HINDUSTAN
INSTITUTE OF TECHNOLOGY & SCIENCE
(DEEMED TO BE UNIVERSITY)

CSB4404 – PROGRAMMING PARADIGMS

B.Tech – VII Semester

Dr. Muthukumaran M
Associate professor
School of Computing Sciences,
Department of Computer Science and Engineering

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)

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!)

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.

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 , Then, for each of the criteria, Sebesta discusses how each of the characteristics relates to it.

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			•

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.

Readability Cont….

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).

Readability Cont….

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

```
ADDL op1 op2
```

where each of the opi'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 op1 to refer to a register and op2 to refer to a memory location, the other of which requires both to refer to registers. This is lacking in orthogonality.

Readability Cont….

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 { ... }
```

Readability Cont….

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.

Writability

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

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".)

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.

Writability Contd…

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!

Reliability

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.)

Reliability Contd…

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.)

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.

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

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.