Ceng 111 – Fall 2018
Week 6

The World of Programming

Binary Representation

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.

Today

- The world of programming
- Binary representations

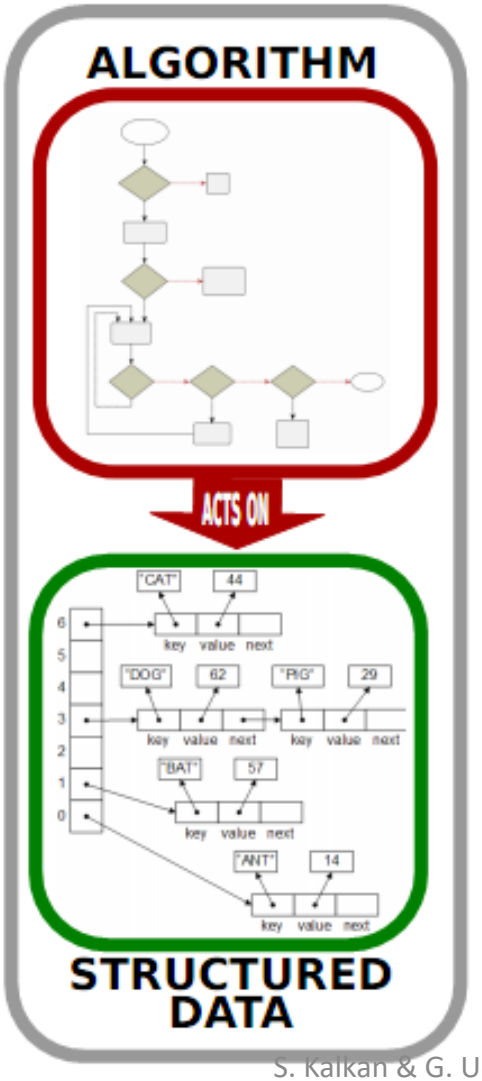
Administrative issues

- Live sessions schedule change
  - Tue 13:40: Common session
  - Wed 10:40: Section 3
  - Wed 15:40: Section 2
  - Thu 15:40: Section 1
- Social session
- The labs
- Midterm: 11 December 17:40

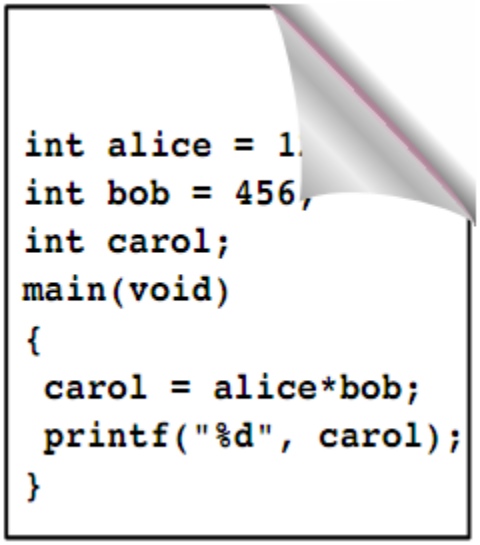
Program, Programming

WORLD PROBLEM

TRANSFORMED



IMPLEMENTED

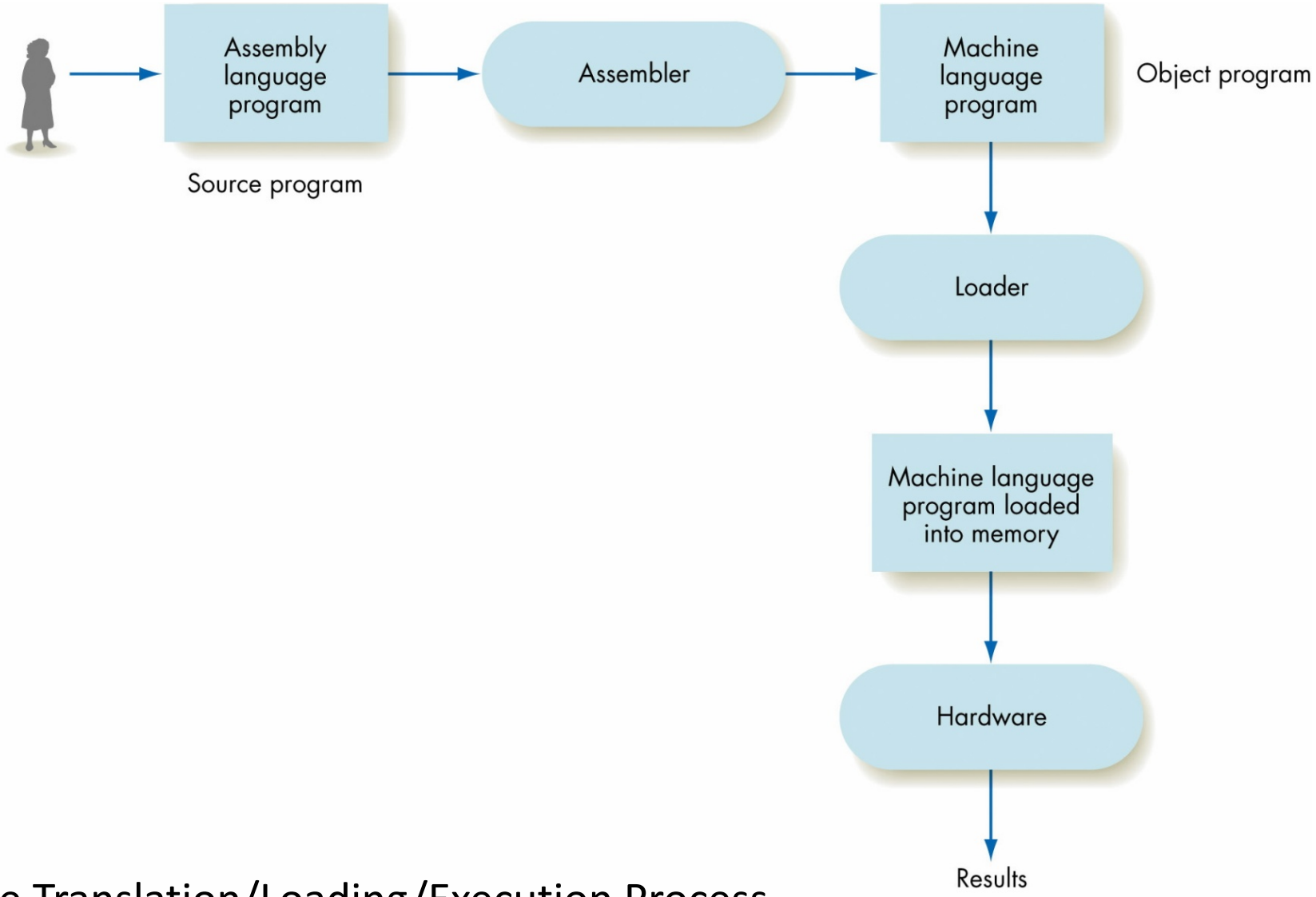


```
int alice = 1
int bob = 456,
int carol;
main(void)
{
 carol = alice*bob;
 printf("%d", carol);
}
```

PROGRAM

Program, Programming

- Program:
  - "a series of steps to be carried out or goals to be accomplished"
- A recipe for cooking a certain dish is also a program (but not a computer program).



The Translation/Loading/Execution Process

Assembly Language (continued)

- Source program
  - An assembly language program
- Object program
  - A machine language program
- Assembler
  - Translates a source program into a corresponding object program

Assembly Language (continued)

- Advantages of writing in assembly language rather than machine language
  - Use of symbolic operation codes rather than numeric (binary) ones
  - Use of symbolic memory addresses rather than numeric (binary) ones
  - Pseudo-operations that provide useful user-oriented services such as data generation

```
.BEGIN      --This must be the first line of the program.
      ⋮     --Assembly language instructions like those in Figure 6.5.
    HALT    --This instruction terminates execution of the program
      ⋮     --Data generation pseudo-ops such as
            --.DATA are placed here, after the HALT.
.END  --This must be the last line of the program.
```

Structure of a Typical Assembly Language Program

Examples of Assembly Language Code

- Arithmetic expression

  A = B + C – 7

  (Assume that B and C have already been assigned values)

Examples of Assembly Language Code (continued)

- Assembly language translation

```
            LOAD        B       --Put the value B into register R.
            ADD         C       --R now holds the sum (B + C).
            SUBTRACT SEVEN      --R now holds the expression (B + C - 7).
            STORE       A       --Store the result into A.
              :
              :                 --These data should be placed after the HALT.
A:          .DATA       0
B:          .DATA       0
C:          .DATA       0
SEVEN:      .DATA       7       --The constant 7.
```

Examples of Assembly Language Code (continued)

- Problem
  - Read in a sequence of non-negative numbers, one number at a time, and compute their sum
  - When you encounter a negative number, print out the sum of the non-negative values and stop

Step	Operation
1	Set the value of Sum to 0
2	Input the first number *N*
3	While *N* is not negative do
4	Add the value of *N* to Sum
5	Input the next data value *N*
6	End of the loop
7	Print out Sum
8	Stop

Algorithm to Compute the Sum of Numbers

```
            .BEGIN                  --This marks the start of the program.
            CLEAR       SUM         --Set the running sum to 0 (line 1).
            IN          N           --Input the first number N (line 2).
--The next three instructions test whether N is a negative number (line 3).
AGAIN:      LOAD        ZERO        --Put 0 into register R.
            COMPARE     N           --Compare N and 0.
            JUMPLT      NEG         --Go to NEG if N < 0.
--We get here if N ≥ 0. We add N to the running sum (line 4).
            LOAD        SUM         --Put SUM into R.
            ADD         N           --Add N. R now holds (N + SUM).
            STORE       SUM         --Put the result back into SUM.
--Get the next input value (line 5).
            IN          N
--Now go back and repeat the loop (line 6).
            JUMP        AGAIN
--We get to this section of the program only when we encounter a negative value.
NEG:        OUT         SUM         --Print the sum (line 7)
            HALT                    --and stop (line 8).
--Here are the data generation pseudo-ops
SUM:        .DATA       0           --The running sum goes here.
N:          .DATA       0           --The input data are placed here.
ZERO:       .DATA       0           --The constant 0.
--Now we mark the end of the entire program.
            .END
```

Assembly Language Program to Compute the Sum of Nonnegative Numbers

Translation and Loading

- Before a source program can be run, an assembler and a loader must be invoked
- Assembler
  - Translates a symbolic assembly language program into machine language
- Loader
  - Reads instructions from the object file and stores them into memory for execution

Translation and Loading (continued)

- Assembler tasks
  - Convert symbolic op codes to binary
  - Convert symbolic addresses to binary
  - Perform assembler services requested by the pseudo-ops
  - Put translated instructions into a file for future use

Programming Languages

C code for the example problem:

```
int alice = 123;
int bob = 456;
int carol;
main(void)
{
   carol = alice*bob;
}
```

Programming Languages

Assembly code for the example problem:

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    alice(%rip), %edx
        movl    bob(%rip), %eax
        imull   %edx, %eax
        movl    %eax, carol(%rip)
        movl    $0, %eax
        leave
        ret
alice:
        .long   123
bob:
        .long   456
```

Programming Languages

Machine code for a simple problem/program:

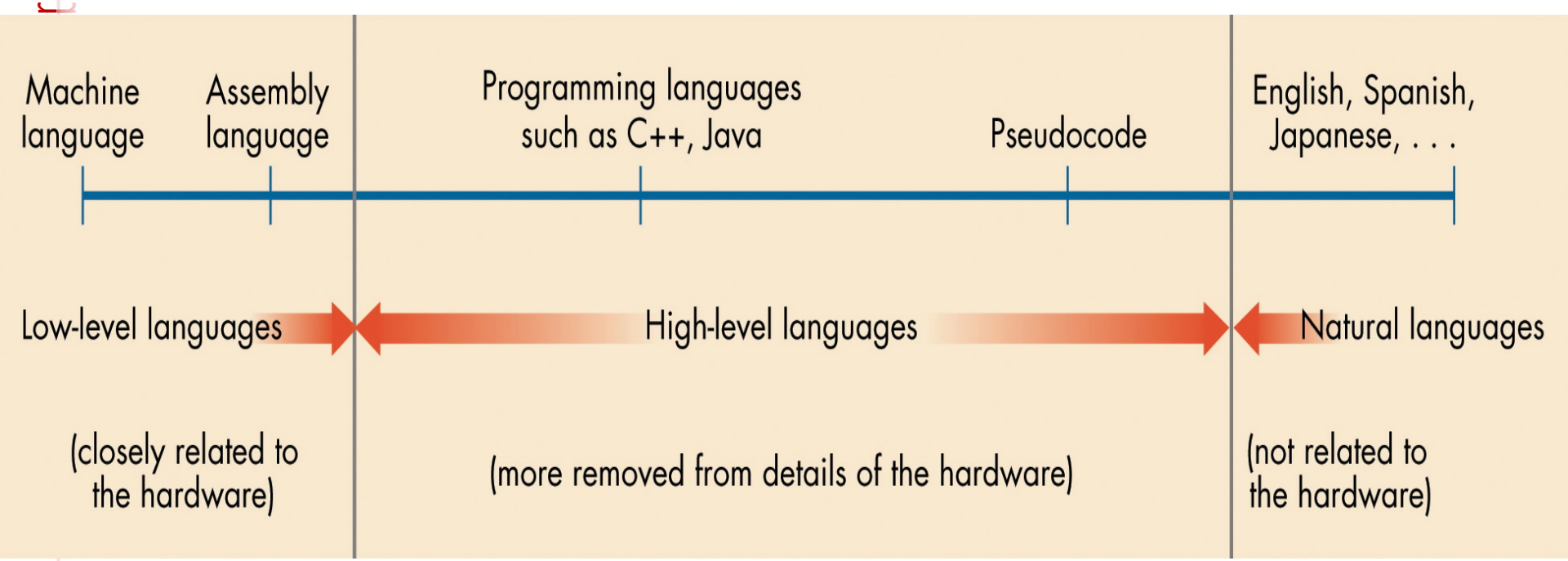
```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000  00000000 00000000
```

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000  00000000 00000000
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    alice(%rip), %edx
        movl    bob(%rip), %eax
        imull   %edx, %eax
        movl    %eax, carol(%rip)
        movl    $0, %eax
        leave
        ret
alice:
        .long   123
bob:
        .long   456
```

```
int alice = 123;
int bob = 456;
int carol;
main(void)
{
  carol = alice*bob;
}
```

The Spectrum of Programming Languages



- There is a limit to how high a language can get.
- Why can't we write programs in our spoken language?

Programming Language Paradigms

- Classification / Categorization of programming languages.
  - Imperative Paradigm
  - Functional Paradigm
  - Logical-declarative Paradigm
  - Object-oriented Paradigm
  - Concurrent Paradigm
  - Event-driven Paradigm

Imperative Paradigm

Statement_1

Statement_2

Statement_3

Statement_4

Statement_5

From C:

```
int a = 2;
int b = a * 2;
int c;

c = -b – sqrt(b*b - 4*a*c) / (2*a);
```

Functional Paradigm

- Data environment is restricted.
- Functions receive their inputs and return their results to the data environment.
- Programmer's task:
  - decompose the problem into a set of functions such that the composition of these functions produce the desired result.

Functional Paradigm

Imperative Version of Fibonacci Numbers

```
# Fibonacci numbers, imperative style
N=10

first = 0     # seed value fibonacci(0)
second = 1    # seed value fibonacci(1)
fib_number = first + second      # calculate fibon
for position in range(N-2):      # iterate N-2 tim
    first = second               # update the valu
    second = fib_number
    fib_number = first + second # update the resu
print fib_number
```

Functional Version of Fibonacci Numbers

```
# Fibonacci numbers, functional style
def fibonacci(N):   # Fibonacci number N (for N >=
    if N <= 1: return N     # base cases
    else: return fibonacci(N-1) + fibonacci(N-2)

print fibonacci(10)
```

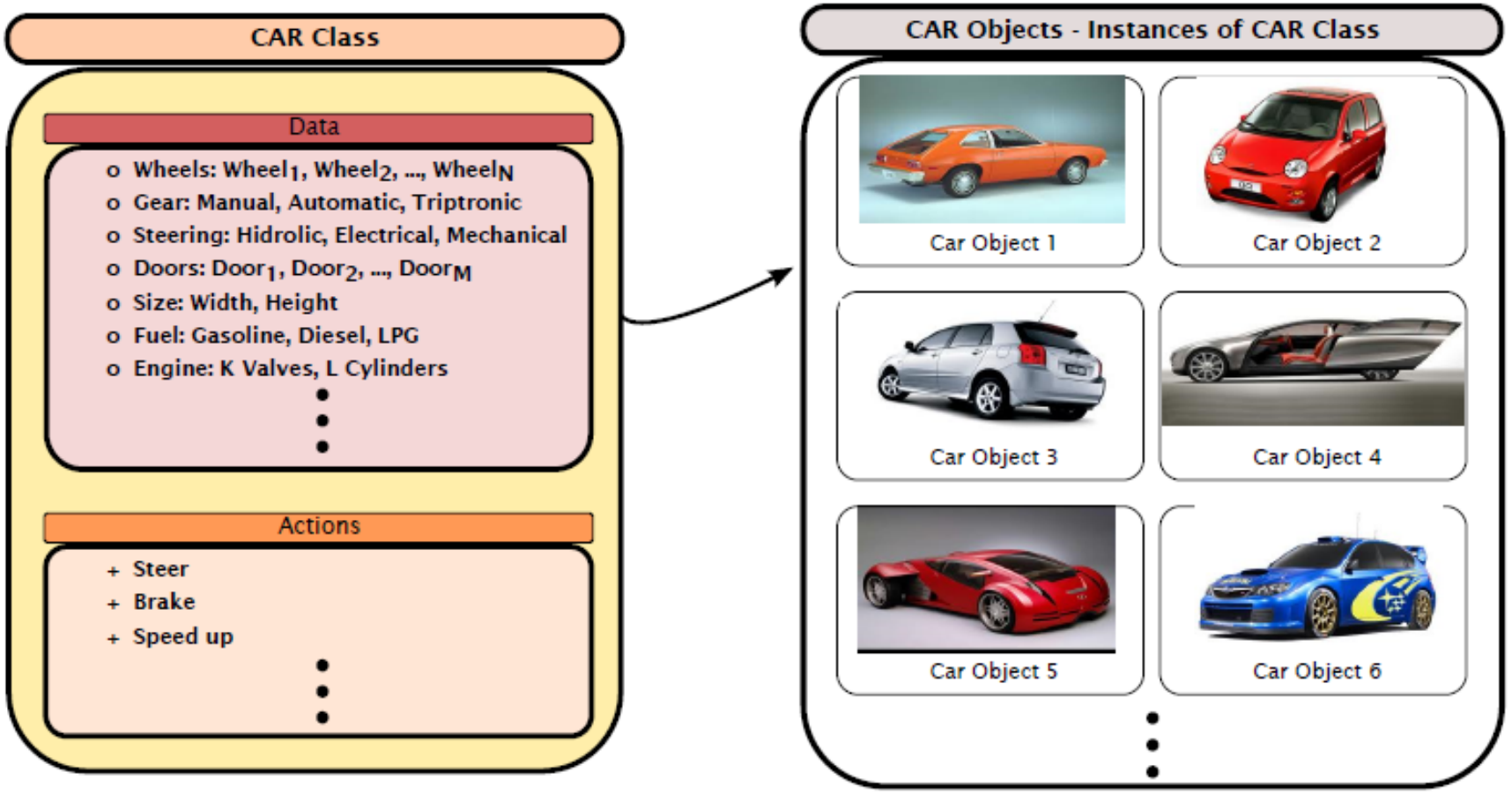
Logical-declarative Paradigm

- The data and the relations are states as rules, or facts.
- The problem is solved by writing new rules/facts.

From Prolog:

```
mother(matilda,ruth).
mother(trudi,paggy).
mother(eve,alice).
mother(zoe,sue).
mother(eve,trudi).
mother(matilda,eve).
mother(eve,carol).
grandma(X,Y) :- mother(X,Z), mother(Z,Y).
```

Object Oriented Paradigm



Object Oriented Paradigm

- Problem is decomposed into objects which hold data and the corresponding functions on the data.
- Objects can be defined using other objects as a basis; the new object inherits from the basis objects.

From C++:

```
class Item
{
string Name;
float Price;
string Location;
...
};

class Book : Item
{
string Author;
string Publisher;
...
 };

 class MusicCD : Item
 {
 string Artist;
 string Distributor;
 ...
 };
```

Concurrent Paradigm

- Programming using multiple CPUs concurrently.
- The task is to assign the overall flow & data to individual CPUs.
- With the bottleneck in CPU power, this paradigm is going to be the trend in the future.

Event-Driven Paradigm

- A program is composed of events and what to do in case of events.
- The task is to decompose a problem into a set of events and the corresponding functionalities that will be executed in case of events.
- Suitable for Graphical User Interface design.

The hyperspace of languages

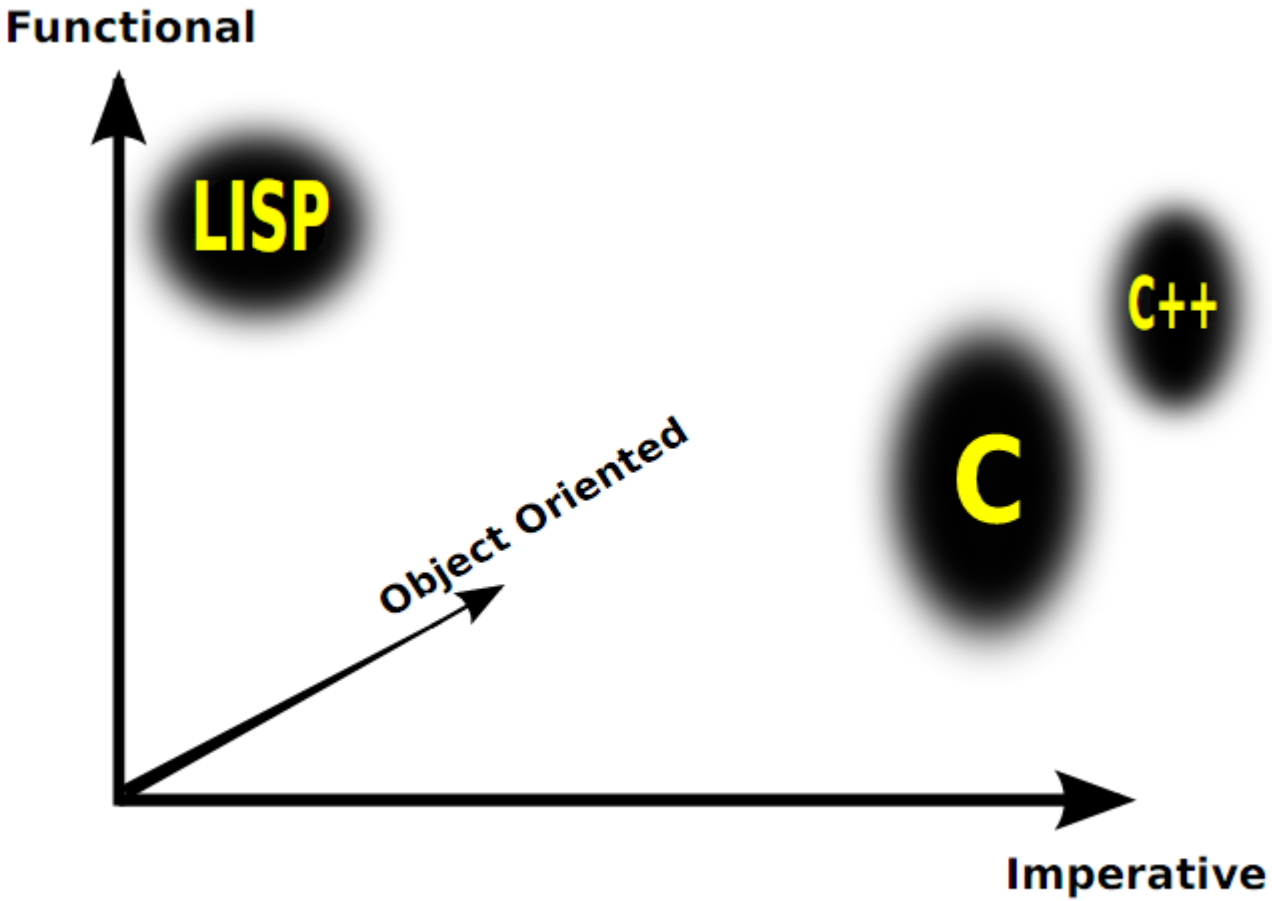


Figure 1.4: The hyperspace of programming *(only 3 axes displayed)*

Zoo of Programming Languages

- Around 700 programming languages!

- But, why do we not have a programming language that serves all paradigms/purposes/requirements?

Choosing a PL

- Ex: Moving soil with a shovel and a grader

Factors that affect choosing a PL

- Domain & Technical Nature of the Problem

a) Finding the pixels of an image with RGB value of [202,130,180] with a tolerance of 5.4% in intensity.

b) A proof system for planar geometry problems.

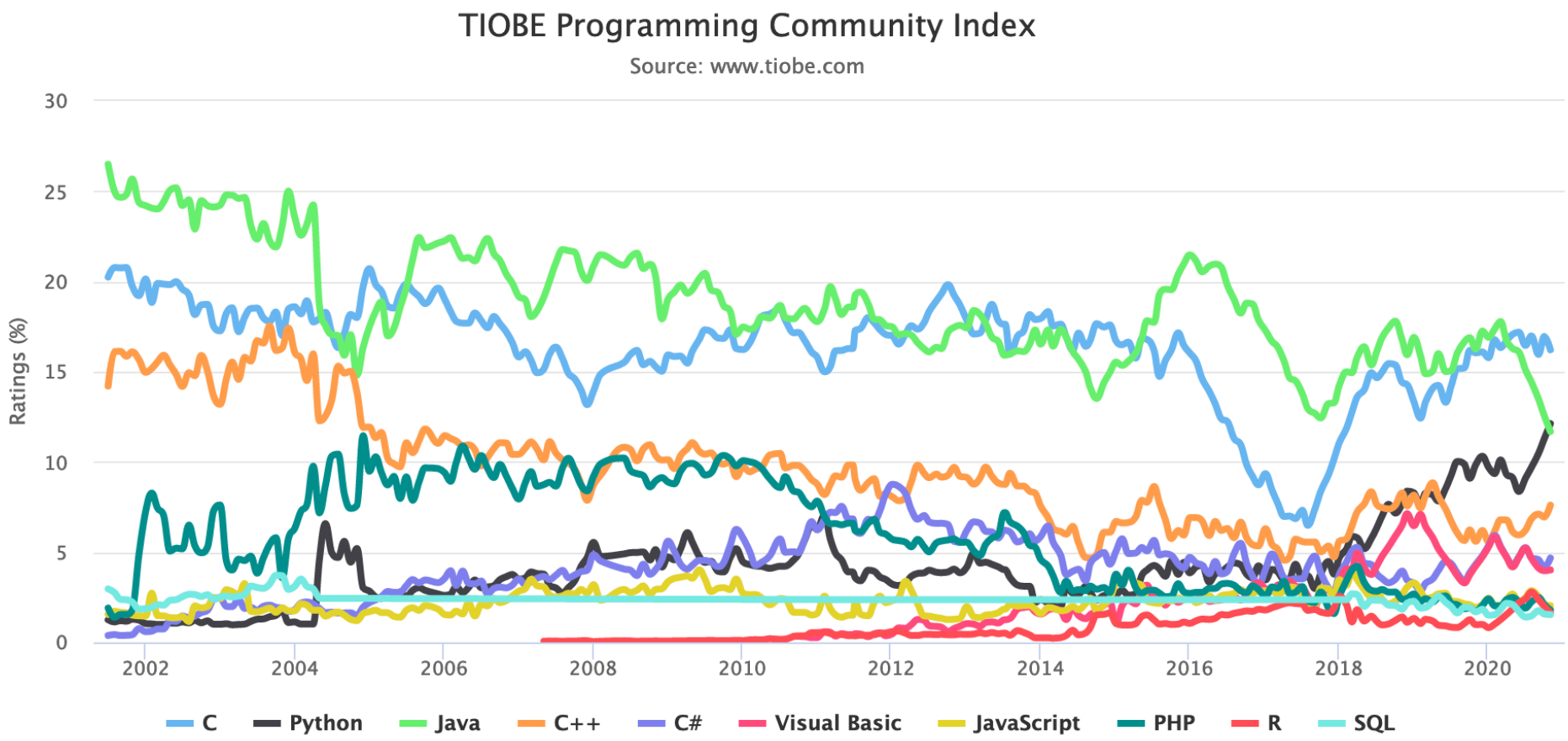
c) A computer game platform which will be used as a whole or in parts and may get extended even rewritten by programmers at various levels.

d) Payroll printing.

Factors that affect choosing a PL

- Personal taste and preference
- Circumstance-imposed constraints
  - e.g., time limit.
- Current trend

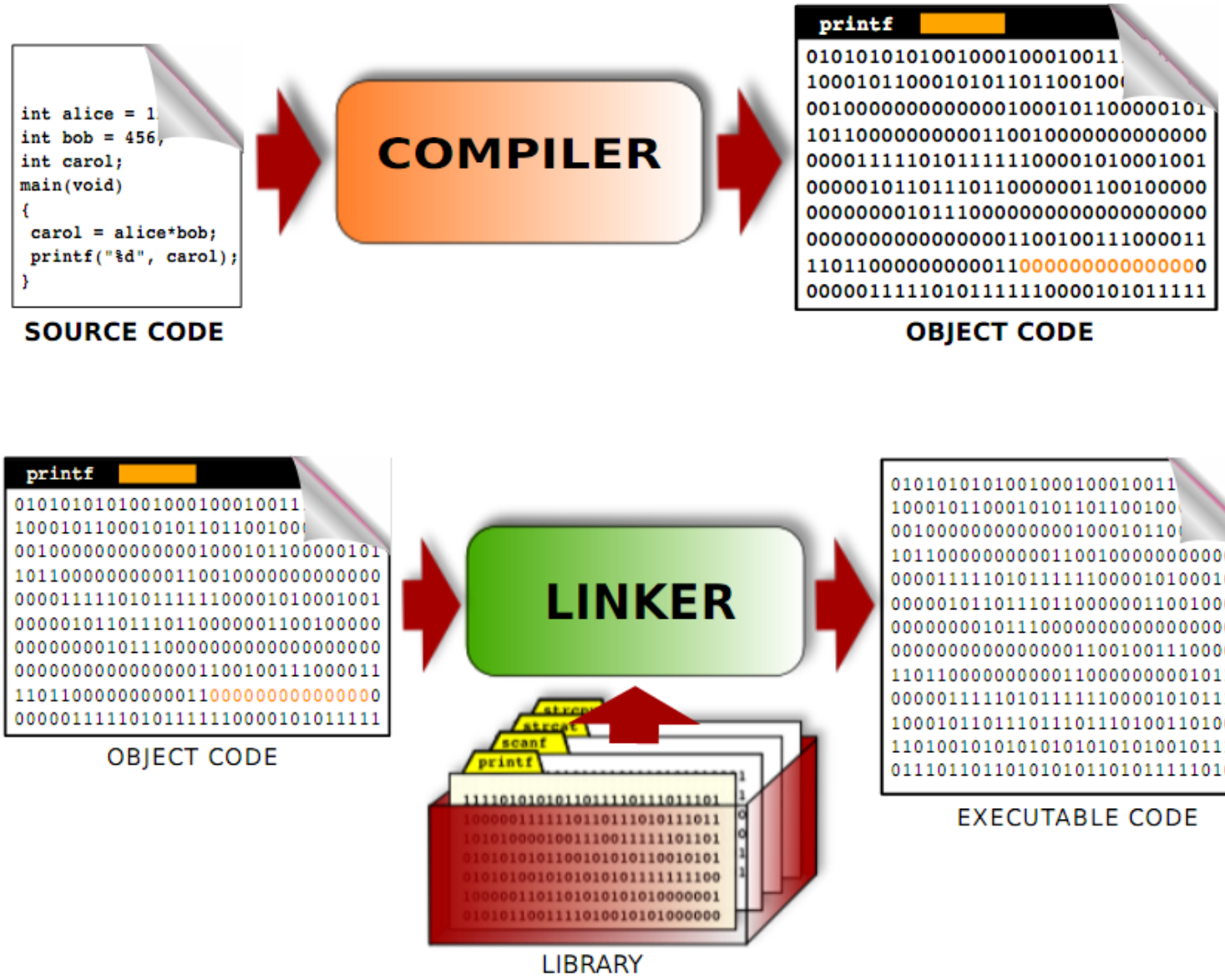
Current trend



http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

How are languages implemented

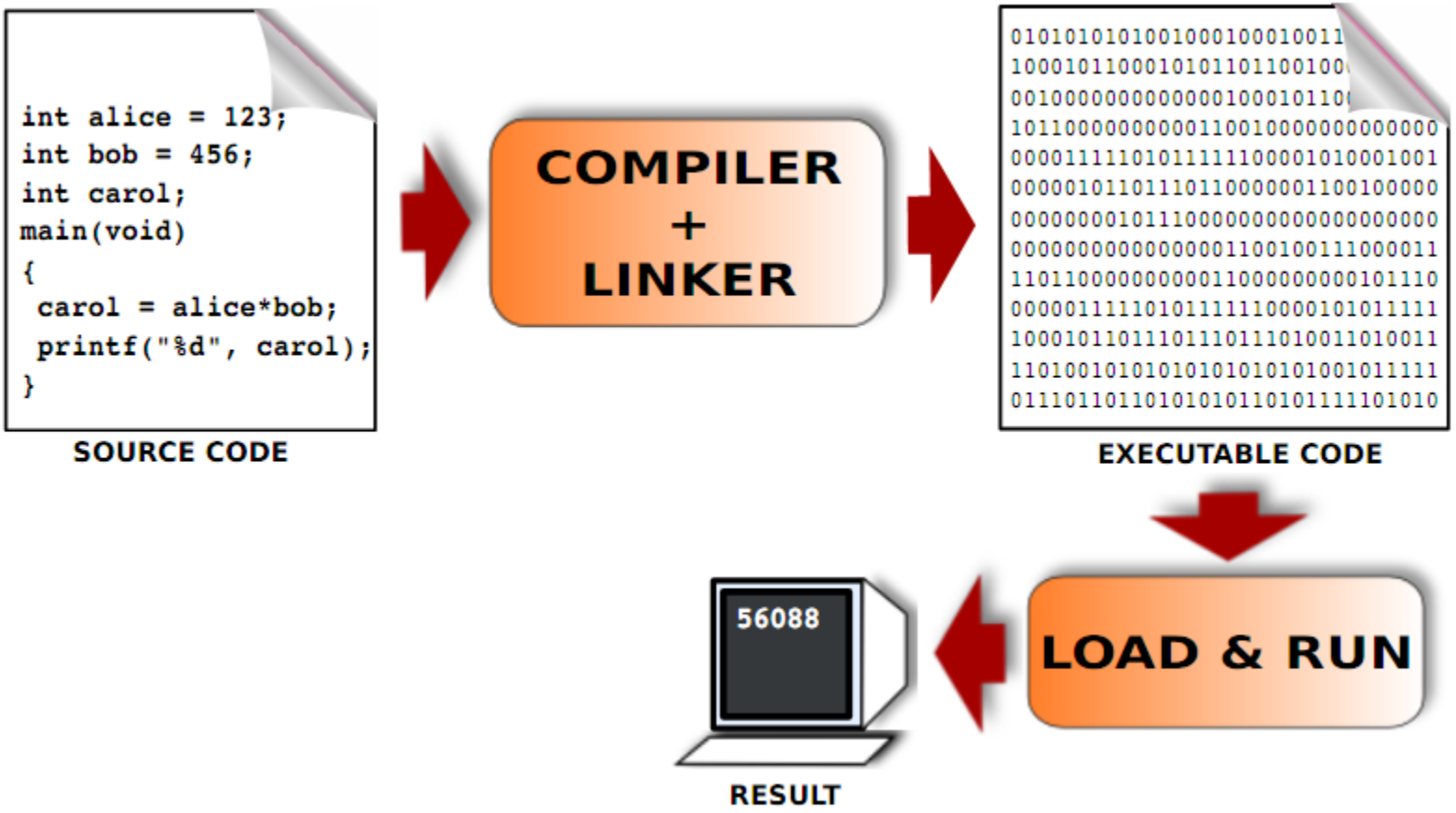
COMPILATIVE APPROACH



How are languages implemented

METU Computer Engineering

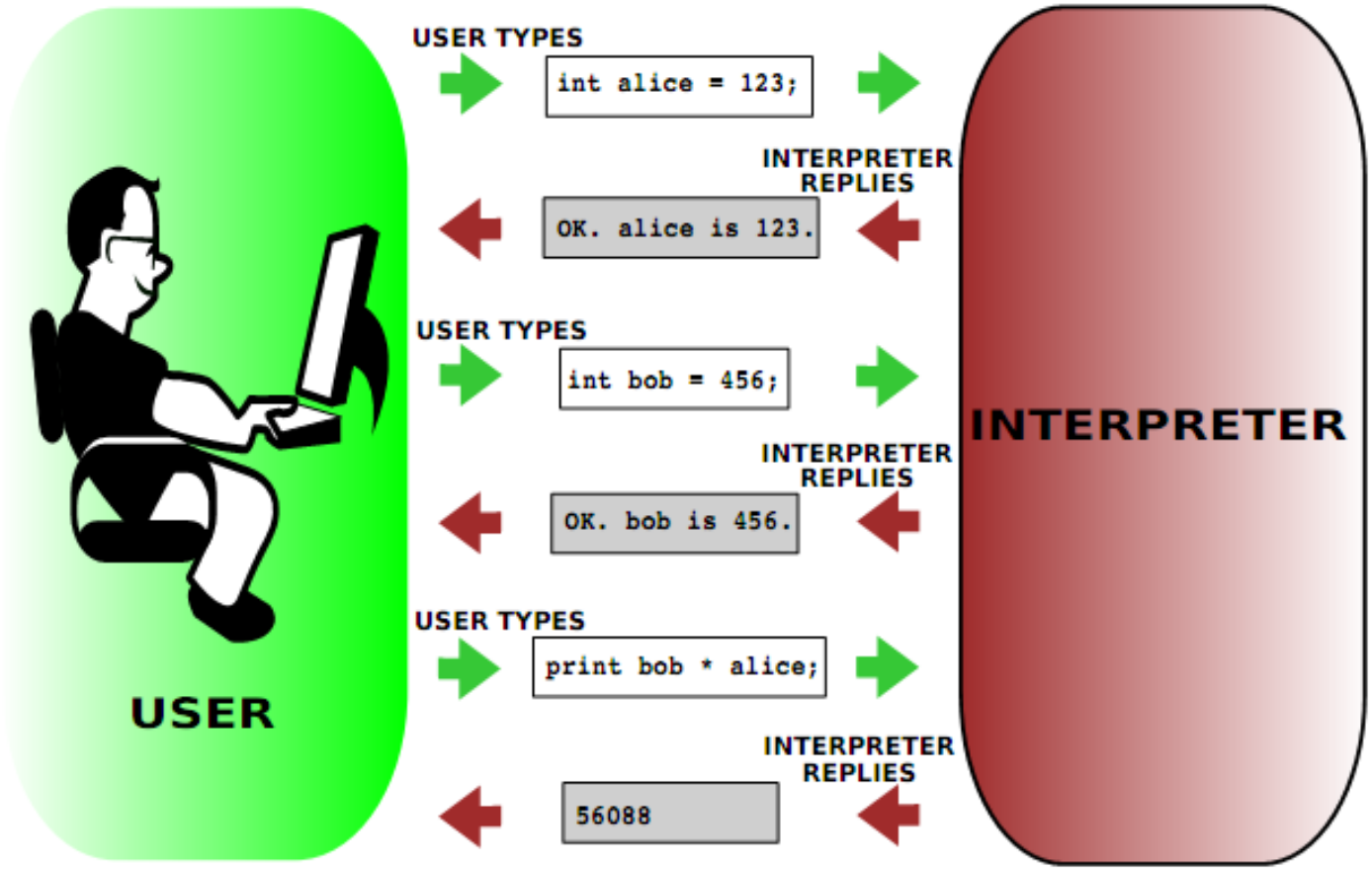
COMPILATIVE APPROACH



How are languages implemented

METU Computer Engineering

INTERPRETIVE APPROACH

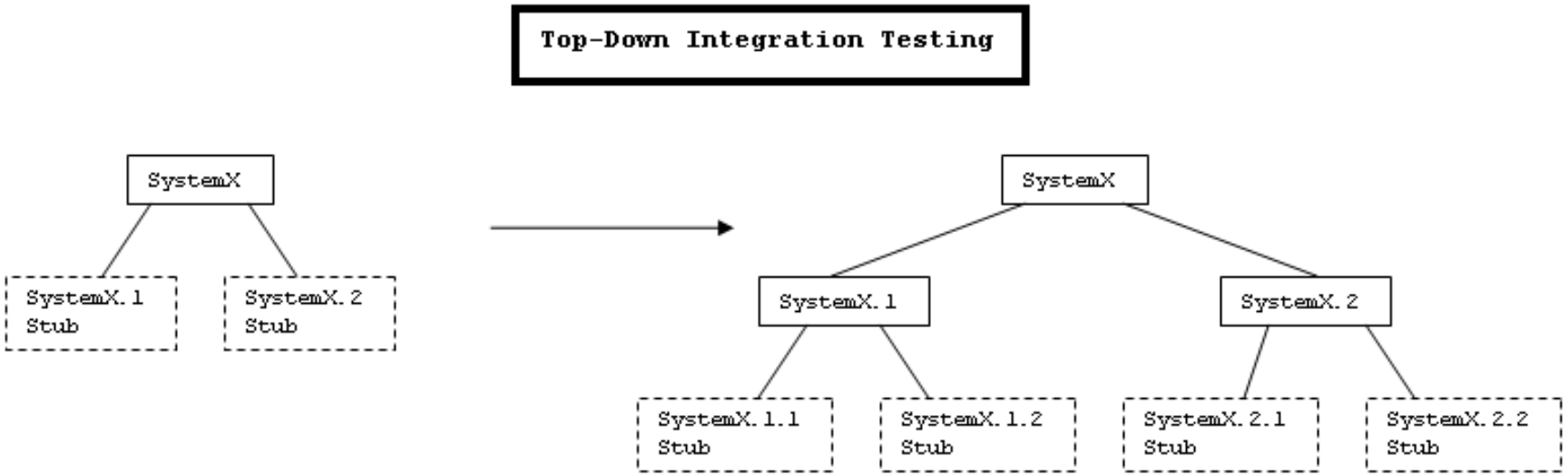


How is a program written?

- Modular & Functional Breakdown
- For example:
  - User interface module
  - Database module
  - Control module

Testing

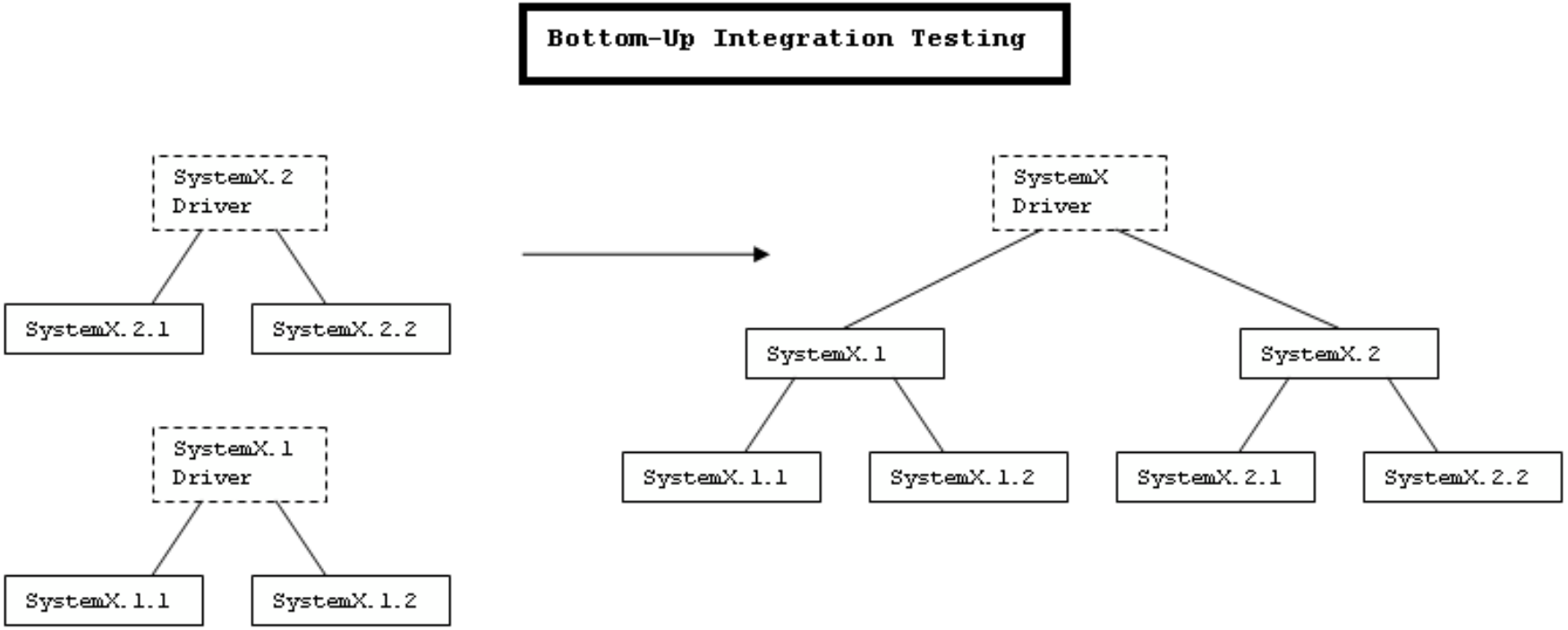
- Top-down Testing



http://sce.uhcl.edu/whiteta/sdp/subSystemIntegrationTesting.html

Testing

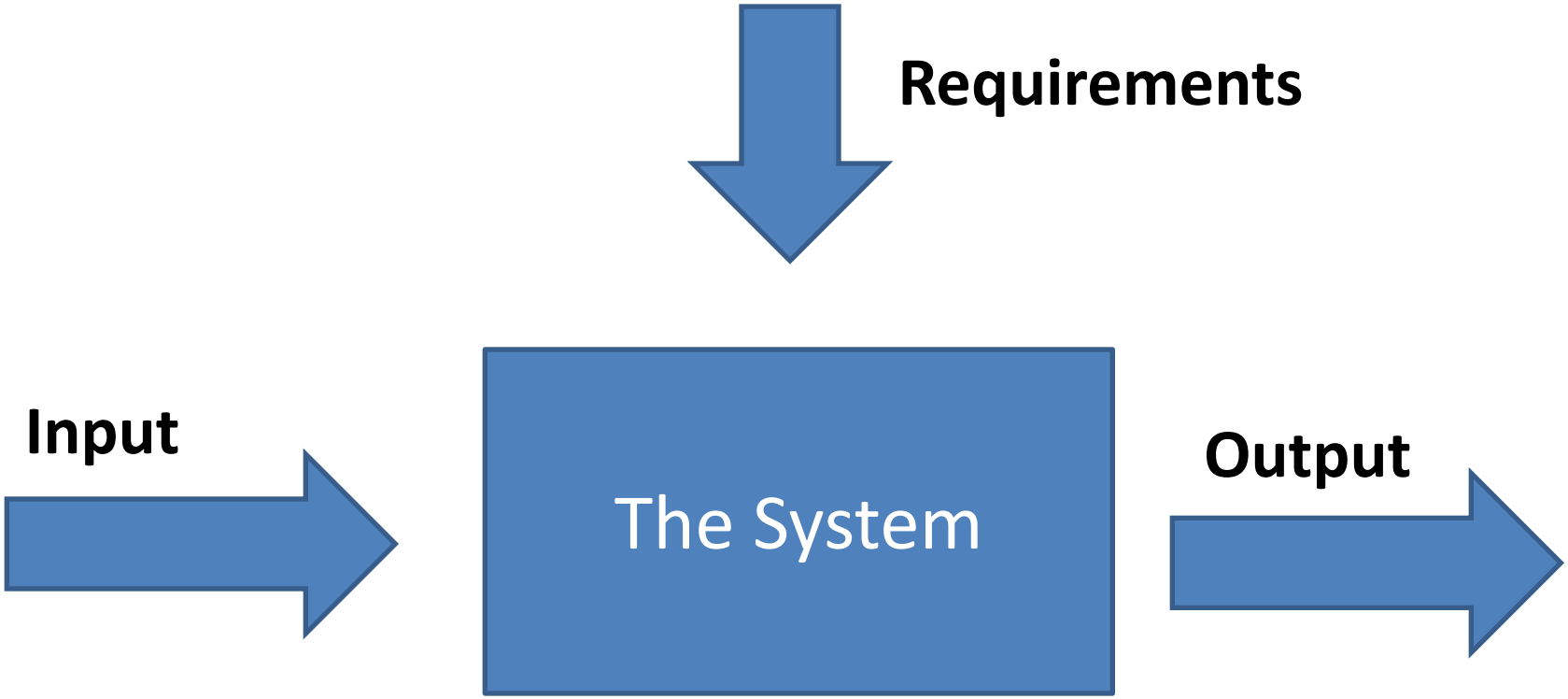
- Bottom-up Testing



http://sce.uhcl.edu/whiteta/sdp/subSystemIntegrationTesting.html

Testing

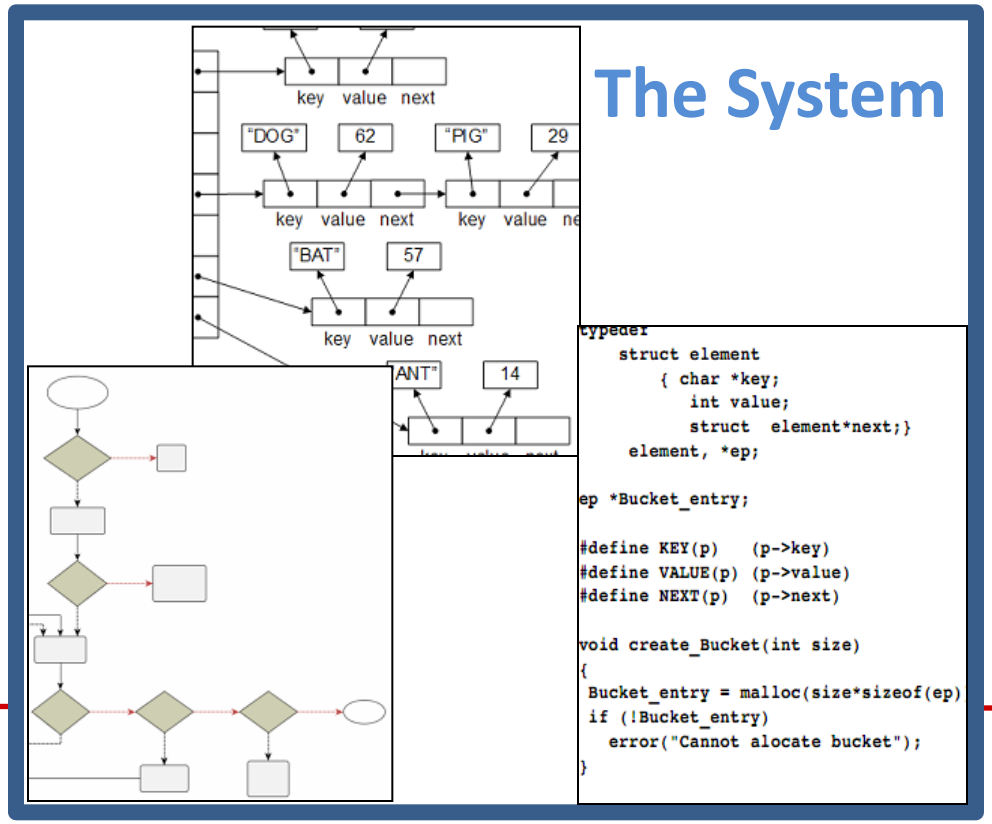
- Black-box Testing



Testing

- White-box Testing

Requirements



Input

Output

Bugs, Errors

- Syntax Errors

  Area = 3.1415 * R * R

  Area = 3.1415 x R x R

- Run-time Errors

```
>>> def SqrtDelta(a,b,c):
>>>       return sqrt(b*b - 4*a*c)
>>>
>>> print SqrtDelta(1,3,1)
2.236067977497898
>>> print SqrtDelta(1,1,1)
ValueError: math domain error
```

Bugs, Errors

- Logical Errors

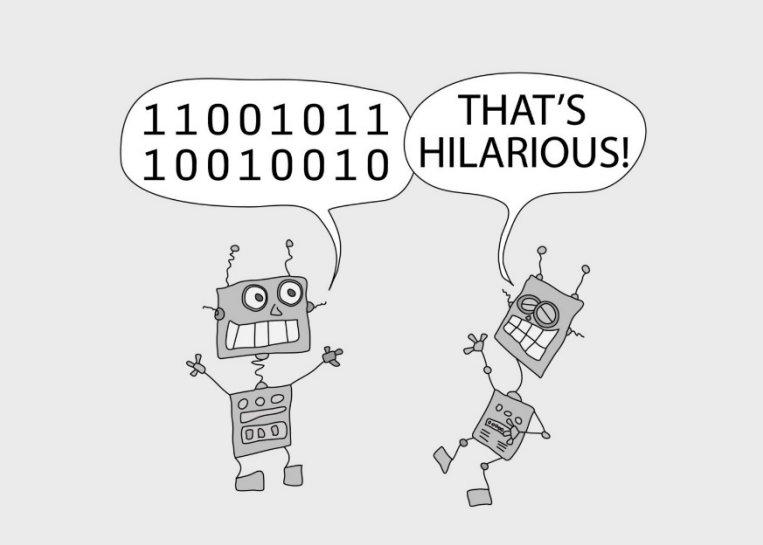
$$root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

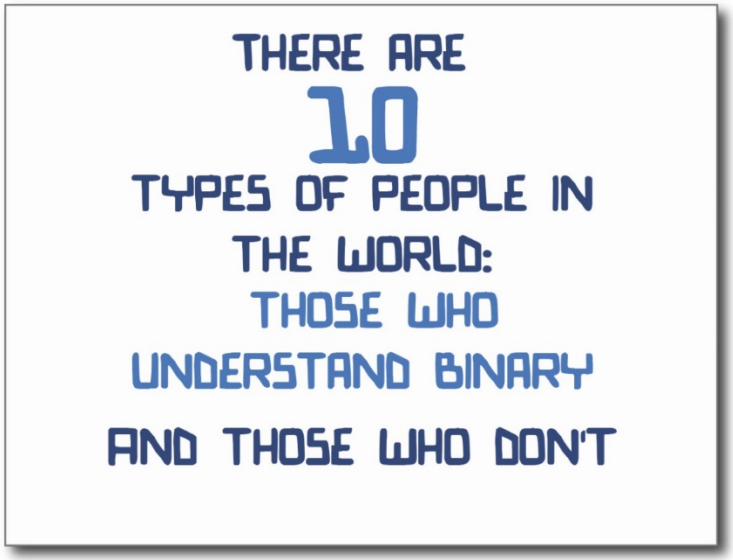
```
>>> root1 = (- b + sqrt(b*b - 4*a*c)) / 2*a
```

- Design Errors

$$x^3 + ax^2 + bx + c = 0$$

$$root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$





BINARY REPRESENTATION

Data Representation

- Based on 1s and 0s
  - So, everything is represented as a set of binary numbers
- We will now see how we can represent:
  - Integers: 3, 1234435, -12945 etc.
  - Floating point numbers: 4.5, 124.3458, -1334.234 etc.
  - Characters: /, &, +, -, A, a, ^, 1, etc.
  - …

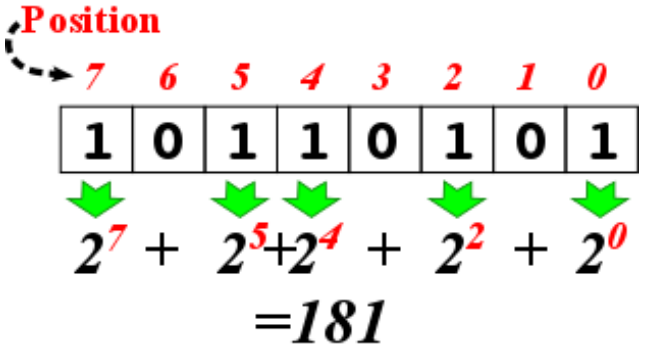
Binary Representation of Numeric Information

- Decimal numbering system
  - Base-10
  - Each position is a power of 10

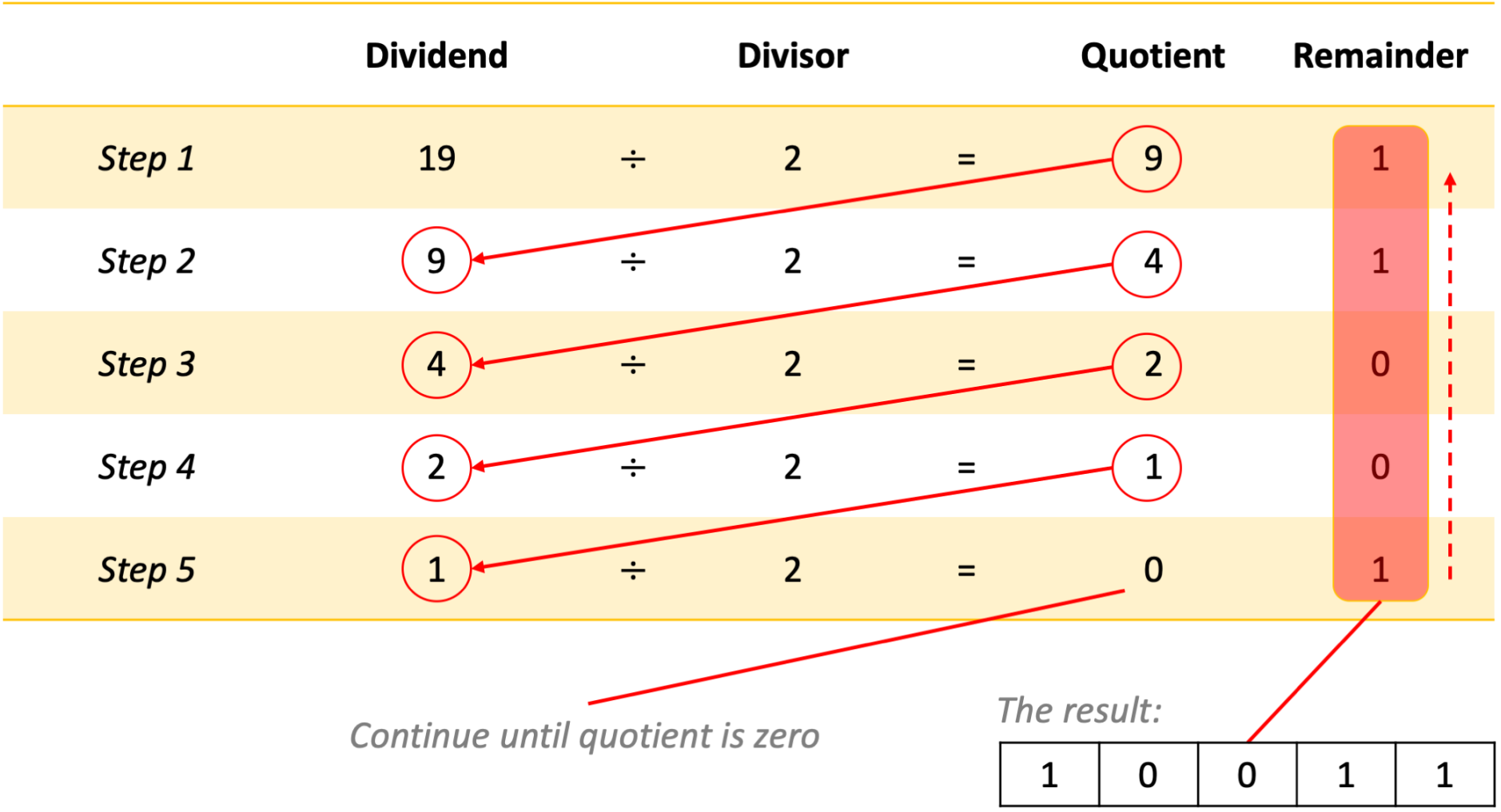
    $3052 = 3 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$

- Binary numbering system
  - Base-2
  - Uses ones and zeros
  - Each position is a power of 2

    $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$



Decimal-to-binary Conversion



Binary Representation of Numeric Information (continued)

- Representing integers
  - Decimal integers are converted to binary integers
  - Question: given k bits, what is the value of the largest integer that can be represented?
  - $2^k - 1$
    - Ex: given 4 bits, the largest is $2^4-1 = 15$
- Signed integers must also represent the sign (positive or negative) - ***Sign/Magnitude notation***

Binary Representation of Numeric Information (continued)

- Sign/magnitude notation

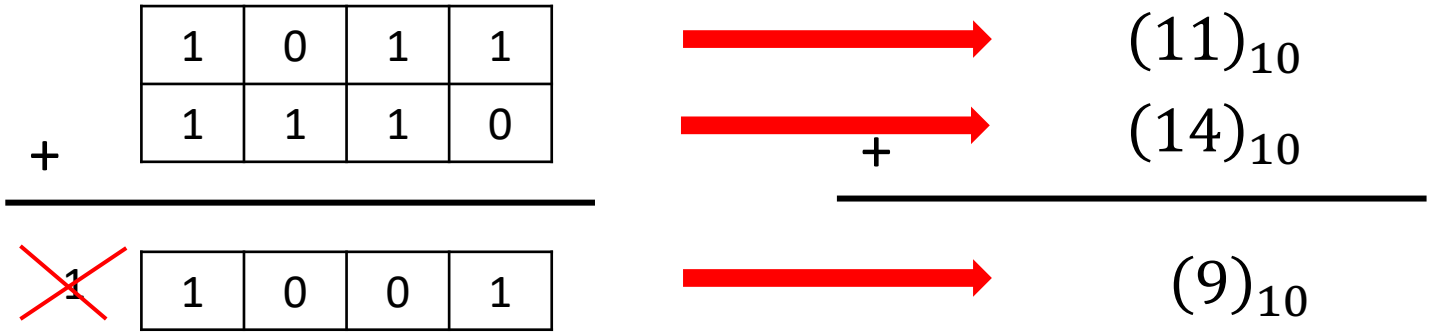
  1 101 = - 5

  0 101 = + 5

- Problems:
  - Two different representations for 0:
    - 1 000 = -0
    - 0 000 = +0
  - Addition & subtraction require a watch for the sign! Otherwise, you get wrong results:
    - 0 010 (+2) + 1 010 (-2) = 1 100 (-4)

Arithmetic in Computers is Modular

Let's add two numbers in binary
(Assume that there is no sign bit)



In other words:

- Numbers larger than or equal to 16 (2^4) are discarded in a 4-bit representation.
- Therefore, $11 + 14$ yields 9 in this 4-bit representation.
- This is actually modular arithmetic:

$$11 + 14 \; mod \; 16 \equiv 9 \; mod \; 16$$

Binary Representation of Numeric Information (continued)

- Two's complement instead of sign-magnitude representation
  - Positive numbers have a leading 0.
    - 5 => 0101
  - The representation for negative numbers is found by subtracting the absolute value from 2^N for an N-bit system:
    - -5 => $2^4 - 5 = 16 - 5 = (11)_{10} => (1011)_2$
- Advantages:
  - 0 has a single representation: +0 = 0000, -0 = 0000
  - Arithmetic works fine without checking the sign bit:
    - 1011 (-5) + 0110 (6) = 0001 (1)
    - 1011 (-5) + 0011 (3) = 1110 (-2)

Binary Representation of Numeric Information (continued)

- Shortcut to convert from "two's complement" :
  - If the leading bit is zero, no need to convert.
  - If the leading bit is one, invert the number and add 1.
- What is our range?
  - With 2's complement we can represent numbers from -2^{N-1} to $2^{N-1} - 1$ using N bits.
  - 8 bits: -128 to +127.
  - 16 bits: -32,768 to +32,767.
  - 32 bits: -2,147,483,648 to +2,147,483,647.

Binary Number	Decimal Value	Value in Two's Complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Binary Representation of Numeric Information (continued)

- Example:
  - We want to compute: 12 – 6
  - 12 => 01100
  - -6 => -(00110) => (11001)+1 => (11010)
- 12 – 6 =

```
  01100
+ 11010
-----------
  00110 => 6
```

So, addition and subtraction operations are simpler in the Two's Complement representation

Binary Representation of Numeric Information (continued)

- Due to its advantages, two's complement is the most common way to represent integers on computers.

Why does Two's Complement work?

- 1st Perspective:
  - We divide the range of unsigned numbers using N bits into two halves:
  - 0 to $2^{N-1} - 1$ => Positive numbers
  - 2^{N-1} to $2^N - 1$ => Negative numbers (-2^{N-1} to -1)

Binary Number	Decimal Value	Value in Two's Complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Why does Two's Complement work?

- A 2nd perspective:
  - Inversion and addition of a 1-bit correspond effectively to subtraction from 0 – i.e., negative of a number.
  - Negative of a binary number X: $(00\ldots00)_2 - (X)_2$
  - Note that $(00\ldots00)_2 = (11\ldots11)_2 + (1)_2$
  - In other words:
    - $(00\ldots00)_2 - (X)_2 = (11\ldots11)_2 - (X)_2 + (1)_2$.

(i.e., how we find two's complement)

Inversion

Why does Two's Complement work?

- A 2nd perspective:
  - i - j mod 2^N = i + (2^N-j) mod 2^N
  - Example:
    - Consider X and Y are positive numbers.
    - $X + (-Y) = X + (2^N - Y)$
      $= 2^N - (Y - X) = -(Y - X) = X - Y$

Why does Two's Complement work?

- A smart trick used in mechanical calculators
  - To subtract b from a, invert b and add that to a. Then discard the most significant digit.

http://en.wikipedia.org/wiki/Method_of_complements

Binary Representation of Real Numbers

Conversion of the digits after the dot into binary:

- 1st Way:
  - 0.375 → 0x½ + 1x¼ + 1x1/8 → 011
- 2nd Way:

	Fraction		Multiplier		Whole		Fraction
Step 1	0.375	×	2	=	0	.	75
Step 2	0.75	×	2	=	1	.	5
Step 3	0.5	×	2	=	1	.	0

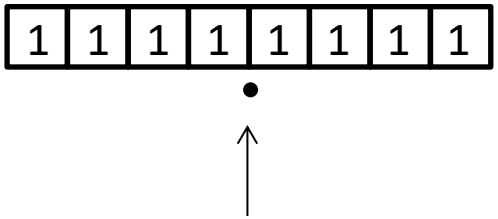
The result:

.	0	1	1

Continue until fraction is zero

Binary Representation of Real Numbers

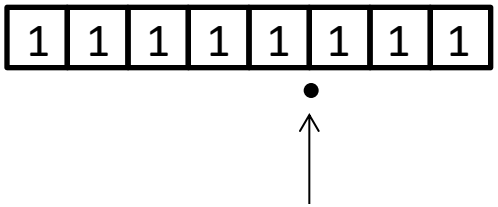
- Approach 1: Use fixed-point
  - Similar to integers, except that there is a decimal point.
  - E.g.: using 8 bits:



$= 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} + 1\times2^{-4}$
$= 15.9375$

Binary Representation of Real Numbers

- Location of the decimal point changes the value of the number.
  - E.g.: using 8 bits:



$= 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$
$1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 31.875$

Assumed decimal point

Binary Representation of Real Numbers

- Problems with fixed-point:
  - Limited in the maximum and minimum values that can be represented.
  - For instance, using 32-bits, reserving 1-bit for the sign and putting the decimal point after 16 bits from the right, the maximum positive value that can be stored is slightly less than 2^{15}.
  - Allowing larger values gives away from the precision (the decimal part).

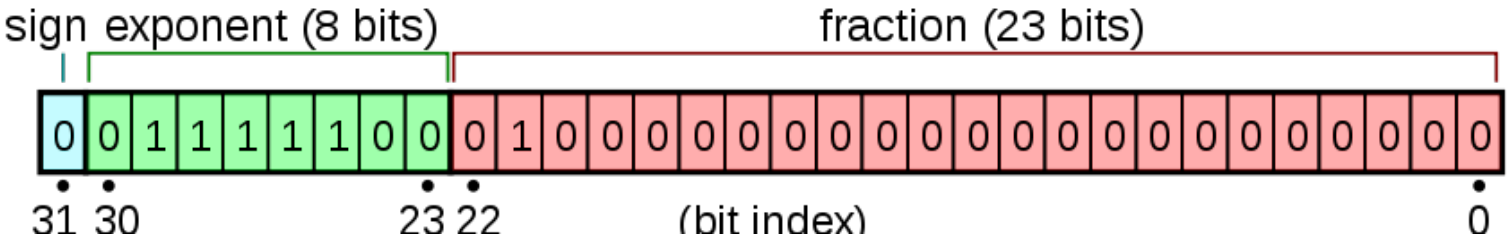
Binary Representation of Real Numbers

- Solution: Use scientific notation: a x 2^b *(or* $\pm M \times B^{\pm E}$*)*
  - Example: 5.75
    - 5 → 101
    - 0.75 → ½ + ¼ → $2^{-1}+2^{-2}$ → $(0.11)_2$
    - 5.75 → $(101.11)_2$ x 2^0
- Number is then normalized so that the first significant digit is immediately to the left of the binary point
  - Example: 1.0111×2^2
- We take and store the mantissa and the exponent.

Binary Representation of Real Numbers

- This needs some standardization for:
  - where to put the decimal point
  - how to represent negative numbers
  - how to represent numbers less than 1

IEEE 32bit Floating-Point Number Representation



= 0.15625

$$= (-1)^{\text{sign}}(1.b_{-1}b_{-2}...b_{-23})_2 \times 2^{e-127}$$

- M x 2^E $(2-2^{-23})\times 2^{127}$
- Exponent (E): 8 bits
  - Add 127 to the exponent value before storing it
  - **E can be 0 to 255 with 127 representing the real zero.**
- Fraction (M - Mantissa): 23 bits
- $2^{128} = 1.70141183 \times 10^{38}$

IEEE 32bit Floating-Point Number Representation

- Example: 12.375
- The digits before the dot:
  - $(12)_{10} \rightarrow (1100)_2$
- The digits after the dot:
  - 1st Way: 0.375 → 0x½ + 1x¼ + 1x1/8 → 011
  - 2nd Way: Multiply by 2 and get the integer part until 0:
    - 0.375 x 2 = 0.750 = 0 + 0.750
    - 0.750 x 2 = 1.50 = 1 + 0.50
    - 0.50 x 2 = 1.0 = 1 + 0.0
- $(12.375)_{10} = (1100.011)_2$
- NORMALIZE (move the point): $(1100.011)_2 = (1.100011)_2 x 2^3$
- Exponent: 3, adding 127 to it, we get 1000 0010
- Fraction: 100011
- Then our number is: 0 10000010 10001100000000000000000

Why add bias to the exponent?

- It helps in comparing the exponents of the same-sign real-numbers without looking out for the sign of the exponent.

Binary Number	Decimal Value	Value in Two's Complement	Value with bias 7
0000	0	0	-7
0001	1	1	-6
0010	2	2	-5
0011	3	3	-4
0100	4	4	-3
0101	5	5	-2
0110	6	6	-1
0111	7	7	0
1000	8	-8	1
1001	9	-7	2
1010	10	-6	3
1011	11	-5	4
1100	12	-4	5
1101	13	-3	6
1110	14	-2	7
1111	15	-1	8

To read more on this:
https://blog.angularindepth.com/the-mechanics-behind-exponent-bias-in-floating-point-9b3185083528

IEEE 32bit Floating-Point Number Representation

- Zero:
  - Exponent: All zeros
  - Fraction: All zeros
  - +0 and -0 are different numbers but they are equal!
- Infinity:
  - Exponent: All ones
  - Fraction: All zeros
- Not a number (NaN):
  - Exponent: All ones
  - Fraction: non-zero fraction.

http://steve.hollasch.net/cgindex/coding/ieeefloat.html

IEEE 32bit Floating-Point Number Representation

- What is the maximum positive IEEE floating point value that can be stored?
  - Just less than 2^{128} $[(2-2^{-23})\times 2^{127}$ to be specific]
  - Why? 2^{128} is reserved for NaN.

- Check out these useful links:
  - http://steve.hollasch.net/cgindex/coding/ieeefloat.html
  - http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

IEEE 32bit Floating-Point Number Representation

- Now consider 4.1:
  - 4 => $(100)_2$
  - 0.1 =>
    - x 2 = 0.2 = 0 + 0.2
    - x 2 = 0.4 = 0 + 0.4
    - x 2 = 0.8 = 0 + 0.8
    - x 2 = 1.6 = 1 + 0.6
    - x 2 = 1.2 = 1 + 0.2
    - x 2 = 0.4 = 0 + 0.4
    - x 2 = 0.8 = 0 + 0.8
    - …….

- So,
  - Representing a fraction which is a multiple of $1/2^n$ is lossless.
  - Representing a fraction which is not a multiple of $1/2^n$ leads to accuracy loss.

BINARY REPRESENTATION OF TEXT ETC.

Binary Representation of Textual Information

- Characters are mapped onto binary numbers
  - ASCII (American Standard Code for Information Interchange) code set
    - Originally: 7 bits per character; 128 character codes
  - Unicode code set
    - 16 bits per character
  - UTF-8 (Universal Character Set Transformation Format) code set.
    - Variable number of 8-bits.

Binary Representation of Textual Information (cont'd)

ASCII
7 bits long

Decimal	Binary	Val.
48	00110000	0
49	00110001	1
50	00110010	2
51	00110011	3
52	00110100	4
53	00110101	5
54	00110110	6
55	00110111	7
56	00111000	8
57	00111001	9
58	00111010	:
59	00111011	;
60	00111100	<
61	00111101	=
62	00111110	>
63	00111111	?
64	01000000	@
65	01000001	A
66	01000010	B

Hex.	Unicode	Charac.
0x30	0x0030	0
0x31	0x0031	1
0x32	0x0032	2
0x33	0x0033	3
0x34	0x0034	4
0x35	0x0035	5
0x36	0x0036	6
0x37	0x0037	7
0x38	0x0038	8
0x39	0x0039	9
0x3A	0x003A	:
0x3B	0x003B	;
0x3C	0x003C	<
0x3D	0x003D	=
0x3E	0x003E	>
0x3F	0x003F	?
0x40	0x0040	@
0x41	0x0041	A
0x42	0x0042	B

Unicode
16 bits long

Partial listings only!

Your book is wrong about the number of characters in the ASCII table.

To store the 4-character string "BAD!" in memory, the computer would store the binary representation of each individual character using the above 8-bit code.

BAD! = 00000010 00000001 00000100 10000001
B A D !

We have indicated above that the 8-bit numeric quantity 10000001 is interpreted as the character "!". However, as we mentioned earlier, the only way a computer knows that the 8-bit value 10000001 represents the symbol "!" and not the unsigned integer value 129 (128 + 1) or the signed integer value –1 (sign bit = negative, magnitude is 1) is by the context in which it is used. If these 8 bits are sent to a display device that expects to be given characters, then this value will be interpreted as an "!". If, on the other hand, this 8-bit value is sent to an arithmetic unit that adds unsigned numbers, then it will be interpreted as a 129 in order to make the addition operation meaningful.

To facilitate the exchange of textual information, such as word processing documents and electronic mail, between computer systems, it would be most helpful if everyone used the same code mapping. Fortunately, this is pretty much the case. Currently the most widely used code for representing characters internally in a computer system is called **ASCII**, an acronym for the *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange. ASCII is an international standard for representing textual information in the majority of computers. It uses 8 bits to represent each character, so it is able to encode a total of $2^8 = 256$ different characters. These are assigned the integer values 0 to 255. However, only the numbers 32 to 126 have been assigned so far to printable characters. The remainder either are unassigned or are used for nonprinting control characters such as tab, form feed, and return. Figure 4.3 shows the ASCII conversion table for the numerical values 32–126.

However, a new code set called **UNICODE** is rapidly gaining popularity because it uses a 16-bit representation for characters rather than the 8-bit format of ASCII. This means that it is able to represent $2^{16} = 65{,}536$ unique characters instead of the $2^8 = 256$ of ASCII. It may seem like 256 characters are more than enough to represent all the textual symbols that we would ever need—for example, 26 uppercase letters, 26 lowercase letters, 10 digits, and a few dozen special symbols, such as +=–{}][\;'"?><.,;'%$#@. Add that all together and it still totals only about 100 symbols, far less than the 256 that can be represented in ASCII. However, that is true only if we limit our work to Arabic numerals and the Roman alphabet. The world grows more connected all the time—helped along by computers, networks, and the Web—and it is critically important that computers represent and exchange textual information using alphabets in addition to these 26 letters and 10 digits. When we start assigning codes to symbols drawn from alphabets such as Russian, Arabic, Chinese, Hebrew, Greek, Thai, Bengali, and Braille, as well as mathematical symbols and special linguistic marks such as tilde, umlaut, and accent grave, it becomes clear that ASCII does not have nearly enough room to represent them all. However, UNICODE, with space for over 65,000 symbols, is large enough to accommodate all these symbols and many more to come. In fact, UNICODE has defined standard code mappings for more

UTF-8 Illustrated

Bits	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+007F	0xxxxxxx					
11	U+07FF	110xxxxx	10xxxxxx				
16	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx			
21	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Character		Binary code	Binary UTF-8
$	U+0024	0100100	00100100
¢	U+00A2	00010100010	11000010 10100010
€	U+20AC	0010000010101100	11100010 10000010 10101100
𤭢	U+24B62	000100100101101100010	11110000 10100100 10101101 10100010

How about a text?

- Text in a computer has two alternative representations:

1. A fixed-length number representing the length of the text followed by the binary values of the characters in the text.
   - Ex: “ABC” =>

   00000011 01000001 01000001 01000001 (3 ‘A’ ‘B’ ‘C’)

2. Binary values of the characters in the text ended with a unique marker, like “00000000” which has no value in the ASCII table.
   - Ex: “ABC” =>

   01000001 01000001 01000001 00000000 (‘A’ ‘B’ ‘C’ END)

Binary Representation of Sound and Images

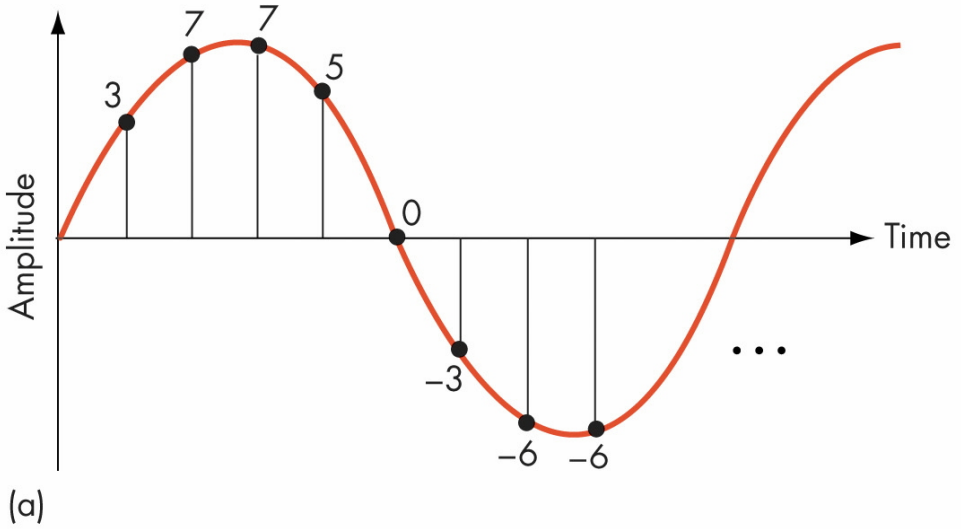
- Multimedia data is sampled to store a digital form, with or without detectable differences
- Representing sound data
  - Sound data must be digitized for storage in a computer
  - Digitizing means periodic sampling of amplitude values

Binary Representation of Sound and Images (continued)

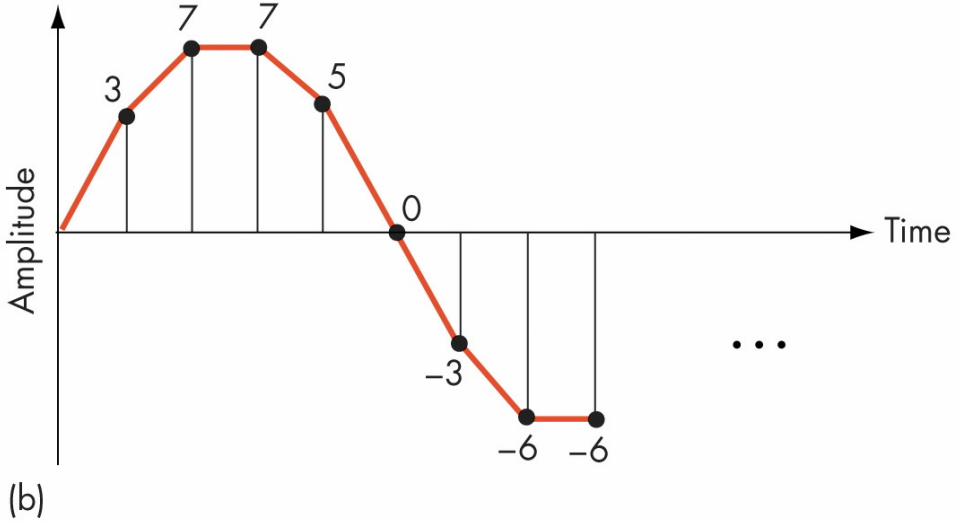
- From samples, original sound may be approximated
- To improve the approximation:
  - Sample more frequently (*increase sampling rate)*
  - Use more bits for each sample value *(↑ bit depth)*

Digitization of an Analog Signal

(a) Sampling the Original Signal

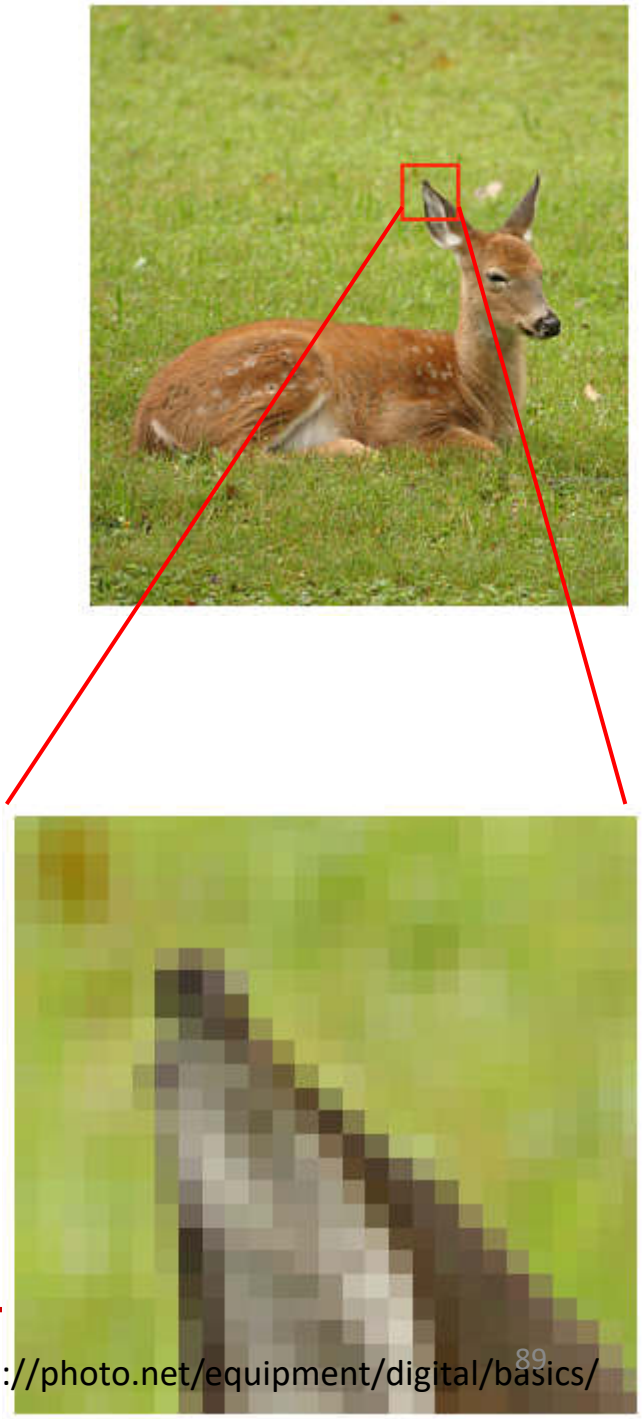


(b) Recreating the Signal from the Sampled Values



Binary Representation of Sound and Images (continued)

- Representing image data
  - Images are sampled by reading color and intensity values at even intervals across the image
  - Each sampled point is a pixel
  - Image quality depends on number of bits at each pixel
  - More image information: http://cat.xula.edu/tutorials/imaging/grayscale.php



http://photo.net/equipment/digital/basics/

The Reliability of Binary Representation

- Electronic devices are most reliable in a bistable environment
- Bistable environment
  - Distinguishing only two electronic states
    - Current flowing or not, or
    - Direction of flow
- Computers are bistable: hence binary representations

Further reading

https://pp4e-book.github.io/chapters/ch3_data_representation.html