MODIBBO ADAMA UNIVERSITY, YOLA

DEPARTMENT OF COMPUTER SCIENCE

CSC306: ORGANISATION OF PROGRAMMING LANGUAGES

(3 Units)

Textbooks:

- Programming Language Concepts by Peter Sestoft.
- Organization of Programming Languages by Bernd Teufel.
- Programming Languages: Principles and Practice, by Kenneth C. Louden and Kenneth A. Lambert.
- Concepts of Programming Languages by Robert W. Sebesta.

Course Outline:

- Brief history of Programming Languages.
- Overview of Programming Languages.
- Programming Paradigms.
- The effects of scale on programming methodology
- Language Semantics.
- Declarations and types.
- Modules in programming languages.
- Review of basic data types.

REASONS FOR STUDYING CONCEPTS OF PROGRAMMING LANGUAGES

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. In fact, many now believe that there are more important areas of computing for study than can be covered in a five-year university curriculum. The following are compelling list of potential benefits of studying concepts of programming languages:

1. *Increased ability to express ideas/algorithms:*
   In Natural language, the depth at which people think is influenced by the expressive power of the language they use. In programming language, the complexity of the algorithms that people Implement is influenced by the set of constructs available in the programming Language.
   Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

2. *Improved background for choosing appropriate Languages:*
   Many programmers use the language with which they are most familiar, even though poorly suited for their new project. It is ideal to use the most appropriate language.
   If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem.
   Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

3. *Increased ability to learn new languages:*
   For instance, knowing the concept s of object-oriented programming OOP makes learning Python significantly easier than those who have never used those concepts and also, knowing the grammar of one's native language makes it easier to learn a second language.
   Furthermore, learning a second language has the benefit of teaching you more about your first/native language.

4. *Better Understanding of Significance of implementation:*

In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they
are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices. Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issue is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

5. *Better use of languages that are already known:*

Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously
unknown and unused parts of the languages they already use and begin to use those features.

6. *The overall advancement of computing*:

Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available. In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.
For example, many people believe it would have been better if ALGOL
60 had displaced Fortran in the early 1960s, because it was more elegant and had much better control statements, among other reasons. That it did not, is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60. They found its description difficult to read (which it was) and even more difficult to understand. They did not appreciate the benefits of block structure, recursion, and well-structured control statements, so they failed to

see the benefits of ALGOL 60 over Fortran. The fact that computer users were generally unaware of the benefits of the language played a significant role.

HISTORY OF PROGRAMMING LANGUAGES

The first programming languages were the machine and assembly languages of the earliest computers, beginning in the 1940s. Hundreds of 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. Most new programming languages arise as a reaction to some language that the designer knows (and likes or dislikes) already, so one can propose a family tree or genealogy for programming languages, just as for living organisms. There are many more languages than those shown, in (**Fig 1)** below in particular if one counts also more domain-specific languages such as Matlab, SAS and R, and strange "languages" such as spreadsheets. In general, languages lower in the diagram (near the time axis) are closer to the real hardware than those higher in the diagram, which are more "high-level" in some sense. In Fortran77 or C, it is fairly easy to predict what instructions and how many instructions will be executed at run-time for a given line of program. The mental machine model that the C or Fortran77 programmer must use to write efficient programs is close to the real machine. Conversely, the top-most languages (SASL, Haskell, Standard ML, F#, Scala) are functional languages, possibly with lazy evaluation, with dynamic or advanced static type systems and with automatic memory management, and it is in general difficult to predict how many machine instructions are required to evaluate any given expression. The mental machine model that the Haskell or Standard ML or F# or Scala programmer must use to write efficient programs is far from the details of a real machine, so he can think on a rather higher level. On the other hand, he loses control over detailed efficiency. It is remarkable that the recent mainstream languages Java and C#, especially their post-2004 incarnations, have much more in common with the academic languages of the 1980's than with those languages that were used in the "real world" during those years (C, Pascal, C++).

However, In the early days of programming, machines were extremely slow and memory was scarce. Program speed and memory usage were, therefore, the prime concerns. Also, some programmers still did not trust compilers to produce efficient executable code (code that required the fewest number of machine instructions and the smallest amount of memory). Thus, one principal design criterion really mattered: efficiency of execution. For example, FORTRAN was specifically designed to allow the programmer to generate compact code that executed quickly.

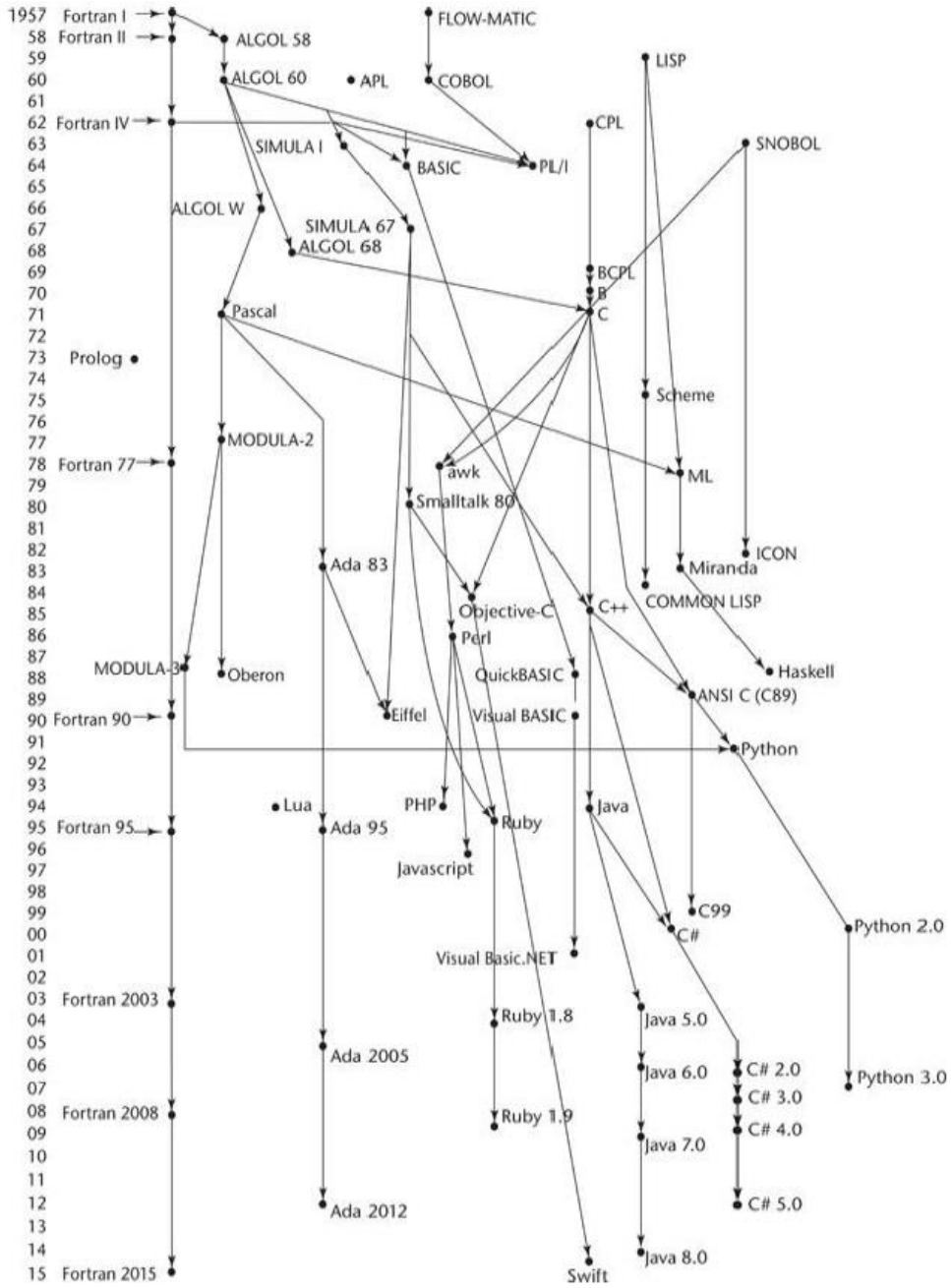


***Figure 1**: The genealogy of programming languages.*

Indeed, with the exception of algebraic expressions, early FORTRAN code more or less directly mapped to machine code, thus minimizing the amount of translation that the compiler would have to perform. Judging by today's standards, creating a high-level programming language that required the programmer to write code nearly as complicated as machine code might seem counterproductive. After all, the whole point of a high-level programming language is to make life easier for the programmer. In the early days of programming, however, writability—the quality of a language that enables a programmer to use it to express a computation clearly, correctly, concisely, and quickly—was always subservient to efficiency. Moreover, at the time that FORTRAN was developed, programmers were less concerned about creating programs that were easy for people to read and write, because programs at that

time tended to be short, written by one or a few programmers, and rarely revised or updated except by their creators.

By the time COBOL and Algol60 came on the scene, in the 1960s, languages were judged by other criteria than simply the efficiency of the compiled code. For example, Algol60 was designed to be suitable for expressing algorithms in a logically clear and concise way—in other words, unlike FORTRAN, it was designed for easy reading and writing by people. To achieve this design goal, Algol60's designers incorporated block structure, structured control statements, a more structured array type, and recursion. These features of the language were very effective. For example, C. A. R. Hoare understood how to express his QUICKSORT algorithm clearly only after learning Algol60.

COBOL's designers attempted to improve the readability of programs by trying to make them look like ordinary written English. In fact, the designers did not achieve their goal. Readers were not able to easily understand the logic or behavior of COBOL programs. They tended to be so long and verbose that they were harder to read than programs written in more formalized code. But human readability was, perhaps for the first time, a clearly stated design goal. In the 1970s and early 1980s, language designers placed.

In the 1970s and early 1980s, language designers placed a greater emphasis on simplicity and abstraction, as exhibited by Pascal, C, Euclid, CLU, Modula-2, and Ada. Reliability also became an important design goal. To make their languages more reliable, designers introduced mathematical definitions for language constructs and added mechanisms to allow a translator to partially prove the correctness of a program as it performed the translation. However, such program verification systems had limited success, primarily because they necessitated a much more complex language design and translator, and made programming in the language more difficult than it would be otherwise. However, these efforts did lead to one important related development, strong data typing, which has since become standard in most languages.

In the 1980s and 1990s, language designers continued to strive for logical or mathematical precision. In fact, some attempted to make logic into a programming language itself. Interest in functional languages has also been rekindled with the development of ML and Haskell and the continued popularity of Lisp/Scheme.

However, the most influential design criterion of the last 25 years has come from the object-oriented approach to abstraction. As the popularity of the object-oriented languages C++, Java, and Python soared, language designers became ever more focused on abstraction mechanisms that support the modeling of real-word objects, the use of libraries to extend language mechanisms to accomplish specific tasks, and the use of object-oriented techniques to increase the flexibility and reuse of existing code.

Thus, we see that design goals have changed through the years, as a response both to experience with previous language designs and to the changing nature of the problems addressed by computer science. Still, readability, abstraction, and complexity control remain central to nearly every design decision.

Despite the importance of readability, programmers still want their code to be efficient. Today's programs process enormous data objects (think movies and Web searches) and must run on miniature computers (think smart phones and tablets). In the next section, we explore the continuing relevance of this criterion to language design.

Attributes of a good language

1. Orthogonality: - every combination of features is meaningful. Features work independently.
2. Clarity, Simplicity and Unity: - provides both a framework for thinking about algorithms and a means of expressing those algorithms.
3. Naturalness for the application: - program structure reflects the logical structure of algorithm.
4. Ease of program verification: - Does a program correctly perform its required function?
5. Programming environment: - external support for the language. Libraries, documentation, community, IDEs etc.
6. Portability of programs: - transportability of the resulting programs from the computer on which they are developed to other computer systems.
7. Cost of use - program execution (run-time), program translation, program creation, and program maintenance.
8. Support for Abstraction: Hide details where you don't need them. Program data reflects the problem you're solving. Algorithm to compute the difference of 5 numbers;

   Step *1: Input the numbers*
   2: Get the lowest numbers
   3: Get the highest number
   4: Get the difference= highest no- Smallest no
   5: Print the difference

OVERVIEW OF SOME PROGRAMMING LANGUAGES

We will describe the development of a collection of programming languages, the environment in which each was designed and focus on the contributions of the language and the motivation for its development.

***Assembly Languages*:**

- Invented by machine designers in early 1950s.
- Machine code is tedious and error-prone.
  - Poor readability.

    - Poor modifiability.
- Shift from machine code to mnemonics.
- First occurrences of reusable macros & subroutines.

***Fortran*:**

- Designed by John Backus at IBM in the early 1950's
- First widely accepted compiled high-level language:
    - Designed for the new IBM 704, which had index registers and floating-point hardware.
    - This led to the idea of compiled programming languages, because there was no place to hide the cost of interpretation (no need for floating-point software).
- Design influenced by environment:
    - Computers were expensive, slow, with small memory.
    - Primary use of computers was for scientific applications.
    - No existing efficient way to program computers.
- (**Fortran I**) was the first implemented version of Fortran (1957, 18 worker years of effort):
    - Names could have up to six characters.
    - Post-test counting loop (DO).
    - Formatted I/O.
    - No dynamic memory allocation.
    - User-defined subprograms (separate compilation added in Fortran II).
    - Three-way selection statement (IF).
    - No data typing statements (I,J,K,L,M,N integers, rest floating point).
    - Code was very fast => quickly became widely used.

***LISP*:**

- Designed by John McCarthy at MIT in the late 1950s.
- Design influenced by AI applications:
    - Symbolic computation (rather than numeric).
        - Ex: differentiation of algebraic expressions.
        - Ex: Advice taker.
    - Process data in lists (rather than arrays):
        - Dynamically allocated linked lists.
        - Implicit deallocation of abandoned lists.
- Implemented on IBM 704.
- (**Pure LISP**) was purely a functional language:
    - No need for variables, assignment, or iteration (loops).
    - Control via recursion and conditional expressions.
    - Syntax is based on lambda calculus.
- Only two data types: Atoms and Lists.

- Atoms are either symbols (identifiers) or numeric literals.
- Two basic list operations: CAR and CDR

***Related Functional Languages*:**

- Scheme (MIT mid-1970s):
  - – Small size, simple syntax and semantics.
  - – Exclusive use of static scoping.
  - – Functions are first class entities.
- Common Lisp, Miranda, Haskell, ML.

***Algol*:**

- International Algorithmic Language.
- Designed by IFIP working group in 1958-1960:
  - John Backus, Peter Naur, John McCarthy, Alan Perlis & others.
  - Syntax specified formally using the Backus-Naur Form (BNF).
- Goals:
  - Universal language for communicating algorithms.
  - Portable, machine independent.
  - Close to mathematical notation.
  - Must be translatable to machine code.

***Simula 67*:**

- Designed by Kristen Nygaard and Ole-Johan Dahl at NCC.
- Superset of Algol 60, for simulations.
- Innovations:
  - Coroutines (subprograms that restart at the position where they previously stopped).
  - First OOP language:
    - Classes (package data structure with manipulating routines).
    - Objects as class instances (local data & code executed at creation).
    - Inheritance, virtual methods.
- Influenced all subsequent OO programming languages:
  - Smalltalk
  - Objective-C
  - C++
  - Eiffel
  - Modula 3
  - Self
  - C#
  - CLOS

***Ada*:**

- Designed for DoD as a high-level language for embedded systems applications:
- Huge design effort, involving hundreds of people, much money, and about eight years.
    - Strawman requirements (April 1975)
    - Woodman requirements (August 1975)
    - Tinman requirements (1976)
    - Ironman equipments (1977)
    - Steelman requirements (1978)
- Named Ada after Augusta Ada Byron, the first programmer
- Major Contributions:
    - Packages -support for data abstraction
    - Exception handling -elaborate
    - Generic program units
    - Concurrency -through the rendezvous synchronization model
- Popularity suffered because the DoD no longer requires its use but also because of popularity of C++.

***C: A Portable Systems Language*:**

- Designed by Dennis Ritchie at Bell Labs in 1972.
- Designed for systems programming:
    - the development of an OS and its utilities.
    - first Unix written in assembly language.
    - B was first high-level language on UNIX (Ken Thompson, 1970)
    - C was developed as a typed language based on B: Huge
- Used as a portable assembly language:
    - Early C++, Modula 3, and Eiffel were translated to C.
- C compilers available for all kinds of architectures:
    - GNU gcc for more than 70 instruction set architectures.

***C++: Combining Imperative and OO Programming*:**

- Developed by Bjarne Stroustrup at Bell Labs in 1980.
- Backward compatible with C:
    - Easy to link C++ code with C code.
- Facilities for OOP related to Simula 67 & Smalltalk:
    - Derived classes & inheritance (1983).
    - Virtual methods, overloaded methods & operators (1984).
    - Multiple inheritance, abstract classes (1989).
    - Templates, exception handling (ISO 1998).
- Large & complex language:
    - Supports both procedural and OO programming through functions & methods.

- Very popular:
    - Availability of good & inexpensive compilers.
    - Suitable for large commercial software projects.
- Microsoft's version (released with .NET in 2002):
    - No multiple inheritance, references for garbage collected objects.

***Java: An Imperative-Based OO Language*:**

- Developed by a team headed by James Gosling at Sun in the early 1990s.
    - C and C++ were not satisfactory for embedded electronic devices.
- Based on C++:
    - Significantly simplified:
        - no struct, union, enum.
        - no pointer arithmetic.
        - eliminated half of the assignment coercions of C++.
        - no multiple inheritance, no operator overloading.
    - Supports only OOP (e.g., no stand-alone subprograms).
    - All objects allocated on the heap & garbage collected.
- Very successful:
    - Eliminated many unsafe features of C++ ⇒simpler, safer design.
    - Supports concurrency (threads, synchronized methods).
    - Libraries for applets, GUIs, database access.
    - Portable:
        - Java Virtual Machine concept, JIT compilers.
    - Widely used for Web programming.
    - Use increased faster than any previous language.
- Java 5.0:
    - Enumeration class, generics, new iteration construct.

***Prolog: Logic Programming*:**

- Developed, by Colmerauer and Roussel (University of Aix-Marseille), with help from Kowalski (University of Edinburgh) in the early 1970s.
- Non-procedural language:
    - describe What as opposed to How.
    - notation based on predicate calculus (Horn clauses).
    - Inference method based on resolution (Robinson 1965).
- Highly inefficient relative to equivalent imperative progs.
- Small application areas in AI and DBMS.
- Program = a collection of statements:
    - Facts:
        - mother(joanne, jake); father(vern, joanne)
    - Rules:
        - parent(X,Y) :-mother(X,Y).
        - parent(X,Y) :-father(X,Y).

        - grandparent(X,Z) :-parent(X,Y), parent(Y,Z).
    - Queries:
        - grandparent(X,jake).

***Scripting Languages for the Web*:**

- JavaScript
    - Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
    - A client-side HTML-embedded scripting language, often used to create dynamic HTML documents
    - Purely interpreted
    - Related to Java only through similar syntax
- PHP
    - PHP: Hypertext Preprocessor, designed by Rasmus Lerdorf
    - A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
    - Purely interpreted
- Python
    - multiparadigm scripting language:
    - imperative
    - functional
    - object oriented
    - Used for CGI programming and form processing.

PROGRAMMING DOMAINS

Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones. Because of this great diversity in computer use, programming languages with very different goals have been developed. We will briefly discuss a few of the most common areas of computer applications and their associated languages.

1. Scientific Applications
2. Data processing Applications
3. Text processing Applications
4. Artificial intelligence Applications
5. Systems Programming Applications
6. Web software

Scientific Applications:

These are Applications which predominantly manipulate numbers and arrays of numbers, using mathematical and statistical principles as a basis for the algorithms.

These algorithms encompass such problem as statistical significance test, linear programming, regression analysis and numerical approximations for the solution of differential and integral equations. Examples are; FORTRAN, Pascal, Math lab.

Data processing Applications:

These are programming problems whose predominant interest is in the creation, maintenance, extraction and summarization of data in records and files. Examples are; COBOL is a programming language that can be used for data processing applications.

Text processing Applications:

These are programming whose principal activity involves the manipulation of natural language text, rather than numbers as their data. SNOBOL and C language have strong text processing capabilities.

Artificial Intelligence Applications:

Those programs which are designed principally to emulate intelligent behavior. They include game playing algorithms such as chess, natural language understanding programs, computer vision, robotics and expert systems. Examples are;

- LISP has been the predominant AI programming language,
- PROLOG using the principle of ‘’Logic programming’’
- Lately AI applications are written in Java, C++ and python.

Systems Programming Applications:

These involve developing those programs that interface the computer system (the hardware) with the programmer and the operator. These programs include compilers, assembles, interpreters, input-output routines, program management facilities and schedules for utilizing and serving the various resources that comprise the system. Examples are; Ada and Modula – 2 are examples of programming languages and C.

Web software:

The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript or PHP.

There is also some markup like languages that have been extended to include constructs that control document processing.

CRITERIA FOR LANGUAGE EVALUATION AND COMPARISION

1. **Expressivity**

The ability of a language to clearly reflect the meaning intended by the algorithm designer (the programmer). An “expressive” language permits an utterance to be compactly stated, and encourages the use of statement forms associated with structured programming (usually “while “loops and “if – then – else” statements).

2. **Well – Defined**

The language’s syntax and semantics are free of ambiguity, are internally consistent and complete. Thus, the implementer of a well-defined language should have, within its definition a complete specification of all the language’s expressive forms and their meanings. The programmer, by the same virtue should be able to predict exactly the behavior of each expression before it is actually executed.

3. **Data types and structures**

The ability of a language to support a variety of data values (integers, real, strings, pointers etc.) and non-elementary collect ions of these.

4. **Modularity**

Modularity has two aspects: the language’s support for sub-programming and the language’s extensibility in the sense of allowing programmer – defined operators and data types. By sub programming, we mean the ability to define independent procedures and functions (subprograms), and communicate via parameters or global variables with the invoking program.

5. **Input-Output facilities**

In evaluating a language’s “Input-Output facilities” we are looking at its support for sequential, indexed, and random-access files, as well as its support for database and information retrieval functions

6. **Portability**

A language which has “portability” is one which is implemented on a variety of computers. That is, its design is relatively “machine – independent”. Languages which are well- defined tend to be more portable than others.

7. **Efficiency**

An “efficient” language is one which permits fast compilation and execution on the machines where it is implemented. Traditionally, FORTRAN and COBOL have been relatively efficient languages in their respective application areas.

8. **Pedagogy**

Some languages have better “pedagogy” than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

9. **Generality**

This means that a language is useful in a wide range of programming applications. For instance, APL has been used in mathematical applications involving matrix algebra and in business applications as well.

PROGRAMMING PARADIGMS

A programming paradigm provides for the programmer the means structure for the execution of a program. A concept by which the methodology of a programming language adheres to. Paradigms are important because they define a programming language and how it works. A great way to think about a paradigm is as a set of ideas that a programming language can use to perform tasks in terms of machine-code at a much higher level.

Common Language paradigms are:

1. Imperative paradigm/languages.
2. Declarative paradigm/languages.
3. Functional paradigm/languages.
4. Object Oriented paradigm/languages.
5. Procedural paradigm/languages.
6. Logic paradigm/languages.
7. Rule- Based paradigm/languages.

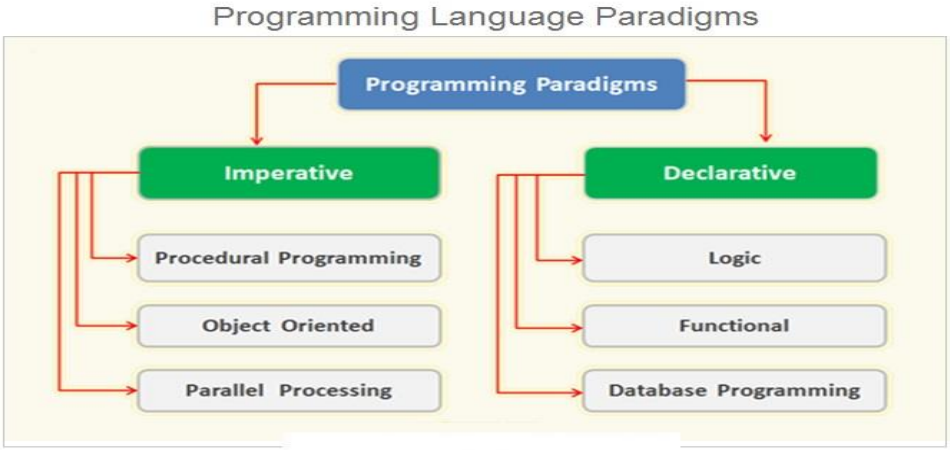


***Figure 2**: Common programming language Paradigms.*

Usually in programming methodology, there are two significant categories that a language will fall into:

A. Declarative
B. Imperative

1. Imperative programming paradigm:

It is one of the oldest programming paradigm. It features close relation to machine architecture. The goal is to understand a machine state (set of memory locations, each containing a value). It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consists of several statements and after execution of all the result is stored. Examples of such Languages are; (FORTRAN, Algol, Pascal, COBOL, Basic and C).

Advantage:

- Very simple to implement
- It contains loops, variables etc.

Disadvantage:

- Complex problem cannot be solved
- Less efficient and less productive
- Parallel programming is not possible

Example: // Participants List

```
$participantlist = [1 => 'Peter', 2 => 'Henry', 3 => 'Sarah'];
$firstnames= [];
foreach ($participantlist as $id => $name) {
    $firstnames[] = $name;
}
```

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

2. Declarative programming paradigm:

It is divided as Logic, Functional, Database. In computer science the declarative programming is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic. It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is

actually doing. It just declares the result we want rather how it has been produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database. Examples of such Languages are; (JavaScript, Scala and Lisp).

Advantage:

- Minimizes mutability
- high level of abstraction
- More understandable code

Disadvantage:

- Sometimes hard to understand for external people
- susceptibility to errors

Example: // Participants List

```
$firstnames = array_values($participantlist);
```

C. Functional programming paradigm

The key principle of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions. The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like Perl, Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket mostly uses this paradigm. The goal is to understand the function that produces the answer. Function composition is major operation (ML, LISP). Syntax: P1 (P2 (P3(X))). Programming consists of building the function that computes the answer

Advantage:

- Pure functions are easier to understand because they don't change any states and depend only on the input given to them.
- More readable and easily understandable
- It adopts lazy evaluation which avoids repeated evaluation

Disadvantage:

- Immutable values and recursion can lead to decrease in performance.
- *Combining them with rest of application and I/O operations is the difficult task*

Example: // Sum of two numbers

```
    sum(x, y)// sum is function taking x and y as arguments
    return x + y// sum is returning sum of x and y without changing them
```

4. Object-Oriented programming paradigm:

The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather than procedure. It can handle almost all kind of real-life problems which are today in scenario. In other words, it sends messages between objects to simulate the temporal evolution of a set of real-world phenomena. Imperative languages that merge applicative design with imperative statements. Programs are built from objects, objects combine functions and data, objects are grouped in classes (e.g., Smalltalk, C++, C#, Java, Python, Ruby)

Advantage:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

Disadvantage:

- Complexity of creating programs
- *Steep learning curve*

Example: // Printing Account Number and Balance

```
Class Account
{
 int account_number;
 int account_balance;
 public void showdata()
 {
 system.out.println("Account Number"+account_number)
 system.out.println("Account Balance"+ account_balance)
 }
}
```

Above you have a Class named as: *Account*
Attributes are: *account_number*, *account_balance*
Method/Action is: *showdata()*

5. Procedural paradigm/languages:

This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability. (e.g., ColdFusion, Pascal, C++, C and Java)

Advantage:

- The coding is easy and simple.
- Reusable in several parts of the program.
- Consumes less memory on the computer.
- Easier tracking of the flow of codes in the program.
- Regarded best for general programming to learn and implement.

Disadvantage:

- Focus on functions rather than data.
- In large program, it is difficult to identify the belonging of global data.
- The modification of global data requires the modification of those functions using it.
- Maintaining and enhancing program code is still difficult.
- Does not model the real-world problem very well.

Example: // Functional way of finding sum of a list

```
import functools
mylist = [11, 22, 33, 44]

# Recursive Functional approach
def sum_the_list(mylist):

    if len(mylist) == 1:
        return mylist[0]
    else:
        return mylist[0] + sum_the_list(mylist[1:])
# lambda function is used
print(functools.reduce(lambda x, y: x + y, mylist))
```

6. Logic paradigm/languages:

Answer a question via search for a solution. Based on axioms, inference rules and queries. Logical paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement. Example: Prolog, Alma-0 and Datalog.

Advantage:

- Removes possibilities of committing errors in classes by suppression.
- Enables the programming of massively parallel computers.

Disadvantage:

- Less efficiency

- There is no suitable method of representing computational concepts originate in a built-in mechanism of state variables like it is found in conventional languages.

Example : // sum of two number in prolog

```
  predicates
  sumoftwonumber(integer, integer)
clauses
  sum(0, 0).
   sum(n, r):-
        n1=n-1,
        sum(n1, r1),
        r=r1+n
```

7. Parallel Processing:

Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posse many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function

Advantage:

- Solve Larger Problems in a short point of time.
- Better suited for modeling, simulating and understanding complex, real-world phenomena.

Disadvantage:

- Costs can sometimes be quite large.
- Power consumption is huge by the multi-core architectures.

Example: // Y = (4 x 5) + (1 x 6) + (5 x 3)

```
//On a single processor, the steps needed to calculate a value for Y might look like:

Step 1: Y = 20 + (1 x 6) + (5 x 3)
Step 2: Y = 20 + 6 + (5 x 3)
Step 3: Y = 20 + 6 + 15
Step 4: Y = 41
```

```
//But in a parallel computing scenario, with three processors or computers, the steps look something like:

Step 1: Y = 20 + 6 + 15
Step 2: Y = 41
```

8. Database/Data driven programming approach:

This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example, SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So, it has its own wide application.

Example: // Creating a database "databaseAddress" with "Addr" table.

```
CREATE DATABASE databaseAddress;
CREATE TABLE Addr (
    PersonID int,
    LastName varchar(200),
    FirstName varchar(200),
    Address varchar(200),
    City varchar(200),
    State varchar(200)
);
```

9. Rule-based paradigm/languages:

Specify rule that specifies problem solution (Prolog, BNF Parsing). Other examples: Decision procedures, Grammar rules (BNF). Syntax: specification rule. Programming consists of specifying the attributes of the answer.

INFLUENCES ON PROGRAMMING LANGUAGE DESIGN

1) **Computer Architecture**: Languages are developed around the prevalent computer architecture, known as the von Neumann architecture

2) **Programming Methodologies**: New software development methodologies (e.g. object-oriented software development) led to new programming paradigms and by extension, new programming languages

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

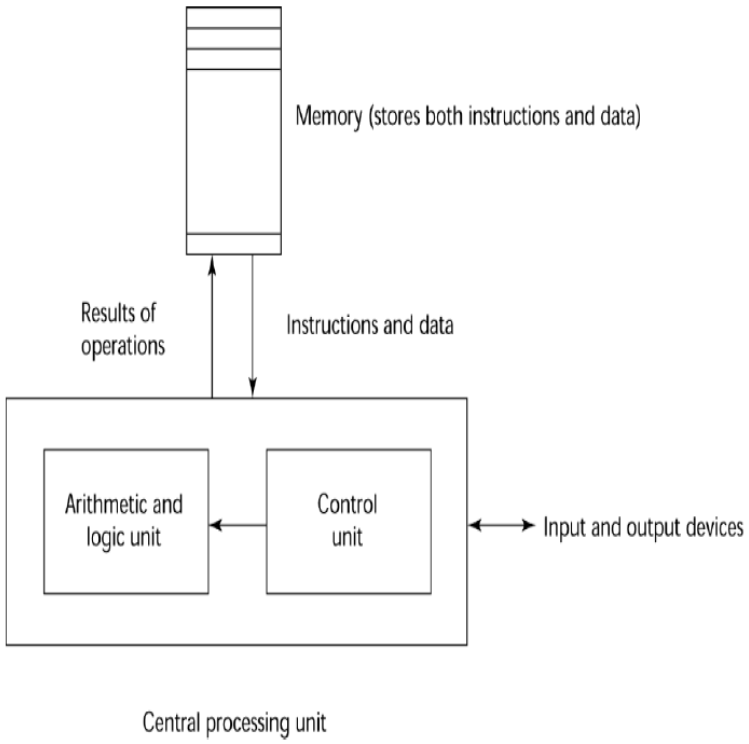


***Figure 3**: The von Neumann Computer Architecture.*

The connection speed between a computer's memory and its processor determines the speed of that computer. Program instructions often can be executed much faster than the speed of the connection; the connection speed thus, results in a bottleneck (Von Neumann bottleneck). It is the primary limiting factor in the speed of computers.

Programming Methodologies

There are many types of programming methodologies prevalent among software developers −

Procedural Programming

Problem is broken down into procedures, or blocks of code that perform one task each. All procedures taken together form the whole program. It is suitable only for small programs that have low level of complexity.

Example − For a calculator program that does addition, subtraction, multiplication, division, square root and comparison, each of these operations can be developed as separate procedures. In the main program each procedure would be invoked on the basis of user's choice.

Object-oriented Programming

Here the solution revolves around entities or objects that are part of problem. The solution deals with how to store data related to the entities, how the entities behave and how they interact with each other to give a cohesive solution.

Example − If we have to develop a payroll management system, we will have entities like employees, salary structure, leave rules, etc. around which the solution must be built.

Functional Programming

Here the problem, or the desired solution, is broken down into functional units. Each unit performs its own task and is self-sufficient. These units are then stitched together to form the complete solution.

Example − A payroll processing can have functional units like employee data maintenance, basic salary calculation, gross salary calculation, leave processing, loan repayment processing, etc.

Logical Programming

Here the problem is broken down into logical units rather than functional units.

Example: In a school management system, users have very defined roles like class teacher, subject teacher, lab assistant, coordinator, academic in-charge, etc. So the software can be divided into units depending on user roles. Each user can have different interface, permissions, etc.

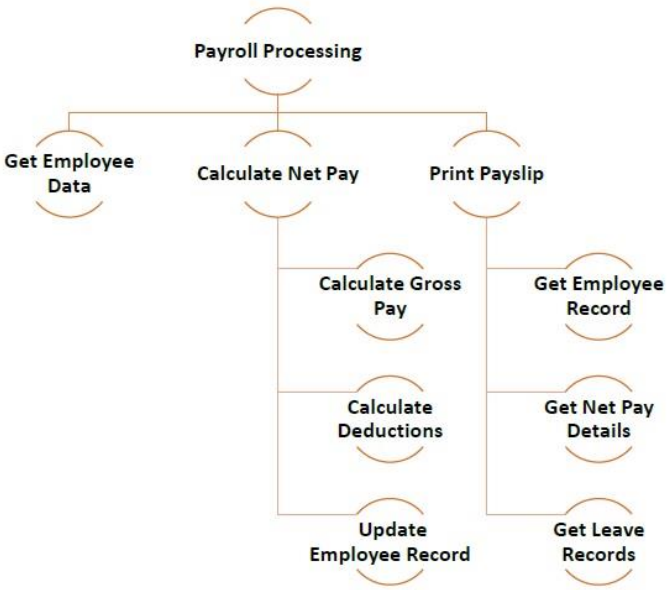
Software developers may choose one or a combination of more than one of these methodologies to develop a software. Note that in each of the methodologies discussed, problem has to be broken down into smaller units. To do this, developers use any of the following two approaches −

- Top-down approach
- Bottom-up approach

Top-down or Modular Approach

The problem is broken down into smaller units, which may be further broken down into even smaller units. Each unit is called a **module**. Each module is a self-sufficient unit that has everything necessary to perform its task.

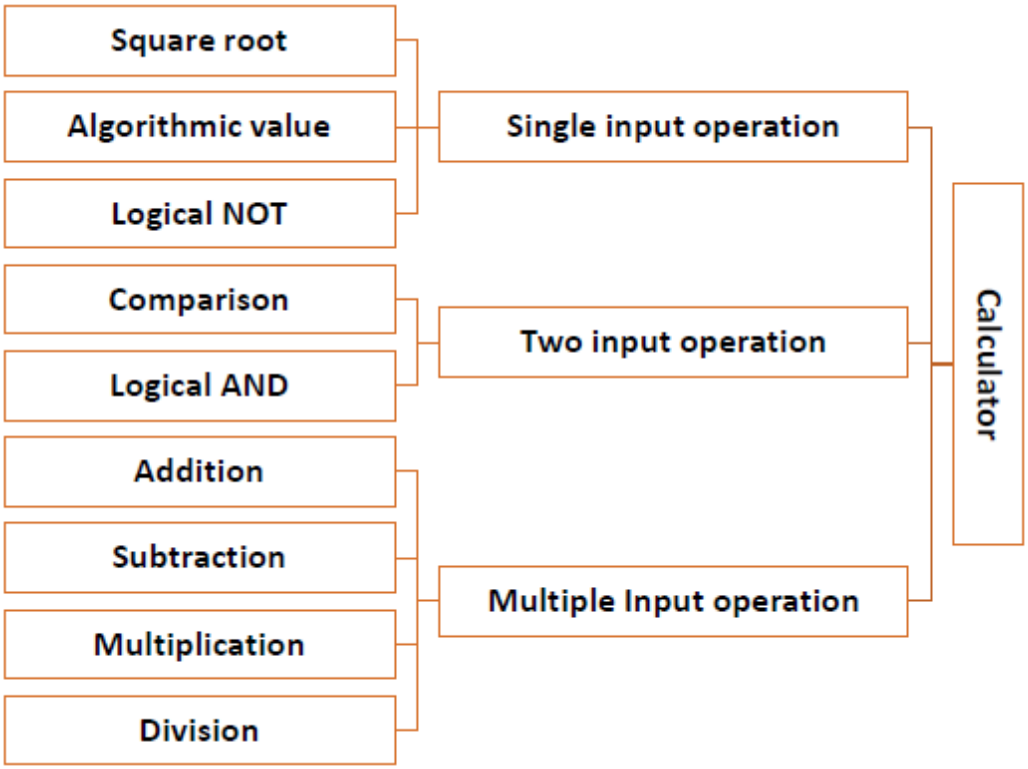
The following illustration shows an example of how you can follow modular approach to create different modules while developing a payroll processing program.



Bottom-up Approach

In bottom-up approach, system design starts with the lowest level of components, which are then interconnected to get higher level components. This process continues till a hierarchy of all system components is generated. However, in real-life scenario it is very difficult to know all lowest level components at the outset. So, bottoms up approach is used only for very simple problems.

Let us look at the components of a calculator program.



TRADE-OFFS IN LANGUAGE DESIGN

1. **Reliability Vs. Cost of Execution:**
   For example, Java demands that all references to array elements be checked for proper indexing, which leads to increased execution costs.

2. **Readability vs. Writability:**
   APL provides many powerful operators land a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.

3. **Writability (Flexibility) vs. reliability:**
   The pointers in C++ for instance are powerful and very flexible but are unreliable.

IMPLEMENTATION METHODS

A. **Compilation**: Programs are translated into machine Language & System calls. Translated high level program (source language) into machine code (machine language)
   - Slow translation, fast execution
   - Compilation process has several phases
     - Lexical analysis converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
     - Syntactic analysis: transforms lexical units into parse trees which represent the syntactic structure of the program.
     - Semantics analysis check for errors hard to detect during syntactic analysis; generate intermediate code.
     - Code generation – Machine code is generated
   - Source program translated ("compiled") to another language

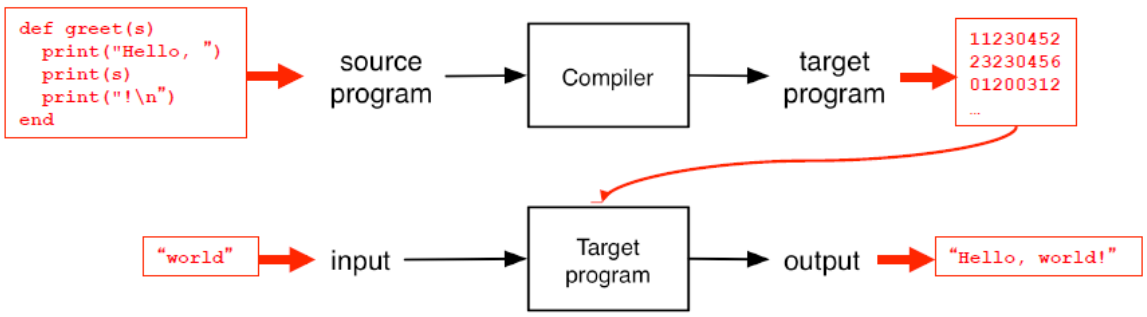


***Figure 4**: How a program runs through a compiler.*

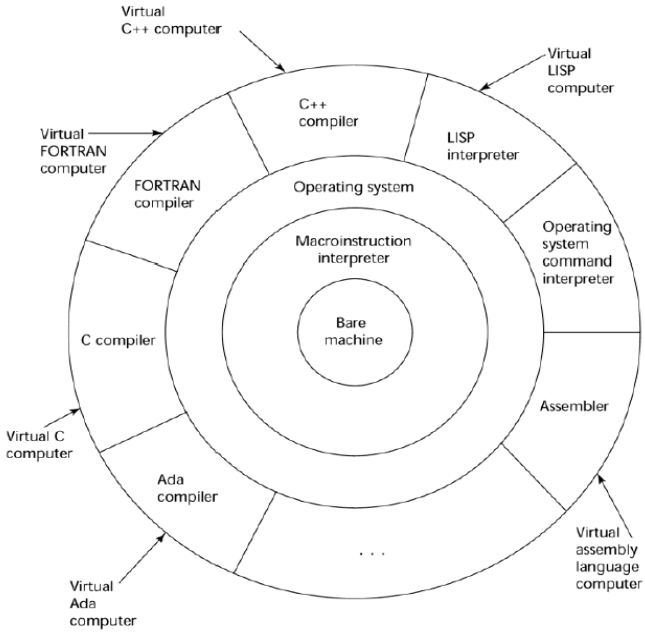


***Figure 4.1**: Layered View of Computer: The operating system and language implementation are layered over Machine interface of a computer.*

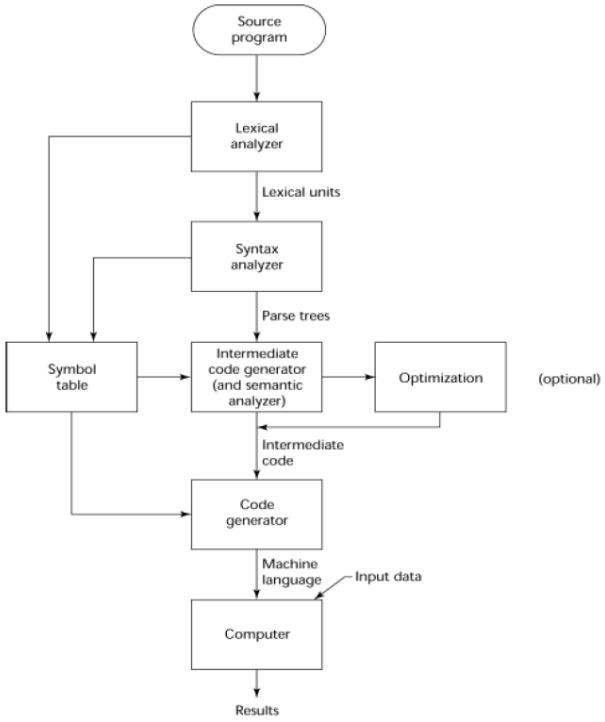


***Figure 4.2**: The Compilation Process*

B. **Interpretation**: Programs are interpreted by another program (an interpreter).
   - Easier implementation of programs (run-time errors can easily and immediately be displayed).
   - Slower execution (10 to 100 times slower than compiled programs)
   - Often requires more memory space and is now rare3 for traditional high-level languages.
   - Significant comeback with some Web scripting languages like PHP and JavaScript.
   - Interpreters usually implement as a read-eval-print loop:
   - Interpreters act as a virtual machine for the source language:
   - Interpreter executes each instruction in source program one step at a time, No separate executable.

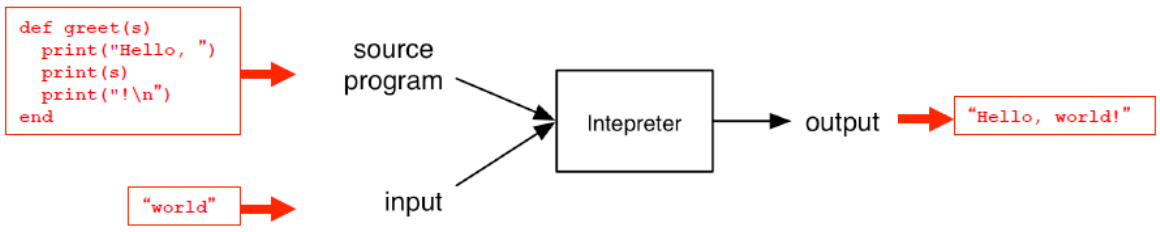


***Figure 4.3**: How a program runs through a compiler.*

C. **Hybrid**: Programs translated into an intermediate language for easy interpretation.
   - This involves a compromise between compilers and pure interpreters. A high-level program is translated to an intermediate language that allows easy interpretation.
   - Hybrid implementation is faster than pure interpretation.
   - Examples of the implementation occur in Perl and Java.
     - 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 run time system (together, these are called Java Virtual Machine).

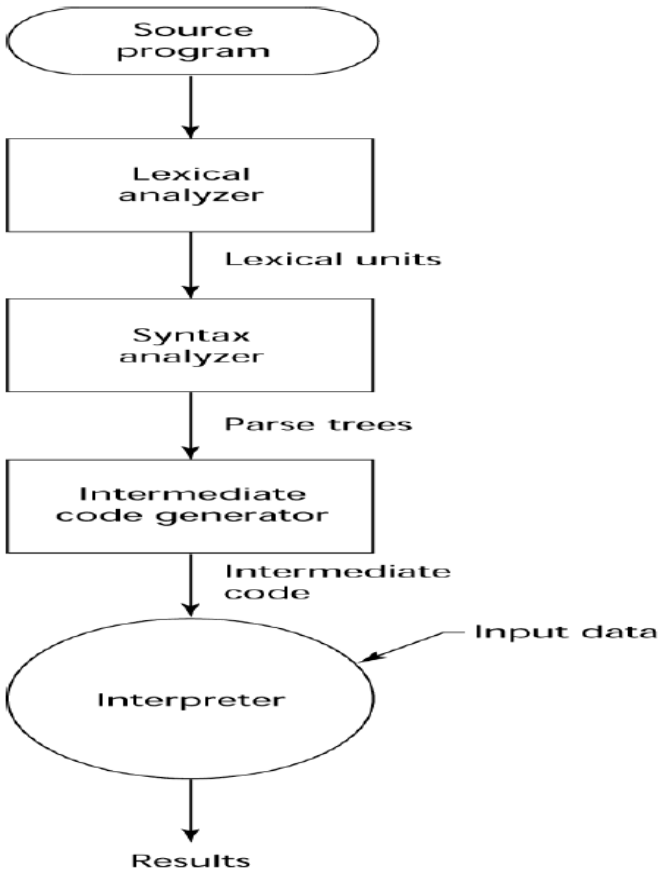


***Figure 4.4**: Hybrid Implementation.*

D. **Just –in-time**: Hybrid implementation, then compile sub programs code the first time they are called.
   - This implementation initially translates programs to an intermediate language then compile the intermediate language of the subprograms into machine code when they are called.
     - Machine code version is kept for subsequent calls. Just-in-time systems are widely used for Java programs. Also, .NET languages are implemented with a JIT system.

COMPILERS VS INTERPRETERS

- **COMPILERS**
  - Generated code more efficient
  - "Heavy"

- **INTERPRETERS**
  - Great for debugging
  - Slow

- **IN PRACTICE**
  - "General-purpose" programming languages (e.g., C, Java) are often compiled, although debuggers provide interpreter support
  - Scripting languages and other special-purpose languages are interpreted, even if general purpose.

SYNTAX AND SEMANTICS OF PROGRAMMING LANGUAGES

Syntax deals with form: is this program text well-formed? Semantics deals with meaning: what does this do?

- **Syntax** – the form of the expressions, statements, and program units
- **Semantics** - the meaning of the expressions, statements, and program units.

Example:

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.

THE GENERAL PROBLEM OF DESCRIBING SYNTAX

- A **sentence** or "**statement**" is a string of characters over some alphabet. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.
- A **language** is a set of sentences.
- A **lexeme** is the lowest level syntactic unit of a language. It includes identifiers, literals, operators, and special word. (e.g. *, sum, begin) A **program** is strings of lexemes.
- A **token** is a **category** of lexemes (e.g., identifier.) An identifier is a token that have lexemes, or instances, such as sum and total.

Example:

*index = 2 * count + 17;*

Lexemes	***Tokens***
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

- **Languages Recognizers** is a recognition device reads input strings of the language and decides whether the input strings belong to the language.
  Example: syntax analysis part of a compiler
- **Languages Generators** is a device that generates sentences of a language. One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator.

FORMAL METHODS OF DESCRIBING SYNTAX

- *Backus-Naur Form (BNF) and Context-Free Grammars*: Most widely known method for describing programming language syntax
- *Extended BNF*: Improves readability and writability of BNF
- *Grammars and Recognizers*

Backus-Naur Form and Context-Free Grammars

- Context-Free Grammars
- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58
  - BNF is equivalent to context-free grammars
  - BNF is a *metalanguage* used to describe another language
  - In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (*nonterminal symbols*)

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules

Examples of BNF rules:

<ident_list> → identifier | identifer, <ident_list>

<if_stmt> → ***if*** *<logic_expr>* ***then*** *<stmt>*

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- A grammar is a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

<stmt> → <single_stmt>

| ***begin*** *<stmt_list>* ***end***

Describing Lists

- Syntactic lists are described using recursion

<ident_list> → ident

| ident, <ident_list>

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Parse Tree

A hierarchical representation of a derivation

An example derivation

<program>⇨<stmts>

⇨<stmt>

⇨<var>=<expr>

⇨a=<expr>

⇨a=<term>+<term>

⇨a=<var>+<term>

⇨a=b+<term>

⇨a=b+const

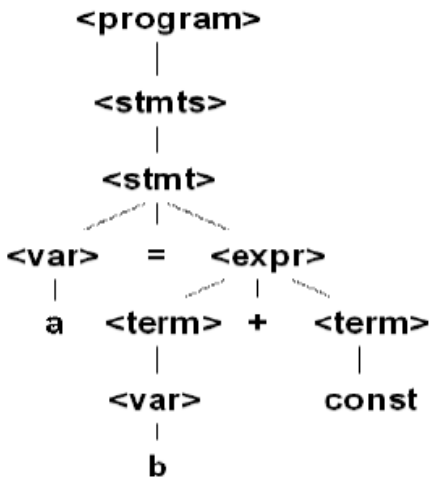


Figure 8*: Parse Tree.*

Derivation

- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Ambiguity in Grammars
A grammar is *ambiguous* if it generates a sentential form that has two or more distinct parse trees

An Unambiguous Expression Grammar
If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

<expr> → <expr> - <term>|<term>
<term> → <term> / const|const

DESCRIBING THE MEANINGS OF PROGRAMS: DYNAMIC SEMANTICS

- There is no single widely acceptable notation or formalism for describing semantics. There are Three primary methods of semantics description ***Operation***, ***Axiomatic*** and ***denotational*.**

Operational Semantics

- Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed
- *A better alternative:* A complete computer simulation
- *The process*:
    - Build a translator (translates source code to the machine code of an idealized computer)
    - Build a simulator for the idealized computer

Axiomatic Semantics:

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called assertions

Denotational Semantics:

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- The process of building a denotational spec for a language:
- Define a mathematical object for each language entity
- Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

- The meaning of language constructs are defined by only the values of the program's variables.
- The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions
- The state of a program is the values of all its current variables

$$s = \{<i1, v1>, <i2, v2>, \cdots, <in, vn>\}$$

- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable

$$VARMAP(ij, s) = vj$$

- Decimal Numbers:
  - The following denotational semantics description maps decimal numbers as strings of symbols into numeric values

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 |5 | 6 | 7 | 8 | 9)
Mdec('0') = 0, Mdec ('1') = 1, ⋯, Mdec ('9') = 9
*Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>)*
*Mdec (<dec_num> '1') = 10 * Mdec (<dec_num>) + 1*
…
*Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9*

DATA TYPES AND VARIABLES

Introduction

A data type defines a collection of data values and a set of predefined operations on those values. Computer programs produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types available in the language being used match the objects in the real world of the problem being addressed. Therefore, it is crucial that a language supports an appropriate collection of data types and structures.

- A data type defines a collection of data objects and a set of predefined operations on those objects.
- A descriptor is the collection of the attributes of a variable.
- An object represents an instance of a user-defined (abstract data) type.
- One design issue for all data types: What operations are defined and how are they specified?

1) Primitive Data Types

Data types that are not defined in terms of other types are called primitive data types. Nearly all programming languages provide a set of primitive data types. Some of the primitive types are merely reflections of the hardware — for example, most integer types. Others require only a little nonhardware support for their implementation. To specify the structured types, the primitive data types of a language are used, along with one or more type constructors.

- ***Numeric Types***

  Some early programming languages only had numeric primitive types. Numeric types still play a central role among the collections of types supported by contemporary languages.

  - **Integer**

    The most common primitive numeric data type is integer. The hardware of many computers supports several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. For example, Java includes four signed integer sizes: byte, short, int, and long. Some languages, for example, C++ and C#, include unsigned integer types, which are types for integer values without signs. Unsigned types are often used for binary data.

  - **Floating-Point**
    - Model real numbers, but only as approximations
    - Languages for scientific use support at least two floating-point types (e.g., float and double; sometimes more
    - Usually exactly like the hardware, but not always
    - IEEE Floating-Point Standard 754

  - **Complex**

    Some programming languages support a complex data type—for example, Fortran and Python. Complex values are represented as ordered pairs of floating-point values. In Python, the imaginary part of a complex literal is specified by following it with a j or J — for example, *(7 + 3j)*

    Languages that support a complex type include operations for arithmetic on complex values.

  - **Decimal**

    Most larger computers that are designed to support business systems applications have hardware support for decimal data types. Decimal data types store a fixed number of decimal digits, with the implied decimal point at a fixed position in the value. These are the primary data types for business data processing and are therefore essential to COBOL. C# and F# also have decimal data types. Decimal types have the advantage of being able to precisely

store decimal values, at least those within a restricted range, which cannot be done with - floating-point. For example, the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type.

- ***Boolean Types***
  This is the simplest of all data types with range values of two elements, one for “true” and one for “false”. It can be implemented as bits, but often as bytes. The advantage of this numerical data type is readability.

- ***Character Types***
  Character data are stored in computers as numeric coding. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters.
  To provide the means of processing coding of single characters, most programming languages include a primitive type for them. However, Python supports single characters only as character strings of length 1.

2) Character String Types

A character string type is one in which the values consist of sequences of characters. Character string constants are used to label output, and the input and output of all kinds of data are often done in terms of strings. Of course, character strings also are an essential type for all programs that do character manipulation.

3) Enumeration Types

An enumeration type is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition. Enumeration types provide a way of defining and grouping collections of named constants, which are called enumeration constants. The definition of a typical enumeration type is shown in the following C# example:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration constants are typically implicitly assigned the integer values, 0, 1, . . . but can be explicitly assigned any integer literal in the type’s definition.

4) 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 kind of slices supported?

5) Associative Arrays

An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys

– User-defined keys must be stored

Design issues:

– What is the form of references to elements?

– Is the size static or dynamic?

6) Record Types

A record is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure. There is frequently a need in programs to model a collection of data in which the individual elements are not of the same type or size. For example, information about a college student might include name, student number, grade point average, and so forth.

Design issues:

– What is the syntactic form of references to the field?

– Are elliptical references allowed?

Definition of Records in COBOL

COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
    02 EMP-NAME.
        05 FIRST PIC X(20).
        05 MID PIC X(10).
        05 LAST PIC X(20).
    02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
        Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

7) Tuple Types

A tuple is a data type that is similar to a record, except that the elements are not named.

Python includes an immutable tuple type. If a tuple needs to be changed, it can be converted to an array with the list function. After the change, it can be converted back to a tuple with the tuple function. One use of tuples is when an array must be write-protected, such as when it is sent as a parameter to an external function and the user does not want the function to be able to modify the parameter. Python's tuples are closely related to its lists, except that tuples are immutable. A tuple is created by assigning a tuple literal, as in the following

example:

```
myTuple = (3, 5.8, 'apple')
```

Notice that the elements of a tuple need not be of the same type.

The elements of a tuple can be referenced with indexing in brackets, as in the following: *myTuple[1]*

This references the first element of the tuple, because tuple indexing begins at 1.

Tuples can be catenated with the plus (+) operator. They can be deleted with the del statement. There are also other operators and functions that operate on tuples.

8) List Types

Lists were first supported in the first functional programming language, Lisp. They have always been part of the functional languages, but in recent years they have found their way into some imperative languages.

Lists in Scheme and Common Lisp are delimited by parentheses and the elements are not separated by any punctuation. For **example**,

```
(A B C D)
```

Nested lists have the same form, so we could have

```
(A (B C) D)
```

In this list, *(B C)* is a list nested inside the outer list.

9) Union Types

A union is a type whose variables are allowed to store different type values at different times during execution

Design issues

– Should type checking be required?

– Should unions be embedded in records?

Discriminated vs. Free Unions

Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union

Type checking of unions require that each union include a type indicator called a discriminant

– Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
        Filled: Boolean;
        Color: Colors;
        case Form is
                when Circle => Diameter: Float;
                when Triangle =>Leftside, Rightside: Integer;
                        Angle: Float;
                when Rectangle => Side1, Side2: Integer;
        end case;
end record;
```

10) Pointer and Reference Types

A pointer type is one in which the variables have a range of values that consists of memory addresses and a special value, nil. The value nil is not a valid address and is used to indicate that a pointer cannot currently be used to reference a memory cell.

Pointers are designed for two distinct kinds of uses. First, pointers provide some of the power of indirect addressing, which is frequently used in assembly language programming. Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in an area where storage is dynamically allocated called a heap.

Design issues

– What are the scope and lifetime of a pointer variable?

– What is the lifetime of a heap-dynamic variable (the value a pointer references)?

– Are pointers restricted as to the type of value to which they can point?

– Are pointers used for dynamic storage management, indirect addressing, or both?

– Should the language support pointer types, reference types, or both?

11) Optional Types

There are situations in programming when there is a need to be able to indicate that a variable does not currently have a value. Some older languages use zero as a nonvalue for numeric variables. This approach has the disadvantage of not being able to distinguish between when the variable is supposed to have the zero value and when the zero indicates that it has no value. Some newer languages provide types that can have a normal value or a special value to indicate that their variables have no value. Variables that have this capability are called optional types. Optional types are now directly supported in C#, F#, and Swift, among others.

C# has two categories of variables, value and reference types. Reference types, which are classes, are optional types by their nature. The null value indicates that a reference type has no value. Value types, which are all struct types, can be declared to be optional types, which allows them to have the value null. A variable is declared to be an optional type by following its type name with a question mark (?), as in

```
int? x;
```

To determine whether a variable has a normal value, it can be tested against **null**, as in

```
int? x;
. . .
if(x == null)
        Console.WriteLine("x has no value");
else
        Console.WriteLine("The value of x is: {0}", x);
```

12) Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of 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. This automatic conversion is called as coercion.
- A type error is the application of an operator to an operand of an inappropriate type
- If all type bindings are static, nearly all type checking can be static

- If type bindings are dynamic, type checking must be dynamic
- Def: A programming language is strongly typed if type errors are always detected

13) Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, EQUIVALENCE
  - Pascal is not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Ada is, almost (UNCHECKED CONVERSION is loophole) (Java is similar)
- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada.