Introduction to programming

Lecture

Imre Varga

University of Debrecen, Faculty of Informatics

For internal use only!

11 February 2017

General information

Teacher:

Dr. Varga, Imre
University of Debrecen
Faculty of Informatics
Department of Informatics Systems and Networks

email: varga.imre@inf.unideb.hu
www: irh.inf.unideb.hu/user/vargai
room: IF13 (building of Faculty of Informatics)

General information

Requirements, conditions for practice:

Maximum number of absences is 3.

Late arrival (more than 20 minutes) means absent from class

There will be two tests during the semester.

There is only one chance to retake!!!

Activity during class means plus score.

Readings:

Adrian Kingsley-Hughes: *Beginning Programming*, Wiley, 2005.

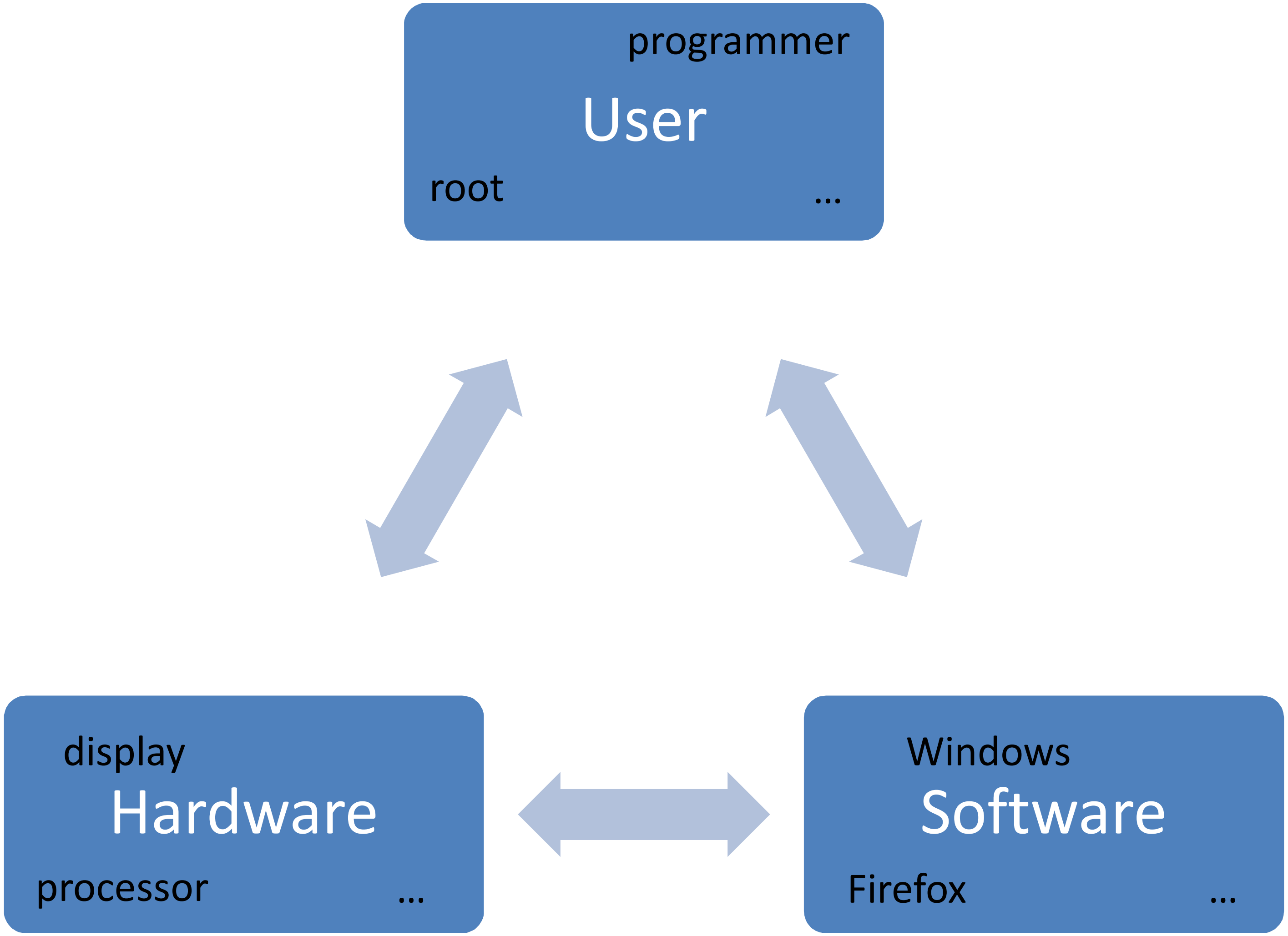
Metrowerks CodeWarrior: *Principles of Programming*

Topics

- What are the basics of Computer Science?
- How does a computer built up and work?
- What is software, application, program?
- How to describe problems and its solution?
- What is **algorithmic thinking**?
- How to describe algorithms?
- What does ‘program writing’ mean?
- *Many more things…*

Computer systems

Computer System



programmer

User

root

…

display

Hardware

processor

…

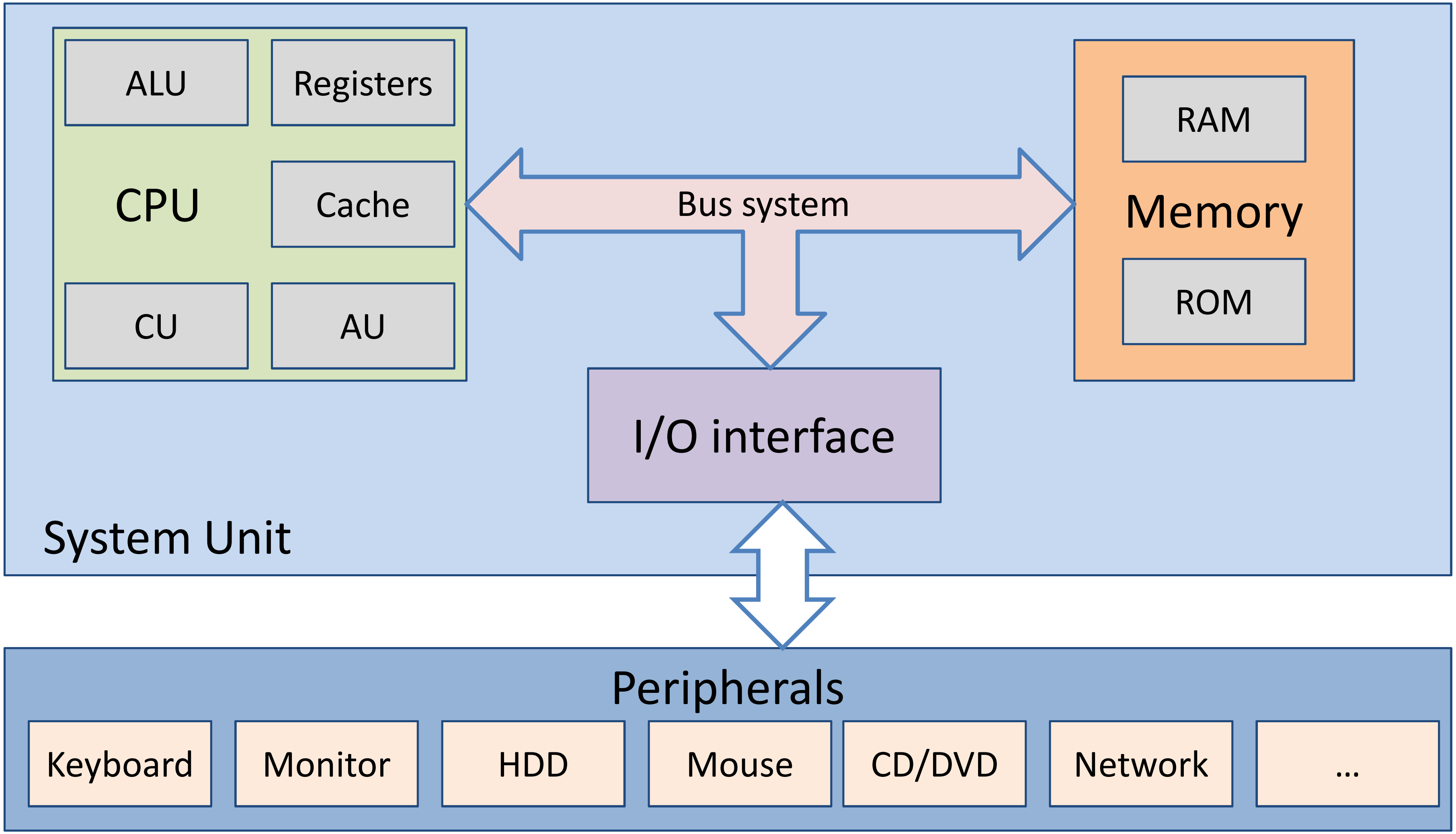
Windows

Software

Firefox

…

Computer architecture (hardware)



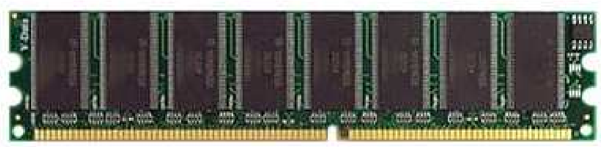
System Unit

Central Processing Unit (CPU):

The brain of computers

Memory:

Contains data and instructions



Input-Output Interface:

Surface between computer
and outer world

Bus system:

Connects together

Central Processing Unit

Control Unit (CU): Says what to do, controls the parts of the CPU

Arithmetic Logic Unit (ALU): Performs operations, does calculations

Registers: Some tinny but very fast memory

Cache: Small, but fast memory

Addressing Unit (AU): Deals with memory addresses at read/write operation

Memory

Random Access Memory (RAM):

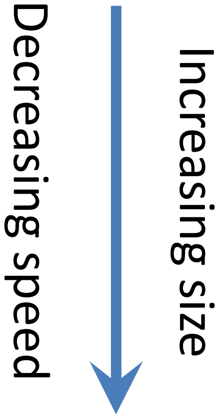
Readable-writeable operative memory

Read Only Memory (ROM)

Not rewritable (eg. BIOS-ROM)

Memory hierarchy:

- Register
- Cache
- Memory
- Hard-disk drive (HDD)



Bus system

Connects the CPU, the Memory and the I/O interfaces

Data bus:

Transports the data from/to CPU

Address bus:

Contains memory address of reading/writing

Control bus:

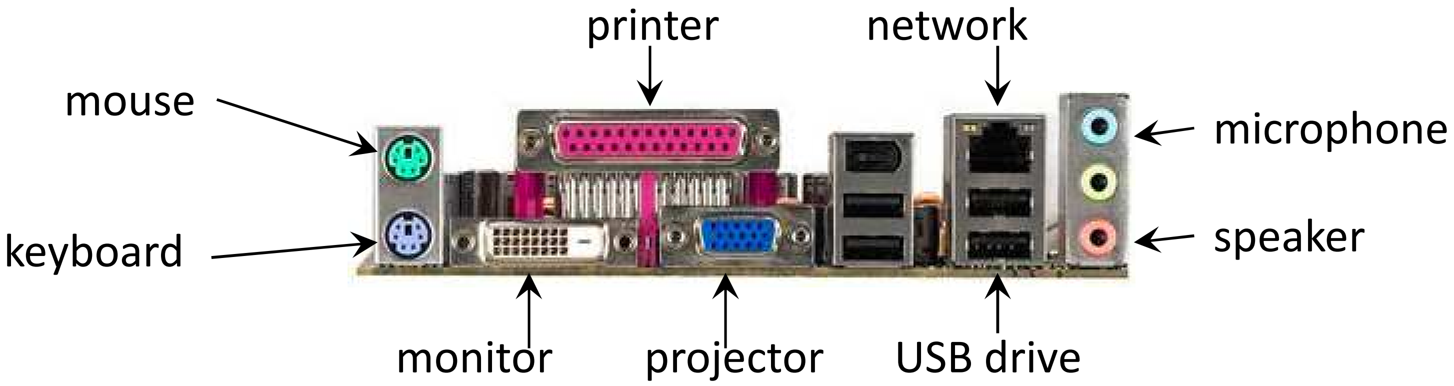
Carries control information

Input-Output Interfaces

It makes the system accessible to peripherals (world)

Connection to

- Input devices
- Output devices
- Storage devices
- Network devices



Peripherals

Input

- Keyboard
- Mouse
- Scanner

Storage

- Winchester (HDD)
- CD/DVD/Blu-ray drive
- USB drive
- Memory Card

Output

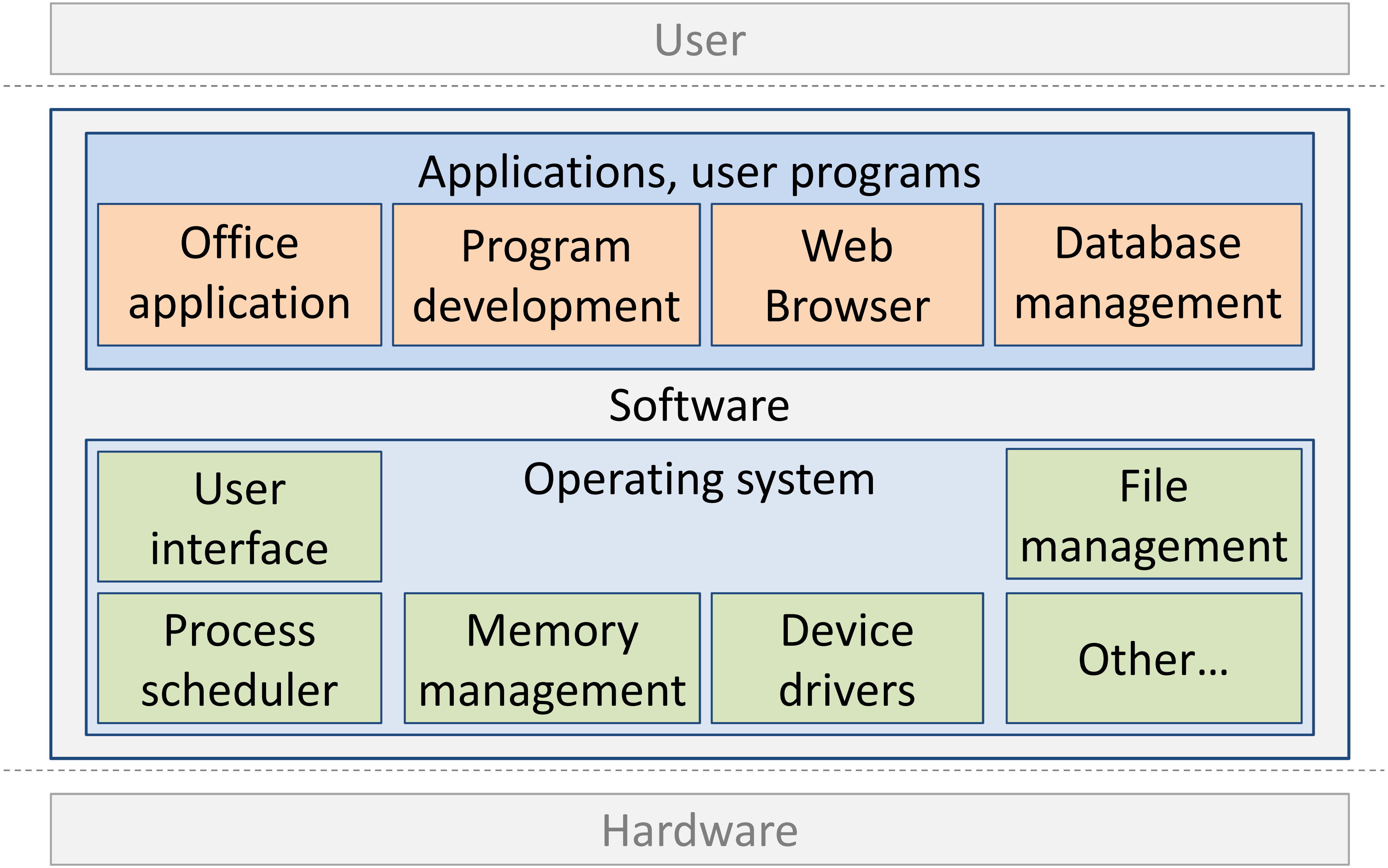
- Monitor
- Printer
- Projector

Network

- Ethernet
- Wi-Fi

Other

Software



User

Applications, user programs

Office application

Program development

Web Browser

Database management

Software

Operating system

User interface

File management

Process scheduler

Memory management

Device drivers

Other…

Hardware

Operating system

Collection of software that manages hardware resources and provides services for other programs

- **User interface:**
  supports human interaction (shell, GUI)
- **Program scheduler:**
  decides which program can run now, for how long time, which will the next
- **File management:**
  handles the files and directories of volume based on a file system

Operating system

- **Memory management:**
  provides ways to dynamically allocate portions of memory to programs at their request
- **Device drivers:**
  software developed to allow interaction with hardware devices
- **Security:**
  protect against illegal operation and access to data
- **Others:**
  Networking, Interrupt management, Utilities, …

User applications

- **File manager:**
  Windows Explorer, Midnight commander, …
- **Office application:**
  Microsoft Word/Excel, OpenOffice Write/Calc, …
- **Web browser:**
  Internet Explorer, Firefox, Chrome, …
- **Database manager:**
  Microsoft Access, MySQL, DB2, …
- **Graphical program:**
  Microsoft Paint, GIMP, Photoshop, …

User applications

- **Media player:**
  Windows Media Player, Flash Player, QuickTime, …
- **Computer game:**
  Minesweeper, Solitaire, NFS, CoD, FIFA, …
- **Anti-virus program:**
  Virus Buster, NOD32, AVG, …
- **Integrated Development Environment (IDE):**
  BorlandC, Netbeans, CodeBlocks, Dev-C++, …
- **Other**:
  …

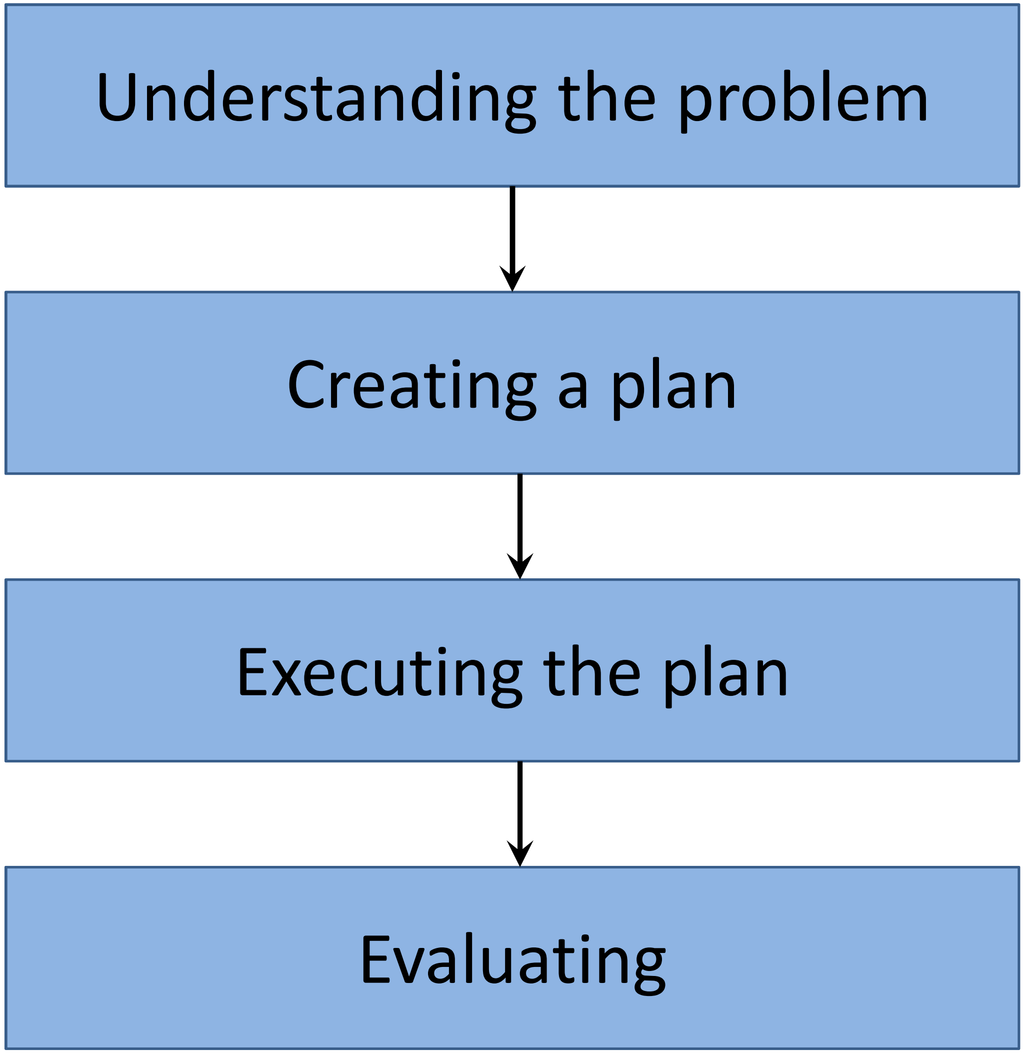
User

Human agent, who uses computer

- **Root:**
  Superuser, system administrator, has high privilege
- **„Simple” user:**
  computer is just a tool, not the purpose of work
- **Programmer:**
  develops computer applications, writes programs

Problem solving

Pólya’s problem solving steps



Understanding the problem

Creating a plan

Executing the plan

Evaluating

Understanding the problem

- What is the task?
- What is the *unknown* (required result)?
- What is the relationship between the given information and the *unknown*?
- Is the given information enough to solve the problem?

Creating a plan

General techniques:

- Finding known similar problems (if exists)
- Reshaping the original problem to a similar known problem
- Divide the problem to shorter solvable problems
- Generalizing a restricted problem
- Finding existing work that can help in the search for a solution

Executing the plan

- Follow the steps of the plan
- Each element of the plan should be checked as it is applied
- If a part of the plan is unsatisfactory, the plan should be revised

Evaluating

The result should be examined

- Is it correct?
- Is it full?
- Is it valid?
- Has the problem been solved?

An example

Problem: What is the sum of
110010110 and 101110101 in binary notation?

1) Understanding: addition of numbers where each digit can be just 0 or 1.

2) Plan: Similar method to addition of decimal numbers just the number of possible digits is 2

3) Execute: Sum of digit pairs taking into account the carry

+	0	1
0	0	1
1	1	0

4) Evaluating: check with subtraction or with conversion

R*esult: 1100001011*

Number systems

Binary systems

Why binary systems are so important?

- There are many binary systems in our environment.
- The **computer is binary** (digital).

no	yes
false	true
absent	present
close	open
switched off	switched on
insulator	conductor
electric current flows	no electric current
0	**1**

Decimal number system

10 different symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

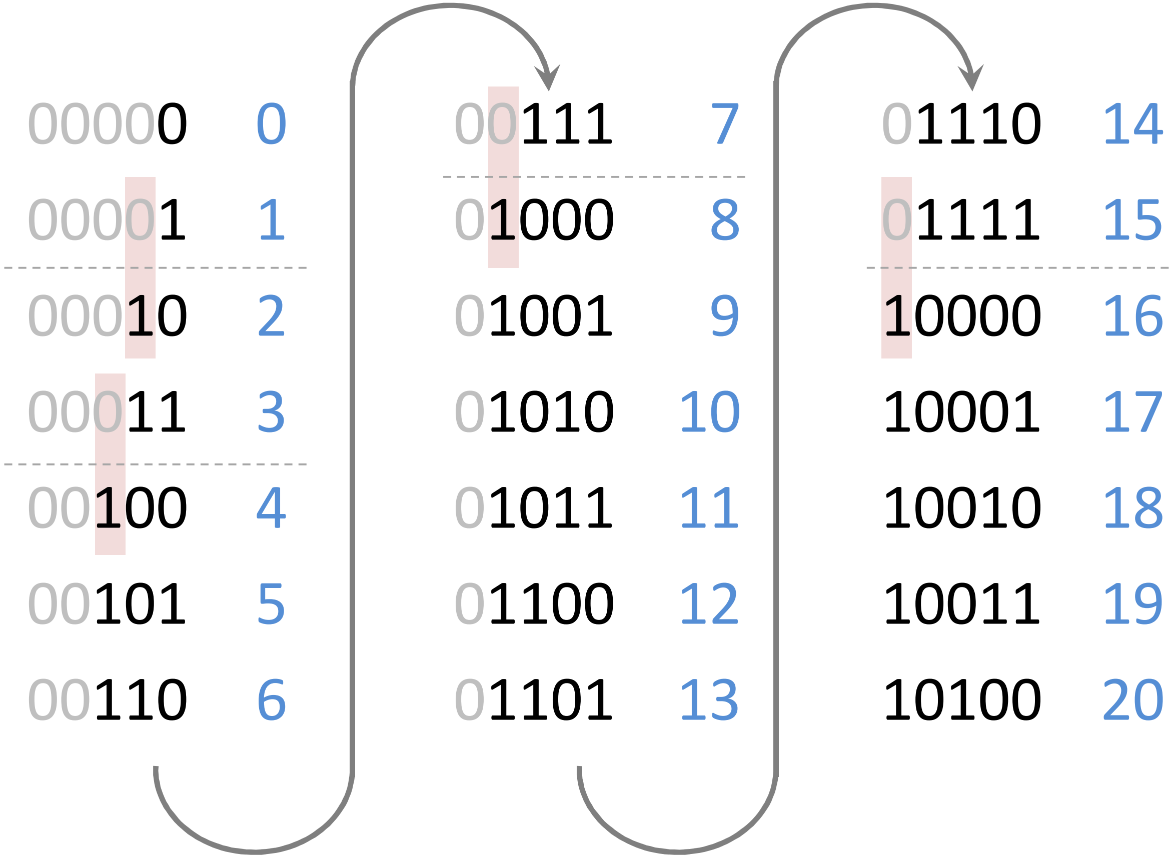
How do we count in decimal?

…	…	…	…	…
08	18	098	598	0998
09	19	099	599	0999
10	20	100	600	1000
11	21	101	601	1001
12	22	102	602	1002
…	…	…	…	…

Binary number system

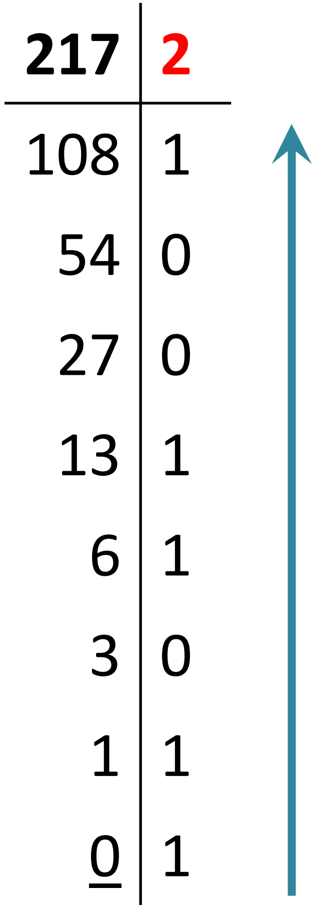
Only 2 different symbols: 0, 1

How do we count using binary?



00000	0
00001	1
00010	2
00011	3
00100	4
00101	5
00110	6
00111	7
01000	8
01001	9
01010	10
01011	11
01100	12
01101	13
01110	14
01111	15
10000	16
10001	17
10010	18
10011	19
10100	20

Conversion from decimal to binary



conversion of 217:	**217**	**2**
217 = 2*108+1	108	1
108 = 2*54+0	54	0
54 = 2*27+0	27	0
27 = 2*13+1	13	1
13 = 2*6+1	6	1
6 = 2*3+0	3	0
3 = 2*1+1	1	1
1 = 2*0+1	$\underline{0}$	1

$217_{10} = 11011001_2$

Conversion from binary to decimal

Decimal (10):

$2495 = 2*1000+4*100+9*10 +5*1$

$2495 = 2*10^3 \quad +4*10^2 +9*10^1+5*10^0$

Binary (2):

$1010011_2 \ = 1*2^6 +0*2^5 \ +1*2^4 \ +0*2^3+0*2^2+1*2^1 +1*2^0$

$1010011_2 \ = 1*64 +0*32 +1*16 +0*8 \ +0*4 +1*2 \ +1*1$

$1010011_2 \ = 64+16+2+1 = 83_{10}$

Binary arithmetic: addition

Addition of digits:

$0 + 0 = \ \ 0$

$0 + 1 = \ \ 1$

$1 + 0 = \ \ 1$

$1 + 1 = \boxed{1}\boxed{0}$

carry

result digit

Example:

$$\begin{array}{r} 1\,1\,0\,0\,1\,0\,1\,1\,0 \\ +\ \underline{\ 1\,0\,1\,1\,1\,0\,1\,0\,1} \\ 1\,1\,0\,0\,0\,0\,1\,0\,1\,1 \end{array} \qquad \begin{array}{r} 4\,0\,6 \\ +\ \underline{\ 3\,7\,3} \\ 7\,7\,9 \end{array}$$

Real numbers in binary

- Fractional numbers from binary to decimal

$1010.011_2 = 1*2^3+0*2^2+1*2^1+0*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}$

$1010.011_2 = 1*8+0*4+1*2+0*1+0*1/2+1*1/4+1*1/8$

$1010.011_2 = 8 + 2 + 1/4 + 1/8 = 10.375_{10}$

- Fractional numbers from decimal to binary

10.375_{10}

10._

	10	**2**
10 = 5*2+0	5	0
5 = 2*2+1	2	1
2 = 1*2+0	1	0
1 = 0*2+1	0	1

_.375

	2	**375**
0+0.750 = 0.375*2	0	750
1+0.500 = 0.750*2	1	500
1+0.000 = 0.500*2	1	000

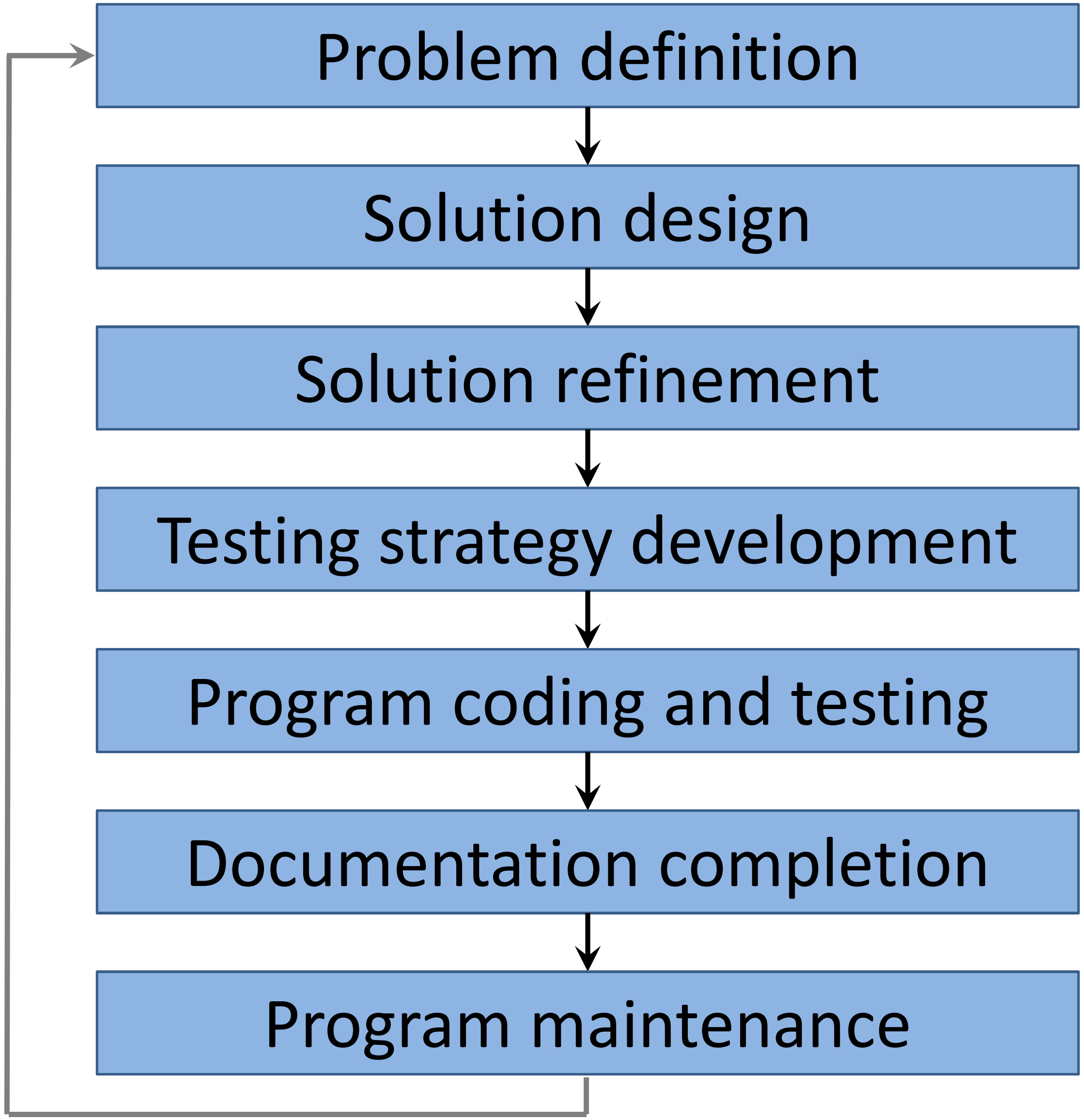
$10.375_{10} = 1010.011_2$

Hexadecimal notation

- 16 different symbols
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Conversion to/from binary
  - Each hexadecimal digit is 4 binary digits (bits)
  - $5E_{16} = 0101\ 1110_2$ $(= 94_{10})$
- Conversion to/from decimal
  - By powers of 16 or by double conversion (16 ⇆ 2⇆10)
- Why it is used in IT?
  - Close relation to binary
  - Short (2 hexadecimal digit is 1 byte)

Software Life Cycle

Software Life Cycle



Problem definition

Solution design

Solution refinement

Testing strategy development

Program coding and testing

Documentation completion

Program maintenance

1: Problem definition

- Similar to Pólya’s first step
- The description of the problem must be precise
- User and programmer must work together
- It leads to complete specifications of the problem, the input data and the desired output

2: Solution design

- Definition of the outline of solution
- Division of the original problem into a number of subproblems
- Subproblems are smaller and easier to solve
- Their solution will be the components of our solution
- „Divide and conquer”
- Finally the problem will be converted to a plan of well-known steps

3: Solution refinement

- Previous step is in very high-level: no indication given how subtasks are to be accomplished
- Refinement is necessary by adding more details
- Avoid any misunderstandings
- A precise method consists of a sequence of well defined steps called an **algorithm**
- Representation: pseudocode, flowchart, etc.

4: Testing strategy development

- It is necessary to try the algorithm with several different combinations of input data to make sure that it will give correct results in all cases
- These different combinations of input data are called **test case**
- It covers not only normal input values, but also extreme input values to test the limits
- Complete test cases can be used to check the algorithm

5: Program coding and testing

- Description of algorithm in previous level cannot be executed directly by computer
- Translation needed to a **programming language**
- After coding program must be tested using our testing strategy
- If an error has been discovered, appropriate revision must be made, and than the test rerun until the program gives correct solution under all circumstances
- Process of coding and testing called **implementation**

6: Documentation completion

- Documentation begins with the first step of development and continues throughout the whole lifetime of the program
- It contains:
  - Explanations of all steps
  - Design decisions that were made
  - Occurred problems
  - Program list
  - User instructions
  - etc.

7: Program maintenance

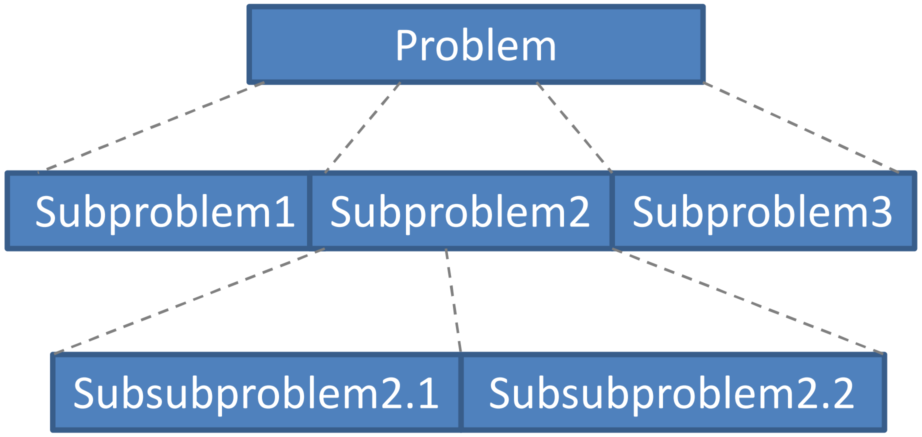
- The program can’t wear out
- Sometimes the program may fail
- The reason of a program fail is that it was never tested for this circumstance
- Elimination of newly detected error is necessary
- Sometimes the users need new features to the program
- Update of documentations is needed

Solution design

by Break-Out Diagrams

Break-Out Diagrams

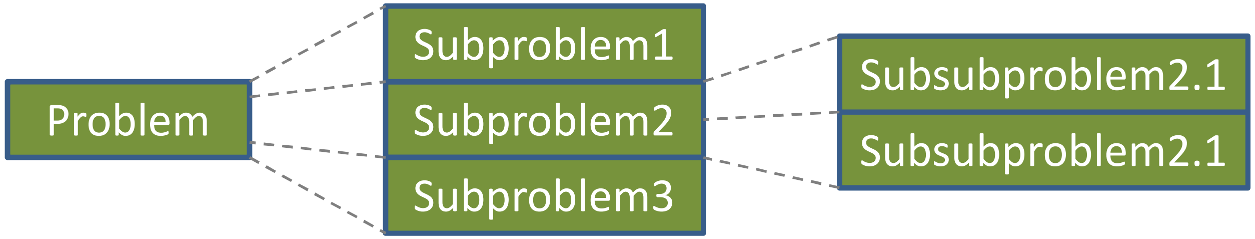
- Useful way to make the problem solving manageable
- Tree-like (hierarchical) skeleton of problems
- For viewing problems in levels
- Styles:
  - Vertical
  - Horizontal



Problem

Subproblem1 Subproblem2 Subproblem3

Subsubproblem2.1 Subsubproblem2.2



Problem

Subproblem1

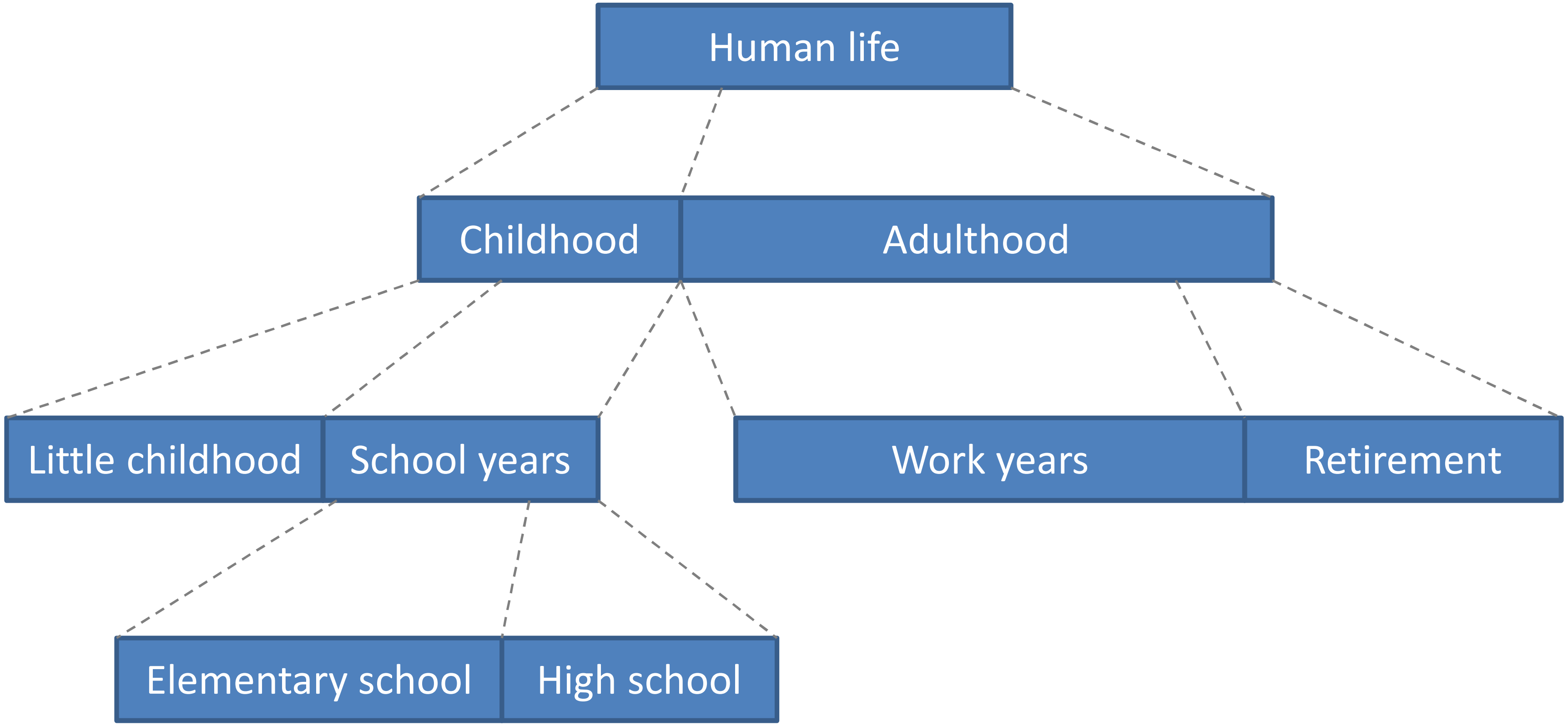
Subproblem2

Subproblem3

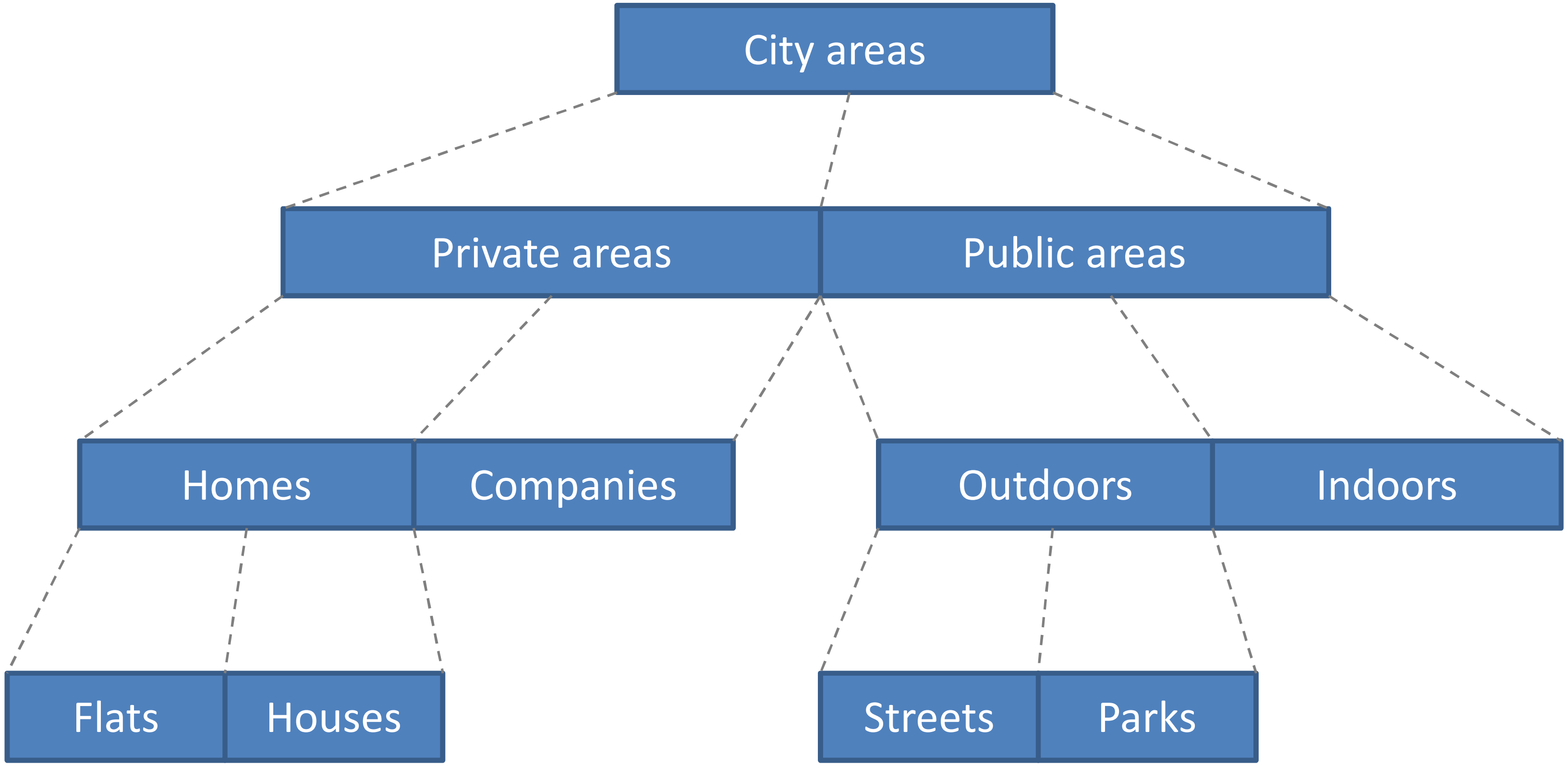
Subsubproblem2.1

Subsubproblem2.1

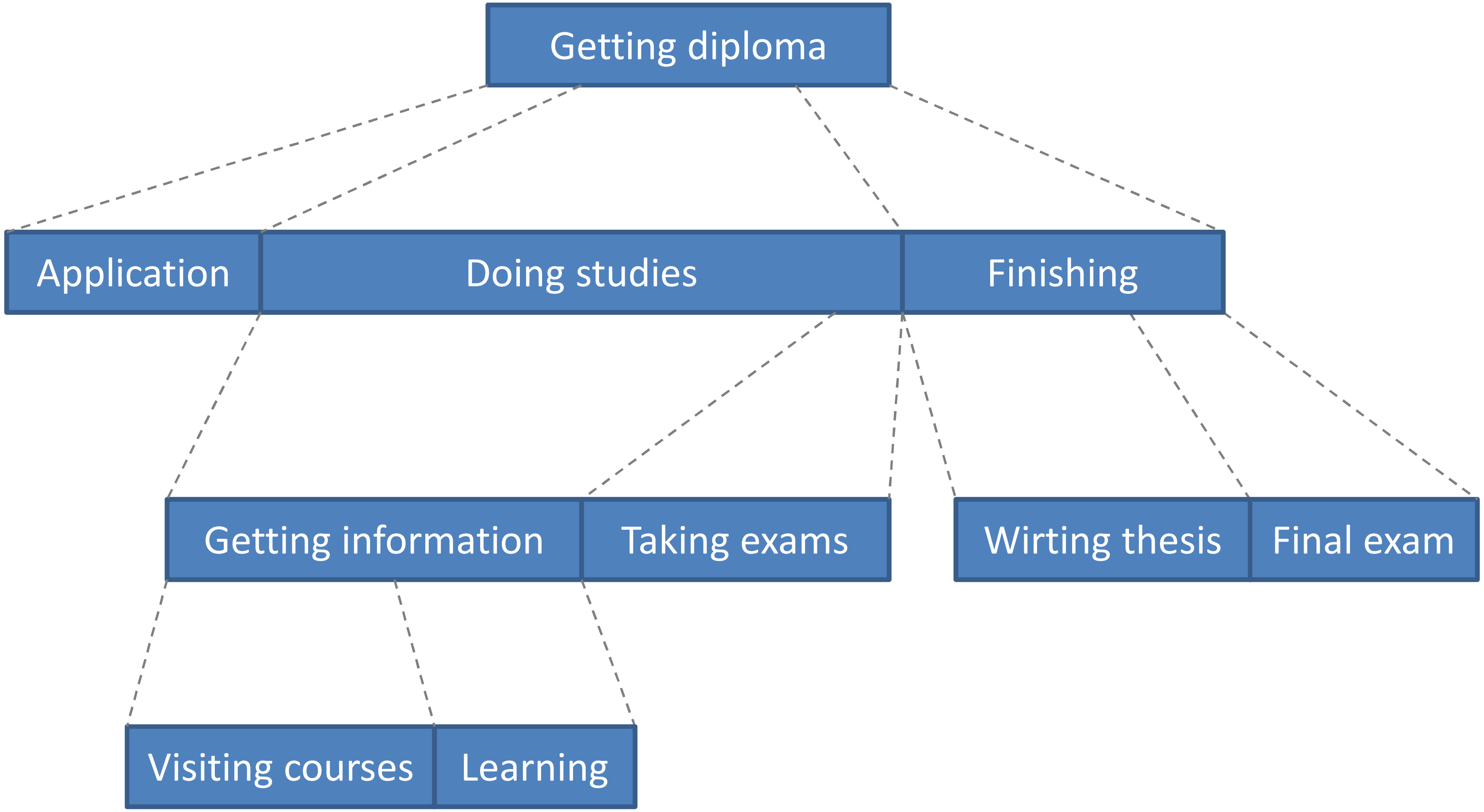
Time BOD


Human life
Childhood
Adulthood
Little childhood
School years
Work years
Retirement
Elementary school
High school

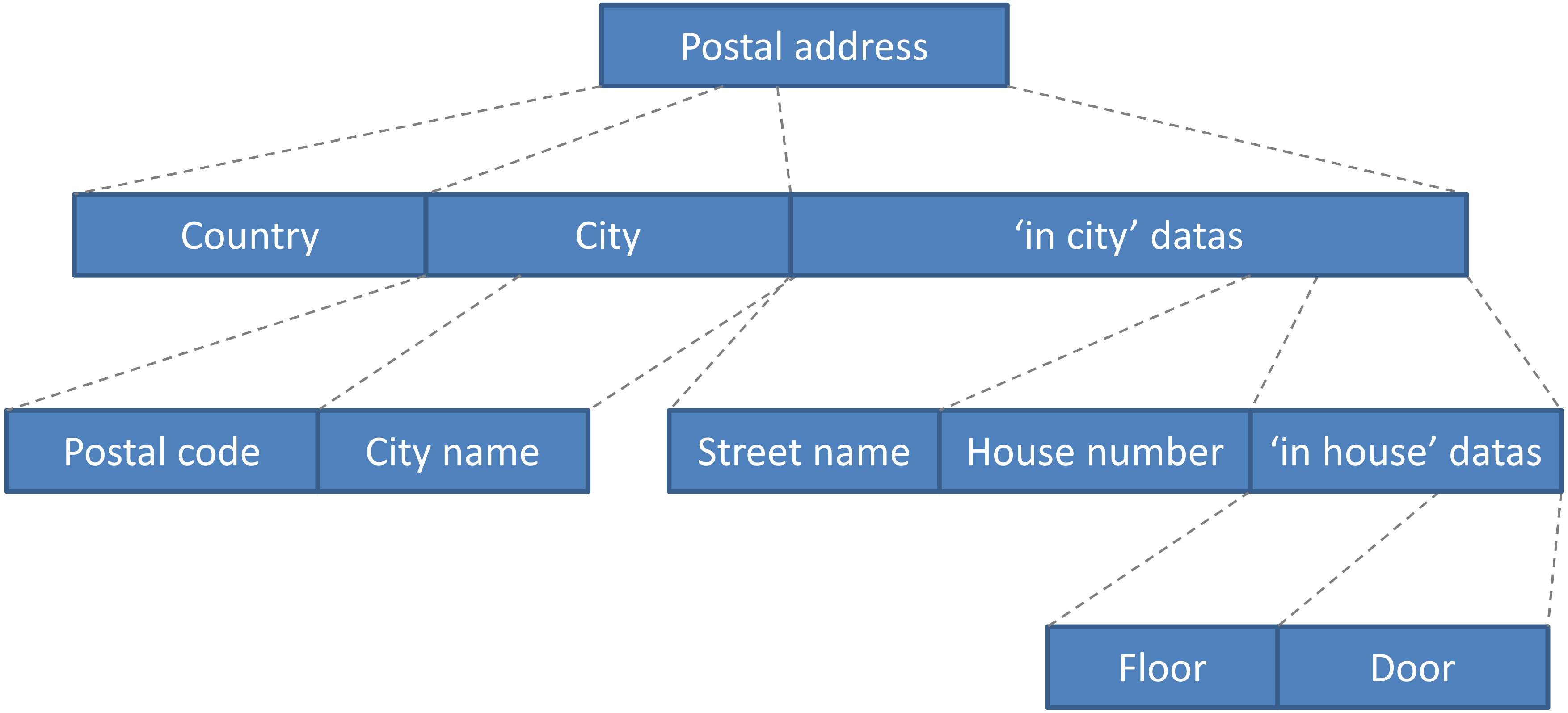
Space BOD


City areas
Private areas
Public areas
Homes
Companies
Outdoors
Indoors
Flats
Houses
Streets
Parks

Action BOD


Getting diploma
Application
Doing studies
Finishing
Getting information
Taking exams
Wirting thesis
Final exam
Visiting courses
Learning

Data BOD

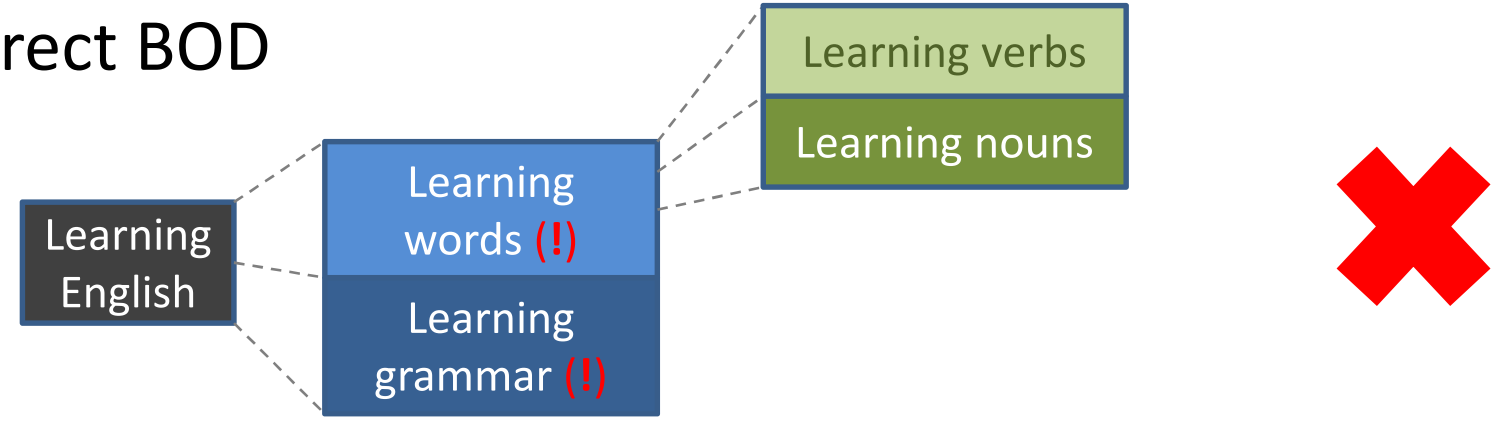

Postal address
Country
City
'in city' datas
Postal code
City name
Street name
House number
'in house' datas
Floor
Door

Properties of BODs

- **Consistent**
  Each break-out must be the same kind.
- **Orderly**
  All blocks at the same level must be separate or independent.
- **Refined**
  Each box of a given level must be break-out of a box at the previous level.
- **Cohesive**
  All of the items within a breakout box must fit together.

Mistakes and corrections

Incorrect BOD



Learning English

Learning words (!)

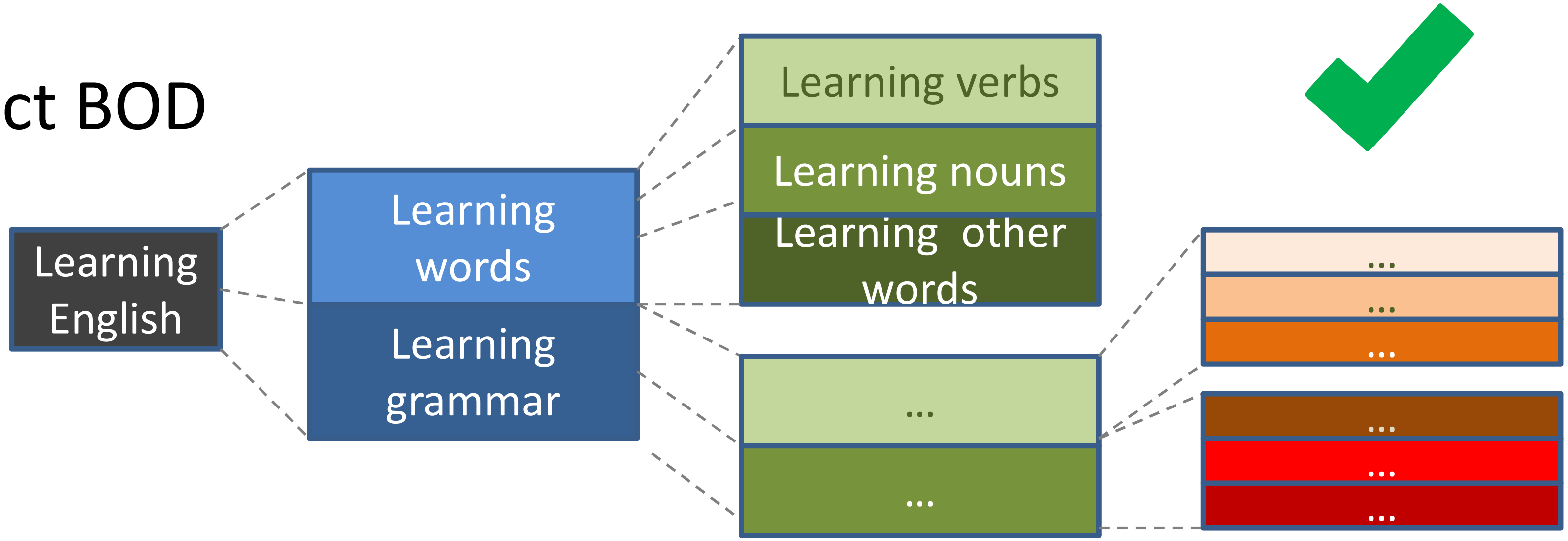
Learning grammar (!)

Learning verbs

Learning nouns

✗

Correct BOD



✓

Learning English

Learning words

Learning grammar

Learning verbs

Learning nouns

Learning other words

…

…

…

…

…

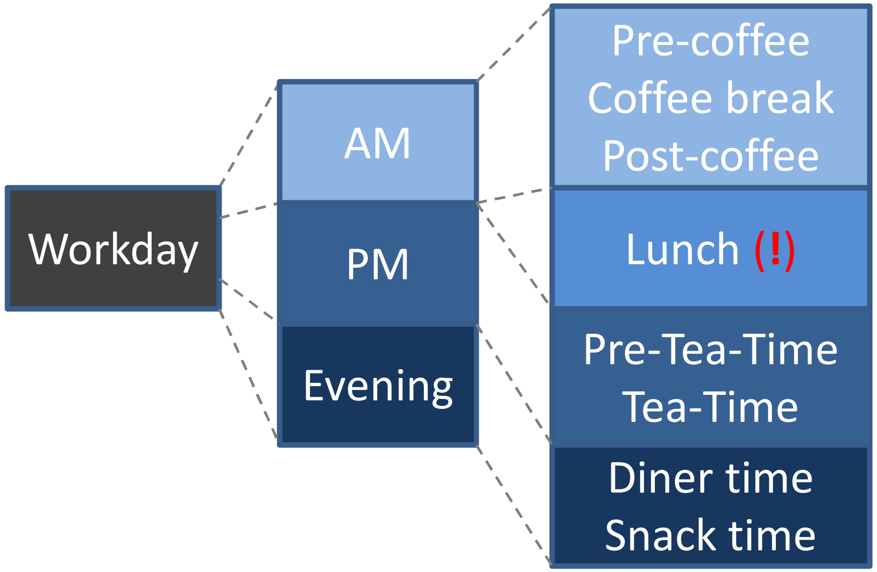
…

…

…

Mistakes and corrections

Incorrect BOD ✗



Workday

AM

PM

Evening

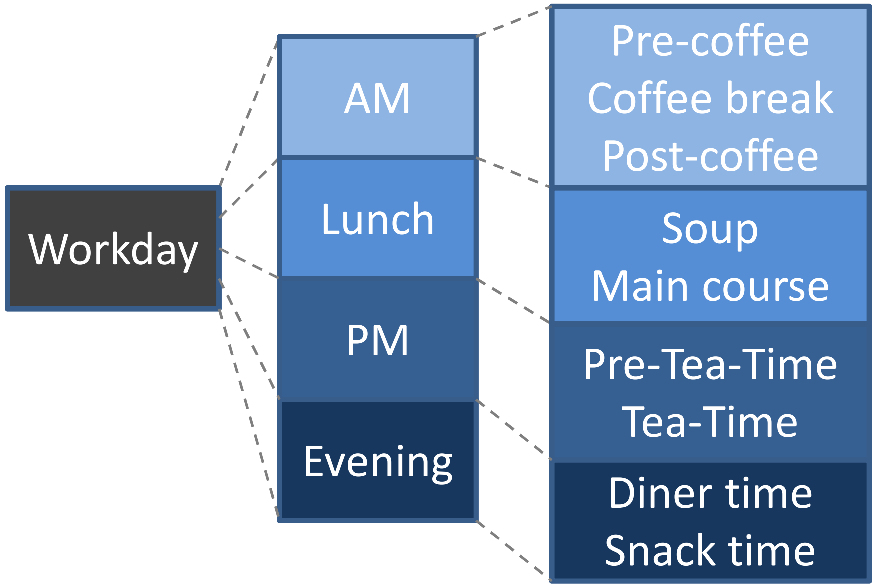
Pre-coffee
Coffee break
Post-coffee

Lunch (!)

Pre-Tea-Time
Tea-Time

Diner time
Snack time

Correct BOD ✓



Workday

AM

Lunch

PM

Evening

Pre-coffee
Coffee break
Post-coffee

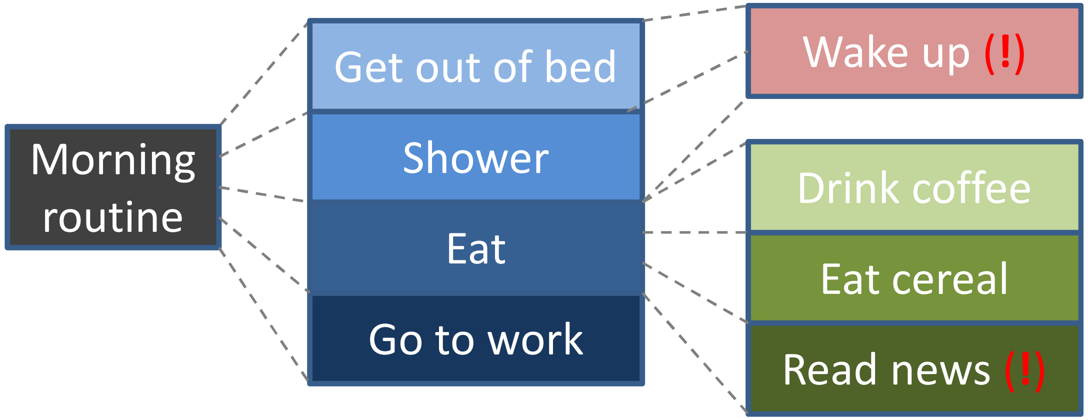
Soup
Main course

Pre-Tea-Time
Tea-Time

Diner time
Snack time

Mistakes and corrections

Incorrect BOD



Morning routine

Get out of bed

Shower

Eat

Go to work

Wake up (!)

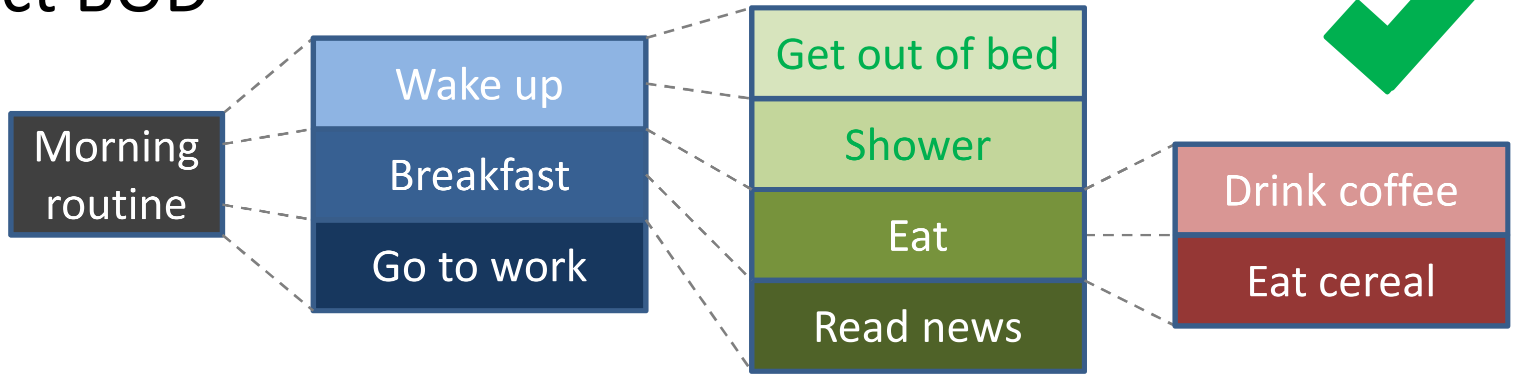
Drink coffee

Eat cereal

Read news (!)

✗

Correct BOD



Morning routine

Wake up

Breakfast

Go to work

Get out of bed

Shower

Eat

Read news

Drink coffee

Eat cereal

✓

Solution refinement

Algorithms

Algorithm

Plan for performing a sequence of well-understood actions to achieve the result.

Precise definition of the actions to be performed to accomplish each task of solution design.

Some properties:

- precise, unambiguous
- specified for all possible cases
- finite sequence of actions
- achieves the result
- efficiency, elegance, easy to use, …

Representation of algorithms

- Algebraic
- Data-flow diagram
- **Flowchart**
- Graphs or plots
- Hierarchical
- **Pseudocode**
- Stuctogram
- Tabular
- **Verbal**

Example

Function **y=sign(x)**

- What is it?
- What does it mean?
- What is the result?
- How is it work?
- How can we determine its value?
- If x is -4, what is the value of y?
- …

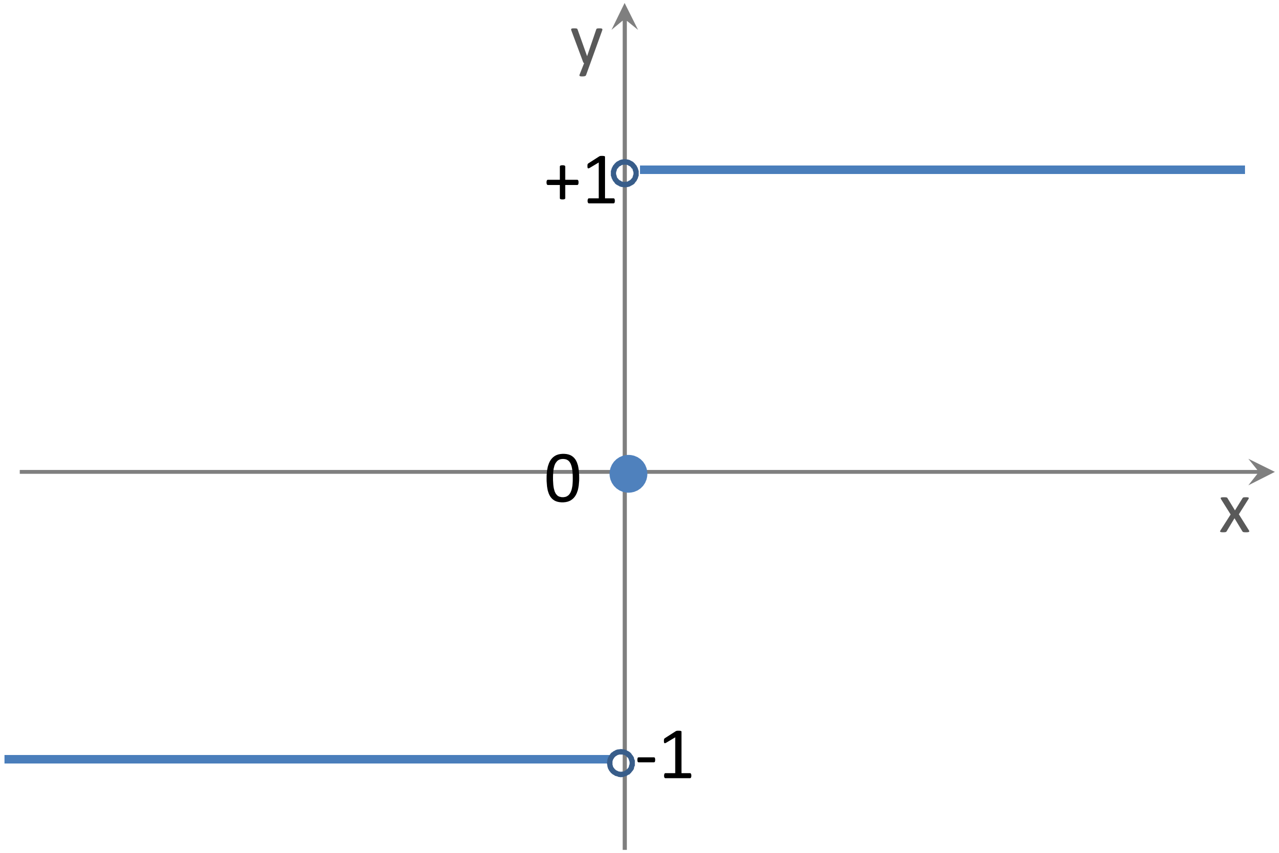
y=sign(x)

Verbal representation:

1. If input value x is 0, set the result to y=0.
2. Otherwise if x>0, let the value of this function +1.
3. Else if x less then 0, give the function -1.

y=sign(x)

Graph representation:



y=sign(x)

‘Algebraic-like’ representation:

$x \in \Re$

$y \in \{-1, 0, +1\}$

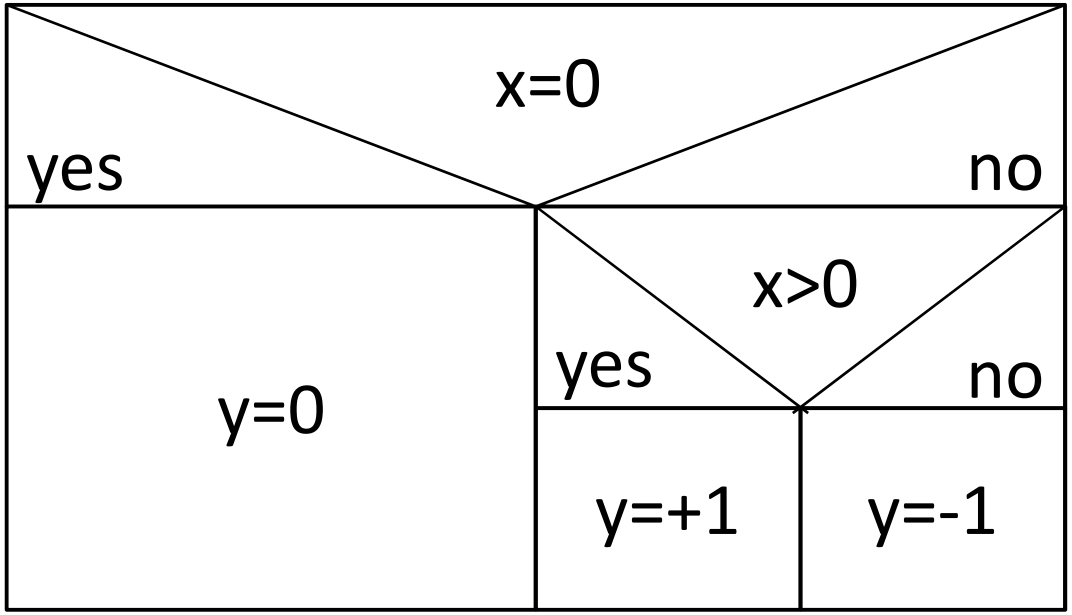
$\forall x, x>0 \Rightarrow y=+1$

$\forall x, x<0 \Rightarrow y=-1$

$x=0 \Rightarrow y=0$

y=sign(x)

Structogram representation:



x=0

yes

no

y=0

x>0

yes

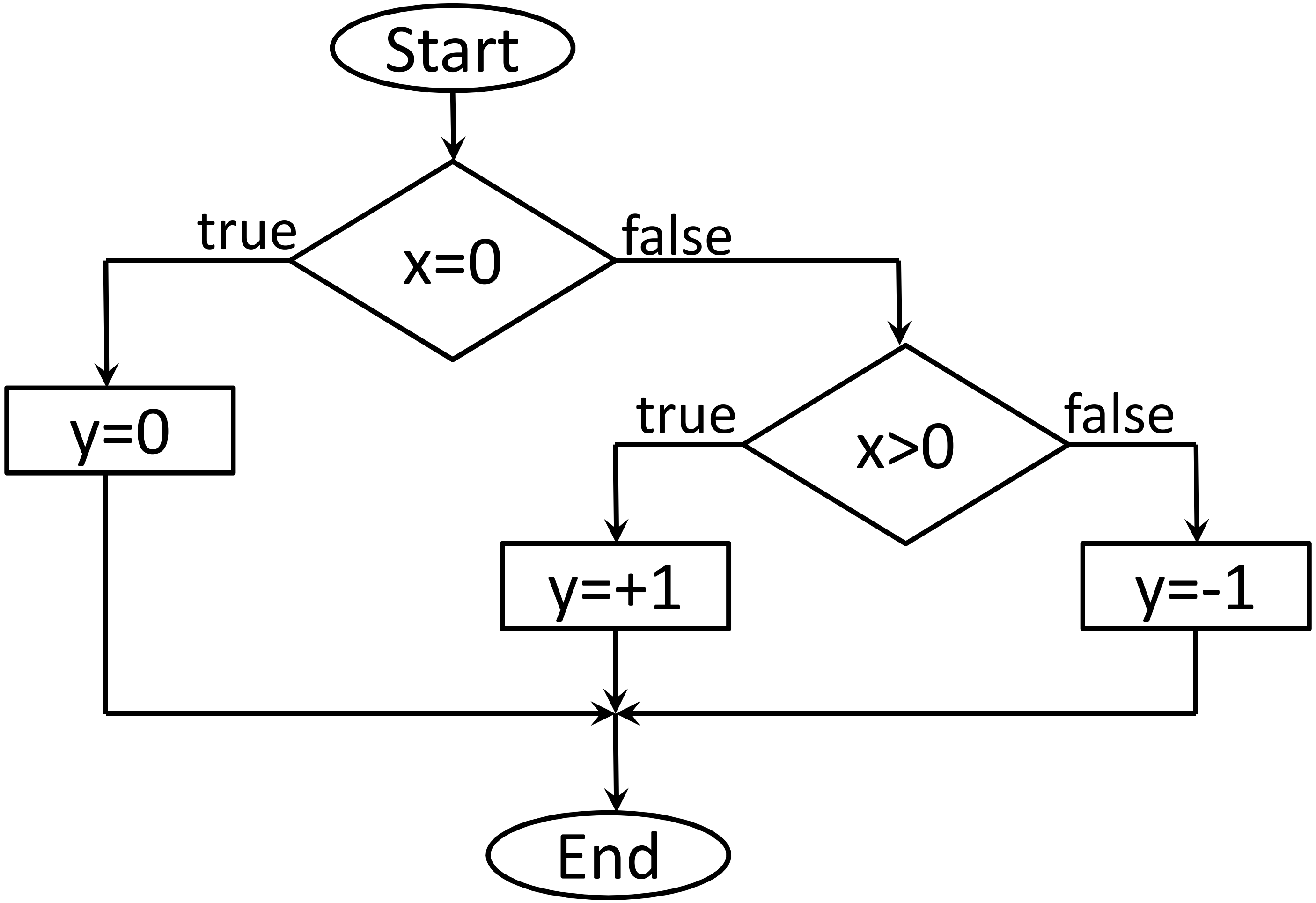
no

y=+1

y=-1

y=sign(x)

Flowchart representation:



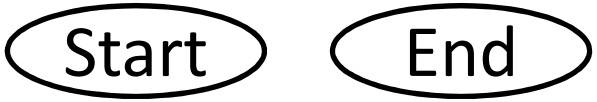
y=sign(x)

Pseudocode representation:

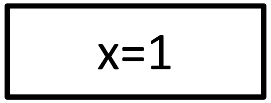
```
if x=0 then
  y=0
else
  if x>0 then
     y=+1
  else
     y=-1
  endif
endif
```

Flowchart

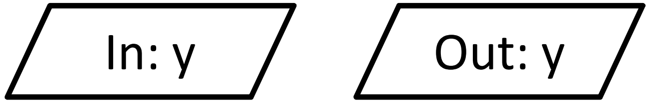
- Starting/finish point



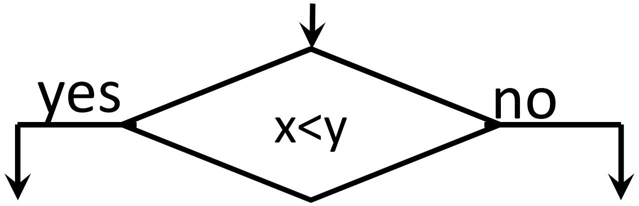
- Atomic instruction



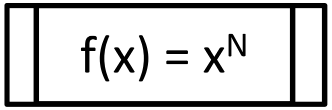
- Input/output



- Condition



- Inserting other algorithm

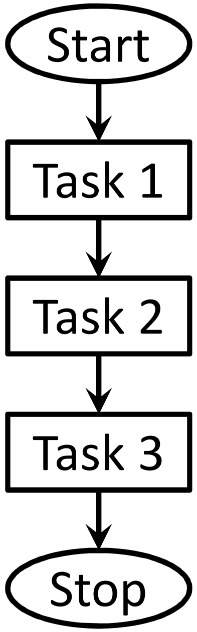


- We can go along arrows.

Base structures of algorithms

by flowcharts

Sequence



Start

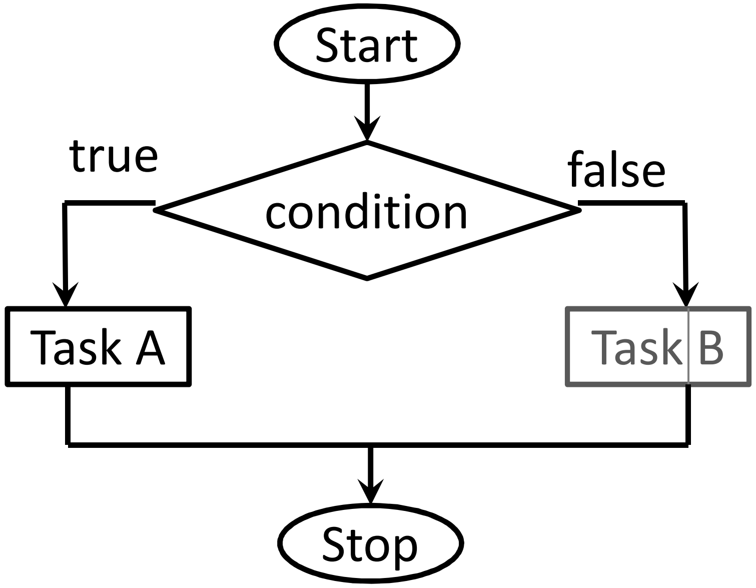
Task 1

Task 2

Task 3

Stop

Selection



Start

true

condition

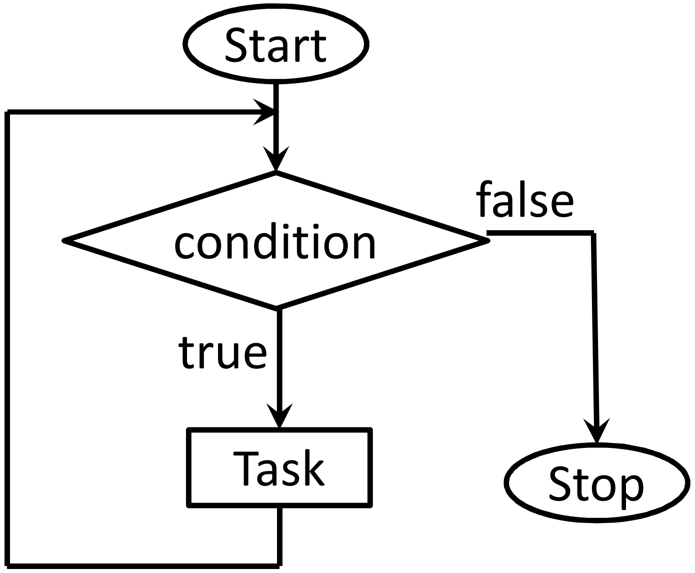
false

Task A

Task B

Stop

Iteration



Start

condition

false

true

Task

Stop

Modifying algorithms

Algorithms often go through many changes to be better.

- Generalizing:
  making them apply to more cases

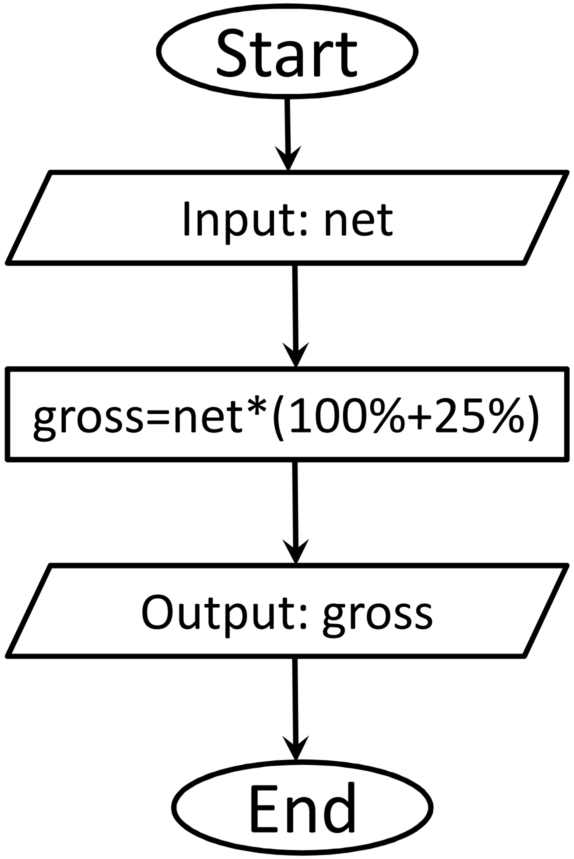
- Extending:
  to include new cases

- Foolproofing:
  making an algorithm more reliable, failsafe or robust

- Embedding:
  re-using that algorithm within another algorithm

Generalizing algorithms

Original:



Start

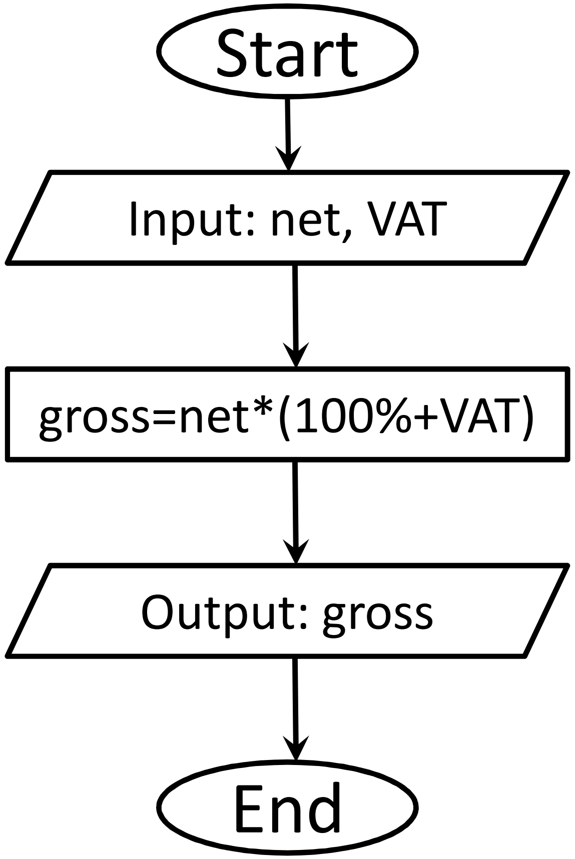
Input: net

gross=net*(100%+25%)

Output: gross

End

Generalized:



Start

Input: net, VAT

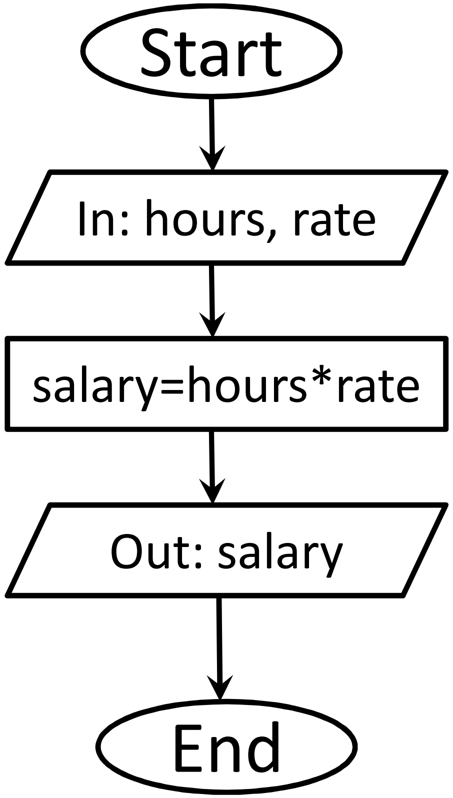
gross=net*(100%+VAT)

Output: gross

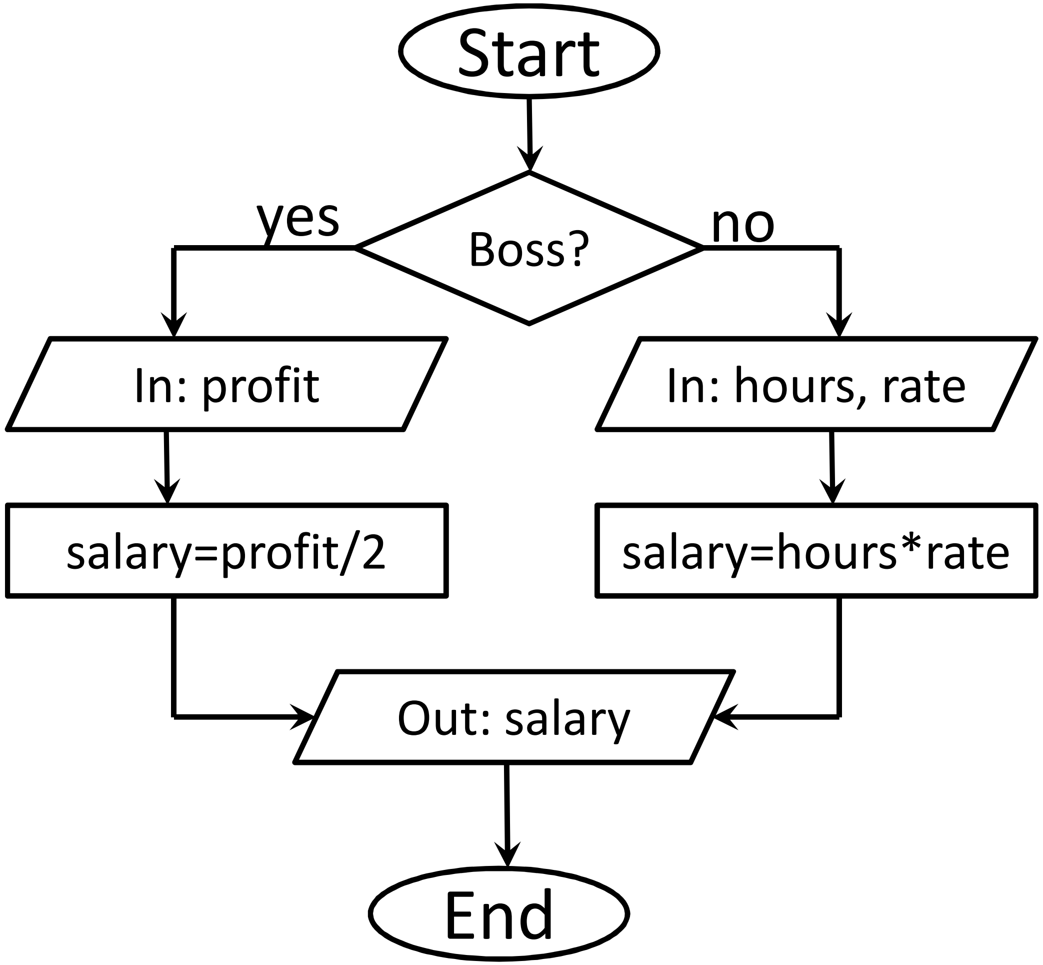
End

Extending algorithms

Original:

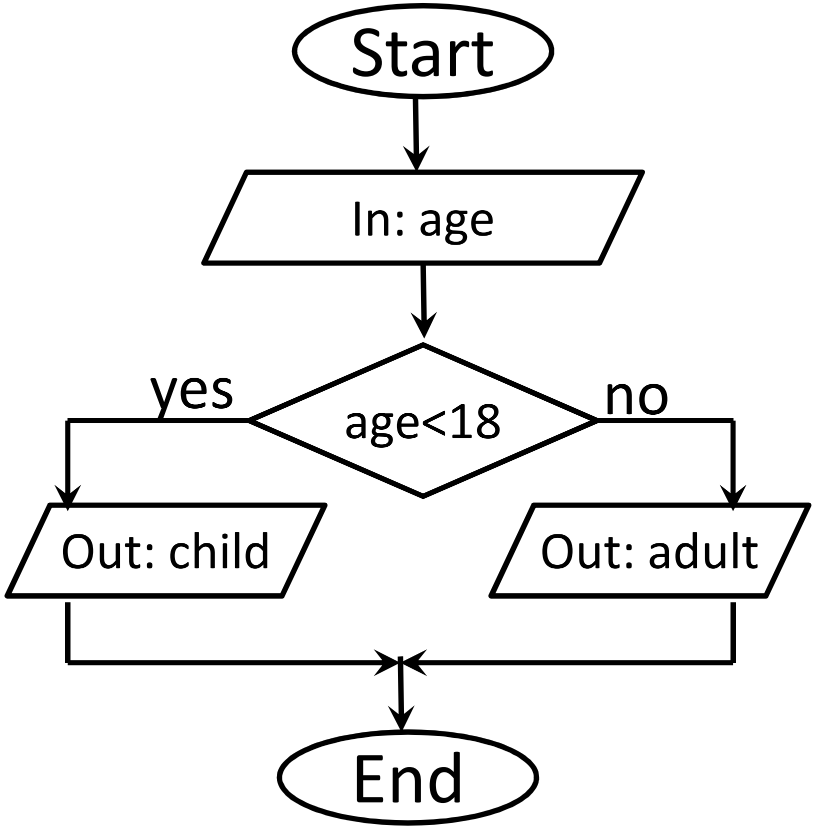


Extended:

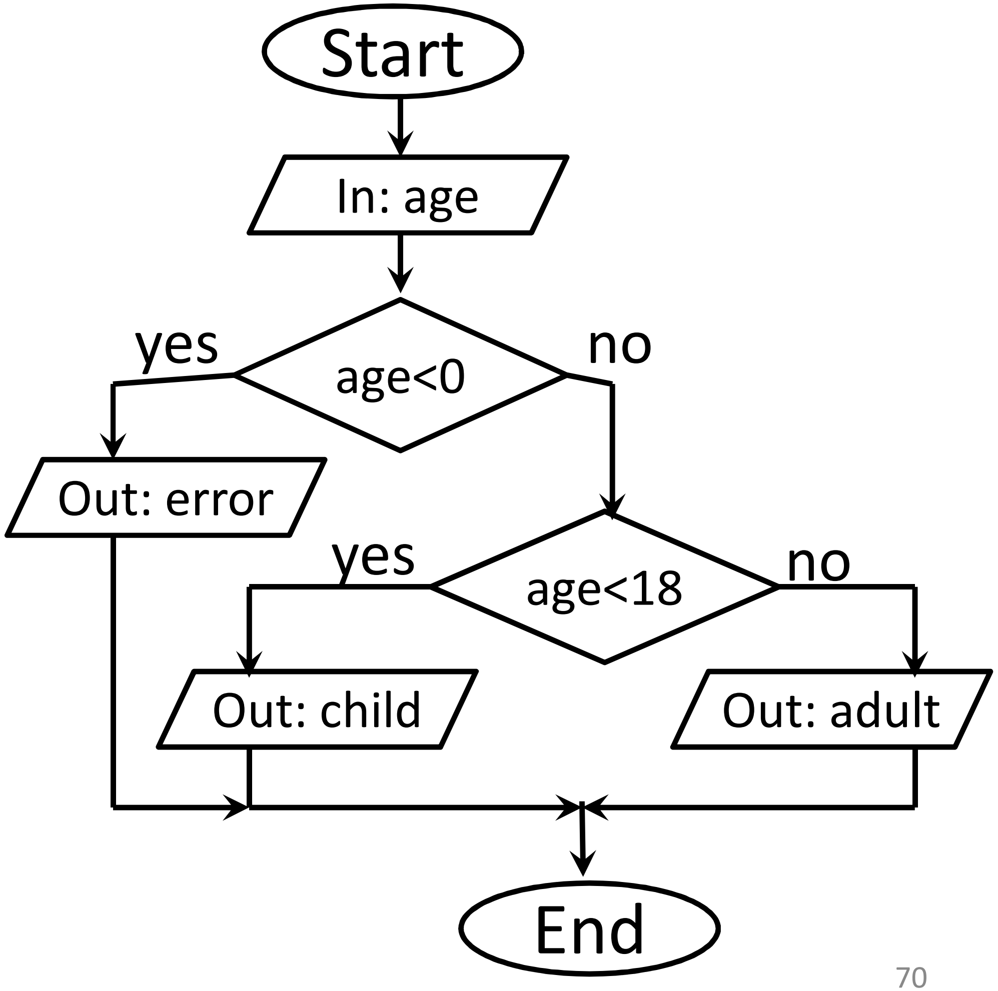


Foolproofing algorithms

Original:

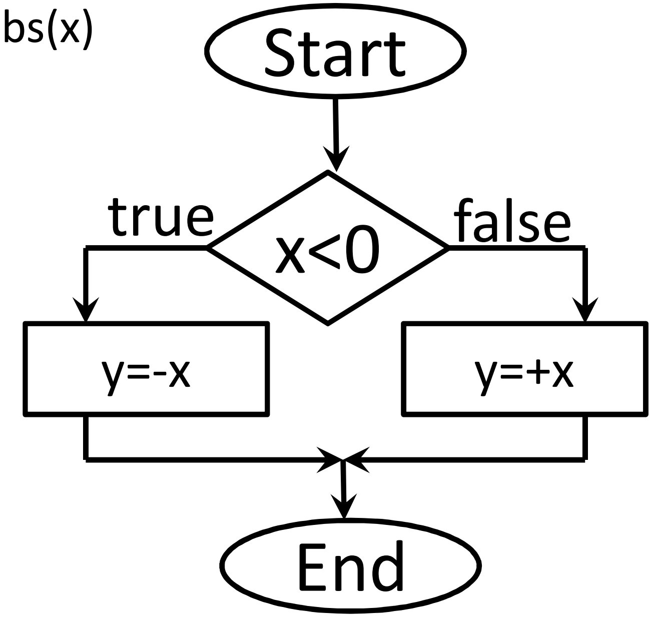


Foolproofed:

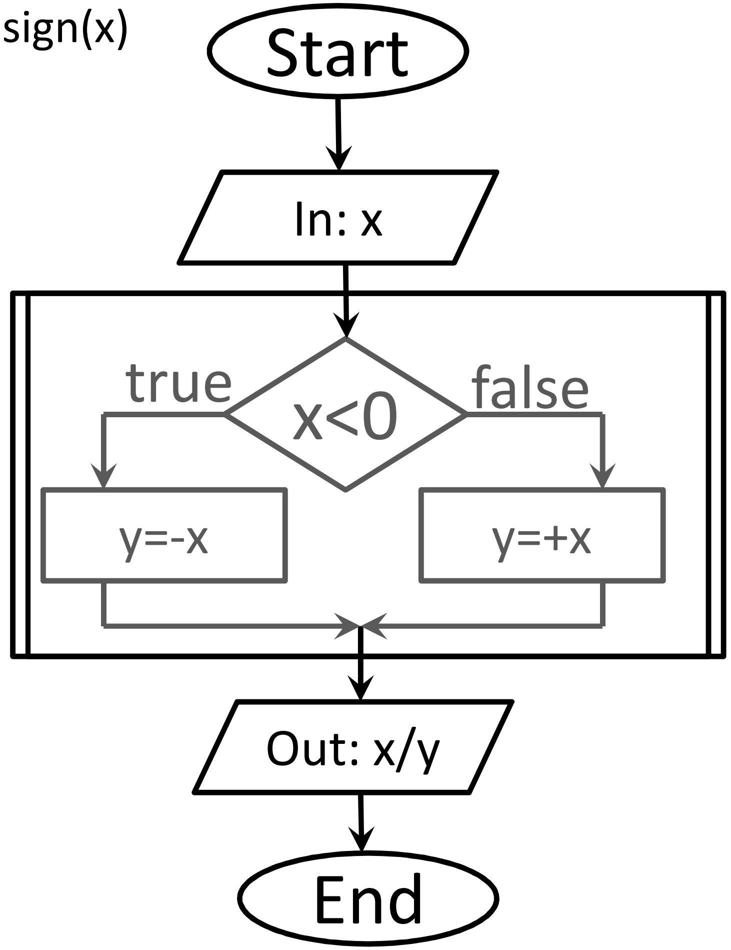


Embedding algorithms

Original:
y=abs(x)



Embedded:
y=sign(x)



Not foolproofed!

Alternative algorithms

There are often many ways to achieve the same thing.

Algorithms can be different in structure, but they can be equivalent in behavior.

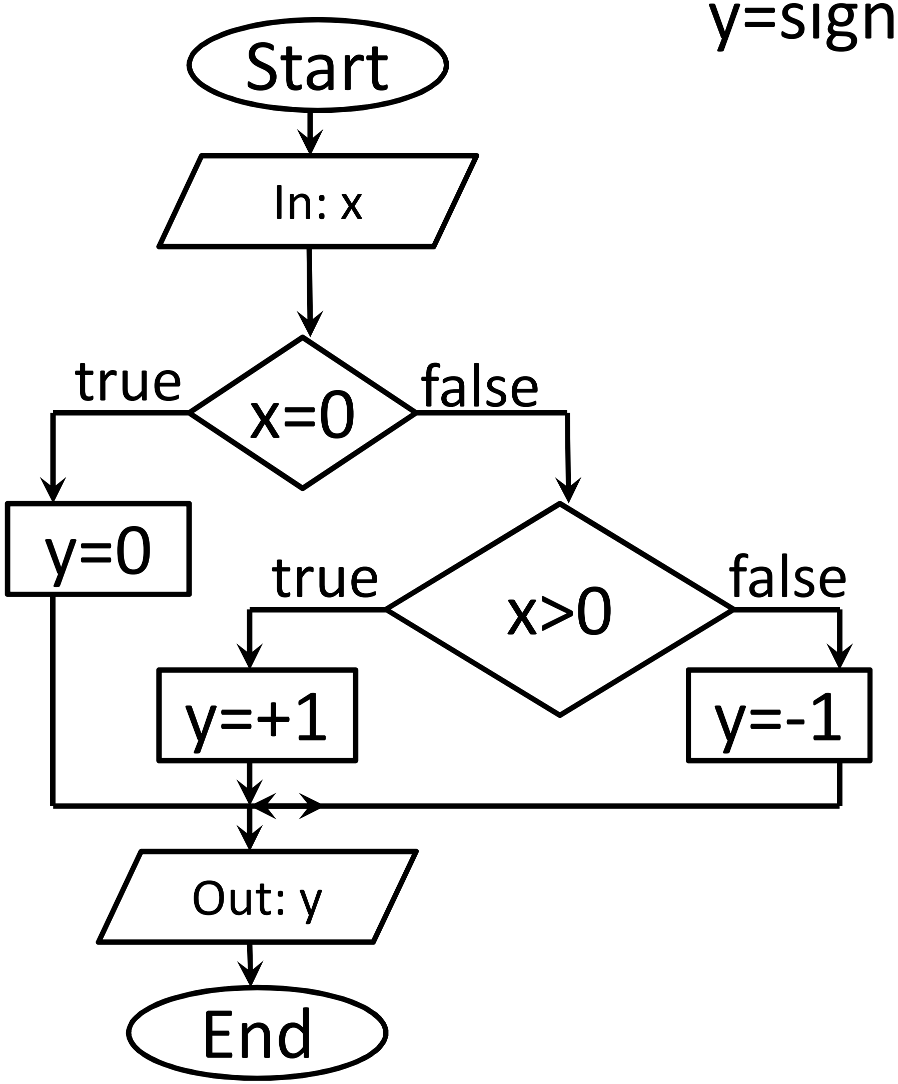
It means: for identical input data, they will produce identical results.

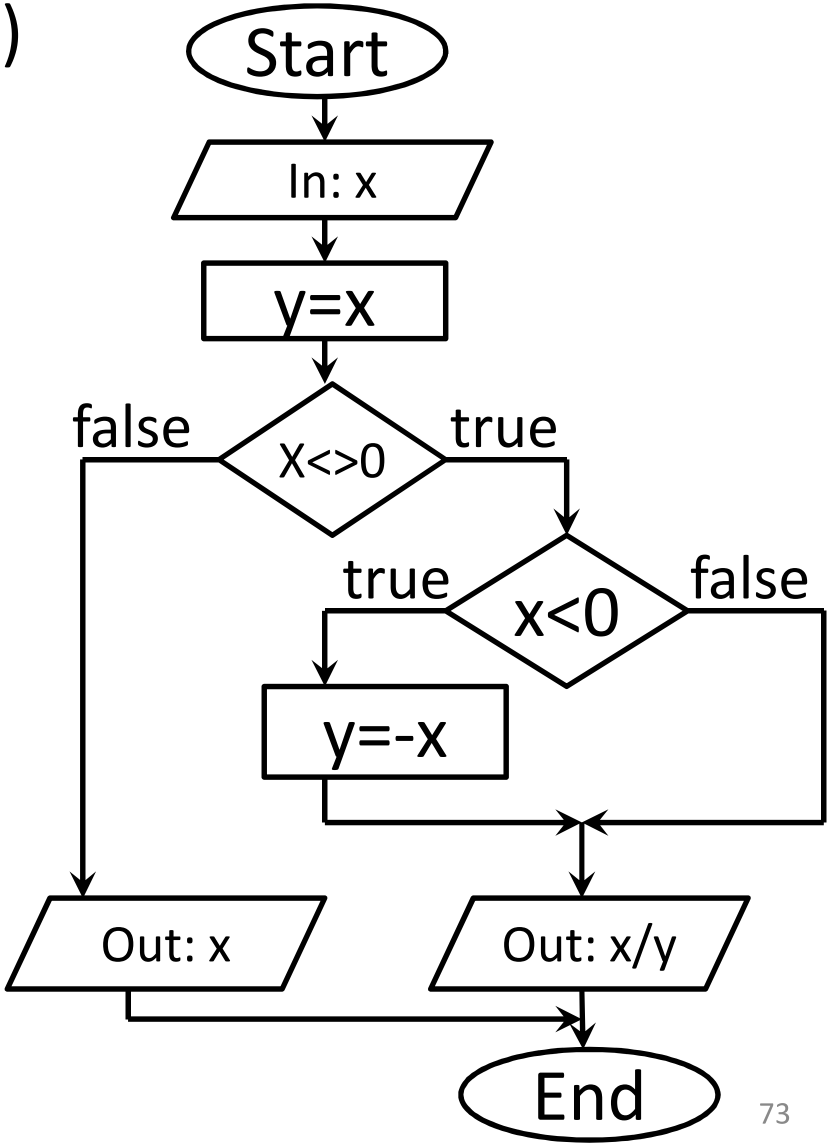
Sometimes there is serious reason to prefer one algorithm over the other, while sometimes there isn’t.

In some cases, one algorithm may be considerably smaller, faster, simpler, or more reliable than another.

Alternative algorithms

y=sign(x)





Properties of algorithms

- Complete:
  all of actions must be exactly defined
- Unambiguous:
  there is only one possible way of interpreting actions
- Deterministic:
  if the instructions are followed, it is certain that the desired result will always be achieved
- Finite:
  the instructions must terminate after a limited number of steps

Wrong algorithms

How to get to the 5th floor from 2nd by elevator?

```
1. Call the lift.
2. Get in.
3. Push ‘5’ button.
4. Wait.
5. If the door opens, get out.
```

Problems (not complete):

- If the list goes downward…
- If the lift stops on 3rd floor for somebody…

Wrong algorithms

How to make fried chicken?

```
1. Put the chicken into the oven.
2. Set the temperature.
3. Wait until it is done.
4. Serve it.
```

Problems (ambiguity):

- What is the optimal temperature (50°C or 200°C)?
- Is the chicken frozen or alive?
- When is it done?

Wrong algorithms

How to be a millionaire?

```
1. Buy a lottery.
2. Choose numbers.
3. Wait for prize or be sad.
```

Problems (stochastic, not deterministic):

- In most of the cases we won’t be a millionaire.
- Not always works.

Wrong algorithms

How to use a bus?

```
1. Wait for the bus.
2. Get on the bus.
3. Buy a ticket.
4. Sit down.
5. Get out of the bus.
```

Problems (infinite):

- If we are not in a bus stop, bus won’t stop.
- If we are in a building, bus will never arrive.

Pseudocode

Possible instructions / keywords:

- input …
- output …
- if … then … else … endif
- while … do … enddo

- exit
- break
- function, procedure, return, call

Base structures of algorithms

by pseudocode

Sequence:

```
statement1
statement2
statement3
…
```

Selection:

```
if condition  then
   statement1
else
   statement2
endif
…
```

Iteration:

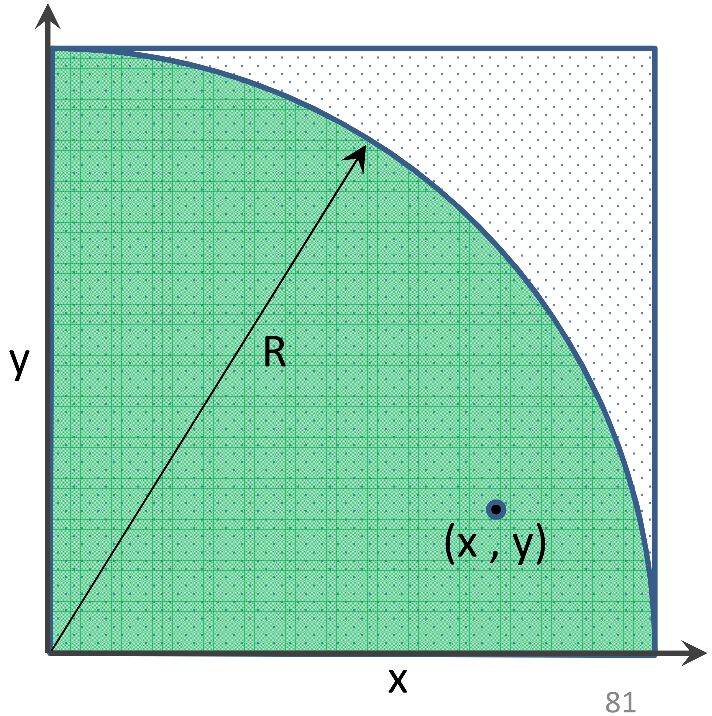
```
while condition do
   statement1
enddo
…
```

Pseudocode example

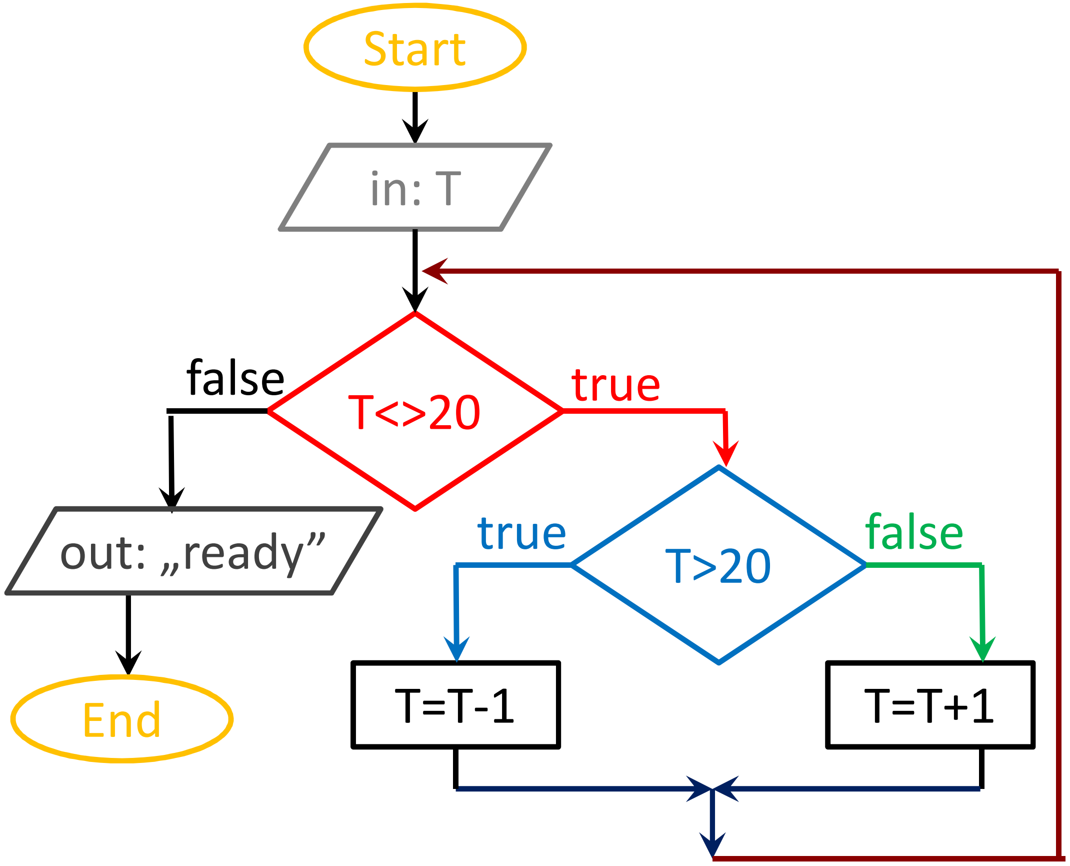
```
input R
i=0
x=0
while x<=R do
   y=0
   while y<=R do
      if x*x+y*y<=R*R then
         i=i+1
      endif
      y=y+1
   enddo
   x=x+1
enddo
output 4*i/(R*R)
```

Approximation of the value of π

Greater R leads to more precise value



Conversion



```
input T
while T<>20 do
   if T>20 then
      T=T-1
   else
      T=T+1
   endif
enddo
output ”ready”
```

Subroutines

Separate unit of algorithms

Tool of **recycling** and **abstraction**

- Recycling: not necessary to type the same code-part at different points of the code. We can teach some activity and then use as a basic instruction. An example of *embedding*.
- Abstraction: not just only one activity, but a set of similar activities, specified by parameters. Realization of *generalization*.

Subroutines

Two sort of subroutines

- **Function**: a set of instructions in order to determine a value somewhere in the code
  E.g. What is the cosine of 45°?
  ```
  x=cos(45)
  ```

- **Procedure**: a set of activities to do something at a given point of the code (no return value)
  E.g. Print your name.
  ```
  output „Imre”
  ```

Procedure example

Definition of procedure to print the * character N-times

name of procedure

(formal) parameter

```
procedure STARS ( N )
  while N>0 do
    output „*”
    N=N-1
  enddo
end procedure
```

Instructions, activity to do

Procedure example

Calling the previous procedure within a code

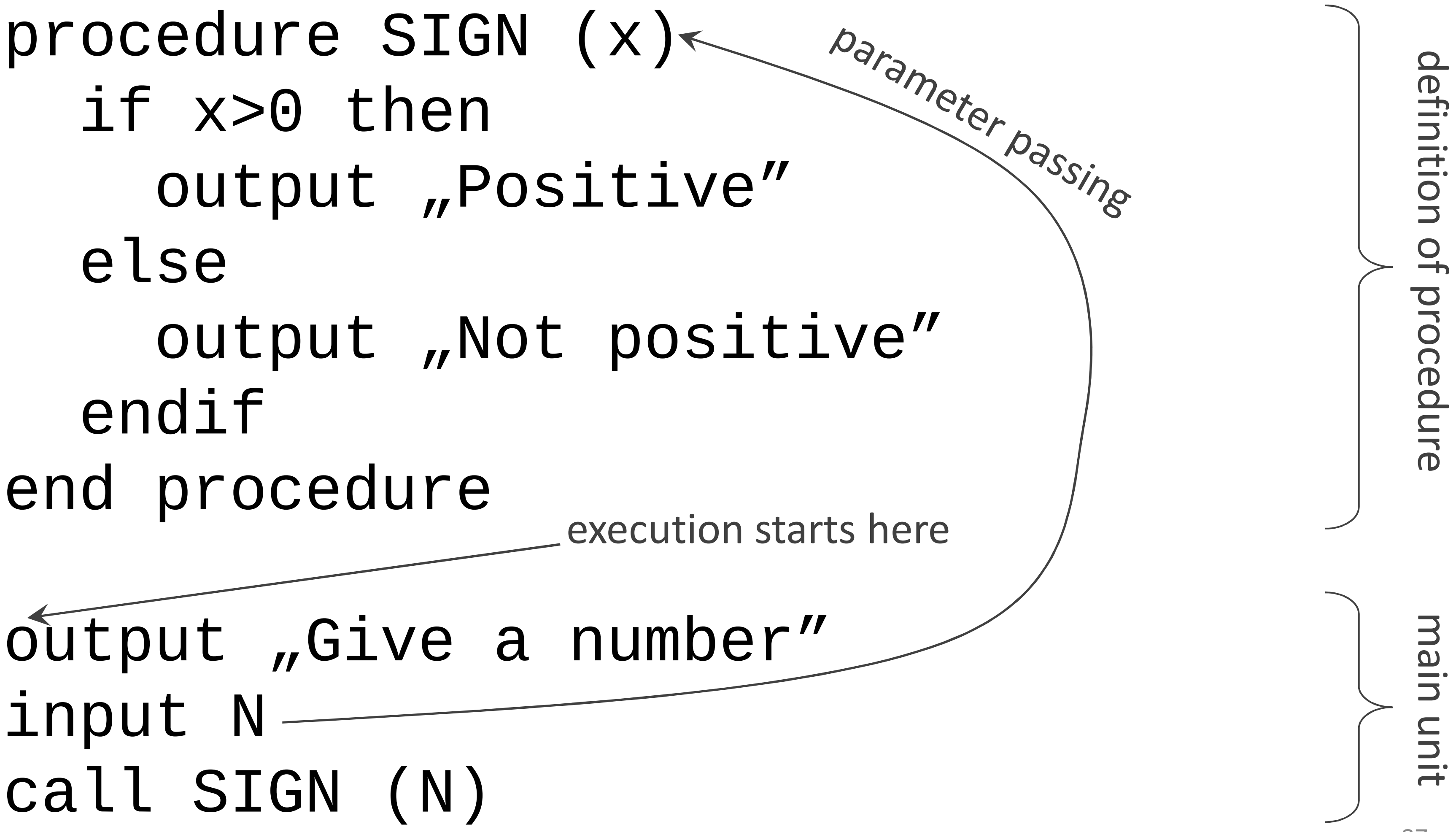
```
output „How many stars you need?”
input S
call STARS (S)
output „I hope, you like it.”
```

(actual) parameter

call of procedure (execution of its instructions)

Procedure example

An other procedure to tell the sign of a number



```
procedure SIGN (x)
  if x>0 then
    output „Positive”
  else
    output „Not positive”
  endif
end procedure

output „Give a number”
input N
call SIGN (N)
```

parameter passing

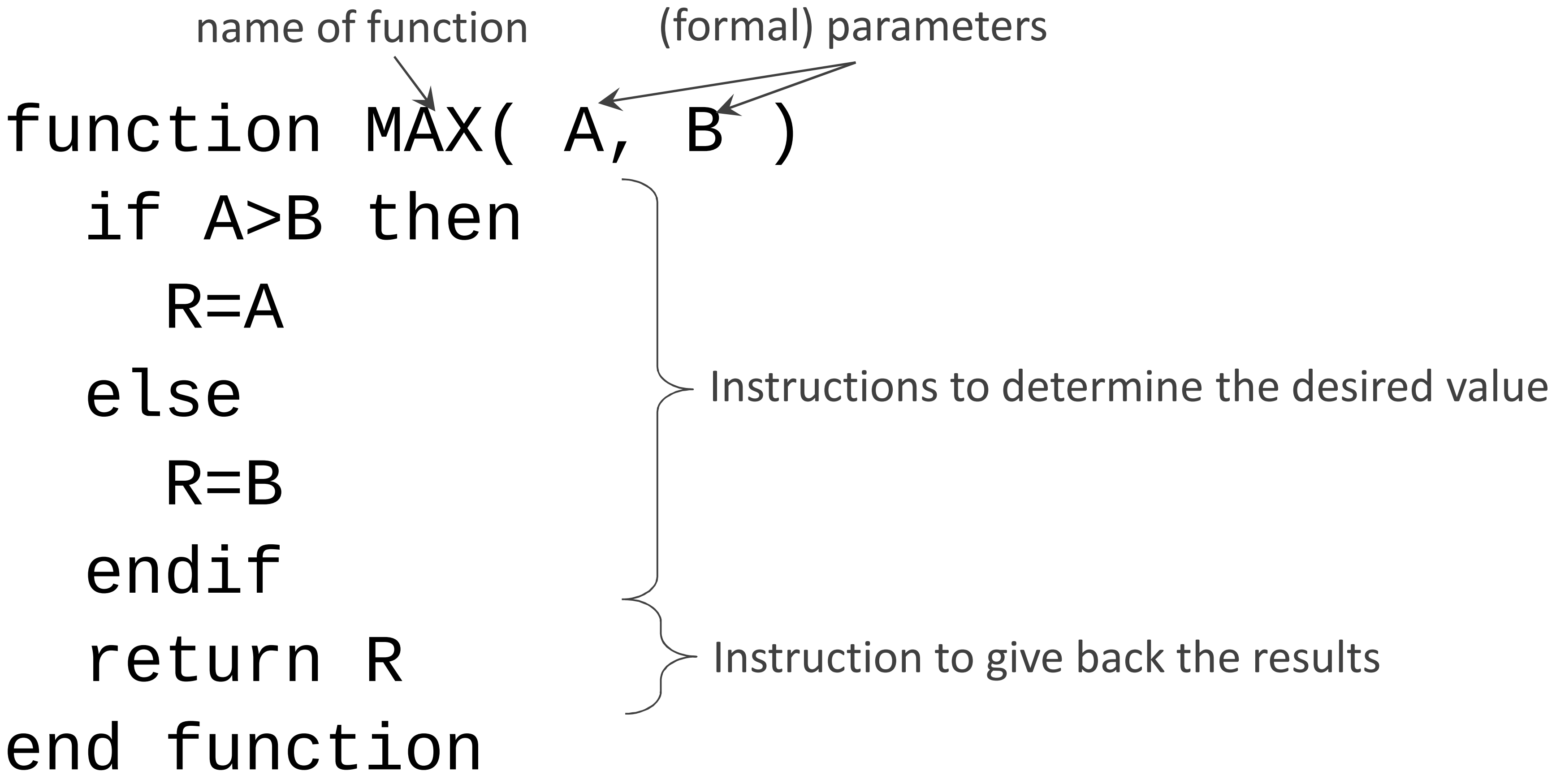
execution starts here

definition of procedure

main unit

Function example

Definition of a function to determine the maximum of two values



name of function

(formal) parameters

```
function MAX( A, B )
  if A>B then
    R=A
  else
    R=B
  endif
  return R
end function
```

Instructions to determine the desired value

Instruction to give back the results

Function example

Calling the previous function within a code

```
output „Give two numbers”
input a, b
c = MAX (a,b)
output „The maximum: ”, c
```

(actual) parameters

call of function (determination of a value)

The returned value is stored in variable c.

Function example

An other function to calculate the absolute value

```
function ABS (x)
  if x<0 then
    x=-1*x
  endif
  return x
end procedure

output „Give a number”
input N
A = abs (N)
output „the absolute value is”, A
```

parameter passing

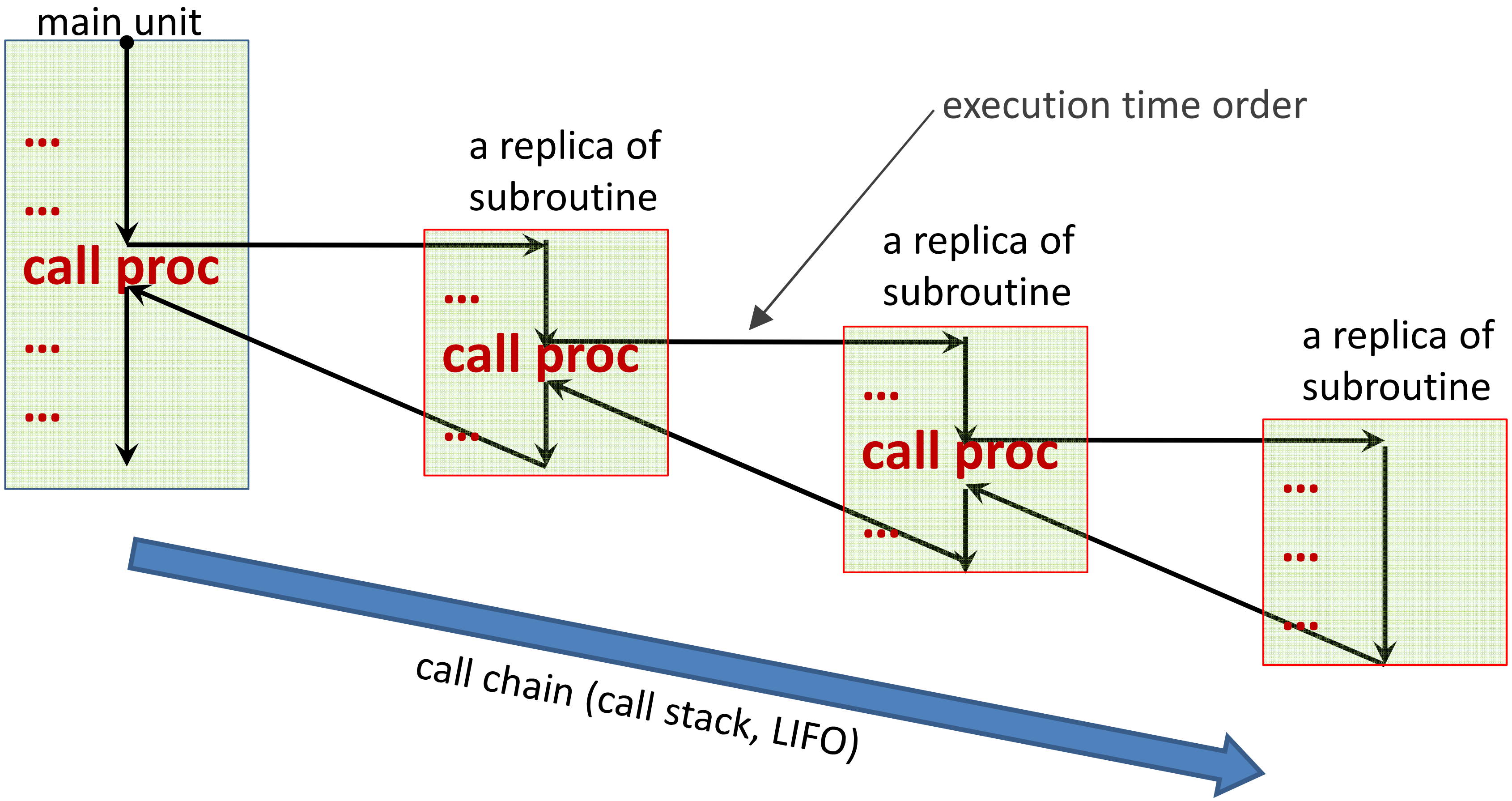
execution starts here

definition of function

main unit

Recursion

- When a subroutine calls itself



Recursion example

```
procedure CONVERT ( N , B )
  if N<>0 then     integer part of the quotient
    call CONVERT ( [N/B] , B)
    output a%b
  endif
  output NEWLINE
end procedure

call CONVERT ( 16 , 8)
call CONVERT ( 19 , 2)
```

- Try to execute the algorithm.
- What is the output?
- How long is the call chain?
- Where the reverse order of digits is coded?

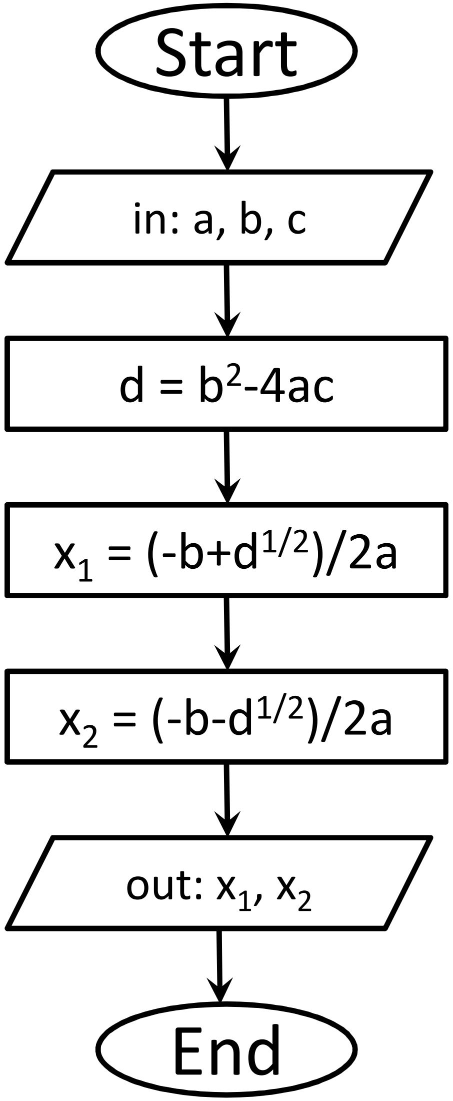
Testing strategy development

Example of testing strategy

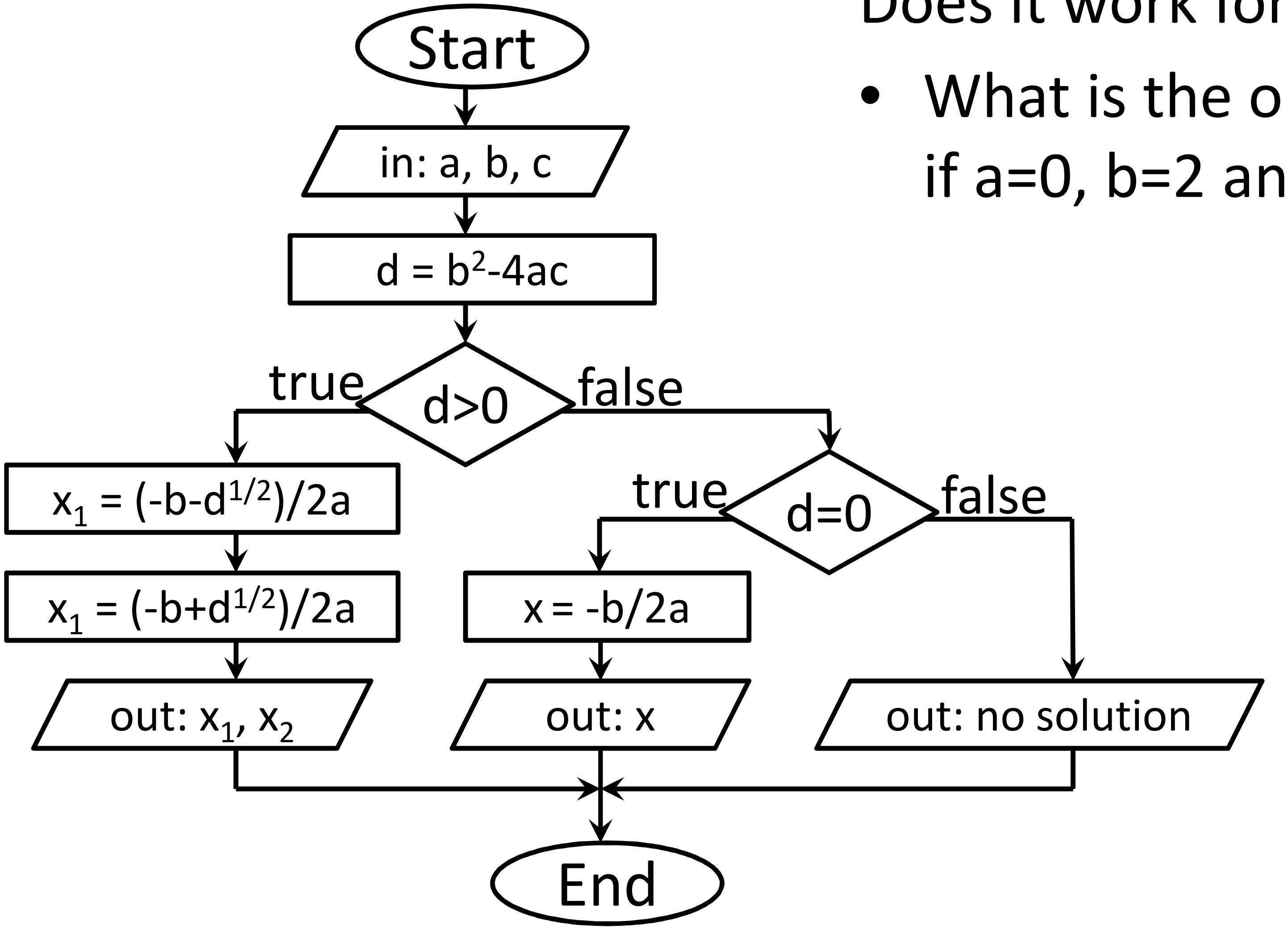
- Solving second degree equation
- General form: $\mathbf{a}x^2 + \mathbf{b}x + \mathbf{c} = 0$
- Input parameters: a, b, c
- Solution: $$\mathbf{x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}}$$

Does it work for all input?

- What is the output if a=1, b=2 and c=1?
- What is the output if a=1, b=2 and c=2?



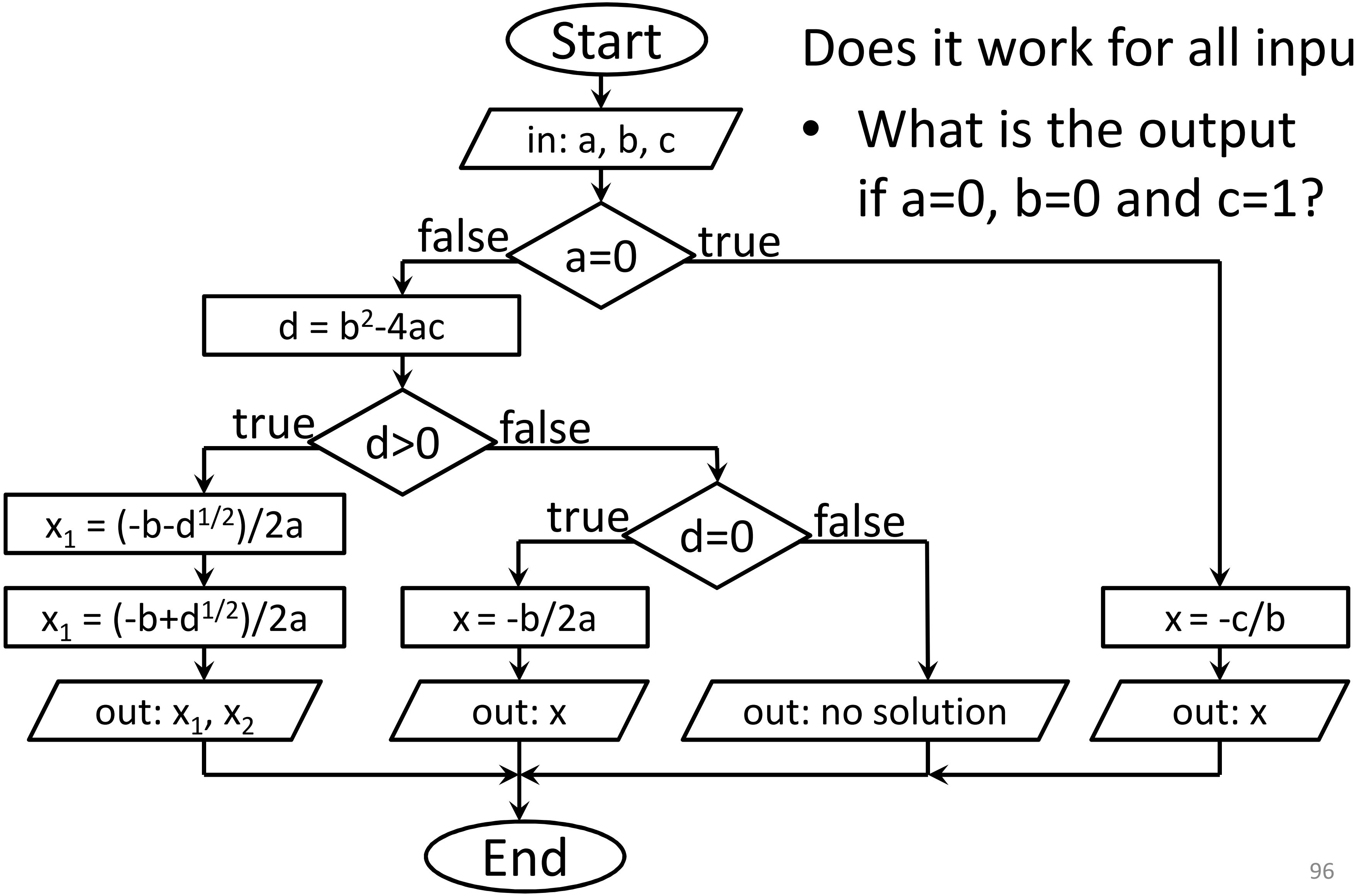
Example of testing strategy



Does it work for all input?

- What is the output if a=0, b=2 and c=6?

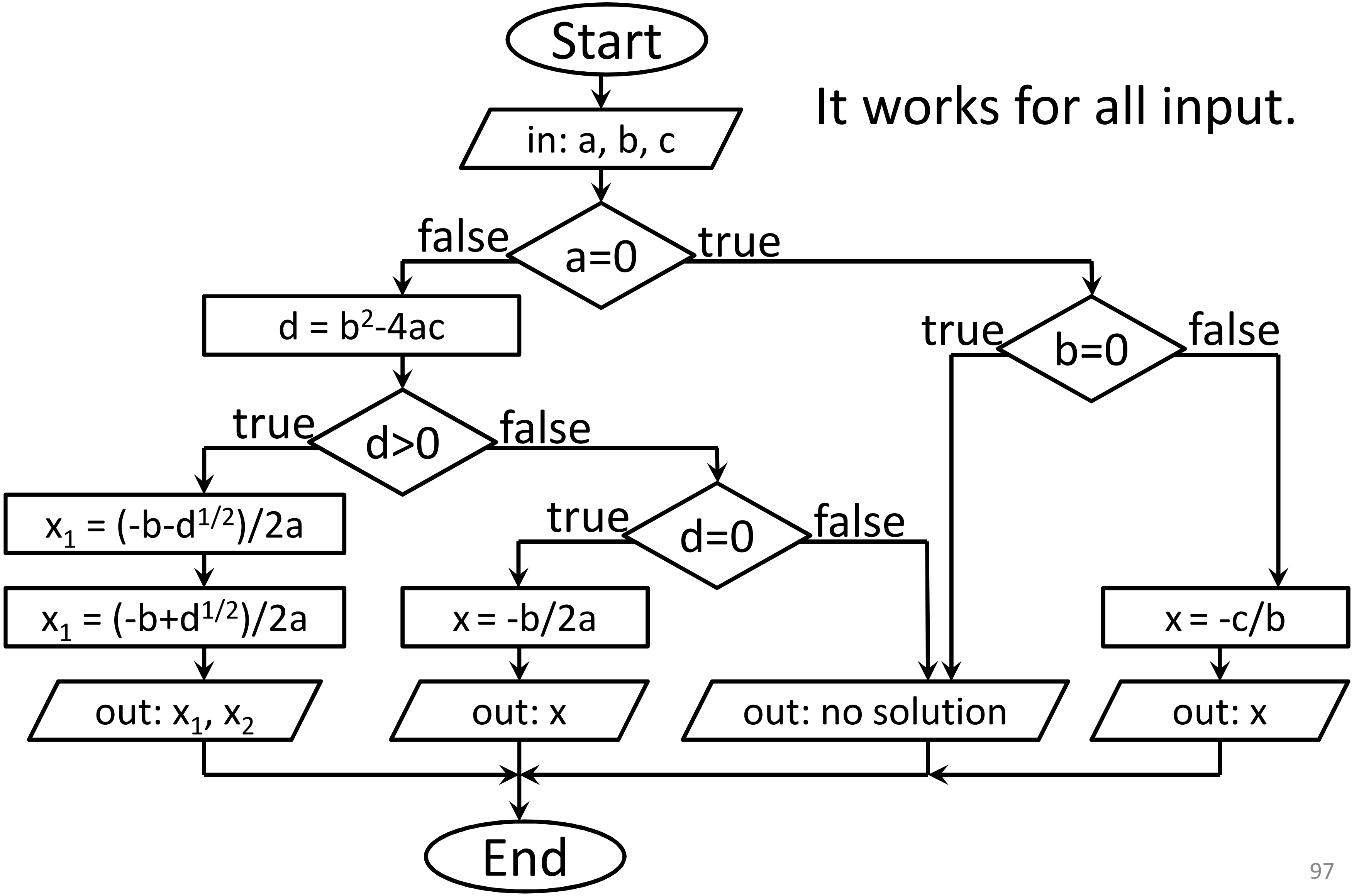
Example of testing strategy



Does it work for all input?

- What is the output if a=0, b=0 and c=1?

Example of testing strategy



It works for all input.

Example of testing strategy

Good solution in pseudocode:

```
input a, b, c
if a=0 then
  if b=0 then
    output error
  else
    x=-c/b
    output x
  endif
else
  d=b*b-4*a*c
  if d>0 then
    x1=(-b+sqrt(d))/(2*a)
    x2=(-b-sqrt(d))/(2*a)
    output x1, x2
  else
    if d=0 then
      x=-b/(2*a)
      output x
    else
      output error
    endif
  endif
endif
```

It works for all input.

To reach this state we have had to test the algorithm with more different input combinations the so called **test cases** and then we have had to modify the algorithm.

We have used testing strategy.

The used testing strategy

a	b	c	reason	OK
3	7	2	general case (not zero, d>0)	✓
0	2	3	**a** is zero (first degree)	✓
2	0	5	**b** is zero (x^2=-c/a)	✓
1	2	0	**c** is zero (x[ax+b]=0)	✓
0	0	1	more zeros (not equation)	✓
3	1	9	**d**<0 (no solution)	✓
2	4	2	**d**=0 (only one solution)	✓
-2	-3	-9	negative inputs	✓
2.3	4.2	0.83	not integer values	✓
0.00001	1000000	1	extreme small/large values	✓

test cases

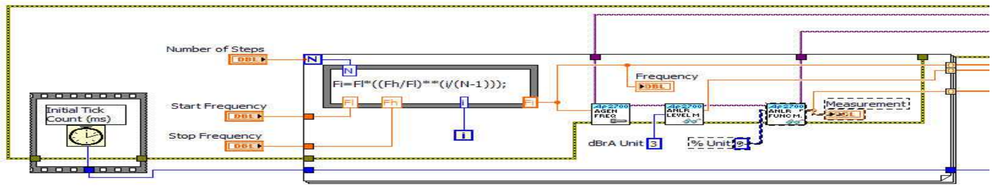
Program coding

Creating source code
in real programming language

Programming levels

High-level programming languages

5GL



4GL SELECT name FROM people WHERE age=20

3GL if (x==0) printf(”zero\n”);

Low-level programming languages

2GL SAVE: STA [$410A, X]

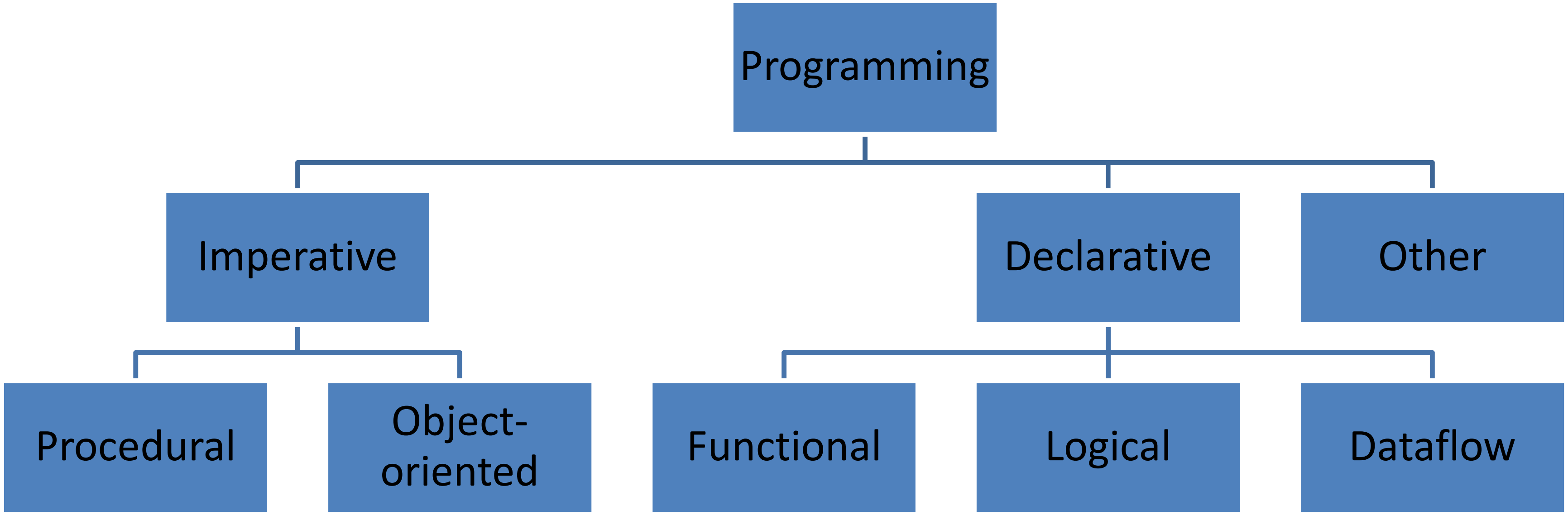
1GL 10010110 11101000 10101101 10000111

time

assembly

machine code

Languages paradigms



Programming

Imperative

Declarative

Other

Procedural

Object-oriented

Functional

Logical

Dataflow

- C
- Pascal
- Fortran

- Java
- C++
- Smalltalk

- R
- LISP
- Haskell

- Prolog

- Labview
- Verilog

Syntax and semantics

Syntax: Formal rules of the program text.

Semantics: Does it describe the desired algorithm?

Example (absolute value):

```
input a
if a>0 then
   a=-1*a
enidf
output a
```

Semantic error

Syntax error

Syntax of programing languages

Fortran:

```
      REAL FUNCTION FAKT(I)
      FAKT=1
      IF (I .EQ. 0 .OR. I .EQ. 1) RETURN
      DO 20 K=2,I
20    FAKT=FAKT*K
      RETURN
      END
```

Pascal:

```
FUNCTION FAKT(I:INTEGER):REAL;
BEGIN
  IF I=0 THEN FAKT:=1
  ELSE FAKT:=FAKT(I-1)*I;
END;
```

C:

```
long fakt(long n){
  if (n<=1) return 1;
  else return n*fakt(n-1);
  }
```

Units and elements of the code

- Character set
- Lexical units
- Syntactic units
- Instructions
- Program units
- Compiling units
- Program

Complexity increase

We use different characters, symbols, special keywords, expressions, and rules in each language.

Interpreter and Compiler

- Processors understand only machine codes
  - High-level source codes need some conversion
- Language implementations use different techniques
  - Compiler
    E.g.: Pascal, C, C++, Labview
  - Interpreter
    E.g.: PHP, JavaScript, Python
  - Combined (bytecode to virtual machine)
    E.g.: Java, C#

Compiler

- A software that creates a so-called **object-code** from the source code
- Compiler makes lexical-, syntactic- and semantic analysis, code generation
  - Source codes have to be syntactically correct
- A co-called **linker** creates executable from object codes, and the **loader** load it to RAM to run
- Compilation once, execution later several times
  - Compilation and execution is separate
- Execution is fast

Interpreter

- Direct execution
  - Analysis and generation at run time
- No object code
- Interpretation of instructions one by one
  - Single instruction as input
- Syntactically incorrect code can be executed
  - Errors may hidden
- Interpreter is needed for any execution
  - Interpretation and execution belong together
- Execution is often slow

Integrated Development Environment

- A (graphical) program to make the software development easy and quick, provides tools/help to the programmer
- **IDE** contains:
  - Language-sensitive editor
  - Compiler/interpreter, linker, loader
  - Debugger
  - Version control tool
  - Project management
  - Simulator
- Examples: Code::Blocks, Dev-C++, NetBeans, Eclipse, MS Visual Studio, Jbuilder, MPLAB, etc.

Data representation, Datatypes

- Every data is stored in the memory in binary
- Different datatypes are used
  - with different representation
  - with different data-domain
  - with different operations
- Most often applied data types:
  - integer (5) 00000000000000000000000000000101
  - float (5.0) 01000000101000000000000000000000
  - char (’5’) 00110101

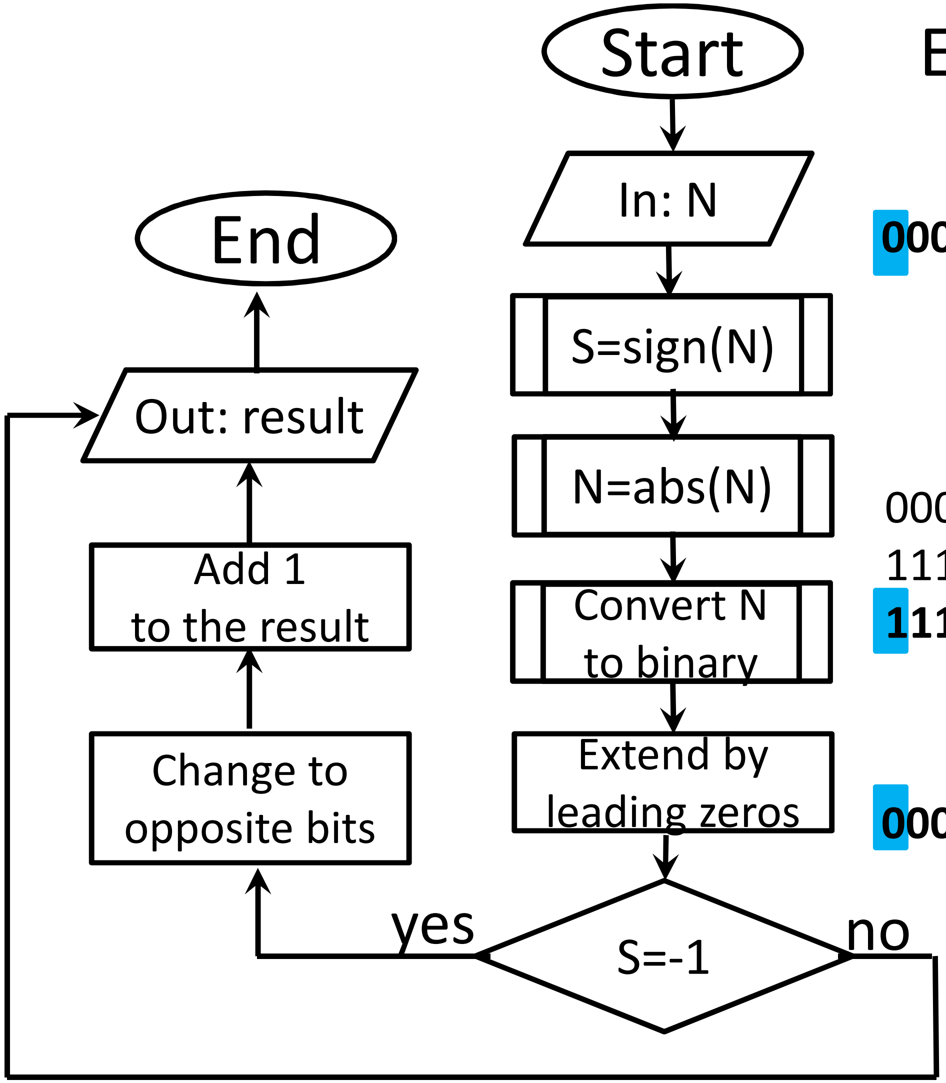
Fixed-point representation

How the computer stores (signed) **integer** numbers?

Steps:

- Store the sign of the given value and convert the absolute value of the integer into binary
- Add leading zeros (if needed) to reach given amount of digits (bits)
- If the sign is -1 (so if the value was negative) then
  - Change every bit to the opposite
  - Add 1 to the result in binary
- The fixed-point representation of the integer is ready

Fixed-point representation



Examples

+195
11000011
00000000 00000000 00000000 11000011

-195
+195
11000011
00000000 00000000 00000000 11000011
11111111 11111111 11111111 00111100
11111111 11111111 11111111 00111101

0
00000000
00000000 00000000 00000000 00000000

MSB = sign bit

Fixed-point representation

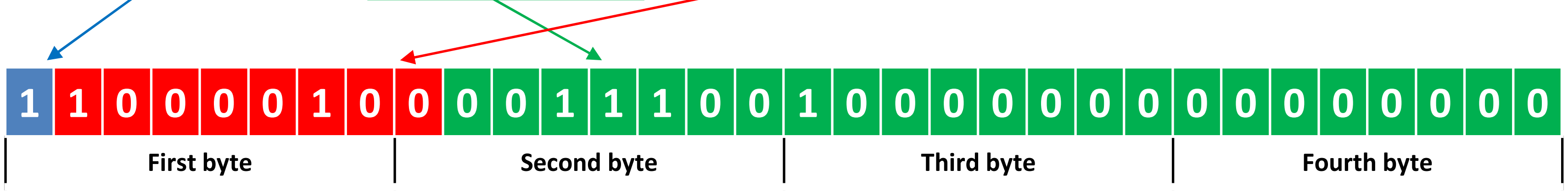
	Representation length	Minimum value	Maximum value
Unsigned	1 byte	0	255
	2 byte	0	65 535
	4 byte	0	4 294 967 295
Signed	1 byte	-128	127
	2 byte	-32 768	32 767
	4 byte	-2 147 483 648	2 147 483 647

Floating point representation

Standardized (IEEE 754) technique to store **real** (fractional, not just integer) numbers in computers

Steps by an example on 4 bytes:

- -19.5625_{10} decimal real
- -10011.1001_2 binary real
- $-1.00111001*2^4$ normal form: Significand * Radix$^{\text{Exponent}}$
- $(-1)^{1} * (1+.00111001) * 2^{131-127}$ $(-1)^S * (1+M) * 2^{K-127}$



1	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Floating point representation

Representation length	4 bytes (single)	8 bytes (double)
# Sign bit	1	1
# Exponent bits	8	11
# Significand bits	23	52
Exponent Bias	127	1023
Minimum absolute value	**$1.175*10^{-38}$**	**$2.225*10^{-308}$**
Maximum absolute value	**$3.403*10^{+38}$**	**$1.798*10^{+308}$**
Accuracy (!)	**~7.2 digits**	**~15.9 digits**

Floating point representation

Not all real numbers in the range can be presented

E.g.: 1.0000000000; 1.0000001192; 1.0000002384; 1.0000003576

Rounding is applied (0.7 → 0.69999998807907…)

Special values:

- `+0.0` (e.g. 1.0-1.0)
- `-0.0` (equal to +0.0; e.g. -1.0*0.0)
- `+Inf` (+∞, e.g. 1.0/0.0)
- `-Inf` (-∞, e.g. -1.0/0.0)
- `NaN` (Not a Number, e.g. +Inf*0.0)

Keywords, identifier, comments

- Keyword

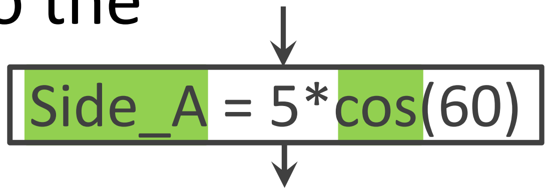
  Sequence of characters with special meaning

  E.g.: if, else, while, do, for, return

- Identifier

  Sequence of characters to give name to the programmer's own tools/objects

  E.g.: `i, Count, var2, abs_val`



Side_A = 5*cos(60)

- Comment

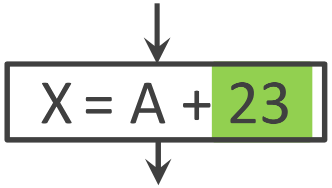
  Text in the code not for the compiler but for the programmer (reader human) as remark

Constants

- Constants (literals) means fix value in the source code that cannot be altered by the program at runtime
- It has type and value
  - The value is defined by itself
  - The type is defined by the form
- Special: Named constant is a fix value with identifier
- Examples

```
-12.34, 5, .5, 5., 0x1F, ’F’, ”F”, ”apple”
```

```
while x>100 do
```



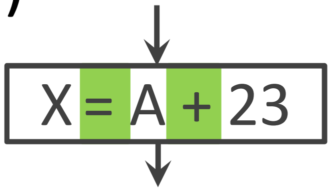
Variables

- A memory location with identifier to store a value
- Most important tool in procedural languages
- Its components
  - Name (identifier)
  - Value (bit series in the RAM)
  - Attributes (type)
  - Address (RAM location)
- Example (in C language)

```
int A = 10;
float Jump1 = 11.5;
```

Operators

- Represents simple operations on data
- Can be unary, binary or ternary
- General groups of operators
  - Arithmetic (E.g.: `+`, `-`, `*`, `/`, `%`)
  - Comparison (E.g.: `>`, `<`, `==`, `>=`, `<=`, `!=`)
  - Logical (E.g.: `&&`, `||`, `!`)
  - Bitwise (E.g.: `&`, `|`, `^`, `~`, `<<`, `>>`)
  - Assignment (E.g.: `=`, `+=`, `*=`)
  - Other (E.g.: `*`, `&`, `? :`, `.`, `->`)



Expressions

- Operators & Operands & Parentheses
- Operand can be: constant, variable, function call
- An expression has type and value (evaluation)
- Form can be
  - Infix (preference/strength is necessary)
    E.g.: 4 + 3 * 2
  - Prefix
    E.g.: + * 3 2 4
  - Postfix
    E.g.: 4 3 2 * +

Instructions

Unit of programs, that can be grouped as

- Declaration
- Assignment
- Conditional statement
  - 2 branch
  - More branch
- Iteration
  - Conditional loop
  - Counted loop
- Other

not executable

executable

Declaration, Assignment

Declaration

- Associate identifier and type
- (Sometimes) initialization of variable
- `int i = 0;`
- `float Weight;`

Assignment

- Giving value to a variable
- `i = 6;`
- `Weight = 80.3 * i;`

Conditional statement

Choosing from 2 execution branch

- Two separate instruction block
- Skip or execute an instruction block
-
  ```
  if(N<0.0)  S=1;
  else       S=0;
  ```

Selecting from several execution branch

-
  ```
  switch (i){
  case 1:  X=1;  break;
  case 2:  X=10; break;
  default: X=100;
  }
  ```

Iteration

Repetition of instructions, activities several times

From operational point of view limiting cases

- Empty loop (the body/core never executed)
- Infinite loop (never stops, semantic error)

Types of iterations

- Conditional loop
  - Pre-condition
  - Post-condition
- Counted loop
- Other (Infinite, Combined)

Pre-conditional loop

The **head** contains a **condition**

Semantics

1. Evaluation of condition
2. If it is true, **body** is executed and evaluate again (1.)
   Else loop ends, go to next instruction behind the loop

It can be empty loop
  if condition is false initially

```
while (x<2){
  i=i+1;
  x=x-2;
}
```

Body must change the condition →

Post-conditional loop

The **end** contains the **condition**

Semantics

1. Execute the **body** once
2. Evaluation of condition
3. If it is true (false), execute again the body (go Step 1.)
   Else loop ends, go to next instruction behind the loop

It cannot be empty loop

body is executed at least once

```
do{
  i=i+1;
  x=x-2;
}
while (x<2);
```

C programming language

Developed: Dennis Ritchie, Bell Labs, 1972

Standards: ANSI C (1989), ISO C99 (1999), ISOC11 (2011)

One of the most popular and widely used language

Procedural (Imperative) paradigm

Close relation to hardware architectures

Purely compiled

Platform-independent

Efficient (fast programs)

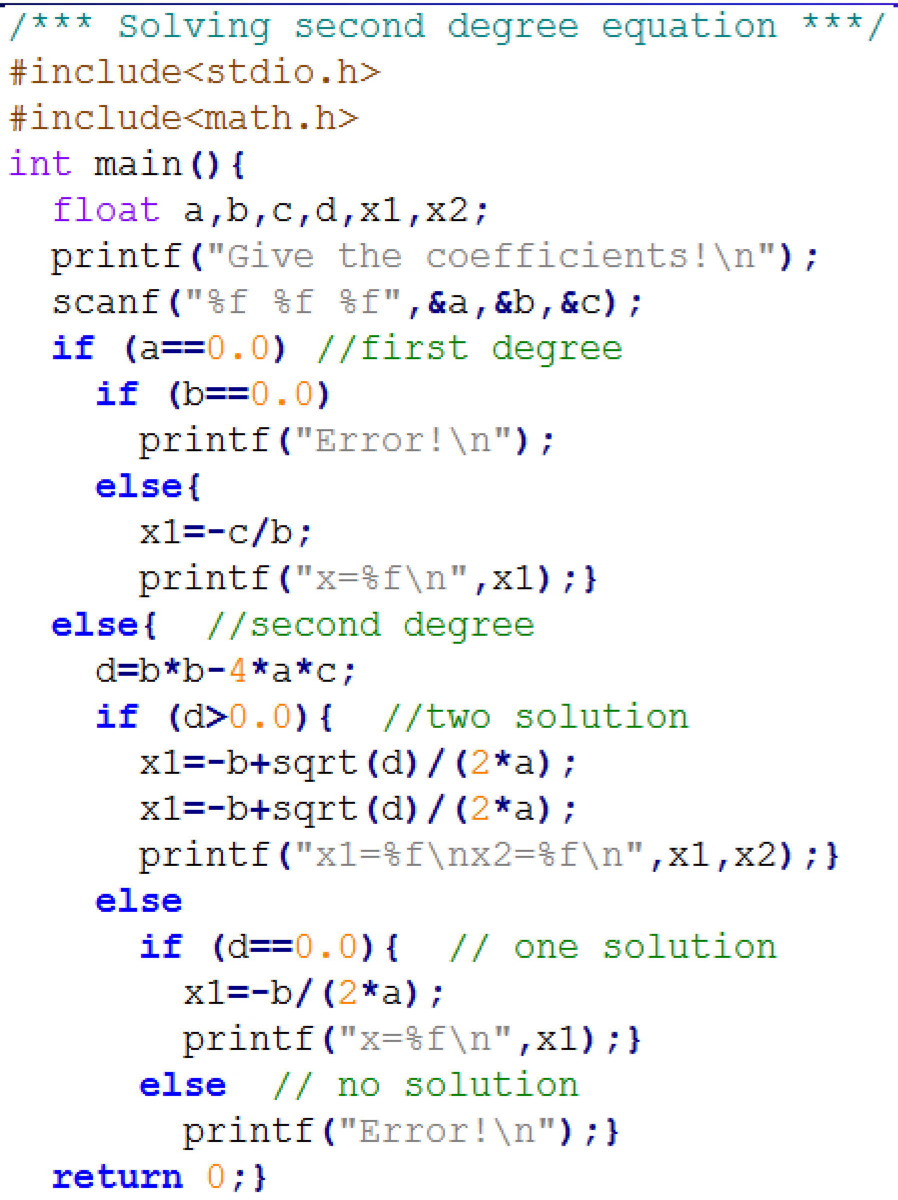
C influenced: C++, Java, PHP, Python, …

C programming language

Some main moment (strongly reduced):

- Comment: /* something */, // something
- Datatypes: int, float, char, …
- Constants: 21, 34.5, ’A’, ”alma”, …
- Operators: +, -, *, /, %, =, ==, >=, <=, !=, &&, ||, …
- Branching: if-else, switch-case
- Iterations: while, for, do-while
- Input/output: printf(), scanf() (built in subroutines)
- etc. …

Examle C source code



```
/*** Solving second degree equation ***/
#include<stdio.h>
#include<math.h>
int main(){
  float a,b,c,d,x1,x2;
  printf("Give the coefficients!\n");
  scanf("%f %f %f",&a,&b,&c);
  if (a==0.0) //first degree
    if (b==0.0)
      printf("Error!\n");
    else{
      x1=-c/b;
      printf("x=%f\n",x1);}
  else{  //second degree
    d=b*b-4*a*c;
    if (d>0.0){  //two solution
      x1=-b+sqrt(d)/(2*a);
      x1=-b+sqrt(d)/(2*a);
      printf("x1=%f\nx2=%f\n",x1,x2);}
    else
      if (d==0.0){  // one solution
        x1=-b/(2*a);
        printf("x=%f\n",x1);}
      else  // no solution
        printf("Error!\n");}
  return 0;}
```

Documentation & Maintenance

Documentation

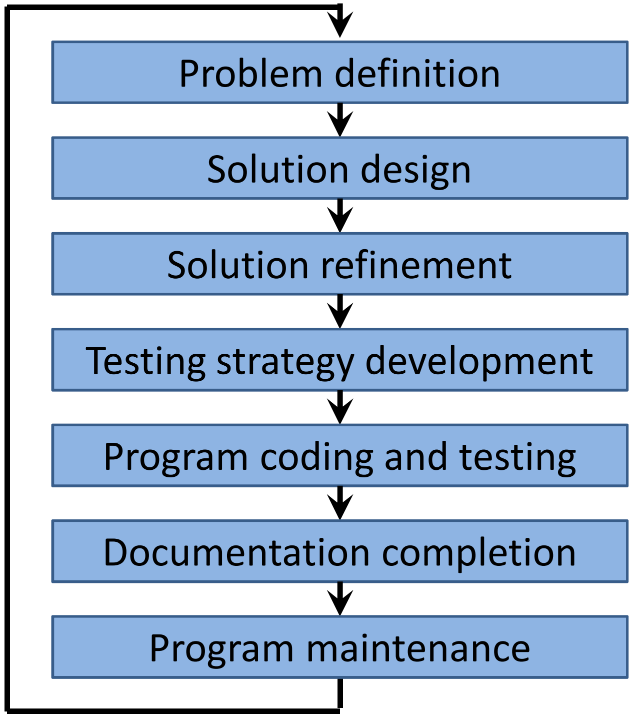
Complete the documentation:

- Always document everything during the program development.
- What is the solution method?
- What are the solved subproblems?
- What are the necessary inputs and the output?
- How does the implemented algorithm work?
- What are the meaning of the variables? (comments)
- How to use the program? (user manual)
- What are the discovered errors and their solutions.

Maintenance

Maintenance the program:

- If the users need correction, extension or changes, developer must update the application.
- Make documentation about all changes.
- If too much change is necessary: Start again from the beginning **Software life cycle**


Problem definition
Solution design
Solution refinement
Testing strategy development
Program coding and testing
Documentation completion
Program maintenance