Principles of Programming Language (PPL)

Unit 1 and 2(Part): Preliminaries, Implementation Methods.

By: Mrs. Neelam Deshpande

UNIT 1. PRELIMINARY CONCEPTS

1.1 Reasons for Studying Concepts of Programming Languages

1.2 Programming Domains ,Language Evaluation Criteria

1.3 Influences on Language Design,Language Categories

1.4 Programming Paradigms-Imperative , Functional Programming language

1.5 Language Implementation-compilation and interpretation

1.6 Preprocessor

1.7 Programming environments

1.8 Programming Language Translation Issues

1.1 Reasons for Studying Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing

1.2 Programming Domains

1. Scientific applications:
   - ➢ Large number of floating point computations
   - ➢ Fortran
2. Business applications
   - ➢ Produce reports, use decimal numbers and characters
   - ➢ COBOL
3. Artificial intelligence
   - ➢ Symbols rather than numbers manipulated
   - ➢ LISP
4. Systems programming
   - ➢ Need efficiency because of continuous use
   - ➢ C
5. Web Software
   - ➢ Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

1.2 Language Evaluation Criteria

- Readability : the ease with which programs can be read and understood
- Writability : the ease with which a language can be used to create programs
- Reliability : conformance to specifications (i.e., performs to its specifications)
- Cost : the ultimate total cost

1.2.1. Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation)
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Control statements
  - The presence of well-known control structures (e.g., while statement)
- Data types and structures
  - The presence of adequate facilities for defining data structures
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

1.2.2. Writability

- Simplicity and Orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of for statement in many modern languages

1.2.3. Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability

1.2.4. Cost

- Training programmers to use language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs Maintaining programs

Others:

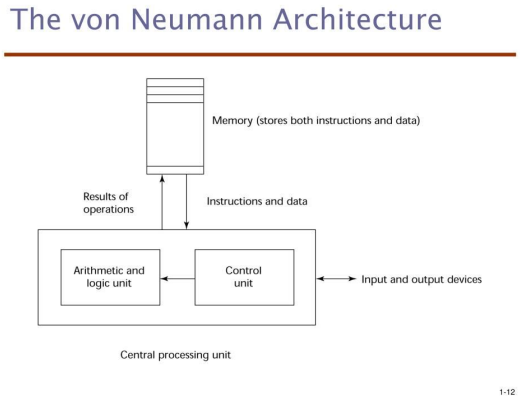
- Portability – The ease with which programs can be moved from one implementation to another
- Generality – The applicability to a wide range of applications
- Well-definedness – The completeness and precision of the language‘s official definition

1.3 Influences on Language Design

- Computer Architecture – Languages are developed around the prevalent computer architecture, known as the von Neumann architecture
- Programming Methodologies – New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

1.3.1. Computer Architecture

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
- Data and programs stored in memory
- Memory is separate from CPU
- Instructions and data are piped from memory to CPU
- Basis for imperative languages Variables model memory cells Assignment statements model piping Iteration is efficient



1.3.2. Programming Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures –
  - ➢ structured programming
  - ➢ top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented – data abstraction
- Middle 1980s: Object-oriented programming – Data abstraction + inheritance + polymorphism

1.4 Language Categories(Paradigms)

- **Imperative**
  - ➢ Central features are variables, assignment statements, and iteration
  - ➢ Examples: C, Pascal
- **Functional**
  - ➢ Main means of making computations is by applying functions to given parameters
  - ➢ Examples: LISP, Scheme
- **Logic**
  - ➢ Rule-based (rules are specified in no particular order)
  - ➢ Example: Prolog
- **Object-oriented**
  - ➢ Data abstraction, inheritance, late binding
  - ➢ Examples: Java, C++
- **Markup** – New; not a programming per se, but used to specify the layout of information in Web documents
  - ➢ Examples: XHTML, XML

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Conflicting criteria – Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria – Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Another conflicting criteria – Example: C++ pointers are powerful and very flexible but not reliably used

1.5 Implementation methods

- Compilation – Programs are translated into machine language
- Pure Interpretation – Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems – A compromise between compilers and pure interpreters

1.5.1 Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:

  i) lexical analysis: converts characters in the source program into lexical units

  ii) syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of program

  iii) Semantics analysis: generate intermediate code

  iv) code generation: machine code is generated

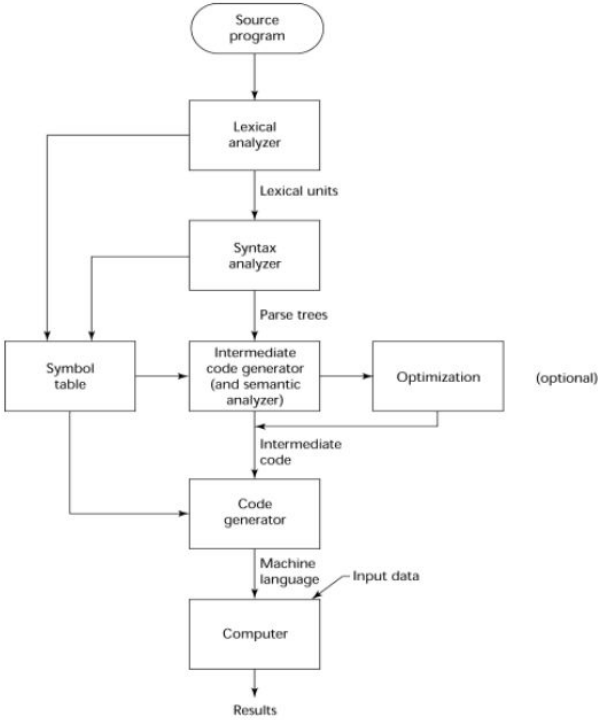


Figure 1.3 The Compilation Process

1.5.1. Compilation contd.

Additional Compilation Terminologies:

- Load module (executable image): the user and system code together
- Linking and loading: the process of collecting system program and linking them to user program

Execution of Machine Code:

- Fetch-execute-cycle (on a von Neumann architecture)

    initialize the program counter

    repeat forever

    fetch the instruction pointed by the counter

    increment the counter

    decode the instruction

    execute the instruction

    end repeat

Von Neumann Bottleneck

Connection speed between a computer‘s memory and its processor determines the speed of a computer

Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a bottleneck

Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

1.5.2. Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages
- Significantly comeback with some latest web scripting languages (e.g., JavaScript)

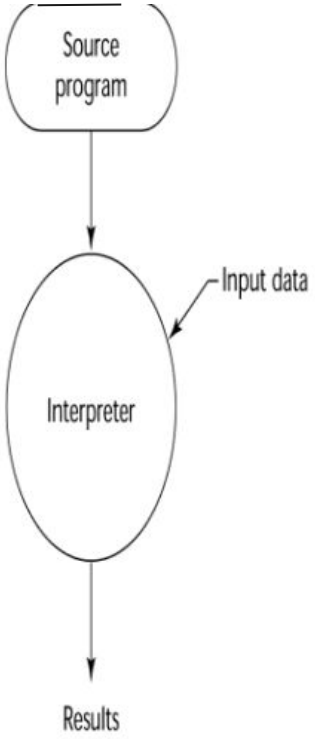


Figure 1.4 Pure Interpreataion

1.5.3. Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation

Examples

– Perl programs are partially compiled to detect errors before interpretation

– Initial implementations of Java were hybrid; the intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a runtime system (together, these are called Java Virtual Machine)

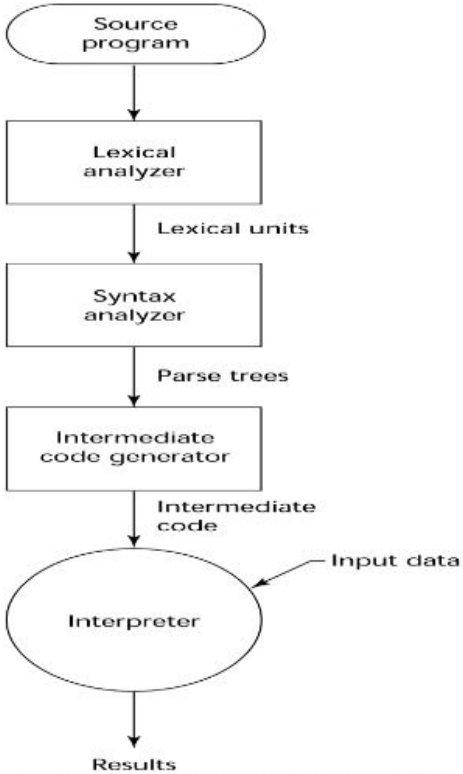


Figure 1.5 Hybrid Implementation

Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system

1.6 Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor

  – expands #include, #define, and similar macros

1.7 Programming Environments

A programming environments is the collection of tools used in the development of software.

This collection may consist:-

- A file system,
- A text editor,
- A linker,
- A compiler,
- Integrated tools

These tools may be access through a uniform interface (GUI).

Examples of Programming Environments

1) UNIX: early(1970s), built around a portable multiprogramming operations system. Support tools for software production and maintenance in variety of languages.

2) JBuilder(not inuse): Integrated Compiler, Editor, Debugger and File system for java Development via GUI.

3) Microsoft Visual Studio .NET, which is a large collection of software development tools, used through a windows interface.

It is used to develop software in following languages-

- C#,
- Visual Basic .NET,
- JScript(MS JavaScript version),
- J# (MS Java version),
- managed C++.

4) NetBeans, Eclipse, Android Studio, Arduino, Turbo C/C++ etc.

1.8. Programming Language Translation Issues

- Programming language Syntax
  - ➢ Key criteria concerning syntax
  - ➢ Basic syntactic concepts
  - ➢ Overall Program-Subprogram structure
- Stages in Translation
  - ➢ Analysis of the source program
  - ➢ Synthesis of the object program
  - ➢ Bootstrapping

1.8.1. Programming Language Syntax

Syntax: the arrangement of words as elements in a sentence to show their relationship

1.8.1.1. General syntactic criteria:

- Readability/writability directed at programmer needs, ease of translation directed at the program translator.
    - Lisp is not very readable or writable but is very easy to translate
    - COBOL is hard to translate due to number of statement forms allowed

1.8.1.2. Syntactic elements of a language

Character set

- Character set choice one of the first choices to be made in syntax design
- Due to various written languages, 16-bit representations must be considered

Identifiers

- widely accepted syntax:string of letters and digits that start with a letter
- Variations between languages include
  - inclusion of special characters (such as . and _)
  - length restrictions

Operator symbols

- Most languages use +
- and - to represent the two basic arithmetic operations

Keywords and reserved words

Noise Words

Comments

Expressions: An expression is a function that accesses data objects in a program and returns some value

Statements: The most prominent syntactic component in imperative languages

1.8.1.3. Overall program-subprogram structure

1. **Separate subprogram definitions**

- C has each subprogram definition treated as a separate syntax unit
  - Each program compiled separately
  - Compiled programs linked at load time
- Object orientation requires information to be passed among the separately compiled units
  - Inheritance requires the compiler to process some of the subprogram issues

2. **Separate data definitions**

- Group together all operations that manipulate a data object
  - Subprogram may consist of all operations that create/print data object
  - The general approach of the class mechanism if Java, C++, Smalltalk

3. **Nested subprogram definitions**

- Important for building modular programs during period of ALGOL, FORTRAN, and Pascal.
- In Pascal, subprogram definitions appeared within main
  - May contain other subprograms nested within their definitions
- This type of subprogram allows for static type checking

1.8.1.3. Overall program-subprogram structure contd.

4. Separate interface definitions

- FORTRAN permits easy compilation of subprograms, but data used across subprograms may have different definitions: compiler will not be able to detect this at compile time
- Pascal allows compiler to have access to all these definitions to aid in finding errors: However, the entire program must be recompiled at the slightest change
- C, ML, and Ada use aspects of both techniques to improve compilation

5. Data descriptions separated from executable statements

- COBOL contains early form of component structure
  - Data declarations/statements divided into data and procedure "divisions"
  - The environment division consists of declarations involving the external operating environment.

6. Unseparated subprogram definitions

- SNOBOL4 represents extreme in program organization
  - No syntactic distinction made between main program and subprogram statements
- In SNOBOL4, statements simply execute
  - Where subprograms begin and end are not different in syntax
  - Execution of a function starts a new subprogram
  - RETURN function ends a subprogram
- Very chaotic! Only useful in allowing run-time translation

1.8.2. Stages in translation

- Translation of a program from syntax to executable is central in every implementation
- Translation is trivial if you're willing to accept slow execution speed
    a. Execution speed is a big deal, that's why we go through the effort of translating programs into efficient executables

Translation can be divided into two major parts:

1. analysis of the source and
2. synthesis of the executable program object.

These processes alternate, often on a statement-by-statement basis.

1.8.2.1. Analysis of the source program

1. **Lexical analysis (scanning)**

The initial phase of any translation is to group the sequence of characters into its parts:

- identifiers
- delimiters
- operator symbols
- numbers
- keywords

The lexical analyzer must:

- identify the type of each lexeme
  - number
  - identifier
  - delimiter
  - etc
- attach a type tag to the lexeme
- convert items to internal representations

2. Syntactic analysis (parsing)

- The second stage of translation
- Larger program structures are identified using lexical items
    - statements
    - declarations
    - expressions
    - etc

Syntactic analysis usually alternates with semantic analysis

3. Semantic analysis

- Many subsidiary functions occur in this stage
  - symbol-table maintenance
  - most error detection
  - expansion of macros
  - execution of compile-time statements
- The output is usually some internal form of the final executable
- This executable is then manipulated by the optimization stage
- Symbol-table maintenance
- Insertion of implicit information: Type guessing
- Error detection: hat does not fit its surrounding context
- Macro processing and compile-time operations: C "#define" lets constants/expressions be evaluated before compilation: C "#ifdef" (if defined) allows for different code to be compiled based on the presence/absence of certain variables

1.8.2.2. Synthesis of the object program(Stages of Translation)

Code Optimization:

- *Code generators* generate object code from the semantic analyzer's produced intermediate code.
- Before generation, optimization of the intermediate code occurs.The register storage and loading is redundant since all data can be kept in the register before storing the result in A

Code generation

- After intermediate code has been optimized, it must be formed into one of
  - assembly language
  - machine code
  - other object form

Linking and loading: Linking and loading: The *linking loader* uses the loader tables to link separate translation code together.

Bootstrapping: Often the translator for a new language is written in that language

Some Review Questions:

1. Write an evaluation of some programming language you know, using the criterias described in the chapter.
2. What are the general methods of implementing a programming language?
3. Discuss your arguments for and against the idea of a single language for all programming domains.
4. Describe some design trade-offs between efficiency and reliability in some language you know.
5. Describe the advantages and disadvantages of some programming environments that you have used.
6. Explain the different aspects of cost of a programming language.
7. Write a short note on various paradigms of programming languages.