What programming languages do you know?
What programming languages do you know?

Some historically interesting and/or currently visible languages:


Don’t forget things like:

- HTML, PHP, other web page description languages
- SQL, other database query languages
- EXCEL formula language
What languages?

Source: spectrum.ieee.org/top-programming-languages-2022
What languages?
What languages?

Source: https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/
For more than half of the fifty years computer programmers have been writing code, O'Reilly has provided developers with comprehensive, in-depth technical information. We've kept pace with rapidly changing technologies as new languages have emerged, developed, and matured. Whether you want to learn something new or need answers to tough technical questions, you'll find what you need in O'Reilly books and on the O'Reilly Network.

This timeline includes fifty of the more than 2500 documented programming languages. It is based on an original diagram created by Éric Lévénez (www.levenez.com), augmented with suggestions from O'Reilly authors, friends, and conference attendees. For information and discussion on this poster, go to www.oreilly.com/go/languageposter.
“Why are coders angry?”

“Programmers...know their position is vulnerable. They get defensive when they hear someone suggest that Python is better than Ruby, because [insert 500-comment message thread here]. Is the next great wave swelling somewhere, and will it wash away Java when it comes? Will Go conquer Python? Do I need to learn JavaScript to remain profitable? Programmers are often angry because they’re often scared. We are, most of us, stumbling around with only a few candles to guide the way. We can’t always see the whole system, so we need to puzzle it out, bit by bit, in the dark.”

- Paul Ford, “What is Code?”, Bloomberg Businessweek, 6/11/15
Learning Objectives

- Know fundamental building blocks and structure of programming languages, and be able to analyze a language into its features.
- Be able to read and manipulate common formalisms for language syntax and semantics.
- Recognize and program in different language styles, including the object-oriented and functional paradigms.
- Understand the role of types in languages and be able to explain how type checking works in general and on specific programs.
- Understand procedural and data abstraction and analyze how they are supported in specific languages.
Course Method

- Conventional survey textbook, with broad coverage of languages
- Organized around key anatomical features of PLs
  - Expressions, control flow, functional abstraction, state, types, objects, modules, ...
- Lab exercises mostly involving implementing interpreters for “toy” languages
- Exercises will use Scala, a modern language that blends the object-oriented (OO) and functional (FP) paradigms
Course Non-Goals

X Teaching you how to program

X Teaching you how to program in Scala
  ✓ although you will learn something about this!

X Surveying the details of lots of languages

X Covering all important programming paradigms
  ✓ e.g. we’ll skip logic programming and concurrency

X Learning how real compilers & interpreters are implemented
“High-level” Programming Languages

Consider a simple algorithm for testing primality.

In Scala, using imperative programming style:

```scala
// return true if n has no divisor in the interval [2,n)
def isPrime(n: Int) : Boolean = {
  for (d <- 2 until n)
    if (n % d == 0)
      return false
  true
}
```
“High-level” Programming Languages

In Scala, using a local recursive function:

```scala
// return true if n has no divisor in the interval [2,n)
def isPrime(n: Int): Boolean = {
    // return true if n has no divisor in the interval [d,n)
def noDivFrom(d: Int): Boolean =
        (d >= n) || (n % d != 0) && noDivFrom(d+1)
    noDivFrom(2)
}
```
In Intel X86 assembler

.globl isprime
isprime:
pushl %ebp ; set up procedure entry
movl %esp,%ebp
pushl %esi
pushl %ebx
movl 8(%ebp),%ebx ; fetch arg n from stack
movl $2,%esi ; set divisor d := 2
cmpl %ebx,%esi ; compare n,d
jge true ; jump if d >= n
loop: movl %ebx,%eax ; set n into ....
cltd ; ... dividend register
idivl %esi ; divide by d
testl %edx,%edx ; remainder 0?
jne next ; jump if remainder non-0
xorl %eax,%eax ; set ret value := false(0)
jmp done
next: incl %esi ; increment d
cmpl %ebx,%esi ; compare n,d
jl loop ; jump if d < n
true: movl $1,%eax ; set ret value := true(1)
done: leal -8(%ebp),%esp ; clean up and exit
popl %ebx
popl %esi
leave
ret
What makes a language “high-level”?
What makes a language “high-level”?

- Complex expressions (arithmetic, logical, …)
- Structured control (loops, conditionals, cases, …)
- Composite types (arrays, records, …)
- Type declarations and type checking
- Multiple data storage classes (global/local/heap/GC?)
- Procedures/functions, with private scope (first class?)
- Non-local control (exceptions, threads, …)
- Data abstraction (ADTs, modules, objects…)

[16]
What does hardware give us?
What does hardware give us?

- Low-level machine instructions
- Control flow based on labels and conditional branches
- Explicit locations (e.g. registers) for values and intermediate results of computations
- Flat memory model
- Explicit memory management (e.g., stacks for procedure local data)
How do we bridge the gap?

Two classic approaches:

- A compiler translates high-level language programs into a lower-level language (e.g. machine code)
How do we bridge the gap?

- Two classic approaches:
  - An interpreter is a fixed program that reads in (the representation of) an arbitrary high-level program and executes it.
How do we bridge the gap?

Two classic approaches:

- A **compiler** translates high-level language programs into a lower-level language (e.g. machine code)

- An **interpreter** is a fixed program that reads in (the representation of) an arbitrary high-level program and executes it

Compilers can generate code that runs much faster than interpreted code.

Interpreters are quicker and easier to write, maintain and **understand**.
Combined approaches

High-level source program

Compiler₁

Intermediate language program

Interpreter

Compiler₂

Low-level target program
Stack machines: an intermediate language

- A stack machine is a simple architecture based on a stack of operand values.
- Each machine instruction pops its operands from the stack and pushes its result back on
- So instructions are very simple, because there’s no need to specify operand locations.
- Often used in abstract machines, such as the Java Virtual Machine (which Scala also uses).
- Often compile from high-level language to stack machine byte code which is then interpreted (or perhaps further compiled to machine code).
Stack machine instructions

Instruction set for a very simple stack machine

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Side effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONST i</strong></td>
<td>$s_1 \ldots s_n$</td>
<td>$i \ s_1 \ldots s_n$</td>
<td>-</td>
</tr>
<tr>
<td><strong>LOAD x</strong></td>
<td>$s_1 \ldots s_n$</td>
<td>$\text{Vars}[x] \ s_1 \ldots s_n$</td>
<td>-</td>
</tr>
<tr>
<td><strong>STORE x</strong></td>
<td>$s_1 \ldots s_n$</td>
<td>$s_2 \ldots s_n$</td>
<td>$\text{Vars}[x] := s_1$</td>
</tr>
<tr>
<td><strong>PLUS</strong></td>
<td>$s_1 \ s_2 \ s_3 \ldots s_n$</td>
<td>$(s_2+s_1) \ s_3 \ldots s_n$</td>
<td>-</td>
</tr>
<tr>
<td><strong>MINUS</strong></td>
<td>$s_1 \ s_2 \ s_3 \ldots s_n$</td>
<td>$(s_2-s_1) \ s_3 \ldots s_n$</td>
<td>-</td>
</tr>
</tbody>
</table>

Here $\text{Vars}[]$ is an auxiliary array mapping variables to values.
Here’s a stack machine program corresponding to the simple statement $c = 3 - a + (b - 7)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Vars[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>CONST 3</td>
<td>3</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>LOAD a</td>
<td>100 3</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>MINUS</td>
<td>-97</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>LOAD b</td>
<td>200 -97</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>CONST 7</td>
<td>7 200 -97</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>MINUS</td>
<td>193 -97</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>PLUS</td>
<td>96</td>
<td>a=100, b=200</td>
</tr>
<tr>
<td>STORE c</td>
<td></td>
<td>a=100, b=200,c=96</td>
</tr>
</tbody>
</table>
Stack machine example

Here’s a stack machine program corresponding to the simple statement \( c = 3 - a + (b - 7) \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST 3</td>
<td>LOAD a</td>
<td>MINUS</td>
</tr>
<tr>
<td>LOAD b</td>
<td>CONST 7</td>
<td>MINUS</td>
</tr>
<tr>
<td>PLUS</td>
<td>STORE c</td>
<td></td>
</tr>
</tbody>
</table>

Is this code sequence unique?

Observe that high-level expressions are more flexible than machine code.
Other themes in the study of programming languages

Paradigms

- Imperative
- Object-oriented
- Functional
- Logic
- Concurrent/Parallel
- Scripting

Language Design Criteria

- Expressiveness
- Efficiency
- Correctness

Scale

- “Programming in the Small”
  - what’s important for $10^2$ lines?
- “Programming in the Large”
  - what’s important for $10^6$ lines?
Course Structure

- Twice-per-week course lectures, live and on Zoom
- Post-class self-study questions (not to hand in)
- Regular reading assignments
- Short on-line quiz each day reading is due
- Weekly lab assignments
  - (Mostly) working with interpreters for “toy” languages that illustrate important language features
  - Implementation in Scala
- You are encouraged to work collaboratively on these assignments (but everyone must submit separately)
- In-class midterm and final
- Overall homework load should be $\leq 10$ hours/week
Books

First edition is available free on line, and is good enough for us.
Grading

- 5% Reading Quizzes
- 45% Weekly Labs
- 20% Midterm (Oct. 26)
- 30% Final exam (Dec. 5)
- Two one-on-one Zoom meetings with instructor are required to pass course
WebLab

- Web-based system for assignments
  - Lab assignments (and reading quizzes) are issued
  - You develop solutions in the embedded editor
    - (or in your preferred stand-alone environment)
  - You test your solutions against your own tests and against (secret) tests we provide
  - *We can help you debug problems via “discussions”*
  - You submit your solutions
  - Your scores are automatically recorded
  - We (usually) publish correct solutions
One-on-one Zoom meetings

- Two **required** meetings with instructor
  - Introductory meeting in first two weeks
  - Second meeting after midterm exam

- About 10 minutes each (15 minute slots)

- Sign-up schedule on course web page

- If these are problematic due to scheduling, technological, or other issues, let instructor know
What to do now:

1. Do post-class self-study questions
   • They can be found on course web page

2. Register to use WebLab
   • See instructions in syllabus

3. Do the assigned reading (Scott 1, 2.1, 6.1) and complete the quiz before Thursday at 4pm
   • Quiz can be found inside WebLab

4. Start working on the first week’s homework assignment, which is due next Tuesday at 4pm
   • The assignment can be found inside WebLab

5. Sign up for first one-on-one Zoom meeting with instructor
   • Sign-up schedule is on course web page