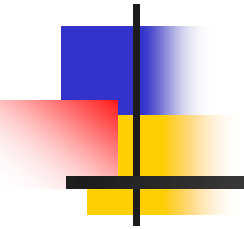
Programming Languages and Compilers (CS 421)

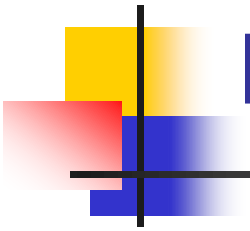


Grigore Rosu

2110 SC, UIUC

http://www.cs.uiuc.edu/class/cs421/

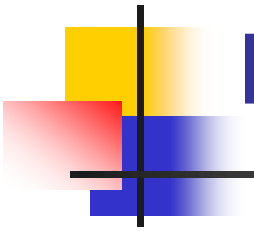
Slides by Elsa Gunter, based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



Programming Language Goals

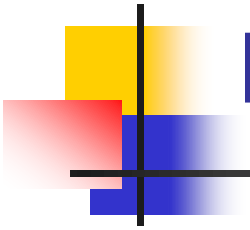
- Original Model:
  - Computers expensive, people cheap; hand code to keep computer busy
- Today:
  - People expensive, computers cheap; write programs efficiently and correctly

Programming Language Goals



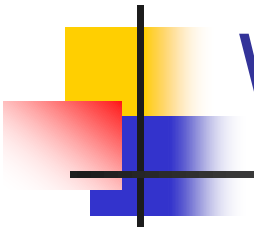
- Mythical Man-Month Author Fred Brookes

  "The most important two tools for system programming … are (1) high-level programming languages and (2) interactive languages"



Languages as Abstractions

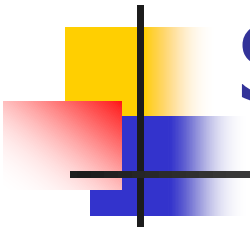
- Abstraction from the Machine
- Abstraction from the Operational Model
- Abstraction of Errors
- Abstraction of Data
- Abstraction of Components
- Abstraction for Reuse



Why Study Programming Languages?

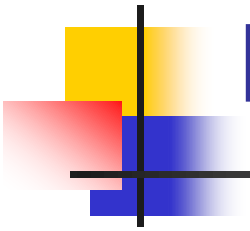
Helps you to:

- understand efficiency costs of given constructs
- reduce bugs by understanding semantics of constructs
- think about programming in new ways
- choose best language for task
- design better program interfaces (and languages)
- learn new languages



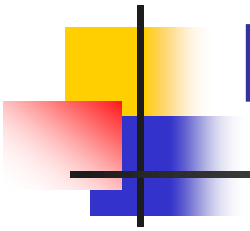
Study of Programming Languages

- Design and Organization
  - Syntax: How a program is written
  - Semantics: What a program means
  - Implementation: How a program runs
- Major Language Features
  - Imperative / Applicative / Rule-based
  - Sequential / Concurrent



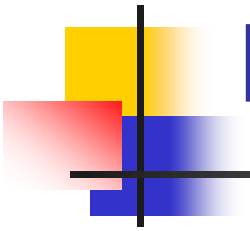
Historical Environment

- Mainframe Era
  - Batch environments (through early 60's and 70's)
    - Programs submitted to operator as a pile of punch cards; programs were typically run over night and output put in programmer's bin



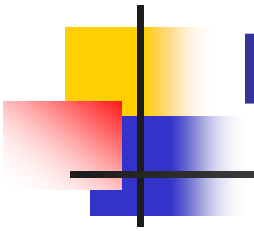
Historical Environment

- Mainframe Era
  - Interactive environments
    - Multiple teletypes and CRT's hooked up to single mainframe
    - Time-sharing OS (Multics) gave users time slices
    - Lead to compilers with read-eval-print loops



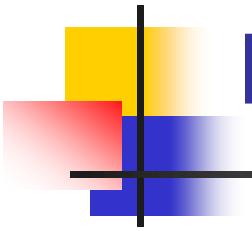
Historical Environment

- Personal Computing Era
  - Small, cheap, powerful
  - Single user, single-threaded OS (at first any way)
  - Windows interfaces replaced line input
  - Wide availability lead to inter-computer communications and distributed systems



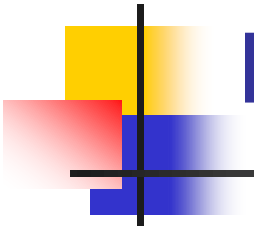
Historical Environment

- Networking Era
  - Local area networks for printing, file sharing, application sharing
  - Global network
    - First called ARPANET, now called Internet
    - Composed of a collection of protocols: FTP, Email (SMTP), HTTP (HMTL), URL



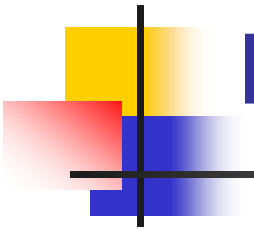
Features of a Good Language

- Simplicity – few clear constructs, each with unique meaning
- Orthogonality - every combination of features is meaningful, with meaning given by each feature
- Flexible control constructs



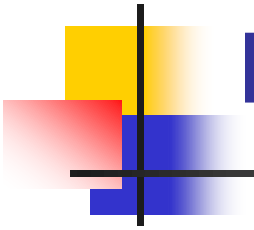
Features of a Good Language

- Rich data structures – allows programmer to naturally model problem
- Clear syntax design – constructs should suggest functionality
- Support for abstraction - program data reflects problem being solved; allows programmers to safely work locally



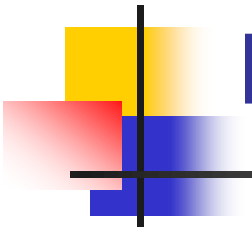
Features of a Good Language

- Expressiveness – concise programs
- Good programming environment
- Architecture independence and portability



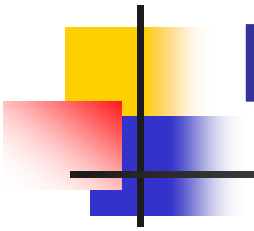
Features of a Good Language

- Readability
  - Simplicity
  - Orthogonality
  - Flexible control constructs
  - Rich data structures
  - Clear syntax design



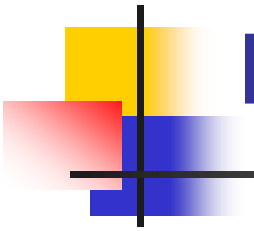
Features of a Good Language

- Writability
  - Simplicity
  - Orthogonality
  - Support for abstraction
  - Expressivity
  - Programming environment
  - Portability



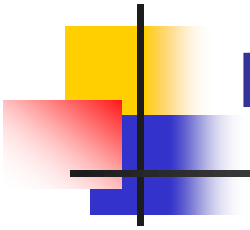
Features of a Good Language

- Usually readability and writability call for the same language characteristics
- Sometimes they conflict:
  - Comments: Nested comments (e.g /*… /* … */ … */) enhance writability, but decrease readability



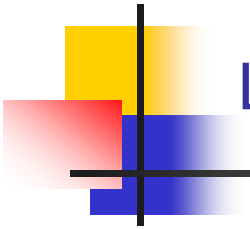
Features of a Good Language

- Reliability
  - Readability
  - Writability
  - Type Checking
  - Exception Handling
  - Restricted aliasing



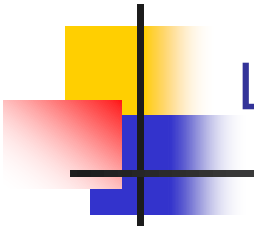
Language Paradigms – Imperative Languages

- Main focus: machine state – the set of values stored in memory locations
- Command-driven: Each statement uses current state to compute a new state
- Syntax: S1; S2; S3; ...
- Example languages: C, Pascal, FORTRAN, COBOL



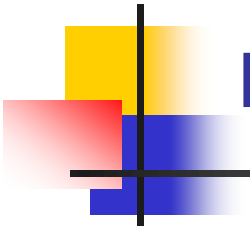
Language Paradigms – Object-oriented Languages

- Classes are complex data types grouped with operations (methods) for creating, examining, and modifying elements (objects); subclasses include (inherit) the objects and methods from superclasses



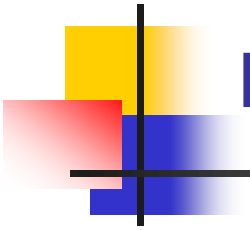
Language Paradigms – Object-oriented Languages

- Computation is based on objects sending messages (methods applied to arguments) to other objects
- Syntax: Varies, object <- method(args)
- Example languages: Java, C++, Smalltalk



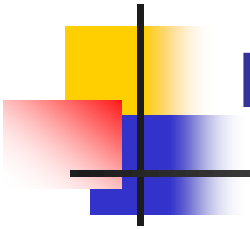
Language Paradigms – Applicative Languages

- Applicative (functional) languages
  - Programs as functions that take arguments and return values; arguments and returned values may be functions



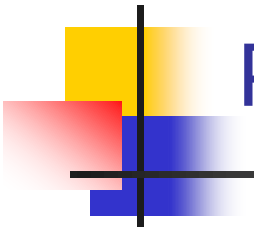
Language Paradigms – Applicative Languages

- Applicative (functional) languages
  - Programming consists of building the function that computes the answer; function application and composition main method of computation
  - Syntax: P1(P2(P3 X))
  - Example languages: ML, LISP, Scheme, Haskell, Miranda



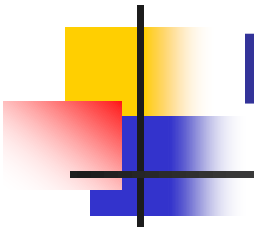
Language Paradigms – Logic Programming

- Rule-based languages
  - Programs as sets of basic rules for decomposing problem
  - Computation by deduction: search, unification and backtracking main components
  - Syntax: Answer :- specification rule
  - Example languages: (Prolog, Datalog,BNF Parsing)



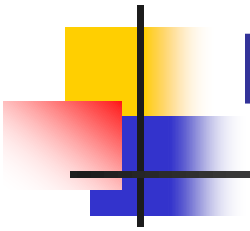
Programming Language Implementation

- Develop layers of machines, each more primitive than the previous
- Translate between successive layers
- End at basic layer
- Ultimately hardware machine at bottom



Basic Machine Components

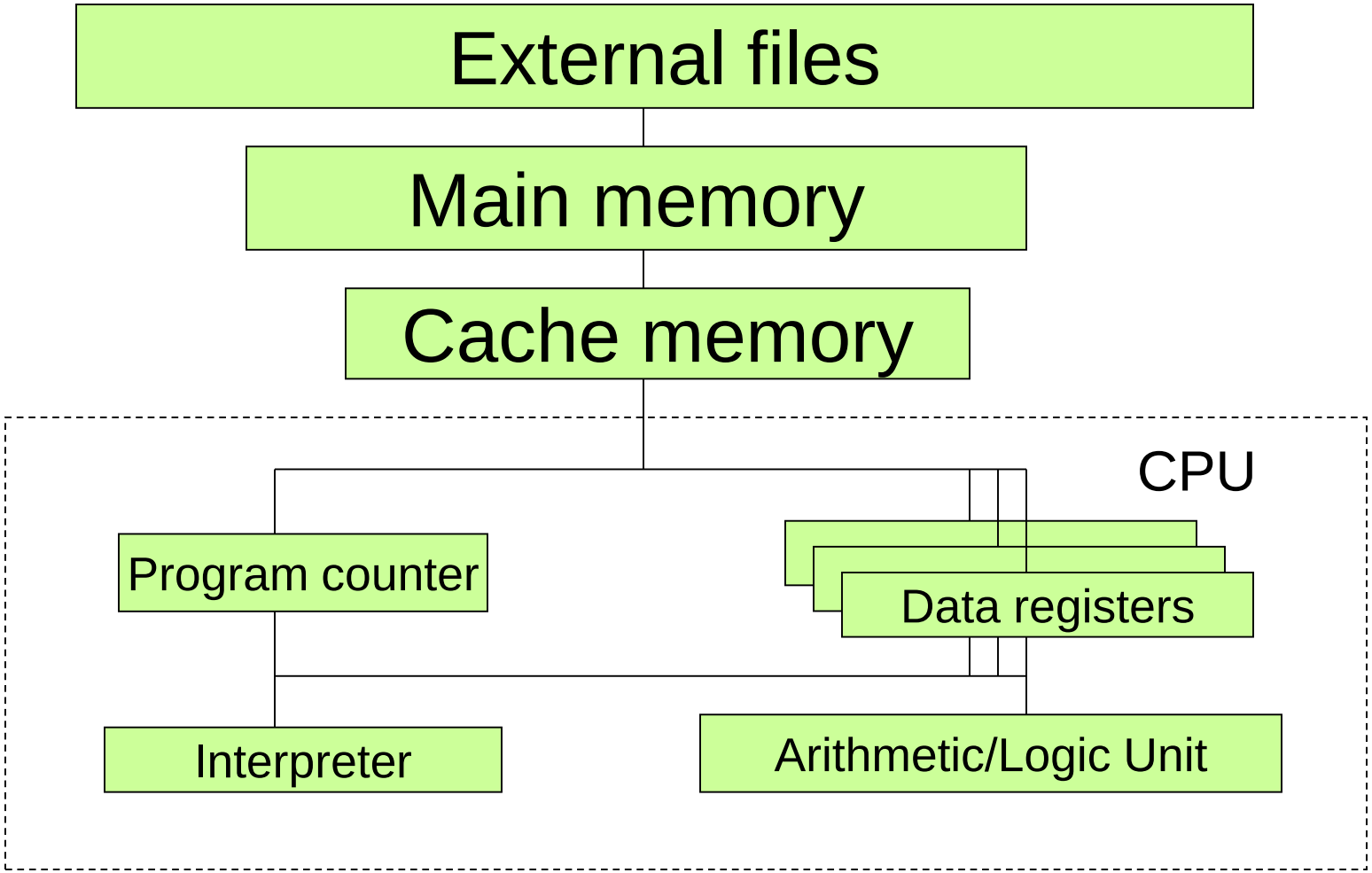
- **Data**: basic data types and elements of those types
- **Primitive operations**: for examining, altering, and combining data
- **Sequence control**: order of execution of primitive operations



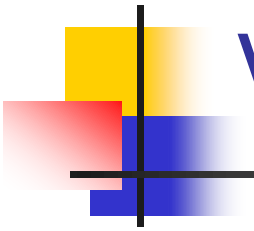
Basic Machine Components

- **Data access**: control of supply of data to operations
- **Storage management**: storage and update of program and data
- **External I/O**: access to data and programs from external sources, and output results

Basic Computer Architecture

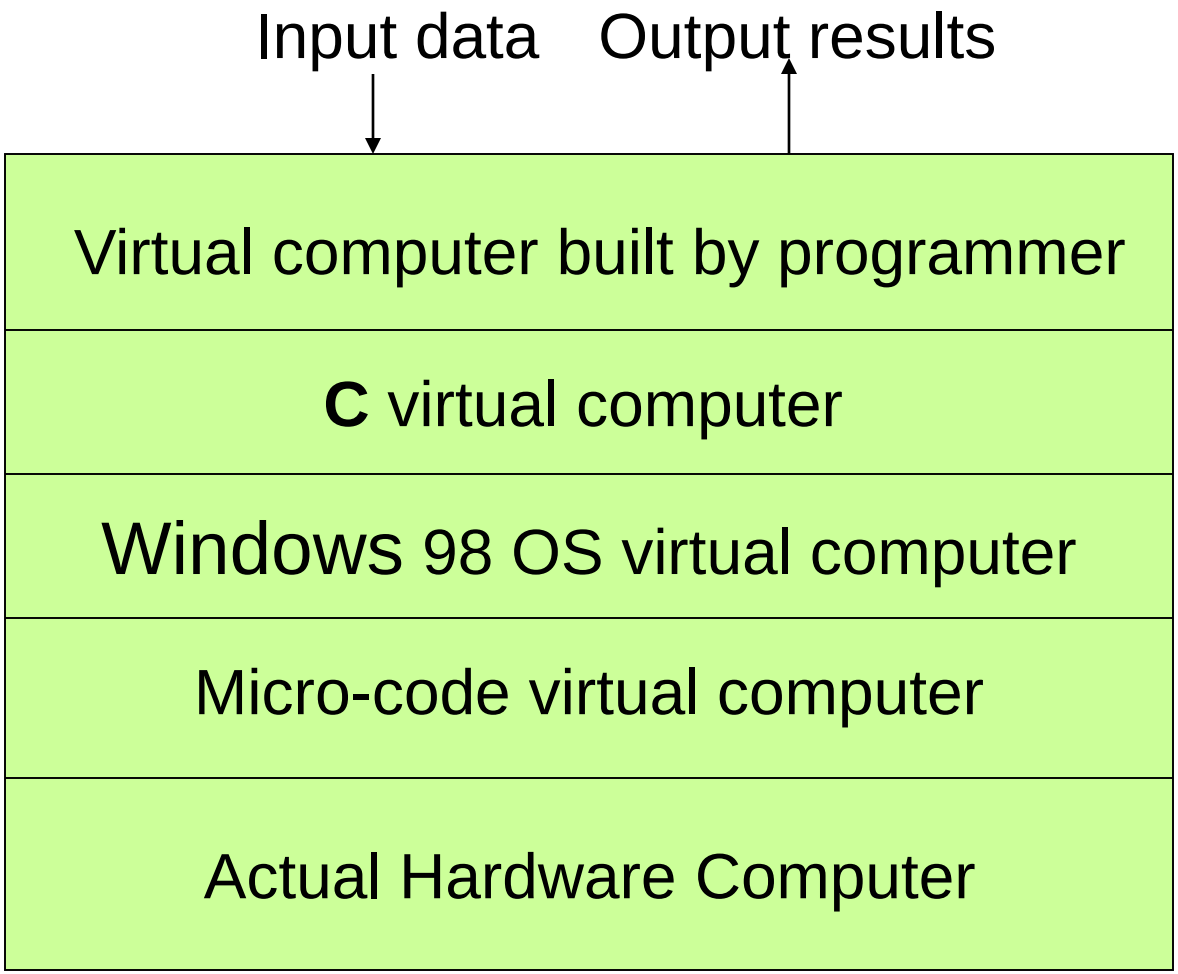


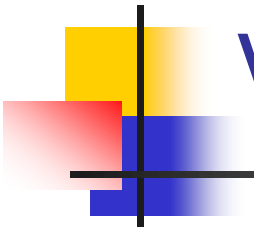
Virtual (Software) Machines



- At first, programs written in assembly language (or at very first, machine language)
- Hand-coded to be very efficient
- Now, no longer write in native assembly language
- Use layers of software (e.g. operating system)
- Each layer makes a *virtual machine* in which the next layer is defined

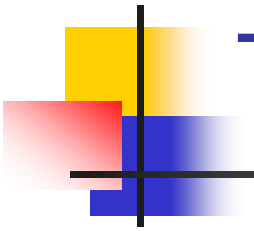
Example Layers of Virtual Computers for a **C** Program





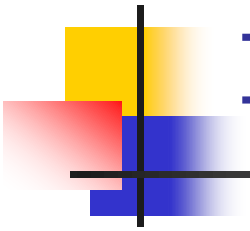
Virtual Machines Within Compilers

- Compilers often define layers of virtual machines
  - Functional languages: Untyped lambda calculus -> continuations -> generic pseudo-assembly -> machine specific code
  - May compile to intermediate language that is interpreted or compiled separately
    - Java virtual machine, CAML byte code



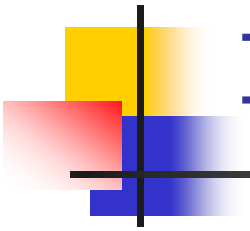
To Class

- Name some examples of virtual machines
- Name some examples of things that aren’t virtual machines



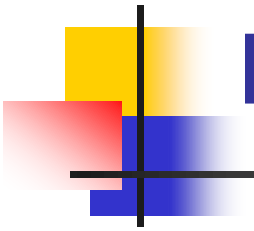
Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning



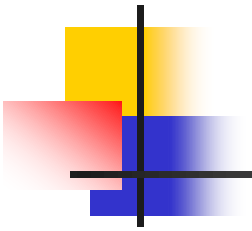
Interpretation Versus Compilation

- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed



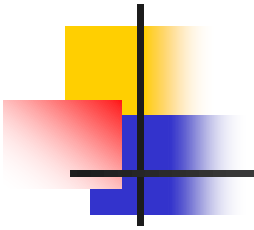
Program Aspects

- Syntax: what valid programs look like
- Semantics: what valid programs mean; what they should compute
- Compiler must contain both information



Major Phases of a Compiler

- Lex
  - Break the source into separate tokens
- Parse
  - Analyze phrase structure and apply semantic actions, usually to build an abstract syntax tree

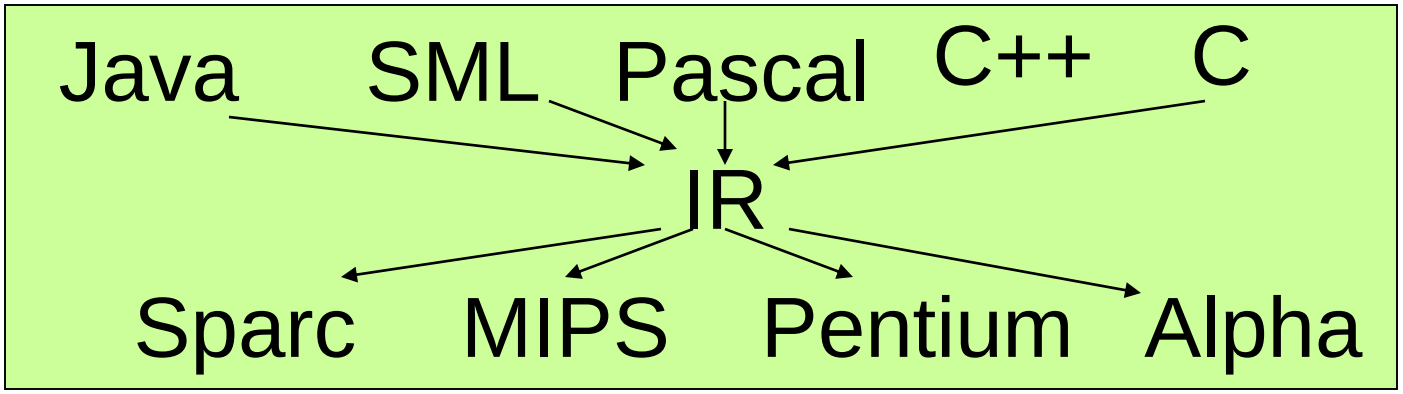


Major Phases of a Compiler

- Semantic analysis
  - Determine what each phrase means, connect variable name to definition (typically with symbol tables), check types

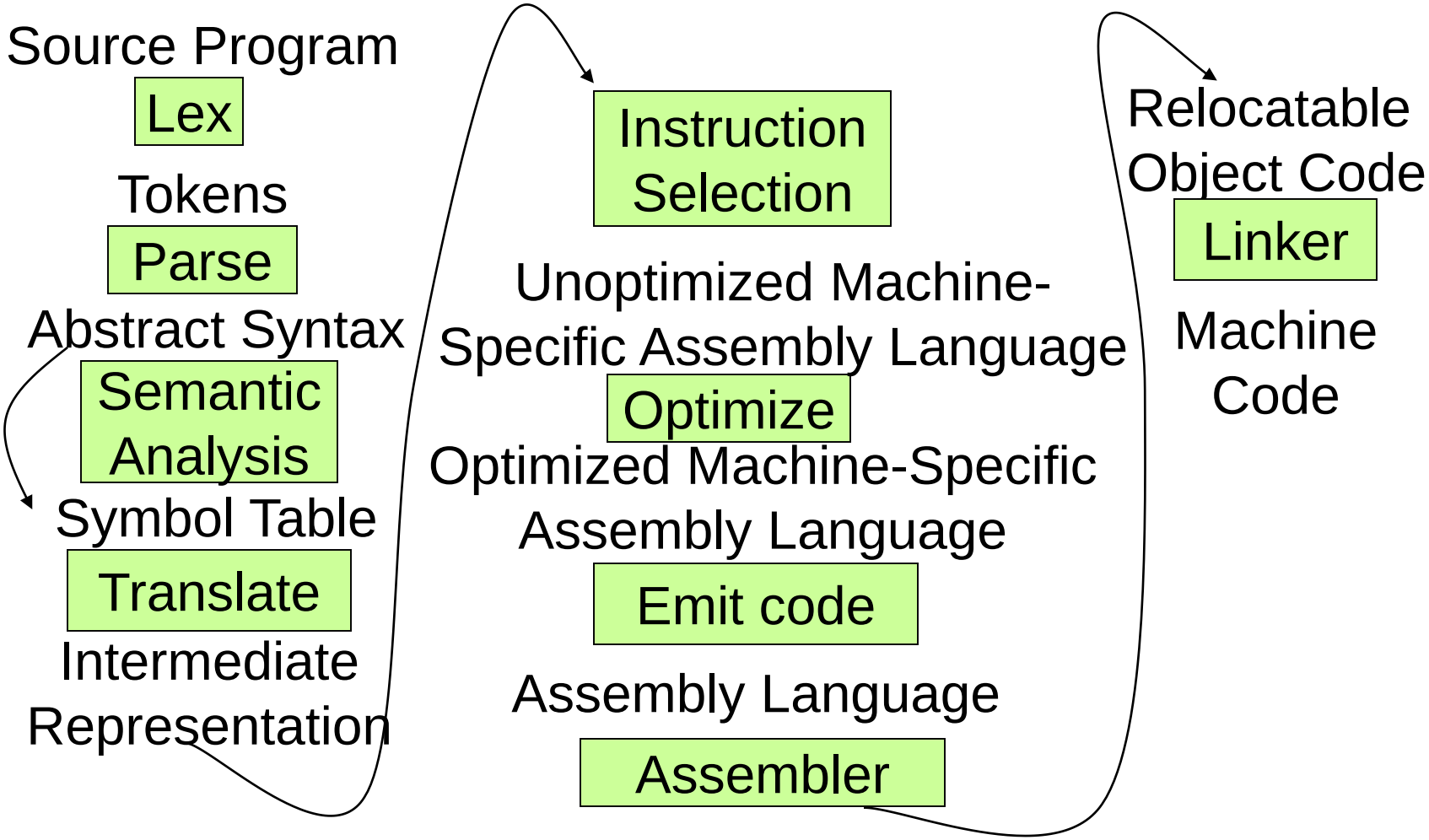
Major Phases of a Compiler

- Translate to intermediate representation



- Instruction selection
- Optimize
- Emit final machine code

Major Phases of a Compiler



Modified from “Modern Compiler Implementation in ML”, by Andrew Appel

Example of Intermediate Representation

- Program code: X = Y + Z + W
  - tmp = Y + Z
  - X = tmp + W
  - Simpler language with no compound arithmetic expressions

Example of Optimization

Program code: X = Y + Z + W

■ Load reg1 with Y ■ Load reg2 with Z ■ Add reg1 and reg2, saving to reg1 ■ Store reg1 to tmp **	■ Load reg1 with tmp ** ■ Load reg2 with W ■ Add reg1 and reg2, saving to reg1 ■ Store reg1 to X

Eliminate two steps marked **