CMSC 330: Organization of Programming Languages

Administrivia

Course Goals

- Understand why there are so many languages
- Describe and compare their main features
- Choose the right language for the job
- Write better code faster
  - Code that is shorter, more efficient, with fewer bugs
  - In a variety of styles
- In short:
  - Become a better programmer with a better understanding of your tools.

Course Activities

- Learn different **types of languages**
- Learn different **language features** and tradeoffs
  - Programming patterns repeat between languages
- Study how languages are **specified**
  - **Syntax**, **Semantics** — mathematical formalisms
- Study how languages are **implemented**
  - Parsing via **regular expressions** (automata theory) and **context free grammars**
  - Mechanisms such as **closures**, **tail recursion**, **lazy evaluation**, **garbage collection**, …
- Language impact on **computer security**

Syllabus

- Dynamic/ Scripting languages (Ruby)
- Functional programming (OCaml)
- Scoping, type systems, parameter passing
- Regular expressions & finite automata
- Context-free grammars & parsing
- Lambda Calculus
- Safe, “zero-cost abstraction” programming (Rust)
- Secure programming
- Comparing language styles; other topics

Calendar / Course Overview

- Tests
  - 4 quizzes, 2 midterm exams, 1 final exam
- Clicker quizzes
  - In class, graded, during the lectures
- Projects
  - Project 1 – Ruby
  - Project 2-4 – OCaml (and parsing, automata)
    - P2 and P4 are split in two parts
  - Project 5 – Rust
  - Project 6 – Security

Clickers

- Turning Technology subscription is free. Physical clicker is preferred.
  - Clicker device: any of LCD, NXT, or QT2 models
  - Phone App: needs wifi

Quiz time!

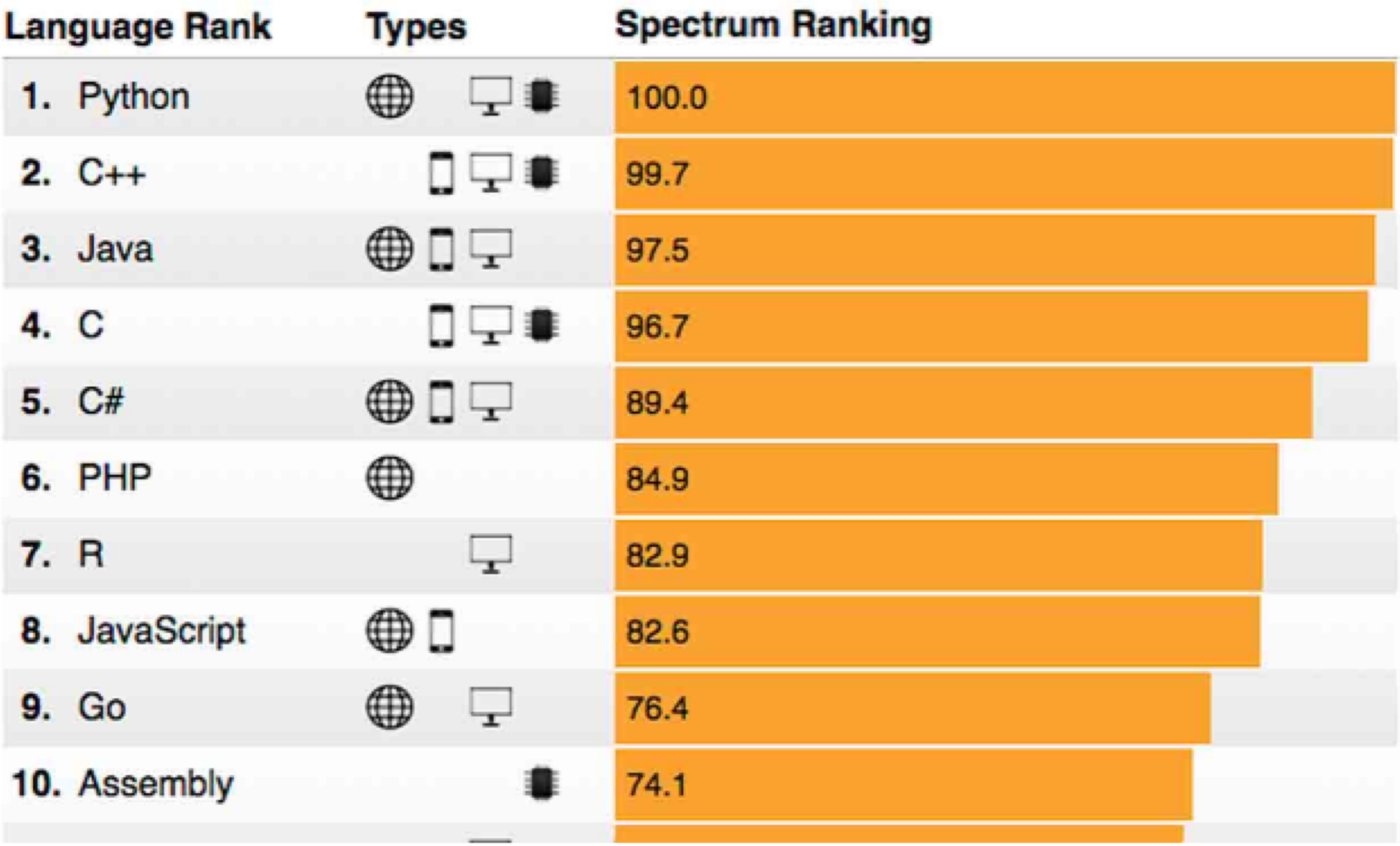
- According to IEEE Spectrum Magazine which is the “top” programming language of 2018?

  A. Java
  B. R
  C. Python
  D. C++

Quiz time!

- According to IEEE Spectrum Magazine which is the “top” programming language of 2018?

  A. Java
  B. R
  C. Python
  D. C++



Python has maintained its grip on the No. 1 spot. Last year it came out on top by just barely beating out C, with Python's score of 100 to C's 99.7. Now C++ is the language nipping at its heels at with a 99.7 score, while C moves down to fourth place at 96.7.

Discussion Sections

- Discussion sections will deepen understanding of concepts introduced in lecture
  - Discussions are smaller, more interactive
- Oftentimes discussion section will consist of programming exercises
  - Bring your laptop to discussion
  - Be prepared to program: install the language in question on your laptop, or remote shell into Grace
- There will also be be quizzes, and some lecture material in discussion sections
  - Quizzes cover non-programming parts of the class

Project Grading

- You have accounts on the Grace cluster
- Projects will be graded using the submit server
  - Software versions on these machines are canonical
- Develop programs on your own machine
  - Generally results will be identical on Dept machines
  - Your responsibility to ensure programs run correctly on the grace cluster
- See web page for Ruby, OCaml, etc. versions we use, if you want to install at home
  - Or install our Linux VM, which has them all

Rules and Reminders

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- Keep ahead of your work
  - Get help as soon as you need it
    - Office hours, Piazza (email as a last resort)
- Don't disturb other students in class
  - Keep cell phones quiet
  - No laptops / tablets in class
    - Prefer hand-written notes (else, sit in back of class)

Academic Integrity

- All written work (including projects) must be done on your own
  - Do not copy code from other students
  - Do not copy code from the web
  - Do not post your code on the web
- Cheaters are caught by auto-comparing code
- Work together on *high-level* project questions
  - Do not look at/describe another student's code
  - If unsure, ask an instructor!
- Work together on practice exam questions

CMSC 330: Organization of Programming Languages

Overview

Plethora of programming languages

- We saw APL. How about

- LISP:
```
(defun double (x) (* x 2))
```
- Prolog:
```
size([],0).
size([H|T],N) :-
     size(T,N1), N is N1+1.
```
- Ocaml:
```
List.iter
(fun x -> print_string x)
     [“hello, ”; s; "!\n”]
```
- Smalltalk:
```
( #( 1 2 3 4 5 )
     select:[:i | i even ] )
```

All Languages Are (kind of) Equivalent

- A language is Turing complete if it can compute any function computable by a Turing Machine
  - *Church-Turing thesis* (1936): Computability by a Turing Machine *defines* "effectively computable"
- Essentially all general-purpose programming languages are Turing complete
  - I.e., any program can be written in any programming language
- Therefore this course is useless?!
  - Learn one programming language, always use it

Studying Programming Languages

- Will make you a better programmer
  - Programming is a human activity
    - Features of a language make it easier or harder to program for a specific application
  - Ideas or features from one language translate to, or are later incorporated by, another
    - Many “design patterns” in Java are functional programming techniques
  - Using the right programming language or style for a problem may make programming
    - Easier, faster, less error-prone

Studying Programming Languages

- Become better at learning new languages
  - A language not only allows you to express an idea, it also shapes how you think when conceiving it
    - There are some fundamental computational paradigms underlying language designs that take getting used to
  - You may need to learn a new (or old) language
    - Paradigms and fads change quickly in CS
    - Also, may need to support or extend legacy systems

Changing Language Goals

- 1950s-60s – Compile programs to execute efficiently
  - Language features based on hardware concepts
    - Integers, reals, goto statements
  - Programmers cheap; machines expensive
    - Computation was the primary constrained resource
    - Programs had to be efficient because machines weren't
      - Note: this still happens today, just not as pervasively

Changing Language Goals

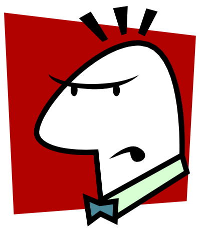
- Today
  - Language features based on design concepts
    - Encapsulation, records, inheritance, functionality, assertions
  - Machines cheap; programmers expensive
    - Scripting languages are slow(er), but run on fast machines
    - They've become very popular because they ease the programming process
  - The constrained resource changes frequently
    - Communication, effort, power, privacy, …
    - Future systems and developers will have to be nimble

Language Attributes to Consider

- Syntax
  - What a program looks like
- Semantics
  - What a program means (mathematically)
- Paradigm and Pragmatics
  - How programs tend to be expressed in the language
- Implementation
  - How a program executes (on a real machine)

Syntax

- The keywords, formatting expectations, and "grammar" for the language
  - Differences between languages usually superficial
    - C / Java	if (x == 1) { … } else { … }
    - Ruby	if x == 1 … else … end
    - OCaml	if (x = 1) then … else …

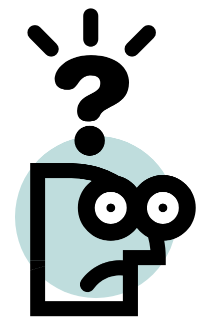


  - Differences initially annoying; overcome with experience
- Concepts such as regular expressions, context-free grammars, and parsing handle language syntax

Semantics

- What does a program *mean*? What does it *do*?
  - Same syntax may have different semantics in different languages!

	Physical Equality	Structural Equality
Java	a == b	a.equals(b)
C	a == b	*a == *b
Ruby	a.equal?(b)	a == b
OCaml	a == b	a = b



- Can specify semantics informally (in prose) or formally (in mathematics)

Why **Formal** Semantics?

- Textual language definitions are often incomplete and ambiguous
  - Leads to two different implementations running the same program and getting a different result!
- A formal semantics is basically a mathematical definition of what programs do
  - Benefits: concise, unambiguous, basis for proof
- We will consider operational semantics
  - Consists of rules that define program execution
  - Basis for implementation, and proofs that programs do what they are supposed to

Paradigm

- There are many ways to compute something
  - Some differences are superficial
    - For loop vs. while loop
  - Some are more fundamental
    - Recursion vs. looping
    - Mutation vs. functional update
    - Manual vs. automatic memory management
- Language's paradigm favors some computing methods over others. This class:

  - Imperative
  - Functional
  - Resource-controlled (zero-cost)
  - Scripting/dynamic

Imperative Languages

- Also called procedural or von Neumann
- Building blocks are procedures and statements
  - Programs that write to memory are the norm

```
int x = 0;
while (x < y) x = x + 1;
```

  - FORTRAN (1954)
  - Pascal (1970)
  - C (1971)

Functional (Applicative) Languages

- Favors immutability
  - Variables are never re-defined
  - New variables a function of old ones (exploits recursion)
- Functions are higher-order
  - Passed as arguments, returned as results

  - LISP (1958)
  - ML (1973)
  - Scheme (1975)
  - Haskell (1987)
  - OCaml (1987)

OCaml

- A mostly-functional language
  - Has objects, but won't discuss (much)
  - Developed in 1987 at INRIA in France
  - Dialect of ML (1973)
- Natural support for pattern matching
  - Generalizes `switch`/`if-then-else` – very elegant
- Has full featured module system
  - Much richer than interfaces in Java or headers in C
- Includes type inference
  - Ensures compile-time type safety, no annotations

A Small OCaml Example

intro.ml:

```
let greet s =
  List.iter (fun x -> print_string x)
    [“hello, ”; s; "!\n”]
```

```
$ ocaml
        Objective Caml version 3.12.1

# #use "intro.ml";;
val greet : string -> unit = <fun>
# greet "world";;
Hello, world!
- : unit = ()
```

Dynamic (Scripting) Languages

- Rapid prototyping languages for common tasks
  - Traditionally: text processing and system interaction
- "Scripting" is a broad genre of languages
  - "Base" may be imperative, functional, OO…
- Increasing use due to higher-layer abstractions
  - Originally for text processing; now, much more

  - sh (1971)
  - perl (1987)
  - Python (1991)
  - Ruby (1993)

```
#!/usr/bin/ruby
while line = gets do
   csvs = line.split /,/
   if(csvs[0] == "330") then
   ...
```

Ruby

- An imperative, object-oriented scripting language
  - Full object-orientation (even primitives are objects!)
  - And functional-style programming paradigms
  - Dynamic typing (types hidden, checked at run-time)
  - Similar in flavor to other scripting languages (Python)
- Created in 1993 by Yukihiro Matsumoto (Matz)
  - “Ruby is designed to make programmers happy”
- Core of Ruby on Rails web programming framework (a key to its popularity)

A Small Ruby Example

intro.rb:

```
def greet(s)
  3.times { print “Hello, ” }
  print “#{s}!\n”
end
```

```
% irb      # you’ll usually use “ruby” instead
irb(main):001:0> require "intro.rb"
=> true
irb(main):002:0> greet("world")
Hello, Hello, Hello, world!
=> nil
```

Theme: Software Security

- Security is a big issue today
- Features of the language can help (or hurt)
  - C/C++ lack of memory safety leaves them open for many vulnerabilities: **buffer overruns**, **use-after-free** errors, **data races**, etc.
  - Type safety is a big help, but so are abstraction and isolation, to help enforce security policies, and limit the damage of possible attacks
- Secure development requires vigilance
  - Do not trust inputs – unanticipated inputs can effect surprising results! Therefore: verify and sanitize

Zero-cost Abstractions in Rust

- A key motivator for writing code in C and C++ is the low (or zero) cost of the abstractions use
  - Data is represented minimally; no metadata required
  - Stack-allocated memory can be freed quickly
  - Malloc/free maximizes control – no GC or mechanisms to support it are needed
- But no-cost abstractions in C/C++ are insecure
- Rust language has safe, zero-cost abstractions
  - Type system enforces use of ownership and lifetimes
  - Used to build real applications – web browsers, etc.

Other Language Paradigms

- We are not covering them all in 330!
- Parallel/concurrent/distributed programming
  - Cilk, Fortress, Erlang, MPI (extension), Hadoop (extension); more on these in CMSC 433
- Logic programming
  - Prolog, λ-prolog, CLP, Minikanren, Datalog
- Object-oriented programming
  - Simula, Smalltalk, C++, Java, Scala
- Many other languages over the years, adopting various styles

Defining Paradigm: Elements of PLs

- Important features
  - Regular expression handling
  - Objects
    - Inheritance
  - Closures/code blocks
  - Immutability
  - Tail recursion
  - Pattern matching
    - Unification
  - Abstract types
  - Garbage collection

- Declarations
  - Explicit
  - Implicit

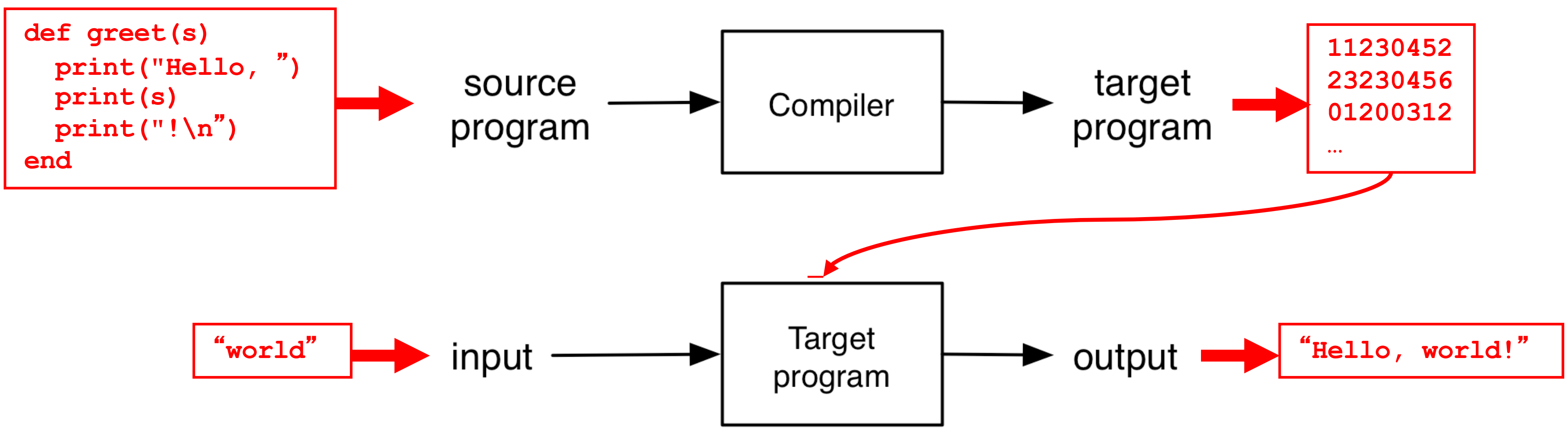
- Type system
  - Static
    - Polymorphism
    - Inference
  - Dynamic
  - Type safety

Implementation

- How do we implement a programming language?
  - Put another way: How do we get program P in some language L to run?

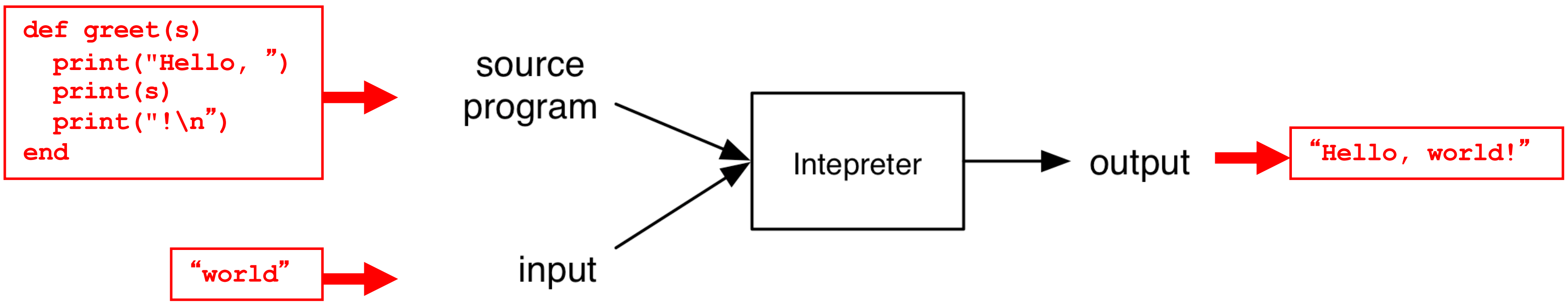
- Two broad ways
  - Compilation
  - Interpretation

Compilation



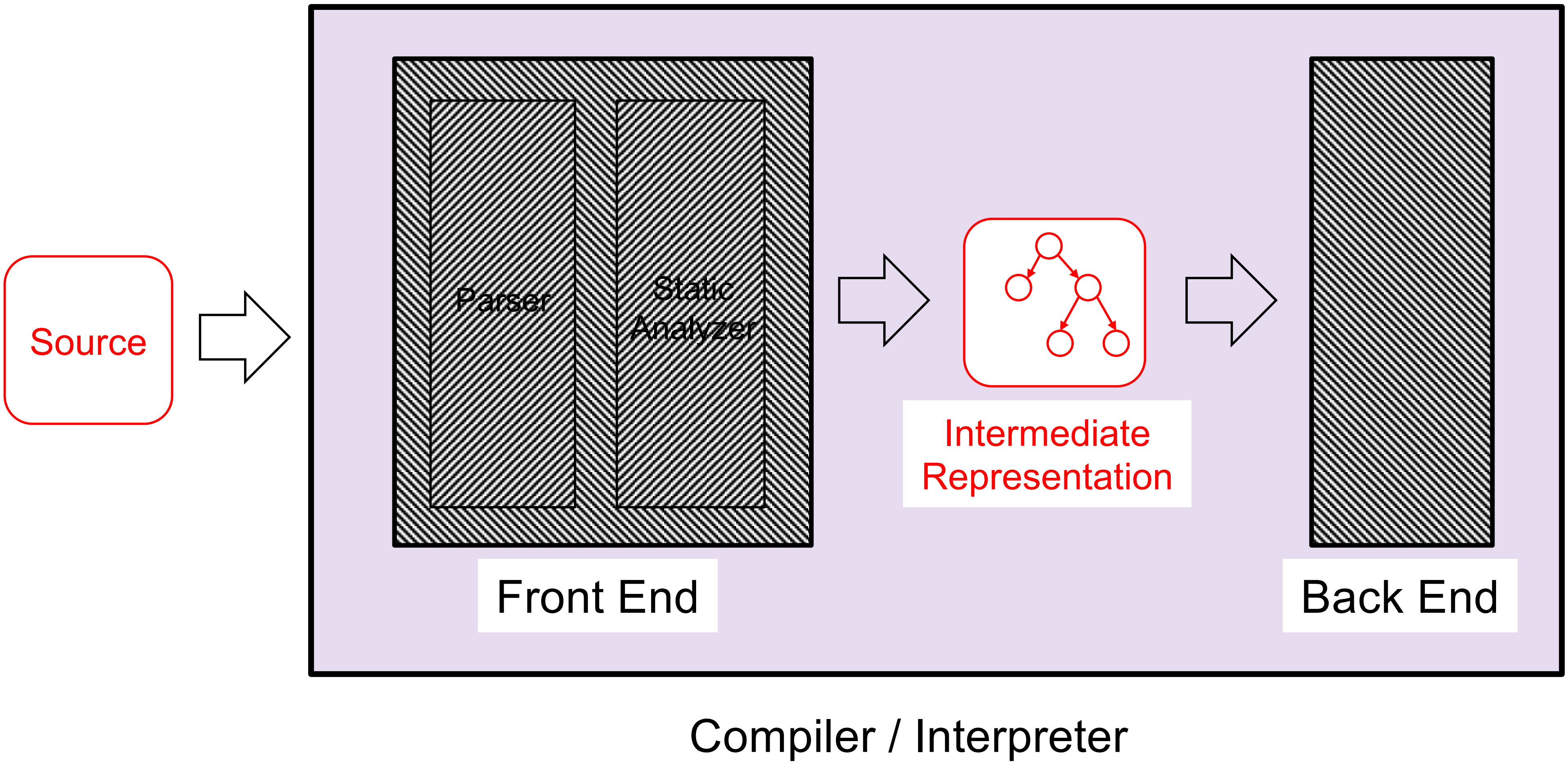
- Source program translated ("compiled") to another language
  - Traditionally: directly executable machine code
  - Generating code from a higher level "interface" is also common (e.g., JSON, RPC IDL)

Interpretation



- Interpreter executes each instruction in source program one step at a time
  - No separate executable

Architecture of Compilers, Interpreters



Front Ends and Back Ends

- Front ends handle syntax
  - Parser converts source code into intermediate format ("parse tree") reflecting program structure
  - Static analyzer checks parse tree for errors (e.g., erroneous use of types), may also modify it
    - What goes into static analyzer is language-dependent!
- Back ends handle semantics
  - Compiler: back end ("code generator") translates intermediate representation into "object language"
  - Interpreter: back end executes intermediate representation directly

Compiler or Intepreter?

- gcc
  - Compiler – C code translated to object code, executed directly on hardware (as a separate step)
- javac
  - Compiler – Java source code translated to Java byte code
- java
  - Interpreter – Java byte code executed by virtual machine
- sh/csh/tcsh/bash
  - Interpreter – commands executed by shell program

Compilers vs. Interpreters

- Compilers
  - Generated code more efficient
  - "Heavy"
- Interpreters
  - Great for debugging
  - Fast start time (no compilation), slow execution time
- In practice
  - "General-purpose" programming languages (e.g., C, Java) are often compiled, although debuggers provide interpreter support
  - Scripting languages languages are often interpreted, even if general-purpose

Summary

- Programming languages vary in their
  - Syntax
  - Semantics
  - Style/paradigm and pragmatics
  - Implementation
- They are designed for different purposes
  - And goals change as the computing landscape changes, e.g., as programmer time becomes more valuable than machine time
- Ideas from one language appear in others