Evaluating Languages

Prof. Evan Chang
Meeting 2, CSCI 3155, Fall 2009

Announcements

• Join the course moodle and introduce yourself. Be sure to upload a picture of yourself to your profile!
• Textbook available in CU Bookstore (thanks Dain)
• Assignment 1 out tonight, due Thu Sep 3 at 11:55pm
  - Find a partner, Post on general forum

Feedback

• Many wanted to get a better “feeling” for different languages (for various applications)
  - stay tuned

Perils of JavaSchools

• Comments?
  - “Know about hardware — pointers”
  - know abstraction layers — why things are hard
  - helps to understand the high-level
  - make low (incl. low-level) = learn lang
  - more easily
  - CS foundation
  - Foundations — ways of thinking
  - about computation

Evaluating Languages
Take-Home Message

- Languages are adopted to fill a void and may have little to do with “quality”
- No universally accepted metrics for good design
- There are some criteria and characteristics that help us evaluate languages
- Table 1.1 (in Sebesta) provides a starting point

Why so many languages?

- Lots of different applications
- Optimized to work well in those applications
- Superlanguage?
  - We use what their users used to
  - Disagreements (think syntax!)
  - Traditions in compiler + push
  - Diff in real — convincing reasons to exist

Opinion on Language Design

- Languages are adopted to fill a void
  - Enable a previously difficult/impossible application
  - Orthogonal to language design quality (almost)
- Training is the dominant adoption cost
  - Languages with many users are replaced rarely
  - But easy to start in a new niche.

So many languages

- Examples:
  - AI: symbolic computation (Lisp, Prolog)
  - Scientific Computing: high performance (Fortran)
  - Business: report generation (COBOL)
  - Systems Programming: low-level access (C)
  - Scripting (Perl, Python, TCL)
  - Distributed Systems: mobile computation (Java)
  - Web (PHP, Ruby, Perl, Javascript)
  - Special purpose languages: ...

Language Paradigms

- Imperative
  - “mutating value”
  - Fortran, Algol, Cobol, C, Pascal
- Functional
  - Lambda calculus, ML, Haskell
- Object-oriented
  - Smalltalk, Eiffel, Self, C++, Java, C#, Javascript
- Logic
  - Prolog
- Concurrent
  - CSP, dialects of the above languages
- Special purpose
  - TEX, PostScript, TrueType, sh, HTML, make

So many languages

- Examples:
  - AI: symbolic computation (Lisp, Prolog)
  - Scientific Computing: high performance (Fortran)
  - Business: report generation (COBOL)
  - Systems Programming: low-level access (C)
  - Scripting (Perl, Python, TCL)
  - Distributed Systems: mobile computation (Java)
  - Web (PHP, Ruby, Perl, Javascript)
  - Special purpose languages: ...

Language Paradigms

- Imperative
  - “mutating value”
  - Fortran, Algol, Cobol, C, Pascal
- Functional
  - Lambda calculus, ML, Haskell
- Object-oriented
  - Smalltalk, Eiffel, Self, C++, Java, C#, Javascript
- Logic
  - Prolog
- Concurrent
  - CSP, dialects of the above languages
- Special purpose
  - TEX, PostScript, TrueType, sh, HTML, make
What makes a good language?

• No universally accepted metrics for design

• "A good language is one people use"?
  - one that's useful
  - "what is good"

What are criteria for evaluating langs?

- Readability, Writability, Reliability

Cost
- Ops - Run-time efficiency
- Maintainability
- How long to learn + train
- Community availability + libraries + frameworks
- Portability

Designing good languages is hard

• Goals almost always conflict.

• Examples:
  - Safety checks cost something in either compilation or execution time.
  - Type systems restrict programming style in exchange for strong guarantees.

Story: Java Bytecode Subroutines

• Java bytecode programs contain subroutines (jsr) that run in the caller's stack frame (why?)

• jsr complicates the formal semantics of bytecodes
  - Several verifier bugs were in code implementing jsr
  - 30% of typing rules, 50% of soundness proof due to jsr

• It is not worth it:
  - In 650K lines of Java code, 230 subroutines, saving 2427 bytes, or 0.02%
  - 13 times more space could be saved by renaming the language back to Oak

What are some characteristics of languages?

• Sebesta, Table 1.1
  - Simplicity
  - Orthogonality
  - Data types and structures
  - Syntax considerations
  - Support for abstraction
  - Expressivity
  - Type checking
  - Exceptions
  - Restricted aliasing
Simplicity

- High-level, loops instead of goto
- How many constructs are there?
- Size of many sets
  - Modula-3, 50 pages
  - SML: 113 pages
  - C++: 657 pages
  - C: 43 pages (no libraries)
- Uniformity
- Easy to learn
- “Pitfalls” (less) - exceptions
- Can have too much

Orthogonality

- “Largeness” 
  - Features can combine “all ways” 
- Can combine arrays + pointers in all ways
- Control statements 
  - for (i=1; i<10; i++)
  - if
  - while (i<10) {
    ... 
  }
  - Very broad "goto"

Data types and structures

- Benefit: for different types
  - Booleans: flag = 1 (flag = 30?);
  - vs. flag = true
  - int compare (x, y);  
    return -1 if x < y
    return 1 if x > y
    return 0 if x = y
  - enum = let if 1 eq

Syntax considerations

- Preferences
  - for vs. for
  - end for vs. end for
  - if vs. if
  - static (multiple names)
  - “twostep” on whitespace

Support for abstraction

- Examples: iterator (enum a data set)
- “Hide some details” 
  - Process abstraction vs. data abstraction 
  - Functions/procedures, large, sets, collections
  - Modules
  - “Hiding details that are not relevant to a user”

Expressivity

- (Writeability)
- Not Readability
- Ability to get your ideas out
- “Naturally” (Quality)
- Number of ways to do something
- Do a lot with little code
- Conflicts: simplicity, orthogonality
- Large, hard-coded expressivity (C++)
- “Automatic features” 
  - high-level
  - up to redefine key features (Python, C++, etc.)
Type checking

- What is it?
  - It checks whether operations are applicable to "well-typed" things
  - Prevents errors: "2+3" x 4
- Restricts programs you can write
- Dynamic vs. static typing
  - Check at runtime/compile at compile-time
- Strong vs. weak

Exception handling

- in reliability

Restricted aliasing

- What is aliasing?
  - Two names for the same thing
  - Java
    ```java
    x = y;
    x += t;
    (y, t changes)
    ```

For Next Time

- Reading
- Online discussion forum
- Start homework assignment 1