DM820
Advanced Topics in Programming Languages

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM820/

COURSE ORGANIZATION

Course Elements

- 7 lectures Thursday 08-10 (Weeks 15–17, 19–22)
- 3 lectures Monday 16-18 (Weeks 16–18)

- 7 discussion sections Wednesday 08-10 (Weeks 16–22)
- 4 presentation sessions Monday 16-18 (Weeks 19–22)

- exam = oral presentation + practical project
  - oral presentation is internal pass/fail
  - practical project is external with grades

Course Goals

- **Learn more ways to program!**

- To this end, you will learn
  - to use advanced concepts in standard programming languages
  - to use non-standard programming languages
  - to use programming in widely differing contexts

- Focus of the course on broadly covering (parts of) the field
- Depth added through individually assigned topics

Course Contract

- I am offering you the following:
  1. I will help you in picking a topic
  2. I am always willing to help you

- From you I expect the following:
  1. You give a good presentation of your assigned topic
  2. You have FUN programming!

- You and I have the right and duty to call upon the contract!

Topics for Lectures 1–6

- Mostly up to you!
- Lectures can be picked from following topics
  a) scripting languages
  b) constraint programming
  c) multi-paradigm programming languages
  d) string/term/graph rewriting
  e) domain specific languages
  f) program verification
  g) extensible programming
  h) aspect-oriented programming
  i) parser generation
- Today (& next time): A little history of programming languages

Topics for Lectures 7–10

- Up to me!
- Selection criteria:
  - Topics must supplement your chosen topics in order to guarantee a broad overview
  - Topics must be fun (in some weird way)

Individually Assigned Topics 1/3

Specific Languages:

A. Scala (Functional programming for JVM)
B. Clojure (Dynamic programming for JVM)
C. CoPriS (Constraint programming in Scala)
D. Objective C (OS X & iOS)
E. Csound (DSL for audio)
F. Functional Logic Programming Languages (e.g. TOY, Curry)
G. Semantic Web Query Languages
H. Web Ontology Languages
I. Distributed Programming Languages (e.g. Erlang)
J. Obscure Languages (e.g. Brainfuck, Whitespace, Unlambda)

Individually Assigned Topics 2/3

Advanced language features:

K. Generators and Yield (in Python or C#)

L. Scripting Language Embedding

M. Parallel Programming in Functional Languages

N. Higher-Level Database Programming (e.g. LINQ)

O. Introspection and Metaprogramming (e.g. Java, Python)

P. GPU Programming (e.g. CUDA, OpenCL)

Analysis of programming languages:

Q. Automated Complexity Analysis

R. Automated Termination Analysis

S. Verification by Contract

T. Theorem Proving

Individually Assigned Topics 3/3

Advanced programming concepts:

U. Design Patterns (e.g. MVC for GUI Programming)

V. Aspect-Oriented Programming

W. Constraint-Handling Rules

Concrete frameworks / systems:

X. AJAX

Y. Rich Client Platform

Z. Eclipse Plugins

Я Google App Engine

Æ XMLVM (Java on the iPhone)

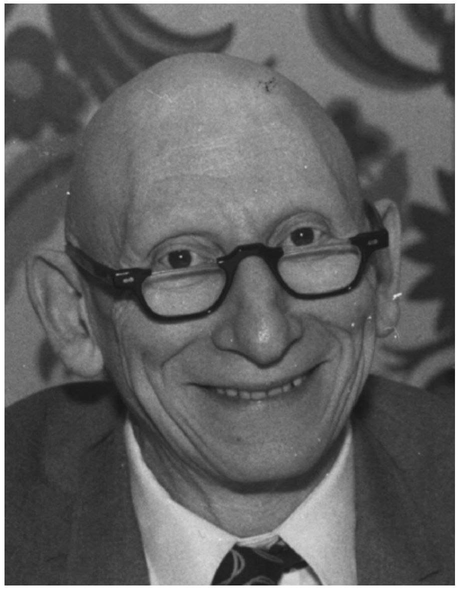
Ø Cloud Computing (e.g. Amazon EC2)

Å App Programming (e.g. Android, iOS)

(shamelessly stolen from Vitaly Shmatikov)

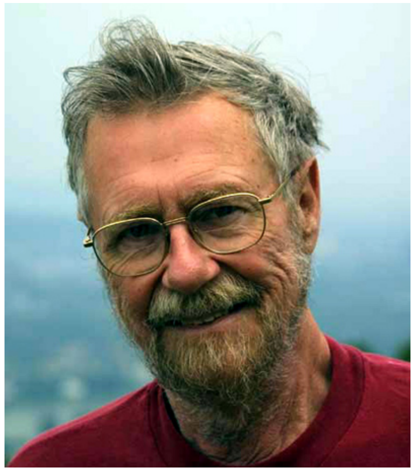
LITTLE HISTORY OF PROGRAMMING LANGUAGES

Quote 1



"A language that doesn't affect
the way you think about programming,
is not worth knowing."
- Alan Perlis

Dijkstra on Language Design



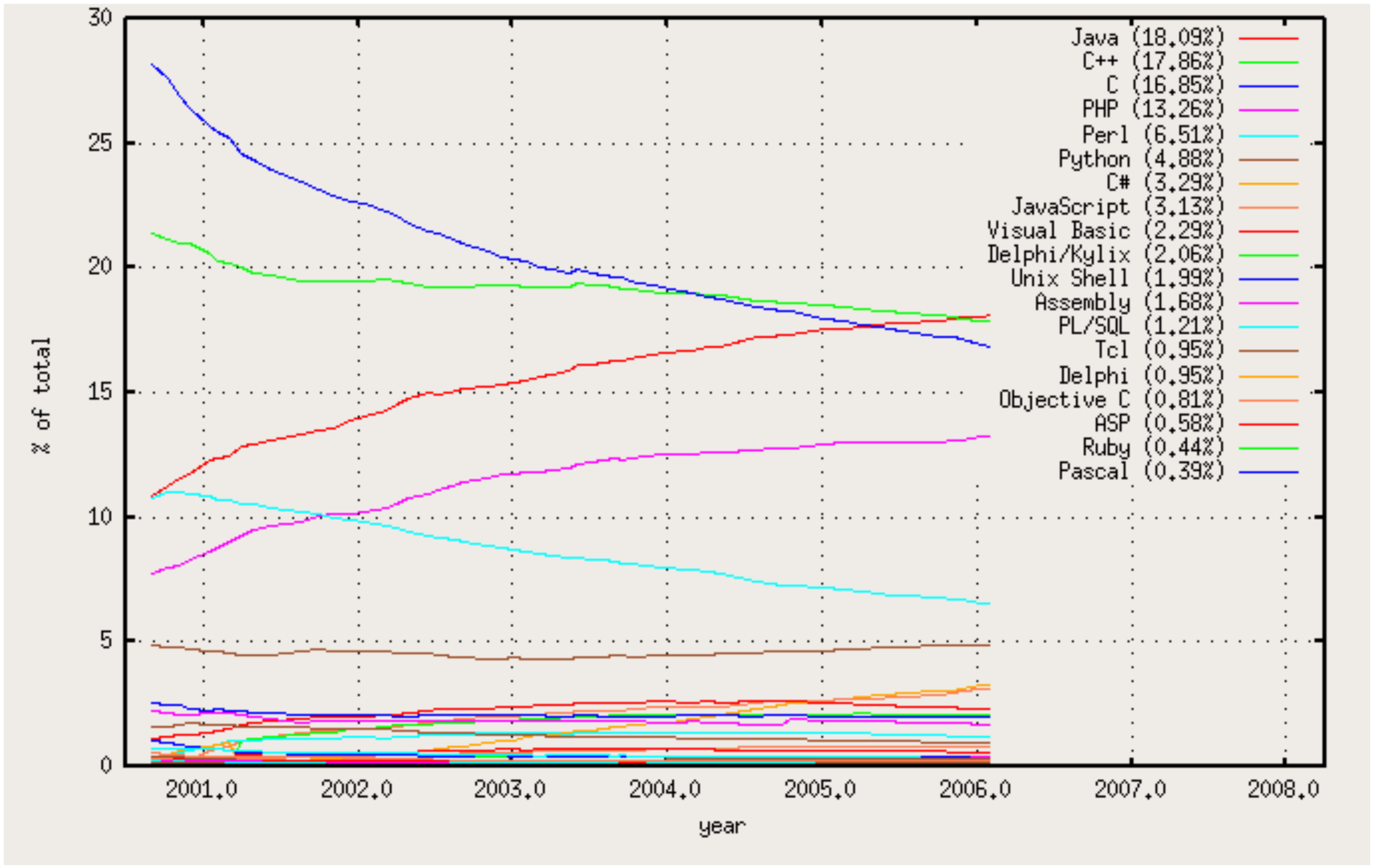
- "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence."
- "APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums."
- "FORTRAN, 'the infantile disorder' … is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use."
- "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

What's Worth Studying?

- Dominant languages and paradigms
  - C, C++, Java… JavaScript?
  - Imperative and object-oriented languages
- Important implementation ideas
- Performance challenges
  - Concurrency
- Design tradeoffs
- Alternative Programming Paradigms, Language Designs, and new Concepts!

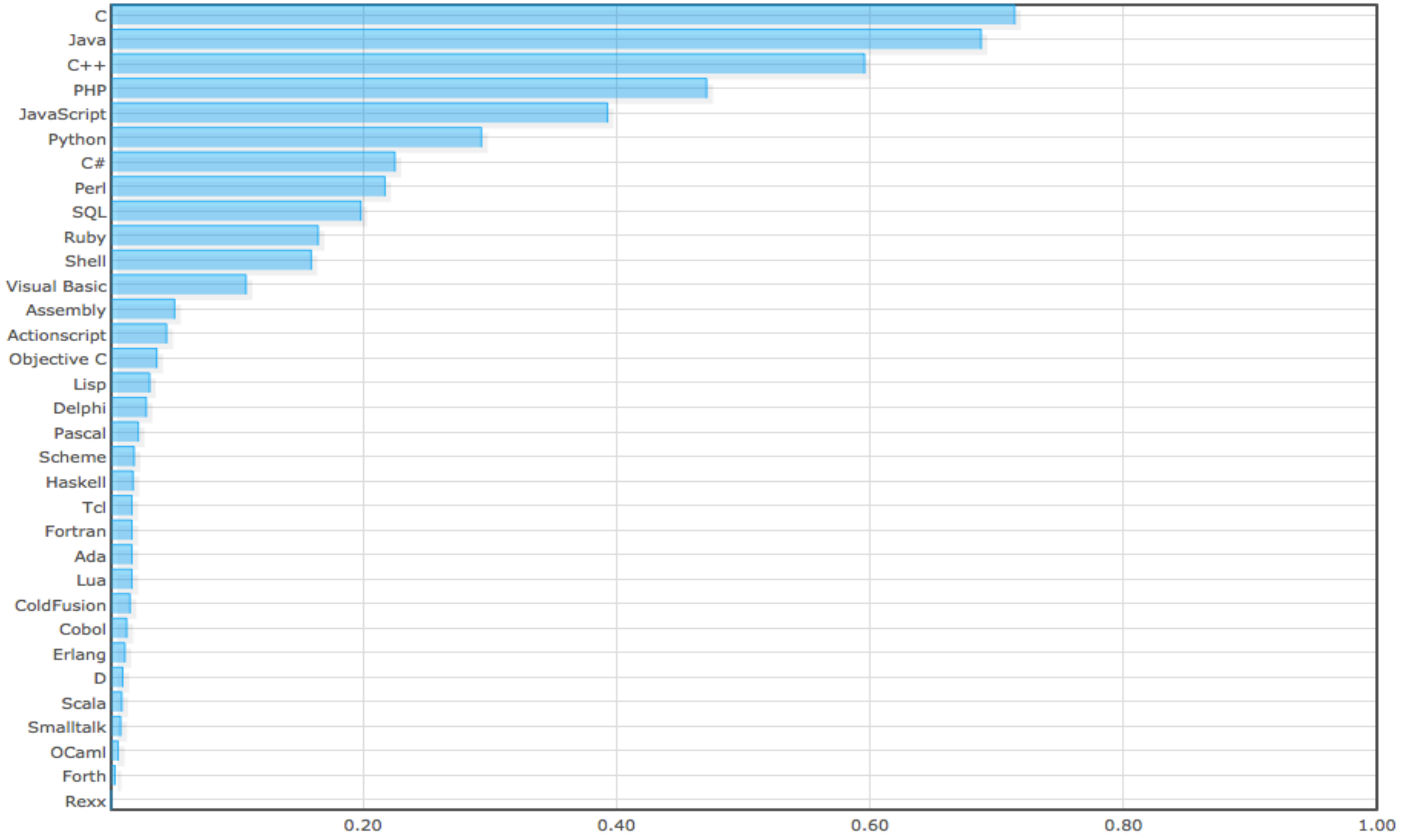
Languages in Common Use

[F. Labelle]



Based on open-source projects at SourceForge

Programming Language Popularity



Flon’s Axiom

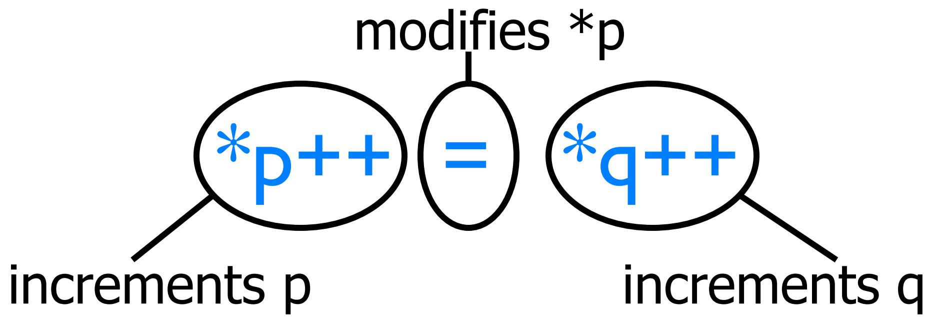
“There is not now, nor has there ever been,
nor will there ever be,
any programming language in which
it is the least bit difficult to write bad code.”

- Lawrence Flon

Latest Trends

- Commercial trends
  - Increasing use of type-safe languages: Java, C#, …
  - Scripting and other languages for web applications
- Teaching trends : Java/Python replacing C
- Research and development trends
  - Modularity
  - Program analysis
    - Automated error detection, programming env, compilation
  - Isolation and security
    - Sandboxing, language-based security, …

What Does This C Statement Mean?



Does this mean…

```
*p = *q;
++p;
++q;
```

… or

```
*p = *q;
++q;
++p;
```

… or

```
tp = p;
++p;
tq = q;
++q;
*tp = *tq;
```

Orthogonality

- A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations.
- Fewer exceptional rules = conceptual simplicity
  - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency

Efficient Implementation

- Embedded systems
  - Real-time responsiveness (e.g., navigation)
  - Failures of early Ada implementations
- Web applications
  - Responsiveness to users (e.g., Google search)
- Corporate database applications
  - Efficient search and updating
- AI applications
  - Modeling human behaviors

Quote 2

"These machines have no common sense; they have not yet learned to `think,' and they do exactly as they are told, no more and no less. This fact is the hardest concept to grasp when one first tries to use a computer."

- Donald Knuth

What Is a Programming Language?

- Formal notation for specifying computations, independent of a specific machine
  - Example: a factorial function takes a single non-negative integer argument and computes a positive integer result
    - Mathematically, written as fact: nat → nat
- Set of imperative commands used to direct computer to do something useful
  - Print to an output device: printf(“hello world\n”);
    - What mathematical function is “computed” by printf?

Computation Rules

- The factorial function type declaration does not convey how the computation is to proceed
- We also need a computation rule
  - fact (0) = 1
  - fact (n) = n * fact(n-1)
- This notation is more computationally oriented and can almost be executed by a machine

Factorial Functions

- C, C++, Java:

```
int fact (int n)  { return (n == 0) ? 1 : n * fact (n-1); }
```

- Scheme:

```
(define fact
    (lambda (n)  (if (= n 0) 1 (* n (fact (- n 1))))))
```

- ML:

```
fun fact n = if n=0 then 1 else n*fact(n-1);
```

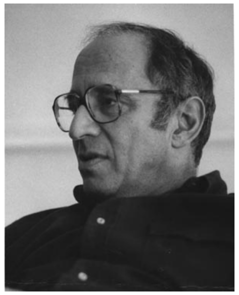
- Haskell:
  - `fact :: Integer->Integer`
  - `fact 0 = 1`
  - `fact n = n*fact(n-1)`

Principal Paradigms

- Imperative / Procedural
- Functional / Applicative
- Object-Oriented
- Concurrent
- Logic
- Scripting

- In reality, very few languages are "pure"
  - Most combine features of different paradigms

Where Do Paradigms Come From?



- Paradigms emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas
  - Thomas Kuhn. "The Structure of Scientific Revolutions."
- Programming paradigms are the result of people's ideas about how programs should be constructed
  - … and formal linguistic mechanisms for expressing them
  - … and software engineering principles and practices for using the resulting programming language to solve problems

Imperative Paradigm

- Imperative (procedural) programs consists of actions to effect state change, principally through assignment operations or side effects
  - Fortran, Algol, Cobol, PL/I, Pascal, Modula-2, Ada, C
  - Why does imperative programming dominate in practice?
- OO programming is not always imperative, but most OO languages have been imperative
  - Simula, Smalltalk, C++, Modula-3, Java, C#
  - Notable exceptions:
    - CLOS (Common Lisp Object System)
    - Scala

Functional and Logic Paradigms

- Focuses on function evaluation; avoids updates, assignment, mutable state, side effects
- Not all functional languages are "pure"
  - In practice, rely on non-pure functions for input/output and some permit assignment-like operators
    - E.g., (set! x 1) in Scheme
- Logic programming is based on predicate logic
  - Targeted at theorem-proving languages, automated reasoning, database applications
  - Recent trend: declarative programming (Why?)

Concurrent and Scripting Languages

- Concurrent programming cuts across imperative, object-oriented, and functional paradigms
- Scripting is a very "high" level of programming
  - Rapid development; glue together different programs
  - Often dynamically typed, with only int, float, string, and array as the data types; no user-defined types; often powerful lists types and hashtables
  - Weakly typed: a variable 'x' can be assigned a value of any type at any time during execution
- Very popular in Web development
  - Especially scripting active Web pages

Unifying Concepts

- Unifying language concepts
  - Types (both built-in and user-defined)
    - Specify constraints on functions and data
    - Static vs. dynamic typing
  - Expressions (e.g., arithmetic, boolean, strings)
  - Functions/procedures
  - Commands

Design Choices

- C: Efficient imperative programming with static types
- C++: Object-oriented programming with static types and ad hoc, subtype and parametric polymorphism
- Java: Imperative, object-oriented, and concurrent programming with static types and garbage collection
- Scheme: Lexically scoped, applicative-style recursive programming with dynamic types
- Standard ML: Practical functional programming with strict (eager) evaluation and polymorphic type inference
- Haskell: Pure functional programming with non-strict (lazy) evaluation.

Abstraction and Modularization

- Re-use, sharing, extension of code are critically important in software engineering
- Big idea: detect errors at compile-time, not when program is executed
- Type definitions and declarations
  - Define intent for both functions/procedures and data
- Abstract data types (ADT)
  - Access to local data only via a well-defined interface
- Lexical scope

Static vs. Dynamic Typing

- Static typing
  - Common in compiled languages, considered "safer"
  - Type of each variable determined at compile-time; constrains the set of values it can hold at run-time
- Dynamic typing
  - Common in interpreted languages
  - Types are associated with a variable at run-time; may change dynamically to conform to the type of the value currently referenced by the variable
  - Type errors not detected until a piece of code is executed

Billion-Dollar Mistake

Failed launch of Ariane 5 rocket (1996)

- $500 million payload; $7 billion spent on development

Cause: software error in inertial reference system

- Re-used Ariane 4 code, but flight path was different
- 64-bit floating point number related to horizontal velocity converted to 16-bit signed integer; the number was larger than 32,767; inertial guidance crashed