MODULE 1

1.1 Reasons for Studying Concepts of Programming Languages

- **Potential benefits of studying concepts of programming languages**:
- ***Increased capacity to express ideas*** *:* It is widely believed that the depth at which people can think is influenced by the expressive power of the language in which they communicate their thoughts. Those with only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction. In other words, it is difficult for people to conceptualize structures they cannot describe, verbally or in writing. 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.
- ***Improved background for choosing appropriate languages*** *:* Many professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through inhouse training programs. Such training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization. Many other programmers received their formal training years ago. The languages they learned then are no longer used, and many features now available in programming languages were not widely known at the time. The result is that many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand. 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.

- ***Increased ability to learn new languages :*** Computer programming is still a relatively young discipline, and design methodologies, software development tools, and programming languages are still in a state of continuous evolution.This makes software development an exciting profession, but it also means that continuous learning is essential. The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general. Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

every software developer must be prepared to learn different languages.Finally, it is essential that practicing programmers know the vocabulary and fundamental concepts of programming languages so they can read and understand programming language descriptions and evaluations, as well as promotional literature for languages and compilers.

- ***Better understanding of the significance of implementation*** *:* In learning the concepts of programming languages, it is 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. 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.

- ***Better use of languages that are already known*** : Many 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.
- ***Overall advancement of computing*** : 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, 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 and even more difficult to understand.

1.2 Programming Domains

- Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones.
- ***Scientific Applications :*** The first digital computers, in the late 1940s and early 1950s, were invented and used for scientific applications. Typically, the scientific applications of that time used relatively simple data structures, but required large numbers of floating-point arithmetic computations. The most common data structures were arrays and matrices; the most common control structures were counting loops and selections. The early high-level programming languages invented for scientific applications were designed to provide for those needs. The first language for scientific applications was Fortran. ALGOL 60 and most of its descendants were also intended to be used in this area, although they were designed to be used in related areas as well. For some scientific applications where efficiency is the primary concern.
- ***Business Applications :*** The use of computers for business applications began in the 1950s. Special computers were developed for this purpose, along with special languages. The first successful high-level 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.

- ***Artificial Intelligence :*** Artificial intelligence (AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations. Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. Also, symbolic computation is more conveniently done with linked lists of data rather than arrays. This kind of programming sometimes requires more flexibility than other programming domains. For example, in some AI applications the ability to create and execute code segments during execution is convenient. The first widely used programming language developed for AI applications was the functional language LISP,Prolog.
- ***Systems Programming :*** The operating system and the programming support tools of a computer system are collectively known as its **systems software**. Systems software is used almost continuously and so it must be efficient. In the 1960s and 1970s, some computer manufacturers, such as IBM, Digital, and Burroughs (now UNISYS), developed special machine-oriented high-level languages for systems software on their machines. For IBM mainframe computers, the language was PL/S. The UNIX operating system is written almost entirely in 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.The dynamic web content 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.

1.3 Language Evaluation Criteria

- The language evaluation criteria are:
- Readability
- Writability
- Reliability
- Cost

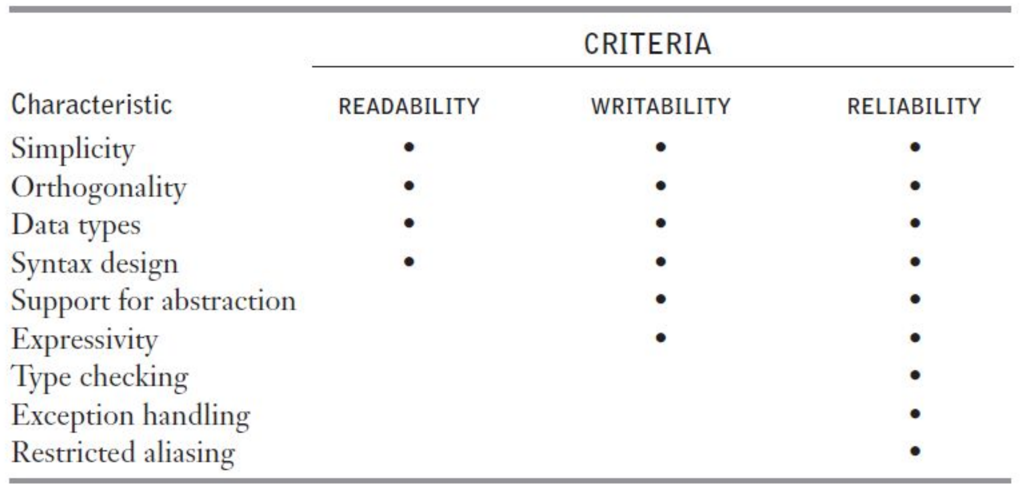


Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

1.3.1 Readability

- One of the most important criteria for judging a programming language is the ease with which programs can be read and understood . The primary positive characteristic of programming languages was efficiency.
- In software life-cycle concept coding has smaller role, and maintenance was recognized as a major part of the cycle, particularly in terms of cost. 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.
- Readability must be considered in the context of the problem domain. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural and convoluted, making it unusually difficult to read.

Overall Simplicity

- The overall simplicity of a programming language strongly affects its readability.
- A language with a large number of basic constructs is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features.
- Readability problems occur whenever the program’s author has learned a different subset from that subset with which the reader is familiar.
- A second complicating characteristic of a programming language is **feature multiplicity**—that is, 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
```

- A third potential problem is **operator overloading**, in which a single operator symbol has more than one meaning.
- For example, it is clearly acceptable to overload + to use it for both integer and floating-point addition. In fact, this overloading simplifies a language by reducing the number of operators
- Simplicity in languages can, be carried too far. For example, the form and meaning of 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, program structure is less obvious; because the statements are simple, far more of them are required than in equivalent programs in a high-level language.

Orthogonality

- **Orthogonality** in a programming language means that 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.
- Suppose a language has four primitive data types (integer, float,double, and character) and two type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types,a large number of data structures can be defined.
- A lack of orthogonality leads to exceptions to the rules of the language. For example, in a programming language that supports pointers, it should be possible to define a pointer to point to any specific type defined in the language. However, if pointers are not allowed to point to arrays, many potentially useful user-defined data structures cannot be defined.

- Consider adding two 32-bit integer values that reside in either memory or registers and replacing one of the two values with the sum. The IBM mainframes have two instructions for this purpose, which have the forms
    - `A Reg1, memory_cell`
    - `AR Reg1, Reg2`
- where `Reg1` and `Reg2` represent registers. The semantics of these are
- `Reg1 ← contents(Reg1) + contents(memory_cell)`
- `Reg1 ← contents(Reg1) + contents(Reg2)`
- The VAX addition instruction for 32-bit integer values is
    - `ADDL operand_1, operand_2`
- whose semantics is
- `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 cells as the operands.
- Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require.
- Too much orthogonality can also cause problems. The most orthogonal programming language is ALGOL 68.

<u>Data Types</u>

- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability. For 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 the following:
    - `timeOut = 1`
- The meaning of this statement 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.
- **<u>Syntax Design</u>**
- The syntax, or form, of the elements of a language has a significant effect on the readability of programs.
- ***<u>Special words</u>***. Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, **`while`**, **`class`**, and **`for`**). Especially important is the method of forming compound statements, or statement groups, primarily in control constructs. Some languages have used matching pairs of special words or symbols to form groups.C and its descendants use braces to specify compound statements. Fortran 95 and Ada make this clearer by using a distinct closing syntax for each type of statement group. For example, Ada uses **`end if`** to terminate a selection construct and **`end loop`** to terminate a loop construct.
    - Another important issue is whether the special words of a language can be used as names for program variables. If so, the resulting programs can be very confusing. For example, in Fortran 95, special words, such as `Do` and `End`, are legal variable names.
- ***<u>Form and meaning</u>***. Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability.

1.3.2 Writability

- Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability.
- For example, the writabilities of Visual BASIC (VB) and C are dramatically different for creating a program that has a graphical user interface, for which VB is ideal.

Simplicity and Orthogonality

- If a language has a large number of different constructs, some programmers might not be familiar with all of them. This situation can lead to a misuse of some features and a disuse of others .
- On the other hand, too much orthogonality can be a detriment to writability. Errors in programs can go undetected when nearly any combination of primitives is legal.

Support for Abstraction

- A**bstraction** means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
- Abstraction is a key concept in contemporary programming language design.This is a reflection of the central role that abstraction plays in modern program design methodologies.
- A simple example of process abstraction is the use of a subprogram to implement a sort algorithm that is required several times in a program. Without the subprogram, the sort code would need to be replicated in all places where it was needed, which would make the program much longer and more tedious to write
- The overall support for abstraction is clearly an important factor in the writability of a language

Expressivity

- Expressivity in a language can refer to several different characteristics. In a language such as APL , it means that there are very powerful operators that allow a great deal of computation to be accomplished.
- For example, in C, the notation `count++` is more convenient and shorter than `count = count + 1`. Also, the **`and then`** Boolean operator in Ada is a convenient way of specifying short-circuit evaluation of a Boolean expression.
- The inclusion of the **`for`** statement in Java makes writing counting loops easier than with the use of **`while.`**

1.3.3 Reliability

- A program is said to be reliable if it performs to its specifications under all conditions.

Type Checking

- **Type checking** 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.
- Because run-time type checking is expensive,compile-time type checking is more desirable. Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs.
- The design of Java requires checks of the types of nearly all variables and expressions at compile time. This virtually eliminates type errors at run time in Java programs.

- One example of how failure to type check, at either compile time or run time, has led to countless program errors is the use of subprogram parameters in the original C language. In this language, the type of an actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function. An **`int`** type variable could be used as an actual parameter in a call to a function that expected a **`float`** type as its formal parameter, and neither the compiler nor the run-time system would detect the inconsistency.
- The current version of C has eliminated this problem by requiring all parameters to be type checked.

Exception Handling

- The ability of a program to intercept run-time errors (as well as other unusual conditions detectable by the program), take corrective measures, and then continue is an obvious aid to reliability. This language facility is called **exception handling**.
- Ada, C++, Java, and C# include capabilities for exception handling.

Aliasing

- A**liasing** is having two or more distinct names that can be used to access the same memory cell. It is now widely accepted that aliasing is a dangerous feature in a programming language.
- Most programming languages allow some kind of aliasing—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 the two changes the value referenced by the other.

Readability and Writability

- Both readability and writability influence reliability.
- Readability affects reliability in both the writing and maintenance phases of the life cycle. Programs that are difficult to read are difficult both to write and to modify.

1.3.4 Cost

- The total cost of a programming language is a function of many of its characteristics
- First, there is **the cost of training programmers** to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers.
- Second, there is the **cost of writing programs** in the language. This is a function of the writability of the language, which depends in part on its closeness in purpose to the particular application.
- Third, there is **the cost of compiling programs** in the language. A major impediment to the early use of Ada was the prohibitively high cost of running the first-generation Ada compilers. This problem was diminished by the appearance of improved Ada compilers.
- Fourth, the **cost of executing programs** written in a language is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler.
- The fifth factor is **the cost of the language implementation system**. One of the factors that explains the rapid acceptance of Java is that free compiler/interpreter systems became available for it soon after its design was released.
- Sixth, there is **the cost of poor reliability**. If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high. The failures of noncritical systems can also be very expensive in terms of lost future business or lawsuits over defective software systems.
- The final consideration is the **cost of maintaining programs**, which includes both corrections and modifications to add new functionality. The cost of software maintenance depends on a number of language characteristics, primarily readability.

1.4 Influences on Language Design
1.5 Language Design Trade-Offs

- ASSIGNMENT portion

1.6 **Implementation Methods**

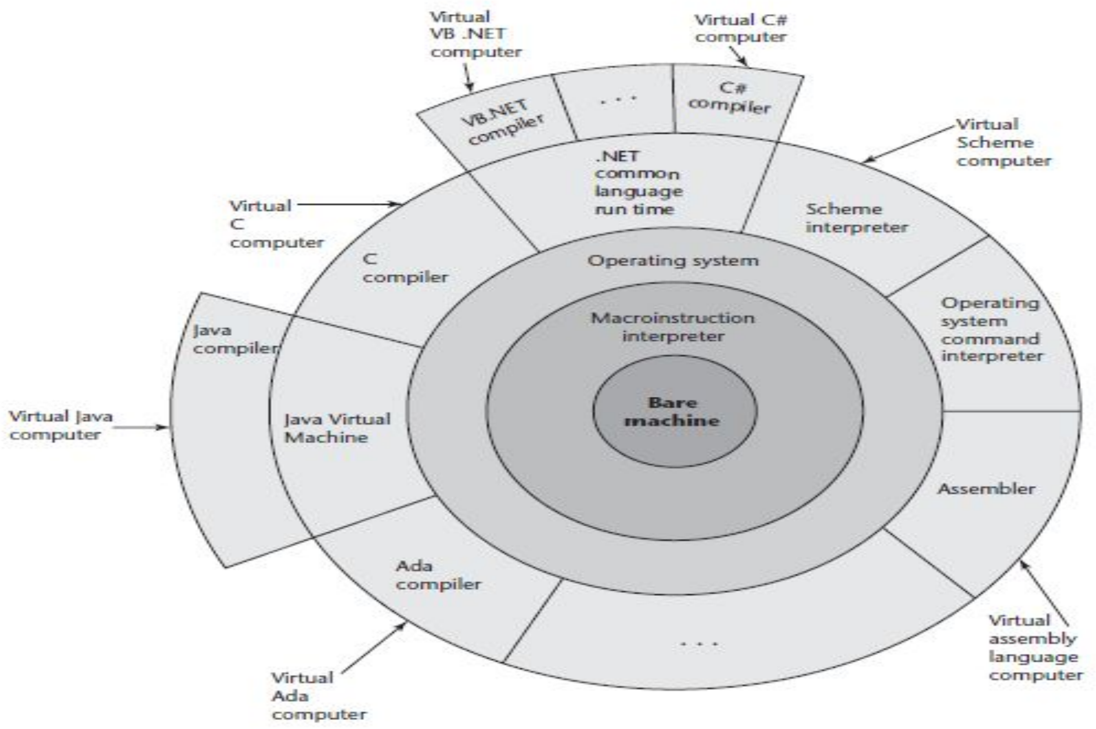
- two of the primary components of a computer are its internal memory and its processor. The internal memory is used to store programs and data. The processor is a collection of circuits that provides a realization of a set of primitive operations, or machine instructions, such as those for arithmetic and logic operations.
- In most computers, some of these instructions, which are sometimes called macroinstructions, are actually implemented with a set of instructions called microinstructions, which are defined at an even lower level.
- The machine language of the computer is its set of instructions. In the absence of other supporting software, its own machine language is the only language that most hardware computers "understand." Theoretically, a computer could be designed and built with a particular high-level language as its machine language, but it would be very complex and expensive.
- It would be highly inflexible, because it would be difficult (but not impossible) to use it with other high-level languages.

Compilation

- Programming languages can be implemented by any of three general methods.
- At one extreme, programs can be translated into machine language, which can be executed directly on the computer. This method is called a **compiler implementation** and has the advantage of very fast program execution, once
- the translation process is complete. Most production implementations of languages, such as C, COBOL, C++, and Ada, are by compilers.
- The language that a compiler translates is called the **source language.**

Figure 1.2

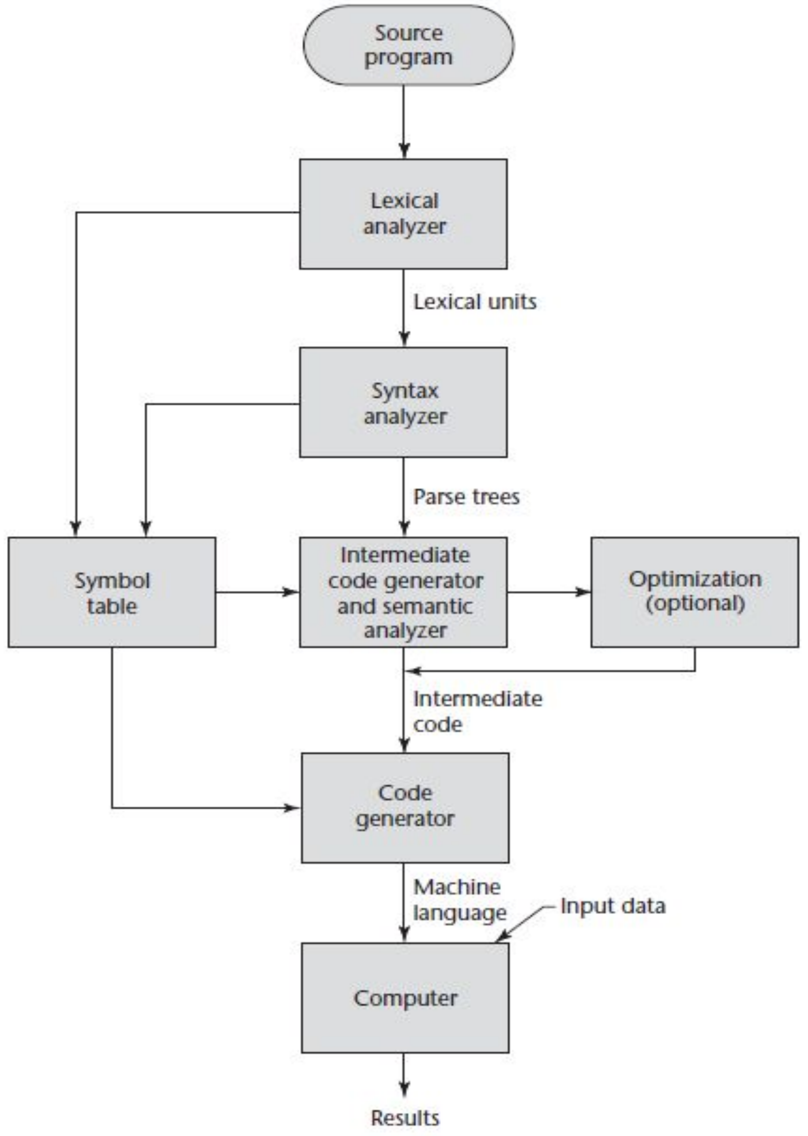
Layered interface of virtual computers, provided by a typical computer system



- The lexical analyzer gathers the characters of the source program into lexical units. The lexical units of a program are identifiers, special words, operators, and punctuation symbols.
- The syntax analyzer takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called **parse trees**. These parse trees represent the syntactic structure of the program.
- The intermediate code generator produces a program in a different language,at an intermediate level between the source program and the final output of the compiler: the machine language program.4 Intermediate languages sometimes look very much like assembly languages.
- The semantic analyzer is an integral part of the intermediate code generator. The semantic analyzer checks for errors, such as type errors, that are difficult, if not impossible, to detect during syntax analysis.
- Optimization, which improves programs by making them smaller or faster or both, is often an optional part of compilation.
- The code generator translates the optimized intermediate code version of the program into an equivalent machine language program.
- The symbol table serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program.
- The user and system code together are sometimes called a **load module**,or **executable image**. The process of collecting system programs and linking them to user programs is called **linking and loading**, or sometimes just **linking.**It is accomplished by a systems program called a **linker**

Figure 1.3

The compilation process



Pure Interpretation

- Pure interpretation lies at the opposite end (from compilation) of implementation methods. With this approach, programs are interpreted by another program called an interpreter, with no translation whatever. The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions.
- This software simulation obviously provides a virtual machine for the language.
- Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units. For example, if an array index is found to be out of range, the error message can easily indicate the source line and the name of the array.
- The disadvantage is that execution is 10 to 100 times slower than in compiled systems.
- Another disadvantage of pure interpretation is that it often requires more space. In addition to the source program, the symbol table must be present during interpretation.
- the approach was rarely used on high-level languages. However, in recent years, pure interpretation was used in some Web scripting languages, such as JavaScript and PHP.

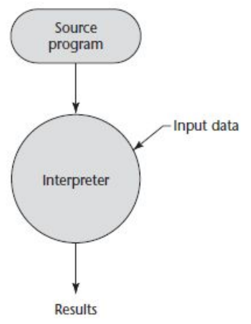


Figure 1.4
Pure interpretation

Hybrid Implementation Systems

- Some language implementation systems are a compromise between compilers and pure interpreters; they translate high-level language programs to an intermediate language designed to allow easy interpretation.
- This method is faster than pure interpretation because the source language statements are decoded only once. Such implementations are called **hybrid implementation systems.**
- Perl is implemented with a hybrid system. Perl programs are partially cor

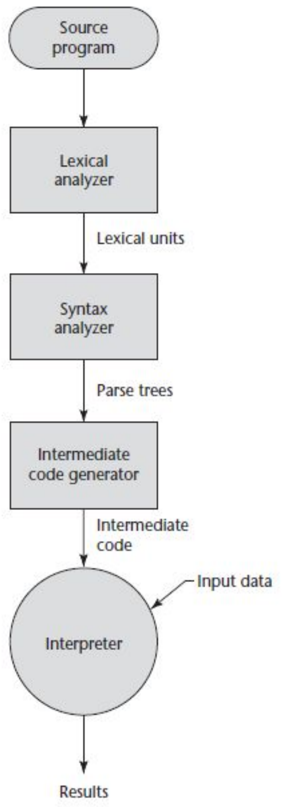
  to detect errors before interpretation and to simplify the interpreter.

  Initial implementations of Java were all hybrid. Its intermediate form,

  called **byte code**,

Figure 1.5

Hybrid implementation system



Preprocessors

- A **preprocessor** is a program that processes a program immediately before the program is compiled. Preprocessor instructions are embedded in programs.
- The preprocessor is essentially a macro expander. Preprocessor instructions are commonly used to specify that the code from another file is to be included.
- For example, the C preprocessor instruction

```
#include "myLib.h"
```

- causes the preprocessor to copy the contents of `myLib.h` into the program at the position of the `#include`.
- Other preprocessor instructions are used to define symbols to represent expressions. For example, one could use

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

- to determine the largest of two given expressions. For example, the expression

```
x = max(2 * y, z / 1.73);
```

- would be expanded by the preprocessor to

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73);
```

Names

- Names are also associated with subprograms, formal parameters, and other program constructs. The term *identifier* is often used interchangeably with *name*.

Design Issues

- The following are the primary design issues for names:
- Are names case sensitive?
- Are the special words of the language reserved words or keywords?

Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- Fortran 95+ allows up to 31 characters in its names. C99 has no length limitation on its internal names, but only the first 63 are significant. Names in Java, C#, and Ada have no length limit, and all characters in them are significant. C++ does not specify a length limit on names.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters (_).
- In the C-based languages, it has to a large extent been replaced by the so-called *camel notation*, in which all of the words of a multiple-word name except the first are capitalized, as in `myStack`

- All variable names in PHP must begin with a dollar sign. In Perl, the special character at the beginning of a variable's name, `$`, `@`, or `%`, specifies its type.
- In Ruby, special characters at the beginning of a variable's name, `@` or `@@`, indicate that the variable is an instance or a class variable, respectively.
- In many languages, notably the C-based languages, uppercase and lowercase letters in names are distinct; that is, names in these languages are **case sensitive**.For example, the following three names are distinct in C++: `rose`, `ROSE`, and `Rose`.

Special Words

- Special words in programming languages are used to make programs more readable by naming actions to be performed. They also are used to separate the syntactic parts of statements and programs.
- In most languages, special words are classified as reserved words, which means they cannot be redefined by programmers, but in some they are only keywords, which means they can be redefined.
- A **keyword** is a word of a programming language that is special only in certain contexts. Fortran is the only remaining widely used language whose special words are keywords. In Fortran, the word `Integer`, when found at the beginning of a statement and followed by a name, is considered a keyword that indicates the statement is a declarative statement. However, if the word `Integer` is followed by the assignment operator, it is considered a variable name.

```
Integer Apple

Integer = 4
```

- A **reserved word** is a special word of a programming language that cannot be used as a name

Variables

- A program variable is an abstraction of a computer memory cell or collection of cells. Programmers often think of variable names as names for memory locations, but there is much more to a variable than just a name.
- <u>A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, and scope)</u>.

<u>Name</u>

- Variable names are the most common names in programs.

<u>Address</u>

- The **address** of a variable is the machine memory address with which it is associated. This association is not as simple as it may at first appear. In many languages, it is possible for the same variable to be associated with different addresses at different times in the program. For example, if a subprogram has a local variable that is allocated from the run-time stack when the subprogram is called, different calls may result in that variable having different addresses.
- The address of a variable is sometimes called its **l-value**, because the address is what is required when the name of a variable appears in the left side of an assignment.
- It is possible to have multiple variables that have the same address. When more than one variable name can be used to access the same memory location, the variables are called **aliases.**
- Aliasing is a hindrance to readability because it allows a variable to have its value changed by an assignment to a different variable.
- For example, if variables named `total` and `sum` are aliases, any change to the value of `total` also changes the value of `sum` and vice versa.

Type

- The **type** of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type. For example, the `int` type in Java specifies a value range of -2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Value

- The **value** of a variable is the contents of the memory cell or cells associated with the variable.
- A variable’s value is sometimes called its **r-value** because it is what is required when the name of the variable appears in the right side of an assignment statement.

The Concept of Binding

- A **binding** is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol.
- The time at which a binding takes place is called **binding time.**
- Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time.
- **Lang Design Time:**Control flow constructs,set of fundamental types
- **Lang Implementation time**: Precision of fundamental types,max.size of stack and heap,handling of run time exception
- **Pgm Writing Time**: Programmers choose alg,data structures and names
- **Compile time**:mapping of high level construct to m/c code
- **Link time**
- **Load time**
- **Run time**

- Consider the following Java assignment statement:

```
count = count + 5;
```

- Some of the bindings and their binding times for the parts of this assignment statement are as follows:
- The type of `count` is bound at compile time.
- The set of possible values of `count` is bound at compiler design time.
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal `5` is bound at compiler design time.
- The value of `count` is bound at execution time with this statement.

Binding of Attributes to Variables

- A binding is **static** if it first occurs before run time begins and remains unchanged throughout program execution.
- If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic**.
- These bindings, however, are maintained by computer hardware, and the changes are invisible to the program and the user

- **Type Bindings**
- Before a variable can be referenced in a program, it must be bound to a data type.2 types

    1.Static type binding

    2.Dynamic type binding

    Static Type Binding

- An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type.
- An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements.
- In this case, the first appearance of a variable name in a program constitutes its implicit declaration. Both explicit and implicit declarations create static bindings to types.
- Implicit variable type binding is done by the language processor, either a compiler or an interpreter.
- The simplest of these is naming conventions.
- For example, in Fortran, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention: If the identifier begins with one of the letters `I`, `J`, `K`, `L`, `M`, or `N`, or their lowercase versions, it is implicitly declared to be `Integer` type; otherwise, it is implicitly declared to be `Real` type.

- implicit declarations can be detrimental to reliability because they prevent the compilation process from detecting some typographical and programmer errors.
- Some of the problems with implicit declarations can be avoided by requiring names for specific types to begin with particular special characters.
- For example, in Perl any name that begins with `$` is a scalar, which can store either a string or a numeric value. If a name begins with `@`, it is an array; if it begins with a `%`, it is a hash structure.4 This creates different namespaces for different type variables. In this scenario, the names `@apple` and `%apple` are unrelated, because each is from a different namespace.
- Another kind of implicit type declarations uses context. This is sometimes called **type inference**. In the simpler case, the context is the type of the value assigned to the variable in a declaration statement. For example, in C# a **`var`** declaration of a variable must include an initial value, whose type is made the type of the variable. Consider the following declarations:

```
var sum = 0;

var total = 0.0;

var name = "Fred";
```

- The types of `sum`, `total`, and `name` are **`int`**, **`float`**, and **`string`**, respectively.

Dynamic Type Binding

- With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name.
- Instead,the variable is bound to a type when it is assigned a value in an assignment statement.
- When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment.
- When the type of a variable is statically bound, the name of the variable can be thought of being bound to a type, in the sense that the type and name of a variable are simultaneously bound.
- When a variable’s type is dynamically bound, its name can be thought of as being only temporarily bound to a type. In reality, the names of variables are never bound to types. Names can be bound to variables and variables can be bound to types.
- The primary advantage of dynamic binding of variables to types is that it provides more programming flexibility. For example, a program to process numeric data in a language that uses dynamic type binding can be written as a generic program, meaning that it is capable of dealing with data of any numeric type.
- In Python, Ruby, JavaScript, and PHP, type binding is dynamic. For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

- Regardless of the previous type of the variable named `list`, this assignment causes it to become the name of a single-dimensioned array of length 2. If the statement

```
list = 47;
```

- followed the previous example assignment, `list` would become the name of a scalar variable.
- In c# variable can be declared to use dynamic type binding by including the **`dynamic`** reserved word in its declaration, as in the following example:

```
dynamic any;
```

- There are two disadvantages to dynamic type binding. First, it causes programs to **be less reliable**, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings.
- the greatest disadvantage of dynamic **type binding is cost**. The cost of implementing dynamic attribute binding is considerable, particularly in execution time.

Storage Bindings and Lifetime

- 3 storage allocation mechanism
- 1. Static objects-given an absolute address ie retained through out the pgm execution
- 2. Stack objects-allocated and deallocated in LIFO order
- 3. Heap objects-allocated and deallocated at arbitrary times

Scope

SCOPE RULES

- The **scope** of a variable is the range of statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced in that statement.

- The textual region of the program in which a binding is active is its *scope*.
- In most modern languages, the scope of a binding is determined statically, that is, at compile time.
- In C, for example, a new scope is introduced upon entry to a subroutine.We create bindings for local objects and deactivate bindings for global objects that are "hidden" by local objects of the same name.
- On subroutine exit, destroy bindings for local variables and reactivate bindings for any global objects that were hidden. These manipulations of bindings may appear to be run-time operations, but they do not require the execution of any code: the portions of the program in which a binding is active are completely determined at compile time. For this reason, <u>C is said to be *statically scoped*</u>

- Other languages, including APL, Snobol, and early dialects of Lisp, are *dynamically scoped*: their bindings depend on the flow of execution at run time.
- At any given point in a program's execution, the set of active bindings is called the current *referencing environment*. The set is principally determined by static or dynamic *scope rules*.
- referencing environments also depend on what are called *binding rules*.
- 2 options:
- *deep binding*, in which the choice is made when the reference is first created, and
- *shallow binding*, in which the choice is made when the reference is finally used.

Static Scoping(lexical scoping)

- In a language with static (lexical) scoping, the bindings between names and objects can be determined at compile time.
- The simplest static scope rule is probably that of early versions of Basic, in which there was only a single, global scope.
- In Fortran distinguishes between global and local variables. The scope of a local variable is limited to the subroutine in which it appears; it is not visible elsewhere.
- Variable declarations are optional. If a variable is not declared,it is assumed to be local to the current subroutine and to be of type integer if its name begins with the letters I–N, or real otherwise
- Global variables in Fortran may be partitioned into common blocks, which are then “imported” by subroutines.

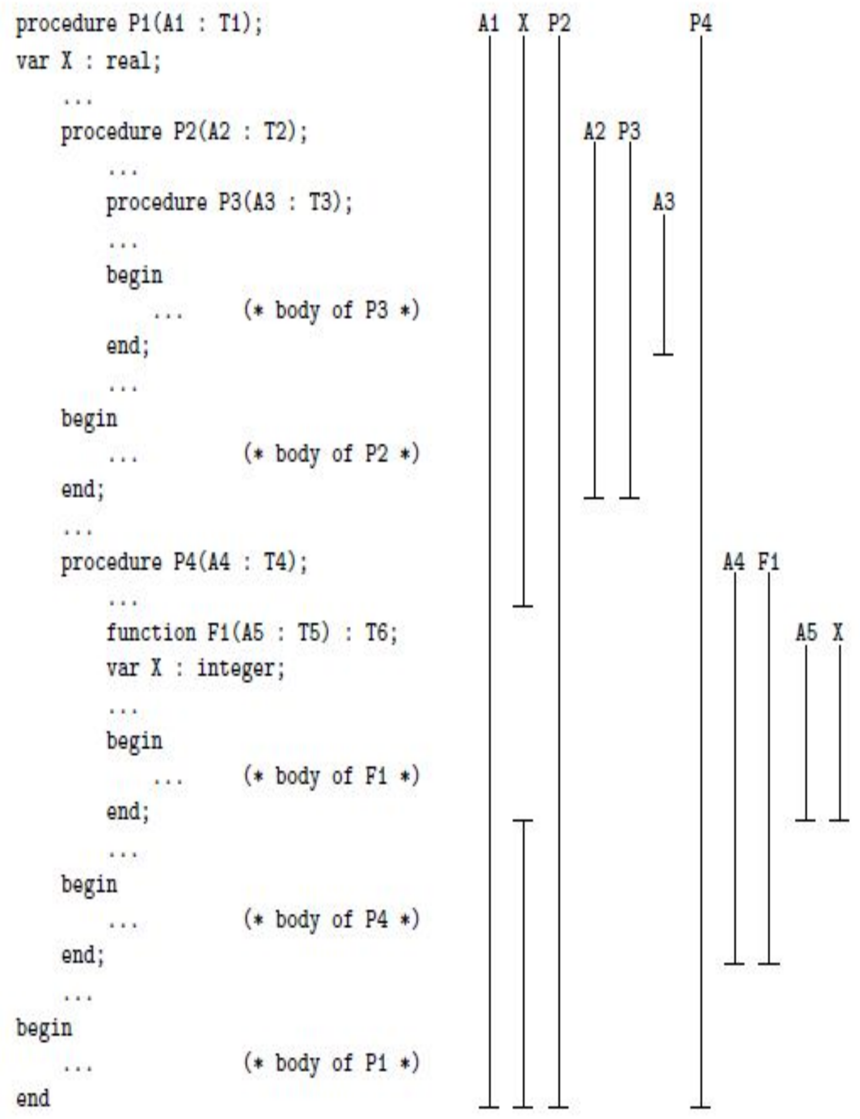
- Consider the following JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function big() {
function sub1() {
var x = 7;
sub2();
}
function sub2() {
var y = x;
}
var x = 3;
sub1();
}
```

- Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`. This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there. The search continues in the static parent of `sub2`, `big`, where the declaration of `x` is found. The `x` declared in `sub1` is ignored, because it is not in the static ancestry of `sub2`

- **Nested Subroutines**
- The ability to nest subroutines inside each other, introduced in Algol 60,
- a feature of many modern languages, including Pascal, Ada, ML, Python, Scheme, Common Lisp, including C and its descendants
- Algol-style nesting gives rise to the *closest nested scope rule* for bindings from names to objects: a name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is *hidden* by another declaration of the same name in one or more nested scopes

- To find the object corresponding to a given use of a name, we look for a declaration with that name in the current, innermost scope.If there is one, it defines the active binding for the name.
- Otherwise, we look for a declaration in the immediately surrounding scope. We continue outward, examining successively surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared.
- If no declaration is found at any level, then the program is in error.



```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...     (* body of P3 *)
        end;
        ...
    begin
        ...         (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...     (* body of F1 *)
        end;
        ...
    begin
        ...         (* body of P4 *)
    end;
    ...
begin
    ...             (* body of P1 *)
end
```

Figure 3.4 **Example of nested subroutines in Pascal.** Vertical bars show the scope of each name (note the hole in the scope of the outer X).

nested subroutines are
able to access the parameters and
local variables (and other local
objects) of the
surrounding scope(s). In our
example, P3 can name (and modify)
A1, X, and
A2, in addition to A3. Because P1 and
F1 both declare local variables
named X,
the inner declaration hides the outer
one within a portion of its scope.
Uses of
X in F1 refer to the inner X; uses of X
in other regions of the code refer to
the
outer X.

- A name-to-object binding that is hidden by a nested declaration of the same name is said to have a *hole* in its scope.
- In most languages the object whose name is hidden is inaccessible in the nested scope .
- Some languages allow the programmer to access the outer meaning of a name by applying a *qualifier* or *scope resolution operator*.
- In Ada, for example, My_proc.X, for example,refers to the declaration of X in subroutine My_proc,

Access to Nonlocal Objects

- The simplest way in which to find the frames of surrounding scopes is to maintain a *static link* in each frame that points to the "parent" frame: the frame of the most recent invocation of the lexically surrounding subroutine.
- If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time.
- If a subroutine is nested *k* levels deep,then it, will form a *static chain* of length *k* at run time.

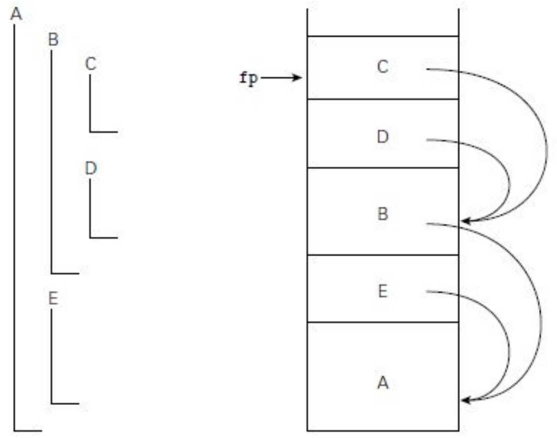
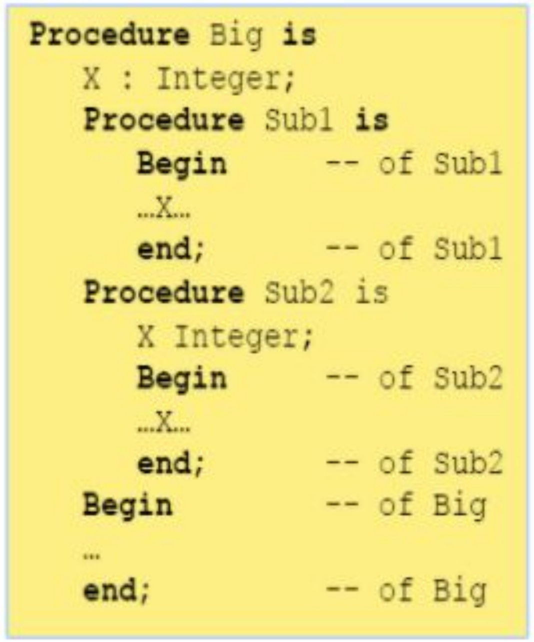


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by

Static scope rules

- Under static scoping, the reference to the var X in Sub1 is to the X declared in the procedure Big.
- This is true b/c the search for X begins in the procedure in which the reference occurs, Sub1, but no declaration for X is found there.
- The search thus continues in the **static parent of Sub1 is Big,** where the declaration of X is found.

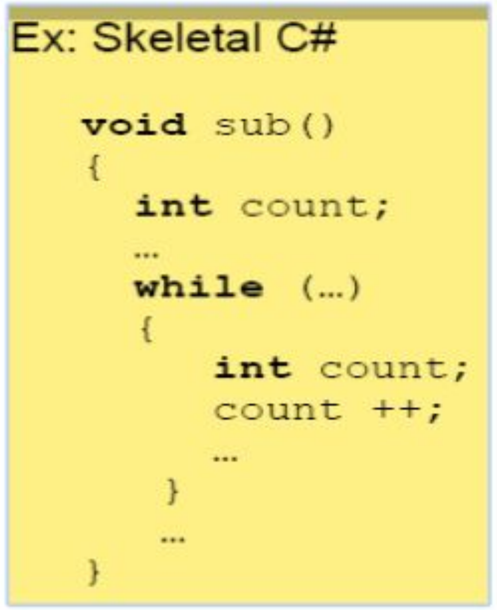


```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin          -- of Sub1
      ...X...
      end;           -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin          -- of Sub2
      ...X...
      end;           -- of Sub2
   Begin             -- of Big
   ...
   end;              -- of Big
```

Static scope rules

- The count of sub is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a var with the same name in a larger enclosing scope.
- C++ and Ada allow access to these "hidden" variables
- - In Ada: Main.X
- -In C++: class_name::name.
- The reference to count in the while

  loop is to that loop's local count.


Ex: Skeletal C#

```
void sub()
{
  int count;
  …
  while (…)
  {
      int count;
      count ++;
      …
    }
    …
}
```

• BLOCKS

- In Scheme, a `let` construct is a call to the function `LEA` with the following form:

```
(LET (
(name1 expression1)
. . .
(namen expressionn))
expression
)
```

- The semantics of the call to `LET` is as follows: The first *n* expressions are evaluated and the values are assigned to the associated names. Then, the final expression is evaluated and the return value of `LET` is that value.
- Consider the following call to `LET`:

```
(LET (
(top (+ a b))
(bottom (- c d)))
(/ top bottom)
)
```

- This call computes and returns the value of the expression `(a + b) / (c – d)`.
- In ML, the form of a **`let`** construct is as follows:

```
let
val name1 = expression1
...
val namen = expressionn
in
expression
end;
```

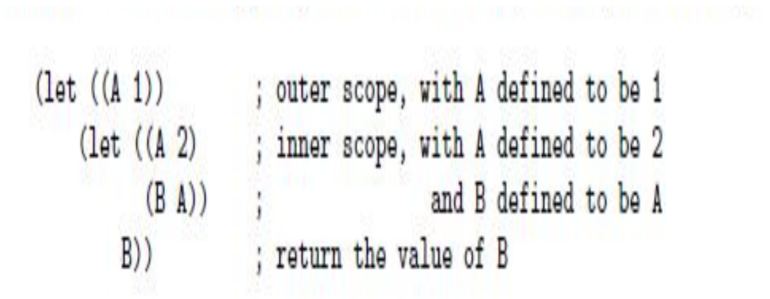
- Consider the following code:

```
let n1 =
let n2 = 7
let n3 = n2 + 3
n3;;
let n4 = n3 + n1;;
```

- The scope of `n1` extends over all of the code. However, the scope of `n2` and `n3` ends when the indentation ends. So, the use of `n3` in the last **`let`** causes an error. The last line of the **`let`** `n1` scope is the value bound to `n1`; it could be any expression.

Declaration order

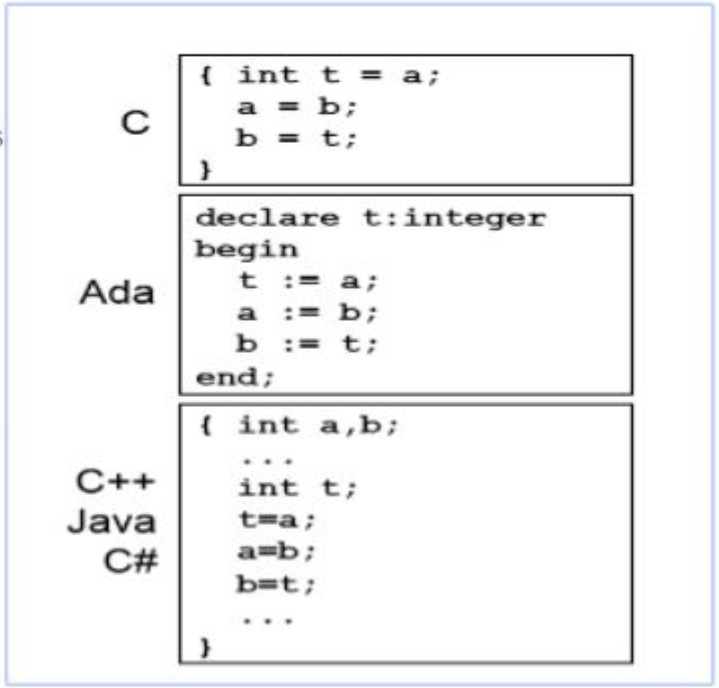
In Scheme,Here the nested declarations of A and B don’t take effect until after the end of the declaration list. Thus when B is defined, the redefinition of A has not yet taken effect. B is defined to be the *outer* A, and the code as a whole returns 1.



```
(let ((A 1))        ; outer scope, with A defined to be 1
   (let ((A 2)      ; inner scope, with A defined to be 2
         (B A))     ;              and B defined to be A
      B))           ; return the value of B
```

NESTED BLOCK

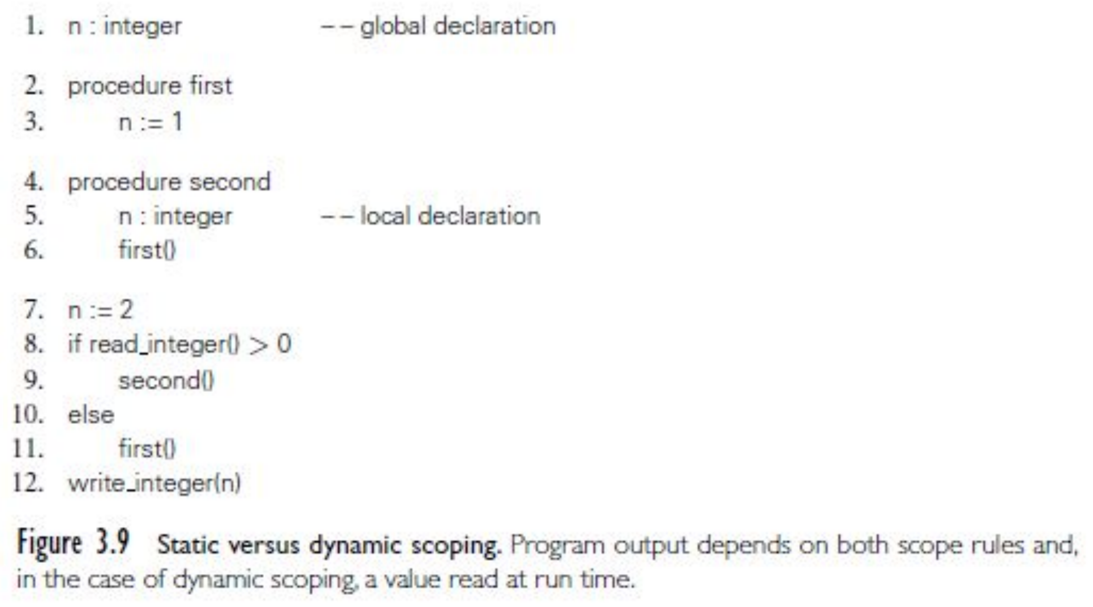
- In several languages local variables are declared in a block or compound statement
- – At the beginning of the block (Pascal, ADA, …)
  – Anywhere (C/C++, Java, …)
- • Local variables declared in nested blocks in a single function are all stored in the subroutine frame for that function (most programming languages, e.g. C/C++, Ada, Java


C
Ada
C++
Java
C#

```
{ int t = a;
  a = b;
  b = t;
}
```

```
declare t:integer
begin
  t := a;
  a := b;
  b := t;
end;
```

```
{ int a,b;
  ...
  int t;
  t=a;
  a=b;
  b=t;
  ...
}
```



```
 1.  n : integer             -- global declaration

 2.  procedure first
 3.      n := 1

 4.  procedure second
 5.      n : integer         -- local declaration
 6.      first()

 7.  n := 2
 8.  if read_integer() > 0
 9.      second()
10.  else
11.      first()
12.  write_integer(n)
```

Figure 3.9 **Static versus dynamic scoping.** Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.

If static scoping is in effect, this program prints 1. If dynamic scoping is in effect, the output depends on the value read at line 8 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1.

```
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
int x = 20;
return f();
}

int main()
{
printf("%d", g());
printf("\n");
return 0;
}
```

```
// Since dynamic scoping is very uncommon in // the familiar languages, we consider the

// following pseudo code as our example. It// prints 20 in a language that uses dynamic //
scoping.

int x = 10;

// Called by g()

int f()

{

return x;

}

// g() has its own variable

// named as x and calls f()

int g()

{

int x = 20;

return f();

}

main()

{

printf(g());

}
```

Global Scope

- C and C++ have both declarations and definitions of global data. Declarations specify types and other attributes but do not cause allocation of storage. Definitions specify attributes *and* cause storage allocation.
- A global variable in C is implicitly visible in all subsequent functions in the file, except those that include a declaration of a local variable with the same name. A global variable that is defined after a function can be made visible in the function by declaring it to be external, as in the following:

```
extern int sum;
```

- In PHP Global variables can be made visible in functions in their scope in two ways: (1) If the function includes a local variable with the same name as a global, that global can be accessed through the $GLOBALS array, using the name of the global as a string literal subscript, and (2) if there is no local variable in the function with the same name as the global, the global can be made visible by including it in a global declaration statement. Consider the following example:

```
$day = "Monday";
$month = "January";
function calendar() {
$day = "Tuesday";
global $month;
print "local day is $day <br />";
$gday = $GLOBALS['day’];
print "global day is $gday <br \>";
print "global month is $month <br />";
}
calendar();
```

- Interpretation of this code produces the following:
- local day is Tuesday
- global day is Monday
- global month is January

- In Python global variable can be assigned in a function only if it has been declared to be global in the function. Consider the following examples:

```
day = "Monday"
def tester():
print "The global day is:", day
tester()
```

- The output of this script, because globals can be referenced directly in functions,is as follows:

```
The global day is: Monday
```

- The following script attempts to assign a new value to the global `day`:

```
day = "Monday"
def tester():
print "The global day is:", day
day = "Tuesday"
print "The new value of day is:", day
tester()
```

- This script creates an `UnboundLocalError` error message, because the assignment to `day` in the second line of the body of the function makes `day` a local variable.

```
Consider,

day = "Monday"
def tester():
global day
print "The global day is:", day
day = "Tuesday"
print "The new value of day is:", day
tester()
```

- The output of this script is as follows:

```
The global day is: Monday
The new value of day is: Tuesday
```

Dynamic Scoping

- with dynamic scoping, the bindings between names and objects depend on the flow of control at run time
- Languages with dynamic scoping include APL, Snobol, TEX, and early dialects of Lisp.
- Because the flow of control cannot in general be predicted in advance, the bindings between names and objects in a language with dynamic scoping cannot in general be determined by a compiler.

Dynamic Scoping

- with dynamic scoping, the bindings between names and objects depend on the flow of control at run time
- Languages with dynamic scoping include APL, Snobol, TEX, and early dialects of Lisp.
- Because the flow of control cannot in general be predicted in advance, the bindings between names and objects in a language with dynamic scoping cannot in general be determined by a compiler.

```
function big() {
function sub1() {
var x = 7;
}
function sub2() {
var y = x;
var z = 3;
}
var x = 3;
}
```

- Assume that dynamic-scoping rules apply to nonlocal references. The meaning of the identifier `x` referenced in `sub2` is dynamic—it cannot be determined at compile time. It may reference the variable from either declaration of `x`, depending on the calling sequence

- Consider the two different call sequences for `sub2` in the earlier example.First, `big` calls `sub1`, which calls `sub2`. In this case, the search proceeds from the local procedure, `sub2`, to its caller, `sub1`, where a declaration for `x` is found. So, the reference to `x` in `sub2` in this case is to the `x` declared in `sub1`.
- Next, `sub2` is called directly from `big`. In this case, the dynamic parent of `sub2` is `big`, and the reference is to the `x` declared in `big`
- if static scoping were used, in either calling sequence discussed, the reference to `x` in `sub2` would be to `big`’s `x`.

Scope and Lifetime

- For example, consider a variable that is declared in a Java method that contains no method calls.
- The scope of such a variable is from its declaration to the end of the method. The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.
- Scope and lifetime are also unrelated when subprogram calls are involved.
- Consider the following C++ functions:

```
void printheader() {
. . .
} /* end of printheader */
void compute() {
int sum;
. . .
printheader();
} /* end of compute */
```

- The scope of the variable `sum` is completely contained within the `compute` function
- the lifetime of `sum` extends over the time during which `printheader` executes.

Referencing Environments

- The **referencing environment** of a statement is the collection of all variables that are visible in the statement.
- The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible.
- Python skeletal program:

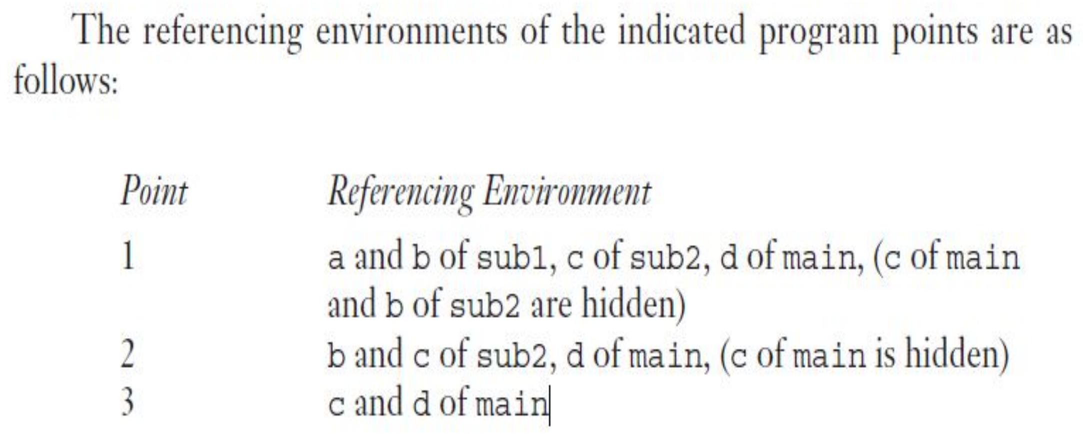
```
g = 3; # A global
def sub1():
a = 5; # Creates a local
b = 7; # Creates another local
. . . 1
def sub2():
global g; # Global g is now assignable here
```

```
c = 9; # Creates a new local
. . . 2
def sub3():
nonlocal c: # Makes nonlocal c visible here
g = 11; # Creates a new local
. . . 3
```

- The referencing environments of the indicated program points are as follows:
- *Point* *Referencing Environment*
- 1 local `a` and `b` (of `sub1)`, global `g` for reference, but not for assignment
- 2 local `c` (of `sub2)`, global `g` for both reference and for assignment
- 3 nonlocal `c` (of `sub2)`, local `g` (of `sub3)`
- A subprogram is **active** if its execution has begun but has not yet terminated.
- The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active

- Consider the following example program. Assume that the only function calls are the following: `main` calls `sub2`, which calls `sub1`.

```
void sub1() {
int a, b;
. . . 1
} /* end of sub1 */
void sub2() {
int b, c;
.. . . 2
sub1();
} /* end of sub2 */
void main() {
int c, d;
. . . 3
sub2();
} /* end of main */
```



The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main`, (`c` of `main` and `b` of `sub2` are hidden)
2	`b` and `c` of `sub2`, `d` of `main`, (`c` of `main` is hidden)
3	`c` and `d` of `main`