Organisation of Programming Languages CMP 401 Introduction

Egena Onu, PhD.

Computer Science Department,

Bingham University.

Course Content

- Language Definition Structure
- Data Types and Structure
- Review of:
  - List
  - Tree
  - Control Structure
  - Data Flow
- Runtime Consideration
- Interpretative Language
- Lexical Analysis and Parsing

Why do computers need a language?

- Computers have been applied in many different areas from controlling power plants to video games in mobile phones.
- The great diversity in the use of computers have necessitated the development of programming languages with different goals.
- Some of the areas of computer applications include:
  - Scientific Applications
  - Business Applications
  - Artificial Intelligence
  - Web Software

Natural Language

- The natural language is a structured system of communication used by humans through speech, signs, gestures and writing.
- The structure of a language is in its grammar and vocabulary.

Natural Languages

- Some examples of natural language include:
  - Igala
  - Idoma
  - Egbira
  - Igede
  - Chinese

Programming Languages

- Programming language is the vocabulary and a set of grammatical rules used to instruct computers.

- Some examples of programming language include:
  - Java
  - C/C++
  - COBOL
  - Prolog
  - Python
  - Etc.

Categories of Programming Languages

- Procedural Programming Languages
  - These are used to execute a sequence of statements
  - These languages use multiple variables and heavy loops to execute their problems.
  - In procedural languages, functions may control variables other than return values. For example, printing.
  - Examples of procedural languages include:
    - Basic, C/C++, Java, FORTRAN and Pascal.

Categories of Programming Languages

- Functional Programming Languages
  - These languages use stored data and frequently avoid loops in favour of recursive functions to execute their problems.
  - These languages focus primarily on the return values of functions.
  - Functional languages are easier and allow programmers to easily focus on abstract issues.
  - Examples include:
    - Python, Lisp and Erlang.

Categories of Programming Languages

- Object-oriented Programming Language
  - These programming languages view the world as a group of objects that have internal data and parts of the data can be accessed externally.
  - The aim is to think about a collection of objects and services that can be offered in specific problems.
  - The major principle here is encapsulation which puts everything an object would need into the object.
  - These languages thrive on reusability and inheritance.
  - Examples include:
    - Java, Python, C++, Lisp, and Perl

Why study programming languages?

- To improve our ability to develop effective Algorithms
- To improve our use of your existing Programming Languages
- To increase our vocabulary of useful programming constructs
- To allow a better choice of programming language
- To make it easier to learn a new language
- To make it easier to design a new language

Who defines and standardises a language?

- Languages can be defined/designed by any individual/group.

- Language standards are defined by national standards bodies:
  - ISO - International Standards organization
  - IEEE - Institute of Electrical and Electronics Engineers
  - ANSI - American National Standards Institute

- These organisations all work in a similar way:
  - Working group of volunteers set up to define standard
  - Agree on features for new standard
  - Vote on standard
  - If approved by working group, submitted to parent organization for approval.

Why Standards?

- Standards ensure language conformity with rules and laws.
- Standards define behaviour/constructs of language - one that meets the rules of the language standard.
- In general (except for Ada), behaviour of non-conforming program is not specified, so any extensions to a standards conforming compiler may still be standards conforming, even though the program is not standards conforming.
- Standards are reviewed every five years.

When to standardise a language?

- Problem: When to standardize a language?
  - If too late - many incompatible versions - FORTRAN in 1960s was already a de facto standard, but no two were the same.
  - If too early - no experience with language - Ada in 1983 had no running compilers.
  - Just right – Java, 1995 (28 years ago) stable version is now Java SE 17 as of September, 2021.

Language Definition Structure

Language Definition

- Languages are defined on the basis of characters and symbols which are strung up to form words or signs.
- The input/output of any algorithm must be presented as a finite string of symbols – the raw contents of some contiguous portion of the computer's memory.
- Reasoning about computation, from this view therefore requires reasoning about strings.
- Suppose there is an arbitrary set Σ called alphabet.
- The elements of Σ are called symbols or characters.

Language Definition

- A **string** (or word) over Σ is a finite sequence of zeros or more symbols from Σ.
- Formally, a string w over Σ is defined (recursively) as either:
  - The empty string, denoted by ε (epsilon), or
  - An ordered pair *(a,x)*, where *a* is a symbol in Σ and x is a string over Σ.
- Normally, either *a* ·x or simply *ax* is used to denote *(a,x)*.
- Similarly, explicit strings are simply written as a sequence of symbols instead of nested ordered pair.
- For example, STRING is a convenient shorthand for the formal expression (S,(T,(R,(I,(N,(G,ε)))))).

Language Definition

- The set of strings over Σ is denoted by Σ* (sigma star).
- All the elements in Σ* is a set of finite string, although Σ* is an infinite set containing strings of every possible finite length.
- The length |w| of a string is formally defined as:

$$|w| := \begin{cases} 0 & if\ w = \varepsilon \\ 1 & if\ w = ax \end{cases}$$

- For example, the string SEVEN has length = 5.

Language Definition

- The Concatenation of two strings x and y, denoted either x·y or simply xy, is the unique string containing the characters of x in order followed by the characters in *y* in order.

- For example, the string NOWHERE is the concatenation of the strings NO and WHERE. That is NO ·WHERE = NOWHERE.

- Concatenation is formally defined as:

$$w \cdot z := \begin{cases} z & if\, w = \varepsilon \\ a \cdot (x \cdot z) & if\; w = ax \end{cases}$$

- When describing concatenation of more that two strings, dots and parentheses are normally omitted.

- This is done by simply writing *wxyz* instead of *(w ·(x ·y) ·z*.

Formal Languages

- Following from strings, a formal language is a set of finite alphabet Σ, or equivalently, an arbitrary subset of Σ*.

- For example, each of the following sets is a language:
  - The empty set ∅.
  - The set {ε}.
  - The set {0,1}.
  - The set {THE, OXFORD, ENGLISH, DICTIONARY}.

- Formal languages are not languages in the same sense that English, Python and Klingon are languages.

- Strings in a formal language do not necessarily carry any meaning nor are they assembled into larger units.

Language Evaluation Criteria

- When it comes to computing, except they are standardised, not many concepts are agreed upon.
- This is the case on the criteria for evaluating programming languages.
- As a programmer, it is important to understand the underlying concepts of the various constructs and capabilities of the language.
- Some of the commonly used criteria for evaluating a programming language are:
  - Readability
  - Writability
  - Reliability and
  - Cost
- Though it may be seen as rather controversial, a list of evaluation criteria is presented in the following table.

Language Evaluation Criteria

- Readability
  - The ease with which a program can be read and understood is one of the most important criteria for judging programming languages.
  - There was a time when programming was largely thought of in terms of writing codes.
  - At this stage, the primary positive characteristic of programming languages was efficiency.
  - Language construct were designed more from the point of view of the machines instead of the users.
  - Things changed when it SDLC integrated maintenance in the cycle.
  - Readability became an important criteria for measuring the quality of programming languages because maintenance depends largely on readability.

Language Evaluation Criteria

- Writability
  - Writability is a measure of how easily a language can be used to create programs for a chosen problem domain.
  - Like readability, writability is also considered in the context of the target problem domain of the language.
  - Writability eases the ability to chose languages in solving problems in particular domains.
  - For example, the writabilities of Visual BASIC (VB) and C are dramatically different for creating a program that has a graphical user interface (GUI) for which VB is particularly designed for.

Language Evaluation Criteria

- Reliability
  - A program is said to be reliable if it performs to its specification under all conditions.
  - Reliability is measured based on characteristics such as:
    - Type checking
    - Exception handling
    - Aliasing
    - Readability and writability.

Language Evaluation Criteria

- Cost
  - The total cost of a programming language is a function of many of its characteristics:
    1. Cost of training programmers to use the language
       - This is a function of the simplicity and orthogonality and experience of the programmer.
       - Although more powerful not necessarily more difficult to learn, they often are.
    2. Cost of writing programs in the language
       - This is a function of the writability of language which depends in part on its closeness in purpose to the particular application.
       - The original efforts to design and implement high level languages were driven by the desire to lower the costs of creating software.

Language Evaluation Criteria

- Cost

3. Cost of compiling
   - For example, the major impediment to early use of Ada was the prohibitively high cost of running the first generation of Ada compilers.

4. Language design
   - The cost of executing programs written in a language isi greatly influenced by the design.
   - Some language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler.
   - A simple trade off can be made between compilation cost and execution speed of the compiled code using optimisation.
   - Optimisation is a collection of techniques that compilers use decrease the size and/or increase the execution speed of the code they produce.

Influences on Language Design

- Apart from the language evaluation criteria, there are other factors that influence the basic design of programming languages.

- The most important of these factors are computer architecture and programming design methodologies.

Influences on Language Design

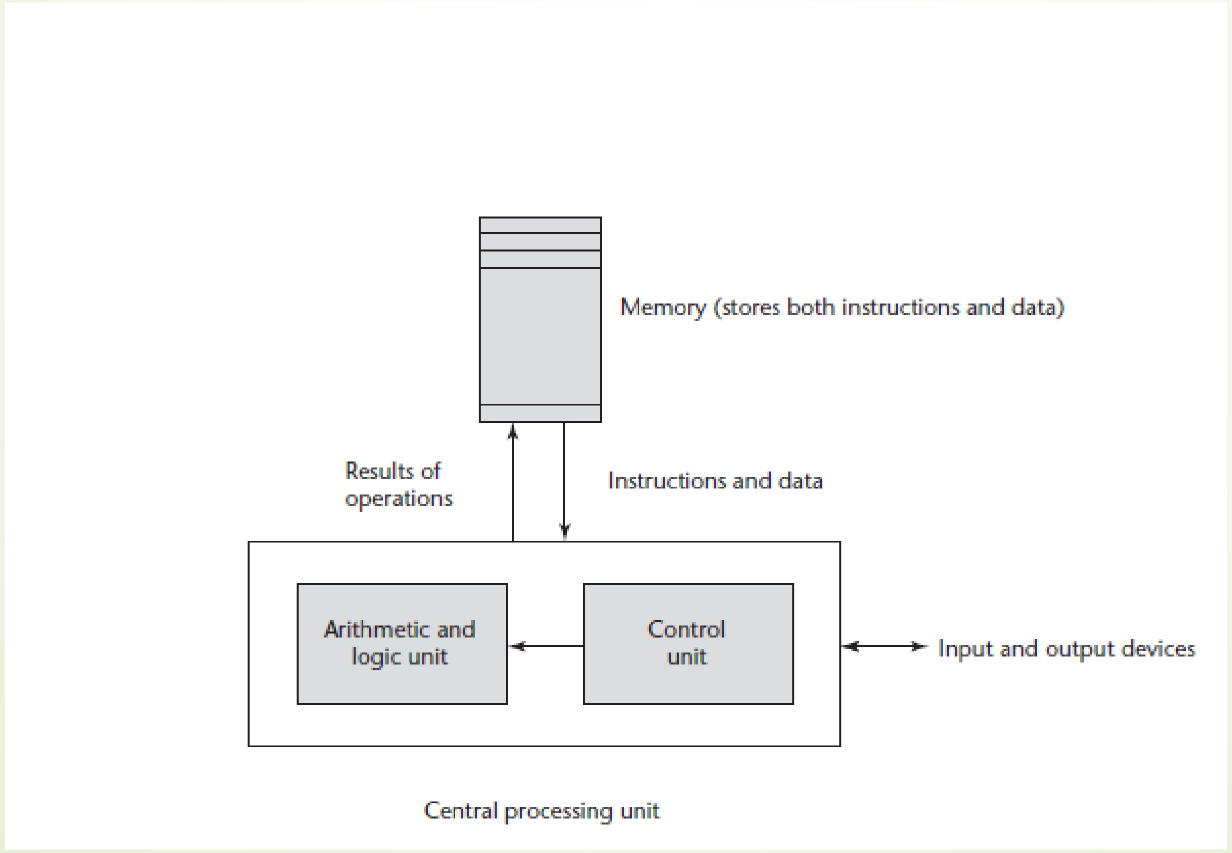
- Computer Architecture
  - The basic architecture of computers has had a profound effect on language design.
  - Most of the popular languages of the past 60 years have been designed around the prevalent computer architecture, called the von Neumann architecture.
  - John von Neumann is one of the originators of this architecture therefore it is named after him.

Influences on Language Design

- Von Neumann Architecture
  - In a von Neumann computer, both the data and programs are stored in the same memory.
  - The central processing unit (CPU) which executes the instruction is separate from the memory.
  - Therefore, instructions and data must be transmitted, or piped from memory to the CPU.
  - Results of operations in the CPU are then moved back to the memory.
  - Nearly all the digital computers built since the 1940s are based on this architecture.

Influences on Language Design

- Von Neumann Architecture



Influences on Language Design

- Von Neumann Architecture
  - The execution of a machine code program on a von Neumann architecture occurs in a process called the **fetch-execute cycle.**
  - Always remember that the program resides in the memory but are executed by the CPU.
  - Each instruction to be executed must be moved from memory to the processor.
  - The address of the next instruction to be executed is maintained in a register called the **program counter.**

Influences on Language Design

- The fetch-execute cycle is as follows:

```
initialize the program counter
repeat forever
fetch the instruction pointed to by the program counter
increment the program counter to point at the next instruction
decode the instruction
execute the instruction
end repeat
```

Language Implementation Methods

- From the von Neumann Architecture, two of the primary components of a digital computer are: the processor and internal memory.

- The internal memory stores program and data.

- The processor is a collection of circuits that provides a realisation of a set of primitive operations or machine instructions such as arithmetic and logic operations.

- Some of these machine instructions are sometimes called microinstructions which are defined at lower levels.

Language Implementation Methods

- Language implementation methods provide the facilities through which high level programming languages get to speak the language of the machine.

- Implementation methods require a large collection of programs (operating system) which supplies higher-level primitives than those of the machine language.

- These primitives provide
  - System resource management
  - Input and output operations
  - File management system
  - text/ and/or program editors
  - And a host of other functionalities.

Language Implementation Methods

- Because language implementation systems need many of the operating system facilities, they interface with the OS rather than directly with the processor.

- The OS and language implementations are layered over the machine language interface of the computer.

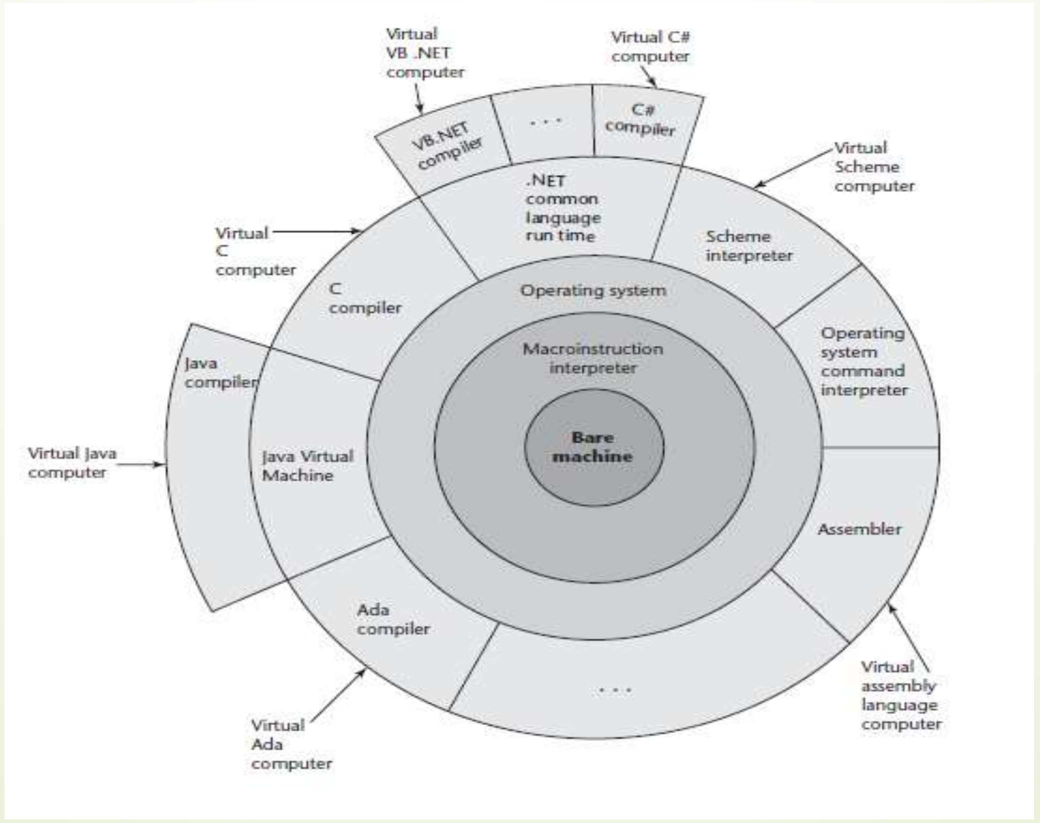
- These layers can be thought of a virtual computers, providing interface to the user at higher levels.

- Most computer systems provide several different virtual computer.

- The user programs form another layer over the top of the layer of virtual computers.

Language Implementation Methods

- Layered interface of virtual computers, provided by a typical computer system



Language Implementation Methods

- There are several methods of implementing a programming language:
  - Compilation
  - Pure Interpretation
  - Hybrid Implementation Methods

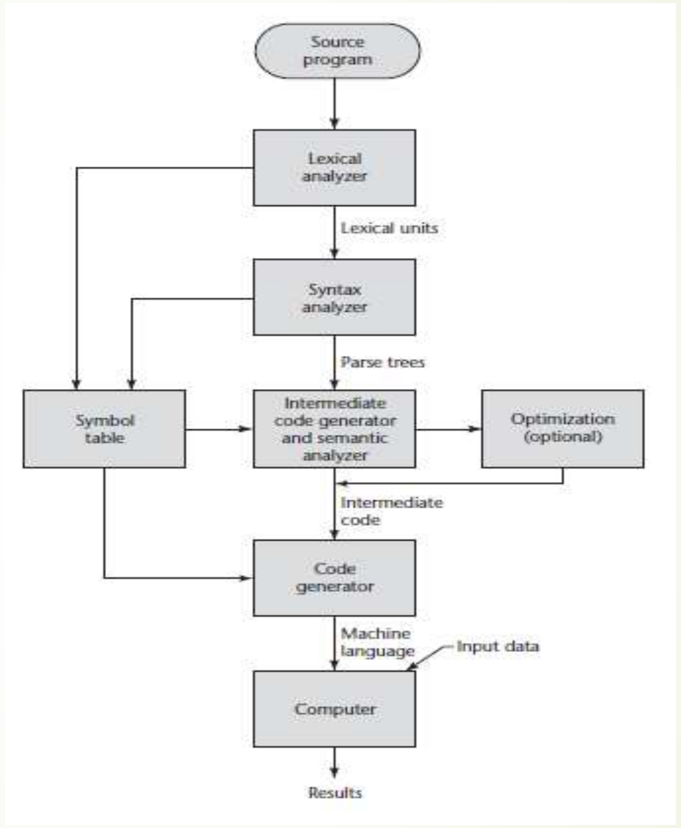
Implementation Methods

- A language is implemented using any three of the fundamental methods:
  - Compilation
  - Pure interpretation
  - Hybrid implementation

Compilation

- This method uses compilers to translate programs into machine languages which are directly executed on the computer.
- Languages that use this method have the advantage of being very fast, once the translation process is complete.
- Languages such as C, C++ and COBOL are implemented using the compilation method.
- The process of compilation and program execution involves many phases. The most important ones are itemised in the following figure.

Compilation



Compilation method

Compilation

- The language which the compiler translates is the *source code.*
- 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.

Compilation

- 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.
- In many cases, no actual parse tree structure is constructed; rather, the information that would be required to build a tree is generated and used directly.

Compilation

- 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 actual assembly languages.
- In other cases, the intermediate code is at a level somewhat higher than an assembly language.

Compilation

- The **semantic analyzer** is an integral part of the intermediate code generator.
- The semantic analyzer checks for errors, such as type errors, that are difficult, if not impossible, to detect during syntax analysis.

Compilation

- **Optimization** improves programs (usually in their intermediate code version) by making them smaller or faster or both.
- Because many kinds of optimization are difficult to do on machine language, most optimization is done on the intermediate code.
- The **code generator** translates the optimized intermediate code version of the program into an equivalent machine language program.

Compilation

- 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 and syntax analyzers and is used by the semantic analyzer and the code generator.

Pure Interpretation

- In pure interpretation method, programs are interpreted by another program called an **interpreter**, with no translation whatever.
- The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions.
- This software simulation obviously provides a virtual machine for the language.

Pure Interpretation

- Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units.
- For example, if an array index is found to be out of range, the error message can easily indicate the source line of the error and the name of the array.

Pure Interpretation

- On the other hand, this method has the serious disadvantage that execution is 10 to 100 times slower than in compiled systems.

- The primary source of this slowness is the decoding of the high- level language statements, which are far more complex than machine language instructions (although there may be fewer statements than instructions in equivalent machine code).

Pure Interpretation

- Furthermore, regardless of how many times a statement is executed, it must be decoded every time.

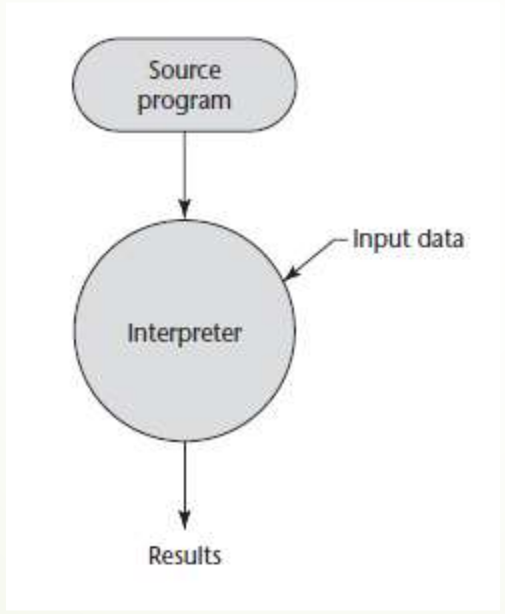
- Therefore, statement decoding, rather than the connection between the processor and memory, is the bottleneck of a pure interpreter.

- Another disadvantage of pure interpretation is that it often requires more space.

- In addition to the source program, the symbol table must be present during interpretation.

- In addition, the source program may be stored in a form designed for easy access and modification rather than one that provides for minimal size.

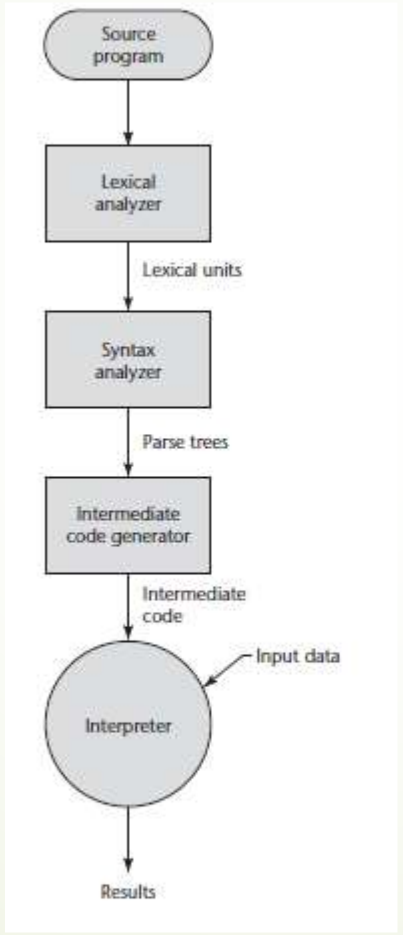
Pure Interpretation



Hybrid Implementation

- Some language implementation systems are a compromise between compilers and pure interpreters.
- Here, high-level language programs are translated to an intermediate language designed to allow easy interpretation.
- This method is faster than pure interpretation because the source language statements are decoded only once.
- Such implementations are called *hybrid implementation systems*.

Hybrid Implementation



Hybrid Implementation

- Instead of translating intermediate language code to machine code, it simply interprets the intermediate code.
- The initial implementations of Java were all hybrid implementation.
- Its intermediate form, called *byte code*, provides portability to any machine that has a byte code interpreter and an associated run-time system.
- Together, these are called the Java Virtual Machine.
- There are now systems that translate Java byte code into machine code for faster execution.
- Such systems are called the “Just-in-Time (JIT)” systems.

Hybrid Implementation

- A Just-in-Time (JIT) implementation system initially translates programs to an intermediate language.
- Then, during execution, it compiles intermediate language methods into machine code when they are called.
- The machine code version is kept for subsequent calls.
- JIT systems now are widely used for Java programs.

Hybrid Implementation

- Also, the .NET languages are all implemented with a JIT system.
- Sometimes an implementer may provide both compiled and interpreted implementations for a language.
- In these cases, the interpreter is used to develop and debug programs. Then, after a (relatively) bug-free state is reached, the programs are compiled to increase their execution speed.

Language Structure

Syntax and Semantics

- The task of providing a concise yet understandable description of a programming language is difficult but essential to the success of the language.
- One of the problems in describing a language is the diversity of the people who must understand it.
- Among these are initial evaluators, implementers, and users.

Syntax and Semantics

- Programming language implementers obviously must be able to determine how the expressions, statements, and program units of a language are formed, and also their intended effect when executed.

- The difficulty of the implementers' job is, in part, determined by the completeness and precision of the language description.

Syntax and Semantics

- The study of programming languages, like the study of natural languages, can be divided into examinations of syntax and semantics.
- The **syntax** of a programming language is the form of its expressions, statements, and program units.
- **Semantics** is the meaning of those expressions, statements, and program units.

Syntax and Semantics

- For example, the syntax of a Java **while** statement is

  while (boolean_expr) statement

- The semantics of this statement form is that:

  i. when the current value of the Boolean expression is true, the embedded statement is executed.

  ii. Then control implicitly returns to the Boolean expression to repeat the process.

  iii. If the Boolean expression is false, control transfers to the statement following the **while** construct.

Syntax and Semantics

- Although they are often separated for discussion purposes, syntax and semantics are closely related.
- In a well-Designed programming language, semantics should follow directly from syntax.
- That is, the appearance of a statement should strongly suggest what the statement is meant to accomplish.
- Describing syntax is easier than describing semantics, partly because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

Syntax and Semantics

- To be continued with:
  - The Problem of Describing Syntax

Questions!!!