1.1 Programming language design

There are two very important principles in PL design.

Leave it to the compiler: Design the language such that the compiler does most of the problem solving, testing, etc. This needs robust language features.

Make the progammer say once, at most: If the compiler cannot solve the complete problem, make the programmer specify all related information at one place rather than in a diffuse fashion throughout the program.

The following some formal design/evaluation criteria.

expressive power: Conside the case of strings in Fortran, C, and Perl. In Fortran, strings have very limited support. In C, one can create strings of any fixed size. Perl supports strings of varying sizes and also pattern matching. So Perl is highly expressive, as far as strings are concerned.

As another example, consider the use of classes in C, C++, and Java. C does not have the notion of classes. C++ supports classes. In Java, one cannot program without classes. This restriction facilitates disciplined programming, code reuse, etc.

So a PL should be sufficiently expressive.

well-defined syntax and semantics: There should be no implementation dependence. Some implementation dependent issues are: variable initialization, result of divide by zero, etc.

reliability: This is the ability to detect syntactic and logical errors. Multiple occurrences of the same information reduces reliability. For example, in C, one can write the follwing code.

```
#include "myinclude.h"

myfun(myargs);
```

where the declaration for `myfun` is contained in `myinclude.h`. The code itself for `myfun` might be in some independent object file. It is difficult to check whether all the different pieces of information are coherent. Java does not have this problem. There is a single `.class` file which holds all the information.

Lexical coherence refers to the notion that related pieces of information should be lexically close. In Pascal, the variable declaration and use might be separated by intervening procedure definitions.

```
program longDist(input, output);
var result: int;
      (*        HUGE PROCEDURE *)
begin
     result := 10.5; (* is result int or real? *)
end.
```

In C++ and Java, variables can be defined anywhere, even inside `for` loops, This greatly enhances lexical coherence.

```
for (int i=0; i < max; i++) {   // declare i
     /*   something done */
}
// i becomes undefine here
```

fast translation: Many languages have features that make the compiler's job simiple. In particular, most languages try to make the compiler "single pass". That is, look at the source once and produce code. C uses forward declarations to achieve this.

```
example in C
      int f2(int);        // forward declaration

      int f1(int x) {
          f2(x);  // function call
      }

      int f2(int x) {  // function definition
          /* some code */
      }
```

But forward declarations is antagonistic to lexical coherence.

efficient object code: The principle here is to make the programmer supply as much information as possible. The compiler uses this information to produce efficient object code. For example, the increment operator, `++`, tells the compiler that the variable is incremented by one. Most processors have a single instruction for increment by one. This notation enables the compiler to use such instructions.

orthogonality: Orthogonality means two things: non-interacting features and irredundant features.

An example of interacting features can be found in Perl. Perl has many features: automatic string–number conversion, context sensitivity, etc. The result is that "0.000" evaluates to true while "0.000" + 0 evaluates to false. This result is counter intuitive. You can execute following script and check for yourself.

```
#!/usr/bin/perl

if ("0.0") { print "exp1 is true\n";}
if ("0.0"+0) { print "exp2 is true\n";}
```

Many languages have redundant features. For example, there two ways to represent comments in C++: `//` and `/* ... */`. In this case, redundancy actually helps the programmer.

Machine independence: Most languages have machine dependent primitive types. Java is an exception.

consistency with commonly used notions: This is also called the "law of least surprise". One example is the Perl example mentioned above. Another example is: 25 + 1/3 evaluates to 5.333333 in PL/1.

subsets: The programmer should be expected to know only what is required for the task and not other features. This means that "what you don't know won't hurt you".

1.2 Horowitz's 10 steps to design a language

1. Choose an application area
2. Make the design committee as small as possible
3. Choose some precise design goals
4. Release version one to small set of people
5. Revise language definition
6. Build a prototype compiler
7. Revise language definition again
8. Write the manual
9. Write a good compiler and distribute it
10. Write primers