Potential Criteria (Partly Overlapping/Interrelated) for Analyzing and Evaluating a Programming Language

Andrés Gómez de Silva Garza

1) Expressivity. Can instructions be written in a language at the same level of abstraction and using the same types of symbols and/or operations used to express algorithms in the application domain? Are operators and pre-defined instructions used/interpreted consistently?

Example 1:
In FORTRAN, calculating 6 squared is done like this:

```
6**2
```

In BASIC, calculating 6 squared is done like this:

```
6^2
```

In Java, calculating 6 squared is done like this:

```
Math.pow(6,2)
```

Conclusion: FORTRAN and BASIC are more expressive than Java when it comes to exponential operations. Neither FORTRAN nor BASIC use the same notation used in mathematics for exponents (the exponent as a superscript to the right of the base, with no additional symbol between them), but at least they both introduced a special symbol/operator to represent exponents. Java requires the use of a pre-defined function in a separate (pre-defined) package, which leads to more "verbose" statements.

Example 2:
In Pascal the + symbol is used to represent the addition operator, as in mathematics/science/engineering.

In Java the + symbol can be used to represent the addition operator or the concatenation operator, depending on the context. Compare:

```
int a=1,b=2,c;
c=a+b;
```

with

```
int a=1;
String s="Student";
System.out.println(s+1);  // Output text: Student1
```

Conclusion: the interpretation of symbols (operators) is more uniform/consistent/unambiguous in Pascal than in Java.

Example 3:
In Pascal the comparison (for equality) and assignment operators are represented by different symbols:

```
if a=2 then
      x:=40;
```

In Java the comparison (for equality) and assignment operators are represented by different symbols (both different from those of Pascal):

```
if(a==2)
```

```
        x=40;
```

In BASIC the comparison (for equality) and assignment operators are represented by the same symbol:

```
If a=2 then
      x=40
End If
```

Conclusion: Pascal and Java are more expressive than BASIC when it comes to visually differentiating between the conceptually different operations of comparison (for equality) and assignment.

2) Well-definedness. Are the syntax and semantics of a language specified unambiguously, consistently, and completely? Can a programmer predict a program's behavior before executing it (or at least retroactively understand the behavior, if it is not the predicted one, based on the language's rules of syntax and semantics, after executing the program)?

Example:
In old versions of FORTRAN it was not necessary to declare variables and spaces were ignored by the compiler. Therefore an attempt to define a cycle/loop within a program (that assigns the value 1 to the variable *I* ten times) like:

```
DO 10 I=1
```

was actually understood by the compiler as an assignment of a 1 value to a variable named *DO10I*.

```
DO10I=1
```

3) Data types and structures. Does a language support (i.e., know the semantics of, or allow the programmer to specify the semantics of) a variety of data types and collections? Does the compiler/interpreter verify the correct use of the same?

Examples:
LISP, Python, old versions of BASIC and FORTRAN, and some new versions of these last two languages, do not require variables to be declared (i.e., their names mentioned and their types defined by the programmer) before using them. Pascal, C, and Java are strictly typed languages and require all variables to be declared before using them.

In the original versions of LISP there were only two types of data: atomic values or lists. Later on it was determined that it would be good for the language to also distinguish numeric from non-numeric data (and to distinguish between different types of numeric data). Java provides many more predefined data types: *int* (and several variations on it according to size/precision), *float* (and variations), *char*, *boolean*—all of which are considered primitive types—plus String and many other predefined classes.

In Java there are two predefined collection types: Strings (which group together several *char*s) and arrays (which can be used to group together multiple values of their declared type). In Python there are six predefined collection types: strings, lists, sets, frozen sets, tuples, and dictionaries. Python provides more flexibility and variety than Java as far as the programmer being able to group together values under a wider set of options of different behavioral and structural characteristics and restrictions.

4) Modularity. Does a language permit, or even force, a structured (modular) approach to defining both code and data types? Is it easy to develop programs piece-meal, whether by individual programmers or teams of them?

Examples:
Pascal is designed so that programs are divided into multiple functions and procedures (with one unnamed main procedure having to be included at the end of every program, which is where the execution of the program begins)—see CDZ.PAS. Java extends this idea (though the main procedure does have to have a specific name: *main*) by allowing multiple classes (user-defined types, including the methods that perform operations on their attributes) to each be defined in its own sub-program. Programs in old versions of BASIC were always one long, visually undivided (even if conceptually sub-divided), sequence of statements (see DEMO.BAS).

5) Input-output facilities. How easy/flexible is it to program the reading/writing of data from/to secondary storage devices in a language? How much support does a language provide for database access/queries?

Examples:
The original specification for LISP did not really pay attention to input/output, though most dialects developed implemented platform-specific instructions that allowed input/output (usually non-formatted) operations (e.g., PRINT). Input/output in C is not really supported other than at the byte level. Java input/output is cumbersome, but a lot more flexible.

6) Portability/standardization. How easy is it to take executable code or source code written in one computer platform and use it in a different platform? Are there lots of dialects of the language or has there been a limited number of widely-used standards developed?

Examples:
Many dialects have been developed of LISP, BASIC, and PROLOG (usually platform-specific, and many of them existing concurrently), and code that works in one platform, with one dialect, will not necessarily run under a different platform, with a different dialect. On the other hand, FORTRAN, Pascal, and C have had few variations developed, and these have normally been done in standard fashion by committees that "control" the evolution of the languages, so there has only been one "standard" in operation at any one time (and usually newer versions have been designed to be backward-compatible), so there are less issues (though there are still some) with porting code across platforms or versions. Java has many deprecated methods but these have been well-documented and the new versions of the compiler will advise on the fact (and the language documentation discusses the "replacement" methods), so it's relatively easy to upgrade code. Java's virtual machine design easily enables cross-platform portability.

7) Efficiency. How quickly can a programmer develop, test, and compile code in a language? How much effort does the language make to produce code that executes quickly?

Examples:
Languages that are generally interpreted such as BASIC and PROLOG make it easier to test code (or fragments of code) during development compared to compiled languages (where extra testing code has to be written as if it were part of the program for each program fragment that needs to be tested). Languages that are "closer" to the hardware (and thus less higher-level), in the sense of providing instruction sets that more directly reflect those available as primitive instructions in the micro-processor, such as C, tend to produce code that executes more quickly than equivalent code written in higher-level languages such as Java.

8) Pedagogy/simplicity. How easy is it to teach/learn a language? Are there enough text books, documentation, tutorials, etc., available for potential programmers?

Examples:
BASIC, Pascal and Modula-2 were designed with pedagogy in mind. They all follow simple basic design principles in systematic fashion, making the language and its philosophy easier to teach/learn. Languages such as C (with its flexible, and therefore often inconsistently-used and difficult-to-decrypt, syntax) and Java (with its extensive use of, and support for, widely-available libraries and features that allow handling of all sorts of things, from graphics to input/output formatting to threads to security issues), are less easy to learn/teach completely.

9) Generality. Does a language support a wide range of potential applications or is its use restricted to highly-specific domains/situations? How much support does the language provide for reusability (programming generic code that can be easily applied to more general situations than it was originally designed for)?

Examples:
The languages that are widely used tend to be the ones that are more general-purpose, such as BASIC, Pascal, C, and Java. Languages that were designed with specific types of applications in mind (e.g., APL and SNOBOL) or that do not easily support (or support in a standardized manner) certain types of important operations (e.g., input/output in the case of LISP or Prolog) are not very widely used. Languages like Java and C++ support reusability by providing constructs that allow the definition of generic data and/or code. Languages like Pascal that require the types of all variables (including function/procedure parameters) to be clearly specified, and that distinguish between arrays containing five elements and arrays containing ten elements, for example, require the re-definition of code for re-use even in very-similar situations, and are thus not that flexible/general.

10) Modeling ability. Can a wide variety of "real-world" situations be modeled easily/directly in a language? Does the language provide enough support for abstraction?

Examples:
APL is really good for heavy-duty vector and matrix (and other higher-order array) calculations and manipulations, but is not much good for anything else, whereas Java can be used (not necessarily in as compact or readable or friendly a manner as APL) for heavy-duty array operations but also for text processing, artificial intelligence, computer games, and a wide variety of other applications because of its flexibility in modeling the world. Java's modularity and generics permit a high degree of abstraction when defining data and code whereas older versions of COBOL had no support for abstraction in defining data and code.

11) Readability. How easy is it for programmers that did not write a specific piece of code in a language to understand it? Does the syntax of the language permit cryptic instructions or does it promote ease of understandability?

Example:
How easy is it to "decipher" the fact that the following method written in Java prints the number of characters that appear to the left of a specified character within a given character string (and prints the total number of characters in the string if the specified character does not appear in said string)?

```
public static void x(char c,String s) {
	int n=0;
	for(n;n<s.length()&&s.charAt(n)!=c;n++);
	System.out.println(n);
}
```

The same method could have been re-written in a much more readable fashion, and without writing many more characters than in the previous version, as follows (though the names of the method and perhaps its variables, as well as the message that is output to the console, could still be enhanced for even better readability and user-friendliness):

```
public static void x(char c,String s) {
	int n=0;
	while(n<s.length()&&s.charAt(n)!=c)
		n=n+1;
	System.out.println(n);
}
```

12) Semantic clarity. Does every legal statement in a language have a clear, unambiguous meaning? Does this meaning remain constant throughout a program's execution?

Example:
The use of the increment (++) operator is ambiguous in C and Java; when used in one statement together with one or more other instructions/operators its semantics (the order in which it is performed in relation to the other instructions in the statement) varies depending on if it was used as a prefix or postfix operator. The first two versions of the following recursive method for printing the contents of an array of integers will work just fine (though the second one is slightly more legible, because the quantity and type of operations that are occurring in the computer's memory are more directly/explicitly visible, than the first), whereas the last version will remain stuck in an infinite loop because the increment will be performed after (except that there is no after) the sequence of recursive calls ends. The difference in semantics is clear to the compiler, but it is very easy to be overlooked by (or simply not be known to) the programmer:

```
public static void printArray(int[] a,int i) {
	if(i<a.length) {
		System.out.print(a[i]+" ");
		printArray(a,++i);
	}
}

public static void printArray(int[] a,int i) {
	if(i<a.length) {
		System.out.print(a[i]+" ");
		printArray(a,i+1);
	}
}

public static void printArray(int[] a,int i) {
	if(i<a.length) {
		System.out.print(a[i]+" ");
		printArray(a,i++);
	}
}
```