

Chapter 1

Preliminaries

1.1 Reasons for Studying Concepts of Programming Languages	2
1.2 Programming Domains	5
1.3 Language Evaluation Criteria	7
1.4 Influences on Language Design	17
1.5 Language Categories	20
1.6 Language Design Trade-Offs	21
1.7 Implementation Methods	22
1.8 Programming Environments	29
Summary • Review Questions • Problem Set	30

Chapter 1

Preliminaries

1.1 Reasons for Studying Concepts of Programming Languages 2

- *Increased ability to **express** ideas*
 - It is believed that the depth at which we think is influenced by the expressive power of the language in which we communicate our thoughts. It is difficult for people to conceptualize structures they **can't describe**, verbally or in writing.
 - Language in which they develop software places **limits** on the kinds of control structures, data structures, and abstractions **they can use**.
 - Awareness of a **wider variety** of programming language features can **reduce such limitations** in software development.
 - Can language constructs be simulated in other languages that do not support those constructs directly? **Associative arrays** in Perl vs. C
- *Improved background for **choosing** appropriate languages*
 - Many programmers, when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to new projects.
 - If these programmers were familiar with other languages available, they would be in a better position to make informed language choices.
- *Increased ability to learn **new** languages*
 - Programming languages are still in a state of continuous evolution, which means continuous learning is essential.
 - Programmers who understand the concept of **object oriented programming** will have easier time learning **Java** and **Ruby**.
 - Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes easier to see how concepts are incorporated into the design of the language being learned.
 - The TIOBE Programming Community issues an index (<https://www.tiobe.com/tiobe-index>) that is an indicator of the relative popularity of programming languages. For example, according to the index, Java, C, C++, and C# were the four most **popular** languages in use in February 2017.
- *Better understanding of significance of **implementation***
 - Understanding of implementation issues leads to an understanding of why languages are designed the way they are.
 - This in turn leads to the ability to use a language more intelligently, as it was designed to be used.

- For example, programmers who know little about how recursion is implemented often do not know that a **recursive** algorithm can be far **slower** than an equivalent **iterative** algorithm.
- *Better use of languages that are already known*
 - The more languages you gain knowledge of, the better understanding of programming languages concepts you understand.
 - Programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.
- *Overall advancement of computing*
 - In some cases, a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.
 - Many believe that ALGOL 60 was a **better** language than Fortran; however, Fortran was most widely used. It is attributed to the fact that the programmers and managers **didn't** understand the conceptual design of ALGOL 60.

1.2 Programming Domains 5

- **Scientific applications**
 - In the early 40s computers were invented for scientific applications.
 - The applications require **large number of floating point computations**.
 - **Fortran** was the first language developed scientific applications.
 - **ALGOL 60** was intended for the same use.
- **Business applications**
 - The first successful language for business was **COBOL**.
 - Business languages are characterized by facilities for producing elaborate **reports**, **precise** ways of describing and storing **decimal numbers and character** data, and the ability to specify decimal arithmetic operations.
 - The arrival of PCs started new ways for businesses to use computers.
 - **Spreadsheets and database systems** were developed for business.
- **Artificial intelligence**
 - Symbolic rather than numeric computations are manipulated.
 - Symbolic computation is more suitably done with linked lists than arrays.
 - **LISP** was the first widely used AI programming language.
 - An alternative approach to AI applications: **Prolog**
 - **Scheme**, a dialect of LISP
- **Systems programming**
 - The OS and all of the programming supports tools are collectively known as its system software.
 - Need efficiency because of continuous use.
 - A language for this domain must provide **fast execution**. Furthermore, it must have **low-level features** that allow the software interfaces to external devices to be written.
 - The **UNIX** operating system is written almost entirely in **C**.
- **Web software**
 - The World Wide Web is supported by an eclectic collection of languages: markup languages (e.g. HTML), scripting (e.g. PHP), general-purpose (e.g. Java)
 - **JavaScript** is used mostly as a **client-side** scripting language. JavaScript is **embedded** in HTML documents and is **interpreted** by a browser that finds the code in a document that is being displayed.
 - **PHP** is a scripting language used on Web server systems. Its code is **embedded** in HTML documents. The code is **interpreted** on the **server** before the document is sent to a requesting browser.

1.3 Language Evaluation Criteria 6

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

1.3.1 Readability

- The most important criteria for judging a programming language is the ease with which programs can be **read and understood**.
- Language constructs were designed **more** from the point of view of the computer than the users.
- Because ease of **maintenance** is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages. The result is a crossover from focus on machine orientation to focus on human orientation.
- The most important criterion “ease of use”
- **Overall simplicity** “Strongly affects readability”
 - **Too many features** make the language difficult to learn. Programmers tend to learn a subset of the language and ignore its other features.
 - **Multiplicity of features** is also a complicating characteristic “having more than one way to accomplish a particular operation. For example, in **Java**, a user can increment a simple integer variable in **four** different ways:

```
count = count + 1
count += 1
count ++
++count
```

- Although the last two statements have slightly different meaning from each other and from the others, all four have the same meaning when used as stand-alone expressions.
- **Operator overloading** where a single operator symbol has more than one meaning.
- Although this is a useful feature, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly.
- Most **assembly language** statements are models of simplicity.
- This very simplicity, however, makes assembly language programs less readable. Because they lack more complex control statements.

- **Orthogonality**

- Makes the language easy to learn and read.
- A relatively small set of primitive constructs can be **combined** in a relatively small number of **ways** to build the control and data structures of the language.
- Every possible combination is legal and meaningful.
- The **more** orthogonal the design of a language, the **fewer exceptions** the language rules require.
- Example: In C language, parameters are passed by **value**, unless they are arrays, in which they are, in effect, passed by **reference** (because the appearance of an array name without a subscript in a C program is interpreted to be the **address** of the array's **first element**).
- Example: Adding two 32-bit integer values that reside in either memory or registers and replacing one of two values with the sum.
 - The **IBM** mainframes have two instructions:

```
A      Reg1, memory_cell
      //Reg1 <- contents (Reg1) + contents(memory_cell)
AR     Reg1, Reg2
      //Reg1 <- contents (Reg1) + contents(Reg2)
where Reg1 and Reg2 represent registers.
```

- The **VAX** addition instruction for 32-bit integer value is:

```
ADDL  operand_1, operand_2
      //operand_2 <- contents(operand_1) + contents(operand_2)
In this case, either operand can be a register or a memory cell.
```

- The VAX instruction design is **orthogonal** in that a single instruction can use either registers or memory cell as the operands. The IBM design is **not** orthogonal.
- Too much orthogonality can also cause problems.
- Example: the most orthogonal programming language is ALGOL 68. Every language construct in ALGOL 68 has a type, and there are no restrictions on those types.
 - For example, a **conditional** can appear as the left side of an assignment, along with declarations and other assorted statements, as long as the result is an address.
 - This form of orthogonality leads to **unnecessary** complexity.

- **Control Statements**

- It became widely recognized that indiscriminate use of **goto** statements severely reduced program readability.
- Example: Consider the following nested loops written in C

```
while (incr < 20)
{
    while (sum <= 100)
    {
        sum += incr;
    }
    incr++;
}
```

if C didn't have a loop construct, this would be written as follows:

```
loop1:
    if (incr >= 20) go to out;
loop2:
    if (sum > 100) go to next;
    sum += incr;
    go to loop2;
next:
    incr++;
    go to loop1;
out:
```

- Basic and **Fortran** in the early 1970s lacked the control statements that allow strong restrictions on the use of **gotos**, so writing highly readable programs in those languages was difficult.
- Since then, languages have included sufficient control structures. The control statement design of a language is now a **less** important factor in readability than it was in the past.

- **Data Types**

- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to reliability.
- Example: suppose a numeric type is used for an indicator flag because there is **no Boolean** type in the language. In such a language, we might have an assignment such as

```
timeout = 1
```

- Whose meaning is unclear, whereas in a language that includes Boolean types we would have the following:

```
timeout = true
```

The meaning of this statement is perfectly **clear**

- **Syntax Considerations**

- The syntax of the elements of a language has a significant effect on readability.
- The following are examples of syntactic design choices that affect readability:
 - *Identifier forms*: Restricting identifiers to very short lengths detract from readability.
 - Example: In Fortran 77, identifiers can have **six** characters at most.
 - Example: ANSI BASIC (1978) an identifier could consist only of a single letter or a single letter followed by a single digit.
 - *Special Words*: Program appearance and thus program readability are strongly influenced by the forms of a language's special words (**while**, **class**, **for**).
 - Example: **C** uses braces for pairing control structures. It is difficult to determine which group is being ended.
 - Example: **Fortran 95** and **Ada** allows programmers to use special names as legal variable names. Ada uses **end if** to terminate a selection construct, and **end loop** to terminate a loop construct.
 - *Form and Meaning*: Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability. Semantic should follow directly from syntax, or form.
 - Example: In C the use of **static** depends on the context of its appearance.
 - If used as a variable inside a function, it means the variable is created at **compile time**.
 - If used on the definition of a variable that is outside all functions, it means the variable is **visible** only in the file in which its definition appears. It is not exported from that file.

1.3.2 Writability

- It is a measure of how easily a language can be used to **create** programs for a chosen problem domain.
- Most of the language characteristics that affect readability also affect writability.
- **Simplicity and orthogonality**
 - A smaller number of primitive constructs and a consistent set of rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives.
 - A programmer can design a solution to a complex problem after learning only a simple set of primitive constructs.
- **Support for abstraction**
 - **Abstraction** means the ability to define and then use complicated structures or operations in ways that allow many of the details to be **ignored**.
- **Expressivity**
 - It means that a language has relatively **convenient**, rather than cumbersome, ways of specifying computations.
 - Ex: `++count` vs. `count = count + 1` // `++count` more convenient and shorter

1.3.3 Reliability

- A program is said to be **reliable** if it performs to its specifications under all conditions.
- **Type checking:** It is simply testing for type errors in a given program, either by the compiler or during program execution.
 - Type checking is an important factor in language reliability.
 - Run-time type checking is expensive; compile-time type checking is more desirable. Furthermore, the earlier errors are detected, the less expensive it is to make the required repairs.
 - **Java** requires type checking of nearly all variables and expressions at compile time.
- **Exception handling:** the ability to intercept **run-time** errors, take corrective measures, and then continue is a great aid to reliability.
- **Aliasing:** it is having two or more distinct referencing methods, or names, for **the same memory cell**. In C, **union** members and pointers set to point to the same variable.
 - It is now widely accepted that aliasing is a **dangerous** feature in a language.
 - For example, two pointers set to point to the same variable, which is possible in most languages. In such a program, the programmer must always remember that changing the value pointed to by one of two changes the value referenced by the other.
- **Readability and writability:** Both readability and writability influence reliability.

1.3.4 Cost

- The total cost of a programming language is a function of many of its characteristics
 - Training programmers to use the language
 - Writing programs in the language
 - Compiling programs
 - Executing programs
 - Language implementation system
 - One of the factors that explains the rapid acceptance of Java is the free compiler/interpreter system
 - Reliability
 - poor reliability leads to high costs
 - Maintaining programs
- Others
 - Portability
 - The ease with which programs can be moved from one implementation to another
 - Generality
 - The applicability to a wide range of applications
 - Well-definedness
 - The completeness and precision of the language's official defining document

1.4 Influences on Language Design 17

1.4.1 Computer architecture

- We use imperative languages, at least in part, because we use **von Neumann** machines
 - Data and programs stored in same memory
 - Memory is separate from CPU
 - Instructions and data are piped from memory to CPU
 - Results of operations in the CPU must be moved back to memory
 - Basis for imperative languages
 - Variables model memory cells
 - Assignment statements model piping
 - Iteration is efficient

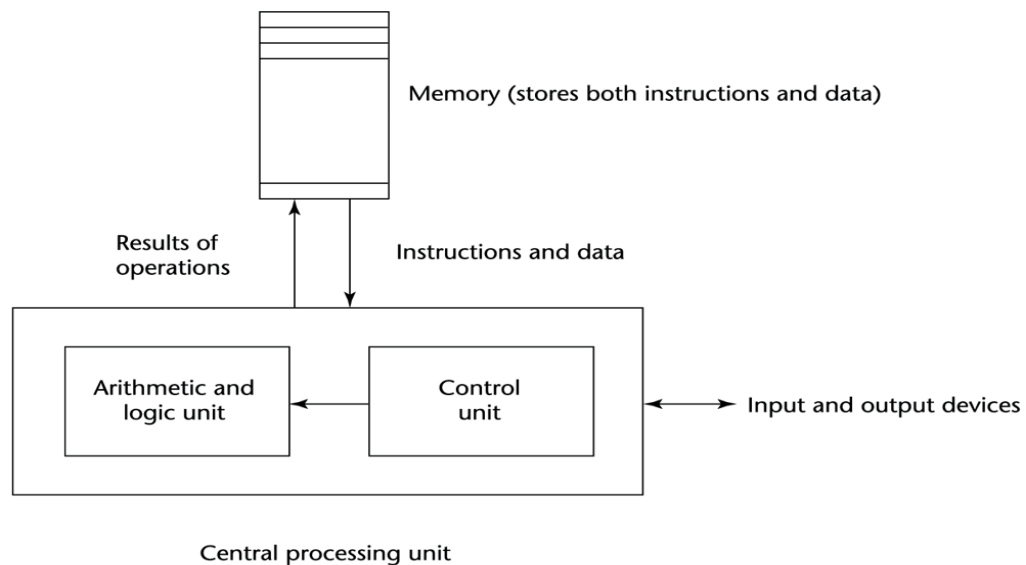


Figure 1.1 The von Neumann computer architecture

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

```
initialize the program counter
repeat forever
    fetch the instruction pointed by the counter
    increment the counter
    decode the instruction
    execute the instruction
end repeat
```

1.4.2 Programming methodologies

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

1.5 Language Categories 20

- Imperative
 - Central features are variables, assignment statements, and iteration
 - Include languages that support object-oriented programming
 - Include scripting languages
 - Include the visual languages
 - Examples: **C, Java, Perl, JavaScript, Visual BASIC .NET, C++**
- Functional
 - Main means of making computations is by applying functions to given parameters
 - Examples: **LISP, Scheme, ML, F#**
- Logic
 - Rule-based (rules are specified in no particular order)
 - Example: **Prolog**
- Markup/programming hybrid
 - Markup languages extended to support some programming
 - Examples: **JSTL, XSLT**

1.6 Language Design Trade-Offs 21

- Reliability vs. Cost of execution
 - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs.
 - C does **not** require index range checking
 - C programs execute **faster** than semantically equivalent Java programs, although Java programs are more reliable.
- Readability vs. Writability
 - Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
 - Well-known author Daniel MaCracken once noted that it took him four hours to read and understand a four-line APL program.
- Writability (flexibility) vs. Reliability
 - Example: C pointers are powerful and very flexible but are unreliable
 - Because of the potential reliability problems with pointers, they are **not** included in Java

1.7 Implementation Methods 22

- The major methods of implementing programming languages are compilation, pure interpretation, and hybrid implementation
 - **Compilation:** Programs are translated into machine language
 - **Pure Interpretation:** Programs are interpreted by another program known as an interpreter
 - **Hybrid Implementation Systems:** A compromise between compilers and pure interpreters
- The operating system and language implementation are **layered** over machine interface of a computer.
- These layers can be thought of as **virtual computers**, providing interfaces to the user at higher levels

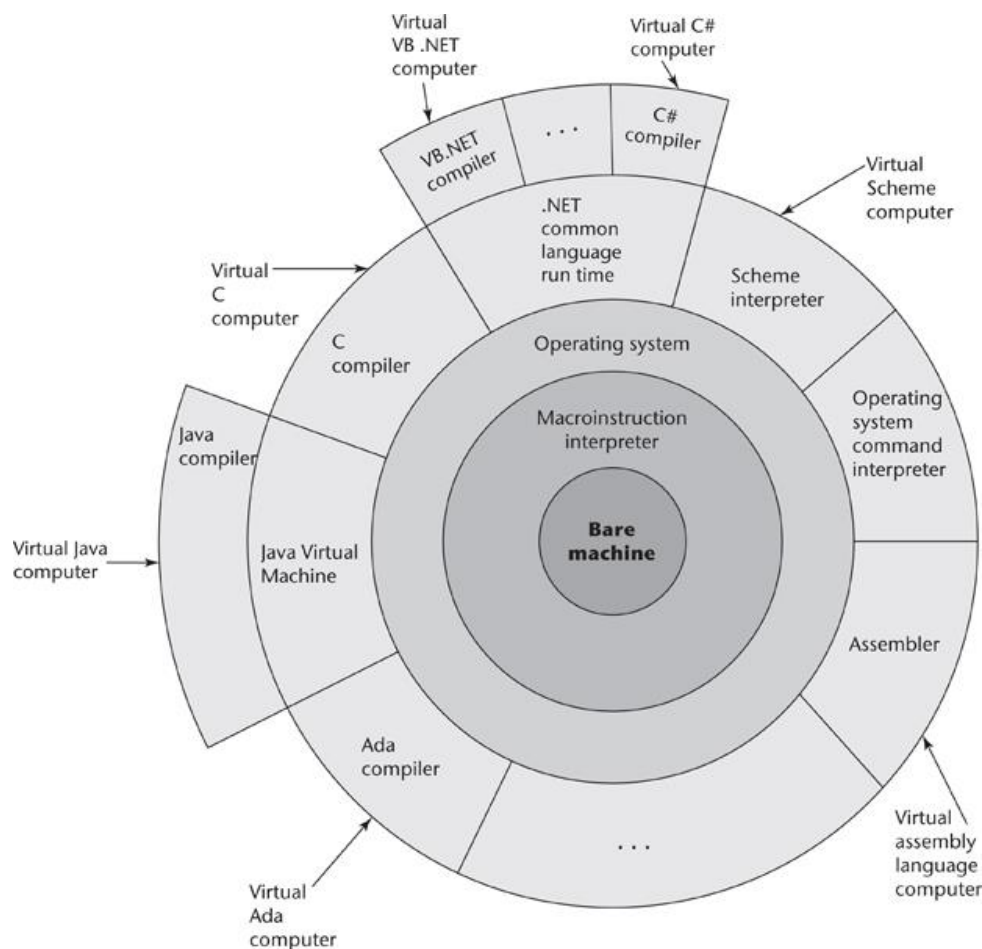


Figure 1.2 Layered interface of virtual computers, provided by a typical computer system

1.7.1 Compilation

- Translate high-level program (source language) into **machine code** (machine language)
- Slow translation, fast execution
- **C, COBOL, C++, and Ada** are by compilers.
- Compilation process has several phases:
 - Lexical analysis: converts characters in the source program into **lexical units**
 - The lexical units of a program are identifiers, special words operators, and punctuation symbols.
 - Syntax analysis: transforms lexical units into **parse trees**
 - These parse trees represent the syntactic structure of program
 - Semantics analysis: generate intermediate code
 - Intermediate languages sometimes look very much like assembly languages and in fact sometimes are actual **assembly language**.
 - **Symbol table**: the type and attribute information of each **user-defined name** in the program
 - **Optimization**: improve programs by making them smaller or faster or both
 - Code generation: machine code is generated

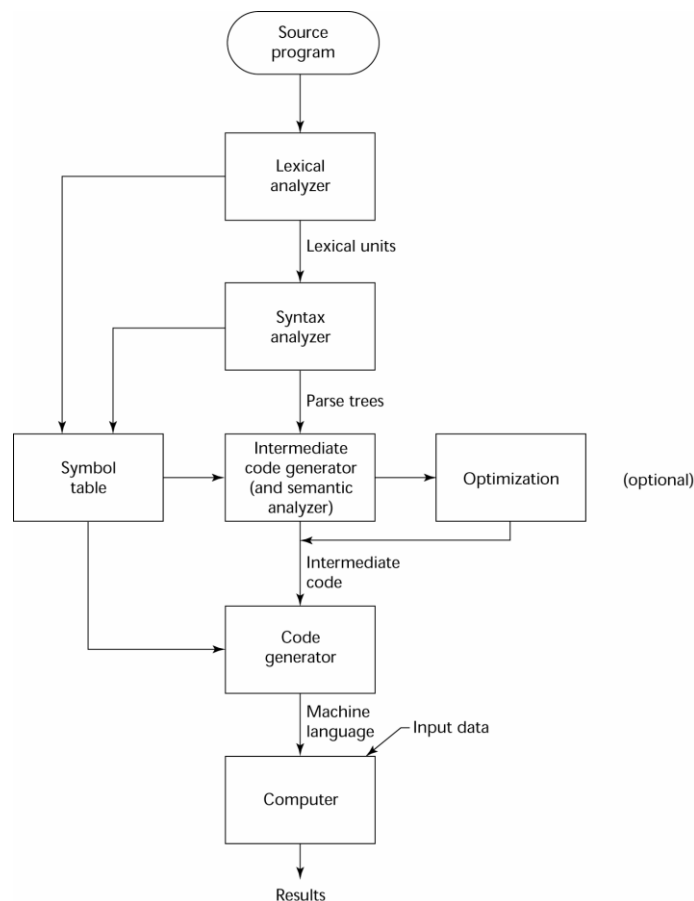


Figure 1.3 The Compilation Process

- Von Neumann Bottleneck
 - The speed of the connection between a computer's memory and its processor determines the speed of a computer.
 - Program instructions often can be executed a lot **faster** than they can be moved to the processor for execution.
 - This **connection speed** is called the von Neumann bottleneck; it is the primary limiting factor in the speed of computers

1.7.2 Pure Interpretation

- Programs are interpreted by another program called an interpreter, with **no translation**.
- Advantage: Easier implementation of many source-level debugging operations, because all **run-time** error messages can refer to **source-level** units.
- Disadvantage: Slower execution (**10 to 100 times** slower than compiled programs)
- **Bottleneck: Statement decoding**, rather than the connection between the processor and memory, is the bottleneck of a pure interpreter.
- Significant comeback with some Web scripting languages (e.g., **JavaScript** and **PHP**).

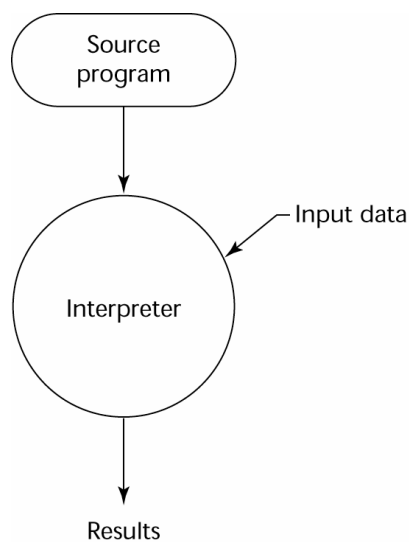


Figure 1.4 Pure Interpretation Process

1.7.3 Hybrid Implementation Systems

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

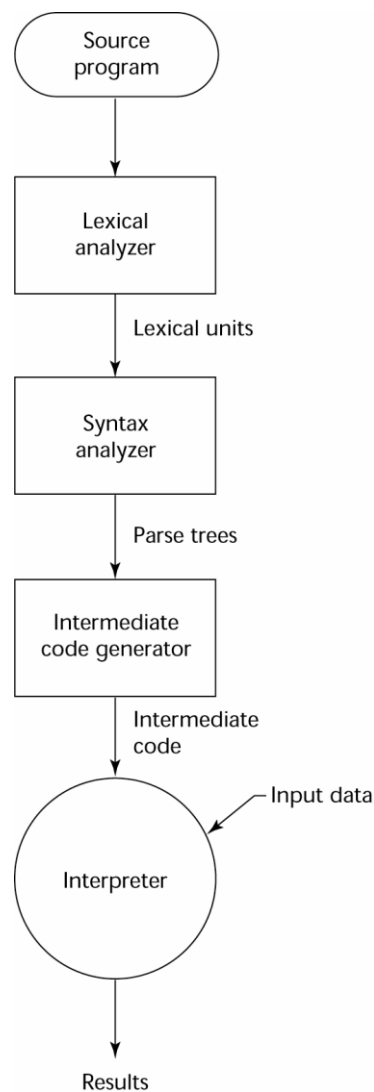


Figure 1.5 Hybrid Implementation System

1.8 Programming Environments 29

- The collection of tools used in software development
- UNIX
 - An older operating system and tool collection
 - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
- Borland JBuilder
 - An integrated development environment for Java
- Microsoft Visual Studio.NET
 - A large, complex visual environment
 - Used to program in C#, Visual Basic.NET, JScript (Microsoft's version of JavaScript), F# (a functional language), and C++
- NetBeans
 - Primarily used for Java application but also supports JavaScript, Ruby, and PHP

Summary

- The study of programming language is valuable for a number of important reasons:
 - Increases our capacity to use different constructs in **writing programs**
 - Enables us to **choose** languages for projects more intelligently
 - Makes learning **new** languages easier
- Among the most important criteria for **evaluating languages** are:
 - **Readability**
 - **Writability**
 - **Reliability**
 - **Overall cost**
- The major methods of implementing program languages are
 - **Compilation**
 - **Pure interpretation**
 - **Hybrid implementation**