CSE 130
Programming Language Principles & Paradigms

Thomas A. Powell
tpowell@pint.com
Housekeeping

• Syllabus review
• Direct class page link
  http://www.pint.com/classes/cse130
• Updated syllabus, notes, homework
  specifications, etc. will always be posted there
• Complete the student survey for end of class
• History of this class
• My goals for the future of this class
• Standard academic issues (esp. cheating)
What is the best/most powerful PL?

- Java
- C
- C++
- C#
- Pascal
- Visual Basic
- Lisp
- Scheme
- ML
- Ruby
- Modula-2
- JavaScript
- Fortran
- Cobol
- Haskell
- Oberon
- Prolog
- PHP
- Perl
- Python
- Ada
Clear Winner?

- Obviously there is no “best” language for all situations, the best language might depend on a variety of things:
  - Type of program
  - Reason program is built
  - Size and scope of program
  - Programmer familiarity
  - Time available
  - Cost
  - Legacy and external issues

- Power is a loaded word, try appropriate

*Real CS professionals don’t get religious about their languages, they get pragmatic and understand the trade-offs in their language choices. There are ALWAYS trade-offs.*
Reasons to Study Programming Language Theory

- Improves ability to express ideas in primary language
  - Languages influence the way you think and approach problems
  - As you study new language features it may help you utilize or extend your own language skills
  - Simulate a useful feature in your primary language
  - Personal example from JS and C (&, |)
Reasons to Study Programming Language Theory

• Improved background for choosing appropriate languages
  - Helps you understand the trade-offs in languages rather than immediately assuming your known language is the best one for the job
  - Gives you the background to communicate to others in a logical way the choices necessary to make an informed language decision
Reasons to Study Programming Language Theory

• ** Increased ability to learn new languages **
  - There is significant similarity in the constructs provided by languages so that learning a new language is often just a matter of syntax.
    • Selection (if and case)
    • Loops (while, for, do)
    • Jumps (goto, break, continue)
    • Data types (strict or loose, int, char, string, object)
    • Functions

• ** Obvious Question: What then is the Latin of programming languages? **
Reasons to Study Programming Language Theory

• Helps you understand the significance of implementation
  - Most things don’t happen by chance, there is often a reason behind the way a language was built.
  - Some implementation issues are obviously related to technology such as hardware or operating system influences.
  - Many aspects of PLs are related to softer issues.
    • Who built it
    • How/why the language was promoted
    • Target programmer and their understanding of the value of certain constructs
    • State of the industry
      - Does it need the language? Is it ready for the language?
Reasons to Study Programming Language Theory

• Increased ability to design new languages
  - You probably will be designing a language of some sort sometime in your career
    • Unlikely a full blown programming language, but maybe an XML schema or full markup language DTD, a mini-scripting language for controlling a system, a configuration file language to control software, a simple API/language for data interchange, and so on.
  - Just because you can, it doesn’t mean you should

Real language value is often very much related to number of people using it.
Reasons to Study Programming Language Theory

- Other reasons
  - Sebesta promotes “advancement of computing”
  - Agree but the point is not limited to just programming language, but Computer Science in general.
  - Extremely useful for understanding compilers class!
Programming Domains

• Scientific applications
  - Focus on calculations
    • Fortran still alive and kicking

• Business applications
  - Reporting as well as calculations
    • Cobol, reporting languages (e.g. Crystal Reports), scripting environments of business systems like SAP, Siebel, etc.

• Artificial intelligence
  - Natural language, string manipulation, and logic needs
    • LISP family (Common Lisp, Scheme, ML), Prolog
Programming Domains

- **Systems programming**
  - Speed! Safety can be a problem
    - Machine level -> assembly -> C

- **Scripting languages**
  - Little code, lots of value
  - Often for non-programmers
  - Generally domain specific
  - Usually interpreted
    - JavaScript, Excel macros, sh, csh, awk, etc.

- **Sebesta breaks off Web programming in this edition.**
Programming Domains

• Yes Web programming seems different but how related to PLs
  - Network issues
  - Cross platform possibilities
  - Security or lack of
  - Type of developer (designer not programmer?) - environmental
  - The Diverse Programming Stack
    • Java, ASP.NET/C#, JavaScript, Server-side scripting environments (PHP, Classic ASP, CFM), Flash, Ruby, Perl, Python, etc.
  - Markup language intersection is troubling and interesting
    • What do you make of XSL
    • Ajax or DHTML? - This is a how used and a stew idea
    • How about CFML? <CF_IF> vs. if
Language Evaluation Criteria: Readability

- Readability describes the ease of which programs can be read and understood.

- It is probably the single most important criteria for programming language evaluation.

- It is so important because it significantly effects the maintainability of code which has in turn been a major cost for programs. In some sense this suggests that ultimately everything is about costs once you get beyond the theory of it.

- **Factors effecting readability:**
  - Overall simplicity - clearly too many features is bad, plain and simple more to learn
  - Multiplicity of ways of doing the same thing is generally bad
    - Not a real good example but illustrates the problem
      - `count = count + 1; count += 1; count++; ++count;`
    - Other coder uses a different coding style/dialect than you, can you read it easily?
  - Operator overloading can be trouble
    - `5 + 6 “test” + “it” “test” + 5` - might seem ok
    - “4” * 5 = “44444” or 20 or error?
    - `[5, 6, 1] + [1, 3, 4] = [6, 9, 5]` or `[5, 6, 1, 1, 3, 4]` or 20?

*Of course too simple can be just as much trouble--isn’t machine languages just 0 and 1! Is binary highly readable? Depends on who you ask! (CPUs or humans)*

- **Orthogonality** - small number of primitive constructs combined in a relatively small number of ways and everything can be combined with everything else.

- A lack of orthogonality is when you have exceptions to the rules
  - Some ESL folks might say this is English’s problem

- Orthogonality is supposed to make the language easy to learn and read, but in practice this doesn’t always seem to be the case.

- **Other factors**
  - Control statements
    - Watching program flow with jumps
  - Data types and structures
    - Not enough structures leads to work-arounds which can reduce clarity
    - No Boolean just use 0 and 1!
  - Syntax Considerations
    - Identifier length and form - two short equals bad variable names
    - Common Fortran complaint coming back on the Web due to download and intellectual property concerns
    - Block structure (end vs. end-if and end-loop)
    - Reserved words
    - Meaning of construct depending on location and use
      - Ex: In JavaScript the var statement
Language Evaluation Criteria: Writability

- **Writability** is the measure of how easily a language can be used to create programs for a given domain.
  - Be careful not to compare things which shouldn’t be. For example, in Rebol you can fetch and parse Web pages with a single command or two, try that in C or even Java.
  - Simplicity aids writability (less constructs = less mistakes)
  - Orthogonality may not help writability and exchange easy to deal with syntax errors with significant logic errors
  - Abstraction which allows you to define and reuse code is a key aspect of writability
    - Two forms of abstraction: process (e.g. procedures and function) and data (e.g. data structures like trees, arrays, etc.)
  - Expressiveness aids writability by make it convenient and easy to specify things (e.g. count = count + 1 vs. count++)
  - Clear tension between readability and writability
# Summary of Language Evaluation Criteria

**Table 1.1** Language evaluation criteria and the characteristics that affect them.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Readability</th>
<th>Writability</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplicity/orthogonality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control structures</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Data types and structures</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Syntax design</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Support for abstraction</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Expressivity</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Type checking</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Exception handling</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Restricted aliasing</td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
Language Evaluation Criteria: Reliability

- Reliable programs work under all conditions
  - I am sure you hope the airline autopilot software doesn’t crash!
- Type checking aids reliability
  - Static types tend to be safer and easier to check for than dynamic types
- Exception handling helps us deal with problems when they do occur
  - Not every language has adequate exception handling features
  - Argue that a Web application should have lots of exception handling particularly to deal with client issues, security and network effects
- Aliasing (when two or more things point to the same memory location) has been shown to be very dangerous
- The better written and supported the code the more reliable it tends to be, thus reliability is also a function of readability and writability
Language Evaluation Criteria: Cost

- Cost is in some ways the bottom line and is influenced by a variety of factors including:
  - Training of programmers
  - Cost/Time of writing the actual program
  - Compiler cost (time and development environment cost)
  - Execution cost (deployment environment cost)
  - Bugs and reliability issues
  - Maintenance
Evaluating Programming Languages

• There are other possibilities for language evaluation including:
  - Generality
  - Portability
  - Popularity

• Of course the way you evaluate a language is heavily influenced by the point of view and background of the evaluator.
Language Design Trade-offs

- Many of the aspects of programming language both theory and practice seem to be in conflict
- Even at the theory level there are issues
  - Reliability vs. Cost of execution
    (e.g. range checking and exception handling)
  - Writability vs. readability
    • Perl’s many ways to do the same thing
  - Flexibility vs. safety

A good PL designer (and programmer for that matter) understands trade-offs and makes decisions rather than skirt them.
Biggest Language Influence

- The hardware really does influence the software
- The standard computer architecture (von Neumann machine) pretty much dominates language design
  - Variables represent memory values
  - Assignments operate on memory values
  - Sequential execution of instructions
- Most languages fall into this *imperative* class of languages
- Some languages like functional or logic languages appear not to follow this form at least from the programmers point of view, but consider their run time environments actual do. **At the end of the day you have to execute on the standard architecture.**
The Evolution of Languages and Methodologies

• How people need to use languages effected the design of programming languages and associated programming paradigms
  - 1950s and early 1960s
    • Simple applications; scientific and big business focus
    • Hardware big $$$ thus a worry about machine efficiency
  - Late 1960s
    • Less worry about machine efficiency and more on people efficiency
    • Better control structures and improved readability, but still little methodology
  - Mid- late 1970s
    • Worry about reuse and maintenance
    • Rise of structured programming and data abstraction
The Evolution of Languages and Methodologies

- Mid-late 1980s
  - Rising complexity and costs
  - Introduction of Object-oriented programming
- Late 1990s - 2000s
  - The Internet (data focus, network issues, interoperability) = back to the 60s?

- Overlooked by Sebesta the people issue, who is doing the programming and what is their background
  - Often PhD in the 1960s, and today many coders have absolutely no formal background
  - Language design has sometimes embraced that fact and sometimes flown in the face of it
Programming Language Categories

- **Imperative**
  - Standard procedural languages (e.g. Pascal, C)
- **Functional**
  - No assignments, tons of recursion, and less focus on ordering (e.g. Lisp or Scheme)
- **Logic**
  - Tell the computer what to do but not necessarily how to do it (e.g. Prolog)
- **Object-oriented**
  - Closely related to imperative (e.g. C++, Java)
  - Question inclusion in categorization, maybe a flavor or even a methodology!
- **Concurrent / Network**
- **Scripting or Very High Level?**
- **Markup Hybrids?**
Imperative Programming Example

function gcd(u, v: in integer) return integer is
    y, t, z: integer;
begin
    z := u;
    x := v;
    loop
        exit when y = 0;
        t := y;
        y := z mod y;
        z := t;
    end loop
    return z;
end gcd;

• This was written in Ada just to show a potentially unfamiliar imperative style of code.
Functional Programming Example

(define (gcd u v)
  (if (= v 0) u
      (gcd v (modulo u v)))))

• This example was written in Scheme, a dialect of Lisp.
• Real functional languages are sometimes dubbed applicative because of the reliance on the application of function calls on values.
Logic Programming Example

gcd(U, V, U) :- V = 0

gcd(U, V, X):- not (V = 0)
    Y is U mod V,
    gcd (V, Y, X)

• This example was written in Prolog.
• Logic programming is sometimes called *declarative* programming because you declare or make assertions, but no execution sequence is specified.
public class IntWithGcd
{
    public IntWithGcd( int val ) { value = val; }
    public int intValue() { return value; }
    public int gcd ( int v )
    {
        int z = value;
        int y = v;
        while ( y != 0)
        {
            int t = y;
            y = z % y;
            z = t;
        }
        return z;
    }
    private int value;
}

• Example written in Java. Note complexity of OOP (or other acronym)
Purity of Language Distinctions

- Given the functional example, you might also try to rewrite the recursive style in C or some other language as well.
  - Some languages are more forceful in style than others, professor prefers those languages which are adaptive to the diversity of programmers
- You also might be able to apply objects to a functional language or even do an object oriented style of coding in a non-OOP language like Pascal.
- Interestingly enough it often turns out that languages are not pure and styles of coding are also important.
Language Implementation

• There are three possible approaches to translating human readable code to machine code:
  1. Compilation
     • Fast speed of execution, optimization possible
     • Slow for development, difficulty dealing with runtime errors
  2. Interpretation
     • Slow execution
     • Fast development and easy run-time error handling
  3. Hybrid
     • Portability possibility
     • A compromise? Does it satisfy?
Compilation

Source program

Lexical analyzer

Lexical units

Syntax analyzer

Parse trees

Symbol table

Intermediate code generator (and semantic analyzer)

Optimization (optional)

Intermediate code

Code generator

Machine language

Input data

Computer

Results
Interpretation

Source program

Interpreter

Input data

Results
Hybrid

Diagram:

1. Source program
2. Lexical analyzer
   - Lexical units
3. Syntax analyzer
   - Parse trees
4. Intermediate code generator
   - Intermediate code
5. Interpreter
   - Input data
6. Results
Hybrid Language Implementation Details

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called Java Virtual Machine)
  - Many Web environments have a byte code intermediate form often used in acceleration of server-parsed Web languages like PHP
Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
Preprocessors

- A preprocessor is a program that processes a program immediately before a program is compiled
  - Basically a macro expander
  - Example: `#define` and `#include` in C
- Preprocessors can be very useful but can also wreck havoc particularly on readability and safety if used in a bad way
Host Environment

- In reality it isn’t quite so simple, in order for a program to work it needs a lot of help from the operating system and in some cases the a potential host application.
  - Think of a language like JavaScript without a browser and associated object model!

- Usually the support to run the translated program is added in by the run-time environment or by linking associated system calls in automatically for you.
Layered Interface for Languages

Figure 1.2
Layered Interface of virtual computers, provided by a typical computer system.
Programming Environments

- In many cases the environment for programming is difficult to separate from the language and one influences the other.
- Programming environments range from simple (UNIX, emacs, cc, link) to complex graphical IDEs like Visual Studio.
- Programmer productivity seems to be better in complex integrated environments that rid developers of common system and interface chores and allow them to focus on programming logic.
Summary

• The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier

• Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost

• Major influences on language design have been machine architecture and software development methodologies

• The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation