COS 301

Programming Languages

Preliminaries

Topics

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Why Study Concepts of Programming Languages?

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Why Study Concepts of Programming Languages?

- To improve expressiveness
  - Expressiveness increases with vocabulary
  - Ability to visualize solutions increases with new programming constructs and depth of abstraction
  - If the only tool you know is a hammer, the only solutions you can see involve nails
  - Even if a language does not directly support a particular abstract feature (e.g. associative array) you can simulate it as part of solution

Gain improved background for choosing appropriate languages

- A fundamental result of theory is that all languages are equivalent in computing power
- Anything that can be computed in Java or Python can also be computed in BASIC, COBOL or FORTRAN
  - But some languages are better in terms of expressiveness for a given problem
  - Learning just one or two languages in great depth can be hazardous to your career
- Let's consider some problems…

Improve your ability to learn new languages

- First language or two are difficult and time-consuming to learn
- These languages also constrict your view of languages
  - As you learn the abstract concepts underlying programming languages you acquire a framework for understanding a new language
  - Eventually you can read many unfamiliar languages with little or minimal effort
  - Current trend is to combine relatively simple language with a huge library. Why re-invent the wheel?

The Tiobe Index

- See http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
- Counts top languages by counting hits of the most popular search engines
- See http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm

Gain a better understanding of significance of implementation

- Programming languages provide a virtual machine with which we express problems and their solutions.
- The virtual machine has to execute on a real one
- Understanding the implementation of a programming language helps to
  - Achieve more efficient problem solutions
  - Avoid subtle bugs caused by particular language implementations

Gain better use of current languages

- As languages evolve, they become large and complex compared to earlier languages
- Many modern languages are multi-paradigm languages – procedural, functional, object oriented
- A broad understanding of concepts and paradigms allows you to better leverage your knowledge of current languages

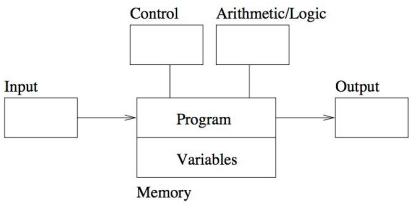
Programming Paradigms

- A programming paradigm is a pattern of problem-solving thought that underlies a particular genre of programs and languages.
- Many people classify languages into four main paradigms:
  - Imperative (Procedural)
  - Object-oriented
  - Functional
  - Logic (declarative)

Imperative or Procedural Paradigm

- The oldest programming model
- Follows the classic von Neumann-Eckert model:
  - Program and data are indistinguishable in memory
  - Program = a sequence of commands
  - State = values of all variables when program runs
- Large programs use procedural abstraction
- Language has assignments, loops, conditionals, procedure and functional calls, control flow
- Example imperative languages:
- Cobol, Fortran, C, Ada, Perl, ...

A Von Neumann Computer



Object Oriented (OO) Paradigm

- An OO Program is a collection of objects that interact by passing messages that transform the state.
- Concepts of OOP:
  - Sending Messages
  - Inheritance
  - Polymorphism
- Example OO languages:
  - SmallItalk, Java, C++, C#, and Python
- We will look at Smalltalk in detail later. Languages such as Java and C++ are OOP concepts grafted onto a procedural core

Functional Paradigm

- Functional programming models a computation as a collection of mathematical functions.
  - Input = domain
  - Output = range
- Functional languages are characterized by:
  - Functional composition
  - Recursion
  - Conditional evaluation
- Example functional languages:
  - Lisp, Scheme, ML, Haskell, …
  - Surprisingly Javascript supports functional programming very well!

Logic Paradigm

- Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.
- When studying logic programming we see:
  - Programs as sets of constraints on a problem
  - Programs that achieve all possible solutions
  - Programs that are nondeterministic
- Example logic programming languages:
  - Prolog

Declarative Languages

- Prolog is a unique language with its basic ideas of facts, rules and atoms combined into a program often called a database
- There are other declarative languages where a "program" consists only of statements declaring what the result should be, not how to do it
- SQL (Structured Query Language) is the prime example
  - But all SQL implementations have procedural extensions (PL/SQL, T-SQL, etc.)
- XPath is another example

Markup Hybrid Languages

- Pure markup languages are mostly based on Standardized General Markup Language (SGML)
- HTML and XML are examples of two pure markup languages
  - No processing is done by the language
  - Markup provides directions to an external tool
- But markup languages have been extended to support some programming
- Examples: JSTL, XSLT

Markup Hybrids

- XSLT (Extensible Stylesheet Language Transformations) is a declarative XML based language used for transforming from XML to XML
- Often used to transform XML documents into HTML (a subset of XML) for presentation in browsers
- Requires a processing engine that takes XML and XSLT as input and outputs XML (or HTML)
- Capabilities overlap XQuery

Programming Domains

- Scientific applications
  - Large numbers of floating point computations; use of arrays
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

Programming Domains

- Computer software operates in a large number of different domains
  - Scientific applications
  - Business applications
  - Artificial intelligence
  - Systems programming
  - The Internet
  - Multimedia
  - Embedded Systems
  - Industrial Control
- Some programming languages have been designed for specific domains
- Others were explicitly design as general purpose

Scientific Applications

- AKA Number Crunching. The first computer applications (1940's)
- Normally involve simple data structures, simple control structures and large numbers of floating point computations
- Abstraction is usually applied OUTSIDE the language before the computational problem is expressed in a language

Scientific Applications

- Earliest high-level languages (FORTRAN, from FORmula TRANslation) were designed for scientific computing
- Efficiency is primary concern -- competition with assembler
- Algol 60 was designed to provide better expressiveness and abstraction but failed to catch on
- FORTRAN is still very much in use – only recently has C begun to be adopted in the domain

Business Applications

- Early 1950's people realized that computers were very good at doing tedious and boring tasks
- Computers were specifically designed for business (high I/O capabilities) as were languages
- COBOL (COmmon Business Oriented Language) was first designed in 1959 (based on Grace Hopper's Flow-Matic language)
- In 2002 object-oriented extensions were grafted onto the language

COBOL

- More lines of COBOL are in existence and still running than any other computer language
- If you want job security learn COBOL – there's always demand for people who can maintain the COBOL corpus
- Other developments in business applications are of little interest to computer science; e.g., RPG (Report Program Generator)
- Not discussed much in text

Database Management Systems (DBMS) and associated languages

- Usually neglected in programming language texts
  - A DBMS is a storage system and programming environment designed for optimal, error-free and efficient storage and retrieval of data
  - The "back end" of both business and scientific systems
- Most well known are the relational DBMS, all of which support Structured Query Language (SQL)
- Many interesting OO languages and DBMS are evolving

DBMS Languages

- DBMS programming languages are no different from other programming languages with the exception that they operate against, and are tightly bound to, a persistent store of data

SQL

- SQL is a "non-procedural language"
  - You say what you want rather than expressing an algorithm and hoping that the results are what you want
  - In practice it not possible to express everything as a query, so all DBMS support procedural extensions to SQL
- Other language families are associated with other storage systems
- The distinguishing feature is the close binding between the features of the language and the internal storage structure of DBMS

Artificial Intelligence (AI)

- Characterized by symbolic rather than numeric manipulation of data
- Goals include mimicking of human intelligence and design of systems that can perform well in areas that humans perform well
- List structures seem lend to themselves well to expressiveness in this area
- First AI language was LISP (1958) aka "Lots of Irritating Stupid Parentheses"). A program is a list itself.

AI Languages

- LISP is still used today
- LISP descendants include Scheme, ML, Haskell, etc.
- Since the 1970's Prolog – a language with a unique paradigm – has also been widely used in the AI environment

Systems Programming

- Operating systems and device drivers control the basic functions of a computer
  - Designed to provide a safe, stable and efficient interface to hardware
  - Efficient code is essential – maximize resources devoted to the user programs not the OS
- The C language was designed specifically for writing Unix – to avoid rewriting the kernel in assembler for each new hardware platform
  - Other languages such as BLISS and PL/S were designed for OS writing
- Key features:
  - Low level
  - Reliance on programmer to avoid mistakes.

Internet and Web Programming

- Web started as static content expressed in a markup language (HTML)
- Dynamic web content led to demand for easy to write, dynamically interpreted languages with strong string manipulation facilities – because they are designed to emit a stream of HTML
- Dynamic content also led to the migration of DBMS's and languages from their primary business orientation

Internet and Web Programming

- Today's web is a mish-mash of different technologies and languages
  - Markup languages (HTML, DHTML, XML, XHTML, CSS)
  - Server side scripting languages and engines (VBScript, PHP, Python, Ruby, .NET, Java)
  - Client side scripting languages (Javascript, Actionscript (Flash))
  - Database systems and languages, including SQL and XML
- Pure markup languages such as HTML are turning into programming languages (HTML 5)

Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications)
- **Cost**: the ultimate total cost

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Evaluation Criteria: Readability

- Prior to the 1970's machine readability was more important than human readability
- Then people realized that software maintenance was more expensive than software development
  - Development of more readable languages was aided by the drop in cost of memory.
  - It's much more difficult to fit a compiler into 32KB than 32MB or 32GB
- Now human readability is an important criterion

Evaluation Criteria: Readability

- Overall simplicity
  - How large is the set of features and constructs?
    - What if reader and writer use different subsets?
  - Feature multiplicity
    - Different ways to do the same thing, ex

```
j = j + 1;      j += 1;
j++;            ++j;
```

  - Operator overloading
    - Can be an issue especially with user defined overloads
    - In many SQL variants and in Javascript + is used for integer addition, floating point addition, decimal addition and string concatenation

      What is '123' + 4.0?

Readability: Orthogonality

- Orthogonality means:
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination of primitives is legal

Orthogonality example

- 32bit addition, from IBM/VAX assembly languages:
- IBM

```
A  reg, mem          ;adds mem into reg
AR reg1, reg2        ;adds reg2 into reg1
```

- VAX

```
addl     op1, op2    ; op1 and op2 are any reg/mem
```

- The VAX instruction set exhibits orthogonality

Non-Orthogonality: examples from C

- The C language supports arrays and structures (structs); functions can return a struct but not an array
- An struct member can be any data type except **void** or a structure of the same type
- An array element can be any data type except **void** or a function
- Parameters are passed by value, except arrays (we'll discuss this further later)

Syntax considerations

- Identifier forms: flexible composition
  - Many older languages restricted identifiers to a small number of characters ; ex Fortran 77, 6 chars, some BASICs allowed only 2
- Special words and methods of forming compound statements
  - Curly brace languages can be difficult to read when deeply nested control structures are used
- Form and meaning: self-descriptive constructs, meaningful keywords
  - In C "static" means one thing inside a function and something entirely different outside a function

Evaluation Criteria: Writability

- How easily can the language be used to create programs for a particular domain?
- Consider Visual Basic (VB) and C for
  - A graphical game program
  - An embedded controller for automotive brakes

Simplicity and orthogonality

- Few constructs, a small number of primitives, a small set of rules for combining them
- With large feature sets it is possible to use unknown features accidentally
- Ex. What does this mean in C++

```
cout << (i << 2)
```

- Does it mean the same as this?

```
cout << i << 2
```

Support for abstraction

- The ability to define and use complex structures or operations in ways that allow details to be ignored
- The ability to model a problem in terms of that problem rather than terms of the computer language

Procedural or Process Abstraction

- A procedural abstraction is a specification
  - Describes effects on outputs for given inputs – what it does, not how it does it – ignores implementation
  - Treats procedure or function as a "black box"
  - Can be applied in any language
- Classic example: sorting
  - We just want to call a sort routine when we need to sort something
  - We don't want to clutter up code by implementing a sort algorithm every time

Data abstraction

- Data abstraction
  - The ability to handle data in meaningful, natural ways
- Abstract Data Types:
  - Extend procedural abstraction to data
  - Example: type float
  - Extends imperative notion of type by:
    - Providing encapsulation of data/functions
    - Separation of interface from implementation

Expressivity

- A set of relatively convenient ways of specifying operations – domain specific
- Strength and number of operators and predefined functions
- APL is a language famous for expressivity in matrix operations (also known as a write-only language)

Evaluation Criteria: Reliability

- Performing to specifications under all conditions
- Many design considerations contribute to reliability
- Very often we have trade-offs involving expressivity, writability and reliability
  - There is a good reason why the very verbose COBOL language is over 50 and going strong!

Type Checking

- Checking for type errors either at runtime or compile time
- Runtime checks are significantly more expensive
- Early detection is less expensive than correcting released code
  - But forcing early detection of type errors reduces writability and expressiveness
  - Many scripting languages freely cast types at runtime which can result in bizarre and difficult to detect errors later

Exception Handling

- Refers to built-in mechanisms to intercept runtime errors, handle them and continue normal execution
- Possible in any language, but some languages have better facilities than others

Aliasing

- Two or more distinct names that refer to the same memory location or object
- Widely regarded as a fruitful source of error
- Difficult to detect and almost impossible to prevent

Readability and Writability

- A language that does not support natural ways of expressing an algorithm will require the use of "unnatural" approaches, and hence reduced reliability
- Some languages are known as "write-only" languages
- It is possible to write unreadable code in any language
  - Speaking of which we are running the Obfuscated Code Contest again this year
  - Cash prizes for bad code!

Evaluation Criteria: Cost

- The total cost of a language is a function of many variables
  - Training programmers to use the language
  - Writing programs (closeness to particular applications)
  - Compiling programs (fast or optimized?)
  - Executing programs (efficiency)
  - Language implementation system: availability of free compilers
  - Reliability: poor reliability leads to high costs (consider software for a nuclear reactor)
  - Maintaining programs is often THE most expensive factor

Evaluation Criteria: Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language's official definition

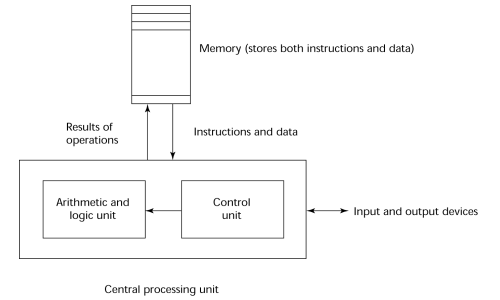
Influences on Language Design

- Computer Architecture
  - Imperative languages were developed around the prevalent computer architecture (*von Neumann)* rather than around the way users think
  - Functional and logic programming languages were based on mathematical models
  - OO attempts to encapsulate a human view of programming problems
- Programming Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture

- Dominant model of computer architecture since the 1940's is Von Neumann architecture
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables are models of memory cells
    - Assignment statements model piping
    - Iteration is efficient

The von Neumann Architecture



The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the
  counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```


Programming Design Methodologies

- Before the 1960's hardware was much more expensive than software

Programming Design Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Language Design Trade-Offs

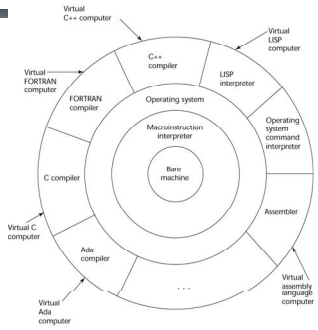
- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability
  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

Implementation Methods

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

Layered View of a Computer

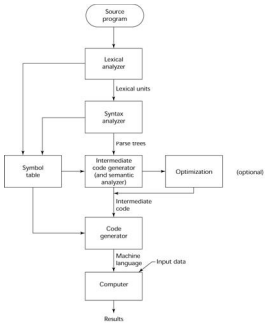
The operating system and language implementation are layered over machine interface of a computer



Compilation

- Translate high-level program (source language) into object code (language independent) which is then linked into machine code
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: object code is generated
- Object code modules are language-independent linker inputs

The Compilation Process



After Compilation

- **Linking**: the process of collecting system program units and linking them to a user program
  - Object code is input to a linker
  - The linker resolves addresses and outputs machine code
- **Program Loader**(executable image):
  - The Operating system provides program loading facilities
  - Executable code turned into an "executable image"
    - Unresolved references are resolved
    - Memory is allocated and mapped out

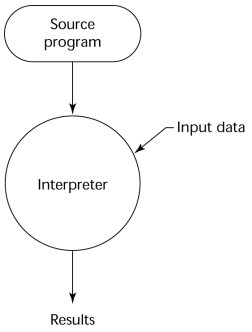
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process



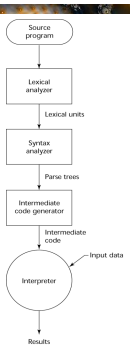
Hybrid Implementation Systems

- 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

Hybrid 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*)
- Smalltalk compiles methods to byte code
- Microsoft Common Language Runtime is a byte code interpreter

Hybrid Implementation Process



Just-in-Time (JIT) Compilers

- Compilers initially translate programs to an intermediate language (Java byte code, CLR byte code, etc)
  - Then the intermediate language is compiled into machine code when subprograms are are called
  - Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- JIT technology is largely responsible for making Java competitive with fully compiled languages at runtime

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands `#include`, `#define`, and similar macros
- C++ templates (used for generic classes) and handled by macro expansion in the preprocessor

Programming Environments

- A collection of tools used in software development
- Includes compilers, editors, debuggers, profilers, linkers and other tools
- UNIX
  - An older operating system and tool collection
  - Many command line tools are still used extensively today; e.g., make, grep, awk sed
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX

IDEs

- Integrated Development Environment
- Typically includes compiler, editor, build automation
- May also include source control system, class browser, object inspector, etc.
- Some support multiple languages
  - Eclipse, ActiveState Komodo
- Others are more tightly integrated with specific languages

Microsoft.NET

- A collection of languages, technologies and development environment
  - Used to build Web applications and non-Web applications in any .NET language
- Most common Languages include C++, C#, Visual Basic
  - Dozens available
  - See http://www.dotnetpowered.com/languages.aspx
- Usually thought of as a large, complex visual environment
  - All operations can be performed entirely with command line tools
  - The .NET SDK has always been available as a free download
- Target output language is machine-independent byte for the Common Language Runtime (CLR)

NetBeans

- The Java answer to .NET
- Primarily used for Java, also supports C, PHP, Ruby, C++, Javascript, Groovy and many others
- Written in Java
- Extensible with modules