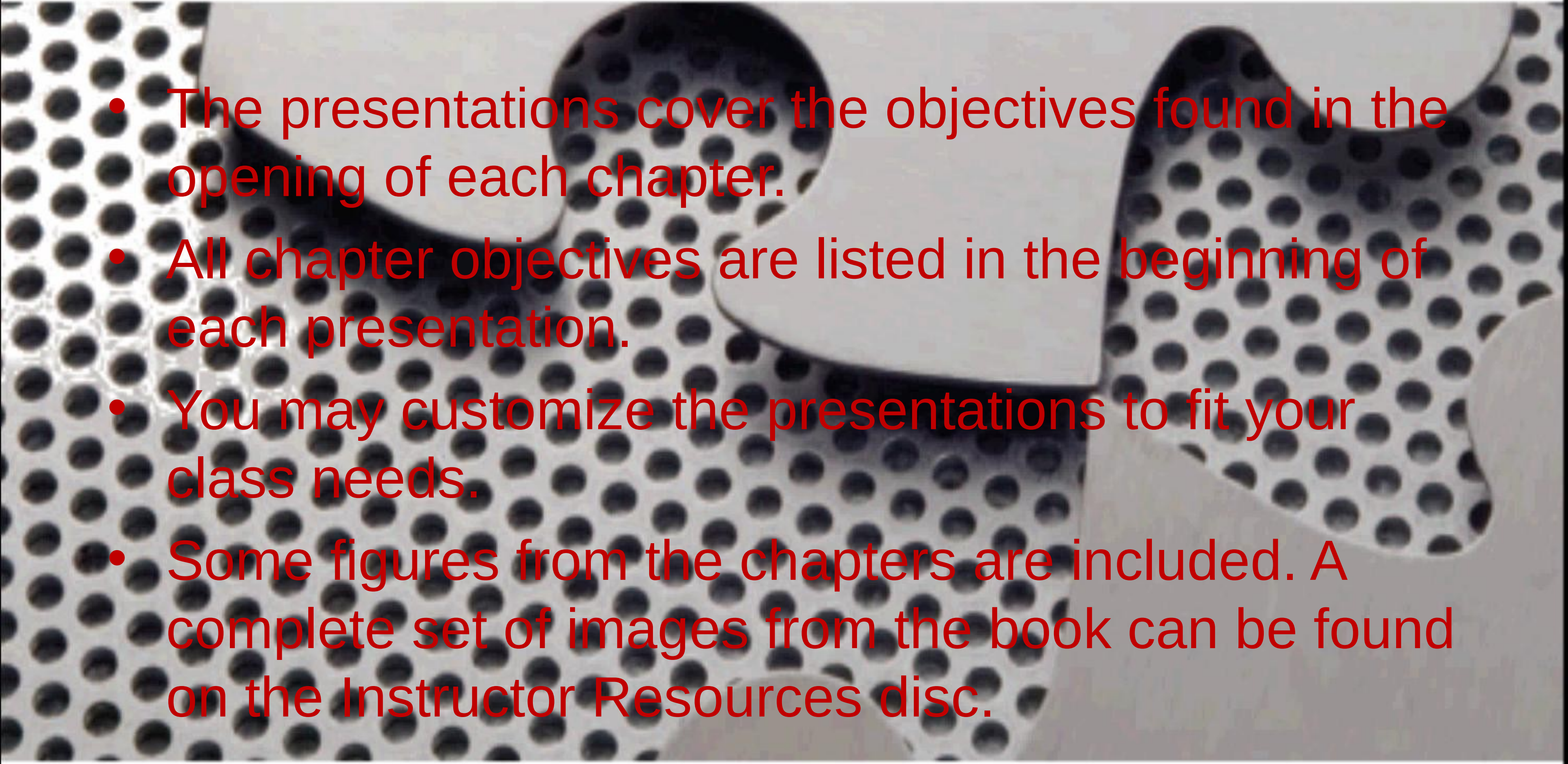
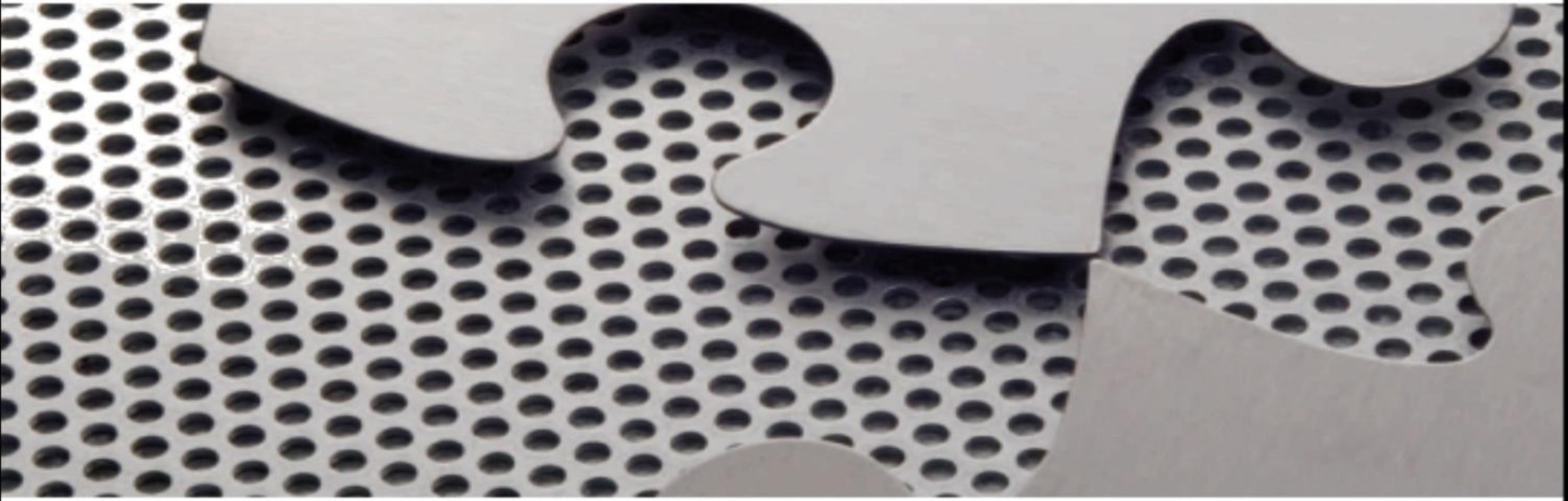
About the Presentations



- The presentations cover the objectives found in the opening of each chapter.
- All chapter objectives are listed in the beginning of each presentation.
- You may customize the presentations to fit your class needs.
- Some figures from the chapters are included. A complete set of images from the book can be found on the Instructor Resources disc.

Programming Languages Third Edition



Chapter 1

Introduction

Objectives

- Describe the origins of programming languages
- Understand abstractions in programming languages
- Understand computational paradigms
- Understand language definition
- Understand language translation
- Describe the future of programming languages

Introduction

- How we program computers influences how we think about computation, and vice versa
- Basic principles and concepts of programming languages are part of the fundamental body of knowledge of computer science
  - The study of these principles is essential to programmers and computer scientists
- This chapter introduces basic notions of programming languages and outlines some basic concepts

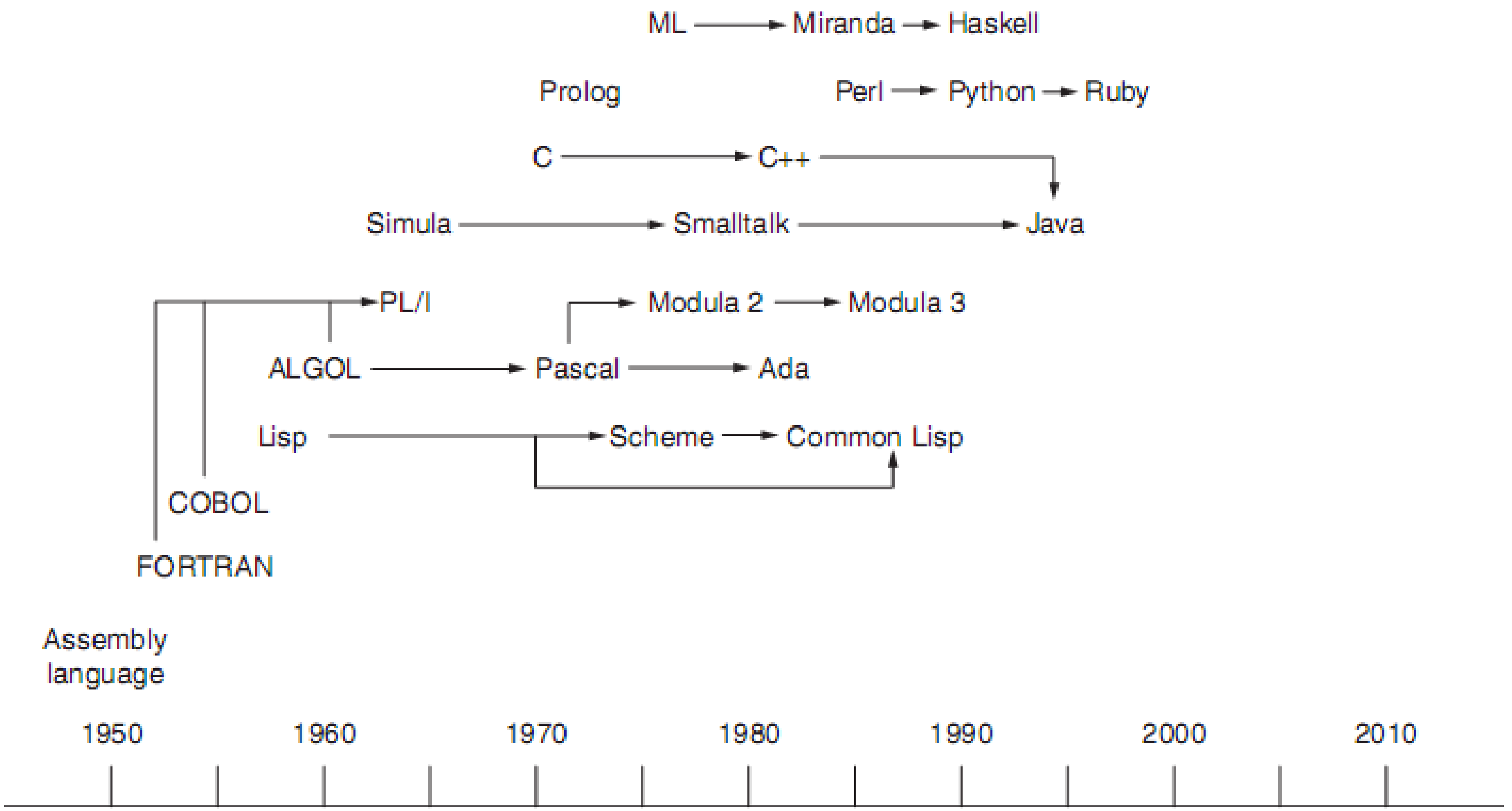


Figure 1.1 A programming language timeline

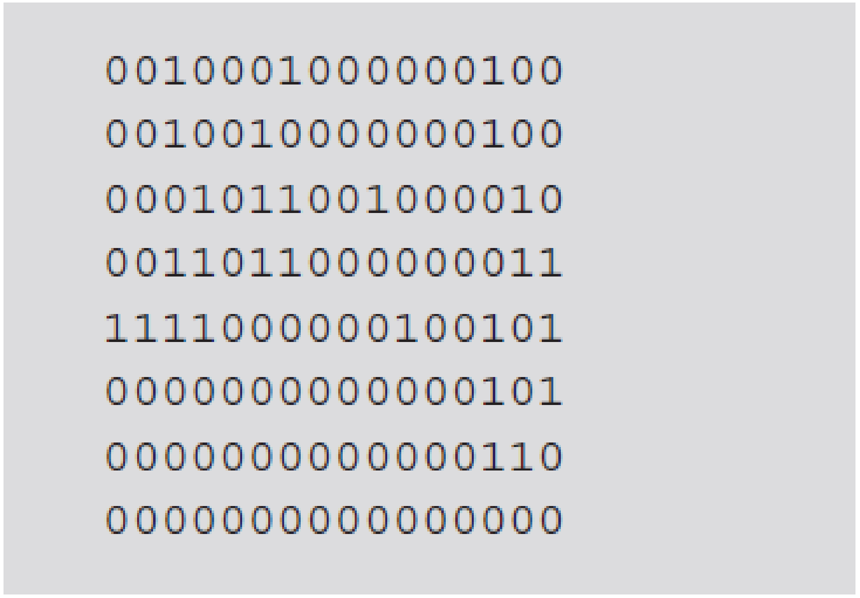
Origins of Programming Languages

- **Programming language**: often defined as “a notation for communicating to a computer what we want it to do”
- Before the mid 1940s, computer operators set switches to adjust the internal wiring of a computer to perform the requested tasks
- Programming languages allowed computer users to solve problems without having to reconfigure hardware

Machine Language and the First Stored Programs

- **John von Neumann**: proposed that computers should be permanently hardwired with a small set of general-purpose operations
  - Would allow the operator to input a series of binary codes to organize the basic hardware operations to solve more specific problems
  - Operators could flip switches to enter these codes, called machine language, into memory

Machine Language and the First Stored Programs (cont'd.)



```
0010001000000100
0010010000000100
0001011001000010
0011011000000011
1111000000100101
0000000000000101
0000000000000110
0000000000000000
```

Figure 1.2 A machine language program

Machine Language and the First Stored Programs (cont'd.)

- Each line of code has 16 bits or binary digits
  - Represents either a single machine language instruction or a single data value
- Program execution begins with the first line of code
  - Code is fetched from memory, decoded (interpreted), and executed
- Control then moves to the next line of code and continues until a halt instruction is reached
- **Opcode**: the first 4 bits of a line of code
  - Indicates the type of operation to be performed

Machine Language and the First Stored Programs (cont’d.)

- Next 12 bits contain code for the instruction’s operands
- Operand codes are either the numbers of machine registers or relate to addresses of other data or instructions in memory
- Machine language programming was tedious and error prone

Assembly Language, Symbolic Codes, and Software Tools

- **Assembly language**: a set of mnemonic symbols for instruction codes and memory locations
  - Example: `LD R1, FIRST`
- **Assembler**: a program that translates the symbolic assembly language code to binary machine code
- **Loader**: a program that loads the machine code into computer memory
- Input devices:
  - Keypunch machine
  - Card reader

Assembly Language, Symbolic Codes, and Software Tools (cont'd.)

```
.ORIG x3000        ; Address (in hexadecimal) of the first instruction
LD  R1, FIRST      ; Copy the number in memory location FIRST to register R1
LD  R2, SECOND     ; Copy the number in memory location SECOND to register R2
ADD R3, R2, R1     ; Add the numbers in R1 and R2 and place the sum in
                   ; register R3
ST  R3, SUM        ; Copy the number in R3 to memory location SUM
HALT               ; Halt the program
FIRST  .FILL #5    ; Location FIRST contains decimal 5
SECOND .FILL #6    ; Location SECOND contains decimal 6
SUM    .BLKW #1    ; Location SUM (contains 0 by default)
.END               ; End of program
```

Figure 1.3 An assembly language program that adds two numbers

Assembly Language, Symbolic Codes, and Software Tools (cont'd.)

- Mnemonic symbols were an improvement over binary machine codes but still had shortcomings
  - Lacks abstraction of conventional mathematical notation
  - Each type of computer hardware architecture has its own machine language instruction set and requires its own dialect of assembly language
- Assembly languages first appeared in the 1950s and are still used today for low-level system tools or for hand-optimization

FORTRAN and Algebraic Notation

- **FORTRAN**: FORmula TRANslation language
  - Developed by John Backus in the early 1950s
  - Reflected the architecture of a particular type of machine
  - Lacked the structured control statements and data structures of later high-level languages
- Popular with scientists and engineers for its support for algebraic notation and floating-point numbers

The ALGOL Family: Structured Abstractions and Machine Independence

- **ALGOL**: ALGOrithmic Language released in 1960
  - Provided a standard notation for computer scientists to publish algorithms in journals
  - Included structured control statements for sequencing (`begin-end` blocks), loops (`for` loop), and selection (`if` and `if-else` statements)
  - Supported different numeric types
  - Introduced the array structure
  - Supported procedures, including recursive procedures

The ALGOL Family (cont’d.)

- ALGOL achieved machine independence with the requirement for an ALGOL compiler with each type of hardware
- **Compiler**: translates programming language statements into machine code
- ALGOL was the first language to receive a formal specification or definition
  - Included a grammar that defined its features for both programmers and for compiler writers

The ALGOL Family (cont’d.)

- Large number of high-level languages descended from ALGOL, including:
  - **Pascal**: language for teaching programming in the 1970s
  - **Ada**: for embedded applications of U.S. Dept. of Defense

Computation without the von Neumann Architecture

- High-level languages still echoed the underlying architecture of the von Neumann model of a machine
  - Memory area for programs and data are stored
  - Separate central processing unit that sequentially executes instructions fetched from memory
- Improvements in processor speed and increasing abstraction in programming languages led to the information age

Computation without the von Neumann Architecture (cont'd.)

- Progress in language abstraction and hardware performance ran into separate roadblocks:
  - Hardware began to reach the limits of improvements predicted by Moore's Law, leading to the multi-core approach
  - Large programs were difficult to debug and correct
  - Single-processor model of computation cannot be easily mapped into new architecture of multiple CPUs executing in parallel

Computation without the von Neumann Architecture (cont'd.)

- Solution: languages need not be based on a particular model of hardware but need only to support models of computation suitable for styles of problem solving
- **Lambda calculus**: computational model developed by mathematician Alonzo Church
  - Based on the theory of recursive functions
- **Lisp**: programming language that uses the functional model of computation

Computation without the von Neumann Architecture (cont'd.)

- Other languages modeled on non-von Neumann models of computing that lend themselves to parallel processing include:
  - A formal logic model with automatic theorem proving
  - A model involving the interaction of objects via message passing

Abstractions in Programming Languages

- Two types of programming language abstractions:
  - Data abstraction
  - Control abstraction
- **Data abstractions**: simplify the behavior and attributes of data for humans
  - Examples: numbers, character strings, search trees
- **Control abstractions**: simplify properties of the transfer of control
  - Examples: loops, conditional statements, procedure calls

Abstractions in Programming Languages (cont'd.)

- Abstractions can also be categorized by levels (measures of the amount of information contained or hidden in the abstraction)
- **Basic abstractions**: collect the most localized machine information
- **Structured abstractions**: collect intermediate information about the structure of a program
- **Unit abstractions**: collect large-scale information in a program

Data: Basic Abstractions

- **Basic data abstraction**:
  - Hides internal representation of common data values
    - Values are also called “primitive” or “atomic” because the programmer cannot normally access the component parts or bits of the internal representation
  - **Variables**: use of symbolic names to hide computer memory locations containing data values
  - **Data types**: names given to kinds of data values
  - **Declaration**: the process of giving a variable a name and a data type

Data: Basic Abstractions (cont’d.)

- Basic data abstractions (cont’d.):
  - Standard mathematical operations, such as addition and multiplication

Data: Structured Abstractions

- **Data structure**: collects related data values into a single unit
  - Hides component parts but can be constructed from parts, and parts can be accessed and modified
- Examples:
  - Employee record contains name, address, phone, salary (different data types)
  - **Array**: sequence of individually indexed items with the same data type
  - Text file: a sequence of characters for transfer to and from an external storage device

Data: Unit Abstractions

- **Information hiding**: defining new data types (data and operations) that hide information
- **Unit abstraction**: often associated with the concept of an abstract data type
  - A set of data values and the operations on those values
- Separates the interface from the implementation
  - **Interface**: set of operations available to the user
  - **Implementation**: internal representation of data values and operations

Data: Unit Abstractions (cont'd.)

- Examples:
  - **Module** in ML, Haskell, and Python
  - **Package** in Lisp, Ada, and Java
  - Class mechanism in object-oriented languages
- Unit abstraction also provides **reusability**
- Typically, **components** are entered into a **library** and become the basis for library mechanisms
  - **Interoperability** allows combination of units
- **Application programming interface (API)**: gives information about the resource's components

Control: Basic Abstractions

- **Basic control abstractions**: statements that combine a few machine instructions into an abstract statement that is easier to understand
- **Syntactic sugar**: a mechanism that allows you to replace a complex notation with a simpler, shorthand notation
  - Example: x += 10 instead of x = x + 10

Control: Structured Abstractions

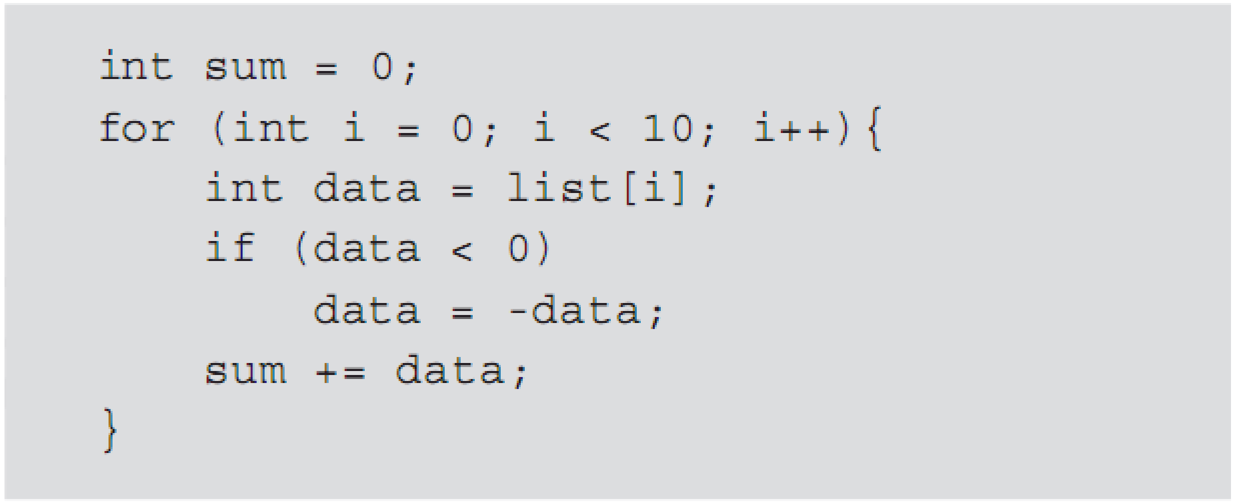
- **Structured control abstractions**: divide a program into groups of instructions nested within tests that govern their execution
  - Help to express the logic of primary control structures of sequencing, selection, and iteration
- **Branch instructions**: instructions that support selection and iteration to memory locations other than the next one

Control: Structured Abstractions (cont'd.)

```
      LEA   R1, LIST        ; Load the base address of the array (the first cell)
      AND   R2, R2, #0      ; Set the sum to 0
      AND   R3, R3, #0      ; Set the counter to 10 (to count down)
      ADD   R3, R3, #10
WHILE LDR   R4, R1, #0      ; Top of the loop: load the datum from the current
                            ; array cell
      BRZP INC              ; If it's >= 0, skip next two steps
      NOT   R4, R4          ; It was < 0, so negate it using twos complement
                            ; operations
      ADD   R4, R4, #1
INC   ADD   R2, R2, R4      ; Increment the sum
      ADD   R1, R1, #1      ; Increment the address to move to the next array
                            ; cell
      ADD   R3, R3, #-1     ; Decrement the counter
      BRP   WHILE           ; Goto the top of the loop if the counter > 0
      ST    R2, SUM         ; Store the sum in memory
```

Figure 1.4 An array-based loop in assembly language

Control: Structured Abstractions (cont'd.)

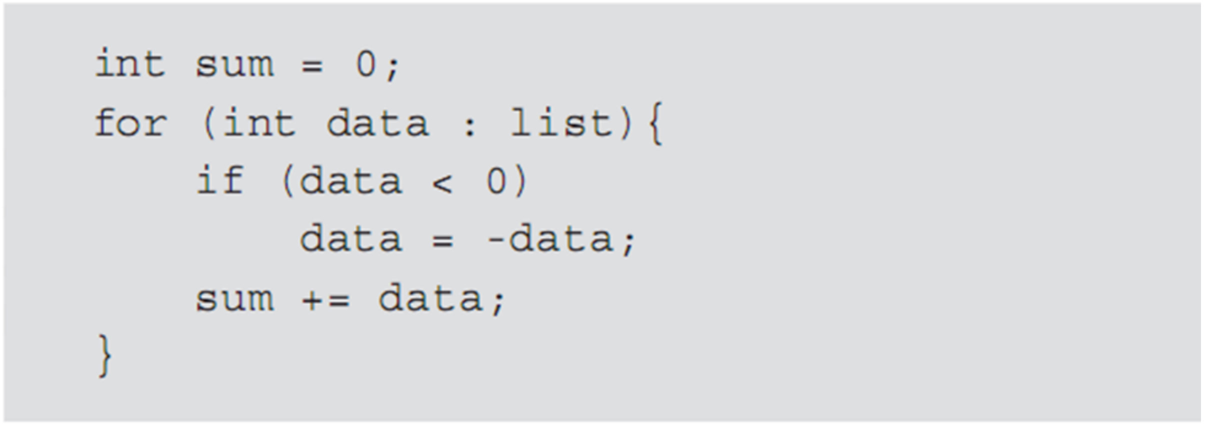


```
int sum = 0;
for (int i = 0; i < 10; i++){
    int data = list[i];
    if (data < 0)
        data = -data;
    sum += data;
}
```

Figure 1.5 An array-based loop in C++ or Java

Control: Structured Abstractions (cont'd.)

- **Iterator**: an object associated with a collection (such as array, list, set, or tree)
  - Open an iterator on a collection, then visit all the elements by using the iterator's methods
- Syntactic sugar for iterator in Java: enhanced for loop

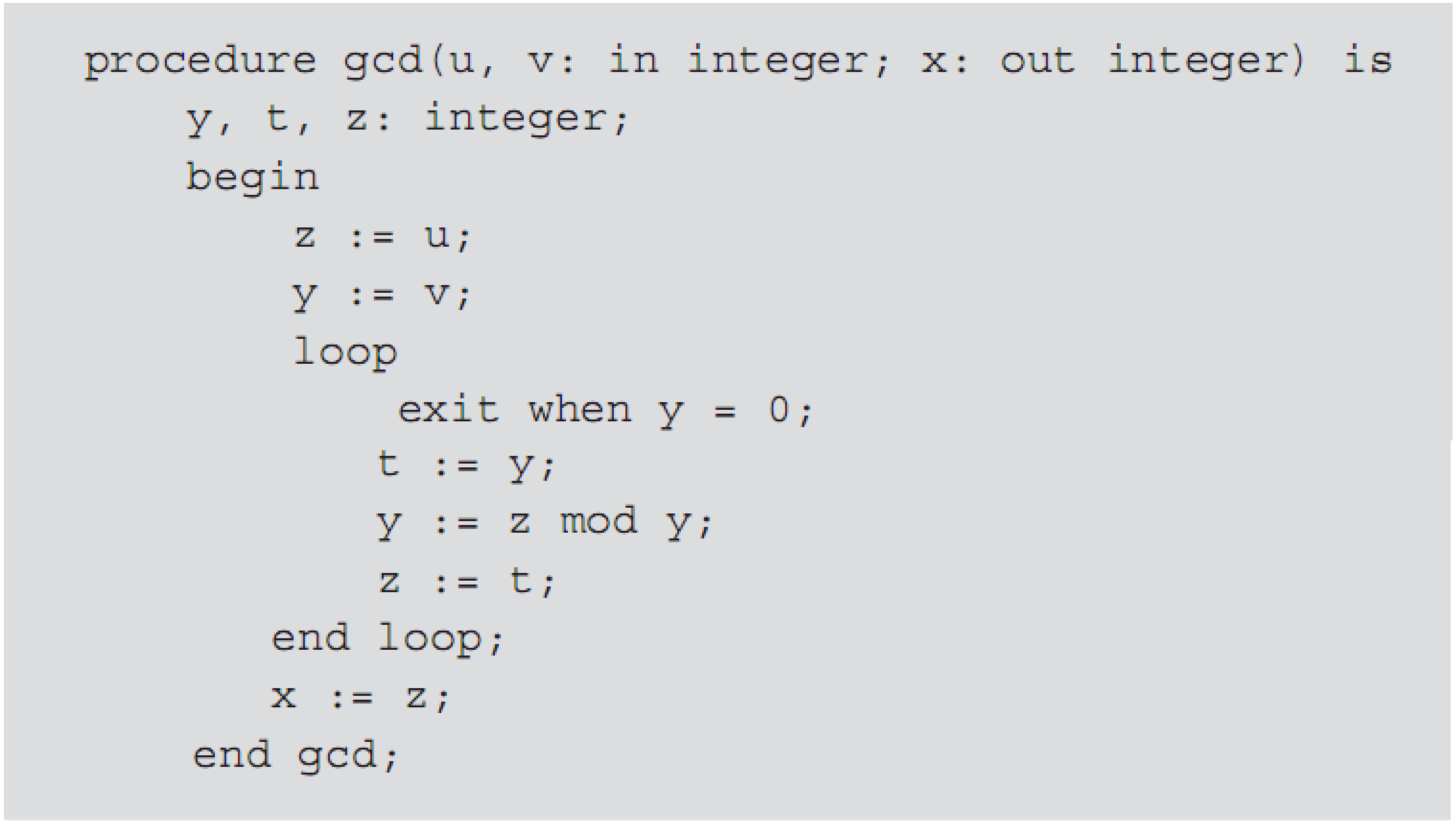


```
int sum = 0;
for (int data : list){
    if (data < 0)
        data = -data;
    sum += data;
}
```

Control: Structured Abstractions (cont'd.)

- **Procedure** (or **subprogram** or **subroutine**): groups a sequence of actions into a single action that can be called or invoked from other points in the program
  - **Procedure declaration**: names a procedure and associates it with the actions to be performed
  - **Invocation** (or **procedure activation**): the act of calling the procedure
  - **Parameters**: values that can change from call to call
  - **Arguments** (or **actual parameters**): values supplied by the caller for the parameters

Control: Structured Abstractions (cont'd.)



```
procedure gcd(u, v: in integer; x: out integer) is
    y, t, z: integer;
    begin
        z := u;
        y := v;
        loop
            exit when y = 0;
           t := y;
           y := z mod y;
           z := t;
       end loop;
       x := z;
    end gcd;
```

Figure 1.6 An Ada gcd procedure

Control: Structured Abstractions (cont'd.)

- **Runtime environment**: the system implementation of the program
  - Stores information about the condition of the program and the way procedures operate
- **Function**: closely related to a procedure
  - Returns a value or result to its caller
  - Can be written to correspond more closely to mathematical abstractions
- **Recursion**: a mechanism that further exploits the abstraction mechanism

Control: Structured Abstractions (cont'd.)

```
function gcd(u, v: in integer) return integer is
    begin
        if v = 0
            return u;
        else
            return gcd(v, u mod v);
        end if;
    end gcd;
```

Figure 1.7 An Ada gcd function

Control: Structured Abstractions (cont'd.)

- **Higher-order functions**: functions that can accept other functions as arguments and return functions as values
- Example: **map** function
  - Expects another function and a collection as arguments
  - Applies the argument function to each element in the argument collection and returns a list of results

```
(map abs (list 33 -10 66 88 -4)    ; Returns (33 10 66 88 4)
```

Control: Unit Abstractions

- **Unit**: a stand-alone collection of procedures providing logically related services to other parts of a program
  - Allows a program to be understood as a whole without needing to know the details of the services provided by the unit
- **Threads**: separately executed control paths within the Java system
- **Processes**: other programs executing outside the Java system
- **Task**: mechanism in Ada for parallel execution

Computational Paradigms

- **Imperative language**: a language with three properties
  - Sequential execution of instructions
  - Use of variables representing memory locations
  - Use of assignment to change the values of variables
- Represents one **paradigm** (pattern) for programming languages
- **von Neumann bottleneck**: requirement that a program be described as a sequence of instructions

Computational Paradigms (cont’d.)

- **Functional paradigm**:
  - Based on the abstract notion of a function in lambda calculus
- **Logic paradigm**:
  - Based on symbolic logic
- Both functional and logic paradigms correspond to mathematical foundations
  - Makes it easier to determine if a program will execute correctly

Computational Paradigms (cont'd.)

- **Object-oriented paradigm**:
  - Reusable code that operates in a way to mimic behaviors of real-world objects

Language Definition

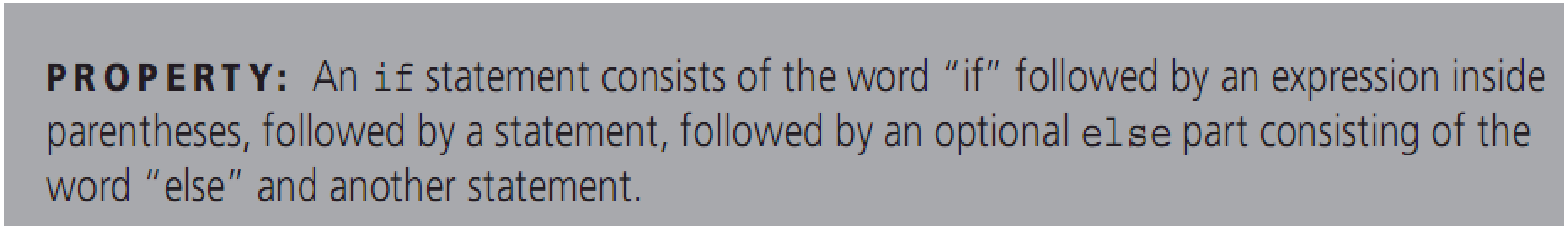
- Formal language definition provides benefits:
  - Helps to allow you to reason mathematically about programs
  - Promotes standardization for machine or implementation independence
  - Defines program behavior and interaction
  - Ensures discipline when a language is designed
- Language definition can be loosely divided into:
  - **Syntax**, or structure
  - **Semantics**, or meaning

Language Syntax

- **Language syntax**: similar to the grammar of a natural language
- **Grammar**: formal definition of the language’s syntax
- **Lexical structure**: structure of the language’s words
  - Similar to spelling in natural languages
- **Tokens**: the language’s words
  - Includes keywords, identifiers, symbols for operations, special punctuation symbols, etc.

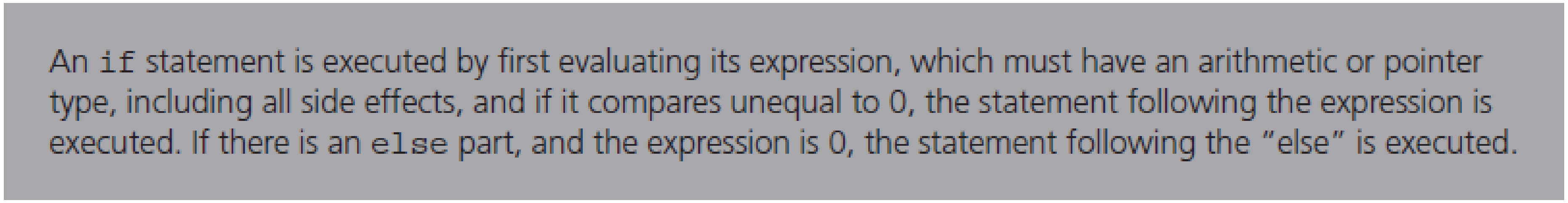
Language Syntax (cont'd.)

- Example: `if` statement in C


PROPERTY: An `if` statement consists of the word "if" followed by an expression inside parentheses, followed by a statement, followed by an optional `else` part consisting of the word "else" and another statement.

Language Semantics

- **Semantics**: meaning of a language
  - Describes the effects of executing the code
  - Difficult to provide a comprehensive description of meaning in all contexts
- Example: `if` statement in C


An `if` statement is executed by first evaluating its expression, which must have an arithmetic or pointer type, including all side effects, and if it compares unequal to 0, the statement following the expression is executed. If there is an `else` part, and the expression is 0, the statement following the "else" is executed.

Language Semantics (cont'd.)

- No generally accepted formal method for describing semantics
- Several notational systems have been developed:
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics

Language Translation

- **Translator**: a program that accepts other programs and either directly executes them or transforms them into a form suitable for execution
- Two major types of translators:
  - **Interpreter**: executes a program directly
  - **Compiler**: produces an equivalent program in a form suitable for execution

Language Translation (cont'd.)

- Interpretation is a one-step process
  - Both the program and the input are provided to the interpreter, and the output is obtained

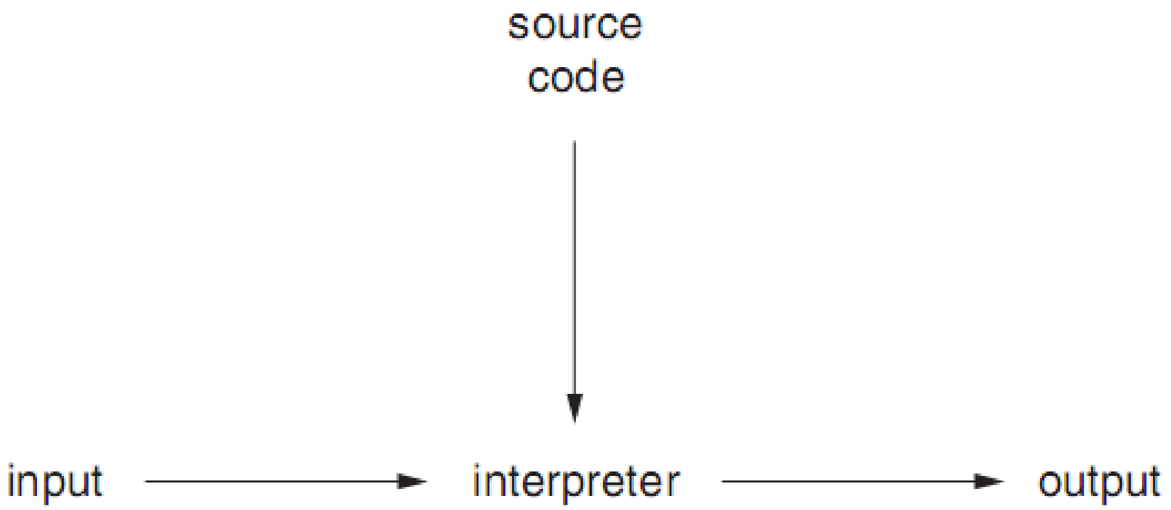


Figure 1.8 The interpretation process

Language Translation (cont'd.)

- Compilation requires at least two steps
  - **Source program** is input to the compiler
  - **Target program** is output from the compiler
- Target language is often assembly language, so the target program must be:
  - Translated by an **assembler** into an object program
  - **Linked** with other object programs
  - **Loaded** into appropriate memory locations

Language Translation (cont'd.)

- Target language may be **byte code** (a form of low-level code)
  - Byte code is then executed by an interpreter called a virtual machine
- **Virtual machine**: written differently for different hardware architectures
  - Byte code is machine-independent
  - Examples: Java, Python

Language Translation (cont'd.)

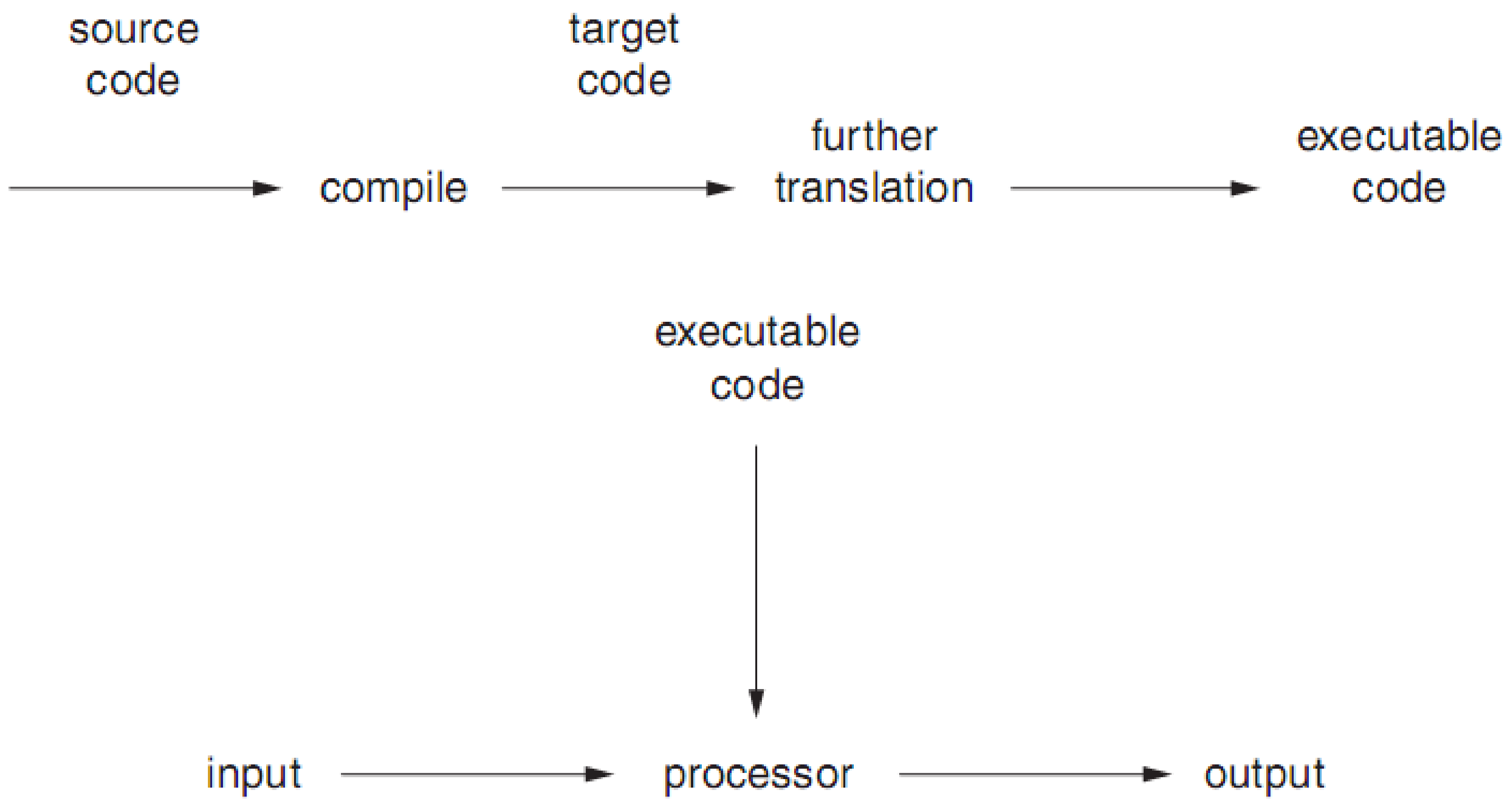


Figure 1.9 The compilation process

Language Translation (cont'd.)

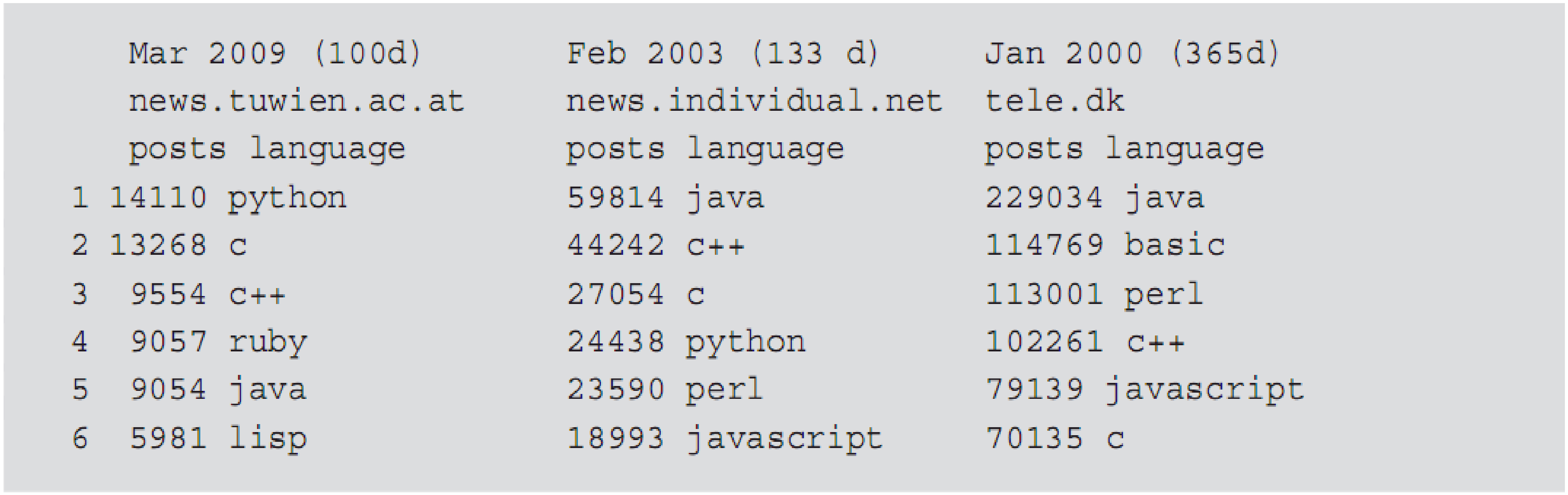
- Possible for a language to be defined by the behavior of a particular interpreter or compiler
  - Called a **definitional** translator
  - Not common

Future of Programming Languages

- In the 1960s, computer scientists wanted a single universal programming language to meet all needs
- In the late 1970s and early 1980s, they wanted specification languages that would allow users to define the needs and then generate the system
  - This is what logic programming languages attempt to do
- Programming has not become obsolete
  - New languages will arise to support new technologies that arise

Future of Programming Languages (cont'd.)

- Relative popularity of programming languages since 2000, based on number of posts on newsgroups:



	Mar 2009 (100d)		Feb 2003 (133 d)		Jan 2000 (365d)	
	news.tuwien.ac.at		news.individual.net		tele.dk	
	posts	language	posts	language	posts	language
1	14110	python	59814	java	229034	java
2	13268	c	44242	c++	114769	basic
3	9554	c++	27054	c	113001	perl
4	9057	ruby	24438	python	102261	c++
5	9054	java	23590	perl	79139	javascript
6	5981	lisp	18993	javascript	70135	c

Figure 1.10 Popularity of programming languages (source: *www.complang.tuwien.ac.at/anton/comp.lang-statistics/*)