CS323 Lecture: Language Evaluation Criteria Last revised 1/6/09

Objectives:

1. To consider criteria for evaluating programming languages.

Introduction

A. The design and evaluation of programming languages is a challenging area because - as we shall see - there is no such thing as a "best" language. Instead, existing languages are strong by some criteria and weak by the others, so that the choice of a language for a particular purpose is tied to a decision as to which criteria are most important.

B. We will now consider a number of criteria which can be used in developing a language design, or evaluating an existing one. The criteria listed below come from books by Alan Tucker and Ellis Horowitz's - which have good chapters on this topic. I have grouped them somewhat differently, though - under three main categories:

1. Criteria Relating to Ease of Using a Language

2. Criteria Relating to Software Engineering

3. Criteria Relating to Performance

C. You should apply these criteria when you are doing language evaluations as part of your Mini-Projects.

I. Criteria Relating to Ease of Using a Language

A. Programming languages are used by programmers to write programs. Thus, a good language should make it easy for a programmer to express what needs to be done. Several criteria contribute to making a language easy to use.

B. The first criterion we will consider is WELL-DEFINEDNESS. Both the syntax and the semantics of the language should be clearly defined.

1. Syntax answers the question "What forms does the language allow?"

a. This is important, so that a programmer knows how to construct statements that will be accepted by the compiler. If the syntax definition is ambiguous, then the programmer may have to resort to trial and error. Even worse, different compilers for the same language may differ in their interpretation of an ambiguity, leading to portability problems.

b. There are a number of notation systems which can be used to spell out the syntax of a language. (We will study these later.) Any one of these systems can be used to spell out syntax unambigously, though some are more readable for humans than others are.

2. Semantics answers the question "what does this form mean?".

a. The importance of this for the programmer is obvious. Ambiguity here may again force the programmer to resort to trial and error.

Example: a classic example of ambiguous semantics is the "dangling else" problem:

should a construct of the form

```
if (B1) if (B2) S1 else S2
```

[ where B1 and B2 are boolean expressions and S1 and S2 are statements ]

be interpreted as:

```
if (B1)
    if (B2)
        S1
    else                          -- else goes w/second if
        S2
```

or as

```
if (B1)
    if (B2)
        S1
else                              -- else goes w/first if
     S2
```

- Obviously, the problem can always be avoided by using braces (or its equivalent) around the inner if. But this is inconvenient and programmers don't usually do this.

- ALGOL handled this by forbidding the "then" part of an if from being another if. Should a programmer want a construct like this, he would be FORCED to use begin .. end.

- Some newer languages require an explicit "end if" to terminate an if .. then .. else, also avoiding the problem (e.g. FORTRAN77, Modula-2, Ada)

- Most languages resolve the ambiguity as Java does, by matching the else with the NEAREST if that has no else (the first interpretation above.) But this rule is often not clearly stated in the language manual (though in the case of Java it is, but in a strange way!)

b. Further, if the semantics of language construct are ambiguous, and two compilers interpret the ambiguity differently, then when a program is ported from one compiler to another it may no longer run correctly - an even worse problem than that arising from a syntactic ambiguity, since the compiler will give the programmer no indication of the problem.

Example: in a boolean expression like (i < max) && (x[i] > 0), is the second comparison done if the first fails? This is an important question if the second comparison would cause a run-time error if the first comparison is false.

- Java addresses this question by saying that, in a case like this, the second comparison is NEVER done if the first fails.

- Ada addresses this question by saying that both comparisons are ALWAYS done. It provides a different construct for use when one wants to not avoid doing the second comparison: (i < max) and then (x[i] > 0)

- Some other languages leave this question unresolved, and different compilers may produce different results. (In fact, in some cases the SAME compiler may handle this differently in different contexts - this lead to some interesting problems with our old Pascal compiler!)

c. Here, unfortunately, the descriptive tools are not as well developed as they are for syntax. We will survey some formal tools later; but often semantics are simply described in English.

C. Another criterion is CONSISTENCY WITH COMMONLY USED NOTATION, or what Tucker calls EXPRESSIVITY. This point can be illustrated best by looking at some violations of this criterion.

1. In writing mathematical expressions, the normal practice is to write the arithmetic operators in infix form - e.g. we write

```
                x + 1   (infix)
instead of
                + x 1   (prefix)
or              x 1 +   (postfix)
```

   Most programming languages adhere to this convention, though some don't - notably LISP (prefix) and FORTH (postfix). This makes programs in these languages harder to read and write - though one eventually gets used to it.

2. Again, in writing mathematical expressions, certain conventions are normally understood with regard to operator precedence. For example,

```
       3 * x + 2
```

   is normally understood to mean

```
       (3 * x) + 2
```

   Most programming languages adhere to conventional rules of operator precedence, but some do not. For example, in APL the unparenthesized expression wouldbe interpreted as

```
       3 * (x + 2)
```

3. In conventional mathematical notation, the = operator means the ASSERTION that two things are equal, as distinct from the assignment operation MAKE two things equal.

   a. Thus, Algol (and its desendants through Pascal) used = for comparison and := for assignment.

b. Some other languages use = for both purposes, leading to possible misreadings. (Examples: BASIC, COBOL).

c. Other languages use = for assignment and something else for comparison, contrary to conventional mathematical usage. (Examples: FORTRAN, and the C descendants of ALGOL including Java).

D. A language should have good facilities for INPUT-OUTPUT.

1. This turns out to be one of the hardest parts of a language to design, because there is an inevitable dependence here on distinctive characteristics of different hardware devices. Facilities for interactive IO to terminals, for example, must be somewhat different from those for reading/writing disk files.

2. One issue that langauges handle quite differently is whether the IO facilities of a language should be part of the language definition itself, or provided by library procedures.

a. FORTRAN, COBOL, and a number of others take the first approach, with the language including READ and WRITE statements.

b. C and its descendants (including Java) and Ada (among others) take the second approach. IO operations are done through calling library procedures. A basic IO library is furnished with the language system, but a programmer is free to extend it to handle special needs.

3. Another issue is facilities for FORMATTING output. COBOL is very strong on this count, and others (like FORTRAN) include good facilities; while other languages are quite weak. (For example, formatted output is very hard to do in Java).

E. A language should also be UNIFORM. That is, similar constructs should have similar meanings. Again, this can be illustrated by a counter-examples:

1. In the C family of languages (including Java), parameters to functions are normally passed by value. Thus, given the C function definition

```
int f(x)
  int x;
  {
     x = 2 * x;
     return x;
  };
```

and a call to the function

```
int a = 2;
b = f(a);
// a still has the value 2 here
```

The assignment to the formal parameter x in f has no effect on the actual parameter a.

2. But if the parameter is an array, then it is passed by reference instead. Thus, given the very similar function definition:

```
int f(x)
  int x[];
  {
     x[1] = 2 * x[1];
     return x[1];
  };
```

and a call to the function

```
int a[2];
a[0] = 2;
b = f(a);
// The value of a[0] is now 4!
```

The assignment to the formal parameter x in f WILL ALTER the first element of the actual parameter a.

F. A similar concept is that a language should be ORTHOGONAL. An orthogonal language is one which has a limited number of features, each of which can be understood by itself, which can be combined in any way to produce a variety of results.

1. This was a major design goal of ALGOL68, but has been less characteristic of most other languages.

2. As a counter-example, consider the matter of types and functions in Pascal.

   a. In Pascal, a variable can be declared to be of any type, and a variable of a given type can be assigned a value of that type by an assignment statement.

   b. A function can take parameters of any type, but cannot return values of certain types. (Specifically, it cannot return an array or a record.)

   c. That is, to use Pascal, one must not only learn about types and functions, but must also learn the rule that functions cannot return certain types. These two otherwise unrelated concepts in the language interact in an unusual way. In a truly orthogonal language, functions would be able to return values of any type.

   d. The difficulty can be pictured this way:

| | | Use | | |
|---|---|---|---|---|
| | | Variable | Parameter | Return value |
| Type | Scalar | OK | OK | OK |
| | Structured | OK | OK | No! |

(In an orthoganal language, all six squares would be OK)

G. A debatable criterion is whether a language should be GENERAL - i.e. capable of tackling any type of problem.

1. Carrying this too far can lead to failure. For example, in the late 1960's, IBM promoted a language called PL/1 (programming language 1) that was intended to replace FORTRAN, COBOL and ALGOL - among others - by incorporating facilities that would allow one to do everything one could do with FORTRAN and COBOL, with the elegance of ALGOL. This attempt, however, failed miserably; and though PL/1 is still in use, few use it.

2. In fact, some of the most useful languages are those that are specifically designed for a particular class of problems - e.g. those designed for programming numerically-controlled machine tools, or solving civil engineering problems, or the like.

3. However, to gain wide use, a language does have to have broad usefulness for different kinds of problems, so generality is generally a good thing!

H. Finally, a language should have good PEDAGOGY - it should be easy to learn.

1. Several of the features we have already considered contribute to pedagogy - e.g. consistency with commonly used notation, uniformity, orthogonality.

2. On the other hand, abundance of features tends to make a language hard to use.

a. That is, generality can sometimes conflict with pedagogy.
b. However, it is possible to achieve a good balance between both. One of the reasons for Pascal's popularity as a teaching language is that it is powerful, but relatively small. One can master the entire language in an introductory course sequence.

3. For larger languages, one approach that has sometimes been taken to pedagogy is the development of subsets - i.e. smaller versions of the language that include necessary features while excluding minor ones. Perhaps the most thorough example of this is a set of subsets of a variant of PL/I, known as SP/1, SP/2, SP/3 ... - each of which includes more features than the preceeding one. In the case of Ada, though, this approach was explicitly ruled out the Department of Defense, which holds the copyright to the name Ada.

II. Criteria Relating to Software Engineering

A. Beyond ease of use, it is important that a programming language support the development of CORRECT software, even when writing large systems. The next group of criteria we consider pertain to support for good software engineering.

B. It is important that the language make it difficult to make careless errors that go undetected by the compiler. Horowitz calls this characteristic RELIABILITY.

1. Many early programming languages - and some recent ones - do not require that a variable be explicitly declared by the programmer.

a. For example, in FORTRAN, a variable that is not explicitly declared is implicitly declared the first time it is used, with its type being determined by the first letter of its name. (Names beginning with I..N are integers; all others are real)

b. Why is this bad? ASK

c. Consider what happens if a programmer makes a typographical error, misspelling the name of a variable. In a language that requires that all variables be declared, this will almost always be caught by the compiler as an "undeclared identifier" (unless the typo happens to come out the same as another variable.) In languages like FORTRAN, though, the compiler will usually not catch such an error.

d. Of course, it may be argued that the requirement of declaring every variable is an inconvenience for the programmer. But, in this case, the inconvenience of having to track down a subtle bug due to a mistyped variable is even worse, so it's worth it.

2. A reliability feature related to the requirement of declaring variables is type checking.

a. As you know, languages like Java check that the use of a variable is consistent with its declaration. For example, the following will be detected as erroneous:

```
boolean b;
System.out.println(b*2);
```

b. Many languages do not do this. For example, consider the following legal FORTRAN program:

```
      SUBROUTINE SUB(A)
      INTEGER A(100)
      ...
      DO 10 I = 1,100
100       A(I) = A(I) + 1
      END
      ...
      REAL B, C
      ...
      CALL SUB(B)
      ...
```

The compiler will permit this, and on the first time through the loop the subroutine will treat the bit pattern for the real parameter B as if it were an integer, producing strange results from the addition. Even worse, on subsequent times through the loop it will move past the memory allocated to B. Thus, the operation on A(2) will probably be done on C, and the operation of A(3) .. A(100) on who knows what - perhaps even the code will be damaged!

3. Finally, the commenting conventions of a language are also a factor in reliability.

   a. Languages tend to approach commenting in one of three ways.

      i. In some languages, comments occupy entire lines unto themselves. For example, FORTRAN uses a C in column 1 to specify a comment line, and COBOL uses a * in column 7 for the same purpose.

      ii. Other languages use pairs of comment delimiters to bracket comments. A comment may start and end anywhere, so you can have comments in the middle of a line or extending over a full line or many lines. For example, Pascal uses (* and *) or { and } this way; C, Java and PROLOG use /* and */, etc.

      iii. Finally, some languages use a symbol to start a comment, which may appear anywhere on the line. The comment occupies the rest of that line - i.e. the end of the line closes the comment. For example, Lisp uses ; this way and Ada uses -- this way:

```
if DISCR < 0.0 then  -- roots are complex
    S := sqrt(-DISCR)
    ...
```

      iv. Also, there are languages that use multiple approaches - e.g. Java supports "remainder of line" comments using //

   b. What convention is most reliable? Obviously, one will get differences of opinion on this point, but one approach does have the edge.

      i. The approach taken by FORTRAN and COBOL tends to discourage commenting, because a comment occupies an entire line. In particular, it is hard to associate a comment about what a variable is used for with its declaration - a good practice.

      ii. The approach taken by Java and many other languages suffers from the danger of mistyping the closing comment bracket. For example, almost everyone at some time has added some comments to a working program. If one mistypes the closing bracket (e.g. as * / or the like), then all of the code between that bracket and the end of the NEXT comment is "commented out". If the code happens to still be syntactically correct, a working program may cease to work without warning.

      iii. The designers of Ada studied typical errors made by programmers, and concluded that the convention they adopted was the most reliable.

C. Another software engineering consideration is the language's support for MODULARITY. A large software project is typically constructed of modules, each of which interfaces with the rest of the system in certain well-defined ways.

   1. A subroutine facility, such as that found in FORTRAN or COBOL, or a block-structured procedure facility, as in ALGOL or Pascal, is one way to address this need; but such facilities are not as good as they might be.

2. Some languages, such as Modula-2 and Ada, provide more sophisticated features to support modular software, as we shall see later in the case of Ada.

3. One of the great strengths of object orientation is the modularity inherent in the notion of a class.

D. A closely related issue is SUPPORT FOR SEPARATE COMPILATION.

1. For small programs, it is common for the entire program to reside in a single file that is compiled as a single unit. But for larger programs, it is almost essential to allow the program to be spread over multiple files (perhaps 1000's) compiled separately. In this way, when a change is made, only the affected file(s) need to be recompiled.

2. Many languages support this by adding a separate step to the program build process called LINKING.

   a. Each source file is compiled to produce an object file.

   b. A separate linking step combines all the object files, together with needed code from libraries, into a single executable file.

   c. Example: a command such as gcc or g++ actually invokes both of' these steps.

   d. Of course, the longer the program, the more work the linking step has to do.

3. To make this work, there must be a mechanism whereby a module being compiled can be aware of "public" features of other modules. This is often handled by splitting a module into two files.

   a. Example: the .h and .c/.cc files used by C/C++.

   b. Some languages use a single source file, but the compiler produces two object files - one containing just declaration information and the other the actual code. Other modules need to explicitly import the former, while the linker uses the latter.

      Examples: Ada, Modula

   c. Some languages include declaration information in a single output file produced by compilation that other modules can use via import.

      Example: Java .class files

   d. One tricky issue is ensuring that, when the interface of a module is changed, other modules that depend on it are recompiled. There is no totally general solution to this problem, short of a "clean build".

E. Another important consideration is the different DATA TYPES AND DATA STRUCTURING FACILITIES available in the language. Ideally, the type structure should be EXTENSIBLE, allowing the programmer to easily create and use new data types to fit the problem at hand.

   1. FORTRAN is an example of a language that is particularly weak on this score, having only arrays as structured types - no records or pointer variables. Thus, what would be done with structs/classes in C like languages will have to be done with individual variables in FORTRAN, and linked structures can only be implemented by using arrays of nodes - dynamic storage allocation is not possible. There is no facility for declaring new data types, either. A number of other languages share this shortfall, including APL and BASIC.

   2. Most languages include at least arrays, records, and pointers as structuring facilities. Also, they typically include a data type creation facility, though the operations available on user-created types are limited. Thus, user-created types are "second-class citizens".

   3. Object-oriented languages such as Java carry this even further, of course.

   4. Some languages (e.g. Ada, C++) even allow the standard operators to be redefined for user-defined data types.

F. Another consideration that is not as often considered in choosing a language is PROVABILITY - the extent to which the language lends itself to using formal methods to prove the correctness of a program.

   1. As you recall, it is possible to construct a program proof by embedding precondition and postcondition assertions into the program - e.g.

      ```
      /* data is an array of integers */

      max = data[0];
      for (int i = 1; i < data.length; i ++)
          if (data[i] > max) max = data[i];

      /* max is the largest element of the array data */
      ```

   2. It would be nice if a programming language would make construction of proofs like this fairly straightforward. Unfortunately, two characteristics found in many programming languages tend to make constructing proofs difficult.

      a. The goto statement complicates proofs, because one cannot be sure what preconditions apply to a statement if it can be reached in more than one way.

      b. The possibility of two variables being ALIASES for one another complicates proofs - e.g. under some circumstances the postcondition given below might not be valid:

         ```
         /* a == a0 && b == b0 */
         a ++;
         /* a == a0 + 1 && b == b0 */
         ```

In particular, if a and b are aliases, then the assertion b = b0 no longer holds.

3. To facilitate proofs, some languages do not have a goto statement (e.g. Java, which not only does not have the goto, but also makes goto a reserved word one cannot use!) and others have sufficient control structure flexibility to make its use almost always unnecessary. A few languages also have mechanisms to prevent aliasing from occurring (though none that we will study in this course.)

III. Criteria Relating to Performance

A. Last on our list - but not unimportant - are criteria relating to how the language performs.

B. First, we consider the performance of language translators. Ideally, the language should lend itself to FAST COMPILATION.

1. In general, the more complex the syntax of the language, the longer a program of a given length will take to compile. This was rather dramatically illustrated by the compilers we had on our PDP-11/70. Student projects in Pascal and FORTRAN would compile quite quickly. Programs written in BASIC-PLUS-TWO would take much longer, and COBOL programs of the same length would seem to take forever!

2. When developing large programs, it is nice to have a separate compilation facility that allows the program to be spread over several files. When a small change is made, only the affected file, and perhaps others that depend on it, need to be recompiled. Most current languages support this.

C. Also important in many cases (but not all) is that the compiler produce EFFICIENT OBJECT CODE.

1. In part, this is a matter of compiler technology; but some language features make this harder.

2. Of course, this goal conflicts with the goal of fast compilation. An optimizing compiler spends extra time during compilation to produce better object code. This is nice for production software, but is not as pleasant during program development.

3. Some languages are supported by two compiler versions - a fast "checkout" compiler that produces less than optimal code, but which can be used during debugging; and an optimizing compiler that is slower but produces production-quality code. Or, a single compiler may include a command line option to turn optimization on or off or even specify the degree of optimization desired - trading off compilation time for execution time.

Example: PL/I implementations included both a "checkout" compiler and an optimizing compiler.

Example: the gnu C and other compilers include a -O switch with possible values 0 (no optimization) or 1, 2, 3 (increasing degrees of optimization.)

(Unfortunately, sometimes the two compilers or the one compiler with different optimization settings don't process the same language constructs in exactly the same way. Of course, this is more of a compiler problem than a language problem per se)

D. Last - but by no means least - we consider the matter of MACHINE INDEPENDENCE or PORTABILITY.

1. One of the original reasons for adopting higher level languages was the desire to be able to move a program from one type of machine to another without rewriting it. To some extent, all higher-level languages achieve this goal; but some do much better than others.

2. Most important for portability is the existence of a well-defined and accepted language standard.

   a. Many languages have been standardized by formal bodies like ANSI or ISO. For others, the original report by the language author may serve as a standard - though not as strong of a one. Some languages have no clear-cut standard, though.

   b. A standard does not help much if different implementers of the language choose to go beyond the standard with various extensions, each in his own way. The classic example of this is BASIC. Despite the existence of an ANSI standard, no two implementations are the same, due to different extensions. (Actually, some also implement less than the standard.) Thus, it is important that the set of features contained in the standard be complete enough to help implementers resist the temptation to add incompatible extensions.

   c. In the case of Ada, the Department of Defense copyrighted the name Ada, in order to ensure that ALL implementations handle exactly the same language, thus facilitating portability. (In order to use the name Ada, a compiler must pass a validation test that ensures it handles the langauge exactly as specified.)

3. Standardization by itself is not enough, though, even when the standard is adhered to. Certain characteristics of the underlying machine have a way of showing up unavoidably in the implementation.

   a. For example, every machine has a basic word length which determines the range of integers that can be processed by regular machine instructions.

      i. Historically, microprocessor systems often used 16 bit integers, restricting the range of integers to -32768 .. 32767.

      ii. Many systems today use 32 bits, leading to integers ranging from -2 billion to +2 billion.

      iii. Still other systems use 64 bits, leading to integers ranging from -100 trillion to +100 trillion.

      iv. A program which relies on the range of integers available on one machine may not run correctly on another machine whose range of values is smaller.

      (A similar phenomenon arises with the range of values and precision of real numbers.)

b. Again, different machines use different coding schemes to represent characters, with ASCII, EBCDIC, and Unicode having wide use. This can lead to problems, as follows:

   i. In ASCII, the codes for alphabetic characters are contiguous, without any gaps. This is not true in EBCDIC. For example, the EBCDIC code for I is 201, while that for J is 209. A program that relies on the letters being contiguous - such as a cipher program - will fail if ported to an EBCDIC machine.

   ii. In ASCII, upper case letters have codes 32 less than corresponding lower case letters. On EBCDIC machines, their code is 64 greater! A program that converts between lower and upper case letters may have a problem here.

c. Java addresses these issues by stipulating as part of the standard that byte, short, int, and long are - respectively - 8, 16, 32 and 64 bit integers; and that characters are represented internally using Unicode.