CSC305
PROGRAMMING PARADIGM

1.1
Reasons for Studying Concepts of Programming Languages

1. Increased capacity to express ideas

- Available features (control structures, data structures, abstractions, etc.) of a language can limit how programming concepts/ ideas can be expressed in that language.
- Awareness of wide variety of programming language features can reduce such limitations in software development.
- The study of programming language concepts builds an appreciation for valuable language features and encourages programmers to use them.

1.1
Reasons for Studying Concepts of Programming Languages

2. Improved background for choosing appropriate languages

- Experience in many languages can help the programmers to choose more appropriate languages for new projects.

3. Increased ability to learn new languages

- Programming languages (along with software development methodologies/ tools) are constantly evolving.
- Familiarity with fundamental concepts of programming languages makes it easier to learn new languages.

1.1
Reasons for Studying Concepts of Programming Languages

4. Better understanding of the significance of implementation

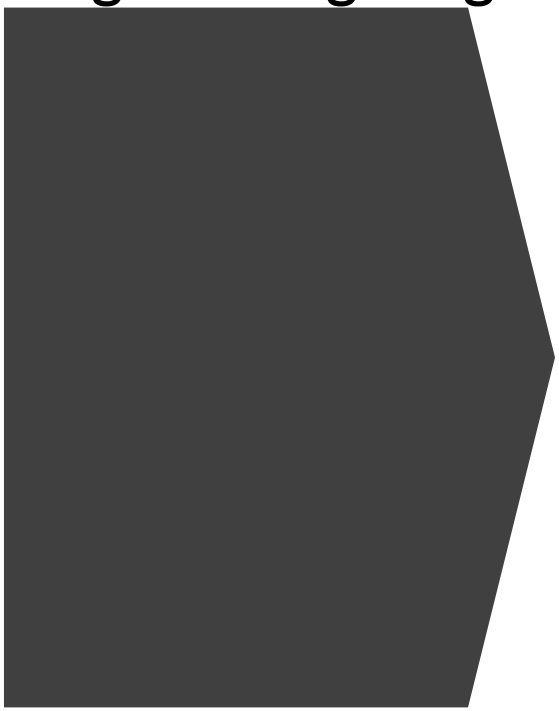
- The ability to use the language more intelligently, as it was designed to be used.
- Become better programmers by understanding the choices among programming language constructs and the consequences of those choices.

5. Overall advancement of computing

1.2 A Brief History of Programming Languages

- The first programming languages were the machine languages of the earliest computers, designed in the 1940's.
- Several hundred programming languages and dialects have been developed since that time.
- Most have had a limited life span and utility, while a few have enjoyed widespread success in one or more application domains.
- Many have played an important role in influencing the design of future languages.

1.2 A Brief History of Programming Languages



Programming Languages		**Paradigms**
Machine Language – 1940's, the first programming language		
Fortran – late 1950's Cobol – late 1950's Algol – 1960's PL/I – 1960's	Pascal – 1970's C – 1970's Fortran 77 – 1970's Modula 2 – 1980's – 1980's Cobol 85 – 1980's Fortran 90 – 1990's	Imperative
C++ - 1980's Smalltalk – 1980's	Eiffel – 1990's Java – 1990's	Object-Oriented
Lisp – 1960's Scheme – 1970's	ML – 1970's Haskell – 1990's	Functional
Prolog – 1970's CLP – 1980's		Logic

1.2 A Brief History of Programming Languages

- **What's in a name?**
- Here are a few programming language names and their meanings:
- Ada : named after the woman who is thought to be the first computer programmer in the 1800s, Ada Lovelace.
- Algol: short for "Algorithmic Language", designed by an international committee in 1959.
- C : designed in the 1970's primarily to support the implementation of the Unix operating system.
- C++ : designed as an extension of C in the 1980's to provide new features that would support object-oriented programming.

1.2 A Brief History of Programming Languages

- Cobol : first designed in 1960, Cobol stands for “Common Business Oriented Language” and uses English as a basis for its syntax.
- Fortran : designed by IBM in 1954 for scientific programming. Fortran is an abbreviation for “Formula Translator” and is probably the most widely used scientific programming language.
- Lisp : short for “List Processor” and designed in 1960 as a tool for writing programs for symbol manipulation and list processing in the field of artificial intelligence.
- Pascal : is a high-level, general-purpose programming language developed in 1971.

1.3 Introduction to Language Paradigms

Programming Languages

- Programming Languages are designed to communicate ideas about algorithms between people and computers.

Programming Paradigm

- Programming paradigm is the way of representing algorithmic expressions to support computing applications for certain application domain. The programming paradigms are developed by the programming communities in their own application area.

1.3 Introduction to Language Paradigms

Imperative Programming

- The program is a series of steps, each of which performs a calculation, retrieves input, or produces output. Procedural abstraction is an essential building block for imperative programming, as are assignments, loops, sequences, and conditional statements. Major imperative programming languages are Cobol, Fortran, Pascal, C, and C++.

1.3 Introduction to Language Paradigms

Object-Oriented Programming

- The program is a collection of objects that interact with each other by passing messages that transform their state. Object modeling, classification, and inheritance are fundamental building blocks for Object-oriented programming. Major object-oriented languages are Smalltalk, Java, C++, and Eiffel.

1.3 Introduction to Language Paradigms

Functional Programming

- The program is a collection of mathematical functions, each with an input (domain) and a result (range). Functions interact and combine with each other using functional composition, conditionals, and recursion. Major functional programming languages are Lisp, Scheme, Haskell, and ML.

1.3 Introduction to Language Paradigms

Logic Programming

- The program is a collection of logical declarations about what outcome a function should accomplish rather than how that outcome should be accomplished. Execution of the program applies these declarations to achieve a series of possible solutions to a problem. The major logic programming language is Prolog.

1.3 Introduction to Language Paradigms

Event-Driven Programming

- The program is a continuous loop that responds to events that are generated in an unpredictable order. These events originate from user actions on the screen (mouse clicks, key stroke etc.). Major event-driven programming languages include Visual Basic and Java.

1.3 Introduction to Language Paradigms

Concurrent Programming/ Parallel Programming

- The program is a collection of cooperating processes, sharing information with each other from time to time. Concurrent programming languages include SR and High-Performance Fortran.

1.3 Introduction to Language Paradigms

- Some programming languages are intentionally designed to support more than one paradigm.
- For examples:
- C++ is an imperative and object-oriented language.
- Java supports object-oriented and event-driven paradigms.
- The experimental language Leda is designed to support the imperative, object-oriented, functional, and logic programming paradigms.

1.4 Application Domains/ Programming Domains

- The programming communities that represent distinct application domains can be grouped in the following way:

1.4 Application Domains/ Programming Domains

Scientific Computing

- Scientific programming is primarily concerned with making complex calculations very fast and very accurately.
- They are primarily implemented using the imperative programming paradigm. The most common data structures are arrays and matrices.
- The parallel programming languages are used for scientific application like weather systems or ocean flow. Modern scientific programming languages include Fortran 90, C, and High-Performance Fortran.

1.4 Application Domains/ Programming Domains

Business Applications

- The systems include an organization's payroll system, accounting system, online sales and marketing systems, inventory and manufacturing systems, and etc.
- Traditionally, business application systems have been developed in programming languages like Cobol, RPG and SQL.
- The online ordering systems are developed using event-driven languages like Java and Tcl/Tk.

1.4 Application Domains/ Programming Domains

Artificial Intelligence

- This programming community is concerned about developing programs that model human intelligent behavior, logical deduction, and cognition.
- The paradigms of functional programming and logic programming have evolved largely through the efforts of artificial intelligence programmers.
- Functional programming languages - Lisp, Scheme, Haskell and ML. Logic programming languages - Prolog and CLP.

1.4 Application Domains/ Programming Domains

Systems Programming

- Systems programmers are those who design and maintain the basic software that runs the systems – operating system components, network software, programming language compilers and debuggers, virtual machines and interpreters, and so on.
- The paradigms that are used include imperative, parallel, and event-driven. About 95 percent of the code of the Unix system is written in C language.

1.4 Application Domains/ Programming Domains

Web Software

- The most dynamic area of new programming community growth is the World Wide Web – for electronic commerce, government, industry, academic etc.
- The programming paradigms that are used are object-oriented and event-driven. Programming languages that support web applications include Perl, Tcl/Tk, JavaScript, PHP, ASP and etc.

1.5 Language Evaluation Criteria

- To evaluate the capabilities of programming languages, we need a set of evaluation criteria.

1.5 Language Evaluation Criteria

Readability

- One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.
- Readability must be considered in the context of problem domain. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural, making it difficult to read.
- The characteristics that contribute to the readability of a programming languages are overall simplicity, orthogonality, control statements, data types and structures, and syntax considerations.

1.5 Language Evaluation Criteria

Writability

- Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability.
- As is the case with readability, writability must be considered in the context of the target problem domain of a language. For example, the writabilities of COBOL and Fortran are different for creating a program to deal with two-dimensional arrays, for which Fortran is ideal.
- Their writabilities are also quite different for producing financial reports with complex formats, for which COBOL was designed. The factors influencing the writability of a language are simplicity and orthogonality, support for abstraction, and expressivity.

1.5 Language Evaluation Criteria

Reliability

- A program is said to be reliable if it performs to its specifications under all conditions. The language features that have a significant effect on the reliability of programs in a given language are type checking, exception handling, aliasing, and readability and writability. Both readability and writability influence reliability.

1.5 Language Evaluation Criteria

Cost

- The ultimate total cost of a programming language is a function of many of its characteristics. First is the cost of training programmers to use the language.
- Second is the cost of writing programs in the language. This is a function of the writability of the language, which depends in part on its closeness in purpose to the particular application.
- Third is the cost of compiling programs in the language.
- Fourth, the cost of executing programs written in a language is greatly influenced by that language's design.

1.5 Language Evaluation Criteria

- The fifth factor in the cost of a language is the cost of the language implementation system. One of the factors that explains the rapid acceptance of Java is that free compiler/interpreter systems have been available after its design was first released.
- A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of ever becoming widely used. Sixth is the cost of poor reliability.
- If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high. The final consideration is the cost of maintaining programs, which includes both corrections and modifications to add new capabilities.

1.6 Compilation Process

- The language that compiler translates is called the **source language**. The process of compilation and program execution takes place in several phases, the most important of which are shown in Figure 1.1.

1.6 Compilation Process

- The **lexical analyzer** gathers the characters of the source program into lexical units. The lexical units of a program are identifiers, special words, operators, and punctuation symbols.
- The lexical analyzer ignores comments in the source program, because the compiler has no use for them.
- The **syntax analyzer** takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called **parse trees**. These parse trees represent the syntactic structure of the program.

1.6 Compilation Process

- The **intermediate code generator** produces a program in a different language, at an intermediate level between the source program and the final output of the compiler, the machine language program.
- Intermediate languages sometimes look very much like assembly languages and in fact sometimes are actually assembly languages.

1.6 Compilation Process

- The **semantic analyzer** is an integral part of the intermediate code generator. The semantic analyzer checks for errors that are difficult if not impossible to detect during syntax analysis, such as type errors.
- **Optimization**, which improves programs (usually in their intermediate code version) by making them smaller or faster or both, is often an optional part of compilation. In fact, some compilers are incapable of doing any significant optimization.

1.6 Compilation Process

- The **code generator** translates the optimized intermediate code version of the program into an equivalent machine language program.
- The **symbol table** serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical analyzer and syntax analyzer and is used by the semantic analyzer and the code generator.

1.6 Compilation Process

- Although the machine language generated by a compiler can be executed directly on the hardware, it must nearly always be run along with some other code.
- Most user programs also require programs from the operating system. Among the most common of these are programs for input and output.
- Before the machine language programs produced by a compiler can be executed, the required programs from the operating system must be found and linked to the user program.
- The process of collecting system programs and linking them to user programs is called **linking and loading**, or sometimes just **linking**. It is accomplished by a system programs program called a **linker**.

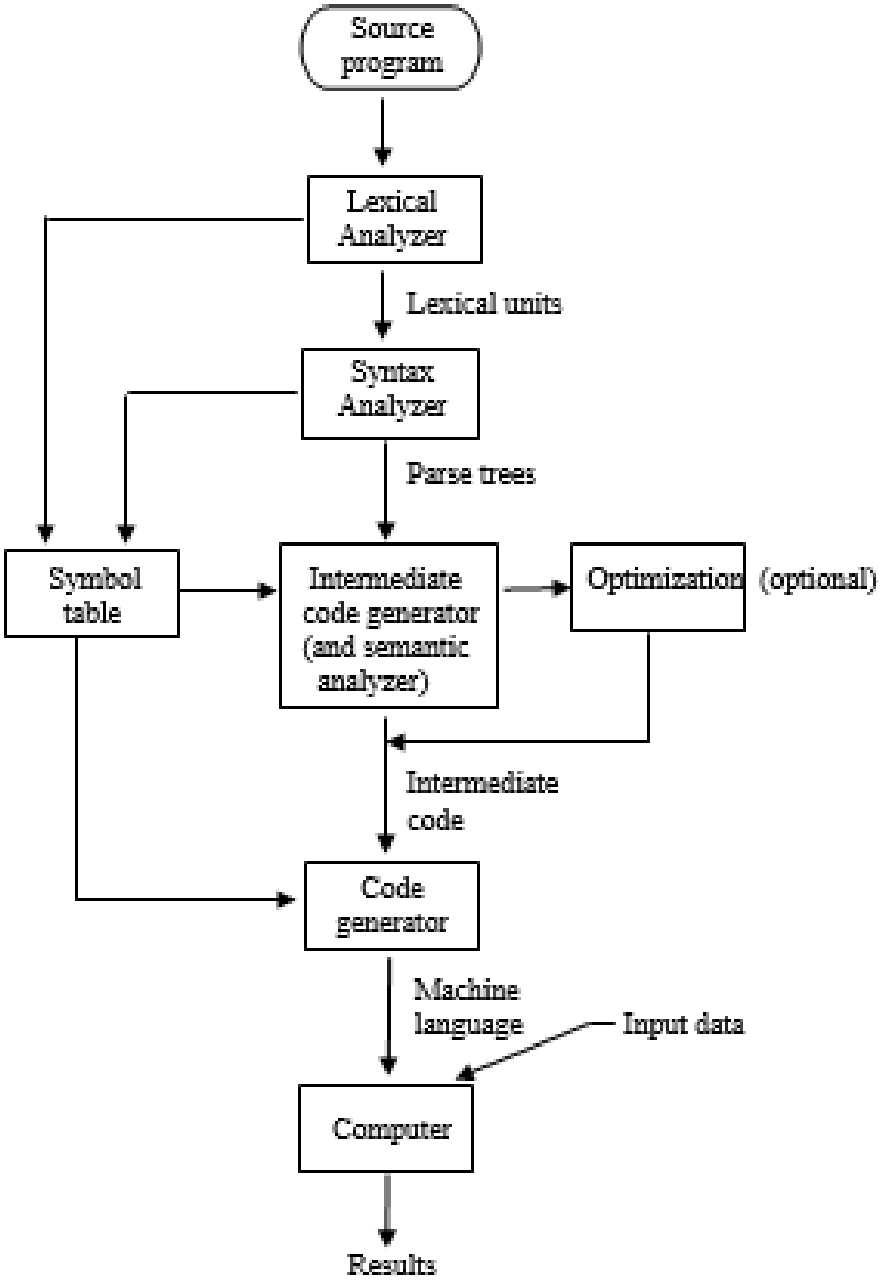


Figure 1.1 : The compilation process


Thank You
:)