

CMSC 330: Organization of Programming Languages

Administrivia

Course Goal

Learn how programming languages work

- Languages you know:
 - C,C++,Java, MIPS, Ruby, Python, R, PHP, ...
- Why are there so many programming languages?
 - not every language is perfect for every task
 - new programming paradigm
 - advances in hardware

Course Goal

Learn how programming languages work

- ▶ Broaden your language horizons
 - Different programming languages
 - Different language features and tradeoffs
 - Useful programming patterns
- ▶ Study how languages are described / specified
 - Mathematical formalisms
- ▶ Study how languages are implemented
 - What **really** happens when I write `x.foo(...)`?
 - (CMSC 430 goes much further)

Our new language: umd

File: while.umd

Only one
function:UMD



```
int UMD() {  
    int umd_a;  
    umd_a=10;  
    int umd_b;  
    umd_b=1;  
    while(umd_b<umd_a) {  
        printf(umd_b);  
        umd_b =umd_b+1;  
    }  
}
```

All variabe names
start with umd_



Course Subgoals

- ▶ Learn some fundamental programming-language concepts
 - Regular expressions
 - Automata theory
 - Context free grammars
 - Computer security
- ▶ Improve programming skills
 - Practice learning new programming languages
 - Learn how to program in a new style

Syllabus

- ▶ Dynamic/ Scripting languages (Ruby)
- ▶ Functional programming (OCaml)
- ▶ Scoping, type systems, parameter passing
- ▶ Regular expressions & finite automata
- ▶ Context-free grammars & parsing
- ▶ Lambda Calculus
- ▶ Logic programming (Prolog)
- ▶ Secure programming
- ▶ Comparing language styles; other topics

Calendar / Course Overview

- ▶ Tests
 - 4 quizzes, 1 midterm exam, 1 final exam
- ▶ Clickers Quizzes
 - In class, combined with lectures
- ▶ Projects
 - Project 1 – Ruby
 - Project 2-4 – OCaml (and parsing, automata)
 - Project 5 – Prolog
 - Project 6 – Security ? **(Maybe optional)**

Clickers

- ▶ Turning Technology clicker is **required**
 - Clicker ISBN is posted on the course webpage
 - You can get any of LCD, NXT, or QT2 models



Discussion Sections

- ▶ Lectures introduce the course content
- ▶ Discussion sections will deepen understanding
 - These are smaller, and thus can be 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

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, SWI-Prolog versions we use, if you want to install at home
 - **We provided a VM.**

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
 - Except for taking notes (please 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
 - We're using Moss; cheaters will be caught
- ▶ 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

All Languages Are (Kind of) Equivalent

- ▶ A language is **Turing complete** if it can compute any function computable by a Turing Machine
- ▶ 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 only 1 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

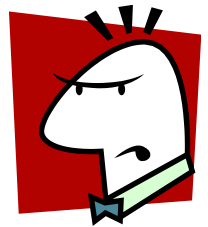
- Language features based on design concepts
 - Encapsulation, records, inheritance, functionality, assertions
- Machines cheap; programmers expensive
 - Scripting languages are slow(er), but run on fast machines
 - They've become very popular because they ease the programming process
- The constrained 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
- ▶ **Paradigm**
 - How programs tend to be expressed in the language
- ▶ **Semantics**
 - What a program means (mathematically)
- ▶ **Implementation**
 - How a program executes (on a real machine)

Syntax

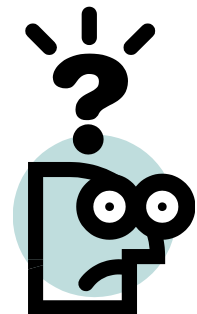
- ▶ The keywords, formatting expectations, and “grammar” for the language
 - Differences between languages usually superficial
 - C / Java `if (x == 1) { ... } else { ... }`
 - Ruby `if x == 1 ... else ... end`
 - OCaml `if (x = 1) then ... else ...`
 - Differences initially annoying; overcome with experience
- ▶ 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 | <code>a == b</code> | <code>a.equals(b)</code> |
| C | <code>a == b</code> | <code>*a == *b</code> |
| Ruby | <code>a.equal?(b)</code> | <code>a == b</code> |
| OCaml | <code>a == b</code> | <code>a = b</code> |



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

Formal (Mathematical) Semantics

- ▶ What do my programs mean?

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

```
let fact n =  
  let rec aux i j =  
    if i = 0 then j  
    else aux (i-1) (j*i) in  
  aux n 1
```

- ▶ Both OCaml functions implement “the factorial function.” How do I know this? Can I prove it?
 - Key ingredient: a mathematical way of specifying what programs do, i.e., their semantics
 - Doing so depends on the semantics of the language

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
 - Logic
 - 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 = ()
```

Logic-Programming Languages

- ▶ Also called **rule-based** or **constraint-based**
- ▶ Program rules constrain possible results
 - Evaluation = constraint satisfaction = search
 - “A :- B” – If B holds, then A holds (“B *implies* A”)
 - `append([], L2, L2) .`
 - `append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs) .`
 - **PROLOG (1970)**
 - Datalog (1977)
 - Various expert systems

Prolog

- ▶ A logic programming language
 - 1972, University of Aix-Marseille
 - Original goal: Natural language processing
- ▶ Rule based
 - Rules resemble pattern matching and recursive functions in Ocaml, but more general
- ▶ Execution = search
 - Rules specify relationships among data
 - Lists, records, “atoms”, integers, etc.
 - Programs are queries over these relationships
 - The query will “fill in the blanks”

A Small Prolog Example

```
/* A small Prolog program */  
  
female(alice).  
male(bob).  
male(charlie).  
father(bob, charlie).  
mother(alice, charlie).  
  
% "X is a son of Y"  
son(X, Y) :- father(Y, X), male(X).  
son(X, Y) :- mother(Y, X), male(X).
```

Lowercase logically terminates

Program consists of facts and rules

Uppercase denotes variables

User types ; to request additional answer

Query

```
?- son(X,Y).  
X = charlie,  
Y = bob;  
X = charlie,  
Y = alice.
```

Multiple answers

User types return to complete request

Object-Oriented Languages

- ▶ Programs are built from objects
 - Objects combine functions and data
 - Often into “classes” which can inherit

```
class C { int x; int getX() {return x;} ... }  
class D extends C { ... }
```
- ▶ “Base” may be either imperative or functional
 - Smalltalk (1969)
 - C++ (1986)
 - OCaml (1987)
 - Ruby (1993)
 - Java (1995)

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
    ...
  end
end
```

Ruby

- ▶ An imperative, object-oriented scripting language
 - 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)
 - Similar in flavor to many other scripting languages
 - Much cleaner than perl
 - Full object-orientation (even primitives are objects!)

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
```

Concurrent / Parallel Languages

- ▶ Traditional languages had one thread of control
 - Processor executes one instruction at a time
- ▶ Newer languages support many threads
 - Thread execution conceptually independent
 - Means to create and communicate among threads
- ▶ Concurrency may help/harm
 - Readability, performance, expressiveness
- ▶ Won't cover in this class
 - Threads covered in 132 and 216; more in 412, 433

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** facilities, 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

Other Languages

- ▶ There are lots of other languages w/ various features
 - COBOL (1959) – Business applications
 - Imperative, rich file structure
 - BASIC (1964) – MS Visual Basic
 - Originally designed for simplicity (as the name implies)
 - Now it is object-oriented and event-driven, widely used for UIs
 - Logo (1968) – Introduction to programming
 - Forth (1969) – Mac Open Firmware
 - Extremely simple stack-based language for PDP-8
 - Ada (1979) – The DoD language
 - Real-time
 - Postscript (1982) – Printers- Based on Forth

Beyond Paradigm

▶ 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

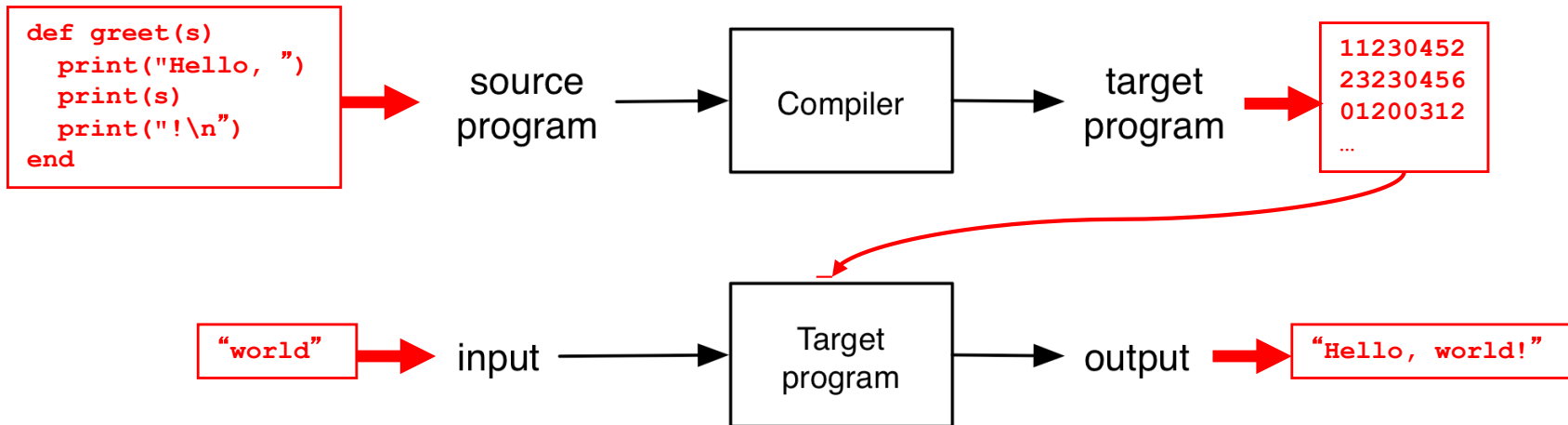
- Static
 - Polymorphism
- Dynamic
- Type safety

Implementation

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

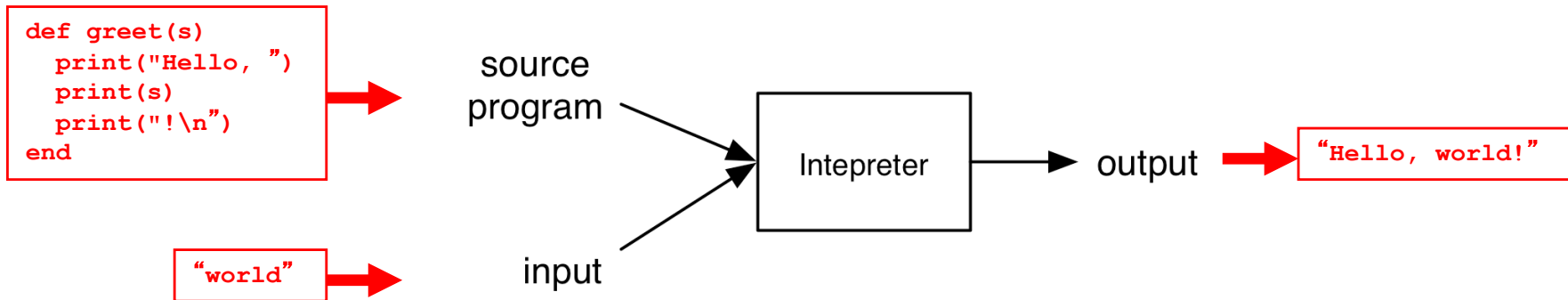
- ▶ Two broad ways
 - Compilation
 - Interpretation

Compilation



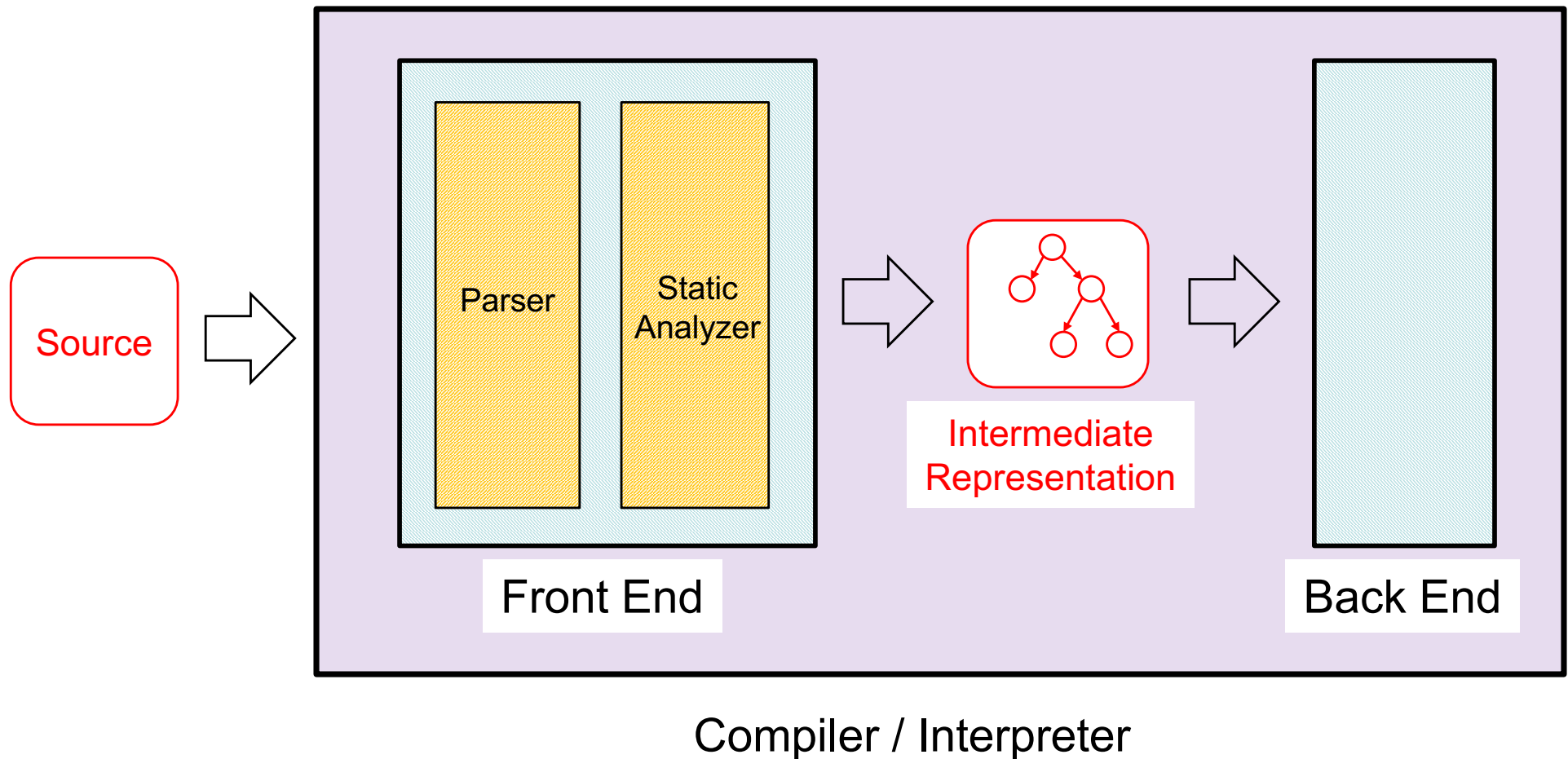
- ▶ Source program translated (“compiled”) to another language
 - Traditionally: directly executable machine code
 - Generating code from a higher level “interface” is also common (e.g., JSON, RPC IDL)

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 Interpreter?

- ▶ **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 and other special-purpose languages are interpreted, even if general purpose

Attributes of a Good Language

- Cost of use
 - Program execution (run time), program translation, program creation, and program maintenance
- Portability of programs
 - Develop on one computer system, run on another
- ▶ Programming environment
 - External support for the language
 - Libraries, documentation, community, IDEs, ...

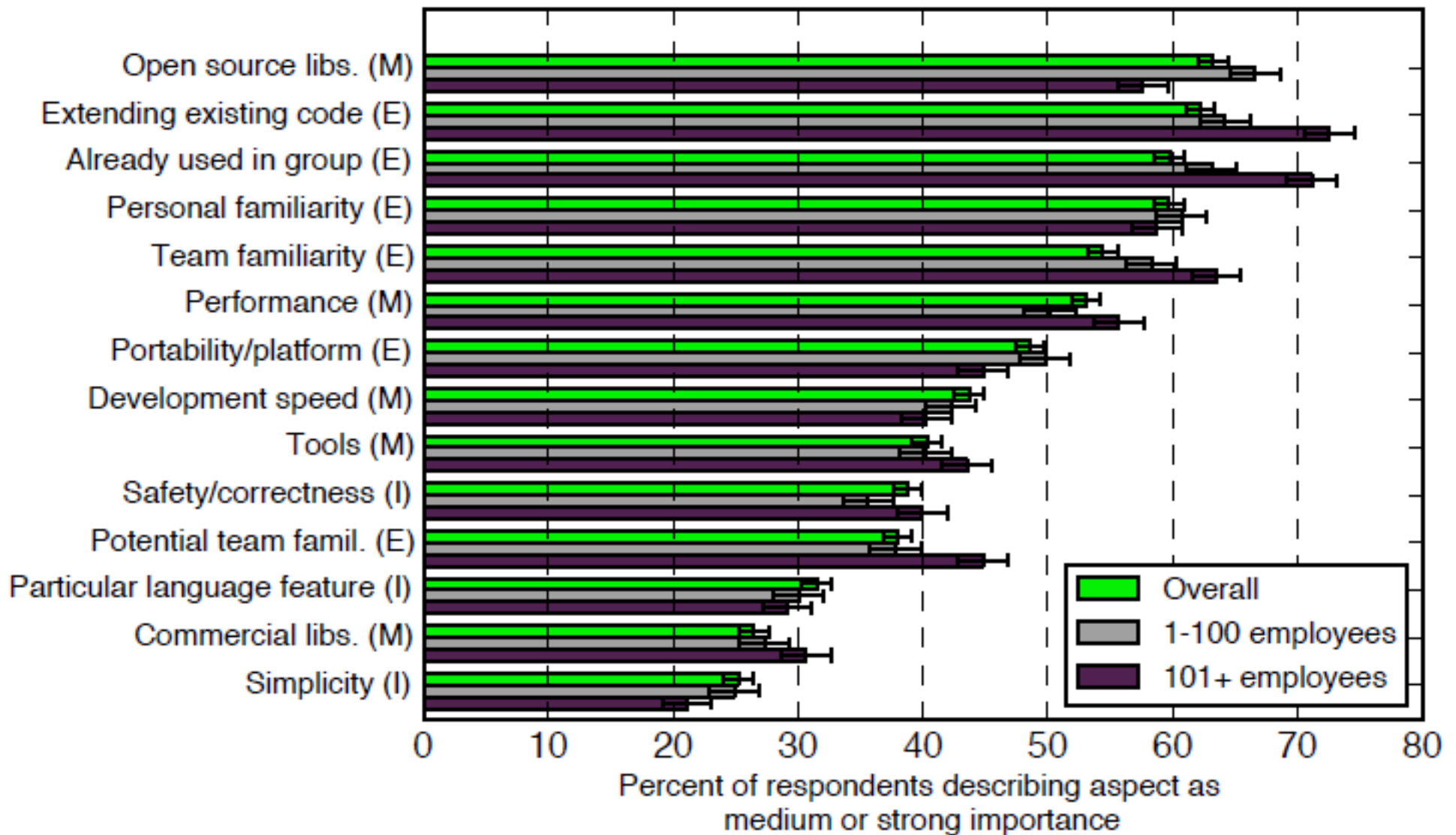
Attributes of a Good Language

- Clarity, simplicity, and unity
 - Provides both a framework for thinking about algorithms and a means of expressing those algorithms
- Orthogonality
 - Every combination of features is meaningful
 - Features work independently
- Naturalness for the application
 - Program structure reflects the logical structure of algorithm

Attributes of a Good Language

- Support for abstraction
 - Hide details where you don't need them
 - Program data reflects the problem you're solving
- Security & safety
 - Should be very difficult to write unsafe programs
- Ease of program verification
 - Does a program correctly perform its required function?

What Programmers Want In a PL



Summary

- ▶ Programming languages vary in their
 - Syntax
 - Style/paradigm
 - Semantics
 - 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