Language Design Criteria

- Efficiency
- Regularity
- Simplicity
- Expressiveness
- Machine independence
- Security
- Consistency
- Preciseness
- Extensibility/Restrictability
Efficiency (Cost)

- Execution
- Translation
- Implementability
- Programming
- Maintenance
Efficiency in Execution

- Fortran
  - close to hardware
  - minimizes run-time overhead
    - no recursion
    - no dynamic memory alloc
    - no array bounds checking
- C++ allows statically typed vars
Efficiency in Translation

- Pascal, C: no forward references, one-pass compiler
- C++ needs two passes
- Algol 68: prohibition against dangling references
- skip error checking – tradeoff with reliability
Efficiency in Implementability

- Algol 60 stack-based structure not understood
- ML awaited unification algorithm for type inference
- Ada
  - language is large and complex
  - compilers must be certified
- Algol 68
  - orthogonality, generality difficult
  - prohibition on dangling references
Efficiency in Programming

- cost of training programmers
- cost of writing programs
- Lisp and Prolog: concise syntax
- depends on expressiveness, abstraction mechanisms, writability
Efficiency in Maintenance

• big concern
• ease with which errors can be found and fixed
• ease with which new features can be added
• depends on readability
Regularity

- How well features are integrated
- Restrictions on the use of features
- Strange interactions among constructs
- Surprising behavior

Subdivisions:
  - Generality
  - Orthogonality
  - Uniformity
Generality

- avoids special cases in the use of constructs
- combines closely related constructs into a single more general one
- avoidance of special cases in availability or use of constructs
- combination of closely related constructs into a single, more general one
- restrictions that do not depend on context are non-generalities
Example: Functions and Procedures

- * C: no nesting, but variables and return values
- * Pascal: nesting but no procedure variables
- * Scheme: fully general: parameters, variables, return values, nesting
Example: Parameter Passing

* Fortran
  · one parameter passing mechanism
  · call by reference
* C, Algol 68
  · one parameter passing mechanism
  · call by value
  · achieves generality by pointers
* Pascal, Ada – multiple mechanisms
Example: Arrays

* variable or fixed length
* both – more general
* Pascal – fixed length
* C, Ada, C++ – both
* Fortran, Modula 2
  • fixed in declarations
  • variable as parameters
Example: Constants

* Fortran – no named constants
* Pascal – cannot be expressions
  \texttt{const C1 = 10*20} not allowed
* Modula 2 no function calls
  \texttt{CONST PI =4.0*ATAN(1.0); not allowed}
* Ada – fully general
Orthogonality

- constructs should not behave differently in different contexts
- constructs can be combined in any meaningful way
- interaction of constructs should not cause unexpected restrictions or behavior
- context dependent restrictions are non-orthogonalities
- Best adherence to orthogonality: Algol 68
Example: Return Types

• Pascal: only scalar or pointer types
• C: all types except arrays and functions
• Ada, most functional languages: all types
Example: Declarations

• C: at start of a block
• C++: anywhere in a block, but before use
Example: Assignments

- Cannot assign to Pascal file pointers
- in Modula 2, if $|str1| > |str2|$ then str2 can be assigned to str1 but not the reverse
Uniformity

• things that are similar should look similar

• things that are different should look different

Example: Classes and Functions

• C++ class, semicolon required:
  class A {...};

• C++ function, semicolon forbidden:
  int f () {...}
Example: Pascal Control Structures

- repeat: no begin ... end
- while and if use begin ... end

Example: Termination vs. Separation

- terminating semicolon: \{ s1;s2; \}
- separating semicolon:
  for (k=1; k < n; k++)...
Example: Operators

- \( = \) vs \( == \)
- \& vs \&\&

Example: Return Statement

- Pascal and Fortran have return statements that look like assignments
Simplicity

- Pascal, Basic and C are simple
- difficult to achieve
- not regularity – Algol 68
- not small number of constructs
- can make use complex
- the trade-off between orthogonality and simplicity is a central decision in language design
Expressiveness

• Ease with which language can express complex processes and structures

• example: recursion (Lisp, Algol 60)
• in conflict with simplicity
  – three loops (while, for, repeat)
    expressive but not simple

• may not be readable
  – string copy from K&R
    while(*s++ = *t++)
Machine Independence

- hard to impossible
- example: floating point numbers
- isolate and identify dependencies
Security

- discourage programming errors
- allows errors to be discovered and reported
- features that support security
  - typing, type checking
  - variable declarations
  - subscript range checking
- at odds with expressiveness and conciseness
Consistency with Accepted Notations and Conventions

• should be easy to learn by experienced programmers

• standard concepts should be recognizable
  fns, vars, if, while
Preciseness

• precise definition
  manual or standards document

• benefit: predictable translator

• should be comprehensible
  Algol 68 used W-grammar
Extensibility/Restrictability

- extensibility examples:
  user-defined types, fns from libraries
- restrict to a useful subset