Programming Languages

- *Programming languages* bridge the gap between people and machines; for that matter, they also bridge the gap among people who would like to share algorithms in a way that immediately allows others to study and execute those algorithms on a machine
- Put another way, the area of programming languages is where these ideas meet:
    - The way a machine operates at a "low level"
    - The algorithms that people design
    - The desire to get machines to perform these algorithms

Programming Language Evolution

- Recall that computers only really understand bits; a *machine language* defines the basic operations that a computer can perform
- Over time, we've tried to make things easier for people to express programs:
    - *Assembly language* — Replaces numeric instructions with "mnemonic" abbreviations (e.g., *mov* instead of 0100; *addi* instead of 0101)
    - *"High level" languages* — Adds more concepts that are closer to how we (people) understand, represent, and express our ideas
- The search hasn't ended — we keep looking for easier and more reliable ways to express algorithms

From Language to Bits

- So how do we get from this:

```
var total = 0;
var n = 252;
var i = 0;
while (i < n) {
     total = total + i;
     i = i + 1;
}
```

- …to this (using a hypothetical machine language where *1* is "assign," *b* is "branch," *a* is "add," and *2* is "jump"):

```
1500 16fc 1700 b476 aa57 ab71 2ffe 5000
```

- As a reminder, hex is just a "more compact" binary:

```
0001 0101 0000 0000 0001 0110 1111 1100 0001 0111 0000 0000 1011 0011 0111 0110
1010 1010 0101 0111 1010 1011 0111 0001 0010 1111 1111 1110 0101 0000 0000 0000
```

- Central to programming languages is the need to *translate* the human-readable program (called *source code*) into a computer-readable (and runnable!) form
- There are two primary approaches to this translation:
  - ◇ With *compilation*, a translator processes the source code once, and produces an *executable* that users then invoke — once the executable is produced, the source code is no longer needed, at least until it's time to modify or improve the program
  - ◇ With *interpretation*, a translator processes *and* performs the source code "as it goes" — in a sense, the "executable" is determined along the way, and isn't saved into a separate form (JavaScript is interpreted, for example: the browser serves as its *interpreter*)

Machine or Platform Independence

- You may have noticed that JavaScript is not married to a specific type of machine or operating system — this is because most *platforms* have a Web browser that can interpret JavaScript
- This is a nice side effect of programming language implementation — it opens the possibility of *machine* or *platform independence*: the ability for the same source code to work well on different devices
- Note how machine or assembly language isn't that way: it *has* to run on the CPU for which it was written

- Interpreted languages attain platform independence when interpreters are available for multiple devices — how about compiled languages, which produce *executables* for machines to read and run directly?
- There's actually a "little white lie" in the language translation example from the previous page: the translated version of a program doesn't immediately have to be CPU-level machine code
- Many languages compile their source code into intermediate *byte code* or *p-code* — instead of a "hardware machine" language, this can be viewed as a "software machine" language
- This way, a software or *virtual* machine can be used to execute the code — this is how Java is implemented

Programming Paradigms

- Somewhat analogously to how writers, artists, and musicians belong to certain "styles" or "genres," it turns out that programming languages also tend to favor certain approaches or philosophies
- These approaches are referred to as programming *paradigms* — and so far, we have actually only seen one such paradigm: the *imperative* or *procedural paradigm*
- Imperative or procedural programming views a program as a series of commands — "do this, then do that" — that affect some state-of-the-world

- Not surprisingly, the imperative approach is one of the oldest paradigms, since machines, at the physical level, are ultimately built that way: recall *fetch-decode-execute*
- It turns out that many other paradigms have evolved over the years, allowing us to write programs in ways other than "do this, then do that:"
  - ◇ *Declarative paradigm* — Perhaps the closest thing to an "opposite" of imperative programming, declarative programming focuses on stating "what problem to solve" instead of "how to solve the problem"
    - This paradigm requires a *general-purpose algorithm* for solving a given problem; while there is no such thing for solving *any possible* problem, there *are* general algorithms for *special classes* of problems, such as database retrieval, simulation, and formal logic
  - ◇ *Functional paradigm* — This paradigm views a program as a set of self-contained units, each of which takes some input and produces some output; instead of "doing" things, this paradigm "evaluates" or "calculates" things
  - ◇ *Object-oriented (OO) paradigm* — This paradigm represents a program as a set of *objects* that interact with each other through their *methods*; it's the reigning industry paradigm as of now, with languages like Java, JavaScript, C#, and Objective-C all having OO capabilities

Programming Concepts

- It should be no surprise if the following ideas sound somewhat familiar — we've seen them all before
- This time, however, we address them from the perspective of how these constructs are common across many programming languages, though they may look different from one language to another
- This is ultimately the secret to being multilingual — the ability to recognize ideas that are common from one language to another, even if those ideas are written or expressed in a different way

- *Literals* — These are the "directly-typed" values in a language: numbers, text, booleans; some languages allow more complex literals such as arrays, maps, and objects
- *Variables and assignments* — Of course, once we've specified some values, we want to manipulate them; to do this, we *assign* them to the named "boxes" that are variables
- *Expressions* — A key part of manipulating values is putting them together; these are *expressions*, which can vary widely in complexity (and readability!):
    - `a + b`
    - `factorial(n)`
    - `str[i++].substring(mark, length * 2)`
    - `((*handle)->location.x - screen->dimensions.x) / 2`
- *Control flow* — Many algorithms require the ability to selectively control what to do next (as opposed to non-changing, linear sequences of activities); these elements fall under the category of *control flow*, and include conditional statements, loop constructs
- *Data types* — The recurring theme of finding ways to make digital data (bits) easier for people to understand leads to the concept of *data types*, or how some set of bits must be interpreted: whether bits are interpreted as numbers, other times as text, and so on
- *Procedures and functions* — Technically a part of control flow, but complex enough to merit their own mention, procedures and functions (or subroutines) allow us to "abstract" common activities under a single, reusable, named block of code
- *Modules* — "Old code never dies, they just get placed in modules:" the *module* constructs in many languages allow us to organize our code into self-contained units, for easy reuse and *encapsulation* of "private" code from the rest of a program

Data Structures

- Where assignments, procedures, functions, loops, conditionals, etc. constitute the "verbs" in a program, its variables and data types form its "nouns"
- Note how many of these "verbs" aren't really directly understood by an underlying CPU; a programming language implementation *translates* these higher-level concepts into machine instructions
- The same is true for the "nouns" used by a programming language: *data structures* allow us to envision entities beyond registers and main memory

- Note how we have already gone beyond simple integer or text values: for example, we could not reason about *search* and *sort* without a *list* or *array* data structure
- Many problems are difficult to think about without having a proper data structure:
  - The *descendants* algorithm (or anything that has to do with hierarchies, for that matter) requires that we can assign *ancestors* or *children* to items in our program
  - A *route calculator* (such as those on various map Web sites) requires a representation that can connect items and attach values (like distances) to them
  - *Server* algorithms like a print scheduler or database need to place "clients" in a line or a *queue* so that we know "who's up next" for service

A Data Structure Catalog

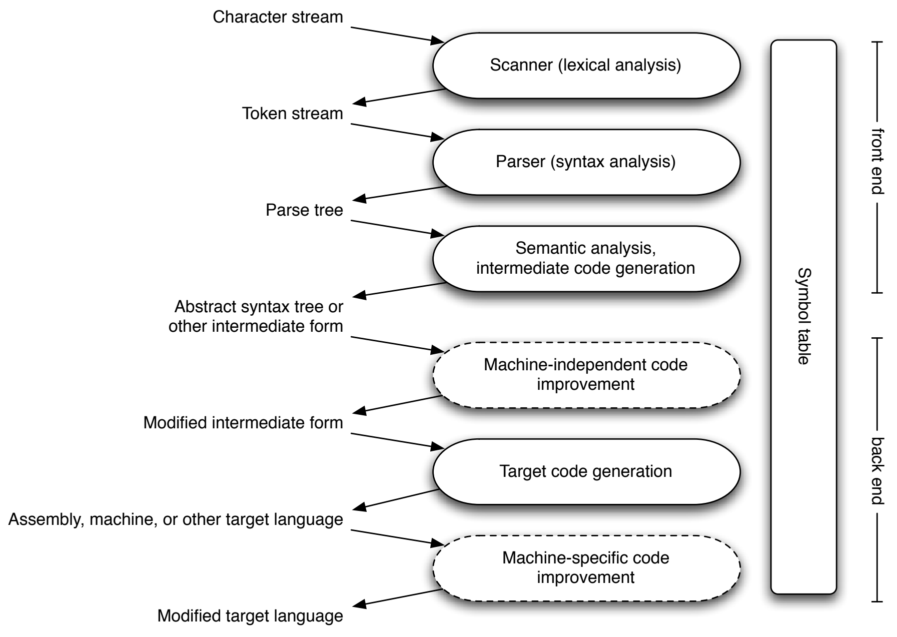
- *Arrays* — Ordered lists of items, typically the same type (e.g., an array of integers, an array of movies, etc.) and usually accessed with an integer index (e.g., *a*[3])
- *Records* — Composite items, consisting of named fields or properties (e.g., an *Element* record consisting of *atomicWeight* and *atomicNumber*)
- *Maps* (a.k.a. *dictionaries*, *hash tables*) — Sets of "keys" that have matching "values" (the way dictionary words have matching definitions): note how a map whose keys are integers resembles an array

- *Trees* (a.k.a. hierarchies) — Items with zero or more children; the "top" item (the one with no ancestors) is the *root*; note how every *subtree* in a tree is also a tree
- *Queues* — Structurally similar to an array (queues are ordered sequences of items), but with specific *first-in-first-out* (FIFO) behavior: items are always added to the "tail" of the queue and items are taken from its "head"
- *Stacks* — Also similar to array, but now with *last-in-first-out* (LIFO) behavior: needed for undo/redo; correct subroutine behavior relies on stacks too
- *Graphs* — Items that are connected to each other via *edges*, usually with *values* attached to each edge: used by "driving directions" algorithms, plus seminal problems such as the *traveling salesman problem*

Implementation Details

- If you think about it, the process of compiling or interpreting a programming language is yet another algorithm — albeit one of the most complex and sophisticated ones around
- This subject of *compiler construction* is one of the final highlights of an undergraduate computer science education, and can be specialized upon even further in both graduate school and industry
- We won't say much more about it here, except to provide an overview of what a compiler does

- No need to sweat the details — this is just meant to give you an idea of how a genuinely complex algorithm may look like:


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis, intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement
Modified intermediate form
Target code generation
Assembly, machine, or other target language
Machine-specific code improvement
Modified target language
Symbol table
front end
back end