Language design principles

The language design problem

- **Language design is difficult, and success is hard to predict:**
  - **Pascal a success, Modula-2 a failure**
  - **Algol60 a success, Algol68 a failure**
  - **FORTRAN a success, PL/I a failure**
- **Conflicting advice**

Efficiency

- The "first" goal (FORTRAN): *execution efficiency*.
- Still an important goal in some settings (C++, C).
- Many other criteria can be interpreted from the point of view of efficiency:
  - programming efficiency: writability or expressiveness (ability to express complex processes and structures)
  - reliability (security).
  - maintenance efficiency: readability.

(saw this as a goal for first time in Cobol)

Other kinds of efficiency

- efficiency of execution (optimizable)
- efficiency of translation. Are there features which are extremely difficult to check at compile time (or even run time)? e.g. Alogol – prohibits assignment to dangling pointers
- Implementability (cost of writing translator)

Features that aid efficiency of execution

- **Static data types allow efficient allocation and access.**
- **Manual memory management avoids overhead of "garbage collection".**
- **Simple semantics allow for simple structure of running programs (simple environments - Chapter 8).**

Note conflicts with efficiency

- **Writability, expressiveness: no static data types (variables can hold anything, no need for type declarations). [harder to maintain]**
- **Reliability, writability, readability: automatic memory management (no need for pointers). [runs slower]**
- **Expressiveness, writability, readability: more complex semantics, allowing greater abstraction. [harder to translate]**

Internal consistency of a language design: Regularity

- **Regularity is a measure of how well a language integrates its features, so that there are no unusual restrictions, interactions, or behavior. Easy to remember.**
- **Regularity issues can often be placed in subcategories:**
  - **Generality: are constructs general enough? (Or too general?)**
  - **Orthogonality: are there strange interactions?**
  - **Uniformity: Do similar things look the same, and do different things look different?**

Generality deficiencies

- In pascal, procedures can be passed as parameters, but no procedure variable.
- Pascal has no variable length arrays –length is defined as part of definition (even when parameter)

Orthogonality: independence

- **Not context sensitive**

Seems similar to "generality" but more of an "odd" decision rather than a limitation.

- **For example, if I buy a sweater, I may have the following choices:**
  - **short sleeve, long sleeve, or sleeveless**
  - **small, medium, or large**
  - **red, green, or blue**

Limitations to sweater example:

- **If it is not possible to get sleeveless sweaters, that may be a lack of generality.**
- **If any combination of any attributes can be used together, it is orthogonal.**
- **If red sweaters cannot be purchased in a small size, but other sweaters can, it is non-orthogonal**

Orthogonality

a relatively small set of primitive constructs can be combined in a relatively small number of ways. Every possible combination is legal.

For example - in IBM assembly language there are different instructions for adding memory to register or register to register (non-orthogonal).

In Vax, a single add instruction can have arbitrary operands.

Closely related to simplicity - the more orthogonal, the fewer rules to remember.

For examples of non-orthogonality consider C++:

- **We can convert from integer to float by simply assigning a float to an integer, but not vice versa. (not a question of ability to do – generality, but of the way it is done)**
- **Arrays are pass by reference while integers are pass by value.**
- **A switch statement works with integers, characters, or enumerated types, but not doubles or Strings.**

Regularity examples from C++

- Functions are not general: there are no local functions (simplicity of environment).
- Declarations are not uniform: data declarations must be followed by a semicolon, function declarations must not.
- Lots of ways to increment – lack of uniformity (++i, i++, i = i+1)
- i=j and i==j look the same, but are different. Lack of uniformity

What about Java?

- **Are function declarations non-general?**
  - **There are no functions, so a non-issue. (Well, what about static methods?)**
- **Are class declarations non-general?**
  - **No multiple inheritance (but there is a reason: complexity of environment).**
  - **Java has a good replacement: interface inheritance.**
- **Do declarations require semicolons?**
  - **Local variables do, but is that an issue? (Not really - they look like statements.)**

Java regularity, continued

- **Are some parameters references, others not?**
  - **Yes: objects are references, simple data are copies.**
  - **This is a result of the non-uniformity of data in Java, in which not every piece of data is an object.**
  - **The reason is efficiency: simple data have fast access.**
- **What is the worst non-regularity in Java?**
  - **My vote: arrays. But there are excuses.**

Other design principles

Simplicity: make things as simple as possible, but not simpler (Einstein). (Pascal, C)

- **We can make things so simple that it doesn't work well – no string handling, no reasonable I/0**
- **Can be cumbersome to use or inefficient.**

Other design principles

Expressiveness: make it possible to express conceptual abstractions directly and simply. (Scheme)

- **Helps you to think about the problem.**
- **Perl, for example, allows you to return multiple arguments:**

```
($a,$b) = swap($a,$b);
```

Other design principles

- **Extensibility: allow the programmer to extend the language in various ways. (Scheme, C++)**

  Types, operators

- **Security: programs cannot do unexpected damage. (Java)**
  - **discourages errors**
  - **allows errors to be discovered**
  - **type checking**

Other design principles (cont.)

- **Preciseness: having a definition that can answer programmers and implementors questions. (Most languages today, but only one has a mathematical definition: ML).**

  If it isn't clear, there will be differences.

Example: Declaration in local scope (for loop) unknown/known after exit

Example: implementation of switch statement

Example: constants – expressions or not?

Example: how much accuracy of float?

Other design principles (cont.)

- **Machine-independence: should run the same on any machine. (Java- big effort)**
- **Consistent with accepted notations – easy to learn and understand for experienced programmers (Most languages today, but not Smalltalk & Perl)**
- **Restrictability: a programmer can program effectively in a subset of the full language. (C++: avoids runtime penalties)**

Wikipedia moment:

- Syntactic sugar is a term coined by Peter J. Landin for additions to the syntax of a computer language that do not affect its expressiveness but make it "sweeter" for humans to use. Syntactic sugar gives the programmer (designer, in the case of specification computer languages) an alternative way of coding (specifying) that is more practical, either by being more succinct or more like some familiar notation.

C++ case study

- **Thanks to Bjarne Stroustrup, C++ is not only a great success story, but also the best-documented language development effort in history:**
  - **1997: The C++ Programming Language, 3rd Edition (Addison-Wesley).**
  - **1994: The Design and Evolution of C++ (Addison-Wesley).**
  - **1993: A History of C++ 1979-1991, SIGPLAN Notices 28(3).**

Major C++ design goals

- OO features: class, inheritance
- Strong type checking for better compile-time debugging
- Efficient execution
- Portable
- Easy to implement
- Good interfaces with other tools

Supplemental C++ design goals

- C compatibility (but not an absolute goal: no *gratuitous* incompatibilities)
- Incremental development based on experience.
- No runtime penalty for unused features.
- Multiparadigm
- Stronger type checking than C
- Learnable in stages
- Compatibility with other languages and systems

C++ design errors

- **Too big?**
  - **C++ programs can be hard to understand and debug**
  - **Not easy to implement**
  - **Defended by Stroustrup: multiparadigm features are worthwhile**
- **No standard library until late (and even then lacking major features)**
  - **Stroustrup agrees this has been a major problem**