Programming Language Concepts

Mooly Sagiv
msagiv@acm.org
Tuesday 11-13, Schriber 317

TA: Oded Padon
Email: odedp@mail.tau.ac.il

http://www.cs.tau.ac.il/~msagiv/courses/pl17.html

Inspired by Stanford John Mitchell CS'242

Prerequisites

- Software Project
- Computational models

Textbooks

- J. Mitchell. Concepts in Programming Languages
- B. Pierce. Types and Programming Languages
- Semantics with Applications by Flemming Nielson and Hanne Riis Nielson
- Real World Ocaml by Anil Madhavapeddy, Jason Hickey, and Yaron Minsky
- JavaScript: The Good Parts by Douglas Crockford

Course Grade

- 50% Assignments (5 assignments)
  - 2-3 person teams
- 50% Exam
  - Must pass exam

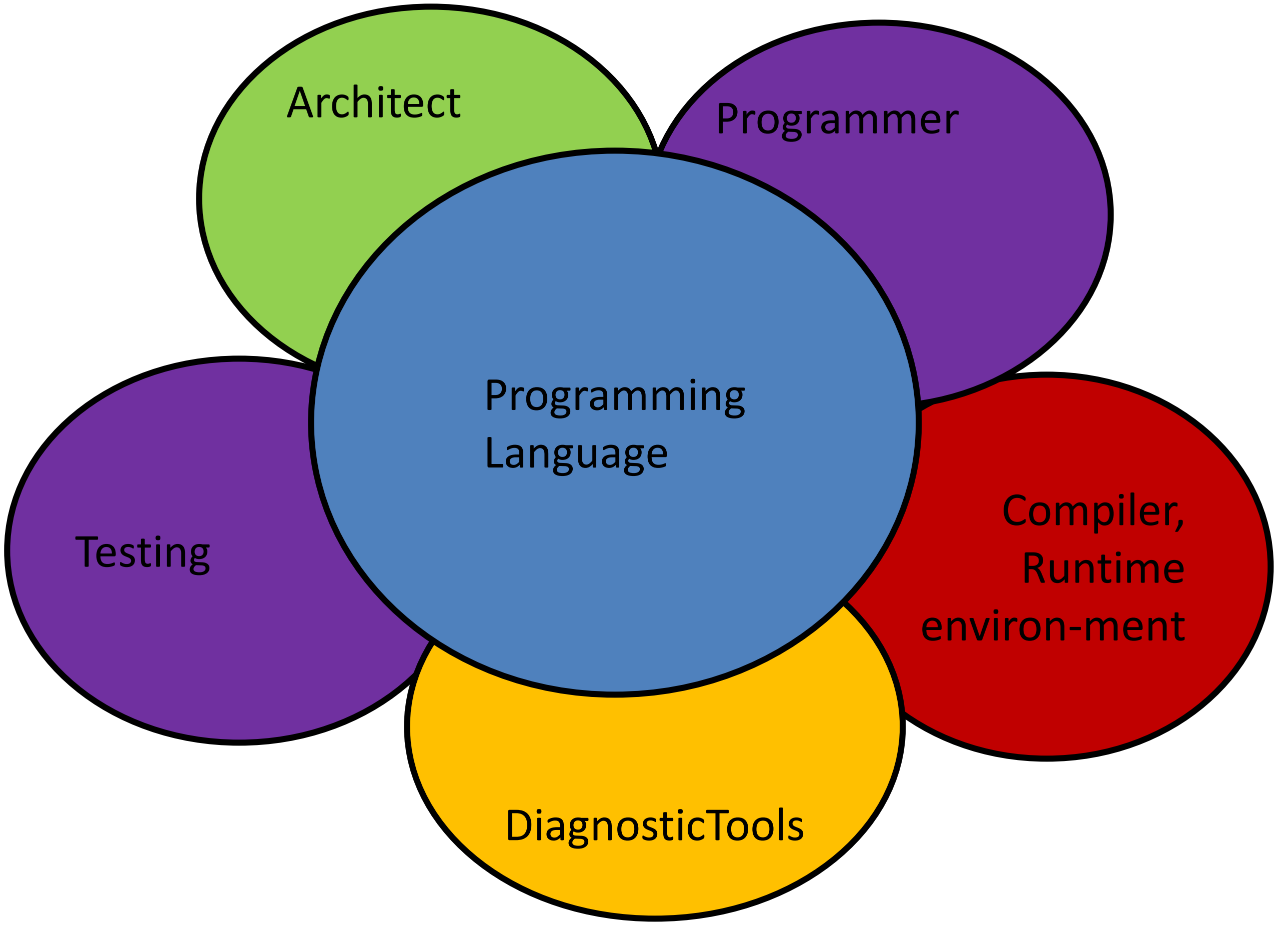
Goals

- Learn about cool programming languages
- Learn about useful programming languages
- Understand theoretical concepts in programming languages
- Become a better programmer in your own programming language
- Have fun

Course Goals (Cont)

- Programming Language Concepts
  - A language is a "conceptual universe" (Perlis)
    - Framework for problem-solving
    - Useful concepts and programming methods
  - Understand the languages you use, by comparison
  - Appreciate history, diversity of ideas in programming
  - Be prepared for new programming methods, paradigms, tools
- Critical thought
  - Identify properties of *language*, not syntax or sales pitch
- Language *and* implementation
  - Every convenience has its cost
    - Recognize the cost of presenting an abstract view of machine
    - Understand trade-offs in programming language design

Language goals and trade-offs



What's new in programming languages

- Commercial trend over past 5+ years
  - Increasing use of type-safe languages: Java, C#, Scala
  - Scripting languages, other languages for web applications JavaScript
- Teaching trends
  - Java replaced C as most common intro language
    - Less emphasis on how data, control represented in machine
- Research and development trends
  - Modularity
    - Java, C++: standardization of new module features
  - Program analysis
    - Automated error detection, programming env, compilation
  - Isolation and security
    - Sandboxing, language-based security, …
  - Web 2.0
    - Increasing client-side functionality, mashup isolation problems

What's worth studying?

- Dominant languages and paradigms
  - Leading languages for general systems programming
  - Explosion of programming technologies for the web
- Important implementation ideas
- Performance challenges
  - Concurrency
- Design tradeoffs
- Concepts that research community is exploring for new programming languages and tools
- Formal methods in practice
  - Grammars
  - Semantics
  - Types and Type Systems

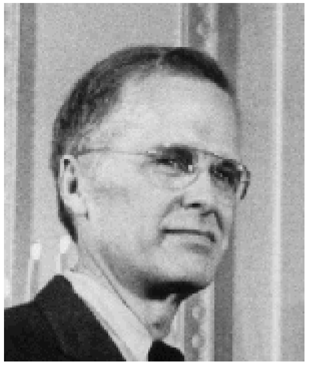
  …

Related Courses

- Seminar in programming Language
- Compilers
- Semantics of programming languages
- Program analysis
- Software Verification

The Fortran Programming Language

- FORmula TRANslating System
- Designed in early 50s by John Backus from IBM
  - Turing Award 1977
  - Responsible for Backus Naur Form (BNF)
- Intended for Mathematicians/Scientists
- Still in use



Lisp

- The second-oldest high-level programming language
- List Processing Language
- Designed by John McCarty 1958
  - Turing Award for Contributions to AI
- Influenced by Lambda Calculus
- Pioneered the ideas of tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, and the self-hosting compiler

Lisp Design Flaw: Dynamic Scoping

```
procedure  p;
        var x: integer
        procedure  q ;
                begin { q }
                …
                x
                …
                end { q };
        procedure   r ;
        var x: integer
        begin { r }
        q ;
        end;  { r  }
begin { p }
        q ;
        r ;
end { p }
```

The Algol 60

- ALGOrithmic Language 1960
- Designed by Researchers from Europe/US
- Led by Peter Naur 2005 Turing Award
- Pioneered: Scopes, Procedures, Static Typing

Name	Year	Author	Country
X1 ALGOL 60	1960	Dijkstra and Zonneveld	Netherlands
Algol	1960	Irons	USA
Burroughs Algol	1961	Burroughs	USA
Case ALGOL	1961		USA
…	….	…	…

Algol Design Flaw: Power

- E ::= ID | NUM | E + E | E – E | E * E | E / E | E ** E

C Programming Language

- Statically typed, general purpose systems programming language
- Computational model reflects underlying machine
- Designed by Dennis Ritchie, ACM Turing Award for Unix
- (Initial) Simple and efficient one pass compiler
- Replaces assembly programming
- Widely available
- Became widespread

Simple C design Flaw

- Switch cases without breaks continue to the next case

```
switch (e) {
 case 1:  x = 1;
 case 2:  x = 4 ;
          break;
 default: x = 8;
}
```

A Pathological C Program

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

Conflicting Arrays with Pointers

- An array is treated as a pointer to first element (syntactic sugar)
- E1[E2] is equivalent to ptr dereference: *((E1)+(E2))
- a[i] == i[a]
- Programmers can break the abstraction
- The language is not type safe
  - Even stack is exposed

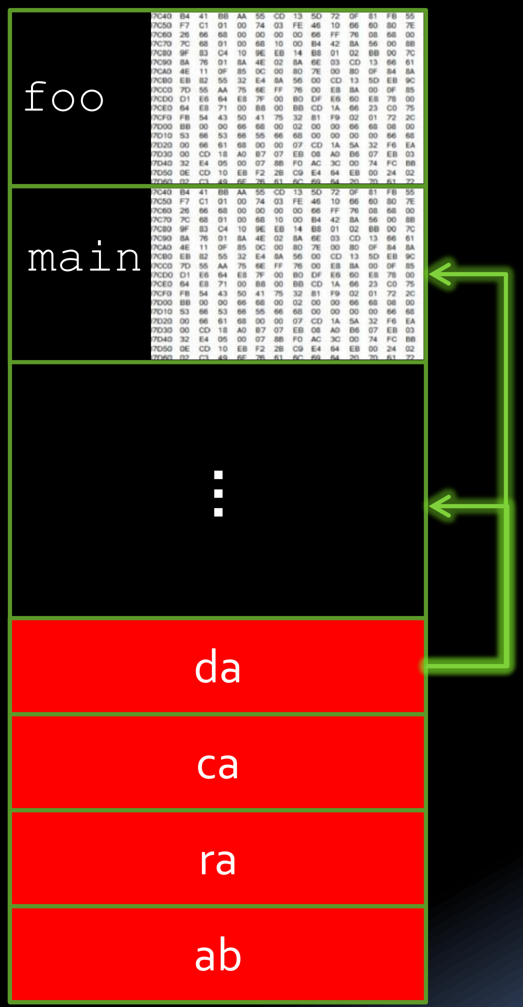
Buffer Overrun Exploits

```
void foo (char *x) {
  char buf[2];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

source code

```
> ./a.out abracadabra
Segmentation fault
```

terminal



memory

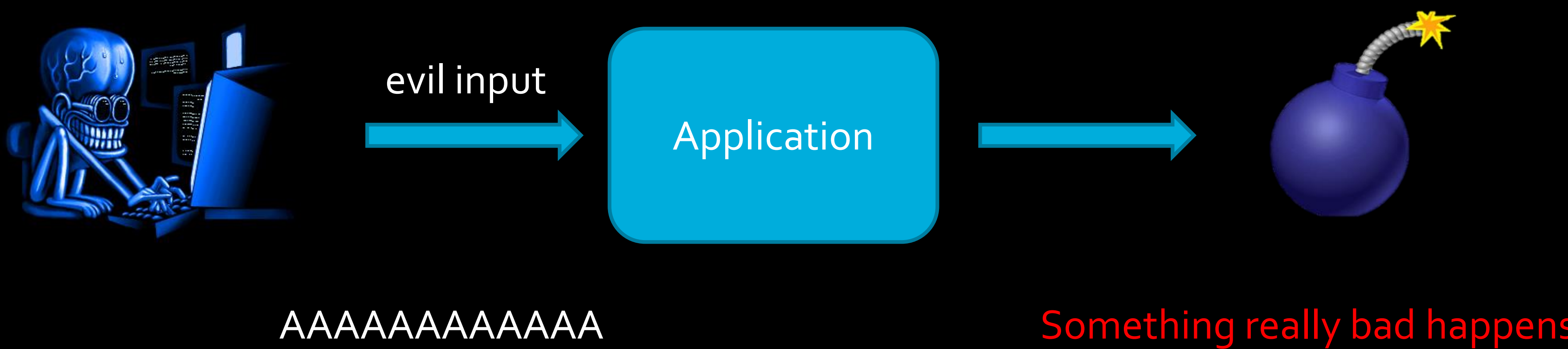
Buffer Overrun Exploits

```
int check_authentication(char *password) {
 int auth_flag = 0;
 char password_buffer[16];

 strcpy(password_buffer, password);
 if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
 if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
  return auth_flag;
}
int main(int argc, char *argv[]) {
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("     Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");  }
   else
      printf("\nAccess Denied.\n");
}
```

(source: "hacking – the art of exploitation, 2nd Ed")

Exploiting Buffer Overruns



Summary C

- Unsafe
- Exposes the stack frame
  – Parameters are computed in reverse order
- Hard to generate efficient code
  – The compiler need to prove that the generated code is correct
  – Hard to utilize resources
- Ritchie quote
  – “C is quirky, flawed, and a tremendous success”

The Java Programming Language

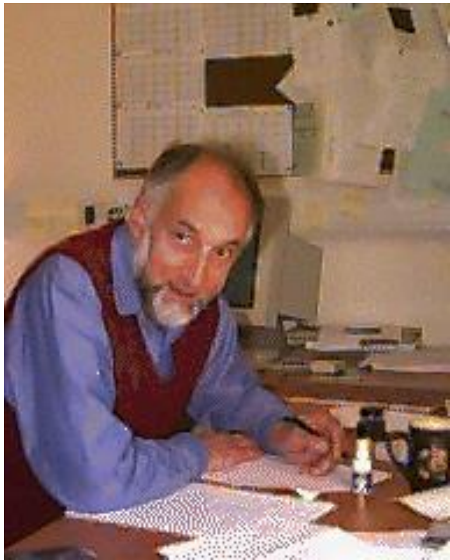
- Designed by Sun 1991-95
- Statically typed and type safe
- Clean and Powerful libraries
- Clean references and arrays
- Object Oriented with single inheritance
- Interfaces with multiple inheritance
- Portable with JVM
- Effective JIT compilers
- Support for concurrency
- Useful for Internet

Java Critique

- Downcasting reduces the effectiveness of static type checking
  - Many of the interesting errors caught at runtime
    - Still better than C, C++
- Huge code blowouts
  - Hard to define domain specific knowledge
  - A lot of boilerplate code
  - Sometimes OO stands in our way
  - Generics only partially helps
  - Array subtype does not work

ML programming language

- Statically typed, general-purpose programming language
  - “Meta-Language” of the LCF theorem proving system
- Designed in 1973
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
  - Encapsulated side-effects



Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …

Haskell

- Haskell programming language is
  - Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, interactive and compiled use
  - Different from ML: lazy evaluation, purely functional core, rapidly evolving type system
- Designed by committee in 80's and 90's to unify research efforts in lazy languages
  - Haskell 1.0 in 1990, Haskell '98, Haskell' ongoing
  - "A History of Haskell: Being Lazy with Class" HOPL 3

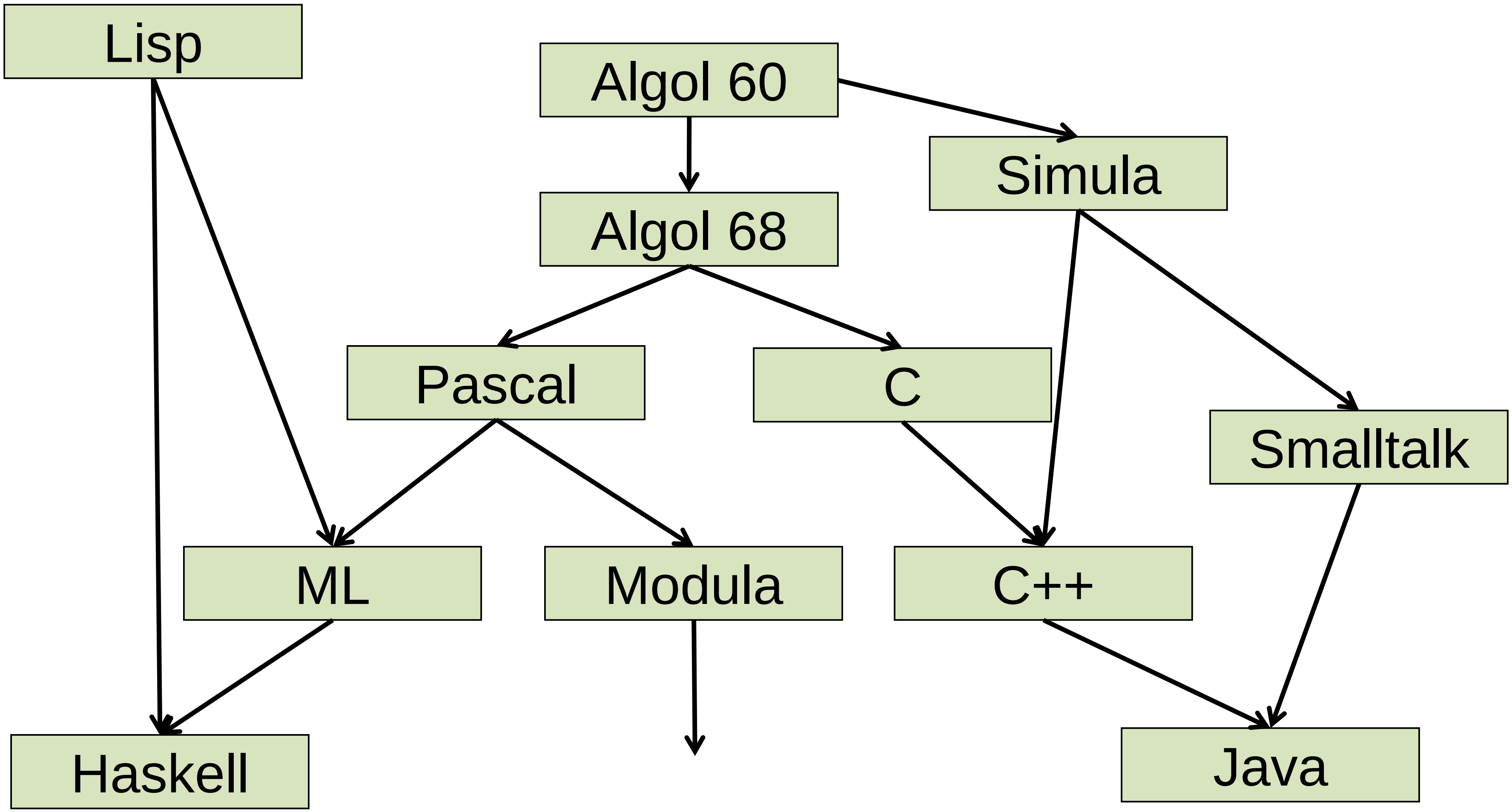
Paul Hudak

John Hughes

Simon
Peyton Jones

Phil Wadler

Language Evolution

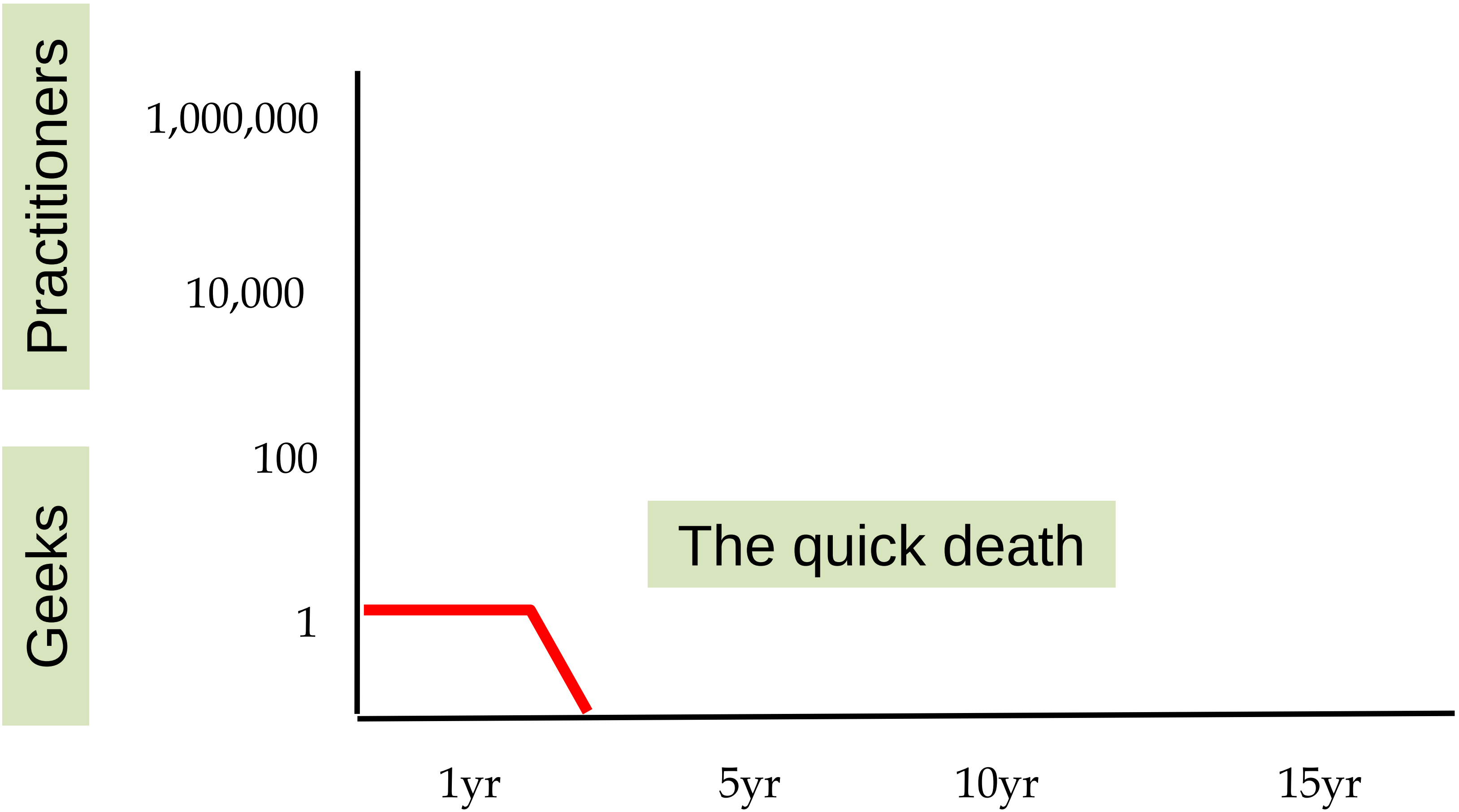


Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#, Scala, go

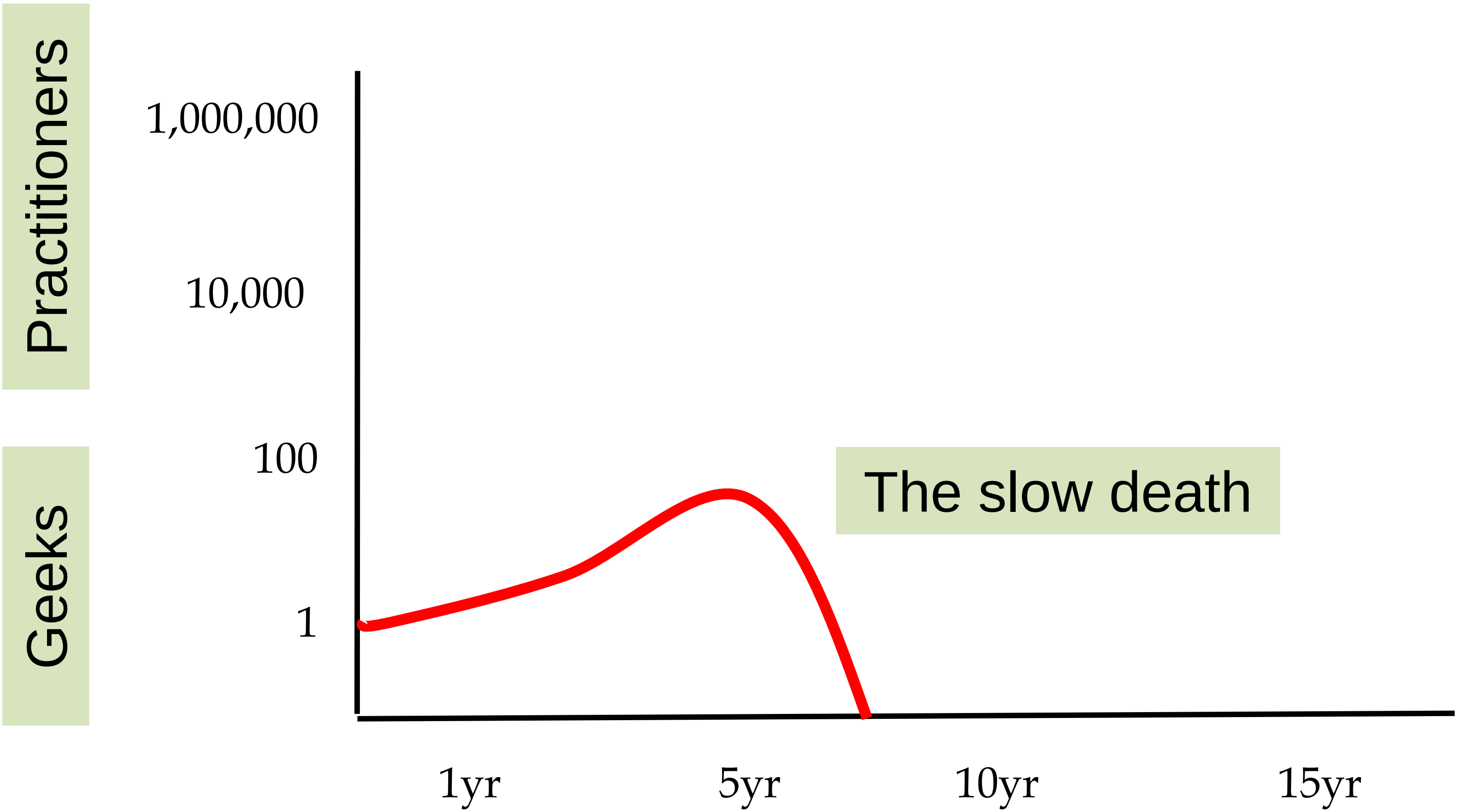
Scala

- Designed and implemented by Martin Odersky [2001-]
- Motivated towards "ordinary" programmers
- Scalable version of software
  - Focused on abstractions, composition, decomposition
- Unifies OOP and FP
  - Exploit FP on a mainstream platform
  - Higher order functions
  - Pattern matching
  - Lazy evaluation
- Interoperates with JVM and .NET
- Better support for component software
- Much smaller code

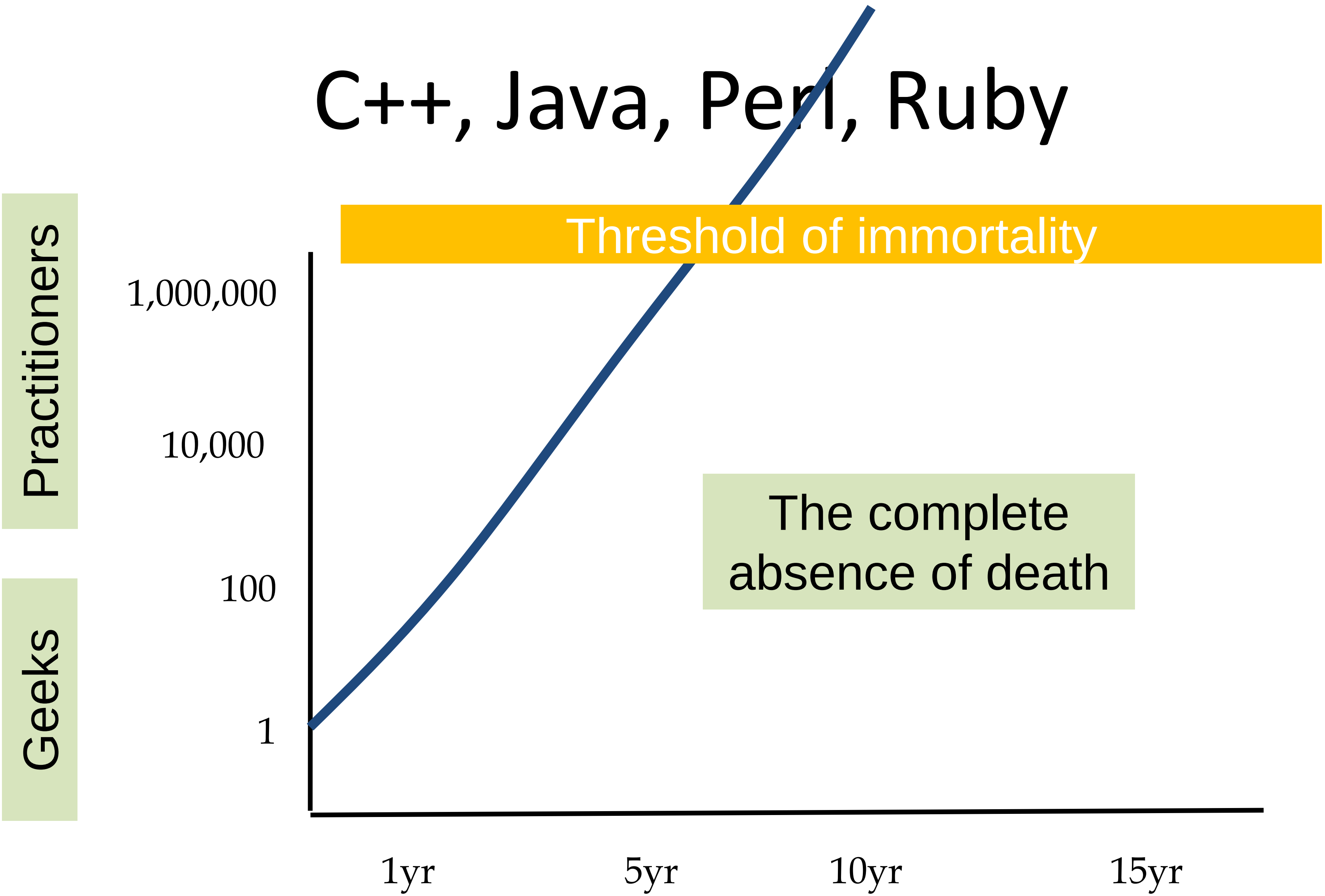
Most Research Languages



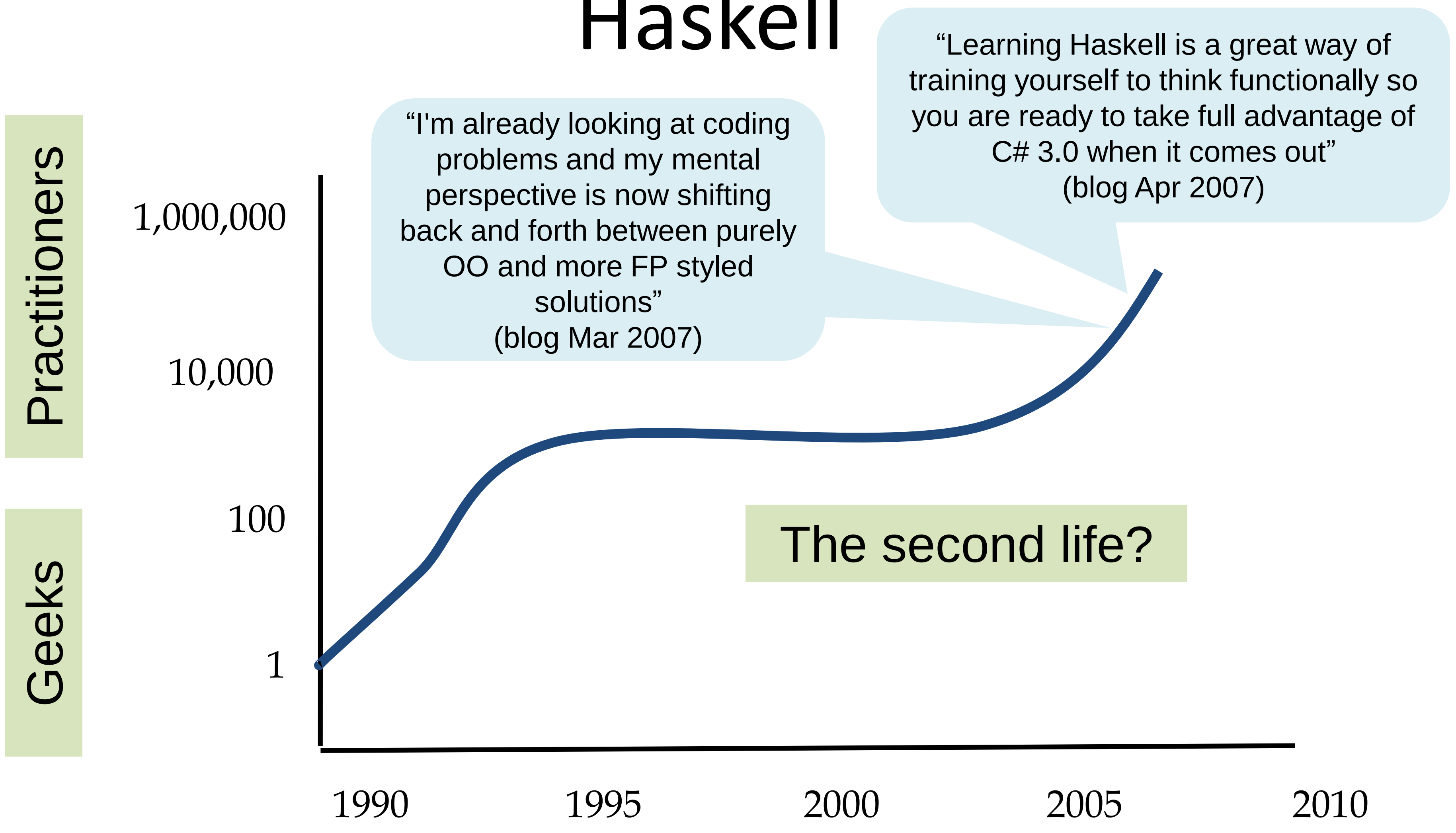
Successful Research Languages



C++, Java, Perl, Ruby



Haskell


Practitioners
Geeks
1,000,000
10,000
100
1
1990
1995
2000
2005
2010
“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)
The second life?

Programming Language Paradigms

- Imperative
  - Algol, PL1, Fortran, Pascal, Ada, Modula, and C
  - Closely related to “von Neumann” Computers
- Object-oriented
  - Simula, Smalltalk, Modula3, C++, Java, C#, Python
  - Data abstraction and ‘evolutionary’ form of program development
    - Class An implementation of an abstract data type (data+code)
    - Objects Instances of a class
    - Fields Data (structure fields)
    - Methods Code (procedures/functions with overloading)
    - Inheritance Refining the functionality of a class with different fields and methods
- Functional
  - Lisp, Scheme, ML, Miranda, Hope, Haskel, OCaml, F#
- Functional/Imperative
  - Rubby
- Logic Programming
  - Prolog

Other Languages

- Hardware description languages
  - VHDL
  - The program describes Hardware components
  - The compiler generates hardware layouts
- Scripting languages
  - Shell, C-shell, REXX, Perl
  - Include primitives constructs from the current software environment
- Web/Internet
  - HTML, Telescript, JAVA, Javascript
- Graphics and Text processing
  TeX, LaTeX, postscript
  - The compiler generates page layouts
- Domain Specific
  - SQL
  - yacc/lex/bison/awk
- Intermediate-languages
  - P-Code, Java bytecode, IDL, CLR

What make PL successful?

- ~~Beautiful syntax~~
- Good design
- Good productivity
- Good performance
- Safety
- Poretability
- Good environment
  - Compiler
  - Interpreter
- Influential designers
- Solves a need
  - C efficient system programming
  - Javascript Browsers

Instructor’s Background

- First programming language Pascal
- Soon switched to C (unix)
  - Efficient low level programming was the key
  - Small programs did amazing things
- Led a big project was written in common lisp
  - Semi-automatically port low level IBM OS code between 16 and 32 bit architectures
- The programming setting has dramatically changed:
  - Object oriented
  - Garbage collection
  - Huge programs
  - Performance depends on many issues
  - Productivity is sometimes more importance than performance
  - Software reuse is a key

Other Lessons Learned

- Futuristic ideas may be useful problem-solving methods now, and may be part of languages you use in the future
  - Examples
    - Recursion
    - Object orientation
    - Garbage collection
    - High level concurrency support
    - Higher order functions
    - Pattern matching

More examples of practical use of futuristic ideas

- Function passing: pass functions in C by building your own closures, as in STL “function objects”
- Blocks are a nonstandard extension added by Apple to C that uses a lambda expression like syntax to create closures
- Continuations: used in web languages for workflow processing
- Monads: programming technique from functional programming
- Concurrency: atomicity instead of locking
- Decorators in Python to dynamically change the behavior of a function
- Mapreduce for distributed programming

Unique Aspects of PL

- The ability to formally define the syntax of a programming language
- The ability to formally define the semantics of the programming language (operational, axiomatic, denotational)
- The ability to prove that a compiler/interpreter is correct
- Useful concepts: Closures, Monads, Continuations, …

Theoretical Topics Covered

- Syntax of PLs
- Semantics of PLs
  - Operational Semantics
  - λ calculus
- Program Verification
  - Floyd-Hoare style verification
- Types

Languages Covered

- Python (Used but not taught)
- ML (Ocaml)
- Javascript
- Scala
- Go & Cloud computing

Interesting Topics not covered

- Concurrency
- Modularity
- Object orientation
- Aspect oriented
- Garbage collection
- Virtual Machines
- Compilation techniques

Part 1: Principles

Date	Lecture	Targil	Assignment
30/10	Overview	No Targil	
6/11	Syntax of Programming Languages	Recursive Decent Parsing	Ex. 1 – Syntax
13/11	Natural Operational Semantics	=	Ex. 2 – Semantics
20/11	Small Step Operational Semantics (SOS)	=	
27/3	Lambda Calculus	=	
4/12	Typed Lambda Calculus	=	Ex3--– Lambda Calculus
11/12		More lambda calculus	

Part 2: Applications

Date	Lecture	Targil	Assignment
11/12	Basic ML	More lambda calculus	Ex 4– ML Project
18/12	Advanced ML	ML	
25/12	No lecture	ML	
1/1	Type Inference	ML	
8/1	Basic Javascript	Type Inference	Ex. 5– JavaScript Project
15/1	Advanced Javascript	Javascipt	
22/1	Go	Javascript	
29/1	Exam Rehersal	No targil	

Summary

- Learn cool programming languages
- Learn useful programming language concepts
- But be prepared to program
  - Public domain software