What is a programming language?

A programming language is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks. It is a formal language that has rules, grammar, and logical sequence

A brief History of computer programming

In the 1840s, Ada Lovelace became the first computer programmer, she designed a program for the analytical machine even though it wasn’t ever manufactured.

She was also the first person to suggest that a computer could be more than just an oversized calculator. Her idea was that the numerical values produced by the computer could be used to represent something other than numbers: symbols, musical notes or well, pretty much anything.

Lovelace wrote an algorithm for the Analytical Engine, the first computer program, to compute Bernoulli numbers.

Mr. Babbage gave a lecture on this machine in the University of Turin which was transcribed in French. Ada took a year to translate the script to English because she wasn’t just translating the script, she added her own notes, this make the transcript being 3 times longer. Her notes made it clear that she understood the working of the Analytical machine

**Programming Language Generations**

Programming languages have been developed over the year in stages. This is referred to as ***programming language generation.*** With each stage of development, the programming languages become more user-friendly, easier to use and more powerful.

There are 5 stages/phases in which programming languages are grouped

**First Generation Language (Machine language) 1GL 1940-1956**

The programming language in this generation is machine language which is a collection of binary digits. Programmers had to use machine language because there was no other option available. It was the only programming language available.

**Second Generation language (Assembly Language) 2GL 1956-1963**

The first step in making software development easier was the creation of Assembly language. This language is also classified as low-level language because detail knowledge of the hardware is still required.

**Third Generation languages (High-Level Languages) 3GL 1964-1971**

The third-generation programming languages were designed to overcome the various limitations of the first- and second-generation programming languages.

Languages within this generation are high level language. They enable the programmer to concentrate only on the logic of the programs without considering the internal architecture of the computer system. 3G introduced many programmer-friendly features to code such as loops, conditionals, classes etc. This means that one line of third generation code can produce many lines of object (machine) code, saving a lot of time when writing programs.

**Third generation codes are imperative language**

**Fourth Generation Language (Very High Level) 4GL 1971- Present**

The languages of this generation were considered as very high-level programming languages. The fourth-generation programming languages were designed and developed to reduce the time, cost and effort needed to develop different types of software applications.

Designed to be closer to natural language than 3GL. They normally consist of statements that are similar to statements in the human language. These are used mainly in database programming and scripting.

4th generation programming languages are the declarative language

**Fifth Generation Language (Artificial Intelligence Language) 5GL Present and beyond**

The programming languages of this generation focus mainly on the use of visual tools to develop a program.

The major fields in which the fifth-generation programming language are employed are Artificial Intelligence and Artificial Neural Networks

Comparison chart of programming language generation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1GL 1940-1956 | 2GL 1956-1963 | 3GL  1964-1971 | 4GL  1971-Present | 5GL  Present and beyond |
| Machine code | binary or hexadecimal | High level language | High level language | Understand natural language |
| Difficult to use and debug | Closer to English  Uses mnemonics | Output was visualized on the monitors | Had GUI  Use a Mouse | Recognize human speech |
| Easy to make errors | Known as assembly language | Simplified programming languages i.e. BASIC | Uses different programming language eg. Pascal | Implement neural networks  Play interactive games |
| Don’t need translations | Uses assembler as the translator | Complier and Interpreter used | Complier and Interpreter used | Exercise heuristic classifications |
| Programs runs quickly | Machine dependent | Multitasking  Machine independent | Multitasking  Machine independent | See the world in three-dimensional perspective |
| Uses Vacuum tube for the circuit  Large in size | Transistor | Integrated chips (IC) slightly smaller  Used motherboards | Microprocessor. The CPU contains thousands of transistors  Were much smaller | ULSI (Ultra Large Scale Integration) technology |
| Memory speed 300 IPS (instructions Per sec) | 300 IPS  (million inst. Per sec.) | 1MIPS  (1 million inst. Per sec.) | Faster than its predecessors | Implement expert input in medical and other complex fields |

Types of Programming Languages

* Low level programming language. (LLL)
* High level programming language (HLL).

Low level language (LLL)

Is often known as a computer's native language and is very close to writing actual machine instructions. It, relate to the specific architecture and hardware of a particular type of computer. It works to control a computer's operational semantics and provides little or no abstraction (hiding complexity and unnecessary details from the users.) of programming ideas

Types

* Machine code
* Assembly language

Machine code

is the lowest and the simplest level of Programming language and it was the first form of programming language to be acquired. Machine Language is fundamentally the only language which a computer can understand. A manufacturer produces a computer to understand only one language which is the machine code, and it is represented inside the computer in a form of binary digits 0 and 1. The symbol 0 stands for the absence of Electric pulse and 1 stands for the presence of an electric pulse . A computer is able to recognize electric signals, therefore, it understands machine language

Instructions in machine code comprise of a certain number of bits. If instructions for a particular processor are 8 bits, for example, the first 4 bits part (the opcode) tells the computer what to do and the second 4 bits (the operand) tells the computer what data to use.

Advantages

* Machine language makes fast and efficient use of the computer.
* Efficient use of storage for instructions and data
* It requires no translator to translate the code. It is directly understood by the computer.

Disadvantages

* It is machine dependent
* Take a long time to write codes in binary form
* Computer storage location must be addressed directly
* It requires high level of programming skills

Low level language -Assembly language

Assembly language is not typically considered a programming paradigm itself. Instead, assembly language is a low-level programming language that represents a specific way of interacting with the hardware of a computer. It provides a human-readable representation of machine code instructions, which are directly executed by the computer's processor.

Assembly language is often associated with the imperative programming paradigm because it involves specifying a sequence of instructions that are executed one after the other. However, assembly language itself does not provide higher-level abstractions or concepts that characterize paradigms like procedural or object-oriented.

Assembly language is a low-level programming language that provides a human-readable representation of machine code instructions. It is specific to the architecture of a particular computer or processor. Assembly language is considered a bridge between machine code and higher-level programming languages.

Assembly language,

* instructions are written using mnemonics that represent specific operations or actions to be performed by the computer's processor. Each mnemonic corresponds to a specific machine code instruction that the processor can directly execute.
* allows for specifying memory addresses and data manipulation.
* a good understanding of the computer's architecture is needed when writing assembly language, including its registers, memory organization, and instruction set.
* often used in areas where performance is crucial, such as operating systems, device drivers, embedded systems, and real-time systems. It is also used in reverse engineering and low-level system programming.
* Each computer architecture has its own assembly language. Examples include x86 assembly language for Intel processors, ARM assembly language for ARM-based processors, and MIPS assembly language for MIPS processors. Assemblers are used to convert assembly language code into machine code that can be executed by the processor.

The structure of assembly language

The structure of assembly language and machine code instructions is the same.

Each instruction has an

* opcode that identifies the operation to be carried out by the CPU.
* an operand that identifies the data to be used by the opcode.

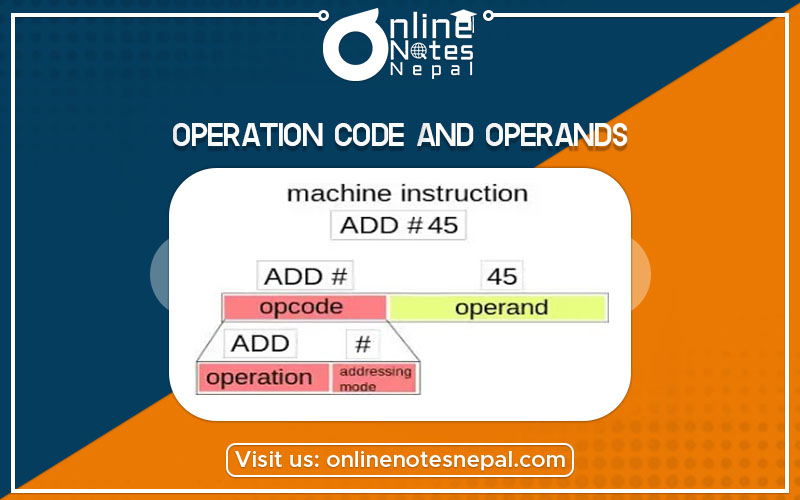
The opcode (operation code) is a part of the assembly language instruction that specifies the operation to be performed. It's a mnemonic representation of an underlying machine code instruction, making it easier for humans to read and write.

Examples of opcodes include MOV, ADD, SUB, JMP, and so on. Each opcode corresponds to a specific operation the CPU understands, like moving data, performing arithmetic, or changing the flow of control.

The operand

The operand(s) of an assembly instruction specify the data or memory address to be operated on or the data locations. The number and type of operands depend on the specific instruction. Operands can be registers, immediate values, memory addresses, or labels. Operands can be immediate values (like a constant number), CPU registers, or memory addresses.

A diagram of a graph

Description automatically generated with medium confidence

**What is the relationship between assembly language and machine code?**

**In summary**

Assembly language and machine code are closely related, as they both deal with low-level programming instructions used by computers.

Machine code is the lowest level of programming language that a computer can understand directly. It consists of binary instructions, typically represented as a sequence of 0s and 1s, which correspond to specific operations that the computer's hardware can execute. Each instruction in machine code is directly executed by the computer's central processing unit (CPU).

Assembly language, on the other hand, is a human-readable representation of machine code. It uses mnemonic codes (abbreviations or symbols) to represent the binary instructions of the machine code. Each mnemonic corresponds to a specific machine instruction. For example, instead of writing a sequence of binary digits, you can write "ADD" to represent the addition operation or "MOV" to represent a data movement operation.

Assembly language provides a more readable and understandable way for programmers to write low-level code compared to machine code. It allows programmers to use symbolic names for registers, memory locations, and other elements, making the code more self-explanatory and easier to maintain.

To translate assembly language code into machine code, you need an assembler. An assembler is a program that converts the mnemonic instructions in assembly language into their corresponding machine code representations. The resulting machine code can then be executed directly by the computer's hardware.

High-Level Language

A program written in high level language is written in a form that is close to human language. It enables the programmer to just focus on the problem to be solved. No particular knowledge of the hardware is needed as high-level languages create programs that are portable and not tied to a particular computer or microchip. It is generally independent of the computer’s hardware architecture. Uses English like statements for writing of codes. Based on what the program will do rather than the components of the computer it will be used with. Its portable. Program written can be used on other computers.

It has a higher level of abstraction from the computer and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization

Advantages

* Easier to modify as it uses English like statements
* Easier/faster to write code as it uses English like statements
* Easier to debug during development due to English like statements
* Portable code – not designed to run on just one type of machine (Machine independent)

Disadvantages

* It takes additional translation times to translate the source to machine code.
* Slower than low level programs.
* Less memory efficient.
* Cannot communicate directly with the hardware

High-level and low-level comparison chart

|  |  |
| --- | --- |
| **High-Level Language** | **Low-Level Language** |
| Easy to understand and widely used in today’s times. | Are very difficult to understand by human beings |
| These are very easy to execute. | These are very difficult to execute. |
| Require the use of a translator | requires an assembler |
| Very low memory efficiency. Uses a lot of memory. | High memory efficiency. Uses less memory. |
| These are portable from any one device to another. | Codes are not portable. |
| Human-friendly. | Machine-friendly. |
| Machine independent | Machine-dependent |
| It is very easy to debug. | Not easy to debug. |
| Simple and comprehensive maintenance technique. | complex to maintain |
| Take more time for execution, require a translation program. | Less time for code execution. |
| Allow a higher abstraction. | Allow very little abstraction or no abstraction at all. |
| One does not require a knowledge of hardware for writing programs. | Having knowledge of hardware is a prerequisite to writing programs. |
| Do not provide various facilities at the hardware level. | Very close to the hardware. Help in writing various programs at the hardware level. |
| Some examples of high-level languages include Perl, BASIC, COBOL, Pascal, Ruby, etc. | Some examples of low-level languages include the Machine language and Assembly language |

Translators

**Translators**/convertors

Every high-level language source's code should be converted to low level language so that computer's microprocessor can understand and execute that.

For that reason, every language comes with its own translator, which is either a compiler or an interpreter.

Every programming language source's code should be converted to low level language so that computer's microprocessor can understand and execute that

* A translator will convert the source code into machine code (object code)
* Source code is the language that the code is written in by the programmer usually in high level language
* Object code is the language that understood by the computer usually machine code

Types of Translators

**INTERPRETER**:

* It helps programmers find the error correction while writing programs and testing them.
* translates line by line and reports the error once it encountered during the translation process.
* converts each source code’s line and executes it.
* help's programmers find the error correction while writing programs and testing them.
* able to read, translate and execute one statement at a time from a high-level language program.
* only works with a source code and do not produce an independent executable file.
* Work with an emulator and doesn't work directly over two microprocessor. This saves computer from crashing when the program performs any illegal operation.

**COMPILER** :

* It reads the whole program and generates an executable file (exe), which can be easily distributed without the fear of source code being copied.
* Debugging is tough with compilers as they convert the whole program and gives out all the errors in that program compiler runs directly over the microprocessor, thus can crash the computer.
* convert programs in high-level language to low-level language.
* translates the entire program and then reports the errors in source program encountered during the translation.

**Assembler:**

Every new architecture has its own assembly language.

* Any source code that is written for that particular architecture should be translated to machine language through a translator made in assembly for that particular architecture called Assembler.
* Assemblers are used to translate a program written in a low-level assembly language into a machine code (object code) file so it can be used and executed by the computer.
* Once assembled, the program file can be used again and again without re-assembly.

Types of Assemblers

There are several different types of assemblers used to translate assembly language code into machine code. The two main types are

* One pass Assembler
* Two pass assemblers

**One-pass/single assemblers**: These assemblers translate assembly language code to machine code in just one scan through the source code. They require that each symbol or label be defined before it is used, as they cannot resolve forward references. (references to symbols that are defined later in the code). They do not store any information about symbols or labels in memory, so all references must be resolved during the initial pass. This limitation makes them faster but less flexible compared to multi-pass assemblers. They are well-suited for simple assembly languages or when the programmer can ensure that symbols are defined before use. They are well-suited for simple assembly languages or when the programmer can ensure that symbols are defined before use. One pass assembler may generate inefficient code if there are complex interdependencies between symbols.

**Two pass assemblers**

Two-pass assemblers perform the translation in two scans (two-step process)of the source code:

* **First Pass:** called the "analysis pass," the assembler reads the entire assembly code, builds a symbol table, and resolves any forward references. It collects information about labels, addresses, and other symbols used in the code to build the symbol table without generating any machine code.
* **Second Pass:** called the "synthesis pass," the assembler generates the actual machine code, substituting addresses and values based on the information collected in the first pass. It uses the symbol table to translate instructions into machine code and to resolve all symbol references, including forward references.

Two-pass assemblers can handle more complex interdependencies than one-pass assemblers, but they take longer to run due to the need for multiple passes.

Example of how the two pass assembler works when translating assembly language (source code) into machine code (object code)

START: LOAD A, 5 ; Load the value 5 into register A

ADD A, B ; Add the value of label B (to be defined later) to A

STORE A, RESULT ; Store the result in a memory location labelled RESULT

JUMPZ END ; Jump to label END if the result is zero

B: DATA 3 ; Define B with the value 3

RESULT: DATA 0 ; Reserve space for the result, initialized to 0

END: NOP ; No operation, end of program

**First Pass**

During the first pass, the assembler scans the program to build the symbol table without generating any machine code. The main goal is to record the addresses of all labels (e.g***., START, B, RESULT, END***).

**Symbol Table** (***addresses are hypothetical***):

* START -> **0x0000**
* B -> **0x0004**
* RESULT -> **0x0005**
* END -> **0x0006**

The assembler calculates the address of each label based on the instruction sizes and data declarations. At this stage, it also determines the memory layout of the program.

**Second Pass**

In the second pass, the assembler translates the assembly instructions into machine code, using the symbol table to resolve addresses for labels and variables.

* *LOAD A, 5* becomes machine code equivalent to load 5 into register A.
* *ADD A, B* now, that the assembler knows B's address (0x0004), it generates the appropriate machine code to add the value at this address to register A.
* *STORE A, RESULT* translates into machine code that stores the content of register A into the memory location associated with RESULT (0x0005).
* *JUMPZ END* is translated into machine code that performs a conditional jump to the address of *END* (0x0006) if the zero flag is set.

The assembler also handles the *DATA* directives by allocating space and setting initial values as defined.

Important to note

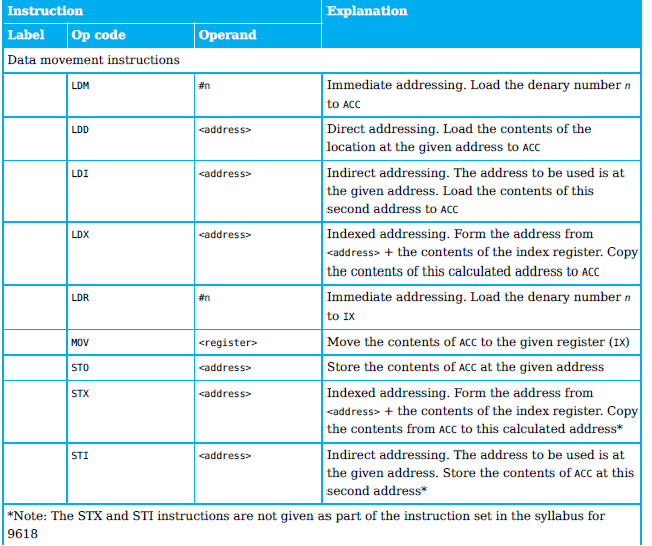
* **Memory Addresses**: In this example, the addresses are hypothetical. In a real scenario, the actual addresses would depend on the assembler's memory model and the starting address of the program.
* **Machine Code Translation**: The specific machine code generated would depend on the target architecture (e.g., x86, ARM). The mnemonics (LOAD, ADD, STORE, JUMPZ, NOP) would be replaced by their binary equivalents according to the CPU's instruction set.
* **Label Resolution**: The key part of the second pass is resolving the symbolic references to their actual memory addresses using the symbol table created during the first pass.

Assembly Language instruction set.

What is an instruction set?

An instruction set, often abbreviated as ISA (Instruction Set Architecture), is a collection of instructions that a processor can execute. It includes a variety of instructions that specify different operations, such as arithmetic and logical operations, data movement, control flow, and input/output operations.

Each instruction typically has an opcode that represents the specific operation to be performed, as well as the addressing mode that specifies how the operands (data or memory addresses) for the operation are accessed

Data Movement Instruction

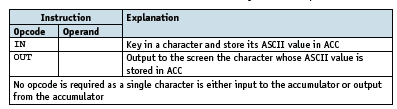
Arithmetic Operation

A white and blue box with black text

Description automatically generated

*Input and output of data instructions*

These instructions allow data to be read from the keyboard or output to the screen.



*Unconditional and conditional instructions*

A white and blue text with black text

Description automatically generated

*Compare instructions*

A close-up of a list of information

Description automatically generated

Addressing mode

* **Direct /Absolute addressing –**The instruction specifies the memory address where the operand is located. The address of the operand is given directly.

**Example: MOV AX, [1234H] (Move the content at memory address 1234H into register AX).**

* **Indirect addressing** –The memory address of the operand is contained in a register, and the instruction specifies the register. The CPU uses the register's content as the address to access the operand.

**Example: MOV AX, [BX] (Move the content at the memory address contained in register BX into AX).**

* **Indexed addressing** – the contents of the memory location found by adding the contents of the index register (IR) to the address of the memory location in the operand are used.   
  *(Combines a base address held in a register with an immediate offset to calculate the effective address of the operand. It's useful for accessing array elements.)*

Example: MOV AX, [BX+4] (Move into AX the content from memory at the address computed by adding 4 to the content of BX).

* **Immediate addressing** –The operand is specified directly in the instruction itself. It is a constants or fixed values within the instruction.

**Example: MOV AX, 5 (Move the immediate value 5 into register AX).**

* **Relative addressing** – the memory address used is the current memory address added to the operand.   
  *(Specifies an operand's address by giving an offset relative to the current instruction pointer (program counter). This mode is often used for branching instructions.)*
* **Example:** **JMP [IP+20H] (Jump to an instruction located 20H bytes away from the current instruction pointer**).
* **Symbolic addressing** – only used in assembly language programming. A label is used instead of a value.
* **Register Addressing Mode**

The operand is located in a register, and the instruction specifies the register directly.

**Example**: **MOV BX, AX (Move the contents of register AX into register BX).**