For Ruby (and most other languages we will cover), you are given a ticket. We can’t possibly cover everything you need to know, but we have allowed you admittance to the “show”.

You know how to build a project. You can run hello world. You understand key differences. You know where the documentation is. What you do with your admission ticket is up to you!

Note – for resumes, “exposure” is likely the correct terminology

Chapter 1 - Introduction
Course Motivation

- Why are programming languages the way they are?
- How are particular language features implemented/supported?
- Terminology for communication

Course Motivation cont…

- understand the underlying ideas of the main programming paradigms
- know more about the huge variety of programming languages
- understand how the syntax and semantics of languages can be defined precisely.
- have a deeper understanding of the history and rationale behind languages like C++.
Relationship between thought and language.

- The Sapir-Whorf hypothesis in linguistics states that the **structure of one's native-tongue influences the way one's mind perceives the world.** It has found at best very limited experimental support, at least in its strong form.
- One study has shown that subjects in memory tests are more likely to remember a given color if their native language includes a word for that color.
- Example – if you had no identified concept of recursion, how would that affect the ability to reason about it?
- Array accessing – what is concept?
- First programming language learned affects ability to think in Object Oriented terms

---

Why study programming languages?

- Understanding of terminology
- Increased capacity to express ideas
- Improved background for choosing language
- Increased ability to learn and retain new languages - something to hang on to
- Better understanding of significance of implementation. Efficiency is key – not just ease of programming. Practical.
Why study programming languages? (cont)

- Ability to design new languages - or user interfaces
- Choose among alternative ways to express things
- Make good use of debuggers, assemblers, linkers, and related tools.
- Simulate useful features in languages that lack them. Ex: iterators can be imitated with subroutines and static variables.

Example

- Beginning students – always want to know specific answers: Can I do X? What happens if I do Y?
- Often hadn’t tried the specific test, but could reason about it from general knowledge of implementation.
- Ex: What happens if I try to return a reference to a local variable? Is this a compile time or run time issue? How will the system respond?
Programming languages bridge the human-computer semantic gap

**Human:**
- Interested in modeling the real world
- More interested in *what* computer should do than *how*

**Computer:**
- Only data it can manipulate is sequences of zeros and ones.
- Understands low-level “how” instructions.

---

**What are programming languages...**

High-level languages bridge the human-computer semantic gap by providing a higher level notation that can still be executed by computer

**Definition:** A programming language is a notational system for describing computation in machine-readable and human-readable form.
Computation:

- Described by a **Turing Machine** - a very simple computer that can carry out all known computations (albeit not very efficiently).
- A programming language is **Turing complete** if it can be used to describe any computation performed by a Turing Machine.

Turing Machine – 1936 Alan Turing

- "If your **state** is #42 and the symbol you see is a '0' then replace this with a '1', move one symbol to the right, and assume state #17 as your new state."
- A Turing machine consists of:
  - A **tape** which is divided into cells. Each cell contains a symbol from some finite alphabet. The alphabet contains a special **blank** symbol and one or more other symbols. The tape is assumed to be arbitrarily extendible to the left and to the right,
  - A **head** that can read and write symbols on the tape and move left and right.
  - A **state register** stores the state of the Turing machine.
  - An **action table** (or transition function) that tells the machine what symbol to write, how to move the head ('L' for one step left, and 'R' for one step right) and what its new state will be, given the symbol it has just read on the tape and the state it is currently in.
What is needed for Turing completeness?

- integer variables
- arithmetic
- sequentially execution of statements, which include assignment, selection (if) and loop (while) statements.

What is needed for a programming language?

**Machine-readability:**

- Basically, the existence of a (more or less) linear-time translation algorithm.
  - Usually boils down to:
    - The syntax must be given by a context-free grammar.

We will discuss context-free grammars. Basically, the language must have rules which aid parsing.
What is needed for a programming language?

**Human-readability:**
- This is the *real* issue!
- Virtually all the complex details of a programming language are there to (supposedly) enhance human readability.
- Still not very well understood.
- Is strongly dependent on good choice of abstractions.
  - Abstraction is the result of generalization by reducing the information content of a concept, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to a ball retains only the information on general ball attributes and behavior.

What about human “writability??”

- Aren’t programming languages there to promote the *writing* of programs, not the *reading* of them?
- Readability is the real goal: many people are going to have to read your program after you have written it.
  - What is the relationship between readability and writability?
Abstractions:

<table>
<thead>
<tr>
<th>Simple</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>int, char</td>
</tr>
<tr>
<td>Control</td>
<td>goto, =</td>
</tr>
</tbody>
</table>

Computational Paradigms

- Programming languages began by imitating the operations of a computer.
- It is not surprising that the kind of computer for which they were written had significant effect on their design.
  - variables representing memory
  - assignment to change values
  - sequential execution of statements
Language Paradigms:

- **Imperative (procedural):** traditional sequential programming (passive data, active control). Characterized by variables, assignment, and loops.
  - Von Neumann: C, Ada, Fortran
  - Scripting: Perl, Ruby, Python, PHP
  - Object Oriented: Smalltalk, Eiffel, Java
- **Declarative:** what the computer is to do not how.
  - Functional: Lisp/Scheme, ML, Haskell
  - Dataflow (parallel): Id, Val
  - Logic, constraint-based: Prolog, spreadsheets
  - template based: XSLT
  - Regular expressions

Von Neumann Languages

- **The isomorphism between von Neumann programming languages and architectures is in the following manner:**
  - program variables ↔ computer storage cells
  - control statements ↔ computer test-and-jump instructions
  - assignment statements ↔ fetching, storing instructions
  - expressions ↔ memory reference and arithmetic instructions
Within the declarative and imperative families there are subclasses. Categories overlap (like saying lacrosse is a team sport or a racket sport, but not all team sports have rackets and not all racket sports have teams).

- **Von Neumann** – computation based on modification of stored variables (Fortran, C, Java...)
- **Functional**: passive data, but no sequential control; all action by function evaluation (“call”), particularly recursion. No local variables! Similar to mathematics. Ex. Haskell. A program is considered a function from its inputs to its outputs.

---

**Language Paradigms:**

Example Haskell (functional style):

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

square x = x * x

squarelist lis =
  if (null lis) then lis
  else square(head lis):squarelist (tail lis)
```

---
Language Paradigms (cont.):

- **Logic**: Assertions are the basic data; logic inference the basic control. Again, no sequential operation. Similar to mathematics. Ex Prolog
  
  Ex: I am your sister if I am female and we have common parents.

- **Dataflow**: Data flows to operations – inherently parallel model. Operations are triggered by arrival of data.

- **Scripting** – (subset of Von Neumann) – glue components developed as independent programs.

Languages and paradigms

- **Imperative**: C, Pascal, core Ada, FORTRAN

- **Functional**: Lisp (Scheme), ML, Haskell

- **Object-oriented**: C++, Java, Smalltalk, Ruby

- **Logic**: Prolog

- **Parallel**: Java (threads), Ada (tasks)

- **Scripting**: job control languages, javascript, perl, Ruby, shellscript