facility missing from many early computers was indexing, the ability to add a variable index quantity to a fixed address in order to address an element of any array. So, most of the actual execution time was spent inside the floating point and indexing subroutines.

* **Pseudo-Code Interpreters were invented**

Since the program was going to spend most of its time in the floating-point and indexing subroutines anyway, why not simplify programming by providing an entire new instruction code that was easier to use than the machine’s own. i.e pseudo-code, that is, an instruction code that is different from that provided by the machine-and presumably better.

The significance of pseudo code interpreter is that they implemented a virtual computer with its own set of data types (eg.Floating point) and operations (eg. indexing) in terms of real computer with its own set of data types and operations.

* **Compiling Routines were also used**

At this same time, another approach was being used for implementing pseudo-codes—*compiling routines*. Grace Murray and others began to use programs to extract subroutines from libraries and combine them (a process called compiling) under the direction of a pseudo code. Since this process was done once, at compilation time, it did not involve the overhead that resulted from interpretation. Therefore, the compiled program could run considerably faster than an interpreted program.

**THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES**

There are many languages that people use to control and interact with computers. These can all be referred to as computer languages. Many of these languages are used for special purposes. e.g., for editing text, conducting transactions with a bank, or generating reports. These special-purpose languages are not programming languages because they cannot be used for general programming. We reserve the term *programming language* for a computer language that can be used, at least in principle, to express any computer program.

**Definition:**

A **programming language** is a [formal language](https://en.wikipedia.org/wiki/Formal_language) that [specifies](https://en.wikipedia.org/wiki/Programming_language_specification) a [set of instructions](https://en.wikipedia.org/wiki/Instruction_set) that can be used to produce various kinds of [output](https://en.wikipedia.org/wiki/Input/output). Programming languages generally consist of [instructions](https://en.wikipedia.org/wiki/Machine_instruction) for a [computer](https://en.wikipedia.org/wiki/Computer). Programming languages can be used to create [programs](https://en.wikipedia.org/wiki/Program_(machine)) that implement specific [algorithms](https://en.wikipedia.org/wiki/Algorithm).

-*A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program.*

The choice of the first programming language and the corresponding programming paradigm is critical for later development of a programmer. Despite the huge number of programming languages introduced over the last fifty years, the key issues in programming education remain the same and choosing appropriate first programming language is still challenging.

In the modern society, relying on information technologies, programming education is extremely important. It is clear that the choice of the first programming language and the corresponding programming paradigm is critical for later development of an IT professional. Over the last fifty years, there was thousands of programming languages introduced, belonging to several programming paradigms. However, despite the big number of programming languages, there are just a few truly important programming concepts and there are not many languages that survived for more than ten years. It is very important to detect what are suitable features of a programming language, especially in the context of education. It is important to consider these issues both in terms of individual programming languages and in terms of programming paradigms. Over the last decades, several programming paradigms emerged and profiled. The most important ones are: imperative, object-oriented, functional, and logic paradigm.

**Challenges of Programming Education**

Acquiring and developing knowledge about programming are a highly complex process. Novice programmers have to overcome a wide range of difficulties. Programming courses are regarded as difficult, and often have very low passing rates. In this section we discuss some of the goals and problems of programming education.

**Goals, Objectives, and Outcomes**: There are five overlapping domains that students should acquire in an introductory course:

• **General orientation** — the capabilities and applications of programs;

• **The notional machine** — an abstract model of the computer used for executing programs;

• **Notation** — the syntax and semantics of a particular programming language;

• **Structures** — the structuring of basic operations into schemas and plans;

• **Pragmatics** — the skills of planning, developing, testing, debugging, documenting, etc.

None of these issues are entirely separable from the others. That is the main source of difficulties for students since they attempt to overcome all these different kinds of difficulties at once .With respect to the above domains, goals and objectives of an introductory programming course can be summarized as follows:

• **Main goals:**

– Become familiar with the fundamental concepts of computer science.

– Develop proficiency in an engineering problem solving and design methodology.

– Understand the importance of advanced information technologies.

**• Main objectives:**

– Use computers and application software as tools to solve problems.

– Analyze, design, build and test operational solutions.

– Acquire the foundation of algorithmic processes.

– Learn to exploit the educational and professional resources available on the Internet and World Wide Web.

– Develop a framework for considering the ethical implications of advanced information technology.

While goals and objectives of any particular programming course can be stated clearly, the desired characteristics of the resulting programmer are still a bit fuzzy. It is generally accepted that it takes about ten years of experience to turn a novice into an expert programmer, but there seems to be very few metrics for what constitutes a good programmer. Some of the relevant questions are:

**• Is it more important to write efficient code or readable code?**

**• Is it more important to have a clumsy bug-free code or to have an elegant algorithm?**

**• Does it matter if students’ programming is not entirely pure, as long as it meets all the functional requirements for the task?**

In order to answer these questions, it is first necessary to define terms such as “readable” and “elegant”, which have never had clearly defined operational definitions. With or without these nontrivial definitions, it is still very difficult to give precise answers that can guide teachers through the process of teaching programming languages.

**Programming Paradigms**

In computer science, several programming paradigms can be recognized. Moreover, the four main problem-solving approaches, i.e., programming paradigms, are recognized as fundamental. Each of these approaches involves a distinct way of thinking and each is supported by a range of programming languages. These paradigms are:

• Imperative paradigm,

• Object-oriented paradigm,

• Functional paradigm,

• Logical paradigm.

**Imperative Programming**

*In*[*computer science*](https://en.wikipedia.org/wiki/Computer_science)*,****imperative programming****is a*[*programming paradigm*](https://en.wikipedia.org/wiki/Programming_paradigm)*that uses*[*statements*](https://en.wikipedia.org/wiki/Statement_(computer_science))*that change a program's*[*state*](https://en.wikipedia.org/wiki/State_(computer_science))*.*

The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940’s. Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory. This is reflected in the design of the imperative languages, with

• States — representing memory cells with changing values,

• Sequential orders — reflecting the single sequential CPU, and

• Assignment statements — reflecting piping.

Imperative programs are sequences of directions (or orders) for performing an action. Therefore, imperative programming is characterized by programming with states and commands which modify these states. Imperative programming languages provide a variety of commands in order to structure the code and to manipulate the states. Usually, in imperative programming languages, a sequence of commands can be named and the name can be used to invoke the sequence of commands. Named sequence of commands is called subprogram, procedure or function. When imperative programming is combined with subprograms it is called procedural programming.

Imperative paradigm is supported by languages such as FORTRAN (introduced in 1954), Cobol (1959), Pascal (1970), C (1971), and Ada (1979), . . .

**Object-Oriented Programming**:

The object-oriented programming is a generalization of imperative programming. The conceptual model of this paradigm is developed from simulation of events. The main underlying idea of this model is: the structure of the simulation should reflect the environment that is being simulated. If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations. As in the real world, objects interact—so, object-oriented programming uses message passing to capture interactions between objects.

A programming language supporting this concept and using objects is called object-based. Object-oriented programming languages support additional features, with the following most important ones:

• Abstract data type definitions are used to define properties of classes of objects;

• Inheritance is a mechanism that allows definition of one abstract data type by deriving it from an existing abstract data type—the newly defined type inherits the properties of the parent type;

• Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes;

• Dynamic binding of function calls supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

Object-oriented paradigm is supported by languages such as Smalltalk (1969), C++ (1983), and Java (1995).

**Functional Programming**:

The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calculus. It allows the programmer to think about the problem at a higher level of abstraction—it encourages thinking about the nature of the problem rather than about sequential nature of the underlying computing engine. Functional languages are motivated and developed by the following questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

A functional programming language usually has three main sets of components:

• data objects — such as a list or an array;

• built-in functions — for manipulating the basic data objects;

• functional forms — also called high-order functions, for building new functions (such as composition and reduction).

Functional programming languages are called applicative since the functions are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.

Functional paradigm is supported by languages such as LISP (1958), ML (1973), Scheme (1975), Miranda (1982), and Haskell (1987).

**Logic Programming**

The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications. The problem description is used by an inference engine to find a solution. More precisely, a logic program consists of:

• axioms — defining facts about objects,

• rules — defining ways for inferencing new facts,

• a goal statement — defining a theorem, potentially provable by given axioms and rules.

Logic programming is characterized by programming with relations and inference. The programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computational resources than procedural and object-oriented languages. Logic paradigm is supported by languages such as Prolog (1970), and G¨odel (1994). Curry (1997) is a multiparadigm programming language merging elements of functional and logic programming.

**Program DESIGN Notations**

Complexity in programming and understanding programs led to development of *program design notations.* These were designed to help the programmer, not to be interpreted by computers. Some of these notations include:

Memory layout

Control flow

•  *Flow Diagrams*

•  Later: Flowcharts

Mnemonics

• Helps to remember instruction codes like assembly language today

**SHORTCOMING OF EARLIER COMPUTER HARDWARE**

**a) Floating Point Arithmetic**

Floating point are harder to support by a hardware due to its complex representation. The  earliest built-in floating point support was provided by IBM 704. Before that, it had to be *simulated using following steps:*

* Manual scaling
* Multiply by constant factor
* Use integer processor
* Manually scale back result
* Complicated and error-prone process

**b) Indexing**

Array is one of most common data structures used in the computer programming, useful for various computations. Array uses a technique called indexing to store and retrieve data from consecutive memory locations. Indexing is like “Adding a variable index quantity to a fixed address in order to access the element of an array”. Indexing was not supported by early computers. They used address modification by:

* Altering the program’s own data accessing instruction
* Computing actual address from pointer and offset, then write into instruction’s data address portion

However it was very error prone process.

**Program DESIGN Notations**

Complexity in programming and understanding programs led to development of *program design notations.* These were designed to help the programmer, not to be interpreted by computers.

Some of these notations helped the programmer to design the:

Memory layout

Control flow

•  *Flow Diagrams*

•  Later: Flowcharts

Mnemonics

• Helps to remember instruction codes like assembly language today

**SHORTCOMING OF EARLIER COMPUTER HARDWARE**

**a) Floating Point Arithmetic**

Floating point is harder to support by hardware due to its complex representation. The  earliest built-in floating point support was provided by IBM 704 in 1953. Before that, it had to be *simulated using following steps:*

* Manual scaling, a technique in which numbers were multiplied by scale factor in order to keep them within the range of the integer arithmetic facilities of the computer
* The difficulty of manual scaling led to the development of floating-point subroutines, that is, of subroutines for performing basic floating-point operations(addition, subtraction, multiplication, division, square root, etc)
* Multiply by constant factor
* Use integer processor
* Manually scale back result
* Complicated and error-prone process

**b) Indexing**

Array is one of most common data structures used in the computer programming, useful for various computations. Array uses a technique called indexing to store and retrieve data from consecutive memory locations. Indexing is like “Adding a variable index quantity to a fixed address in order to access the element of an array”. Indexing was not supported by early computers. They used address modification by:

* Altering the program’s own data accessing instruction
* It also consumed much of the scarce memory with this address modification code
* For this reason it was common to use subroutines to perform indexing
* Most of the actual execution time was spent inside the floating-point and indexing subroutines.

However it was very error prone process.

This justified the use of pseudo-code interpreters, which we will discuss next.

**PseudoCode**

It is an instruction code that is different than that provided by the real machine. Pseudo-code offered floating point support and indexing which was actually not supported by earlier generation computers. It provided entirely new instruction set not offered by the real hardware.

**Need of PseudoCode**

During first generation of computer, programming was very difficult as the programmer need to know about the hardware specification of every machine that the program was intended for. For example in 1950’s for IBM 650 which has following characteristics:

* No programming language was available (not even assembler)
* Memory was only a few thousand words.
* Stored program and data on rotating drum.
* Instructions included address of next instruction so that rotating drum was under next instruction to execute and no full rotations were wasted.

**PseudoCode Interpreters**

Pseudo-Code Interpreter is an interpretive subroutine (The **subroutine** is an important part of any **computer** system's **architecture**. A **subroutine** is a group of instructions that usually performs one task, it is a reusable section of the software that is stored in memory once, but used as often as necessary.) developed to run the pseudocode. They are used for saving memory since the pseudocode is more compact than machines real program code. It implements a virtual computer which allows us to use functionality with its own data types (e.g. floating point) and operations (e.g. indexing) not provided by the actual hardware in which the virtual machine resides but abstracted by the virtual machine. It has own data types and operations and we can view all programming languages this way. The pseudocode was at the higher level and provided facilities more suitable to applications and it eliminated many details from programing. In general it is an example of “Automation Principle” of programming language.

Pseudo-Code interpreters were commonly used to perform floating-point operations and indexing. Consistent use of these simplified the programming process and this simulated instructions not provided by the hardware.

Pseudo-Code interpreter (a primitive, interpreted programming language) implements:

* A virtual computer
* New instruction set
* New data structures

Virtual computer:

* Higher level than actual hardware
* Provides facilities more suitable to applications
* Abstracts away hardware details

PseudoCode follows two basic *Principles of Programming*

***The Automation Principle***

Automate mechanical, tedious, or error prone activities.

***The Regularity Principle***

Regular rules, without exceptions, are easier to learn, use, describe, and implement.

**Design of a Pseudo-Code**

The design of PseudoCode is based on the capabilities and constraints of the first generation computers. In 1950 Capabilities expected by the programmers and not support by the hardware at that time are:

* Floating point operation support (+,-,\*,/,…)
* Comparisons (=,≠,<,≤,>,≥)
* Indexing
* Transfer of control
* Input/output

**Hardware Assumptions**

The IBM 650 will serve as the hardware

* 1 word: 10 decimal digits + 1 sign
* 2000 byte memory

•  1000 for data

•  1000 for program

***The design of pseudocode must follow impossible error principle because “making errors impossible to commit is preferable to detecting them after their commission”.***E.g.: Cannot modify the program accidentally, since memory modifying operations are for “data memory” only.

**Language Design**

1 word can be enough to specify a 3- operand instruction. Operation contains a sign (+, –, etc.) and 1 digit. It supports 20 operations. Three 3-digit operands each accesses memory locations in data area.

***Language design should be orthogonal which means:***

* Operations should be more intuitive than machine code
* Use the *sign* to get more orthogonality

***Language design must follow orthogonality principle i.e. independent functions should be controlled by independent mechanisms.***

***Instruction format:***

**Operand src1 src2 dst**

E.g.: x + y 🡪 z : +1 010 150 200

This instruction instructs to “Add values at location 010 and 150, and save it to location 200”

To follow Orthogonal design the subtract operation should be ‘-1’

**Arithmetic Operations**

**Comparisons**

Comparisons alter control flow of the program

If *x* = *y* then go to *z*

For example, the instruction

+4 200 201 035

Means: if the contents of (data) location 200 equal the contents of (data) location 201, then go to the instruction in (program) location 035.

**Moving**

Data could be moved by “adding 0” to an address, but that operation is expensive so the new operation should be assigned to move data.

+ 0 150 000 200 (First operand is source, third operand is destination while second operand is not used)

**Indexing**

One of the justifications for our pseudo-code was that it provided built-in indexing, so we will turn to the design of this facility next. To perform indexing we will need the address of the array and the index variable, thus consuming two of the three address fields in the instruction. Therefore, the only operations we can perform directly on array elements are to move them to or from other locations. We can use the codes +6 and -6 to move from or to an array: xi  z and x yi.

The formats of these operations are

*+6 xxx iii zzz*

*-6 xxx yyy iii*

E.g. if there is a 100-element array beginning at location 250 in data memory, and location 050 contains 17, then

+6 250 050 803

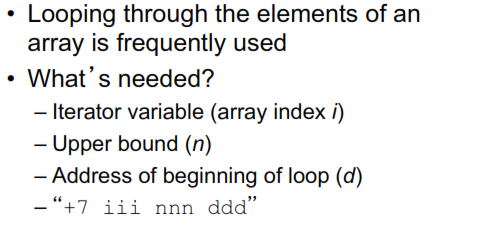
will move the contents of location 267(=250+17) to location 803.

Similarly,

-6 722 250 050

will move the contents of location 722 to location 267.

**Looping**



The operation increments location iii and loops to instruction ddd if the result is less than the contents of nnn.

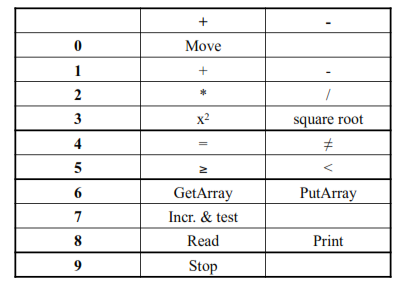
**Input/output**

Program needs to read data from input and write data to output.

* Needs only a memory location to read from or write to
* Read: “+8 000 000 dst”
* Print: “-8 000 000 src”

***Language design must follow the abstraction principle i.e. avoid requiring something to be stated more than once; factor out the recurring pattern.***

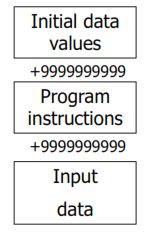
**Complete Pseudo-Code operations**



**Program Structure**

We now know how to write individual instructions, but we have not designed a means of constructing the program as a whole. The simplest solution to this problem is to have interpreter read initialization cards and their content in consecutive memory locations.

Thus, the structure of a program is:



We have used a card containing the “flag value” +9999999999 to separate the initial value from the program and the program from the input data. The loader reads in the initial data values and stores them in consecutive locations (starting with 000) in the data memory. Then the loader reads in the program instructions and stores them in consecutive locations (starting with 000) in the program memory. The loader does not read the input data; this is read by the user’s program whenever it executes a +8 instruction.

Therefore, the general structure of a program is (1) declarations (2) executable statements, and (3) input data.

**Implementing the Interpreter**

The interpreter for our pseudo-coded program can be implemented as:

* Model interpreter behavior after manual execution
* Cheat: Implement using a high-level language
* We have to simulate the hardware in software

The second option is impossible as high level language is not developed during 1950s. We will focus on the third implementation method.

**Data Structures**

Data structures are needed to simulate the IBM 650 which contains:

* Data memory
* Program memory
* Instruction pointer

**Structure of the Interpreter**

000 000

**Data**

**Memory**

**Program**

**Memory**

999 999

IP

[ Fig. Interpreter Data Structure]

**The Read-Execute Cycle is the Heart of an Iterative Interpreter**

We can now consider how a program is actually interpreted. Roughly, what we will do is read the next instruction to be executed (as indicated by the instruction pointer), determine the operation encoded by the instruction, and then perform that operation. When execution of the operation is completed. This process is called the *read-execute cycle*, and can be summarized as follows:

1. Read the next instruction
2. Decode the instruction
3. Execute the operation
4. Continue from step 1

The instruction pointer (IP) must be updated each time an instruction is read and the best option to that is in step 1 because if we update at last the branch instruction may overwrite the IP.

Typical code for step 1 is

*instruction : = Program[IP];*

*IP : = IP+1;*

**Decoding Instructions by extracting their parts**

PseudoCode has been designed with a regular structure, decoding is simple; we simply extract the sign, operation code, and the three address fields. For example, the destination address could be extracted by

*dest := abs (instruction) mod 1000*

(Where ‘x mod y’ gives the remainder of dividing x by y). We assume that the names of these extracted parts are **sign, op, opnd1, opnd2, and dest**.

The operations types are:

•  Select operation

Switch-statement (case-statement)

•  Arithmetic operations

Straight-forward

•  Control-flow

IP may also need to be altered

**Labeling**

Labeling is the aid to the coding of the program. Labels make programs readable however they are not converted into object codes. Labels are used for replacing absolute addressing in the programs particularly in the loops and branching.

We can define a label in pseudocode using label definition operator as:

  -7 0LL 000 000

defines the statement number, or label, LL.

Only 100 numeric labels are possible (00-99)

The control flow instructions then jump to labels instead of absolute address.

**Interpreting Labels**

Label Interpreting Approaches:

**Look through all instructions from beginning of program?**

Yes, but that is slow. This is how some interpreters work. (BASIC, for instance)

**Create label table with absolute addresses for labels and bind addresses**

|  |  |
| --- | --- |
| **Label** | **Location** |
| 20 | 001 |
| 40 | 005 |
| 50 | 009 |

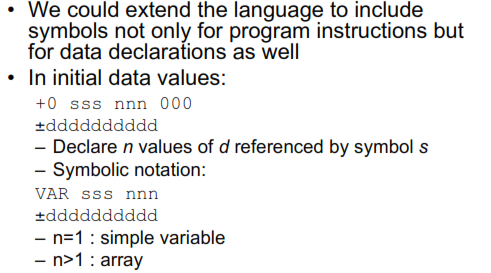
Much faster. Compilers do it this way.

***The labeling principle must be followed i.e. do not require users to know absolute numbers or addresses. Instead associate labels with number or addresses.***

**Data Labels**

We could use labels for variables as well by constructing a symbol table. This idea is easily extended to instructions as well to form a symbolic pseudo-code.

**Data Declaration**



**Debugging?**

Debugging always has to be done and it can be facilitated by debugging by printing instructions executed in order. Also, Interpreter can include *trace*, to get a trace of the execution of the program, that is, a record of the instructions it has executed.

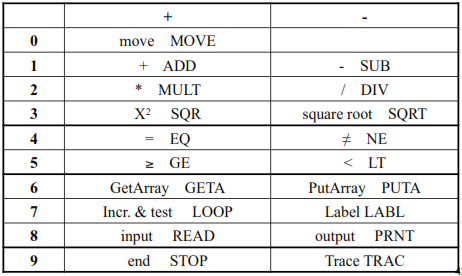
*Read Next Instruction:*

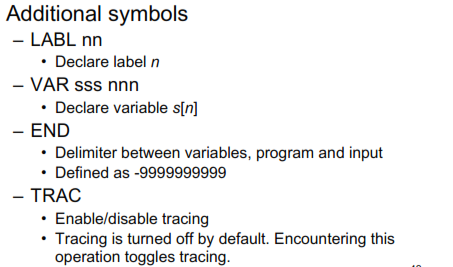
*If trace is enabled then*

*print IP, instruction;*

*IP := Ip+1.*

**Complete Symbolic Language**





**Sample Program**

VAR ZRO 1 Constant Zero

+0000000000

VAR I 1 Index

+0000000000

VAR SUM 1 Sum of array

+0000000000

...

END

READ N Read number of elements

LABL 20

READ TMP Read into TMP

GE TMP ZRO 40 If +ve, skip to 40

SUB ZRO TMP TMP Negate TMP

LABL 40

PUTA TMP DTA I Move TEMP into the ith element

LOOP I N 20 Loop for all array elements

...

STOP

END

***PseudoCode must follow security principle i.e. no program that violates the definition of the language, or its own intended structure, should escape detection.***

**Phenomenology of programming languages**

Programming language is used for solving the problems so it can be taken as a tool. Some of the phenomenon’s of programming language as tool from the investigations of Don Ihde are:

* ***Tools are Ampliative and Reductive***

To better understand the phenomenology of programming languages, we may begin with a simpler tool. Ihde contrasts the experience of using your hands to pick fruit with that of using a stick to knock the fruit down. On the one hand, the stick is ampliative: it extends your reach to otherwise inaccessible fruit. On the other hand, it is reductive: your experience of the fruit is mediated by the stick, for you do not have the direct experience of grasping the fruit and tugging it off the branch. You cannot feel if the fruit is ripe before you pick it.

“Technological Utopians” tend to focus on the ampliative aspect- the increased reach and power- and to ignore the reductive aspect, whereas “technological dystopian” tend to focus on the reductive aspect- the loss of direct, sensual experience - and to diminish the practical advantages of the tool.

* ***Fascination and fear are common to new tools***

When first introduced, programming languages elicited the two typical responses to a new technology: fascination and fear. Utopians tend to become fascinated with the ampliative aspects of new tools, so they embrace the new technology and are eager to use it and to promote it (even where its use is inappropriate); they are also inclined to extrapolation: extending the technology toward further amplification. Dystopian, in contrast, fear the reductive aspects of the tool (so higher level language are fear for their efficiency), or sometimes the ampliative aspects, which may seem dangerous. Ideally, greater familiarity with a technology allows us to grow beyond these reactions.

* ***With mastery, objectification become Embodiment***

A tool replaces immediate (direct) experience with mediated (indirect) experience. Yet, when a good tool is mastered, its mediation becomes transparent. Consider again the stick. If it is a good tool (sufficiently stiff, not too heavy, etc.) and if you know how to use it, then it functions as an extension of your arm, allowing you both to feel the fruit and to act on it. In this way the tool becomes partially embodied. On the other hand, if the stick is unsuitable or you are unskilled in its use, then you experience it as an object separate from your body; you relate to it rather than through it. With mastery a good tool becomes transparent; it is not invisible, for we still experience its ampliative and reductive aspects, but we are able to look through it rather than at it.

As you acquire skill with the language, it becomes transparent so that you can program the machine through the language and concentrate on the project rather than the tool. With mastery, objectification yields (partial) embodiment.

* ***Programming Language influence focus and action***

Tools influence the style of a project. E.g. writing technologies: dip pen, an electric typewriter, and a word processor. In case of dip pen it is slower than the speed of thought, with typewriter the speed is closer to the speed of thought, and with word processor, text can be revised and rearranged in small units, so there is greater tendency to salvage bits of text.

In general, a tool influences focus and action. It influences focus by making some aspects of the situation salient and by hiding others. Like others, programming language influence the focus and actions of programmers and therefore their programming style.

***FORFTRAN PROGRAM***

**PROGRAM INOUT**

C

C This program reads in and prints out a name

C

CHARACTER NAME\*20

PRINT \*,’ Type in your name, up to 20 characters’

PRINT \*,’ enclosed in quotes’

READ \*,NAME

PRINT \*,NAME

END

**PROGRAM AVERAG**

C C THIS PROGRAM READS IN THREE NUMBERS AND SUMS AND AVERAGES THEM.

C

REAL NUMBR1,NUMBR2,NUMBR3,AVRAGE,TOTAL

INTEGER N

N = 3

TOTAL = 0.0

PRINT \*,’TYPE IN THREE NUMBERS’

PRINT \*,’SEPARATED BY SPACES OR COMMAS’

READ \*,NUMBR1,NUMBR2,NUMBR3

TOTAL= NUMBR1+NUMBR2+NUMBR3

AVRAGE=TOTAL/N

PRINT \*,’TOTAL OF NUMBERS IS’,TOTAL

PRINT \*,’AVERAGE OF THE NUMBERS IS’,AVRAGE

END

WRITE A PROGRAM THAT READS N NUMBERS AND FIND THEIR SUM AND AVERAGES.