Programming Languages

Course Summary

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Programming Language Selection and Design

- Choice of best programming language depends on:
  1. Intended application.
  2. Intended programmers.
  3. Intended computers to be used.
- And many general principles!

Issue for Choosing

- Support for:
  1. Abstraction: Can programs be written to hide how they are implemented?
  2. Programming in the Large: Can 100 people work together on a program?
  3. Software reuse: Can you use old or produce new modules which can be used in other programs?

Criteria for Language Design

- Writeability
- Readability — Modifiability and Maintenance
- Reliability
- Fast translation
- Efficient object code
- Machine independence

Simplicity

- The language should be easy to master
- Bad examples: PL/I and Ada — don't throw in kitchen sink
- If too complex, danger of unexpected interactions, subsetting and (if redundant) development of dialects.
- Simple conceptual model of semantics.

  Can go too far: Basic, LISP syntax.

  "Everything should be as simple as possible, but not simpler" Einstein

Uniformity & Expressiveness

- Uniformity
  - Similar syntax should imply similar semantics.
  - Bad example: subscript vs function definition in FORTRAN.
  - Good example: Instance variable vs parameterless function in Eiffel
- Expressiveness
  - Easy to express programs. E.g. support for abstraction
  - Success depends on application

Orthogonality and Generality

- Few basic features — free from interactions. No special restrictions.
- Example: Problems with Pascal: Function can only return simple type, constants only defined for built-in basic types, enumerated types cannot be read or written, etc.
- Algol 68 carries to extremes, everything has value.
- Smalltalk, Eiffel, other pure object-oriented languages do it well.

Clear Syntactic and Semantic Description

- Formal grammar, formal semantics helpful but also need well-written manual.
- Original presentation usually too vague. Refined.
- ALGOL 68 problem with redefining terms.

Readability — Modifiability and Maintenance

- Hoare considered more important than writeability
- Documentation:
  - self-documenting
  - commenting conventions: line vs stream oriented
- Lexical and syntactic conventions:
  - descriptive identifier names,
  - blocking of compound statements - if, fi versus begin, end
  - two-dimensional syntax, syntax directed editors.
- Not synonymous with wordiness:
  - COBOL not easier to read.
  - Importance depends on number of programmers.
  - APL — write-only.

Reliability

- Syntactic and logical (semantic) errors discouraged and easily discovered.
- Related to writeability, readability, and modifiability.
- Static checks
- Separate (but not independent) compilation
- Minor modifications should not require major changes.
- Clear semantics supporting verification.

  Problems: Ada out params, Pascal variant records, side-effects.
- Simplicity of compiler implementation.

Other Qualities

- Fast translation

  A language that is simple to parse for the compiler is also simple to parse for the human programmer, and that can only be an asset. — Wirth

- Efficient Object Code — So people will use program

- Machine independence — hard to achieve completely.

Hoare on Languages

- Base programming language on:
  1. a minimum number of independent concepts combined in a uniform manner.
  2. a comprehensive definition mechanism to provide the breadth.
  3. a small core language on which the extensions are based.
  4. a syntax chosen for its readability.
- Most dangerous approach to problems: add new features

  Trap PL/I, Ada, and now C++ fell into.

Principles from MacLennan

1. *Abstraction*: Avoid requiring something to be stated more than once; factor out the recurring pattern.
2. *Automation*: Automate mechanical, tedious, or error-prone activities. (Provide higher level features, e.g. DO loop)
3. *Defense in Depth*: If an error gets through one line of defense, then it should be caught by the next line of defense. (e.g. if incorrect but syntactically ok, then should fail type-checking: DO, varieties of GOTO in FORTRAN)
4. *Information Hiding*: Modules should be designed so that: (1) The user has all the information needed to use the module correctly, and nothing more. (2) The implementor has all the information needed to implement the module correctly, and nothing more.

5. *Labeling*: We should not require the user to know the absolute position of an item in a list. Instead, we should associate labels with any position that must be referenced elsewhere. (E.g., labelled case as opposed to unlabelled)
6. *Localized Cost*: A user should only pay for what he uses; avoid distributed costs. (Don't build in so much generality that simple things are expensive, e.g ALGOL DO, default parameters)
7. *Manifest Interface*: All interfaces must be apparent (manifest) in the syntax. (Clear how to use, no side-effects)
8. *Orthogonality*: Independent functions should be controlled by independent mechanisms. (Avoid dependencies, e.g. logic and control).
9. *Portability*: Avoid features or facilities that are dependent on a particular machine or a small class of machines.

10. *Preservation of Information*: The language should allow the representation of information that the user knows and that the compiler will need. (E.g. DO loops, subrange types — make it easier to optimize.)
11. *Regularity*: Regular rules, without exceptions, are easier to learn, use, describe, and implement.
12. *Security*: No program that violates the definition of the language, or its own intended structure, should escape detection.
13. *Simplicity*: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.
14. *Structure*: The static structure of a program should correspond in a simple way with the dynamic structure of the corresponding computations.

15. *Syntactic Consistency*: Things which look similar should be similar and things which look different should be different.

16. *Zero-One-Infinity*: The only reasonable numbers in programming language design are zero, one, and infinity. (Avoid special cases, E.g. compound commands in Pascal.)

Why This Course?

- Originally survey of programming languages: COBOL, FORTRAN, PL/I, LISP, APL, SNOBOL, etc. (languages for jobs course!).
- Get better understanding of principle features of programming languages.
- Explore design space of programming languages.
- Introduce different ways of thinking about programming reflected by different programming language paradigms.

Course Overview

1. Syntax (formal) and semantics (informal and formal) of common programming language concepts.

   Structure of compilers / interpreters. Importance of binding time. Variables: static vs. dynamic scoping, lifetime of variables, l-values vs. r-values.

2. Run-time structure of programming languages. Allocation of storage at run-time. Parameter passing mechanisms. Storage reclamation (explicit — under programmer control, automatic — garbage collection and reference counting).

3. Types in programming languages. Available types and their representation. Type structures in important representative languages.

4. Issues in type-checking: Static vs. dynamic type-checking. Type compatibility (structural vs. name equivalence). Holes in type systems (e.g. variant records). Problems with pointers.
5. Abstract data types and representation in programming languages.
6. Expressions: Order of evaluation (including short-circuit evaluation), functions, and side-effects
7. Control structures (including exception handling).
8. Polymorphism — implicit and explicit.
9. Programming in the large: Support from languages (information hiding, encapsulation, separate but not independent compilation, explicit importing and exporting of environments.) Top-down vs. object-based design.

10. Programming language paradigms:
    (a) Imperative: Assignment statements (including order of evaluation, side-effects, and aliasing) and control structures. Iterators. Procedures, coroutines.
    (b) Functional: Expanded expression constructs. Pattern matching and type inference. Higher-order functions. Lazy vs. eager languages. Recursively defined types. Implementation issues (efficiency and possibility of implementation on highly-parallel computers). Program verification. ML as an example.
    (c) Logic: Basis in mathematical logic — extract program from proof. Relational programming (and reversability of computations). Computation as backtracking + unification. Relation to resolution theorem proving. Separation of logic and control (sort of). Importance of ordering of clauses. Use of cut. Closed-world assumption and negation.

Limitations of Prolog as an implementation of logic programming (lack of soundness and completeness).

(d) Object-oriented: Origins in object-based languages supporting ADT's. Key features: objects and classes, message passing and dynamic method invocation, subclasses and inheritance, subtyping, etc. Support for abstractions and information-hiding. Object-oriented view of algorithms. Support for incremental modification through inheritance and subclasses. Difference between subtyping and inheritance. Type-checking problems in popular languages. Complications in semantics of inheritance. Loss of locality in inheritance.

11. Semantics: Operational, Axiomatic, and denotational semantics.

12. Evolution of programming languages. Problems which led to introduction of new constructs.

13. Criteria for language design: Writeability, readability, reliability, efficiency considerations, portability. How to design features and languages. How to choose a language.