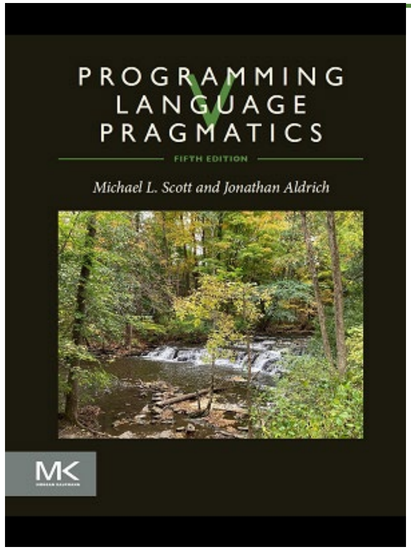
Introduction

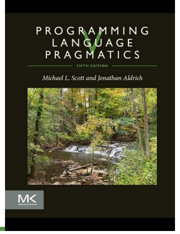

PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

Programming Language Pragmatics

Prof. Jonathan Aldrich

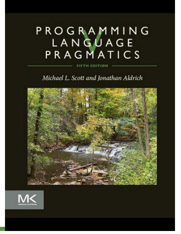
Copyright © 2025 by Elsevier. Slides written by Jonathan Aldrich and Michael L. Scott. Licensed under CC-BY 4.0.

Language design and implementation go together



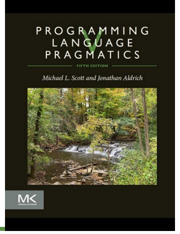
- An implementor has to understand the language
  - To ensure the implementation is correct
- A language designer has to understand implementation issues
  - To ensure the language can be implemented efficiently
- A good programmer has to understand both!
  - To write correct, understandable, and efficient programs

Why are there so many programming languages?


PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

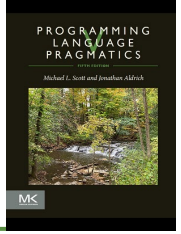
- Evolution: we’ve learned better ways to do things
  - Structured programming over gotos
- Socio-economic factors: proprietary interests, network effects
  - Learn Swift to program iPhone apps, Java for Android apps
- Special purposes
  - JavaScript is good for web programs, Rust for systems programming
- Hardware focus
  - CUDA for GPUs
- Personal preference: diverse ideas about what works well

What makes a language successful?



- Easy to learn (BASIC, Python, LOGO, Scheme)
- Expressive power (C++, Common Lisp, Scala, Rust)
- Easy to implement, freely available (BASIC, Forth, Pascal, Java)
- Safety (Java, Rust)
- Standardization (C, Java, C#)
- Open source (C)
- Efficient (fast/small) code (Fortran, C, Rust)
- Backing of a powerful sponsor (C#, Ada, Swift)
- Market lock-in (Cobol, JavaScript)

Two viewpoints: the programmer & the computer

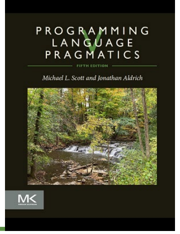

PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

- “Computer Programming is the art of explaining to another human being what you want the computer to do.” - Donald Knuth

- Programmer’s view
  - Language as a way of thinking and expressing algorithms
- Implementer’s view
  - An abstraction of a (virtual) machine

- Both conceptual clarity and efficient implementation are fundamental concerns

Programming language people: Donald Knuth



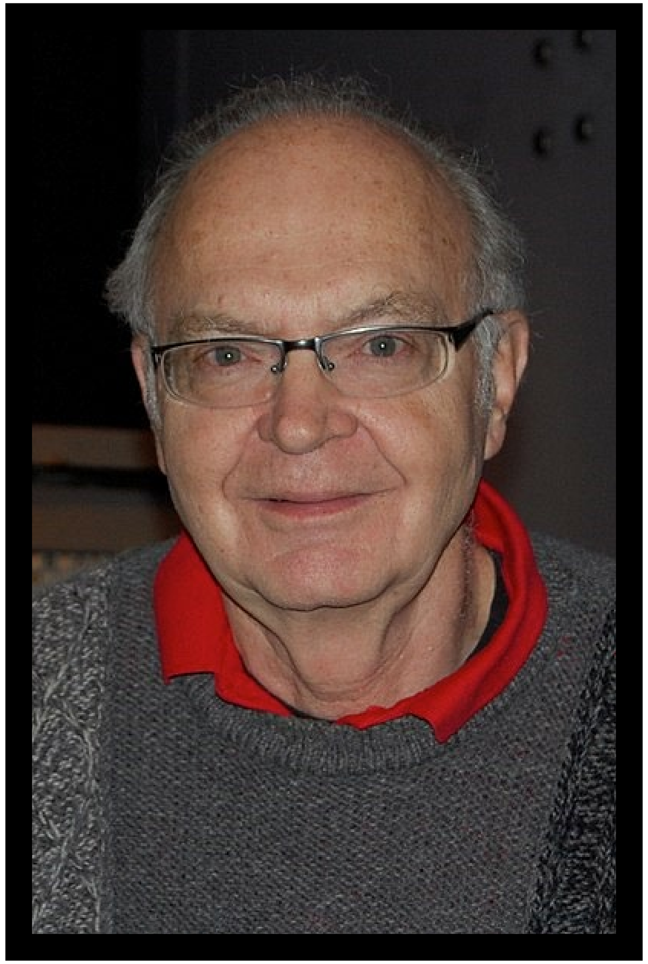


Image by Alex Handy
CC BY-SA 2.0

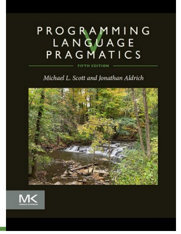
Donald E. Knuth (1938–)

Professor Emeritus,
Stanford University

Known for:

- Design and analysis of algorithms
- The TEX typesetting system
- Literate programming methodology
- The Art of Computer Programming
- ACM Turing Award (1974)

Language Paradigms



PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

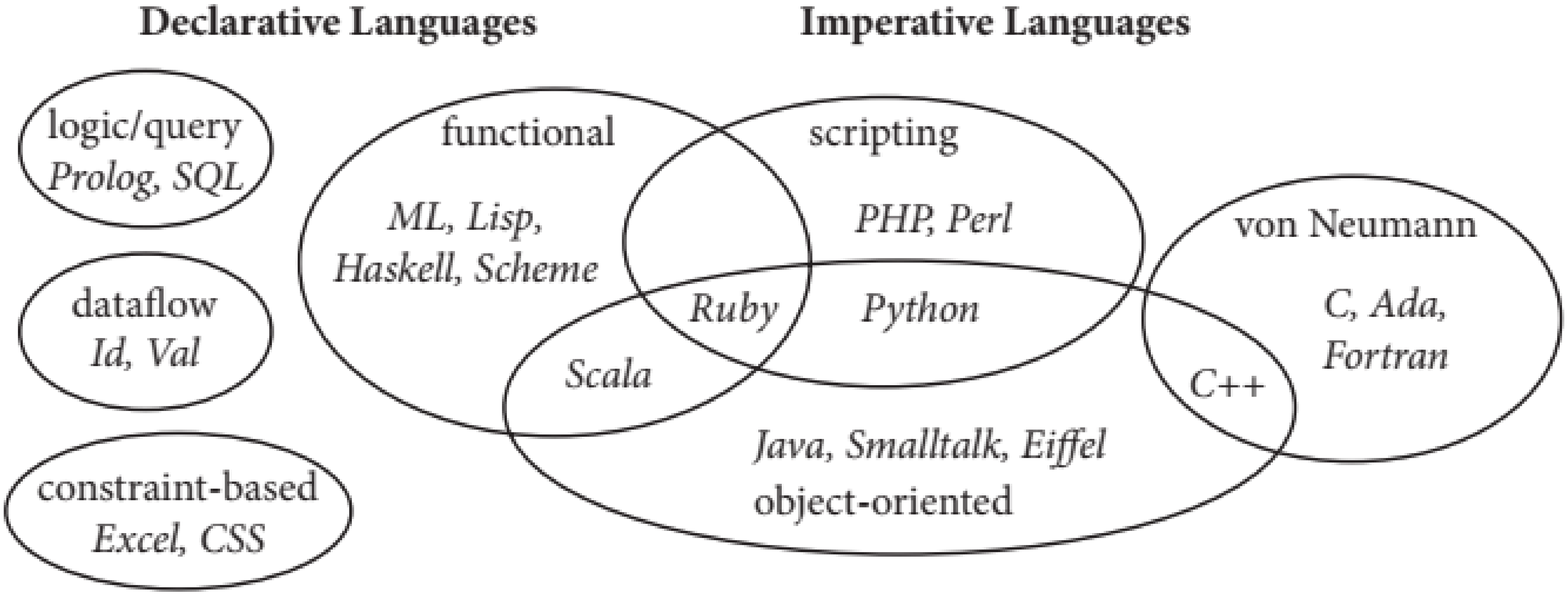
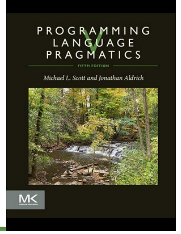


Figure 1.1 **Classification of programming languages.** Note that the categories are fuzzy, and open to debate. Many languages fall into more than one category. Many authors do not consider functional programming to be declarative.



Declarative languages tend to be higher level

- Closer to programmer, further from machine
- Focus on **what** program should do
- Logic/query languages (Prolog, SQL)
  - Find values that satisfy constraints
- Dataflow languages (Id, Val)
  - Model computation as parallel flow of tokens
- Constraint-based (Excel, CSS)
  - Express constraints to be solved/maintained
- Functional languages (Haskell, Scheme)
  - Side-effect-free computation of outputs from inputs using functions, supports unbounded computation using recursion

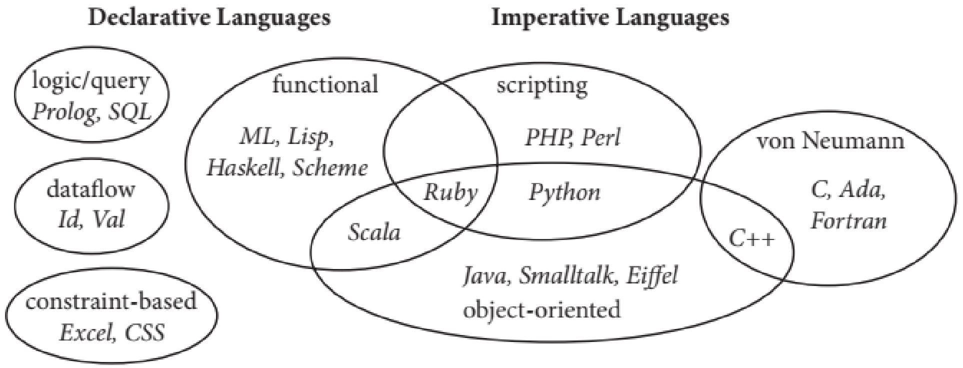
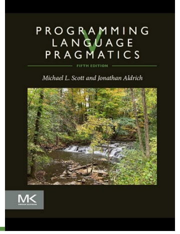


Figure 1.1 **Classification of programming languages.** Note that the categories are fuzzy, and open to debate. Many languages fall into more than one category. Many authors do not consider functional programming to be declarative.



Imperative languages are more algorithmic

- Less abstract, closer to the machine
- Focus on **how** program should operate
- Von Neumann languages (C, Fortran)
  - Computation as modification of variables, unbounded work done through loops
- Object-oriented languages (C++, Java)
  - Computation is structured and distributed among objects, each of which has data and methods
- Scripting languages (Python, JavaScript)
  - Emphasize flexibility, ease of programming, gluing components together

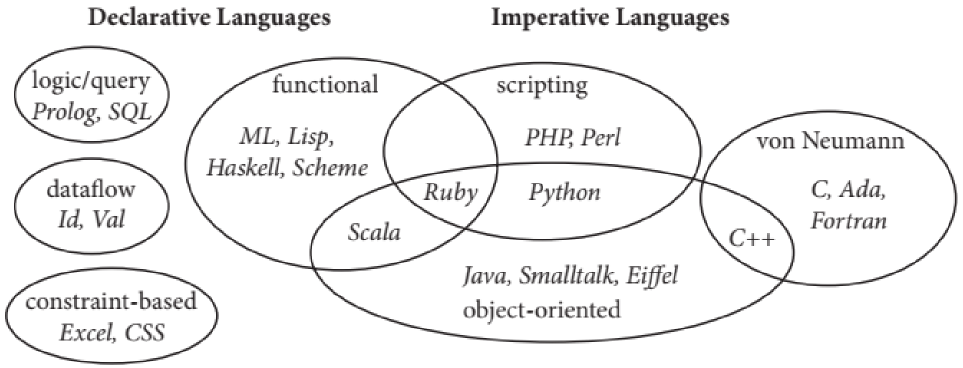
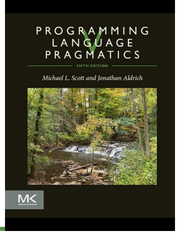


Figure 1.1 **Classification of programming languages.** Note that the categories are fuzzy, and open to debate. Many languages fall into more than one category. Many authors do not consider functional programming to be declarative.


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

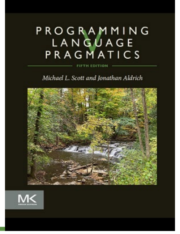
Programming language people: John von Neumann

Image from Los Alamos National Laboratory. Used by permission (see slide notes)

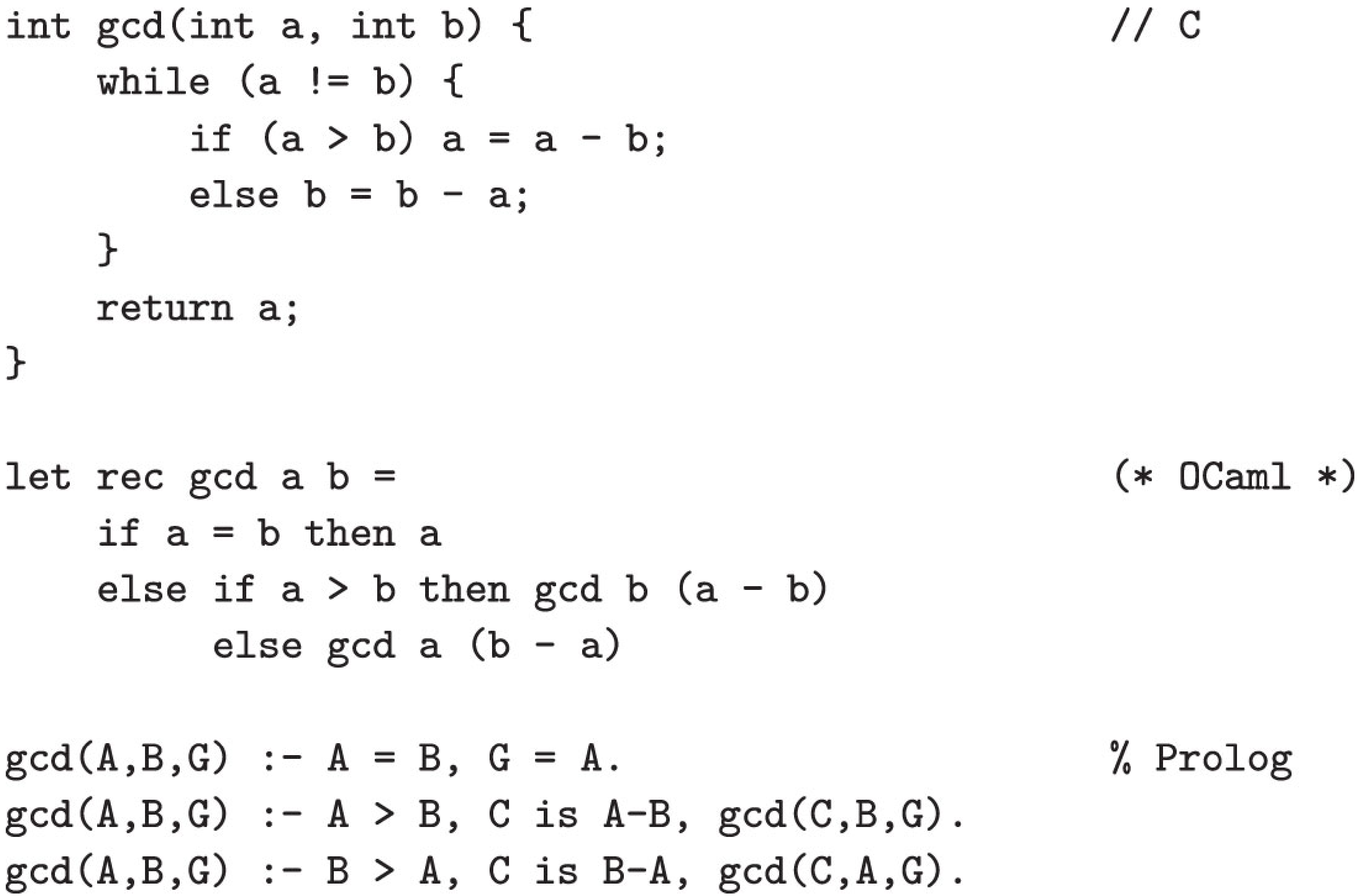
John von Neumann

(1903–1957)

- Mathematician and computer pioneer
- helped to develop the concept of *stored program computing*
  - underlies most computer hardware
  - both programs and data are represented as bits in memory
  - processor repeatedly fetches, interprets, and updates that representation

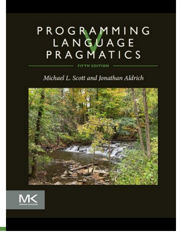


One program, three language families



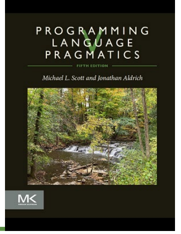
```
int gcd(int a, int b) {                                  // C
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}

let rec gcd a b =                                        (* OCaml *)
    if a = b then a
    else if a > b then gcd b (a - b)
         else gcd a (b - a)

gcd(A,B,G) :- A = B, G = A.                              % Prolog
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

Discussion: compare languages



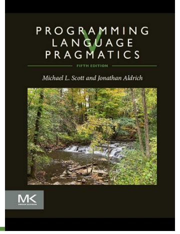
- Think about two different programming languages that you know. For each, name one advantage of using that language.

Why study programming languages?



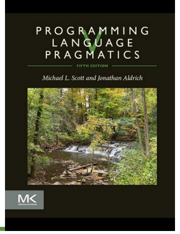
- Help you choose a language
  - What kind of project is Rust good for? JavaScript? Python?
- Learn new languages more easily
  - Leverage concepts that cross-cut languages: types, control structures, …
- Make better use of languages and language technology
  - Understanding obscure features when you need to
  - Choose alternative ways to express things, e.g. based on cost
  - Use tools such as debuggers, assemblers, and linters effectively
  - Know how to work around features missing from your language
  - Languages are everywhere: configuration files, extension languages, scripting, …
- Learn to reason rigorously
  - PL has some of the best intellectual tools!

How is this course different?



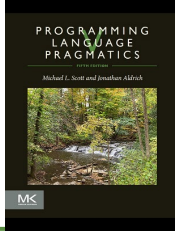
- Overall: emphasizes the interaction between language design and implementation
- Vs. 15-410
  - More focus on language design and theory; fulfills the Logic & Languages elective, not the Systems elective
- Vs. 15-312
  - “Pragmatic” focus – we study ideas and theory in the context of industrial languages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

Course Administration

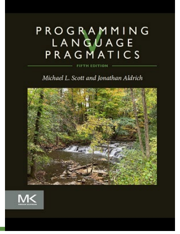

PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

- Lectures 2x/week
  - Active learning exercises in every class
  - In person expectation
    - If you can’t make it, email me—we’ll get you a video & exercises
- Textbook: Programming Language Pragmatics, 5th edition
- Recitation
  - Lab-like, helpful for homework. Bring your laptop!

“How do I get an A?”


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

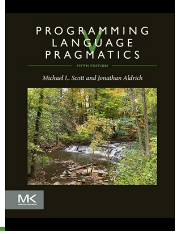
- 50% Homework –due Friday 11:59pm
  - Build a compiler (5 coding assignments, plus a warmup this week)
    - Implementation in Rust – good language for compilers & interesting to study
  - Reason about languages (4 theory assignments)
    - SASyLF educational theorem proving tool
- 20% - 2 midterm exams 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


PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

Communication

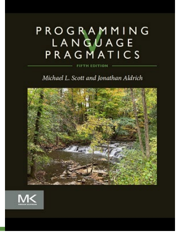
- Website
  - Schedule, syllabus, slides
- Piazza for announcements, communication
  - Use Piazza as much as possible
  - Make questions public if possible, so others can benefit!
- Canvas
  - Assignments, grades
- Office hours TBA shortly (or just come by)

Read the syllabus!


PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

A high level summary of some policies:

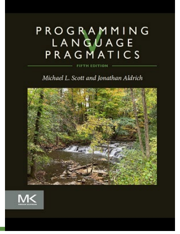
- Late work: 5 free late days
  - 10% penalty per day after these are used up
  - No credit more then 5 days late
  - Special circumstances: contact the instructor
- Collaboration policy
  - Your work must be your own
  - 100% penalty for cheating
  - Read full policy carefully
- No electronics in lecture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


PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

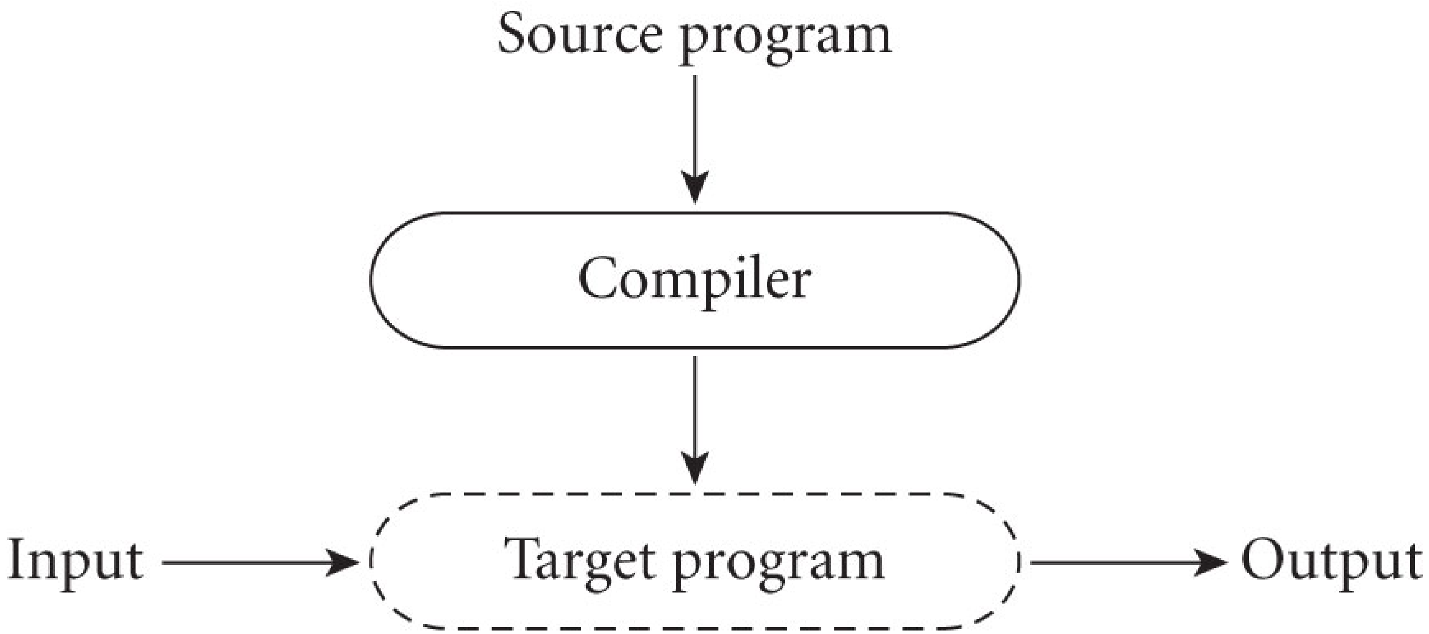
CMU can be pretty intense

- 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

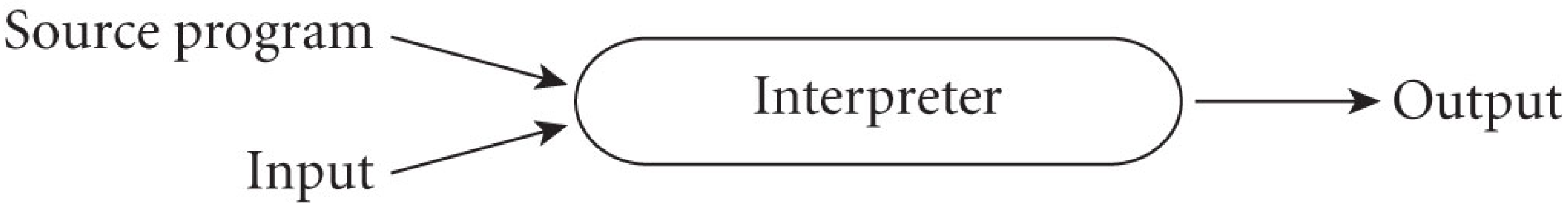
Two approaches to language implementation


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

Compilation:



Interpretation:



Programming lang. people: Grace Murray Hopper

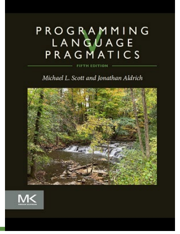
Image by James S. Davis
public domain

Grace Murray Hopper

(1906–1992)

- Mathematics professor, computing pioneer, rear admiral in the US Navy
- While working on the early Mark II computer, Hopper's team discovered a moth in a relay, leading her to write "First actual case of bug being found.''
- Developed the FLOW-MATIC language based on English words to make computing more accessible
- Coined the term "compiler" and later played a key role in the design of Cobol.

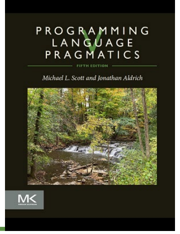
Interpretation vs. compilation

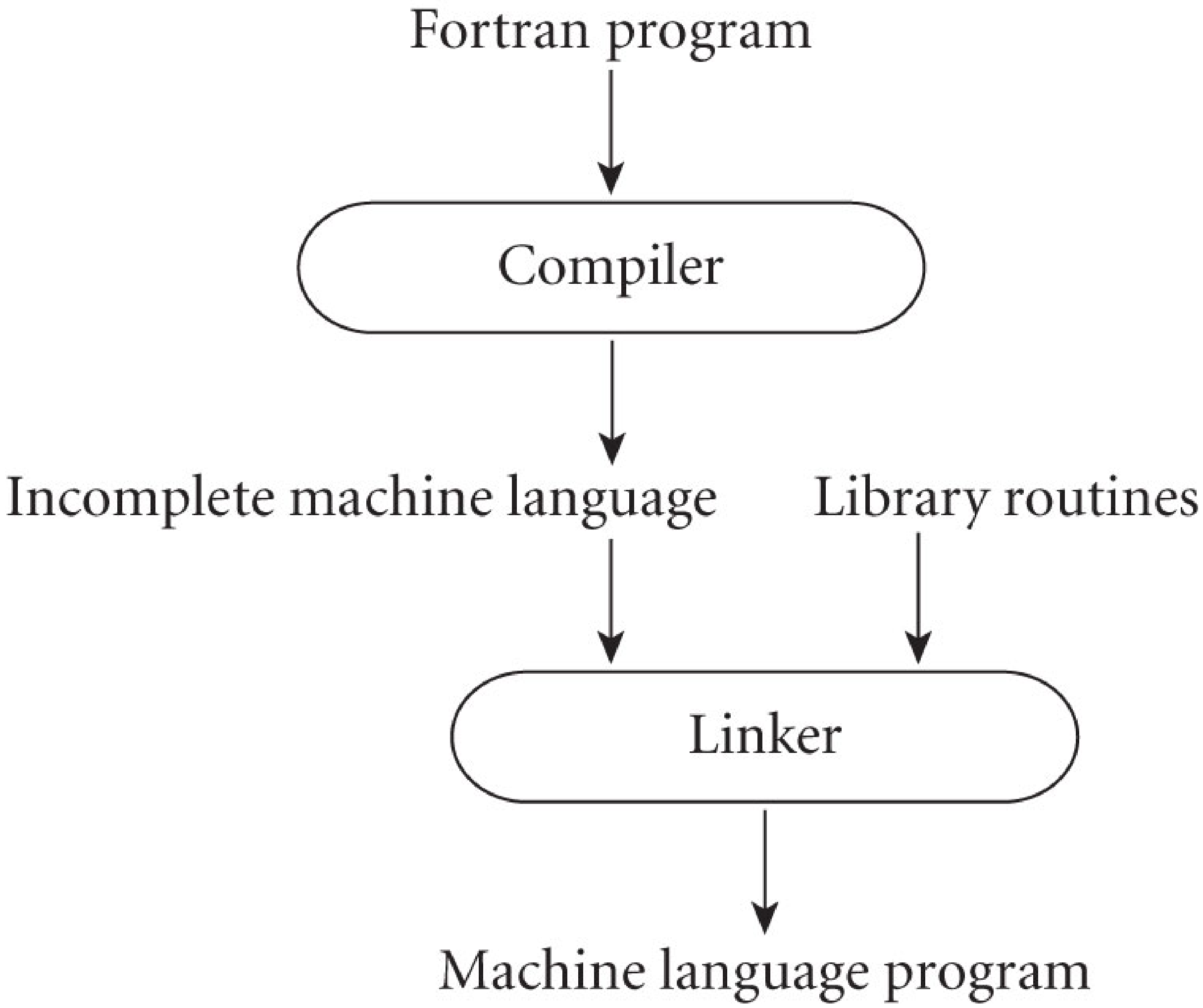


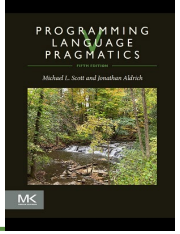
- Interpretation supports more flexibility, better diagnostics
  - Often excellent source-level debuggers

- Compilation generally leads to better performance
  - Decisions made at compile time need not be repeated at run time, saving effort

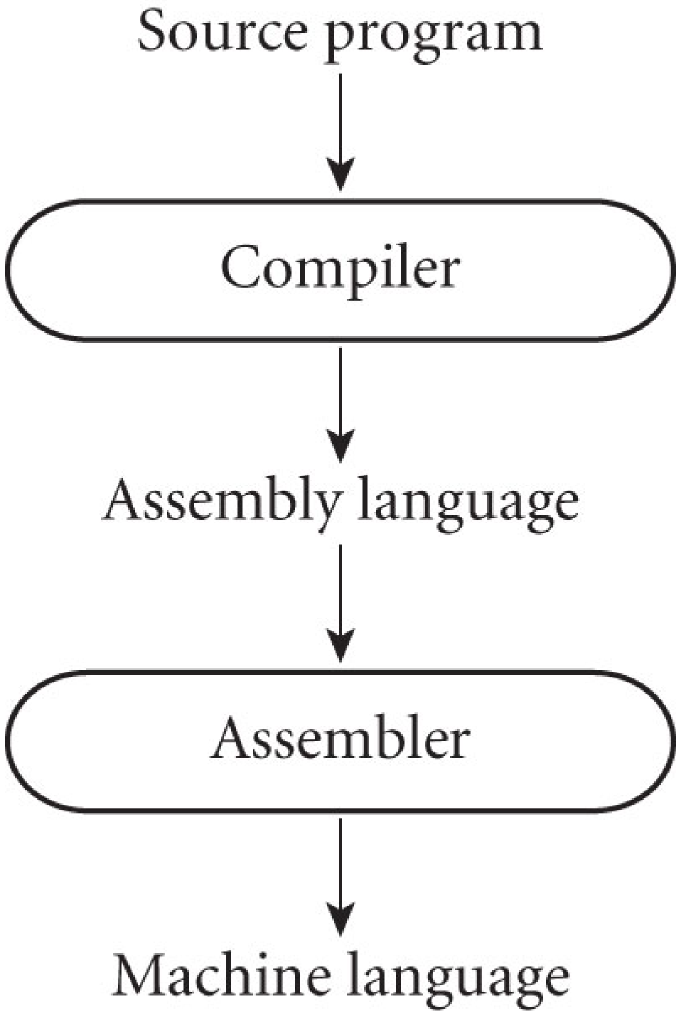
Linking to libraries – for example, in Fortran







Assembly language is often an intermediate step



Source program

Compiler

Assembly language

Assembler

Machine language

Programming language people: Kathleen Booth

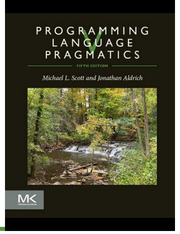
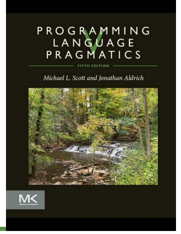


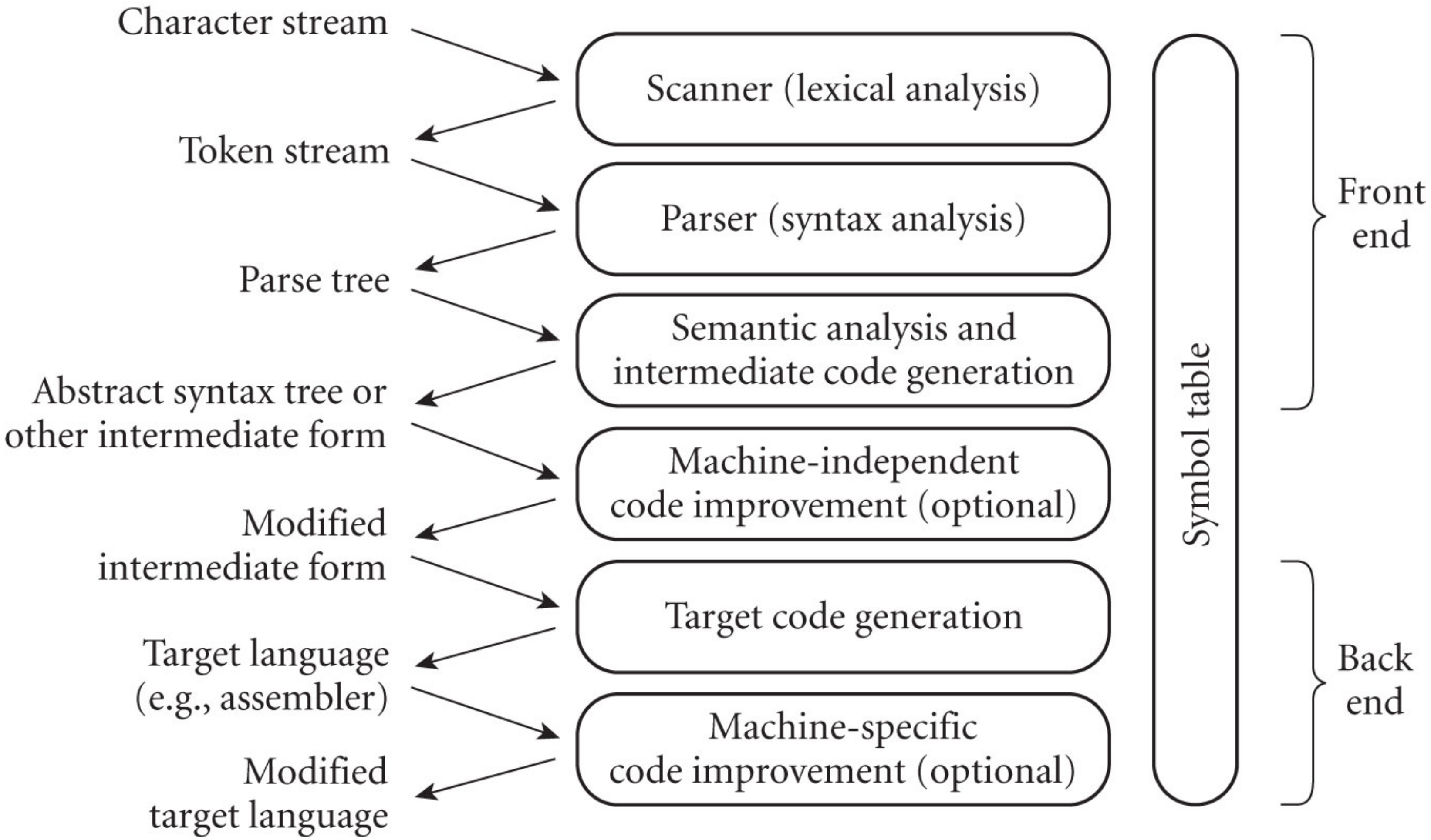
Image by Gibson
Public domain

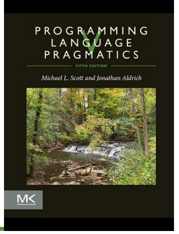
Kathleen Hylda Valerie
Booth (1922–2022)

- Co-founded the Department of Numerical Automation (now School of CS & IS) at Birkbeck College, University of London, in 1957.
- Co-developed the ARC, SEC, and APE(X)C computing systems
- Invented the first assembly language for the ARC, and wrote a book on how to program the APE(X)C.
- Explored neural networks as a way to understand how animals recognize patterns.

An overview of compilation


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

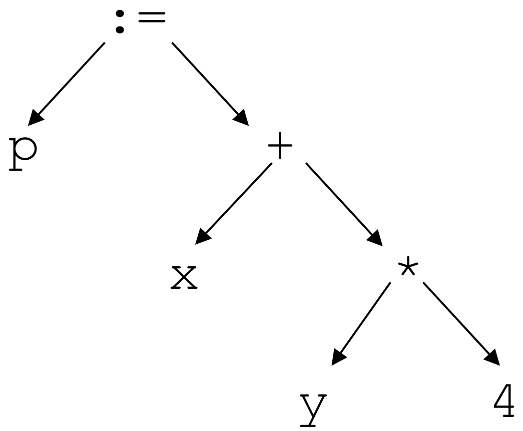


Parsing

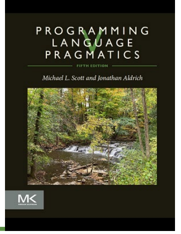
- Parsing converts a sequence of tokens into a parse tree that captures the program structure

`p` `:=` `x` `+` `y` `*` `4`

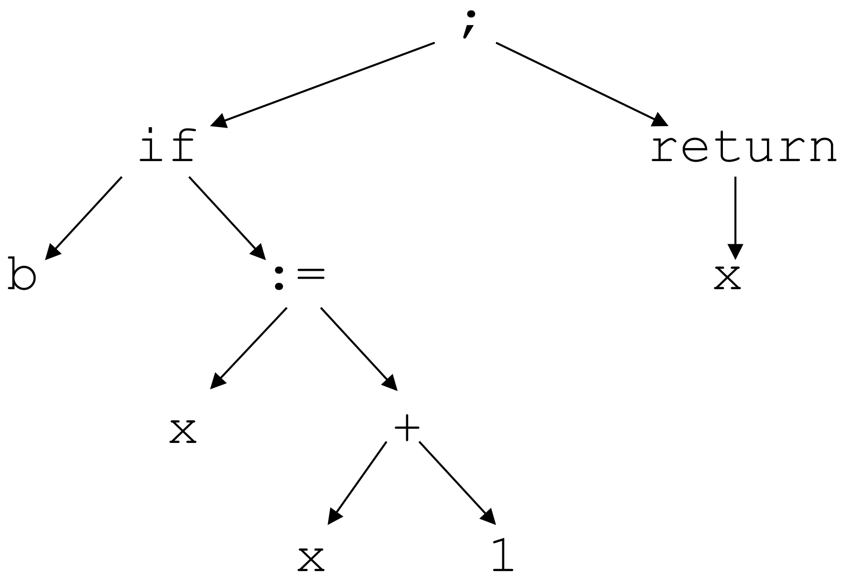
➔



Intermediate code generation

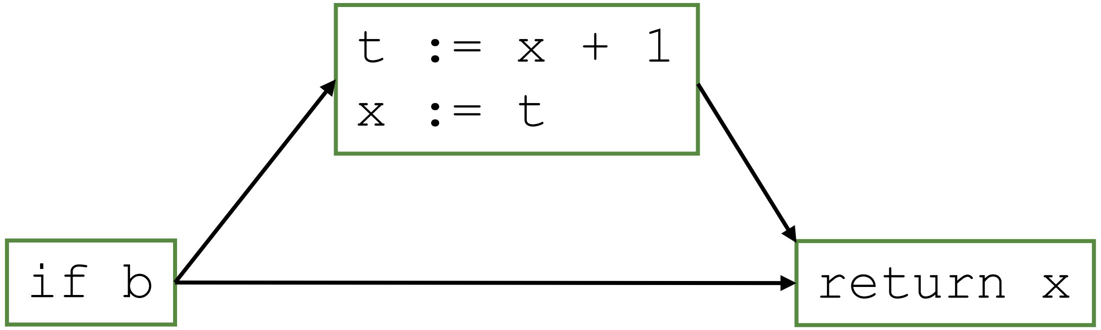


- Intermediate code generation
  - Generates intermediate code in form(s) that are more abstract or closer to the machine than parse trees
  - Examples: Abstract syntax tree (AST) and/or control-flow graph (CFG)



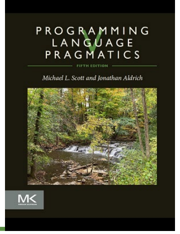
parse tree / abstract syntax tree

→



control flow graph

Target code generation



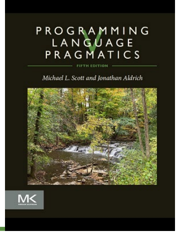
- May generate byte code, assembly code, or machine code
- Traverse symbol table to assign locations to variables
- Traverse intermediate code to generate output instructions
- May generate a symbol table for use by a debugger

```
y := x << 2
z := y + 1
```

→

```
mov eax, [x]
shl eax, 2
mov [y], eax
add eax, 1
mov [z], eax
```

Programming language design and implementation

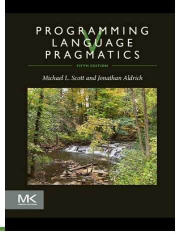

PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

- Language design and implementation are closely linked
  - Different languages and paradigms represent different tradeoffs: closeness to the problem, level of abstraction, performance, and market forces

- Languages can be implemented with interpreters or compilers
  - Tradeoffs between flexibility, diagnostics, and performance
  - Compilers fully analyze the source program and transform it substantially

- Studying programming languages can help you become a better programmer!

For next time



- Read PLP chapter 1 (as soon as you can get the book)
- Homework “zero” is out today, due Friday. Useful:
  - Rust book chapters 1-6, esp. “Programming a Guessing Game” https://rust-book.cs.brown.edu/
  - x86 quick references
    - Stanford https://web.stanford.edu/class/archive/cs/cs107/cs107.1196/guide/x86-64.html
    - Brown https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf