

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 FORTRAN 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, LISP 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.

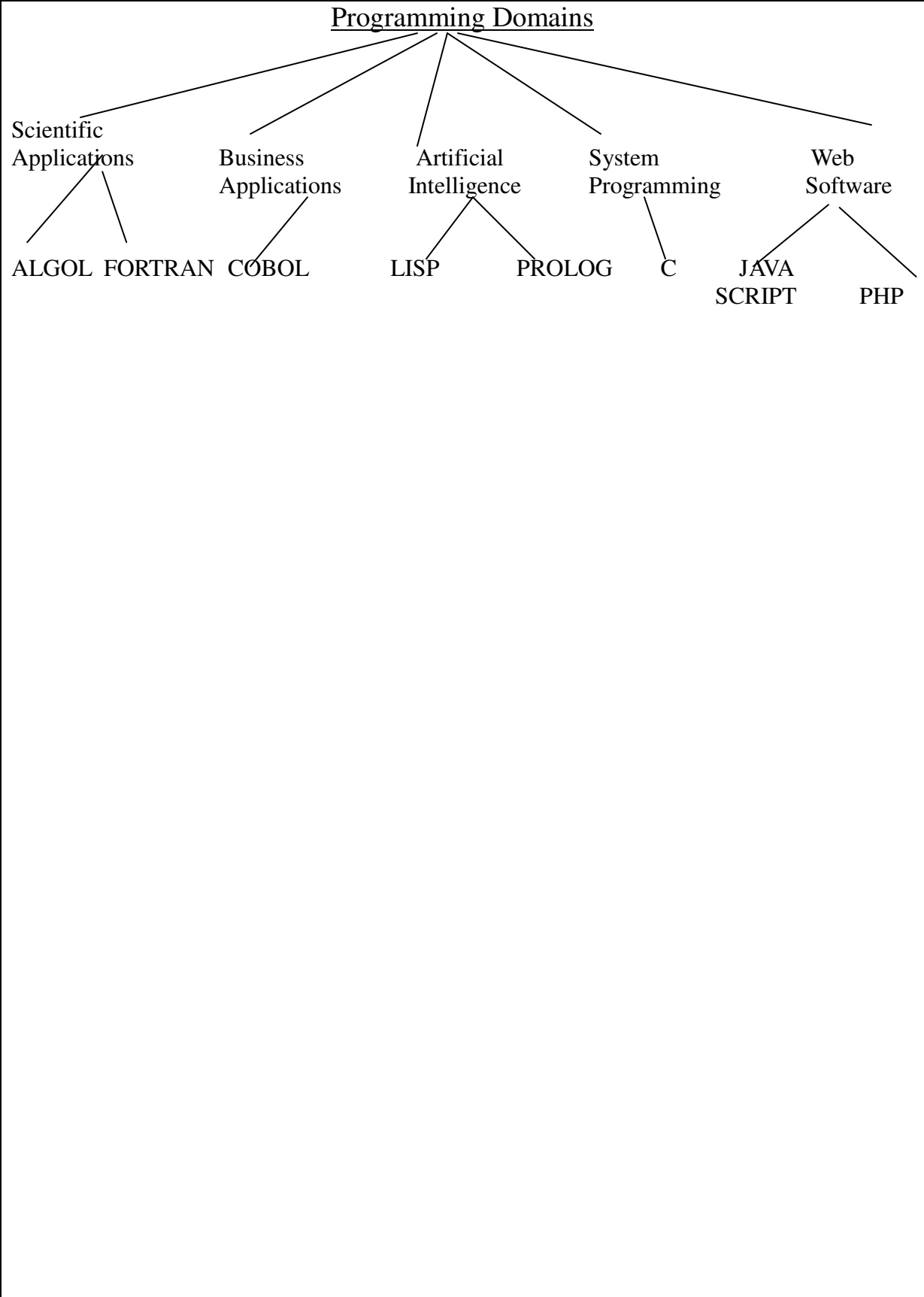
C was developed in 1972 by Dennis Ritchie. All of the features of Pascal, including the new ones such as the CASE statement are available in C. C uses pointers extensively and was built to be fast and powerful at the expense of being hard to read. But because it fixed most of the mistakes Pascal had, 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 Stroustrup 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.



Unit-01/Lecture-02

LANGUAGE EVALUATION CRITERIA

Table 1.1 Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

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)

- 5) 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 $\text{count}++$ is more convenient and shorter than $\text{count}=\text{count}+1$. 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

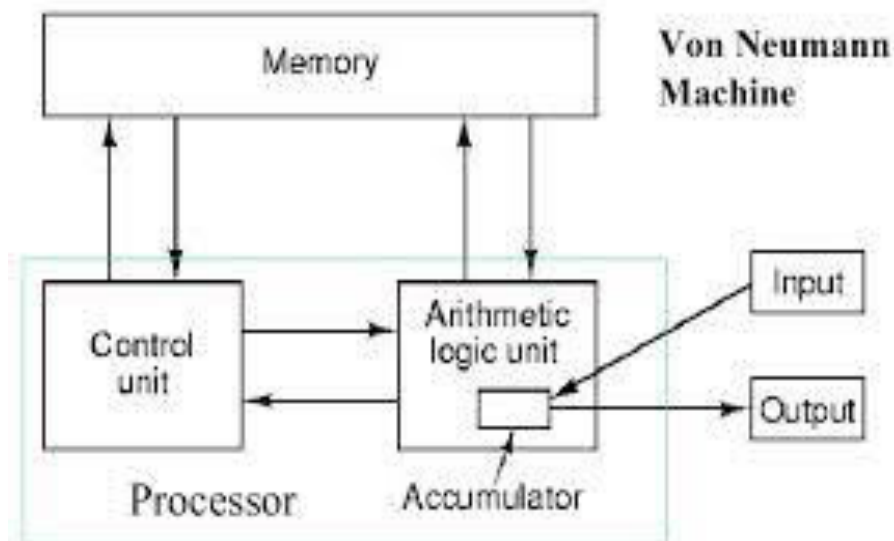
The ultimate cost of a programming language is a function of many of its characteristics

- a) The cost of training programmers
- b) The cost of writing programs
- c) The cost of compiling programs
- d) The cost of executing programs
- e) The cost of Language implementation System
- f) The cost of poor reliability
- g) 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.



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-

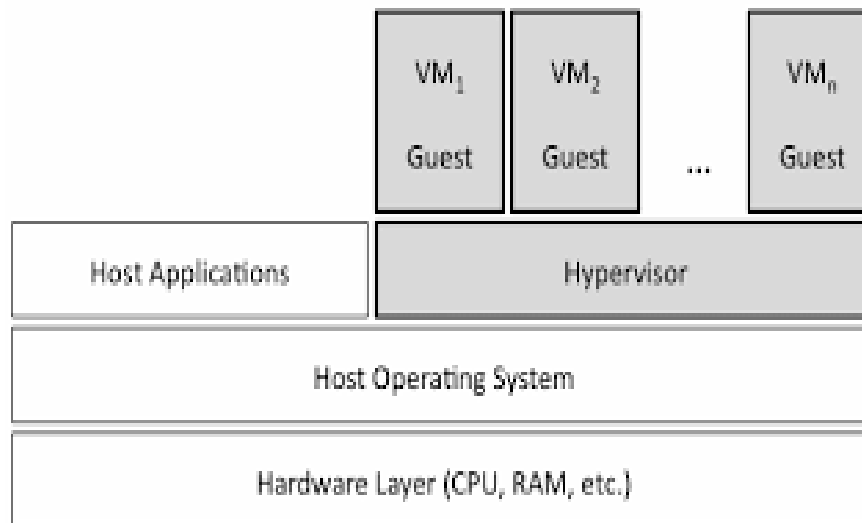
defined data type. Abstraction separates the interface from implementation. Encapsulation insulates the data by wrapping them up using various methods, and allows the internal implementation of a class to be shared by specifying what information in an object can be exchanged with others. Inheritance 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.



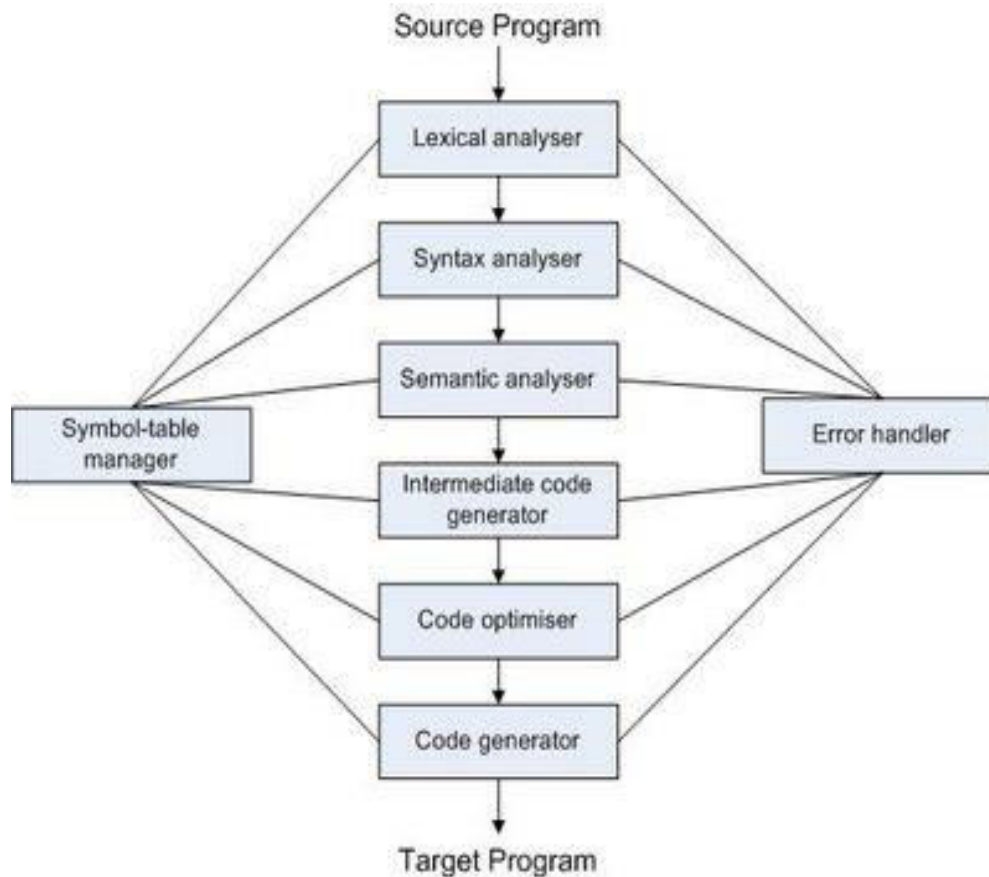
Specialized software called a hypervisor emulates the PC client or server's CPU, memory, hard disk, network and other hardware resources completely, enabling virtual machines to share the resources. The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other. Virtual 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).



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 into 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.

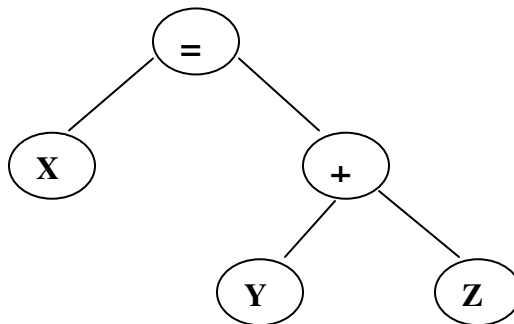
The output of lexical analyzer goes as input to syntax analyzer phase.

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
 - i. Every identifier is declared before it is used.
 - ii. No identifier is used in an inappropriate context (e.g. adding a string to an integer)
 - iii. 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

- i. Every identifier is declared before it is used.
- ii. No identifier is used in an inappropriate context (e.g. adding a string to an integer)
- iii. 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

2. 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 ...
```

3. 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}.

2) Backus Naur Form

3) 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)

$$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$$

- 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

$$\langle \text{stmt} \rangle \rightarrow \langle \text{single_stmt} \rangle \mid \text{begin } \langle \text{stmt_list} \rangle \text{ end}$$

Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical OR

Parse Trees

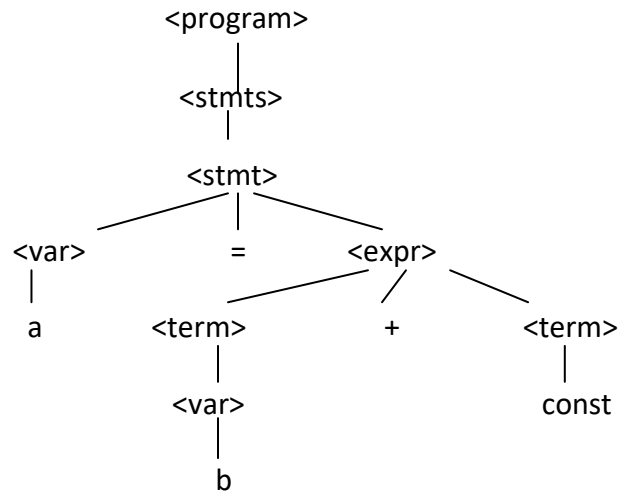
([RGPV/ June 2013 (10)])

([RGPV/ June 2011 (5)])

- Hierarchical structures of the language are called parse trees.
- A parse tree for the simple statement $A = B + \text{const}$.

An example grammar:

$\langle \text{program} \rangle \rightarrow \langle \text{stmts} \rangle$
 $\langle \text{stmts} \rangle \rightarrow \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle$
 $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
 $\langle \text{var} \rangle \rightarrow a \mid b \mid c \mid d$
 $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle - \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{var} \rangle \mid \text{const}$



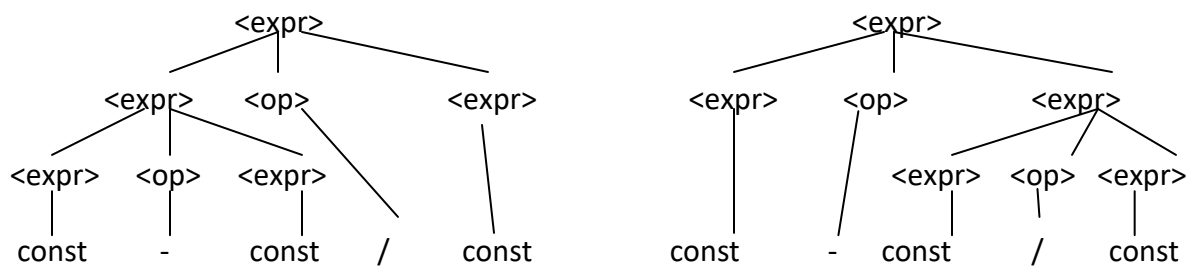
Ambiguity

•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, $\text{const} - \text{const} / \text{const}$

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \text{const}$

$\langle \text{op} \rangle \rightarrow / \mid -$



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

- (i) if <condition 1>
 Statement 1
 else
 Statement 2
- (ii) 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.

UNIT - II

Data Types

Unit-02/Lecture-01

A data type defines a collection of data values and a set of predefined operations on those values. Types provide implicit context. Compilers can infer information, so programmers write less code.

e.g., the expression $a+b$ in Java may be adding two integer, two floats or two strings depending on context.

Types provide a set of semantically valid operation. Compilers can detect semantic mistakes

e.g., Python's list support `append ()` and `pop ()`, but complex numbers do not.

Design Issues for all Data Types

- How is the domain of values specified?
- What operations are defined and how are they specified?
- What is the syntax of references to variables?

Typical primitives include: Boolean, Character, Integral Type, Fixed point type, Floating point type.

Primitive Data Types

- Almost all programming languages provide a set of primitive data types
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require little non-hardware support

Primitive Data Types: Integer

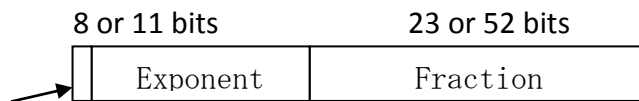
- The most common primitive numeric data type is integer.
- Many computers support several sizes of integers, and these capabilities are reflected in some programming languages For example, Ada allows these: short integer, integer and long integer.

- An integer is represented by a string of bits, with the leftmost representing the sign bit.

FLOATING-POINT

- Floating point data types model real numbers, but the representations are only approximations for most real values.
- On most computers, floating-point numbers are stored in binary, which exacerbates the problem
- Floating-point values are represented as fractions and exponents
- Most new computers use the standard IEEE format
- Most languages use float and double as floating-point types
- The float is stored in 4 bytes of memory
- The double has twice as big of storage.

IEEE Floating Point Standard 754



Sign bit

Precision - The accuracy of the fractional part of a value, measured as the number of bits

Range – a combination of the range of fractions and the range of exponents.

Primitive Data Types: Complex

- Represented as an ordered pair of floating numbers
- Python specifies the imaginary part by following it with a j or $J(7 + 3j)$
- Languages that support a complex type include operations for arithmetic on complex values

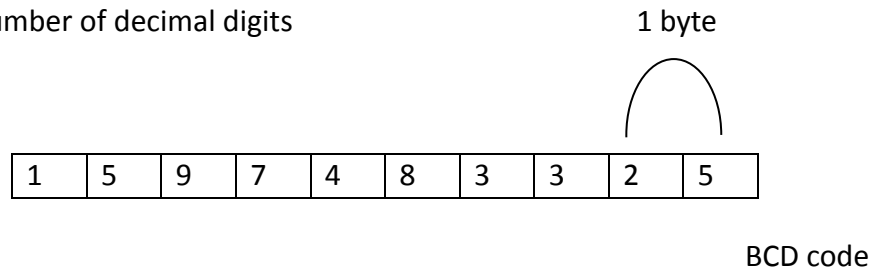
Primitive Data Types: Decimal

- For business applications (money)

–Essential to COBOL

–C# offers a decimal data type

- Store a fixed number of decimal digits



- Advantage: accuracy
- Disadvantages: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes.
- Advantage: readability (compared with using integers to represent switches/flags)

Primitive Data Types: Character

- Stored as numeric coding
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode

–Includes characters from most natural languages

–Originally used in Java

–C# and JavaScript also support Unicode

Unit-02/Lecture-02

CHARACTER STRING TYPES

- A character string type is one in which the values consist of sequences of characters.
- They are used to label output, and input and output of all kinds.
- Design Issues-
 - Is it a primitive type or just a special kind of array?
 - Is the length of objects fixed or variable?
- Operations-
 - Assignment
 - Comparison (=, > etc)
 - Concatenation
 - Substring reference
 - Pattern matching

Array Types

An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kinds of slices allowed?

Array Indexing

- Indexing (or subscripting) is a mapping from indices to elements

array_name (index_value_list) → element

- Index Syntax

–FORTRAN, PL/I, Ada use parentheses

- Ada explicitly uses parentheses to show uniformity between array references and

function calls because both are mappings

–Most other languages use brackets

Associative Arrays

An associative array is an array with strings as index. This stores element values in association with key values rather than in a strict linear index order. In associative arrays the keys are not integers but strings.

The associative arrays are very similar to numeric arrays in term of functionality but they are different in terms of their index. Associative array will have their index as string so that you can establish a strong association between key and values.

To store the salaries of employees in an array, a numerically indexed array would not be the best choice. Instead, we could use the employees names as the keys in our associative array, and the value would be their respective salary.

```
$aFruit = array(
    'color' => 'red'
    , 'taste' => 'sweet'
    , 'shape' => 'round'
    , 'name' => 'apple'
    ,      4 // key will be 0
);
```

Is equivalent with:

```
$aFruit['color'] = 'red';
$aFruit['taste'] = 'sweet';
$aFruit['shape'] = 'round';
$aFruit['name'] = 'apple';
$aFruit[] = 4; // key will be 0
```

User Defined Ordinal Types

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers. Two common kinds:

- Enumeration types
- Sub range types

Enumeration Types

The user enumerates all of the possible values, which are symbolic constants

Design Issue: Should a symbolic constant be allowed to be in more than one type definition?

Examples:

Pascal--cannot reuse constants; they can be used for array subscripts, for variables, case selectors; no input or output; can be compared

Ada--constants can be reused (overloaded literals); can be used as in Pascal; input and output supported

C and C++--like Pascal, except they can be input and output as integers

Java does not include an enumeration type

In other words, an enumeration is a list of values.

```
type TWeekDays = (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
```

Once we define an enumerated data type, we can declare variables to be of that type:

```
var SomeDay : TWeekDays;
```

Sub range Types

- An ordered, contiguous subsequence of another ordinal type
- Design Issue: How can they be used?

Examples:

- Pascal--sub range types behave as their parent types; can be used as for

variables and array indices

```
type pos = 0 .. MAXINT;
```

- Ada--subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants

```
subtype Pos_Type is
```

```
Integer range 0...Integer 'Last;
```

Unit-02/Lecture-03

Record

A record is a data structure composed of a fixed number of components of different types. The components may be heterogeneous, and they are named with symbolic names.

Specification of attributes of a record:

Number of components

Data type of each component

Selector used to name each component.

Implementation:

Storage: single sequential block of memory where the components are stored sequentially.

Selection: provided the type of each component is known, the location can be computed at translation time.

Note on efficiency of storage representation:

For some data types storage must begin on specific memory boundaries (required by the hardware organization). For example, integers must be allocated at word boundaries (e.g. addresses that are multiples of 4). When the structure of a record is designed, this fact has to be taken into consideration. Otherwise the actual memory needed might be more than the sum of the length of each component in the record. Here is an example:

```
struct employee
```

```
{ char Division;
```

```
  int IdNumber; };
```

The first variable occupies one byte only. The next three bytes will remain unused and then the second variable will be allocated to a word boundary. Careless design may result in doubling the memory requirements.

Unions

A union is an aggregate variable that can declare several data types, but only store the value of one variable at any one time; each data type shares the same area of memory. The declaration of a union is similar to that of a structure:

```
union [union_tag]
{
    data_type variable_1;
    data_type variable_2;
    data_type variable_3;
    .
    .
    .
} {union_variable_name};
```

As with a structure, a union_tag is optional if a union_variable_name is present. The compiler allocates only the storage space of the largest data_type declared.

```
union Data
{
    char str[6];
    int y;
    long z;
    float x;
    double t;
} var;
```

The union variable var is given eight(8) bytes of storage; all the elements start at the same address in memory; the compiler gives the amount of storage needed by the largest data

type, in this case double which is eight bytes.

Accessing Members of a union

To access a member of a union the dot operator is used the same as with structures.

```
var.x = 123.45;
```

Unions passed to functions exactly as a structure would be passed.

Pointers and Reference Types

Pointers and references are essentially variables that hold memory addresses as their values. Pointers and references hold the addresses in memory of where we find the data of the various data types that we have declared and assigned. The two mechanisms, pointers and references, have different syntax and different traditional uses.

A pointer is declared as:

```
<Pointer type> *<pointer-name>
```

In the above declaration:

1. Pointer-type: It specifies the type of pointer. It can be int, char, float etc. This type specifies the type of variable whose address this pointer can store.
2. Pointer-name: It can be any name specified by the user. Professionally, there are some coding styles which every code follows. The pointer names commonly start with 'p' or end with 'ptr'

```
int x = 1, y = 2;
```

```
int *ip;
```

```
ip = &x;
```

A reference is treated **exactly** as if we had used the original variable in its place. For example, if we assign to a reference, it is as if we assigned to the original variable:

```
int x = 5;
```

```
int &y = x; // y is an alias for x
```

```
y = 6; // now x == 6
```

Unit-02/Lecture-04

Names

Names are also associated with labels, subprograms, formal parameters and other program constructs.

Name Forms-

A name is a string of characters used to identify some entity in a program. The earliest programming languages used single character names.

FORTRAN1 broke this tradition by allowing names up to 6 characters long. FORTRAN90 and C allow up to 31 characters names. ADA has no length limit.

We can use some connector characters like underscore (`_`) in the string. Some languages like C, C++, a Java are case sensitive. That is, these languages differentiate between uppercase and lowercase letters. Ex- SUN, Sun and sun are distinct in C++.

Special words- Special words are used in programming languages to make programs more readable by naming actions to be performed.

Keyword-

A keyword is a word of programming languages that is special only in certain contexts. FORTRAN is one of the languages whose special word are keywords. For Ex- In FORTRAN, the word `real` when found in the beginning of a statement and followed by a name is considered a keyword that indicates the statement is a declarative statement. However if the word `real` is followed by an assignment operator it considered as a variable name.

Ex- `REAL APPLE`

`REAL = 3.4`

Reserved Word- A reserved word is a special word of a programming language that cannot be used as name.

Ex- The names `printf` and `scanf` are the reserved words which are defined in `<stdio.h>`

VARIABLES

A variable is a symbolic name for (or reference to) information. The variable's name

represents what information the variable contains. They are called variables because the represented information can change but the operations on the variable remain the same. In general, a program should be written with "Symbolic" notation, such that a statement is always true symbolically.

For example, if we want to find the sum of any two numbers we can write:

```
result = a + b;
```

Both 'a' and 'b' are variables. They are symbolic representations of any numbers. For example, the variable 'a' could contain the number 5 and the variable 'b' could contain the number 10. During execution of the program, the statement "a + b" is replaced by the Actual Values "5 + 10" and the result becomes 15.

Variable Properties

There are 6 properties associated with a variable.

1. A Type
2. A Value
3. A Scope
4. A Life Time
5. A Location (in Memory)

Properties

1. A Name

The name is Symbolic. It represents the "title" of the information that is being stored with the variable.

The name is perhaps the most important property to the programmer, because this is how we "access" the variable. Every variable must have a unique name!

2. A Type

The type represents what "kind" of data is stored with the variable.

In C, Java etc, the type of a variable must be explicitly declared when the name is created.

3. A Value

A variable, by its very name, changes over time. Thus if the variable is jims_age and is assigned the value 21. At another point, jims_age may be assigned the value 27.

Default Values

Most of the time, when we "create a variable" we are primarily defining the variable's name and type. Often we will want to provide an initial value to be associated with that variable name. If you forget to assign an initial value, then various rules "kick in" depending on the language.

Example

```
age = 20; //this creates a variable named age with the value 20 (and type Number (double))
```

```
printf('age is %f\n'); // answer: age is 20
```

4. A Scope

Good programs are "chopped" into small self-contained sections (called functions). A variable that is seen and used in one function is NOT available in another section. This allows us to reuse variable names, such as `age`. In one function `'age'` could refer to the age of a student, and in another function `'age'` could refer to the vintage of a fine wine.

Further, this prevents us from "accidentally" changing information that is important to another part of our program.

5. A Life Time

The life time of a variable is strongly related to the scope of the variable. When a program begins, variables "come to life" when the program reaches the line of code where they are "declared". Variables "die" when the program leaves the "scope" of the variable.

6. A Location (in Memory)

Generally we don't have to worry too much about where in the computer hardware the variable is stored. The computer does this for us. But we should be aware that a "bucket" or "envelope" exists in the hardware for every variable we declare. In the case of an array, a "bunch of buckets" exist. Every bucket can contain a single value.

Unit-02/Lecture-05

Concept of Binding

A binding is an association, such as between an attribute and an entity, or between an operation and a symbol. Binding time is the time at which a binding takes place

For example, in C the binding time for a variable type is when the program is compiled (because the type cannot be changed without changing and recompiling the program), but the value of the variable is not bound until the program executes (that is, the value of the variable can change during execution).

Some additional examples of attributes are:

- the meaning of a keyword such as if
- the operation associated with a symbol such as +
- the entity (variable, keyword, etc.) represented by an identifier
- the memory location for the value of an identifier

The most common binding times for attributes are (in chronological order):

1. Language definition
2. Language implementation
3. Program translation (compile time)
4. Link edit
5. Load
6. Program execution (run time)

Classes of Binding Times

1. Execution Time (Run time)

This includes binding performed during program execution. Ex- Binding of variables to their values as well as bindings of variables to particular storage locations.

2. Translation Time (Compile Time)-
 - a) Binding chosen by the programmer

While writing a program, a programmer gives choice for variable names, types for variables, program statement structures and so on that represents binding during translation.

b) Binding chosen by the translator

Some bindings are chosen by translator. Ex- The relative location of a data object in the storage allocated for a procedure, how arrays are stored, how descriptors for the arrays, if any are created all such decisions are made by the language translator.

c) Binding chosen by the Loader- A program usually consists of several subprograms that must be merged into a single executable program. The translator binds variables to addresses within the storage designated for each subprogram.

3. Language Implementation Time- Some aspects of a language definition may vary between implementations. For Ex- The details associated with the representation of numbers and arithmetic operation may be determined by the underlying computer hardware.

4. Language Definition Time- Most of the structure of programming language is fixed at language definition time. Ex-Different data types, data structure types, control elements, program structure and so on are all fixed at language definition time.

Types of Binding

A binding is static if it first occurs before run time and remains unchanged throughout program execution. A binding is dynamic if it first occurs during execution or can change during execution of the program.

Static Type Binding

- If static, the type may be specified by either an explicit or an implicit declaration.

Variable Declarations

- An explicit declaration is a program statement used for declaring the types of variables.
- An implicit declaration is a default mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.

EX: –In FORTRAN, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention: I, J, K, L, M, or N or their lowercase versions is implicitly declared to be Integer type; otherwise, it is implicitly declared as Real type.

–Advantage: writ ability

–Disadvantage: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.

–In FORTRAN, variables that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement

- Ex, JavaScript

list = [2, 4.33, 6, 8]; ->single-dimensional array

list = 47; -> scalar variable

–Advantage: flexibility (generic program units)

–Disadvantages:

–High cost (dynamic type checking and interpretation)

- Dynamic type bindings must be implemented using pure interpreter not compilers.

- Pure interpretation typically takes at least ten times as long as to execute equivalent machine code.

–Type error detection by the compiler is difficult because any variable can be assigned a value of any type.

- Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.

- Ex: i, x -> Integer

 y ->floating-point array

 i = x ->what the user meant to type

 i = y -> what the user typed instead

•No error is detected by the compiler or run-time system. `i` is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Unit-02/Lecture-06**TYPE CHECKING**

Type systems are the biggest point of variation across programming languages. Even languages that look similar are often greatly different when it comes to their type systems.

Definition: A type system is a set of types and type constructors (integers, arrays, classes, etc.) along with the rules that govern whether or not a program is legal with respect to types (i.e., type checking). For example, C++ and Java have similar syntax and the control structures. They even have a similar set of types (classes, arrays, etc.). However, they differ greatly with respect to the rules that determine whether or not a program is legal with respect to types.

As an example, one can do this in C++ but not in Java:

```
int x = (int) "Hello";
```

In other words, Java's type rules do not allow the above statement but C++'s type rules do allow it. Why do different languages use different type systems? The reason for this is that there is no one perfect type system. Each type system has its strengths and weaknesses. Thus, different languages use different type systems because they have different priorities. A language designed for writing operating systems is not appropriate for programming the web; thus they will use different type systems. When designing a type system for a language, the language designer needs to balance the tradeoffs between execution efficiency, expressiveness, safety, simplicity, etc. In other words, the type system affects many of the characteristics. Thus, a good understanding of type systems is crucial for understanding how to best exploit programming languages.

Type Conversion and Coercion**Coercion-**

Implicit type conversion, also known as coercion, is an automatic type conversion by the compiler. Some languages allow, or even require compilers to provide coercion.

In a mixed type expression, a subtype s will be converted to a super type t or some subtypes s_1, s_2, \dots will be converted to a super type t (maybe none of the s_i is of type t) at runtime so that the program will run correctly. For example:

```
double d;
```

```

long l;

int i;

if (d > i) d = i;

if (i > l) l = i;

if (d == l) d *= 2;

```

is legal in a C language program. Although *d*, *l* and *i* belong to different datatypes, they will be automatically converted to the same data type each time a comparison or assignment is executed.

Type Casting

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

Used when the programmer wants to explicitly convert one data type to another. It shows the programmer's intention as being very clear.

```

i = 5.2 / f; // warning "loss of precision"

i = int(5.2 / f) // no warning but still loss

f = float(3 * i)

```

TYPE COMPATIBILITY

Compatible Types:

A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type.

Type compatibility is also called conformance or equivalence.

There are two type compatibility methods:

♣ Name compatibility (also called strict compatibility):

Two variables can have compatible types only if they are in either the same declaration or in

declarations that use the same type name.

It is highly restrictive.

It is easier to implement.

Ada uses name compatibility.

♣Structure type compatibility

Two variables have compatible types if their types have identical structures.

It is more flexible.

It is difficult to implement: compare the whole structure instead of just names.

Two types are structurally compatible if:

1. T1 is name compatible with T2; or
2. T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

Examples:

1. Two records or structure types compatible if they have same structure but different field names?
2. Two single-dimensioned array types in a Pascal or Ada program are compatible if they have the same element type but have different subscript ranges?
3. C uses structural compatibility except for structures.

Unit-02/Lecture-07

Named Constants

A named constant is a variable that is bound to a value only at the time it is bound to a value only at the time it is bound to storage; its value cannot be changed by assignment or by an input statement.

```
Const int Max =30;
```

Here Max is a constant of type integer with value30.

Advantages:-

- It improves program readability and reliability. For Ex- using the name 'Pi' in the program is more readable than using the value 3.14.
- Another advantage of named constant is in the program that process a fixed number of data values say 10. Such programs usually use the constant 10 in number of statements like for declaring array subscript ranges, for loop control limits and other uses.

Ex-

```
void main()
{
const int limit=10;
int A[limit], B[limit], C[limit];
cout<<"enter array A elements";
for (i=0; i<limit; i++)
{
        cin>>A[i];
}
cout<<"enter array B elements";
for (i=0; i<limit; i++)
```

```

{
    cin>>B[i];
}

for (i=0; i<limit; i++)

{
    C[i]= A[i]+B[i];
}

cout<<"the addition of array A and B elements";

for (i=0;i<limit;i++)

{
    cout<<C[i];
}
}

```

The advantage of using named constant 'limit' is that when the array limit needs to be changed say from 10 to 100, then only one line is required to be changed, regardless the number of times it is used in the program.

Variable Initialization

The binding of a variable to a value at the time it is bound to storage is called variable initialization.

If the variable is bound to storage statically binding and initialization occur before runtime. If storage binding is dynamic, initialization is also dynamic.

Ex- int fact =1;

Here the variable 'fact' is initialized with the value 1 statically.

Conditional Statements

A conditional statement is one which makes the computer compare two or more variables in some way and decide that the outcome is either 'true' or 'false', and then feeds this into a function such as 'if' or 'while'.

The If statement

If the condition expression evaluates to true, the statement is ignored.


```
Syntax          if(condition)
                  Statement;
```

```
Ex; -          if (A < B)
{              printf ("A is smaller element");    }
```

If..Else Statement

It is used to test the condition that has true and false part.

```
Syntax: -      if (condition)
                  Statement1;
                  else
                  Statement2;
```

```
Ex-           if (A < B)
                {  printf ("A is smaller element");
                }
                else
                {
                printf "B is smaller element");
                }
```

Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax:

The syntax for a switch statement in C programming language is as follows:

```
switch(expression){  
  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    /* you can have any number of case statements */  
  
    default : /* Optional */  
        statement(s);  
}  
}
```

The?: operator

It is shorthand method for specifying

(If expression)? (Evaluate if true): (else evaluate this)

This reduces the readability of program. This does not in any way speed up execution time.

Unit-02/Lecture-08

LOOPS

Loop statements are used to perform some action repeatedly. Different types of loop control statement used in C language are-

- 1) While statement – The while statement is used to carry out looping operations in which a group of statements is executed repeatedly until some condition has been satisfied.

while (expression)

{

Statement

}

Ex- To print numbers between 1 to 100 using while

```
#include<stdio.h>
```

```
void main()
```

```
{
```

```
    int i=1;
```

```
    while (i<=100)
```

```
    {
```

```
        printf("\n %d",i);
```

```
        i++;
```

```
    }
```

```
}
```

- 2) The do while statement- When a loop is constructed using the while statement the test for continuation of the loop is carried out at the beginning of each pass. Sometime it is desirable to have a loop with the test for continuation at the end of each pass.

do

{

Statement

```
} while (expression);
```

The statement will be executed repeatedly, till the value of the expression is true.

Ex-

```
void main()
{
Int i=1;
do
{
                printf( "%d", i);
                i++;
} while (i<=100);
```

3) The for statement

The for statement is the most commonly used looping statement in C

```
for (expression1; expression2; expression3)
```

```
Statement;
```

where expression1 is used to initialize some parameter that controls the looping action. Expression2 represents a condition that must be true to continue execution and expression3 is used to alter the value of the parameter.

Ex-

```
void main()
{
Int i=1;

    For ( i=1;j<=100; i++)
    {
                Printf("%d", i);
    }
}
```

Unit-02/Lecture-09

SEQUENCE CONTROL

Control Structure in a PL provides the basic framework within which operations and data are combined into a program and sets of programs.

Sequence Control-> Control of the order of execution of the operations

Data Control-> Control of transmission of data among subprograms of program

Sequence Control may be categorized into four groups:

1) Expressions–

They form the building blocks for statements. An expression is a combination of variable constants and operators according to syntax of language. Properties as precedence rules and parentheses determine how expressions are evaluated

2) Statements–

The statements (conditional & iterative) determine how control flows from one part of program to another.

3) Declarative Programming–

This is an execution model of program which is independent of the program statements. Logic programming model of PROLOG.

4) Subprograms–In structured programming, program is divided into small sections and each section is called subprogram. Subprogram calls and co-routines, can be invoked repeatedly and transfer control from one part of program to another.

IMPLICIT AND EXPLICIT SEQUENCE CONTROL

Implicit Sequence Control

Implicit or default sequence control structures are those defined by the programming language itself. These structures can be modified explicitly by the programmer.

eg. Most languages define physical sequence as the sequence in which statements are executed.

Explicit Sequence Control

Explicit sequence control structures are those that programmer may optionally use to modify the implicit sequence of operations defined by the language.

eg. Use parentheses within expressions, or goto statements and labels

Sequence Control within Expressions

Expression is a formula which uses operators and operands to give the output value.

i) Arithmetic Expression –An expression consisting of numerical values (any number, variable or function call) together with some arithmetic operator is called “Arithmetic Expression”.

Evaluation of Arithmetic Expression

Arithmetic Expressions are evaluated from left to right and using the rules of precedence of operators. If expression involves parentheses, the expression inside parentheses is evaluated first.

ii) Relational Expressions –An expression involving a relational operator is known as “Relational Expression”. A relational expression can be defined as a meaningful combination of operands and relational operators.

$(a + b) > c$ $c < b$

Evaluation of Relational Expression

The relational operators $<$, $>$, $<=$, $>=$ are given the first priority and other operators ($=$ and $!=$) are given the second priority. The arithmetic operators have higher priority over relational operators. The resulting expression will be of integer type, true = 1, false = 0

iii) Logical Expression –

An expression involving logical operators is called ‘Logical expression’. The expression formed with two or more relational expression is called logical expression.

Ex. $a > b \ \&\& \ b < c$

Evaluation of Logical Expression

The result of a logical expression is either true or false. For expression involving AND ($\&\&$), OR ($\|\|$) and NOT ($!$) operations, expression involving NOT is evaluated first, then the expression with AND and finally the expression having OR is evaluated.

Sequence Control within Expressions

1 Controlling the evaluation of expressions

a) Precedence (Priority)

If expression involving more than one operator is evaluated, the operator at higher level of precedence is evaluated first

b) Associativity

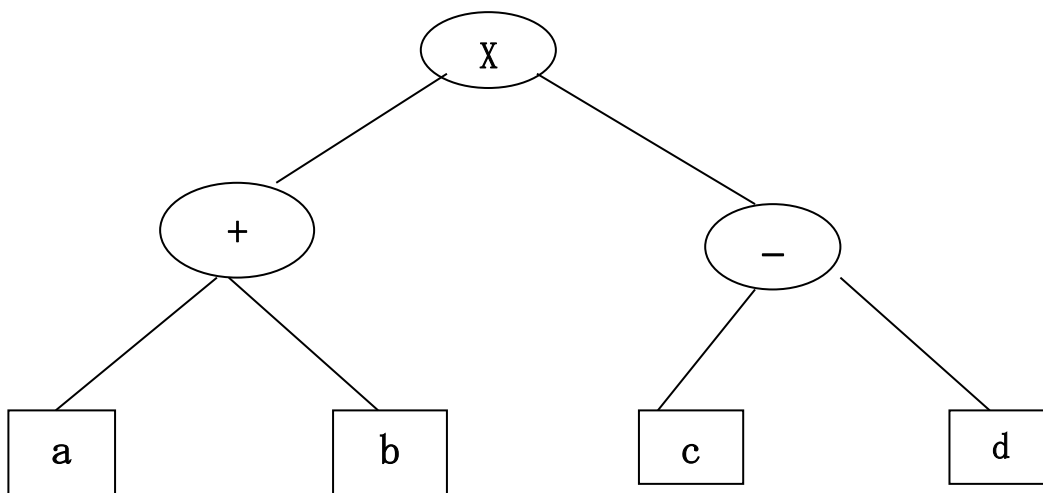
The operators of the same precedence are evaluated either from left to right or from right to left depending on the level. Most operators are evaluated from left to right except

+ (unary plus), -(unary minus) ++, --, !, &

Assignment operators =, +=, *=, /=, %=

Expression Tree

An expression (Arithmetic, relational or logical) can be represented in the form of an "expression tree". The last or main operator comes on the top (root). Example: $(a + b) * (c - d)$ can be represented as



Syntax for Expressions

a) Prefix or Polish notation

Named after polish mathematician Jan Lukasiewicz, refers to notation in which operator symbol is placed before its operands.

*XY, -AB, /*ab -cd

Cambridge Polish- variant of notation used in LISP, parentheses surround an operator and its arguments.

(/*ab)(-cd)

b) Postfix or reverse polish

Postfix refers to notation in which the operator symbol is placed after its two operands.

AB*, XY-

c) Infix notation

It is most suitable for binary (dyadic) operation. The operator symbol is placed between the two operands.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Why pointer is necessary in any programming language?		5 marks
Q.2	Define pointer. Explain various design issues of pointer and pointers in C/C++.		7 marks
Q.3	What is Sequence Control? Explain various categories of sequence control		7 marks
Q.4	Explain the different categories of scalar type variables with their advantages and disadvantages.		7 marks
Q.5	Explain how dynamic type checking affects system performance and improves flexibility with example.		10 marks
Q.6	Differentiate records from variant records with suitable example?		10 marks
Q.7	Explain the following terms: Strong typing Type Coercion Pointers		10 marks

Unit-03/Lecture-01

- Two fundamental abstraction facilities

– Process abstraction

- Emphasized from early days

– Data abstraction

- Emphasized in the 1980s

Fundamentals of Subprograms

General Subprogram Characteristics

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates.

Basic Definition

- A subprogram definition describes the interface to and the actions of the subprogram abstraction
- In Python, function definitions are executable; in all other languages, they are non-executable
- A subprogram call is an explicit request that the subprogram be executed
- A subprogram header is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The parameter profile of a subprogram is the number, order, and types of its parameters
- The protocol is a subprogram's parameter profile and, if it is a function, its return type
- Function declarations in C and C++ are often called prototypes

- A subprogram declaration provides the protocol, but not the body, of the subprogram
- Parameter: A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram
- Argument: An actual parameter represents a value or address used in the subprogram call statement

Argument/Parameter Correspondence

- Positional
- The binding of actual parameters (arguments) to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
- Safe and effective
- Keyword
- The name of the formal parameter to which an actual parameter (argument) is to be bound is specified with the actual parameter
- Advantage: Parameters can appear in any order, thereby avoiding parameter correspondence errors
- Disadvantage: User must know the formal parameter's names

Parameters

Actual/Formal Parameter Keyword Correspondence: Python Example

```
sumer  
  
(length = my_length  
,list = my_list  
,sum = my_sum  
)
```

Parameters: length, list, sum

Arguments: my_length, my_list, my_sum

Unit-03/Lecture-02

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)

- In C++, default parameters must appear last because parameters are position-ally associated

- Variable numbers of parameters

- C# methods can accept a variable number of parameters as long as they are of the same type


—the corresponding formal parameter is an array preceded by parameters.

- In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.

- In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

- **Positional**—The binding of actual parameters to formal parameters is by position. The first actual parameter is bound to the first formal parameter and so forth. It is Safe and effective

Caller `d= f1 (a, b, c)`



`int f1(int x, float y, int z)`

- **Keyword**- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter

In Ada, `Sumer (Length => My_Length, List => My_Array, Sum => My_Sum);`

Advantage: Parameters can appear in any order.

Disadvantage: User must know the formal parameter's names.

Unit-03/Lecture-03

Procedures and Functions

- There are two categories of subprograms
- Procedures are collection of statements that define parameterized computations. These computations are enacted by single call statements. Procedure defines new statements. For Ex- Ada doesn't have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of unavailable sort statement.
- Functions structurally resemble procedures but are semantically modelled on mathematical functions. If a function is a faithful model, it produces no side effects that are; it modifies neither its parameter nor any variables defined outside the function. Such a pure function returns a value that is only desired effect.
- They are expected to produce no side effects.
- In practice, program functions have side effects.

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?

Local Referencing Environments

Variables that are defined inside subprograms are called local variables. Local variables can be either static or stack dynamic “bound to storage when the program begins execution and are unbound when execution terminates.”

Advantages of using stack dynamic:

- a. Stack dynamic variables are essential for recursive subprograms.
- b. Flexibility they provide the subprogram.
- c. Storage for locals is shared among some subprograms.

Disadvantages:

- a. Allocation/de allocation time- There is the cost of the time required to allocate initialize and de allocate such variables for each call to the subprogram.
- b. Indirect addressing “only determined during execution.”- Access to stack dynamic local variables must be indirect whereas accesses to static variables can be direct. This indirectness is because the place in the stack where a particular local variable will reside can be determined only during execution.
- c. Finally when all local variables are stack dynamic, subprograms cannot be history sensitive, that is, they cannot retain data values of local variables between calls.

Ex- for a history- sensitive subprogram is one whose task is to generate pseudo random numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed.

Advantages of using static variables:

- a. Static local variables can be accessed faster because there is no indirection.
- b. No run-time overhead for allocation and de allocation.
- c. Allow subprograms to be history sensitive.

Disadvantages:

- a. Inability to support recursion.
- b. Their storage can't be shared with the local variables of other inactive subprograms.

Ex:

```
int adder( int list[ ], int listlen) {  
  
static int sum = 0;  
  
int count; //count is stack-dynamic  
  
for (count = 0; count < listlen; count++)  
  
sum += list[count];  
  
return sum;  
  
}
```

Here the variable sum is static and count is stack dynamic. Ada, C++, Java and C# have only stack dynamic local variables.

Unit-03/Lecture-04

Parameter Passing Methods

Following are the ways in which parameters are transmitted to and/or from called subprograms

- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

Formal parameters are characterized by one of three distinct semantic models-

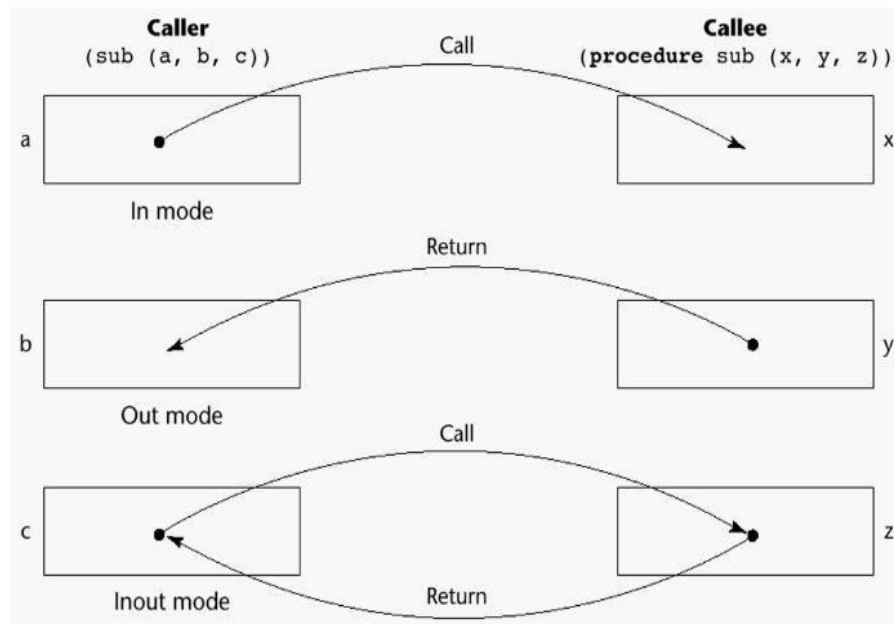
- i. They can receive data from the corresponding actual parameters.
- ii. They can transmit data to the actual parameter.
- iii. They can do both.

These three semantic models are called in mode, out mode, and in out mode respectively.

Data transmission takes place by two ways-

- An actual value is moved.
- An access path is transmitted.

Models of Parameter Passing



1. Pass by Value (In mode)

The value of the actual parameter is used to initialize the corresponding formal parameter

–Normally implemented by copying

–Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)

–Disadvantages (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)

–Disadvantages(if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

2. Pass by Result (out mode)

When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's

actual parameter when control is returned to the caller, by physical move

–Require extra storage location and copy operation

•Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent

the current value of p1

3. Pass by Value Result (In out mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages: –Those of pass-by-result

–Those of pass-by-value

4. Pass by Reference (In out mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages

–Slower accesses (compared to pass-by-value) to formal parameters

–Potentials for unwanted side effects (collisions)

–Unwanted aliases (access broadened)

5. Pass By Name (In out mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

Unit-03/Lecture-05

Overloaded Subprograms

An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment

–Every version of an overloaded subprogram has a unique protocol

–Unique protocol means that the number, order, or types of parameters must differ or the return type must differ

•C++, Java, C#, and Ada include predefined overloaded subprograms and also allow users to write multiple versions of subprograms with the same name.

Because each version of an overloaded subprogram has a unique parameter profile, the compiler can disambiguate occurrences of calls to them by the different type parameters.

But when parameter coercions are allowed, complicate the disambiguation process enormously. The issue is that if no method's parameter profile matches the number and types of the actual parameters in a method call, but two or more methods have parameter profiles that can be matched through coercions which method should be called. A language designer has to decide how to rank all the different coercions so that the compiler can choose the method that best matches the call.

In Ada, the return type of an overloaded function can used to disambiguate calls. Therefore two overloaded functions can have the same parameter profile and differ only in their return types. Ex- If an Ada program has 2 functions named Fun, both of which take an Integer parameter but one returns an Integer and one returns a float the following call would be legal:-

```
A, B: Integer;
```

```
A: = B + Fun (7);
```

In this call, the call to Fun is bound to the version of Fun that returns an Integer because choosing the version that returns a float would cause a type error.

Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Ex- In C++

```
void fun (float b= 0.0);
```

```
void fun();
```

```
.....
```

```
fun ();
```

The call is ambiguous and will cause a compilation error.

Unit-03/Lecture-06

Generic Subprograms

A generic or polymorphic subprogram takes parameters of different types on different activations

- Overloaded subprograms provide ad hoc polymorphism
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism

Examples of parametric polymorphism: C++ template

<Class Type>

```
Type max (Type first, Type second) {
return first > second ? first: second;
}
```

- The above template can be instantiated for any type for which operator > is defined. For example

```
int max (int first, int second) {
return first > second? first : second;
}
```

Another example of Generic Subprogram in C++

A generic subprogram (function) for swapping integer, float and character type elements-

```
# include <iostream.h>
# include <conio.h>
void swap (T &a, T &b)
{
```

```
        T temp;

        temp = a;

        a = b;

        b = temp;

    }

Void main ()

{

    Int x=10, y=20;

    Float a=1.2, b= 2.4;

    Cout<<"Swapping integer values \n";

    Cout<<"values of x and y before swapping\n";

    Cout<<" X="<<x<<"Y="<<y;

    Swap (x,y);

    Cout<<" values of x and y after swapping\n";

    Cout<<" X="<<x<<"Y="<<y;

    Cout<<"Swapping float values \n";

    Cout<<"values of a and b before swapping\n";

    Cout<<" A="<<a<<"B="<<b;

    Swap (a,b);

    Cout<<" values of a and b after swapping\n";

    Cout<<" A="<<a<<"B="<<b;

    getch();

}
```

Unit-03/Lecture-07

Co routines

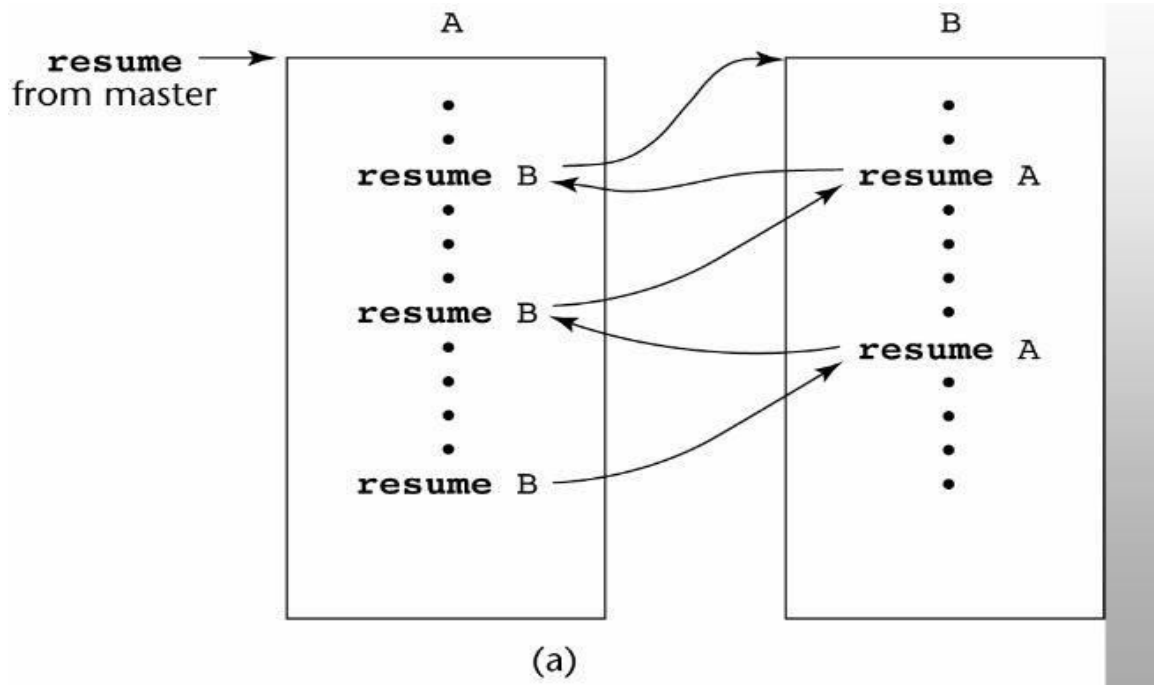
A co routine is a subprogram that has multiple entries and controls them itself

- Also called symmetric control: caller and called co routines are on a more equal basis
- A co routine call is named a resume
- The first resume of a co routine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the co routine.
- Co routines repeatedly resume each other, possibly forever
- Co routines provide quasi-concurrent execution of program units (the co routines); their execution is interleaved, but not overlapped

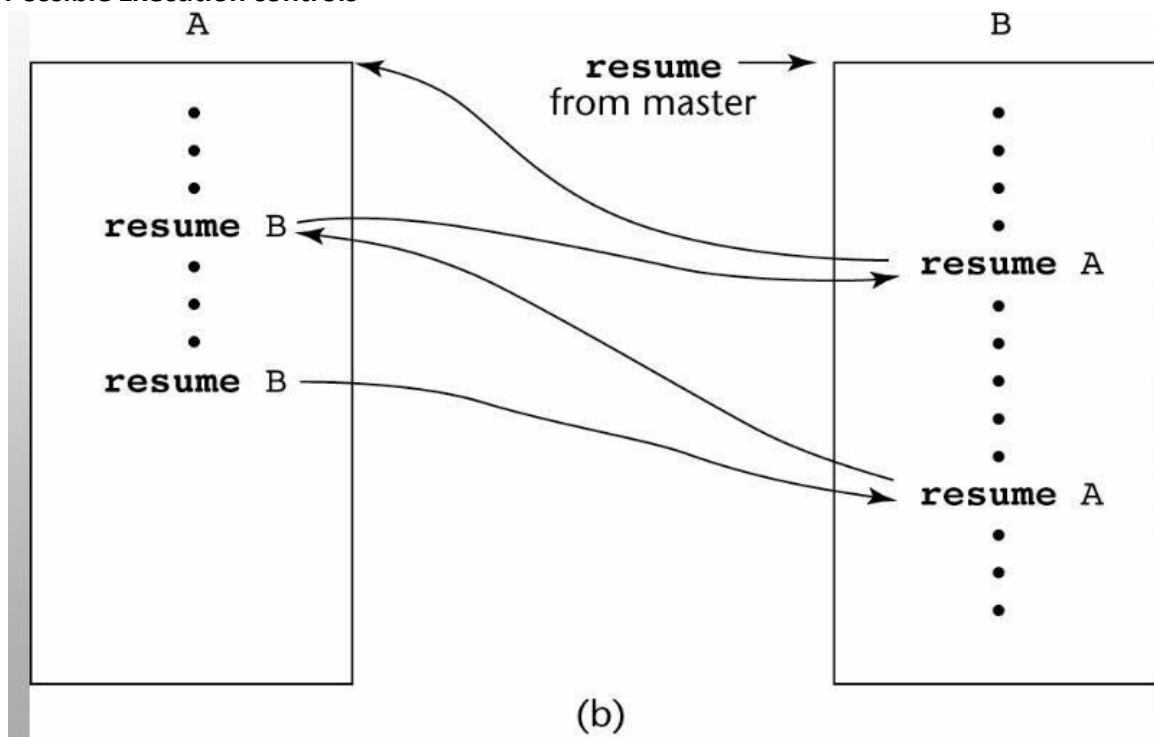
Only one co routine executes at a given time. Rather than executing to their other ends, co routines often partially execute and then transfer control to other routines. When restarted a co routine resumes execution just after the statement it used to transfer control elsewhere. This sort of interleaved execution sequence is related to the way multiprogramming operating systems work. In the case of co routines, this is sometimes called Quasi Concurrency.

Co routines are created in an application by a program unit called Master Unit, which is not a co routine. When created, co routine executes their initialization code and then returns control to that master unit.

Possible Execution controls



Possible Execution controls



Possible Execution Controls with Loops

