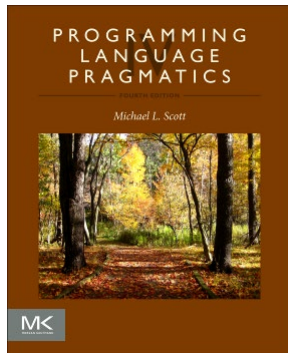


Introduction

17-363/17-663: Programming Language Pragmatics



Next edition: Scott & Aldrich!



Prof. Ben Titzer



Prof. Jonathan Aldrich



Introduction

- Language Design and Language Implementation go together
 - An implementor has to understand the language
 - A language designer has to understand implementation issues
 - A good programmer has to understand both!



Introduction

- Why are there so many programming languages?
 - evolution -- we've learned better ways of doing things over time
 - socio-economic factors: proprietary interests, commercial advantage
 - orientation toward special purposes
 - orientation toward special hardware
 - diverse ideas about what works well (and what people like)



Introduction

- What is your favorite language, and why do you like it?



Introduction

- What makes a language successful?
 - easy to learn (BASIC, Python, LOGO, Scheme)
 - expressive, powerful (C++, Common Lisp, Scala, Rust)
 - easy to implement (BASIC, Forth)
 - possible to compile to very good (fast/small) code (Fortran, C)
 - backing of a powerful sponsor (C#, Ada, Swift)
 - wide dissemination at minimal cost (Pascal, Java)
 - market lock-in (Javascript)



Introduction

- Why do we have programming languages?
What is a language for?
 - way of thinking / way of expressing algorithms
 - languages from the user's point of view
 - abstraction of virtual machine -- way of specifying what you want the hardware to do without getting down into the bits
 - languages from the implementor's point of view



Why study programming languages?

- Help you choose a language.
 - C++ vs. Rust for systems programming
 - Fortran vs. Julia for numerical computations
 - Python vs. JavaScript for web applications
 - Ada vs. C for embedded systems
 - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
 - Java vs. Scala for application servers



Why study programming languages?

- Make it easier to learn new languages
 - Familiarity with related languages
 - Understanding core concepts that reappear
- Use language/compiler ideas in your projects
 - Almost every complex system has a language somewhere!
- Learn how to reason rigorously
 - PL has some of the best intellectual tools!



Why study programming languages?

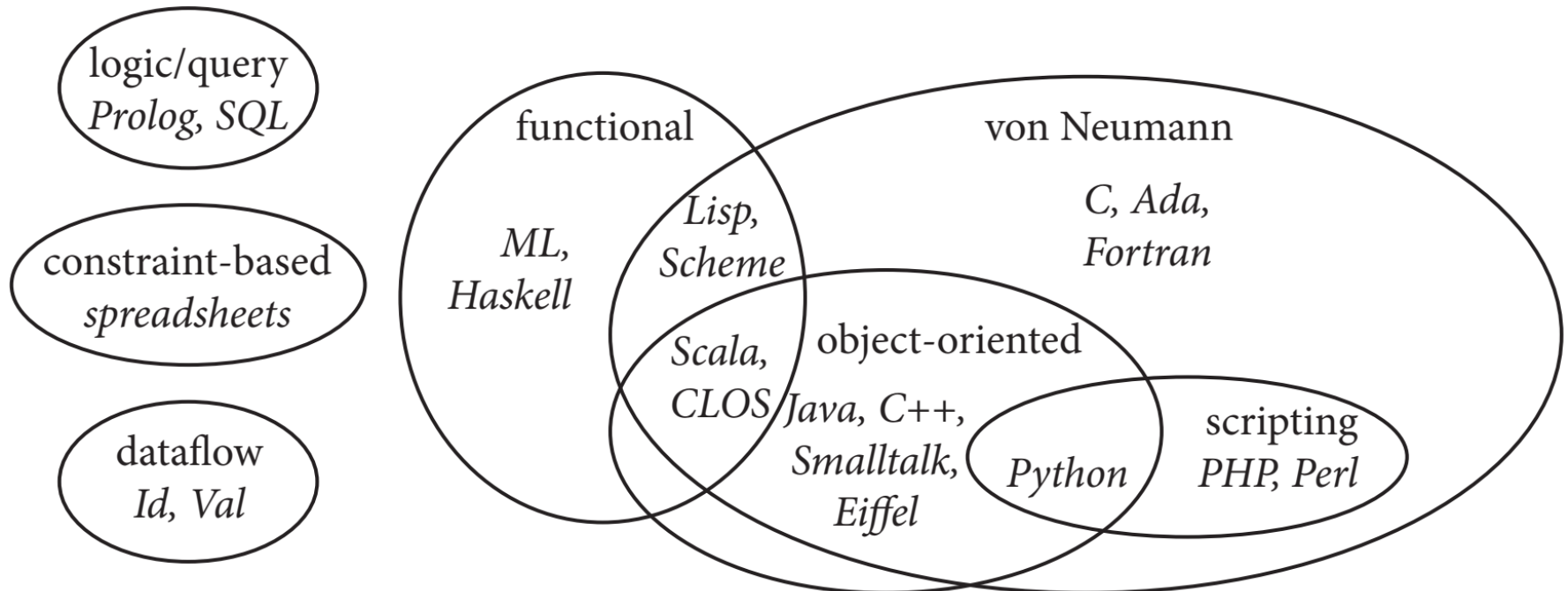
- Help you make better use of whatever language you use
 - Specialized features
 - unions, first-class functions, ...
 - Implementation costs
 - Garbage collection, tail recursion
 - Emulating missing features
 - Recursion (with loops and stacks)
 - First-class functions (with objects)...or vice versa!



Language Paradigms

Declarative Languages

Imperative Languages



How is this course different?

- Overall: emphasizes the interaction between language design and implementation
- Vs. 15-410
 - More focus on language design and theory; fulfills the Logic & Languages elective, not the Systems elective
- Vs. 15-312
 - “Pragmatic” focus – we study ideas and theory in the context of industrial languages and their design choices
 - Use of an educational proof assistant to make theory both more approachable and rigorous



Course Staff



Prof. Ben Titzer



Prof. Jonathan Aldrich



ELSEVIER

Course Administration

- Lectures 2x/week
 - Active learning exercises in every class
 - In person expectation
 - If you can't make it (COVID is not gone, but there may be other reasons too), email us—we'll get you a video & exercises
- Textbook: Programming Language Pragmatics
 - Strongly recommended: supplements lecture with more depth
 - Please give Jonathan feedback—he is coauthoring the next edition!
- Recitation
 - Lab-like, helpful for homework. Bring your laptop!

“How do I get an A?”

- 50% Homework –due Tues/Thurs 11:59pm
 - Small warm-up assignment due this Thursday
 - Build a compiler (4 coding assignments)
 - Implementation in Ocaml – great language & libraries for writing compilers
 - Reason about languages (4 theory assignments)
 - SASyLF educational theorem proving tool
- 20% - 2 midterm exams covering core concepts
- 25% Project
 - Extend the compiler in some interesting way, or explore theory
- 5% Participation (assessed via in-class exercises)
 - Can miss up to 2 sessions (lecture or recitation) w/o losing credit

Communication

- Website
 - Schedule, syllabus, slides
- Piazza for announcements, communication
 - Use Piazza as much as possible
 - Make questions public if possible, so others can benefit!
- Canvas
 - Assignments, grades
- Office hours TBA shortly (or just come by)



Read the Syllabus

A high level summary of some policies:

- Late work: 5 free late days
 - 10% penalty per day after these are used up
 - No credit more than 5 days late
 - Special circumstances: contact the instructor
- Collaboration policy
 - Your work must be your own
 - 100% penalty for cheating
 - Read full policy carefully
- No electronics in lecture
 - But bring them to recitation!



CMU can be pretty intense

- A 12-credit course is expected to take ~12 hours a week.
- We aim to provide a rigorous but tractable course.
 - More frequent assignments rather than big monoliths
 - Two midterm exams to cover core material as you learn it
- Please keep us apprised of how much time the class is actually taking and whether it is interfacing badly with other courses.
 - We have no way of knowing if you have three midterms in one week.
 - Sometimes, we misjudge assignment difficulty.
- If it's 2 am and you're panicking...put the homework down, send us an email, and go to bed.



Executing programs

- Consider the following program
 - In a simple imperative language, Hoare's WHILE

```
y := x;  
z := 1;  
while y > 1 do  
    z := z * y;  
    y := y - 1
```

- How do we run this sequence of characters?



Programs as trees

- What if we organize it as a tree in memory

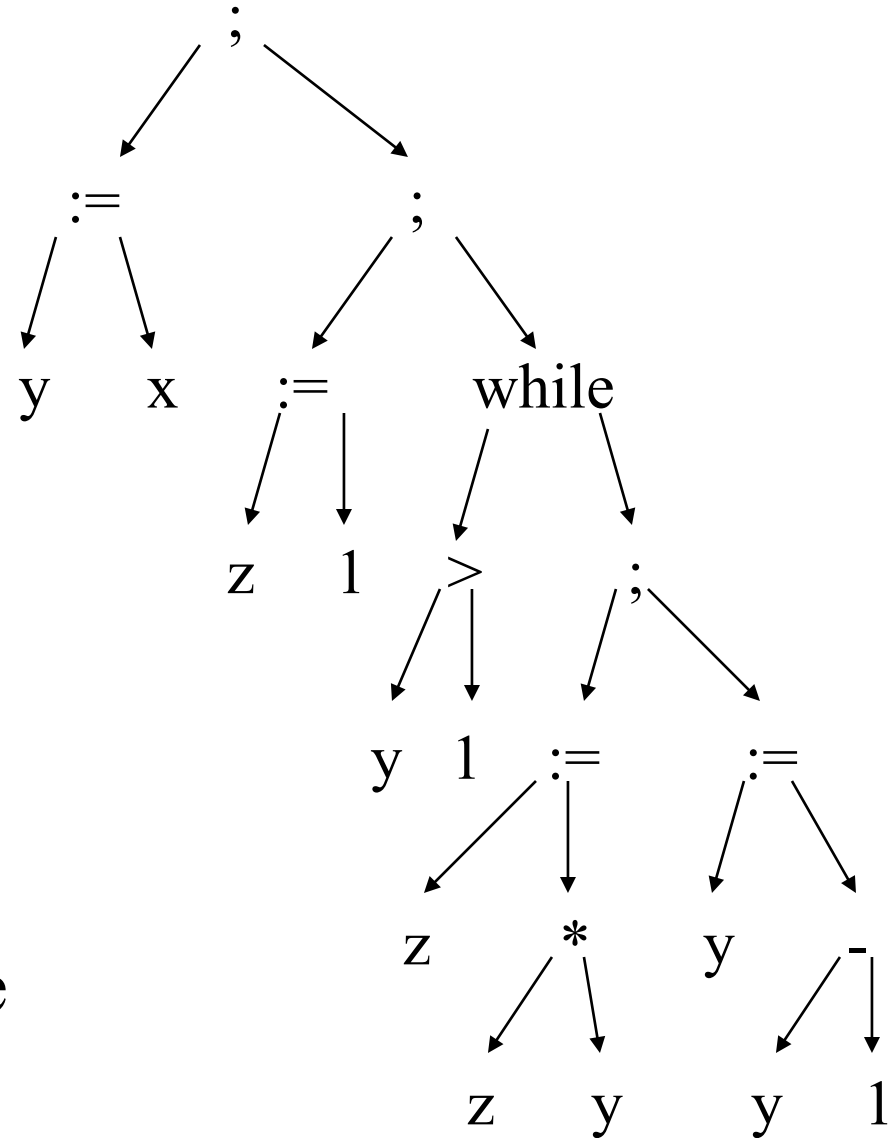
```
y := x;
```

```
z := 1;
```

```
while y > 1 do
```

```
  z := z * y;
```

```
  y := y - 1
```



- Now we can walk the tree and execute it

Interpreters



- Interpreter runs at execution time
 - Operates over the program as a data structure
- A simple and flexible approach—but slow
 - We examine the program to determine what to do, over and over again

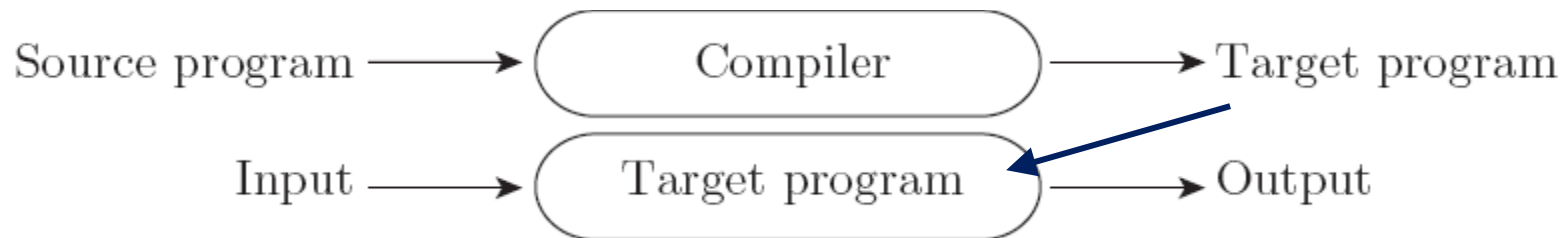
A Pattern for Simple Interpreters

```
function interpret_expr(a : AST) : int
case a of
  int_lit(n) : return n
  bin_op(a1, op, a2) :
    let v1 : int = interpret_expr(a1)
    let v2 : int = interpret_expr(a2)
    case op of
      “+” : return v1 + v2
      “-” : return v1 - v2
      “×” : return v1 * v2
      “÷” : return v1 / v2
```



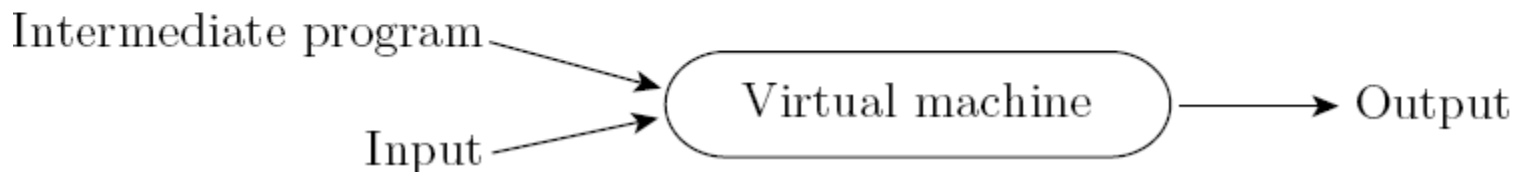
Compilers

- A compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



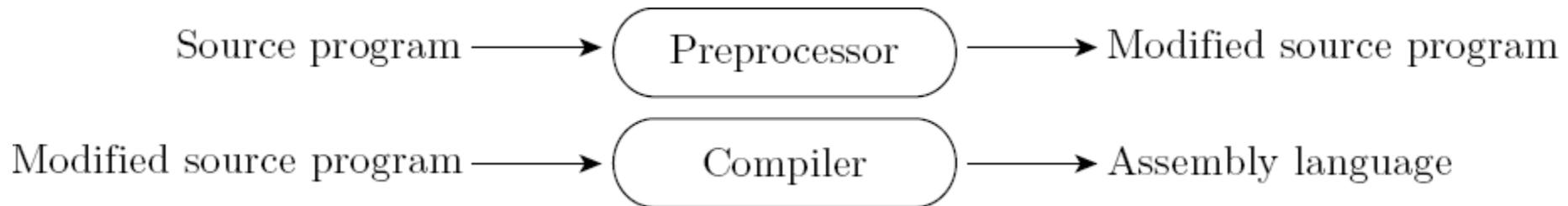
Virtual Machine Targets

- A common case is compilation to a virtual machine target
 - E.g. Java source to JVM bytecode
 - The virtual machine can itself be an interpreter or a compiler
- Why is this useful?



Compilation: Preprocessing

- The C Preprocessor (conditional compilation)
 - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



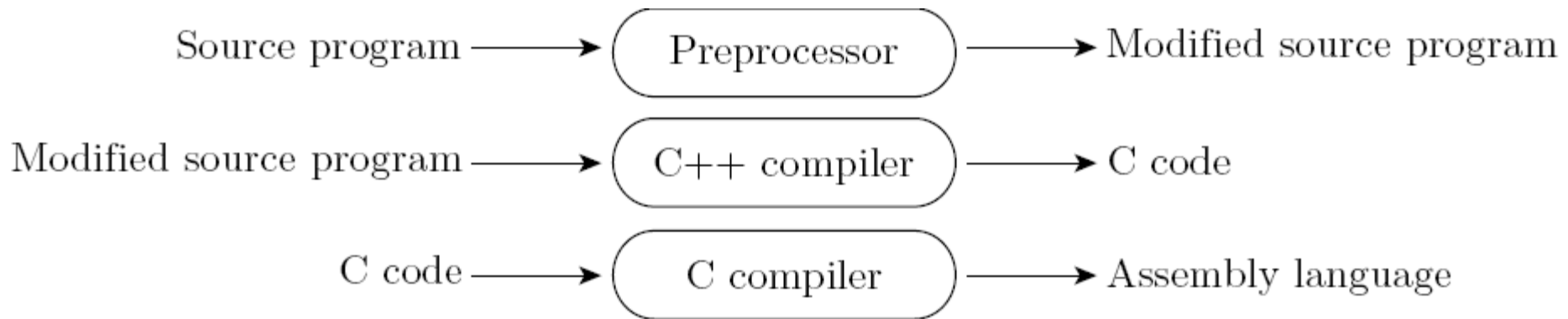
Compilation vs. Preprocessing

- Note that compilation does NOT have to produce machine language for some sort of hardware
- Compilation is *translation* from one language into another, with full analysis of the meaning of the input
- Compilation entails semantic *understanding* of what is being processed; pre-processing does not
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not



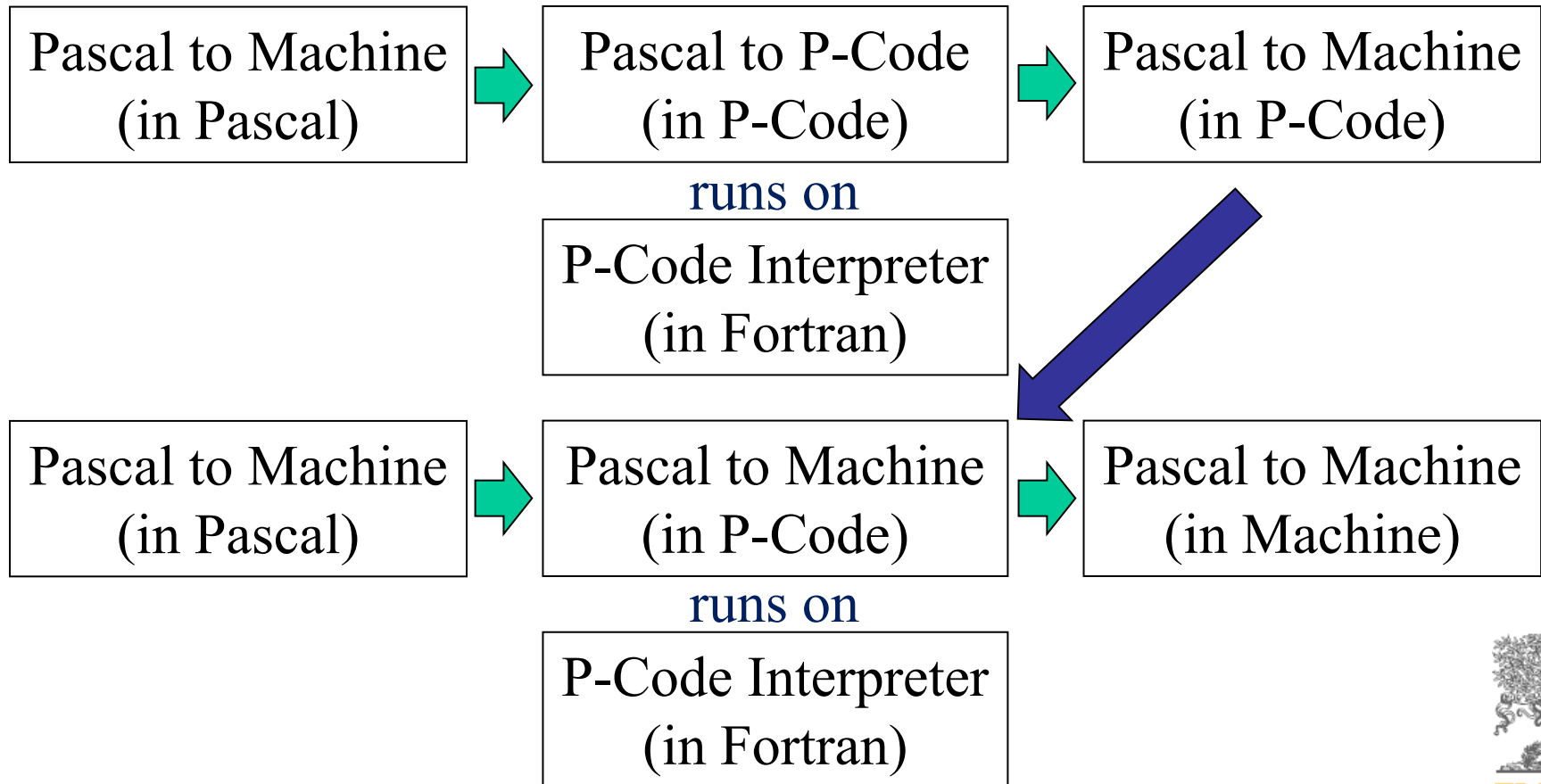
Compilation Strategies

- Source-to-Source Translation (C++)
 - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language:



Compilation Strategies

- Bootstrapping



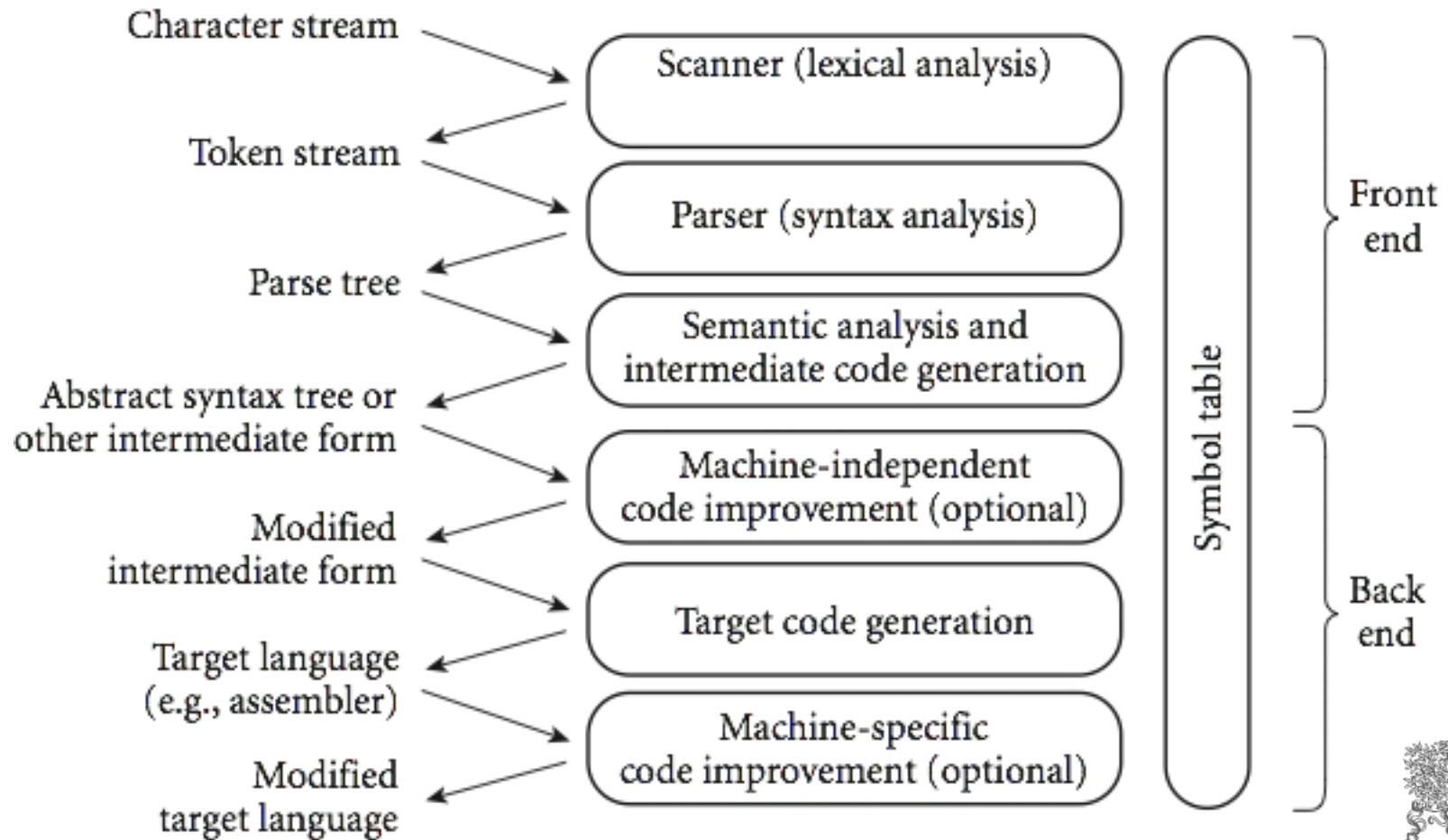
Compilation vs. Interpretation

- Compilation produces the fastest programs
- So why interpret?
 - Allows delaying decisions to run time
 - Names to objects, types of objects, even what code is run
 - Used in dynamic/scripting languages (Scheme, Python, Shell scripts, ...)
 - Compilation can account for these, but becomes complex and somewhat slower anyway
 - Small code size
 - Good diagnostics—interpreter state is available
 - Fast startup (don't have to wait for the compiler)
 - Easy to write and port



An Overview of Compilation

- Phases of Compilation



Scanning / Lexical Analysis

- Input program:

```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1  
od
```

- Output of scanner is a stream of *tokens*:

```
y := x ; z := 1 ; while y > 1 do z := z * y ; y  
:= y - 1 od
```



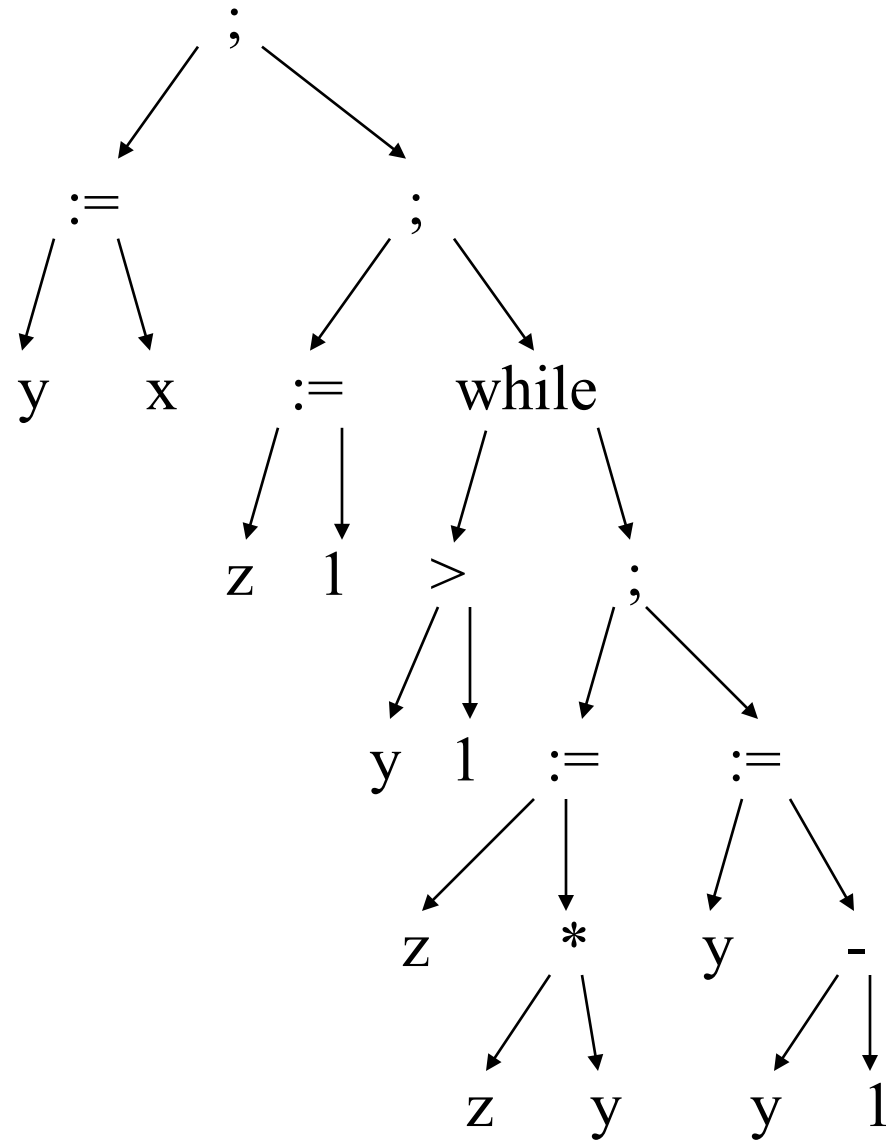
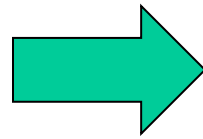
Scanning / Lexical Analysis

- divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
 - scanning is recognition of a *regular language*, e.g., via a DFA
- removes comments
- saves text of identifiers, strings, numbers
- tags tokens with line numbers, for error messages
- main benefits: efficiency, simplifies later stages
 - you can design a parser to take characters instead of tokens as input, but it isn't pretty



Parsing

```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1
```



Semantic analysis

- *Semantic analysis* is the discovery of *meaning* in the program
 - The compiler actually does what is called **STATIC** semantic analysis. That's the meaning that can be figured out at compile time
 - E.g. typechecking, which catches errors and helps generate code (e.g. floating point vs. integer add)
 - Some things (e.g., array subscript out of bounds) usually can't be figured out until run time. Things like that are part of the program's **DYNAMIC** semantics



Concrete vs. Abstract Syntax Trees

- *Concrete* syntax trees capture exactly the syntax in the source program
- *Abstract* syntax trees (ASTs) simplify things
 - E.g. getting rid of parentheses, which are only necessary to show the intended tree structure



An Overview of Compilation

- *Intermediate form* (IF) done after semantic analysis (*if* the program passes all checks)
 - IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
 - They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
 - Many compilers actually move the code through more than one IF



An Overview of Compilation

- *Optimization* takes an intermediate-code program and produces another one that does the same thing faster, or in less space
 - The term is a misnomer; we just *improve* code (but see superoptimization)
 - Can be very complex and take a long time—but also produce significant speedup
 - The optimization phase is optional



An Overview of Compilation

- *Code generation* produces assembly language or (sometime) relocatable machine language
 - Allocating registers to store data
 - Machine-specific optimizations



Programming Language Pragmatics

- PL is an exciting field to study
 - Interesting theory
 - Important impact on practice
 - Lots of applications
 - Will help you become a better programmer
- For next time:
 - Get the textbook and read through chapter 2.2
 - Homework “zero” is out today, due Thursday
 - The first real homework will be out Thursday

