KTU NOTES

MODULE 1

1. Reasons for Studying Concepts of Programming Languages
2. Programming Domains
3. Language Evaluation Criteria
4. Influences on Language Design
5. Language Categories
6. Language Design Trade-Offs
7. Implementation Methods
8. Programming Environments

1.1 Reasons for Studying Concepts of Programming Languages

- **Increased capacity to express ideas.**

  Language influences and limits one's ability to express (and even formulate) ideas, because people tend to "think in a language". Many CS 1 students, for example, have difficulties because they don't yet know the programming language well enough to know what it can do.

  By knowing about the various abstraction mechanisms available in various languages (e.g., recursion, objects, associative arrays, functions as "first-class" entities, etc.), a programmer can more easily solve problems, even if programming in a language lacking an abstraction relevant to the solution.

- **Improved ability to choose an appropriate language.**

  "If all you know is how to use a hammer, every problem looks like a nail."

  All general-purpose programming languages are equivalent (i.e., Turing universal) in terms of capability, but, depending upon the application, one language may be better suited than another.

  Examples: COBOL was designed with business applications in mind, FORTRAN for scientific applications, C for systems programming, SNOBOL for string processing.

- **Increased ability to learn new languages.**

  Given how frequently new programming languages rise in popularity, this is an important skill.

- **Better understanding of implementation issues (i.e., how language constructs are implemented)**

  - Helps in figuring out subtle bugs (e.g., caused by a buffer overrun or aliasing) and in playing tricks to get you "closer" to the hardware in those instances where it is necessary.
  - Helps in tweaking a program to make more efficient. E.g., When M.G. said, "Recursion is bad.", he meant that, in some instances, it is better to use iteration because it can be much faster and use less memory. (Compare a subprogram that recursively sums the elements of an array to one that does it using a **for** or **while** loop.)
  - Affect upon language design (i.e., which constructs are included vs. excluded).

    E.g., FORTRAN was designed to be fast; IBM 704 had three address registers, so arrays were limited to be no more than 3-dimensional.

    E.g., Why are there separate types for integers and reals?

    E.g., Why have "associative arrays" become common as built-in constructs only in recently-introduced languages? Does it have to do with complexity of implementation?

- **Improved use of languages one already "knows"**:

  By learning about programming language constructs in general, you may come to understand (and thus begin making use of) features/constructs in your "favorite" language that you may have not used before.

- **Advancement of computing in general**.

  Here, Sebesta argues that, if programmers (in general) had greater knowledge of programming language concepts, the software industry would do a better job of adopting languages based upon their merits rather than upon political and other forces. (E.g., Algol 60 never made large inroads in the U.S., despite being superior to FORTRAN. Eiffel is not particularly popular, despite being a great language!)

1.2 Programming Domains

Computers have been used to solve problems in a wide variety of application areas, or domains. Many programming languages were designed with a particular domain in mind.

- **Scientific Applications:** Programs tend to involve use of arithmetic on real numbers, arrays/matrices, and "counting" loops. FORTRAN was designed (in late 1950's) for this kind of application, and it remains (in a more modern incarnation) a popular programming language in this area.

- **Business Applications:** Among the desires here are to do decimal arithmetic (to deal with monetary amounts) and to produce elaborate reports. COBOL was designed for this area, and few competitors have ever emerged. (See Robert Glass articles.)
- **Artificial Intelligence:** Tends to require symbolic manipulation rather than numeric, use of lists rather than arrays. Functional languages (e.g., LISP is the classic AI language) or logic languages (e.g., Prolog) are typically better-suited to this area.
- **Systems Programming:** Need for efficiency and for ability to do low-level operations (e.g., as used in device drivers). Classic language is C.
- **Web software:** Typically want good string processing capabilities. Scripting languages such as PHP and JavaScript have become popular here.

1.3 Language Evaluation Criteria

Aside from simply examining the concepts that underlie the various constructs/features of programming languages, Sebesta aims also to **evaluate** those features with respect to how they impact the software development process, including maintenance.

So he sets forth a few evaluation criteria (namely **readability**, **writability**, **reliability**, and **cost**) and several characteristics of programming languages that should be considered when evaluating a language with respect to those criteria.

See Table 1.1 on page 8. Then, for each of the criteria, Sebesta discusses how each of the characteristics relates to it.

1.3.1 Readability: This refers to the ease with which programs (in the language under consideration) can be understood. This is especially important for software maintenance.

One can write a hard-to-understand program in any language, of course (e.g., by using non-descriptive variable/subprogram names, by failing to format code acccording to accepted conventions, by omitting comments, etc.), but a language's characteristics can make it easier, or more difficult, to write easy-to-read programs.

- **1.3.1.1 Simplicity:**
  - **number of basic constructs/features:** if there are too many, the more likely a program will be hard to read (because reader might know a different subset of language than programmer).

    If there are very few (e.g., assembly language), code can be hard to read because what may be a single operation, conceptually, could require several instructions to encode it.

    - **feature multiplicity:** the existence of multiple ways of doing the same operation. Examples: incrementing a variable in C-based syntax, looping constructs (while, do while, for), Java's conditional ternary operator (**?:**).
    - **operator overloading:** can aid readability if used with discretion, but can lessen readability if used unwisely (e.g., by using + as a comparison operator).
- **1.3.1.2 Orthogonality:** In geometry, orthogonal means "involving right angles", but the term has been extended to general use, meaning the property of being independent (relative to something else).

  In the context of a programming language, a set of features/constructs is said to be *orthogonal* if those features can be used freely in combination with each other. In particular, the degree of orthogonality is lessened if

    - particular combinations are forbidden (as exceptional cases) or
    - the meaning of a particular combination is not evident from the meanings of its component parts, each one considered without regard to context.

  Examples of non-orthogonality in C:

    - A function can return a value of any type, except for an array type or a function type.
    - According to Sebesta, an array can hold values of any type, except for a function type or **void**. (Note that material on the WWW indicates that you *can* place *pointers to functions* in an array!)
    - Parameters to functions are passed "by value", except for arrays, which are, in effect, passed "by reference". (Is this a valid criticism? After all, one should understand a variable of an array type to have a value that is actually a pointer to an array. So passing an array to a function is really passing a pointer "by value". This is exactly how Java works when passing objects to methods. What is being passed is really a **reference** (i.e., pointer) to an object, not the object itself.)
    - In the expression `a + b`, the meaning of `b` depends upon whether or not `a` is of a pointer type. (This is an example of *context dependence*.)

  Example from assembly languages: In VAX assembler, the instruction for 32-bit integer addition is of the form

$$\texttt{ADDL } op_1 \; op_2$$

  where each of the op_i's can refer to either a register or a memory location. This is nicely orthogonal.

In contrast, in the assembly languages for IBM mainframes, there are two separate analogous ADD instructions, one of which requires `op`$_1$ to refer to a register and `op`$_2$ to refer to a memory location, the other of which requires both to refer to registers. This is lacking in orthogonality.

Too much orthogonality? As with almost everything, one can go too far. Algol 68 was designed to be very orthogonal, and turned out to be too much so, perhaps. As B.T. Denvir wrote (see page 18 in "On Orthogonality in Programming Languages", ACM SIGPLAN Notices, July 1979, accessible via the ACM Digital Library):

> Intuition leads one to ascribe certain advantages to orthogonality: the reduction in the number of special rules or exceptions to rules should make a language easier "to describe, to learn, and to implement" — in the words of the Algol 68 report. On the other hand, strict application of the orthogonality principle may lead to constructs which are conceptually obscure when a rule is applied to a context in an unusual combination. Likewise the application of orthogonality may extend the power and generality of a language beyond that required for its purpose, and thus may require increased conceptual ability on the part of those who need to learn and use it.

As an example of Algol 68's extreme orthogonality, it allows the left hand side of an assignment statement to be any expression that evaluates to an address!

- **1.3.1.3 Data Types:**

  Adequate facilities for defining data types and structures aids readability. E.g. Early FORTRAN had no record/struct construct, so the "fields" of an "object" could not be encapsulated within a single structure (that could be referred to by one name).

  Primitive/intrinsic data types should be adequate, too. E.g., Early versions of C had no **boolean** type, forcing programmer to use an **int** to represent true/false (0 is false, everything else is true, so `flag = 1;` would be used to set `flag` to true.) How about this statement fragment:

  ```
  if (k = 5) { ... } else { ... }
  ```

- **1.3.1.4 Syntax Design:**
  - **Identifier forms:** Should not be too restrictive on length, such as were FORTRAN 77 and BASIC. In COBOL, identifiers could include dashes, which can be confused with the subtraction operator.

- **Special words:** Words such as **while**, **if**, **end**, **class**, etc., have special meaning within a program. Are such words **reserved** for these purposes, or can they be used as names of variables or subprograms, too?
  The manner in which the beginning/end of a compound statement (e.g., loop) is signaled can aid or hurt readability. E.g., curly braces vs. **end loop**.
- **form and meaning:** Ideally, the semantics of a syntactic construct should be evident from its form. A good choice of special words helps this. (E.g., use **if**, not **glorp**.) It also helps if a syntactic form means the same thing in all contexts, rather than different things in different contexts. C violates this with its use of **static**.

1.3.2 Writability: This is a measure of how easily a language can be used to develop programs for a chosen problem domain.

- **1.3.2.1 Simplicity and Orthogonality:** Sebesta favors a relatively small number of primitive constructs (simplicity) and a consistent set of rules for combining them (orthogonality). (Sounds more like a description of "learnability" than of "writability".)
- **1.3.2.2 Support for Abstraction:** This allows the programmer to define and use complicated structures/operations in ways that allow implementation details to be ignored. This is a key concept in modern language design.

  Data abstraction and process (or procedural) abstraction.

- **1.3.2.3 Expressivity:** This is enhanced by the presence of powerful operators that make it possible to accomplish a lot in a few lines of code. The classic example is APL, which includes so many operators (including ones that apply to matrices) that it is based upon an enlarged character set. (There are special keyboards for APL!)

  Typically, assembly/machine languages lack expressivity in that each operation does something relatively simple, which is why a single instruction in a high-level language could translate into several instructions in assembly language.

  Functional languages tend to be very expressive, in part because functions are "first-class" entities. In Lisp, you can even construct a function and execute it!

1.3.3 Reliability: This is the property of performing to specifications under all conditions.

- **1.3.3.1 Type Checking:** This refers to testing for type errors, either during compilation or execution. The former is preferable, not only because the latter is expensive in terms of running time, but also because the earlier such errors are found, the less expensive it is to make repairs.

  In Java, for example, type checking during compilation is so tight that just about the only type errors that will occur during run-time result from explicit type casting by the programmer (e.g., when casting a reference to an object of class A into one of subclass B in a situation where that is not warranted) or from an input being of the wrong type/form.

  As an example of a lack of reliability, consider earlier versions of C, in which the compiler made no attempt to ensure that the arguments being passed to a function were of the right types! (Of course, this is a useful trick to play in some cases.)

- **1.3.3.2 Aliasing:** This refers to having two or more (distinct) names that refer to the same memory cell. This is a dangerous thing. (Also hurts readability/transparency.)
- **1.3.3.3 Readability and Writability:** Both have an influence upon reliability in the sense that programmers are more likely to produce reliable programs when using a language having these properties.

1.3.4 Cost: The following contribute to the cost of using a particular language:

- Training programmers: cost is a function of simplicity of language
- Writing and maintaining programs: cost is a function of readability and writability.
- Compiling programs: for very large systems, this can take a significant amount of time.
- Executing programs: Having to do type checking and/or index-boundary checking at run-time is expensive. There is a tradeoff between this item and the previous one (compilation cost), because **optimizing** compilers take more time to work but yield programs that run more quickly.
- Language Implementation System: e.g., Java is free, Ada not
- Lack of reliability: software failure could be expensive (e.g., loss of business, liability issues)

Other criteria (not deserving separate sections in textbook):

Portability: the ease with which programs that work on one platform can be modified to work on another. This is strongly influenced by to what degree a language is standardized.

Generality: Applicability to a wide range of applications.

Well-definedness: Completeness and precision of the language's official definition.

The criteria listed here are neither precisely defined nor exactly measurable, but they are, nevertheless, useful in that they provide valuable insight when evaluating a language.

1.7 Language Design Trade-offs

Not surprisingly, a language feature that makes a language score higher on one criterion may make it score lower in another. Examples:

- Reliability vs cost (of execution): The increase in reliability resulting from index range checking (such as is mandated in Java and Ada) is offset by an increase in the running time of a program.
- Writability vs. readability: A language that allows you to express computations/algorithms very concisely (such as APL, with its collection of powerful array and matrix operators) also tends to be difficult to read.
- Writability vs. reliability: A language that gives the programmer the convenience/flexibility of "breaking an abstraction" (e.g., by interpreting the contents of a chunk of memory in a way that is not consistent with the type declaration(s) of the data that is held there) is more likely to lead to subtle bugs in a program.

1.4 Influences on Language Design

1.4.1 Computer Architecture: By 1950, the basic architecture of digital computers had been established (and described nicely in John von Neumann's EDVAC report). A computer's machine language is a reflection of its architecture, with its assembly language adding a thin layer of abstraction for the purpose of making easier the task of programming. When FORTRAN was being designed in the mid to late 1950's, one of the prime goals was for the compiler to generate code that was as fast as the equivalent assembly code that a programmer would produce "by hand". To achieve this goal, the designers —not surprisingly— simply put a layer of abstraction on top of assembly language, so that the resulting language still closely reflected the structure and operation of the underlying machine. To have designed a language that deviated greatly from that would have been to make the compiler more difficult to develop and less likely to produce fast-running machine code.

The style of programming exemplified by FORTRAN is referred to as **imperative**, because a program is basically a bunch of commands. (Recall that, in English, a command is referred to as an "imperative" statement, as opposed to, say, a question, which is an "interrogative" statement.)

This style of programming has dominated for the last fifty years! Granted, many refinements have occurred. In particular, OO languages put much more emphasis on designing a program based upon the data involved and less on the commands/processing. But the notion of having variables (corresponding to memory locations) and changing their values via assignment commands is still prominent.

Functional languages (in which the primary means of computing is to apply functions to arguments) have much to recommend them, but they've never gained wide popularity, in part because they tend to run slowly on machines with a von Neumann architecture. (The granddaddy of functional languages is Lisp, developed in about 1958 by McCarthy at MIT.)

The same could be said for Prolog, the most prominent language in the logic programming paradigm.

Interestingly, as long ago as 1977 (specifically, in his Turing Award Lecture, with the corresponding paper appearing in the August 1978 issue of Communications of the ACM), John Backus (famous for leading the team who designed and implemented FORTRAN) harshly criticized imperative languages, asking "Can Programming be Liberated from the von Neumann Style?" He set forth the idea of an FP (functional programming) system, which he viewed as being a superior style of programming. He also challenged the field to develop an architecture well-suited to this style of programming.

Here is an interesting passage from the article:

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our thirty year old belief that there is only one kind of computer is the basis of our belief that there is only one kind of programming language, the conventional —von Neumann— language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. ...

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.

1.4.2 Programming Method(ologie)s: Advances in methods of programming also have influenced language design, of course. Refinements in thinking about **flow of control** led to better language constructs for selection (i.e., **if** statements) and loops that force the programmer to be disciplined in the use of jumps/branching (by hiding them). This is called **structured programming**.

An increased emphasis on data (as compared to process) led to better language support for data abstraction. This continued to the point where now the notions of **abstract data type** and **module** have been fused into the concept of a **class** in object-oriented programming.

1.5 Language Categories

The four categories usually recognized are **imperative**, **object-oriented**, **functional**, and **logic**. Sebesta seems to doubt that OO is deserving of a separate category, because one need not add all that much to an imperative language, for example, to make it support the OO style. (Indeed, C++, Java, and Ada 95 all are quite imperative.) (And even functional and logic languages have had OO constructs added to them.)

1.7 Implementation Methods

Computers execute machine code. Hence, to run code written in any other language, first that code has to be translated into machine code. Software that does this is called a **translator**. If you have a translator that allows you to execute programs written in language X, then, in effect, you have a **virtual X machine**. (See Figure 1.2.)

There are three general translation methods: **compilation**, **interpretation**, and a hybrid of the two.

1.7.1 Compilation: Here, a compiler translates each compilation unit (e.g., class, module, or file, depending upon the programming language) into an **object module** containing **object code**, which is like machine code except that two kinds of references have not yet been put into machine code form: **external references** (i.e., references to entities in other modules) and **relative references** (i.e., references expressed as an offset from the location of the module itself). Also —for the purpose of making subsequent steps in the translation possible— an object module contains tables in which are listed

- identifiers declared within the module that are free to be referenced in other modules,
- references in the module to entities defined elsewhere,
- relative references within the module

A **linker** is responsible for linking together the object modules that comprise a program, which means that it uses the tables in each object module to "resolve" all the external references. The result of the linker is a **load module**, which is a "relocatable" machine code program, i.e., one in which the only unresolved references are the relative references. When the time comes to execute the program, a **relocating loader** puts the code into the appointed area in memory, at the same time replacing all relative references by the actual memory addresses.

See Figure 1.3 for a depiction of the various phases that occur in compilation. The first two phases, **lexical** and **syntax** analysis, are covered in Chapter 4. The job of a lexical analyzer, or **scanner**, is to transform the text comprising a program unit (e.g., class, module, file) into a sequence of **tokens** corresponding to the logical units occurring in the program. (For example, the substring `while` is recognized as being one unit, as is each occurrence of an identifier, each operator symbol, etc.) The job of the syntax analyzer is to take the sequence of tokens yielded by the scanner and to "figure out" the program's structure, i.e., how those tokens relate to each other.

To draw an analogy with analyzing sentences in English, lexical analysis identifies the words (and possibly their parts of speech) and punctuation, which the syntax analyzer uses to determine the boundaries between sentences and to form a diagram of each sentence. Example sentence: The gorn killed Kirk with a big boulder.

```
    S      V       D.O.

  gorn | killed | Kirk
-------+--------+-------
 \T       \w
  \h       \i
   \e       \t
(adj)        \h   boulder
              --------------
                 \a  \b
                      \i
        (prep.         \g
         phrase)
```

1.7.2 Pure Interpretation: Let X be a programming language. An X **interpreter** is a program that simulates a computer whose "native language" is X. That is, the interpreter repeatedly fetches the "next" instruction (from the X program being interpreted), decodes it, and executes it. A computer is itself an interpreter of its own machine language, except that it is implemented in hardware rather than software.

1.7.3 Hybrid: Here, a program is translated (by the same means as a compiler) not into machine code but rather into some **intermediate** language, typically one that is at a level of abstraction strictly between language X and machine code. Then the resulting intermediate code is interpreted. This is the usual way that Java programs are processed, with the intermediate language being **Java bytecode** (as found in `.class` files) and the **Java Virtual Machine** (JVM) acting as the interpreter.

Alternatively, the intermediate code produced by the compiler can itself be compiled into machine code and saved for later use. In a **Just-in-Time** (JIT) scenario, this latter compilation step is done on a piecemeal basis on each program unit the first time it is needed during execution. (Subsequent uses of that unit result in directly accessing its machine code rather than re-translating it.)

1.8 Programming Environments

Collection of tools that aid in the program development process.

Chapter 5

Names, Bindings, Type Checking, and Scopes

Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants
- Variable Initialization

Chapter 5

Names, Bindings, Type Checking, and Scopes

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of memory
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

Design issues for names:

- Maximum length?
- Are connector characters allowed?
- Are names case sensitive?
- Are special words reserved words or keywords?

Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- If too short, they cannot be connotative
- Language examples:
  - FORTRAN I: maximum 6
  - COBOL: maximum 30
  - FORTRAN 90 and ANSI C: maximum 31
  - Ada and Java: **no limit**, and all are significant
  - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_).
- Although the use of the _ was widely used in the 70s and 80s, that practice is far less popular.
- C-based languages (C, C++, Java, and C#), replaced the _ by the “camel” notation, as in myStack.

- Prior to Fortran 90, the following two names are equivalent:

```
Sum Of Salaries  // names could have embedded spaces
SumOfSalaries    // which were ignored
```

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - worse in C++ and Java because predefined names are mixed case (e.g. **IndexOutOfBoundsException**)
    - In C, however, exclusive use of lowercase for names.
  - C, C++, and Java names are case sensitive ➔ rose, Rose, ROSE are distinct names "What about Readability"

Special words

- An aid to readability; used to delimit or separate statement clauses
- A **keyword** is a word that is special only in certain contexts.
- Ex: Fortran

```
Real Apple       // Real is a data type followed with a
                 name, therefore Real is a keyword
Real = 3.4       // Real is a variable name
```

- **Disadvantage**: poor readability. Compilers and users must recognize the difference.
- A **reserved word** is a special word that **cannot** be used as a user-defined name.
- As a language design choice, reserved words are **better** than keywords.
- Ex: In Fortran, one could have the statements

```
Integer Real     // keyword "Integer" and variable "Real"
Real Integer     // keyword "Real" and variable "Integer"
```

Variables

- A variable is an abstraction of a memory cell(s).
- Variables can be characterized as a sextuple of attributes:
    - Name
    - Address
    - Value
    - Type
    - Lifetime
    - Scope

Name

- Not all variables have names: **Anonymous**, heap-dynamic variables

Address

- The memory address with which it is associated
- A variable name may have different addresses at different places and at different times during execution.

  ```
  // sum in sub1 and sub2
  ```

- A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time **stack** when the subprogram is called, different calls may result in that var having different addresses.

  ```
  // sum in sub1
  ```

- The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement.

Aliases

- If **two variable** names can be used to access **the same memory location**, they are called **aliases**
- Aliases are created via **pointers**, **reference variables**, C and C++ **unions.**
- Aliases are harmful to readability (program readers must remember **all** of them)

Type

- Determines the **range** of values of variables and the set of **operations** that are defined for values of that type; in the case of floating point, type also determines the precision.
- 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.
- Abstract memory cell - the physical cell or collection of cells associated with a variable.
- A variable’s value is sometimes called its ***r*-value** because that is what is required when a variable appears in the **right** side of an assignment statement.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
- The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
- Possible binding times:
    - *Language design time*: bind operator symbols to operations.
        - For example, the asterisk symbol (*) is bound to the multiplication operation.
    - *Language implementation time*:
        - A data type such as **int** in C is bound to a **range** of possible values.
    - *Compile time*: bind a variable to a **particular data type** at compile time.
    - *Load time*: bind a variable to a **memory cell** (ex. C **static** variables)
    - *Runtime*: bind a **nonstatic** local variable to a memory cell.

Binding of Attributes to Variables

- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

Type Bindings

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.

Variable Declarations

- An **explicit declaration** is a program statement used for declaring the types of variables.
- An **implicit declaration** is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.
- EX:
    - In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
      I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
    - **Advantage**: writability.

  - **Disadvantage**: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
  - In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
- Less trouble with **Perl**: Names that begin with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
  - In this scenario, the names `@apple` and `%apple` are unrelated.
- In **C and C++**, one must distinguish between declarations and definitions.
  - **Declarations** specify types and other attributes but do **no** cause allocation of storage. Provides the type of a var defined external to a **function** that is used in the function.
  - **Definitions** specify attributes and cause storage allocation.

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript

```
list = [2, 4.33, 6, 8];     ➔ single-dimensioned array
list = 47;                  ➔ scalar variable
```

- Advantage: **flexibility** (generic program units)
- Disadvantages:
  - **High cost** (dynamic type checking and interpretation)
    - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
    - Pure interpretation typically takes at least **ten times** as long as to execute equivalent machine code.
  - **Type error detection by the compiler is difficult** because **any** variable can be assigned a value of **any** type.
    - Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
    - Ex:

```
i, x  ➔ Integer
y     ➔ floating-point array
i = x ➔ what the user meant to type
i = y ➔ what the user typed instead
```

    - **No error** is detected by the compiler or run-time system. i is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Type Inference (ML, Miranda, and Haskell)

- Rather than by assignment statement, types are determined from the context of the reference.
- Ex:

```
fun circumf(r) = 3.14159 * r * r;
      The argument and functional value are inferred to
      be real.

fun times10(x) = 10 * x;
      The argument and functional value are inferred to
      be int.
```

Storage Bindings & Lifetime

- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes: **static**, **stack-dynamic**, **explicit heap-dynamic**, and **implicit heap-dynamic**

Static Variables:

- bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- e.g. all FORTRAN 77 variables, C static variables.
- **Advantages**:
  - **Efficiency**: (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocating and deallocating vars.
  - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- **Disadvantage**:
  - Storage **cannot** be shared among variables.
  - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Stack-dynamic Variables:

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- Ex:
  - The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.
- Ex:
  - Local variables in C subprograms and Java methods.
- **Advantages**:
  - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
  - In the absence of recursion it conserves storage b/c all subprograms share the same memory space for their locals.

- **Disadvantages**:
  - Overhead of allocation and deallocation.
  - Subprograms cannot be history sensitive.
  - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.

Explicit Heap-dynamic Variables:

- Nameless memory cells that are allocated and deallocated by explicit directives "run-time instructions", specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;
…
intnode = new int; // allocates an int cell
…
delete intnode;  // deallocates the cell to which
                 // intnode points
```

- An explicit heap-dynamic variable of int type is created by the new operator.
- This operator can be referenced through the pointer, intnode.
- The var is deallocated by the **delete** operator.
- Java, all data except the primitive scalars are **objects**.
- Java objects are explicitly heap-dynamic and are accessed through **reference variables**.
- Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- **Advantage**:
  - Provides for dynamic storage management.
- **Disadvantage**:
  - Inefficient "Cost of allocation and deallocation" and unreliable "difficulty of using pointer and reference variables correctly"

Implicit Heap-dynamic Variables:

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment statements**.

- All their attributes are bound every time they are assigned.
    - e.g. all variables in APL; all strings and arrays in Perl and JavaScript.
- **Advantage**:
  - Flexibility allowing generic code to be written.
- **Disadvantages**:
    - Inefficient, because all attributes are dynamic “run-time.”
    - Loss of error detection by the compiler.

Type Checking

- **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 a **coercion**.
- Ex: an **int** var and a **float** var are added in Java, the value of the **int** var is coerced to **float** and a floating-point is performed.
- A **type error** is the application of an operator to an operand of an inappropriate type.
- Ex: in C, if an **int** value was passed to a function that expected a **float** value, a type error would occur (compilers **didn't** check the types of parameters)
- If all type bindings are static, nearly all type checking can be static.
- If type bindings are dynamic, type checking must be dynamic and done at run-time.

Strong Typing

- A programming language is strongly typed if type errors are **always** detected. It requires that the types of all operands can be determined, either at compile time or run time.
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors.
- **Java and C#** are strongly typed. Types can be explicitly cast, which would result in type error. However, there are no implicit ways type errors can go undetected.
- The coercion rules of a language have an important effect on the value of type checking.
- Coercion results in a loss of part of the reason of strong typing – error detection.
- Ex:

```
int a, b;
float d;
a + d;    // the programmer meant a + b, however
```

– The compiler would not detect this error. Var a would be coerced to **float**.

Scope

- The scope of a var is the range of statements in which the var is visible.
- A var is **visible** in a statement if it can be referenced in that statement.
- **Local var** is local in a program unit or block if it is declared there.
- **Non-local var** of a program unit or block are those that are visible within the program unit or block but are not declared there.

Static Scope

- Binding names to non-local vars is called **static scoping**.
- There are two categories of static scoped languages:
  - Nested Subprograms.
  - Subprograms that can't be nested.
- Ada, and JavaScript allow **nested** subprogram, but the C-based languages do not.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- Ex: Suppose a reference is made to a var **x** in subprogram **Sub1**. The correct declaration is found by first searching the declarations of subprogram Sub1.
- If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its **static parent**.
- If a declaration of x is not found there, the search continues to the next larger enclosing unit (the unit that declared Sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. ➔ an undeclared var error has been detected.
- The static parent of subprogram Sub1, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of Sub1.

Ex: Ada procedure:

```
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
```

- 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, Big, where the declaration of X is found.
- Ex: Skeletal C#

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

- The reference to count in the while loop is to that loop's local count. 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

Blocks

- Allows a section of code to have its own local vars whose scope is minimized.
- Such vars are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- From ALGOL 60:
- Ex:

C and C++:

```
for (...)
{
  int index;
  ...
}
```

Ada:

```
declare LCL : FLOAT;
begin
...
end
```

Dynamic Scope

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic.
- Based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Ex:

```
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
```

- Big calls Sub1
  - The dynamic parent of Sub1 is Big. The reference is to the X in **Big**.
- Big calls Sub2 and Sub2 calls Sub1
  - The search proceeds from the local procedure, Sub1, to its caller, **Sub2**, where a declaration of X is found.
- Note that **if static scoping** was used, in either calling sequence the reference to X in Sub1 would be to **Big's X**.

Scope and Lifetime

- Ex:
```
void printheader()
{
…
}    /* end of printheader */
void compute()
{
   int sum;
   …
   printheader();
}    /* end of compute */
```

- The **scope** of sum in contained within compute.
- The **lifetime** of sum extends over the time during which printheader executes.
- Whatever storage location sum is bound to before the call to printheader, that binding will continue during and after the execution of printheader.

Referencing environment

- It is the collection of all names that are visible in the statement.
- In a **static-scoped language**, it is the local variables plus all of the visible variables in all of the enclosing scopes.
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to non-local vars in both static and dynamic scoped languages.
- A subprogram is active if its execution has begun but has not yet terminated.
- In a **dynamic-scoped language**, the referencing environment is the local variables plus all visible variables in all active subprograms.
- Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin      -- of Sub1
      …                          ← 1
       end       -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin   -- of Sub3
         …                       ← 2
         end;    -- of Sub3
    begin  -- of Sub2
    …                            ← 3
     end;   { Sub2}
begin
   …                             ← 4
end;        {Example}
```

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

Point	*Referencing Environment*
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

- Ex, **dynamic-scoped language**
- Consider the following 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
2	b and c of sub2, d of main
3	c and d of main

Named Constants

- It is a var that is bound to a value only at the time it is bound to storage; its value **can't** be change by assignment or by an input statement.
- Ex, Java

```
final int LEN = 100;
```

- **Advantages**: readability and modifiability

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, Java
  int sum = 0;