VILNIUS GEDIMINAS TECHNICAL UNIVERSITY
DEPARTMENT OF INFORMATION SYSTEMS

B. Pliuskuvienė

PROGRAMMING 1

Teaching material

2007

Table of content

1. Binary logic

1.1. Scales of notation

Scale of notation is the whole of ways and means that allows to write down or to present numbers differently. The meaning of the figure depends on its position in the number.

The basis of the scale of notation is considered to be a number, which shows how many times the meaning of one and the same figure, increases or decreases, when it is moved into one of the positions that is close to it. Any of the numbers can be presented by this formula:

$$N_q = a_n \times q^n + a_{n-1} \times q^{n-1} + a_{n-2} \times q^{n-2} + \ldots\ldots + a_1 \times q^1 + a_0 \times q^0;$$

Here: N_q - the given number;

q - the basis of notation scale;

n+1 – the amount of figures in the number as the least right segment of the number is not marked by a unit but by zero.

In case a number is fractional its shorter part is marked by negatives n.

$$N_q = a_n \times q^n + a_{n-1} \times q^{n-1} + a_{n-2} \times q^{n-2} + \ldots\ldots + a_1 \times q^1 + a_0 \times q^0 + a_{-1} \times q^{-1} + \ldots\ldots;$$

The meaning of each figure in the fractional part is q times less than the same before going figure:

$$a_1 q^{-1} = \frac{a_1}{q^1}; \ldots$$

At present in computer science it is impossible to do without three scales of notation: decimal (q= 10), binary (q= 2) and hexadecimal (q= 16).

The decimal notation is used as generally accepted and acknowledged system, which has ten different symbols to mark figures.

The binary notation system has only two symbols. They are: 0 and 1. It is the basis of electronic circuitries as only such system allows defining the state of electronic circuitries with the help of meanings of figures. The existing of electricity charge in certain points of circuitry can be considered a one and absence of it is zero (or vice versa).

In order a man could understand the numbers presented for computer processing and received results and in order the computer could do such processing it is necessary to be able to move numbers from decimal notation to binary notation and by vice versa from a binary notation to a decimal notation. Such number transferring isn't easy and convenient therefore the intermediate hexadecimal notation is being used and hexadecimal symbols can directly be transferred into the binary notation.

Table 1.

q_{16}	q_2	q_{10}
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

The direct transferring of a hexadecimal number into the binary number means that every figure of hexadecimal number, for e.g., 1F3 will be expressed by the corresponding binary tetrad (the group of four binary figures):

$$\begin{array}{ccc} 1 & F & 3 \\ \Downarrow & \Downarrow & \Downarrow \\ 0001 & 1111 & 0011, \end{array}$$

or $$1F3_{16} = 000111110011_2 = 111110011_2.$$

The decimal number 6783 is transferred into the hexadecimal with the help of simple division and subtraction:

$$6783 : 16 = 423 \text{ and residual } 15 = F$$
$$423 : 16 = 26 \text{ and residual } 7$$
$$26 : 16 = 1 \text{ and residual } 10 = A.$$

It means that:

$$6783_{10} = 1A7F_{16} = 1101001111111_2.$$

And vice versa - the hexadecimal number 1A7F is transferred into decimal by the same simple succession of multiplication and addition:

$$1 \times 16 = 16,$$
$$16 + A = 26,$$
$$26 \times 16 = 416,$$
$$416 + 7 = 423,$$
$$423 \times 16 = 6768,$$
$$6768 + F = 6783_{10}.$$

The octal notation is analogical to hexadecimal notation and it can also helps to directly transfer numbers from decimal notation into binary or from binary into decimal notation. The figures of octal notation are transferred into binary notation by tetrads (in groups of three binary figures):

Table 2.

q_8	0	1	2	3	4	5	6	7	10	11
q_2	000	001	010	011	100	101	110	111	001000	001001
q_{10}	0	1	2	3	4	5	6	7	8	9

If given octal number is 763 the binary number is going to be:

$$\begin{array}{ccc} 7 & 6 & 3 \\ \Downarrow & \Downarrow & \Downarrow \\ 111 & 110 & 011, \end{array}$$

Then $763_8 = 111110011_2.$

It is easy to notice that $1F3_{16}$ and 763_8 are the same binary number 111110011_2, so $1F3_{16} = 763_8$.

Having transmitted $1F3_{16}$ and 763_8 into the decimal notation

$$1 \times 16 = 16 \qquad\qquad 7 \times 8 = 56$$
$$16 + F = 31 \qquad\qquad 56 + 6 = 62$$
$$31 \times 16 = 496 \qquad\qquad 62 \times 8 = 496$$
$$496 + 7 = 499_{10} \qquad\qquad 496 + 3 = 499_{10}$$

In spite of the notation in which the number is presented its size remains the same:

$$499_{10} = 1F3_{16} = 763_8 = 111110011_2.$$

Fractions from decimal notation into binary notation are transferred in some other way in comparison with whole numbers. Let's say the given fraction is $0.\ 257_{10}$:

$$.257 \times 16 = 4.112 = 4 + .112,$$
$$.112 \times 16 = 1.792 = 1 + .792,$$
$$.792 \times 16 = 12.672 = C + .672,$$
$$.6727 \times 16 = 10.752 = A + .752,$$
$$.752 \times 16 = 12.032 = C + .032,$$
$$.257_{10} = .41CAC\ldots._{16} = .0100\ 0001\ 1100\ 1010\ 1100_2.$$

The obtained hexadecimal fraction $.41CAC_{16}$ will be transferred into the decimal fraction:

$$12 : 16 = .75,$$
$$10 + .75 = 10.75,$$

$$10.75 : 16 = .671875,$$
$$12 + .671875 = 12.671875,$$
$$12.671875 : 16 = .791992188,$$
$$1 + .791992188 = 1.791992188,$$
$$1.791992188 : 16 = .111999512,$$
$$4 + .111999512 = 4.111999512,$$
$$4.111999512 :16 = .25699997 \approx .257_{10}.$$

When transferring decimal fractions into hexadecimal the operation can be very long (until the fraction part will be equal to zero) therefore mostly it is necessary to finish the transferring until the desired precise is achieved.

When transferring hexadecimal fractions into decimal mostly it is necessary to approximate fractions because the transferring can be limitless.

When transferring numbers, made of the whole and fractional parts, from one scale of notation into another, both the fractional part and whole part are being transferred separately.

The transferring of numbers makes only a little part of the whole manipulation of digital data.

The binary arithmetic and Boolean algebra play the most important role here.

1.2. Binary arithmetic

The basis of the binary arithmetic is the operations of additions, subtraction and multiplication shown in the table 3.

Table 3.

Addition	**Subtraction**	**Multiplication**
0 + 0 = 0	0 - 0 = 0	0 × 0 = 0
1 + 0 = 1	1 - 0 = 1	1 × 0 = 0
0 + 1 = 1	1 – 1 = 0	0 × 1 = 0
1 + 1 = 10	10 - 1 = 1	1 × 1 = 1

While adding numbers the transferring of one is going to the bigger (the left) when 1 + 1 = 10 are being added.

$$(1011 + 1010 = 10101)_2,\ (B + A = 15)_{16},\ (11 + 10 = 21)_{10}.$$

There shouldn't be any doubts that B + A = 15, as

Table 4.

q_{16}	…	A	B	C	D	E	F	10	11	12	13	14	15	…
q_{10}	…	10	11	12	13	14	15	16	17	18	19	20	21	…

When subtracting if you have to transfer (borrow) the unit of the older segment, two units (i.e.10) appear in the latter segment, and one unit appears in each intermediate negative segment, as another unit goes to the smallest segment into which we borrow. Therefore before subtracting two binary units appear in the latter segment.

$$(16 - 9 = D)_{16},\quad (10110 - 1001 = 1101)_2,\quad (22 - 9 = 13)_{10}.$$

Operations of binary numbers multiplications and division are analogical to the operations of decimal numbers multiplication and division.

```
   1101
 x  101
 ------
   1101
  0000
 1101
-------
1000001
```

$(D \times 5 = 41)_{16}$

$(13 \times 5 = 65)_{10}$

$65_{10} = 41_{16}$

```
_1000110 | 111
 111     | 1010
   _111
    111
 -------
      0
```

$(46 : 7 = A)_{16}$

$(70 : 7 = 10)_{10}$

$10_{10} = A_{16}$

After the first operation of subtraction a unit from the fifth dividend segment is transferred, but 11 is not enough to divide by 111. Therefore we write zero into the second segment of the quotient and we take on more unit from the sixth divided segment. We directly transfer the rest zero from the seventh dividend segment into the quotient, i.e. all operations are analogical to operations of decimal numbers subtraction.

1.3. Logical operations

From 1.1. and 1.2. sections it is obvious that four binary segments or tetrad is enough to write down any hexadecimal figure. But it is impossible to write down other signs in the tetrad because it wouldn't be able distinguish them from the hexadecimal figures. Therefore one more tetrad is needed for writing down letters, mathematical symbols and other sings. So, eight binary segments, called **bits**, are used for writing down numbers and other symbols, and the whole structure of eight bits is called **a byte**. Any 0 and 1 combinations from 00000000 to 11111111, i.e. from 00_{16} to FF_{16} or $1+255 = 256_{10}$ in byte, because $FF_{16} = 255_{10}$.

It is evident that the binary arithmetic doesn't suit for the processing of non-figure symbols. For that case logical operations are used. The main categories of the logical operations are: the truth, right, which are marked by 1, and unreal, the false, lie are marked by number 0.

Logical operations are called OR (logical addition, disjunction), AND (logical multiplication, conjunction), XOR (exclusive or), NOT (logical negative, inversion).

In this section we will analyze only four logical operations though there may exist more such operations (nonexclusive, implication and others).

Though we stressed that it is not enough to use the structure of four binary sings (tetrad) for synonymous determining of binary symbols, this can be quite sufficient for the essential analysis of logical operations.

Logical operations are marked by symbols:

$$\text{OR} - \vee,\ \text{AND} - \wedge,\ \text{XOR} - \oplus,\ \text{NOT} - \neg.$$

The essence of logical operations is showed in the following table 5:

Table 5.

The name of operations	**OR**	**AND**	**XOR**	**NOT**
Markings	$\vee$	$\wedge$	$\oplus$	$\neg$
Operations	0101 $\vee$1001 1101	0101 $\wedge$1100 0100	1010 $\oplus$1100 0110	$\neg$0101 1010

It is common to all logical operations that the bits, which are in the equal distance from the beginning of the number, are compared among themselves and the result of their comparing is always 1 or 0 and it never (as in the binary arithmetic) goes up into higher segments. That is why the results of logical operation may be only true (1) or false (0) and may not be any other.

The internecine differences of the operations must be noted.

In the logical operation OR, also called addition or disjunction, the comparable result of the bits in all cases is 1, except these cases, when zeros are compared except the case when 1s are compared among themselves.

In the logical operation AND, also called logical multiplication or conjunction, the results are equal to zero in all cases when comparing bites, except when ones are being compared. In the latter case the result is 1.

In the logical operation XOR different bits, having 1, as their result, and homogenous having a zero, are compared. As it has already been mentioned, this operation otherwise is called as non-equivalence.

The logical NOT operates with one binary symbol, transferring it into the so-called inverse symbols. It means, ones of the result in the analogical places are changed into zeros and zeros – into ones.

The operations given in the table 5 use four figure binary numbers for visual clearness. In practice such operations may be performed with the binary numbers or symbols of any size, having the same and different amount of the bits.

The possibility to know about the binary structure and to change them accordingly makes the essence of the logical operations. It is especially important as it is impossible to see directly and visually the binary structures being in the computer's memory. It is when the so-called pattern, i.e. the binary structure known in advance is presented.

Further we will give the examples of using logical operations. Let's write down in one byte the control information of the certain processes. It means that having switched, i.e. having put down the ones into the certain bit segments this bit participates in the process, and if the bit is the zero, then this segment doesn't participate in the concrete process. So, as it has been mentioned earlier, the byte may have 256 switching positions of the control processes all, from the position HEX '00' to the position HEX 'FF'. The letters HEX mean the hexadecimal number. The same hexadecimal number is shown in the apostrophes.

The same form of the data presentation is used for all the operations.

The byte before the operation	HEX '00'	0000 0000
		?
The byte – the pattern	HEX '00'	0000 0000
The byte after the operation	HEX '00'	0000 0000

Here the hexadecimal numbers are given: the content of the byte before the operation, the byte – the pattern and the content of the byte after the operation. Besides, every number is given in the binary form, which helps visually to conceive the process of the decisive problem. The byte – the pattern is the data selected so that it enables to solve the concrete problem under a certain primary state of byte and using the logical operation whose marking symbol is written instead of the symbol "?".

Operations OR

1) Change the first bit content into one (the position is "switched") in such a way that the bits would not change.

The byte before the operation	HEX '0F'	0000 1111
		v
The byte – the pattern	HEX '80'	1000 0000
The byte after the operation	HEX '8F'	1000 1111

2) Switch two switches HEX '28', though only one of them was switched before the operation.

The byte before the operation	HEX '20'	0010 0000
		v
The byte – the pattern	HEX '08'	0000 1000
The byte after the operation	HEX '28'	0010 1000

3) Set the content only of the lower (right) byte tetrad.

The byte before the operation	HEX '49'	0100 1001
		v
The byte – the pattern	HEX 'F0'	1111 0000
The byte after the operation	HEX 'F9'	1111 1001

Operations AND

1) Turn off the switches HEX '26'.

The byte before the operation	HEX 'FF'	1111 1111
		^
The byte – the pattern	HEX 'D9'	1101 1001
The byte after the operation	HEX 'D9'	1101 1001

2) Turn all the switches, which were switched.

The byte before the operation	HEX '92'	1001 0010
		^
The byte – the pattern	HEX '00'	0000 0000
The byte after the operation	HEX '00'	0000 0000

Operations XOR

1) Let's say we exclusive XOR operation.

The byte before the operation	HEX '7D'	0111 1101
		⊕
The byte – the pattern	HEX 'FF'	1111 1111
The byte after the operation	HEX '82'	1000 0010

We can use this operation with the given pattern HEX 'FF' when we want to know if the byte hadn't been a zero: HEX 'FF' before the operation. With the help of this operation we can also change the content of the first (primary) byte vice verse, i.e. negative bits are changed into ones and ones into zeros.

2) Switch the higher byte state into the opposite state, leaving the lower tetrad not changed.

The byte before the operation	HEX '96'	1001 0110
		⊕
The byte – the pattern	HEX 'F0'	1111 0000
The byte after the operation	HEX '66'	0110 0110

The usage of the logical operations in data searching.

Let's have a certain list of persons who are characterized by the content of the byte:

1 bit	Man / not	1 / 0
2 bit	Married / not	1 / 0
3 bit	25 years old / older	1 / 0
4 bit	University education / not	1 / 0
5 bit	Born in Lithuania / not	1 / 0
6 bit	Hasn't got a house / has got	0 / 1
7 bit	Hasn't got a driving license / has got	0 / 1
8 bit	Lithuanian / not	1 / 0

Having in mind the essence of each bit in every byte of the list, we can choose properly the byte – the pattern, in order to know the number of persons in the list, who have necessary characteristics. If we choose the byte – pattern HEX '80', so logically multiplying (the operation AND - $\wedge$) it from one byte in the list, we will obtain the result in the byte HEX '80'. It means that the man was characterized in the same byte of the list. If a woman is described in the byte of the list, the result of the byte will be HEX '00'. Logically multiplying every byte of the list to the pattern and after every multiplication operation the result of the byte to the binary word a = HEX '0000', having finished operations with the list we will obtain the number in the word a, which showing how many men are there in the list.

While similarly doing we may group all the persons of the list into 256 lists according to their characteristics and calculate how many persons are there in each of those lists.

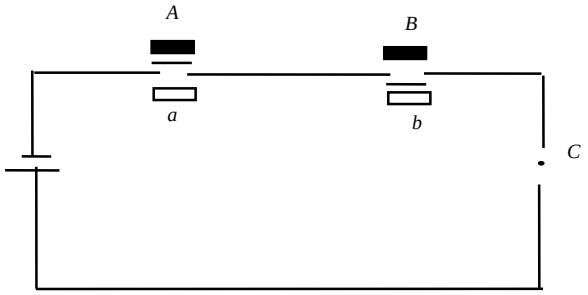
If the byte of people's list is HEX '00', it means, that in this byte the following are being characterized: a woman, single, elder than 25, without higher education, born not in Lithuania, not owning house, having no driving license, not a Lithuanian. If the byte of the list is HEX 'FF', it means, that a person characterized in this byte has such characteristics: a man, married, 25 years old, having higher education, born in Lithuania, having his own house and a driving license, a Lithuanian. In order to know all mentioned personal characteristics it is necessary to present the byte – the pattern HEX 'FF', and to perform the logical operation AND. It is evident that having presented in byte – the pattern of any other combination, we will establish whether the analyzed person has such characteristics or only a part of them.

The place of two bytes is chosen for summing up the results of the logical multiplication for we could store the bigger analyzed person's number in it, which are included in one byte than 255. Two bytes will contain the number $FFFF_{16}$, i.e. 65535_{10}.

While using the logical arithmetic operations suitably choosing the rows of the operation processes and various patterns, i.e. the parameters of the operation processes and variables, we practically have unlimited possibilities of the manipulation by the binary data. Therefore modern computers that are using the binary logic have also the same possibilities.

1.4. Binary operations in discretionary elements of computer

In this chapter the principal possibility of arithmetical and logical process of the operations in computer is shown. Attention is not paid to the demonstration of the operations towards physical realization of the computer. The elements of the computer are always changing and improving while the mathematics of the binary operations remains the same. In order to have the demonstration of the binary operations in discretionary computer element more visual and easier perceptible, that element is represented by the scheme, which is made by the chain of electricity, consisting of magnetic (*a* and *b*) and electromagnetic (*A* and *B*) mechanisms, of contacts and the source of the electricity current and of going out or the point *C* (picture 1.1.) of the operation result.



Picture 1.1. Discrete computer element

The binary information for the scheme is presented by existence or non-existence of the electricity current in the points *A* and *B* then magnets *a* and *b* act on the contacts, and if the current exists, stronger electromagnets act on the contacts at the point *A* and *B.* If two contacts conjoin in the point *C* the tension exists of the result (outlet) and if at least one contact is not joined, there is no tension in the point *C*. The existence of the tension in the points *A*, *B* and *C* means the one (1), and non-existence – the zero (0).

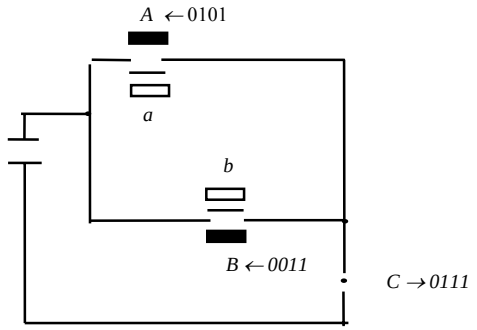
Let's present the data into the points *A* and *B* by the binary combinations:

$$A \leftarrow 0101, B \leftarrow 0011,$$

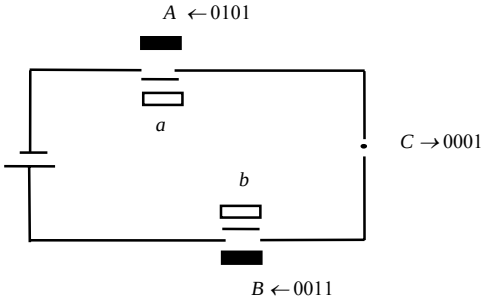
And the result information shown in the point *C* will depend on the scheme, which realizes the operations $\vee$, $\wedge$, $\neg$ and $\oplus$.

The primary results and the results of their binary interaction in the schemes are always presented as being in the same distance from the start of all three binary numbers.

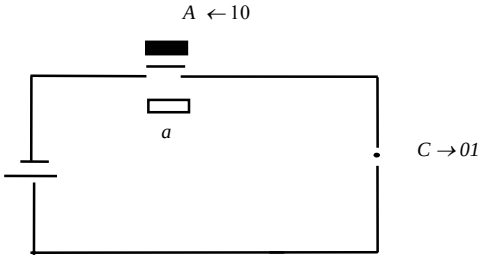
Below the principal schemes of technical realization of the main logical operations are shown as discrete computer elements.



Picture 1.2. The scheme operation OR (disjunction)



Picture 1.3. The scheme operation AND (conjunction)



Picture 1.4. The scheme operation NOT (inversion)

The operation XOR is more complicated and in the simple schemes, which were given here, can't be realized for the reason that the result of operations of both zeros and ones must be simultaneously the same, i.e. equal to the zero.

The operation XOR may be performed by joining a certain set of discrete elements into one system, which maybe expressed by the formula:

$$(m \vee n) \wedge [\neg (m \wedge n)] \qquad (1.1)$$

There m and n are any quadri-digit binary numbers. The operation OR ($m \vee n$) shows in which segments of m and n the numbers coincide. The result will be zero there. The operation AND ($m \wedge n$) shows in which segments of m and n the ones coincide. Even if there weren't such coincidence, the essence of the operation wouldn't chance. The operation NOT is being applied for the result of the operation AND (logical multiplication) and it changes it with the reverse. It means that both obtained binary numbers have zeros in the segments where two prime data were the same, i.e. either ones, or zeros. Thus by logically multiplying those two obtained binary numbers, we obtain ones in the segments where the prime data were different. This is the result of the operation XOR (table 5).

Let's say $m = 0101$ and $n = 0011$, then

$$(m \vee n) \wedge [\neg (m \wedge n)] =$$

$$=(0101 \vee 0011) \wedge [\neg (0101 \wedge 0011)]=$$

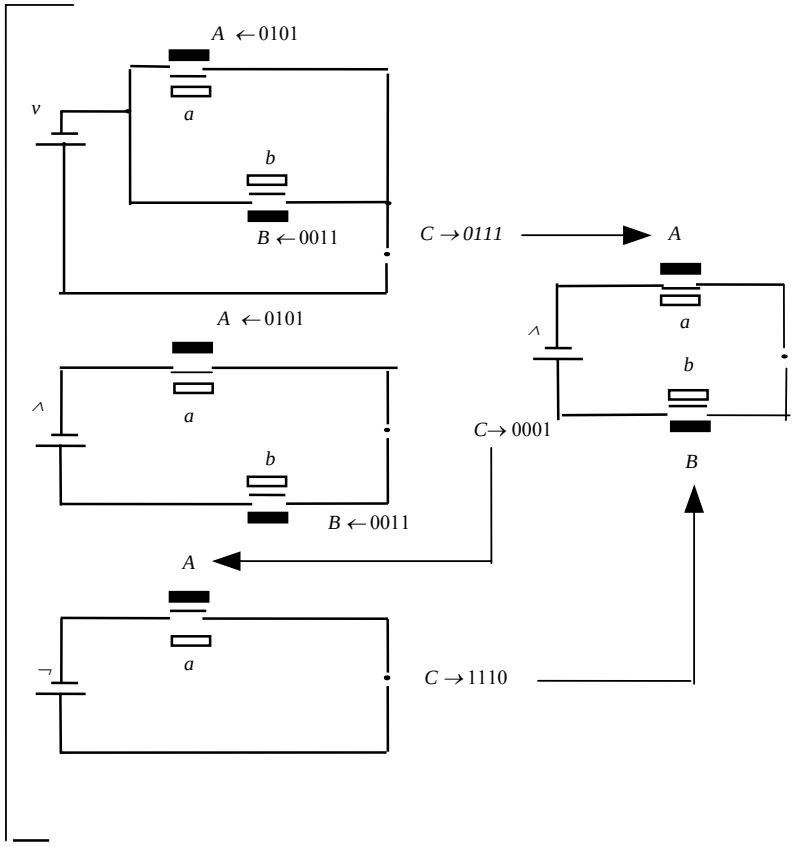
$$= 0111 \wedge (\neg (0001)) = 0111 \wedge 1110 = 0110.$$

The number 0110 is the one that would be obtained if the operation XOR was made from the outset:

$$m \oplus n = 0101 \oplus 0011 = 0110.$$

By conjoining discrete computer elements given in pictures 1.2, 1.3, and 1.4 into some certain set according the formula (1.1) the scheme of the operation XOR is obtained (picture 1.5).

It is easy to get sure that by similarly constructing schemes, as it was made for the obtaining of operations OR, AND, NOT, and especially XOR, it is possible to make schemes for the realization of any arithmetical or logical operation. It is especially apparent when discrete elements are made for elementary (undivided) binary procedures. In such an operation only two binary uni-digit numbers. It is sufficient to have fifteen such binary operations and their schemes with two binary uni-digit numbers in order to be able to make schemes how to make any arithmetical or logical operation (picture 1.5).



Picture 1.5. The scheme of the XOR operation.

The prime data of elementary operations 1, 2, 3, and 4 (table 6) make all possible binary combinations, and their results are equal to zero. The results of the same operations are equal to ones (operations 5, 6, 7, and 8). The operations 9 and *A* are inversions. The operations *B*, *C*, *D*, and *E* are not the repetition of some certain operations 1 – 8, as the results of the operation are being associated with the position of one and zero; it means that the result of the operation depends on which of the operands (the first or the second) is one or zero.

The operation *F* is only conditionally attributed to elementary operations as it can be realised only with the help of more complicated discrete scheme of computer element. By conjoining this operation with other elementary operations from 1 to *E*, all previously mentioned arithmetical operations could be made.

As the amounts of elementary operations and the variety of the combinations how to integrate them are practically little limited in modern computers, it is possible to integrate various schemes the way that enables them to make any necessary amount of logical-mathematical operations of any level of difficulty.

The table 6 of elementary operations.

Codes of the operation	**1 2 3 4**	**5 6 7 8**	**9 *A***	***B C***	***D E***	***F***
Prime data	0 0 1 1 0 1 0 1	0 0 1 1 0 1 0 1	 1 0	1 0 0 1	1 0 0 1	1 1
Results of the operation	0 0 0 0	1 1 1 1	0 1	0 0	1 1	1 0

The previously mentioned arithmetical and logical operations enable to understand the essence of binary manipulation with data and it also shows unlimited possibilities of manipulation with data of this process. On the other hand, the given way of realisation of arithmetic and logic can be absolutely different. It means that in data byte one can be given in a different way as it was given in chapter 1.1. It is important that physical scheme would “realise” that it is having business with one or another number or symbol and that it is able to process it in the right way according the function of its own purpose.

Varies code schemes of figures, letters or any other symbols are being used in practice. Even some computers of different producers have different inner codes. Such differences do not interrupt into data exchange, as the processes of recoding are very simple and are easily physically realised.

The table of 256 bytes are taken for the recoding, in the bytes of which binary symbols of the coding system into which we want to recode, are being written. The recoded symbols are considered numbers that show the place of the needed byte in the table. The equivalent of this recoded symbol is written in that byte. For example, if in some certain coding system *A* is a hexadecimal number *F*1, then

its equivalent, symbol *A*, has to be written in 241 byte of the table, in which that *A* is being recoded, counting from the beginning of that table, because

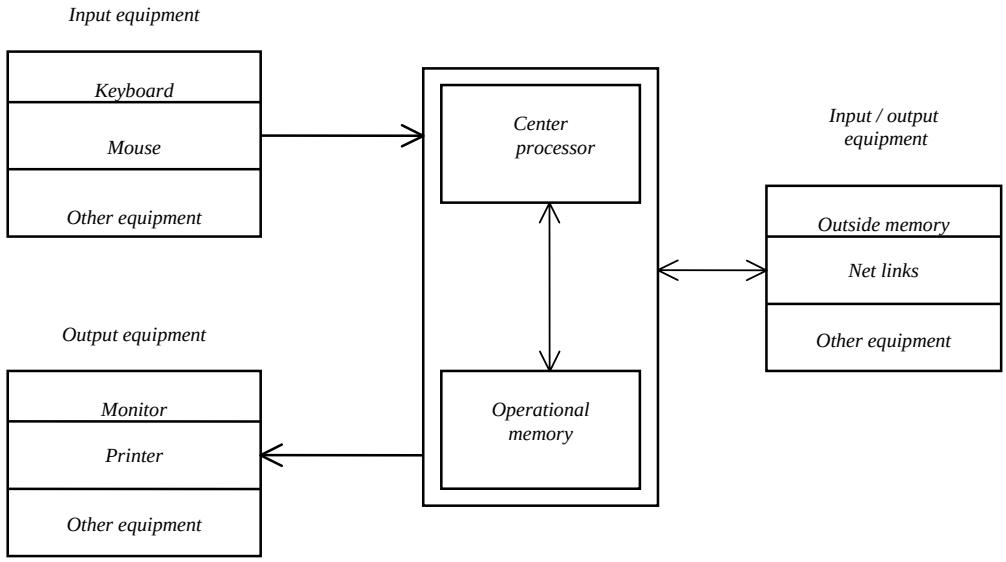
$$F1_{16} = 241_{10}.$$

It is notable that the purpose of the computer depends on the level of the realisation of schemes. If the schemes are realised in such a way that the computer solves tasks of final consumer, then such a computer is usually a computer of special purpose, i.e. it has been created for solving of limited amount of certain tasks. It can be various computer techniques from the usage of computer elements in various facilities to special computers that rule whole technological processes. If the schemes of the computer are realised in a way that does not enable it to directly solve the tasks of final consumer then such a computer is considered universal. The schemes of such computers solve some certain intermediate tasks and the their combinations supply the needs of the final consumer. In this case the variety of tasks that are being solved can be very big, that is why such computers are considered universal.

1.5. The equipments of universal computer

The names of the equipment of a universal computer and the scheme of their links are given in picture 1.6. The whole content of this book is oriented towards computers of specific purpose. Usually a universal computer consists of:

- central processor;
- operational memory;
- equipment of input and output and their links.



Picture 1.6. The linking scheme of the equipment of a universal computer.

Central processor is the main device of controlling computer system and data processing.

Computer memory is the main device of data recalling. This device has two strains: operational and outside memory. The central processor does operations of data processing in operational memory, and the outside memory is used to remember and store big amounts of data.

The equipment of input / output is divided between output equipment and input-output equipment.

A universal computer can be joined or not joined to local or global computer networks of various types, therefore the possibility of actuating networks is shown as physical link in the scheme and not as concrete equipment.

Computer monitor is apparatus device that enables to visually provide the consumer with data and helps to visually observe the data that were brought to computer with the help of keyboard or in any other way.

A mouse is a manipulator that enables to control the process of data processing more operationally than a keyboard. The purpose of the keyboard and the printer is shown in their names.

The descriptions of discrete computer elements and the latter scheme of computer equipment and their inner links end the descriptions of physical computer elements given in this book. The rest part of the book is devoted to the subjects of applied mathematics.

2. Information and data

Knowledge, that are expressed in words, is accepted to call information, and more concrete knowledge, expressed in numbers is called data. Generally, the concepts of information and data are very similar, as most knowledge of various structure and configuration can be called both information and data. In this work, as in other work of informatics of mathematical nature, knowledge that is given in a way easily understood by people is called information, and the same knowledge, that is expressed mathematically, is called data.

Thus, information processing consist of methods and ways in which information can be expressed mathematically, by transferring it into data, redone, process these data and give them as information, which is new, different or given in another way and so on. In order to be able to process the information, it is necessary to have information and data that shows how to do it. So, it is necessary to have information about information, and data about data. The latter can be called metainformation and metadata. The concepts “metainformation” and “metadata” are not popular in practical. The information about information and the data about data are called differently and it various ways. For example, one of the kinds of metainformations can be called algorithms, and one of the kinds of metadata can be called programs. Further in this chapter the basic concepts of information, data, algorithms and programs are given.

2.1. Information

Information can be considered the totality of knowledge about the real world. That totality consists of knowledge about various objects, from elementary to very complicated. Therefore, the information as totality can be divided according various indications. The names of the kinds of information depend on those indications.

According the stages of information processing, information can be primary intermediate and final, devoted to the final using of that information. According the type of tasks, information can be operational, closed, commercial, scientific, economic and so on; according the meaning – correct, incorrect, total, incomplete, discrepant, unnecessary and other. It is easy to see that information of one and the same title can correspond to criterions not of one, but also of some kinds. For instance, economic information can be both primary, and incomplete, and scientific. Therefore, such categorizing of information in computer science doesn’t have factual value, but it lets understand the essence, variety and abundance of it.

In this material the information that can be characterised by the following factors is being operated:

- information belongs to final consumer;
- information requires computer processing of certain level;
- information can be expressed in some certain concrete mathematical constructions.

The final consumer is considered a person or an organization (physical or judicial person), that have their own information, that has to be, is necessary to or purposive to process with the help of a computer. Besides such information can't be completely processed with existing programming systematical, applicable and typical or any other software. It means that it is necessary to create special software for processing of such information that could solve the tasks of the mentioned information processing independently or together with typical, applicable or any other software.

These are the cases of the levels of information processing:

1. The computer processing of information doesn't change the information, we don't get any new information, computer is being used as physical tools devoted to improve human work;
2. The consumer gets information of another consumer, that is processed in a form, sequence processing duration and the like that the consumer needs;
3. The result of information processing is new information that is expressed in numbers, words, symbols, codes, that has the meanings of known or new attributes.

It is obvious, that the given levels of information are conventional. On the other hand, this conventionality is sufficient, that by using only the third level of processing, the consumer has to find out about the creation of special means of processing. The example of the first level of processing can be various text editors. The text is being edited by a man that gains new text (new information). A computer is only a too, that makes the human work easier in different aspects. The computer system can't create any new information in this case.

The form of providing the information is table.

A table is usual, apparent, often met in practice in information processing not only with the help of computers for any consumer. On the other hand, information provided in tables are easily transformed into mathematical constructions, that is into relational sets. The latter also have flat quadrangle shape. The relational sets are also called relations, interfacing, interrelation. The usage of relational sets is purposive also because the sets have a developed fundamental mathematical apparatus that can be modified for the needs of applicable mathematics that is by transforming information into data.

It is noticeable, that most technical, technological, economical, scientific and other kinds of information have the expressing of relational sets.

2.2. Data

In this chapter some initial knowledge about data is given. These are basic ways how to transfer information into data, fundamentals of formalization and description of data in formal expressions that are given in binary symbols in computer memory. The concepts of data given in this chapter makes it possible to understand other objects of this part of the book: algorithms, programs, programming systems and other.

Datum is considered a statement that is given in logically independent, prime sentence. Such a sentence has a logical subject and bases, and also objects and attributes. The latter indicates the main characteristics and features of the subject and bases.

The sentence "IN THE INSTITUTION *b*, AT THE BEGINNING OF THE YEAR *d*, *c* WORKERS WERE WORKING, THAT HAVE *e* YEAR WORKING EXPERIENCE AT THIS INSTITUTION" can be an example of datum.

The logical subject of this statement is the word " WORKERS", the bases is *c*. All other concepts only add specify and mark some certain characteristics of the subject and the basis or mark some certain features of theirs.

The given statement is logically independent and it cannot lose or gain new features or objects. If it happened, it would be another, i.e. a new statement. For example, if one part of the sentence was taken away: "THAT HAVE *e* YEAR WORKING EXPERIENCE AT THIS INSTITUTION", the basic should change and we would have two data (or two sentences) about the workers of the institution *b*. even in case *c* didn't change, there would be two data about the workers, as it would be apparent, that *c* workers are working at the institution and all of them have the same working experience, i.e. *e*.

If we have not one, but several or more data about the institutions and the working experience of their workers, then we can make the hypothetical table of such data. When making such a table, first of all we single out common features of all data. In this case the moment of time, i.e. the beginning of the year can be such a feature. This feature subjected out of descriptions of all data is usually given in front of the table and it serves at the part of the indicator of the table.

Another part of the indicator should be a subject, for the data of which the table is denoted. In this case it is the certain part of the totality of the institutions, all institutions of some business, educational or science institutions, or institutions working in some types of activities. Then the scheme of the table is being established and coded, i.e. the attributes given in some certain row are being coded.

Thus the identified scheme of the table consists of the scheme of identifier attributes and the scheme of the table itself.

The scheme of the identifier:

a – the code of the table itself, as the independent derivative of data;

B – the attribute of the subject – a reference showing what sphere of activities the institutions, the data of which are given in the table, cover;

D – the code of the factor of time, in this case it indicates the beginning of the year, and by adding the table it is being indicated data of the beginning of what years are given in the table.

The scheme of the table – the list of its attributes and codes:

Table 7

The name of institution	Total number of the worker	Working experience of___ years	Working experience of___ years	Working experience of___ years
C_1	C_2	C_3	C_4	C_5

In this scheme and further in the tables the attributes will be given in capital letters, and the meanings of the attributes will be given in the same not capital letters. Thus, *a* is not an attribute but the meaning of the attribute, i.e. the code of the specific table itself about the staff of the institutions. In the scheme of the table in the attributes C_3, C_4 and C_5 specific years of working experience are not given. It is done on purpose as the form and structure of the data are being analyzed. The giving of the forming of working groups according the specific number of years of the work at that institution would be the condition for solving the task of working experience. The latter object is not the goal of the analysis of these data. Besides, it shows the reader the essence of division between the subject of informatics and the tasks of some certain sphere that uses informatics. It helps computer science, but it doesn't and cannot change neither the theory no the practice of other spheres of activities.

The example of the table filled with data:

a, $B - b_1$, $D - d_1$

Table 8

C_1	C_2	C_3	C_4	C_5
C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
C_{21}	C_{22}	C_{23}	C_{24}	C_{25}
C_{31}	C_{32}	C_{33}	C_{34}	C_{35}
---	---	---	---	---
---	---	---	---	---
---	---	---	---	---

The number of type *a* data c_{ij} (i = 1, 2, 3,…; j = 1, 2, 3, 4, 5) in the table depends on what data are necessary, i.e. of how many institutions and on the working experience of the beginning of what year. Despite the abundance of data c_{ij} , it is sufficient to have only one scheme in computer memory, and to keep data in computer memory separately from the scheme together only with their identifiers. Then we have one register for the assessment of the structure of data:

$$a, B, D; C_1, C_2, C_3, C_4, C_5$$

The meanings of attributes will have the shown form; their amount will be influenced by different number of *b* and *d*:

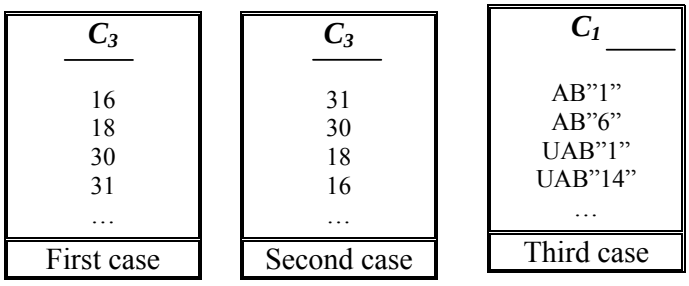
$$< a, b, d; c_{ij}> \qquad (2.1)$$

In such a brief and formal way we can give the structure of the table (2.1) by one formula that is "convenient" to keep in computer memory. Besides, the above given table ab_1d_1 can also be the part of the sets of the latter formulae.

Thus *b*, *d* and c_{ij} are the meanings of given attributes $B, D, C_1, C_2, C_3, C_4, C_5$, that are different to each element of the set of meanings, i.e. to independently identified table of data that does not have directly attributed scheme. That scheme is attributed to all meanings of attributes; after adequation and finding that codes *a,* that identify both structures coincide.

The rows in the table of data can be given in different order – according the increasing or decreasing of the numbering of any attribute, according the alphabet of the first letters of the meanings of attributes and the like.

Below the rows according the increasing of the meanings of attribute C_3 (I), according the decreasing (II), and according the alphabet of the first letters of the names of the institutions C_1 (III) are given.



Picture 2.1. The ways of data rows in the table.

It is apparent that data do not change because of the difference of ways the rows are given. But it is necessary to have the ability to separate one row from the other. That is way one of the attributes is considered a key attribute, and its meanings become key rows, i.e. identifiers. If the meaning of one attribute is not sufficient, as there are repetitive meanings of key attribute, and then two or more attributes are considered key attributes. In the given table *a* key attributes will be C_1, i.e. the name of the institution. If for any reasons (for example, because of the mistake) there are two or more meanings of the same key and two or more meanings of other equal (coinciding) attributes, all the rows are taken away from the table, we leave only one of them. It is apparent that we doubled the same data needlessly. If there is the same key for two rows, other meanings of the attributes differ, than it is obvious that the given data are incorrect.

For example, different number of workers at the beginning of the same year at the same institution cannot have the same working experience. Only when the rows are being doubled, such a mistake is possible. For example,

C_1	C_2	C_3	…
AB”1”	100	26 ?	…
…	…	…	…
…	…	…	…
…	…	…	…
AB”1”	100	25 ?	…

Picture 2.2. The doubling of the rows in the table that made the giving of incorrect data possible

In this case only the owner of the data can find the mistake and correct it, i.e. it is necessary to find the prime source of the data.

The number of the scheme of attributes in the table is called the rank of the table. The number of the rows in the table is called the set of the table. If the rank of the table is a constant number, the set of the table may vary. For example, the rank of the table *a* is 5, then the set for each filling of the table

(for different *d*) can be different and it depends on how many institutions there are in the object *b* at the beginning of particular year.

The variety of the content and the identifiers of the table are being under consideration in theoretical and practical aspects because of the quantitative and structural differences. From the theoretical point of view the sets of the attributes C_1 and their meanings c_{ij} , that make the content of the table, are infinite, as they reflect the variety of the real world.

The sets of the identifiers *a*, *b,* and *d* from the table and of their attributes are different in their nature and numerousness.

The set of the identifiers of the table schemes $\{a\}$ can be unlimitedly big, as it is theoretically possible to control other new schemes of attributes for various objects, processes, phenomena and other things of the real world. Besides, a different number of one and the same scheme is possible.

For example:

$a; C_1, C_2.$
$a; C_1, C_2, C_3.$
$a; C_1, C_2, C_3, C_4.$
$a; C_2, C_3.$
$a; C_2, C_3, C_4, C_5.$
\- - - - -
\- - - - -
\- - - - -
\- - - - -

new schemes are being obtained. In all obtained schemes of attributes, the attributes can be put commutatively, we can obtain n! schemes from each new scheme. All this show that from the theoretical point of view the number of elements in the set of schemes $\{a\}$ is not finite. Each of these *a* identifiers different scheme of attributes, that can be filled unlimited number of times with the meanings of attributes c_{ij}**.** Such filling also depends on unlimited amount of the objects and subjects *b* that reflect the real world.

The variety and abundance of all the tables mentioned above is radically supplemented by the part of the identifier – the factor of time. The factor of time is the certain fixation of time or the reference of time interval, in which c_{ij} is significant. From the theoretical point of view it is possible to make new tables for each factual time interval or moment from all previously mentioned sets of the infinite data tables.

In practice the number of the table schemes of one institution, that has nothing to do with the industry of data processing, is usually between one and several tens. Besides, the bigger part of these schemes filled with content, i.e. c_{ij}, is being processed with common or typical software, that is being received from other institutions, various computer networks and the like. Usually only the small part of

data is being processed with special software created for that institution. Thus, ten to twenty schemes are usually sufficient to provide certain, characteristic of one certain institution data. The number of data filling of these schemes is varies very much and it depends on the nature of certain tasks. But in this stage of examining data it is the most important to realize the dependence of data identification on the features of the data and vice versa.

Thus, data, the schemes and attributes of which, have factual meanings of attributes can be identified:

- *a* – only by the code of the scheme;
- *a, b* – by *a* and the codes of the subject;
- *a, d* – by *a* and the codes of the factor of time;
- *a, b, d* – by *a* and the codes of subject and the factor of time;
- *a, b, d,* ? – three factors of identifying are not sufficient.

If the code *a* alone is sufficient for the identifying of the data table, it means, that the data c_{ij} of that table do not belong to a certain subject and do not depend on the factor of time. Then *a* is made of various constants and/or normative data, as the normative data are conditionally constant.

The code or the name of the data that are being identified *a* and the subject *b* could be some certain technical device, the attribute meanings of which are given as the aggregates, parts, their numbers in the device, their nomenclature numbers and other things of the device, as the amounts of aggregates are usually (but not always) independent on the factor of time.

If the attribute "KAINA" is added to the data, identified *a, b,* in the given example, and its meaning is expressed in money units, then the identifier should be supplemented with the factor of time *d*. This is because the price can change after some time or by taking another moment of time, different from the one given in the table at the time when it is being filled. Then we would have a new table with the changed prices and the old table with the old prices. The new table would then be identified as a, b, d.

It often happens that a, b, d are not sufficient to identify the data tables. It happens because the identifiers of more than one table coincide. It wouldn't interfere in the solving of some tasks, but there might be such tasks for the solving of which it is necessary to know how many tables there are with the same *a, b, d*. It can happen because of many reasons when accumulating data for the solving of easy tasks. For example, let's say that data are given in data tables about a certain operation in the technological process. It is not important if these data are the same or not, it is important to have them and to know how many times the operation had had been carried out. If the beginning of the operation is measured using whole numbers (by rounding them down), then two operations during the shorter

summary duration than the measure unit will show that two operations were carried out at one and the same moment of time *d*. In practise an example of such an operation could be two telephone conversations taking place between two people during one minute if the number of the moment of time is being rounded down by accuracy to one minute, when the fractional part of the number is being eliminated.

Generalizing everything that has been said about homogeneous identifiers, it can be stated, that sometimes identifier made from the elements *a, b* and *d* is not sufficient for the homologous identification of the tables. Then it is possible to point out one or several addresses of the data table in the identifier and to write the content of the address (instead of the address) in the identifier. Additional data in the identifier will make it possible to unambiguously identify any data table. The addresses are pointed out in the table with number of the column *j* or the number of the row and the column *ij*.

2.3. Algorithms and programs-data

Before defining what an algorithm is and what a program is we can say that they are data. These are data about the data that were described in the last chapter 2.2. So data about data are usually called metadata.

Drawing a parallel between the concepts of the first chapter of the book about the binary expressions of data in computer memory and the descriptions of the real objects in this chapter, it is obvious that computer elements will be able to process these data only when they will appear in computer memory and will be expressed in binary form. **Such "mediators" that can logically and physically join the data about the real objects with computer schemes are algorithms and programs.** Thus, such totality of references and rules, that defines the process of data processing in a way that enables to receive data that are the processing result or the prime data, is called algorithms.

A person can memorise an easy algorithm but it is necessary to write more difficult algorithms down into electronic equipment on paper or the like. In any case this algorithm will have nothing to do with data processing until it is transferred into a program and until this program isn't expressed in binary form and written into the computer memory. Thus, there might be only two basically different kind of data in computer memory:

- data about real objects;
- data about data or programs that enable to transfer, process, bring data into and out of computer memory and so on.

Thus, data that are devoted to control the components of data processing and for the carrying out of the algorithms are called a program.

The control of the components of data processing means that in data program there must be references to other more common programs of the tasks of data processing, in order they would do some helpful works and some certain moments of realization of algorithms. Such components might be the bringing of data into and out, the control of correct acts of data and programs, the security of programs and data in order they would not be erased and etc.

If the program in computer memory worked alone, it would have to do a lot of helpful works that would have to be programmed together with the applied program each time according the particularity of that program. In order to avoid such works in program making, the schemes of common nature are made, that provide applied programs. Those schemes are called operational, or disk operational, schemes of the translator and other. They change together with the changing of the fluctuation of physical features of the computer, they improve, get old on their own, new are being created and etc. These schemes make up one part of the computer artificial intelligence that is the imitation of human intelligence features with the help of the computer.

Ending this short chapter, devoted to the definition of the essence of programs and algorithms, we can see that not only the content of the data but also their purpose must differ in binary computer memory. Data being coded in various ways and methods reflect the structure, content and semantics of the data. Program codes show what and how needs to be done to receive the result of the processing of data. Computer schemes (chapter 1) analyse binary text of the programs and carry out some certain acts with data.

3. Computer data

Computer data are data about factual objects of the real world and metadata that have a structure and expressions that a computer memory device can memorize them. This device is usually called computer memory, and such data are called machine or device data.

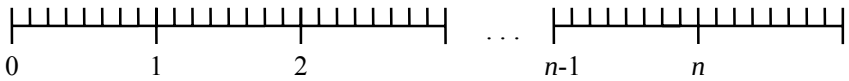
The aim of this chapter is to give a concept about how the data that are expressed in elementary binary form in a computer memory don't lose the content and purport of the factual object. Besides, new data obtained during the binary process that have both new structure and new content and different meaning from the one that have been before the binary processing. Giving the answer to the given question, in this chapter the principles of addressing computer memory, the structures of the data and programs and their relations in computer memory, the concept of the fields of addresses and the examples, and the ways to code the content are being analyzed.

3.1. Memory addressing and data structures

Computer memory is technically complicated device that improves when he is being reduced by his physical parameters and increased by his capacity.

Looking from the mathematical point of view, it is sufficient to understand computer memory as the vector of binary segments or their groups. The group of eight segments makes the least addressed memory cell that is a byte.

In the vector of memory bytes the address of the first byte is a zero, and the address of the last byte is a whole number n, that shows the capacity or largeness of the memory.



Picture 3.1. The vector of memory bytes

The content of data byte, record, field, array of any other structure can be set only if the address of the beginning of the structure is known in the memory. Only then it is possible to analyse the content of the bytes and to recognise the necessary data according the setout of the 0 and 1. Metadata show what those data are and what to do with them. The metadata are the algorithm, the algorithm is a task.

The element of the data collection (file, field, array) is considered the data record, the types of the inner elements of which can be different.

An identified set of logically and/or physically connected records is considered a data file.

Data array is a named set of the same type, where records are identified with the help of indexes.

A data field is named part of logically or/and physically connected file or record array, or a memory part, which is devoted to records but isn't filled yet.

If the data field is made from some certain set picked from the data collection but which is given not in the same row as used to be in mentioned data structures, then such a set will be called a data suit.

When it is not important whether the analysed data are a file, an array, a field or suit, then such a named data derivative is called a data collection.

In the future the concept of the task of data processing will be often used or its synonym – applied task. Such a task means the data processing task of the final consumer and it is not directly connected with the solving of systematic, service or any other tasks that shouldn't directly interest the final consumer.

If data record are

$$A_1, A_2, \ldots, A_k ,$$

where 1,2, …, k are the numbers of the record row, then the data file can be written this way:

$$A_i \subset F,$$

when i varies from 0 to k.

In this instance data field L will be subset F

$$L \subset F, \text{ where } A_d \in L,$$

when number $d < k$.

If data records are

$$\boldsymbol{A_{k1}, A_{k2}, A_{k3}, \ldots, A_{k1},}$$

where k_i ($1 \geq i \geq l$) are record identifiers, then the data array M will be made of records

$$\boldsymbol{A_{k1} \subset M,}$$

and the field of this array could be $L \subset M,$

where $\boldsymbol{A_{kt} \subset L,}$

but $t < i$, or data identifiers and there should be less records in data field than in the array. If $t = i$ then

$$L \equiv M.$$

It is apparent that any data arrays can be found and given for the processing, if the addresses of those data structures are known. These are usually the computer memory addresses of the first bytes of those structures. The attention to the just given concepts should be paid:

-"… can be found…";

-"… given for the processing…";

-"… if the addresses… are known…".

There is a question: what can find, give for the processing and obviously what can process? Finally to what the addresses should be known? The answer is short: to the programs. The programs of data processing.

This way before the main programs or the concepts of programming are formed it is possible to understand the essence of those concepts.

The earlier mentioned file addresses are the binary addresses of computer memory, which are called absolute addresses. It is obvious, that it is very difficult to give absolute addresses to the programs. That is why they are being established with the help of some certain methods, that are being realized with certain computer programs. These programs don't directly participate in the solving of consumer task, but it serves the computer process of that solving.

The following are the main concepts about:

- record formats;
- operational and exterior memory;
- the ways to collect data arrays;
- data finding ways;
- data addresses and ways of addressing;
- data bases.

3.1.1. Records formats

The main difference (excluding the content difference) of the records in one data file or array is their permanent or mutable length. It means one and the same data file or array can consist of either permanent or mutable length of records when counting in symbols or bytes.

The records can naturally have a permanent length. It means that some certain data consisting of records about real objects have the same number of symbols. An example of such a record can be corteges of a relational set (i.e. the rows of data table), where the meanings of one and the same attribute have the same number of symbols.

	B_1	B_2	B_3	B_4
Record 1-	*abc*	*22*	*366*	*481*
Record 2-	*bcd*	*93*	*661*	*121*
Record 3-	*cde*	*41*	*332*	*101*

Picture 3.2. Records of naturally permanent length

In practice, when we want the records to have a permanent length, such a length is artificially made. First of all the maximum length of each component (datum) is being established. Then to receive a maximum length of the datum special symbols of datum lengthening are being inscribed in its beginning (usually for numbers) or in its ending (usually for textual data). Usually such symbols are symbols – gaps (␣).

	B_1	B_2	B_3	B_4
Record 1-	*a b*␣	␣*2*	␣*64*	*481*
Record 2-	*c*␣ ␣	*34*	*686*	␣*33*
Record 3-	*d e f*	*43*	␣␣*1*	*961*

Picture 3.3. Artificially made record data of permanent length

Knowing that the maximum length of the records given in Picture 3.3. is of 11 symbols excluding symbols that separate one meaning of the attribute from the other, (3+2+3+3=11), an artificially made length can be made in another way as well: the number of till 11 missing symbols is being filled in gaps (␣) the record in its end.

	B_1	B_2	B_3	B_4
Record 1-	*ab*	*2*	*64*	*481*␣␣␣
Record 2-	*c*	*34*	*686*	*33*␣␣␣
Record 3-	*def*	*43*	*1*	*961*␣␣

Picture 3.4. Artificially made records of permanent length.

In all cases of the making of permanent record shown in Pictures 3.2., 3.3., and 3.4. , between the separate data of the record a space is being inscribed that enables to find the ending of each datum and the end of the whole record is being established by knowing the length of them all. It is essential that it is not necessary to repeat the attributes A_1, A_2, A_3, A_4 and B_1, B_2, B_3, B_4 near each record or record group, the attributes are given only when the consumer receives the data. Thus, the records of permanent length will have the following expressions in computer memory: Picture 3.2. – the first expression, Picture 3.3. – the second, Picture 3.4. – the third.

1.	*abc*␣22␣366␣481*bcd*␣93␣661␣121*cde*␣41␣332␣101...
2.	*ab*␣␣␣2␣␣64␣453*c*␣␣␣34␣686␣␣33*def*␣43␣␣␣1␣961...
3.	*ab*␣2␣64␣481␣␣␣*c*␣34␣686␣33␣␣␣*def*␣43␣1␣961␣␣...

Picture 3.5. Records of permanent length in computer memory.

It is notable that in the second case of permanent record the data of the record are always separated by gap (␣) and the other gaps are given only when the datum does not have a maximum length.

One data file can consist of record groups of the first and the second types. The third group is not included in the general file as its records are identical to the records of the second group for it differs only because of the artificial way of data lengthening.

The third group given in records of mutable length will be shown in Picture 3.6.

ab␣2␣64␣481/*c*␣34␣686␣33/*def*␣43␣1␣961/...

Picture 3.6. Records of mutable length.

Records of permanent and mutable length have some certain advantages and disadvantages.

Records of permanent length are easy to find when they being looked for or through successively, as the beginning of each of them (starting with the second record in the collection of data) are behind the same number of bytes. When an incorrect record is found, it is easy to replace it with the correct one.

The negative side or the usage of the records is that computer memory is used irrationally. As the length of all records is being formed according the maximally possible length of the record, then in practice such maximally possible length of the record occurs very seldom. Most records are noticeably shorter. Thus, 40-70% of memory fields stays not filled or they are being filled with meaningless symbols.

Another disadvantage of the usage of the records of permanent length that after using records for a longer time, their maximum length can change. If the length decreases then the irrational usage of the computer increases even more. If the maximum length increases then the normal rhythm of task solving can fall into disarray, then they will need to use records of two different lengths. In this case some certain correctives may be necessary to be used in the programs of task solving, in order they will be able to solve tasks with records of two lengths, or to make a special program that would equalize the lengths of the records of data arrays. Both the correctives and the special programs in some cases can be complicated and they can complicate the usage of the system of data processing.

The usage of data of mutable length in the task solving of data processing do not have the disadvantages typical to records of permanent length. That is why such records enable to solve tasks of data processing more stable. But there are also negative aspects of the usage of such data. First of all the separating sigh (/) between the records is necessary. If it is impossible to avoid mistakes when the data are being formed then we have to correct the mistakes. When the mistakes are being corrected the

incorrect records are being replaced with the correct ones. If the length of the new (correct) record is shorter than the length of the incorrect record the some certain "holes" are being made in the array of data which are filled with meaningless symbols. If the length of the correct record is longer than the record that is needed to be corrected than the pushing aside of this record is needed in order the correct record fit in. if the correct record can be written into the end of data structure then not used memory field will again be left in the place of the incorrect record. Thus, by using records of mutable length it is sometimes necessary to resend data arrays from one place to another in memory.

Another negative aspect of using long records is that each time when looking for a new record, its length has to be estimated in a programming way.

In given formats, that have permanent of alternate length, record files of arrays are practically being used for doing easy tasks. Besides usually one array or file are used by programs of one task. After having processed that array of data, the same data are seldom used for doing other tasks.

For the doing of more difficult tasks, the records have identifiers that enable to use the data more than one time, to intercept the needed part of the data and their arrays, to pick up data in some certain row, which does not necessarily coincide with the order of the setting of records in the data array.

One of the ways of giving records can be a relational set, which is described in the chapter 2.2., having the expression

$$<a, b\ d;\ c_{ij}>.$$

Here *a, b, d* are the identifier of the record, and the table of the meanings of the attributes c_{ij} :

$$\begin{array}{c} c_{11}\ ,\ c_{12}\ ,\ \ldots,\ c_{1n} \\ c_{21}\ ,\ c_{22}\ ,\ \ldots,\ c_{1n} \\ \text{----------} \\ c_{m1}\ ,\ c_{m2}\ ,\ \ldots,\ c_{mn} \end{array}$$

when $1 \leq i \leq m$, $1 \leq j \leq n$.

The records can be corteges (rows) of any relational sets. They always have the keys that identify them. It is obvious, that the identifiers of the corteges are the meanings of the attributes. That is why these identifiers are primitive, can repeat often and their usage is limited. The domains (columns, graphs) of the relational set cannot be records when difficult tasks of data processing are being done, as they cannot be identifiers. All the data in the domain of the attribute are the meanings of that attribute.

By now when data records were being analysed, we confined to two characteristics of theirs: record length and its identifier. In the main parts of this book the variety of the data meaning and features of the content and their formal expressions will be analysed. In this chapter we confine only to the influence of the record length and identifier to the process of data processing. The minimum and maximum of record length have the direct influence to the making of the structure of computer inner – operational and outer memory.

Some important characteristics of data processing (the speed of finding data, the time of their processing, etc.) depend on the structure of the memory.

3.1.2. The operational and outer computer memory

The common purpose of the operational and outer computer memory is to memorise the data.

When the computer is switched off, the information, which is in the operational memory, (usually) rubs out, and the information, which is in the outer memory, is saved. Thus by continuing the work, that is when we continue to process the data and in many other cases, the exchange of data goes on between the operational and outer memory. Such an exchange of data is being carried out by systematic programming means, thus it could seem that there is no need to go deeper into the concept of the memory, but actually the final consumer has to understand the purpose of the process of the exchange. The tasks of the consumer differ in their algorithms very much when all the data of the concrete task go into one operational memory and when they do not go into one operational memory. Special and often not the same programming exertions are needed in order the data are processed when they all do not go into the operational memory. Despite the operational memory is always being expanded, with improvement of the technical part, the arrays of the data are also always increasing. Besides the bigger and bigger part of the operational memory is being occupied with operational systems and other means of systematic programming means. The obvious example of the mentioned phenomenon, when the data of one task go and do not go into the operational memory, can be the screening of data records. Later it will be shown how many additional and difficult procedures are needed in order the records are screened when they all do not go into the operational memory at the same time.

The peculiarities of the operational and outer memory show up, when they are being compared according some features of theirs. Such features are:

- the purpose;
- the structure;
- the capacity;

- the speed of data memorising;
- the arrangement of the records;
- the search of the records.

The purpose of the operational memory is to directly participate in the process of data processing. It means that any changes that take place during the processing are being recorded in the operational memory in order we can change them and process them later. The outer memory participates in data processing as well. Part processed intermediate data (usually when there is not enough space in the operational memory), not ending the processing is being remembered in the outer memory and again brought into the operational memory for the later processing. The main purpose of the outer memory is to save the data when the process of data processing is being interrupted for various reasons. For instance, after having finished the doing of the tasks of data processing, the results are being remembered. The intermediate data are being remembered when the data are being stored asunder, when the data processing is being interrupted deliberately (at the end of the working day, having finished some certain stage of data processing and so forth).

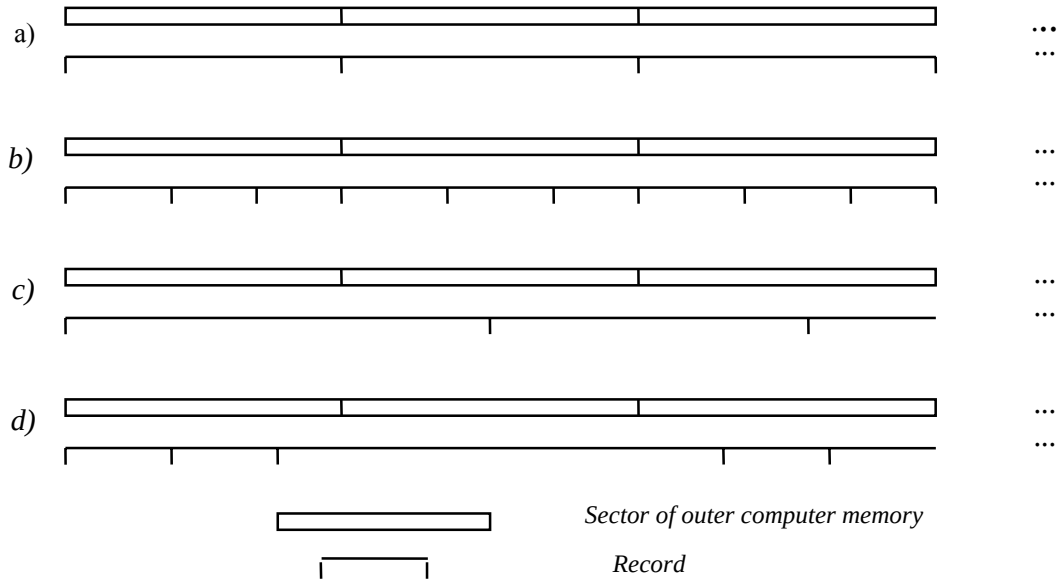
It is sufficient to understand the structure of the operational memory as homogeneous, i.e. indiscrete, integral data vector. It consists of symbols, numbers, letters that are written into memory bites that are being addressed in whole numbers (0, 1, 2, …). It absolutely not important of what nature or content the data that are written into memory bites are: data about real objects, data addresses bites, programs or other. Structure of outer computer memory is not integral. They have some certain memory units. For instance, the unit of disk memory can be a sector with fixed (the same) capacity (256 bites). Thus the transfer of data from the outer into the operational memory can be carried out only with whole units of the first one (in this case with sectors). Only later, in a programmable way the part of data that are necessary for the task, which is being solved, can be excluded from the sector. Obviously, it is not necessary to take sectors only successively. It is not difficult to use them in any succession necessary for the task.

From the above content of the text it is apparent that the capacity of the operational memory is smaller than the capacity of the outer memory. Besides, the speed of data exchange or memorising of the outer memory is not as big as of the operational memory.

The proportion of the length (capacity) of the unit of records and outer memory plays an important role in the set out of data records in the outer memory. As the units of outer memory are being identified with whole numbers that come one after another then when the length of the record and the length of the outer memory unit coincide, it is easy to look for the records. It is sufficient to apply to the number of the outer memory unit. Such coincidences are very rare in practice. Most

often some or more records can go into one unit of outer memory or on the contrary, one record involve two or more sectors. In two latter cases, if the lengths of the records are changeable, than it is obvious that the search of the needed record is complicated. Most of the search is being made having transferred memory arrays or files into the operational memory according the content of the data records.

The following are the schematically given proportions of the record length and the size of the memory unit.



Picture 3.7. Records of various length in the outer memory.

In order the record search is successful, i.e. in order it is possible to find all the records or their needed part fast and in the necessary succession, first of all effective ways of creating data files and arrays have to be used.

3.1.3. Ways of creating data collections and record search

The ways of creating data collections mean the setout of data records in the collections the way that the data processing task programs could effectively find the records. The effective finding of the records mean that the needed records are being found with the speed that could make the task in the time that the consumer needs and the task which is being solved does not interrupt the apropos solving of other tasks. Other factors (the required amount of data that is being found, necessary records and they are given for the processing in the row that is being established by the algorithm of the task which is being solved) are also attributed to the effective record finding.

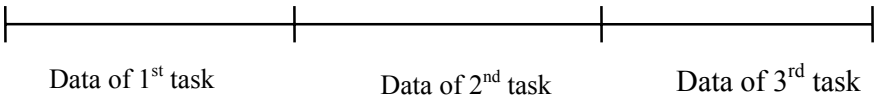
The ways and methods of creating data collections depend on many and various factors. One of the main factors is how many and what tasks are being made by using one and the same data collection. If for the processing of data collection only one task is being used or strictly defined group of tasks (that can be understood as some certain system of tasks) data records are written into the collection by using some certain ways and methods and data collections are called files or arrays. If the amount of tasks is not established in advance for data processing and if that amount can change, the ways and methods of data organization differ very much from the ones previously mentioned and the collection of such data is called database.

The ways and methods of creating data collections and the description of the search of data records will be given.

The associations of the algorithm of data processing with the primary data records determines the ways and methods that are being used for the creation of making data collections.

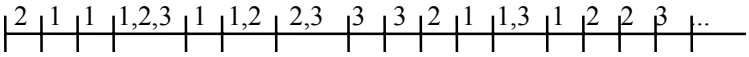
If the algorithm is such that the records are being processed all and in turn, in the order they were memorised, then the way of creating a collection is trivial. Records are being taken from the problematic field (that are brought by a person, by using a keyboard, taken from various sensors of technical devises or taken from the computer memory as the results of other tasks) and they are being memorised without any additional efforts in turn in the mentioned place of the memory. In this case the records are being processed where they are inscribed, if they are not replaced during the processing and they are not later used by other task making programs. It also happens if two or several programs of task solving are using data of one collection.

But there are cases when forming a collection of data it is rational to divide data in the collection into records that go successively, for each task separately.



Picture 3.8. Together and separately memorised primary data of making different tasks

The simplest way of creating a structure of data collection was given above. The following, ignoring intermediate ways of structure complications, practically the most complex case of creating files or structure of arrays and compatibility with task algorithms is given. Some records of data collection are being used only for solving a concrete task, the others for solving several tasks. Besides, those records are being memorised in data collection in the succession they were acquired.



Picture 3.9. Common and different records of primary data of making different tasks

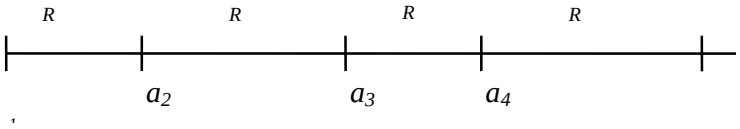
In Picture 3.9. records are being shown, and the numbers given together show for what tasks (1st, 2nd, or 3rd) they are being used. It is obvious that these records can be separated from each other only according their content. It means that they can be identified according the issue factor, codes of attributes or supplements, identifiers of data tables or the keys of the corteges of data tables. It depends on what is considered a record from data structure at a given time: an issue, a table, or a cortege.

In the case being analysed, if the structure of the data is not big, i.e. its multifold full revision when finding the records, which have the necessary attribute for each task, does not interrupt the work of the main system in respect of apropos task solving, then it is not necessary to make additional efforts when looking for data of actions. If data collections are big enough and can interrupt task solving then it is necessary to have one or another way to make one or another methods to create fields of address searching.

When forming a data collection each record that is being inscribed into the memory field devoted to the collection gains its address. That address is usually the address of the memory byte of the first record symbol. In the following process of data processing it is possible to operate not with records themselves but with their addresses. It is especially convenient because the record is being found offhand when addressing to its address.

If the field dedicated to data is called *A*, the addresses are called *a*, and the records are called *R*, then

A:



Picture 3.10. Fields of data records.

the addresses are being inscribed into the field *B*.

B: $a_1, a_2, a_3, a_4, \ldots$

When the records are needed, their addresses are being taken and any needed record is being found offhand.

If according the identifiers of the records it is known in advance what records are needed for what tasks then record addresses are grouped according the tasks. It means, that record data are being

inscribed into record address fields devoted to separate tasks. In these fields record addresses can repeat. It means, that one and the same record is being used not in the only one-task solving but also in the solving of the tasks where its address in the address field repeats.

Address field $\boldsymbol{B_1}$ of the records of the first task	. . .	Address field $\boldsymbol{B_2}$ of the records of the second task	. . .
a_1		a_2	
a_2		a_4	
a_3		…	
a_4			
…			

Picture 3.11. Data address fields

If the records needed for some certain tasks have to be sorted according some attribute, for example, according the increasing of the identifier, then it is possible to inscribe records together with the sorting attributes into the address field and to sort those attributes.

If

$$\boldsymbol{B}\text{: } a_1 6,\ a_2 3,\ a_3 1,\ a_4 5,\ \ldots$$

then after sorting addresses according the increasing of sorting attributes the address field is being received

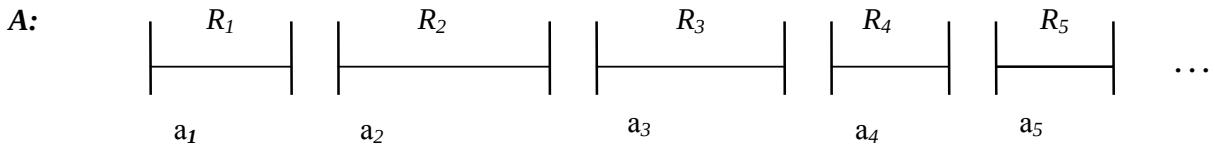
$$\boldsymbol{B}\text{: } a_3 1,\ a_2 3,\ a_4 5,\ a_1 6,\ldots$$

or just

$$\boldsymbol{B}\text{: } a_3,\ a_2,\ a_4,\ a_1,\ldots$$

Such preparation of data collection is often sufficient for solving various data processing tasks when one or several tasks have their "own", i.e. dedicated to them collection of initial data. But also possible are the cases when previously mentioned organising ways of data collection are not effective. It can happen when it is necessary to correct initial data because of mistakes or any other reasons and the data records become longer. Then the record addresses that go after the corrected record in the row change as many times as the records are being corrected when they are becoming longer. The change of the addresses mean that some certain part of the collection is being moved into another place in the memory. It is obvious, that the content of address fields dedicated to tasks have to change as well. The latter changes can have a rather complicated algorithm if the address fields were resorted or changed in any other way. We have to remember that the latter ways or organising data collections, as it has been mentioned earlier, are being used only when the address collections are comparatively big.

Schematically the address changes can be expressed the way when for the sake of the simplicity only one correction of the record when it becomes longer is used. Let's say the given record field is A:



Address field B_1 is sorted for the first task

$$B_1: a_2, a_5, a_3, a_4, a_1, \ldots,$$

and the address field B_2 will not require any sorting for the second task

$$B_2: a_1, a_2, a_3, a_4, a_5, \ldots .$$

Let's say the record R_2 that becomes tree bytes longer because of the mistake. Then the other records that go after this record, i.e. records the addresses of which are longer than the address a_2, addresses change in three bytes as well:

$$a_3 + 3, \quad a_4 + 3, \quad a_5 + 3.$$

The part of the data collection from the address a_3 till the end of the collection has to be moved after three bytes into the right side.

The correction of the address field B_1 is not difficult:

$$B_2: a_1, a_2, a_3 + 3, \quad a_4 + 3, \quad a_5 + 3.$$

The field B_2 can be corrected in various ways, but in any case the algorithm of correction is more complicated, as it is necessary to establish the previous place of each address, i.e. whether it has been in front of or behind the corrected address. If the address row of the address field was changed in the most complicated ways, for example it was sorted according some certain samples, then the algorithm of addresses becomes more complicated as the way of creating the sample has to be estimated every time in the algorithm of address changing.

From everything that has been made it is obvious that when there are many corrections and big data collections, the task-solving period of data processing can become impermissibly long and interrupt the work of the system. So it is the most rational to use data collection for the solving of only one task and that collection is being made only for that task and with the help of the programs of the

same task or if it is purposive to use data base and its methods of organising data. The ways of processing data collections, which were analysed till now, have some important features that are characteristic only to them. Thus it is useful to get acquainted with the mentioned things before moving to the principles of organising databases. It enables to understand the principle differences, similarities and negative aspects between data collections and databases in some certain situations of the data processing system.

Data processing in the collections has these characteristics:

- stable, reciprocal relations exist between the structure of the collection and the content and the programs, which process data that are in that collection;
- in the complicated collections of the data processing system a part of initial data repeat;
- data that are processed as a rule are not used anymore for the processing of other tasks, as such processing is impossible or very complicated.

Stable reciprocal relations between the data and the programs that process them originate because these processing programs create the structure of primary data, control the reliability, accuracy, fullness of the data, correct the mistakes and correct the collection of data before the beginning of purposive processing. This is because the concrete data collection is intended only for one task solving of data processing. Thus, because of outer changes in data structure, it is necessary also to change processing programs. Reverse relation also exists. When it is necessary to process data in some other way, it is often necessary to also organize the data structure in another way, to add new data to it and so forth. Such changing of data and programs usually requires much time and materiel expenses. Thus in the setting where the problematic sphere is not stable (data and processing algorithms can often change) it is not advisable to use direct way of processing data collection.

In the data collections of different subsystems of automation information system data processing tasks of one object a part of data can repeat. For instance, a situation, that task data collections of prognostication, planning and accounting do not have a part of the same tasks, is practically impossible. As the data processing is direct, no systemic means exist in order the programs of one task could use the data collection of another task. The final consumer is usually interested only in the results of task solving and their various interpretations that can be obtained also only in the direct way of automation. The direct data processing way in the mentioned way, when a part of initial data in different data collections repeats, can have negative consequences.

Repetitive, i.e. the data that are necessary for solving of various tasks are practically impossible to relate the way that their repetitiveness is being recorded automatically as each task in this case is being solved separately. It is possible that incorrect data (from those that repeat in the collections

intended for the solving of other tasks) can be found only by the solving algorithms of some tasks. For instance when making tasks of accounting, incorrect data can be noticed when the same tasks are being made for the last period. When mistakes are found they are being corrected and some tasks of accounting from the previous period can be remade. But it is not always possible to find and remake incorrect data that have already been used in other subsystems. For example, it is impossible in the subsystem of prognostication of tasks. Thus the tasks of prognostication can provide with especially inaccurate results of solving that become noticeable only later as the period of accounting task solving is usually much shorter than the period of prognostication task solving.

Thus in the information system the results of not interconnected tasks that are being solved in programming and algorithm way, can be inaccurate and even contradictory. If databases are being used for the solving of similar tasks that have been mentioned it is possible to avoid a bigger part of such consequences.

3.1.4. Databases

Database is the collection of interconnected data, which is being used by many programs for the solving of various data processing tasks. Data providing to the task solving systems is being carried out by the control system of databases.

From this definition it is apparent that:

- by storing data in the database their correctness, fullness are being controlled, the correction is being carried out without reference to the tasks and their programs;
- after new tasks appear, when there is a lack of data for their solving, new data are being added to the database also without reference to the new tasks but in the way that the correct data are sufficient for the solving of new tasks.

The aims of the organization and usage of database are to avoid the negative phenomena of data processing (which were described in chapter 3.1.3.) when data collections are being processed. First of all:

- in the database it is much easier to avoid data redundancy because according the organizing principles of the concrete database the data are being identified in one way or another, that is why data that have the same identifiers and meanings are not being inscribed into the base as they are purposeless because they repeat the same meaning;
- the programs of data processing tasks (of applied tasks), as it has been mentioned, do not have to be concerned with problems of storing data, but it is necessary according

the requirements of the concrete database to prepare a direction (some certain program) into the database in order the necessary data are obtained.

On the other hand, the system of databases also has some fundamental negative things. First of all, as the data are not directly oriented towards concrete tasks, a part of data is rarely used, i.e. they are not dynamically active but they change, that is why the whole database and the rarely used data are necessary to be kept in relevant state. This is undoubtedly connected with big material and labour expenditures. Secondly, the database which is being brought up-to-date or corrected and the part, which is rarely used, are impossible to forecast whether it is going to be used in the future at all. Not only new data processing tasks can appear but also old can disappear. This is why it is difficult for the consumer to understand which data are not necessary anymore and which are partially but rarely useful. Necessary data, which are not in the base, are always being established very easily also by using programs of the control of the base and of the direction towards the base. The program of directions straight after the start of their working indicates that there are no data in the database necessary to them as its further work is impossible.

The mentioned negative aspects of the usage of databases are being applied to the bases, which are being created, by the systems of the control of databases of the common character. If the systems of the control of databases are intended for some certain databases of types of tasks then it is should be noted that many negative aspects of the usage of databases could be localized, i.e. not to have tangible influence on the process of data processing.

Very many control systems of databases have been created, both of the common and specific character. Some common ways of creating databases will be given, which will show the principal opportunity:

- about how the data record lists are being created, which enable to select all the necessary data for the applicable task from the database;
- about how it is possible to avoid or decrease the repetition of the same records in the database.

The data record can be a table, the cortege of the table, separate statement (look 2.2.) or structures adequate to them. It is the most important for the record to be unambiguously identified with the expressed features of the search, i.e. with the codes of the features of the record. Only this way it is possible to find the necessary records, which have some features necessary for the applicable task.

Let’s say we have n different data and two memory fields. In the field *A* all the data intended for the database are being inscribed successively (in the row they were given). Each record that gets into the database has its own address of the place in it. Thus later its digital or symbolic meaning is

identifying this record. The addresses of the records are being memorised in the field *B* according the attributes of the records. In the part B_1 of the field *B* the addresses of the records that have the attribute 1 are being memorised, in the part B_2 of the field *B* the addresses of the records that have attribute 2 are being memorised and so forth. Thus in the field *B* the addresses of the records are being sorted according their attributes.

B:	$B_1, B_2, B_3, \ldots, B_n$	*A:*	a_1___a_2_____a_4_____
	a_1 a_5 a_2 a_6		a_4_____a_5_____
	a_3		a_6___a_7_____ ...
	a_4		
	... ...		

Picture 3.13.data records and their addresses sorted according their attributes

The database organised in such a way is of the primitive structure and practically unusable, but it illustrates the principle how to organise database – the separation of the data collection in the computer from the data processing tasks and their programs. Besides when analysing the database of this structure it is possible to understand some types of different applicable tasks and to separate some systematic principles of reorganising data.

Using this database the algorithm of doing applicable tasks is apparent, when:

- primary data of applicable tasks consist of records, which have one or several attributes;
- when applicable tasks use a part of the records of same attribute as primary data.

We attribute to the tasks of the reorganization of the base:

- tasks of data correcting;
- tasks how to avoid the partial dubbing of the data;
- tasks of data reorganization and facelift.

The applicable tasks the primary data of which consist of the records only of one attribute have a simple structure of direction towards the database. The attribute of the data is being indicated according the requirements of the control system of the concrete database and it is sufficient for the realisation of the direction. When the applicable task requires data of more than one attribute the direction is not much more difficult. Then the attributes and any row of records according the attributes necessary for the task are being indicated. It is much more difficult to address to the data when not all records of some attributes are necessary for the applicable task. In this case all the records of the given task are being got in and the necessary tasks have to be selected by the programs that realize the applicable task.

The procedures of task correcting are being carried out by the control system of the database. When there is a mistake the procedure, which is analogous to the procedure of data input into the base, is being carried out. If the new record is shorter or its length is equal to the old incorrect record then when the record is found it is being replaced with the new one. If there is some space of the old record left, it is being filled with some certain symbols to show that the space of such a field is empty. The first address of the empty space is being memorised in the field *B*, in the address of that field B_{n+1}. If the record does not contain into the place of the old record, the same action are being carried out as if the new record was shorter than the old one. After filling the place of the old record with special symbols its address and the chains of the addresses of the attribute are being carried into B_{n+1} the place of the field *B* and the new record itself is being inscribed into the end of the field A after the new address has been inscribed into the place of the old address. Thus if the chain of the places of empty records exists any record which is being inscribed into the field A is first of all attempted to insert into the fields of the empty records and only if it does not contain into any of these it is being inscribed at the end of the records of the field A and the address is being memorised in the corresponding address field of the attribute.

In order to avoid the dubbing of the records the contents of the record addresses, which are in the address chain of the same attribute, are being equated with the content of the new record. The record that coincides with any other record is not being inscribed into the database. Despite this, it is practically very difficult to totally avoid the dubbing. If in the data table or in the data cortege of the table as in independent records the at least one meaning of the attribute differs from the meaning of the attribute of any other record these records are considered incorrect records. But other meanings of the attributes that mean the same can duplicate. It is possible to totally avoid the dubbing if the record is being considered as statements or the meaning of one identified attribute. But the latter way of data organizing requires much bigger fields *A* and *B* of databases and it markedly lengthens data statement.

After some certain exploitation period of the database is over more and more not unused spaces appear in the fields *A* and *B*. Thus, the base itself irrationally gets bigger in its span of empty spaces and the procedures of data input and search become longer. Databases usually have programming means of “compressing” systemic base. The latter remove “empty” spaces of the database and define new data addresses and memorise them in the field *B* according the attributes of the records.

In practice the structure of the bases that are being used is markedly more difficult than the structure, which was described here.

3.2. The connection between data and programs

In this chapter data that are understood as object of processing are being described; programs are understood as means of processing data.

The connection between them is being analysed when both of them are in the operational memory of the computer. The connection between data and programs can be divided between formal and causal or semantic.

Formal connections include such connections that directly do not depend on the separate commands of the program and on their collections and also on the data structure and their content.

Causal connections are such connections the realization aim of which is data processing and obtaining other result data, i.e. doing the task of applicable character.

Some decades have already passed when applicable programs in computer memory do tasks not alone. Now abundant and complicated systematic programming equipment help them. In the level of programming concepts it is sufficient to treat the applicable task as if its programs when doing tasks did everything on their own. On the other hand, it is now possible to give the most common and substantial concept of systematic software. Let's say we have quite a big packet of applicable programs, i.e. such programs that do quite many and different tasks of final consumer. Let's say each task is being done absolutely independently. It means that the program itself outer and inner devices of computer takes care of data correctness, various corrections, their memorising, input, output and other. Thus without deeper analysis and deeper consideration it is obvious that on the one hand most programming works repeat in every totally independent program. On the other hand, when doing most tasks the coordination between decisions and data memorising, in order the data do not erase each other and the tasks do not interrupt in one another's doing. Thus software, which is the same for all abundant and different applicable programs can be separated, systemised by using various and difficult methodology and technique and only purposive realisation of task function is being left for the applicable program. Such common software is called systematic software.

The software of the final consumer can be divided between typical equipment and special software. Typical software includes tasks of common nature, which spread very widely. For example, modern programming projects Word, i.e. some certain typing machine. Special software is intended for realisations of concrete institutions, production, scientific or any other original project.

3.2.1. Formal relations between data and programs

First of all formal relations between data and programs are being analysed in the following aspects:

- the variety of layout of data and programs in respect of each other in computer memory;
- address relations of data and programs;
- data coding and recoding;
- physical discrete computer elements and binary codes.

When one task is being done in computer memory and that is being realised by one program, then the fragments of that program and the fields of processed data can be laid differently in respect of each other. If data fields are being marked $d = d_1, d_2, d_3, \ldots$, and fragments of the program are $p = p_1, p_2, p_3, \ldots$, then the fragments of the program and data fields in an even (without spaces that are filled with other data or programs) memory field can be:

$p_1 , p_2 , p_3 , ..., d_1 , d_2 , d_3 , ...$
$p_1 , d_1 , p_2 , d_2 , p_3 , d_3, , ...$
- - - - - - - - - - - - - - - - - - -
$d_1 , d_2 , d_3 , ..., p_1 , p_2 , p_3 , ...$

It is more difficult to make a layout of intermediate data *t* of the task, which is being done, and the result data *r*. It is not always possible to write *t* and *r* into some *d*, i.e. the place of initial data. First of all it is not always possible to identify in advance if *t* and *r* fits into the place of *d*, secondly for many tasks until they are done the initial data are necessary. In this case intermediate data and result data are being inscribed into the place of initial data, the latter are being unmade. If the operational memory is big enough, then the mentioned problems are being solved formally and simply. After the programs and initial data a sufficient space in the memory for intermediary and result data are being devoted:

$$p, d, t, r.$$

It is necessary to emphasize here that despite big modern operational memory, its shortage always remains because first of all the amount of common or systematic programs is improving and getting bigger, secondly, there are more and more tasks, which cover big memory fields. These are various pictures, drawings, maps and the like.

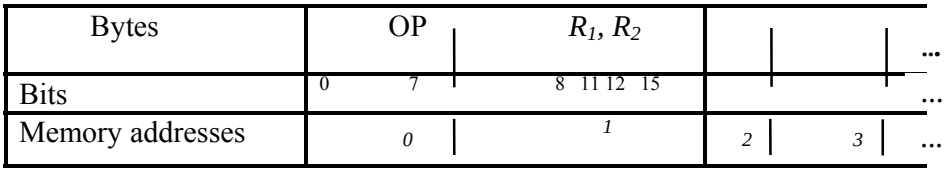
It is obvious that when where is a lack of operational memory, the program translates most fragments and data fields by itself by keeping them in outer memory. Both the fragments of the program and separate fields are being called into the operational memory when they are needed. After using them, the main fragment of the program changes them into other fragments of the programs and data fields. But then, it is obvious, that the common period of doing the tasks gets longer.

The situation in the operational memory becomes more difficult, if not one but several tasks are being done separately one after another or partially, i.e. in parallel. It is not an artificial thing, and the significance of doing such tasks becomes apparent then one or several tasks use their own data, or the intermediate or final data of other tasks, i.e. result data. Not going deeper into more difficult relations between the programs and data we will state that in all cases those relations are formal and directly not associated with the data semantics. All that has been said in this chapter so far is intended to the addresses of formal manipulation in various formats of data when looking for the necessary data to illustrate, when the data can change their residence in the operational memory, as well as the programs themselves can change their residence subject to the situation.

Formal address relations of data and programs are complicated as physically for the processing of data with the same programs both the programs and the data must be able to be in different places of operational memory. This is determined by yet another common situation in the operational memory, as because of big changes, i.e. yet another usage, the usual configuration of common software changes both in the operational and outer memory. Thus the address relation between the programs and data is the setting of new addresses of programs and their commands' residence, according their current position in the operational memory, for the processing of each datum. This setting of data and program addresses is usually called site drafting and it will be given in details later.

The most usual site drafting is being carried out starting with the fact that a non-working program, which is being memorised in the outer memory, is being considered amended according the zero address of the operational memory. It means that the address of the first byte of the first operand of the command of the first program is zero.

Let's say we have the simplest command *OP* R_1 R_2, where OP is the code of the operation, and R_1 and R_2 are the registers and operands.



Picture 3.14. First addresses of the commands

From this portraiture in the operational memory we can see that the code of the operation of the program covers one (the first) byte and is zero in the address.

If the commands are not connected with moving to another place in the operational memory, by over passing one or more commands, then they are being carried out successively, one after another. If it is being moved to another command, which does not go successively, then the address of the

program to which it is being moved should be indicated in one of the operands of the program from which it is being moved. That address is being indicated considering the zero meaning of the first address of the whole program. If the program is being taken out into the operational memory not from its zero address, then the first address is being changed into the address into which the program is being taken out b the site editor. Other addresses of moving commands to other commands are being added to the figure meaning of the first address and with this the editing of site addresses finishes. It is necessary to discriminate the addresses of transferring control programs from the addresses of the commands which are being formed in the elements of their operands and which indicate the mechanical addresses of data arrays or their parts or fields. The editing of those site addresses is totally different than that of other commands.

Do not forget that the data, which have to be processed by the program, can be memorized in the operational memory between separate parts of that program at the beginning or the end of the whole program or they can be "remote" from the program. It means that in the latter case there empty memory fields between the program and the data or other programs or other data of the program. In all cases a universal method of basic address and shift is being used for the identification of the data place (for the mechanical address) in the operational memory. When you want to identify the address of the data array or to move across the data array in the step of the length of the necessary variable (which is being expressed in the number of bytes) by indicating the addresses of the smaller parts of the data array. These questions where discussed in detail when binary programming with changeable length bytes of the commands in computer memory was being analyzed.

At the beginning of this chapter it was indicated that to the formal data and program relations we can attribute both data coding and recoding into such coding systems that enable the programs to identify various symbols and figures in which the data are being expressed.

In chapter 1.4. it was shortly mentioned that the binary codes of the same data symbols and figures can be various and different. It is only important that the inner codes of discrete computer elements coincide with data codes. For example, it absolutely unimportant how one (a number) is being expressed in binary one 0000 0001 or in any other binary combination in the byte, but the computer element system must enable to recognize it as a one. If it is not the cases, the data have to be recoded. In one and the same computer different devices can have not the same inner data codes. That is why it is apparent, that data codes only formally express the data and only formally according the codes the program recognizes the data.

Formal data recoding can vary very much, but their mechanisms are not difficult. Some recoding examples are given below. Let's say we have fragments of two earlier widely spread coding

systems ASCII (American Standard Code for Information Interchange) and EBCDIC (Expanded Binary Code Decimal Interchange Code).

Symbol	*..., C, D, E, ..., K, L, M, ..., 1, 2, 3, ..., /, &, ...*
ASCII	*..., A3, A4, A5, ..., AB, AC, AD, ..., 51, 52, 53, ..., 4F, 46,...*
EBCDIC	*..., C3, C4, C5, ..., D2, D3, D4, ..., F1, F2, F3, ..., 61, 50, ...*

Picture 3.15. The binary meanings of some symbols and numbers in different coding systems.

By recoding from one coding system ASCII into the system EBCDIC, the hexadecimal codes of ASCII are being considered the numbers of the row of bytes in the recoding table, and the codes of EBCDIC are being inscribed into the place of the table, which matches the same symbol, or number that is in the indicated place in the table. More clearly the recoding are given in the pictures 3.16 and 3.17.

3.3.2 The causality of data and programs

The goal of this chapter is to reveal the essence and variety of causality between the data and their supporting programs.

The knowing of such things enables you to do one and the same data processing task more rationally. It means that data structure (lay out) has to carried out in the way that makes the processing program as simple and if necessary easily replaced or improved as possible and that data processing requires minimal recourses of time for doing that task and memory.

In objective-problematic sphere as shown in picture 3.18 the causality between programs and data are indirect. The moderator of such relations is the algorithm the realization of which creates a program and thus the direct causality between the programs and data are being established. Undoubtedly the indirect relations exist because the features and peculiarities of the program realization of the algorithm depend on the way it is being made. In fact these are pre-computer causalities even when the data are being formed in some certain structure by a computer in order to be later given to be processed by the program that had realized the algorithm. In the example (in picture 3.18) on the right of the dotted line the scheme of causalities between data and the programs that process them is given. The types of causalities and to some extent independent from one another groups of those relations also reflect.

ASCII codes, as successively going hexadecimal numbers	00_{16},...,46_{16},...,$4F_{16}$,..., 51_{16}, 52_{16}, 53_{16},..., $A3_{16}$, $A4_{16}$, $A5_{16}$,...AB_{16},AC_{16},AD_{16},...,FF_{16}
EBCDIC	... *50* ... *61* ... F*1* F*2* F*3* ... C*3* C*4* C*5* ... D*2* D*3* D*4* ...
Numbers and Symbols	*&* */* *1* *2* *3* C D E K L M

Picture 3.16 recoding from ASCII to EBCDIC

EBCDIC codes, as successively going hexadecimal numbers	00_{16}, ..., 50_{16},..,61_{16}, ...,$C3_{16}$, $C4_{16}$, $C5_{16}$,..., $D2_{16}$,$D3_{16}$,$D4_{16}$ ...,$F1_{16}$,$F2_{16}$ $F3_{16}$,..., FF_{16}
ASCII	...,46, ...,4F ,..., A3, A4, A5, ... AB, AC, AD, ...,51, 52, 53, ...
Numbers and Symbols	*&* */* *C* *D* *E* *K* *L* *M* 1 2 3

Picture 3.17 Recoding from EBCDIC to ASCII

From the given examples it is apparent that the mechanism of recoding is a formal phenomenon because in the table of the recoding it is possible to inscribe any necessary code into the byte that is respectably remote from its beginning (according the hexadecimal meaning of the byte of the symbol, which is being recoded), Into which the mentioned number that shows the distance from the beginning of the table is being recoded. On the other hand, the table itself makes the field of 256 bytes, i.e. from 00_{16} to FF_{16}. Also, any decimal combination from 00000000_2 to 11111111_2, i.e. from 00_{16} to FF_{16}, can be in any byte of the table.

The causality does not appear also if the recoded data match the data, which process the inner codes of the devices. This is just a formal fact. The causality appears only when the recoded data are being set in some certain strict order (row) and strictly some certain data and not any other. About these causalities we are going to talk.

First of all the program and data of the final consumer are considered in picture 3.18. Despite the surroundings of a difficult programming system in which that program works the given groups of causalities remain. Besides, these groups of relations are characteristic of separate causalities of programming modules of the programming system that have their own systematic data. As it was apparent from the description of the groups of causalities, the features of mutual relations and reversible causalities are more characteristic of systematic software than of tasks of applicable nature.

Structural group of causalities between data and the program (shown as 1 in picture 3.18) is a group of mutual relations, the peculiarity of which is being shown by the structural relation between

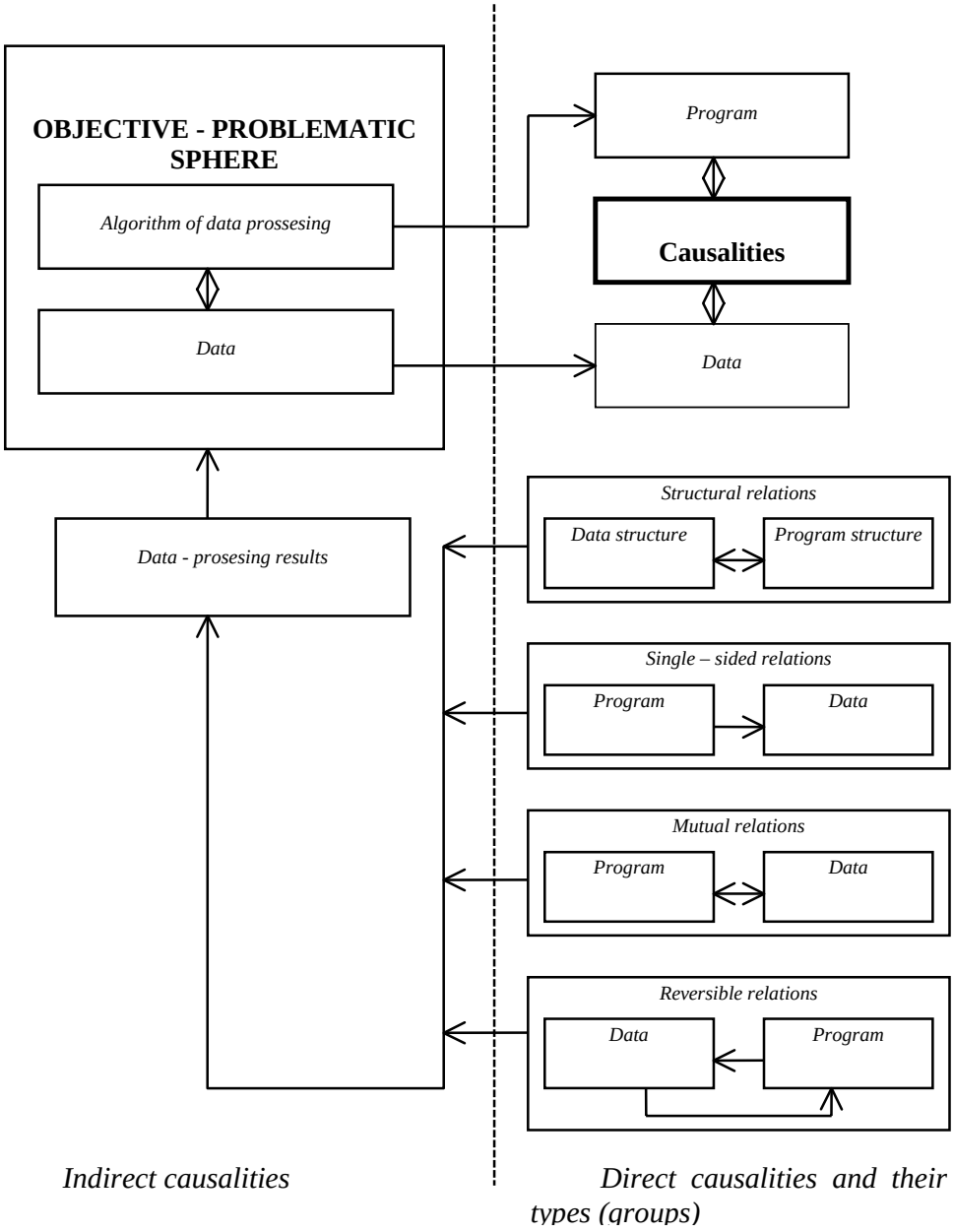
the data and the processing program. If the data are set in one way in their structure the separate parts of the program will have some certain setting. All this is apparent when we try to change the structure (layout) of data. Then it is easy to notice that we have to change the layout or programming procedures. For example, if the data are given in tables, then such data (without changing the data content) can be given in various different settings in the tables of various rating and row. The number of the tables and the amount of data and their place in the table will depend on the rating and row. That is why the ways of data search will necessarily change even if the tables are in some certain system of the database.

In the latter case the changed programming actions are being carried out not by the applied but by the control programming system of the database.

The essence of the single-sided causalities is that most often most programming procedures do not require estimating most results of processed data. For example, the arithmetical actions are being done, the data are being obtained that are the results of those actions and the procedure finishes with this. But, especially in more difficult programs of data processing the mutual relations between the program and the data exist. It means that the changing results of data processing are additionally analysed or the initial data are being analysed before the processing and the way to continue that program depends on the results of those analyses. For example, when there are negative figure results of data processing one way how to continue the program is being chosen and when there positive figure results of data processing another way to continue the program is being chosen, because the procedure set depends on the sigh of the number-result. If we obtain a negative number by deducting the amount of the material necessary for the production from the material remains in the storehouse, then such an operation can be carried out in the objective spheres. Then all what remains is given for producing, the final data consumer is being informed that there are not enough material as had been required therefore it is necessary to refill the storehouse. It is obvious that we would behave in a totally different way if there were enough material in the storehouse.

The reversible causality between the data and the program is being established when depending on the data it is necessary to change the program itself. For example the existing amount of data does not go into data fields foreseen and reserved in the program. Then the program itself is forced to change the size of the reserved fields. Without going deeper into the details of such changing it should be emphasized that such procedures are often difficult because they can also make temporal changes to the place in computer operational memory of other programs that are not connected with the program, which is being analysed. Another more difficult example of reversible relation can be a case when with the help of some certain programming means a part of some commands is being changed in the

working program because the usage of different subprograms of the program is not effective because of the abundance of such subprograms.



Picture 3.18. Causalities between data and programs

Summing up it is necessary to emphasize that causalities are being realized by using formal relations discussed in chapter 3.2.1. Because one or another formal means which are being used because of some certain reason, because one way of using can make the data processing task, and another cannot. Thus, the totality of causalities and formal relations between the data and programs which is being used at the same time reveals the essence of the programming because it explains how

to transfer data about the real world into binary expressions and how to transfer data about data into binary expressions which are called programs by using some certain strictly set formal relations they change data into the processing results and then they again express them from the binary form in the form that is understandable to the final consumer. In the next chapter the elements of programming will be given that are directly connected with the concepts given in this chapter.

Literature

1. ABEL, P. IBM PC Assembly Language and Programming. 5th ed. Prentice Hall International, Inc., 2000. 545 p.
2. BROOKSHAR, J. G. Computer science: An overview. 5th ed. Library of Congress Cataloging in Publication Data. USA. 1997. 483 p.
3. BROOKSHAR, J. G. Computer science: An overview. 8th ed. Reading, MA: Addison Wesley, 2004.
4. HAMACHER, V. C.; VRANNESIC, Z.G,;ZAKY, S.G. Computer organization, 6th ed. New York: McGrow-Hill, 2003.
5. PATTERSON, D.A.; HENNESSY, J.L. Computer Organization and Design. San Francisco: Morgan Kaufman, 1994.
6. SCHACH, R.S. Classical and object-Oriented Software Engineering, 3th ed. Chicago: Irwin, 1996.
7. TANENBAUM, A.S. Structured Computer Organization, 3th ed. Englewood Cliffs, N., J.: Prentice-Hall, 1990.
8. TEOREY, T.J. database Modelling and Design: The Entity-Relationship Approach. San Mateo, Colit.: Morgan Kaufmann, 2000.