

ECS 140A: Programming Languages

Course Objectives: After completing this course, you should be able to demonstrate your understanding of a number of important programming language concepts and constructs by

- predicting the results of programs that use the concepts
- developing small programs or program fragments that make essential use of these concepts
- explaining how the constructs are typically implemented.

This Course . . .

- is not a “grand tour” of programming languages (“It is Wednesday, so it must be ADA-DAY”),
- is a survey of programming language concepts (rather than specific languages).

Organizing Theme: *Abstraction*

- Study concepts in order of increasing abstraction relative to underlying machine
- programming language features that support creation of new abstractions get special attention

Administrative Matters

Instructor

- Raju Pandey, 3041 EU II, 752-3584
- Email: pandey@cs.ucdavis.edu
- Office Hrs: T/Th 4:30 - 6:00 PM or by appointment

TA

- TA: to be announced
- Readers: to be announced
- Office hours: to be announced

Communication

- Newsgroups:
 1. `ucd.class.ecs140a`: For class announcements (to be used by instructor and TA only)
 2. `ucd.class.ecs140a.d`: For discussions.
- Home page:
<http://www.cs.ucdavis.edu/~pandey>
 - Class handouts
 - Homeworks and corrections
 - Grades
 - Links to other relevant programming language resources

Prerequisites and Activities

Prerequisites

- ECS 50, ECS 110; Drop if you do not have the pre-requisites.
- Good understanding of basic concepts

Homeworks

- Programming (40%): (i) C++ (10%) (ii) Java (10%) (iii) Lisp (10%) (iv) Prolog (10%)
- Two problem sets (4% each)

Tests

- One midterm (20%); One 2 hours final (32%)
- Open book open notes

Grading

- Not on the curve:
 - Assign A, B, C, D (min/max) for each homework/tests (based on my expectations)
 - Weighted average of standard grades averages for A, B, C, D
 - Use those to determine your weighted average

I reserve the right to change this scheme.

- Significant difference between homework scores and exam scores \Rightarrow alternate grading scheme.
 - 90% on homeworks and $< D$ in midterm and final $\Rightarrow F$ for course

Policies

- Written homeworks due before class: **NO DELAYS/NO EXCEPTIONS**
- Projects:
 - Due at 11:59 PM on due date. You loose 10% every hour after that.
 - Backup your directory periodically; We are not responsible for problems.
 - Make sure your programs run on Linux machines; Grading will be done there.
- Re-grades within **one week** of returning homework; After one week, No regrades.
- Attendance is not mandatory. However, responsible for all material
- **NO MAKEUP EXAMS**; So plan to be for midterm and final
- **NO CHEATING**:
 - Do your own work. May discuss general approach, but develop your own solution. **NO CODE SHARING: Even data structures**
 - DO NOT BUY YOUR SOLUTION.
 - Solutions from other sources prohibited.. If you pick up an algorithm from somewhere:
 - * Let me, TA, or readers know
 - * Cite your resources
 - Note: TA, reader, and I compare answers
 - Any instance of suspected cheating \Rightarrow referral to Office of Student Judicial Affairs.

Textbooks

- Primary text book: Louden
- Java: Gosling (Any Java book will do)
- Common Lisp: Wilensky
- Prolog: Clocksin and Mellish.

Lecture

- Copies of transparencies handed out in class; Leftovers in the pocket outside my office.
Postscript versions may be accessible from the web page.

Computing

- CSIF DEC, SGI, and HP Machines (Basement EU II)
- Accounts available: check with support staff
- Software:
 - Java: JDK; Different versions on different machines. Details to be posted shortly.
 - Common Lisp: clisp,
 - Prolog: bp
- PC/Macs okay but need to be able to run on local machines
Your responsibility to make sure it works; TA or reader is not going to debug it for you.
- More on it in first discussion section

What is a programming language?

- A formal notation that describes computations
 - A set of primitives such as data types, control structures, and other abstractions, and
 - a set of rules for combining primitives and user-defined abstractions for defining computations
- Used for specifying, organizing, and reasoning about computations
- Similar to the general notion of languages except
 - more precise. For instance, mostly context free.

How can programming language help?

- Programming's biggest problem: *Dealing with complexity*
- Be as simple as possible by
 - suppressing unnecessary detail
 - consisting of a few parts that fit together in simple ways
- support modularization
 - programs are readily built up from separate subprograms
 - modules' interfaces are clear and simple
- Supports sharing and reuse of program parts

What do languages contain?

- Syntax: How will a program be represented?
- Semantics: What does a program mean?
- Abstractions:
 - Control
 - Data
- Computational paradigm:
 - Imperative: computation is a sequence of actions.
 - Functional: Computation as a set of functions and function applications.
 - Logic: Computation as a verification of an assertion against a set of logical truths.
 - Object-oriented: Computation as a set of autonomous interacting objects.
- Execution model: how does execution take place?
 - Compilation → execution
 - Interpretation

How do we judge a programming language L ?

- What do we want to judge? suitability of L for an application
- Many criteria, some conflicting \Rightarrow no ideal language.
- **Efficiency:** How efficiently can a program written in L be
 - translated?
Does language contain constructs that cannot be checked at compile time? Example: array index value.
Does it force the compiler to go through a source program more than once? Example: Usage of a variable before its definition.
 - executed?
Does language contain features that are difficult to implement?
Does language contain information that can be used for efficient implementation? Indeed, one of the major motivations in programming language design: give enough information to the compiler so that it can generate efficient code.
Example: use type information for runtime code organization and code generation.
- **Expressiveness:** How easy is it to write complex processes and structures? Is language concise?
Expressibility interacts with efficiency, readability, and reliability.

How do we judge a programming language *L*? - cont'd.

- **Generality:** Are constructs general or special cases? Are there instances of constructs that do not apply in general case?
Example: Pascal allows procedure declarations and procedure parameters but no procedure variables \Rightarrow procedure is not being treated in its most general form.
- **Orthogonality:** Different language constructs should represent independent concepts so that they can be combined in many different ways.
Example: In Pascal, data types and procedures are orthogonal concepts, with rules for passing and returning values. However, file types are special types, which cannot be passed in a conventional way.
- **Uniformity:** Do similar things behave in a similar way.
Example: In Pascal, Repeat-Until start their own begin-end block but not while-do loop.
- **Simplicity:** Are concepts simple?
- **Preciseness:** Does the language have precise semantics? Can programs written in this language be verified?
- **Machine independence:** Can the program be ported from one machine to another?
- **Programming environment:** Does the language support efficient compilers, interpreters, editors, debuggers, testing and maintenance packages?

What kinds of programming languages?

- A language is at a higher level if it is independent of the underlying machine.
- A language is a general purpose language if it can be applied for a wide range of problems.
- Assembly programming languages: MIPS, x86, Sparc etc.
- High level programming languages: FORTRAN, APL, LISP, COBOL, C++, Eiffel, etc.
- Unix Shells: `cs`h, `tc`sh, `z`sh, etc.
- Command line options: `cc -o test test.c`
- Languages of utilities and tools: Emacs, sed etc.
- Imperative, functional, logic, object-oriented etc.
- Libraries: STL, UNIX OS etc.
- Meta-languages: languages that describe languages
- Observation: different languages for different domains.
- Our focus: languages for describing general computation

Historical Perspective

1. Machine languages (1940's) and assembly languages (1950's)
 - Machine-specific
 - Verbose
 - modularity: zero
2. FORTRAN, ALGOL, COBOL (1960's) — the first high level languages
 - Portable, concise
 - Modularity better (subroutines, procedures)

The transition 1 \rightarrow 2 brought about

- *small increase* in cost of use as programs became slightly bigger and slower
- *large reduction* in cost of development as programs became much cheaper, more portable, and more reliable.

For most purposes, level-2 languages have replaced level-1 languages.

The quest for another similar increase in programmer productivity continues to this day – so far without any success.

Major

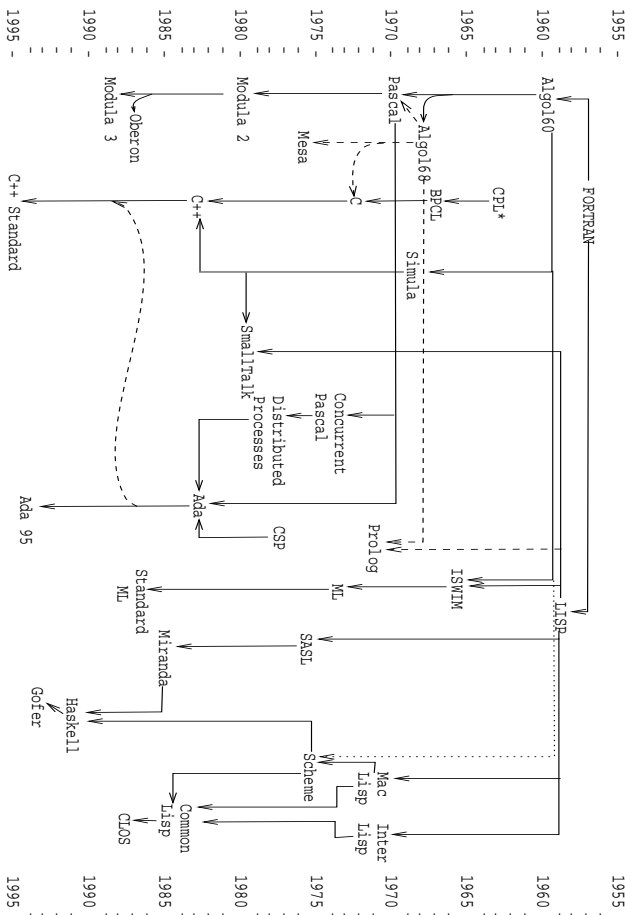
- 1950–55:
Hardware: Vacuum-tube computers
Methods: Assembly languages; foundation concepts: subprograms, data structures;
Languages: experimental use of expression compilers
- 1956–60: Small, slow and expensive computers,
Hardware: Magnetic tape storage; core memories; transistor circuits
Method: Compilers, software interpreters, code optimization, dynamic storage management, linked data structures, BNF grammars
Languages: FORTRAN, ALGOL58 and Algol60, COBOL, LISP
- 1961–65:
Hardware: Large expensive computers, Magnetic-tape storage systems,
Methods: Operating systems, Multi-programming, Syntax-directed compilers,
Languages: FORTRAN IV, Algol60-revised, SNOBOL. APL

Major Influences and Programming Languages

- 1966-70:
 - Hardware: Microprogramming, minicomputers, ICs
 - Methods: Time-sharing interactive operating systems, Optimizing compilers, Translator writing systems,
 - Languages: PL/I, FORTRAN 66 (Standard), COBOL 65 (Std), Algol 68, Snobol 4, Simula 67, BASIC, APL
- 1971-1975:
 - Hardware: Microcomputers, Small inexpensive storage systems
 - Methods: Proofs of program correctness, Structured programming, Software engineering, Reactions against large complex languages
 - Languages: Pascal, COBOL 74 (Std), PL/I (Std)
- 1976-1980:
 - Hardware: Powerful inexpensive computers, Large inexpensive storage systems, Distributed computers systems,
 - Methods: Concurrent and real-time programming using high level languages, Interactive programming environments, Data abstraction, Software components, Formal semantic definitions, Reliability and ease of maintenance as language design goals
 - Languages: Ada, Fortran 77, ML

Major Influences and Programming Languages - cont'd.

- 1981-85: Personal computers, first workstations, large mass storage systems, distributed computing
 - Object-oriented programming; interactive environments; syntax-directed editors;
 - Languages: Turbo Pascal, Smalltalk-80, Ada 83, Postscript.
- 1986-90:
 - Hardware: Microcomputers; Rise of engineering workstations; RISC architectures; global networking; Internet
 - Methods: client-server computing
 - Languages: FORTRAN 90, C++, SML
- 1991-95:
 - Hardware: Very fast inexpensive workstations, Massively parallel architectures, voice, video, multi-media
 - Methods: Open Systems, Environmental frameworks; Information super-highway
 - Languages: Ada 95, TCL etc.



Applications and Languages

Era	Application	Major Languages	Other languages
1960s	Business	Cobol	Assembler
	Scientific	FORTRAN	Algol, BASIC, APL
	System	Assembler	JOVIAL, FORTH
	AI	LISP	SNOBOL
	Business	Cobol and SpreadSheets	C, PL/I, 4GLs
Today	Scientific	FORTRAN, C, C++	BASIC, Pascal
	System	C, C++	Pascal, Ada, BASIC, Modula
	AI	LISP, Prolog	
	Publishing	TeX, Postscript, Word	
	Process	Unix shell, TCL, Perl	Marvel
New Paradigms	ML, SmallTalk, Java	Eiffel	