|  |
| --- |
| **UNIT 1** |
| **INTRODUCTION** |
| **Unit-01/Lecture-01** |
| **HISTORY OF PROGRAMMING LANGUAGES**  In 1957, the first of the major languages appeared in the form of FORTRAN. Its name stands for FORmula Translating system. The language was designed at IBM for scientific computing. The components were very simple, Today, this language would be considered restrictive as it only included IF, DO, and GOTO statements, but at the time, these commands were a big step forward. The basic types of data in use today got their start in FORTRAN, these included logical variables (TRUE or FALSE), and integer, real, and double-precision numbers. Though FORTAN was good at handling numbers, it was not so good at handling input and output, which mattered most to business computing.  Business computing started in 1959, and because of this, COBOL was developed. It was designed as the language for businessmen. Its only data types were numbers and strings of text. It also allowed for these to be grouped into arrays and records, so that data could be tracked and organized better. COBOL statements also have a very English-like grammar, making it quite easy to learn  The Algol language was created by a committee for scientific use in 1958. Its major contribution is being the root of the tree that has led to such languages as Pascal, C, C++, and Java. It was also the first language with a formal grammar, known as Backus-Naur Form or BNF. Though Algol implemented some novel concepts, such as recursive calling of functions his lead to the adoption of smaller and more compact languages, such as Pascal.  Pascal was designed in a very orderly approach; it combined many of the best features of the languages in use at the time, COBOL, FORTRAN, and ALGOL. The combination of features, input/output *and* solid mathematical features, made it a highly successful language. Pascal also improved the "pointer" data type, a very powerful feature of any language that implements it. It also added a CASE statement that allowed instructions to branch like a tree. Pascal also helped the development of dynamic variables, which could be created while a program was being run, through the NEW and DISPOSE commands. However, Pascal did not implement dynamic arrays, or groups of variables, which proved to be needed and led to its downfall.  In 1958, LISt Processing (or LISP) language was developed for Artificial Intelligence (AI) research. Because it was designed for such a highly specialized field, its syntax has rarely been seen before or since. The most obvious difference between this language and other languages is that the basic and only type of data is the list, denoted by a sequence of items enclosed by parentheses. LISP programs themselves are written as a set of lists, so that LISP has the unique ability to modify itself, and hence grow on its own.  C was developed in 1972 by Dennis Ritchie. All of the features of Pascal, including the new ones such as the CASE statement are available in C. C uses pointers extensively and was built to be fast and powerful at the expense of being hard to read. But because it fixed most of the mistakes Pascal had, it won over former-Pascal users quite rapidly. C is very commonly used to program operating systems such as UNIX, Windows, the MacOS, and Linux.  In the late 1970's and early 1980's, a new programming method was being developed. It was known as Object Oriented Programming, or OOP. Objects are pieces of data that can be packaged and manipulated by the programmer. Bjarne Stroustroup liked this method and developed extensions to C known as "C with Classes."  In early 1990s Sun Microsystems developed a new object oriented language called Java was introduced. Java is first Programming language which is not attached with any particular hardware or operating system. Program developed in java can be executed anywhere and on any system.  A high-level programming language that is interpreted by another program at runtime rather than compiled by the computer's processor as other programming languages (such as C and C++) is. Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements. These types of languages are client-side scripting languages, affecting the data that the end user sees in a browser window. Other scripting languages are server-side scripting languages that manipulate the data, usually in a database, on the server.  JavaScript, ASP, JSP, PHP, Perl, and Python are examples of scripting languages.  Programming Domains  Scientific  Applications Business Artificial System Web  Applications Intelligence Programming Software    ALGOL FORTRAN COBOL LISP PROLOG C JAVA  SCRIPT PHP    **Unit-01/Lecture-02**  **LANGUAGE EVALUATION CRITERIA**  criteria.png  Readability  The ease with which programs can be read and understood is called readability. The following describe characteristics that contribute to the readability of a PL   1. Simplicity- The language that has large no. of basic components is more difficult to learn than one with a small no of basic components. The language should not have multiplicity of commands. For e.g. I = I + 1 ; I + = 1 ;I + + ; + + I . The language should not have operator overloading in which a single operator symbol has more than one meaning. 2. Orthogonal – It means that a relatively small number of primitive constructs can be combined in a number of ways to build the program. Orthogonal language is independent of the context of its appearance in the program. 3. Control Statements- A program that can be read from top to bottom is much easier to understand than a program that requires the reader to jump from one statement to some other non adjacent statement. Example- goto statements. 4. Data Types and Structures – The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability.   There should be provision for data structures in a language are another significant aid to readability. There should be provision for data types, for record type of data types(representing an array of employee records)   1. Syntax considerations – Syntax is the form of elements of language. There are 3 types of syntactic design choices that affect readability. Different forms of identifiers, special keywords (reserve words), Form & meaning – constructs that are similar in appearance but different meaning is not readable.   Writ ability  The measure of how easily a language can be used to create programs for a chosen problem domain. The features that affect the readability of a also affect the writ ability apart from them, the factors that influence writability are   1. Simplicity- A large language takes more time to learn. Programmers might learn only a subset. Feature multiplicity (having more than one way to perform a particular operation) is often confusing. For example, in C++ or Java you can decrement a variable in four different ways: x = x – 1; x -= 1; x--; --x. Operator overloading (a single operator symbol has more than one meaning) can lead to confusion. Some languages (e.g. assembly languages), can be "too simple" – too low level. 2, 3, 4, 5 or more statements needed to have the effect of 1 statement in a high-level language 2. Orthogonality- In general use, it means being independent, non-redundant, non-overlapping, or not related. In computer languages, it means a construct can be used without consideration as to how its use will affect something else. A programming language is considered orthogonal if its features can be used without thinking about how their use will affect other features, e.g. objects and arrays. Having fewer constructs and having few exceptions increases readability and writability. Orthogonal languages are easier to learn. Examples:Pointers should be able to point to any type of variable or data structure. 3. Support for abstraction – process & data abstraction both. Abstraction means the ability to define and use complicated structures or operations in ways that allow many of the details to be ignored. For example- To use a subprogram to implement a sort algorithm that is required several times in a program. Without the subprogram the sort code would have to be replicated in all places where it was needed, which would make the program much longer and more tedious to write. 4. Expressivity- The great deal of computation must be accomplished with a very small program. A language must have relatively convenient, rather than cumbersome ways of specifying computations. For example in C, the statement count++ is more convenient and shorter than count=count+1. Also the use of for statement in Java makes writing counting loops easier than with the use of while, which is also possible.   Reliability  A program is said to be reliable if it performs to its specifications under all conditions. Along with all the features that affect readability and writ-ability there are several other features that affect reliability   1. Type checking – It is the testing for type errors in a given program either by compiler or during program execution. Runtime checking is expensive. Examples of failures of type checking(i)Countless loops(ii)Formal and actual parameter being of different types(iii)Array out of bounds 2. Exception Handling - The ability of a program to intercept run-time errors, take corrective measures and then continue is a great aid to reliability. ADA, C++, Java include this capability whereas C, FORTRAN don’t. 3. Aliasing- Aliasing is referencing the same memory cell with more than one name   E.g., in C, both x and y can be used to refer to the same memory cell  int x = 5;  int \*y = &x;  Aliasing is a dangerous feature in a programming language.  Cost  The ultimate cost of a programming language is a function of many of its characteristics   1. The cost of training programmers 2. The cost of writing programs 3. The cost of compiling programs 4. The cost of executing programs 5. The cost of Language implementation System 6. The cost of poor reliability 7. The cost of maintaining program   **Unit-01/Lecture-03**  **Influences on Language Design**  The basic structure proposed in the draft became known as the “von Neumann machine” (or model).a memory, containing instructions and data a processing unit, for performing arithmetic and logical operations a control unit, for interpreting instructions.  index.jpeg  Memory   * 2k x m array of stored bits * Address -unique (k -bit) identifier of location * Contents- m-bit value stored in location   Basic Operations:   * LOAD   read a value from a memory location   * STORE   write a value to a memory location  Processing Unit  Functional Units   * ALU = Arithmetic and Logic Unit * Could have many functional units. some of them special-purpose   (multiply, square root, ... )   * LC-3 performs ADD, AND, NOT   Registers   * Small, temporary storage * Operands and results of functional units * LC-3 has eight registers (R0, .., R7), each 16 bits wide   Input and Output   * Devices for getting data into and out of computer memory * Each device has its own interface, usually a set of registers like the memory’s MAR and MDR * LC-3 supports keyboard (input) and monitor (output)   keyboard: data register (KBDR) and status register (KBSR)  monitor: data register (DDR) and status register (DSR)  Control Unit   * Orchestrates execution of the program * Instruction Register (IR) contains the current instruction * Program Counter (PC) contains the address of the next instruction to be executed. * Control unit * reads an instruction from memory * the instruction’s address is in the PC * interprets the instruction, generating signals that tell the other components what to do   **Unit-01/Lecture-04**  **Programming Paradigms**  (**[**RGPV/ June 2011 (10)]  The most influential programming paradigms include the procedural (also known as the imperative), structured, functional, logic, and object-oriented paradigms.  The Procedural (Imperative) Programming Paradigm  The traditional model of computation, the procedural programming paradigm specifies a list of operations that the program must complete to reach its final goal. It describes the steps that change the computer’s state of memory by providing statements such as assignment statements. This paradigm creates procedures, functions, subroutines, or methods by splitting tasks into small pieces, thus allowing a section of code to be reused in the program to some extent and making it easier for programmers to understand and maintain the program structure. However, it is still difficult to solve problems, especially when they are large and complicated, since procedural programming languages are not particularly close to the way humans think or reason. Procedural programs are difficult to maintain and it is not easy to reuse the code when the program is large and has many procedures or functions. Moreover, if a modification must be made in one of its states or conditions, it is difficult and time consuming to do so. These drawbacks make using this paradigm very expensive. Procedural programming languages include Algol, FORTRAN, COBOL, and BASIC.  The Structured Programming Paradigm  The structured programming paradigm can be seen as a subset of the procedural programming paradigm. Its characteristics include removing or reducing the use of global variables, relying on the GOTO statement, and introducing variables local to blocks such as procedures, functions, subroutines, or methods, which result in variables declared inside a block that are invisible outside it. The structured programming paradigm is often associated with the top-down approach. This approach first decomposes the problem into smaller pieces. These pieces are further decomposed, finally creating a collection of individual problems. Each problem is then solved one at a time. Though this approach is successful in general, it causes problems later when revisions must be made. Because each change requires modifying the program, this approach minimizes the reuse of code or modules. The structured programming paradigm includes languages such as Pascal and C.  The Functional Programming Paradigm  The functional programming paradigm was created to model the problem rather than the solution, thus allowing the programmer to take a high-level view of what is to be computed rather than how. In this paradigm, the program is actually an expression that corresponds to the mathematical function f. Thus, it emphasizes the definition of functions instead of the execution of sequential list of instructions.  Because each function is designed to accomplish a specific task given its arguments while not relying on an external state, the functional programming paradigm increases readability and maintainability. However, since this paradigm corresponds less closely to current hardware such as the Von Neumann Architecture, it can be less efficient, and its time and space usage can be hard to justify. Also, some things are harder to fit into a model in which functions only take inputs and produce outputs. The functional programming paradigm includes languages such as Lisp and Scheme.  The Logic Programming Paradigm  Instead of specifying instructions on a computer, the logic programming paradigm enables the expression of logic. It is therefore useful for dealing with problems where it is not obvious what the functions should be. In this paradigm, programmers specify a set of facts such as statements or relationships that are held to be true, and a set of axioms or rules (i.e., if A is true, then B is true), and use queries to the execution environment to see whether certain relationships hold and to determine the answer by logical inference. This paradigm is popular for database interfaces, expert systems, and mathematical theorem  provers.  In the following example, we declare the facts about some domain. We can then make queries about these facts—for example, are Ryan and brian siblings?  sibling(X,Y) :− parent(Z,X), parent(Z,Y)  parent(X,Y) :− father(X,Y)  parent(X,Y) :− mother(X,Y)  mother (megan, brian).  father(ryan, brian)  father(ryan, molly)  father(mike, ryan)  Under the logic programming paradigm fall languages such as Prolog.  The object-Oriented Programming Paradigm  Object-oriented programming is the newest and the most prevailing paradigm. It suggests new ways of thinking for problem-solving, since its techniques more closely model the way humans solve problems. Traditionally, a program has been viewed as a logical procedure that takes input processes it, and generates output. By contrast, the object-oriented programming paradigm focuses on modeling problems in terms of entities called objects that have attributes and behaviors and that interact with other entities using message passing.  The key characteristics of object-oriented programming include class, abstraction, encapsulation, inheritance, and polymorphism. A class is a template or prototype from which objects are created that contains variables and methods, and that specifies a user-defined data type. Abstraction separates the interface from implementation. Encapsulation insulates the data by wrapping them up using various methods, and allows the internal implementation of a class to be shared by specifying what information in an object can be exchanged with others. Inheritance enables hierarchical relationships to be represented and refined. Polymorphism allows objects of different types to receive the same message and respond in different ways.  The object-oriented programming paradigm has many benefits over traditional ones. Since it emphasizes modular code through the abstraction and encapsulation concepts, facilitates a disciplined software development process, enables building secure programs through the data-hiding concept, and eliminates redundant code and defines new classes from existing ones with little effort through inheritance, it creates enhanced reusability, extensibility, reliability and maintainability. The object-oriented programming paradigm includes languages like Smalltalk and Eiffel.  **Unit-01/Lecture-05**  **VIRTUAL COMPUTERS**  A virtual machine (VM) is an operating system (OS) or application environment that is installed on emulated hardware instead of being physically installed on dedicated hardware. The end user has the same experience on a virtual machine as they would have on dedicated hardware.  index.png  Specialized software called a hypervisor emulates the PC client or server's CPU, memory, hard disk, network and other hardware resources completely, enabling virtual machines to share the resources. The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other. Virtual machines that run, for example, Linux and Windows server operating systems, may share the same underlying physical host.  We can have several virtual machines installed on our system; we’re only limited by the amount of storage we have available for them. Once we’ve installed several operating systems, we can open our virtual machine program and choose which virtual machine we want to boot – the guest operating system starts up and runs in a window on our host operating system, although we can also run it in full-screen mode.   |  |  | | --- | --- | |  |  |   Virtual machines (VMs) are also widely used to run multiple instances of the same operating system, each running the same set or a different set of applications. The separate VM instances prevent applications from interfering with each other. If one app crashes, it does not affect the programs in the other VMs. This approach differs from a dual-boot or multiboot environment, in which the user has to choose only one OS at startup. All virtual machines in the same computer run simultaneously.  **COMPILATION**    Compilation is a process that translates a program in one language (the source language) into an equivalent program in another language (the object or target language).  guide4bankexams_compilerphases.jpg  Lexical Analyzer phase  This is the first phase of a compiler. The compiler scans the source code from left to right, character by character, and groups these characters in to tokens. The main function of this phase is:   * Identify the lexical units in a source statement. * Classify units into different lexical classes (e.g.: reserve words, identifiers, constants etc) and enter them in different tables. * Ignore comments in the source program.   The output of lexical analyzer goes as input to syntax analyzer phase.  Syntax Analysis Phase:  The main function of this phase is:   * Obtain tokens from lexical analyzer * Check whether the expression is syntactically correct. * Report syntax errors if any. * Determine the statement class i.e. it is an assignment statement, condition statement etc. * Group tokens in to statements. * Construct hierarchical structures called parse trees. The parse trees represent the syntactic structures of the program.   Consider the statement X = Y +Z. The parse tree is as follows:  Semantic Analysis Phase:  The main function of this phase is:   * Check phrases for semantic errors e.g. type checking. In C program, int x= 10.5; should be detected as semantic error. * Semantic analyzer keeps track of types of identifiers and expressions to verify their consistent usage. * Using the symbol table the semantic analyzer enforces a large number of rules such as  1. Every identifier is declared before it is used. 2. No identifier is used in an inappropriate context (e.g. adding a string to an integer) 3. Every function contains at least one statement that specifies a return value.   **Unit-01/Lecture-06**  Symbol table  The symbol table is built and maintained by the semantic analysis phase. It maps each identifier to the information known about it. This information includes the identifier’s type, internal structure, and scope. Using the symbol table the semantic analyzer enforces a large number of rules such as   1. Every identifier is declared before it is used. 2. No identifier is used in an inappropriate context (e.g. adding a string to an integer) 3. Every function contains at least one statement that specifies a return value.   Intermediate Code Generation  The intermediate code produces a program in a different language, at an intermediate level between the source code and the machine code. Intermediate languages are sometimes assembly languages. The generation of an intermediate code offers the following advantages-  The intermediate representation should have two important properties:  •      It should be easy to produce,  •      And easy to translate into target program.  Code Optimization Phase  Optimization improves programs by making them smaller or faster or both. The goal of code optimization is to translate a program into a new version that computes the same result more efficiently- by taking less time, memory and other resources.  Code optimisation is achieved in two ways   * Rearranging computations in a program to make them execute more efficiently. * Eliminating redundancies in a program   Code optimization should not change the meaning of program.  Code Generation Phase  The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. The code generated depends on the architecture of the target machine.   * Memory locations are selected for each of the variables used by the program. * Then, the each intermediate instruction is translated into a sequence of machine instructions that perform the same task.   **Programming Environments**  A programming environment is the collection of tools used in the development of software. This collection may consist of only a file system, a text editor, a linker and a compiler or it may include a large collection of integrated tools each accessed through a uniform user interface.  UNIX is an older programming environment. It provides a wide array of powerful support tools for software production and maintenance in a variety of languages. In the past the most important feature absent from UNIX was a uniform interface among its tools. This made it more difficult to learn and to use. However UNIX is now often used through a GUI that runs on top of UNIX. Examples of Unix GUI are the Solaris Common Desktop Environment (SCDE).  Borland JBuilder is a programming environment that provides an integrated compiler, editor, debugger and file system for Java Development, where all four are accessed through a graphical interface.  The latest step in the evolution of software development environments is represented by Microsoft Visual Studio .NET, a large collection of software development tools all used through a windowed interface. This system can be used to develop software in any one of the five .NET languages: C#, Visual Basic .NET, Jscript, J# or managed C++.  **Syntax Directed Control Flow**  A program is structured if the flow of control through the program is evident from the syntactic structure of the program text  Evident is defined as single-entry/single-exit   * Control flow in through a single entry point and flows out through a single exit point. * All statements of Pascal, except gotos are single-entry/single-exit.  1. Composition of Statements   A sequence of statements can be grouped into a compounded statement by enclosing it between begin and end  temp := x; x:=y; y:=temp;  ⇒begin temp:=x; x:=y; y:=temp; end  The statement sequence can be empty  ⇒begin end   * the compounded statement can appear wherever a statement is expected.In Pascal, semicolons separate statements. * A preferred alternative design is for semicolons to terminate statements  1. Selection: Conditional Statement   A conditional statement has the form  if <expression> then <statement1> else<statement2>  If expression is true, control flow through statement1; otherwise, control flows through statement2  A variant is  if <expression> then <statement>  statement is executed when only expression is true.  When conditionals are nested, the following style improves readability  if ...then ...  elseif...then ...  elseif...then ...  else ...   1. Looping Constructs:   While and Repeat  A definite iteration is executed a predetermined number of times.  The number of executions of an indefinite iteration is not known.  while<expression> do<statement>  <statement> is called the body of the while construct  while x<>0 do  begin...end  repeat<statement-list>  until  <expression>  Definite Iteration:  For Each Element  Do for <name>:= < expression>to<expression> do  < statement>  **Unit-01/Lecture-07**  **Formal Methods of Describing Syntax**  Syntax - the form or structure of the expressions, statements, and program units.  Semantics - the meaning of the expressions, statements, and program units.  Ex: while (<Boolean\_expr>)<statement>  The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed.  The form of a statement should strongly suggest what the statement is meant to accomplish.   1. Context Free Grammar-   A CFG consists of the following components:   * A set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar. * A set of *non-terminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the non-terminal symbols. * A set of *productions*, which are rules for replacing non-terminal symbols (on the left side of the production) in a string with other non-terminal or terminal symbols (on the right side of the production). * A *start symbol*, which is a special non-terminal symbol that appears in the initial string generated by the grammar.   A CFG for Arithmetic Expressions  An example grammar that generates strings representing arithmetic expressions with the four operators +, -, \*, /, and numbers as operands is:   1. <expression> --> number 2. <expression> --> ( <expression> ) 3. <expression> --> <expression> + <expression> 4. <expression> --> <expression> - <expression> 5. <expression> --> <expression> \* <expression> 6. <expression> --> <expression> / <expression>   The only non-terminal symbol in this grammar is <expression>, which is also the start symbol. The terminal symbols are {+,-,\*,/,(,),number}.   1. Back us Naur Form 2. Invented by John Backus to describe ALGOL 58 syntax.  * BNF (Backus-Naur Form) is equivalent to context-free grammars used for describing syntax. * A metalanguage is a language used to describe another language “Ex: BNF.” * In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called non-terminal symbols)   <assign> →<var> = <expression>   * This is a rule; it describes the structure of an assignment statement * A rule has a left-hand side (LHS) “The abstraction being defined” and a right-hand side (RHS) “consists of some mixture of tokens, lexemes and references to other abstractions”, and consists of terminal and non-terminal symbols.   Example: total = sub1 + sub2   * A grammar is a finite nonempty set of rules and the abstractions are called non-terminal symbols or simply non-terminals. * The lexemes and tokens of the rules are called terminal symbols or terminals. * A BNF description, or grammar, is simply a collection of rules. * An abstraction (or non-terminal symbol) can have more than one RHS   <stmt> →<single\_stmt> | begin <stmt\_list> end  Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical OR  Parse Trees (**[**RGPV/ June 2013 (10)]  (**[**RGPV/ June 2011 (5)]   * Hierarchical structures of the language are called parse trees. * A parse tree for the simple statement A = B + const.     An example grammar:  <program> →<stmts>  <stmts> →<stmt> | <stmt> ; <stmts>  <stmt> →<var> = <expr>  <var> →a | b | c | d  <expr> →<term> + <term> | <term> - <term>  <term> →<var> | const  <program>  <stmts>    <stmt>  <var> = <expr>  a <term> + <term>  <var> const  b  Ambiguity  •A grammar is ambiguous if it generates a sentential form that has two or more distinct parse trees.  •Ex: Two distinct parse trees for the same sentence, const – const / const  <expr> →<expr> <op> <expr> | const  <op> →/ | -  <expr> <expr>  <expr> <op> <expr> <expr> <op> <expr>  <expr> <op> <expr> <expr> <op> <expr>  const - const / const const - const / const  **Unit-01/Lecture-08**  **Syntactic Elements of a Language**   * *Character set*. The choice of character set is one the first to be made in designing language syntax. * *Identifiers*. The basic syntax for identifiers—a string of letters and digits beginning with a letter—is widely accepted. * *Operator symbols.* + And – are special characters that most language used to represent the two basic arithmetic operations. * *Keywords and reserved words*. A *keyword* is an identifier used as a fixed part of the syntax statement. It is also a reserved word if it may also be used as a programmer –chosen identifier. * *Noise words*. These are optional words that are inserted in statements to improve readability. COBOL provides many options. * *Comments*. In relation to computers, also called *remark*. Text embedded in a computer program for documentation purposes. Comments usually describe what the program does, who wrote it, why it was changed, and so on. Most programming languages have syntax for creating comments so that the comments will be ignored by the compiler or assembler. * Blank (spaces) * Delimiters and brackets- A syntax element used to mark the beginning or end of some syntax unit such as a statement or expression.   Brackets are paired delimiters  Parenthesis  Begin...end pairs   * Free- and fixed-field expressions- syntax is free-field if program statements may be written anywhere on an input line without regard for positioning on the line. Syntax is fix field if the positioning on an input line conveys information. * Expressions. An expression is a piece of a statement that describes a series of computations to be performed on some of the program’s variables, such as X + Y/Z, in which the variables are X, Y, and Z and the computations are addition and division. * Statements. A statement in a program is a basic sentence that expresses a simple idea—its purpose is to give the computer a basic instruction. Statements define the types of data allowed, how data are to be manipulated, and the ways that procedures and functions work   **General Syntactic Criteria**   1. Readability- Program is readable if the underlying structure of the algorithm and data represented by the programmer is apparent from an inspection of the program test. A readable program is often said to be self documenting that is it is understandable without any separate documentation. 2. Writ ability- The syntactic features that make a program easy to write are often in conflict with those features that make it easy to read. The use of structured statements, simple natural statement formats mnemonic operation symbols and unrestricted identifiers usually make program writing easier by allowing the natural structure of the problem, algorithms and data are to be directly represented in the program. 3. Ease of Verifiability- Related to readability and writ ability is the concept of program correctness or program verification. Understanding each programming language statement is relatively easy, but the overall process of creating the correct program is difficult. 4. Ease of Translation- The programs should be easily translated into executable form. Ease of translation relates to the need of the translator that processes the written program. The key to easy translation is the regularity of the structure. 5. Lack of Ambiguity- A language definition ideally provides a unique meaning for every syntactic construct that a programmer may write. An ambiguous construction allows two or more different interpretations.   Ex- C allows two different forms of conditional statements   1. if <condition 1>   Statement 1  else  Statement 2   1. if <condition 1>   if <condition 2>  Statement 1  else  Statement 2  In example 1, if condition 1 is true statement1 is executed else statement2 is executed.  The example2 is dangling else. This statement form is ambiguous, because it is not clear the else statement belongs to which if statement.  **Unit-01/Lecture-09**  **Stages in Translation**  Analysis of the source program   * Lexical analysis (scanning) – identifying the tokens of the programming language:  the keywords, identifiers, constants and other symbols appearing in the language.   In the program  void main()  {  printf("Hello World\n");  }  The tokens are  void, main, (,), {, printf, (, "Hello World\n", ), ;, }   * Syntactic analysis (parsing) – determines the *structure* of the program,  as defined by the language *grammar*. * Semantic analysis - assigns meaning to the syntactic structures   Example:  int variable1;  The meaning is that the program needs 4 bytes in the memory to serve  as a location for *variable1*. Further on, a specific set of operations only can be used  with *variable1*, namely integer operations.  The semantic analysis builds the bridge between analysis and synthesis.  Basic semantic tasks:   * 1. Symbol–table maintenance   2. Insertion of implicit information   3. Error detection   4. Macro processing and compile-time operations   The result of the semantic analysis is an internal representation, suitable to be used  for code optimization and code generation.  Synthesis of the object program  The final result is the executable code of the program. It is obtained in three main steps:  Optimization - Code optimization involves the application of rules and algorithms  applied to the intermediate and/ or assembler code with the purpose to make it more efficient,  i.e. faster and smaller.  Code generation - generating assembler commands with relative memory addresses  for the separate program modules - obtaining the object code of the program.  Linking and loading - resolving the addresses - obtaining the executable code of the program.  For Example, The statement A=B+C+D  The semantic analyzer may generate the following intermediate code  Temp1 = B+C  Temp2 = Temp1 + D  A = Temp2  The code generators may generate the following inefficient code  load register with B  add C to register  store register in Temp1  load register with Temp1  add D to register  store register in Temp2  load register with Temp2  store register in A  The register storage and loading is redundant since all data can be kept in the register before storing the result in A. |